Assignment Questions 18

1. Merge Intervals.
Given an array of intervals where intervals[i] = [starti, endi], merge all overlapping intervals, and return an array of the non-overlapping intervals that cover all the intervals in the input.
Example 1:
Input: intervals = [[1,3],[2,6],[8,10],[15,18]]
Output: [[1,6],[8,10],[15,18]]
Explanation: Since intervals [1,3] and [2,6] overlap, merge them into [1,6].
Example 2:
Input: intervals = [[1,4],[4,5]]
Output: [[1,5]]
Explanation: Intervals [1,4] and [4,5] are considered overlapping.
Constraints:
- `1 <= intervals.length <= 10000`
- `intervals[i].length == 2`
- `0 <= starti <= endi <= 10000`

code:-

```
class Solution {
    public int[][] merge(int[][] intervals) {
        Arrays.sort(intervals, (a, b) -> Integer.compare(a[0], b[0]));
        LinkedList<int[]> merged = new LinkedList<>();
        for (int[] interval : intervals) {
            // if the list of merged intervals is empty or if the current
            // interval does not overlap with the previous, simply append it.
            if (merged.isEmpty() || merged.getLast()[1] < interval[0]) {
                merged.add(interval);
            }
        // otherwise, there is overlap, so we merge the current and previous
            // intervals.
            else {
                merged.getLast()[1] = Math.max(merged.getLast()[1], interval[1]);
            }
        }
        return merged.toArray(new int[merged.size()][]);
    }
}
```

2. Sort Colors
Given an array `nums` with `n` objects colored red, white, or blue, sort them **[in-place](https://en.wikipedia.org/wiki/In-place_algorithm)** so that objects of the same color are adjacent, with the colors in the order red, white, and blue.
We will use the integers `0`, `1`, and `2` to represent the color red, white, and blue, respectively.
You must solve this problem without using the library's sort function.
Example 1:
Input: nums = [2,0,2,1,1,0]
Output: [0,0,1,1,2,2]
Example 2:
Input: nums = [2,0,1]
Output: [0,1,2]
Constraints:
- `n == nums.length`
- `1 <= n <= 300`
- `nums[i]` is either `0`, `1`, or `2`.

code:-

```
class Solution {
    public void sortColors(int[] nums) {
     int low=0;
     int high = nums.length-1;
     int mid=0;
     while(mid<=high){
         if(nums[mid]==0){
             int temp = nums[mid];
             nums[mid] = nums[low];
             nums[low] = temp;
             low++;  mid++;
         }
         else if(nums[mid]==1){
             mid++;
         }else{
             int temp =nums[mid];
             nums[mid] = nums[high];
             nums[high] = temp;
             high--;
         }
     }
    }
}
```

3. First Bad Version Solution

You are a product manager and currently leading a team to develop a new product. Unfortunately, the latest version of your product fails the quality check. Since each version is developed based on the previous version, all the versions after a bad version are also bad.

Suppose you have `n` versions `[1, 2, ..., n]` and you want to find out the first bad one, which causes all the following ones to be bad.

You are given an API `bool isBadVersion(version)` which returns whether `version` is bad. Implement a function to find the first bad version. You should minimize the number of calls to the API.

Example 1:

Input: n = 5, bad = 4

Output: 4

Explanation:

call isBadVersion(3) -> false

call isBadVersion(5) -> true

call isBadVersion(4) -> true

Then 4 is the first bad version.

Example 2:

Input: n = 1, bad = 1

Output: 1

Constraints:

- `1 <= bad <= n <= 2^31 - 1`

code:-

```
/* The isBadVersion API is defined in the parent class VersionControl.
      boolean isBadVersion(int version); */

public class Solution extends VersionControl {
    public int firstBadVersion(int n) {
        int low = 0, high = n , mid = 0;
```

```
        while(low <= high){
            mid = low + (high - low) / 2;
            if(isBadVersion(mid)){
                high = mid - 1;
            }else{
                low = mid + 1;
            }
        }
        return low;
    }
}
```

