NAME – AMEY HUSHAR PATKAR

ASSIGNMENT NO 3

Question 1

Given an integer array nums of length n and an integer target, find three integers

in nums such that the sum is closest to the target.

Return the sum of the three integers.

You may assume that each input would have exactly one solution.

Example 1:

Input: nums = [-1,2,1,-4], target = 1

Output: 2

Explanation: The sum that is closest to the target is 2. (-1 + 2 + 1 = 2).

ANS -

To solve this problem, we can use a similar approach as the three-sum problem. We will sort the array nums in ascending order and then iterate through the array. For each element nums[i], we will use two pointers, left and right, to find a triplet with the closest sum to the target.

Here's the step-by-step approach:

Sort the array nums in ascending order.

Initialize a variable closest\_sum to store the sum of the closest triplet to the target. Set it to a large value initially.

Iterate through the array nums from index 0 to n-2 (since we need three integers).

For each element nums[i], set two pointers: left = i + 1 and right = n - 1.

While left < right, do the following:

Calculate the current sum: current\_sum = nums[i] + nums[left] + nums[right].

If the absolute difference between current\_sum and target is smaller than the absolute difference between closest\_sum and target, update closest\_sum with current\_sum.

If current\_sum is greater than the target, decrement right by 1.

If current\_sum is less than the target, increment left by 1.

If current\_sum is equal to the target, return current\_sum as the closest sum.

After the loop, return closest\_sum as the final result.

Here's the implementation in Python:

def threeSumClosest(nums, target):

nums.sort() # Sort the array in ascending order

n = len(nums)

closest\_sum = float('inf') # Initialize closest\_sum to a large value

for i in range(n-2):

left = i + 1

right = n - 1

while left < right:

current\_sum = nums[i] + nums[left] + nums[right]

if abs(current\_sum - target) < abs(closest\_sum - target):

closest\_sum = current\_sum

if current\_sum > target:

right -= 1

elif current\_sum < target:

left += 1

else:

return current\_sum

return closest\_sum

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

nums = [-1, 2, 1, -4]

target = 1

result = threeSumClosest(nums, target)

print("Output:", result)

The output will be:

Output: 2

It means that the sum of the three integers closest to the target 1 is 2 (-1 + 2 + 1 = 2).

Question 2

Given an array nums of n integers, return an array of all the unique quadruplets

[nums[a], nums[b], nums[c], nums[d]] such that:

● 0 <= a, b, c, d < n

● a, b, c, and d are distinct.

● nums[a] + nums[b] + nums[c] + nums[d] == target

You may return the answer in any order.

Example 1:

Input: nums = [1,0,-1,0,-2,2], target = 0

Output: [[-2,-1,1,2],[-2,0,0,2],[-1,0,0,1]]

ANS – To solve this problem, we can use a similar approach as the three-sum problem. We will sort the array nums in ascending order and then use nested loops to iterate through the array and find unique quadruplets that sum up to the target.

Here's the step-by-step approach:

Sort the array nums in ascending order.

Initialize an empty list result to store the unique quadruplets.

Iterate through the array nums from index 0 to n-4 (since we need four integers).

For each element nums[a], check if a > 0 and nums[a] == nums[a-1]. If true, continue to the next iteration to avoid duplicates.

Iterate through the array nums from index a+1 to n-3.

For each element nums[b], check if b > a+1 and nums[b] == nums[b-1]. If true, continue to the next iteration to avoid duplicates.

Set two pointers: left = b + 1 and right = n - 1.

While left < right, do the following:

Calculate the current sum: current\_sum = nums[a] + nums[b] + nums[left] + nums[right].

If current\_sum is equal to the target, append [nums[a], nums[b], nums[left], nums[right]] to result.

If current\_sum is greater than the target, decrement right by 1.

If current\_sum is less than the target, increment left by 1.

Return result as the final output.

Here's the implementation in Python:

def fourSum(nums, target):

nums.sort() # Sort the array in ascending order

n = len(nums)

result = []

for a in range(n-3):

if a > 0 and nums[a] == nums[a-1]:

continue

for b in range(a+1, n-2):

if b > a+1 and nums[b] == nums[b-1]:

continue

left = b + 1

right = n - 1

while left < right:

current\_sum = nums[a] + nums[b] + nums[left] + nums[right]

if current\_sum == target:

result.append([nums[a], nums[b], nums[left], nums[right]])

left += 1

right -= 1

while left < right and nums[left] == nums[left-1]:

left += 1

while left < right and nums[right] == nums[right+1]:

right -= 1

elif current\_sum < target:

left += 1

else:

right -= 1

return result

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

nums = [1, 0, -1, 0, -2, 2]

target = 0

result = fourSum(nums, target)

print("Output:", result)

The output will be:

Output: [[-2, -1, 1, 2], [-2, 0, 0, 2], [-1, 0, 0, 1]]

It means that the unique quadruplets whose sum is equal to the target 0 are [-2, -1, 1, 2], [-2, 0, 0, 2], and [-1, 0, 0, 1].

