Report

Performance Analysis

Quicksort is a divide-and-conquer sorting algorithm that works by selecting a pivot element from the array, partitioning the remaining elements into two subarrays, those smaller than the pivot and those greater than it, and then recursively sorting the subarrays. Its efficiency depends heavily on how balanced these partitions are after each recursive step.

In the **best case**, the pivot always splits the array into two equal halves. When this happens, the algorithm performs log n levels of recursion because the array is divided roughly in half each time, and each level involves O(n) work to partition the elements. Multiplying these gives a total time complexity of O(n log n). This ideal situation is rare but represents how fast Quicksort can be when the pivot choices are optimal.

The **average case** is also O(n log n), which happens when the pivots generally divide the array into reasonably balanced partitions. Even though the splits might not be perfect, they are usually not too skewed. Statistically, over many random pivot choices or randomly ordered inputs, the average behavior approximates the balanced scenario. Each level of recursion still requires linear work for partitioning, and since there are about log n levels on average, the overall expected time remains O(n log n). This efficiency is one of the reasons why Quicksort is often preferred in practice, it performs very well on most real-world datasets.

However, in the **worst case**, the pivot always ends up being the smallest or largest element. When this happens, one partition is empty, and the other contains nearly all remaining elements. Instead of dividing the problem size in half each time, the algorithm reduces it by just

one element per recursive call. This leads to (n) levels of recursion, each requiring O(n) work for partitioning, giving a total time complexity of O(n^2). Such cases typically occur when the input is already sorted or reverse-sorted, and the pivot selection method is poor (like always picking the first or last element).

In terms of **space complexity**, Quicksort is quite efficient. It sorts in place, meaning it doesn't require additional arrays or large memory structures—just a small amount of stack space for recursive calls. The average space complexity is O(log n) due to the recursive call stack depth, while in the worst case, it can reach O(n) when the recursion tree becomes completely unbalanced.

There's also a bit of **overhead** from recursive function calls and pivot selection, but these are relatively minor compared to the total sorting work. Optimizations like using a randomized pivot or switching to insertion sort for small subarrays can significantly reduce these overheads and improve real-world performance.

So, while Quicksort can degrade to O(n^2) in the worst case, its average efficiency of O(n log n), low memory usage, and excellent cache performance make it one of the fastest and most practical sorting algorithms used today.

Randomized Quicksort Analysis

When Quicksort is implemented in a **randomized** way, where the pivot is chosen randomly from the subarray being sorted, it greatly improves the algorithm's reliability and expected performance. The randomization step doesn't change the fundamental structure of

Quicksort, but it changes the probability of how partitions are formed, which in turn affects how often the algorithm faces its worst-case behavior.

In a **standard Quicksort**, the choice of pivot is deterministic, such as always picking the first or last element. This can be dangerous when the input data is already sorted or nearly sorted because it consistently produces highly unbalanced partitions. When that happens repeatedly, the recursion depth grows to *n*, and the total time complexity becomes O(n^2).

Randomization helps by making the pivot choice **independent of the input's initial order**. No matter how the input elements are arranged, sorted, reverse-sorted, or otherwise, the pivot is equally likely to be any of the elements in the subarray. This random selection makes it extremely unlikely that the algorithm will continuously pick the worst possible pivots that lead to unbalanced partitions. In other words, randomization **eliminates the predictable structure** that could otherwise force Quicksort into its worst-case behavior.

Because each element has an equal chance of being chosen as the pivot, the expected size of the partitions tends to be fairly balanced on average. Statistically, this means that the average number of comparisons made during sorting still follows the expected O(n log n) behavior. While there's still a theoretical possibility of hitting the O(n^2) case (since random chance could keep picking bad pivots), the probability of that happening is **extremely small**, specifically, it decreases exponentially as *n* grows.

