Day-1

1. Remove Element from Array (In-place)

Description:
You are given an array and a value. Remove all instances of the value in-place and return the new length of the array.

- **Do not use extra space**
- **Order of remaining elements can change**
- **It's okay to overwrite the original array beyond the new length**

Example:

```
Input: arr[] = {3, 2, 2, 3}, val = 3
Output: 2
Modified Array: {2, 2, _, _} (underscores represent removed/irrelevant values)
```

Java Code:

```
public class RemoveElement {
    public static int removeElement(int[] arr, int val) {
        int i = 0; // Pointer for the new length
        for (int j = 0; j < arr.length; j++) {
            if (arr[j] != val) {
                arr[i] = arr[j]; // Overwrite value
                i++;
            }
        }
        return i; // i is the new length
    }

    public static void main(String[] args) {
        int[] arr = {3, 2, 2, 3};
        int val = 3;
        int newLength = removeElement(arr, val);
        System.out.println("New length: " + newLength);
        System.out.print("Modified array: ");
        for (int i = 0; i < newLength; i++) {
            System.out.print(arr[i] + " ");
        }
    }
}
```

2. Move Zeroes to the End (Maintain Order)

Description:
Given an array, write a function to move all zeroes to the end **in-place**, while **maintaining the relative order** of the non-zero elements.

- Do not return a new array.
- Minimize operations.
- Must operate on the original array.

Example:

```
Input:  nums[] = {0, 1, 0, 3, 12}
Output: {1, 3, 12, 0, 0}
```

Java Code:

```java
public class MoveZeroes {
    public static void moveZeroes(int[] nums) {
        int insertPos = 0;

        // Move non-zero elements forward
        for (int num : nums) {
            if (num != 0) {
                nums[insertPos++] = num;
            }
        }

        // Fill the remaining positions with 0s
        while (insertPos < nums.length) {
            nums[insertPos++] = 0;
        }
    }

    public static void main(String[] args) {
        int[] nums = {0, 1, 0, 3, 12};
        moveZeroes(nums);
        System.out.print("After moving zeroes: ");
        for (int num : nums) {
            System.out.print(num + " ");
        }
    }
}
```

Day-2

1. Candy Distribution Problem

Problem Statement:
Given a list of `N` children with ratings, distribute candies according to the following rules:

1. Each child must receive **at least one candy**.
2. A child with a **higher rating** than their immediate neighbor must get **more candies** than that neighbor.

Your goal is to **minimize the total number of candies** distributed.

Example:

```
Input: ratings = [1, 0, 2]
Output: 5
Explanation: Candies = [2, 1, 2]

Input: ratings = [1, 2, 2]
Output: 4
Explanation: Candies = [1, 2, 1]
```

Approach: Two Pass (Greedy)

- Do a **left-to-right** pass: if the current rating is higher than the previous, increment candy count.
- Do a **right-to-left** pass: if the current rating is higher than the next, ensure the candy count is at least one more than the next.

Java Code:

```
public class CandyDistribution {
    public static int minCandies(int[] ratings) {
        int n = ratings.length;
        int[] candies = new int[n];

        // Step 1: Give 1 candy to each child
        for (int i = 0; i < n; i++) {
            candies[i] = 1;
        }

        // Step 2: Left to Right - handle increasing ratings
        for (int i = 1; i < n; i++) {
            if (ratings[i] > ratings[i - 1]) {
                candies[i] = candies[i - 1] + 1;
            }
        }

        // Step 3: Right to Left - handle decreasing ratings
        for (int i = n - 2; i >= 0; i--) {
            if (ratings[i] > ratings[i + 1]) {
                candies[i] = Math.max(candies[i], candies[i + 1] + 1);
            }
        }

        // Step 4: Calculate total
        int totalCandies = 0;
        for (int c : candies) {
            totalCandies += c;
        }

        return totalCandies;
    }

    public static void main(String[] args) {
        int[] ratings = {1, 0, 2};
        System.out.println("Minimum candies required: " + minCandies(ratings));
    }
}
```

2 Remove Duplicates from Sorted Array

Given a sorted array, remove the duplicates in place such that each element appear only once and return the new length. Do not allocate extra space for another array, you must do this in place with constant memory.

