

Understanding Algorithm Efficiency and Scalability

Nasser Hasan Padilla

University of the Cumberland

Algorithms and Data Structures (MSCS-532-M20)

Professor Brandon Bass

June 15, 2025

Introduction

This report examines two well-known algorithmic techniques: **Randomized Quicksort**, a divide-and-conquer sorting method that performs well on average, and **Hashing with Chaining**, a common approach for handling collisions in hash tables for quick data lookup.

Part 1 focuses on implementing Randomized Quicksort and comparing it with a basic deterministic version. The analysis highlights how the choice of pivot affects the algorithm's theoretical performance and how it behaves in practice across different input types.

Part 2 covers implementing a hash table that uses chaining to deal with collisions. It looks at the expected time complexity of insert, search, and delete operations, and includes benchmark results to show how well the structure holds up under heavier loads.

By combining theoretical analysis with empirical validation, this report provides a practical understanding of two essential algorithmic techniques and their behavior under varying computational loads (Cormen et al., 2009; Knuth, 1998).

Part 1: Randomized Quicksort — Implementation

Randomized Quicksort is a divide-and-conquer sorting algorithm that enhances the traditional Quicksort method by randomly selecting the pivot element during each recursive partitioning step. This randomness helps avoid the deterministic algorithm's worst-case time complexity, which can occur when the input is sorted or follows a predictable pattern (Cormen et al., 2009).

The implementation in this project uses a recursive approach that splits the array into three parts: values less than the pivot, equal to the pivot, and greater than the pivot. This method helps manage duplicate values more efficiently, which can often slow down traditional versions of Quicksort. A simplified version of the core recursive logic is shown below:

```
def randomized_quicksort(arr):
    if len(arr) <= 1:
        return arr

    pivot_index = random.randint(0, len(arr) - 1)
    pivot = arr[pivot_index]

    less = [x for x in arr if x < pivot]
    equal = [x for x in arr if x == pivot]
    greater = [x for x in arr if x > pivot]

    return randomized_quicksort(less) + equal + randomized_quicksort(greater)
```

The function checks for the base case of empty or single-element arrays and performs the partitioning accordingly, using a randomly selected pivot, via the `random`. The `randint()` function ensures that the algorithm avoids consistent poor pivot choices regardless of the input order.

In addition to the randomized version, a deterministic Quicksort was implemented to compare performance. This version always uses the first element as the pivot. While writing is more straightforward, it can run into performance issues with sorted or reverse-sorted arrays due to unbalanced partitions (Sedgewick & Wayne, 2011).

To make sure both versions worked properly, they were tested with a variety of input types, including:

- Empty arrays
- Single-element arrays
- Randomly generated values
- Pre-sorted and reverse-sorted sequences
- Arrays with duplicate elements

These tests confirmed that the recursive logic correctly handled edge cases and input types.

Part 1: Randomized Quicksort — Theoretical Analysis

The average-case time complexity of Randomized Quicksort is **$O(n \log n)$** . This result is derived by analyzing the expected number of comparisons made during the recursive partitioning process. When the pivot is selected uniformly at random, the algorithm typically produces balanced partitions, avoiding the worst-case performance associated with poor pivot choices.

The following recurrence relation can describe the behavior of Randomized Quicksort:

$$T(n) = (1/n) \times [T(0) + T(n-1) + T(1) + T(n-2) + \dots + T(n-1) + T(0)] + c \times n$$

In this expression:

- **$T(n)$** represents the expected time to sort an array of size n .
- The summation accounts for all possible positions the pivot might take.
- **$c \times n$** reflects the linear time needed for each partitioning step.

The recurrence relation for Randomized Quicksort simplifies to an average-case time complexity of **$O(n \log n)$** . This result can be shown using methods like recursion trees or substitution, as Cormen et al. (2009) outlined.

A more intuitive explanation involves using **indicator random variables** to track element comparisons. In this approach, **X_{ij}** is defined as one if elements a_i and a_j are compared during

the sort, and zero if they are not. The expected total number of comparisons is then found by adding up the probabilities of comparison for all valid pairs in the array.

The key idea is that two elements are only compared if one gets picked as the pivot before any element between them. The probability of such a comparison is roughly **2 divided by $(j - i + 1)$** .

