Introduction and Problem Description

We will continue with our study of divide and conquer algorithms by considering how to count the number of inversions in a list of elements. This is a problem that comes up naturally in many domains, such as collaborative filtering and meta-search tools on the internet. A core issue in applications like these is the problem of comparing two rankings: you rank a set of $n$ movies, and then a collaborative filtering system consults its database to look for other people who had “similar” rankings in order to make suggestions. But how does one measure similarity?

Suppose you and another person rank a set of $n$ movies, labelling them from 1 to $n$ accordingly. A natural way to compare your ranking with the other persons is to count the number of pairs that are “out of order.” Formally, we will consider the following problem: We are given a sequence of $n$ numbers, $a_1, ..., a_n$, which we assume 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 $a_1 < a_2 < ... < a_n$ and should increase as the numbers become more scrambled, taking on the largest possible value if $a_n < a_{n-1} < ... < a_1$.

A natural way to quantify this notion is by counting the number of inversions.

**Definition** Given a sequence of numbers $a_1, a_2, ..., a_n$, we say indices $i < j$ form an inversion if $a_i > a_j$. That is, if $a_i$ and $a_j$ are out of order.

As an example, consider the sequence

$$2, 4, 1, 3, 5$$

There are three inversions in this sequence: $(2, 1), (4, 1)$, and $(4, 3)$. A simple way to count the number of inversions when $n$ is small is to draw the sequence of input numbers in the order they’re provided, and below that in ascending order. We then draw a line between each number in the top list and its copy in the lower list. Each crossing pair of line segments corresponds to an inversion.

![Figure 1](image.png)

Figure 1: There are three inversions: $(2, 1), (4, 1)$, and $(4, 3)$.

Note how the number of inversions is a measure that smoothly interpolates between complete agreement (when the sequence is in ascending order, then there are no inversions) and complete disagreement (if the sequence is in descending order, then every pair forms an inversion and so there are $\binom{n}{2}$ of them).
Designing an Algorithm

A naive way to count the number of inversions would be to simply look at every pair of numbers \((a_i, a_j)\) and determine whether they constitute an inversion. This would take \(O(n^2)\) time.

As you can imagine, there is a faster way that runs in \(O(n \log n)\) time. Note that since there can be a quadratic number of inversions, such an algorithm must be able to compute the total number without every looking at each inversion individually. The basic idea is to use a divide and conquer strategy.

Similar to the other divide and conquer algorithms we’ve seen, the first step is to divide the list into two pieces: set \(m = \lceil \frac{n}{2} \rceil\) and consider the two lists \(a_1, \ldots, a_m\) and \(a_{m+1}, \ldots, a_n\). Then we conquer each half by recursively counting the number of inversions in each half separately.

Now, how do we combine the results? We have the number of inversions in each half separately, so we must find a way to count the number of inversions of the form \((a_i, a_j)\) where \(a_i\) and \(a_j\) are in different halves. We know that the recurrence \(T(n) = 2T(n/2) + O(n)\), \(T(1) = 1\) has solution \(T(n) = O(n \log n)\), and so this implies that we must be able to do this part in \(O(n)\) time if we expect to find an \(O(n \log n)\) solution.

Note that the first-half/second-half inversions have a 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, and \(a_i > a_j\).

To help with counting the number of inversions between the two halves, we will make the algorithm recursively sort the numbers in the two halves as well. Having the recursive step do a bit more work (sorting as well as counting inversions) will make the “combining” portion of the algorithm easier.

Thus the crucial routine in this process is . Suppose we have recursively sorted the first and second halves of the list and counted the inversions in each. We know 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 \(a \in A\) and \(b \in B\) and \(a > b\).

This is closely related to a problem we have previously encountered: namely the “combining” step in MERGE SORT. There, we had two sorted lists \(A\) and \(B\) and we wanted to merge them into a single sorted list in \(O(n)\) time. The difference here is that we want to do something extra: not only should we produce a single sorted list from \(A\) and \(B\), but we should also count the number of inverted pairs \((a, b)\) where \(a \in A\), \(b \in B\) and \(a > b\) as we do so.

We can do this by walking through the sorted lists \(A\) and \(B\), removing elements from the front and appending them to the sorted list \(C\). In a given step, we have a current pointer into each list, showing our current position. Suppose that these pointers are currently at elements \(a_i\) and \(b_j\). In one step, we compare the elements \(a_i\) and \(b_j\), remove the smaller one from its list, and append it to the end of list \(C\).

This takes care of the merging. To count the number of inversions, note that because \(A\) and \(B\) are sorted, it is actually very easy to keep track of the number of inversions we encounter. Every time element \(a_i\) is appended to \(C\), no new inversions are encountered since \(a_i\) is smaller than everything left in list \(B\) and it comes before all of them. On the other hand, if \(b_j\) is appended to list \(C\), then it is smaller than all of the remaining items in \(A\) and it comes after all of them, so we increase our count of the number of inversions
by the number of elements remaining in $A$. This is the crucial idea: in constant time, we have accounted for a potentially large number of inversions.

---

**Merge-and-Count**

*Input:* Two sorted lists $A$ and $B$

*Output:* A sorted list containing all elements in $A$ and $B$, as well as the number of inversions assuming all elements in $A$ precede those in $B$.

```
merge-and-count(A,B)
L = []
currA = 1
currB = 1
count = 0
While currA <= A.length and currB <= B.length:
a = A[currA]
b = B[currB]
if a < b then
    L.append(a)
currA = currA + 1
else
    L.append(b)
    remaining = A.length - currA + 1
    count = count + remaining
    currB = currB + 1
Once one list is empty, append the remainder of the other list to L
Return count and L
```

We use this MERGE-AND-COUNT routine in a recursive procedure that simultaneously sorts and counts the number of inversions in a list $L$.

---

**Sort-and-Count**

*Input:* A list $L$ of $n$ distinct numbers

*Output:* A sorted version of $L$ as well as the number of inversions in $L$.

```
sort-and-count(L)
if n = 1 then
    return 0 and L
m = [n/2]
A = L[1,...,m]
B = L[m+1,...,n]
(r_A,A) = sort-and-count(A)
```
\[(r_B, B) = \text{Sort-and-Count}(B)\]
\[(r, L) = \text{Merge-and-count}(A, B)\]

return \(r_A + r_B + r\) and the sorted list \(L\)

Runtime

The running time of \texttt{Merge-and-Count} can be bounded by the analogue of the argument we used for the original \texttt{Merge} algorithm: each iteration of the \texttt{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 \(O(n)\).

Since \texttt{Merge-and-Count} runs in \(O(n)\) time, the recurrence for the runtime of \texttt{Sort-and-Count} is

\[
T(n) = \begin{cases} 
2T(n/2) + O(n) & n > 1 \\
1 & \text{otherwise}
\end{cases}
\]

This is exactly the same recurrence as we saw in the analysis of \texttt{MergeSort}, so we can immediately say the runtime of \texttt{Sort-and-Count} is

\[
T(n) = O(n \log n)
\]