Assignment Questions 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:-

```
class Solution {
    public int threeSumClosest(int[] nums, int target) {
     Arrays.sort(nums);
        // Length of the array
        int n = nums.length;
        // Result
        int closest = nums[0] + nums[1] + nums[n - 1];
        // Loop for each element of the array
        for (int i = 0; i < n - 2; i++) {
            // Left and right pointers
            int j = i + 1;
            int k = n - 1;
            // Loop for all other pairs
            while (j < k) {
                int sum = nums[i] + nums[j] + nums[k];
                if (sum <= target) {
                    j++;
                } else {
                    k--;
                }
                if (Math.abs(closest - target) > Math.abs(sum - target)) {
                    closest = sum;
                }
            }
        }
        return closest;
    }
}
```

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:-

```
class Solution {
    public List<List<Integer>> fourSum(int[] nums, int target) {
          List<List<Integer>> quadruplets = new ArrayList<>();
        // Base condition
        if (nums == null || nums.length < 4) {
            return quadruplets;
        }
        // Sort the array
        Arrays.sort(nums);
        // Length of the array
        int n = nums.length;
        // Loop for each element in the array
        for (int i = 0; i < n - 3; i++) {
            // Check for skipping duplicates
            if (i > 0 && nums[i] == nums[i - 1]) {
                continue;
            }
            // Reducing problem to 3Sum problem
            for (int j = i + 1; j < n - 2; j++) {
                // Check for skipping duplicates
                if (j != i + 1 && nums[j] == nums[j - 1]) {
                    continue;
                }
                // Left and right pointers
                int k = j + 1;
                int l = n - 1;
                // Reducing to two sum problem
                while (k < l) {
                    int currentSum = nums[i] + nums[j] + nums[k] + nums[l];
                    if (currentSum < target) {
                        k++;
                    } else if (currentSum > target) {
                        l--;
                    } else {
                        quadruplets.add(Arrays.asList(nums[i], nums[j], nums[k],
nums[l]));
                        k++;
                        l--;
                        // Check for skipping duplicates
                        while (k < l && nums[k] == nums[k - 1]) {
                            k++;
                        }
                        while (k < l && nums[l] == nums[l + 1]) {
                            l--;
                        }
                    }
                }
            }
        }
        return quadruplets;
    }
}
```

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:-

```
class Solution {
    public void nextPermutation(int[] nums) {

        // Length of the array
        int n = nums.length;
        // Index of the first element that is smaller than
        // the element to its right.
        int index = -1;
        // Loop from right to left
        for (int i = n - 1; i > 0; i--) {
            if (nums[i] > nums[i - 1]) {
                index = i - 1;
                break;
            }
        }
        // Base condition
        if (index == -1) {
            reverse(nums, 0, n - 1);
            return;
        }
        int j = n - 1;
        // Again swap from right to left to find first element
        // that is greater than the above find element
        for (int i = n - 1; i >= index + 1; i--) {
            if (nums[i] > nums[index]) {
```

```java
                j = i;
                break;
            }
        }
        // Swap the elements
        swap(nums, index, j);
        // Reverse the elements from index + 1 to the nums.length
        reverse(nums, index + 1, n - 1);
    }

    private static void reverse(int[] nums, int i, int j) {
        while (i < j) {
            swap(nums, i, j);
            i++;
            j--;
        }
    }

    private static void swap(int[] nums, int i, int index) {
        int temp = nums[index];
        nums[index] = nums[i];
        nums[i] = temp;
    }
    }
```

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:-

```java
class Solution {
    public int searchInsert(int[] nums, int target)
     {
         int start=0;
         int end=nums.length-1;
         while(start<=end)
```

```
        {
            int mid=(start+end)/2;
            if(nums[mid]==target)
            {
                return mid;
            }
            else if(nums[mid]>target)
            {
                return mid;
                end=mid-1;
            }
            else
            start=mid+1;
        }


                     return start;

    }
}
```

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:-

```
class Solution {
    public int[] plusOne(int[] digits)
    {
        for(int i=digits.length-1;i>=0;i--)
        {
            if(digits[i]==9)
            {
                digits[i]=0;
            }
            else
             {
                 digits[i]++;
                 return digits;
             }
        }
         digits=new int[digits.length+1];
        digits[0]=1;
        return digits;

    }
}
```

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:-

```
class Solution {
    public int singleNumber(int[] A) {
        int ar_size=A.length;
         for (int i = 0; i < ar_size; i++) {

      // Initialize count to 0
      int count = 0;
      for (int j = 0; j < ar_size; j++) {

        // Count the frequency
        // of the element
        if (A[i] == A[j]) {
          count++;
        }
      }

      // if the frequency of the
      // element is one
      if (count == 1) {
        return A[i];
      }
    }

    // if no element exist at most once
    return -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]

Solution:-

```
class Solution {
    public 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;
    }

    private String f(int a, int b) {
        return a == b ? a + "" : a + "->" + b;
    }
}
```

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:-

```
class Solution {
     public 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;
    }
}
```