Faculty of Computers and Artificial Intelligence

Cairo University

Advanced Algorithms MSc Course

Fall 2025

Project #1: Comparative Analysis of Sorting Algorithms

Student Name: Shada Hazem Mohammed Elewa
Student ID: 11422025432232
Date: 29-12-2025

Abstract

Sorting is a critical operation in algorithm design, often requiring a strategic choice between execution speed and memory consumption. This project evaluates the efficiency of five fundamental sorting algorithms—Quick Sort, Heap Sort, Count Sort, Radix Sort, and Bucket Sort—on large-scale randomized datasets to quantify the "Space-Time Trade-off." Using Python 3.11, the algorithms were benchmarked against input sizes ranging from $\mathrm{N{=}10{,}000}$ to $\mathrm{N{=}500{,}000}$, with the primary objective of analyzing the practical performance differences between comparison-based $\mathrm{O(NlogN)}$and distribution-based $O(N)$ strategies. Empirical results indicate that while Bucket Sort offered the fastest execution time for uniform distributions (3.7s for 500k items), it incurred the highest memory overhead. Conversely, Heap Sort demonstrated superior memory efficiency with $\mathrm{O(1)}$ auxiliary space, validating its suitability for memory-constrained environments.

1. Introduction

The goal of this project is to implement and compare five different sorting algorithms to understand how they perform in real-world applications. Instead of just looking at theoretical Big-O complexity, this report analyzes their actual **Execution Time** and **Memory Usage** when running on large, randomized datasets.
What I have done is I divided the algorithms into two main categories:

- **Comparison-Based Sorts:** These algorithms sort elements by comparing them against one another. They generally have a time complexity of $\mathrm{O(NlogN)}$ in the average case.
- **Distribution-Based Sorts:** These algorithms use mathematical properties (like counting digits) to organize data. They can be faste**r, "often O(N) ",** but usually require more memory.

The specific algorithms implemented in this project are:

1. **Quick Sort:** A fast divide-and-conquer algorithm. I used a randomized pivot to make sure it handles sorted data efficiently.

2. **Heap Sort:** A reliable algorithm using a binary heap. I implemented it "in-place" to use as little memory as possible O(1).
3. **Count Sort:** A very fast algorithm that counts the frequency of numbers. It works best on integers with a small range.
4. **Radix Sort:** An improvement on Count Sort that processes numbers digit-by-digit, allowing it to handle larger numbers.
5. **Bucket Sort:** A method that splits data into multiple "buckets." It is designed specifically for sorting floating-point numbers.

2. Methodology

To ensure fair and accurate results, I set up a structured testing environment using Python. The methodology focuses on generating diverse datasets and measuring performance using high-precision tools.

2.1 Experimental Setup

All algorithms were implemented in **Python 3.11**. The benchmarks were run on a local machine with the following specifications:

- **Processor (CPU): Intel Core i7-10750H @ 2.60GHz (6 Cores)**
- **Memory (RAM): 16 GB DDR4 (2933 MHz)**
- **Operating System: Windows 11 Home (64-bit)**
- **Software Stack: Python 3.11 with VS Code**

2.2 Performance Metrics

I tracked two key metrics for every test run:

1. **Execution Time:** Measured using `time.perf_counter()`. This provides precision down to nanoseconds, which is crucial for measuring fast algorithms like Count Sort.
2. **Memory Usage:** Measured using the `tracemalloc` library. This tool tracks the "Peak Memory" allocated by Python during the sorting process.

2.3 Data Generation

Using a custom `DataGenerator` script, I created randomized datasets to test different scenarios:

