# CSE331 - Assignment #1

Jeonghoon Park (20201118)
UNIST
South Korea
hoonably@unist.ac.kr

**Project:** https://hoonably.github.io/cse-archive/sorting-algorithm

## 1 PROBLEM STATEMENT

In many applications, sorting is a fundamental operation. While classical algorithms such as Merge Sort and Quick Sort are widely known, newer or specialized algorithms offer different trade-offs in performance, stability, and space usage. The objective of this project is to implement and evaluate 12 sorting algorithms under unified conditions. Through benchmarking across diverse input types and data sizes, we aim to analyze their actual runtime characteristics, stability, and behavior with different data types. However, in real-world scenarios, performance is not the only concern. There are situations where stability of the sorting algorithm is required, such as when secondary ordering must be preserved after a primary sort. Similarly, many inputs may already be partially or fully sorted, and some algorithms can take advantage of such structure. Moreover, applications often involve sorting not only integers, but also long integers, floating point numbers, or doubles. In environments with limited memory resources, algorithms with in-place behavior or lower space complexity may be preferable. This project investigates not just the theoretical complexity, but practical trade-offs across different algorithmic dimensions. It is designed to provide insights into which algorithms perform best under which conditions, and to expose how data characteristics influence performance and correctness. This project focuses solely on CPU-based implementations to reveal practical algorithmic trade-offs without relying on hardware-level optimizations.

## 2 BASIC SORTING ALGORITHMS

In this section, we review six classical sorting algorithms: Merge Sort [9], Heap Sort [6], Bubble Sort [7], Insertion Sort [9], Selection Sort [9], and Quick Sort [8].

### 2.1 Merge Sort

Merge Sort is a classical divide-and-conquer algorithm introduced by **John von Neumann in 1945**. It recursively splits the input array into two halves, sorts them independently, and then merges the sorted halves. This top-down recursive structure ensures consistent performance across all input distributions, making it suitable for general-purpose use.

**Algorithm 1** Merge Sort

```
Input: Array A, indices left, right
if left < right then
    mid ← ⌊(left + right)/2⌋
    MergeSort(A, left, mid)
    MergeSort(A, mid + 1, right)
    Merge(A, left, mid, right)
```

Merge Sort is particularly effective for **linked lists** and **external sorting** (e.g., large disk-based files), as it does not rely on random access and can process data sequentially. **Time Complexity** remains the same in all cases: Best: $O(n \log n)$, Average: $O(n \log n)$, Worst: $O(n \log n)$. **Space Complexity** is $O(n)$ due to the auxiliary array required for merging. **Stability:** Merge Sort is stable, preserving the relative order of equal elements. **In-place:** No; it requires additional memory proportional to the input size.

### 2.2 Heap Sort

Heap Sort is a comparison-based sorting algorithm that uses a **binary heap** data structure to sort elements. It first builds a **max-heap** from the input array, and then repeatedly extracts the maximum element and moves it to the end of the array. This process continues until the entire array is sorted in-place. The algorithm was introduced by **J. W. J. Williams in 1964** and further optimized by **R. W. Floyd**.

**Algorithm 2** Heap Sort

```
Input: Array A of size n
BuildMaxHeap(A, n)
for i = n − 1 downto 1 do
    Swap A[0] and A[i]
    heapSize ← heapSize − 1
    MaxHeapify(A, 0, heapSize)
```

Unlike Merge Sort, Heap Sort does not require additional memory and operates entirely **in-place**. However, it is **not stable**, as it swaps elements without regard to their original positions. Additionally, its **non-sequential memory access** pattern often leads to suboptimal cache performance. Nonetheless, it provides a reliable **worst-case time complexity of** $O(n \log n)$, making it suitable for applications where guaranteed upper bounds are important. **Time Complexity** in all cases is: Best: $O(n \log n)$, Average: $O(n \log n)$, Worst: $O(n \log n)$. **Space Complexity** is $O(1)$ due to in-place operations. **Stability:** Heap Sort is not stable. **In-place:** Yes; it performs sorting within the input array.

### 2.3 Bubble Sort

Bubble Sort is a simple comparison-based algorithm that repeatedly steps through the array, compares adjacent elements, and swaps them if they are in the wrong order. This process continues until the array is fully sorted. The algorithm gets its name because smaller elements gradually "bubble up" to the front of the array through successive swaps.

Although it is easy to implement and understand, Bubble Sort is highly inefficient for large datasets due to its **quadratic time complexity**. It is primarily used for **educational purposes**. Although

**Algorithm 3** Bubble Sort

```
Input: Array A of size n
for i = 0 to n − 2 do
    for j = 0 to n − i − 2 do
        if A[j] > A[j + 1] then
            Swap A[j] and A[j + 1]
```

it can achieve **linear time** $O(n)$ when the input is already sorted, this requires an **early termination check**—which is not included in the above pseudocode. **Time Complexity** is: Best: $O(n)$ (with early termination), Average: $O(n^2)$, Worst: $O(n^2)$. **Space Complexity** is $O(1)$, as it performs sorting in-place. **Stability:** Bubble Sort is stable, preserving the order of equal elements. **In-place:** Yes; it performs all swaps within the input array.

## 2.4 Insertion Sort

Insertion Sort builds the sorted array one element at a time by repeatedly picking the next element from the input and inserting it into its correct position among the previously sorted elements. It is intuitive and performs efficiently on **small or nearly sorted datasets**, making it a good choice for simple use cases or as a subroutine in hybrid algorithms.

