Merge two Sorted Arrays Without Extra Space

Problem statement: Given two sorted arrays **arr1[]** and **arr2[]** of sizes **n** and **m** in non-decreasing order. Merge them in sorted order. Modify arr1 so that it contains the first N elements and modify arr2 so that it contains the last M elements.

Examples

Brute Force Approach
Algorithm / Intuition

Solution:

In the question, it is clearly stated that the given two arrays are sorted. Based on this we will try to solve this problem.

Naive Approach (Brute-force):

This approach is not the exact solution according to the question as in this approach we are going to use an extra space i.e. an array. But it is definitely one of the solutions if the question does not contain the constraint of not using any extra space. And also this approach will help to understand the optimal approaches.

Approach:

Assume the size of the given arrays are n and m.

The steps are as follows:

1. We will first declare a third array, arr3[] of size n+m, and two pointers i.e. ***left and right***, one pointing to the first index of arr1[] and the other pointing to the first index of arr2[].
2. The two pointers will move like the following:
    1. **If arr1[left] < arr2[right]:** We will insert the element arr1[left] into the array and increase the left pointer by 1.
    2. **If arr2[right] < arr1[left]:** We will insert the element arr2[right] into the array and increase the right pointer by 1.
    3. **If arr1[left] == arr2[right]:** Insert any of the elements and increase that particular pointer by 1.
    4. **If one of the pointers reaches the end,** then we will only move the other pointer and insert the rest of the elements of that particular array into the third array i.e. arr3[].
3. If we move the pointer like the above, we will get the third array in the sorted order.
4. Now, from sorted array arr3[], we will copy first n(*size of arr1[]*) elements to arr1[], and the next m(*size of arr2[]*) elements to arr2[].

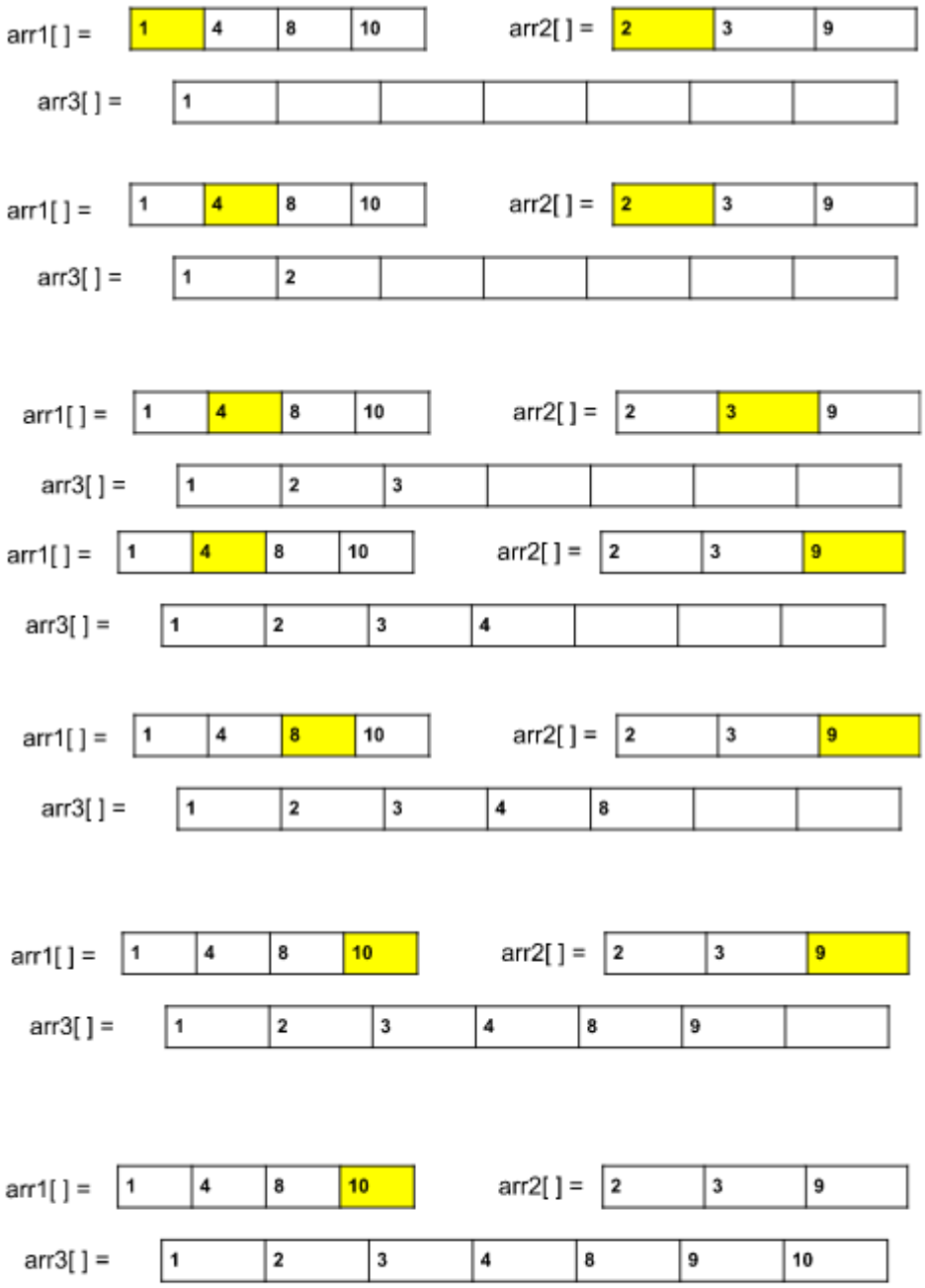
Intuition:

Intuition is pretty straightforward. As the given arrays are sorted, we are using 2 pointer approach to get a third array, that contains all the elements from the given two arrays in the sorted order. Now, from the sorted third array, we are again filling back the given two arrays.

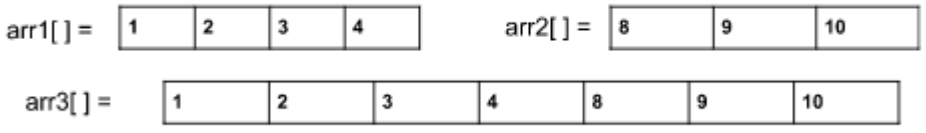
Dry Run:

The following dry run will further simplify the concept:

```
Assume arr1[] = [1 4 8 10] and arr2[] = [2 3 9].
```



Filling back arr1[] and arr2[] from arr3[]:



Note: *For a better understanding of intuition, please watch the video at the bottom of the page.*

Code

```
import java.util.*;

public class Merge {

    public static void merge(long[] arr1, long[] arr2, int n, int m) {

        // Declare a 3rd array and 2 pointers:
        long[] arr3 = new long[n + m];
        int left = 0;
        int right = 0;
        int index = 0;

        // Insert the elements from the 2 arrays
        // into the 3rd array using left and right
        // pointers:

        while (left < n && right < m) {
            if (arr1[left] <= arr2[right]) {
                arr3[index] = arr1[left];
                left++;
                index++;
            } else {
                arr3[index] = arr2[right];
                right++;
                index++;
            }
        }

        // If right pointer reaches the end:
        while (left < n) {
            arr3[index++] = arr1[left++];
        }

        // If left pointer reaches the end:
        while (right < m) {
            arr3[index++] = arr2[right++];
        }

        // Fill back the elements from arr3[]
        // to arr1[] and arr2[]:
        for (int i = 0; i < n + m; i++) {
            if (i < n) {
                arr1[i] = arr3[i];
            } else {
                arr2[i - n] = arr3[i];
            }
        }
    }

    public static void main(String[] args) {
        long[] arr1 = {1, 4, 8, 10};
        long[] arr2 = {2, 3, 9};
        int n = 4, m = 3;
        merge(arr1, arr2, n, m);
        System.out.println("The merged arrays are:");
        System.out.print("arr1[] = ");
        for (int i = 0; i < n; i++) {
            System.out.print(arr1[i] + " ");
        }
        System.out.print("\narr2[] = ");
        for (int i = 0; i < m; i++) {
            System.out.print(arr2[i] + " ");
        }
        System.out.println();
    }

}
```

Output: The merged arrays are: arr1[] = 1 2 3 4 arr2[] = 8 9 10

Complexity Analysis

Time Complexity: O(n+m) + O(n+m), where n and m are the sizes of the given arrays.
Reason: O(n+m) is for copying the elements from arr1[] and arr2[] to arr3[]. And another O(n+m) is for filling back the two given arrays from arr3[].

Space Complexity: O(n+m) as we use an extra array of size n+m.

Optimal Approach 1:
Algorithm / Intuition

Optimal Approach 1 (without using any extra space):

In this optimal approach, we need to get rid of the extra space we were using.

Approach:

The sizes of the given arrays are n(*size of arr1[]*) and m(*size of arr2[]*).

