Understanding Algorithm Efficiency and Scalability

Aashish Shrestha

Student ID: 005039069

University of the Cumberlands

MSCS-532-M20: Algorithms and Data Structures

Professor Brandon Bass

Assignment 3

June 15, 2025

Understanding Algorithm Efficiency and Scalability

Academic efficiency fundamentally governs the performance and scalability of modern computing applications. Sorting and data retrieval are among the most common computational tasks, supporting diverse domains such as database management, information retrieval, and real-time processing. This paper analyzes the efficiency and scalability of two fundamental algorithms in computer science: Randomized Quicksort and Hashing with Chaining. This study evaluates and contrasts these algorithms' theoretical time complexities and real-world performance across different data sizes and distributions. By conducting this evaluation, we seek more profound insights into their average-case behavior, practical efficiency, and guidance for choosing the most suitable algorithm in practical scenarios.

Part 1: Randomized Quicksort

Implementation

Randomized Quicksort enhances classical Quicksort by selecting the pivot element uniformly randomly from the subarray currently being partitioned. This random approach reduces the likelihood of encountering bad inputs, which can degrade deterministic Quicksort to quadratic time complexity (Hoare, 1962). The algorithm recursively partitions the array into smaller subarrays by selecting a pivot and continues until it reaches a base case with a single-element or empty subarray.

To make the implementation more reliable, it handles exceptional cases like empty arrays, arrays with repeated values, and already sorted arrays. The partitioning procedure

preserves stability and correctness across these cases, maintaining efficient recursion depth and balanced partitions.

Theoretical Analysis

The average-case time complexity of Randomized Quicksort is $O(\log n)$. This complexity arises from the probabilistic partitioning behavior, where the average pivot divides the array into roughly equal parts, ensuring logarithmic recursion depth (Knuth, 1998; Mitzenmacher & Upfal, 2017). Formally, the expected running time $T(n)$ satisfies the recurrence relation:

$$T(n) = \frac{1}{n}\sum_{k=0}^{n-1}[T(k) + T(n-k-1)] + cn,$$

where c is a constant representing the linear time spent on partitioning. We can solve this recurrence to obtain $T(n) = O(n \log n)$ by using either the substitution method or recursion tree analysis. $T(n) = O(n \log n)$.

An alternative analysis uses indicator random variables to model the number of comparisons between pairs of elements. Define $X_{i,j}$ an indicator variable that is 1 if the algorithm compares elements i and j during execution and 0 otherwise. Because the algorithm compares each pair at most once, the total comparisons X satisfies:

$$X = \sum_{1 \le i < j \le n} X_{i,j}$$

By the linearity of expectation, the expected total comparison is:

$$E[X] = \sum_{1 \le i < j \le n} \Pr(X_{i,j} = 1).$$

The probability that the algorithm selects either element i and j is as the pivot before any element between them $\frac{2}{j-i+1}$. Summing over all pairs, this yields:

