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).

```
class Solution {
    public int threeSumClosest(int[] nums, int target) {
        int result=nums[0]+nums[1]+nums[nums.length-1];
        Arrays.sort(nums);
        for (int i=0;i<nums.length-2;i++) {
            int start=i+1,end=nums.length-1;
            while(start<end) {
                int sum=nums[i]+nums[start]+nums[end];
                if(sum>target) end--;
                else start++;
                if (Math.abs(sum-target)<Math.abs(result-target)) result=sum;
            }
        }
        return 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]]

```
class Solution {
    public List<List<Integer>> fourSum(int[] nums, int target) {
        Arrays.sort(nums);
        return kSum(nums, target, 0, 4);
    }

    public List<List<Integer>> kSum(int[] nums, long target, int start, int k) {
        List<List<Integer>> res = new ArrayList<>();

        // If we have run out of numbers to add, return res.
        if (start == nums.length) {
            return res;
        }
        // There are k remaining values to add to the sum. The
        // average of these values is at least target / k.
        long average_value = target / k;

        // We cannot obtain a sum of target if the smallest value
        // in nums is greater than target / k or if the largest
```

```
    // value in nums is smaller than target / k.
    if  (nums[start] > average_value || average_value > nums[nums.length - 1]) {
        return res;
    }

    if (k == 2) {
        return twoSum(nums, target, start);
    }

    for (int i = start; i < nums.length; ++i) {
        if (i == start || nums[i - 1] != nums[i]) {
            for (List<Integer> subset : kSum(nums, target - nums[i], i + 1, k - 1)) {
                res.add(new ArrayList<>(Arrays.asList(nums[i])));
                res.get(res.size() - 1).addAll(subset);
            }
        }
    }

    return res;
}

public List<List<Integer>> twoSum(int[] nums, long target, int start) {
    List<List<Integer>> res = new ArrayList<>();
    int lo = start, hi = nums.length - 1;

    while (lo < hi) {
        int currSum = nums[lo] + nums[hi];
        if (currSum < target || (lo > start && nums[lo] == nums[lo - 1])) {
            ++lo;
        } else if (currSum > target || (hi < nums.length - 1 && nums[hi] == nums[hi + 1])) {
            --hi;
```

```
            } else {
                res.add(Arrays.asList(nums[lo++], nums[hi--]));
            }
        }

        return res;
    }
}
```

<aside>

💡 **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]

</aside>

```
public class Solution {
    public void nextPermutation(int[] nums) {
        int i = nums.length - 2;
        while (i >= 0 && nums[i + 1] <= nums[i]) {
            i--;
        }
        if (i >= 0) {
            int j = nums.length - 1;
            while (nums[j] <= nums[i]) {
                j--;
            }
            swap(nums, i, j);
        }
        reverse(nums, i + 1);
    }

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

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

```
class Solution {
    public int searchInsert(int[] nums, int target) {
        // Last and First indexes
        int start = 0;
        int end = nums.length - 1;

         // Traverse an array
        while( start <= end ) {

            int mid = (start + end) / 2;

            //if target value found.
            if(nums[mid] == target) {
                return mid;
            }
```

```
            // If target value is greater then mid elements's value
            else if (target > nums[mid]) {
                start = mid + 1;
            }

            //otherwise target value is less,
             else {
                end = mid - 1;
            }
        }
        // Return the insertion position
        return end + 1;
    }
}
```

<aside>

💡 **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>

```
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;
            }
            digits[i] = 0;
        }
        int[] newNumber = new int [n+1];
        newNumber[0] = 1;
        return newNumber;
    }
}
```

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

```
class Solution {
    public int singleNumber(int[] nums) {

        // Initialize seenOnce and seenTwice to 0
        int seenOnce = 0, seenTwice = 0;

        // Iterate through nums
        for (int num : nums) {
            // Update using derived equations
            seenOnce = (seenOnce ^ num) & (~seenTwice);
            seenTwice = (seenTwice ^ num) & (~seenOnce);
        }

        // Return integer which appears exactly once
        return seenOnce;
    }
}
```

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]

```
public class Solution {
    public List<String> findMissingRanges(int[] nums, int lower, int upper) {
        List<String> result = new LinkedList<String>();
        if (nums.length == 0) {
            findRange(result, lower, upper);
            return result;
        }
        if (nums[0] != Integer.MIN_VALUE) {
```

```
            findRange(result, lower, nums[0] - 1);
        }
        for (int i = 0; i < nums.length - 1; i ++) {
            if (nums[i] == nums[i+1]) {
                continue;
            }
            findRange(result, nums[i] + 1, nums[i+1] - 1);
        }
        if (nums[nums.length - 1] != Integer.MAX_VALUE) {
            findRange(result, nums[nums.length - 1] + 1, upper);
        }

        return result;
    }
    private void findRange(List<String> result, int low, int up) {
        if (low > up){
            return;
        }
        if (low == up) {
            result.add((low) + "");
            return;
        }
        result.add(low + "->" + up);
    }
}
```

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

```
class Solution {
  public boolean canAttendMeetings(int[][] intervals) {
    Arrays.sort(intervals, new Comparator<int[]>() {
      public int compare(int[] i1, int[] i2) {
        return i1[0] - i2[0];
      }
    });
    for (int i = 0; i < intervals.length - 1; i++) {
      if (intervals[i][1] > intervals[i + 1][0])
        return false;
    }
    return true;
  }
}
```