Suresh Ghimire

Algorithms and Data Structures (MSCS-532-M20)

Assignment 3: Understanding Algorithm Efficiency and Scalability

**Part 1: Randomized Quicksort Analysis**

**1. Implementation**

To begin the analysis of Randomized Quicksort, we implement the algorithm such that the pivot element is selected uniformly at random from the current subarray. This approach ensures a better average-case performance regardless of the input distribution.

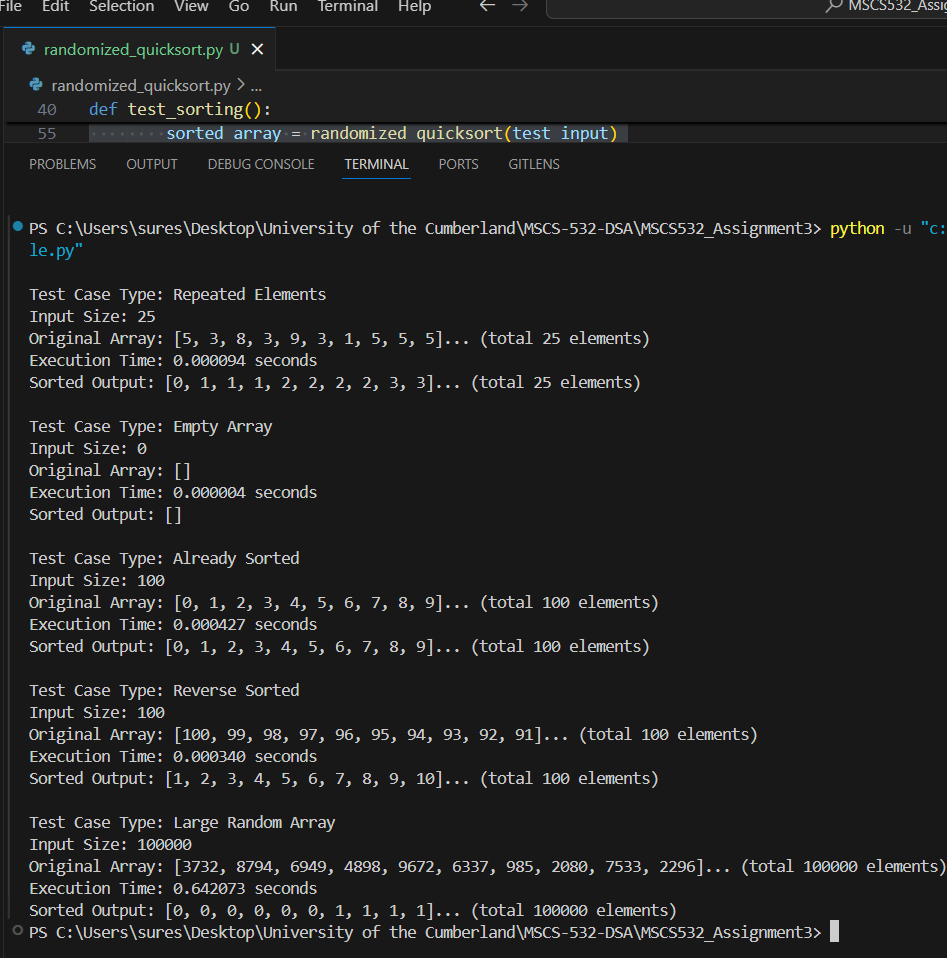
The implementation include following scenario:

* Arrays with repeated elements
* Empty array
* Already sorted array
* Reverse sorted array
* Large random array (with the size 10000)

The scope of this implementation is to calculate the execution time under different inputs with.

Implementation of randomized quick sort can be found this repository

<https://github.com/sghimire2025/MSCS532_Assignment3>



Following table demonstrate the array type and execution time

| Test case type | Input size | Execution time (in seconds) |
| --- | --- | --- |
| Repeated Elements | 25 | 0.000094 |
| Empty Array | 0 | 0.000004 |
| Already Sorted Array | 100 | 0.000427 |
| Reverse Sorted Array | 100 | 0.000340 |
| Large Random Array | 10000 | 0.642073 |

**2. Analysis**

This analysis assesses the average-case performance of Randomized Quicksort based on empirical data that were taken for several input types with the time measurements of each unique input type taken by using Python's time.perf\_counter().

