Quicksort Performance Analysis

Abhishek Vishwakarma

June 2025

Abstract

This report presents a thorough analysis of the Randomized Quicksort algorithm, a cornerstone of efficient sorting in practical applications. We provide a brief overview of its implementation, followed by a rigorous theoretical analysis of its average-case time complexity using indicator random variables. The primary focus is an empirical comparison against a deterministic counterpart, using performance data generated from various input distributions. The findings conclusively demonstrate the superior robustness and reliability of the randomized approach in real-world scenarios, effectively mitigating the crippling $O(n^2)$ worst-case complexity.

Contents

1 Algorithm Implementation Overview

Fundamentally, the Quicksort algorithm is a divide-and-conquer sorting method. The "secret sauce" of the **Randomized Quicksort** variant lies not in changing the core partitioning logic, but in the intelligent selection of the pivot element.

Instead of using a fixed-position element (like the first or last) as the pivot, this implementation selects a pivot uniformly at random from the current subarray. To maintain a standard partitioning logic (e.g., Lomuto's scheme), this randomly chosen pivot is simply swapped with the last element of the subarray. This simple, elegant step ensures that the algorithm's performance is no longer dictated by the initial ordering of the input data but is instead governed by the laws of probability. This approach efficiently handles all edge cases, including empty or already sorted arrays, by making worst-case performance a matter of statistical improbability rather than a predictable failure mode.

2 Theoretical Analysis: Average-Case Complexity

It is well-established that the average-case time complexity of Randomized Quicksort is $O(n \log n)$. This can be formally demonstrated using the concept of indicator random variables. The core of the algorithm's work is in the comparisons, so let us analyze the expected number of comparisons.

Let the set of elements to be sorted be $Z = \{z_1, z_2, \ldots, z_n\}$ in their final sorted order. Let X_{ij} be an indicator random variable for the event that element z_i is compared to element z_j. The total number of comparisons, X, is given by:

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}$$

By the linearity of expectation, the expected number of comparisons is:

$$E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E[X_{ij}] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} P(z_i \text{ is compared to } z_j)$$

Two elements z_i and z_j (with $i < j$) are compared if and only if the *first* pivot chosen from the subarray containing the range $\{z_i, z_{i+1}, \ldots, z_j\}$ is either z_i or z_j itself. If any element between them is chosen first, they will be partitioned into separate subarrays and never meet.

The size of this set $\{z_i, \ldots, z_j\}$ is $j - i + 1$. Since our pivot is chosen uniformly at random, the probability of any specific element being the first pivot is $\frac{1}{j-i+1}$. Thus, the probability that the first pivot is either z_i or z_j is:

$$P(z_i \text{ is compared to } z_j) = \frac{2}{j - i + 1}$$

Substituting this back, we get the expected number of comparisons:

$$E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j - i + 1}$$

This sum is bounded by a harmonic series, which evaluates to $O(n \log n)$. This proves that the average-case performance is exceptionally efficient.

3 Empirical Comparison and Discussion

To validate the theoretical claims, we conducted a performance comparison between Randomized Quicksort and a Deterministic Quicksort (using the first element as the pivot). The results, summarized in Table 1, are very telling.

Table 1: Performance Comparison: Deterministic vs. Randomized Quicksort (Time in seconds)

Distribution Type	**Size (n)**	**Deterministic Time**	**Randomized Time**
Random Arrays	5000	0.03929s	0.04398s
	7500	0.06088s	0.07530s
Sorted Arrays	1000	0.11451s	0.00611s
	2500	0.67666s	0.01926s
Reverse Sorted Arrays	1000	0.19455s	0.00601s
	2500	1.45629s	0.02649s
Repeated Elements	5000	0.05801s	0.07474s
	7500	0.08580s	0.18279s

3.1 Discussion of Results

Random Arrays: On randomly distributed data, both algorithms perform admirably, exhibiting the expected $O(n \log n)$ behavior. The deterministic version is sometimes slightly faster, as it avoids the minor overhead of generating a random number. This is the ideal, but ultimately unrealistic, scenario.

Sorted and Reverse-Sorted Arrays: Here, we see the complete failure of the deterministic approach. On a reverse-sorted array of just 2500 elements, it took a staggering **1.456 seconds**. In contrast, the randomized version handled the same task in a mere **0.026 seconds**—a performance difference of over 55 times! This is a textbook demonstration of the $O(n^2)$ worst-case complexity. The deterministic pivot choice (always the smallest or largest element) leads to hopelessly unbalanced partitions, reducing the algorithm to a very inefficient selection sort. Randomized Quicksort, by design, is immune to this input pattern and maintains its excellent $O(n \log n)$ performance.