4. Maximum Gap
Given an integer array `nums`, return the maximum difference between two successive elements in its sorted form. If the array contains less than two elements, return `0`.
You must write an algorithm that runs in linear time and uses linear extra space.
Example 1:
Input: nums = [3,6,9,1]
Output: 3
Explanation: The sorted form of the array is [1,3,6,9], either (3,6) or (6,9) has the maximum difference 3.
Example 2:
Input: nums = [10]
Output: 0
Explanation: The array contains less than 2 elements, therefore return 0.
Constraints:
- `1 <= nums.length <= 10^5`
- `0 <= nums[i] <= 10^9`

code:-

```
class Solution {
    public int maximumGap(int[] arr) {
        Arrays.sort(arr);
        int l=arr.length;
        if(l<2)
        return 0;
        int max=0;
        for(int i=0;i<l-1;i++)
        {
            int c=arr[i+1]-arr[i];
            if(c>max)
            max=c;

        }
        return max;
    }
}
```

5. Contains Duplicate
Given an integer array nums, return true if any value appears at least twice in the array, and return false if every element is distinct.
Example 1:
Input: nums = [1,2,3,1]
Output: true
Example 2:

Input: nums = [1,2,3,4]
Output: false
Example 3:
Input: nums = [1,1,1,3,3,4,3,2,4,2]
Output: true
Constraints:
- `1 <= nums.length <= 10^5`
- `109 <= nums[i] <= 10^9`

code:-

```
class Solution {
    public boolean containsDuplicate(int[] nums) {
        Set<Integer> set = new HashSet<>();
        for(int num : nums){
            if (set.contains(num)) return true;
            set.add(num);
        }
        return false;
    }
}
```

6.Minimum Number of Arrows to Burst Balloons
There are some spherical balloons taped onto a flat wall that represents the XY-plane. The balloons are represented as a 2D integer array `points` where `points[i] = [xstart, xend]` denotes a balloon whose horizontal diameter stretches between `xstart` and `xend`. You do not know the exact y-coordinates of the balloons.
Arrows can be shot up directly vertically (in the positive y-direction) from different points along the x-axis. A balloon with `xstart` and `xend` is burst by an arrow shot at `x` if `xstart <= x <= xend`. There is no limit to the number of arrows that can be shot. A shot arrow keeps traveling up infinitely, bursting any balloons in its path.
Given the array `points`, return the minimum number of arrows that must be shot to burst all balloons.

Example 1:
Input: points = [[10,16],[2,8],[1,6],[7,12]]
Output: 2
Explanation: The balloons can be burst by 2 arrows:
- Shoot an arrow at x = 6, bursting the balloons [2,8] and [1,6].
- Shoot an arrow at x = 11, bursting the balloons [10,16] and [7,12].
Example 2:
Input: points = [[1,2],[3,4],[5,6],[7,8]]
Output: 4
Explanation: One arrow needs to be shot for each balloon for a total of 4 arrows.
Example 3:
Input: points = [[1,2],[2,3],[3,4],[4,5]]
Output: 2
Explanation: The balloons can be burst by 2 arrows:
- Shoot an arrow at x = 2, bursting the balloons [1,2] and [2,3].
- Shoot an arrow at x = 4, bursting the balloons [3,4] and [4,5].
Constraints:
- `1 <= points.length <= 10^5`
- `points[i].length == 2`
- `231 <= xstart < xend <= 2^31 - 1`