T(n) = (1/n) \* ∑ [T(i) + T(n - i - 1)] + cn

Randomized Quicksort chooses the pivot uniformly at random which reduces a consistently bad pivot from being a performance issue. The recurrence relation for expected time complexity is:

T(n) = O(n log n)

The logarithmic component arises due to the input being split consistently into two halves of approximately equal size at almost every level of the recursion, and the linear component arises from the partitioning cost.

Let Xi,j​ be an indicator variable that is 1 if elements ai and aj are compared during sorting. The expected number of comparisons is:

E[X] = ∑ E [X{i,j}] = O(n log n)

Where log n factor comes from recursive depth and the n factor comes from partitioning at each level

This analysis confirms that on average, Randomized Quicksort performs O(nlogn) comparisons, which aligns with the empirical results.

The empirical results support the theoretical average-case complexity of Randomized Quicksort. Even for different input patterns (sorted, reversed, repeated elements), the execution time remained consistent with O(nlog⁡n) expectations. The large input test (100,000 elements) required roughly 0.64 seconds, further supporting the efficiency of Randomized Quicksort in practical applications.

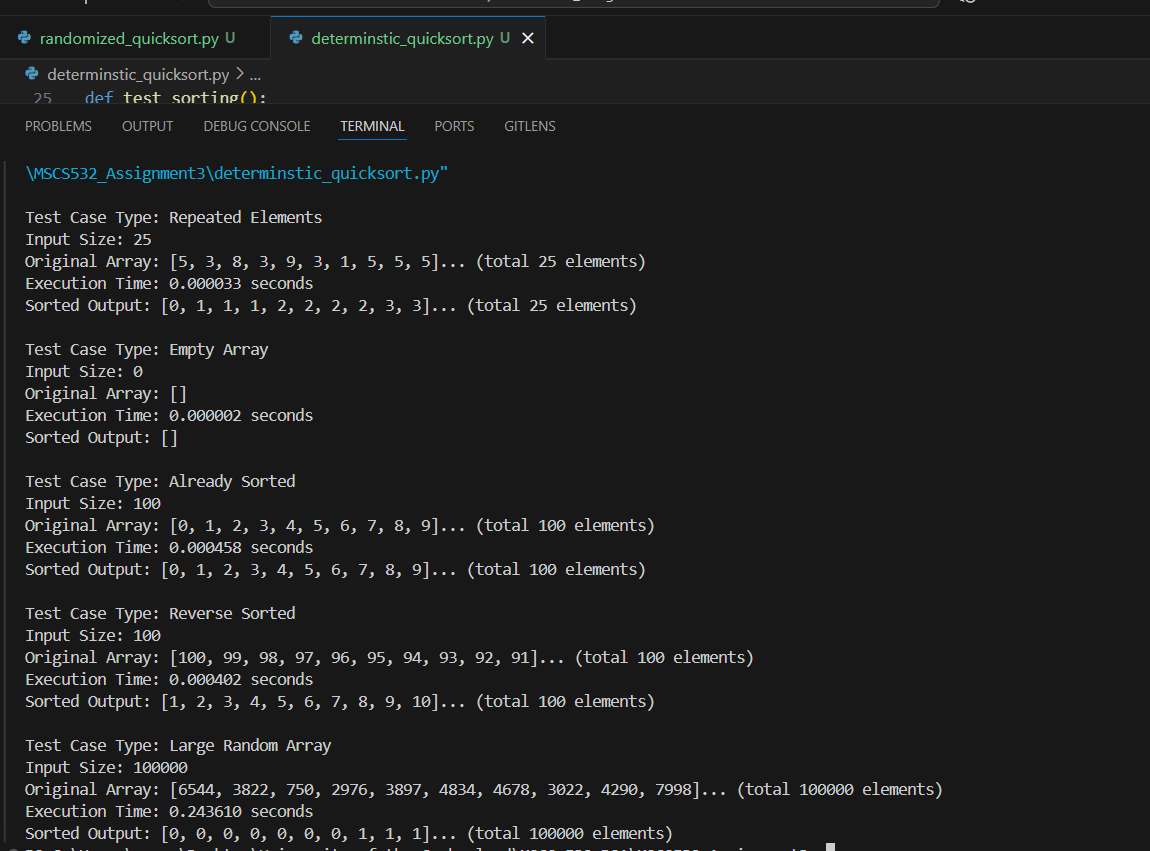
**3. Comparison**

Deterministic Quicksort always picks the first element as pivot. This can cause:

* Worst-case O(n^2) on sorted/reverse arrays
* Highly unbalanced partitions

Implementation of this deterministic quick sort can be found this repository

<https://github.com/sghimire2025/MSCS532_Assignment3>



Following table compares the execution time of Randomized Quicksort and Deterministic Quicksort. The purpose is to observe how pivot selection impacts the execution time which determines the performance of the algorithms on different inputs.