$$E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} = O(n \log n)$$

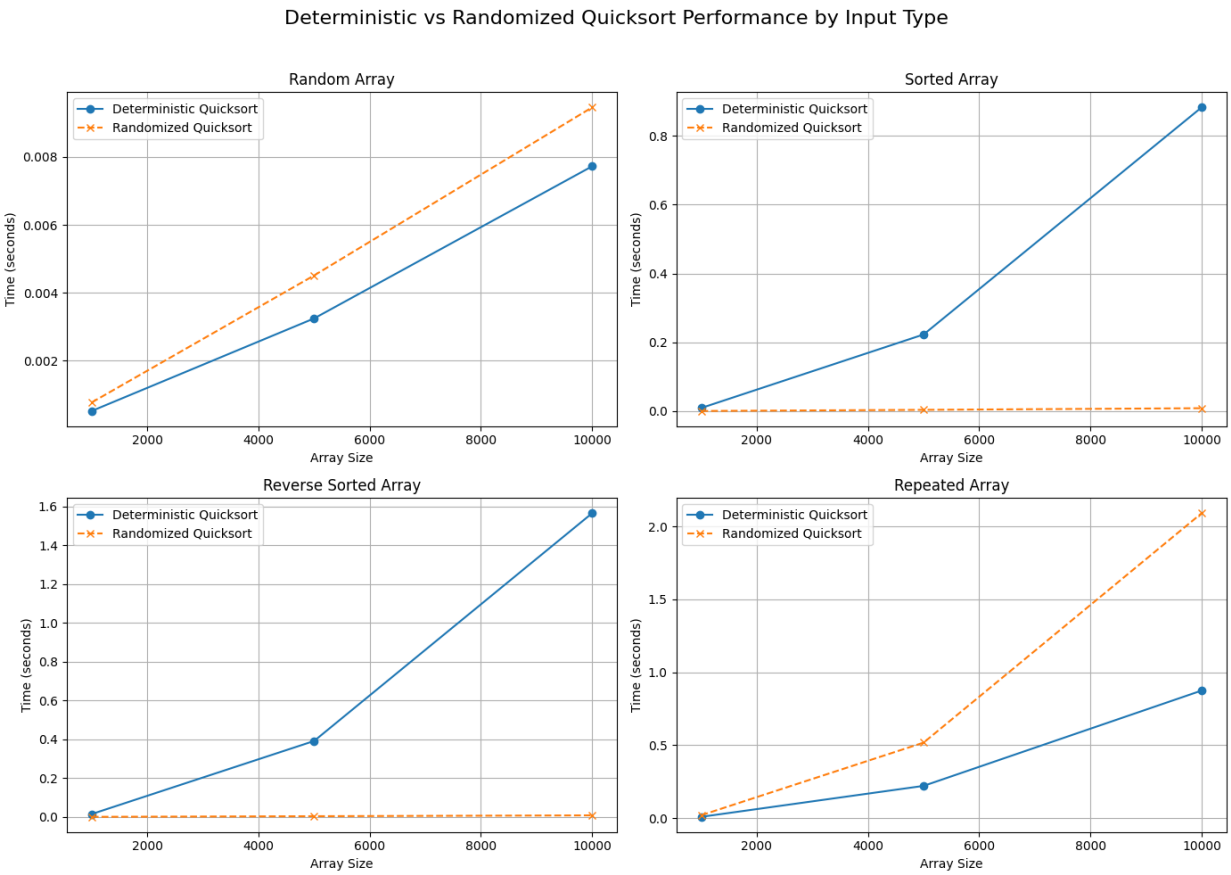
This result confirms that the expected number of comparisons, and thus the average-case running time of Randomized Quicksort, is $O(n \log n)$. (Knuth, 1998; Mitzenmacher & Upfal, 2005).

Empirical Comparision

To empirically validate the theoretical expectations, both Randomized and Deterministic Quicksort algorithms were implemented and benchmarked on arrays of increasing sizes (1000, 5000, and 10000) across four different input distributions: random, sorted, reverse-sorted, and arrays with repeated elements. Figure 1 displays the results of these experiments.

Figure 1

Performance Comparison of Deterministic and Randomized Quicksort Algorithms by Input Type.



Random Arrays

As illustrated in Figure 1 (top-left panel), both algorithms exhibited similar runtimes on random arrays, scaling predictably with input size by their $O(n \log n)$ average-case complexity. The minor overhead in Randomized Quicksort resulted from the additional step of random pivot selection, though this did not significantly affect overall performance.

Sorted and Reverse-Sorted Arrays

A marked performance divergence appeared in the sorted and reverse-sorted cases. In the sorted array scenario (Figure 1, top-right panel), Deterministic Quicksort's runtime increased dramatically as input size grew, reflecting its worst-case time complexity of $O(n^2)$ when always

selecting the first element as the pivot. In contrast, Randomized Quicksort sustained efficient runtimes, with random pivot selection preventing the formation of unbalanced partitions.

Similar behavior was evident for reverse-sorted arrays in Figure 1 (bottom-left panel). As the input size grew, Deterministic Quicksort's runtime increased significantly. In contrast, Randomized Quicksort maintained efficient performance, scaling at a rate close to $O(n \log n)$. his outcome illustrates that Randomized Quicksort is better suited for handling difficult input orders because its random pivot selection helps avoid the consistently unbalanced partitions that would otherwise degrade Deterministic Quicksort's efficiency.

