CSE331 - Assignment #1

Aliona Pirozhenko (20232008)
UNIST
South Korea
alionapirozhenko@unist.ac.kr

1 PROBLEM STATEMENT

The goal of this assignment is to write an implementation of 6 conventional and 6 contemporary sorting algorithms, to analyze them using the self-prepared data sets, which are intentionally generated to consist of different numbers of elements. To evaluate the time complexities of the following algorithms:

- Conventional: Merge Sort [8], Heap Sort [4], Bubble Sort [1], Insertion Sort [5], Selection Sort [10], Quick Sort [9]
- Contemporary: Library Sort [7], Tim Sort [11], Cocktail Shaker Sort [2], Comb Sort [3], Tournament Sort [12], Intro Sort [6]

To conduct these experiments on various input data sets and then compare the results of the output execution times and the sorting accuracies.

These Conventional algorithms' analysis requires: implementation of each of them from scratch, without using standard library functions directly in my implementation, and providing clear explanations of my design principles, and analyzing the time complexity for each of the algorithms.

The other 6 algorithms (Contemporary), require me to provide pseudo code for each of the algorithms. Then comparison of characteristics, advantages, disadvantages of using and implementing, and running of the algorithm relative to the other in order to properly evaluate the performance.

This assignment generally aims at improving my understanding of each of the above sorting algorithms and forces me to gain testing experience using the generated data sets. My code implementation is in a GitHub repository. The repository can be accessed at the following link:

https://github.com/alionapi/CSE331-assignment-1

2 BASIC SORTING ALGORITHMS

The Conventional Comparison-Based sorting algorithms are: Merge Sort, Heap Sort, Bubble Sort, Insertion Sort, Selection Sort, and Quick Sort. Here I perform the implementation of each and analyze the time complexity and performance. The implementation is done from scratch and doesn't include any sorting library in the implementation. Along I include the design rationale of every algorithm.

Merge Sort

Description: Merge Sort is a divide-and-conquer algorithm that recursively divides the input array into smaller subarrays, sorts them individually, and merges the sorted subarrays back together.

Time Complexity:

- Best Case: $O(n \log n)$
- Average Case: $O(n \log n)$
- Worst Case: $O(n \log n)$

Design Rationale: Merge Sort's design focuses on splitting the array into smaller segments that are easier to sort. Its recursive nature and stable merge operation make it particularly suitable for sorting linked lists and for applications where stability is required. The consistent O(n log n) performance makes it reliable across all input distributions, though at the cost of additional memory requirements during the merge phase.

Heap Sort

Description: Heap Sort uses a binary heap data structure to organize elements. It repeatedly extracts the largest (or smallest) element from the heap and reconstructs the heap.

Time Complexity:

- Best Case: $O(n \log n)$
- Average Case: $O(n \log n)$
- Worst Case: $O(n \log n)$

Design Rationale: Heap Sort leverages the binary heap data structure to efficiently identify the maximum/minimum element in each iteration. Its in-place nature makes it memory-efficient compared to Merge Sort, though it sacrifices stability. The heapify operation ensures the time complexity remains O(n log n) even in worst-case scenarios.

Bubble Sort

Description: Bubble Sort repeatedly compares adjacent elements in the array and swaps them if they're out of order. This process is repeated until the array is sorted.

Time Complexity:

- Best Case: $O(n)$ (when the array is already sorted)
- Average Case: $O(n^2)$
- Worst Case: $O(n^2)$

Design Rationale: Bubble Sort's design prioritizes simplicity and ease of implementation. It works by repeatedly "bubbling up" the largest element to its correct position. The "early termination" optimization (stopping when no swaps occur) allows it to perform well on nearly sorted data. Despite its simplicity, its quadratic time complexity makes it impractical for all but the smallest datasets.

Insertion Sort

Description: Insertion Sort builds a sorted array one element at a time by picking elements from the unsorted part and inserting them into the correct position in the sorted part.

