**DSA – ASSIGNMENT 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).

**Solution. :-**

* Sort the array nums in ascending order.
* Initialize a variable closestSum to store the sum closest to the target. Set it to a large value initially.
* Iterate through the array nums from left to right using a pointer i. For each nums[i], perform a two-pointer approach to find the remaining two integers that sum up closest to the target.
  + Initialize two pointers, left and right, with values i+1 and n-1, respectively, where n is the length of the array.
  + b. While left < right, calculate the current sum currentSum = nums[i] + nums[left] + nums[right].
  + c. If the absolute difference between currentSum and target is smaller than the absolute difference between closestSum and target, update closestSum to currentSum.
  + d. If currentSum is greater than the target, decrement right to try and get a smaller sum. e. If currentSum is smaller than the target, increment left to try and get a larger sum.
  + f. If currentSum is equal to the target, return target as the closest sum.
* After the iteration, return the closestSum as the result.

**def threeSumClosest(nums, target):**

**nums.sort()**

**n = len(nums)**

**closestSum = float('inf')**

**for i in range(n - 2):**

**left = i + 1**

**right = n - 1**

**while left < right:**

**currentSum = nums[i] + nums[left] + nums[right]**

**if abs(currentSum - target) < abs(closestSum - target):**

**closestSum = currentSum**

**if currentSum > target:**

**right -= 1**

**elif currentSum < target:**

**left += 1**

**else:**

**return target**

**return closestSum**

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

**target = 1**

**result = threeSumClosest(nums, target)**

**print(result)**

💡 **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]]

**Solution. :-**

* Sort the array nums in ascending order.
* Initialize an empty list result to store the unique quadruplets.
* Iterate through the array nums up to the third-to-last element. Let's call this element a.
* If a is the same as the previous element, continue to the next iteration to avoid duplicates.
* Iterate through the remaining elements of nums starting from a+1 up to the second-to-last element. Let's call this element b.
* If b is the same as the previous element, continue to the next iteration to avoid duplicates.
* Set two pointers, left and right, to the elements following b+1 and n-1, respectively, where n is the length of the array.
* While left < right, calculate the current sum currentSum = nums[a] + nums[b] + nums[left] + nums[right].
* If currentSum is equal to the target, add the quadruplet [nums[a], nums[b], nums[left], nums[right]] to the result list.
* Adjust the pointers left and right based on the comparison of currentSum with the target:
  + If currentSum is less than the target, increment left.
  + If currentSum is greater than the target, decrement right.
  + If currentSum is equal to the target, increment left and decrement right to find more possible quadruplets.
* After the inner loop, update the pointers a and b to their next unique values.
* Repeat steps 4-11 until all unique quadruplets have been found.
* Return the result list.

**def fourSum(nums, target):**

**nums.sort()**

**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:**

**currentSum = nums[a] + nums[b] + nums[left] + nums[right]**

**if currentSum == 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 currentSum < target:**

**left += 1**

**else:**

**right -= 1**

**return result**

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

**target = 0**

**result = fourSum(nums, target)**

**print(result)**

💡 **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]

**Solution. :-**

* Start from the right end of the array and find the first pair of adjacent elements where nums[i] < nums[i+1]. Let's call the index of the first element in this pair partitionIndex. If no such pair is found, it means the array is in descending order, and we return the sorted array in ascending order.
* Start from the right end of the array again and find the first element that is greater than nums[partitionIndex]. Let's call the index of this element swapIndex.
* Swap the elements at partitionIndex and swapIndex.
* Reverse the subarray starting from partitionIndex+1 till the end of the array.
* The array nums now represents the next permutation.

**def nextPermutation(nums):**

**n = len(nums)**

**partitionIndex = -1**

**# Find the partition index**

**for i in range(n - 2, -1, -1):**

**if nums[i] < nums[i + 1]:**

**partitionIndex = i**

**break**

**if partitionIndex == -1:**

**# Array is in descending order, return sorted array**

**nums.sort()**

**return**

**# Find the swap index**

**swapIndex = -1**

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

**if nums[i] > nums[partitionIndex]:**

**swapIndex = i**

**break**

**# Swap the elements**

**nums[partitionIndex], nums[swapIndex] = nums[swapIndex], nums[partitionIndex]**

**# Reverse the subarray**

**left = partitionIndex + 1**

**right = n - 1**

**while left < right:**

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

**left += 1**

**right -= 1**

**nums = [1, 2, 3]**

**nextPermutation(nums)**

**print(nums)**

💡 **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

**Solution. :-**

* Initialize two pointers, left and right, to the start and end of the array, respectively.
* While left <= right, calculate the middle index as mid using the formula mid = (left + right) // 2.
* If the target value is equal to the value at index mid, return mid as the target index.
* If the target value is less than the value at index mid, set right to mid - 1 to search the left half of the array.
* If the target value is greater than the value at index mid, set left to mid + 1 to search the right half of the array.
* After the while loop, if the target value is not found, return left as the index where the target value would be inserted.

**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**

**nums = [1, 3, 5, 6]**

**target = 5**

**result = searchInsert(nums, target)**

**print(result)**

💡 **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].

**Solution. :-**

* Start from the right end of the array and initialize a carry variable to 1, indicating the increment by one.
* Iterate through the array digits in reverse order:
  + Add the carry to the current digit at digits[i].
  + Update the carry by dividing the sum by 10, to get the new carry value for the next iteration.
  + Update the current digit by taking the modulo 10 of the sum, to get the new digit value after incrementing.
* After the iteration, if the carry is still 1, it means there was a carry at the most significant digit. In this case, insert 1 at the beginning of the array.
* Return the updated array digits.

**def plusOne(digits):**

**n = len(digits)**

**carry = 1**

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

**sum = digits[i] + carry**

**digits[i] = sum % 10**

**carry = sum // 10**

**if carry == 1:**

**digits.insert(0, 1)**

**return digits**

**digits = [1, 2, 3]**

**result = plusOne(digits)**

**print(result)**

💡 **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

**Solution. :-**

* Initialize a variable result to 0.
* Iterate through each element num in the array nums.
* Update result by XOR-ing it with num.
* After the iteration, result will contain the single element that appears only once.
* Return result.

**def singleNumber(nums):**

**result = 0**

**for num in nums:**

**result ^= num**

**return result**

**nums = [2, 2, 1]**

**result = singleNumber(nums)**

**print(result)**

💡 **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]]

**Solution. :-**

* Initialize a variable start to lower and an empty list result to store the ranges.
  + Iterate through each number num in the array nums:
  + If num is greater than start, it means there is a gap between start and num. Append the range [start, num-1] to result.
* Update start to num + 1.
* After the iteration, if start is less than or equal to upper, there is a final gap between start and upper. Append the range [start, upper] to result.
* Return result.

**def findMissingRanges(nums, lower, upper):**

**result = []**

**start = lower**

**for num in nums:**

**if num > start:**

**result.append(getRange(start, num - 1))**

**start = num + 1**

**if start <= upper:**

**result.append(getRange(start, upper))**

**return result**

**def getRange(start, end):**

**if start == end:**

**return str(start)**

**else:**

**return str(start) + "->" + str(end)**

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

**lower = 0**

**upper = 99**

**result = findMissingRanges(nums, lower, upper)**

**print(result)**

**Explanation:** The ranges 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

**Solution. :-**

* Sort the intervals based on the start time of each meeting.
* Iterate through each interval starting from the second 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. Return false.
* If there are no overlaps, return true.

**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**

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

**result = canAttendMeetings(intervals)**

**print(result)**