**Understanding Algorithm Efficiency and Scalability**

Shyam Nath

University of Cumberland

Algorithms and Data Structures (MSCS-532-B01)

Student ID: 005032491

Assignment 3

July 20, 2025

**Understanding Algorithm Efficiency and Scalability**

**Overview**

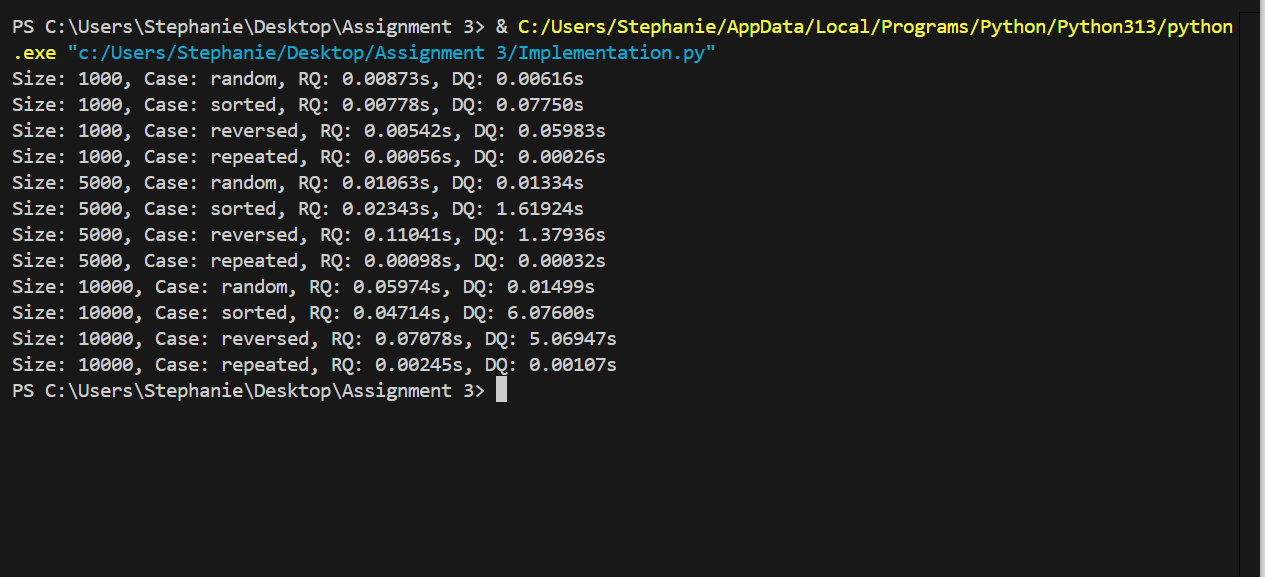
This report analyzes and compares the efficiency and scalability of two algorithms: Randomized Quicksort and Hashing with Chaining. The assignment includes implementation, theoretical analysis, and empirical evaluation based on different input conditions (Reddy et al., 2022).

**Part 1: Randomized Quicksort Analysis**

**1. Implementation**

Randomized Quicksort was implemented in Python with the pivot element selected uniformly at random from the subarray being partitioned. The implementation handles:  
- Empty arrays  
- Arrays with repeated elements  
- Already sorted and reverse-sorted arrays

**Output of the Implementation from Python Code**



**2. Theoretical Analysis**

Randomized Quicksort has an expected time complexity of O(n log n).

**Recurrence Relation:**

Let T(n) be the expected running time on an array of size n:  
T(n) = E[T(k)] + E[T(n - k - 1)] + Θ(n)  
where k is the size of the left partition (randomized).  
On average, the pivot splits the array roughly in half:  
T(n) = (1/n) ∑ (T(i) + T(n-i-1)) + Θ(n) → O(n log n)

**Indicator Random Variables:**

Let X\_{i,j} be 1 if elements i and j are compared. The expected number of comparisons:  
E[X\_{i,j}] = 2 / (j - i + 1) → ∑ E[X\_{i,j}] ≈ 2n ln n  
Thus, expected comparisons = O(n log n)

