Counting Number of Inversions in an Array

B.tech Semester 4th

Harshdeep Singh Pruthi
IIT2019105

Jayaram Naik
IIT2019106

Shivansh Gupta
IIT2019107

***Abstract*—In this report we have devised an algorithm on how to count the Number of Inversions in an Array. We have discussed in detail the Divide and Conquer Approach for this problem and discussed why it works and have compared it with the Naive Approach, on what makes it better.**

***Index Terms*—Inversion, Divide and Conquer, Merge Sort**

I. Introduction

Consider an array A, and two indices i and j, such that $i < j$ and $A[i] > A[j]$. This is called an inversion in the array. Thus, the Inversion Count of an array is the count of all the pairs (i, j) such that $i < j$ and $A[i] > A[j]$

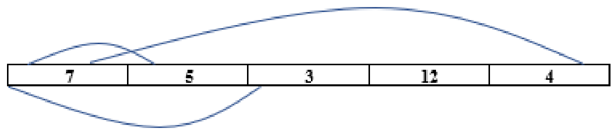

Fig. 1. An example of inversion in array.

Thus, in the above figure indices $(0, 1), (0, 2), (0, 4)$ are called inversions.
An important observation here is that Inversion Count can be seen to be closely associated with Sorting of an array. If we find out all the inversions in an array and replace the elements at the indices i and j to the point that no inversion now exists, the array will be sorted.
Thus, Inversion Count for an array indicates – how far the array is from being sorted. If the array is already sorted, then the inversion count is 0, and it would be maximum when the array is sorted in decreasing order.
This relationship between inversions and a sorted array is going to be the key for devising a Divide and Conquer algorithm for this problem which shall follow in the subsequent sections.

II. Algorithm Description and Analysis

A. Naive

The Naive approach is fairly simple. It involves a Brute Force approach where we iterate over all the pairs of the array (which can be done through a nested loop), and if any pair (i, j) satisfies our condition, we increase our Inversion Count by one.

B. Divide and Conquer

For the Divide and Conquer approach, suppose we divide our array A into two equal, or almost equal parts. Let's call the first one **L** and the other one **R**. Also, let's say we know the Inversion Count of both **L** and **R**. Let's call them inv_1 and inv_2.
Now, suppose we want to count the number of inversions inside the array A. Now any inversion (i, j) in **A** would be of one of the following types:

1) $i \in L$ and $j \in R$.
2) $i \in L$ and $j \in L$.
3) $i \in R$ and $j \in R$.

- Let's consider the first case. First of all, it's clear that the total inversions of **A** now will include the inversions of both **L** and **R** ,i.e., $inv_1 + inv_2$. But there may be some more inversions, for when we will $merge$ **L** and **R**. And that is the major step in this case - computing for each index $i \in L$, the number of indices $j \in R$ such that $L_i > R_j$.
  Now, let's say that we have sorted the arrays **L** and **R**. and their sizes are **N** and **M** respectively.
  Also, now consider we are using k and l to iterate over **L** and **R**. Say, at certain point we encounter $L[l] > R[k]$. Now, because **L** and **R** are sorted, we know that there are going to be **N-i** more inversions in **A**, because all elements to the right of $L[l]$ will also be greater than $R[k]$.
  Note that it doesn't matter that the elements are now sorted, because all the elements of **L** existed to the left of all elements **R** in the original array **A**, so these inversions will be present in one or another order in **A**.

- Now for the second and the third case: If we encounter that situation, we again divide them up into further two arrays and repeat this recursively, until we have arrived on the base case - that includes just one element. And in this case, we know the inversions will be 0, because there is just a single element in the array.

Here, one more important thing is that in the first case, we require the arrays **L** and **R** to be sorted. To achieve that, the algorithm must sort them. In other words, after applying the algorithm to both L and R, they become sorted. To sort **A** we just merge the two sorted arrays **L** and **R** into **A**.

III. Pseudo Code

We will be creating two functions: **enhancedMergeSort** and **mergeTwoArrays**. The former will divide up the array recursively into halves and call the latter function for these divided arrays and the latter function will sort the arrays and in the process return the Inversion Count which is ultimately added up to return the final answer.

Algorithm 1: mergeTwoArrays

```
Data: arr: The Array
beg: Start iterator
mid: Middle iterator
end: End iterator
Result: Returns the total number of inversions in the
        procedure of merging two arrays
        $arr[beg...mid]$ and $arr[mid+1...end]$
1  $i, k = beg$
2  $j = end$
3  $newarr$
4  while $i < mid$ AND $j <= end$ do
       /* No Inversion */
5      if $arr[i] < arr[j]$ then
6          newarr[k]=arr[i]
7          increment k,i
       /* mid-i inversions */
8      else
9          newarr[k]=arr[j]
10         invCount+= $mid - i$
11         increment k,j
12 while $i < mid$ do
13     temp[k] = arr[i]
14     increment k,i
15 while $j, end$ do
16     temp[k] = arr[j]
17     increment k,j
18 it = beg /* Copy all the elements of
       newarr into arr */
19 while $it < end$ do
20     arr[it] = newarr[it] increment it
21 return $invCount$
```

IV. Complexity

A. Time complexity

In the Naive Approach, for any index i, there are **N-i** iterations. Thus the total number of iterations will be the sum of **N,N-1,N-2,...1** ,i.e., $\frac{N(N-1)}{2}$ In the algorithm , the input array is divided into two halves each time it is processed. As such it can be expressed with following recurrence relation,

$$T(n) = 2T(n/2) + \theta(n).$$

Hence, time complexity for the same is O(nlog(n)).

Algorithm 2: enhancedMergeSort

```
Data: arr: The Array
beg: Start iterator
end: End iterator
Result: Returns the total number of inversions in the
        array arr from beg to end
1  $mid, invCount = 0$
2  if $end > beg$ then
3      mid = (beg+end)/2
       /* Call recursively and add
          returned value to invCount */
4      invCount += enhancedMergeSort(arr,left,mid)
5      invCount += enhancedMergeSort(arr,mid+1,right)
       /* Finally merge these divided
          arrays up using the merge
          function which also return
          inversions count for this step
          */
6      invCount += mergeTwoArrays(arr,left,mid+1,right)
7  return $invCount$
```

B. Space complexity

In the process of counting inversions, we are also sorting the input array as the algorithm demands this. If we wish to retain the original array we do need an auxiliary array which is the copy of the input array .
Thus, the space complexity will be O(n) .

Acknowledgment

References

[1] https://www.geeksforgeeks.org/

[2] https://www.stackoverflow.com/