**Algorithm 4** Insertion Sort

```
Input: Array A of size n
for i = 1 to n − 1 do
    key ← A[i]
    j ← i − 1
    while j ≥ 0 and A[j] > key do
        A[j + 1] ← A[j]
        j ← j − 1
    A[j + 1] ← key
```

This algorithm is particularly effective when the input is already **partially sorted**, as it requires fewer comparisons and shifts. In the best case, when the array is sorted, each element is simply compared once, resulting in **linear time** $O(n)$. It is also **stable** and **in-place**, making it well-suited for memory-constrained environments. **Time Complexity** is: Best: $O(n)$ (already sorted), Average: $O(n^2)$, Worst: $O(n^2)$. **Space Complexity** is $O(1)$, as all operations are done within the input array. **Stability:** Insertion Sort is stable. **In-place:** Yes; sorting is performed in-place.

## 2.5 Selection Sort

Selection Sort repeatedly selects the minimum (or maximum) element from the unsorted portion and moves it to the beginning (or end) of the sorted portion. Unlike Insertion Sort, it performs fewer swaps, but makes significantly more comparisons. Its structure is simple and intuitive, which makes it useful for teaching sorting principles, though not for performance-critical applications.

Selection Sort is inefficient for large datasets because its **quadratic time complexity** $O(n^2)$ is incurred regardless of input order. Furthermore, it is **not stable**—swapping can disrupt the order of equal elements. However, it is **in-place** and performs a minimal number of writes, which can be useful in environments where memory writes are costly. **Time Complexity** in all cases is: Best: $O(n^2)$, Average: $O(n^2)$, Worst: $O(n^2)$. **Space Complexity** is $O(1)$, as it operates directly on the input array. **Stability:** Selection Sort is not stable. **In-place:** Yes; sorting occurs within the input array.

**Algorithm 5** Selection Sort

```
Input: Array A of size n
for i = 0 to n − 2 do
    min ← i
    for j = i + 1 to n − 1 do
        if A[j] < A[min] then
            min ← j
    if min ≠ i then
        Swap A[i] and A[min]
```

## 2.6 Quick Sort

Quick Sort is a divide-and-conquer algorithm that recursively partitions the array around a pivot [2, pp. 170–180]. Elements less than the pivot are moved to its left, and elements greater than the pivot to its right. In the CLRS version, the **last element is used as the pivot**, which can lead to highly unbalanced partitions on already sorted data.

**Algorithm 6** Quick Sort

```
Input: Array A, indices low, high
if low < high then
    pivot ← A[high]
    i ← low − 1
    for j = low to high − 1 do
        if A[j] ≤ pivot then
            i ← i + 1
            Swap A[i] and A[j]
    Swap A[i + 1] and A[high]
    p ← i + 1
    QuickSort(A, low, p − 1)
    QuickSort(A, p + 1, high)
```

This structure makes Quick Sort simple and fast on average, with **expected time complexity of** $O(n \log n)$ when balanced partitions are formed. However, if poor pivot choices result in unbalanced splits (e.g., always selecting the last element on sorted input), it degrades to **worst-case** $O(n^2)$. Randomized pivot selection is commonly used in practice to avoid this.

**Time Complexity** is: Best: $O(n \log n)$, Average: $O(n \log n)$, Worst: $O(n^2)$ (e.g., sorted input with fixed pivot). **Space Complexity** is $O(\log n)$ due to recursive stack frames. **Stability:** Quick Sort is not stable. **In-place:** Yes; it operates within the original array. **First Described By:** C. A. R. Hoare in 1961.

# 3 ADVANCED SORTING ALGORITHMS

In this section, we review six advanced or modern sorting algorithms: Library Sort [1], Tim Sort[1], Cocktail Shaker Sort [9], Comb Sort [4], Tournament Sort [12], and Introsort [10].

## 3.1 Library Sort

Library Sort, also known as Gapped Insertion Sort, improves upon classical Insertion Sort by maintaining evenly spaced gaps within an auxiliary array [1]. This reduces the number of element shifts during insertion and improves average-case performance.

We follow the practical scheme proposed by Faujdar and Ghrera [5], using an auxiliary array of size $(1+\varepsilon)n$ initially filled with sentinel gaps. The first element is placed at the center, and subsequent elements are inserted using gap-aware binary search on occupied positions. If the target is occupied, elements shift right until a gap is found. After every $2^i$ insertions, a rebalancing step evenly redistributes elements.

**Algorithm 7** Library Sort

**Input:** Array $A$ of size $n$
Initialize $G \leftarrow$ empty array of size $(1+\varepsilon)n$ filled with GAPs
Insert $A[0]$ into center of $G$
Set $inserted \leftarrow 1$, $round \leftarrow 0$
**for** $i = 1$ to $n-1$ **do**
    **if** $inserted = 2^{round}$ **then**
        Rebalance $G$
        $round \leftarrow round + 1$
    Use gap-aware binary search to find insert index $i$
    **while** $G[i]$ is occupied **do**
        $i \leftarrow i + 1$
    Shift elements to make space at $i$
    Insert $A[i]$ at position $i$
    $inserted \leftarrow inserted + 1$
**return** $G$ with gaps removed

Library Sort has **average-case time complexity** $O(n \log n)$, but may degrade to $O(n^2)$ due to excessive shifts. In ideal conditions, it can reach $O(n)$ in the best case. It is **not stable**, as rebalancing disrupts the order of equal keys, and it is **not in-place**, requiring $O((1+\varepsilon)n)$ memory.

