Problem 88.

**Merge Sorted Array**

Code:

class Solution {

    public void merge(int[] nums1, int m, int[] nums2, int n) {

        // Pointers for nums1, nums2, and the last index of merged array

        int i = m - 1, j = n - 1, k = m + n - 1;

        // While there are elements in both nums1 and nums2

        while (i >= 0 && j >= 0) {

            if (nums1[i] > nums2[j]) {

                nums1[k] = nums1[i];

                i--;

            } else {

                nums1[k] = nums2[j];

                j--;

            }

            k--;

        }

        // If there are remaining elements in nums2, copy them

        while (j >= 0) {

            nums1[k] = nums2[j];

            j--;

            k--;

        }

    }

    public static void main(String[] args) {

        Solution solution = new Solution();

        int[] nums1 = {1, 2, 3, 0, 0, 0};

        int m = 3;

        int[] nums2 = {2, 5, 6};

        int n = 3;

        solution.merge(nums1, m, nums2, n);

        // Output the merged array

        for (int num : nums1) {

            System.out.print(num + " ");

        }

    }

}

**Problem: 278**First Bad Version  
  
**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 left = 1;  // Start of the version range

        int right = n; // End of the version range

        // Perform binary search

        while (left < right) {

            int mid = left + (right - left) / 2; // To avoid overflow

            if (isBadVersion(mid)) {

                right = mid; // The first bad version is at mid or before

            } else {

                left = mid + 1; // The first bad version is after mid

            }

        }

        return left; // When left == right, we've found the first bad version

    }

    public static void main(String[] args) {

        Solution solution = new Solution();

        int n = 5; // Example: total 5 versions

        System.out.println("The first bad version is: " + solution.firstBadVersion(n));

    }

}

**Problem: 56**

Merge Intervals

**Code:**

import java.util.\*;

public class Solution {

    public int[][] merge(int[][] intervals) {

        // Edge case: if intervals array is empty, return empty result

        if (intervals == null || intervals.length == 0) {

            return new int[0][0];

        }

        // Step 1: Sort intervals by the start value

        Arrays.sort(intervals, (a, b) -> Integer.compare(a[0], b[0]));

        // Step 2: Initialize a list to store merged intervals

        List<int[]> merged = new ArrayList<>();

        // Step 3: Iterate through the sorted intervals

        for (int[] interval : intervals) {

            // If the list is empty or no overlap, add the current interval

            if (merged.isEmpty() || merged.get(merged.size() - 1)[1] < interval[0]) {

                merged.add(interval);

            } else {

                // Otherwise, there's an overlap, so merge the intervals

                merged.get(merged.size() - 1)[1] = Math.max(merged.get(merged.size() - 1)[1], interval[1]);

            }

        }

        // Convert the list to a 2D array and return

        return merged.toArray(new int[merged.size()][]);

    }

    public static void main(String[] args) {

        Solution solution = new Solution();

        // Example input

        int[][] intervals = {{1, 3}, {2, 4}, {5, 7}, {6, 8}, {9, 10}};

        int[][] mergedIntervals = solution.merge(intervals);

        // Output the merged intervals

        for (int[] interval : mergedIntervals) {

            System.out.println("[" + interval[0] + ", " + interval[1] + "]");

        }

    }

}