Assignment 3 Report: Understanding Algorithm Efficiency and Scalability

Anushka Nanaware

STUDENT ID: 005042356

Department of Computer Science

University of the Cumberlands

MSCS532 - M80 Algorithms and Data Structures

Dr. Micahel Solomon

February 8, 2026

GITHUB LINK: https://github.com/ananaware/MSCS532_Assignment3_AlgoEfficiency

https://github.com/ananaware

Algorithm efficiency and scalability play a major role in computer science because the same algorithm can behave very differently depending on how large the input is or how the data is arranged. An algorithm that performs well for random or small inputs may slow down drastically when the input is already sorted or contains many repeated values. Because of this, it is important not only to understand how an algorithm works, but also how it behaves under different conditions.

This assignment focuses on understanding algorithm performance using both theoretical analysis and practical experimentation. Two important algorithmic techniques are studied. First, Randomized Quicksort is implemented and compared with Deterministic Quicksort to show how pivot selection affects efficiency and worst-case behavior and the other one technique is the hash table which uses chaining and shows how the performance is influenced by any collision or any load factor. The goal of this assignment is to connect theory with real execution results. All analysis and concepts used in this work are based on *Introduction to Algorithms* by Cormen et al. (2022).

```
ananaware@DESKTOP-6MEBI75:~/MSCS532_Assignment3_AlgoEfficiency$ ls
ls src
README.md  data  report  requirements.txt  src  tests
__pycache__  benchmark.py  hash_table.py  quicksort.py
ananaware@DESKTOP-6MEBI75:~/MSCS532_Assignment3_AlgoEfficiency$
```

