Algorithm Efficiency and Scalability

Minh B. Le

University of the Cumberlands

MSCS-532-B01: Algorithms and Data Structures

Dr. Vanessa Cooper

July 17, 2025

Introduction

In this paper, we explore two fundamental algorithms—Randomized Quicksort and Hashing with Chaining—through both theoretical analysis and practical experiments. We examine how random pivot selection drives Quicksort's average-case $O(nlogn)$ performance and how simple uniform hashing delivers expected constant-time operations when the load factor is controlled. By implementing each method in Python and benchmarking across varied datasets, we validate our analytical insights and reveal real-world considerations like constant factors and memory behavior. This study sharpens our ability to evaluate, select, and tune algorithms for efficiency and scalability in diverse applications. All source codes are stored in this GitHub repository: https://github.com/leminh646/Algorithm-Efficiency-and-Scalability.

Randomized Quicksort Analysis

We will implement a version of quicksort where the pivot is chosen at random at each partition, analyze its theoretical complexity, and then compare it with real-time performances using a benchmark. The source code can be found in the folder randomizedQuicksort in the provided GitHub repository, where a README file is provided on how to run the program.

Analysis

Randomized QuickSort achieves its expected $O(nlogn)$ running time by choosing each pivot uniformly at random, which ensures that no particular input ordering can force consistently unbalanced partitions. One can analyze this behavior by observing that, on average, a randomly selected pivot will split the array into two subarrays whose sizes sum to $n - 1$ and are "balanced" in expectation. If we let $E[T(n)]$ denote the expected time to sort n elements, then $E[T(n)]$ consists of the linear-time work to partition around the pivot plus the expected costs of sorting the two resulting subarrays. Solving this recurrence shows that the partition cost

accumulates across roughly $logn$ levels of recursion, each processing all n elements, giving an overall expected bound proportional to $nlogn$ (Cormen et al., 2022).

By randomizing the pivot choice, QuickSort avoids the pathological unbalanced splits that lead to quadratic behavior in the deterministic version. Instead, it achieves logarithmic recursion depth in expectation, ensuring that even adversarial inputs cannot systematically degrade its performance. This is why Randomized QuickSort is both simple to implement and robustly efficient in practice, running in expected $O(nlogn)$ time on any input.

Comparison

We will compare the running time of Randomized Quicksort with Deterministic Quicksort using the provided code in the GitHub repository. Results are shown in Table 1 below.

Input Size	Distribution	Randomized QS (s)	Deterministic QS (s)
1000	random	0.002506	0.002364
1000	sorted	0.003037	0.082571
1000	reverse	0.003267	0.082748
1000	repeated	0.000565	0.000531
2000	random	0.006862	0.004869
2000	sorted	0.006062	0.321892
2000	reverse	0.005444	0.321692
2000	repeated	0.001091	0.001025
5000	random	0.017799	0.014076
5000	repeated	0.002600	0.002505

Table 1: Randomized and Deterministic QuickSort Comparison

On randomly generated inputs, the benchmark shows that Randomized QuickSort processes 1,000 elements in roughly 0.0025 s and 5,000 elements in about 0.0178 s, while

Deterministic QuickSort (first-element pivot) completes the same tasks in approximately 0.00236 s and 0.01408 s, respectively. Although the random‑pivot selection introduces a small constant overhead, both algorithms scale in a manner consistent with the $nlogn$ growth predicted by theory. In particular, doubling the input size from 1,000 to 2,000 increases runtime by a factor of about 2.7 for the randomized version and about 2.06 for the deterministic version, closely matching the expectation that QuickSort's cost grows proportional to $nlogn$ on average (Cormen et al., 2022).

By contrast, when applied to already sorted or reverse-sorted arrays, Deterministic QuickSort degrades dramatically, taking roughly 0.0826 s at $n = 1,000$ and 0.322 s at $n = 2,000$. These timings scale quadratically, nearly quadrupling when the input size doubles, precisely the behavior of its worst-case $\Theta(n^2)$ bound. Randomized QuickSort on the same sorted inputs, however, requires only about 0.0030 s at $n = 1,000$ and 0.0061 s at $n = 2,000$, demonstrating its resilience to adversarial orderings. Its runtimes roughly double with each doubling of n, confirming that random pivot selection restores the expected $O(nlogn)$ performance even on inputs that are pathological for a fixed-pivot strategy (Cormen et al., 2022).

Arrays with many repeated elements exhibit an interesting intermediate behavior. For $n = 1,000$, Randomized QuickSort sorts in approximately 0.000565 s and Deterministic QuickSort in 0.000531 s; at $n = 5,000$, these times grow to about 0.0026 s and 0.0025 s. Because our implementations group all elements equal to the pivot into a single "equal" partition, the recursion depth is greatly reduced when there are many identical keys. As a result, both algorithms approach linear scaling in practice, with runtimes increasing by factors close to the increase in n. The small differences between randomized and deterministic versions stem primarily from the cost of generating random indices, rather than from fundamentally different

recursion depths. Overall, the empirical findings corroborate the theoretical insight that Randomized QuickSort achieves robust $O(nlogn)$ expected time on all inputs, while the deterministic first-element variant suffers quadratic blow-up on already ordered data (Cormen et al., 2022).

Hashing with Chaining

This part of the paper aims to implement a hash table using chaining for collision resolution. Source code for the implementation can be found in the 'hashTableChaning' folder in the provided GitHub repository.

Analysis

Under the assumption of simple uniform hashing, each key is equally likely to land in any of the m buckets, so the expected number of keys in a given chain is $\alpha = n/m$. Consequently, the expected time to search for, insert, or delete a key is proportional to the length of its chain plus a constant overhead for computing the hash. Formally, each of these operations runs in $O(1 + \alpha)$ time in expectation (Cormen et al., 2022). When α is small, on the order of a constant, the table delivers amortized constant-time performance for all three operations, making chaining an efficient collision-resolution strategy under random hashing.

The load factor α directly governs performance: as more elements n are inserted into a fixed number of buckets m, α grows and with it the average chain length. Longer chains mean each search or deletion must scan more entries, and insertions (when avoiding duplicates) must likewise traverse existing entries before appending. Thus, if one allows α to approach or exceed 1, the expected cost of an operation degrades to $O(\alpha)$, losing the constant-time advantage. Keeping α bounded by a small constant ensures that the expected time remains $O(1)$, even though individual chains may occasionally be longer by chance (Mitzenmacher & Upfal, 2005).

To maintain a low load factor and minimize collisions, hash tables commonly employ dynamic resizing: whenever α exceeds an upper threshold (e.g., 0.75), the table size m is doubled and all entries are rehashed; if α falls below a lower threshold (e.g., 0.25), the table may be halved (Cormen et al., 2022). This amortizes the $O(n)$ cost of rehashing over many $O(1)$ operations, preserving expected constant-time performance. Complementing resizing, one chooses a robust hash function, often from a universal family, to randomize key placement and avoid pathological clustering. Together, these strategies ensure that chaining remains efficient even as the dataset grows or shrinks dynamically.

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

Mitzenmacher, M., & Upfal, E. (2005). Probability and Computing: Randomized Algorithms and Probabilistic Analysis. Cambridge University Press.