

Counting Inversions

Let A be an array of n distinct integers. We say (i, j) is an *inversion* in A if $i < j$ and $A[i] > A[j]$. Given an array A , the problem of *counting inversions* seeks the number of inversions in A , formally written as $|\{(i, j) \mid i < j, A[i] > A[j]\}|$. For example, the number of inversions in array $A[4, 3, 7, 2, 5]$ is 5.

Clearly, a sorted array (of size n) in ascending order contains 0 inversion. A sorted array (of size n) in descending order contains $n(n - 1)/2$ inversion. To see this, note that every pair (i, j) , $i < j$, forms an inversion because $A[i] > A[j]$; therefore, the number of inversions is: $\sum_{i=1}^n \sum_{j=i+1}^n 1 = \sum_{i=1}^n (n - i) = n(n - 1)/2$. Hence, the number of inversions of an array A serves as a good quantitative measure that quantifies how far A is from being (increasingly) sorted. Applications of this problem can be found in Section 5.3 [KT].

We now design algorithms to count inversions. A brute-force algorithm can be easily designed, that simply enumerates all pairs and counts the inversions:

```
Algorithm #1: brute-force-count-inv ( $A[1 \dots n]$ )
  let inv = 0;
  for  $i = 1$  to  $n$ 
    for  $j = i + 1$  to  $n$ 
      if( $A[i] > A[j]$ ): inv = inv + 1;
    end for;
  end for;
  return inv;
end algorithm;
```

Clearly, above brute-force-count-inv algorithm takes $\Theta(n^2)$ time. Can we design a faster algorithm? Let's try the divide-and-conquer idea. We define a recursive function DC-count-inv (A) that returns the number of inversions in A . Naturally, we can divide array A by splitting it in the middle, i.e., $A_1 = A[1 \dots n/2]$ and $A_2 = A[n/2 + 1 \dots n]$. The total number of inversions of A consists of inversions in A_1 , inversions in A_2 , and inversions between A_1 and A_2 . The first two parts can be calculated by recursively calling DC-count-inv(A_1) and DC-count-inv(A_2). To account for the third part, we introduce a new function count-inv-two-arrays ($S[1 \dots m]$, $T[1 \dots n]$), that calculates the inversions between S and T , formally defined as $|\{(i, j) \mid 1 \leq i \leq m, 1 \leq j \leq n, S[i] > T[j]\}|$. We first give the pseudo-code for function DC-count-inv:

```
Algorithm #2: DC-count-inv ( $A[1 \dots n]$ )
  if  $n \leq 1$ : return 0;
   $I_1 = \text{DC-count-inv } (A[1 \dots n/2]);$ 
   $I_2 = \text{DC-count-inv } (A[n/2 + 1 \dots n]);$ 
   $I_3 = \text{count-inv-two-arrays } (A[1 \dots n/2], A[n/2 + 1 \dots n]);$ 
  return  $I_1 + I_2 + I_3;$ 
end algorithm;
```

To complete above algorithm #2 we need to implement count-inv-two-arrays (S , T). Let's again use the “brute-force” idea, that compares all pairs between S and T :

```

function count-inv-two-arrays ( $S[1 \dots m], T[1 \dots n]$ )
    let inv = 0;
    for  $i = 1$  to  $m$ 
        for  $j = 1$  to  $n$ 
            if( $S[i] > T[j]$ ): inv = inv + 1;
        end for;
    end for;
    return inv;
end algorithm;

```

Function count-inv-two-arrays runs in $\Theta(mn)$ time. Hence, count-inv-two-arrays ($A[1 \dots n/2], A[n/2 + 1 \dots n]$) takes $\Theta(n/2 \cdot n/2) = \Theta(n^2)$ time. The running time of entire Algorithm #2, defined as $T(n)$, can be written as $T(n) = 2T(n) + \Theta(n^2)$. By using master's theorem, where $a = 2, b = 2, d = 2$, we conclude that $T(n) = \Theta(n^2)$.