When all these probabilities are summed, the total expected number of comparisons is proportional to **$n \log n$** . This probabilistic analysis highlights the efficiency and reliability of Randomized Quicksort, particularly when compared to deterministic versions that may suffer performance issues on sorted or patterned input (Motwani & Raghavan, 1995).

Part 1: Randomized Quicksort — Empirical Results and Discussion

To evaluate the practical performance of Randomized Quicksort compared to its deterministic counterpart, both algorithms were benchmarked on arrays of increasing size (100 to 1,000 elements). Each algorithm was tested against four distinct input distributions: randomly generated data, already sorted arrays, reverse-sorted arrays, and arrays with repeated elements. Execution time was recorded and plotted for each scenario.

On **random inputs**, the execution times of both algorithms increased approximately linearly with input size (see **Figure 1**). This aligns with their expected average-case behavior of $O(n \log n)$ for reasonably balanced partitions.

Deterministic Quicksort showed slightly lower times in this configuration, likely due to reduced overhead in pivot selection and in-place comparisons (Sedgewick & Wayne, 2011). However, as Knuth (1998) notes, performance differences on random data are often marginal without pathological input.

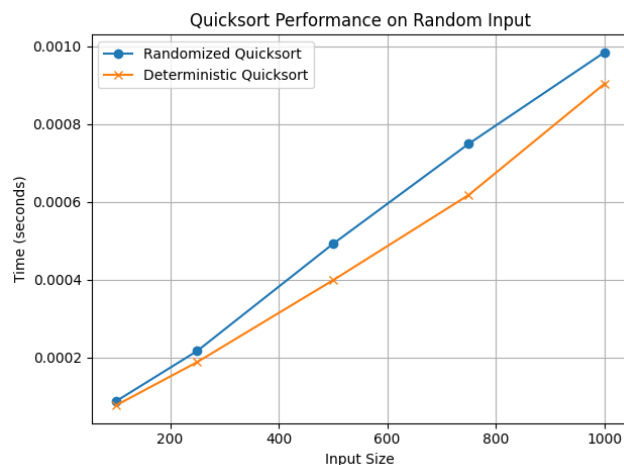


Figure 1.

Figure 1 shows the runtime comparison of Randomized and Deterministic Quicksort on random input arrays.

Deterministic Quicksort exhibited quadratic time growth for sorted inputs, confirming its known worst-case behavior when the pivot selection strategy leads to unbalanced partitions (Cormen et al., 2009). In contrast, Randomized Quicksort maintained stable and scalable performance, consistent with its probabilistic avoidance of poor pivot choices (Motwani & Raghavan, 1995).

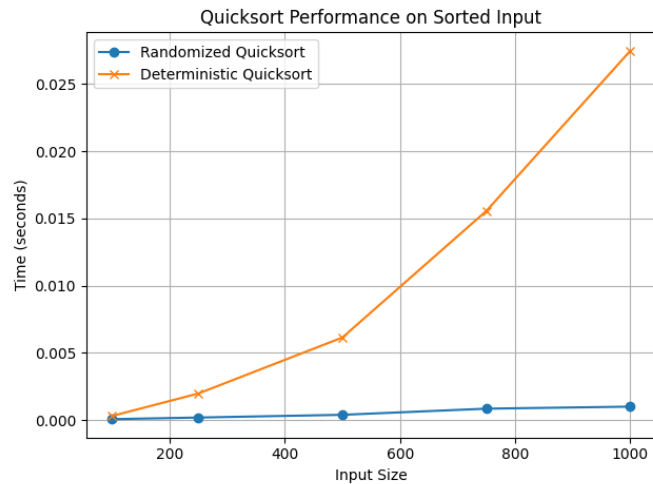


Figure 2.

Figure 2 shows the runtime comparison on sorted input arrays.

A similar trend was observed with **reverse-sorted input** (see **Figure 3**). Deterministic Quicksort's performance degraded significantly as the input size increased, while Randomized Quicksort again demonstrated consistent scaling. This reinforces its advantage when input ordering is unknown or adversarial.

