

Unit 2- Counting inversions

Srinidhi H
Dept of CSE, MSRIT

Topics
1. Counting inversions <ol style="list-style-type: none">Problem.Designing algorithm based on divide and conquer approach.Analysis

Counting Inversions

- We are given a sequence of n numbers a_1, \dots, a_n ; we will assume that all the numbers are distinct.
- We want to define a measure that tells us how far this list is from being in ascending order; the value of the measure should be 0 if $a_1 < a_2 < \dots < a_n$, and should increase as the numbers become more scrambled.

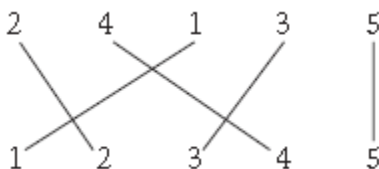
Counting the number of *inversions*

The two indices $i < j$ form an inversion if $a_i > a_j$, that is, if the two elements a_i and a_j are “out of order.” **Counting the number of *inversions*** is to determine the number of inversions in the sequence a_1, \dots, a_n .

Example:

Sequence is 2, 4, 1, 3, 5.

There are three inversions in this sequence: (2, 1), (4, 1), and (4, 3).



- if the sequence is in descending order, then every pair forms an inversion, and so there are
- $n(n-1)/2$ pairs.

Example:

5	4	3	2	1
---	---	---	---	---



{5,4}, {5,3}, {5,2}, {5,1}
{4,3}, {4,2}, {4,1}
{3,2}, {3,1}
{2,1}

Algorithm:

- Clearly, we could look at every pair of numbers (a_i, a_j) and determine whether they constitute an inversion; this would take $O(n^2)$ time.
- The basic idea is to follow the strategy of divide and conquer.
- We set $m = \lfloor n/2 \rfloor$ and divide the list into the two pieces a_1, \dots, a_m and a_{m+1}, \dots, a_n .
- We first count the number of inversions in each of these two halves separately.
- Then we count the number of inversions (a_i, a_j) , where the two numbers belong to different halves.
- Note that these first-half/second-half inversions have a particularly nice form: they are precisely the pairs (a_i, a_j) , where a_i is in the first half, a_j is in the second half.
- Suppose we have recursively sorted the first and second halves of the list and counted the inversions in each. We now have two sorted lists A and B , containing the first and second halves, respectively.
- We want to produce a single sorted list C from their union, while also counting the number of pairs (a, b) with $a \in A$, $b \in B$, and $a > b$.

Algorithm: Counting Inversions

//Purpose: To count the inversions for a given list $L(a_1, a_2, \dots, a_n)$

//Input: An unsorted list of distinct numbers $L(a_1, a_2, \dots, a_n)$

//Output: The number of inversions r for the list $L(a_1, a_2, \dots, a_n)$

Sort-and-Count(L)

If the list has one element
 then there are no inversions

Else

 Divide the list into two halves:

A contains the first $\lfloor n/2 \rfloor$ elements

B contains the remaining $\lfloor n/2 \rfloor$ elements

$(r_A, A) = \text{Sort-and-Count}(A)$

$(r_B, B) = \text{Sort-and-Count}(B)$

$(r, L) = \text{Merge-and-Count}(A, B)$

Endif

Return $r = r_A + r_B + r$, and the sorted list L

Algorithm: Merge two lists A and B from Sort-and-Count to count the number of inversions in the list $L(a_1, a_2, \dots, a_n)$

//Purpose: To Merge two lists A and B from Sort-and-Count to count the number of inversions in the list $L(a_1, a_2, \dots, a_n)$

//Input: An unsorted list of distinct numbers $A(a_1, a_2, \dots, a_{n/2})$ and $B(a_{n/2+1}, a_{n/2+2}, \dots, a_n)$

//Output: The number of inversions r for the sorted list $L(a_1, a_2, \dots, a_n)$

Merge-and-Count (A, B)

Maintain a *Current* pointer into each list, initialized to point to the front elements

Maintain a variable *Count* for the number of inversions, initialized to 0

While both lists are nonempty:

 Let a_i and b_j be the elements pointed to by the *Current* pointer

 Append the smaller of these two to the output list

 If b_j is the smaller element then

 Increment *Count* by the number of elements remaining in A

 Endif

 Advance the *Current* pointer in the list from which the smaller element was selected.