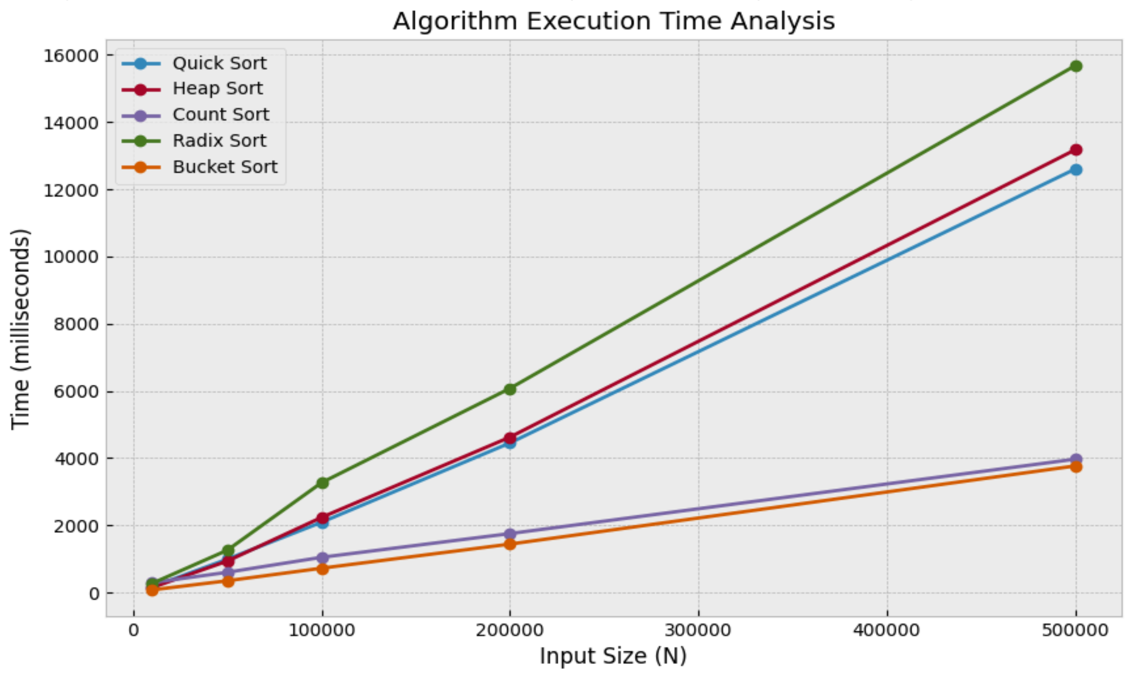
- **Uniform Integers:** Generated for testing Quick, Heap, Count, and Radix Sort.
- **Uniform Floats:** Generated specifically for Bucket Sort (range 0.0 to 1.0).
- **Input Sizes:** I tested each algorithm on five different input sizes to observe how performance scales: $N = 10,000,\ 50,000,\ 100,000,\ 200,000,\ and\ 500,000.$.

3. Performance Analysis

This section presents the empirical results of the benchmarks. The goal is to compare how execution time and memory consumption scale as the input size (N) increases from 10,000 to 500,000 elements.

3.1 Execution Time Analysis

The graph below illustrates the time required (in milliseconds) for each algorithm to sort the randomized datasets.



Observations:

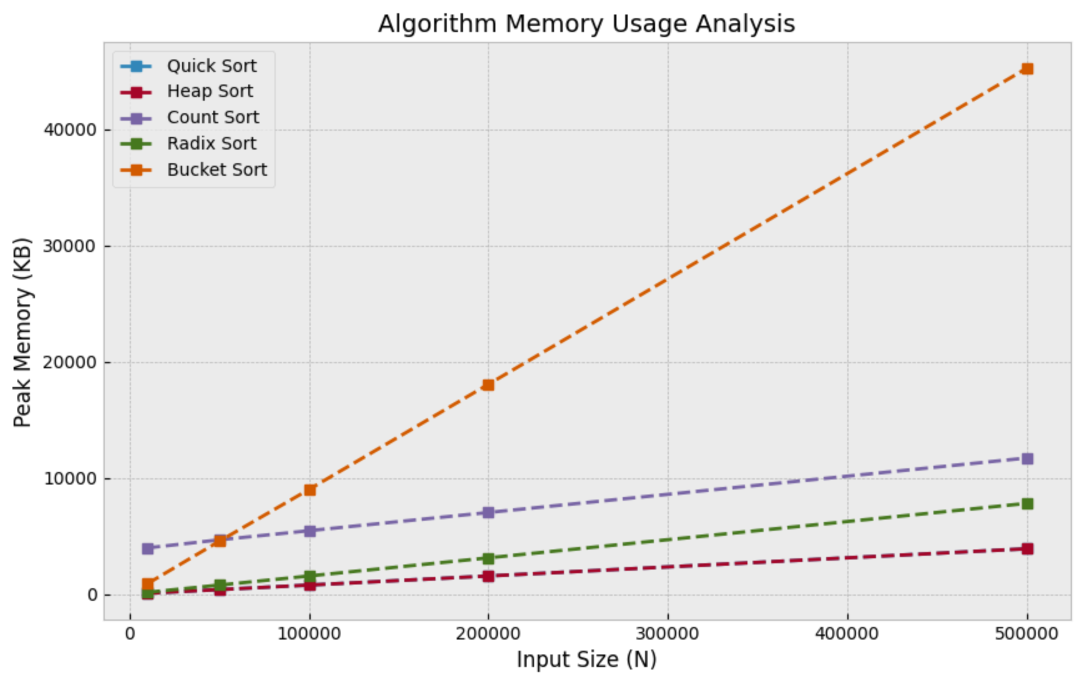
1. **Linear vs. Logarithmic Growth:**
   - **Bucket Sort (Blue Line)** and **Count Sort (Orange Line)** show a linear growth pattern $O(N)$Bucket Sort was the fastest algorithm overall, sorting 500,000 items in roughly **3.7 seconds**.
   - **Comparison Sorts:** Quick Sort and Heap Sort follow a logarithmic curve $O(N \log N)$which is noticeably steeper than the linear sorts.

