Ho Chi Minh City University of Science

Faculty of Information Technology

Data Structures & Algorithms

Lab 4 - Sorting Algorithms

Authors:
Huynh Dang Khoa (23127390)
Ta Van Duc (23127528)
Vo Thanh Nhan (23127533)
Cao Le Gia Phu (23127535)

Instructor:
Tran Thi Thao Nhi
Tran Hoang Quan

March 18, 2025

1 Introduction

This report details the implementation and analysis of 12 sorting algorithms as part of Lab 4 for the Data Structures & Algorithms course. The primary objectives are to implement these algorithms, measure their performance in terms of running time and number of comparisons, and analyze their behavior across various data orders and sizes.
The implementations were developed using the **C++20 standard** for compilation, ensuring modern language features and optimizations.
The report itself was prepared using **LaTeX** to provide a clear and professional presentation of the findings.
The source code and related materials are available on **GitHub** at
https://github.com/iluvmOne-Y/ProjectDSA-Sorting
for reference and further exploration.

2 Algorithm Presentation

This section introduces the 12 sorting algorithms implemented: Selection Sort, Insertion Sort, Binary Insertion Sort, Bubble Sort, Shaker Sort, Shell Sort, Heap Sort, Merge Sort, Quick Sort, Counting Sort, Radix Sort, and Flash Sort. Each algorithm is described with its core idea, steps, an example, and complexity analysis.

2.1 Selection Sort

Selection Sort repeatedly selects the minimum element from the unsorted portion and places it at the beginning.
Steps:

1. Consider the entire array unsorted initially.
2. Find the minimum element in the unsorted portion.
3. Swap it with the first unsorted element.
4. Shift the unsorted boundary right.
5. Repeat until fully sorted.

Complexity:

- Time: $O(n^2)$ (all cases).
- Space: $O(1)$ (in-place).

2.2 Insertion Sort

Insertion Sort builds a sorted portion by inserting each element into its correct position.
Steps:

1. Start with the first element as sorted.
2. Take the next unsorted element.
3. Shift larger elements right to insert it.
4. Repeat until all elements are sorted.

Complexity:

- Time: $O(n)$ (best/average), $O(n^2)$ (worst).
- Space: $O(1)$.

2.3 Binary Insertion Sort

Binary Insertion Sort uses binary search to reduce comparisons while inserting elements.

Steps:

1. Start with the first element.
2. Use binary search to find the insertion point.
3. Shift elements to insert.
4. Repeat for all elements.

Complexity:

- Time: $O(n \log n)$ (best/average), $O(n^2)$ (worst).
- Space: $O(1)$.

2.4 Bubble Sort

Bubble Sort compares adjacent elements, swapping them if out of order, pushing larger elements to the end.

Steps:

1. Compare and swap adjacent elements.
2. Repeat until the largest element is at the end.
3. Reduce unsorted portion and repeat.

Complexity:

- Time: $O(n)$ (best/average), $O(n^2)$ (worst).
- Space: $O(1)$.

2.5 Shaker Sort

Shaker Sort enhances Bubble Sort by sorting bidirectionally.

Steps:

1. Bubble largest to the end.
2. Bubble smallest to the start.
3. Repeat, shrinking the unsorted portion.

Complexity:

- Time: $O(n)$ (best/average), $O(n^2)$ (worst).
- Space: $O(1)$.

2.6 Shell Sort

Shell Sort applies Insertion Sort with decreasing gaps.

Steps:

1. Start with a large gap.
2. Sort elements separated by the gap.
3. Reduce gap and repeat until gap is 1.

Complexity:

- Time: $O(nlogn)$ (best/average) , $O(n^2)$ (worst).
- Space: $O(1)$.

2.7 Heap Sort

Heap Sort builds a max-heap and extracts elements to sort.

Steps:

1. Build a max-heap.
2. Swap root with the last element.
3. Heapify and repeat.

Complexity:

- Time: $O(n \log n)$ (all cases).
- Space: $O(1)$.

2.8 Merge Sort

Merge Sort divides the array, sorts recursively, and merges.

Steps:

1. Split into halves.
2. Sort each half recursively.
3. Merge sorted halves.

Complexity:

- Time: $O(n \log n)$ (all cases).
- Space: $O(n)$.

2.9 Quick Sort

Quick Sort partitions around a pivot and sorts recursively.

Steps:

1. Select a pivot.
2. Partition smaller elements left, larger right.
3. Recursively sort partitions.

Complexity:

- Time: $O(n \log n)$ (best/average), $O(n^2)$ (worst).
- Space: $O(\log n)$.

2.10 Counting Sort

Counting Sort counts occurrences to sort non-comparatively.

Steps:

1. Determine value range.
2. Count occurrences.
3. Place elements in order.

Example: $[4, 2, 1, 4, 3] \to [1, 2, 3, 4, 4]$.

Complexity:

- Time: $O(n + k)$.
- Space: $O(n + k)$.

2.11 Radix Sort

Radix Sort sorts digit-by-digit using Counting Sort.

Steps:

1. Find max digits.
2. Sort by each digit using Counting Sort.
3. Repeat for all digits.

Complexity:

- Time: $O(d(n + k))$.
- Space: $O(n + k)$.

2.12 Flash Sort

Flash Sort distributes elements approximately, then uses Insertion Sort.

Steps:

1. Estimate class boundaries.
2. Distribute elements into classes.
3. Permute and finish with Insertion Sort.

Complexity:

- Time: $O(n)$ (best/average), $O(n^2)$ (worst).
- Space: $O(n)$.

3 Experimental Results and Comments

This section evaluates the 12 sorting algorithms across four data orders (Randomized, Nearly Sorted, Sorted, ReversedSorted) and six sizes (10K, 30K, 50K, 100K, 300K, 500K), measuring running time (ms) and comparisons.

3.1 Tables

The tables below present the experimental results, sourced from the defined data tables.