In practice, **sorted inputs often trigger worst-case behavior** by inducing dense insertions and frequent rebalancing. While asymptotically appealing, these limitations confine its use to experimental or pedagogical contexts.

## 3.2 Tim Sort

Tim Sort is a hybrid sorting algorithm that combines the strengths of Insertion Sort and Merge Sort. Originally designed by Tim Peters for Python in 2002, it is now the default in Python, Java, and Android due to its practical performance.

The algorithm partitions the input array into segments called *runs*, which are either naturally ordered or explicitly sorted using Insertion Sort. These sorted runs are then merged in a bottom-up manner, similar to Merge Sort.

[1]https://mail.python.org/pipermail/python-dev/2002-July/026837.html

**Algorithm 8** Tim Sort

**Input:** Array $A$ of size $n$
Partition $A$ into runs of size RUN
**for** each run **do**
    Sort the run using Insertion Sort
Let run size = RUN
**while** run size $< n$ **do**
    **for** each pair of adjacent runs **do**
        Merge them using Merge Sort logic
    Double the run size
**return** fully merged and sorted array

Tim Sort leverages the simplicity of Insertion Sort on small or nearly sorted subarrays and the efficiency of Merge Sort for large-scale merging. It guarantees a **worst-case time complexity of** $O(n \log n)$ and performs optimally on real-world data, often achieving **best-case** $O(n)$ when the input is already sorted. The algorithm is also **stable**, preserving the relative order of equal elements.

However, Tim Sort is **not in-place**, requiring $O(n)$ **auxiliary space** for merging. Its adaptive nature and reliable performance explain its adoption in modern standard libraries.

## 3.3 Cocktail Shaker Sort

Cocktail Shaker Sort, also known as Bidirectional Bubble Sort or Ripple Sort, is a variation of Bubble Sort that improves its performance slightly by sorting in both directions within each pass. It was first introduced by Edward H. Friend in 1956 [7].

In standard Bubble Sort, larger elements "bubble" toward the end of the array via repeated adjacent swaps. Cocktail Shaker Sort enhances this behavior by also moving smaller elements toward the beginning of the list during the same iteration. Each full iteration consists of a forward pass (left to right) followed by a backward pass (right to left), which helps reduce the number of necessary iterations when the array is partially sorted.

**Algorithm 9** Cocktail Shaker Sort

**Input:** Array $A$ of size $n$
Set $left \leftarrow 0$, $right \leftarrow n-1$
Set $swapped \leftarrow$ true
**while** $swapped$ is true **do**
    $swapped \leftarrow$ false
    **for** $i = left$ to $right - 1$ **do**
        **if** $A[i] > A[i+1]$ **then**
            Swap $A[i]$ and $A[i+1]$
            $swapped \leftarrow$ true
    $right \leftarrow right - 1$
    **for** $i = right$ downto $left + 1$ **do**
        **if** $A[i] < A[i-1]$ **then**
            Swap $A[i]$ and $A[i-1]$
            $swapped \leftarrow$ true
    $left \leftarrow left + 1$
**return** $A$

Cocktail Shaker Sort retains the simplicity and stability of Bubble Sort, but reduces redundant passes by checking both ends of the list in each iteration. While it remains inefficient for large unsorted data due to its **quadratic time complexity of** $O(n^2)$ in the average and worst cases, it performs slightly better on **partially sorted inputs**. When the array is already sorted, it terminates early, achieving **best-case performance of** $O(n)$. The algorithm is **stable**, preserves element order for equal values, and is **in-place**, requiring only constant $O(1)$ **auxiliary space**.

## 3.4 Comb Sort

Comb Sort is a refinement of Bubble Sort introduced by Włodzimierz Dobosiewicz in 1980. It addresses a key inefficiency of Bubble Sort—turtles, which are small elements near the end of the list that require many iterations to move forward.

The core idea is to compare and swap elements that are a fixed distance apart, called the gap, which decreases over time. Initially, the gap is set to the length of the array and is reduced by a shrink factor—typically around 1.3—after each pass. As the gap approaches 1, Comb Sort behaves similarly to Bubble Sort but with most large out-of-place elements already moved closer to their correct positions.

**Algorithm 10** Comb Sort

```
Input: Array A of size n
Set gap ← n, shrink ← 1.3, sorted ← false
while not sorted do
    gap ← ⌊gap/shrink⌋
    if gap ≤ 1 then
        gap ← 1
        sorted ← true
    for i = 0 to n − gap − 1 do
        if A[i] > A[i + gap] then
            Swap A[i] and A[i + gap]
            sorted ← false
return A
```

Comb Sort improves the efficiency of Bubble Sort, particularly on inputs with many small values at the end. Although it still exhibits **worst-case time complexity of** $O(n^2)$, its **average-case performance is** $O(n \log n)$ when using an optimal shrink factor. It also achieves near-**linear time** $O(n)$ in the best case when the input is already sorted.

The algorithm is **not stable**, as swaps may reorder equal elements, but it is **in-place** and requires only $O(1)$ **auxiliary space**.

## 3.5 Tournament Sort

Tournament Sort is a comparison-based sorting algorithm that organizes input elements into a complete binary tree structure, known as a tournament tree. Each element is initially placed at a leaf, and each internal node stores the smaller (i.e., "winning") of its two children. The minimum element is thus located at the root.

