**Assignment Questions and Answer 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).

import java.util.Arrays;

public class sumCloset {

    static int threeSumClosest(int[] nums, int target) {

        Arrays.sort(nums);

        int resultSum = nums[0] + nums[1] + nums[2];

        int closest = Integer.MAX\_VALUE;

        for(int i = 0; i < nums.length - 2; i++){

            int left = i + 1;

            int right = nums.length - 1;

            while(left < right){

                int sum = nums[i] + nums[left] + nums[right];

                if(sum == target){

                    return target;

                }

                if(sum < target){

                    left ++;

                }

                else {

                    right --;

                }

                int diff = Math.abs(sum - target);

                if(diff < closest){

                    resultSum = sum;

                    closest = diff;

                }

            }

        }

        return resultSum;

    }

    public static void main(String[] args) {

        int [] nums = {-1,2,1,-4};

        int target = 1;

        System.out.println("Closest Sum is as -> " + threeSumClosest(nums, target));

    }

}

💡 **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]]

import java.util.\*;

public class FourSum {

    static List<List<Integer>> fourSum(int[] nums, int target) {

        Arrays.sort(nums);

        List<List<Integer>> allQuadruple = new ArrayList<>();

        int n = nums.length;

        for(int i = 0; i < n; i++){

            if(i > 0 && nums[i] == nums[i - 1]){

                continue;

            }

            for(int j = i + 1; j < n; j++){

                if(j != i + 1 && nums[j] == nums[j - 1]){

                    continue;

                }

                int start = j + 1;

                int end = n - 1;

                while(start < end){

                    int sum = nums[i];

                    sum += nums[j];

                    sum += nums[start];

                    sum += nums[end];

                    if(sum == target){

                        allQuadruple.add(Arrays.asList(nums[i], nums[j], nums[start], nums[end]));

                        start++;

                        end--;

                        while(start < end && nums[start] == nums[start - 1]){

                            start ++;

                        }

                        while(start < end && nums[end] == nums[end + 1]){

                            end--;

                        }

                    }

                    else if(sum < target){

                        start++;

                    }

                    else{

                        end--;

                    }

                }

            }

        }

        return allQuadruple;

    }

    public static void main(String[] args) {

        int nums[] = {1,0,-1,0,-2,2};

        int target = 0;

        System.out.println(fourSum(nums, target));

    }

}

💡 **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]

import java.util.\*;

public class nextPermutation {

    static void nextPermutation(int[] arr) {

        int temp=0,i,min=Integer.MAX\_VALUE,pivot=0,pos=0;

        for ( i=arr.length-1;i>=1;i--)

            if (arr[i]>arr[i-1])

            {

                pivot=i;break;

            }

        if (pivot>0)

        {

            for (i=pivot;i<arr.length;i++)

            {

                if (arr[i]>arr[pivot-1])

                    min=Math.min(min,arr[i]);

                if (min==arr[i])

                    pos=i;

            }

            swap(arr,pos,pivot-1);

        }

        int high=arr.length-1;

        while(pivot<=high){

            swap(arr,pivot,high);

            high--;pivot++;

        }

    }

    static void swap(int[] arr,int low,int high){

            int temp=arr[low];

            arr[low]=arr[high];

            arr[high]=temp;

    }

    public static void main(String[] args) {

        int [] nums = {1,2,3};

        nextPermutation(nums);

        System.out.println(Arrays.toString(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

public class serachPostion {

    static int searchInsert(int[] nums, int target) {

        int  start = 0, end = nums.length - 1;

        while(start <= end){

            int mid = start + (end - start)/2;

            if(nums[mid] == target){

                return mid;

            }

            if(nums[mid] < target){

                start = mid + 1;

            }

            else{

                end = mid - 1;

            }

        }

        return start;

    }

    public static void main(String[] args) {

        int [] nums = {1,3,5,6};

        int target = 5;

        System.out.println(searchInsert(nums, target));

    }

}

💡 **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]

import java.util.Arrays;

public class plusOne {

    static int[] plusOne(int[] digits) {

        int n = digits.length;

        for(int i = n-1; i >= 0; i--){

            if(digits[i] < 9){

                digits[i]++;

                return digits;

            }

            else{

                digits[i] = 0;

            }

        }

        if(digits[0] == 0){

            int arr[] = new int[n + 1];

            arr[0] = 1;

            return arr;

        }

        return digits;

    }

    public static void main(String[] args) {

        int [] nums = {9,1,2};

        System.out.println(Arrays.toString(plusOne(nums)));

    }

}

**Explanation:** The array represents the integer 123. Incrementing by one gives 123 + 1 = 124. Thus, the result should be [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

public class singleNumber2 {

    static int singleNumber(int[] nums) {

        int xor = 0;

        for(int n : nums){

            xor = xor^n;

        }

        return xor;

    }

    public static void main(String[] args) {

        int nums[] ={2,2,1};

        System.out.println(singleNumber(nums));

    }

}

💡 **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]

import java.util.\*;

public class missingRange {static List<String> findMissingRanges(int[] nums, int lower, int upper) {

    int n = nums.length;

    List<String> ans = new ArrayList<>();

    if (n == 0) {

        ans.add(f(lower, upper));

        return ans;

    }

    if (nums[0] > lower) {

        ans.add(f(lower, nums[0] - 1));

    }

    for (int i = 1; i < n; ++i) {

        int a = nums[i - 1], b = nums[i];

        if (b - a > 1) {

            ans.add(f(a + 1, b - 1));

        }

    }

    if (nums[n - 1] < upper) {

        ans.add(f(nums[n - 1] + 1, upper));

    }

    return ans;

}

static String f(int a, int b) {

    return a == b ? a + "" : a + "->" + b;

}

public static void main(String[] args) {

    int [] nums = {0,1,3,50,75};

    int lower = 0, upper = 99;

    System.out.println(findMissingRanges(nums, lower, upper));

}

}

💡 **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

import java.util.\*;

public class canAttendAllMeeting {

    static boolean canAttendMeetings(int[][] intervals) {

        int len = intervals.length;

        int[] startTime = new int[len];

        int[] endTime = new int[len];

        int count = 0;

        for(int[] interval: intervals){

            startTime[count] = interval[0];

            endTime[count++] = interval[1];

        }

        Arrays.sort(startTime);

        Arrays.sort(endTime);

        for(int i = 1; i < len; i++){

            if(startTime[i] < endTime[i - 1]) return false;

        }

        return true;

    }

    public static void main(String[] args) {

        int [][] intervals = {{0,30},{5,10},{15,20}};

        System.out.println(canAttendMeetings(intervals));

    }

}