Divide & Conquer SOLUTIONS

rohini.c.jadhav9401.com@gmail.com

Solution 1:

```
class Solution {
   //function to mergeSort 2 arrays
   public static String[] mergeSort(String[] arr, int lo, int hi) {
       if (lo == hi) {
           String[] A = { arr[lo] };
           return A;
       }
       int mid = lo + (hi - lo) / 2;
       String[] arr1 = mergeSort(arr, lo, mid);
       String[] arr2 = mergeSort(arr, mid + 1, hi);

       String[] arr3 = merge(arr1, arr2);
       return arr3;
   }

   static String[] merge(String[] arr1, String[] arr2) {
       int m = arr1.length;
       int n = arr2.length;
       String[] arr3 = new String[m + n];

       int idx = 0;

       int i = 0;
       int j = 0;

       while (i < m && j < n) {
           if (isAlphabeticallySmaller(arr1[i], arr2[j])) {
               arr3[idx] = arr1[i];
               i++;
               idx++;
           }
           else {
               arr3[idx] = arr2[j];
               j++;
               idx++;
           }
```

```
        }
        while (i < m) {
            arr3[idx] = arr1[i];
            i++;
            idx++;
        }
        while (j < n) {
            arr3[idx] = arr2[j];
            j++;
            idx++;
        }
        return arr3;
    }

    // Return true if str1 appears before str2 in alphabetical order
    static boolean isAlphabeticallySmaller(String str1, String str2) {
        if (str1.compareTo(str2) < 0) {
            return true;
        }
        return false;
    }
    public static void main(String[] args) {
        String[] arr = { "sun", "earth", "mars", "mercury" };
        String[] a = mergeSort(arr, 0, arr.length - 1);
        for (int i = 0; i < a.length; i++) {
            System.out.println(a[i]);
        }
    }
}
```

Solution 2:

Approach 1 - Brute Force Approach

Idea : Count the number of times each number occurs in the array & find the largest count.

Time complexity - O(n^2)

```
class Solution {
    public static int majorityElement(int[] nums) {
        int majorityCount = nums.length/2;
```

rohini.c.jadhav9401.com@gmail.com

```
        for (int i=0; i<nums.length; i++) {
            int count = 0;
            for (int j=0; j<nums.length; j++) {
                if (nums[j] == nums[i]) {
                    count += 1;
                }
            }
            if (count > majorityCount) {
                return nums[i];
            }
        }
        return -1;
    }
    public static void main(String args[]) {
        int nums[] = {2,2,1,1,1,2,2};
        System.out.println(majorityElement(nums));
    }
}
```

Approach 2 - Divide & Conquer

Idea : If we know the majority element in the left and right halves of an array, we can determine which is the global majority element in linear time.

Here, we apply a classical divide & conquer approach that recurses on the left and right halves of an array until an answer can be trivially achieved for a length-1 array. Note that because actually passing copies of subarrays costs time and space, we instead pass lo and hi indices that describe the relevant slice of the overall array. In this case, the majority element for a length-1 slice is trivially its only element, so the recursion stops there. If the current slice is longer than length-1, we must combine the answers for the slice's left and right halves. If they agree on the majority element, then the majority element for the overall slice is obviously the same[1]. If they disagree, only one of them can be "right", so we need to count the occurrences of the left and right majority elements to determine which subslice's answer is globally correct. The overall answer for the array is thus the majority element between indices 0 and n.

