Designing Efficient Algorithmic Solutions From the Worst Logical Sort to Mergesort

written by junk'aal on Functor Network
original link: https://functor.network/user/3095/entry/1201

Designing Efficient Algorithmic Solutions From the Worst Logical Sort to Mergesort

June 28, 2025

1 Overview

> *"I do not know how to design efficient algorithms, but I do know how to recognize them."*

Many software developers feel this way. We learn to write step-by-step instructions for tasks like computing interest or registering payments—business processes that *directly map* to code—but when asked to solve a problem efficiently e.g., sorting a list, we realize our methods often remain naive. We can't easily go from a correct but slow solution to an optimized one, because textbooks and interview practices usually present optimized algorithms "fully formed." This essay aims to **fill that gap** by starting with the simplest and slowest solution—**permutation sort**—and incrementally **refining** it until we arrive at mergesort. By witnessing *how* an optimized algorithm emerges, we develop **true** problem-solving skills, rather than memorizing final formulas.

Sorting is just one illustration of how naive-but-correct algorithms can be **refined** into efficient solutions. The **iterative approach** below—beginning with permutation sort, gradually limiting factorial blow-up, merging at each step—**reveals** how mergesort's elegant structure emerges naturally from small, logical improvements on a brute-force baseline. This same process applies across problem-solving domains. Seeing the path from **worst** to **best** not only cements your understanding of how sorting works but also trains you to tackle **any** problem by:

- 1. Capturing it clearly,
- 2. Stating a naive solution even if factorial or exponential,
- 3. Finding ways to prune or restructure the search space,
- 4. Iterating until you reach a solution that's both correct and efficient.

In a world of "Leet coding" interviews, I think recognizing the below journey makes you a **true problem solver**, not just someone who can reproduce a memorized algorithm. You'll be prepared to **invent** or **improve** solutions for problems that go beyond the standard library—precisely the skill set that leads to **real** innovation in software engineering.

1.0.1 Capturing the Problem of Sorting

To understand sorting, it helps to formalize it: 1. We have a list l of n elements, each of which can be compared with another to decide if it is **less than**, **equal to**, or **greater than** another. 2. We say l is **sorted in ascending order** if, for every element e_i in l, $e_i \leq e_{i+1}$. 3. If any element e_i is larger than e_{i+1}, the list is not sorted. The **goal** is: given an unsorted list, produce a new list or modify the same list so that it is *sorted* in ascending order. —

1.0.2 The Naive Starting Point: Permutation Sort

1.0.3 How It Works

1. **Generate all permutations** of your list of n elements. That's $n!$ permutations.

2. **Check** each permutation to see if it's sorted.

3. **Return** the first sorted permutation you find or pick any sorted one.

1.0.4 Complexity

There are $n!$ permutations; each one requires $\mathrm{O}(n)$ time to check if it's sorted.

Overall complexity: $O(n \times n!)$, which quickly becomes intractable for $n > 8$ or so. Despite being **terribly** inefficient, it provides the ultimate baseline: it is obviously correct, and it clarifies the enormity of the sorting search space. If nothing else, this method ensures we understand exactly *what* the problem entails.

1.0.5 A Small Step Forward: Divide the List, Then Permute A first refinement is to realize:

- Permuting n elements yields $n!$ permutations.

- But permuting $\frac{n}{2}$ elements yields $!(\left(\frac{n}{2}\right))$—still factorial, but *less* explosive than $n!$.

Hence:

- 1. **Divide** your list of size n into two halves of size $\frac{n}{2}$.

- 2. **Permutation sort** each half now you're generating $!(\left(\frac{n}{2}\right))$ permutations *twice*.

- 3. **Merge** the two sorted halves. Merging takes $\mathrm{O}(n)$ time.

Why it's better: $!(\left(\frac{n}{2}\right))$ is *much* smaller than $n!$ for larger n. For instance, $8! = 40320$, while $4! = 24$. Doing $2 \times 24 = 48$ permutations is far less than 40320. In big-O notation it's still factorial, but it's an example of a **divide-and-conquer** improvement over enumerating *all* permutations of the entire list at once.

1.0.6 Extending the idea: Recursively Divide & Permute Smaller Sub-Lists Why stop at one division?

We can **recursively** keep dividing each sub-list until it reaches some small size k. Then:

1. **If the sub-list size** $\leq k$, use permutation sort it's affordable if k is small.

2. **Otherwise**, keep dividing in half.

3. **Merge** the sub-lists once they're individually "sorted."

1.0.7 Complexity Discussion

- If you pick $k = n/2$, each half is still large, so you'll face $\left(\frac{n}{2}\right)$. - If k is a **fixed constant** like 2, 3, or 4, then enumerating permutations in a sub-list of size k takes $\mathrm{O}(k!)$ operations—still technically factorial, but *bounded* by a small constant.

- Once each sub-list of size k is sorted, merging them follows a structure similar to merge-sort—linear merges per sub-list, and $\log n$ levels of merging. The total cost of merging is $\mathrm{O}(n \log n)$.

Hence, **if you fix** $k = 2$, enumerating "all permutations" of a 2-element list is effectively just "check and swap if out of order." This is $\mathrm{O}(1)$. All the rest of the work is in merging sorted sub-lists, level by level, which is how mergesort arrives at $\mathrm{O}(n \log n)$.

1.0.8 Realizing This Is Basically Mergesort

Mergesort is normally introduced as:

> 1. If the list has size 1, it's sorted trivial.
> 2. Otherwise, split into halves, recursively sort each half, then merge.

But you can **also** see mergesort as:

> 1. Keep dividing until you have sub-lists of size 2 or 1.
> 2. "Permutation sort" each 2-element sub-list only 2 permutations to consider.
> 3. Merge your way up. Both descriptions produce $\mathrm{O}(n \log n)$ complexity.

The difference is *pedagogical*: typical textbooks skip the "permutation enumeration" angle because it's usually regarded as an impractical approach for sub-lists beyond a tiny size. Yet, from a *teaching perspective*, this path from "**full factorial**" to "**factorial only on *very* small sub-lists**" to "**pure merges**" is (at least to me) enlightening.