Chapter 6

Runtime bounds on comparison sort

Both merge sort and quicksort only use the $<$ operator between elements to compare them. In this manner, they are both “comparison sorting” algorithms. Non-comparison sorting algorithms exploit additional properties of the numbering elements. E.g., we are able to “peel” the most-significant digit off of an integer to perform postman sort (or the least-significant digit to perform radix sort). But if the only thing we know about our elements is the existence of a $<$ operator, comparison sorts are all that we can do.

Here we consider the worst-case runtime of any comparison sort algorithm, even those that have never been proposed. We would like to know if it is ever possible to do better than $O(n \log(n))$ using a comparison sorting on any array of $n$ elements.

An array of $n$ values can be arranged in $n!$ unique permutations. Now consider a comparison sorting algorithm, which will perform as well as possible against a malicious opponent. That is, this unknown comparison sorting algorithm that we dream of will perform as well as possible against a worst-case input.

If we think abstractly, sorting is the process of mapping any of those $n!$ array permutations into the unique sorted array permutation. In an optimal

\[1\] For simplicity, let’s assume that there are no duplicate elements.
world, each comparison will contribute 1 additional bit of information\footnote{This is not true in general if we choose sub-optimal comparisons to perform. For example, if we know that \( a < b \) from one comparison and know that \( b < c \) from a second comparison, then the comparison \( a < c \) must be true, and therefore contributes no additional information.}. In this manner, optimal comparisons will divide the space of \( n! \) unsorted arrays in half.

Using this strategy, we can construct a decision tree, which iteratively divides \( n! \) in half during each comparison, until we reach the unique sorted ordering. The longest path from the root to any leaf will be the number of comparisons that we need to perform against a malicious opponent. Note that an optimal comparison strategy guarantees that our decision tree will be balanced, which guarantees that any initial ordering chosen by that opponent will not be able to make any path of comparisons taking us from the root to the leaf substantially more expensive than any other. The depth of a balanced binary decision tree with \( n! \) leaves will be \( \log_2(n!) \). We can expand \( \log(n!) \) into the following:

\[
\log(n!) = \log(n \cdot (n-1) \cdot (n-2) \cdots 3 \cdot 2 \cdot 1) = \log(n) + \log(n-1) + \log(n-1) + \log(3) + \log(2) + \log(1).
\]

### 6.1 Lower bound on the number of steps required by an optimal comparison sorting algorithm

Under the assumption of an optimal comparison strategy (even if we are not sure precisely how that would work), the number of comparisons needed to multiplex the \( n! \) possible array permutations to the unique sorted ordering will be \( \log_2(n!) \), with which we can derive a lower bound on the number of
comparisons that would need to be performed:

\[
\log(n!) = \log(n) + \log(n-1) + \log(n-2) + \cdots + \log(3) + \log(2) + \log(1)
\]

\[
> \log(n) + \log(n-1) + \log(n-2) + \cdots + \log\left(\frac{n}{2}\right) + 0 + \cdots + 0 + 0 + 0
\]

\[
= \frac{n}{2} \log\left(\frac{n}{2}\right)
\]

\[
= \frac{n}{2} (\log(n) - \log(2))
\]

\[
\in \Omega(n \log(n)).
\]

Amazingly, this demonstrates that no comparison sort, even one that has never been dreamed up, can have a runtime substantially faster than \(n \log(n)\).

### 6.2 Upper bound on the number of steps required by an optimal comparison sorting algorithm

Using a strategy similar to the one above, we can also derive an upper bound on the number of steps necessary by some hypothetical optimal comparison strategy. This is arguably less satisfying than the pessimistic \(\Omega(\cdot)\) bound, because it may be easy to be optimistic under the assumption of some optimal comparison strategy that has not yet been shown. Nonetheless, our upper bound is as follows:

\[
\log(n!) = \log(n) + \log(n-1) + \log(n-2) + \cdots + \log(3) + \log(2) + \log(1)
\]

\[
< \log(n) + \log(n) + \log(n) + \cdots + \log(n) + \log(n) + \log(n)
\]

\[
= n \log(n)
\]

\[
\in O(n \log(n)).
\]

This verifies what we have seen already: our merge sort and quicksort implementations were comparison sorts, and they were in \(O(n \log(n))\). Because we have already seen such algorithms by construction, we no longer need
to worry about our previous constraint that the comparisons be performed optimally. We know that $O(n \log(n))$ is an upper bound on number of steps needed by well-constructed comparison sorts, even in the worst-case (i.e., even against a malicious opponent).

Thus we see that $\log(n!) \in \Theta(n \log(n))$. No comparison sort can be substantially better than $n \log(n)$ steps and no well-constructed comparison sort should be substantially worse than $n \log(n)$ steps. From a runtime perspective, our humble merge sort implementation is within a factor of the optimal possible comparison sort.