**Assignment 3**

Akash Shrestha

University of the Cumberlands

MSCS-532-M20

Professor Brandon Bass

June 15, 2025

**Part 1: Randomized Quicksort Analysis**

**Implementation Overview**

This project implemented two sorting algorithms using Python: Randomized Quicksort and Deterministic Quicksort. The randomized variant selects the pivot element uniformly at random from the subarray, whereas the deterministic variant uses the middle element of the subarray as the pivot. The purpose was to evaluate their efficiency under various input conditions and compare their empirical performance.

Test inputs included:

* Randomly generated arrays
* Already sorted arrays
* Reverse-sorted arrays
* Arrays with repeated elements

Each algorithm was tested for correctness and execution time, and the results were analyzed to validate the theoretical time complexities.

**Theoretical Time Complexity**

***Randomized Quicksort***

Randomized Quicksort has the best-case and average-case time complexity of O(nlogn), with the worst-case of O(n^2). However, the worst case is rare due to the random selection of pivots, which probabilistically balance partitions.

The expected number of comparisons is given by the recurrence:

This result can be derived using indicator random variables and linearity of expectation.

***Deterministic Quicksort***

When using the middle element as a pivot, deterministic quicksort performs well in most practical cases. However, it lacks the probabilistic balancing of Randomized Quicksort and can still exhibit unbalanced partitions depending on input distribution. Its worst-case time complexity is also O(n^2), but the average-case performance remains O(nlogn).

**Empirical Performance Comparison**

The following results were observed by timing both algorithms on arrays of size 1000:

A screenshot of a computer program

AI-generated content may be incorrect.

**Discussion**

Randomized Quicksort performed consistently across all input types, validating its theoretical robustness. Deterministic Quicksort, while efficient with random and repeated inputs, was slightly slower on sorted and reverse-sorted inputs due to less optimal partitioning.

These observations affirm the importance of pivot selection strategy, especially when algorithm performance must be robust to various data distributions.

**Part 2: Hashing with Chaining**

**Implementation Overview**

A hash table was implemented in Python using the chaining technique for collision resolution. The structure consists of an array of lists (buckets), where each index is determined by a hash function. In the case of a collision, elements are appended to the respective bucket list.

The hash table supports three operations:

* Insert(key, value)
* search(key)
* delete(key)

Python’s built-in hash() function was used to generate hash codes for keys.

**Expected Time Complexity**

Assuming simple uniform hashing, the expected time complexity for each operation is:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | Operation | Time Complexity | | Insert | *O*(1+α) | | Search | *O*(1+α) | | Delete | *O*(1+α) | |  |

Where α=n/m is the load factor, with:

* n: number of inserted elements
* m: number of hash table buckets

**Load Factor and Resizing**

The load factor plays a critical role in hash table efficiency. When α≤1, operations approach constant time. When α\alphaα increases due to many collisions, performance degrades.

To maintain optimal performance:

* Monitor the load factor
* Resize and rehash the table when α\alphaα exceeds a threshold (e.g., 0.75)

**Testing and Output**

The following operations were executed to verify functionality:

A screen shot of a computer

AI-generated content may be incorrect.The output confirms that the hash table correctly inserts, updates, searches, and deletes entries.

**Conclusion**

This assignment demonstrated how algorithm choice impacts performance under varying input conditions. Randomized Quicksort proved efficient and robust across all test cases due to its probabilistic pivot selection. Deterministic Quicksort showed slightly less stability but improved with a better pivotal strategy.

The hash table implementation showcased the importance of managing the load factor to maintain efficient average-case operations. These experiments highlight key trade-offs in algorithm design and data structure implementation.

**References**

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). *Introduction to algorithms*. MIT press.