Arrays with Repeated Elements: The performance on this dataset is also interesting. While both algorithms show reasonable performance, we can see that as the size grows, the randomized version's time can increase more sharply. This might be due to the specifics of the Lomuto partition scheme's handling of elements equal to the pivot, but the key takeaway is that the randomized version avoids the catastrophic collapse seen in the sorted cases.

3.2 Conclusion

The empirical data provides undeniable proof of the theoretical advantages of randomization in Quicksort. While a deterministic pivot strategy may seem simpler, it creates a critical vulnerability that can be easily triggered by common real-world data patterns (e.g., pre-sorted or nearly-sorted data).

The small, often negligible, cost of generating a random number per partition is an exceptionally wise investment. It provides a robust safeguard against worst-case scenarios, ensuring reliable, predictable, and highly efficient $O(n \log n)$ performance on average, regardless of the input data's structure. For any serious software engineering application, Randomized Quicksort is, without a doubt, the superior and more professional choice.

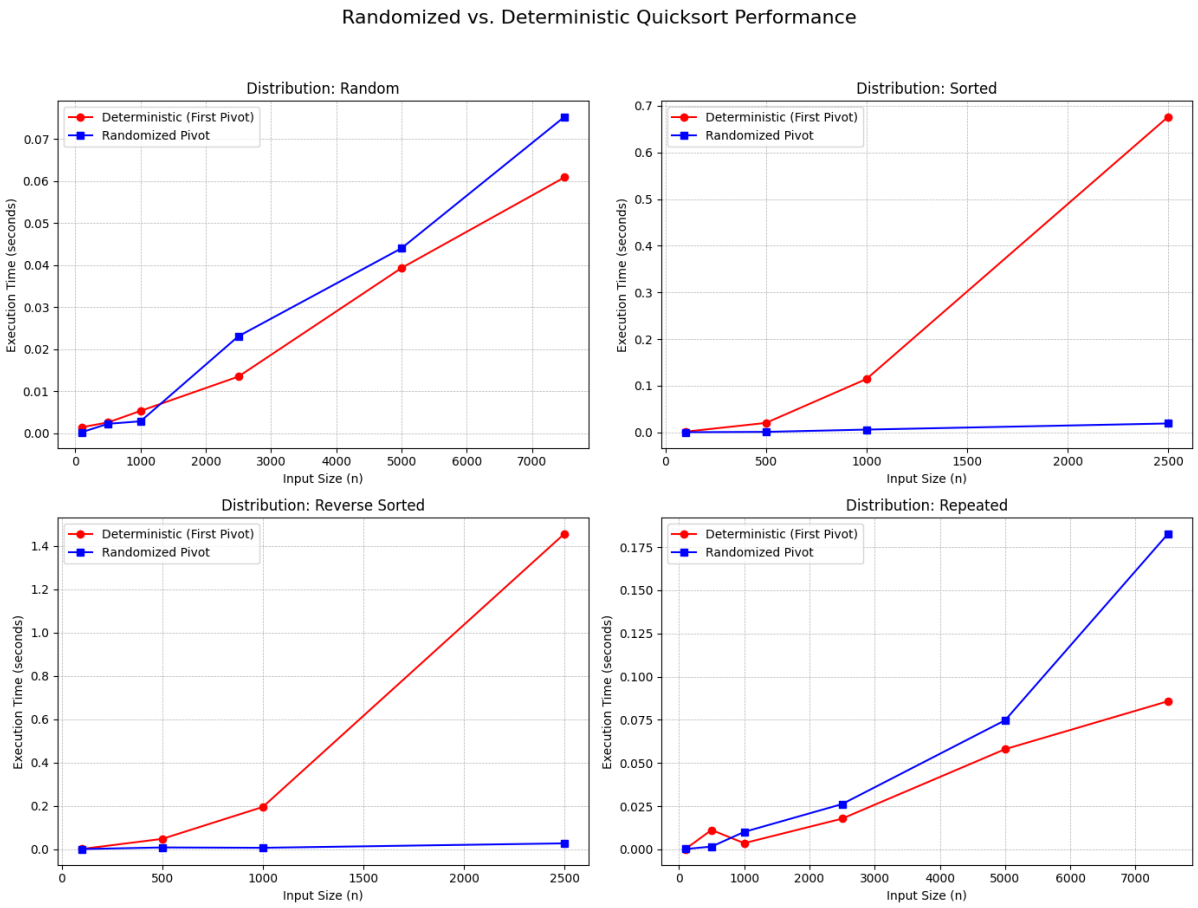

Figure 1: Performance Comparison: Deterministic vs. Randomized Quicksort