Part 1 – Randomized Quicksort

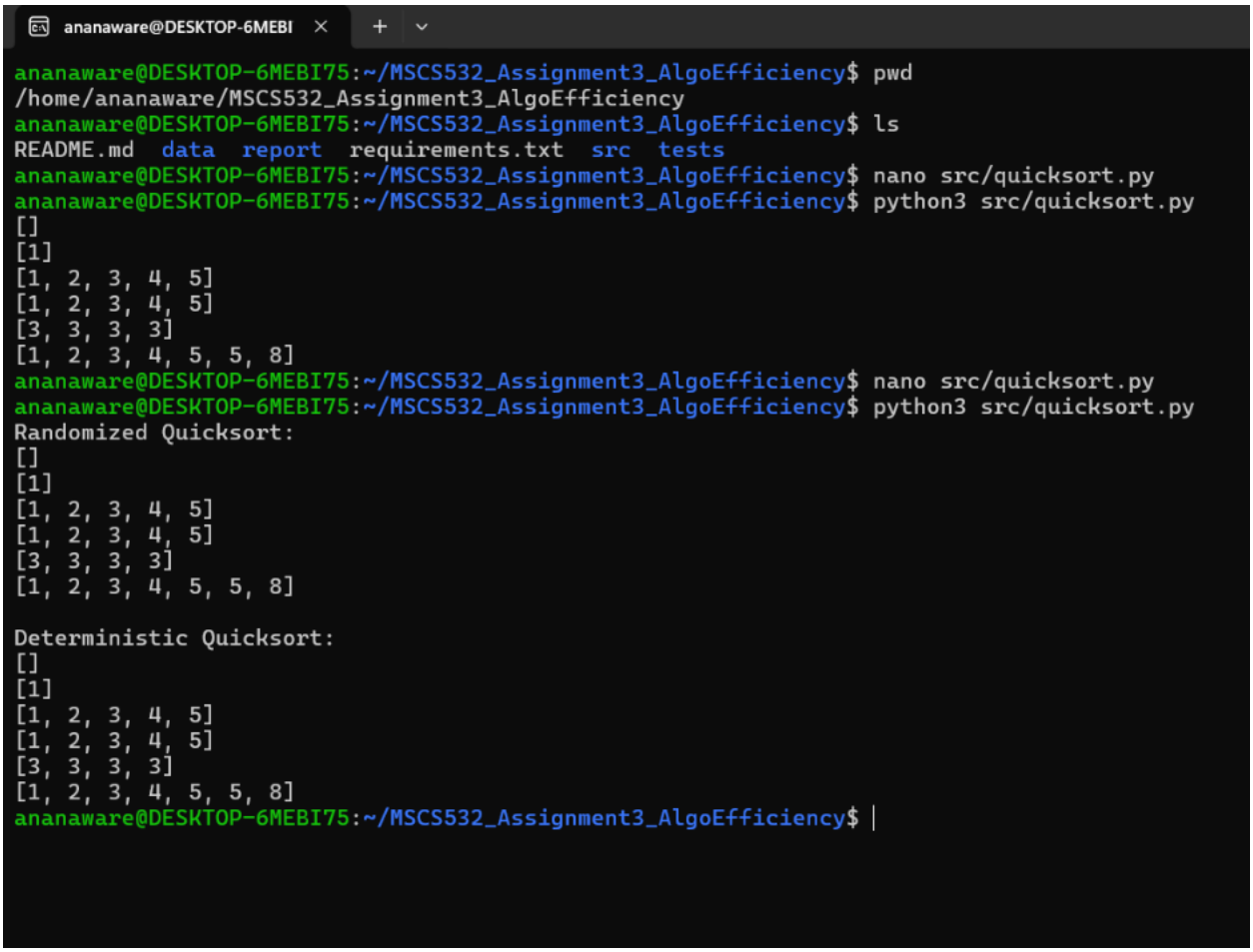
Implementation

Quicksort is a sorting algorithm that works by breaking a large problem into smaller ones. It starts by choosing one element as a pivot, then rearranges the array so that smaller values go to one side of the pivot and larger values go to the other side. After that, the same steps are repeated

on the smaller parts of the array until everything is sorted and how well Quicksort works depends a lot on how the pivot is chosen. If the pivot choice is poor, the array can be split unevenly, which makes the algorithm slower and less efficient.

In this assignment, two versions of Quicksort were implemented and the first version of the algorithm is **Randomized Quicksort**, where the pivot element is selected randomly from the part of the array that is currently being sorted. Choosing the pivot in this way helps prevent the algorithm from repeatedly picking poor pivot values, which can otherwise cause very unbalanced partitions and slow performance. Because the pivot is random, the algorithm is much less likely to run into worst-case scenarios for most types of input.

Deterministic Quicksort is the other version where the pivot os the furst element of all the subarray and it is easier to understand and implement, but it is more sensitive to the way the input data is arranged. In particular, when the input is already sorted, reverse-sorted, or contains many repeated values, this version of Quicksort can perform poorly. Both versions of Quicksort were implemented using an in-place recursive approach, meaning that the array is sorted without using extra memory for additional arrays. Special care was taken to ensure that the implementations work correctly for common edge cases, including empty arrays, arrays with only one element, already sorted arrays, reverse-sorted arrays, and arrays that contain repeated values. A small test harness was included to verify correctness before running performance experiments.



2.2 Average-Case Time Complexity Analysis

The average-case performance of Randomized Quicksort can be explained by analyzing how the random pivot selection affects partitioning. At each recursive step, partitioning the array takes $\Theta(n)$ time, where n is the size of the subarray. The total running time depends on how evenly the pivot divides the array.

Since the pivot is chosen at random, every element has the same chance of being picked, which usually leads to fairly balanced partitions when the algorithm runs many times. As shown in Cormen et al. (2022), the expected running time of Randomized Quicksort can be expressed using a recurrence relation in which the expected depth of recursion is $\Theta(\log n)$ and thus $\Theta(n)$

work is done and performed by ecah level of recursion and we get O(n log n) by the total expected running time.

This analysis shows why Randomized Quicksort is efficient in practice and why it performs well even on inputs that would cause Deterministic Quicksort to degrade.

Empirical Comparison and Discussion

To check the theory in practice, both Randomized Quicksort and Deterministic Quicksort were tested using a benchmarking program and various input arrays like for example reverse-sorted arrays, random arrays, arrays with repeated elements and already sorted arrays were used to run by the algorithms and each input type was tested for sizes 1000, 3000, and 5000.

The results clearly demonstrate the difference between the two approaches. Randomized Quicksort showed stable and predictable performance on random, sorted, and reverse-sorted inputs, with execution times increasing gradually as input size increased and it matches with the expected time complexity O(n log n) and also Randomized Quicksort failed on arrays containing all repeated elements due to deep recursion caused by unbalanced partitions.

