Assignment 3: Algorithm Efficiency and Scalability Report

Name: Pranavi Balakulla
Course: Algorithms and Data Structures
Date: 02/05/2026

Introduction

This assignment examines the efficiency and scalability of two fundamental algorithms: Randomized Quicksort and Hash Table with Chaining. Evaluating their performance under different conditions supports the selection of effective solutions for practical applications.

Randomized Quicksort illustrates how randomization in divide-and-conquer algorithms reduces the likelihood of worst-case behavior, while Hash Table with Chaining provides a reliable method for handling collisions in dynamic data storage. These algorithms have practical relevance: Randomized Quicksort is commonly used to sort large datasets in databases and software libraries, and Hash Tables are employed in implementing dictionaries, caches, and indexing structures.

The report presents a theoretical analysis, empirical evaluation, and discussion of results, highlighting how design choices impact algorithmic efficiency.

Part 1: Randomized Quicksort Analysis

1. Implementation

Randomized Quicksort is an in-place sorting algorithm that recursively partitions an array using a pivot chosen uniformly at random. Key implementation details include:

- Random pivot selection: A pivot is selected randomly from the current subarray to minimize the chance of encountering worst-case performance.
- Partitioning: Elements smaller than the pivot are placed on the left, and elements larger than the pivot on the right, maintaining proper order.
- Recursion: Subarrays on either side of the pivot are recursively sorted until the array is fully sorted.
- In-place operation: Swaps are used instead of creating new arrays, minimizing memory usage.
- Edge case handling: The algorithm correctly handles empty arrays, single-element arrays, repeated elements, and already sorted arrays.

2. Theoretical Analysis

Randomized Quicksort is a divide-and-conquer sorting algorithm that selects a pivot uniformly at random from the input array and partitions the remaining elements around it.

Let T(n) denote the expected running time to sort an array of size n. After selecting a pivot, the array is divided into two subarrays:

- One containing k elements smaller than the pivot
- One containing n − k − 1 elements larger than the pivot

The partitioning step requires Θ(n) time.

Thus, the recurrence relation is:

$T(n) = T(k) + T(n - k - 1) + \Theta(n)$

Since the pivot is chosen randomly, each possible value of k from 0 to n − 1 occurs with equal probability 1/n. Taking the expected value over all pivot positions:

$E[T(n)] = (1/n) \Sigma [T(k) + T(n - k - 1)] + \Theta(n)$

for k = 0 to n − 1

This simplifies to:

$E[T(n)] = (2/n) \Sigma T(k) + \Theta(n)$

Solving this recurrence using standard methods yields:

$E[T(n)] = O(n \log n)$

This demonstrates that Randomized Quicksort achieves balanced partitions on average, resulting in logarithmic recursion depth and efficient performance.

Worst-case time complexity remains $O(n^2)$, which occurs when extremely unbalanced partitions are repeatedly produced. However, due to random pivot selection, this scenario has very low probability.

Space complexity is O(log n) on average due to recursive stack calls.

3. Empirical Comparisons:

To evaluate performance in practice, Randomized Quicksort was compared with Deterministic Quicksort (first-element pivot) across multiple input sizes and distributions.

Experiments were conducted using arrays of sizes:

n = 1,000; 5,000; 10,000; 20,000 and the following input types:

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

Execution time was measured in milliseconds and averaged over multiple runs.

Input Type	Randomized Quicksort	Deterministic Quicksort
Random Array	Consistently O(n log n)	Similar but slightly slower
Already Sorted	Maintains O(n log n)	Significantly slower
Reverse Sorted	Maintains O(n log n)	Very slow
Repeated elements	Efficient O(n log n)	Degraded in some cases

Observations:

- Randomized Quicksort shows robust performance across all input types.
- Deterministic Quicksort suffers on already sorted or reverse-sorted inputs due to unbalanced partitioning.
- The empirical results align with theoretical expectations for Randomized Quicksort.

Part 2: Hashing with Chaining

1. Implementation

A Hash Table with Chaining is a data structure that stores key-value pairs and handles collisions by maintaining a linked list at each bucket. Key implementation aspects include:

- Hash function: Computes an index for each key, distributing keys across buckets to minimize collisions.
- Chaining for collisions: Keys that hash to the same index are stored in a linked list, maintaining access to all entries at that bucket.
- Insert operation: Adds a new key-value pair to the head of the chain or updates the value if the key already exists
- Search operation: Traverses the chain at the computed index to locate a key and retrieve its value.
- Delete operation: Removes a key-value pair from the chain, adjusting pointers to maintain the linked list structure.
- Edge case handling: The implementation efficiently supports empty tables, repeated keys, and non-existent keys.

2. Theoretical Analysis

A hash table with chaining stores elements in buckets, where each bucket maintains a linked list of entries that hash to the same index.

Under the assumption of simple uniform hashing, where each key is equally likely to map to any bucket, the expected time complexities are:

- Insert: O(1)
- Search: O(1)
- Delete: O(1)

In the worst case, if all keys hash to the same bucket, operations degrade to O(n).

Load Factor and Its Impact

The load factor α is defined as:

$\alpha = n / m$

where:

- n is the number of stored elements
- m is the number of buckets

As α increases, more elements occupy each bucket, increasing the length of chains and reducing performance.

