Assignment 5: Quicksort Algorithm: Implementation, Analysis, and Randomization

Anushka Nanaware

STUDENT ID: 005042356

Department of Computer Science

University of the Cumberlands

MSCS532 - M80 Algorithms and Data Structures

Dr. Micahel Solomon

February 22, 2026

GITHUB LINK:

Sorting is one of the most fundamental operations in computer science. Many real-world systems rely on efficient sorting, including database indexing, search engines, data processing pipelines, and large-scale distributed systems such as Hadoop and Spark. Because sorting is so common, choosing an efficient algorithm is critical for performance and scalability.

Quicksort, introduced by C. A. R. Hoare, is one of the most widely used comparison-based sorting algorithms and the strategy that is used by it is called as the divide-and-conquer strategy:

1. A pivot element will be chosen
2. The array will be divided/partition at the point of pivot.
3. Recursively sort the two subarrays.

Although Quicksort has a worst-case time complexity of **$O(n^2)$**, its expected running time is **$O(n \log n)$**, which makes it extremely efficient in practice (Cormen et al., 2022).

This assignment implements both deterministic and randomized versions of Quicksort, analyzes their theoretical performance, and compares them empirically using multiple trials and statistical measurements.

Deterministic Quicksort Implementation

The deterministic implementation follows the **Lomuto partition scheme** described in Chapter 7 of *Introduction to Algorithms* (Cormen et al., 2022).

In this version:

- The **last element** of the subarray is chosen as the pivot.
- The placing of the element that are either same as or less than the pivot will be at left.
- Elements greater than the pivot are placed to the right.
- The sorting of the two partitions will be done by the algorithm.

The algorithm operates **in-place**, meaning it does not require an additional array proportional to input size. The partition step runs in **Θ(n)** time, and the recursive structure reduces the problem until the entire array is sorted.

However, this version is highly sensitive to pivot selection. If the pivot consistently produces unbalanced partitions, performance degrades significantly.

Theoretical Time Complexity Analysis

Best Case

When partition of the array is done into two halves (that are almost the) at every step by the pivot, that's when the best case happens. This is when the recurrence relation will become:

$T(n) = 2T(n/2) + \Theta(n)$

Using the Master Theorem, this solves to:

$T(n) = \Theta(n \log n)$

In this case, recursion depth is proportional to log n, resulting in efficient performance (Cormen et al., 2022).

Average Case

Here, there will be a fair balance throughout all the recursive calls. While they may not be perfectly equal, the expected height of the recursion tree remains proportional to log n.

Cormen et al. (2022) show that the expected running time of Quicksort is:

$E[T(n)] = \Theta(n \log n)$

This explains why Quicksort performs well in practice even though its worst case is quadratic.

Worst Case

This case will occur when there is no balance between the partitions done by the pivot and This happens when:

- The sorting of the array is already done.
- The sorting of the array is done in reverse order.
- Either the smallest or most large element.

Thus the recurrence will become:

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

Solving this gives:

$T(n) = \Theta(n^2)$

In this case, recursion depth becomes n, and performance degrades significantly.

Space Complexity

Quicksort is considered in-place because it does not allocate additional arrays proportional to input size. However, recursion requires stack space.

- Best and average case recursion depth: $O(\log n)$
- The recursion depth of the worst case is: $O(n)$

Therefore:

- O (log n) will be the average of the space complexity
- O(n) will be the worst possible case space complexity.

Randomized Quicksort

To improve robustness, a randomized version was implemented. Instead of always selecting the last element as pivot, this version randomly selects an index between *low* and *high*, swaps it with the last element, and then performs partitioning.

According to Cormen et al. (2022), the expected running time of Randomized Quicksort is:

$\Theta(n \log n)$

The worst-case complexity remains $O(n^2)$, but the probability of consistently encountering worst-case partitions becomes extremely small.

Randomization prevents structured or adversarial input from forcing poor pivot choices repeatedly.

Empirical Performance Analysis

To compare both implementations, experiments were conducted using three input distributions:

- When the input is random: Random input
- When the input is sorted: Sorted input

- When the input is sorted in a reverse order: Reverse-sorted input

The sizes of the arrays that were tested were starting from 1000 elements then 500 elements then 10000 elements.

Experimental Methodology

To reduce measurement noise and improve reliability, **each experiment was executed 10 times** for every input size and distribution.

For each configuration:

- The recording of the average of the running time was done.
- The standard deviation was computed.
- Results were saved to a CSV file for reproducibility.

Using multiple trials provides more stable and statistically meaningful measurements than a single execution.

Random Input Results

For random input, both deterministic and randomized Quicksort showed similar performance. Execution time increased approximately proportional to n log n.

The deterministic version was sometimes slightly faster due to the absence of random number generation overhead. However, the difference was minimal.

This confirms that under typical input conditions, both implementations perform efficiently.

Sorted Input Results

For sorted input, the deterministic version showed dramatic performance degradation.

For example, at 10,000 elements:

- Deterministic: ~3+ seconds
- Randomized: ~0.01 seconds

This reflects the theoretical worst-case $O(n^2)$ behavior.

The randomized version maintained near $O(n \log n)$ performance because pivot selection was independent of input structure.

Reverse-Sorted Input Results

Reverse-sorted input produced similar behavior. The deterministic version again demonstrated quadratic growth, while the randomized implementation remained stable.

The small standard deviation values across trials indicate consistent performance for the randomized approach.

The expectations that are set theoretically are supported by the empirical results that we got:

- When given random data, the deterministic quicksort will perform well on it.
- It performs poorly on structured input due to poor pivot selection.

- The chances of any worst possible case to happen will lessen when randomized quicksort is used.
- The observed runtime growth aligns with the recurrence relations described in CLRS.

This experiment highlights the importance of pivot strategy. While deterministic Quicksort may be acceptable in many cases, randomized pivot selection provides stronger performance guarantees in practice.

This assignment provided a comprehensive study of Quicksort, including implementation, theoretical analysis, and empirical evaluation.

The deterministic implementation confirmed the theoretical complexities described in CLRS (2022). Empirical results clearly demonstrated worst-case quadratic growth on sorted and reverse-sorted inputs.

The randomized version improved robustness by preventing consistent unbalanced partitions and maintained expected O(n log n) performance across all tested distributions.

Quicksort remains one of the most efficient and practical comparison-based sorting algorithms due to its simplicity, speed, and adaptability.

References

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). *Introduction to algorithms* (4th ed.). MIT Press.