

Assignment 3: Understanding Algorithm Efficiency and Scalability

Nitish Dhinakaran

University of the Cumberland

MSCS-532-M20 Algorithms and Data Structures

Dr. Brandon Bass

June 15, 2025

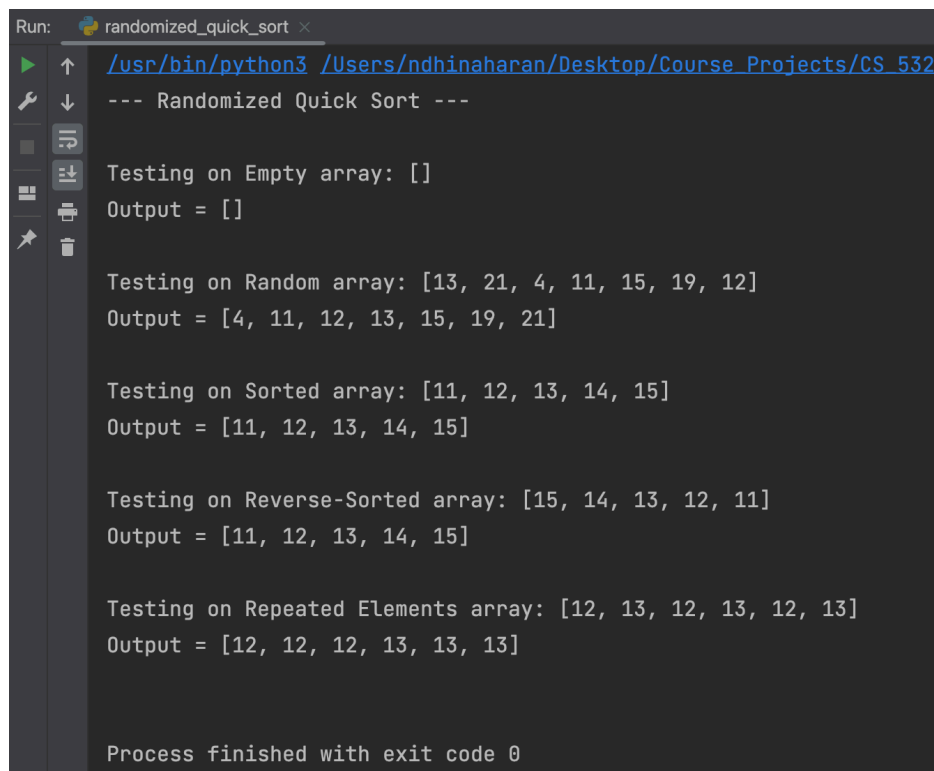
Github Link to Source Code of Implementation

<https://github.com/ndhinaharan36295/MSCS532-Assignment3>

Randomized Quick Sort

Implementation and Execution

I implemented the randomized quicksort algorithm in Python. This algorithm selects a pivot element uniformly at random from the current subarray during each recursive call. This randomness in pivot selection helps reduce the chance of encountering worst-case time complexity on specific input patterns, making the algorithm more robust in practice (Cormen et al., 2022). As it can be seen in Figure 3.1, my implementation executes the sorting and correctly returns the sorted output. It can also be seen that it handles all the necessary edge cases such as empty arrays, already sorted arrays, reverse-sorted arrays and arrays with repeated elements.



```
Run: randomized_quick_sort x
/usr/bin/python3 /Users/ndhinaharan/Desktop/Course Projects/CS_532
--- Randomized Quick Sort ---

Testing on Empty array: []
Output = []

Testing on Random array: [13, 21, 4, 11, 15, 19, 12]
Output = [4, 11, 12, 13, 15, 19, 21]

Testing on Sorted array: [11, 12, 13, 14, 15]
Output = [11, 12, 13, 14, 15]

Testing on Reverse-Sorted array: [15, 14, 13, 12, 11]
Output = [11, 12, 13, 14, 15]

Testing on Repeated Elements array: [12, 13, 12, 13, 12, 13]
Output = [12, 12, 12, 13, 13, 13]

Process finished with exit code 0
```

Figure 3.1 Randomized Quicksort

Analysis

The average-case time complexity of randomized quicksort is $O(n \cdot \log n)$. We can analyze and prove this using recurrence relations. As it can be seen in Figure 3.2, is the proof of the analysis using recurrence relations that I created the proof using LaTeX (Shil, 2019).

