Sorting Algorithm Benchmark Report

Michał Łasocha

June 5, 2025

Contents

1 Introduction

This report will go over presentation, implementation and benchmarking of multiple sorting algorithm, using go programming language.

2 Selected algorithms

I have selected 7 different algorithms for the purpose of this report:

- **Bubble Sort**
  - *Design*: Simple comparison-based algorithm that repeatedly swaps adjacent elements if they are out of order, "bubbling" the largest element to the end on each pass.
  - *Implementation*: Self-implemented
  - *Best Case Complexity*: $O(n)$ — when the array is already sorted (with optimization).
  - *Worst Case Complexity*: $O(n^2)$ — when the array is sorted in reverse.
- **Bucket Sort**
  - *Design*: Distributes elements into buckets, sorts each bucket (usually with another algorithm), then concatenates the buckets. Works best with uniformly distributed data.
  - *Best Case Complexity*: $O(n + k)$ — with k buckets and evenly distributed data.
  - *Implementation*: Self-implemented
  - *Worst Case Complexity*: $O(n^2)$ — if all elements fall into one bucket.
- **Go's Built-in Sort (sort.Ints)**
  - *Design*: Hybrid adaptive sort (introsort) combining quicksort, heapsort, and insertion sort based on input.
  - *Implementation*: Go standard library
  - *Best Case Complexity*: $O(n \log n)$
  - *Worst Case Complexity*: $O(n \log n)$ — guaranteed by switching to heapsort if needed.
- **Heap Sort**
  - *Design*: Builds a max-heap and repeatedly extracts the maximum element, maintaining the heap property.
  - *Implementation*: Self-implemented
  - *Best Case Complexity*: $O(n \log n)$
  - *Worst Case Complexity*: $O(n \log n)$
- **Insertion Sort**
  - *Design*: Builds the sorted array by inserting each element into its correct position.
  - *Implementation*: Self-implemented
  - *Best Case Complexity*: $O(n)$ — when the array is already sorted.
  - *Worst Case Complexity*: $O(n^2)$ — when the array is reverse sorted.
- **QuickSort**
  - *Design*: Divide-and-conquer algorithm selecting a pivot, partitioning the array, then recursively sorting subarrays.
  - *Implementation*: Self-implemented
  - *Best Case Complexity*: $O(n \log n)$ — balanced partitions.
  - *Worst Case Complexity*: $O(n^2)$ — consistently unbalanced partitions.

- **Radix Sort (Signed)**
  - *Design*: Non-comparative integer sort processing digits from least significant to most significant, adapted for signed integers.
  - *Implementation*: Self-implemented
  - *Best Case Complexity*: $O(nk)$, where k is the number of digit positions.
  - *Worst Case Complexity*: $O(nk)$ — depends mainly on digit count, not input order.
- **Threaded Merge Sort (Parallel Merge Sort)**
  - *Design*: Splits array into parts, sorts them in parallel threads, then merges sorted subarrays.
  - *Implementation*: Self-implemented
  - *Best Case Complexity*: $O\left(\frac{n \log n}{p} + \text{overhead}\right)$, where p is number of threads.
  - *Worst Case Complexity*: $O\left(\frac{n \log n}{p} + \text{overhead}\right)$.

3 Experimental Setup and Methodology

Following system and hardware were used in order to obtain the data presented later in the report:

3.1 Environment

- *Hardware*: Ryzen 9 7900X (12C/24T), 32 GB DDR5 RAM
- *Operating system*: Gnu/Linux 6.14.6-zen1-1-zen
- *Programming Language and runtime*: go version go1.24.3 linux/amd64

3.2 Methodology

All Algorithms were implemented or imported in main.go package, with use of go standard library. Self-implemented algorithms were validated using unit tests provided inside of the main_test.go package.

Each algorithm was written with integer sorting in mind, with minimal non-compiler optimizations, using slices in place of arrays, and goroutines in place of threads.

Threaded algorithms were written in mind to adapt to user's thread count and use worker groups in order to maximal efficiency and avoid thread starting overhead where possible.

All sorting Algorithms would be benched using the implemented BenchmarkSort and BenchmarkSort-NoReturn functions that measure time in milliseconds. Program generates multiple differently sized arrays of random integers and runs each algorithm while measure time over all runs, returning average time. All times are logged into a csv file for further data processing using LibreOffice Calc.

3.3 Early predictions

Go's built in sort's hybrid design should deliver the best results for average cases, while remaining competitive with massive data sets. Algorithms such as insertion sort will perform well on small data sets, but will fail to compete with other algorithms with large datasets. Parallel Merge Sort will have poor results with small data sets, due to overhead caused by its threaded design, however it should by far scale the best with extreme size data sets due to it being able to split the workload across multiple threads.

