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: https://github.com/ananaware/QuicksortAlgorithm_MSCS532_Assignment5

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:

- Whatever, the most last element is in the subarray that will be selected as the pivot point.

- The placing of the element that is either the same as or less than the pivot will be at the left.
- Any elements that are actually larger than then the pivot then its positioning will be done in 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 almost same halves/partitions at every step by the pivot, 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.

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

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

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.

- 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.

Time Complexity Analysis

Quicksort is a divide-and-conquer sorting algorithm. Its time complexity depends on how balanced the partitioning is at each recursive step.

Best Case: O(n log n)

In this case:

- All the n elements will be processed by each level of recursion.
- log n will be the depth of the recursion

Therefore, the total work is:

$T(n) = n + n + n + ...$ (log n times)

This results in:

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

Average Case: O(n log n)

According to CLRS (Cormen et al., 2022), the expected number of comparisons made by Quicksort is proportional to n log n. Although some partitions may be uneven, random distribution ensures that extremely unbalanced splits are unlikely across all levels.

Thus, the expected running time of Quicksort is:

$O(n \log n)$

This explains why Quicksort performs efficiently in most practical scenarios.

Worst Case: $O(n^2)$

This happened when either the biggest or the smallest element is selected as a pivot in the array.

- The array is already sorted
- The array is reverse sorted
- A poor deterministic pivot rule is used

In this situation:

- $n-1$ elements are in one partition and 0 elements in other partition.

$T(n) = T(n-1) + O(n)$

Solving this recurrence gives:

$T(n) = O(n^2)$

During empirical testing, deterministic Quicksort on sorted input caused extremely deep recursion, which reflects this theoretical worst-case behavior.

Space Complexity (In palce sorting algorithm)

No requirement of additional arrays proportional to n.

It uses recursion, which consumes stack space.

- O(log n) is the recursion depth either in the best or the case
- O(n) is the recursion depth in the worst case.

Thus:

- O(log n) and O(n) are the s the average and worst-case space complexity respectively.

This behavior was observed during benchmarking, where deterministic Quicksort required increasing Python's recursion limit for large sorted inputs due to deep recursive calls.

Randomized Quicksort Analysis

Impact of Randomization

Randomized Quicksort selects the pivot randomly from the subarray. This reduces the probability of consistently poor partitions.

Even if the input is sorted, random pivot selection makes it unlikely that the smallest or largest element is chosen repeatedly.

- The expected running time remains $O(n \log n)$
- The probability of worst-case $O(n^2)$ becomes extremely small

Empirical results confirmed that randomized Quicksort maintained stable performance across random, sorted, and reverse-sorted inputs.

Connecting Empirical Results to Theory

Empirical Results and Theoretical Alignment

Each experiment was executed 10 times for every input size and distribution. The average execution time and standard deviation were computed.

- Balanced partitions → $O(n \log n)$
- Highly unbalanced partitions → $O(n^2)$

Thus, the empirical findings strongly support the theoretical complexity analysis presented in CLRS (Cormen et al., 2022)

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

A very close performance was showcased by randomized and deterministic Quicksort. Execution time increased approximately proportional to n log n. 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

Similar behavior was seen here as well . 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.