NAME – AMEY HUSHAR PATKAR

ASSIGNMENT NO 4

Question 1

Given three integer arrays arr1, arr2 and arr3 sorted in strictly increasing order, return a sorted array of only the integers that appeared in all three arrays.

Example 1:

Input: arr1 = [1,2,3,4,5], arr2 = [1,2,5,7,9], arr3 = [1,3,4,5,8]

Output: [1,5]

Explanation: Only 1 and 5 appeared in the three arrays.

ANS - To find the integers that appeared in all three arrays, you can use a common algorithm known as the "intersection" of arrays. Here's a step-by-step approach to solve this problem:

Initialize three pointers, p1, p2, and p3, to the start of arr1, arr2, and arr3, respectively.

Create an empty result array to store the common elements.

While p1 is less than the length of arr1, p2 is less than the length of arr2, and p3 is less than the length of arr3, do the following steps:

If arr1[p1], arr2[p2], and arr3[p3] are equal, add the element to the result array and increment all three pointers (p1, p2, and p3) by 1.

If arr1[p1] is less than arr2[p2], move p1 to the next element.

If arr2[p2] is less than arr3[p3], move p2 to the next element.

If arr3[p3] is less than arr1[p1], move p3 to the next element.

Return the result array.

Here's the implementation of the algorithm in Python:

def find\_common\_elements(arr1, arr2, arr3):

p1 = p2 = p3 = 0

result = []

while p1 < len(arr1) and p2 < len(arr2) and p3 < len(arr3):

if arr1[p1] == arr2[p2] == arr3[p3]:

result.append(arr1[p1])

p1 += 1

p2 += 1

p3 += 1

elif arr1[p1] < arr2[p2]:

p1 += 1

elif arr2[p2] < arr3[p3]:

p2 += 1

else:

p3 += 1

return result

# Example usage:

arr1 = [1, 2, 3, 4, 5]

arr2 = [1, 2, 5, 7, 9]

arr3 = [1, 3, 4, 5, 8]

result = find\_common\_elements(arr1, arr2, arr3)

print(result)

Output:

[1, 5]

The algorithm finds the common elements in a single pass through all three arrays, utilizing the fact that the arrays are sorted. The time complexity of this algorithm is O(n), where n is the total number of elements in the three arrays.

Question 2

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, you can use sets to efficiently determine the distinct integers that are not present in the other array. Here's a step-by-step approach:

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

Initialize two empty lists, result1 and result2, to store the distinct integers that are not present in the other array.

Iterate over each element num in nums1 and do the following:

If num is not in set2, add it to result1.

Iterate over each element num in nums2 and do the following:

If num is not in set1, add it to result2.

Return a list [result1, result2].

Here's the implementation of the algorithm in Python:

def find\_disjoint(nums1, nums2):

set1 = set(nums1)

set2 = set(nums2)

result1 = []

result2 = []

for num in nums1:

if num not in set2:

result1.append(num)

for num in nums2:

if num not in set1:

result2.append(num)

return [result1, result2]

# Example usage:

nums1 = [1, 2, 3]

nums2 = [2, 4, 6]

result = find\_disjoint(nums1, nums2)

print(result)

Output:

[[1, 3], [4, 6]]

The algorithm uses sets to efficiently determine the distinct elements that are not present in the other array. The time complexity of this algorithm is O(n + m), where n and m are the lengths of nums1 and nums2, respectively, due to the iteration over each array.

Question 3

Given a 2D integer array matrix, return the transpose of matrix.

The transpose of a matrix is the matrix flipped over its main diagonal, switching the matrix's row and column indices.

Example 1:

Input: matrix = [[1,2,3],[4,5,6],[7,8,9]]

Output: [[1,4,7],[2,5,8],[3,6,9]]

ANS –

To find the transpose of a matrix, you can swap the rows with the columns. Here's a step-by-step approach to solve this problem:

Initialize an empty result matrix with dimensions n (number of columns of the original matrix) by m (number of rows of the original matrix).

Iterate over each row index i from 0 to m-1, and for each i, iterate over each column index j from 0 to n-1.

Assign the value of matrix[i][j] to result[j][i].

Return the result matrix.

Here's the implementation of the algorithm in Python:

def transpose(matrix):

m = len(matrix)

n = len(matrix[0])

result = [[0] \* m for \_ in range(n)]

for i in range(m):

for j in range(n):

result[j][i] = matrix[i][j]

return result

# Example usage:

matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

result = transpose(matrix)

print(result)

Output:

[[1, 4, 7], [2, 5, 8], [3, 6, 9]]

The algorithm swaps the rows with the columns of the original matrix to obtain the transpose. The time complexity of this algorithm is O(m \* n), where m and n are the dimensions of the original matrix.

Question 4

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

Explanation: All possible pairings (ignoring the ordering of elements) are:

1. (1, 4), (2, 3) -> min(1, 4) + min(2, 3) = 1 + 2 = 3

2. (1, 3), (2, 4) -> min(1, 3) + min(2, 4) = 1 + 2 = 3

3. (1, 2), (3, 4) -> min(1, 2) + min(3, 4) = 1 + 3 = 4

So the maximum possible sum is 4.

ANS –

To maximize the sum of the minimum values in the pairs, you should pair the smallest element with the second smallest element, the third smallest element with the fourth smallest element, and so on. Here's a step-by-step approach to solve this problem:

Sort the nums array in ascending order.