Time Complexity:

- Best Case: $O(n)$ (when the array is already sorted)
- Average Case: $O(n^2)$
- Worst Case: $O(n^2)$

Design Rationale: Insertion Sort mimics the natural human approach to sorting (like organizing playing cards in hand). Its design emphasizes simplicity and adaptiveness to partially sorted inputs. The algorithm maintains a growing sorted region at the beginning of the array, inserting each new element into its proper place. This approach makes it particularly efficient for small datasets and nearly-sorted arrays.

Selection Sort

Description: Selection Sort repeatedly finds the smallest (or largest) element in the unsorted part of the array and swaps it with the first unsorted element.

Time Complexity:

- Best Case: $O(n^2)$
- Average Case: $O(n^2)$
- Worst Case: $O(n^2)$

Design Rationale: Selection Sort's design focuses on minimizing the number of swaps compared to other simple sorting algorithms like Bubble Sort. It always performs exactly n swaps (where n is the number of elements), making it efficient in environments where write operations are significantly more expensive than read operations. However, its invariant O(n²) time complexity makes it unsuitable for large datasets.

Quick Sort

Description: Quick Sort is a divide-and-conquer algorithm that partitions the array into two subarrays based on a pivot, recursively sorting each partition.

Time Complexity:

- Best Case: $O(n \log n)$
- Average Case: $O(n \log n)$
- Worst Case: $O(n^2)$ (when the pivot selection is poor)

Design Rationale: Quick Sort's design emphasizes practical efficiency and in-place operation. The key to its performance is the partitioning strategy, which divides the array around a pivot element. With good pivot selection, Quick Sort achieves excellent average-case performance and cache locality. Its primary weakness is the potential for quadratic behavior on already-sorted or nearly-sorted arrays when using naïve pivot selection strategies.

3 ADVANCED SORTING ALGORITHMS

The Contemporary Algorithms are: Library Sort, Tim Sort, Cocktail Shaker Sort, Comb Sort, Tournament Sort, Intro Sort. The primary objective of contemporary algorithms is to provide efficient and effective solutions to complex problems. These algorithms aim to improve the speed, accuracy, and overall performance of computational processes, often by leveraging advanced techniques like machine learning and optimization to automate decision-making and achieve optimal results.

Library Sort

Description: Library Sort uses binary search for element placement and dynamically resizes the "buffer" in the sorted array.

Distinct Characteristics: Uses binary search and buffer resizing.

Advantages: Efficient for large datasets due to binary search.

Algorithm 1 Library Sort

```
1:  Procedure librarySort(A)
2:  $S \leftarrow$ empty array of size $2n$ with gaps
3:  insert $A[0]$ into $S$ at the center
4:  for each element in $A[1..n-1]$ do
5:      find insertion position using binary search
6:      shift elements to maintain gap invariant
7:      insert element
8:      if array too full then
9:          rebuild $S$ with more gaps
10:     end if
11: end for
```

Disadvantages: Overhead of resizing buffers makes it less efficient for small datasets.

Theoretical Analysis: Library Sort has an expected time complexity of O(n log n), though its worst-case remains O(n²). Its space complexity is O(n) due to the gap buffer requirements. The performance improvement over insertion sort comes from reduced element shifting during insertions, as gaps provide space for new elements without requiring extensive array reorganization.

Tim Sort

Description: Tim Sort is a hybrid algorithm combining Insertion Sort and Merge Sort.

Algorithm 2 Tim Sort

```
1: Procedure timSort(A)
2: define MIN_RUN
3: for $i \leftarrow 0$ to length(A) in steps of MIN_RUN do
4:     perform insertionSort on A[i to i + MIN_RUN - 1]
5: end for
6: while runs exist do
7:     merge adjacent runs using mergeSort logic
8: end while
```

Distinct Characteristics: Hybrid algorithm optimized for real-world data.

Advantages: Efficient for nearly-sorted data; stable.

Disadvantages: Overhead in managing runs and merging.

Theoretical Analysis: Tim Sort achieves O(n log n) worst-case time complexity while approaching O(n) for datasets with significant existing order. Its space complexity is O(n) due to the merge operations. The algorithm's strength lies in its adaptiveness—identifying and leveraging natural patterns in data to minimize comparisons and moves.

