# **Assignment 3: Understanding Algorithm Efficiency and Scalability**

Jacob Jeffers

MSCS 532

Dr. Vanessa Cooper

Fall 2024

11/10/2024

Assignment 3: Understanding Algorithm Efficiency and Scalability

Part 1: Randomized Quicksort Analysis

Implementation: For Part 1, I had previously worked on a Quicksort Algorithm, so for this assignment, I could use a lot of my previous work and then add to it. The code is set to handle edge cases with repeated elements, empty arrays, and already sorted arrays.

A screenshot of a computer program

Description automatically generated

Analysis: The average-case time complexity of randomized quicksort is O(nlogn), achieved by selecting a random pivot at each step, which balances the partitioning on average. This recursive process divides the array into two subarrays at each step, requiring O(n) work per level for partitioning. Since the array is likely split approximately in half each time, we have around logn levels of recursion. This leads to the recurrence T(n)=2T(n/2)+O(n), which solves to O(nlogn) using the Master Theorem (Cormen et al., 2022). Thus, randomized quicksort efficiently maintains an average complexity of O(nlogn) by reducing the likelihood of unbalanced partitions (Cormen et al., 2022).

Comparison: For this portion of the assignment, I had to work with my code to compare the Deterministic Quicksort with the Randomized Quicksort algorithm to determine which method is faster. I created code that tests up to 1000 recursions (the depth limit in VS Code). I arrived at this number because any more recursions would throw an error. The results indicate that deterministic quicksort algorithms do very well with random or shuffled listed, while the randomized quicksort performs better for specific kinds of input.

A computer screen shot of a program

Description automatically generated

Part 2: Hashing with Chaining

Implementation: To implement a hash table with chaining, I used a list of lists called a bucket, where each bucket can hold multiple key-value pairs. In the event of a collision, the pairs are stored in a list within that bucket. It is a fundamental hash function implementation that uses an Insert, Search, and Delete operation. Doing it this way prevents data from being lost because even when multiple keys hash the same index, we can apply these to a list within a particular bucket.

A screen shot of a computer

Description automatically generated

Analysis: To get actual-time data for the Search, Insert, and Delete operations in a hash table, I created a script that measures the execution time under different conditions. I started with the hash table created in the first part of the assignment. I added a time measurement to the Python code. I created a benchmark function that performs different operations and calculates the total time taken for each over a certain number of operations. The screenshot below illustrates the time over 10, 50, 100, 500, and 1000 operations. The insert time shows how long It takes to insert elements as they grow in size. The search time measures how long it takes to find an element that has already been inserted. The delete time measures how long it takes to remove an element. A screenshot of a computer

Description automatically generated

For insert time, as the size of the hash table increases from 10 to 1000, the insert time decreases. This suggests that, for small hash table sizes, the overhead of handling fewer elements might take slightly more time due to the initialization and management of the table. For sizes 10 and 50, insertions take longer (in the range of 0.001 to 0.0006 seconds), but as the size increases to 100 and beyond, the insert time decreases further (around 0.0004 to 0.0005 seconds). This could be because the hash table is more balanced or there is less overhead relative to the operations.

The search time follows a similar pattern. It is highest at size 10 (0.0015 seconds), then decreases slightly as the table size grows, reaching around 0.00035 seconds at size 1000.

This could be because the average load per bucket decreases with more entries in the hash table. As the size increases, search operations may be faster due to fewer collisions and better distribution of keys across buckets.

The delete time is consistently low (under 0.001 seconds) and does not change significantly as the table size increases. This suggests that deletions are relatively fast and efficient, possibly because, even with larger tables, deletion only involves searching for the key and removing it.

GitHub Repository for Assignment 3: <https://github.com/jakejeffers/MSCS-532-Assignment-3>

References

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). *Introduction to Algorithms, fourth edition*. MIT Press.