Let $T(n)$ be the expected time complexity on an input of size n .

The pivot splits the array into subarrays of size i and $n - i - 1$ with equal probability for all $i \in [0, n - 1]$.

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

here cn represents the cost of partitioning the array, which is linear n .

Using symmetry and changing indices:

$$T(n) = cn + \frac{2}{n} \sum_{i=0}^{n-1} T(i)$$

Proof by substitution method

Assume the hypothesis:

$$T(i) \leq ai \log i \quad \text{for all } i < n$$

We want to prove:

$$T(n) \leq an \log n$$

Substitute the hypothesis into the recurrence:

$$T(n) \leq cn + \frac{2a}{n} \sum_{i=1}^{n-1} i \log i$$

$$\sum_{i=1}^{n-1} i \log i = O(n^2 \log n) \Rightarrow \frac{1}{n} \sum_{i=1}^{n-1} i \log i = O(n \log n)$$

Therefore:

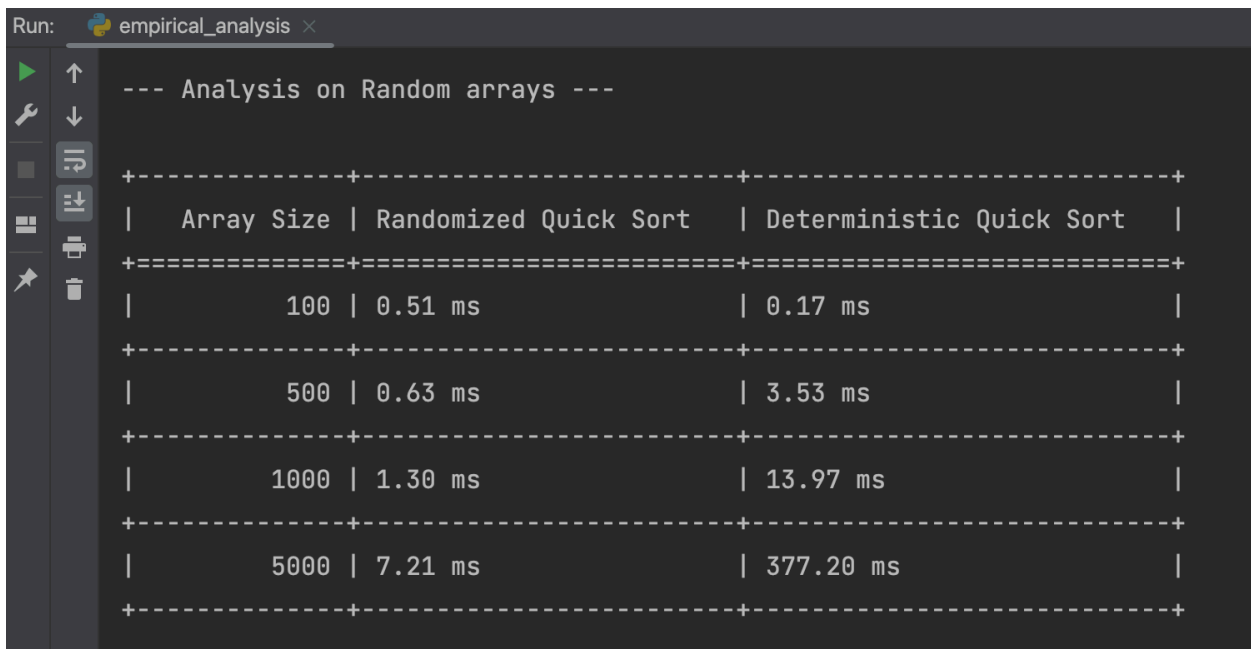
$$T(n) \leq cn + 2a \cdot O(n \log n) = O(n \log n)$$