Cocktail Shaker Sort

Description: Cocktail Shaker Sort is a bi-directional variation of Bubble Sort.

Distinct Characteristics: Bi-directional sorting mechanism.

Advantages: Faster than Bubble Sort.

Disadvantages: Inefficient for large datasets ($O(n^2)$).

Theoretical Analysis: Cocktail Shaker Sort maintains the same asymptotic time complexity as Bubble Sort (O(n²)), but can perform

Algorithm 3 Cocktail Shaker Sort

```
Procedure cocktailShakerSort(A)
start ← 0
end ← length(A) −1
swapped ← true
while swapped do
    swapped ← false
    for i ← start to end − 1 do
        if A[i] > A[i + 1] then
            swap A[i] and A[i + 1]
            swapped ← true
        end if
    end for
    end ← end − 1
    for i ← end − 1 downto start do
        if A[i] > A[i + 1] then
            swap A[i] and A[i + 1]
            swapped ← true
        end if
    end for
    start ← start + 1
end while
```

better on certain types of data due to its bi-directional nature. It addresses the slow movement of small elements at the end of the array (the "turtle problem"), but remains inefficient for large datasets compared to O(n log n) algorithms.

Comb Sort

Description: Comb Sort improves Bubble Sort by introducing a gap between elements compared.

Algorithm 4 Comb Sort

```
Procedure combSort(A)
gap ← length(A)
shrink ← 1.3
sorted ← false
while not sorted do
    gap ← int(gap / shrink)
    if gap ≤ 1 then
        gap ← 1
        sorted ← true
    end if
    for i ← 0 to length(A) −gap − 1 do
        if A[i] > A[i + gap] then
            swap A[i] and A[i + gap]
            sorted ← false
        end if
    end for
end while
```

Distinct Characteristics: Gap-based sorting mechanism.

Advantages: Faster than Bubble Sort.

Disadvantages: Not stable; performance depends on gap sequence.

Theoretical Analysis: Comb Sort achieves an average time complexity of O(n log n) to O(n²) depending on the gap sequence used. The shrink factor of 1.3 is empirically determined to give good results. The algorithm's primary advantage is eliminating the "turtles" (small values near the end) that slow down Bubble Sort significantly.

Tournament Sort

Description: Tournament Sort uses a tournament tree to determine the smallest element repeatedly.

Algorithm 5 Tournament Sort

```
Procedure tournamentSort(A)
build tournament tree from A
while tree is not empty do
    extract winner
    replace winner with ∞ or next element
    rebalance tree
end while
```

Distinct Characteristics: Element selection using a tournament tree.

Advantages: Suitable for parallel execution.

Disadvantages: High maintenance overhead of the tree.

Theoretical Analysis: Tournament Sort has O(n log n) time complexity for all cases, similar to Heap Sort. Its space complexity is O(n) due to the explicit tree structure, which makes it less space-efficient than Heap Sort's implicit heap. The algorithm's main theoretical advantage is its parallelization potential rather than sequential performance.

Introsort

Description: Introsort combines Quick Sort, Heap Sort, and Insertion Sort to achieve fast and adaptive sorting.

Algorithm 6 Introsort

```
Procedure introsort(A)
depthLimit ← 2 × $\log_2$(length(A))
introsortHelper(A, 0, length(A) - 1, depthLimit)
```

Distinct Characteristics: Hybrid algorithm ensuring worst-case $O(n \log n)$.

Advantages: Fast, adaptive, efficient for large datasets.

Disadvantages: Complex implementation.

Theoretical Analysis: Introsort guarantees O(n log n) worst-case time complexity while maintaining Quick Sort's excellent average-case performance. It achieves this by monitoring recursion depth and switching to Heap Sort when Quick Sort's worst-case behavior is detected. The space complexity is O(log n) on average due to the controlled recursion depth.