After extracting the root, the winner's original leaf is replaced with a sentinel value, and the path from that leaf to the root is updated to reflect the new tournament result. This process is repeated $n$ times until all elements are extracted in sorted order.

**Algorithm 11** Tournament Sort

```
Input: Array A of size n
Build complete binary tree T with n leaves storing elements of A
for each internal node do
    Store minimum of its two children
for i = 1 to n do
    Let winner ← T[1] (the root)
    Output winner to result
    Replace winner's original leaf with ∞
    Update tree values upward along winner's path
return sorted result
```

Tournament Sort has a consistent **time complexity of** $O(n \log n)$ in all cases, as each insertion and update requires traversing from a leaf to the root in a binary tree of depth $\log n$. However, the overhead of maintaining a complete tree makes it **slower in practice** than most $O(n \log n)$ sorts.

The algorithm is **not in-place**, requiring $O(n)$ **auxiliary space** for the tree, which contains approximately $2n$ nodes. While it can be made **stable** by consistently breaking ties in favor of the left child, such behavior depends on implementation details and language-specific min-comparison behavior. Due to its overhead, Tournament Sort is rarely used for internal sorting but remains useful for external sorting and stream merging.

## 3.6 Introsort

Introspective Sort, or Introsort, is a hybrid sorting algorithm introduced by **David Musser** that starts with Quick Sort and switches to Heap Sort if the recursion depth exceeds a threshold (typically $2 \log n$), ensuring **worst-case time complexity** $O(n \log n)$ while preserving Quick Sort's average-case performance.

It applies Quick Sort recursively, monitoring the depth to prevent unbalanced partitions. If the depth limit is reached, it falls back to Heap Sort. For small subarrays, Insertion Sort is used instead. This adaptive approach balances speed and reliability, making Introsort suitable for general-purpose use.

**Algorithm 12** Introsort

```
procedure IntroSort(A, p, r, depth_limit)
    if r − p + 1 ≤ threshold then
        Use Insertion Sort on A[p ... r]
    else if depth_limit = 0 then
        Use Heap Sort on A[p ... r]
    else
        q ← Partition(A, p, r)
        IntroSort(A, p, q − 1, depth_limit − 1)
        IntroSort(A, q + 1, r, depth_limit − 1)
```

By adapting to both input structure and recursion depth, Introsort avoids Quick Sort's worst-case scenarios while remaining fast on typical inputs. It is **not stable**, but **in-place** and requires only $O(\log n)$ **auxiliary space**. It is the algorithm behind `std::sort` in C++ standard libraries due to its performance and robustness.

# 4 EXPERIMENTAL RESULTS AND ANALYSIS

## 4.1 Experimental Environment

All benchmarks were conducted on a **MacBook Pro (2023, 14-inch)** equipped with an **Apple M2 Pro** chip featuring a **10-core CPU**, running **macOS 15.4**. All sorting algorithms were implemented in **C++17** and compiled using **Apple Clang** (g++) with the -O2 optimization flag to ensure reasonable performance tuning.

The benchmarking framework was written in **Python**, and automated the full evaluation pipeline including **compilation**, **execution**, **timing measurements** [11], and **output validation**.

## 4.2 Performance Across Algorithms

***Method.*** Each sorting algorithm was evaluated on the same randomly generated dataset of size $n = 10^6$. The benchmark was repeated 10 times per algorithm, and the average runtime was recorded. Figure 1 shows the latency for all methods, revealing clear performance tiers and highlighting the relative efficiency of each algorithm.

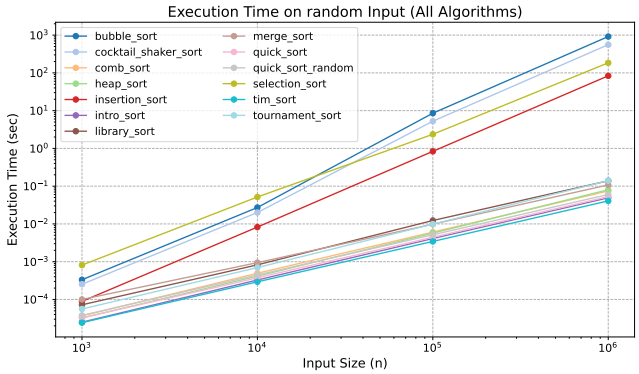


**Figure 1: Execution Time for Random Inputs**

***Algorithms with log-linear complexity.*** The following algorithms have average-case time complexity of $O(n \log n)$ and perform efficiently even on large input sizes.

**Merge Sort** divides the array into halves recursively and merges sorted subarrays in linear time. Its consistent and stable performance makes it a reliable benchmark.

**Heap Sort** builds a max heap and repeatedly extracts the maximum element. It avoids extra memory allocation, but performs slightly slower due to non-local access patterns.

**Quick Sort** partitions the array using a fixed pivot, last element. Though it can degrade on sorted inputs, it performs well on random data with balanced splits.

**Quick Sort (Random Pivot)** selects the pivot randomly to avoid worst-case scenarios. It achieves stable log-linear performance regardless of input structure.

**Intro Sort** begins as Quick Sort and switches to Heap Sort if recursion depth exceeds a threshold. This hybrid approach guarantees worst-case log-linear behavior while maintaining practical speed.

**Tim Sort** detects ascending or descending runs and merges them efficiently. By combining Insertion Sort for small runs and Merge Sort for merging, it achieves excellent real-world performance with log-linear guarantees.

**Library Sort** inserts elements into a partially filled array using binary search and rebalancing. Our improved version approaches log-linear time, though its gap management still incurs overhead compared to other optimized algorithms.