Algorithm #2 is no better/faster than algorithm #1. They are essentially the same algorithm. Although algorithm #2 uses the divide-and-conquer framework, it in fact also enumerates all pairs. To see this, note that all pairs between $A[1 \dots n/2]$ and $A[n/2 + 1 \dots n]$ are compared by count-inv-two-arrays; all pairs within $A[1 \dots n/2]$ and all pairs within $A[n/2 + 1 \dots n]$ are also compared, as they recursively call count-inv-two-arrays which calls count-inv-two-arrays (on subarrays of $n/4$) as well. So, algorithm #2 enumerates all pairs. Hence, not a surprise, it's running time is $\Theta(n^2)$.

Can we really design faster algorithm for this problem? Let's borrow some ideas from merge-sort. Recall that, in merge-sort, the combining/merging step (i.e., function merge-two-sorted-arrays) takes $\Theta(n)$ time. And the reason is that, the two subarrays are already sorted, which is taken advantage of by the merging step. We therefore hope that, the inversions between two *sorted* arrays, can be counted in a faster way. We will demonstrate that, this is indeed true.

Here is algorithm #3. It counts the inversions *while* sorting the array. We define a recursive function DC-sort-count-inv (A) that returns the sorted array of A and the number of inversions in A . The framework of algorithm #3 is the same with that of algorithm #2. It's pseudo-code is given below, where the merging step, merge-count-inv-two-sorted-arrays, takes two *sorted* arrays as input, and return the merged/sorted array and the number of inversions between the two input arrays.

```

Algorithm #3: DC-sort-count-inv ( $A[1 \dots n]$ )
    if  $n \leq 1$ : return ( $A, 0$ );
     $(A'_1, I_1) = \text{DC-sort-count-inv } (A[1 \dots n/2]);$ 
     $(A'_2, I_2) = \text{DC-sort-count-inv } (A[n/2 + 1 \dots n]);$ 
     $(A', I_3) = \text{merge-count-inv-two-sorted-arrays } (A'_1, A'_2);$ 
    return ( $A', I_1 + I_2 + I_3$ );
end algorithm;

```

The remaining (key) step is to implement function merge-count-inv-two-sorted-arrays ($S[1 \dots m], T[1 \dots n]$). Again, we aim for doing it faster (than brute-force which takes $\Theta(mn)$ time) by making use of the fact that both S and T are already sorted. It's pseudo-code is given below; it just adds a single line to the merge-two-sorted-arrays function.

```

function merge-count-inv-two-sorted-arrays ( $S[1 \dots m], T[1 \dots n]$ )
    init an empty array  $C$ ;
    init pointers  $i = 1$  and  $j = 1$ ;
    init inv = 0;
     $S[m+1] = M$  and  $T[n+1] = M$ ;
    for  $k = 1 \rightarrow m+n$ 
        if  $S[i] \leq T[j]$ 
             $C[k] = S[i]$ ;
             $i = i + 1$ ;
        else
             $C[k] = T[j]$ ;
             $j = j + 1$ ;
            inv = inv + (m - i + 1);
        end if;
    end for;
    return (C, inv);
end algorithm;

```

Let's explain/prove why above algorithm works. We have proved that C is the merged, sorted array of S and T (in Lecture 01) so now we focus on showing that inv stores the correct number of inversions between S and T , i.e., $|\{(i, j) \mid S[i] > T[j], 1 \leq i \leq m, 1 \leq j \leq n\}|$. The above algorithm maintains two pointers i and j that (always) point to the first number (also the smallest number) in S and T respectively that haven't been added to C . In case of $S[i] \leq T[j]$, which means (i, j) does not form an inversion, we do not increase inv . In case of $S[i] > T[j]$, which means (i, j) forms an inversion, we increase inv by $m - i + 1$. Why? This is because S is sorted, i.e., $S[i] < S[i+1] < \dots < S[m]$ and therefore all numbers after $S[i]$ also form inversions with $T[j]$. In other words, $(i, j), (i+1, j), (i+2, j), \dots, (m, j)$ — all these are inversions. The number of these inversions are $(m - i + 1)$; hence we increase inv by this amount. Note that also at this time point, $T[j]$ is copied/moved to C and j advances. $T[j]$ won't have a chance to be explicitly compared with any of the $S[i+1], \dots, S[m]$. These comparisons are saved, as they are guaranteed to be forming inversions with $T[j]$, and therefore counted right here (when $S[i] > T[j]$ happens).