4 Results

4.1 Raw data

The raw data represents the direct timing outputs from the benchmark runs, collected over multiple iterations for each algorithm and input size. It is stored in CSV format without filtering. Algorithms such as Insertion sort and especially Bubble sort were skipped at 10 million and 100 million element arrays, as time for soting would be measured in hours, making it completely unusable.

Size	Bubble Sort	Bucket Sort	Go's built-in Sort	Heap Sort	Insertion Sort	QuickSort	Radix Sort	Threaded Merge Sort	Total Result
100	3.02	1.8	0.01	1.11	1.03	8.87	1.09	4.48	21.41
1000	487.13	13.68	7.9	21.67	106.79	165.14	18.98	40.26	861.55
10000	28210	278.61	353.36	550.6	9263.46	1584.91	182.7	307.7	40731.34
100000	6180190	4044.6	4327.9	7614.9	863296.8	21757.8	3221.3	4045.1	7088498.4
1000000		32787.8	47876	96219.2		203230.6	26929.6	18241.8	425285
10000000		551177.8	550829.4	1650828.2		2047687.6	297298	157711	5255532
Total Result	6208890.15	588304.29	603394.57	1755235.68	872668.08	2274434.92	327651.67	180350.34	12810929.7

Table 1: Sorting algorithm benchmark times (ms) by input size.

4.2 Processed data

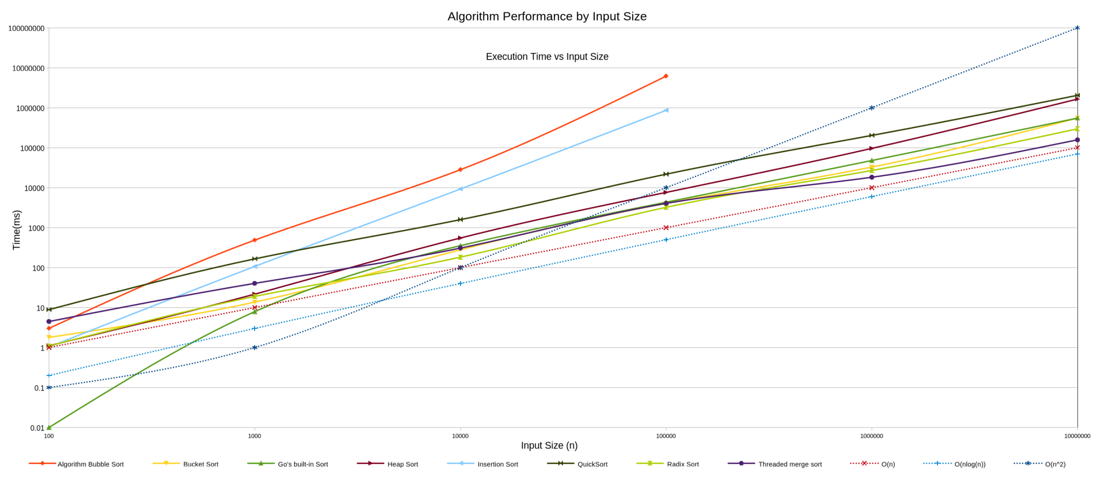

Figure 1: Sorting algorithm benchmark times (ms) by input size, represented as a line chart

5 Analysis and Conclusions

The benchmark outcome clearly shows the large difference in performance of the various sorting algorithms for different input sizes. As predicted, straightforward quadratic algorithms like Bubble Sort and Insertion Sort have bad scalability, with running times increasing quickly as the size of the input increases. For example, Bubble Sort's time leaps from a few milliseconds on small inputs to several million milliseconds for bigger arrays, making it impractical for large data sets. On the other hand, algorithms with better asymptotic complexities, such as

QuickSort, Go native sort (introsort), Heap Sort, and Threaded Merge Sort, exhibit considerably better performance and scalability. QuickSort tends to be consistently fast due to its average-case $O(n \log n)$ complexity, while the Go native sort utilizes very optimized introsort, leading to the lowest execution times in most scenarios. Radix Sort and Bucket Sort, that utilize non-comparison based sorting

methods, execute competitively especially on large sets of data, capitalizing on their linear or near-linear time complexities under optimal conditions. The results also confirm the expected trade-offs between algorithm complexity, implementation concerns, and parallelism (for Threaded Merge Sort). Exclusion of Bubble Sort and Insertion Sort for the largest dataset was justified given their infeasible runtimes. Generally, this evidence corroborates the theoretical time complexities and affirms the importance of choosing an optimal sorting algorithm based on dataset size and performance requirements.