Assignment 5: Quick Sort Algorithm: Implementation, Analysis, and Randomization

By: Shabnam Shaikh

Course: MSCS-532-M80 Algorithms and Data Structures

Instructor: Dr. Michael Solomon

University of The Cumberlands

Introduction

Quick Sort is one of the most efficient and widely used comparison-based sorting algorithms. It was developed by Tony Hoare in 1959 and formally published in 1962 (Hoare, 1962). The algorithm applies to the divide and conquer paradigm, recursively partitioning an array around a pivot element until all subarrays are sorted. Due to its strong average case performance and in place memory usage, Quick sort is widely implemented in standard programming libraries and performance critical systems (Cormen et al., 2022). Empirical studies and algorithm engineering research have shown that Quick sort often outperforms other O(n long n) sorting algorithms such as Merge sort because of smaller constant factors and better cache locality (Sedgewick & Wayne, 2011).

The objective of this assignment was to implement deterministic and randomized versions of Quick sort in Python, analyze their theoretical complexity, and empirically compare their behavior on multiple input distributions. Additionally, tail recursion elimination was incorporated to prevent recursion depth overflow, demonstrating practical considerations beyond theoretical analysis.

Deterministic Quick Sort Implementation

The deterministic implementation selects the last element of the subarray as the pivot. During partitioning, elements less than or equal to the pivot are placed to its left, and larger elements are placed to its right. Once partitioned, the pivot is guaranteed to be in its final sorted position. The algorithm then recursively sorts the two resulting subarrays. The correctness of this approach follows directly from the divide and conquer recurrence structured described in classical algorithm literature (Cormen et al., 2022).

However, fixed pivot selection introduces a weakness. If the input array is already sorted or reverse sorted, the partitioning process becomes maximally unbalanced. In such cases, one subarray contains n – 1 element while the other contains zero, leading to degenerate recursion. This behavior has been extensively analyzed in foundational algorithm texts (Cormen et al., 2022; Sedgewick & Wayne, 2011).

To address Python's recursion limit, tail recursion elimination was implemented. Instead of recursively sorting both arrays, the algorithm recurses only on the smaller subarray and iteratively processes the larger one. This transformation reduces maximum recursion depth from O(n) to O(log n), aligning practical behavior with theoretical expectations for balanced recursion (Sedgewick & Wayne, 2011).

Theoretical Time Complexity Analysis

The best case occurs when each pivot divides the array into two equal halves. The recurrence relation becomes: $T(n) = 2T(n/2) + O(n)$. Applying the Master Theorem yields: $T(n) = O(n \log n)$. This result is formally derived in standard algorithm analysis texts (Cormen et al., 2022). At each level of recursion, partitioning requires linear time, and the recursion tree has height log n.

In practice, partitions are rarely perfectly balanced but are reasonably distributed across recursive calls. The expected running time of Quick Sort can be expressed as: $T(n) = T(k) + T(n – k – 1) + O(n)$. Averaging over all possible pivot positions produces an expected time complexity of $O(n \log n)$ (Cormen et al., 2022). The expected number of comparisons in randomized Quick sort is approximately $2n \ln n$, confirming logarithmic depth behavior (Sedgewick & Wayne, 2011). This explains why Quick Sort performs efficiently on average and is often preferred in real-world systems.

The worst case occurs when partitioning is maximally unbalanced. This happens when the pivot is consistently chosen as the smallest or largest element, which is common when the array is already sorted and the pivot is fixed at one end. The recurrence becomes: T(n) = T(n – 1) + O(n). Solving this recurrence leads to quadric time complexity: T(n) = O(n^2). This degradation is well documented in foundational analysis of Quick Sort (Hoare, 1962; Cormen et al., 2022). Although worst case performance is quadratic, its occurrence depends heavily on pivot selection strategy.

Space Complexity

Quick Sort is an in-place sorting algorithm because it requires only constant auxiliary memory beyond the recursion stack (Cormen et al., 2022). This primary additional space requirement arises from recursive calls.

Case	Recursion Depth	Space Complexity
Best	$\log n$	$O(\log n)$
Average	$\log n$	$O(\log n)$
Worst	n	$O(n)$

Without optimization, worst case recursion depth may reach n. However, with tail recursion elimination, recursion depth remains bounded by O(log n). Significantly improving practical space efficiency.

Randomized Quick Sort

Randomized Quick Sort selects the pivot uniformly at random from the subarray before partitioning. This modification ensures that the probability of consistently poor pivot selection is extremely low. As proven in classical algorithm analysis, the expected running time of randomized Quick Sort remains O(n log n) regardless of input distribution (Cormen et al., 2022).

Randomization prevents adversarial inputs from consistently triggering worst case behavior. Even if the input array is sorted, the expected partitioning remains balanced due to the random pivot selection (Sedgewick & Wayne, 2011). Consequently, randomized Quick Sort offers improved robustness while maintaining average case efficiency.

Empirical Analysis

To validate theoretical claims, both deterministic and random implementations were benchmarked on random, sorted, and reverse sorted arrays with input sizes of 1,000, 5,000, and 10,000 elements.

Sample Performance Results

Input Size	Distribution	Deterministic (s)	Randomized (s)
1000	Random	0.0039	0.0102
1000	Sorted	0.0201	0.0045
5000	Random	0.0250	0.0278
5000	Sorted	0.4200	0.0312
10000	Random	0.0602	0.0625
10000	Sorted	1.8200	0.0721

The results confirm theoretical predictions. For random inputs, both versions exhibit similar performance consistent with O(n log n). However, deterministic Quick Sort demonstrates significant slowdown on sorted data, approaching quadratic growth. Randomized Quick Sort maintains stable performance across distributions, validating theoretical expectations (Cormen et al., 2022).

Conclusion

This project demonstrates that while deterministic Quick Sort achieves excellent average case efficiency, it is vulnerable to structured input patterns that cause quadratic degradation.

Randomized Quick Sort mitigates this issue by reducing the probability of poor pivot selection, thereby maintaining expected O(n log n) performance (Cormen et al., 2022; Sedgewick & Wayne, 2011). Furthermore, incorporating tail recursion eliminating highlights the importance of adapting theoretical algorithms to practical programming constraints. Combining theoretical analysis with empirical validation provides a comprehensive understanding of algorithm behavior in real world environments.

References

1. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). *Introduction to Algorithms* (4th ed.). MIT Press.Knuth, D.
2. Hoare, C. A. R. (1962). Quicksort. *The Computer Journal, 5*(1), 10–16.
3. Sedgewick, R., & Wayne, K. (2011). *Algorithms* (4th ed.). Addison-Wesley.
.