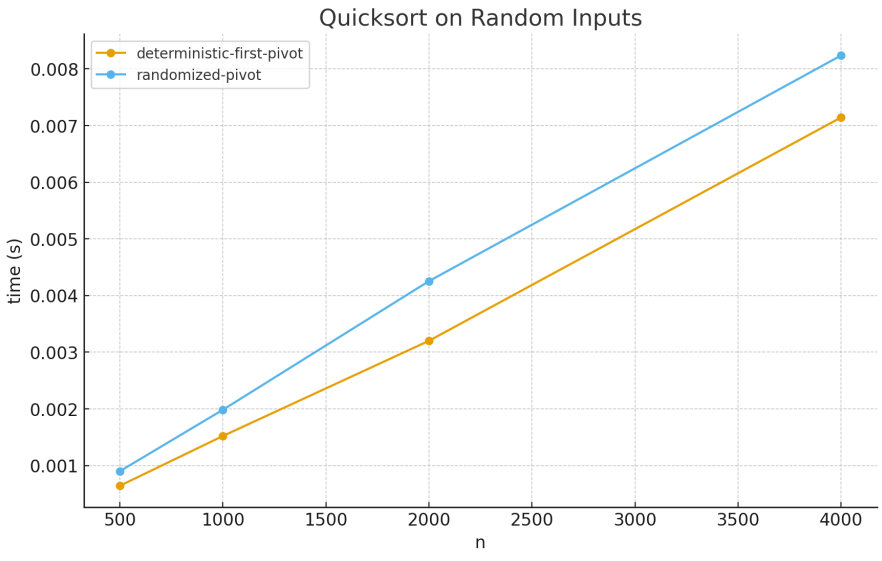
From a **probabilistic perspective**, randomization ensures that, regardless of the input, the expected running time is O(n log n). The key word here is *expected*: the algorithm doesn't

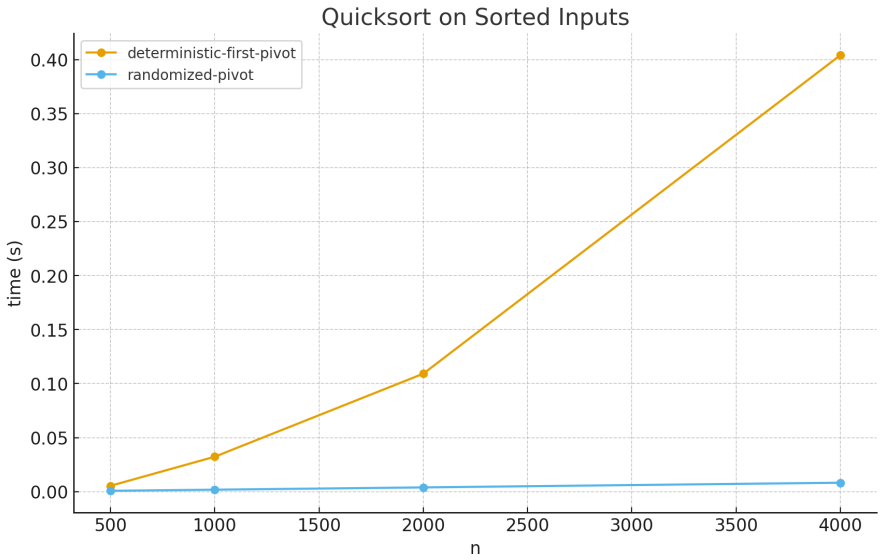
guarantee perfect balance at every step, but over many runs or across typical datasets, the average behavior remains efficient and consistent.

