7 Count Inversions [KT 5.3]

Definition and motivation. *Collaborative filtering*: try to match your preferences with those of other people out on the Internet.

Meta search: synthesize the results from multiple search engine. Need to measure the similarity of two rankings.

The following example gives the definition of the problem.

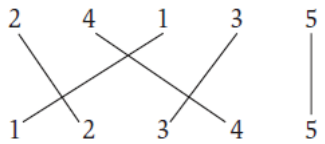

Figure 5.4 Counting the number of inversions in the sequence 2, 4, 1, 3, 5. Each crossing pair of line segments corresponds to one pair that is in the opposite order in the input list and the ascending list—in other words, an inversion.

Re-label items of Alice's list as $\{1, \ldots, n\}$, and Bob's list becomes $\{a_1, \ldots, a_n\}$. The #inversions is defined as #pairs (i, j) in Bob's list such that $a_i > a_j$.

Solution: We restate the problem as follows. In the input we are given an array $A[]$ of n (distinct) numbers. The goal is to count the number of inversion pairs, where a pair $A[i]$ and $A[j]$ is called an inversion if $i < j$ and $A[i] > A[j]$. While the naive method of enumerating all pairs of i and j takes $O(n^2)$ time, we would like to design a more efficient algorithm via divide and conquer.

Let us first divide the problem into two subproblems using the most straightforward idea. We take $k = n/2$ and divide A into an array $X[] = A[1, k]$ of k numbers and an array $Y[] = A[k+1, n]$ of $(n - k)$ numbers. We first (recursively) solve the two subproblems, i.e. figure out c_1: the number of inversions in $X[]$ and c_2: the number of inversions in $Y[]$. However, the total number of inversions in $A[]$ consists of c_1, c_2, and another part, namely c_3: the number of pairs $X[i]$ and $Y[j]$ such that $X[i] > Y[j]$. If we think a little more carefully, c_3 is the sum of the number of $X[i]$'s those are greater than $Y[j]$ for each $Y[j]$. (Similarly, c_3 is also the sum of the number of $Y[j]$'s those are less than $X[i]$ for each $X[i]$. However we cannot add both sums together; otherwise, we would double count each pair.)

Now let us design an algorithm to figure out how many $X[i]$'s are greater than $Y[j]$ for each $Y[j]$. We find that the task is simpler if we assume both $X[]$ and $Y[]$ are sorted.

Merge-and-Count($X[], Y[]$)

$i = j = k = 1; c_3 = 0$

WHILE ($i \leq X.len$ or $j \leq Y.len$) DO

IF (($j > Y.len$) or ($i \leq X.len$ and $X[i] < Y[j]$))

THEN $Z[k++] = X[i++]$

ELSE $c_3 = c_3 + (n_1 - i + 1)$; $Z[k++] = Y[j++]$

RETURN $(Z[], c_3)$

We see that the only difference between Merge-and-Count and the Merge process in merge sort is that Merge-and-Count uses $c_3 = c_3 + (n_1 - i + 1)$ to count the number of inversion pairs between $X[]$ and $Y[]$. Indeed, $(n_1 - i + 1)$ is the number of $X[]$'s those are greater than $Y[j]$ at that particular time.

Now we can easily design the Sort-and-Count process as follows.

Sort-and-Count($n, A[]$)

IF ($n <= 1$) return $(A, 0)$;

$k = n/2$;

$(X[], c_1) =$ Sort-and-Count($k, A[1, k]$);

$(Y[], c_2) =$ Sort-and-Count($n - k, A[k + 1, n]$); // two divide steps

$(Z[], c_3) =$ Merge-and-Count($X[], Y[]$); // conquer step

RETURN $(Z, c_1 + c_2 + c_3)$;

We see that the procedure is almost the same as the merge sort algorithm, except for that we also keep track of the number of inversion pairs along the way. Since we also sorted the two arrays in the divide steps, Merge-and-Count will take two sorted arrays as input, meeting our original assumption. Of course, Merge-and-Count will also return a sorted array. The time complexity of Sort-and-Count is the same as that of merge sort, which is $O(n \log n)$.

The Master Theorem. Suppose to solve a problem of size n, we divide it into b subproblems where each subproblem is of size n/a (where $a > 1, b \geq 1$ are constants), and use time $f(n)$ in the conquer step. We have the following recurrence for the time complexity $T(n)$ for a solving a problem of size n:

$$T(n) = bT\left(\frac{n}{a}\right) + f(n) \text{ for } n > 1,$$

and $T(n) = \Theta(1)$ when $n \leq 1$.

- When $f(n) = \Theta(n^q)$ where the constant $q > \log_a b$, we have $T(n) = \Theta(n^q)$.
- When $f(n) = \Theta(n^q \log^k n)$ where the constant $q = \log_a b, k \geq 0$, we have $T(n) = \Theta(n^q \log^{k+1} n)$.
- When $f(n) = \Theta(n^q)$ where the constant $q < \log_a b$, we have $T(n) = \Theta(n^{\log_a b})$.