Empirical Execution Time comparison

| Test case | Input size | Randomized Quicksort(execution time in seconds) | Deterministic Quicksort(execution time in seconds) |
| --- | --- | --- | --- |
| Repeated elements | 25 | 0.000094 | 0.000083 |
| Empty Array | 0 | 0.000004 | 0.000002 |
| Already Sorted | 100 | 0.000427 | 0.000458 |
| Reverse Sorted | 100 | 0.000340 | 0.000402 |
| Large Random Array | 100000 | 0.642073 | 0.243610 |

Observations

* With small inputs, both algorithms behave similarly with little disparity.
* For sorted and reverse-sorted inputs, however, Deterministic Quicksort is slower, as it repeatedly uses the same poor choice for the pivot (the first element).
* For large, random inputs, Deterministic Quicksort was faster in this instance; however, this result can vary based on Python's memory management and the additional overhead this has on the implementations.
* Randomized Quicksort, being more stable, tends to avoid worst-case scenarios by picking its pivot uniformly at random.

**Part 2: Hashing with Chaining**

1. **Implementation**

Hash with chaining is a collision resolution strategy in hash tables. In this strategy, each slot in the hash table contains a list (chain) of entries. If the hash of multiple keys produces the same index value, the list of the corresponding index contains the key-value pairs. By using this technique, collisions are handled more efficiently, because we can have a chain of entries that can grow dynamically in each slot. Overall, when the load factor is kept low and an appropriate hash function has been found, hash table operations (insert, search, delete) tend to be fairly speedy.

