# Assignment 3 (2D-Arrays)

**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).

**Sol:-**

```java
// Time Complexity: O(n^2logn)
// Space Complexity: O(1)
class Solution {
    public int threeSumClosest(int[] nums, int target) {
        Arrays.sort(nums);
        int n = nums.length;
        int closest_sum = nums[0] + nums[1] + nums[2]; // initialize closest sum
        for (int i = 0; i < n - 2; i++) {
            int left = i + 1, right = n - 1;
            while (left < right) { // two-pointer approach
                int sum = nums[i] + nums[left] + nums[right];
                if (sum == target) { // sum equals target, return immediately
                    return sum;
                } else if (sum < target) {
                    left++;
                } else {
                    right--;
                }
                  if (Math.abs(sum - target) < Math.abs(closest_sum - target)) { // update
closest sum
                    closest_sum = sum;
                }
            }
        }
        return closest_sum;
    }
}
```

**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]]

**Sol:-**

```java
// Time Complexity: O(n^3)
// Space Complexity: O(n)
```

```
class Solution {
    public List<List<Integer>> fourSum(int[] nums, int target) {
        List<List<Integer>> quadruplets = new ArrayList<>();
        int n = nums.length;
        // Sorting the array
        Arrays.sort(nums);
        for (int i = 0; i < n - 3; i++) {
            // Skip duplicates
            if (i > 0 && nums[i] == nums[i - 1]) {
                continue;
            }
            for (int j = i + 1; j < n - 2; j++) {
                // Skip duplicates
                if (j > i + 1 && nums[j] == nums[j - 1]) {
                    continue;
                }
                int left = j + 1;
                int right = n - 1;
                while (left < right) {
                    long sum = (long) nums[i] + nums[j] + nums[left] + nums[right];
                    if (sum < target) {
                        left++;
                    } else if (sum > target) {
                        right--;
                    } else {
                                quadruplets.add(Arrays.asList(nums[i],  nums[j],  nums[left],
nums[right]));
                        // Skip duplicates
                        while (left < right && nums[left] == nums[left + 1]) {
                            left++;
                        }
                        while (left < right && nums[right] == nums[right - 1]) {
                            right--;
                        }
                        left++;
                        right--;
                    }
                }
            }
        }
        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]

## Sol:-

```
class Solution{
    public void nextPermutation(int[] nums) {
    int n = nums.length;
    int i = n - 2;

    // Find the first element from the right that can be modified
    while (i >= 0 && nums[i] >= nums[i + 1]) {
        i--;
    }

    if (i >= 0) {
        // Find the smallest element greater than nums[i] to the right of nums[i]
        int j = n - 1;
        while (j >= 0 && nums[i] >= nums[j]) {
            j--;
        }

        // Swap nums[i] and nums[j]
        swap(nums, i, j);
    }

    // Reverse the elements from i+1 to the end
    reverse(nums, i + 1);
}

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

private void reverse(int[] nums, int start) {
    int i = start;
    int j = nums.length - 1;
    while (i < j) {
        swap(nums, i, j);
        i++;
        j--;
    }
}
}
```

**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

```
// Time Complexity: O(logn)
// Space Complexity: O(1)
class Solution {
    public int searchInsert(int[] nums, int target) {
       int start = 0;
       int end = nums.length - 1;
       while(start <= end){
           int mid = start + (end-start)/2;
           if(nums[mid] == target){
               return mid;
           }
           else if(target < nums[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].

## Sol:-

```
// Time Complexity: O(n)
// Space Complexity: O(n)
class Solution {
    public int[] plusOne(int[] digits) { // this is method signature
        for(int i=(digits.length)-1; i>=0;i--){ // loop iterates over elements of the array
in reverse order, starting from last digit.
            if(digits[i]<9){ // true
                digits[i] = digits[i]+1;//  adding one to the digits
                return digits;
            }else if(digits[i]==9){
                digits[i]=0; //
            }
        }
         int[] newarr = new int[digits.length+1]; // a new array (newarr) is created with a
length one greater than the original digits
        newarr[0] = 1;
        return newarr;
```

```
    }
}
// Approach:
// 1. As its already an int[] array given the last element of the array will be considered as
the units place of the number we need to add 1 for last digit if its less than 9
// 2. if the last element is greater than 9 we need to check for before element and increment
if its less than 9..
// 3. if all the elements are 9 we need add another element to the array
//for example: [9,9] --> [1,0,0]
```

**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

## Sol:-

```
// TimeComplexity: O(n)
// Space Complexity: O(1)
class Solution {
    public int singleNumber(int[] nums) {
        int result = 0;
        for(int num : nums){
            result = result ^ num; // XOR operation b/w the curr element and result variable
using the '^' operator
        }
        return 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]]

**Explanation:** The ranges are: [2,2] [4,49] [51,74] [76,99]

## Sol:-

```
// Time Complexity: O(n)
// Space Complexity: O(n)
class Solution {
    public List<String> findMissingRanges(int[] nums, int lower, int upper){
        ArrayList<String> result = new ArrayList<>();
        // traverse the whole array
        for (int i = 0; i < nums.length; i++) { // O(n)
            if (lower < nums[i]){
                if(nums[i] - lower == 1){
                result.add(lower + “”);
}else{
Result.add(lower + “->” + (nums[i] - 1));
}
            if (nums[i] == Integer.MAX_VALUE){
return result;
        }
```

```
lower = nums[i] + 1;
    }
If(lower < upper){
result.add(lower + “->” + upper)
}
Else if(lower == upper){
result.add(lower + “”);
}
return result;
}
}
```

**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

## Sol:-

```
// Time complexity: O(nlogn)
// Space Complexity: O(n)
class Solution {
    public int[][] merge(int[][] intervals) {
        Arrays.sort(intervals, (a,b)-> Integer.compare(a[0],b[0]));
        Stack<int[]> stack = new Stack();
        stack.add(intervals[0]);

        for(int i = 0; i<intervals.length; i++){
            int startPoint2 = intervals[i][0];
            int endPoint2 = intervals[i][1];
            int[] popArray = stack.pop();
            int startPoint1 = popArray[0];
            int endPoint1 = popArray[1];

            int endMax = Math.max(endPoint2, endPoint1);

            if(endPoint1 >= startPoint2){
                return false;
            }
            else{
                stack.add(intervals[i]);
            }
        }
        return false;
    }
}
```