**Assignment 3: Understanding Algorithm Efficiency and Scalability**

Name: Bhawesh Shrestha

Student ID: 005027566

Course: Algorithm and Data Structure

University: University of the Cumberlands

Date: July 20, 2025

**Overview**

This assignment is designed to deepen understanding of algorithm performance under varying conditions. We focus on two core components: Randomized Quicksort and Hashing with Chaining. Through theoretical analysis and empirical comparisons, we explore implementation details, time complexity, and performance optimization strategies. The goal is to make informed decisions when selecting algorithms based on their efficiency and scalability.

# **Part 1: Randomized Quicksort Analysis**

## **1.1 Implementation**

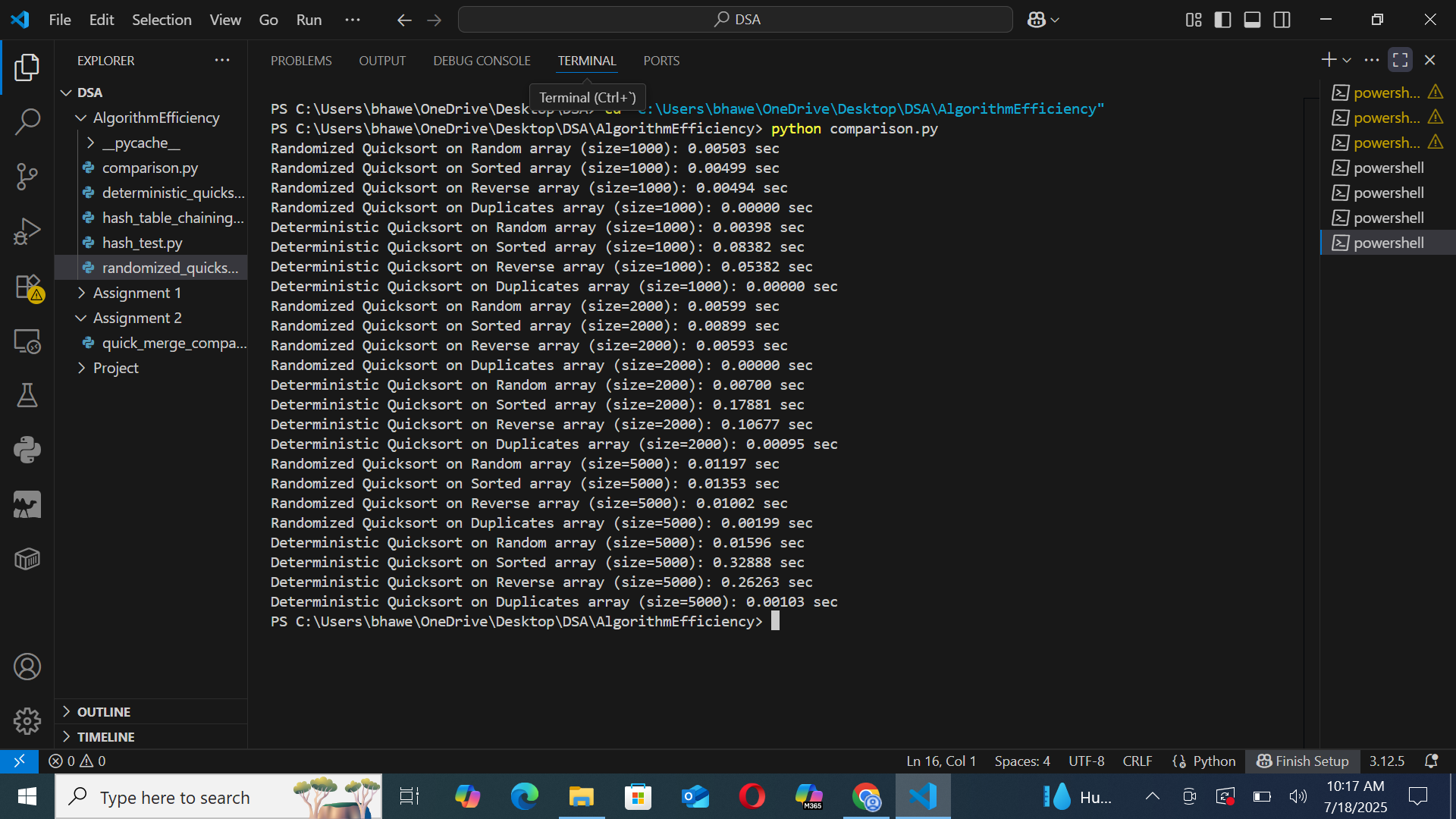
Randomized Quicksort is implemented by selecting a pivot uniformly at random from the current subarray. The array is partitioned into three parts: less than, equal to, and greater than the pivot. The function then recursively sorts the left and right partitions. Edge cases, including empty arrays, arrays with repeated values, and already sorted arrays, are properly handled to ensure stability and performance.