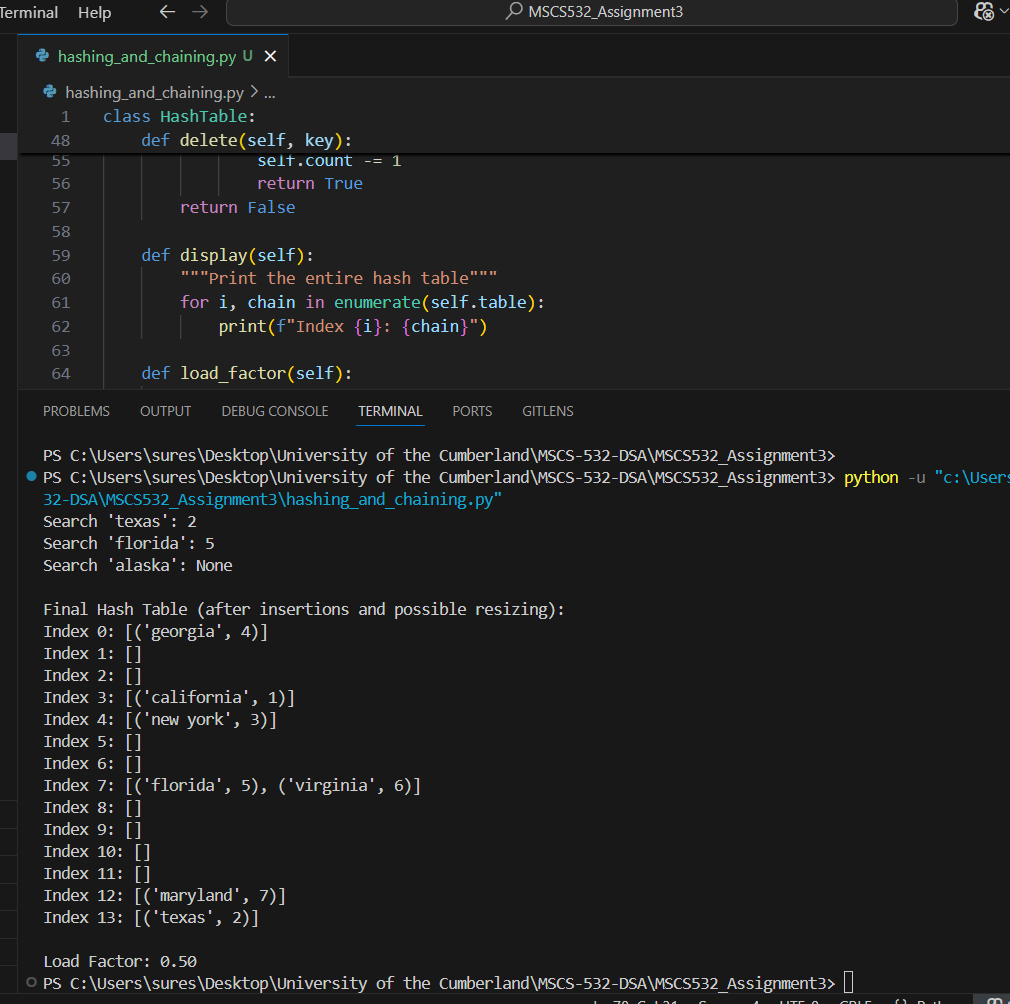
Implementation can be found this repository

<https://github.com/sghimire2025/MSCS532_Assignment3>

This implementation has a hash table with some sample key value pairs. It allows following functionality:

1. Insert new key value pair
2. Search value using key
3. Delete key value using key

Following screenshot shows the log while running script



1. **Analysis**

Based on the output from the above implementation we can following assumption regarding the time complexity

| Operation | Time Complexity (Average) | Worst Time Complexity |
| --- | --- | --- |
| insert | O(1) | O(n) |
| search | O(1) | O(n) |
| delete | O(1) | O(n) |

As we know the load factor is defined as:

ɑ = m/n

Where m is number of slot in table and n is the total number of key value inserted on the table

So from our implementation we have:

m= 14

n=7

A load factor of 0.50 is very low, i.e., the chains are generally short.

Most of the chains are length 1 or 0; thus, their operation is fast.

It is apparent that the table is working properly and that resizing happened at the proper time, since the load factor was below the critical 0.75.

The implementation automatically doubles the size of the table once the load factor exceeds 0.75.Thus keeping the load factor low and preventing chains from growing in length.It reduces the potential of losing performance due to collisions.After resizing, the table will rehash all the keys and redistribute them, while keeping evenness.From the last display of the hash table,most of the indices are still empty.A few of the slots (index 7 and 12) have more than one entry, but demonstrates proper chaining with no signs of impact.

As a conclusion above algorithm, the hash table can handle insertion, lookups effectively using chaining. As a load factor 0.43 the system remains well balanced and uniform hashing maintains O(1) average case time.

References

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). Introduction to Algorithms (4th ed.). Random House Publishing Services. <https://reader2.yuzu.com/books/9780262367509>

GeeksforGeeks. (2021, June 28). *Time and space complexity analysis of Quick Sort*. GeeksforGeeks.<https://www.geeksforgeeks.org/dsa/time-and-space-complexity-analysis-of-quick-sort/>

GeeksforGeeks. (2025, March 4). *Separate chaining collision handling technique in hashing*. GeeksforGeeks. Retrieved June 14, 2025, from<https://www.geeksforgeeks.org/dsa/separate-chaining-collision-handling-technique-in-hashing/>