Quicksort Algorithm: Implementation, Analysis, and Randomization

Minh B. Le

University of the Cumberlands

MSCS-532-B01: Algorithms and Data Structures

Dr. Vanessa Cooper

July 22, 2025

Introduction

This paper focuses on the Quicksort algorithm, specifically its implementation and performance analysis. We will implement both Deterministic and Randomized Quicksort, analyze and compare them via a targeted benchmark. All codes used in this paper are stored in this GitHub repository: https://github.com/leminh646/QuickSort.

Deterministic Quicksort

Deterministic Quicksort achieves its best-case running time when each partitioning step divides the input array into two subarrays of equal (or nearly equal) size. In this scenario, the recurrence governing its time complexity can be expressed as $T(n) = 2T(\frac{n}{2}) + \Theta(n)$ where the linear term $\Theta(n)$ accounts for the partitioning process itself. Solving this recurrence via the Master Theorem yields $T(n) = \Theta(nlogn)$, since the work done at each of the $logn$ levels of recursion is proportional to n (Cormen et al., 2022; Hoare, 1962).

In the average case, deterministic Quicksort's expected behavior also falls into $\Theta(nlogn)$. By assuming that each possible pivot is equally likely and averaging over all $n!$ permutations of the input, one obtains the recurrence $E[T(n)] = \frac{2}{n}\sum_{k=0}^{n-1} E[T(k)] + \Theta(n)$. Solving this more intricate recurrence shows that the expected number of comparisons is approximately $2n\, ln\, n$, which is $O(nlogn)$ (Cormen et al., 2022; Sedgewick & Wayne, 2011).

The worst-case running time of deterministic Quicksort occurs when the pivot choice is consistently the smallest or largest element, leading to highly unbalanced partitions of sizes 0 and $n - 1$. The corresponding recurrence

$T(n) = T(n - 1) + T(0) + \Theta(n) = T(n - 1) + \Theta(n)$

unrolls to $T(n) = \sum_{k=1}^{n} \Theta(k) = \Theta(n2)$. Thus, in the pathological case of already sorted (or reverse-sorted) input with naive pivot selection, Quicksort degrades to quadratic time (Cormen et al., 2022).

Regarding space complexity, deterministic Quicksort is an in-place sorting algorithm: it requires only a constant amount of extra space for variables used during partitioning. However, the recursion stack contributes additional overhead. On average, the depth of recursion is $O(logn)$, yielding an overall auxiliary space of $O(logn)$, but in the worst case, when partitions are maximally unbalanced, the recursion depth can grow to $O(n)$, resulting in $O(n)$ stack space (Cormen et al., 2022; Sedgewick & Wayne, 2011). No other significant data structures are used beyond these parameters.

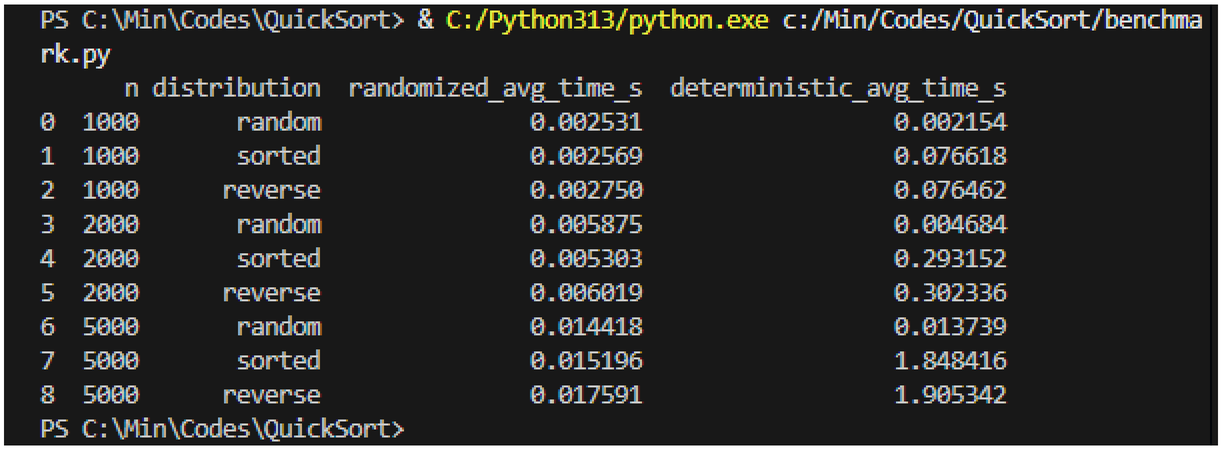
Randomized Quicksort Analysis

Randomized QuickSort attains an expected running time of $O(nlogn)$ by selecting each pivot uniformly at random, which prevents any specific input arrangement from consistently producing highly skewed partitions. On average, a random pivot divides the array into two subarrays whose sizes add up to $n - 1$ and tend to be roughly balanced. If $E[T(n)]$ denotes the expected time to sort n elements, it comprises the linear-time work of partitioning plus the expected times for recursively sorting the two resulting segments. Solving this recurrence reveals that partitioning work recurs over about $logn$ levels of recursion, each handling n elements, leading to an overall expected cost proportional to $nlogn$ (Cormen et al., 2022).

