Fakultät CB

HOCHSCHULE MITTWEIDA
University of Applied Sciences

Permutation-Sorting Algorithms & Complexity

Practical ISO-Date String Experiments

Georgii Burdin

hs-mittweida.de

Why and what for?

- Sorting is a fundamental operation in computer science, mathematics, and data analysis.
- Any sorting algorithm works by transforming an initial permutation of data into the sorted (identity) permutation.
- Understanding how algorithms handle permutations helps us analyze their efficiency and correctness.
  Real-world uses of sorting algorithms:
- Preparing data for time-series analysis and forecasting
- Applications in networking (packet reordering), cryptography, and more
- Organizing transaction or event logs by time.
- * I will use ISO-8601 date strings ("YYYY-MM-DD") as a concrete example for sorting. This reflects a common real-world need: sorting events by date (eg in reading logs of the system).

What Is a Permutation?

Definition: A *permutation* of a finite set $S = \{1, 2, \ldots, n\}$ is a bijective function $\pi : S \to S$

Notation:

- A permutation can be written as a sequence $(\pi(1), \pi(2), \ldots, \pi(n))$, meaning: 1 goes to $\pi(1)$, 2 goes to $\pi(2)$, etc.
- Example: For $n = 4$, one possible permutation is $(3, 1, 4, 2)$. This means:
  - $1 \mapsto 3$
  - $2 \mapsto 1$
  - $3 \mapsto 4$
  - $4 \mapsto 2$
- The total number of permutations of n objects is $n!$ (factorial).

Experiment Summary

- Implemented Bubble, Insertion, Merge, Quick and Heap in Python.
- Generated n sequential ISO date strings from a random starting date.
- Shuffled the data into random, reversed, and already sorted orders.
- Measured runtime of 1000 random initialized datasets with `time.perf_counter` and memory usage with `tracemalloc`.
- Compared and analyzed performance results for the five sorting algorithms.
- Made conclusions about which algorithms are fastest, slowest, and most efficient in terms of memory and time.

Algorithms Overview

Algorithm	Core Idea	Best	Worst	Space
Bubble sort	Swap adjacent out-of-order items	$O(n)$	$O(n^2)$	$O(1)$
Insertion sort	Insert each element into sorted prefix	$O(n)$	$O(n^2)$	$O(1)$
Merge sort	Divide, sort, merge (stable)	$O(n \log n)$	$O(n \log n)$	$O(n)$
Quick sort	Partition around pivot (in-place)	$O(n \log n)$	$O(n^2)$	$O(\log n)$
Heap sort	Build max-heap, extract max	$O(n \log n)$	$O(n \log n)$	$O(1)$

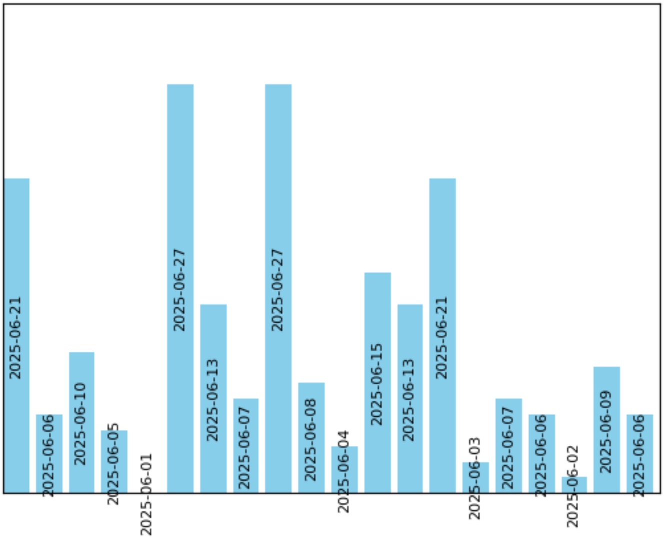
Bubble Sort

1. **Scan:** Iterate through the list from the first element to the last.
2. **Compare & Swap:** For each adjacent pair (a_j, a_{j+1}), if $a_j > a_{j+1}$ swap them.
3. **Repeat:** After each full pass the largest unsorted element "bubbles" to its correct position at the end. Continue passes until no swaps occur.