The steps are as follows:

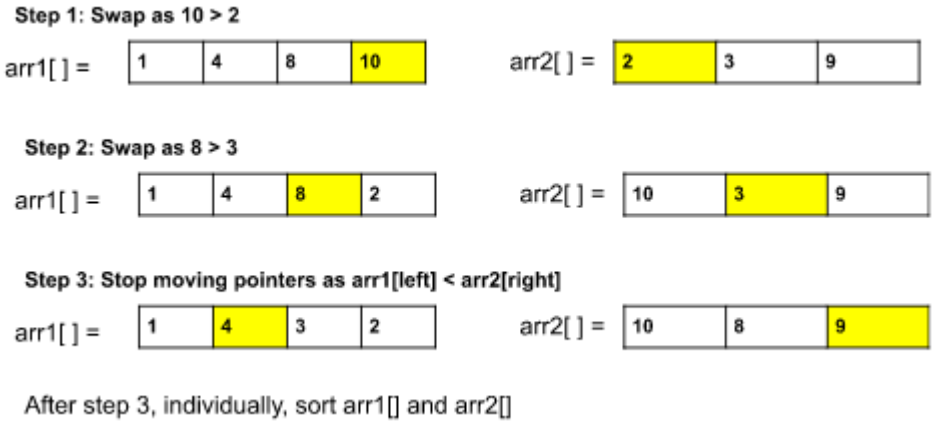
1. We will declare two pointers i.e. **left** and **right.** The left pointer will point to the last index of the arr1[](*i.e. Basically the maximum element of the array*). The right pointer will point to the first index of the arr2[](*i.e. Basically the minimum element of the array*).
2. Now, the left pointer will move toward index 0 and the right pointer will move towards the index m-1. While moving the two pointers we will face 2 different cases like the following:
    1. **If arr1[left] > arr2[right]:** In this case, we will swap the elements and move the pointers to the next positions.
    2. **If arr1[left] <= arr2[right]:** In this case, we will stop moving the pointers as arr1[] and arr2[] are containing correct elements.
3. Thus, after step 2, arr1[] will contain all smaller elements and arr2[] will contain all bigger elements. Finally, we will sort the two arrays.

Intuition:

If we merge the given array, one thing we can assure is that arr1[] will contain all the smaller elements and arr2[] will contain all the bigger elements. This is the logic we will use. Using the 2 pointers, we will swap the bigger elements of arr1[] with the smaller elements of arr2[] until the minimum of arr2[] becomes greater or equal to the maximum of arr1[].

Dry run:

The dry run will further clarify the concepts:



Note: *For a better understanding of intuition, please watch the video at the bottom of the page.*

Code

```
import java.util.*;

public class Merge {

    public static void merge(long[] arr1, long[] arr2, int n, int m) {

        // Declare 2 pointers:
        int left = n - 1;
        int right = 0;

        // Swap the elements until arr1[left] is
        // smaller than arr2[right]:
        while (left >= 0 && right < m) {
            if (arr1[left] > arr2[right]) {
                long temp = arr1[left];
                arr1[left] = arr2[right];
                arr2[right] = temp;
                left--;
                right++;
            } else {
                break;
            }
        }

        // Sort arr1[] and arr2[] individually:
        Arrays.sort(arr1);
        Arrays.sort(arr2);
    }

    public static void main(String[] args) {
        long[] arr1 = {1, 4, 8, 10};
        long[] arr2 = {2, 3, 9};
        int n = 4, m = 3;
        merge(arr1, arr2, n, m);
        System.out.println("The merged arrays are:");
        System.out.print("arr1[] = ");
        for (int i = 0; i < n; i++) {
            System.out.print(arr1[i] + " ");
        }
        System.out.print("\narr2[] = ");
        for (int i = 0; i < m; i++) {
            System.out.print(arr2[i] + " ");
        }
        System.out.println();
    }

}
```

Output: The merged arrays are: arr1[] = 1 2 3 4 arr2[] = 8 9 10

Complexity Analysis

Time Complexity: O(min(n, m)) + O(n*logn) + O(m*logm), where n and m are the sizes of the given arrays.
Reason: O(min(n, m)) is for swapping the array elements. And O(n*logn) and O(m*logm) are for sorting the two arrays.

Space Complexity: O(1) as we are not using any extra space.

Optimal Approach 2:
Algorithm / Intuition

Optimal Approach 2 (Using gap method):

This gap method is based on a sorting technique called shell sort. The intuition of this method is simple.

Intuition:

Similar to optimal approach 1, in this approach, we will use two pointers i.e. left and right, and swap the elements if the element at the left pointer is greater than the element at the right pointer.

