Raghul Krishnan

University of Cumberlands

2025 Spring - Algorithms and Data Structures (MSCS-532-A01) - First Bi-term

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

**Report on Quicksort Algorithm Implementation and Analysis**

**Introduction**

One of the most effective sorting algorithms still in use today is the Quicksort algorithm, created by Tony Hoare in 1960. It is highly regarded for its ease of use and exceptional average-case performance (Edelkamp, 2022). The Quicksort algorithm's deterministic and randomised implementations are described in length in this article. In order to comprehend how various design decisions impact their robustness and efficiency, it also examines how well they operate under various circumstances (Shorman, 2024).

**Design Choices**

A 'pivot' element is chosen, and the other elements are divided into those that are more than and those that are less than the pivot. The partitions are then sorted recursively using the Quicksort method. Two important design elements that affect the algorithm's performance are the pivot selection and the partitioning technique (Edelkamp, 2022).

**Deterministic Quicksort**:

* **Pivot Selection**: The deterministic approach always selects the last element of the array as the pivot. This method is straightforward and eliminates the need for additional overhead to determine the pivot, making the implementation simpler and slightly faster in terms of constant factors.
* **Partitioning Scheme**: The standard Lomuto partition scheme was chosen for its ease of implementation. It involves a single scan through the array, swapping elements as necessary to ensure all elements less than the pivot are on the left, and all greater are on the right.

**Randomized Quicksort**:

* **Pivot Selection**: Instead of fixing the pivot, the randomized version selects a pivot randomly from the array segment to be sorted (Edelkamp, 2022). This strategy aims to minimize the probability of degrading to worst-case performance, which is crucial for data sets that might be predisposed to worst-case scenarios, like nearly sorted or reverse-ordered arrays (Shorman, 2024).
* **Partitioning Scheme**: The same partitioning method as deterministic Quicksort is used for consistency in comparing the two versions.

**Implementation Details**

The implementation was carried out in Python due to its widespread use and ease of understanding. Both versions of the algorithm were implemented with a focus on clarity and efficiency, using recursive methods.

**Code Structure**:

* **Base Case**: The recursion terminates when the 'low' index is greater than or equal to the 'high' index, which means the sub-array contains zero or one element and is therefore already sorted.
* **Recursive Case**: For arrays of size greater than one, the partition function is called to place the pivot into its correct position and return the index of the pivot. The function then recursively sorts the elements before and after the pivot (Shorman, 2024).

**Error Handling and Input Validation**: The code includes basic checks to ensure that the indices passed to the Quicksort functions are within the bounds of the array, thus avoiding runtime errors related to index out of range.

**Performance Analysis**

Performance analysis focused on both theoretical aspects and empirical testing. The theoretical analysis examined the time complexity in best, average, and worst-case scenarios, while the empirical analysis involved running the algorithms on different data sets (Rahman & Munro, 2023).

**Theoretical Analysis**:

* **Best and Average Case**: Both versions of Quicksort have an average-case complexity of O(nlog⁡n)O(n \log n)O(nlogn), achieved when the pivot divides the array into two nearly equal parts. The best case also falls into this complexity.
* **Worst Case**: The deterministic version can degrade to O(n2)O(n^2)O(n2) when the pivot is consistently the smallest or largest element in the array, which happens with sorted or reverse-sorted data. The randomized version, however, rarely encounters this degradation due to the randomness in pivot selection.

**Empirical Analysis**:

* **Setup**: The algorithms were tested on arrays of varying sizes and types (random, sorted, reverse-sorted).
* **Metrics**: Execution times were recorded for each run, providing a direct comparison of performance under different conditions (Rahman & Munro, 2023).
* **Results**: The deterministic version performed well on random arrays but showed significant slowdowns on sorted and reverse-sorted arrays. In contrast, the randomized version maintained more consistent times across all types of inputs, substantiating its effectiveness in avoiding worst-case scenarios.

**Conclusion**

The study and empirical findings show that the deterministic Quicksort might perform noticeably worse for sorted datasets, even when it is effective for randomly ordered data (Rahman & Munro, 2023). For cases where input patterns are unpredictable, the randomised variant offers a more reliable solution by reducing the effect of input order. These insights are crucial for selecting the appropriate sorting strategy in both academic study and real-world applications, highlighting the trade-offs between simplicity and performance robustness.

In addition to highlighting Quicksort's value in contemporary computer tasks—from real-time systems to data analysis frameworks—this research offers a thorough reference for future development and modification of this classic algorithm (Edelkamp, 2022).

**References**

Edelkamp, S., & Weiss, A. (2022). Recent Advances in Quicksort. ACM Computing Surveys, 54(7), Article 157.

Rahman, S., & Munro, J. I. (2023). Enhanced Quicksort Techniques for Large Data Sets. Journal of Computer Science and Technology, 38(2), 349-361.

Shorman, D. (2024). Performance Analysis of Quicksort Algorithm: An Experimental Study of Its Variants. International Journal of Advanced Science and Computer Applications, 5(1).