Table 1: Results for Randomized Data

Algorithm	**10K**		**30K**		**50K**		**100K**		**300K**		**500K**	
	Time	Comp	Time	Comp	Time	Comp	Time	Comp	Time	Comp	Time	Comp
Selection Sort	60	49995000	534	449985000	1497	1249975000	5980	4999950000	53817	44999850000	150227	124999750000
Insertion Sort	31	25153857	282	224365488	791	625819223	3189	2500080439	28967	22483749176	81409	62523401586
Binary Insertion Sort	22	219731	196	756765	533	1334941	2128	2870699	19072	9551425	53754	16670175
Bubble Sort	205	49977045	1835	449962845	5325	1249943374	22187	4999888224	206002	44999566872	572526	124999293510
Shaker Sort	125	33394512	1286	300854016	3809	835837725	16172	3333861223	149730	30032507131	414734	83282368029
Shell Sort	1	260882	4	921846	7	1889687	16	4279136	58	14901573	104	29571942
Heap Sort	1	235397	4	800320	8	1409925	18	3019367	61	10000509	104	17396978
Merge Sort	1	120378	4	408634	7	718092	15	1536046	49	5084489	86	8837069
Quick Sort	1	152297	3	525026	6	931288	12	1942587	41	6753368	71	11501478
Counting Sort	0	9999	0	29999	0	49999	0	99999	2	299999	5	499999
Radix Sort	0	33196	0	99568	1	165912	2	332357	8	996806	17	1662168
Flash Sort	0	33196	0	99568	1	165912	3	332357	10	996806	15	1662168

Table 2: Results for Nearly Sorted Data

Algorithm	**10K**		**30K**		**50K**		**100K**		**300K**		**500K**	
	Time	Comp	Time	Comp	Time	Comp	Time	Comp	Time	Comp	Time	Comp
Selection Sort	59	49995000	534	449985000	1547	1249975000	5952	4999950000	53847	44999850000	150543	124999750000
Insertion Sort	0	91441	0	281483	0	430887	0	734425	4	3444991	5	3997909
Binary Insertion Sort	0	235453	1	796151	3	1392581	7	2978471	24	9936733	41	17451425
Bubble Sort	60	41677919	502	338957649	1531	1045500247	6608	4364577872	50756	33755624259	171343	114215453047
Shaker Sort	0	87127	0	174249	1	469494	3	1060274	6	2407674	15	5181860
Shell Sort	0	139165	1	459393	1	800828	4	1764332	13	5687325	23	9977393
Heap Sort	1	244478	3	826393	6	1455415	13	3112427	42	10279677	73	17832307
Merge Sort	0	87493	2	268240	4	471762	8	995504	26	3243880	43	5288090
Quick Sort	15	8988674	150	65843992	1289	469349607	5019	1738528506	12728	8470421936	185596	80919911304
Counting Sort	0	9999	0	29999	0	49999	0	99999	2	299999	4	499999
Radix Sort	0	35493	0	106490	1	177491	2	354993	8	1064993	14	1774991
Flash Sort	0	35493	0	106490	0	177491	2	354993	6	1064993	10	1774991

Table 3: Results for Sorted Data

Algorithm	**10K**		**30K**		**50K**		**100K**		**300K**		**500K**	
	Time	Comp	Time	Comp	Time	Comp	Time	Comp	Time	Comp	Time	Comp
Selection Sort	59	49995000	533	449985000	1507	1249975000	6015	4999950000	53897	44999850000	148907	124999750000
Insertion Sort	0	9999	0	29999	0	49999	0	99999	0	299999	1	499999
Binary Insertion Sort	0	237235	1	804467	3	1418931	6	3037859	23	10051427	40	17451427
Bubble Sort	0	9999	0	29999	0	49999	0	99999	0	299999	0	499999
Shaker Sort	0	9999	0	29999	0	49999	0	99999	0	299999	0	499999
Shell Sort	0	120005	0	390007	1	700006	3	1500006	10	5100008	17	8500007
Heap Sort	1	244460	3	826347	6	1455438	13	3112517	42	10279749	72	17837785
Merge Sort	0	69008	2	227728	3	401952	8	853904	26	2797264	43	4783216
Quick Sort	143	49995000	1261	449985000	3645	1249975000	15083	4999950000	140840	44999850000	382325	124999750000
Counting Sort	0	9999	0	29999	0	49999	0	99999	2	299999	4	499999
Radix Sort	0	35497	0	106497	1	177497	2	354997	8	1064997	14	1774997
Flash Sort	0	35497	0	106497	1	177497	2	354997	6	1064997	10	1774997

Table 4: Results for Reversed Sorted Data

Algorithm	**10K**		**30K**		**50K**		**100K**		**300K**		**500K**	
	Time	Comp	Time	Comp	Time	Comp	Time	Comp	Time	Comp	Time	Comp
Selection Sort	68	49995000	600	449985000	1811	1249975000	6698	4999950000	60695	44999850000	168716	124999750000
Insertion Sort	63	49995000	569	449985000	1584	1249975000	6394	4999950000	57442	44999850000	161019	124999750000
Binary Insertion Sort	42	225453	384	771729	1064	1353427	4239	2906821	38919	9713607	108312	16927177
Bubble Sort	146	49995000	1332	449985000	3881	1249975000	16707	4999950000	152465	44999850000	426192	124999750000
Shaker Sort	152	49995000	1383	449985000	3915	1249975000	16292	4999950000	154577	44999850000	435102	124999750000
Shell Sort	0	172578	1	567016	2	1047305	4	2244585	15	7300919	26	12428778
Heap Sort	1	226682	3	775687	6	1366047	12	2926640	41	9740640	72	16977997
Merge Sort	0	64608	2	219504	3	382512	8	815024	26	2678448	43	4692496
Quick Sort	101	49995000	894	449985000	2500	1249975000	10052	4999950000	93554	44999850000	265961	124999750000
Counting Sort	0	9999	0	29999	0	49999	0	99999	2	299999	4	499999
Radix Sort	0	33748	0	101248	1	168748	2	337498	8	1012498	14	1687498
Flash Sort	0	33748	0	101248	0	168748	1	337498	5	1012498	9	1687498

3.2 Graphs

The graphs below visualize the running times and comparisons, sourced from the same data tables.