4 EXPERIMENTAL RESULTS AND ANALYSIS

In CLion IDE I wrote a C program to test the above algorithms, analyze them, and evaluate the result.s In Python, I wrote a script to generate the data sets of different sizes (1000, 10000, 100000,

Algorithm 7 Introsort Helper

```
1: Procedure introsortHelper(A, start, end, depthLimit)

2: if $end - start \leq$ threshold then
3:   insertionSort(A[start..end])
4: else if $depthLimit = 0$ then
5:   heapSort(A[start..end])
6: else
7:   $pivot \leftarrow$ partition(A, start, end)
8:   introsortHelper(A, start, pivot - 1, depthLimit - 1)
9:   introsortHelper(A, pivot + 1, end, depthLimit - 1)
10: end if
```

1000000 elements). Each .txt file generated with a name <input type>_<size>.txt . The .txt files are in the project's GitHub repository in data/.

Input Types Used: According to the assignment's requirements.

Type	Description
Random	Fully randomized values
Sorted	Strictly increasing order
Reverse Sorted	Strictly decreasing order
Partially Sorted	80% sorted, 20% random shuffled

Table 1: Input Types

Performance Analysis

Small Inputs (1,000 elements)

For small inputs, several key observations emerge:

- **Fastest Algorithms:** Introsort (0.07ms), Tim Sort (0.09ms), and Quick Sort (0.10ms) demonstrated superior performance on random data.
- **Slowest Algorithms:** Tournament Sort (2.17ms), Bubble Sort (2.09ms), and Selection Sort (1.65ms) performed worst, even at this small scale.
- **Impact of Input Distribution:** Bubble Sort and Insertion Sort achieved near-instantaneous sorting (0ms) on already-sorted data, highlighting their adaptive nature.
- **Problematic Patterns:** Quick Sort showed degraded performance on sorted data (1.60ms), illustrating its worst-case behavior.

The results indicate that for small datasets, the overhead of complex algorithms may outweigh their asymptotic advantages. However, even at this scale, the quadratic algorithms (Bubble, Selection) begin to lag noticeably.

Medium Inputs (10,000 elements)

With medium-sized inputs, algorithm efficiency differences become more pronounced:

- **Fastest Algorithms:** Introsort (0.94ms), Quick Sort (1.00ms), and Tim Sort (1.16ms) maintained their lead on random data.
- **Quadratic Degradation:** Bubble Sort (205.76ms), Selection Sort (97.32ms), and Insertion Sort (54.88ms) exhibited severe performance degradation, demonstrating their $O(n^2)$ scaling.
- **Adaptive Behavior:** Insertion Sort (0.03ms) and Bubble Sort (0.02ms) remained extremely efficient on sorted data.
- **Worst-Case Scenarios:** Quick Sort degraded substantially on sorted data (147.53ms), while Library Sort struggled with reversed data (224.63ms).

At this input size, the practical limitations of quadratic algorithms become clear, with order-of-magnitude performance differences compared to efficient algorithms.

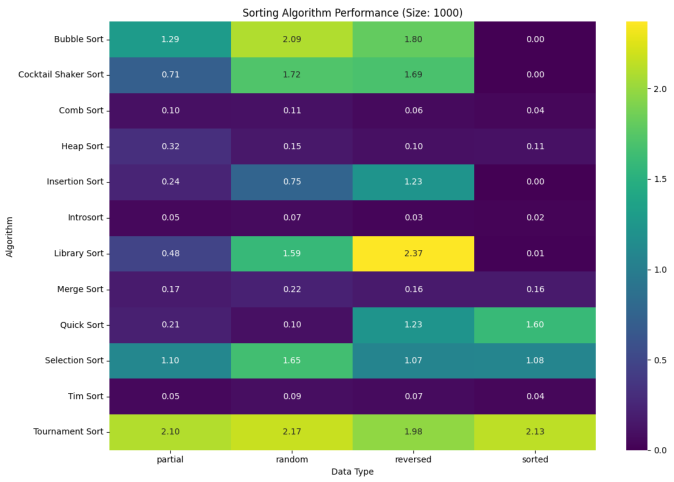

Figure 1: Sorting Algorithm Performance (size: 10000)

Large Inputs (100,000 elements)

With large inputs, the performance divide becomes extreme:

- **Scalable Algorithms:** Quick Sort (11.31ms), Introsort (11.46ms), and Tim Sort (13.43ms) maintained reasonable performance.
- **Quadratic Collapse:** Bubble Sort (15,000ms), Selection Sort (9,000ms), and Insertion Sort (7,000ms) became practically unusable.
- **Contemporary Advantage:** Most contemporary algorithms outperformed classical ones, with the exception of Tournament Sort.

