Divide and Conquer: Counting Inversions[1]

1 Counting Inversions

We now look at a closely related problem to merge-sort. Given an array $A[1:n]$, the pair (i,j) for $1 \leq i < j \leq n$ is called an *inversion* if $A[i] > A[j]$. For example, in the array $[10, 20, 30, 50, 40]$, the pair $(4,5)$ is an inversion.

COUNTING INVERSION
Input: An array $A[1:n]$
Output: The number of inversions in A.
Size: n, the size of the array.

There is a naive $O(n^2)$ time algorithm: go over all pairs and check if they form an inversion or not. We now apply the divide-and-conquer paradigm to do better.

If $n = 1$, then the number of inversions is 0. Otherwise, suppose we divide the array into two: $A[1:n/2]$ and $A[n/2+1:n]$. Recursively, suppose we have computed the number of inversions in $A[1:n/2]$ and $A[n/2+1:n]$. Let these be I_1 and I_2, respectively. Note that any inversion (i,j) in $A[1:n]$ satisfies

(a) either $i < j \leq n/2$, which implies (i,j) is an inversion in $A[1:n/2]$, or

(b) $n/2+1 \leq i < j$, which implies (i,j) is an inversion in $A[n/2+1:n]$, or

(c) $i \leq n/2 < j$, and these are the extra inversions over $I_1 + I_2$ that we need to count.

Let's call any (i,j) of type (c) above a *cross* inversion, and let C denote this number. Then by what we said above, we need to return $I_1 + I_2 + C$. To obtain a "win", we need to see if we can calculate C "much faster" than $O(n^2)$ time. How do we do that?

After you think about it for a while, there may not seem to be any easy way to calculate C faster than $O(n^2)$. Indeed, there could be $\Theta(n^2)$ inversions in $A[1:n]$ and so shouldn't it take that muct time to count them? How do we get around this? There are two crucial observations that help here.

- The number of cross-inversions between $A[1:n/2]$ and $A[n/2+1:n]$ is the same as between $\mathsf{sort}(A[1:n/2])$ and $\mathsf{sort}(A[n/2+1:n])$, where $\mathsf{sort}(P)$ is the sorted order of an array P.
- If $A[1:n/2]$ and $A[n/2+1:n]$ were sorted, then the cross-inversions can be calculated in $O(n)$ time. This may not be immediate, but if you understand the COMBINE subroutine above, then it should ring a bell. We elaborate it on it below.

Cross-Inversions between Sorted Arrays. Given two sorted arrays $P[1:p]$ and $Q[1:q]$, we can count the number of cross-inversion pairs (i,j) such that $P[i] > Q[j]$ in $O(n)$ time using the two-pointer trick. As in COMBINE we start off with two pointers i, j initialized to 1. We also store a counter num initialized to 0 which, at the end, is supposed to contain the answer C. We check if $P[i] > Q[j]$ or not. If it isn't,

[1] *Lecture notes by Deeparnab Chakrabarty. Last modified : 19th Mar, 2022*
These have not gone through scrutiny and may contain errors. If you find any, or have any other comments, please email me at deeparnab@dartmouth.edu. Highly appreciated!

that is if $P[i] \leq Q[j]$, then (i,j) is not a cross-inversion and we simply increment $i = i+1$. Otherwise, if $P[i] > Q[j]$, then we increment $\mathsf{num} = \mathsf{num} + (p-i+1)$ and $j = j+1$. Why do you increment by so much? Didn't you find (i,j) is an inversion and so you should increment by only $+1$? Well, not only is (i,j) a cross-inversion, so are $(i+1,j)$, $(i+2,j)$, and so on till (p,j). This is crucially using the fact that P is sorted. By doing a single comparison, because of sortedness, we discover a "bunch" of inversions. This gives us the "win" we were looking for. We provide loop invariants which help us formalize the above.

```
1: procedure CountCrossInv($P[1:p], Q[1:q]$):
2:     ▷ $P$ and $Q$ are sorted; outputs the number of $(i,j)$ with $P[i] > Q[j]$.
3:     $i \leftarrow 1$; $j \leftarrow 1$; $\mathsf{num} \leftarrow 0$.
4:     ▷ Inv 1: $\mathsf{num} = |\{(a,b) \; : \; a < b, P[a] > Q[b], \;\; 1 \leq a \leq p, 1 \leq b \leq j-1\}|$.
5:     ▷ In plain English, num counts all inversions of the form $(a,b)$ with $b < j$
6:     ▷ Inv 2: $P[i-1] \leq Q[j]$
7:     while $i < p+1$ and $j < q+1$ do:
8:         if $(P[i] > Q[j])$ then:
9:             $\mathsf{num} \leftarrow \mathsf{num} + (p-i+1)$ ▷ Increment the number of inversions $Q[j]$ participates in
10:            $j \leftarrow j+1$ ▷ Inv 2 holds due to sortedness of $Q$. Inv 1 holds due to sortedness of $P$
11:        else:
12:            $i \leftarrow i+1$ ▷ Inv 2 holds by branch of if-statement. Inv 1 holds since $j$ hasn't changed.
13:    return num.
```

Theorem 1. CountCrossInv counts the number of cross inversions between P and Q in time $O(p+q)$.

Proof. The running time is hopefully clear since each while loop is $O(1)$ time and there are $p+q-2$ such loops at most.