For example, given input array A = [1,1,2], your function should return length = 2, and A is now [1,2].

1.1 Analysis

The problem is pretty straightforward. It returns the length of the array with unique elements, but the original array need to be changed also. This problem is similar to Remove Duplicates from Sorted Array II

Java Solution

```
public static int removeDuplicates(int[] A) {
if (A.length < 2)
return A.length;
int j = 0;
int i = 1;
while (i < A.length) {
if (A[i] != A[j]) {
j++;
A[j] = A[i];
}
i++;
}
return j + 1;
}
```

Day-3

1. Remove Duplicates from Sorted Array II

Follow up for "Remove Duplicates": What if duplicates are allowed at most twice?

For example, given sorted array A = [1,1,1,2,2,3], your function should return length = 5, and A is now [1,1,2,2,3].

So this problem also requires in-place array manipulation.

Java Solution 1

We can not change the given array's size, so we only change the first k elements of the array which has duplicates removed.

```
public int removeDuplicates(int[] nums) {
if(nums==null){
return 0;
}
if(nums.length<3){
return nums.length;
}
int i=0;
int j=1;
while(j<nums.length){
if(nums[j]==nums[i]){
if(i==0){
i++;
j++;
}else if(nums[i]==nums[i-1]){
j++;
}else{
i++;
nums[i]=nums[j];
j++;
}
}else{
i++;
nums[i]=nums[j];
j++; } }
return i+1;
}
```

Java Solution 2

```
public int removeDuplicates(int[] nums) {
if(nums==null){
return 0;
}
if (nums.length <= 2){
return nums.length;
}
/*
1,1,1,2,2,3
i j
*/
int i = 1; // point to previous
int j = 2; // point to current
while (j < nums.length) {
if (nums[j] == nums[i] && nums[j] == nums[i - 1]) {
```

```
j++;
} else {
i++;
nums[i] = nums[j];
j++;
}
}
return i + 1;
}
```

2. Median of Two Sorted Arrays

Problem Statement:

Given two **sorted arrays** `A` and `B` of sizes `m` and `n`, find the **median** of the merged sorted array without fully merging it.

- **Time Complexity Requirement:** `O(log(min(m, n)))`

Example:

```
Input: A = [1, 3], B = [2]
Output: 2.0
Merged = [1, 2, 3] → Median = 2

Input: A = [1, 2], B = [3, 4]
Output: 2.5
Merged = [1, 2, 3, 4] → Median = (2 + 3) / 2 = 2.5
```

Approach: Binary Search on the Shorter Array

- Always do binary search on the shorter array (`A`)
- Use partitioning such that:
  - Left half has `(m + n + 1) / 2` elements
  - All elements in left of partition A ≤ right of B and vice versa
- Use `maxLeft` and `minRight` to compute median

Java Code:

```java
public class MedianSortedArrays {
    public static double findMedianSortedArrays(int[] A, int[] B) {
        if (A.length > B.length) {
            return findMedianSortedArrays(B, A); // Ensure A is smaller
        }

        int m = A.length;
        int n = B.length;
        int low = 0, high = m;

        while (low <= high) {
            int i = (low + high) / 2;
            int j = (m + n + 1) / 2 - i;

            int maxLeftA = (i == 0) ? Integer.MIN_VALUE : A[i - 1];
            int minRightA = (i == m) ? Integer.MAX_VALUE : A[i];

            int maxLeftB = (j == 0) ? Integer.MIN_VALUE : B[j - 1];
            int minRightB = (j == n) ? Integer.MAX_VALUE : B[j];

            if (maxLeftA <= minRightB && maxLeftB <= minRightA) {
                if ((m + n) % 2 == 0) {
                    return (Math.max(maxLeftA, maxLeftB) + Math.min(minRightA, minRightB))
/ 2.0;
                } else {
                    return Math.max(maxLeftA, maxLeftB);
                }
            } else if (maxLeftA > minRightB) {
                high = i - 1;
            } else {
                low = i + 1;
            }
        }

        throw new IllegalArgumentException("Input arrays are not sorted.");
    }

    public static void main(String[] args) {
        int[] A = {1, 3};
        int[] B = {2};
        System.out.println("Median: " + findMedianSortedArrays(A, B));  // Output: 2.0
    }
}
```