Time complexity - O(n*logn)

```
class Solution {
    private static int countInRange(int[] nums, int num, int lo, int hi) {
        int count = 0;
        for (int i = lo; i <= hi; i++) {
            if (nums[i] == num) {
```

rohini.c.jadhav9401.com@gmail.com

```
                count++;
            }
        }
        return count;
    }

    private static int majorityElementRec(int[] nums, int lo, int hi) {
        // base case; the only element in an array of size 1 is the majority
        // element.
        if (lo == hi) {
            return nums[lo];
        }

        // recurse on left and right halves of this slice.
        int mid = (hi-lo)/2 + lo;
        int left = majorityElementRec(nums, lo, mid);
        int right = majorityElementRec(nums, mid+1, hi);

        // if the two halves agree on the majority element, return it.
        if (left == right) {
            return left;
        }

        // otherwise, count each element and return the "winner".
        int leftCount = countInRange(nums, left, lo, hi);
        int rightCount = countInRange(nums, right, lo, hi);

        return leftCount > rightCount ? left : right;
    }

    public static int majorityElement(int[] nums) {
        return majorityElementRec(nums, 0, nums.length-1);
    }

    public static void main(String args[]) {
        int nums[] = {2,2,1,1,1,2,2};
        System.out.println(majorityElement(nums));
    }
}
```

rohini.c.jadhav9401.com@gmail.com

(You can also find this problem at - https://leetcode.com/problems/majority-element/)

Solution 3 :

Approach 1 - Brute Force Approach

Idea : Traverse through the array, and for every index, find the number of smaller elements on its right side of the array. This can be done using a nested loop. Sum up the counts for all indexes in the array and print the sum.

- Traverse through the array from start to end
- For every element, find the count of elements smaller than the current number up to that index using another loop.
- Sum up the count of inversion for every index.
- Print the count of inversions.

Time complexity - O(n^2)

```
class Solution {
   public static int getInvCount(int arr[]) {
       int n = arr.length;
       int invCount = 0;
       for (int i = 0; i < n - 1; i++) {
           for (int j = i + 1; j < n; j++) {
               if (arr[i] > arr[j]) {
                   invCount++;
               }
           }
       }
       return invCount;
   }
   public static void main(String[] args) {
       int arr[] = {1, 20, 6, 4, 5};
       System.out.println("Inversion Count = "+ getInvCount(arr));
   }
}
```