Example.

$$[4, 3, 2, 1] \to [3, 4, 2, 1] \to [3, 2, 4, 1] \to [3, 2, 1, 4] \to \cdots \to [1, 2, 3, 4]$$

Bubble Sort – June 2025 (swaps=107)



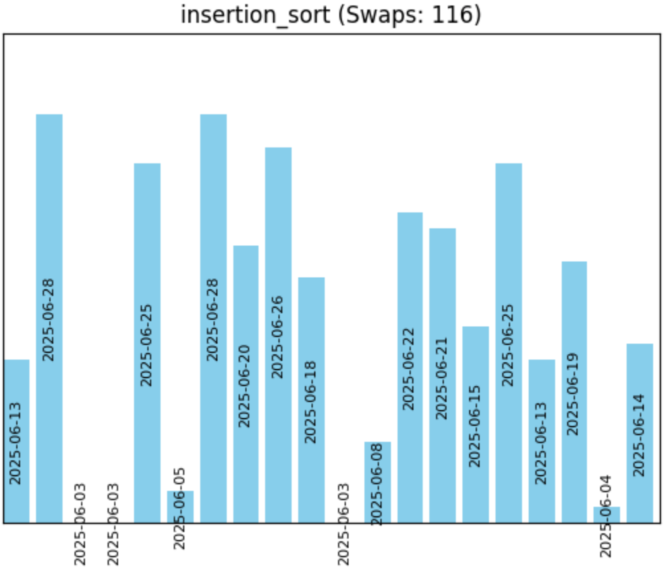
Insertion Sort Step-by-Step

1. **Pick the key** — the first item in the *unsorted* tail.
2. **Shift larger items right** until you reach the key's sorted position.
3. **Drop the key** into that gap. The sorted prefix is now one element longer.

Example — ISO dates

Init:	[05, 02, 04, 01, 03]	(start: first element is the 1-item sorted prefix)
1:	[02, 05, 04, 01, 03]	Key = 02; shift 05 right, insert 02
2:	[02, 04, 05, 01, 03]	Key = 04; shift 05, insert 04
3:	[01, 02, 04, 05, 03]	Key = 01; shift 05, 04, 02, insert 01
4:	[01, 02, 03, 04, 05]	Key = 03; shift 05, 04, insert 03 (sorted)

where 05 = 2025-06-05, 02 = 2025-06-02, etc.


insertion_sort (Swaps: 116)
2025-06-13
2025-06-28
2025-06-03
2025-06-03
2025-06-25
2025-06-05
2025-06-28
2025-06-20
2025-06-26
2025-06-18
2025-06-03
2025-06-08
2025-06-22
2025-06-21
2025-06-15
2025-06-25
2025-06-13
2025-06-19
2025-06-04
2025-06-14

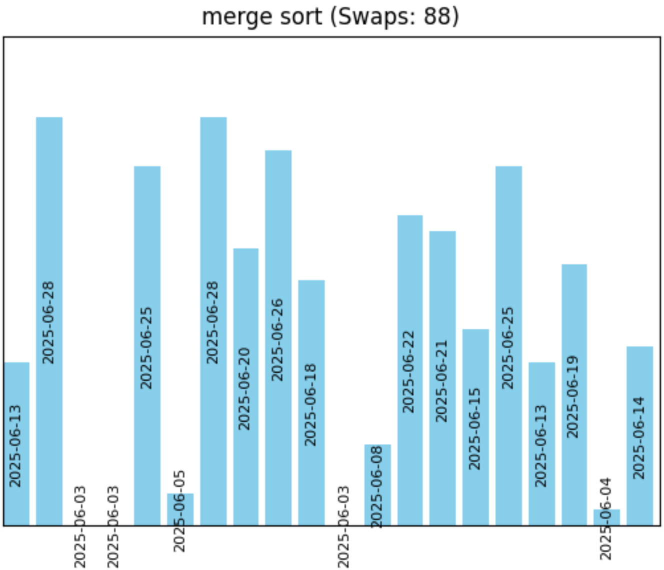
Merge Sort

1. **Divide:** Recursively split the list into halves until sublists of size 1 remain.
2. **Merge:** Repeatedly merge two sorted sublists into a single sorted list:
   - Compare the smallest elements of each sublist,
   - Copy the smaller element into the output buffer,
   - Continue until all elements from both sublists are merged.
3. **Copy:** Copy the merged buffer back to the original list segment.