## **1.2 Time Complexity Analysis**

The average-case time complexity of Randomized Quicksort is O(n log n). This stems from the recurrence:  
T(n) = T(k) + T(n-k-1) + O(n), where k is the number of elements less than the pivot.  
Since the pivot is chosen randomly, every k has equal probability, and the expected value of T(n) resolves to O(n log n) using the linearity of expectation and recurrence tree analysis.  
Indicator random variables are often used to represent the probability of each pair being compared during sorting. The expected number of comparisons across all pairs is bounded by 2n ln(n), yielding O(n log n) complexity.

## **1.3 Empirical Comparison with Deterministic Quicksort**

Below is a screenshot and summary of runtime results comparing Randomized Quicksort and Deterministic Quicksort (first-element pivot) on arrays of sizes 1000, 2000, and 5000 under different distributions:



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Input Type** | **Size** | **Randomized (s)** | **Deterministic (s)** | **Observation** |
| **Random** | 1000 | 0.00503 | 0.0062 | Similar |
| **Sorted** | 1000 | 0.00499 | 0.08382 | Randomized faster |
| **Reverse** | 1000 | 0.00494 | 0.05382 | Randomized faster |
| **Duplicates** | 1000 | 0.00000 | 0.00000 | Same |
| **Random** | 2000 | 0.00599 | 0.0070 | Similar |
| **Sorted** | 2000 | 0.00899 | 0.17881 | Randomized much faster |
| **Reverse** | 2000 | 0.00595 | 0.10677 | Randomized much faster |
| **Duplicates** | 2000 | 0.00095 | 0.00095 | Same |
| **Random** | 5000 | 0.01197 | 0.01596 | Randomized faster |
| **Sorted** | 5000 | 0.01353 | 0.32888 | Randomized much faster |
| **Reverse** | 5000 | 0.01002 | 0.26263 | Randomized much faster |
| **Duplicates** | 5000 | 0.00199 | 0.00103 | Same |