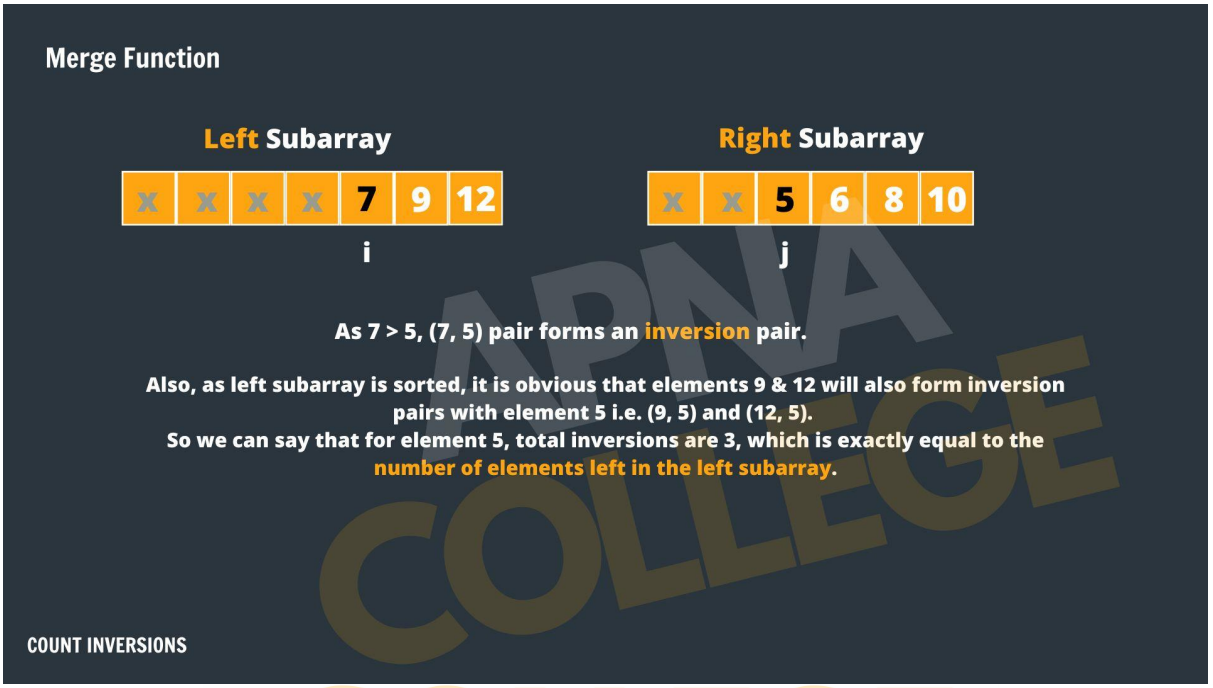
Approach 2 - Modified Merge Sort

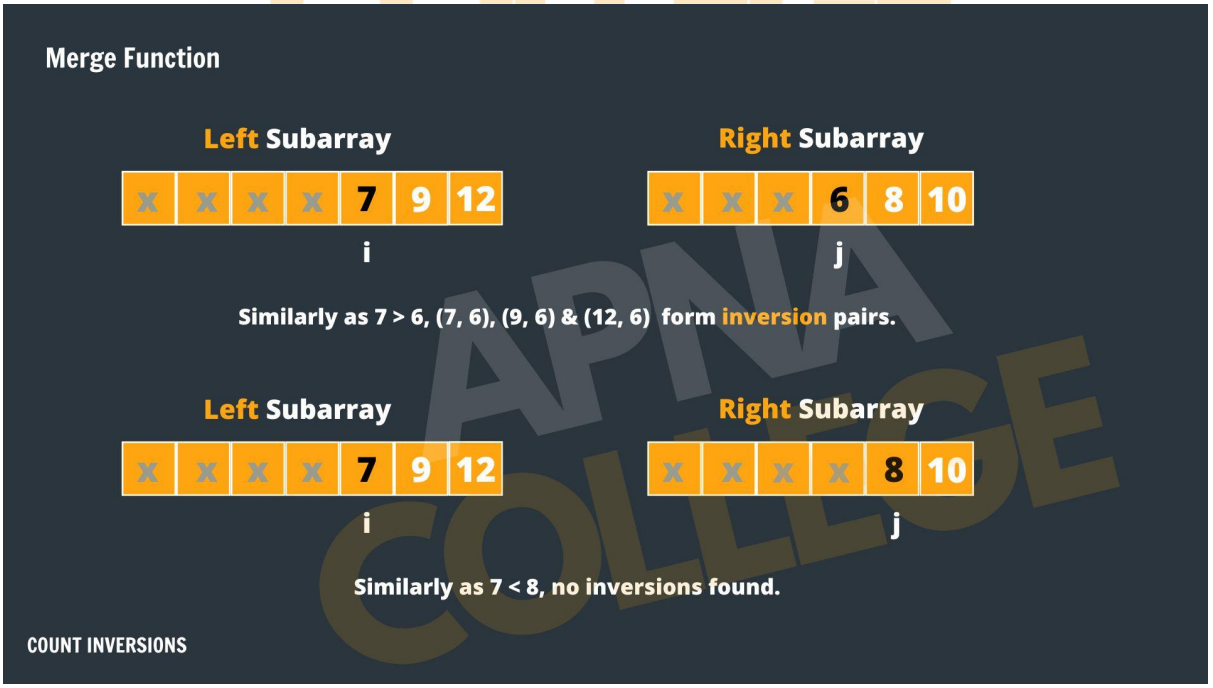
Idea : Suppose the number of inversions in the left half and right half of the array (let be inv1 and inv2); what kinds of inversions are not accounted for in Inv1 + Inv2? The answer is – the inversions that need to be counted during the merge step. Therefore, to get the total number of inversions that need to be added are the number of inversions in the left subarray, right subarray, and merge().