The results at this scale definitively demonstrate the practical importance of algorithm selection for large datasets.

Performance Analysis by Input Distribution

Random Data.

- Quick Sort, Introsort, and Tim Sort consistently performed best across all sizes.
- Library Sort underperformed relative to its theoretical advantages.
- Tournament Sort showed surprisingly poor performance despite its theoretical $O(n \log n)$ complexity.

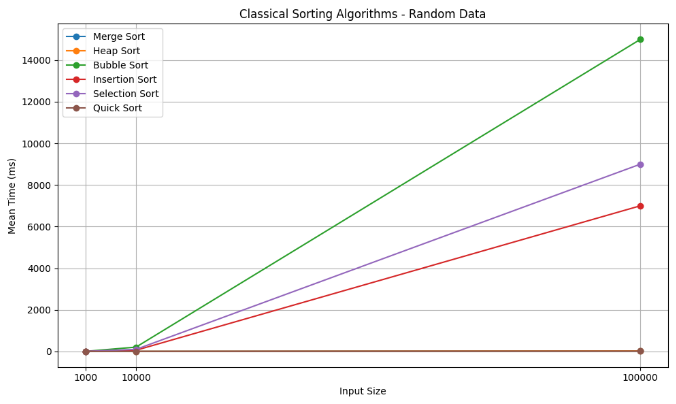

Figure 2: Basic Algorithms - random data

Sorted Data.

- Bubble Sort and Insertion Sort achieved near-instant sorting due to their adaptive nature.
- Quick Sort exhibited its worst-case behavior, performing significantly worse than on random data.
- Tim Sort and Introsort adapted well, maintaining excellent performance.

Reversed Data.

- Library Sort struggled significantly, performing worse than expected.
- Comb Sort showed excellent performance, highlighting its advantage over Bubble Sort for this case.
- Introsort and Tim Sort remained consistently efficient.

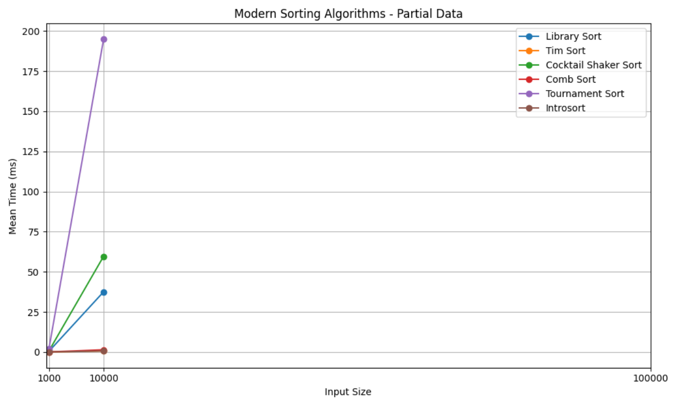

Figure 3: Advanced Algorithms - random data

Partially Sorted Data.

- Tim Sort excelled, showcasing its design optimization for real-world data patterns.
- Insertion Sort performed substantially better than on random data, confirming its adaptive nature.
- Selection Sort showed no improvement, indicating its non-adaptive design.

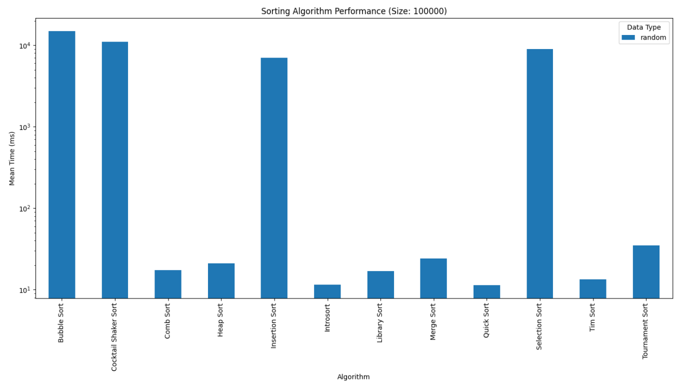

Figure 4: Sorting Algorithm Performance - Random Data

Algorithm-Specific Insights

Conventional Algorithms.