Figure 3.2 Analysis of average-case time complexity

Empirical Comparison

I implemented the deterministic quicksort algorithm and ran both of the algorithms with different types of inputs (random arrays, sorted arrays, reverse-sorted arrays and array with repeated elements) and input sizes (100, 500, 1000, 5000) to perform comparisons of the runtimes of these algorithms and how the runtime varies based on different inputs and different input sizes. I used Python's in-built tabulate package to showcase the runtimes which would make it easier to compare and analyse.

Firstly, I ran both of the algorithms on random arrays with the above mentioned different input sizes. As it can be seen in Figure 3.3, for random arrays, randomized quicksort consistently outperforms deterministic quicksort (whose runtime increases exponentially as the input size increases). The deterministic algorithm performs poorly due to consistently choosing the first element as pivot which leads to highly unbalanced recursive calls and $O(n^2)$ runtime complexity (Cormen et al., 2022).

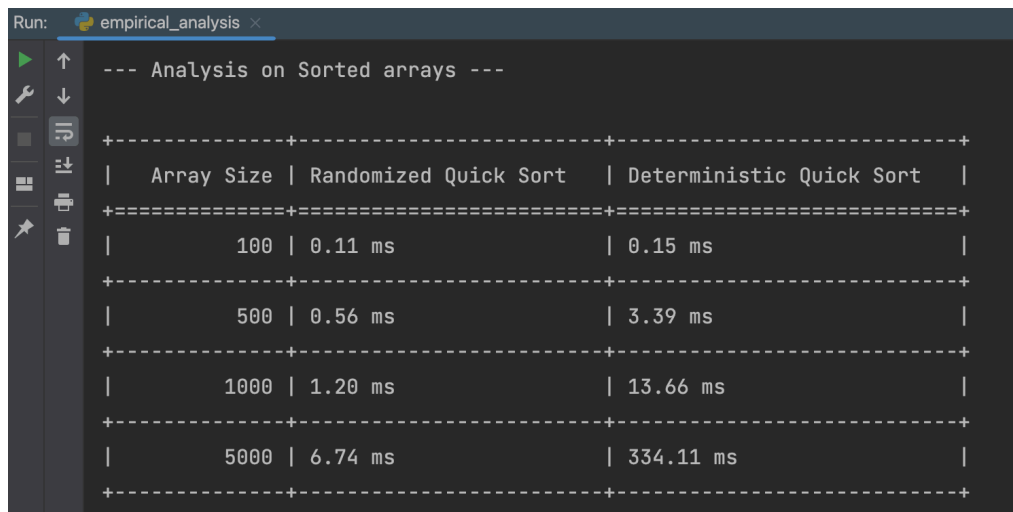


```
Run: empirical_analysis x
--- Analysis on Random arrays ---
+-----+-----+-----+
| Array Size | Randomized Quick Sort | Deterministic Quick Sort |
+-----+-----+-----+
|      100 | 0.51 ms                | 0.17 ms                  |
+-----+-----+-----+
|      500 | 0.63 ms                | 3.53 ms                  |
+-----+-----+-----+
|     1000 | 1.30 ms                | 13.97 ms                 |
+-----+-----+-----+
|     5000 | 7.21 ms                | 377.20 ms                |
+-----+-----+-----+
```

Array Size	Randomized Quick Sort	Deterministic Quick Sort
100	0.51 ms	0.17 ms
500	0.63 ms	3.53 ms
1000	1.30 ms	13.97 ms
5000	7.21 ms	377.20 ms

Figure 3.3 Runtime analysis on random arrays

Next, I ran the algorithms on sorted arrays with different input sizes. As it can be seen in Figure 3.4, the performance of each algorithm on this data set of sorted arrays is very similar to the performance in random arrays as well – where randomized quicksort massively outperformed deterministic quicksort, as the input sizes grew.



