Assignment 3: Average-Case Time Complexity Analysis of Randomized Quicksort

Ashim Sedhain

University of the Cumberlands

MSCS-532-M20: Algorithms and Data Structures

June 14, 2025

Average-Case Time Complexity Analysis of Randomized Quicksort

This section presents a rigorous analysis of the average-case time complexity of the Randomized Quicksort algorithm. Using probabilistic analysis and recurrence relations, it is shown that the expected runtime is $O(n \log n)$. Empirical benchmarking results on various input types are also discussed and compared with the theoretical model, highlighting both alignment and divergence due to data-specific behaviors and implementation details.

Theoretical Average-Case Complexity

Randomized Quicksort is a divide-and-conquer algorithm that selects a pivot uniformly at random from the subarray and partitions the remaining elements into two subarrays: those less than the pivot and those greater than or equal to it. The algorithm recursively sorts the two partitions.

Let $T(n)$ denote the expected number of comparisons required to sort an array of size n. Since the pivot is chosen randomly, each pair of distinct elements is compared at most once. Let us define indicator random variables $X_{i,j}$ $1 \leq i < j \leq n$, where:

$$X_{i,j} = \begin{cases} 1 \; if \; elements \; a_i \; and \; a_j \; are \; compared \\ 0 \qquad\qquad\qquad\qquad otherwise \end{cases}$$

The expected total number of comparisons is:

$$\mathbb{E}[T(n)] = \sum_{1 \leq i \leq j \leq n} \mathbb{E}\left[X_{i,j}\right]$$

A pair (a_i, a_j) is compared only if one of them is the first to be chosen as a pivot among the elements in the subarray containing both. Since each element has an equal chance to be selected as the pivot, the probability that a_i and a_j are compared is:

$$\mathbb{E}[X_{i,j}] = \frac{2}{j - i + 1}$$

Summing over all such pairs, we get:

$$\mathbb{E}[T(n)] = \sum_{1 \le i \le j \le n} \frac{2}{j - i + 1} = \mathrm{O}(\mathrm{n} \log n)$$

Thus, the expected number of comparisons and, by extension, the average-case time complexity of Randomized Quicksort is $O(n \log n)$.

Empirical Results and Observations

Benchmarking results from the implemented Python version of Randomized Quicksort are summarized below:

Input Type	**Randomized Quicksort (s)**	**Deterministic Quicksort (s)**
Random Array	0.0907	0.0605
Sorted Array	0.0740	1.0296

Input Type	Randomized Quicksort (s)	Deterministic Quicksort (s)
Reverse-Sorted Array	0.0747	1.7737
Repeated Elements	2.2840	2.2527

These observations align with theoretical expectations:

1. **Random Inputs:**

   Both randomized and deterministic versions show good performance on random arrays. The deterministic variant happens to be slightly faster here, likely due to fewer random operations and the small array size (1000 elements). However, the difference is marginal and not asymptotically significant.

2. **Sorted and Reverse-Sorted Arrays:**

   Deterministic Quicksort performs poorly, as always picking the first element as pivot leads to unbalanced partitions and $O(n^2)$ time. In contrast, Randomized Quicksort avoids this degenerate case by randomly selecting the pivot, maintaining its $O(n \log n)$ average-case performance even on sorted inputs.

3. **Repeated Elements:**

   Both versions exhibit a significant slowdown on inputs with repeated elements. This is due to inefficient partitioning, as many elements are equal to the pivot and contribute no progress in reducing problem size. Randomized Quicksort does not inherently optimize for repeated elements, so the performance degrades toward $O(n^2)$ in practice.

Discussion of Discrepancies

Theoretically, the average-case performance of Randomized Quicksort is robust across input types. However, empirical measurements may exhibit deviations due to the following reasons:

- **Constant Factors and Language Overhead:** Python's function calls and recursion overhead can make algorithms like deterministic Quicksort appear faster on small inputs due to fewer random operations.
- **System-Level Noise:** Factors such as cache behavior, memory access patterns, and Python's dynamic typing can introduce variability.
- **Data Distribution:** The worst-case scenarios for deterministic Quicksort are triggered precisely by sorted or reverse-sorted inputs, which aligns with the large observed runtimes. In contrast, Randomized Quicksort benefits from input obliviousness, which maintains expected performance.

Conclusion

Randomized Quicksort achieves $O(n \log n)$ expected time complexity in the average case, supported both by rigorous probabilistic analysis and empirical measurements. Its robustness across input types makes it a practical sorting algorithm in real-world applications. Nevertheless, its performance can be degraded on inputs with many repeated elements, highlighting the importance of algorithmic refinements such as multi-way partitioning in such cases.