Assignment 3: Understanding Algorithm Efficiency and Scalability

Gregory W. Renteria Villar

Student ID: 005039058

School of Computer and Information Sciences, University of the Cumberlands

MSCS-532-M20: Algorithms and Data Structures

Dr. Brandon Bass

June 15, 2025

Understanding Algorithm Efficiency and Scalability

Part 1: Randomized Quicksort Analysis

The average case time complexity of Randomized Quicksort is $O(n \log n)$, and this result can be rigorously demonstrated using indicator random variables and properties of expected values. The key idea is that the number of comparisons performed by the algorithm dominates its running time, so by analyzing the expected number of comparisons, we can determine the average-case time complexity.

To begin, consider an array of n distinct elements. During the execution of Randomized Quicksort, two elements a_i and a_j are compared only if one of them is chosen as a pivot before any element that lies between them in sorted order. To capture this idea, we define an indicator variable X_{ij} which is 1 if a_i and a_j are compared, and 0 otherwise. The total number of comparisons is then the sum of all X_{ij} over all $i < j$, and the expected number of comparisons is the sum of the expected values $E[X_{ij}]$.

The probability that a_i and a_j are compared is $\frac{2}{j-i+1}$, because this is the chance that either a_i or a_j is selected as the first pivot among the elements between them. Summing this over all pairs (i, j), we get the expected number of comparisons as $\sum_{1 \le i < j \le n} \frac{2}{j-i+1}$. This sum simplifies to $O(n \log n)$, since it resembles a harmonic series weighted by decreasing linear terms.

An alternative approach using recurrence relations confirms the result. If $T(n)$ is the expected time to sort an array of size n, then we can express it as $T(n) = n - 1 + \frac{1}{n}\sum_{i=1}^{n}[T(i-1) + T(n-i)]$, where the $n - 1$ term accounts for partitioning and the summation represents the recursive calls on the left and right partitions. This recurrence solves to $T(n) = O(n \log n)$.

Comparison

To empirically compare Randomized Quicksort and Deterministic Quicksort (which always uses the first element as the pivot, we tested both algorithms on arrays of varying sizes and characteristics. The goal was to observe their performance on different types of input: randomly generated arrays, already sorted arrays, reverse-sorted arrays, and arrays with repeated elements. Execution time was measured across increasing input sizes, ranging from 1,000 to 100,000 elements, with multiple runs averaged to reduce measurement noise.

On randomly generated arrays, both versions of Quicksort performed similarly well. Aas expected, the exhibited average case time complexity of $O(n \log n)$. In these cases, even the deterministic choice of the first element as pivot often resulted in reasonably balanced partitions, so no significant performance difference was observed.

However, for already sorted arrays and reverse-sorted arrays, the performance diverged significantly. Deterministic Quicksort was much slower, showing quadratic $O(n^2)$ behavior. This is because always selecting the first element as pivot leads to the worst case scenario: one partition is of size zero, and the other is of size $n - 1$, causing deep recursion and many redundant comparisons. In contrast, Randomized Quicksort maintained good performance, since the randomly selected pivot usually avoids such extreme imbalance, preserving the expected $O(n \log n)$ behavior.

Both algorithms performed relatively well when tested on arrays with many repeated elements, but Randomized Quicksort was typically faster. While repeated elements do not inherently lead to poor performance for Deterministic Quicksort, the fixed pivot choice can still lead to unbalanced splits, especially if the pivot value is overrepresented. Randomized Quicksort benefits from the probabilistic chance of choosing a pivot that helps divide duplicates more evenly,

particularly if a three-way partitioning scheme is used.

Part 2: Hashing with Chaining Analysis

Under the assumption of simple uniform hashing, each key is equally likely to be hashed to any slot in the table, independently of other keys. In such scenario, a hash table using chaining for collision resolution has expected constant time $O(1)$ for the search, insert, and delete operations. This is because, on average, each slot or "bucket" will contain only a small number of elements if the table is not overloaded. More formally, the expected time for these operations is $O(1 + \alpha)$, where α is the load factor, defined as $\alpha = \frac{n}{m}$, with n being the number of elements stored and m the number of slots.

The load factor is critical in determining performance. When α is low (much less than 1), most buckets are empty or contain only one element, so operations are nearly constant time. However, as α increases (i.e., more elements are added without increasing the table size), the average length of the chains grows linearly with α, and the time complexity for search and delete operations degrades toward $O(\alpha)$.Insertions remain efficient since they usually involve appending to a list, but even that can become slower if the chains are long.

To maintain efficient performance, it's important to keep the load factor low. One common strategy is dynamic resizing: when the number of elements exceeds a threshold (e.g., when $\alpha > 0.75$), the table is resized, and all existing elements are rehashed into the new table. This process is expensive, but because it happens infrequently, the amortized cost of maintaining constant-time operations remains low. Additionally, using a good hash function, such as one from a universal hashing family, helps distribute keys uniformly across the table and minimizes the likelihood of collisions, further preserving the expected $O(1)$ time per operation.

Screenshots

Randomized Quicksort

```
Original: []
Sorted:   []

Original: [1]
Sorted:   [1]

Original: [3, 1, 2]
Sorted:   [1, 2, 3]

Original: [5, 5, 5, 5]
Sorted:   [5, 5, 5, 5]

Original: [1, 2, 3, 4, 5]
Sorted:   [1, 2, 3, 4, 5]

Original: [9, 7, 5, 3, 1]
Sorted:   [1, 3, 5, 7, 9]

Original: [10, -1, 2, 5, 0, 6, 4]
Sorted:   [-1, 0, 2, 4, 5, 6, 10]
```

Hashing with Chaining

```
Search apple: 5
Search grape: None
Updated apple: 10
After deleting banana: None

Full Table:
 [[], [], [('orange', 8)], [], [], [], [], [], [], [], [('apple', 10)], [], [], [], [], [], []]
```