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: All elements with values less than the pivot are placed to its left, while elements with values greater than the pivot are placed to its right.
- 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 robustly handles empty arrays, single-element arrays, repeated elements, and pre-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 subset containing k elements smaller than the pivot.
- One subset containing $n - k - 1$ elements larger than the pivot.

The partitioning step requires $\Theta(n)$ 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 an average because of recursive stack calls.

3. Empirical Comparisons:

To assess practical performance, Randomized Quicksort was tested against Deterministic Quicksort, which uses the first element as the pivot, across various input sizes and data 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

The running time was tracked in milliseconds and calculated as the average over multiple trials.

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 high performance for all input types.
- Deterministic Quicksort suffers on already sorted or reverse-sorted inputs due to unbalanced partitioning.
- Experimental observations closely match the theoretical behavior predicted 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.

Assuming a simple uniform hashing model, in which each key has an equal probability of being assigned to any bucket, the expected time complexities are as follows.

- 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 highlights how algorithm design choices significantly affect performance and scalability.

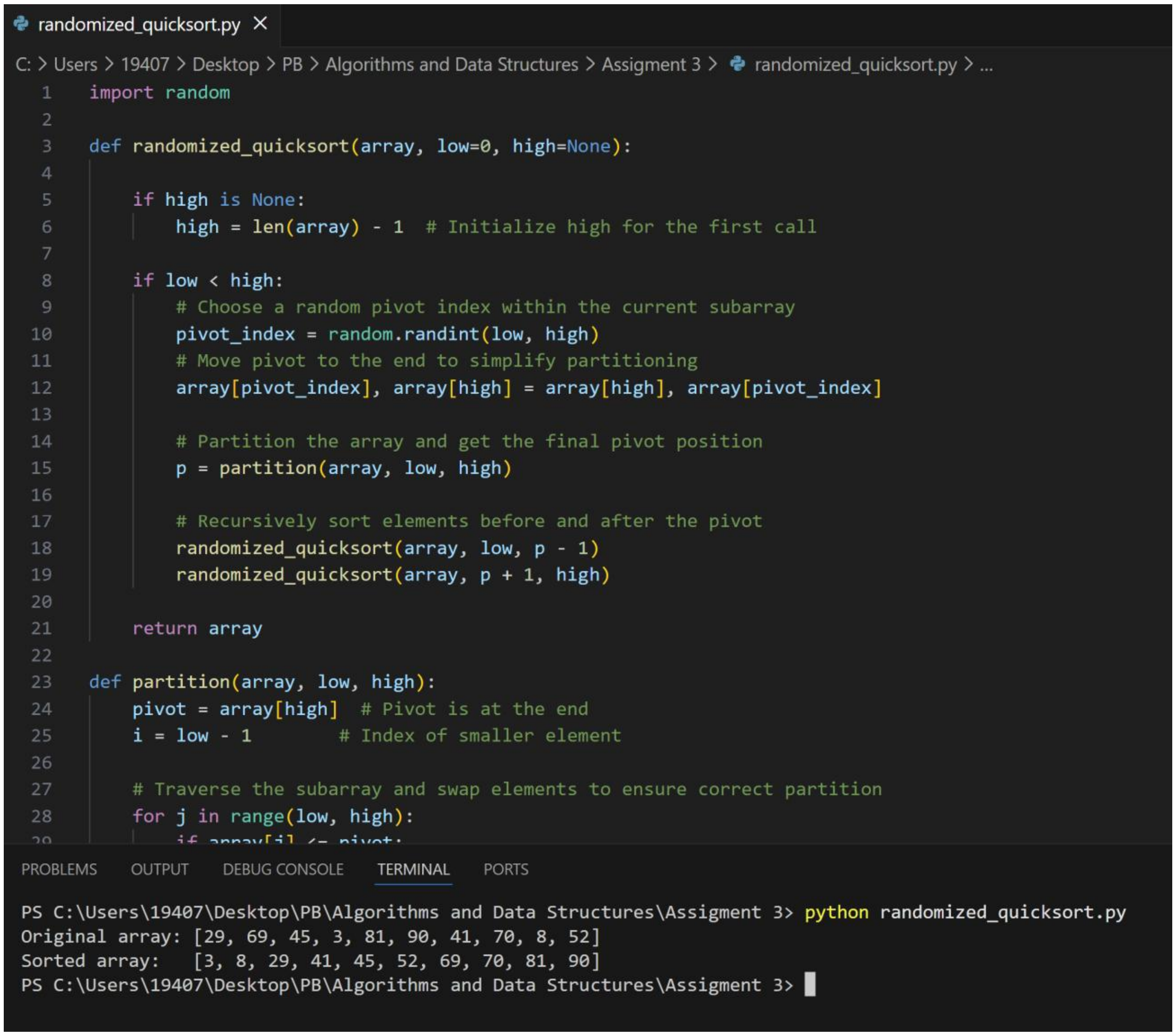
- Randomized Quicksort offers robust and scalable sorting performance across diverse input types, achieving an average-case time complexity of O(n log n).
- Hash Table with Chaining efficiently supports dynamic key operations and handles collisions effectively.
- The combination of theoretical and empirical 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