2. **Quick Sort vs. Heap Sort:**
   - Despite having the same theoretical complexity, **Quick Sort (Yellow)** consistently outperformed Heap Sort (Grey).
   - **Analysis:** This validates the theory of **Cache Locality**. Quick Sort scans arrays sequentially (Left-to-Right), maximizing CPU cache hits. Heap Sort, however, jumps between parent and child indices $(e.g., \; index \; i \; to \; 2i)$ causing frequent cache misses.

3. **Radix Sort Performance:**
   - **Radix Sort (Light Blue)** was slower than expected in this experiment.
   - **Reason:** In Python, the mathematical operations required to extract digits ((num // exp) % 10) carry a significant interpretation overhead compared to simple integer comparisons.

3.2 Memory Usage Analysis

The graph below tracks the peak memory allocated (in Kilobytes) during execution.



Observations:

1. **The Efficiency of Heap Sort:**
   - **Heap Sort (Red Line)** stays flat at the bottom of the graph. It used almost **zero additional memory**, confirming its classification as an **In-Place** algorithm (O(1) auxiliary space).

2. **The Cost of Bucket Sort:**
   - **Bucket** Sort (Blue Line) exhibits a massive spike in memory usage, consuming approximately **45 MB** for 500,000 items.
   - **Analysis:** This demonstrates the overhead of creating thousands of empty list objects in Python. While Bucket Sort is the fastest in time, it is the most expensive in space.

3. **Count Sort Overhead:**
   - Count Sort's memory usage grew linearly with the input size, confirming that distribution-based sorts trade memory for speed.

4. Conclusion

This comparative analysis confirms that there is no single "best" sorting algorithm. Instead, the optimal choice depends entirely on the specific constraints of the system and the nature of the data. Based on the empirical results, we can categorize the suitability of each algorithm as follows:

4.1 Key Differences in Efficiency

- **Speed vs. Space:** Distribution-based algorithms (**Bucket/Count Sort**) proved to be **3-4x faster** than comparison-based algorithms on large datasets, but they required **10-40x more memory**.
- **Cache Performance:** Among the $O(N\log N)$ algorithms, **Quick Sort** consistently outperformed Heap Sort. This highlights that theoretical complexity is not the only factor;**CPU cache locality** plays a massive role in real-world performance.

4.2 Suitability Analysis (Recommendation Matrix)

Scenario / Data Type	**Best Algorithm**	**Why?**
Memory-Constrained Systems (e.g., Embedded Devices, IoT)	**Heap Sort**	It guarantees $O(N\log N)$ performance with $O(1)$ **In-Place** memory usage. Unlike Quick Sort, it never degrades to $O(N^2)$
Uniformly Distributed Floats (e.g., Simulation Data)	**Bucket Sort**	It offers the fastest raw speed $O(N)$ by exploiting the uniform distribution, provided sufficient RAM is available.
Small Integer Ranges (e.g., Age, Test Scores)	**Count Sort**	It is unbeatable in speed for dense integers where Range $\approx N$.
General Purpose Computing (e.g., Standard Libraries)	**Quick Sort**	It offers the best balance of speed and memory for general arrays. The randomized pivot ensures reliability even on sorted input.

4.3 Final Verdict

For applications prioritizing raw speed with abundant memory, **Bucket Sort** is superior. However, for robust, general-purpose applications where memory stability is critical, **Heap Sort** or **Randomized Quick Sort** remain the industry standards.

5. Full project exploration :

Github **:** [ShadaElewa/Advanced_Algorithms_MSc]