code:-

```
class Solution {
    public int findMinArrowShots(int[][] points) {

        int minNumArrows = 1;
        Arrays.sort(points, new Comparator<int[]>(){
            @Override
            public int compare(int[] i1, int[] i2)
            {
                if(i1[0] < i2[0])
                    return -1;
                else if (i1[0] > i2[0])
                    return 1;
                return 0;
            }
        });

        // This is where they will trip you up ( at the merge stage )
        // Wait ... do we actually have to merge here? The intervals have been
sorted already
        // No you must merge
        // See if they can be merged
        // If mergeable - overwrite OR write into a new subintervals code ( new
ArrayList )
        // Ok ... so first we compare (a1,a2) and then next step compare (a2,a3)
        // Now if (a1,a2) had an overlap -> why not make the next a2 =
merged(a1,a2)?
        // That would do a carry over effect then
        int n = points.length;
        int[] candid = new int[2];  // always first interval anyways
        candid[0] = points[0][0];
        candid[1] = points[0][1];
        for(int i = 1; i < n; i++)
        {
            // System.out.printf("Current set = (%d,%d)\n", candid[0], candid[1]);
            int[] next = points[i];
            if(hasOverlap(candid,next))
            {
                int[] merged = mergeInterval(candid,next);
                candid[0] = merged[0];
                candid[1] = merged[1];
            }
            else
            {
                candid[0] = next[0];
                candid[1] = next[1];
                minNumArrows++;
            }
        }

        return minNumArrows;
    }

    public boolean hasOverlap(int[] i1, int[] i2)
    {
        boolean hasOverlap = false;
        if(i1[0] <= i2[0] && i2[0] <= i1[1])
            hasOverlap = true;
        if(i2[0] <= i1[0] && i1[0] <= i2[1])
```

```
            hasOverlap = true;
        return hasOverlap;
    }

    public int[] mergeInterval(int[] i1, int[] i2)
    {
        int[] merged = new int[2];
        merged[0] = Math.max(i1[0],i2[0]);
        merged[1] = Math.min(i1[1],i2[1]);
        return merged;
    }
}
```

7. Longest Increasing Subsequence
Given an integer array `nums`, return *the length of the longest strictly increasing subsequence
Example 1:
Input: nums = [10,9,2,5,3,7,101,18]
Output: 4
Explanation: The longest increasing subsequence is [2,3,7,101], therefore the length is 4.
Example 2:
Input: nums = [0,1,0,3,2,3]
Output: 4
Example 3:
Input: nums = [7,7,7,7,7,7,7]
Output: 1
Constraints:
- `1 <= nums.length <= 2500`
- `-10^4 <= nums[i] <= 10^4`

code:-

```
// Binary Search
// TC -> O(nlogn)
// SC -> O(n)

class Solution {
    public int lengthOfLIS(int[] nums) {
        int n = nums.length;
        ArrayList<Integer> list = new ArrayList<>();
        list.add(nums[0]);

        for(int i=1;i<n;i++){
            if(nums[i]>list.get(list.size()-1))
                list.add(nums[i]);
            else
                list.set(upperBound(list,nums[i]),nums[i]);
        }
        return list.size();
    }

    private int upperBound(ArrayList<Integer> list,int target){
        int i=0, j=list.size()-1;
        while(i<=j){
            int mid = (i+j)/2;

            if(list.get(mid)<=target){
```

```
                i=mid+1;
            }
            else{
                j=mid-1;
            }
        }
        return list.get(Math.max(j,0)) == target? j:i; // return last targetIndex |
insertionIndex
    }
}
```

8. 132 Pattern
Given an array of `n` integers `nums`, a 132 pattern is a subsequence of three integers `nums[i]`, `nums[j]` and `nums[k]` such that `i < j < k` and `nums[i] < nums[k] < nums[j]`.
Return `true` if there is a 132 pattern in `nums`, otherwise, return `false`.
Example 1:
Input: nums = [1,2,3,4]
Output: false
Explanation: There is no 132 pattern in the sequence.
Example 2:
Input: nums = [3,1,4,2]
Output: true
Explanation: There is a 132 pattern in the sequence: [1, 4, 2].
Example 3:
Input: nums = [-1,3,2,0]
Output: true
Explanation: There are three 132 patterns in the sequence: [-1, 3, 2], [-1, 3, 0] and [-1, 2, 0].
Constraints:
- `n == nums.length`
- `1 <= n <= 2 * 10^5`
- `-10^9 <= nums[i] <= 10^9`

code:-

```
class Solution {
    public boolean find132pattern(int[] nums) {
        int x=Integer.MIN_VALUE;
         int largest_num=0;
        Stack<Integer> s=new Stack<>();
        for(int i=nums.length-1;i>=0;i--)
      {
            largest_num=nums[i];
            if(nums[i]<x)
                return true;
            while(!s.isEmpty() && nums[i]>s.peek())
            {
                 x=s.peek();
                largest_num=Math.max(largest_num,s.peek());
                s.pop();
            }
            s.push(nums[i]);
        }
        return false;
    }
}
```