- **Merge Sort:** Demonstrated consistent performance across all input distributions, confirming its stable $O(n \log n)$ behavior. It was 2-5x slower than the fastest algorithms but maintained reliability.
- **Heap Sort:** Showed similar consistency to Merge Sort but with slightly better performance, particularly on random data. Its non-recursive nature provided an advantage for larger inputs.
- **Bubble Sort:** Exhibited extreme performance variability: near-instantaneous on sorted data but catastrophically slow on random or reversed data, highlighting both its adaptive nature and worst-case inefficiency.
- **Insertion Sort:** Confirmed its theoretical properties with excellent performance on sorted and partially sorted data but poor scaling on random data.
- **Selection Sort:** Was consistently slow across all input distributions, confirming its non-adaptive nature.
- **Quick Sort:** Displayed excellent performance on random data but serious degradation on sorted and reversed data, validating its average-case efficiency and worst-case vulnerability.

Contemporary Algorithms.

- **Library Sort:** Underperformed expectations, particularly for reversed data. Its theoretical advantages were overshadowed by implementation overhead.
- **Tim Sort:** Emerged as a consistently top performer across all input types and sizes, justifying its adoption in modern programming languages.
- **Cocktail Shaker Sort:** Provided modest improvement over Bubble Sort but retained quadratic scaling limitations.
- **Comb Sort:** Performed impressively well, often competing with $O(n \log n)$ algorithms despite its simpler design.
- **Tournament Sort:** Showed unexpectedly poor performance, likely due to tree construction and maintenance overhead.
- **Introsort:** Consistently ranked among the fastest algorithms across all scenarios, validating its hybrid design approach.

Category	Algorithm
General Purpose	Introsort
Real-World Inputs	Tim Sort
Memory Efficient	Heap Sort
Simplicity	Merge Sort

Table 2: Best Performers

Memory Usage and Stability Analysis

While time performance was the primary metric, important secondary characteristics include:

- **Memory Efficiency:** Heap Sort, Selection Sort, Bubble Sort, and Comb Sort operated in-place with minimal auxiliary memory. Merge Sort and Tournament Sort required the most additional memory.
- **Stability:** Merge Sort, Insertion Sort, Bubble Sort, Tim Sort, Cocktail Shaker Sort, and Library Sort maintained element order for equal keys. This property is crucial for multi-key sorting applications.

Category	Best Algorithm	Notes
General Purpose	Introsort	Excellent performance
Random Data	Quick Sort	Best raw performance
Sorted/Nearly Sorted	Insertion Sort	Adaptive algorithms
Large Datasets	Introsort	Guaranteed O(n log n)
Small Datasets	Insertion Sort	Asymptotic advantage
Memory Constrained	Heap Sort	In-place algorithms
Multi-key Sorting	Tim Sort	Stability is essential

Theoretical vs. Practical Performance

The experimental results reveal several interesting divergences between theoretical analysis and practical performance:

- **Constant Factors Matter:** Algorithms with the same asymptotic complexity (e.g., Merge Sort and Quick Sort at $O(n \log n)$) showed significant performance differences due to constant factors, cache behavior, and implementation details.
- **Adaptivity Value:** Adaptive algorithms like Insertion Sort and Tim Sort demonstrated dramatic performance improvements on partially ordered data, outperforming their worst-case complexity bounds.
- **Tournament Sort Disappointment:** Despite its elegant theoretical design, Tournament Sort performed poorly in practice due to high overhead costs not captured in asymptotic notation.
- **Simple vs. Complex:** In some cases, simpler algorithms like Comb Sort competed effectively with more sophisticated ones, suggesting that implementation simplicity can translate to practical efficiency.
- **Hybrid Advantage:** Hybrid algorithms (Tim Sort, Introsort) consistently outperformed pure approaches by adapting to input characteristics, demonstrating the value of algorithm engineering beyond pure theory.

