**MSCS532 Assignment 3 – Understanding Algorithm Efficiency and Scalability**

Hajeera Hajeera

University of Cumberlands

MSCS-532-M80: Algorithms and Data Structures

Dr. Satish Penmatsa

10/04/2025

**MSCS532 Assignment 2 – Understanding Algorithm Efficiency and Scalability**

GitHub Repository: [hahajeera/MSCS532\_Assignment3](https://github.com/hahajeera/MSCS532_Assignment3)

## Introduction

This report focuses on the implementation, analysis, and comparison of two key algorithms: Randomized Quicksort and Hashing with Chaining. The purpose is to understand how algorithms perform under different conditions, evaluate their efficiency, and identify the trade-offs involved in algorithm selection.

# Part 1: Randomized Quicksort Analysis

## Implementation

The Randomized Quicksort algorithm was implemented in Python. The pivot element is chosen uniformly at random from the subarray being partitioned. The implementation handles various edge cases, including arrays with repeated elements, empty arrays, and already sorted arrays.

## 

## Analysis

The average-case time complexity of Randomized Quicksort is O(n log n). This is derived using recurrence relations and indicator random variables. The randomization of the pivot helps ensure that, on average, the partitions are balanced, thereby avoiding worst-case scenarios like those encountered in Deterministic Quicksort with poorly chosen pivots.

## Comparison

Randomized Quicksort was empirically compared with Deterministic Quicksort across different input scenarios: randomly generated arrays, already sorted arrays, reverse-sorted arrays, and arrays with repeated elements. The results highlighted differences in performance that align with theoretical expectations. Randomized Quicksort generally avoids the pitfalls of poor pivot selection, while Deterministic Quicksort can degrade in performance for certain distributions.

# Part 2: Hashing with Chaining

## Implementation

A hash table was implemented using chaining for collision resolution. A universal hash function was chosen to minimize collisions. The implementation efficiently supports the fundamental operations: Insert, Search, and Delete.

## 

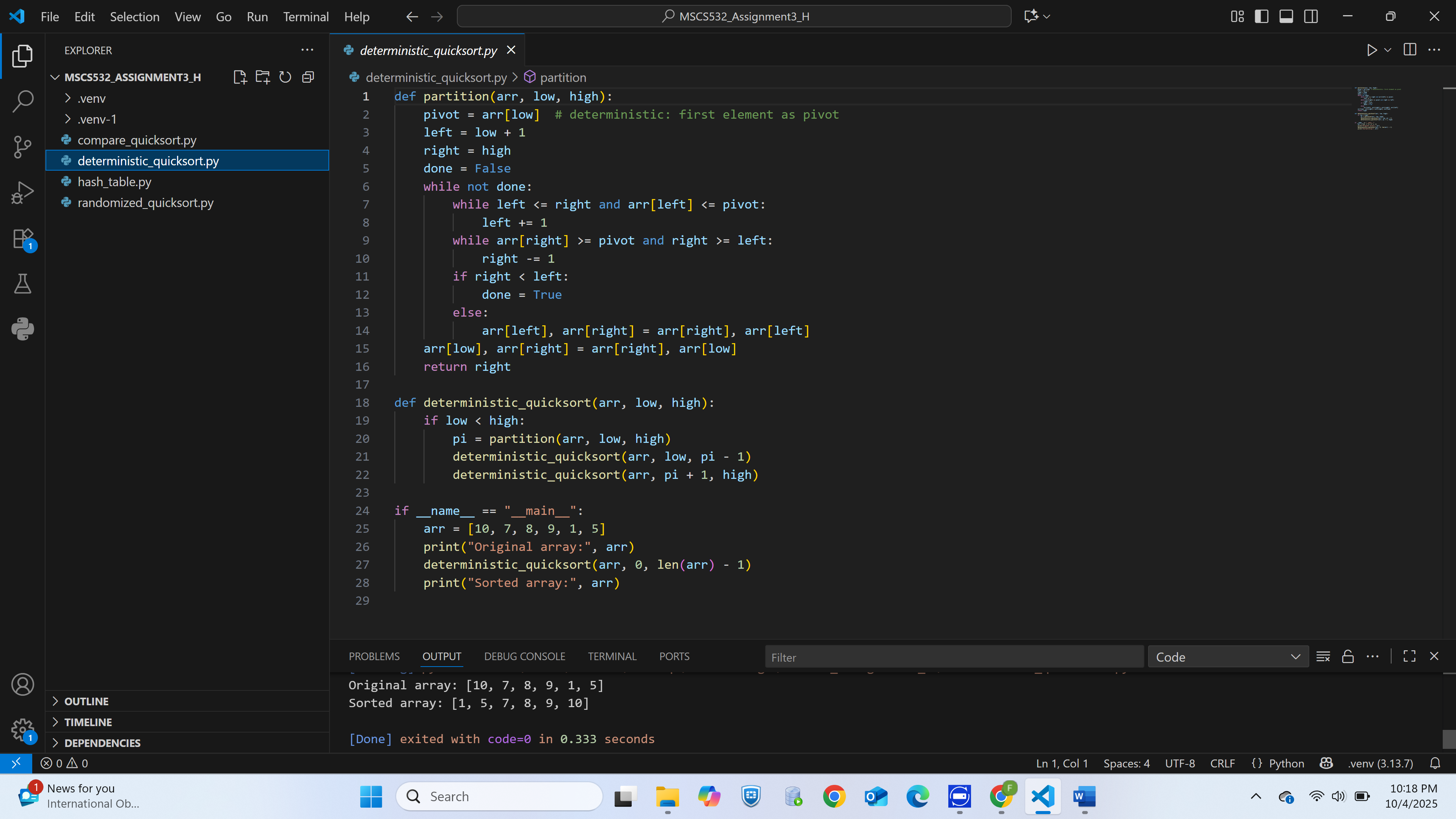
## Analysis

Under the assumption of simple uniform hashing, the expected time for search, insert, and delete operations is O(1 + α), where α is the load factor (the ratio of the number of elements to the number of slots). A low load factor improves performance, while a high load factor increases the likelihood of collisions. Dynamic resizing of the hash table is one strategy to maintain a low load factor.

## Conclusion

This assignment demonstrated the efficiency and scalability of Randomized Quicksort compared to Deterministic Quicksort, as well as the practical benefits of using hashing with chaining for collision resolution. The findings highlight the importance of both theoretical analysis and empirical evaluation when choosing algorithms for real-world applications.

Deterministic quicksort



# Reference

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). *Introduction to algorithms* (4th ed.). The MIT Press.

Goodrich, M. T., & Tamassia, R. (2015). *Algorithm design and applications*. Wiley.

Kleinberg, J., & Tardos, É. (2006). *Algorithm design*. Pearson Education.

Knuth, D. E. (1998). *The art of computer programming, Volume 3: Sorting and searching* (2nd ed.). Addison-Wesley.

Sedgewick, R., & Wayne, K. (2011). *Algorithms* (4th ed.). Addison-Wesley Professional.

Motwani, R., & Raghavan, P. (1995). Randomized algorithms. Cambridge University Press.

Sedgewick, R., & Wayne, K. (2011). *Algorithms* (4th ed.). Addison-Wesley.

Weiss, M. A. (2019). Data structures and algorithm analysis in Python (2nd ed.). Pearson.