randomized_quicksort.py

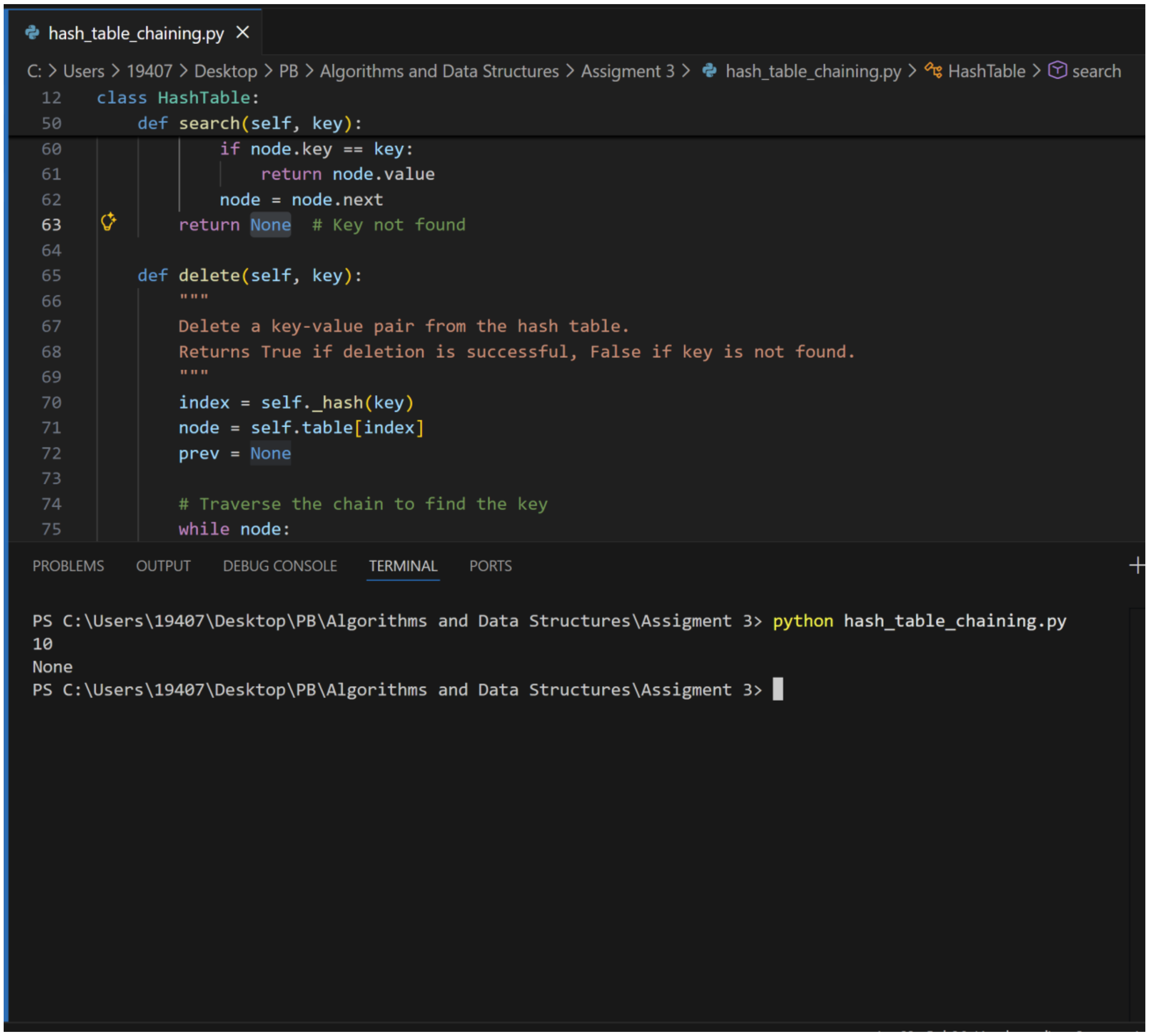
C: > Users > 19407 > Desktop > PB > Algorithms and Data Structures > Assigment 3 > randomized_quicksort.py > ...

```python
import random

def randomized_quicksort(array, low=0, high=None):

    if high is None:
        high = len(array) - 1  # Initialize high for the first call

    if low < high:
        # Choose a random pivot index within the current subarray
        pivot_index = random.randint(low, high)
        # Move pivot to the end to simplify partitioning
        array[pivot_index], array[high] = array[high], array[pivot_index]

        # Partition the array and get the final pivot position
        p = partition(array, low, high)

        # Recursively sort elements before and after the pivot
        randomized_quicksort(array, low, p - 1)
        randomized_quicksort(array, p + 1, high)

    return array

def partition(array, low, high):
    pivot = array[high]  # Pivot is at the end
    i = low - 1        # Index of smaller element

    # Traverse the subarray and swap elements to ensure correct partition
    for j in range(low, high):
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
PS C:\Users\19407\Desktop\PB\Algorithms and Data Structures\Assigment 3> python randomized_quicksort.py
Original array: [29, 69, 45, 3, 81, 90, 41, 70, 8, 52]
Sorted array:   [3, 8, 29, 41, 45, 52, 69, 70, 81, 90]
PS C:\Users\19407\Desktop\PB\Algorithms and Data Structures\Assigment 3>
```

Hash Table with Chaining



hash_table_chaining.py

C: > Users > 19407 > Desktop > PB > Algorithms and Data Structures > Assigment 3 > hash_table_chaining.py > HashTable > search

```python
class HashTable:
    def search(self, key):
                if node.key == key:
                    return node.value
                node = node.next
            return None  # Key not found

    def delete(self, key):
        """
        Delete a key-value pair from the hash table.
        Returns True if deletion is successful, False if key is not found.
        """
        index = self._hash(key)
        node = self.table[index]
        prev = None

        # Traverse the chain to find the key
        while node:
```

```
PS C:\Users\19407\Desktop\PB\Algorithms and Data Structures\Assigment 3> python hash_table_chaining.py
10
None
PS C:\Users\19407\Desktop\PB\Algorithms and Data Structures\Assigment 3>
```