But the placing of the pointers will be based on the gap value calculated. The formula to calculate the initial gap is the following:

Initial gap = ceil((size of arr1[] + size of arr2[]) / 2)

Assume the two arrays as a single continuous array and initially, we will place the left pointer at the first index and the right pointer at the (left+gap) index of that continuous array.

Now, we will compare the elements at the left and right pointers and move them by 1 place each time after comparison. While comparing we will swap the elements if ***the element at the left pointer > the element at the right pointer***. After some steps, the right pointer will reach the end and the iteration will be stopped.

After each iteration, we will decrease the gap and will follow the same procedure until the iteration for gap = 1 gets completed. Now, after each iteration, the gap will be the following:

gap = ceil(previous gap / 2)

The whole process will be applied to the imaginary continuous array constructed using arr1[] and arr2[].

Approach:

The steps are as follows:

1. First, assume the two arrays as a single array and calculate the gap value i.e. ceil((size of arr1[] + size of arr2[]) / 2).
2. We will perform the following operations for each gap until the value of the gap becomes 0:
    1. Place two pointers in their correct position like the left pointer at index 0 and the right pointer at index (left+gap).
    2. Again we will run a loop until the right pointer reaches the end i.e. (n+m). Inside the loop, there will be 3 different cases:
        1. **If the left pointer is inside arr1[] and the right pointer is in arr2[]:** We will compare arr1[left] and arr2[right-n] and swap them if arr1[left] > arr2[right-n].
        2. **If both the pointers are in arr2[]:** We will compare arr1[left-n] and arr2[right-n] and swap them if arr1[left-n] > arr2[right-n].
        3. **If both the pointers are in arr1[]:** We will compare arr1[left] and arr2[right] and swap them if arr1[left] > arr2[right].
    3. After the right pointer reaches the end, we will decrease the value of the gap and it will become ceil(current gap / 2).
3. Finally, after performing all the operations, we will get the merged sorted array.

Note: *For a better understanding of intuition, please watch the video at the bottom of the page.*

Code

```
import java.util.*;

public class Merge {

    public static void swapIfGreater(long[] arr1, long[] arr2, int ind1, int ind2) {
        if (arr1[ind1] > arr2[ind2]) {
            long temp = arr1[ind1];
            arr1[ind1] = arr2[ind2];
            arr2[ind2] = temp;
        }
    }

    public static void merge(long[] arr1, long[] arr2, int n, int m) {

        // len of the imaginary single array:
        int len = n + m;

        // Initial gap:
        int gap = (len / 2) + (len % 2);

        while (gap > 0) {
            // Place 2 pointers:
            int left = 0;
            int right = left + gap;
            while (right < len) {
                // case 1: left in arr1[]
                //and right in arr2[]:
                if (left < n && right >= n) {
                    swapIfGreater(arr1, arr2, left, right - n);
                }
                // case 2: both pointers in arr2[]:
                else if (left >= n) {
                    swapIfGreater(arr2, arr2, left - n, right - n);
                }
                // case 3: both pointers in arr1[]:
                else {
                    swapIfGreater(arr1, arr1, left, right);
                }
                left++; right++;
            }
            // break if iteration gap=1 is completed:
            if (gap == 1) break;

            // Otherwise, calculate new gap:
            gap = (gap / 2) + (gap % 2);
        }
    }

    public static void main(String[] args) {
        long[] arr1 = {1, 4, 8, 10};
        long[] arr2 = {2, 3, 9};
        int n = 4, m = 3;
        merge(arr1, arr2, n, m);
        System.out.println("The merged arrays are:");
        System.out.print("arr1[] = ");
        for (int i = 0; i < n; i++) {
            System.out.print(arr1[i] + " ");
        }
        System.out.print("\narr2[] = ");
        for (int i = 0; i < m; i++) {
            System.out.print(arr2[i] + " ");
        }
        System.out.println();
    }

}
```

Output: The merged arrays are: arr1[] = 1 2 3 4 arr2[] = 8 9 10

Complexity Analysis

Time Complexity: O((n+m)*log(n+m)), where n and m are the sizes of the given arrays.
Reason: The gap is ranging from n+m to 1 and every time the gap gets divided by 2. So, the time complexity of the outer loop will be O(log(n+m)). Now, for each value of the gap, the inner loop can at most run for (n+m) times. So, the time complexity of the inner loop will be O(n+m). So, the overall time complexity will be O((n+m)*log(n+m)).

Space Complexity: O(1) as we are not using any extra space.