Ques\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).

```
class Solution {
  public List<List<Integer>> threeSum(int[] nums) {
    List<List<Integer>> res= new ArrayList<>();
    Arrays.sort(nums);
    for(int i = 0; i<nums.length-2; i++){
      if(i==0 || (i>0 && nums[i] != nums[i-1])){
     int target=0 -nums[i];
     int j= i+1;
     int k = nums.length-1;

     while(j<k){
       if(nums[j]+nums[k]== target){
         ArrayList<Integer> al = new ArrayList<>();
         al.add(nums[i]);
         al.add(nums[j]);
         al.add(nums[k]);

         res.add(al);
         while(j<k && nums[j]== nums[j+1])
           j++;
         while(j>k && nums[k]== nums[k-1])
           k--;
         j++;
         k--;

       }
       else{

         if(nums[j]+nums[k]> target){

            k--;
          }else{
             j++;
       }
       }
     }
      }

    }

     return res;
    }
    }
```

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.ArrayList;
import java.util.Arrays;
import java.util.List;

public class QuadrupletSum {
    public static List<List<Integer>> fourSum(int[] nums, int target) {
        List<List<Integer>> result = new ArrayList<>();

        // Sort the array to utilize the two-pointer technique
        Arrays.sort(nums);

        int n = nums.length;

        // Fix the first and second numbers using two nested loops
        for (int i = 0; i < n - 3; i++) {
            // Skip duplicates for the first number
            if (i > 0 && nums[i] == nums[i - 1])
                continue;

            for (int j = i + 1; j < n - 2; j++) {
                // Skip duplicates for the second number
                if (j > i + 1 && nums[j] == nums[j - 1])
                    continue;
                int left = j + 1; // Pointer for the third number
                int right = n - 1; // Pointer for the fourth number

                while (left < right) {
                    int sum = nums[i] + nums[j] + nums[left] + nums[right];

                    if (sum == target) {
                        // Found a quadruplet that sums up to the target
                        result.add(Arrays.asList(nums[i], nums[j], nums[left], nums[right]));

                        // Skip duplicates for the third number
                        while (left < right && nums[left] == nums[left + 1])
                            left++;

                        // Skip duplicates for the fourth number
                        while (left < right && nums[right] == nums[right - 1])
                            right--;

                        // Move the pointers towards the center
                        left++;
                        right--;
                    } else if (sum < target) {
                        // Sum is smaller than the target, move the left pointer to increase the
sum
                        left++;
                    } else {
                        // Sum is larger than the target, move the right pointer to decrease the
sum
                        right--;
                    }
                }
            }
        }

        return result;
    }

    public static void main(String[] args) {
        int[] nums = {1, 0, -1, 0, -2, 2};
        int target = 0;
        List<List<Integer>> result = fourSum(nums, target);
        System.out.println(result); // Output: [[-2,-1,1,2],[-2,0,0,2],[-1,0,0,1]]
    }
}
```

Ques: 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]

```
public class NextPermutation {

    public static void nextPermutation(int[] nums) {

        int n = nums.length;

        int i = n - 2;


        // Find the first decreasing element

        while (i >= 0 && nums[i] >= nums[i + 1]) {
            i--;
        }

        if (i >= 0) {
            int j = n - 1;
            // Find the smallest element in the subarray nums[i+1:] that is greater than nums[i]
            while (j >= 0 && nums[j] <= nums[i]) {
                j--;
            }
            swap(nums, i, j);
        }

        // Reverse the subarray nums[i+1:]
        reverse(nums, i + 1, n - 1);
    }

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

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

    public static void main(String[] args) {
        int[] nums = {1, 2, 3};
        nextPermutation(nums);
        System.out.println(Arrays.toString(nums)); // Output: [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

```
class Solution {
  public int searchInsert(int[] nums, int target) {

    int i=0;
    int j= nums.length-1;
    while(i<=j){
      int mid= (i+j) /2;
     // System.out.println(mid);
      if(target== nums[mid]){
        return mid;
      } else if(target< nums[mid]){
        j= mid-1;
      }else
        i= mid+1;



    }
    return j+1;

  }
}
```

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].

</aside>

```java
class Solution {
  public 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;
      }
    }

    int[] res = new int[n + 1];
    res[0] = 1;

    return res;
  }
}
```

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

```java
class Solution {
  public int singleNumber(int[] nums) {
    int value=0 ;
    HashSet<Integer> set= new HashSet<>();
    for(int i: nums){
      if(!set.add(i)) set.remove(i);
      else{
        set.add(i);

      }
    }
    Iterator<Integer> iterator = set.iterator();

    if (iterator.hasNext()) {
      value = iterator.next();
    }
    return value;
  }
}
```

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]

```
package in.training.main;

import java.util.ArrayList;
import java.util.List;

public class Solution1 {
    public static List<List<Integer>> 
findMissingRanges(int[] nums, int lower, int upper) {
        List<List<Integer>> res1 = new ArrayList<>();
        int next = lower;

        for (int i = 0; i < nums.length; i++) {
            if (lower == Integer.MAX_VALUE)
                return res1;
            if (nums[i] < next) {
                continue;
            }

            if (nums[i] == next) {
                next++;
                continue;
            }

            List<Integer> res = new ArrayList<>();
            res.add(next);
            res.add(nums[i] - 1);
            res1.add(res);

            if (nums[i] == Integer.MAX_VALUE)
                return res1;

            next = nums[i] + 1;
        }

        if (next <= upper) {
            List<Integer> res = new ArrayList<>();
            res.add(next);
            res.add(upper);
            res1.add(res);
        }

        return res1;
    }

    public static void main(String[] args) {
        int[] nums = {0, 1, 3, 50, 75};
        List<List<Integer>> res1 =
findMissingRanges(nums, 0, 99);
        System.out.println(res1);
    }
}
```

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

```java
package in.training.main;

import java.util.Arrays;

public class Solution {
    public static boolean canAttendAllMeetings(int[][] intervals) {
        // Sort the intervals based on the start time
        Arrays.sort(intervals, (a, b) -> a[0] - b[0]);

        // Check for overlapping intervals
        for (int i = 1; i < intervals.length; i++) {
            // 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
            if (intervals[i][0] <= intervals[i - 1][1]) {
                return false;
            }
        }

        // No overlaps found, the person can attend all meetings
        return true;
    }

    public static void main(String[] args) {
        int[][] intervals = {{0, 30}, {5, 10}, {15, 20}};
        boolean canAttend = canAttendAllMeetings(intervals);
        System.out.println(canAttend); // Output: false
    }
}
```