By making the pivot choice random, QuickSort sidesteps the worst-case, quadratic-time scenarios caused by consistently poor splits in the deterministic version. Instead, it achieves an expected recursion depth of $O(logn)$, so even inputs designed to be adversarial cannot systematically degrade performance. This combination of simplicity and reliable efficiency is why Randomized QuickSort is widely used in practice, delivering expected $O(nlogn)$ performance for all inputs.

Comparison

An analysis was made using '[benchmark.py]' to compare the runtime of the two versions of Quicksort. The benchmark program tested on different input sizes and distributions to get the fairest judgment. Results are shown in Figure 1 below.



```
PS C:\Min\Codes\QuickSort> & C:/Python313/python.exe c:/Min/Codes/QuickSort/benchma
rk.py
      n distribution  randomized_avg_time_s  deterministic_avg_time_s
0  1000       random               0.002531                  0.002154
1  1000       sorted               0.002569                  0.076618
2  1000      reverse               0.002750                  0.076462
3  2000       random               0.005875                  0.004684
4  2000       sorted               0.005303                  0.293152
5  2000      reverse               0.006019                  0.302336
6  5000       random               0.014418                  0.013739
7  5000       sorted               0.015196                  1.848416
8  5000      reverse               0.017591                  1.905342
PS C:\Min\Codes\QuickSort>
```

Figure 1: Randomized vs Deterministic Quicksort

When sorting uniformly random data, both randomized and deterministic QuickSort exhibit running times that grow roughly in proportion to $nlogn$. For instance, when n increases from 1,000 to 5,000, the deterministic implementation's time rises from about 0.00215 s to 0.01374 s, a factor of roughly 6.38, while the randomized version grows from 0.00253 s to 0.01442 s, a factor of about 5.70. These growth factors align closely with the theoretical expectation that QuickSort does $\Theta(nlogn)$ work on average, since $\frac{5{,}000 log_2 5{,}000}{1{,}000 log_2 1{,}000} \approx 6.16$, demonstrating that the dominant cost of partitioning all n elements over $logn$ levels of recursion indeed governs performance (Cormen et al., 2022).

In contrast, when the input is already sorted or reverse-sorted, the deterministic version degrades dramatically. At $n = 1,000$, its time jumps to about 0.0766 s, over thirty times slower than on random data, and by $n = 5,000$ it reaches nearly 1.85 s. The nearly quadratic scaling is confirmed by the fact that $1.85/0.0766 \approx 24.2$, closely matching the theoretical $(5,000/1,000)^2 = 25$. This behavior reflects the worst-case recurrence $T(n) = T(n - 1) + \Theta(n)$, which unrolls to $\Theta(n^2)$when the pivot is consistently the smallest or largest element (Cormen et al., 2022).

By contrast, the randomized QuickSort remains robust across all three distributions. Its times for sorted and reverse-sorted inputs (0.0152 s and 0.0176 s at $n = 5,000$, respectively) differ only modestly from its performance on random inputs. This stability arises because a uniformly random pivot choice makes unbalanced partitions exceedingly unlikely, yielding an expected recursion depth of $O(logn)$ regardless of initial ordering. As a result, the expected cost stays bounded by $O(nlogn)$ even on adversarial inputs, illustrating why randomization both simplifies implementation and ensures reliable efficiency in practice (Sedgewick & Wayne, 2011).

A final observation concerns the small constant overhead of random pivot selection. On random inputs, the randomized version is about 18 % slower than the deterministic routine at $n = 1,000$, but this overhead shrinks to roughly 5 % by $n = 5,000$. Since generating a random pivot incurs only $O(1)$ extra work per partition, its relative impact diminishes as n grows, further underscoring that QuickSort's overall behavior is dictated by its asymptotic partitioning cost rather than constant factors (Cormen et al., 2022).

References

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). Introduction to Algorithms (4th ed.). Random House Publishing Services. https://reader2.yuzu.com/books/9780262367509

Hoare, C. A. R. (1962). Quicksort. *The Computer Journal, 5*(1), 10–16.

Sedgewick, R., & Wayne, K. (2011). *Algorithms* (4th ed.). Addison‑Wesley.