**Comb Sort** improves on Bubble Sort by eliminating small disordered elements early using a shrinking gap. While not strictly log-linear in the worst case, it performs comparably to faster algorithms in random input scenarios.

***Algorithms with quadratic complexity.*** These algorithms have time complexity of $O(n^2)$ and become inefficient as the input size grows.

**Bubble Sort** repeatedly swaps adjacent elements. This results in $n^2$ comparisons, making it highly inefficient.

**Insertion Sort** builds a sorted section by shifting larger elements to the right. Although effective for nearly sorted data, it is costly on random input.

**Selection Sort** selects the smallest remaining element in each pass. It performs a fixed number of comparisons and is consistently among the slowest.

**Cocktail Shaker Sort** improves on Bubble Sort by alternating forward and backward passes. This bidirectional approach slightly reduces the number of iterations, but the overall time complexity remains quadratic, with performance still far behind log-linear algorithms.

**Tournament Sort** theoretically has log-linear complexity, but in practice performs poorly due to heavy overhead from maintaining and updating a complete binary tree. For each extraction, multiple full-width data copies occur, resulting in performance closer to that of quadratic algorithms under random input.

**Table 1: Latency for Random Inputs (in seconds)**

| Algorithm | $n = 10^3$ | $n = 10^4$ | $n = 10^5$ | $n = 10^6$ |
|---|---|---|---|---|
| Merge Sort | 0.0001000 | 0.0009431 | 0.0098381 | 0.1078913 |
| Heap Sort | 0.0000323 | 0.0004130 | 0.0057043 | 0.0803684 |
| Bubble Sort | 0.0003324 | 0.0271842 | 8.5452850 | 916.6450000 |
| Insertion Sort | 0.0000888 | 0.0082530 | 0.8353202 | 83.6130300 |
| Selection Sort | 0.0008122 | 0.0516557 | 2.3630430 | 184.6850000 |
| Quick Sort | 0.0000325 | 0.0003742 | 0.0044425 | 0.0528438 |
| Quick Sort (Random) | 0.0000378 | 0.0004457 | 0.0051551 | 0.0602114 |
| Library Sort | 0.0000721 | 0.0008245 | 0.0123183 | 0.1386819 |
| Cocktail Shaker Sort | 0.0002524 | 0.0200263 | 5.2499610 | 558.3070000 |
| Tim Sort | 0.0000243 | 0.0002927 | 0.0034355 | 0.0407254 |
| Comb Sort | 0.0000367 | 0.0004937 | 0.0060670 | 0.0745422 |
| Tournament Sort | 0.0000558 | 0.0006971 | 0.0098557 | 0.1399134 |
| Intro Sort | 0.0000250 | 0.0003273 | 0.0040994 | 0.0491607 |

***Summary.*** As shown in Table 1 and Figure 1, the algorithms clearly separate into two performance tiers. Among all tested methods, **Tim Sort** consistently outperformed others across all input sizes, confirming its exceptional efficiency on unstructured data. **Intro Sort** ranked second overall in performance and is notable for being the basis of `std::sort` in the C++ standard library.

## 4.3 Performance Sensitivity to Input Order

***Method.*** Each algorithm was tested on three types of input: 50% partially sorted, fully sorted in ascending order, and fully sorted in descending order. Arrays were of type `int`, and execution time was measured for $n \in \{10^3, 10^4, 10^5, 10^6\}$. Each configuration was run 10 times and averaged.

Grouped line plots illustrate how input order affects runtime, highlighting which algorithms benefit from existing order and which ones degrade under structured inputs.

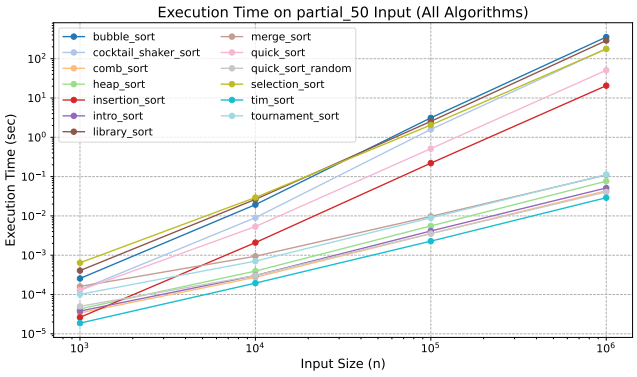


**Figure 2: Execution Time for 50% Partial Sorted Inputs**

***Unexpected Behavior at Small Scales.*** On partially sorted inputs of size $n = 10^3$, **Insertion Sort** ranked as the second fastest algorithm, outperforming several $O(n \log n)$ methods. This behavior can be attributed to its low overhead and tight inner loop. For small inputs, the absolute number of required shifts is limited, and branch prediction and memory locality are highly favorable. In contrast, more complex algorithms like Merge Sort or Heap Sort incur fixed setup costs (e.g., recursion, heapification) that outweigh their asymptotic advantages at small scales. Thus, despite its quadratic worst-case complexity, Insertion Sort can outperform log-linear algorithms when input size is small and partially ordered.