Day-4

1. Merge Intervals

Problem Statement:
Given an array of intervals where each interval is represented as `[start, end]`, merge all overlapping intervals and return an array of the non-overlapping intervals that cover all the intervals in the input.

Example:

```
java
CopyEdit
Input: intervals = [[1,3],[2,6],[8,10],[15,18]]
Output: [[1,6],[8,10],[15,18]]
```

Explanation:

- `[1,3]` and `[2,6]` overlap → merged to `[1,6]`
- `[8,10]` and `[15,18]` don't overlap → stay as they are

Approach:

1. **Sort intervals** based on the start time.
2. Initialize a list to hold merged intervals.
3. Traverse the sorted intervals and:
   - If the current interval overlaps with the last merged one → **merge**.
   - Otherwise, **add it as a new interval**.

Java Code:

```
import java.util.*;

public class MergeIntervals {
    public static int[][] merge(int[][] intervals) {
        if (intervals.length <= 1) return intervals;

        // Step 1: Sort by start time
        Arrays.sort(intervals, (a, b) -> Integer.compare(a[0], b[0]));

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

        // Step 2: Merge intervals
        int[] current = intervals[0];

        for (int i = 1; i < intervals.length; i++) {
            if (current[1] >= intervals[i][0]) {
                // Overlapping intervals: update the end
                current[1] = Math.max(current[1], intervals[i][1]);
            } else {
                // No overlap: add current to result and update current
                merged.add(current);
                current = intervals[i];
            }
        }

        // Add the last interval
        merged.add(current);

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

    public static void main(String[] args) {
        int[][] intervals = {{1,3},{2,6},{8,10},{15,18}};
        int[][] result = merge(intervals);

        System.out.print("Merged Intervals: ");
        for (int[] interval : result) {
            System.out.print("[" + interval[0] + "," + interval[1] + "] ");
        }
    }
}
```

2. Merge Sorted Array (In-place)

Problem Statement:
You are given two sorted arrays:

- `A[]` with size `m + n`, where the first `m` elements are valid, and the rest are empty space (`0` or garbage).
- `B[]` with `n` elements.

Merge `B[]` into `A[]` **in-place** such that the resulting array is sorted.

Example:

```
java
CopyEdit
Input:
A = [1, 3, 5, 0, 0, 0], m = 3
B = [2, 4, 6], n = 3
```

```
Output:
A = [1, 2, 3, 4, 5, 6]
```

Approach: Start from the End (Reverse Two-Pointer)

- Set pointers at the **end** of both arrays:
    - `i = m - 1` → end of valid elements in `A`
    - `j = n - 1` → end of `B`
    - `k = m + n - 1` → end of array `A` (space available)
- Compare and place the larger element at index `k` in A.

Java Code:

```
java
CopyEdit
public class MergeSortedArray {
    public static void merge(int[] A, int m, int[] B, int n) {
        int i = m - 1; // Pointer for end of A's valid elements
        int j = n - 1; // Pointer for end of B
        int k = m + n - 1; // Pointer for end of merged array

        // Merge from the back
        while (i >= 0 && j >= 0) {
            if (A[i] > B[j]) {
                A[k--] = A[i--];
            } else {
                A[k--] = B[j--];
            }
        }

        // If B has remaining elements
        while (j >= 0) {
            A[k--] = B[j--];
        }
    }

    public static void main(String[] args) {
        int[] A = {1, 3, 5, 0, 0, 0};
        int[] B = {2, 4, 6};
        int m = 3, n = 3;

        merge(A, m, B, n);

        System.out.print("Merged array: ");
        for (int num : A) {
            System.out.print(num + " ");
        }
    }
}
```