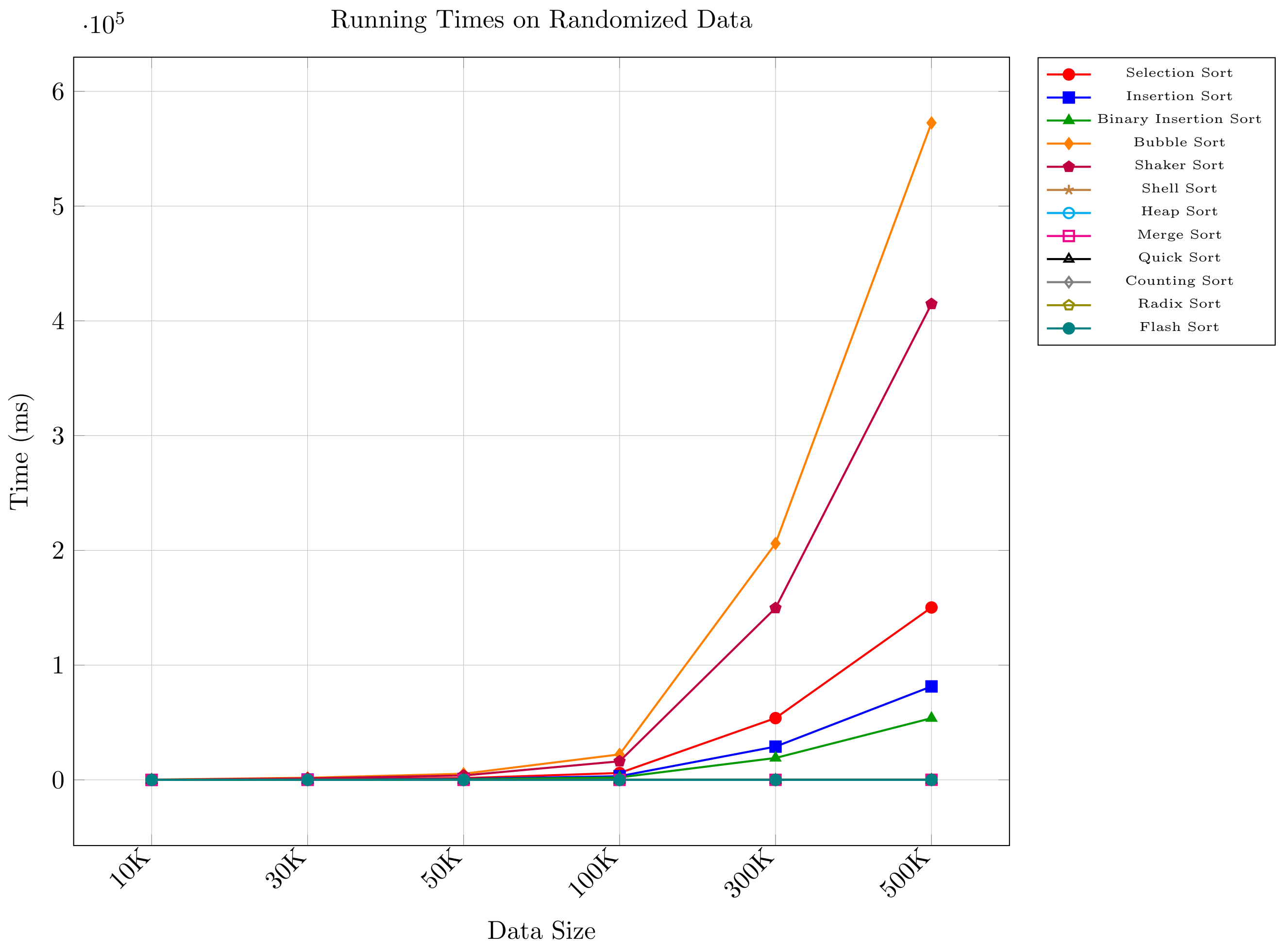

Figure 1: Running times of sorting algorithms on randomized data.

