**Name: AJAL RC**

**Student ID: 005039456**

**Algorithms and Data Structures (MSCS-532-B01)**

**Assignment 2: Understanding Algorithm Efficiency and Scalability**

**Part 1: Randomized Quicksort Analysis**

* 1. **Implementation**

Randomized Quicksort is a version of Quicksort that picks a random pivot for each recursive call. This is important because it helps the algorithm avoid slow behavior on sorted or nearly sorted data. This is a problem that basic deterministic Quicksort suffers from. Here is the snapshot of the code that works in arrays with either repeated values, empty arrays, or sorted elements. A screen shot of a computer program

AI-generated content may be incorrect.

* 1. **Analysis**

The average-case time complexity of Randomized Quicksort is O(n log n). This means that, on average, it will sort an array of n elements in about n log n steps. This happens because picking a pivot at random prevents the algorithm from getting stuck, always making bad splits, which can happen with deterministic quicksort if the input is already sorted or reversed.

Using probability and a tool called indicator random variables, mathematicians have shown that the expected number of comparisons is O(n log n) (Sedgewick & Wayne, 2022).

The recurrence relation for the expected running time is T(n) = n + (1/n) \* sum of [T(k) + T(n-k-1)] for k = 0 to n-1, which solves to O(n log n) (Knuth, 1998).

* 1. **Comparison**

Testing was done on Randomized Quicksort and Deterministic Quicksort (which always picks the first element as the pivot) on random arrays, already sorted arrays, reverse-sorted arrays, and arrays with many repeated values.

The running time for each test was measured with Python’s time module. The results were printed in a table, and a screenshot is below. A screenshot of a computer program

AI-generated content may be incorrect.

**Observations**

* Both algorithms were fast on random arrays, with running times almost the same.
* Randomized Quicksort stayed fast on already sorted and reverse-sorted arrays, but Deterministic Quicksort became much slower or even crashed with a RecursionError for larger arrays greater than 1000. This is because the recursion depth reached Python’s limit due to always making the worst possible split.
* Both algorithms stayed fast on arrays with repeated elements because the splits were balanced enough.

The theory says that Randomized Quicksort should perform well on all input types, while Deterministic Quicksort may fail badly on sorted or reverse-sorted data. My results match this exactly. When deterministic quicksort hit a recursion error, this showed its theoretical worst-case in action, which is not a bug, but a fundamental limitation.

**Part 2: Hashing With Chaining**

* 1. **Implementation**

A hash table with chaining stores data in “buckets.” If two keys map to the same bucket, they are kept in a list (chain) so no data is lost. Here is the screenshot for the code.

A screen shot of a computer program

AI-generated content may be incorrect.

* 1. **Analysis**

In a hash table, chaining, searching, inserting, and deleting each take O(1) time on average, as long as the hash function spreads out the keys evenly and the table is not too full. The load factor (average number of elements per bucket) is important as operations slow down if it gets too high because the chains get longer.

To keep operations efficient, the table should be made bigger (resized and rehashed) when it gets too full. This process is called dynamic resizing or rehashing (Knuth, 1998; Williams, 1964).

Here is the sample output from the hash table demo file.

A black screen with white text

AI-generated content may be incorrect.

**Conclusion**

This assignment taught me that Randomized Quicksort is much safer and faster than Deterministic Quicksort when sorting unknown data. It avoids the “bad cases” that can cause a deterministic quicksort crash. The assignment also showed that hash tables with chaining are highly efficient if the load factor is kept low and the hash function is good. This practical experience confirmed what the theory says about both types of algorithms.

**References:**

Sedgewick, R., & Wayne, K. (2022). Algorithms (4th ed.). Addison-Wesley. <https://algs4.cs.princeton.edu/home/>

Williams, J. W. J. (1964). Algorithm 232: Heapsort. Communications of the ACM, 7(6), 347–348. <https://dl.acm.org/doi/10.1145/512274.3734138>

Knuth, D. E. (1998). The art of computer programming, volume 3: Sorting and searching (2nd ed.). Addison-Wesley.