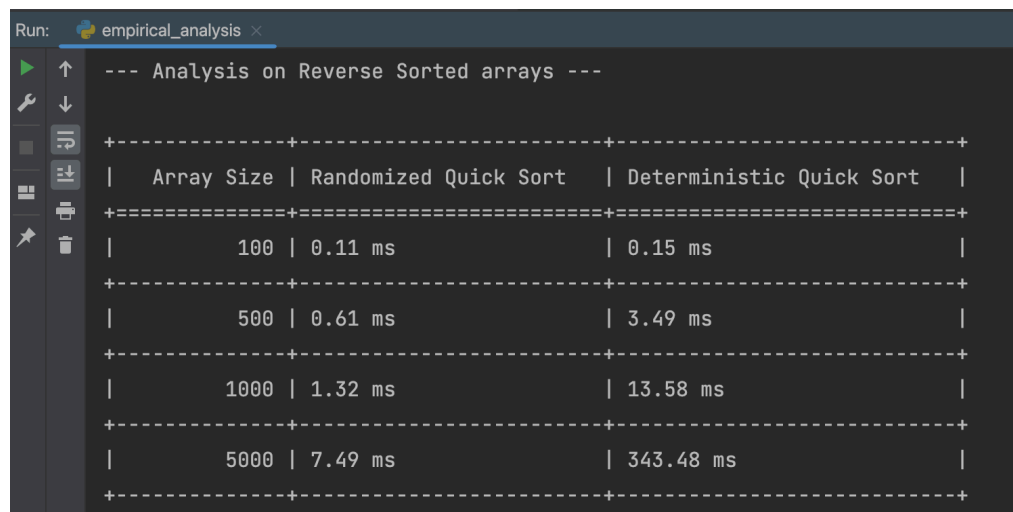
```

Run: empirical_analysis x
--- Analysis on Sorted arrays ---
+-----+-----+-----+
| Array Size | Randomized Quick Sort | Deterministic Quick Sort |
+-----+-----+-----+
|      100 | 0.11 ms                | 0.15 ms                  |
+-----+-----+-----+
|      500 | 0.56 ms                | 3.39 ms                  |
+-----+-----+-----+
|     1000 | 1.20 ms                | 13.66 ms                 |
+-----+-----+-----+
|     5000 | 6.74 ms                | 334.11 ms                |
+-----+-----+-----+

```

Figure 3.4 Runtime analysis on sorted arrays

Next, I ran the algorithms on reverse sorted arrays with different input sizes. As it can be seen in Figure 3.5, the performance for each of the algorithms was, again, very similar to that of sorted arrays.