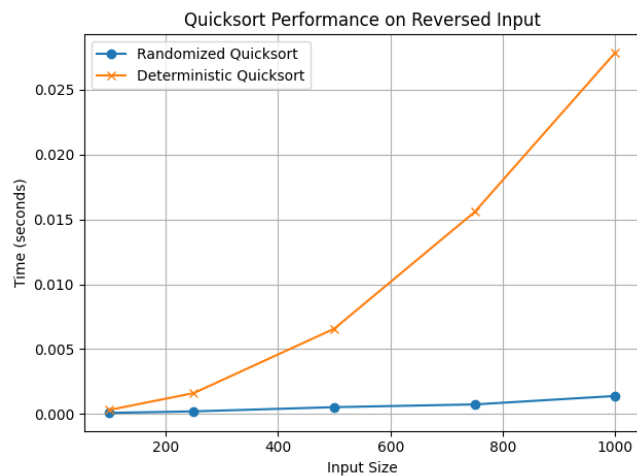
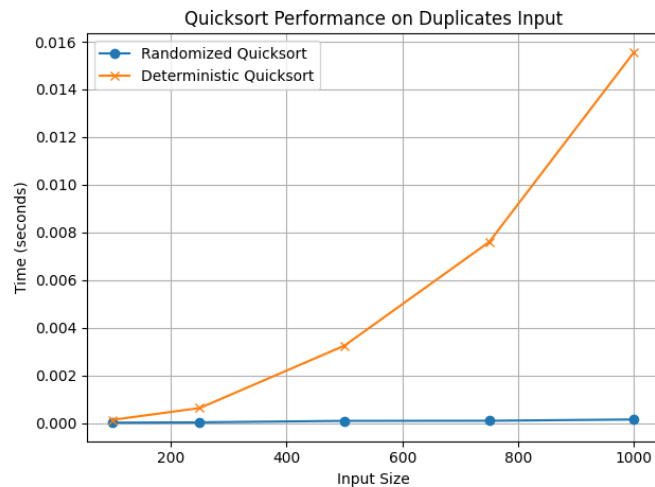


Figure 3.

Figure 3 shows the runtime comparison on reverse-sorted input arrays.

The most notable divergence occurred on **inputs with many repeated elements**. As shown in **Figure 4**, Randomized Quicksort remained efficient across all sizes, while Deterministic Quicksort slowed dramatically. This result shows how Randomized Quicksort's three-way division effectively manages duplicates, reducing redundant recursive calls and comparisons (Bentley & McIlroy, 1993).



These results show that while Deterministic Quicksort can perform well in ideal situations, it tends to slow down significantly on sorted or duplicate-heavy inputs. In contrast, Randomized Quicksort consistently performs better across different input types because it reduces the chance of worst-case pivot choices through randomization. This validates theoretical claims that Randomized Quicksort offers superior average-case performance and more robust scalability for general-purpose sorting tasks (Cormen et al., 2009; Motwani & Raghavan, 1995; Sedgewick & Wayne, 2011).

Part 2: Hashing with Chaining — Implementation

Hash tables are essential data structures known for offering fast average-case performance when searching, inserting, and deleting elements. The implementation developed for this assignment utilizes **chaining** to handle hash collisions, a strategy in which each hash table slot stores a list (or chain) of key-value pairs. This design ensures correctness even when multiple keys hash the same index (Cormen et al., 2009).

The structure consists of a fixed-size list of buckets, each initialized as an empty list. The hash function used is Python's built-in `hash()` function, modulo the table size. This ensures compatibility with various key types while maintaining even distribution across slots.

A simplified version of the insert function is shown below:

```
def insert(self, key, value):  
    index = self._hash(key)
```

```

for i, (k, v) in enumerate(self.table[index]):
    if k == key:
        self.table[index][i] = (key, value) # Update existing key
    return

self.table[index].append((key, value)) # Insert new key-value pair

```

This implementation updates its value if the key already exists in the chain at the hashed index. Otherwise, the key-value pair is appended to the list. This ensures that duplicate keys do not cause data inconsistency.