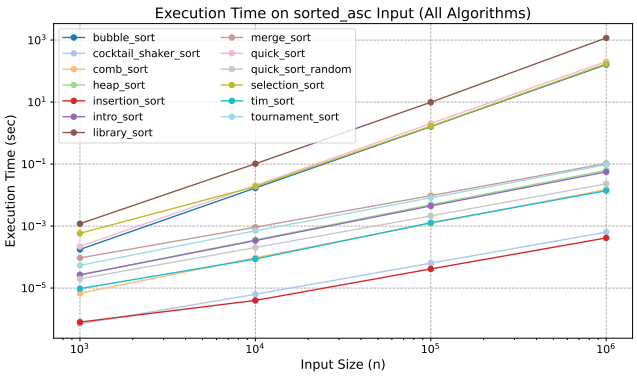


**Figure 3: Execution Time for Ascending Sorted Inputs**

***Ascending Sorted Inputs.*** Figure 3 shows the performance of all algorithms on ascending sorted inputs. In this scenario, several algorithms achieve their best-case behavior due to minimal disorder in the data.

**Insertion Sort** reaches its optimal $O(n)$ time, as no shifts are needed. Our measurements confirm that it becomes one of the fastest algorithms for sorted input, with latency dropping to nearly zero even at large scales.

**Cocktail Shaker Sort** terminates after a single forward and backward pass, achieving near-linear runtime.

**Quick Sort** (with last-element pivot) performs poorly. On sorted input, its partitions become maximally unbalanced, leading to worst-case $O(n^2)$ behavior. This causes significant latency spikes despite the input being fully ordered.

**Library Sort** also slows down on sorted inputs. Although it has a theoretical linear best case, our measurements show that frequent insertions into dense areas and delayed rebalancing cause it to perform worse than on random input.

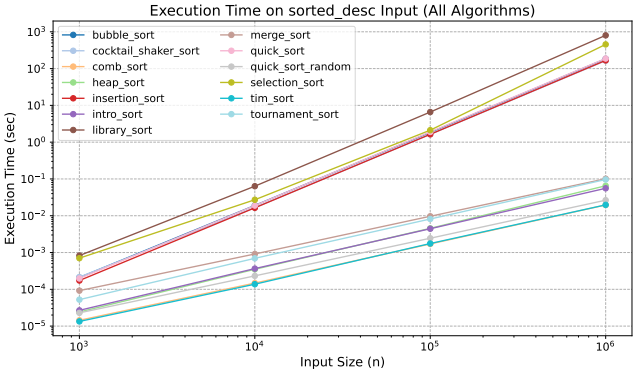


**Figure 4: Execution Time for Descending Sorted Inputs**

***Descending Sorted Inputs.*** Figure 4 illustrates the performance of each algorithm on reverse-sorted input. This case often triggers worst-case behavior in insertion-based or pivot-sensitive algorithms, leading to significant slowdowns.

**Insertion Sort** suffers a dramatic performance drop. On descending input, every new element must be shifted across the entire sorted portion, resulting in $O(n^2)$ time. As shown in our measurements, latency increased by over 400× compared to the ascending case.

**Bubble Sort** similarly reaches its worst-case behavior. The absence of early termination forces all comparisons and swaps, making it one of the slowest algorithms on descending input.

**Quick Sort** again performs poorly due to unbalanced partitions caused by fixed pivot selection. The recursive depth increases significantly, and performance degrades to quadratic time, nearly doubling its latency compared to the random input.

**Tim Sort**, in contrast, remains robust. It quickly identifies long descending runs and reverses them efficiently. As a result, its performance remains nearly identical to the ascending case.

## 4.4 Performance Sensitivity to Data Type

***Method.*** Each algorithm was benchmarked on arrays of size $n = 10^6$, using identical values cast to four types: `int`, `long long`, `float`, and `double`. All values were uniformly sampled integers in the range $[0, 10^6)$, ensuring consistent content across types.

This isolates the impact of data representation and arithmetic overhead, while avoiding rounding artifacts from floating-point inputs.

**Table 2: Latency by Data Type for $10^6$ Random Inputs**

| Algorithm | int | long long | float | double |
|---|---|---|---|---|
| Merge Sort | 0.1052s | 0.1172s | 0.1288s | 0.1405s |
| Heap Sort | 0.0817s | 0.0900s | 0.1005s | 0.1064s |
| Bubble Sort | 921.3660s | 926.2290s | 1113.9800s | 1127.9900s |
| Insertion Sort | 72.1336s | 77.4562s | 74.7377s | 81.7258s |
| Selection Sort* | 175.8300s | 232.0120s | 245.9670s | 293.1080s |
| Quick Sort | 0.0534s | 0.0526s | 0.0650s | 0.0652s |
| Quick Sort (Random) | 0.0584s | 0.0581s | 0.0703s | 0.0706s |
| Library Sort | 0.1384s | 0.1528s | 0.1607s | 0.1808s |
| Cocktail Shaker Sort | 560.8050s | 562.9870s | 683.0070s | 702.3440s |
| Tim Sort | 0.0403s | 0.0442s | 0.0638s | 0.0667s |
| Comb Sort | 0.0760s | 0.0759s | 0.0913s | 0.0927s |
| Tournament Sort* | 0.2014s | 0.2679s | 0.2511s | 0.4492s |
| Intro Sort | 0.0492s | 0.0494s | 0.0611s | 0.0615s |

***Impact of Numeric Type on Sorting Performance.*** Although `long long` occupies 64 bits and `int` only 32 bits, most algorithms exhibited nearly identical performance. As a result, integer bit-width had negligible impact on sorting latency.

