# Assignment 3: Understanding Algorithm Efficiency and Scalability

## Sahaj Shrestha

## MSCS-532-B01 Algorithms and Data Structures

**Part 1: Randomized Quicksort**

**1. Implementation**

Randomized Quicksort is implemented by choosing a pivot uniformly at random from the subarray being partitioned. This approach minimizes the likelihood of encountering worst-case scenarios, such as already sorted or reverse-sorted arrays.

**2. Theoretical Analysis**

The average-case time complexity of Randomized Quicksort is:

T(n) = O(n log n)

Explanation:  
 In Randomized Quicksort, the pivot is chosen randomly. On average, this results in balanced partitions. The recurrence relation for average-case can be expressed as:

T(n) = (1/n) ∑ [T(k) + T(n−k−1)] + O(n), for k = 0 to n−1  
 This solves to O(n log n) using recurrence tree or probabilistic analysis with indicator random variables.

**3. Empirical Comparison**

The table below compares the performance (in seconds) of Randomized Quicksort vs Deterministic Quicksort (pivot = first element) for 10000 inputs:

|  |  |  |
| --- | --- | --- |
| Input Type | Randomized Quicksort | Deterministic Quicksort |
| Random | 0.02712 s | 0.04452 s |
| Sorted | 0.04047 s | 2.14154 s |
| Reversed | 0.01918 s | 3.83376 s |
| Duplicates | 0.57813 s | 0.24876 s |

**Discussion and Conclusions:**

Randomized Quicksort maintains stable performance across all inputs due to balanced recursion. Deterministic Quicksort degrades to O(n²) time in sorted or reverse-sorted scenarios. Duplicates affect both, but not drastically due to proper comparison logic.

**Part 2: Hashing with Chaining**

**1. Implementation**

We implemented a hash table using chaining (linked lists at each index) to handle collisions. The Python built-in hash() function was used, modulo the table size.

Supported operations are : Insert: Add a key-value pair , Search: Retrieve value by key and Delete: Remove a key-value pair.

**2. Complexity Analysis**

Under the simple uniform hashing assumption:

Insert: O(1 + α)

Search: O(1 + α)

Delete: O(1 + α)

where α = n / m is the load factor (n = number of elements, m = table size).

**3. Load Factor and Collision Minimization**

* High load factor leads to longer chains like slower operations.
* Maintain α < 0.75 for optimal performance.
* Resize table dynamically when load factor crosses threshold.
* Use good hash functions (for example from a universal family) to minimize collisions.

**4. Github Link:**

https://github.com/sahajshrestha/MSCS532\_Assignment3

**5. Screenshots of the Sample Run:**

A screenshot of a computer program

AI-generated content may be incorrect.

**Conclusion**

Randomized Quicksort is resilient to poor input distributions and offers consistent performance across array types.Deterministic Quicksort suffers with sorted and reverse-sorted arrays.Hashing with Chaining is efficient when the load factor is controlled. Collision resolution is simple and effective.This assignment demonstrates both theoretical and empirical understanding of algorithm efficiency and scalability.