More precisely, when $S[i] > T[j]$ happens, $S[i]$ must be the smallest number in S that is larger than $T[j]$. In other words, it is not possible to have i' such that $i' < i$ and $S[i'] > T[j]$. Let's prove this by contradiction. Assume that such i' exists. When $S[i] > T[j]$ happens, $S[i']$ must have been added to C simply because $i' < i$. Also right after $S[i] > T[j]$ happens, $T[j]$ will be added to C . This implies that $S[i']$ is added to C before $T[j]$. This is a contradiction because $S[i'] > T[j]$ but we proved (in Lecture 01) that C will be sorted (in ascending order). The claim that, when $S[i] > T[j]$ happens $S[i]$ is the smallest number in S that is larger than $T[j]$, implies that there are *exactly* $m - i + 1$ inversions involving $T[j]$; all numbers before $S[i]$ do not form inversions with $T[j]$ because they are smaller than $T[j]$.

Hence, above algorithm does this: when $T[1]$ is added to C , it counts exactly all inversions that involves $T[1]$, i.e., inversions in the form of $(\cdot, 1)$; when $T[2]$ is added to C , it counts exactly all inversions that involves $T[2]$, i.e., inversions in the form of $(\cdot, 2)$; and so on. Since each number in T will be added once and exactly once to C , eventually all inversions are precisely counted.

Here is a working example: $S = [2, 4, 5, 7], T = [1, 3, 6, 9]$. Let's run above algorithm to verify these claims.

$k = 1, i = 1, j = 1, S[1] > T[1]$, so $C = [1]$ and $\text{inv} = 4$; **4 is the #inversions $(\cdot, j = 1)$**
 $k = 2, i = 1, j = 2, S[1] < T[2]$, so $C = [1, 2]$;
 $k = 3, i = 2, j = 2, S[2] > T[2]$, so $C = [1, 2, 3]$ and $\text{inv} = 4 + 3$; **3 is the #inversions $(\cdot, j = 2)$**
 $k = 4, i = 2, j = 3, S[2] < T[3]$, so $C = [1, 2, 3, 4]$;
 $k = 5, i = 3, j = 3, S[3] < T[3]$, so $C = [1, 2, 3, 4, 5]$;
 $k = 6, i = 4, j = 3, S[4] > T[3]$, so $C = [1, 2, 3, 4, 5, 6]$ and $\text{inv} = 4 + 3 + 1$; **1 is the #inversions $(\cdot, j = 3)$**
 $k = 7, i = 4, j = 4, S[4] < T[4]$, so $C = [1, 2, 3, 4, 5, 6, 7]$;
 $k = 8, i = 5, j = 4, S[5] > T[4]$, so $C = [1, 2, 3, 4, 5, 6, 7, 9]$ and $\text{inv} = 4 + 3 + 1 + 0$. **0 is the #inversions $(\cdot, j = 4)$**

The running time of merge-count-inv-two-sorted-arrays is clearly $\Theta(m + n)$, the same to the merge-two-sorted-arrays function. Why is it faster than count-inv-two-arrays (both functions count the number of inversions between two arrays)? The merge-count-inv-two-sorted-arrays function successfully avoid the all-vs-all comparisons, by taking advantage of the fact that S and T are sorted. (Again, key: when $S[i] > T[j]$ happens, $S[i+1], S[i+2], \dots, S[m]$ are all guaranteed to be forming inversions with $T[j]$ so these comparisons are avoided.)

Let's analyze the running time of DC-sort-count-inv. Since merge-count-inv-two-sorted-arrays (A'_1, A'_2) takes $\Theta(|A'_1| + |A'_2|) = \Theta(n)$ time, the total running time of DC-sort-count-inv, denoted as $T(n)$, can be written as $T(n) = 2T(n/2) + \Theta(n)$. This gives $T(n) = \Theta(n \log n)$.