Why do all comparison-based sorting algorithms have to run in $O(n \log n)$?

Ajitesh Dasaratha

September 2025

Ever noticed how all the best comparison-based (i.e., they work by comparing elements from the list) sorting algorithms like QuickSort and MergeSort have an $O(n \log n)$ worst-case runtime? I used to think that theoretically, some genius could come up with a better algorithm, but no one has yet. But it turns out any sorting algorithm you come up with *has* to have an $O(n \log n)$ worst-case runtime.

You can view any sorting algorithm as running a certain permutation on the input. For example, if you were to sort the list $\{3, 1, 2\}$, you need to do the following permutation of 3 elements: move the first element to the third slot, move the second and third elements one slot to the left.

In general, the number of ways to permute n elements is $n!$. And so the sorting n elements problem can be viewed as: given a list of n elements, which of its $n!$ permutations will sort it?

Every time we compare two elements a, b, there are two possible results: $a < b$ or $b < a$ (for simplicity, assume no two elements are equal). **You can have your algorithm do different things(i.e., swap/permute a different set of elements) in both cases. In some cases, if that comparison tells you you're done sorting, you could also return.**

For example, here is how you could sort 3 elements:

Algorithm 1 Brute-force sort of 3 distinct elements

```
procedure SORT3($a, b, c$)
if $a < b$ then
    if $b < c$ then
        return $(a, b, c)$
    else
        if $a < c$ then
            return $(a, c, b)$
        else
            return $(c, a, b)$
        end if
    end if
else
    if $a < c$ then
        return $(b, a, c)$
    else
        if $b < c$ then
            return $(b, c, a)$
        else
            return $(c, b, a)$
        end if
    end if
end if
```

(Note: real sorting algorithms almost never look like this; usually after each comparison, they some, and then decide what comparison comes next, then do swaps after that if needed, and so on. The combination of all these swaps results in a permutation at the end. I've just written the code like this to make this perspective of viewing the sorting problem in terms of cases very explicit.)

We treat each of the 6 permutations we might need to perform on the input as its own case. Notice that the most comparisons we ever need with this approach is 3. In some cases, we can return in just 2 comparisons: like if $a < b$ and $b < c$, we are already done.

How does this generalize to the problem with n elements?

The problem of sorting n elements has $n!$ cases, (that's how many ways you could permute n elements). We need enough comparisons to cover all cases. If you're allowed c comparisons, how many cases could you cover?
In any algorithm you make, after each comparison, you either:

- split your path into 2: if $a < b$, do x, otherwise do y
- return (as we did when we had $a < b, b < c$).

If you are maximizing the number of cases covered, you would always split your path into 2 (note: real algorithms always return in some cases, I'm only choosing the split every time to show the maximum number of paths). Since each comparison can give one of 2 results, if there are c comparisons, then you get a maximum of 2^c paths, covering that many cases.

As stated earlier, our problem has $n!$ possible cases. We need at least enough comparisons to cover all of those cases, so we need to pick a c large enough that:

$$2^c \;\geq\; n!.$$

Taking logarithm on both sides,

$$c \;\geq\; \log_2(n!).$$

It's not immediately obvious how to simplify the $\log_2(n!)$, or what that has to do with $n \log n$, so we need to do some simplifying. We can make the factorial term much easier to simplify by using Stirling's approximation, which states that for large n,

$$n! \;\approx\; \sqrt{2\pi n}\;\left(\tfrac{n}{e}\right)^n.$$

Using this, we get

$$\begin{aligned}\log_2(n!) &\approx \log_2\Big(\sqrt{2\pi n}\;\left(\tfrac{n}{e}\right)^n\Big)\\ &= \log_2(\sqrt{2\pi n}) + \log_2\big(\left(\tfrac{n}{e}\right)^n\big)\\ &= \log_2(\sqrt{2\pi n}) + n\log_2\left(\tfrac{n}{e}\right).\end{aligned}$$

Since

$$\log_2\left(\tfrac{n}{e}\right) = \log_2 n - \log_2 e,$$

the final result is

$$c > \log_2(n!) \;\approx\; \log_2(\sqrt{2\pi n}) + n\log_2 n - n\log_2 e.$$

Of these terms, the $n \log_2 n$ is the one that grows fastest asymptotically as n gets large, and so there you have it! The number of comparisons needed to sort n items grows at $O(n \log n)$, meaning any comparison-based sorting algorithm at best will have that runtime.