# Project 8: Divide & ConquAAD

Empirical Analysis of Quicksort Variants
(Standard vs. Randomized vs. Dual-Pivot)

Laksh Mittal

December 1, 2025

## 1 Introduction

Sorting is a fundamental operation in computer science, serving as a building block for search algorithms, data compression, and complexity analysis. Among comparison-based sorting algorithms, **Quicksort** is widely regarded as the most efficient in practice due to its cache locality and low overhead.

However, the standard deterministic implementation suffers from a severe vulnerability: its performance degrades to $O(n^2)$ on sorted or reverse-sorted data. This project implements and empirically compares three variants to analyze their robustness and scalability:

1. **Standard Quicksort:** The textbook Lomuto partition scheme.
2. **Randomized Quicksort:** A Las Vegas algorithm that selects random pivots to defeat adversarial inputs.
3. **Dual-Pivot Quicksort:** An advanced variant (used in Java's `Arrays.sort`) that uses two pivots to reduce recursion depth.

## 2 Theoretical Background

### 2.1 Standard Quicksort (Lomuto)

Quicksort is a Divide and Conquer algorithm. The core operation is *Partitioning*. **Mechanism:** Given an array $A[low \dots high]$, the algorithm selects the last element $A[high]$ as the **pivot**. It rearranges the array such that all elements smaller than the pivot are to its left, and all elements greater are to its right. It then recursively sorts the sub-arrays.

**The Flaw:** If the array is already sorted, the pivot (last element) is always the maximum. The partition produces one sub-problem of size $n-1$ and one of size 0. This creates a recursion tree of height $n$, leading to $O(n^2)$ complexity.

### 2.2 Randomized Quicksort

To mitigate the worst-case scenario, we introduce randomness into the pivot selection. **Mechanism:** Before partitioning, we select a random index $r \in [low, high]$ and swap

$A[r]$ with $A[high]$.

$$P(\text{Worst Case}) \approx 0 \tag{1}$$

By sampling the pivot uniformly at random, the algorithm becomes agnostic to the input pattern. The probability of consistently picking bad pivots becomes astronomically low ($1/n!$), ensuring an expected runtime of $O(n \log n)$ on *any* input.

### 2.3 Dual-Pivot Quicksort

Proposed by Yaroslavskiy in 2009, this variation uses two pivots, $P_1$ and $P_2$ (where $P_1 \leq P_2$). **Mechanism:** The array is partitioned into three regions:

1. Elements $< P_1$
2. Elements between $P_1$ and $P_2$
3. Elements $> P_2$

This reduces the height of the recursion tree (logarithm base changes from 2 to 3) and effectively reduces the number of memory accesses, often yielding better performance on large datasets.

## 3 Implementation & Complexity Analysis

### 3.1 Asymptotic Complexity

| Algorithm | Best Case | Average Case | Worst Case |
|---|---|---|---|
| Standard Quicksort | $O(n \log n)$ | $O(n \log n)$ | $O(n^2)$ |
| Randomized Quicksort | $O(n \log n)$ | $O(n \log n)$ | $O(n^2)$ (Probabilistic) |
| Dual-Pivot Quicksort | $O(n \log n)$ | $O(n \log n)$ | $O(n^2)$ |

Table 1: Theoretical Time Complexity Comparison

**Space Complexity:** All three variants are in-place sorts, requiring $O(1)$ auxiliary memory for swapping. However, they require $O(\log n)$ stack space for recursion. In the worst case (Standard Quicksort on Sorted Data), this stack usage grows to $O(n)$, which causes Stack Overflow errors on large inputs.

### 3.2 Implementation Details & Design Choices

- **Language & Environment:** Implemented in C++ to minimize runtime overhead. We used `std::chrono::high_resolution_clock` for microsecond-precision timing.
- **Data Structures:** `std::vector<int>` was used for dynamic array management. Vectors provide contiguous memory allocation, which is critical for leveraging the cache locality benefits of Quicksort.
- **Random Number Generation:** We used the standard `rand()` function seeded with `time(0)`. While not cryptographically secure, it provides sufficient uniform distribution for pivot selection in algorithmic benchmarking.

### 3.3 Implementation Challenges

1. **Stack Overflow on Sorted Inputs:** During the testing of Standard Quicksort on sorted arrays of size $N > 10,000$, the recursion depth reached $N$, causing segmentation faults. We restricted the sorted tests to $N = 20,000$ to maintain stability.

2. **Accurate Benchmarking:** For small $N$, the execution time was dominated by OS noise. To counter this, we implemented a loop that runs each test case 20 times, collecting raw data points to visualize the variance (shown in error bands) rather than just simple averages.

## 4 Empirical Analysis & Results

### 4.1 Performance on Random Inputs (Average Case)

We tested all variants on random permutations of integers. As illustrated in Figure 1, all three algorithms exhibit $O(n \log n)$ growth.

- **Observation:** The lines for Standard (Red) and Randomized (Green) are nearly identical. This confirms that on random data, the "last element" strategy of Standard Quicksort effectively acts as a random pivot.

- **Dual-Pivot Advantage:** The Dual-Pivot implementation (Blue) is consistently faster as $N$ increases. This validates the theory that ternary partitioning (splitting into 3 parts) reduces the recursion depth and improves cache efficiency.

- **The Cache Cliff:** A noticeable spike occurs between $N = 5000$ and $N = 10000$. This nonlinear jump is attributed to the dataset exceeding the CPU's L1 Cache size, forcing slower L2/L3 cache access.

