

**Experiment No. 6**

**Title: Inversion Count**

**Theory/Description of the Problem Statement:**

Inversion Count for an array indicates – how far (or close) the array is from being sorted. If the array is already sorted, then the inversion count is 0, but if the array is sorted in reverse order, the inversion count is the maximum.

Create a function merge that counts the number of inversions when two halves of the array are merged, and 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].

Create a recursive function to divide the array into halves and find the answer by summing the number of inversions in the first half, the number of inversions in the second half and the number of inversions by merging the two. The base case of recursion is when there is only one element in the given half. At the end print the answer.

**Algorithm:**

**Algorithm : Inversion count using merge sort**

* function mergeSort(arr, left, right):
* if left < right:
* mid = (left + right) / 2
* left\_count = mergeSort(arr, left, mid)
* right\_count = mergeSort(arr, mid + 1, right)
* merge\_count = merge(arr, left, mid, right)
* return left\_count + right\_count + merge\_count
* return 0
* function merge(arr, left, mid, right):
* i = left
* j = mid + 1
* k = 0
* temp = []
* count = 0
* while i <= mid and j <= right:
* if arr[i] <= arr[j]:
* temp[k] = arr[i]
* i += 1
* else:
* temp[k] = arr[j]
* count += mid - i + 1
* j += 1
* k += 1
* while i <= mid:
* temp[k] = arr[i]
* i += 1
* k += 1
* while j <= right:
* temp[k] = arr[j]
* j += 1
* k += 1
* for i in range(k):
* arr[left + i] = temp[i]
* return count

**Analysis of the Algorithm**

**Time Complexity**

O(n \* log n), The algorithm used is divide and conquer i.e. merge sort whose complexity is O(n log n).

Auxiliary Space: O(n), Temporary array.

**Experiment and result:**

Code**:**

// C++ program to Count

// Inversions in an array

// using Merge Sort

#include <bits/stdc++.h>

using namespace std;

int \_mergeSort(int arr[], int temp[], int left, int right);

int merge(int arr[], int temp[], int left, int mid,

        int right);

// This function sorts the

// input array and returns the

// number of inversions in the array

int mergeSort(int arr[], int array\_size)

{

    int temp[array\_size];

    return \_mergeSort(arr, temp, 0, array\_size - 1);

}

// An auxiliary recursive function

// that sorts the input array and

// returns the number of inversions in the array.

int \_mergeSort(int arr[], int temp[], int left, int right)

{

    int mid, inv\_count = 0;

    if (right > left) {

        // Divide the array into two parts and

        // call \_mergeSortAndCountInv()

        // for each of the parts

        mid = (right + left) / 2;

        // Inversion count will be sum of

        // inversions in left-part, right-part

        // and number of inversions in merging

        inv\_count += \_mergeSort(arr, temp, left, mid);

        inv\_count += \_mergeSort(arr, temp, mid + 1, right);

        // Merge the two parts

        inv\_count += merge(arr, temp, left, mid + 1, right);

    }

    return inv\_count;

}

// This function merges two sorted arrays

// and returns inversion count in the arrays.

int merge(int arr[], int temp[], int left, int mid,

        int right)

{

    int i, j, k;

    int inv\_count = 0;

    i = left;

    j = mid;

    k = left;

    while ((i <= mid - 1) && (j <= right)) {

        if (arr[i] <= arr[j]) {

            temp[k++] = arr[i++];

        }

        else {

            temp[k++] = arr[j++];

            // this is tricky -- see above

            // explanation/diagram for merge()

            inv\_count = inv\_count + (mid - i);

        }

    }

    // Copy the remaining elements of left subarray

    // (if there are any) to temp

    while (i <= mid - 1)

        temp[k++] = arr[i++];

    // Copy the remaining elements of right subarray

    // (if there are any) to temp

    while (j <= right)

        temp[k++] = arr[j++];

    // Copy back the merged elements to original array

    for (i = left; i <= right; i++)

        arr[i] = temp[i];

    return inv\_count;

}

// Driver code

int main()

{

    int arr[] = { 1, 20, 6, 4, 5 };

    int n = sizeof(arr) / sizeof(arr[0]);

    int ans = mergeSort(arr, n);