***Floating-Point Computation as the Primary Overhead.*** Compared to integer types, both `float` and `double` introduced consistent slowdowns across all algorithms. This overhead stems from two main sources: (1) floating-point comparisons are generally slower due to hardware complexity, and (2) arithmetic operations incur additional cost from rounding, normalization, and precision handling. Interestingly, `double` did not perform significantly worse than `float`, suggesting that logic overhead—not data size—is the dominant factor. This slowdown is further amplified on memory-bound algorithms, where increased operand size results in higher cache miss rates and memory transfer latency. On modern architectures, even small increases in per-element size can lead to disproportionate performance drops due to vectorization and pipeline stalls. Thus, algorithmic sensitivity to data type is not just computational but also deeply architectural.

***Exceptions: Bit-Width as a Bottleneck in Select Algorithms.*** While most algorithms are bottlenecked by comparisons, a few are sensitive to data size due to frequent element movement. In particular, **Selection Sort** performs $O(n^2)$ swaps, so increasing element width leads to more memory traffic and cache stress. **Tournament Sort** maintains a binary tree of $2n$ full-width elements, amplifying memory transfer overhead with wider types. In these cases, performance degradation is primarily due to memory movement rather than computation.

## 4.5 Stability Analysis

To assess sorting stability, we generated inputs of 1,000 random `(value, index)` pairs with duplicate values $[0, 50)$ but unique original indices. Each algorithm was run 10 times, and marked **Stable** only if it preserved the relative order of equal elements in all trials.

**Table 3: Stability Results**

| Algorithm | Stability | Algorithm | Stability |
|---|---|---|---|
| Merge Sort | Stable | Library Sort | Unstable |
| Heap Sort | Unstable | Cocktail Shaker Sort | Stable |
| Bubble Sort | Stable | Tim Sort | Stable |
| Insertion Sort | Stable | Comb Sort | Unstable |
| Selection Sort | Unstable | Tournament Sort | Stable* |
| Quick Sort | Unstable | Intro Sort | Unstable |

***Stable Algorithms.*** Merge Sort, Insertion Sort, Bubble Sort, Cocktail Shaker Sort, and Tim Sort consistently preserved order among equal elements.

***Unstable Algorithms.*** Heap Sort, Selection Sort, Quick Sort, Library Sort, Comb Sort, and Intro Sort all showed unstable behavior, mainly due to swaps and partitioning logic that disrupt relative positions.

***Stability Sensitivity in Tournament Sort.*** Tournament Sort was stable in our tests, but this depends on how tie-breaking between equal values is handled. Since `std::min` does not guarantee a consistent order for equal inputs [3], we explicitly enforced left-child preference to ensure stable behavior.

## 4.6 Memory Usage Across Algorithms

***Method.*** Each algorithm was tested on random input of size $n = 10^5$ using integer arrays. Memory usage was sampled at three points: before loading the input, after loading the input into a `std::vector`, and at peak usage during sorting. We define **vector_only** as the difference between memory after and before loading the input, and **sort_overhead** as the difference between the sorting peak and the vector baseline.

In all memory trace graphs, the blue segment represents memory usage during vector construction, and the red segment shows memory fluctuations during sorting.

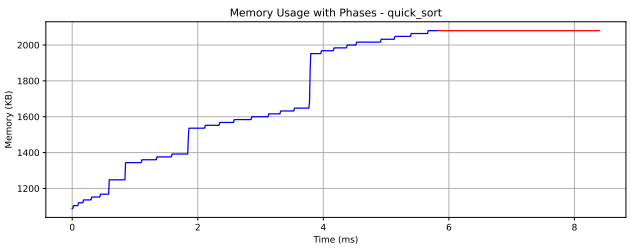


**Figure 5: Quick Sort Memory Usage (KB)**

**Table 4: Memory Usage (KB) for Random Inputs ($n = 10^5$)**

| Algorithm | Vector Only | Sort Overhead |
|---|---|---|
| Merge Sort | 1043.2 | 947.2 |
| Heap Sort | 1024.0 | 0.0 |
| Bubble Sort | 1028.8 | 0.0 |
| Insertion Sort | 1014.4 | 0.0 |
| Selection Sort | 1017.6 | 0.0 |
| Quick Sort | 1020.8 | 0.0 |
| Quick Sort (Random) | 1022.4 | 0.0 |
| Library Sort | 1017.6 | 2601.6 |
| Tim Sort | 1024.0 | 476.8 |
| Cocktail Shaker Sort | 1014.4 | 0.0 |
| Comb Sort | 1011.2 | 0.0 |
| Tournament Sort | 1011.2 | 1454.4 |
| Intro Sort | 1020.8 | 0.0 |

***In-Place Sorting Verification.*** The `Vector Only` column indicates memory used to store 100,000 integers in a `std::vector`, typically around 1010–1040 KB. Algorithms such as **Bubble Sort**, **Insertion Sort**, **Selection Sort**, **Heap Sort**, and **Quick Sort** (Figure 5) reported zero additional memory usage, confirming they are in-place.

***Expected Linear Overheads.*** **Merge Sort** showed 947 KB of overhead, which closely matches the memory required to allocate an auxiliary array of the same size as the input vector. **Tim Sort**, a hybrid of Insertion and Merge Sort, added 476.8 KB of overhead. This moderate increase likely arises from allocating temporary buffers to store sorted runs and merge segments efficiently, though not for the full array at once. **Tournament Sort** incurred over 1.4 MB of additional memory usage. This is expected given its use of a binary tree structure for comparisons, which requires roughly $2n$ space to represent internal nodes and leaves, effectively doubling the input size in memory.