rohini.c.jadhav9401.com@gmail.com

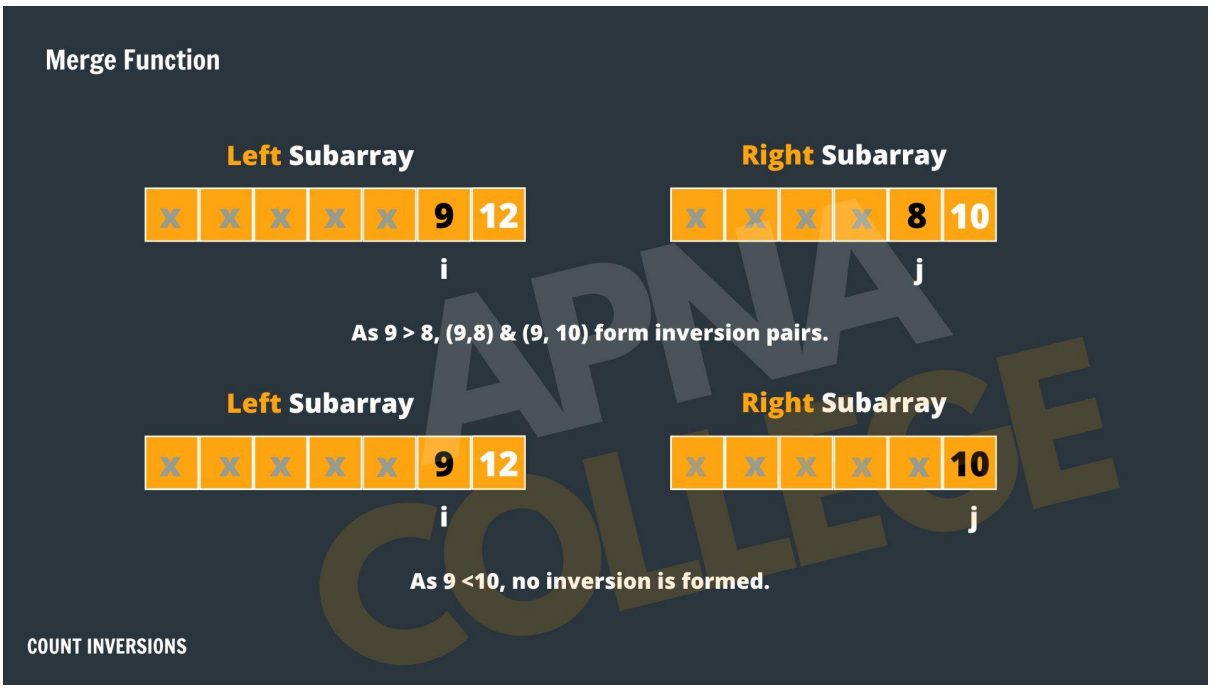
Basically, for each array element, count all elements more than it to its left and add the count to the output. This whole magic happens inside the merge function of merge sort.