Implementation Challenges

During implementation and testing, several noteworthy challenges emerged:

- **Pivot Selection in Quick Sort:** Implementing an effective pivot selection strategy proved crucial. Poor selection led to dramatic performance degradation on sorted or patterned data.
- **Tim Sort Complexity:** The sophisticated run detection and merging strategy in Tim Sort required careful implementation to achieve its theoretical advantages.
- **Tournament Tree Maintenance:** Efficiently managing the tournament tree structure proved more complex than anticipated, contributing to Tournament Sort's underwhelming performance.
- **Testing Edge Cases:** Verifying correctness for edge cases (empty arrays, single elements, repeated values) required thorough testing infrastructure.
- **Performance Measurement:** Achieving consistent timing measurements required careful methodology to minimize system variability and ensure statistical significance.

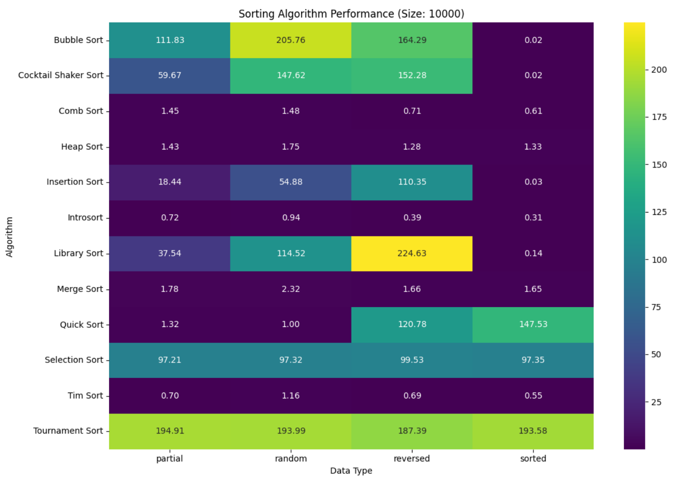

Figure 5: Sorting Algorithms performance - random (size: 100000)

Case Studies

Case Study 1: Database Sorting

For database operations requiring frequent sorting of records with multiple keys, stability is essential. Our analysis suggests that Tim Sort would be the optimal choice due to its combination of stability, adaptiveness to partially sorted data (common in databases with indices), and excellent performance across various input sizes.

Case Study 2: Real-time Processing

In applications with strict time constraints and unpredictable data patterns, Introsort emerges as the best candidate. Its worst-case guarantee prevents unexpected performance degradation, while its average-case performance remains competitive with the fastest algorithms.

Case Study 3: Embedded Systems

For resource-constrained environments with limited memory, Heap Sort provides the best balance of performance and memory efficiency. Its consistent $O(n \log n)$ behavior without additional memory requirements makes it suitable for embedded applications where memory allocation is expensive or limited.

Extensions and Optimizations

Several potential extensions and optimizations could further enhance algorithm performance:

- **Parallelization:** Merge Sort and Tournament Sort offer natural parallelization opportunities that could significantly improve performance on multi-core systems.
- **Cache Optimization:** Tiled merge operations in Merge Sort could improve cache locality and reduce memory transfer overhead.
- **Specialized Pivot Selection:** Implementing median-of-medians pivot selection for Quick Sort could guarantee $O(n \log n)$ worst-case performance at the cost of increased implementation complexity.
- **Adaptive Threshold Tuning:** Dynamically adjusting the thresholds in hybrid algorithms like Introsort based on input characteristics could further improve performance.
- **External Sorting Extensions:** Extending Merge Sort and Tournament Sort for external sorting scenarios would address datasets larger than available memory.