Example.

$$
\begin{aligned}
&[\texttt{"2025-06-04"}, \texttt{"2025-06-01"}, \texttt{"2025-06-03"}, \texttt{"2025-06-02"}]\\
\rightarrow\ &[\texttt{"2025-06-04"}, \texttt{"2025-06-01"}] + [\texttt{"2025-06-03"}, \texttt{"2025-06-02"}]\\
\rightarrow\ &[\texttt{"2025-06-01"}, \texttt{"2025-06-04"}] + [\texttt{"2025-06-02"}, \texttt{"2025-06-03"}]\\
\rightarrow\ &[\texttt{"2025-06-01"}, \texttt{"2025-06-02"}, \texttt{"2025-06-03"}, \texttt{"2025-06-04"}]
\end{aligned}
$$

Mer



merge sort (Swaps: 88)
2025-06-13
2025-06-28
2025-06-03
2025-06-03
2025-06-25
2025-06-05
2025-06-28
2025-06-20
2025-06-26
2025-06-18
2025-06-03
2025-06-08
2025-06-22
2025-06-21
2025-06-15
2025-06-25
2025-06-13
2025-06-19
2025-06-04
2025-06-14

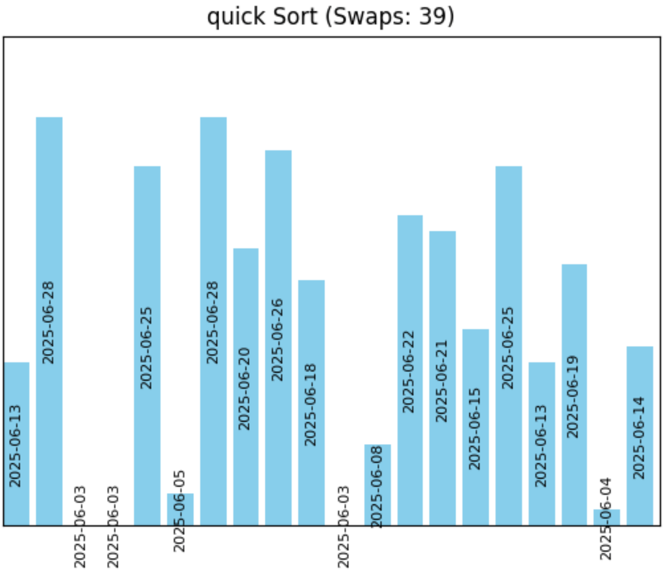
Quick Sort

Algorithm steps:

1. **Choose a pivot:** Select an element from the array (e.g., first, last, or random).
2. **Partition:** Rearrange elements so that:
   - All elements $\leq$ pivot are moved to the left of the pivot,
   - All elements $>$ pivot are moved to the right.
3. **Recursively sort:** Apply the same process to the left and right subarrays (excluding the pivot, which is now in its final position).
4. **Base case:** Arrays of length 0 or 1 are already sorted.

Example (pivot = 5).

$$6, 2, 9, 4, 8, \boxed{5} \rightarrow 2, 4, \boxed{5}, 9, 8, 6 \text{ (then iteratively sort left and right subsets)}$$


quick Sort (Swaps: 39)
2025-06-13
2025-06-28
2025-06-03
2025-06-03
2025-06-25
2025-06-05
2025-06-28
2025-06-20
2025-06-26
2025-06-18
2025-06-03
2025-06-08
2025-06-22
2025-06-21
2025-06-15
2025-06-25
2025-06-13
2025-06-19
2025-06-04
2025-06-14