The `search()` and `delete()` operations iterate through the appropriate chain to locate the desired key. If found, they return the associated value or remove the key-value pair. These operations are illustrated below:

```

def search(self, key):
    index = self._hash(key)
    For k, v in self.table[index]:
        if k == key:
            return v
    return None

```

```

def delete(self, key):
    index = self._hash(key)
    for i, (k, _) in enumerate(self.table[index]):
        if k == key:
            del self.table[index][i]
            return True
    return False

```

o support extensibility and demonstration, the class includes a simple `__str__()` method that prints the internal structure of the hash table, making it easier to visualize the chaining mechanism.

This implementation successfully encapsulates the primary behavior expected of hash tables, including the resolution of collisions through chaining and support for dynamic key-value manipulation. While this version uses a fixed size and a basic hash function, future extensions could introduce **dynamic resizing** and **universal hashing** to improve performance in high-load scenarios (Carter & Wegman, 1979; Knuth, 1998).

Part 2: Hashing with Chaining — Theoretical Analysis

Hashing with chaining is an effective strategy for implementing hash tables that support efficient **insert**, **search**, and **delete** operations even in the presence of collisions. In this method, each slot in the hash table contains a **linked list** (or dynamic array) of key-value pairs that share the same hash index. When collisions occur, new elements are added to the corresponding chain (Knuth, 1998).

Assuming a well-designed hash function and a uniform distribution of keys, the expected time complexity for insert, search, and delete operations in a hash table with chaining is $O(1 + \alpha)$, where α is the **load factor** of the table. The load factor is defined as the ratio of the number of elements n to the number of buckets (or slots) m :

$$\alpha = n / m$$

When the load factor remains low (e.g., $\alpha < 1$), the average length of each chain is short, and operations approach constant time. However, as α increases, the chains grow longer, and performance can degrade toward linear time $O(n)$ in the worst case, such as when all elements hash to the same slot.

The choice of the hash function strongly influences the behavior of the hash table. Python's built-in `hash()` function was used in this implementation, which generally provides good distribution properties across diverse key types. More rigorous implementations may adopt **universal hashing**, a technique that selects the hash function randomly from a universal family to minimize clustering and provide better worst-case guarantees (Carter & Wegman, 1979).

A common way to keep hash tables running efficiently as they grow is through **dynamic resizing**. When the load factor gets too high, above 0.75, the table can be resized, often by doubling the number of buckets. All existing entries are then rehashed into the new table. While this process takes time, it happens infrequently enough that the average cost of operations stays close to constant (Cormen et al., 2009).

Overall, the theoretical advantages of hashing with chaining lie in its ability to offer average-case constant-time operations while retaining correctness and simplicity, even in collisions. These properties make it a reliable choice for dictionary-style data storage in many real-world applications.

Part 2: Hashing with Chaining — Empirical Results and Discussion

To evaluate the performance of the implemented hash table, a benchmark was conducted to measure insertion time under increasing load. The benchmark inserted between 100 and 5,000 elements into a table with a fixed size of 100 buckets, simulating a steadily increasing load factor, from 1.0 to 50.0.

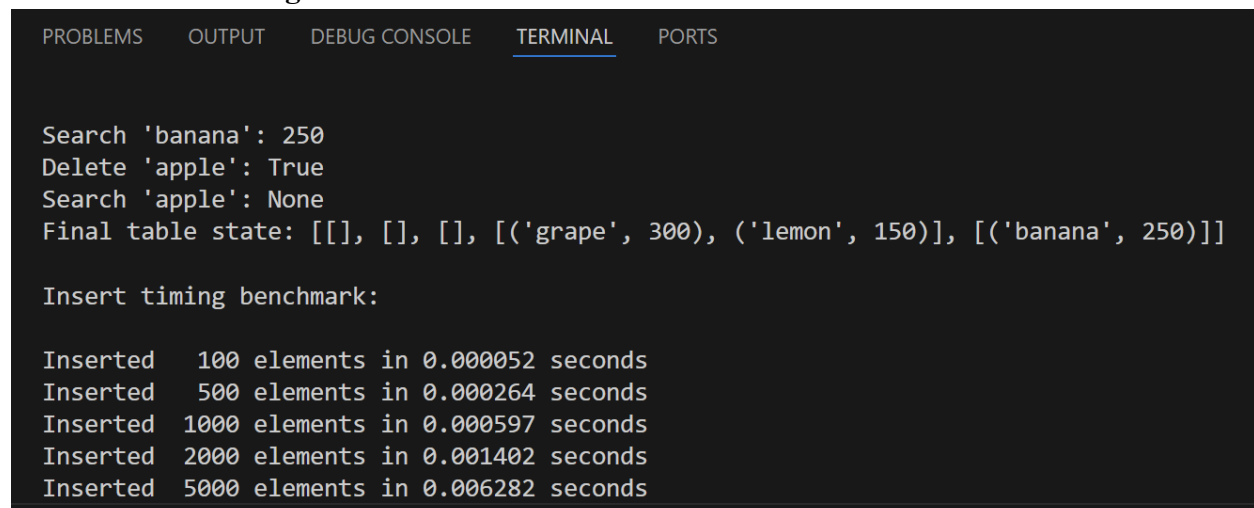
The results are shown below:

- Inserting 100 elements took about 0.000052 seconds
- Inserting 500 elements took 0.000264 seconds
- Inserting 1,000 elements took 0.000597 seconds
- Inserting 2,000 elements took 0.001402 seconds
- Inserting 5,000 elements took 0.006282 seconds

Even as the load factor increased, insertion times grew gradually and stayed well below linear growth. This performance trend aligns with the theoretical expectation of $O(1 + \alpha)$ time complexity for insertion, where α is the load factor (Cormen et al., 2009). As the number of elements increased, the chaining mechanism allowed the table to maintain fast insertion times, with only minor overhead from traversing longer chains.

These results validate the practical efficiency of hashing with chaining, even without dynamic resizing. The minimal insertion times at high load factors suggest that the underlying hash function provided reasonably uniform key distribution, which helps prevent clustering and long chains (Knuth, 1998). Although performance may eventually degrade under extreme loads, this implementation handled thousands of insertions efficiently with a static table size.

A screenshot of the terminal output confirming the program's functional correctness and timing results is shown in **Figure 5**.



```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

Search 'banana': 250
Delete 'apple': True
Search 'apple': None
Final table state: [[], [], [], [('grape', 300), ('lemon', 150)], [('banana', 250)]]

Insert timing benchmark:

Inserted  100 elements in 0.000052 seconds
Inserted  500 elements in 0.000264 seconds
Inserted 1000 elements in 0.000597 seconds
Inserted 2000 elements in 0.001402 seconds
Inserted 5000 elements in 0.006282 seconds
```

Figure 5.

Figure 5 shows the terminal output showing hash table correctness tests and insertion benchmark results.

These results show that chaining hash tables can handle a wide range of real-world workloads while still performing efficiently, even as the number of elements grows (Sedgewick & Wayne, 2011).

Conclusion

This report looked at the performance and scalability of **Randomized Quicksort** and **Hashing with Chaining** using theory and practical testing. Each method was tested with various input types to see how they performed under different conditions, and the results confirmed that both are practical for real-world use.

Randomized Quicksort delivered steady average-case performance and handled all input types efficiently. It outperformed the deterministic version, which had trouble with sorted and duplicate-heavy data.

The **hash table with chaining** also performed well, proving a reliable way to handle collisions, even as more elements were added.

Even as the number of elements increased, insert times stayed low, supporting the idea that this approach maintains near-constant performance when using a good hash function.

Overall, these experiments highlight the importance of choosing the correct algorithm based on the data and the workload. Randomized algorithms and well-designed data structures—like the ones explored here—continue to be dependable, high-performance solutions in everyday computing tasks.

References

- Bentley, J. L., & McIlroy, M. D. (1993). Engineering a sort function. *Software: Practice and Experience*, 23(11), 1249–1265. <https://doi.org/10.1002/spe.4380231105>
- Carter, J. L., & 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](https://doi.org/10.1016/0022-0000(79)90044-8)
- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to algorithms* (3rd ed.). MIT Press.
- Knuth, D. E. (1998). *The art of computer programming, volume 3: Sorting and searching* (2nd ed.). Addison-Wesley.
- Motwani, R., & Raghavan, P. (1995). *Randomized algorithms*. Cambridge University Press.
- Sedgewick, R., & Wayne, K. (2011). *Algorithms* (4th ed.). Addison-Wesley.