Question 3

A permutation of an array of integers is an arrangement of its members into a

sequence or linear order.

For example, for arr = [1,2,3], the following are all the permutations of arr:

[1,2,3], [1,3,2], [2, 1, 3], [2, 3, 1], [3,1,2], [3,2,1].

The next permutation of an array of integers is the next lexicographically greater

permutation of its integer. More formally, if all the permutations of the array are

sorted in one container according to their lexicographical order, then the next

permutation of that array is the permutation that follows it in the sorted container.

If such an arrangement is not possible, the array must be rearranged as the

lowest possible order (i.e., sorted in ascending order).

● For example, the next permutation of arr = [1,2,3] is [1,3,2].

● Similarly, the next permutation of arr = [2,3,1] is [3,1,2].

● While the next permutation of arr = [3,2,1] is [1,2,3] because [3,2,1] does not

have a lexicographical larger rearrangement.

Given an array of integers nums, find the next permutation of nums.

The replacement must be in place and use only constant extra memory.

Example 1:

Input: nums = [1,2,3]

Output: [1,3,2]

ANS –

To find the next permutation of an array, we need to follow these steps:

Start from the rightmost element of the array and find the first pair of consecutive elements where the left element is smaller than the right element. Let's call the index of the left element i.

If no such pair is found, it means the given permutation is the last possible permutation. In this case, we need to rearrange the array in ascending order, which will be the lowest possible order. Return the sorted array.

If we found a pair at index i, we need to find the smallest element to the right of nums[i] that is larger than nums[i]. Let's call the index of this element j.

Swap the elements at indices i and j.

Reverse the elements from index i+1 to the end of the array. This step ensures that the elements to the right of nums[i] are in ascending order.

Return the modified array.

Here's the implementation in Python:

def nextPermutation(nums):

# Find the first pair of consecutive elements where nums[i] < nums[i+1]

i = len(nums) - 2

while i >= 0 and nums[i] >= nums[i+1]:

i -= 1

if i >= 0:

# Find the smallest element to the right of nums[i] that is larger than nums[i]

j = len(nums) - 1

while j > i and nums[j] <= nums[i]:

j -= 1

# Swap the elements at indices i and j

nums[i], nums[j] = nums[j], nums[i]

# Reverse the elements from index i+1 to the end

left = i + 1

right = len(nums) - 1

while left < right:

nums[left], nums[right] = nums[right], nums[left]

left += 1

right -= 1

return nums

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

nums = [1, 2, 3]

result = nextPermutation(nums)

print("Output:", result)

The output will be:

Output: [1, 3, 2]

It means that the next permutation of the array [1, 2, 3] is [1, 3, 2].

Question 4

Given a sorted array of distinct integers and a target value, return the index if the

target is found. If not, return the index where it would be if it were inserted in

order.

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

Example 1:

Input: nums = [1,3,5,6], target = 5

Output: 2

ANS –

To solve this problem efficiently with a runtime complexity of O(log n), we can use a modified version of the binary search algorithm.

Here's the algorithm:

Set the left pointer left to 0 and the right pointer right to the length of the array nums minus 1.

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

a. Calculate the middle index mid as the average of left and right, rounded down to the nearest integer.

b. If the value at nums[mid] is equal to the target, return mid.

c. If the value at nums[mid] is greater than the target, update right to mid - 1.

d. If the value at nums[mid] is less than the target, update left to mid + 1.

If the target is not found, return left.

This algorithm performs a binary search by comparing the middle element of the array with the target value. Based on the comparison, it adjusts the search range by updating either the left or right pointer.

Here's the implementation in Python:

def searchInsert(nums, target):

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 left

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

nums = [1, 3, 5, 6]

target = 5

result = searchInsert(nums, target)

print("Output:", result)

The output will be:

Output: 2

It means that the target value 5 is found at index 2 in the array [1, 3, 5, 6].

Question 5

You are given a large integer represented as an integer array digits, where each

digits[i] is the ith digit of the integer. The digits are ordered from most significant

to least significant in left-to-right order. The large integer does not contain any

leading 0's.

Increment the large integer by one and return the resulting array of digits.

Example 1:

Input: digits = [1,2,3]

Output: [1,2,4]

Explanation: The array represents the integer 123.

Incrementing by one gives 123 + 1 = 124.

Thus, the result should be [1,2,4].

ANS –

To solve this problem, we can start from the least significant digit (the last element of the array) and perform the increment operation by considering carry.

Here's the algorithm:

Initialize a carry variable as 1 to represent the initial increment value.

Traverse the digits array from right to left:

a. Add the carry to the current digit.

b. Update the current digit as the result of the modulo operation of the sum by 10.

c. Update the carry as the result of the integer division of the sum by 10.

d. If the carry becomes 0, no further increment is required, so break out of the loop.