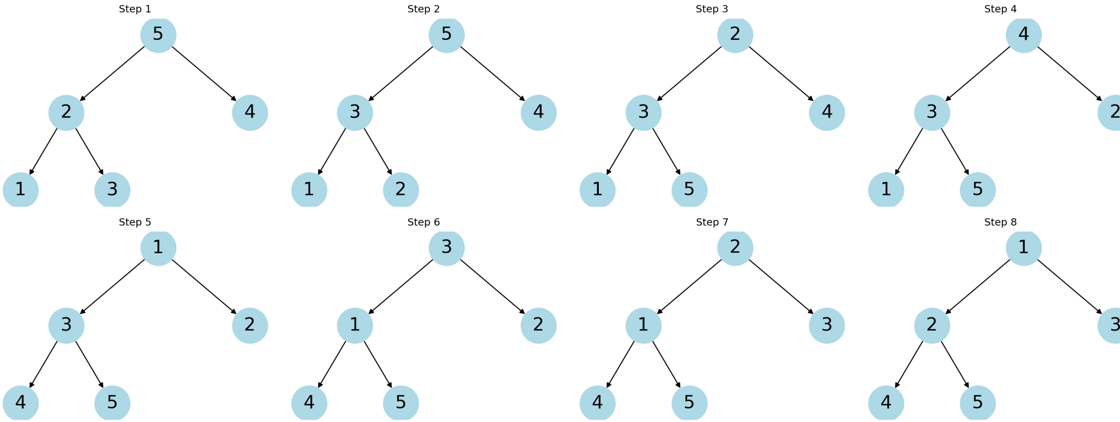
Heap Sort:

Algorithm steps:

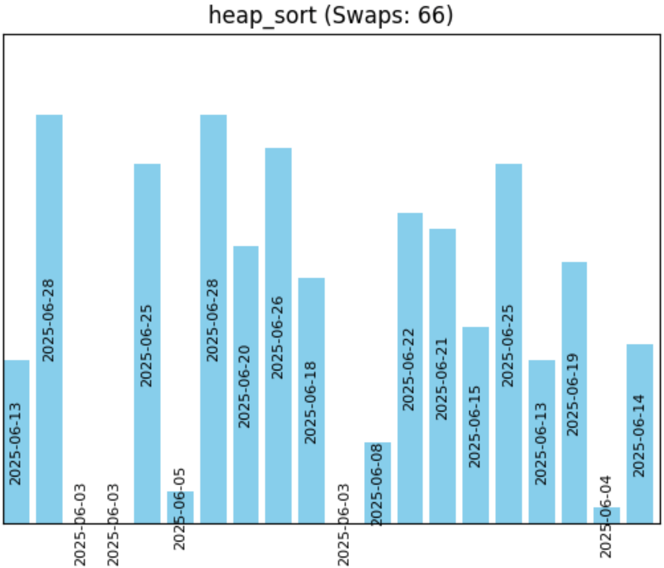
1. **Build a max-heap:** Transform the array into a max-heap, where each parent node is greater than or equal to its children.
2. **Sort:**
   - Swap the first element (the maximum) with the last element of the heap.
   - Reduce the heap size by one (ignore the last sorted element).
   - Restore the heap property ("sift down" the new root as needed).
   - Repeat until the heap size is 1.
3. **Result:** The array is now sorted in-place.

```
  ["2025-06-05", "2025-06-02", "2025-06-04", "2025-06-01", "2025-06-03"]
→ ["2025-06-03", "2025-06-02", "2025-06-01", "2025-06-04", "2025-06-05"]
→ ["2025-06-01", "2025-06-02", "2025-06-03", "2025-06-04", "2025-06-05"]
```