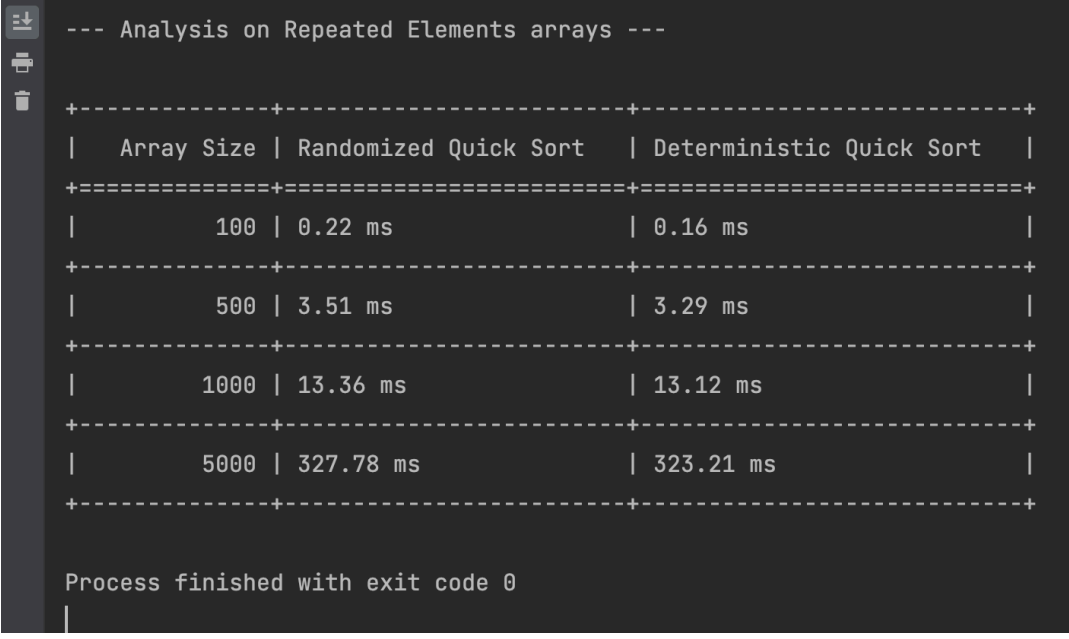
```

Run: empirical_analysis x
--- Analysis on Reverse Sorted arrays ---
+-----+-----+-----+
| Array Size | Randomized Quick Sort | Deterministic Quick Sort |
+-----+-----+-----+
|      100 | 0.11 ms                | 0.15 ms                  |
+-----+-----+-----+
|      500 | 0.61 ms                | 3.49 ms                  |
+-----+-----+-----+
|     1000 | 1.32 ms                | 13.58 ms                 |
+-----+-----+-----+
|     5000 | 7.49 ms                | 343.48 ms                |
+-----+-----+-----+

```

Figure 3.5 Runtime analysis on reverse sorted arrays

Finally, I ran the algorithms on arrays with repeated elements. As it can be seen in Figure 3.6, both Randomized and Deterministic quicksort perform similarly on arrays with repeated elements. This is due to the presence of duplicate values, leading to minimal gains from pivot selection. Partitions remain unbalanced regardless of pivot strategy, causing comparable degradation in performance as input size grows (Cormen et al., 2022).



```

--- Analysis on Repeated Elements arrays ---

+-----+-----+-----+
| Array Size | Randomized Quick Sort | Deterministic Quick Sort |
+=====+=====+=====+
|      100 | 0.22 ms                | 0.16 ms                  |
+-----+-----+-----+
|      500 | 3.51 ms                | 3.29 ms                  |
+-----+-----+-----+
|     1000 | 13.36 ms               | 13.12 ms                 |
+-----+-----+-----+
|     5000 | 327.78 ms              | 323.21 ms                |
+-----+-----+-----+

Process finished with exit code 0

```

Array Size	Randomized Quick Sort	Deterministic Quick Sort
100	0.22 ms	0.16 ms
500	3.51 ms	3.29 ms
1000	13.36 ms	13.12 ms
5000	327.78 ms	323.21 ms

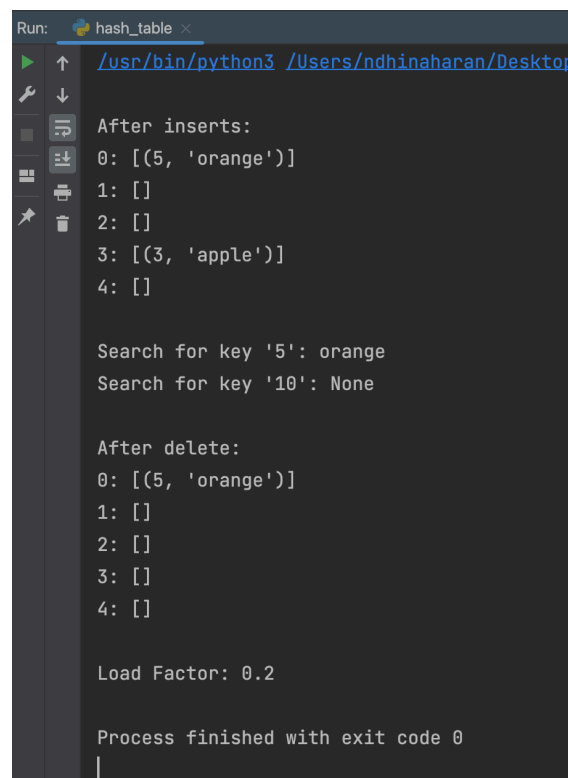
Figure 3.6 Runtime analysis on arrays with repeated elements

