Q1: Merge two arrays by satisfying given constraints

Given two sorted arrays X[] and Y[] of size m and n each where m >= n and X[] has exactly n vacant cells,

merge elements of Y[] in their correct position in array X[], i.e., merge (X, Y) by keeping the sorted order.

For example,

Input: X[] = { 0, 2, 0, 3, 0, 5, 6, 0, 0 }

Y[] = { 1, 8, 9, 10, 15 } The vacant cells in X[] is represented by 0

Output: X[] = { 1, 2, 3, 5, 6, 8, 9, 10, 15 }

Merge two arrays by satisfying given constraints using

i for X[]

j for Y[]

k for merge array

Traverse through X[] and Y[] simultaneously:

If the current element of X[] is 0, replace it with the current element of Y[] and increment both i and j.Otherwise, move i to the next position in X[].

Continue until both arrays are completely traversed. Finally, if there are still elements left in Y[], append them to the end of X[].

Perform a Task (input) :

```
public class MergeArrays {
    public static void mergeArrays(int[] X, int[] Y) {
        int m = X.length;
        int n = Y.length;

        // Initialize pointers
```

```
    int i = 0; // Pointer for X[]
    int j = 0; // Pointer for Y[]
    int k = 0; // Pointer for merged array

    // Traverse through X[] and Y[] simultaneously
    while (i < m && j < n) {
        if (X[i] == 0) {
            X[i] = Y[j];
            i++;
            j++;
        } else {
            i++;
        }
    }

    // Append remaining elements of Y[] to the end of X[]
    while (j < n) {
        X[i++] = Y[j++];
    }
}

public static void main(String[] args) {
    int[] X = {0, 2, 0, 3, 0, 5, 6, 0, 0};
    int[] Y = {1, 8, 9, 10, 15};

    mergeArrays(X, Y);

    // Print the merged array
    for (int num : X) {
        System.out.print(num + " ");
    }
```

```
    }
}
```

Output : This code will output: 1 2 3 5 6 8 9 10 15.

Q2:Find maximum sum path involving elements of given arrays

Given two sorted arrays of integers, find a maximum sum path involving elements of both arrays whose sum is maximum.

We can start from either array, but we can switch between arrays only through its common elements.

For example,

Input: X = { 3, 6, 7, 8, 10, 12, 15, 18, 100 }

Y = { 1, 2, 3, 5, 7, 9, 10, 11, 15, 16, 18, 25, 50 }

The maximum sum path is: 1 —> 2 —> 3 —> 6 —> 7 —> 9 —> 10 —> 12 —> 15 —> 16 —> 18 —> 100

The maximum sum is 199

We can solve this problem using dynamic programming. Here's the Java code to find the maximum sum path involving elements of both arrays:

```
public class MaxSumPath {
    public static int maxSumPath(int[] X, int[] Y) {
        int m = X.length;
        int n = Y.length;

        int i = 0, j = 0;
        int sumX = 0, sumY = 0, result = 0;

        while (i < m && j < n) {
            if (X[i] < Y[j]) {
                sumX += X[i++];
```

```
            } else if (X[i] > Y[j]) {
                sumY += Y[j++];
            } else { // When elements are equal, consider maximum sum and reset sums
                result += Math.max(sumX, sumY) + X[i];
                sumX = 0;
                sumY = 0;
                i++;
                j++;
            }
        }

        while (i < m) { // Add remaining elements of X[]
            sumX += X[i++];
        }

        while (j < n) { // Add remaining elements of Y[]
            sumY += Y[j++];
        }

        result += Math.max(sumX, sumY); // Add maximum sum of remaining elements

        return result;
    }

    public static void main(String[] args) {
        int[] X = {3, 6, 7, 8, 10, 12, 15, 18, 100};
        int[] Y = {1, 2, 3, 5, 7, 9, 10, 11, 15, 16, 18, 25, 50};

        System.out.println("The maximum sum path is: " + maxSumPath(X, Y));
    }
}
```

This code will output: The maximum sum path is: 199

Q3:Write a Java Program to count the number of words in a string using HashMap.

Java program that counts the number of words in a string using HashMap:

import java.util.HashMap;

(1) We use a HashMap to store the words as keys and their respective counts as values.

The input string is split into words using the split() method with a regular expression \s+, which matches one or more whitespace characters.

(2) We iterate through the array of words and remove any non-alphabetic characters using replaceAll() and convert the word to lowercase.

(3) If the word is not empty after removing non-alphabetic characters, we update the count in the HashMap using getOrDefault() method.

Finally, we print the word count stored in the HashMap.

Q4:Write a Java Program to find the duplicate characters in a string.

(1) We use a HashMap to store characters as keys and their frequencies as values.

(2) We convert the string to lowercase to consider case-insensitive duplicates.

(3) We iterate through each character in the string and update the count in the HashMap.

Finally, we iterate through the HashMap to find characters with frequencies greater than 1, indicating duplicates, and print them along with their counts. We also ensure that only alphabetic characters are considered.

This is code of Java Program to count the number of words in a string using HashMap.