If the carry is still 1 after the loop, it means there was a carry from the most significant digit. In this case, prepend a new element with a value of 1 to the digits array.

Return the modified digits array.

Here's the implementation in Python:

def plusOne(digits):

carry = 1

n = len(digits)

for i in range(n - 1, -1, -1):

digits[i] += carry

carry = digits[i] // 10

digits[i] %= 10

if carry == 0:

break

if carry == 1:

digits.insert(0, 1)

return digits

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

digits = [1, 2, 3]

result = plusOne(digits)

print("Output:", result)

The output will be:

Output: [1, 2, 4]

It means that by incrementing the number represented by the digits [1, 2, 3] by one, we get the resulting digits [1, 2, 4].

Question 6

Given a non-empty array of integers nums, every element appears twice except

for one. Find that single one.

You must implement a solution with a linear runtime complexity and use only

constant extra space.

Example 1:

Input: nums = [2,2,1]

Output: 1

ANS –

To find the single element that appears only once in the given array nums, we can utilize the bitwise XOR operation.

The XOR operation (^) returns 0 when two bits are the same, and 1 when they are different. Therefore, if we XOR all the elements in the array, the result will be the element that appears only once, as the duplicate elements will cancel each other out.

Here's the algorithm:

Initialize a variable result to 0.

Iterate through each element num in nums.

Update result by performing the XOR operation between result and num.

After the iteration, result will hold the value of the single element that appears only once.

Return the value of result.

Here's the implementation in Python:

def singleNumber(nums):

result = 0

for num in nums:

result ^= num

return result

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

nums = [2, 2, 1]

result = singleNumber(nums)

print("Output:", result)

The output will be:

Output: 1

It means that the element that appears only once in the array [2, 2, 1] is 1.

Question 7

You are given an inclusive range [lower, upper] and a sorted unique integer array

nums, where all elements are within the inclusive range.

A number x is considered missing if x is in the range [lower, upper] and x is not in

nums.

Return the shortest sorted list of ranges that exactly covers all the missing

numbers. That is, no element of nums is included in any of the ranges, and each

missing number is covered by one of the ranges.

Example 1:

Input: nums = [0,1,3,50,75], lower = 0, upper = 99

Output: [[2,2],[4,49],[51,74],[76,99]]

Explanation: The ranges are:

[2,2]

[4,49]

[51,74]

[76,99]

ANS -

To solve this problem, we can iterate through the sorted nums array and track the missing ranges. We'll compare each element with its previous element to determine if there are any missing numbers between them.

Here's the algorithm:

Initialize an empty list result to store the missing ranges.

Initialize the variable start as lower, which represents the starting point of a potential missing range.

Iterate through each element num in the nums array.

If num is equal to start, increment start by 1 and continue to the next iteration.

If num is greater than start by 1, we have found a missing range. Append the range [start, num-1] to result.

Update start to num + 1 to check for the next potential missing range.

After the iteration, if start is less than or equal to upper, append the range [start, upper] to result.

Return the result list containing all the missing ranges.

Here's the implementation in Python:

def findMissingRanges(nums, lower, upper):

result = []

start = lower

for num in nums:

if num == start:

start += 1

elif num > start:

result.append([start, num - 1])

start = num + 1

if start <= upper:

result.append([start, upper])

return result

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

nums = [0, 1, 3, 50, 75]

lower = 0

upper = 99

result = findMissingRanges(nums, lower, upper)

print("Output:", result)

The output will be:

Output: [[2, 2], [4, 49], [51, 74], [76, 99]]

It means that the missing ranges in the given inclusive range [0, 99] for the sorted nums array [0, 1, 3, 50, 75] are [[2, 2], [4, 49], [51, 74], [76, 99]].

Question 8

Given an array of meeting time intervals where intervals[i] = [starti, endi],

determine if a person could attend all meetings.

Example 1:

Input: intervals = [[0,30],[5,10],[15,20]]

Output: false

ANS –

To determine if a person could attend all meetings, we need to check if any two meetings overlap. If there is any overlap between meetings, it means that the person cannot attend all the meetings.

Here's an algorithm to solve this problem:

Sort the meeting time intervals based on the start time in ascending order.

Iterate through the sorted intervals starting from the second interval.

For each interval, compare its start time with the end time of the previous interval.

If the start time of the current interval is less than or equal to the end time of the previous interval, there is an overlap, so return False.

If there are no overlaps after iterating through all the intervals, return True.

Here's the implementation in Python:

def canAttendMeetings(intervals):

intervals.sort(key=lambda x: x[0]) # Sort intervals based on start time

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

if intervals[i][0] < intervals[i-1][1]:

return False

return True

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

intervals = [[0, 30], [5, 10], [15, 20]]

result = canAttendMeetings(intervals)

print("Output:", result)

The output will be:

Output: False

It means that based on the given meeting time intervals [[0, 30], [5, 10], [15, 20]], a person cannot attend all the meetings since there is an overlap between the first and third meetings.