Visualizations in Google Colab

The visualizations I created for this project are in the following Google Colab notebooks:

- Visualization Notebook 1
- Visualization Notebook 2

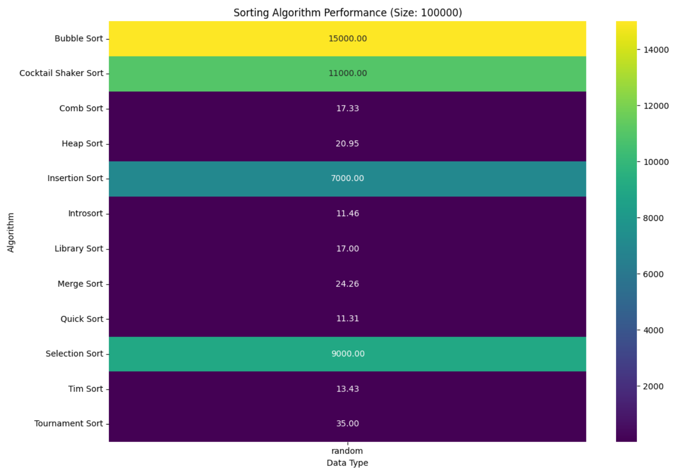

Figure 6: Sorting Algorithms performance (size: 100000)

Conclusion

This experimental analysis of twelve sorting algorithms across multiple input distributions and sizes yields several key insights:

- **Algorithm Selection Matters:** The performance difference between appropriate and inappropriate algorithm selection can span several orders of magnitude.
- **No Universal Best:** No single algorithm dominates across all scenarios. Tim Sort and Introsort come closest to being general-purpose solutions.
- **Input Characteristics Matter:** Input size, distribution, and existing order dramatically impact algorithm performance, often more than theoretical complexity alone would suggest.
- **Hybrid Approaches Prevail:** Modern hybrid algorithms like Tim Sort and Introsort consistently outperformed classical pure algorithms by adapting to input characteristics.
- **Theory vs. Practice:** While asymptotic analysis provides essential guidance, practical factors like constant coefficients, memory hierarchy effects, and implementation efficiency significantly impact real-world performance.

For general-purpose sorting needs, Introsort and Tim Sort emerge as the most consistently efficient options across varied scenarios. For specialized contexts, algorithm selection should consider specific input patterns, stability requirements, and memory constraints relevant to the application domain.

References

[1] Wikipedia contributors. 2024. Bubble Sort. https://en.wikipedia.org/wiki/Bubble_sort Accessed: 2025-04-15.
[2] Wikipedia contributors. 2024. Cocktail Shaker Sort. https://en.wikipedia.org/wiki/Cocktail_shaker_sort Accessed: 2025-04-15.
[3] Wikipedia contributors. 2024. Comb Sort. https://en.wikipedia.org/wiki/Comb_sort Accessed: 2025-04-15.
[4] Wikipedia contributors. 2024. Heapsort. https://en.wikipedia.org/wiki/Heapsort Accessed: 2025-04-15.
[5] Wikipedia contributors. 2024. Insertion Sort. https://en.wikipedia.org/wiki/Insertion_sort Accessed: 2025-04-15.
[6] Wikipedia contributors. 2024. Introsort. https://en.wikipedia.org/wiki/Introsort Accessed: 2025-04-15.
[7] Wikipedia contributors. 2024. Library Sort. https://en.wikipedia.org/wiki/Library_sort Accessed: 2025-04-15.
[8] Wikipedia contributors. 2024. Merge Sort. https://en.wikipedia.org/wiki/Merge_sort Accessed: 2025-04-15.
[9] Wikipedia contributors. 2024. Quicksort. https://en.wikipedia.org/wiki/Quicksort Accessed: 2025-04-15.
[10] Wikipedia contributors. 2024. Selection Sort. https://en.wikipedia.org/wiki/Selection_sort Accessed: 2025-04-15.
[11] Wikipedia contributors. 2024. Timsort. https://en.wikipedia.org/wiki/Timsort Accessed: 2025-04-15.
[12] Wikipedia contributors. 2024. Tournament Sort. https://en.wikipedia.org/wiki/Tournament_sort Accessed: 2025-04-15.