The correctness follows from the invariants. If the invariants were true at the end, then either $j = q+1$ at the end in which case Inv 1 implies num is precisely the number of inversions. Or, $i = p+1$ in which case Inv 2 implies $Q[j] \geq P[p]$ and thus $Q[b]$ for $b \geq j$ doesn't participate in any inversions. And so Inv 1 implies num is the precise answer.

The invariants hold vacuously at the beginning (when $i = j = 1$). Let's say they hold for all while loops up to ℓ (for some number ℓ) and let's consider the ℓth while loop. Two cases arise.

Case 1: $P[i] > Q[j]$. Since P is sorted increasing, (a,j) are cross-inversions for $i \leq a \leq p$, and so $Q[j]$ participates in *at least* $(p-i+1)$ many inversions. Inv 2 implies $Q[j] \geq P[i-1]$ and again since P is sorted, there are *no* inversions (a,j) for $1 \leq a \leq i-1$. So, $(p-i+1)$ is precisely the number of inversions $Q[j]$ participates in. The next line increments j and this makes Inv 1 hold. Incrementing j keeps Inv 2 true since Q is sorted increasing.

Case 2: $P[i] \leq Q[j]$. In this case we increment i and so Inv 1 remains true since j wasn't incremented. Inv 2 holds because of the case we are in. This completes the proof. □

Now we are armed to describe the divide-and-conquer algorithm for counting inversions.

```
1:  procedure CountInv1($A[1:n]$):
2:      ▷ Counts the number of inversions in $A[1:n]$
3:      if $n = 1$ then:
4:          return $0$. ▷ Singleton Array
5:      $m \leftarrow \lfloor n/2 \rfloor$
6:      $I_1 \leftarrow$ CountInv1($A[1:m]$)
7:      $I_2 \leftarrow$ CountInv1($A[m+1:n]$)
8:      $B_1 \leftarrow$ MergeSort($A[1:m]$)
9:      $B_2 \leftarrow$ MergeSort($A[m+1:n]$)
10:     $C \leftarrow$ CountCrossInv($B_1, B_2$)
11:     return $I_1 + I_2 + C$.
```

Let's analyze the time complexity. As always, let $T(n)$ be the worst case running time of CountInv1 on an array of length n. Let $A[1:n]$ be the array attaining this time, and let's see the run of the algorithm on this array. The time taken by Line 6 and Line 7 are $T(\lfloor n/2 \rfloor)$ and $T(\lceil n/2 \rceil)$ respecively. The time taken by Line 10 takes $O(n)$ time by what we described above. Furthermore, the Line 8 and Line 9 takes $O(n \log n)$ time. Together, we get the following recurrence

$$T(n) \leq T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + O(n \log n)$$

The above 'almost' looks like the merge-sort recurrence, and indeed the recurrence solves[2] to $T(n) = O(n \log^2 n)$.

But there is something wasteful about the above algorithm. In particular, if you run it on a small example by hand you will see that you are sorting a lot. And often the same sub-arrays. Whenever you see this, often you can exploit this observation and get a faster algorithm?

Let us use this opportunity to introduce a new idea in the divide-and-conquer paradigm: *get more by asking for more*. This "asking for more" technique is something you may have seen while proving statements by induction where you can prove something you want by actually asking to prove something stronger by induction. In this problem, we ask our algorithm to do more: given an array $A[1:n]$ it has to count the inversions ***and*** also has to sort the array too. Now note that in this case Line 8 and Line 9 are not needed any more; this is returned by the new stronger algorithm. We however need to also return the sorted array : but this is what Combine precisely does[3]. So the final algorithm for counting inversions is below.

[2]Warning: Master Theorem doesn't apply. But if you go back to the kitty method proof, you should be able to recreate the $T(n) = O(n \log^2 n)$. Indeed, every "round" one puts in $\leq Cn \log n$ instead of $\leq Cn$, and possibly even smaller.

[3]Actually, the CountCrossInv code looks so much like Combine, you should not really have two lines (Line 8 and Line 9 in Sort-and-Count), but wrap both of these into a single subroutine. I have separate lines for conceptual clarity at the expense of running time inefficency (but not in a way that the big-Oh picture is muddled). In your coding assignment, hopefully you will keep this in mind.

```
procedure SORT-AND-COUNT($A[1:n]$):
    ▷ Returns $(B, I)$ where $B = \mathsf{sort}(A)$ and $I$ is the number of inversions in $A[1:n]$
    if $n = 1$ then:
        return $(A, 0)$. ▷ Singleton Array
    $m \leftarrow \lfloor n/2 \rfloor$
    $(B_1, I_1) \leftarrow$ SORT-AND-COUNT($A[1:m]$)
    $(B_2, I_2) \leftarrow$ SORT-AND-COUNT($A[m+1:n]$)
    $C \leftarrow$ COUNTCROSSINV($B_1, B_2$)
    $B \leftarrow$ COMBINE($B_1, B_2$)
    return $(B, I_1 + I_2 + C)$
```

Now we see that the recurrence for the running time of SORT-AND-COUNT is precisely

$$T(n) \leq T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + O(n)$$

Theorem 2. SORT-AND-COUNT returns the number of inversions of an array $A[1:n]$ in $O(n \log n)$ time.

As an application, you are now ready to solve Problem 1 of PSet 0. Go ahead and try it again!