Overall, from the above results, we can see that the runtimes of the algorithms on the experimental data supports the theoretical complexity analysis, with discrepancies only due to input characteristics (e.g., presence of duplicates). Randomized quicksort performed in the expected average-time complexity similar to of $O(n * \log n)$, while Deterministic quicksort can degrade to $O(n^2)$ due to poor pivot choices.

Hashing with Chaining

Implementation

I implemented a hash table using chaining to handle collisions. The table consists of an array of buckets, where each bucket is a list of key-value pairs. A hash function maps each key to an index in the array, and any collisions are resolved by storing multiple elements in the corresponding bucket. The implementation supports the following operations: insert (adds or updates a key-value pair), search (retrieves the value associated with a key), delete (removes a key-value pair from the table, if it exists, based on the given key). To ensure scalability, I implemented dynamic resizing — when the load factor exceeds 0.75, the table size is doubled and all elements are rehashed (Cormen et al., 2022). Figure 3.7 shows the execution of some example operations of insert, search and delete.

A screenshot of a terminal window titled 'hash_table'. The terminal shows the output of a Python script. It starts with 'After inserts:' followed by a dictionary representation of a hash table with 5 buckets. Bucket 0 contains [(5, 'orange')], bucket 3 contains [(3, 'apple')], and the others are empty. Then it shows 'Search for key '5': orange' and 'Search for key '10': None'. Next is 'After delete:' followed by the same dictionary, but bucket 3 is now empty. Finally, it shows 'Load Factor: 0.2' and 'Process finished with exit code 0'.

```
Run: hash_table x
/usr/bin/python3 /Users/ndhinaharan/Desktop/...

After inserts:
0: [(5, 'orange')]
1: []
2: []
3: [(3, 'apple')]
4: []

Search for key '5': orange
Search for key '10': None

After delete:
0: [(5, 'orange')]
1: []
2: []
3: []
4: []

Load Factor: 0.2

Process finished with exit code 0
```

Figure 3.7 Hash table operations

Time complexity analysis and Load Factor affecting performance

Time complexity analysis of hash table operations (assuming simple uniform hashing):

Let, n = number of elements, m = number of buckets, $\alpha = n/m$ (load factor).

Then, the expected time complexity of the basic operations is:

$\text{insert}() = O(1+\alpha)$, $\text{search}() = O(1+\alpha)$ and $\text{delete}() = O(1+\alpha)$

Under the assumption of simple uniform hashing—where each key is equally likely to hash into any bucket—collisions are evenly distributed, and average bucket length is proportional to α — load factor (Cormen et al., 2022).

The load factor α directly affects performance of the hashing operations. If $\alpha \leq 1$, operations remain close to constant time. As α grows (due to more insertions without resizing), performance degrades because more elements accumulate in each bucket (Cormen et al., 2022).

Strategies to maintain low load factor

One of the main strategies to maintain a low load factor is dynamic hashing. To maintain efficient operations, we resize the table by doubling its size when $\alpha > 0.75$. On resizing, the number of buckets is typically doubled, and all existing elements are rehashed into the new table. This keeps α low and distributes keys more evenly (Cormen et al., 2022).

An important strategy to minimize collisions is choosing an effective hash function. This ensures uniform distribution of the elements. A good hash function distributes keys uniformly across buckets, minimizing the chance of collisions (Cormen et al., 2022).

References

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). *Introduction to algorithms*.

The Mit Press.

Shil, S. (2019). *Lecture 8: Randomized Quick-Sort*. University at Buffalo. Retrieved from

<https://cse.buffalo.edu/~shil/courses/CSE632-Fall2019/Notes/8-Quicksort.pdf>