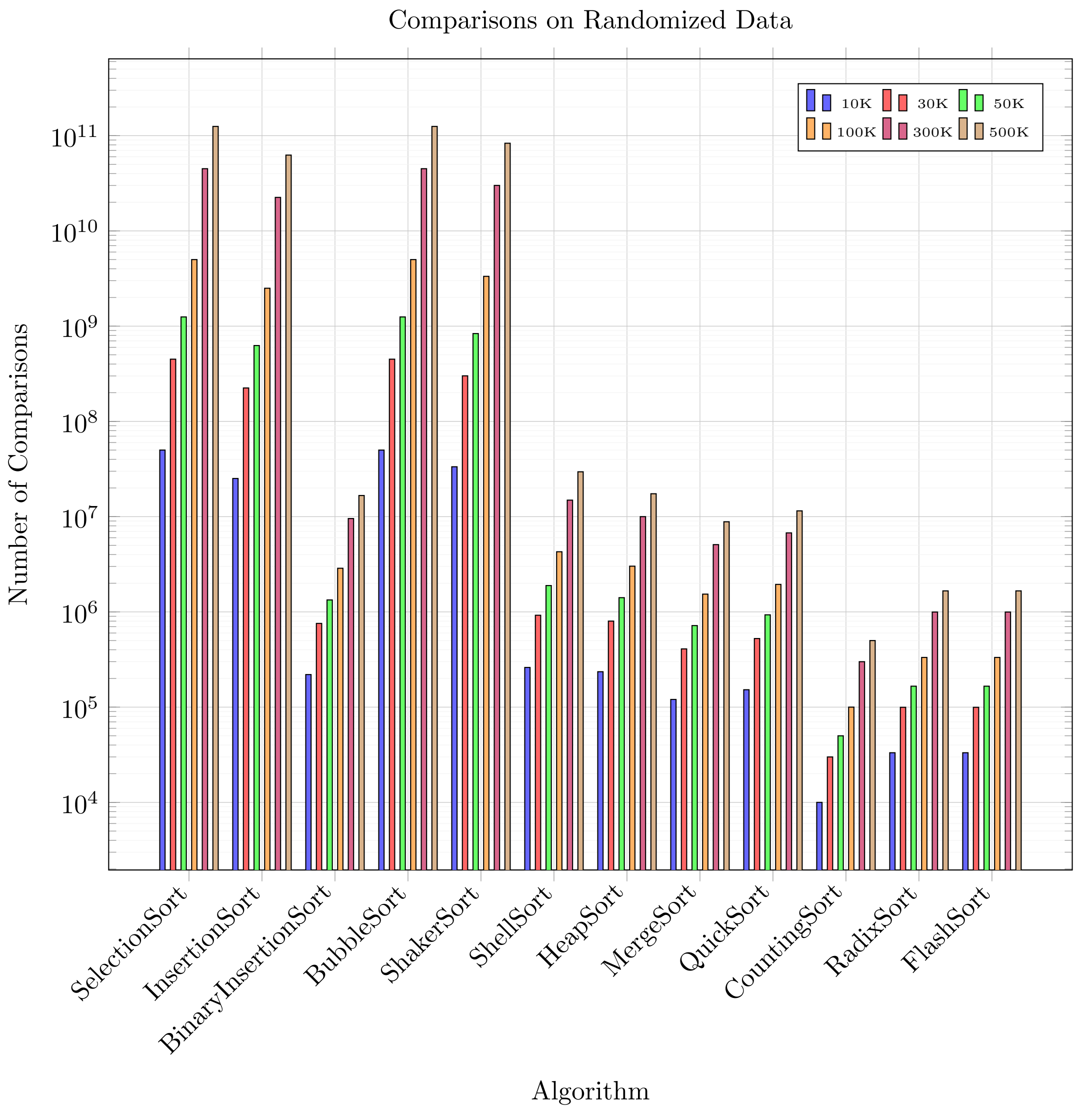

Figure 2: Number of comparisons of sorting algorithms on randomized data.

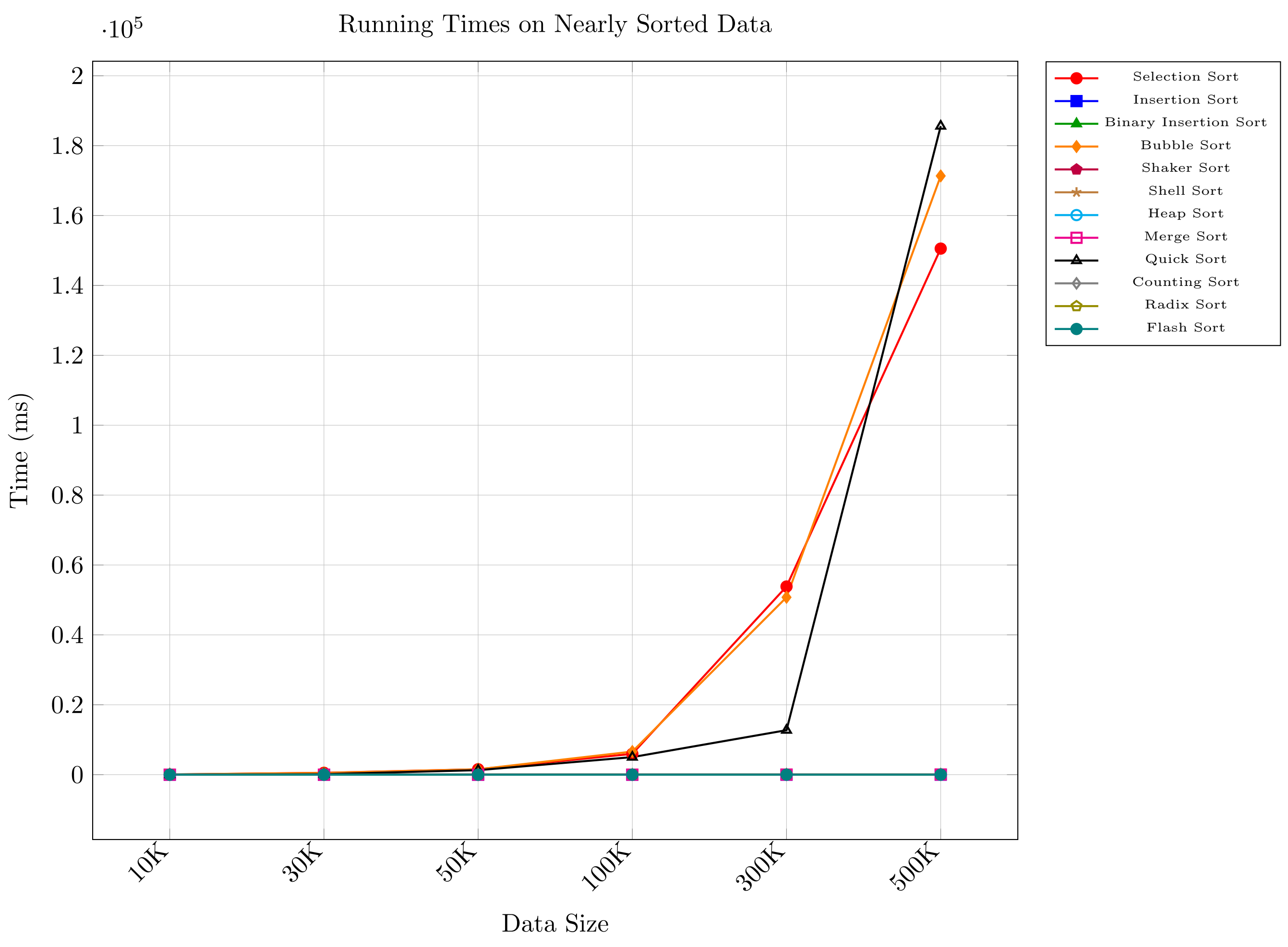

Figure 3: Running times of sorting algorithms on nearly sorted data.

