# Assignment 5: Quicksort Algorithm: Implementation, Analysis, and Randomization

Ronit Pawar

005040714

University of Cumberlands

Algorithms and Data Structures

Satish Penmatsa

16 October 2025

## 1. Implementation Overview & Output

The Quicksort implementation developed for this experiment includes two primary versions: a deterministic Quicksort and a randomized Quicksort. Both algorithms were tested using identical datasets of various input sizes (1,000, 5,000, and 10,000 elements) under three different input orderings: random, sorted, and reverse-sorted to empirically observe performance differences. The timing results were measured using Python’s time library.

**Output:**

A screenshot of a computer

AI-generated content may be incorrect.

## 2. Analysis

The Quicksort algorithm, first introduced by Tony Hoare in 1962, is one of the most efficient general-purpose sorting algorithms due to its divide-and-conquer strategy and in-place partitioning. In this implementation, both the deterministic and randomized versions follow the standard Quicksort paradigm of selecting a pivot, partitioning the array into two subarrays based on the pivot, and recursively sorting each part.

### 2.1 Theoretical Time Complexity

**Best Case: O(n log n)**

The best-case performance occurs when the pivot divides the array into two nearly equal subarrays at every recursive step. In this scenario, the recursion tree has height log n, and each level performs O(n) work to partition the elements around the pivot. Hence, the total time complexity is:

For example, when sorting random inputs of size 10,000, the deterministic Quicksort completed in approximately 0.00817 seconds, while the randomized version required 0.01048 seconds. Both demonstrate efficient partitioning and logarithmic recursion depth, validating the O(n log n) performance under balanced splits.

**Average Case: O(n log n)**

In the average case, Quicksort assumes that the pivot is equally likely to be any of the n elements. This yields an expected balanced partition on average, even though some individual partitions may be uneven. The recurrence of expected time complexity can be expressed as:

Solving this recurrence gives **O(n log n)** for the expected runtime. The empirical data strongly supports this. For randomized inputs, runtimes grew approximately linearly with n log n:

* 1,000 elements → 0.00083 s
* 5,000 elements → 0.00467 s
* 10,000 elements → 0.01048 s

This proportional growth confirms the theoretical scaling relationship, as doubling n increases runtime by slightly more than a factor of two, consistent with the logarithmic term’s modest influence.

**Worst Case: O(n²)**

The worst-case scenario occurs when the pivot consistently divides the array extremely unevenly, such as always selecting the smallest or largest element. In this case, the recursion tree becomes degenerate (height n), and the recurrence simplifies to:

This scenario manifests most clearly with deterministic Quicksort on already sorted or reverse-sorted data, where the last element (chosen as pivot) always yields a poor partition.  
Empirical evidence from this implementation shows the quadratic explosion clearly:

|  |  |  |  |
| --- | --- | --- | --- |
| Input Type | n = 1000 | n = 5000 | n = 10000 |
| Sorted Input (Deterministic) | 0.02402s | 0.64329s | 2.68001s |
| Reverse-Sorted Input (Deterministic) | 0.01703s | 0.36874s | 1.30224s |

These results show that runtime increases roughly with the square of input size. E.g., increasing from 1,000 to 10,000 elements increases time by more than **100×** approximately, closely matching the O(n²) prediction.

### 2.2 Effect of Randomization

The randomized pivot strategy effectively eliminates deterministic bias in partitioning. By randomly selecting a pivot at each recursive step, the algorithm ensures that the probability of repeatedly poor partitions is exceedingly low. Mathematically, random pivot selection leads to an expected recurrence identical to the average case:

Empirical evidence supports this: even for sorted and reverse-sorted arrays of 10,000 elements, randomized Quicksort maintained runtimes around 0.010 seconds, in stark contrast to the multi-second durations of deterministic Quicksort. This demonstrates that randomization stabilizes performance across all input distributions, yielding a consistent and predictable runtime profile.

### 2.3 Space Complexity and Overheads

Quicksort’s in-place partitioning minimizes memory usage, requiring only O(1) auxiliary space for pivot and index variables. The main space overhead arises from recursive stack depth:

* **Average case:** Balanced partitions produce recursion depth of O(log n).
* **Worst case:** Highly unbalanced partitions (as in deterministic Quicksort on sorted input) yield depth O(n).

To mitigate this, a common optimization is recursing on the smaller partition first which is employed to limit maximum stack depth and prevent overflow. Compared to algorithms like Merge Sort, which requires O(n) additional memory for merging, Quicksort’s memory efficiency provides a major practical advantage despite its recursive nature.

## 3. Discussion and Interpretation

The results collectively confirm both the theoretical and practical properties of Quicksort:

1. **Deterministic Quicksort:** performs optimally (O(n log n)) on random data but deteriorates to O(n²) on sorted or reverse-sorted inputs.
2. **Randomized Quicksort:** maintains stable O(n log n) performance regardless of input distribution, demonstrating resilience against adversarial cases.
3. **Empirical scaling:** across input sizes (1,000 → 10,000) aligns with theoretical expectations, with runtime roughly proportional to n log n in non-adversarial cases.
4. **Space efficiency:** remains a strong advantage, as both implementations use minimal auxiliary memory beyond recursive stack space.

In conclusion, Quicksort exemplifies an elegant balance between theoretical efficiency and practical performance. The randomized variant, in particular, combines the low memory footprint of in-place sorting with statistically robust runtime guarantees, making it one of the most efficient and widely used sorting algorithms in both academic and industrial contexts.