Counting Inversions

Isaac DeJager

March 15, 2023

Click here to view the HackerRank page

1 Problem Description

In an array, arr, the elements at indices i and j (where $i < j$) form an inversion if $arr[i] > arr[j]$. In other words, inverted elements $arr[i]$ and $arr[j]$ are considered to be "out of order". To correct an inversion, we can swap adjacent elements.

Example
$arr = [2, 4, 1]$
To sort the array, we must perform the following two swaps to correct the inversions:

- swap 1: (4, 1) $\rightarrow$ [2, 1, 4]
- swap 2: (2, 1) $\rightarrow$ [1, 2, 4]

The sort has two inversions. Given an array arr, return the number of inversions to sort the array.

Function Description
Complete the function countInversions which has the following parameter:

- int $arr[n]$: an array of integers to sort

Returns

- int: the number of inversions

Input Format
The first line contains an integer, d, the number of datasets. Each of the next d pairs of lines is as follows:

- The first line contains an integer, n, the number of elements in arr

- The second line contains n space-separated integers, $arr[i]$

Constraints

- $i \leq d \leq 15$
- $1 \leq n \leq 10^5$
- $1 \leq arr[i] \leq 10^7$

Sample Input

```
2
5
1 1 1 2 2
5
2 1 3 1 2
```

Sample Output

```
0
4
```

Explanation

We sort the following $d = 2$ datasets:

- $arr = [1, 1, 1, 2, 2]$ is already sorted, so there are no inversions for us to correct.
- $arr = [2, 1, 3, 1, 2] \rightarrow [1, 2, 3, 1, 2] \rightarrow [1, 2, 1, 3, 2] \rightarrow [1, 1, 2, 3, 2] \rightarrow [1, 1, 2, 2, 3]$

We performed a total of 4 swaps to correct inversions.

2 Solution

```
long x;

vector<int> mergeArrays(vector<int> A, vector<int> B) {
    vector<int> arr;

    int i1 = 0;
    int i2 = 0;
    for (int i = 0; i < A.size() + B.size(); i++) {
        if (i2 >= B.size() || (i1 < A.size() && A[i1] <= B[i2])) {
            arr.push_back(A[i1]);
            i1++;
        } else {
            arr.push_back(B[i2]);
            i2++;
            x += A.size() - i1;
        }
    }

    return arr;
}

vector<int> mergeSort(vector<int> arr) {
    if (arr.size() == 1) {
        return arr;
    }

    int n = arr.size() / 2;
    vector<int> arr1 (arr.begin(), arr.begin() + n);
    vector<int> arr2 (arr.begin() + n, arr.end());

    arr1 = mergeSort(arr1);
    arr2 = mergeSort(arr2);

    return mergeArrays(arr1, arr2);
}

long countInversions(vector<int> arr) {
    x = 0;
    arr = mergeSort(arr);

    return x;
}
```

3 Analysis

The solution to this problem is nothing more than a small add-on to a merge sort. Consider the following array as an example:

$[1, 3, 4, 5, 2, 2, 4, 8]$

The logic of merge sort is as follows: split the array into two halves. Sort them seperately using merge sort, and then merge the halves together. This problem can be solved in the process of remerging the arrays. In the above example, we separate the array into two halves.

$[1, 3, 4, 5]$ $[2, 2, 4, 8]$

For the sake of convenience, the halves already happen to be sorted. Now, when merging the arrays, we can count the number of inversions required to do so.

We loop through the arrays and build a new array that contains all elements sorted. The array starts with:

$[1]$

No inversions were required for this, as 1 was taken from the left array. Next:

$[1, 2]$

Since 2 was taken from the right array, we can count how many numbers it needed to jump (a.k.a. how many inversions it takes to put 2 in the correct spot). We can see from the left array that it needs to hop $3, 4$ and 5. So it takes 3 inversions. Next:

$[1, 2, 2]$

Again, since this was taken from the right array it takes 3 inversions. And so, the process continues until the halves of the array are merged.

The above code is nearly all just a merge sort. The only addition is a global long variable x. Each time countInversions is called, x is reset to 0. And when the subarrays are merged, x is increased depending on how many numbers are remaining in the left array.

The time complexity for this is very simple. Since a merge sort takes $O(n * log(n))$ time, so does this. The space complexity is less ideal. Since a merge sort has to recursively split arrays in half until it is just a series of arrays with one element, the space complexity is $O(n)$ as that is number of arrays that are created.