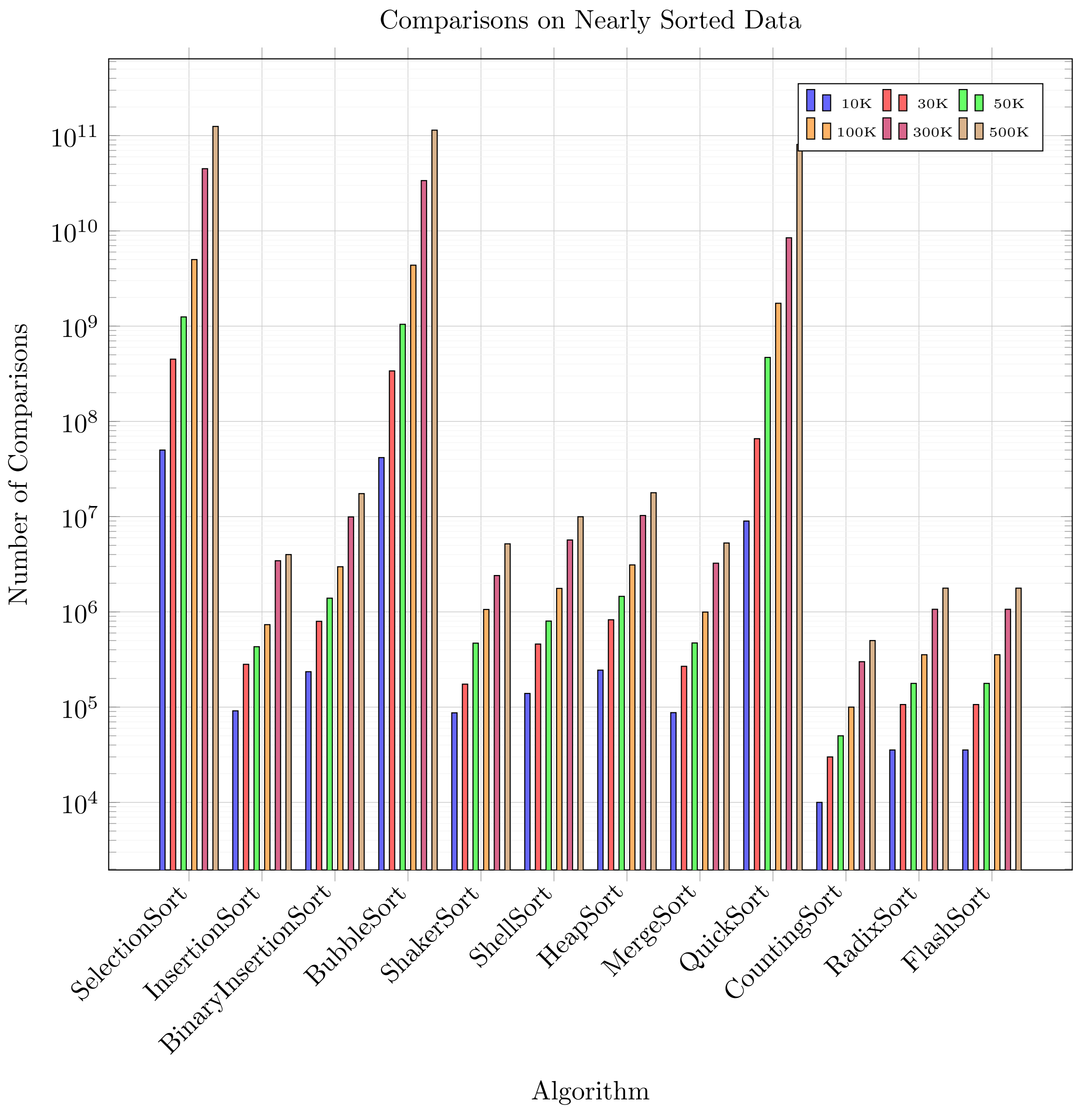

Figure 4: Number of comparisons of sorting algorithms on nearly sorted data.

