Count Inversions of an Array:-

Given an integer arr[] of size n, find the inversion count in the array. Two array elements arr[i] & arr[j] form an inversion if arr[i] > arr[j] & i < j.

Note:- Inversion count of an array indicates that how far (or close) the array is from being sorted. If the array is already sorted, then its IC = 0, but if the array is sorted in reverse order then the IC is maximum.

eg. I/p : arr[] = {4, 3, 2, 1}
O/p : 6

[Naive Approach]: Using Two Nested Loops - O(n²) Time & O(1) Space :-

[Naive Approach] Using Two Nested Loops – O(n^2) Time and O(1) Space

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

Solution Code :-

```cpp
// C++ program to Count Inversions in an array
// using nested loop

#include <iostream>
#include <vector>
using namespace std;

// Function to count inversions in the array
int inversionCount(vector<int> &arr) {
    int n = arr.size();
    int invCount = 0;

    // Loop through the array
    for (int i = 0; i < n - 1; i++) {
        for (int j = i + 1; j < n; j++) {

            // If the current element is greater
            // than the next, increment the count
            if (arr[i] > arr[j])
                invCount++;
        }
    }
    return invCount;
}

int main() {
    vector<int> arr = {4, 3, 2, 1};
    cout << inversionCount(arr) << endl;
    return 0;
}
```

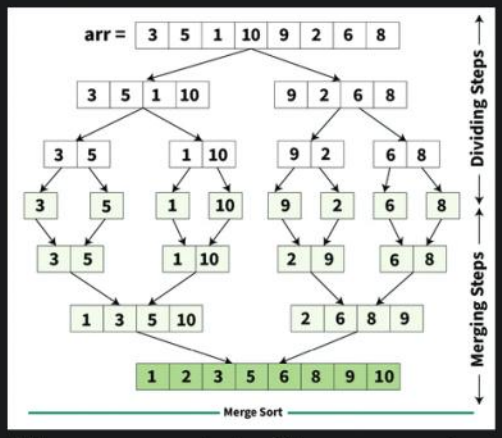
Expected Approach:-

Using Merge step of Merge Sort.

O(n log n) Time & O(n) space

*We can use merge sort to count the **inversions** in an array. First, we divide the array into two halves: **left half** and **right half.** Next, we recursively count the inversions in both halves. While merging the two halves back together, we also count how many elements from the **left half** array are greater than elements from the **right half** array, as these represent **cross inversions** (i.e., element from the left half of the array is greater than an element from the right half during the **merging process** in the **merge sort algorithm**). Finally, we sum the inversions from the **left half**, **right half**, and the **cross inversions** to get the total number of inversions in the array. This approach efficiently counts inversions while sorting the array.*

Let's understand the above intuition in more detailed form, as we get to know that we have to perform the **merge sort** on the given array. Below images represents **dividing** and **merging steps** of merge sort.



During each merging step of the merge sort algorithm, we count cross inversions by comparing elements from the left half of the array with those from the right half. If we find an element **arr[i]** in the left half that is greater than an element **arr[j]** in the right half, we can conclude that all elements after **i** in the **left half** will also be greater than **arr[j]**. This allows us to count multiple inversions at once. Let's suppose if there are **k** elements remaining in the left half after **i,** then there are **k** cross inversions for that particular **arr[j]**. The rest of the merging process continues as usual, where we combine the two halves into a sorted array. This efficient counting method significantly reduces the number of comparisons needed, enhancing the overall performance of the inversion counting algorithm.

C++ C Java Python C# JavaScript

```cpp
// C++ program to Count Inversions in an array using merge sort

#include <iostream>
#include <vector>
using namespace std;

// This function merges two sorted subarrays arr[l..m] and arr[m+1..r]
// and also counts inversions in the whole subarray arr[l..r]
int countAndMerge(vector<int>& arr, int l, int m, int r) {

    // Counts in two subarrays
    int n1 = m - l + 1, n2 = r - m;

    // Set up two vectors for left and right halves
    vector<int> left(n1), right(n2);
    for (int i = 0; i < n1; i++)
        left[i] = arr[i + l];
    for (int j = 0; j < n2; j++)
        right[j] = arr[m + 1 + j];

    // Initialize inversion count (or result) and merge two halves
    int res = 0;
    int i = 0, j = 0, k = l;
    while (i < n1 && j < n2) {

        // No increment in inversion count if left[] has a
        // smaller or equal element
        if (left[i] <= right[j])
            arr[k++] = left[i++];

        // If right is smaller, then it is smaller than n1-i
          // elements because left[] is sorted
        else {
            arr[k++] = right[j++];
```



```cpp
    }

    // Merge remaining elements
    while (i < n1)
        arr[k++] = left[i++];
    while (j < n2)
        arr[k++] = right[j++];

    return res;
}

// Function to count inversions in the array
int countInv(vector<int>& arr, int l, int r){
    int res = 0;
    if (l < r) {
        int m = (r + l) / 2;

        // Recursively count inversions in the left and
        // right halves
        res += countInv(arr, l, m);
        res += countInv(arr, m + 1, r);

        // Count inversions such that greater element is in
          // the left half and smaller in the right half
        res += countAndMerge(arr, l, m, r);
    }
    return res;
}

int inversionCount(vector<int> &arr) {
      int n = arr.size();
      return countInv(arr, 0, n-1);
}

int main(){
    vector<int> arr = {4, 3, 2, 1};

    cout << inversionCount(arr);
    return 0;
}
```

```
6
```