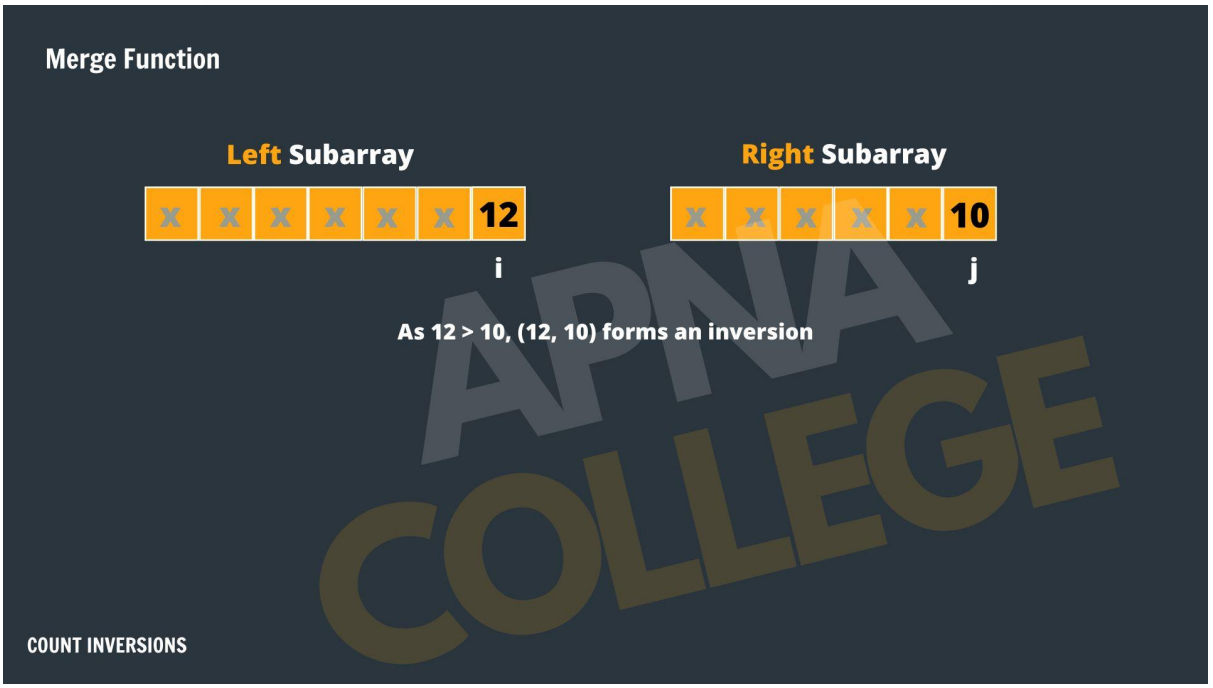
Let's consider two subarrays involved in the merge process:





rohini.c.jadhav9401.com@gmail.com





Algorithm:

- Split the given input array into two halves, left and right similar to merge sort recursively.
- Count the number of inversions in the left half and right half along with the inversions found during the merging of the two halves.
- Stop the recursion, only when 1 element is left in both halves.
- To count the number of inversions, we will use a two pointers approach. Let us consider two pointers i and j, one pointing to the left half and the other pointing towards the right half.

rohini.c.jadhav9401.com@gmail.com

- While iterating through both the halves, if the current element A[i] is less than A[j], add it to the sorted list, else increment the count by mid – i.

Time complexity - O(n* logn)

```java
public class Solution {
   public static int merge(int arr[], int left, int mid, int right) {
       int i = left, j = mid, k = 0;
       int invCount = 0;
       int temp[] = new int[(right - left + 1)];

       while ((i < mid) && (j <= right)) {
           if (arr[i] <= arr[j]) {
               temp[k] = arr[i];
               k++;
               i++;
           }
           else {
               temp[k] = arr[j];
               invCount += (mid - i);
               k++;
               j++;
           }
       }

       while (i < mid) {
           temp[k] = arr[i];
           k++;
           i++;
       }

       while (j <= right) {
           temp[k] = arr[j];
           k++;
           j++;
       }

       for (i = left, k = 0; i <= right; i++, k++) {
           arr[i] = temp[k];
       }
```

rohini.c.jadhav9401.com@gmail.com

```
        return invCount;
    }

    private static int mergeSort(int arr[], int left, int right) {
        int invCount = 0;

        if (right > left) {
            int mid = (right + left) / 2;

            invCount = mergeSort(arr, left, mid);
            invCount += mergeSort(arr, mid + 1, right);
            invCount += merge(arr, left, mid + 1, right);
        }
        return invCount;
    }

    public static int getInversions(int arr[]) {
        int n = arr.length;
        return mergeSort(arr, 0, n - 1);
    }

    public static void main(String args[]) {
        int arr[] = {1, 20, 6, 4, 5};
        System.out.println("Inversion Count = " + getInversions(arr));
    }
}
```

rohini.c.jadhav9401.com@gmail.com