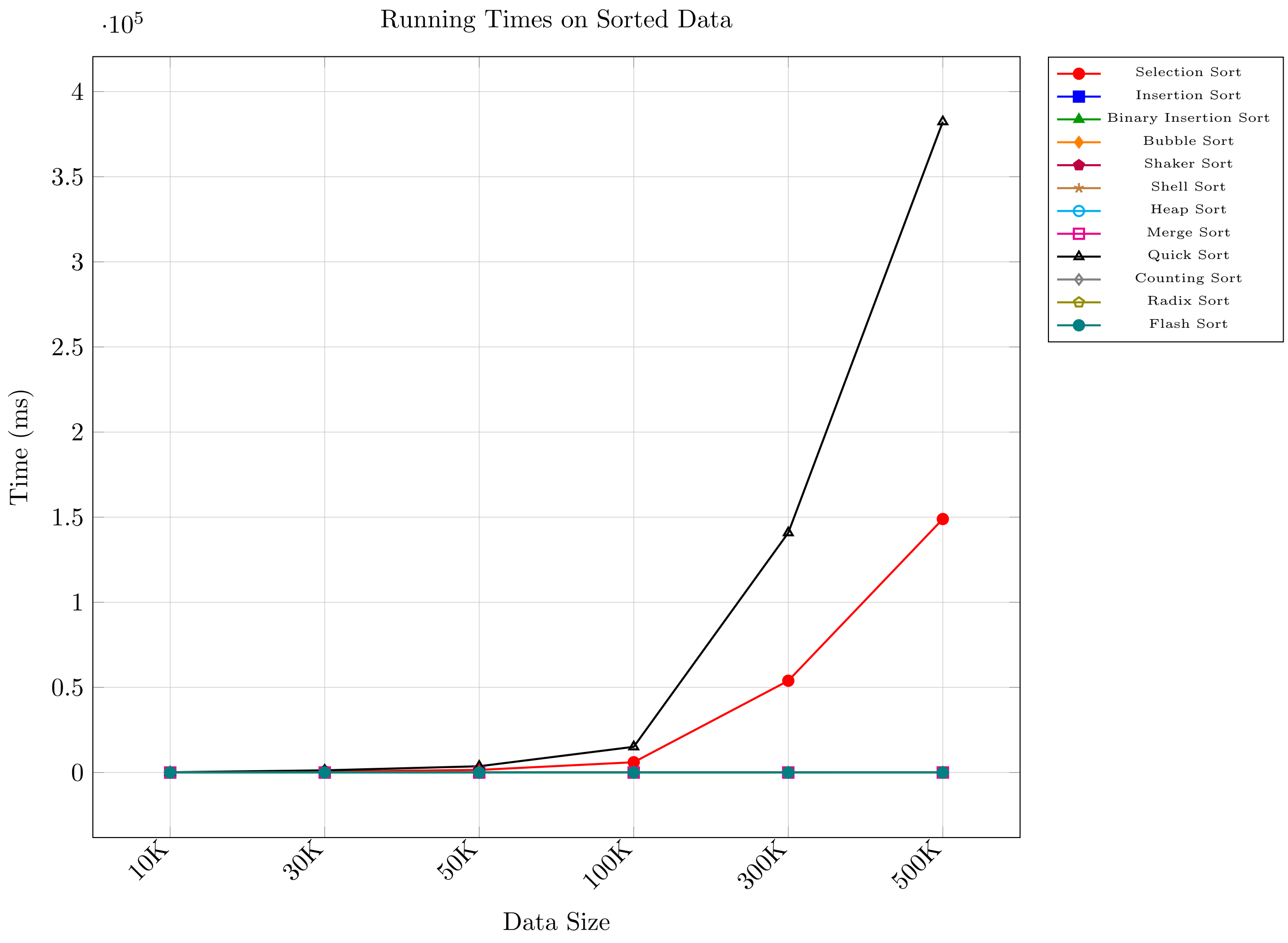

Figure 5: Running times of sorting algorithms on sorted data.

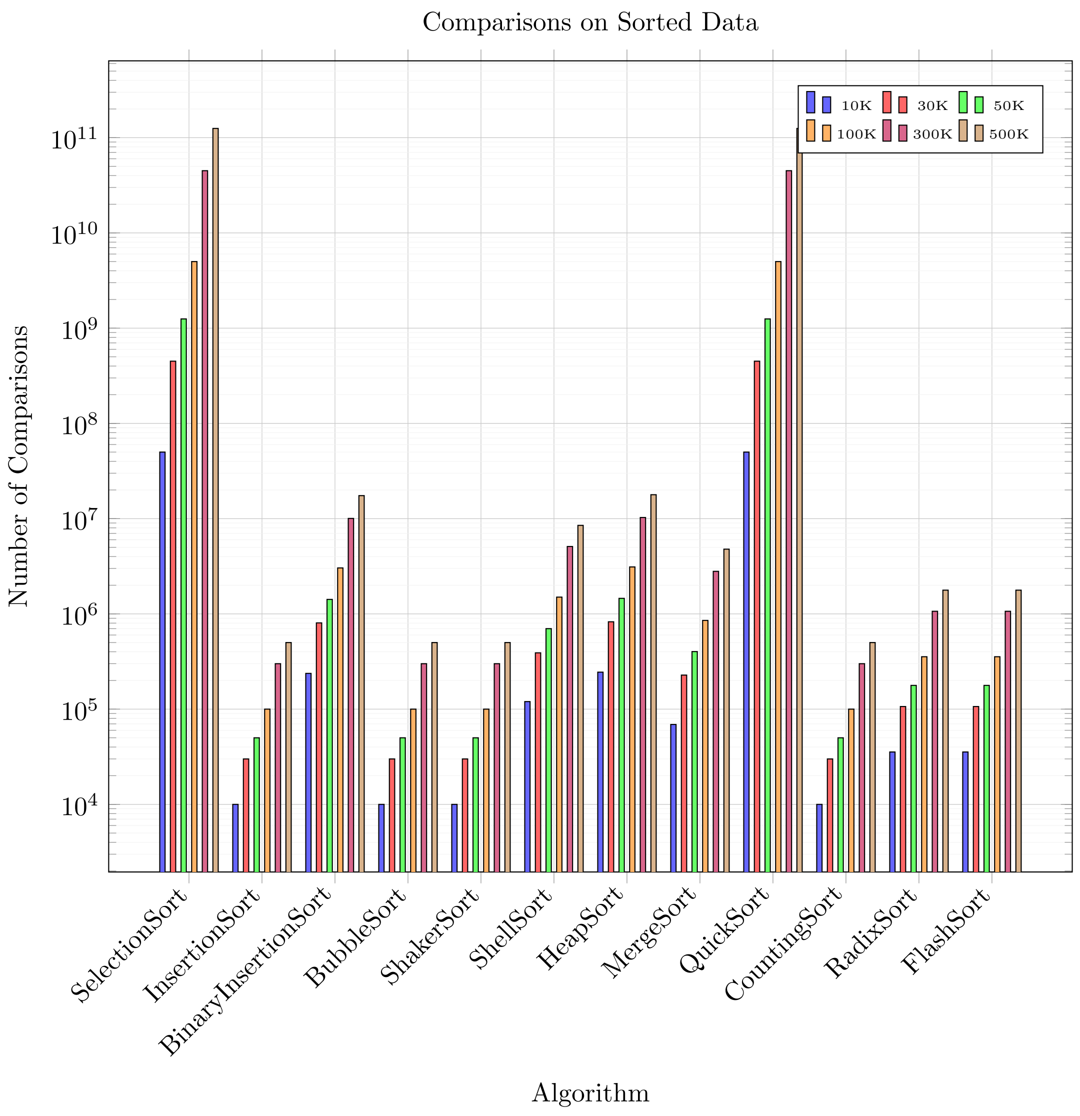

Figure 6: Number of comparisons of sorting algorithms on sorted data.