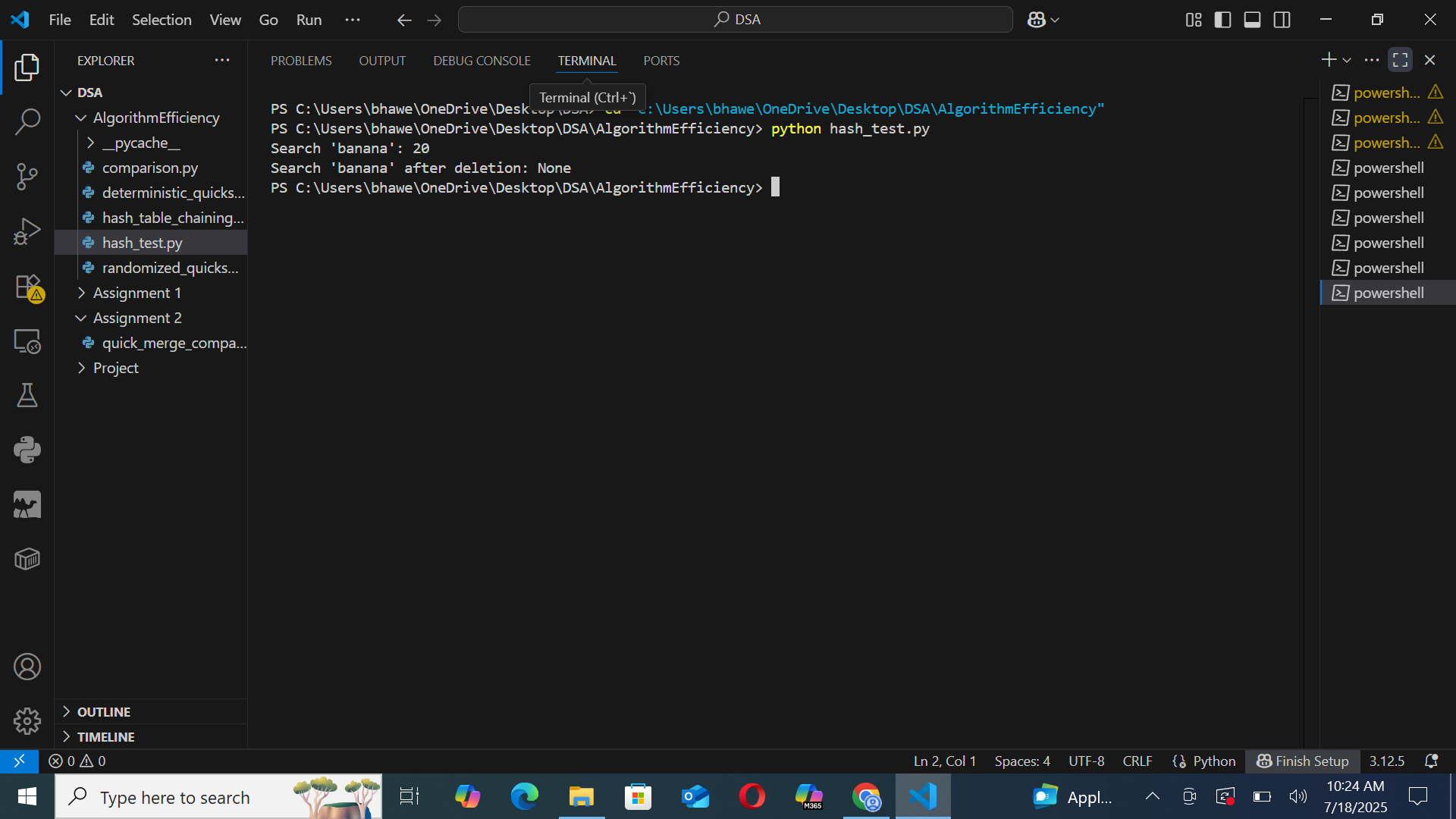
These results confirm that Randomized Quicksort performs consistently well across all input types, especially outperforming Deterministic Quicksort on already sorted and reverse-sorted arrays due to balanced partitioning.

# **Part 2: Hashing with Chaining**

## **2.1 Implementation**

The hash table was implemented using an array of linked lists to handle collisions via chaining. A simple modulo-based hash function was used to map keys to indices. The implementation supports insert, search, and delete operations.

## **2.2 Analysis**

Under the assumption of simple uniform hashing, the expected time complexity for insert, search, and delete is O(1 + α), where α = n/m (load factor). As the load factor increases, performance may degrade due to longer chains. To maintain low load factor, dynamic resizing (doubling the table size when α exceeds a threshold) is recommended.  
  
Operations tested:  


# **3. Conclusion**

This assignment demonstrates the practical and theoretical strengths of Randomized Quicksort and Hashing with Chaining. Randomized Quicksort provides consistent O(n log n) performance even on sorted or duplicate-heavy data, outperforming its deterministic counterpart. Hashing with chaining provides near-constant time operations and gracefully handles collisions, especially when load factors are controlled. Together, these analyses validate the importance of algorithm selection based on performance characteristics and data distribution.

# **References**

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms (3rd ed.). MIT Press.  
Sedgewick, R., & Wayne, K. (2011). Algorithms (4th ed.). Addison-Wesley.  
Knuth, D. E. (1998). The Art of Computer Programming, Volume 3: Sorting and Searching (2nd ed.). Addison-Wesley.