Heap Sort: Step by Step



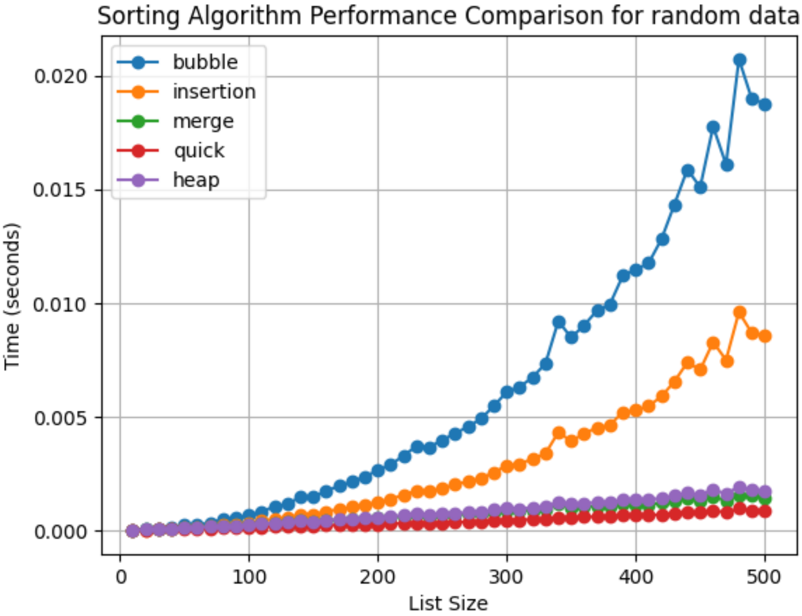
Hea

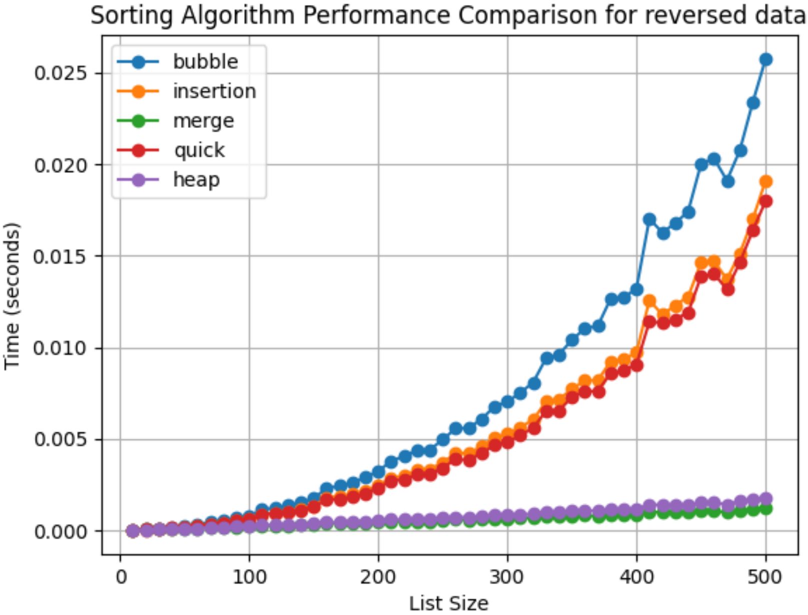


heap_sort (Swaps: 66)
2025-06-13
2025-06-28
2025-06-03
2025-06-03
2025-06-25
2025-06-05
2025-06-28
2025-06-20
2025-06-26
2025-06-18
2025-06-03
2025-06-08
2025-06-22
2025-06-21
2025-06-15
2025-06-25
2025-06-13
2025-06-19
2025-06-04
2025-06-14

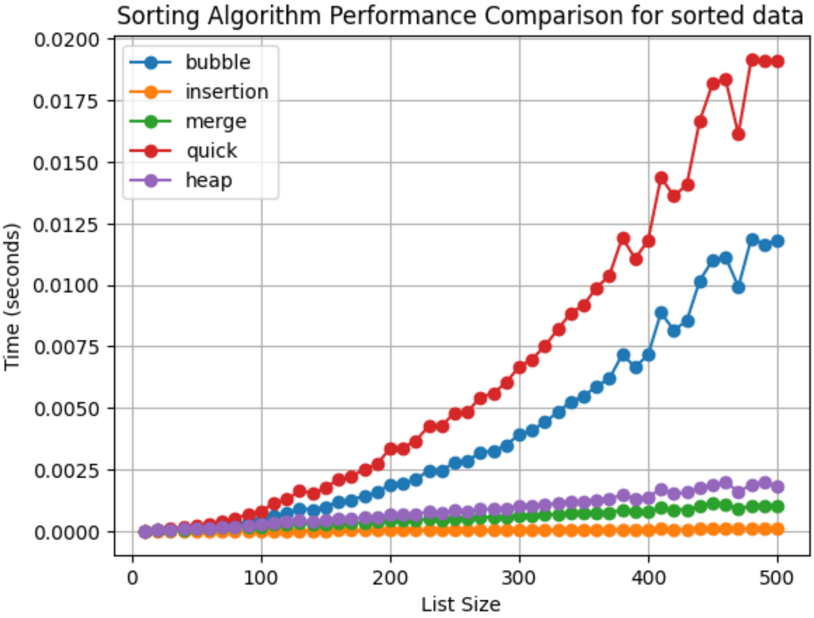
Empirical Results ($n = 100$, average over 100 runs)