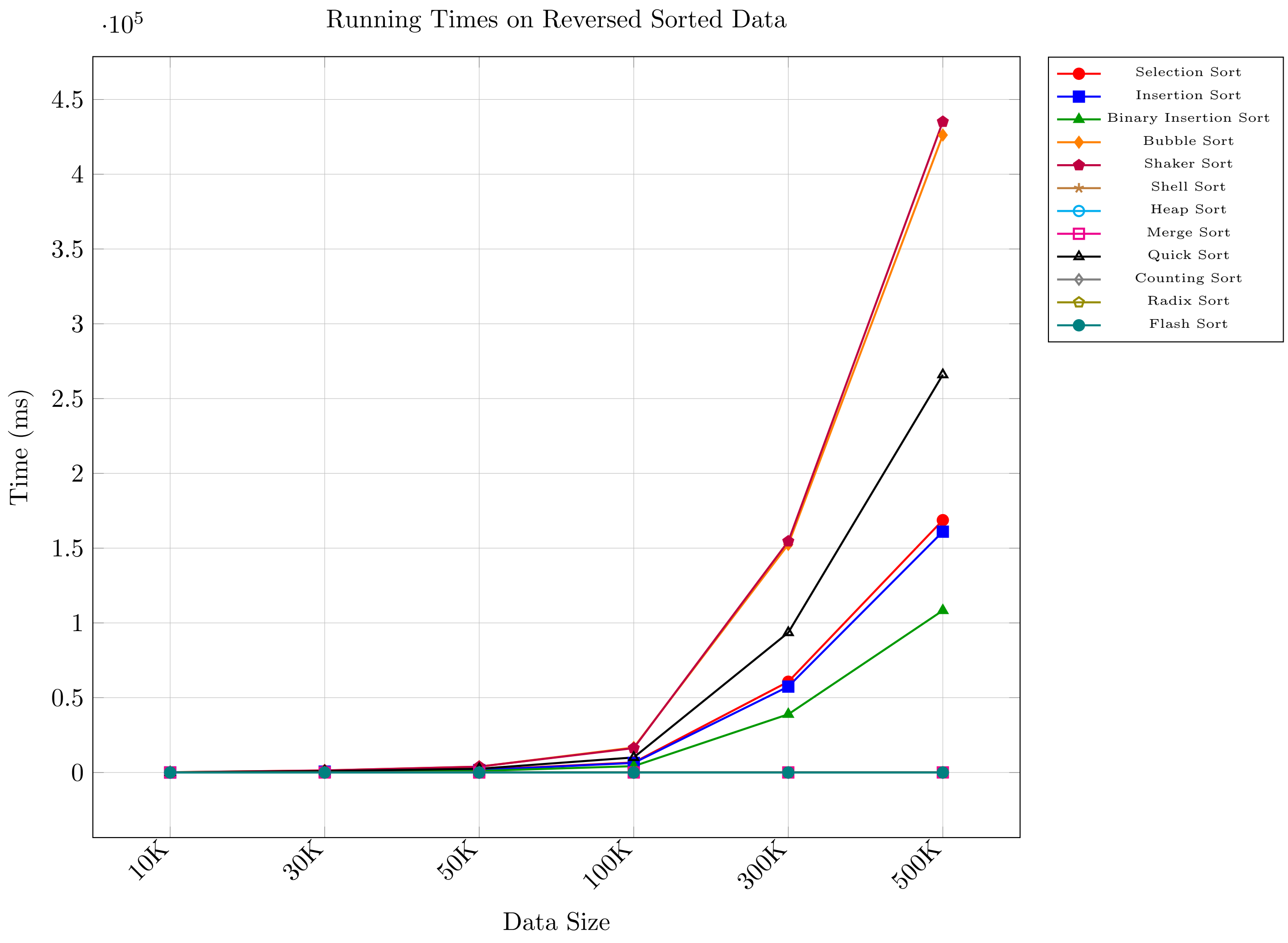

Figure 7: Running times of sorting algorithms on reversed sorted data.

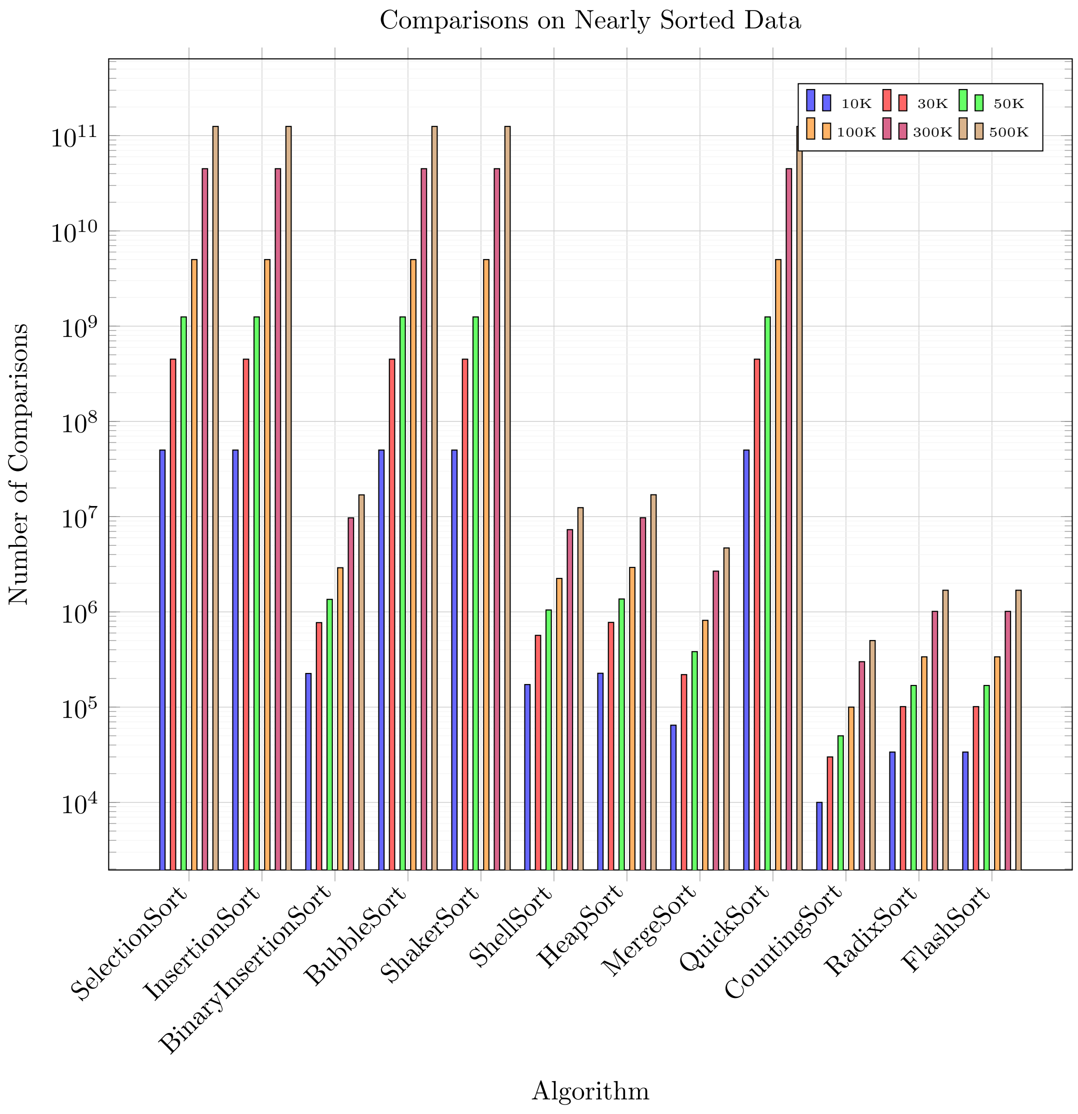

Figure 8: Number of comparisons of sorting algorithms on nearly sorted data.