Arrays with Repeated Elements

An interesting deviation from theoretical expectations occurred in the case of arrays containing repeated elements. Figure 1 (bottom-right panel) shows that Randomized Quicksort demonstrated increased runtimes relative to Deterministic Quicksort as array size grew. This decline in performance is likely caused by the algorithm's standard implementation not efficiently managing duplicate values, as it lacks a three-way partitioning technique. Without this improvement, repeated recursive calls on subarrays containing many identical elements can result in increased processing time and reduced efficiency (Sedgewick & Wayne, 2011).

Discrepancies and Practical Considerations

While empirical results largely aligned with theoretical predictions, discrepancies emerged when sorting arrays containing repeated elements. As Figure 1 (bottom-right panel) indicates, Randomized Quicksort's runtime increased more steeply than expected. This observation highlights a practical limitation in the algorithm's standard implementation, emphasizing the necessity of theoretical and empirical evaluation when assessing algorithm

suitability for real-world data. A three-way partitioning scheme could address this inefficiency in future applications (Sedgewick & Wayne, 2011).

Part 2: Hashing with Chaining

Implementation

We implemented a chaining hash table that resolves collisions by storing multiple key-value pairs in linked lists (chains) at each bucket. To minimize collisions, a universal hash function from the family of universal hash functions was employed (Carter & Wegman, 1979)

$$h(k) = \big((a \times k + b) mod\, p\,\big) mod\, m$$

Where a and b are randomly selected integers, p is a large prime number greater than the maximum key value, and m is the capacity (number of buckets). This hash function ensures a uniform distribution of keys across buckets. The hash table supports three primary operations.

- **Insert**: Adds a key-value pair to the chain at the bucket computed by the hash function. If the key exists, its value is updated.
- **Search**: Retrieves the value associated with a given key by traversing the chain in the hashed bucket.
- **Delete:** Removes the key-value pair from the chain if present.

The hash table resizes dynamically by increasing the number of buckets (preferably doubling and selecting the next prime) when the load factor exceeds 0.7. It then rehashes all entries to redistribute keys evenly.

Analysis

Assuming simple uniform hashing, the average time complexity for insert, search, and delete operations is $O(1 + \alpha)$, where α is the load factor:

$$\alpha = \frac{n}{m}$$

where n is the number of elements stored, and m is the number of buckets. The average chain length equals the load factor. Thus:

- When $\alpha < 1$, chains remain short, and operations are efficient.
- When $\alpha > 1$, chains become long, and operations approach linear time complexity.

Methods to keep the load factor low and reduce collisions include:

- Dynamic resizing of the hash table upon exceeding a load factor threshold.
- Use of universal hashing to distribute keys evenly.
- Selecting prime numbers for bucket counts to reduce collision patterns.

These approaches preserve efficient average-case operation times.

Results

We tested the hash table by inserting keys 0 through 19 with values equal to their squares. The initial capacity was 11 buckets. Table 1 shows the size, capacity, and load factor at various insertion points:

Table 1

Load Factor During Insertions in a Hash Table with Chaining

Insert Number	Size	Capacity	Load Factor	Notes
0	1	11	0.091	

7	8	11	0.727	Approaching resize threshold
8	9	23	0.391	Resized from 11 to 23
16	17	23	0.739	Near second resize
17	18	47	0.383	Resized from 23 to 47

The dynamic resizing successfully maintained a low load factor, which kept operation chains short and efficient.