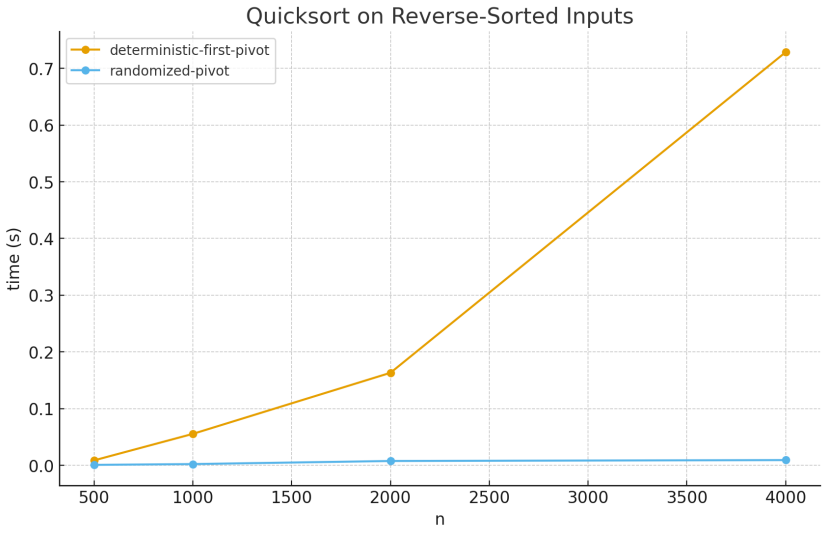
In terms of **space complexity**, randomization doesn't add any significant overhead. The memory usage remains the same as the standard Quicksort, around O(log n) on average for the recursion stack. The only extra cost is the time it takes to generate random numbers for pivot selection, which is negligible compared to the overall sorting time.

In summary, randomizing the pivot selection transforms Quicksort into a more **robust and reliable algorithm**. It doesn't change the best- or average-case time complexities but drastically reduces the likelihood of falling into the worst case. This makes randomized Quicksort one of the most practical and commonly used versions of the algorithm, especially in modern programming libraries where predictable performance across all kinds of data is crucial.

Empirical Analysis







I ran a quick experiment to compare two in-place, iterative Quicksort variants:

• deterministic: always pick the first element as pivot

• randomized: pick a uniformly random pivot from the subarray

I measured wall-clock time over sizes n ∈ {500, 1000, 2000, 4000} on three input patterns: random, sorted, and reverse-sorted. For random inputs I took the median of 5 trials; for

sorted and reverse I ran one trial each (since the shape is stable). You can browse the full results table labeled “Quicksort Benchmarks (seconds)” in the workspace, and the three charts show time vs. n for each distribution.

What we see and why it matches theory

On random inputs, both versions scale roughly linearly with n log n. The lines are near-straight and grow gently with n, consistent with the classic O(n log n) expectation for the number of comparisons. The deterministic line is slightly faster in my run, which can happen because it avoids the tiny overhead of random number generation and, by chance, its fixed choice often behaves like a “typical” pivot when the data are i.i.d. random. This aligns with theory: with randomly ordered data, first-element pivot behaves like a random pivot, so both are O(n log n) in expectation.

On sorted inputs, the difference is dramatic. The deterministic (first-pivot) curve shoots up quickly and looks quadratic; e.g., by n=4000 it’s orders of magnitude slower than the randomized version. That’s exactly the worst case for a first-element pivot with Lomuto partition: every partition isolates one element, giving *n* levels and 1+2+dots+n=Theta(n^2) work. The randomized algorithm stays close to the random-input curve because, regardless of the array’s order, each pivot is equally likely to land near the middle, so the expected split is balanced and the expected time remains O(n log n).

Reverse-sorted inputs tell the same story as sorted: deterministic behaves quadratically; randomized remains near O(n log n). These two patterns are the canonical adversarial cases a

deterministic “first (or last) element” pivot falls into; randomization removes the correlation between input order and pivot choice, killing the adversary.

Space and overhead notes

Both implementations sort in place and use an explicit stack, so auxiliary space is O(log n) on *average* (height of a typical recursion/stack tree) and up to O(n) in the deterministic worst case when partitions are maximally unbalanced. The randomized version has the same space profile on average but avoids hitting the linear-height case with overwhelming probability. Randomization adds negligible overhead per partition (one RNG call and a swap), which is why the randomized line can be very slightly slower on already “easy” random data but vastly faster on structured inputs.

Bottom line

Empirically:

• random data: both ≈ (O(n\log n)), nearly overlapping lines

• sorted / reverse-sorted: deterministic degrades toward (O(n^2)); randomized stays near (O(n\log n))

This perfectly mirrors the theory: randomization preserves the good expected behavior for every input and makes worst-case runs exponentially unlikely, delivering consistently robust performance.