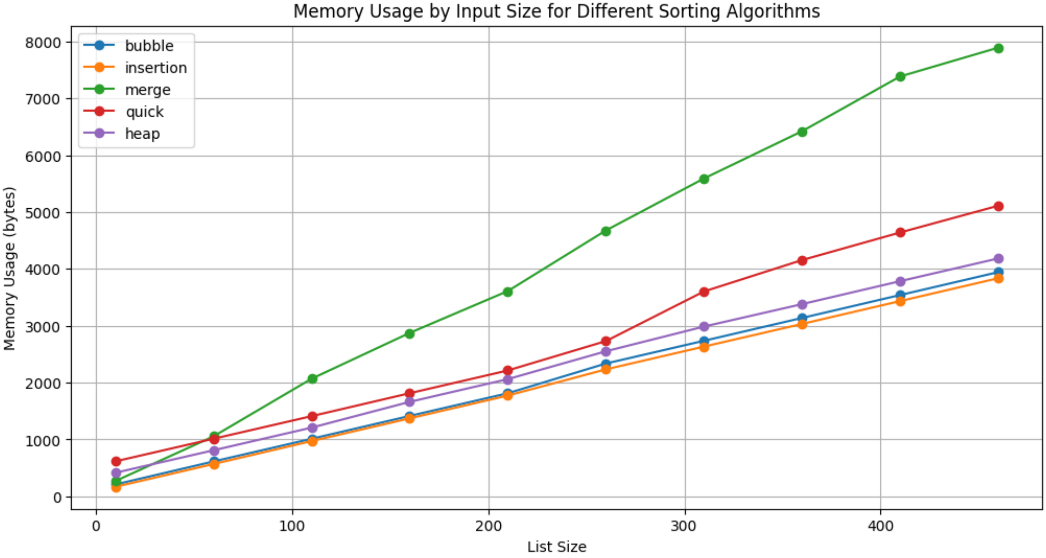
Algorithm	Average Swaps	Peak Mem (bytes)
Bubble	2469	12132593
Insertion	388	335898
Merge	580	502194
Quick	2547	2199561
Heap	672	583718


Sorting Algorithm Performance Comparison for random data
bubble
insertion
merge
quick
heap
0.020
0.015
0.010
0.005
0.000
Time (seconds)
0
100
200
300
400
500
List Size



Sorting Algorithm Performance Comparison for reversed data
bubble
insertion
merge
quick
heap
Time (seconds)
0.000
0.005
0.010
0.015
0.020
0.025
List Size
0
100
200
300
400
500


Sorting Algorithm Performance Comparison for sorted data
bubble
insertion
merge
quick
heap
Time (seconds)
0.0200
0.0175
0.0150
0.0125
0.0100
0.0075
0.0050
0.0025
0.0000
0
100
200
300
400
500
List Size


Memory Usage by Input Size for Different Sorting Algorithms
bubble
insertion
merge
quick
heap
Memory Usage (bytes)
8000
7000
6000
5000
4000
3000
2000
1000
0
0
100
200
300
400
List Size

Conclusion

- Bubble sort and insertion sort are simpler but slower ($O(n^2)$)
- Merge sort, quick sort, and heap sort are faster ($O(n \log n)$)
- Merge sort performs stable on all inputs but Merge Sort may require additional space
- Quick sort can have problems with already sorted lists
- Different algorithms perform better on different types of input Choosing an algorithm depends on input size, memory budget, and requirements such as stability or in-place operation.

Future Research Topics

- Parallel and distributed sorting algorithms
- External-memory and cache-aware sorts
- Cryptographic shuffles and secure permutation generation