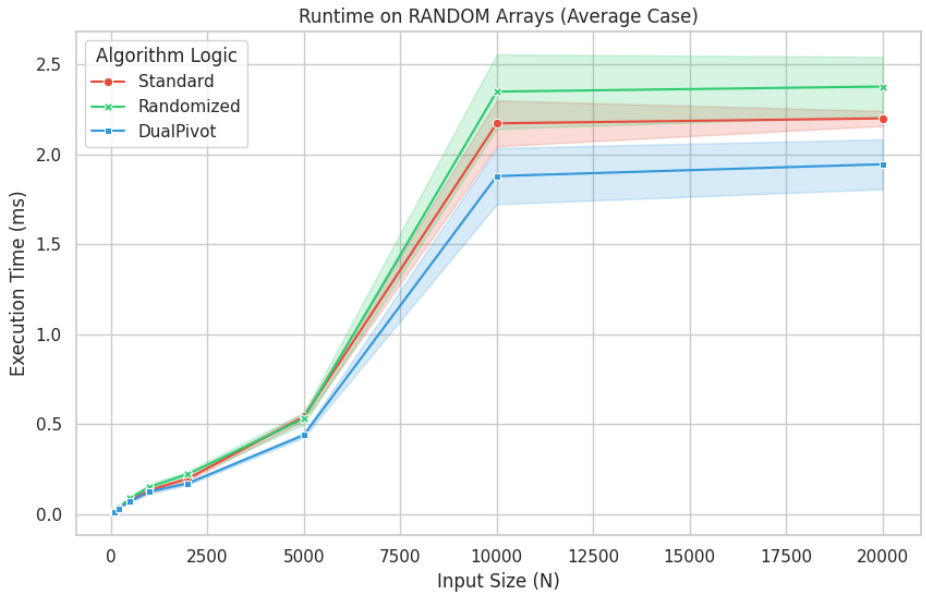


Figure 1: Runtime on Random Arrays. Dual-Pivot shows slight superiority.

### 4.2 The "Sorted Array" Vulnerability (Worst Case)

The true differentiation occurs when the input is already sorted.

- **Standard Quicksort Failure:** The runtime explodes quadratically ($O(n^2)$). At $N = 20,000$, Standard Quicksort took $\approx 947$ ms.
- **Randomized Resilience:** Randomized Quicksort maintained its $O(n \log n)$ performance, sorting the same $N = 20,000$ array in just $\approx 1.5$ ms. This represents a speedup factor of **600x**.
- **Dual-Pivot Determinism:** Interestingly, our deterministic Dual-Pivot implementation also degraded on sorted inputs (taking $\approx 180$ ms), though it was roughly $5\times$ faster than Standard Quicksort. This proves that Dual-Pivot also requires randomization to be truly robust against sorted adversaries.

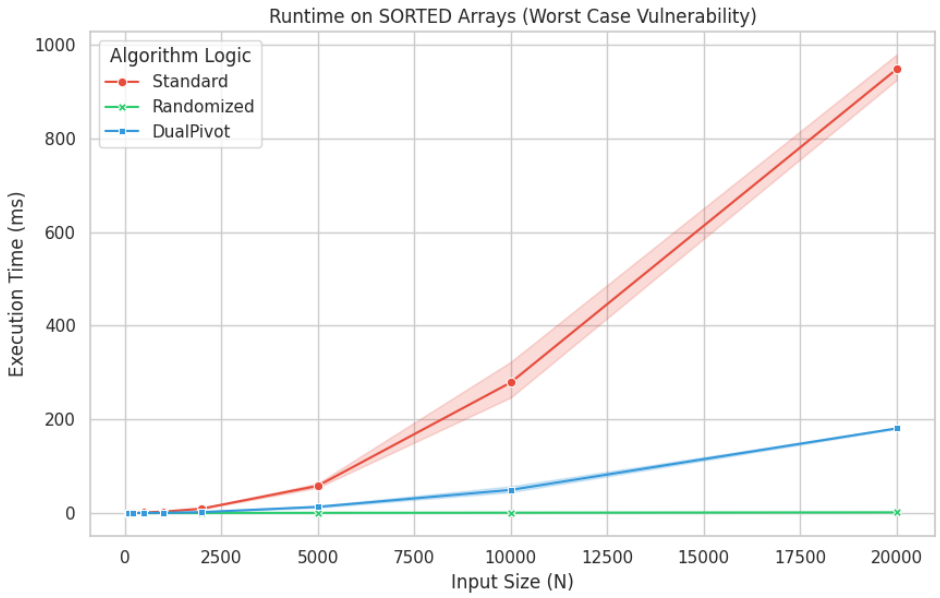


Figure 2: Runtime on Sorted Arrays. Standard Quicksort degrades to quadratic time; Randomized remains efficient.

## 5 Conclusion

This project empirically validated the theoretical bounds of Quicksort variants.

1. **Randomization is non-negotiable** for robustness. Without it, Quicksort is dangerously vulnerable to sorted inputs, degrading to bubble-sort-like performance ($O(n^2)$).
2. **Dual-Pivot is the superior architecture** for general cases, offering tangible speedups on random data due to reduced recursion depth.
3. **Best Practice:** The ideal sorting algorithm (like `std::sort` or Java's sort) combines these insights: using Dual-Pivot partitioning combined with randomized pivot selection (or falling back to Heapsort via Introsort) to guarantee safety and speed.