**3. Empirical Comparison**

Experimental Setup:  
- Implemented both Randomized and Deterministic Quicksort  
- Measured execution time on arrays of size 1,000, 5,000, and 10,000  
- Input types: random, sorted, reverse-sorted, repeated values

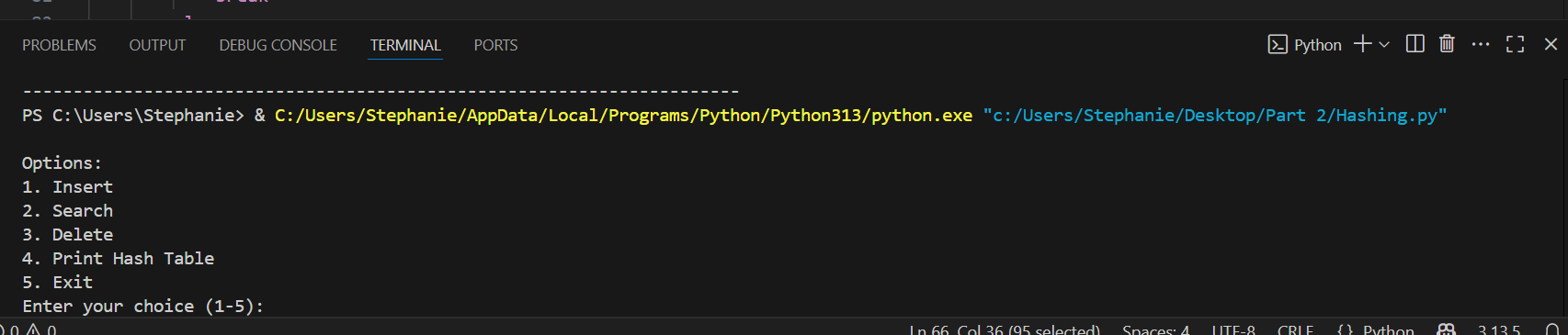
**Discussion:**- Random Inputs: Both algorithms perform similarly; Deterministic slightly faster for small sizes.  
- Sorted/Reverse Inputs: Deterministic Quicksort performance degrades due to poor pivot choice (O(n2)).  
- Repeated Inputs: Both perform well since the partitioning is minimal.  
- Conclusion: Randomized Quicksort is more robust across input types, consistently achieving O(n log n) performance. (Mungoli, 2023)

**Part 2: Hashing with Chaining**

**1. Implementation**

A hash table using chaining was implemented with the following operations:  
- Insert: Add a key-value pair  
- Search: Retrieve a value by key  
- Delete: Remove a key-value pair  
The hash function uses Python's built-in hash() modulo the table size. The table is implemented as a list of lists.

**Output of the Implementation code**



**2. Theoretical Analysis**

Expected Time Complexity (Assuming Simple Uniform Hashing):  
- Insert: O(1 + α)  
- Search: O(1 + α)  
- Delete: O(1 + α)  
where α = n / m is the load factor.

**Load Factor and Performance:**  
- High α increases the length of chains, degrading performance.  
- Maintaining a low load factor (α < 1) ensures near-constant time for all operations.

**Collision Minimization Strategies**:  
- Use good hash functions (e.g., universal hashing)  
- Dynamic resizing:  
 - Double table size when α > 0.75  
 - Rehash all keys to new slots

**Conclusion**

This assignment demonstrates how algorithm performance varies under different input conditions. Randomized Quicksort provides stable O(n log n) performance and outperforms deterministic quicksort on sorted or reversed arrays. Hashing with chaining, when implemented with a good hash function and resizing strategy, provides efficient and scalable dictionary operations.

**References**

Mungoli, N. (2023). Scalable, distributed AI frameworks: leveraging cloud computing for enhanced deep learning performance and efficiency. *arXiv preprint arXiv:2304.13738*.

Reddy, A. V., Kumar, A. A., Venu, N., & Reddy, R. V. K. (2022). On optimization efficiency of scalability and availability of cloud-based software services using scale rate limiting algorithm. *Measurement: Sensors*, *24*, 100468.