```
MERGE-INVERSIONS(A, p, q, r)
n_1 = q − p + 1
n_2 = r − q
let L[1 . . n_1 + 1] and R[1 . . n_2 + 1] be new arrays
for i = 1 to n_1
    L[i] = A[p + i − 1]
for j = 1 to n_2
    R[j] = A[q + j]
L[n_1 + 1] = ∞
R[n_2 + 1] = ∞
i = 1
j = 1
inversions = 0
counted = FALSE
for k = p to r
    if counted == FALSE and R[j] < L[i]
        inversions = inversions + n_1 − i + 1
        counted = TRUE
    if L[i] ≤ R[j]
        A[k] = L[i]
        i = i + 1
    else A[k] = R[j]
        j = j + 1
        counted = FALSE
return inversions
```

The initial call is COUNT-INVERSIONS$(A, 1, n)$.

In MERGE-INVERSIONS, the boolean variable *counted* indicates whether we have counted the merge-inversions involving $R[j]$. We count them the first time that both $R[j]$ is exposed and a value greater than $R[j]$ becomes exposed in the L array. We set *counted* to FALSE upon each time that a new value becomes exposed in R. We don't have to worry about merge-inversions involving the sentinel ∞ in R, since no value in L will be greater than ∞.

Since we have added only a constant amount of additional work to each procedure call and to each iteration of the last **for** loop of the merging procedure, the total running time of the above pseudocode is the same as for merge sort: $\Theta(n \lg n)$.