UNDERSTANDING ALGORITHM EFFICIENCY AND SCALABILITY

Assignment 3

JULY 16, 2025

BISHNU SHARMA
ID: 005036007

Introduction

This report explores and analyzes two fundamental algorithmic techniques named Randomized Quicksort and Hashing with Chaining. These algorithms are widely used in software development and data processing because of their efficient performance in sorting and searching tasks. The goal of this assignment was to implement both algorithms in Python, analyze their theoretical efficiency, and compare how they perform in different scenarios.

Part 1.1: Randomized Quicksort

Implementation Overview

Randomized Quicksort is a variation of the standard Quicksort algorithm, where the pivot element is selected randomly instead of deterministically. This adjustment helps avoid worst-case performance on sorted or reverse-sorted arrays. The implementation uses recursion, dividing elements into those less than or equal to the pivot and those greater than it. This method handles various input types, including duplicates and unbalanced data, quite well.

Part 1.2: Deterministic Quicksort

Implementation Overview

In the deterministic version of Quicksort, the pivot is always chosen as the first element of the array. This approach is simple to implement but can lead to poor performance if the input is already sorted or nearly sorted. In such cases, the partitions become unbalanced, leading to increased recursion depth and more comparisons. While deterministic Quicksort has the same average-case time complexity as the randomized version, its worst-case time complexity is $O(n^2)$, which occurs more frequently when the pivot selection is not ideal (Cormen et al., 2009).

Time Complexity Analysis

The average-case time complexity of Randomized Quicksort is O(nlogn). This result comes from analyzing the expected number of comparisons made during sorting. Since the pivot is chosen randomly, the likelihood of balanced partitioning is high. The recurrence relation for the expected time is:

$$T(n) = \frac{1}{n}\sum_{k=0}^{n-1}(T(k) + T(n-k-1)) + cn$$

Solving this leads to O(nlogn) using recursive analysis or expected value techniques (Sedgewick & Wayne, 2011). On average, the algorithm performs well across diverse inputs and scales efficiently for larger arrays.

Empirical Comparison

For performance comparison, a deterministic Quicksort was implemented with the first element consistently chosen as the pivot. We then tested both algorithms on arrays with random elements, already sorted elements, reverse-sorted elements, and arrays with repeated values. The randomized version performed more consistently across all test cases. In contrast, the deterministic version was slower on sorted and reverse-sorted arrays, confirming that its fixed pivot choice leads to poor partitioning in those cases.

The table below shows the actual running times (in milliseconds) for each version when sorting arrays of size 10,000. These measurements were obtained using Python's built-in time module. The results show that Randomized Quicksort consistently handles sorted and reversed inputs efficiently, while Deterministic Quicksort performs poorly in those cases due to unbalanced

partitions. Interestingly, Randomized Quicksort was much slower on arrays with repeated elements, likely due to recursive calls on nearly identical partitions.

Input type	Size	Deterministic Quicksort (ms)	Randomized Quicksort (ms)
Random Array	10000	17.47	25.26
Already sorted	10000	3892.9	26.24
Reverse sorted	10000	4147.0	19.61
Repeated elements	10000	1.02	5031.07

These results validate the theoretical understanding that pivot choice has a significant impact on partition quality, especially for structured or uniform data.

Part 2: Hashing with Chaining

Implementation Overview

The implementation of the hash table employs chaining as a collision resolution strategy. Every slot in the hash table contains a list of associated key-value pairs. A simple modulo-based hash function was used:

```
hash(key) % table_size
```

The table supports insertion, lookup, and deletion of key-value pairs.

Time Complexity and Load Factor

When simple uniform hashing is assumed, the expected time complexity for insertion, lookup, and deletion is $O(1+\alpha)$, where α is the load factor defined as the ratio of elements to the table's size. A lower load factor means fewer collisions and shorter chains, resulting in faster operations (Goodrich et al., 2014).

To keep performance high, it's important to monitor the load factor and resize the table when it becomes too large. The process involves allocating a larger table and re-inserting all current elements using the updated hash function. In practice, resizing is triggered when the load factor exceeds a threshold, such as 0.75.

Conclusion

This assignment demonstrated how algorithm design choices impact performance in practical applications. Randomized Quicksort, with its use of random pivot selection, showed better consistency and scalability compared to a fixed-pivot version. The hash table with chaining provided efficient support for insertion, lookup, and deletion, while highlighting the importance of load factor and collision handling. Overall, both implementations reinforced the importance of analyzing algorithm efficiency from both theoretical and empirical perspectives.

References

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms (3rd ed.). MIT Press.

Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2014). Data Structures and Algorithms in Python. Wiley.

Sedgewick, R., & Wayne, K. (2011). Algorithms (4th ed.). Addison-Wesley.