EndWhile

Once one list is empty, append the remainder of the other list to the output

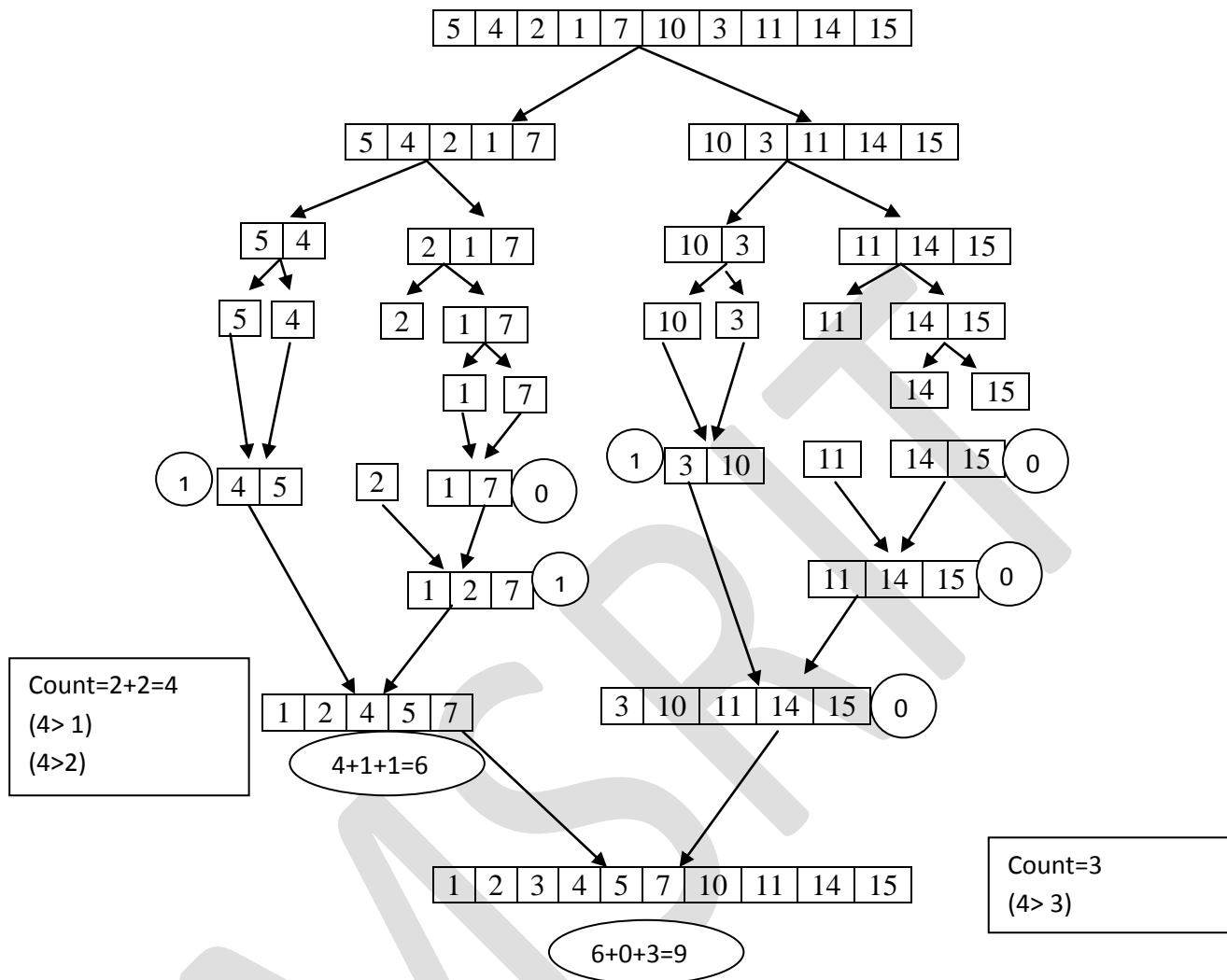
Return *Count* and the merged list

Analysis

- The running time of Merge-and-Count can be bounded by the analogue of the argument we used for the original merging algorithm at the heart of Mergesort: each iteration of the While loop takes constant time, and in each iteration we add some element to the output that will never be seen again.
- Thus the number of iterations can be at most the sum of the initial lengths of A and B, and so the total running time is $O(n)$.
- The Sort-and-Count algorithm correctly sorts the input list and counts the number of inversions; it runs in $O(n \log n)$ time for a list with n elements.

Examples using divide and conquer approach

- Count the number of inversions for the following array.



The total no of inversions=9