Created by Dmytro Krainyk

All code is in my repo on github

Assignment I: Getting things in order

Introduction

In this assigqnment, I focused on the empirical analysis of sorting algorithms and efficient data processing. The work consists of implementing classic comparison-based algorithms and solving a specific data matching problem using a linear-time sorting approach. All solutions are implemented in C++ for maximum performance.

Task 1A: “Ordnung muss sein” (Sorting Analysis)

Implementation Details

I implemented four fundamental sorting algorithms to compare their behavior on random datasets:

- **Bubble Sort & Insertion Sort:** Simple comparison-based algorithms with quadratic time complexity $O(n^2)$. Useful for educational purposes or extremely small datasets.
- **Merge Sort & Quick Sort:** Efficient divide-and-conquer algorithms with log-linear complexity $O(n \log n)$. Quick Sort is generally preferred in practice due to lower constant factors and in-place sorting capabilities.

Performance Analysis

I tested these algorithms on arrays ranging from 10 to 5,000 elements. The results are visualized below.

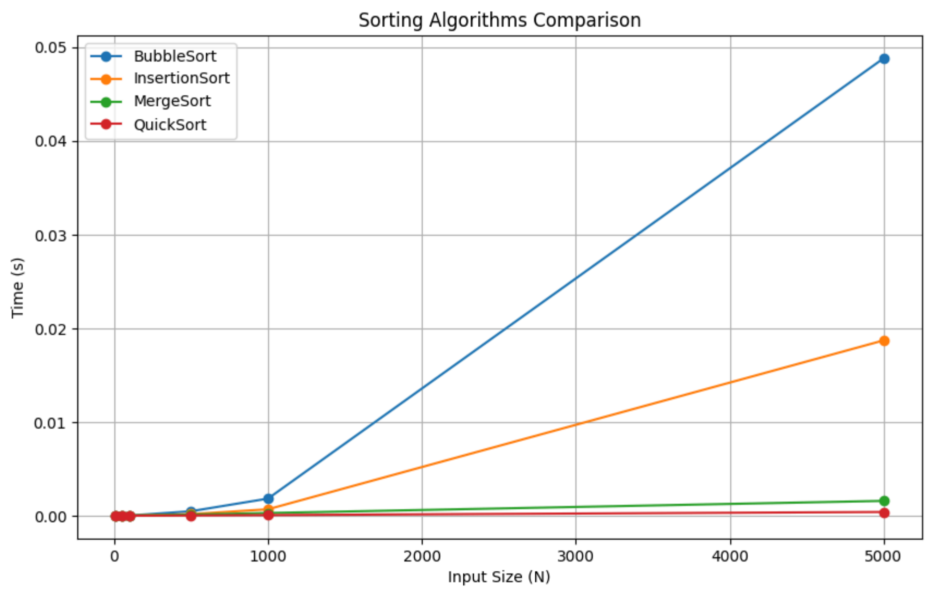

Figure 1: Fig 1. Empirical comparison of sorting algorithms. Note the logarithmic scale on the Y-axis.

Observations:

1. **The Quadratic Trap:** As seen in the graph, the execution time for Bubble Sort (Blue line) skyrockets as the input size increases. This empirically confirms the $O(n^2)$ complexity.
2. **Efficiency of Divide-and-Conquer:** Quick Sort (Red) and Merge Sort (Green) remain computationally inexpensive even as N grows, adhering to the $O(n \log n)$ trendline. Quick Sort proved to be the fastest on random data.

Task 1B: “The Olsen Gang” (Linear Sort)

Problem & Solution

The task required matching two large datasets of credit cards (one sorted, one shuffled) in **linear time**. Standard sorting would take $O(n \log n)$, which is suboptimal for massive datasets.

I chose Radix Sort (Least Significant Digit variant) as the linear solution.

Key Implementation Decisions

- **Composite Key:** Instead of sorting by Date and PIN separately, I combined them into a single integer key (e.g., `MMYYPIN`). This simplified the sorting logic.
- **Counting Sort Subroutine:** Radix Sort uses stable Counting Sort for each digit, ensuring $O(N + k)$ complexity, where k is the range of digits (0-9).

Linear vs Log-Linear Comparison

I compared my custom Radix Sort against the optimized C++ standard library sort (`std::sort` - Introsort).

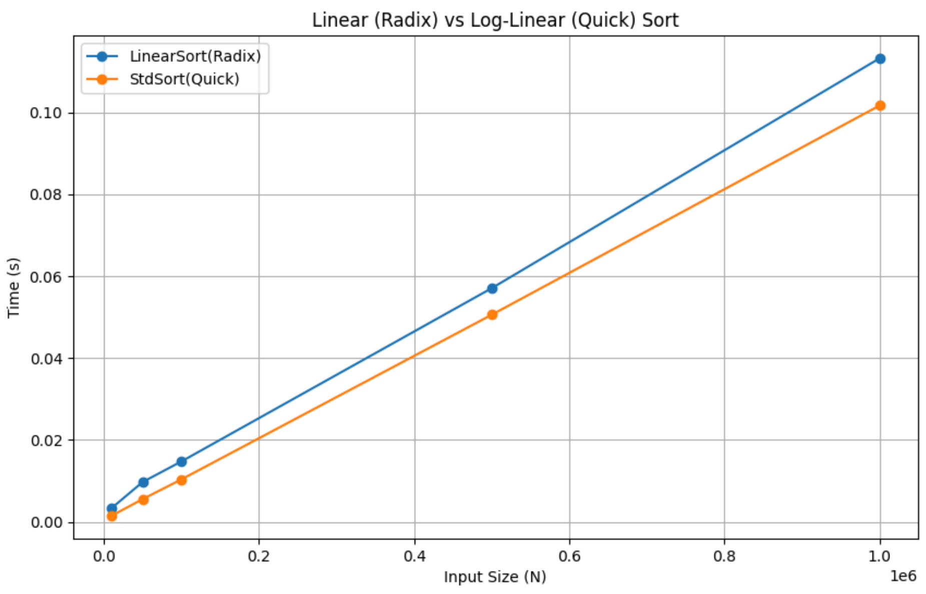

Figure 2: Fig 2. Performance of Linear Radix Sort vs Standard Quick Sort.

Findings:

- **Linear Trend:** The Radix Sort (Blue line) demonstrates a purely linear growth pattern. It does not suffer from the $\log n$ multiplier overhead that comparison sorts have.
- **Crossover Point:** While Radix Sort is theoretically superior ($O(N)$), the graph shows that `std::sort` (Orange) remains highly competitive on datasets up to 1 million elements due to extreme optimization of the C++ STL. However, for significantly larger datasets or simpler keys, the linear nature of Radix Sort would eventually gain absolute supremacy.