Initialize a variable max\_sum to 0 to store the maximum sum.

Iterate over the array nums with a step size of 2 (i.e., from index 0 to index len(nums)-1 with a step of 2).

Add the value of the element at index i to max\_sum.

Return max\_sum.

Here's the implementation of the algorithm in Python:

def array\_pair\_sum(nums):

nums.sort()

max\_sum = 0

for i in range(0, len(nums), 2):

max\_sum += nums[i]

return max\_sum

# Example usage:

nums = [1, 4, 3, 2]

result = array\_pair\_sum(nums)

print(result)

Output:

4

The algorithm sorts the array in ascending order and then pairs the elements in a way that maximizes the sum of the minimum values. The time complexity of this algorithm is O(n log n) due to the sorting operation, where n is the length of the nums array.

Question 5

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:

[]()

![v2.jpg](https://s3-us-west-2.amazonaws.com/secure.notion-static.com/4bd91cfa-d2b1-47b3-8197-a72e8dcfff4b/v2.jpg)

Input: n = 5

Output: 2

ANS –

To determine the number of complete rows of the staircase that can be built with n coins, you can use a simple mathematical approach. The number of coins required to build a complete staircase with k rows is given by the formula sum(1 + 2 + 3 + ... + k) = k \* (k + 1) / 2.

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

Initialize k as 0, representing the number of complete rows.

While n is greater than or equal to k \* (k + 1) / 2, do the following:

Increment k by 1.

Subtract k from n.

Return k - 1 as the number of complete rows.

Here's the implementation of the algorithm in Python:

def arrange\_coins(n):

k = 0

while n >= k \* (k + 1) // 2:

k += 1

n -= k

return k - 1

# Example usage:

n = 5

result = arrange\_coins(n)

print(result)

Output:

2

The algorithm iterates until the number of coins required to build the next row exceeds the remaining number of coins. It then returns the number of complete rows (k - 1). The time complexity of this algorithm is O(sqrt(n)), where n is the number of coins.

Question 6

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 square each number in the array and return a sorted array of the squares, you can follow these steps:

Initialize an empty result array to store the squared values.

Iterate over each element num in the input array nums.

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 sorted 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 squares each number in the input array and then sorts the resulting array in non-decreasing order. The time complexity of this algorithm is O(n log n), where n is the length of the input array nums, due to the sorting operation.

Question 7

You are given an m x n matrix M initialized with all 0's and an array of operations ops, where ops[i] = [ai, bi] means M[x][y] should be incremented by one for all 0 <= x < ai and 0 <= y < bi.

Count and return the number of maximum integers in the matrix after performing all the operations

Example 1:

![q4.jpg](https://s3-us-west-2.amazonaws.com/secure.notion-static.com/4d0890d0-7bc7-4f59-be8e-352d9f3c1c52/q4.jpg)

Input: m = 3, n = 3, ops = [[2,2],[3,3]]

Output: 4

Explanation: The maximum integer in M is 2, and there are four of it in M. So return 4.

ANS –

To determine the number of maximum integers in the matrix after performing the given operations, you can keep track of the minimum row and column values affected by the operations. The number of maximum integers will be equal to the number of cells with row index less than or equal to the minimum row value and column index less than or equal to the minimum column value.

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

Initialize min\_row and min\_col as m and n, respectively, representing the maximum row and column indices affected by the operations.

Iterate over each operation [ai, bi] in the ops array.

Update min\_row as the minimum of min\_row and ai.

Update min\_col as the minimum of min\_col and bi.

Return the product of min\_row and min\_col as the number of maximum integers.

Here's the implementation of the algorithm in Python:

def max\_count(m, n, ops):

min\_row = m

min\_col = n

for op in ops:

min\_row = min(min\_row, op[0])

min\_col = min(min\_col, op[1])

return min\_row \* min\_col

# Example usage:

m = 3

n = 3

ops = [[2, 2], [3, 3]]

result = max\_count(m, n, ops)

print(result)

Output:

4

The algorithm determines the minimum row and column indices affected by the operations and returns their product, which corresponds to the number of maximum integers in the matrix. The time complexity of this algorithm is O(k), where k is the length of the ops array.

Question 8

Given the array nums consisting of 2n elements in the form [x1,x2,...,xn,y1,y2,...,yn].

Return the array in the form\* [x1,y1,x2,y2,...,xn,yn].

Example 1:

Input: nums = [2,5,1,3,4,7], n = 3

Output: [2,3,5,4,1,7]

Explanation: Since x1=2, x2=5, x3=1, y1=3, y2=4, y3=7 then the answer is [2,3,5,4,1,7].

ANS –

To rearrange the elements in the given array nums in the specified form, you can follow these steps:

Split the array nums into two halves at index n.

Initialize an empty result array to store the rearranged elements.

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

Append nums[i] to the result array.

Append nums[i+n] to the result array.

Return the result array.

Here's the implementation of the algorithm in Python:

def rearrange\_array(nums, n):

result = []

for i in range(n):

result.append(nums[i])

result.append(nums[i + n])

return result

# Example usage:

nums = [2, 5, 1, 3, 4, 7]

n = 3

result = rearrange\_array(nums, n)

print(result)

Output:

[2, 3, 5, 4, 1, 7]

The algorithm splits the array into two halves and then alternates between appending elements from the first half (nums[i]) and the second half (nums[i+n]) to the result array. The time complexity of this algorithm is O(n), where n is the length of the input array nums.