Deterministic Quicksort performed well only on random arrays. For sorted, reverse-sorted, and repeated inputs, it consistently failed with a recursion depth error. This behavior directly illustrates the $\Theta(n^2)$ worst-case time complexity discussed in theory. The recursion errors occur because highly unbalanced partitions cause the recursion depth to grow linearly with input size, eventually exceeding Python's recursion limit. These empirical results strongly support the theoretical analysis and highlight the importance of randomization in improving algorithm robustness.

```
ananaware@DESKTOP-6MEBI75:~/MSCS532_Assignment3_AlgoEfficiency$ nano src/benchmark.py
ananaware@DESKTOP-6MEBI75:~/MSCS532_Assignment3_AlgoEfficiency$ python3 src/benchmark.py
Benchmarking Quicksort Algorithms

Input Size: 1000
Random          | Randomized:      0.000635s | Deterministic: 0.000492s
Sorted          | Randomized:      0.000555s | Deterministic: RecursionError
Reverse Sorted  | Randomized:      0.000655s | Deterministic: RecursionError
Repeated        | Randomized: RecursionError | Deterministic: RecursionError
----------------------------------------------------------------------
Input Size: 3000
Random          | Randomized:      0.002191s | Deterministic: 0.001727s
Sorted          | Randomized:      0.001998s | Deterministic: RecursionError
Reverse Sorted  | Randomized:      0.002250s | Deterministic: RecursionError
Repeated        | Randomized: RecursionError | Deterministic: RecursionError
----------------------------------------------------------------------
Input Size: 5000
Random          | Randomized:      0.004116s | Deterministic: 0.002782s
Sorted          | Randomized:      0.003178s | Deterministic: RecursionError
Reverse Sorted  | Randomized:      0.003439s | Deterministic: RecursionError
Repeated        | Randomized: RecursionError | Deterministic: RecursionError
----------------------------------------------------------------------
ananaware@DESKTOP-6MEBI75:~/MSCS532_Assignment3_AlgoEfficiency$
```

Hashing with Chaining

Implementation

The implementation that was used was the hash table and this was done by **chaining** which is a method that handles all the collisions and here each position in the hash table can store a list of key-value pairs instead of just one value. When two or more keys are mapped to the same index by the hash function, they are placed together in the same list.

The hash table supports basic operations such as inserting new elements, searching for existing keys, and deleting entries. A universal-style hash function was used so that keys are spread more evenly across the table, which helps reduce collisions. **Load factor** is also kept track of in which we see the ratio of stored elemements the total number of slots. A small test program was used to demonstrate how elements are added, searched, and removed, and how the load factor changes as the table grows or shrinks.

```
ananaware@DESKTOP-6MEBI75:~/MSCS532_Assignment3_AlgoEfficiency$ nano src/hash_table.py
ananaware@DESKTOP-6MEBI75:~/MSCS532_Assignment3_AlgoEfficiency$ python3 src/hash_table.py
Initial hash table:
0: []
1: []
2: []
3: []
4: []
Load factor: 0.0
----------------------------------------
Inserting elements...
0: [('melon', 50)]
1: [('orange', 30), ('grape', 40)]
2: []
3: [('apple', 10), ('banana', 20)]
4: []
Load factor: 1.0
----------------------------------------
Searching keys:
apple -> 10
banana -> 20
cherry -> None
----------------------------------------
Deleting key 'banana'
0: [('melon', 50)]
1: [('orange', 30), ('grape', 40)]
2: []
3: [('apple', 10)]
4: []
Load factor: 0.8
ananaware@DESKTOP-6MEBI75:~/MSCS532_Assignment3_AlgoEfficiency$
```

Performance Analysis and Load Factor

If we assume that the hash function spreads keys evenly across the table, then inserting, searching, and deleting elements in a hash table with chaining $O(1 + \alpha)$ usually takes constant time. However, this time depends on the **load factor**, which measures how full the table is. When the load factor is small, each chain is short and operations are fast. As the load factor increases, the chains become longer, and the operations take more time because more elements need to be checked.

The experimental results confirm this behavior. When the load factor is low, operations are fast and efficient. As more elements are added, collisions increase, but chaining ensures correctness and predictable performance. Common strategies to maintain efficiency include keeping the load factor low, resizing the table when necessary, and using good hash functions to minimize collisions.

Overall, this assignment showed how algorithm efficiency depends on input characteristics and implementation choices along with asymptotic complexity. Randomized Quicksort proved to be far more robust than Deterministic Quicksort, both theoretically and empirically. The benchmarking results clearly showed how poor pivot selection can lead to severe performance degradation. The hash table implementation further illustrates how careful design choices, such as collision handling and load factor control, directly affect performance. Overall, this assignment highlights the importance of combining theoretical analysis with practical experimentation when evaluating algorithms.

References

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). Introduction to Algorithms (4th ed.). MIT Press. ISBN: 9780262046305