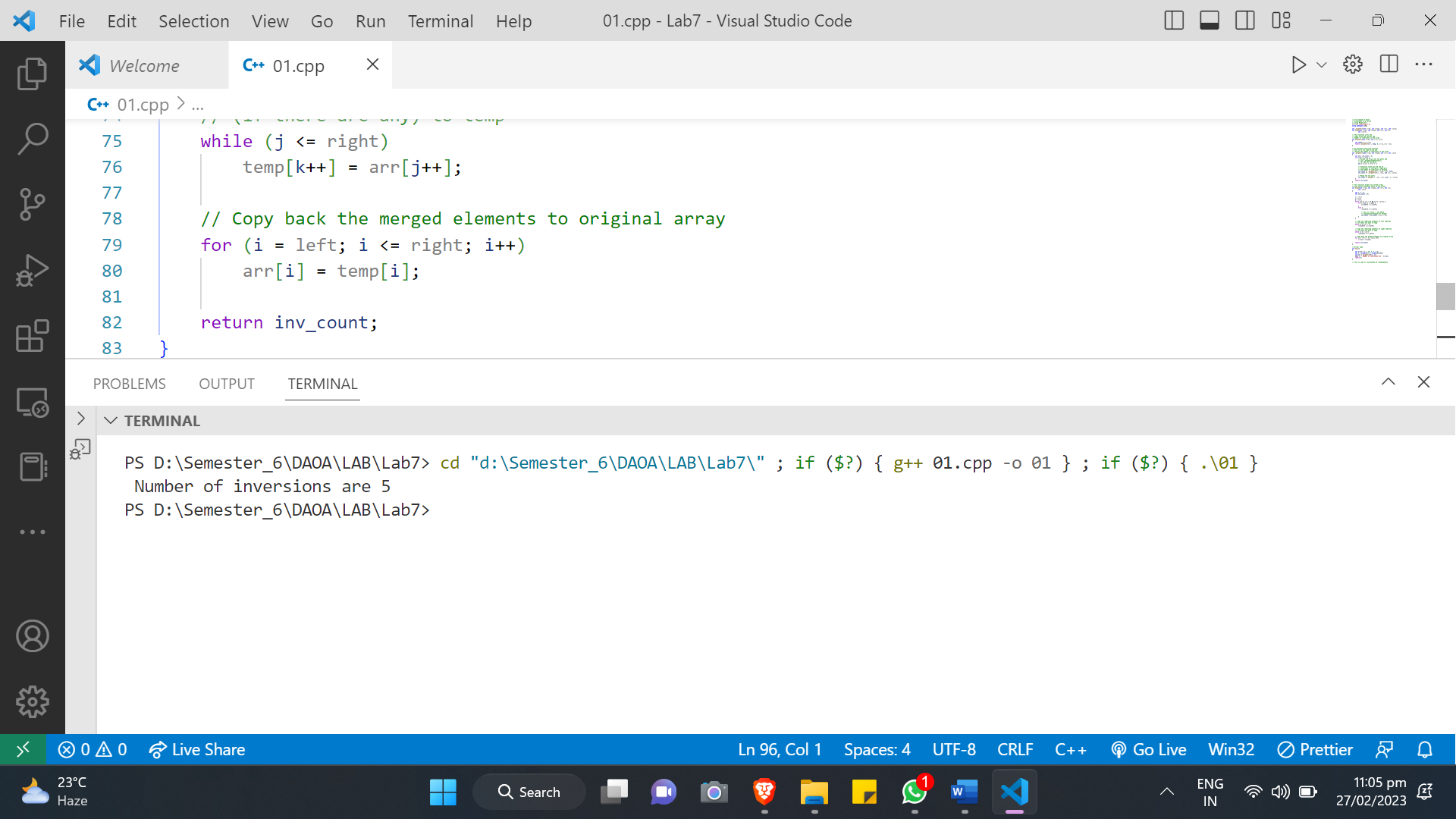
    cout << " Number of inversions are " << ans;

    return 0;

}

// This is code is contributed by rathbhupendra

Output:



**Conclusions:**

The inversion count algorithm using merge sort is an efficient way to compute the number of inversions in an array. It works by recursively dividing the array into subarrays, sorting and merging them while counting the inversions, and returning the total inversion count. The algorithm has a time complexity of O(n log n) since it uses the divide and conquer strategy of merge sort, which is faster than a brute-force approach that would take O(n^2) time. Overall, the algorithm is an excellent choice for problems that require counting the number of inversions in an array.