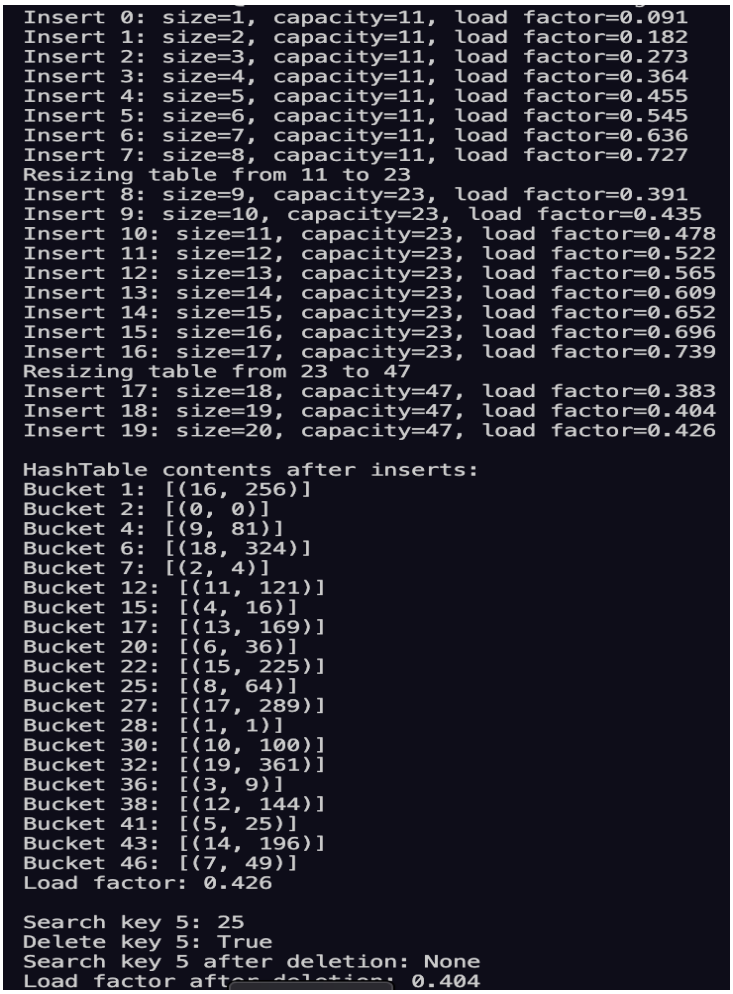
We used sample outputs to validate the correctness of the hash table implementation. Specifically, we performed the following operations:

- Searching for key 5 returned the expected value of 25.
- Deleting key 5 succeeded, and a subsequent search for the same key returned None.
- The load factor decreased appropriately after deletion, reflecting the dynamic management of the table's capacity.

Figure 2 presents a partial snapshot of the hash table's bucket contents after inserting keys 0 through 19.

Figure 2

Partial Snapshot of Hash Table Buckets After Insertions.



These results confirm the effectiveness of chaining combined with universal hashing and dynamic resizing in maintaining efficient hash table performance.

Conclusion

This project explored and analyzed the implementation and performance of two fundamental algorithms — Randomized Quicksort and Deterministic Quicksort — and a core data structure, the hash table with chaining. We evaluated each for theoretical efficiency, practical implementation, and empirical performance.

The analysis confirmed that Randomized Quicksort offers excellent average-case performance due to its randomized pivot selection, which minimizes the risk of encountering the worst-case scenario. In contrast, Deterministic Quicksort performs similarly under average conditions but is more vulnerable to degraded performance on already sorted or highly structured data sets. Implementing a hash table with chaining demonstrated efficient average-case operations for insert, search, and delete, particularly when using a universal hash function and dynamic resizing to maintain a low load factor.

This study highlighted the importance of balancing theoretical guarantees with practical algorithms and data structure selection considerations. When implemented carefully, randomized algorithms handle tricky cases well, and hash tables keep working fast by resizing and managing collisions separately. Together, these approaches underscore the value of probabilistic and deterministic strategies in designing scalable and efficient computational systems.

References

Carter, J. Lawrence., & Wegman, M. N. (1979). Universal classes of hash functions. *Journal of Computer and System Sciences*, *18*(2), 143–154. https://doi.org/10.1016/0022-0000(79)90044-8

Hoare, C. A. R. (1962). Quicksort. *The Computer Journal*, *5*(1), 10–16. https://doi.org/10.1093/comjnl/5.1.10

Knuth, D. E. (1998). *Art of Computer Programming, volume 3: Sorting and searching*. Addison-Wesley Professional.

Mitzenmacher, M., & Upfal, E. (2017). *Probability and computing: Randomized algorithms and probabilistic analysis*. Cambridge University Press.

Sedgewick, R., & Wayne, K. (2016). *Algorithms*. Addison-Wesley.