Count Inversions

Given an array of integers. Find the Inversion Count in the array.

***Inversion Count**:* For an array, inversion count indicates how far (or close) the array is from being sorted. If array is already sorted then the inversion count is 0. If an array is sorted in the reverse order then the inversion count is the maximum.
Formally, two elements a[i] and a[j] form an inversion if a[i] > a[j] and i < j.

Example 1:

```
Input: N = 5, arr[] = {2, 4, 1, 3, 5}
Output: 3
Explanation: The sequence 2, 4, 1, 3, 5
has three inversions (2, 1), (4, 1), (4, 3).
```

METHOD 1 (Simple)

- **Approach:** 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 index in the array and print the sum.
- **Algorithm:**

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

```
static int getInvCount(int[] arr, int n)

{

    int inv_count = 0;

    for (int i = 0; i < n - 1; i++)

        for (int j = i + 1; j < n; j++)

            if (arr[i] > arr[j])

                inv_count++;

    return inv_count;

}
```

- **Complexity Analysis:**
    - **Time Complexity:** O(n^2), Two nested loops are needed to traverse the array from start to end, so the Time complexity is O(n^2)
    - **Space** Complexity**:**O(1), No extra space is required.

METHOD 2(Enhance Merge Sort)

Approach:
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 needs to be added are the number of inversions in the left subarray, right subarray, and merge().

- **Algorithm:**
    0. The idea is similar to merge sort, divide the array into two equal or almost equal halves in each step until the base case is reached.
    1. Create a function merge that counts the number of inversions when two halves of the array are merged, create two indices i and j, i is the index for the first half, and j is an index of the second half. if a[i] is greater than a[j], then there are (mid – i) inversions. because left and right subarrays are sorted, so all the remaining elements in left-subarray (a[i+1], a[i+2] … a[mid]) will be greater than a[j].
    2. Create a recursive function to divide the array into halves and find the answer by summing the number of inversions is the first half, the number of inversion in the second half and the number of inversions by merging the two.
    3. The base case of recursion is when there is only one element in the given half.
    4. Print the answer

```
public class GFG {

  private static int mergeAndCount(int[] arr, int l,  int m, int r)

  {

    // Left subarray

    int[] left = Arrays.copyOfRange(arr, l, m + 1);

    // Right subarray

    int[] right = Arrays.copyOfRange(arr, m + 1, r + 1);

    int i = 0, j = 0, k = l, swaps = 0;


    while (i < left.length && j < right.length) {

      if (left[i] <= right[j])

        arr[k++] = left[i++];

      else {

        arr[k++] = right[j++];

        swaps += (m + 1) - (l + i);
```

```
            }
        }
        while (i < left.length)
            arr[k++] = left[i++];
        while (j < right.length)
            arr[k++] = right[j++];
        return swaps;
    }
    // Merge sort function
    private static int mergeSortAndCount(int[] arr, int l,  int r)
    {
        // Keeps track of the inversion count at a
        // particular node of the recursion tree
        int count = 0;
        if (l < r) {
            int m = (l + r) / 2;
            // Total inversion count = left subarray count
            // + right subarray count + merge count

            // Left subarray count
            count += mergeSortAndCount(arr, l, m);
            // Right subarray count
            count += mergeSortAndCount(arr, m + 1, r);
            // Merge count
            count += mergeAndCount(arr, l, m, r);
        }
        return count;
    }
```

Complexity Analysis:

- **Time Complexity:** O(n log n), The algorithm used is divide and conquer, So in each level, one full array traversal is needed, and there are log n levels, so the time complexity is O(n log n).
- **Space** Complexity**:** O(n), Temporary array.