Arrays - Solutions

NOTE - Only functions will be given in the code. Please complete the entire code by writing themain function & class on your own.

Question 1 (DSA Sheet #4)

Approach 1 - Brute Force (O(n^2))

```
public boolean containsDuplicate(int[] nums) {
   for(int i=0; i<nums.length-1 ; i++) {
       for(int j=i+1; j<nums.length ; j++ ) {
          if( nums[i] == nums[j] ) {
              return true ;
           }
       }
   }
   return false;
}
```

Approach 2 - using Sets (O(n))

/* You should have a basic knowledge about Java HashSets before following Approach 2. It willbe taught to you in later chapters. */

```
public boolean containsDuplicate(int[] nums) {
   HashSet<Integer> set = new HashSet<>();
   for(int i=0; i<nums.length; i++) {
       if(set.contains(nums[i])) {
           return true;
       } else {
           set.add(nums[i]);
       }
   }

   return false;
}
```

Question 2 (DSA Sheet #6)

Approach - Based on Binary Search

```
public int search(int[] nums, int target) {
    //min will have index of  minimum element of nums
    int min = minSearch(nums);
    //find in sorted left
    if(nums[min] <= target && target <= nums[nums.length-1]){
         return search(nums,min,nums.length-1,target);
    }
    //find in sorted right
    else{
        return search(nums,0,min,target);
    }
}

//binary search to find target in left to right boundary
public int search(int[] nums,int left,int right,int target){
    int l = left;
    int r = right;
    // System.out.println(left+" "+right);
    while(l <= r){
        int mid = l + (r - l)/2;
        if(nums[mid] == target){
            return mid;
        }
        else if(nums[mid] > target){
            r = mid-1;
        }
        else{
            l = mid+1;
        }
    }
    return -1;
}

//smallest element index
public int minSearch(int[] nums){
    int left = 0;
    int right = nums.length-1;
```

```
    while(left < right){
        int mid = left + (right - left)/2;
        if(mid > 0 && nums[mid-1] > nums[mid]){
            return mid;
        }
        else if(nums[left] <= nums[mid] && nums[mid] > nums[right]){
            left = mid+1;
        }
        else{
            right = mid-1;
        }
    }

    return left;
}
```

Question 3 (DSA Sheet #8)

Approach

```
public int maxProfit(int[] prices) {
    int buy = prices[0];
    int profit = 0;

    for (int i=1; i<prices.length; i++) {
        if (buy < prices[i]) {
            profit = Math.max(prices[i] - buy, profit);
        }
        else {
            buy = prices[i];
        }
    }

    return profit;
}
```

Question 4 (DSA Sheet #11)

/* This problem can be a little difficult for beginners to solve. Please analyze the solution if youare not able to come up with the code. */

Approach

```
public int trap(int[] height) {
    int n = height.length;

    int res = 0, l = 0, r = n - 1;
    int rMax = height[r], lMax = height[l];

    while (l < r) {
        if (lMax < rMax) {
            l++;
            lMax = Math.max(lMax, height[l]);
            res += lMax - height[l];
        } else {
            r--;
            rMax = Math.max(rMax, height[r]);
            res += rMax - height[r];
        }
    }

    return res;
}
```

Question 5 (DSA Sheet #16)

/* This problem uses List to return the numbers & HashSets to store them. These are new data structures that we will study about in later chapters. */

Approach

Let us try to understand the problem statement. The first part of the problem statement is clear, we are asked to find out all the triplets in the given array whose sum is equal to zero. A triplet is nothing but a set of three numbers in the given array. For example, if nums=[1,2, 3,4] is the given array, [1,2,3] [2,3,4] [1,3,4] etc are its triplets.

What does the condition i != j, i != k, and j != k mean?

It means that we are not allowed to reuse any number from the array within a triplet. Example, for the given array nums = [1,2,3,4], triplets [1,1,1] or [1,1,2] or [1,2,2] etc are not considered validtriplets.

The last condition that we need to consider is that we cannot have duplicate triplets in our final result. Example if [-2,-2,0,2] is the given array, we can only consider one of [-2,0,2] from indexes 0, 2, 3 and [-2,0,2] from indexes 1, 2, 3 in our final result.

BRUTE FORCE - The simple solution to this problem is to find every possible triplet from the given array, see if its sum is equal to zero and return the result (ensuring there are no duplicatetriplets in the result).

This algorithm involves the following steps:

1. **Use three loops to generate all possible triplets for the given array, with each loop variable keeping track of 1 triplet element each.**
2. **Next we calculate the sum for each triplet generated in step 1.**
3. **If the sum is equal to 0 we need to check if it is a unique triplet (not already in our result set). We can ensure the triplets in our result set are unique by sorting the triplets and adding it to a hashmap (hashmap overwrites data if the same value iswritten to the same key multiple times thereby eliminating duplicates).**
4. **Once we have added all the triplets whose sum is equal to 0 into the hashmap, we iterate through the hashmap and add it to our result array.**
5. **Finally we return the result array.**


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


   for(int i=0; i<nums.length; i++) {
       for(int j=i+1; j<nums.length; j++) {
           for(int k=j+1; k<nums.length; k++) {
               if(nums[i] + nums[j] + nums[k] == 0) {
```

```
                    List<Integer> triplet = new ArrayList < Integer > ();
                    triplet.add(nums[i]);
                    triplet.add(nums[j]);
                    triplet.add(nums[k]);
                    Collections.sort(triplet);
                    result.add(triplet);
                }
            }
        }
    }
    result = new ArrayList<List<Integer>> (new LinkedHashSet<List<Integer>> (result));
    return result;
}
```