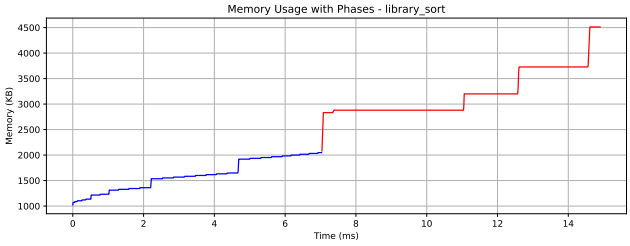


**Figure 6: Library Sort Memory Usage (KB)**

***Memory Growth Pattern in Library Sort.*** As shown in Figure 6, the memory trace of Library Sort exhibits four distinct jumps during the sorting phase. These spikes correspond to repeated reallocations triggered by gap exhaustion as elements are inserted. Each time the current array becomes too full, a larger array is allocated and the existing elements are copied over. The final jump likely results from compaction or copying of the sorted output.

## 4.7 Accuracy Analysis

***Method.*** Accuracy was evaluated using the *adjacent pair violation rate*, computed as the proportion of index pairs $(i, i+1)$ in the sorted output where $A[i] > A[i+1]$. For a result array of size $n$, we counted the number of such violations and defined the accuracy score as:

$$\text{Accuracy} = 1 - \frac{\#\text{ of violations}}{n-1}$$

This metric was measured alongside runtime during the experiments in Sections 4.2 and 4.3.

**Table 5: Accuracy and runtime of Library Sort**

| Size | Input Type | Time (s) | Accuracy (%) |
|---|---|---|---|
| $10^3$ | partial_50 | 0.000406 | 99.20 |
| $10^3$ | random | 0.000072 | 98.80 |
| $10^3$ | sorted_asc | 0.001195 | 100.00 |
| $10^3$ | sorted_desc | 0.000826 | 100.00 |
| $10^4$ | partial_50 | 0.026388 | 99.24 |
| $10^4$ | random | 0.000825 | 99.57 |
| $10^4$ | sorted_asc | 0.102631 | 100.00 |
| $10^4$ | sorted_desc | 0.063526 | 100.00 |
| $10^5$ | partial_50 | 2.538908 | 99.31 |
| $10^5$ | random | 0.012318 | 99.27 |
| $10^5$ | sorted_asc | 9.880547 | 100.00 |
| $10^5$ | sorted_desc | 6.584756 | 100.00 |
| $10^6$ | partial_50 | 289.247667 | 99.02 |
| $10^6$ | random | 0.138682 | 99.13 |
| $10^6$ | sorted_asc | 1175.510000 | 100.00 |
| $10^6$ | sorted_desc | 801.178000 | 100.00 |

***Overall Accuracy Result.*** **All algorithms except Library Sort achieved perfect accuracy (100%) across all inputs.** Table 5 summarizes the accuracy of Library Sort across various input sizes and distributions.

***Error Characteristics of Library Sort.*** Library Sort showed minor violations (typically under 1%) on random and partially sorted inputs. These errors are not due to incorrect final ordering, but rather due to the way we define accuracy: as the absence of adjacent pair inversions. In Library Sort, when a newly inserted element causes a long rightward shift or when rebalancing is delayed, temporary local disorder can occur. This may leave a few inversions unresolved when the algorithm terminates, especially if the insertion density near the end becomes high and a final rebalance does not take place.

***Perfect Accuracy on Sorted Inputs.*** When the input is already sorted—either in ascending or descending order—each new element is inserted in the correct position without needing to shift other elements or resolve conflicts. Because of this, Library Sort can complete without creating any local disorder in the array. As a result, the final output has no adjacent violations, and the algorithm achieves exactly 100% accuracy in these cases.

## References


[1] M. A. Bender, M. Farach-Colton, and M. A. Mosteiro. 2006. Insertion Sort is O(n log n). *Theory of Computing Systems* 39 (2006), 391–397.

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms* (3rd ed.). MIT Press.

[3] cppreference contributors. 2024. std::min - cppreference.com. https://en.cppreference.com/w/cpp/algorithm/min.

[4] W. Dobosiewicz. 1980. An Efficient Variation of Bubble Sort. Unpublished report or workshop paper.

[5] Neetu Faujdar and Satya Prakash Ghrera. 2015. A detailed experimental analysis of library sort algorithm. *2015 Annual IEEE India Conference (INDICON)* (2015), 1–6. doi:10.1109/INDICON.2015.7443165

[6] G. E. Forsythe. 1964. Algorithms. *Commun. ACM* 7, 6 (1964), 347–349.

[7] E. H. Friend. 1956. Sorting on Electronic Computer Systems. *Journal of the ACM (JACM)* 3, 3 (1956), 134–168.

[8] C. A. R. Hoare. 1961. Algorithm 64: Quicksort. *Commun. ACM* 4, 7 (1961), 321.

[9] Donald E. Knuth. 1998. *The Art of Computer Programming: Sorting and Searching, Volume 3.* Addison-Wesley Professional.

[10] David R. Musser. 1997. Introspective Sorting and Selection Algorithms. *Software: Practice and Experience* 27, 8 (1997), 983–993.

[11] Python Software Foundation. 2024. *time — Time access and conversions.* https://docs.python.org/3/library/time.html.

[12] Alexander Stepanov and Allan Kershenbaum. 1986. *Using Tournament Trees to Sort.* Technical Report 86–13. Polytechnical Institute of New York, CATT.