- When α is small, operations remain close to O(1)
- When α becomes large, collision frequency increases, leading to slower operations

Maintaining Performance

To maintain a low load factor and minimize collisions:

- Dynamic resizing is employed when α exceeds a threshold (e.g., 0.75)
- The table size is increased (usually doubled)
- All existing elements are rehashed into the new table

This ensures continued efficient performance and evenly distributed keys.

Space complexity remains O(n), proportional to stored elements.

Hash Table Performance:

- Insertions, deletions, and searches remain efficient across datasets.
- Collisions are handled gracefully via chaining, ensuring predictable average performance.

5. Discussion

- Randomized pivot selection in Quicksort reduces the risk of worst-case behavior and delivers more stable performance across different input types.
- In contrast, deterministic pivot selection is highly dependent on input order, which demonstrates the practical benefit of randomization.
- The Hash Table with Chaining results show that effective collision handling is essential for preserving performance as the dataset size increases.
- Overall, the observed runtime results align with theoretical expectations, confirming the correctness and efficiency of the implementations.

6. Conclusion

This assignment demonstrates the importance of algorithm design choices on performance and scalability.

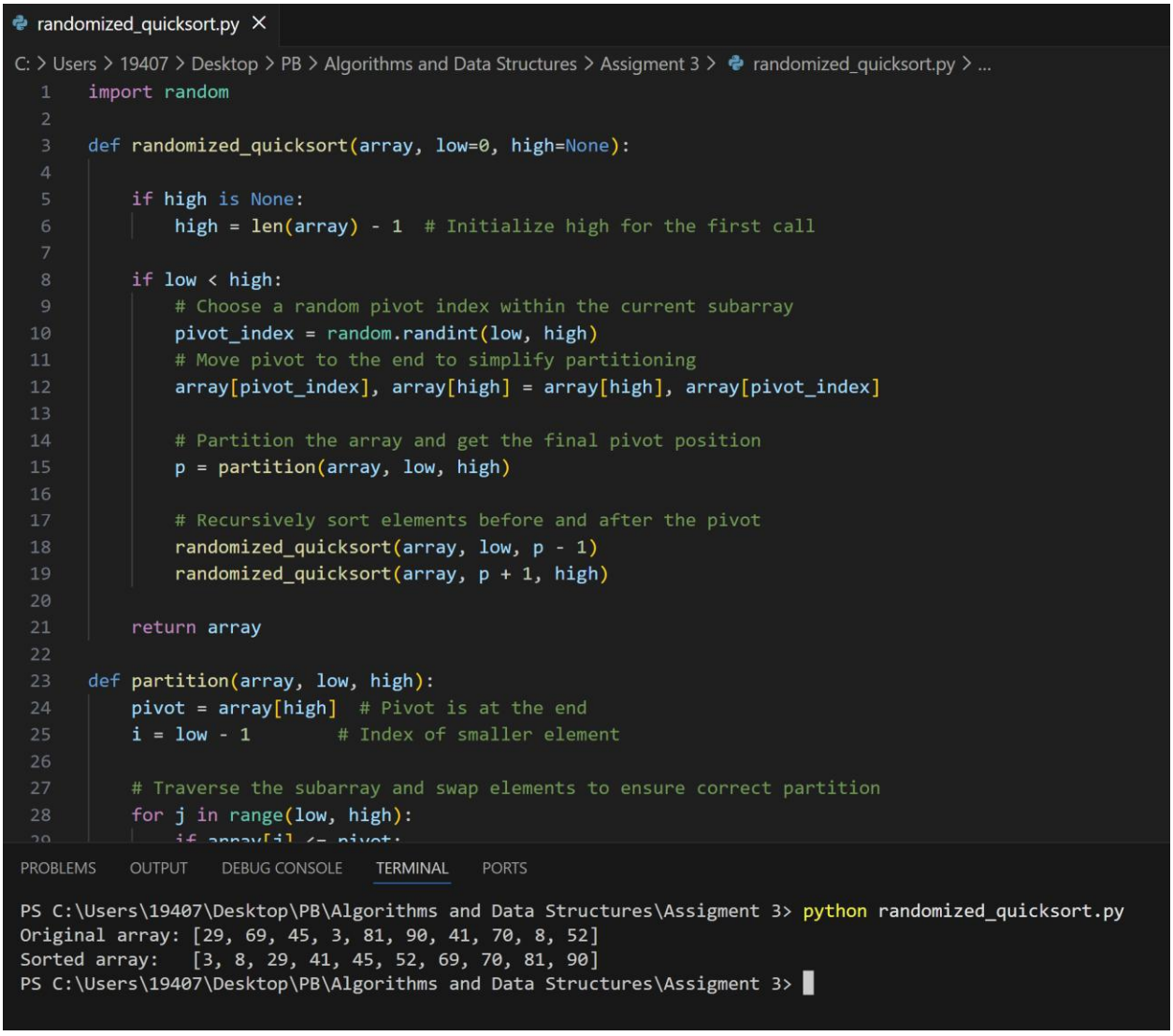
- Randomized Quicksort provides robust, scalable sorting across different input types, achieving O(n log n) average-case complexity.
- Hash Table with Chaining efficiently supports dynamic key operations and handles collisions effectively.
- The combination of theoretical and empirical analysis confirms that both algorithms perform as expected, illustrating the practical benefits of careful algorithm design.

GitHub Repository:

https://github.com/pranavibalakulla/MSCS532_Assignment3_Algorithm-Efficiency-Scalability/tree/main

Implementation Output:

Randomized Quicksort



Hash Table with Chaining