Assignment 3 (Arrays) Solution

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.
Solution:

```
public int threeSumClosest(int[] nums, int target) {
        Arrays.sort(nums);
        int closestSum = Integer.MAX_VALUE;
        int curSum = 0;

        for (int i = 0; i <nums.length - 2; i++) {
            int left = i + 1;
            int right=nums.length - 1;

            while (left<right) {
                curSum = nums[i] + nums[left] + nums[right];
                if (curSum==target) {
                    return curSum;
                }
                else if (Math.abs(target - curSum) < Math.abs(target -
closestSum)) {
                    closestSum=curSum;
                }
                if (curSum <= target) {
                    left += 1;
                    while (nums[left] == nums[left - 1] && left < right) {
                        left += 1;
                    }
                } else {
                    right -= 1;
                }
            }
        }
        return closestSum;
    }
```

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.
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.

Solution:

```
 public void nextPermutation(int[] nums) {
        int ind1=-1;
        int ind2=-1;
        // step 1 find breaking point
        for(int i=nums.length-2;i>=0;i--){
            if(nums[i]<nums[i+1]){
                ind1=i;
                break;
            }
        }
        // if there is no breaking  point
        if(ind1==-1){
            reverse(nums,0);
        }

        else{
            // step 2 find next greater element and swap with ind2
            for(int i=nums.length-1;i>=0;i--){
                if(nums[i]>nums[ind1]){
                    ind2=i;
                    break;
                }
            }


            swap(nums,ind1,ind2);

            // step 3 reverse the rest right half

            reverse(nums,ind1+1);

        }

    }

    void swap(int[] nums,int i,int j){

        int temp=nums[i];

        nums[i]=nums[j];

        nums[j]=temp;

    }

    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.

Solution:

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

    int low = 0;

       int high = nums.length-1;

       int mid = (low + high )/2;

       while(low<=high){

           mid = (low + high)/2;

           if(nums[mid] == target){

               return mid;

           }

           if(nums[mid] > target){

               high = mid - 1;

           }

           if(nums[mid] < target){

               low = mid + 1;

           }

       }

       if(nums[mid] > nums[nums.length - 1]){

           return nums.length;

       }

       if(nums[mid] < target){
           return ++mid;
       }
       return mid;
   }
```

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.

Solution:

```
 public int[] plusOne(int[] digits) {


        if(digits[digits.length - 1] == 9){



            for(int i = digits.length-1;i>=0;i--){
                if(digits[i] == 9){
                    digits[i] = 0;
                }else{
                    digits[i]++;
                    return digits;
                }
            }
            if(digits[0] == 0){
                int[] arr = new int[digits.length + 1];
                arr[0] = 1;
                for(int i =  1;i<arr.length;i++){
                    arr[i] = 0;
                }
                return arr;
            }
        }else{
            digits[digits.length -1]++;
            return digits;
        }

        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.

Solution:

```
 public int singleNumber(int[] nums) {
        Arrays.sort(nums);
        for(int i =0;i<nums.length;i++){
            if(nums.length==1){
                return nums[i];
            }
            if(i==nums.length-1){
                return nums[i];
            }
            if(nums[i]==nums[i+1]&&nums[i+1]==nums[i+2]){
                i =i+2;
            }
            else{
                return nums[i];
        }
        }
        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.

Solution:

```
 public List<String> summaryRanges(int[] nums) {
        List<String> result = new ArrayList<String>();
```

```
if(nums == null || nums.length==0)
    return result;
if(nums.length==1)
{
    result.add(nums[0]+"");
}
int pre = nums[0]; // previous element
int first = pre; // first element of each range
for(int i=1; i<nums.length; i++)
{
    if(nums[i]==pre+1)
    {
        if(i==nums.length-1)
        {
            result.add(first+"->"+nums[i]);
        }
    }
    else
    {
        if(first == pre)
        {
            result.add(first+"");
        }
```

```
            else

            {

               result.add(first + "->"+pre);

            }

            if(i==nums.length-1)

            {

               result.add(nums[i]+"");

            }

            first = nums[i];

         }

         pre = nums[i];

      }

      return result;


   }
```

Question 8 Given an array of meeting time intervals where intervals[i] = [starti, endi], determine if a person could attend all meetings.

Solution:

```
static boolean canAttendAllMeetings(int[][] meetings){

      ArrayList<Integer> list = new ArrayList<>();


      for(int i = 0;i<meetings.length;i++){
        for(int j = 0;j<meetings[i].length;j++){

            list.add(meetings[i][j]);

        }

    }


    for (int i = 0; i < list.size()-1; i++) {

        if(list.get(i+1) < list.get(i)){

            return false;

        }

    }


    return true;


}
```