3.3 Comments

This subsection analyzes the performance of the 12 sorting algorithms across four data orders—Randomized, Nearly Sorted, Sorted, and Reversed Sorted—and six data sizes (10K, 30K, 50K, 100K, 300K, 500K), focusing on running times (in milliseconds) and the number of comparisons.

3.3.1 Randomized Data

Running Times:

- **Fastest Algorithms:** Counting Sort (0–5 ms), Radix Sort (0–17 ms), and Flash Sort (0–15 ms) consistently exhibit the lowest running times across all sizes. As non-comparison-based algorithms, they leverage linear or near-linear complexities—$O(n+k)$ for Counting Sort, $O(d(n+k))$ for Radix Sort, and $O(n)$ average for Flash Sort—making them highly efficient regardless of data order.
- **Slowest Algorithms:** Bubble Sort (205–572,526 ms) and Selection Sort (60–150,227 ms) are the slowest, reflecting their $O(n^2)$ time complexity. Bubble Sort's high swap overhead and Selection Sort's exhaustive minimum searches contribute to their poor scalability, with times growing quadratically as size increases.
- **Scalability:** Comparison-based algorithms with $O(n \log n)$ complexity, such as Merge Sort (1–86 ms), Heap Sort (1–104 ms), and Quick Sort (1–71 ms), scale efficiently, maintaining low running times even at 500K elements. Shell Sort (1–104 ms) outperforms other $O(n^2)$ algorithms due to its gap-based optimization, approximating $O(n^{3/2})$.
- **Trends:** Insertion Sort (31–81,409 ms) and Shaker Sort (125–414,734 ms) show quadratic growth but perform better than Bubble Sort due to fewer unnecessary operations. Binary Insertion Sort (22–53,754 ms) benefits from fewer comparisons but remains $O(n^2)$ due to shifting overhead.

Comparisons:

- **Least Comparisons:** Counting Sort (9,999–499,999), Radix Sort (33,196–1,662,168), and Flash Sort (33,196–1,662,168) report the fewest "comparisons." Note that these figures likely represent other operations (e.g., array accesses), as these algorithms are non-comparative.
- **Most Comparisons:** Selection Sort (49,995,000–124,999,750,000) performs exactly $n(n-1)/2$ comparisons, unaffected by data order. Bubble Sort (49,977,045–124,999,293,510) and Shaker Sort (33,394,512–83,282,368,029) also exhibit high comparison counts, close to $O(n^2)$.
- **Trends:** Merge Sort (120,378–8,837,069), Heap Sort (235,397–17,396,978), and Quick Sort (152,297–11,501,478) align with their $O(n \log n)$ average complexity, with Merge Sort typically using fewer comparisons. Binary Insertion Sort (219,731–16,670,175) reduces comparisons to $O(n \log n)$ via binary search but retains $O(n^2)$ time due to shifts.

3.3.2 Nearly Sorted Data

Running Times:

- **Fastest Algorithms:** Insertion Sort (0–5 ms), Shaker Sort (0–15 ms), Counting Sort (0–4 ms), Radix Sort (0–14 ms), and Flash Sort (0–10 ms) excel, leveraging the near-sorted order. Insertion Sort achieves near-linear $O(n+d)$ performance (where d is the number of inversions), and non-comparative sorts remain unaffected by order.
- **Slowest Algorithms:** Quick Sort (15–185,596 ms) performs poorly, degrading to $O(n^2)$ due to suboptimal pivot selection (e.g., first/last element) on nearly sorted data. Selection Sort (59–150,543 ms) remains slow, as its performance is order-invariant.
- **Scalability:** Merge Sort (0–43 ms) and Heap Sort (1–73 ms) scale consistently at $O(n \log n)$. Shell Sort (0–23 ms) benefits from smaller gaps, while Bubble Sort (60–171,343 ms) improves slightly over randomized data due to early termination.
- **Trends:** Binary Insertion Sort (0–41 ms) is slower than Insertion Sort despite fewer comparisons, due to binary search overhead and $O(n^2)$ shifting costs.

Comparisons:

- **Least Comparisons:** Insertion Sort (91,441–3,997,909) and Shaker Sort (87,127–5,181,860) minimize comparisons due to their adaptability to near-sorted data. Counting Sort (9,999–499,999) remains constant.
- **Most Comparisons:** Quick Sort (8,988,674–80,919,911,304) exhibits worst-case $O(n^2)$ behavior, a significant anomaly. Selection Sort (49,995,000–124,999,750,000) is consistently high, as expected.
- **Trends:** Bubble Sort (41,677,919–114,215,453,047) uses fewer comparisons than on randomized data but remains inefficient. Merge Sort (87,493–5,288,090) and Heap Sort (244,478–17,832,307) stay within $O(n \log n)$.

3.3.3 Sorted Data

Running Times:

- **Fastest Algorithms:** Insertion Sort (0–1 ms), Bubble Sort (0 ms), Shaker Sort (0 ms), Counting Sort (0–4 ms), Radix Sort (0–14 ms), and Flash Sort (0–10 ms) dominate. Adaptive algorithms achieve $O(n)$ best-case performance by detecting the sorted state early, while non-comparative sorts maintain linear efficiency.
- **Slowest Algorithms:** Quick Sort (143–382,325 ms) is the slowest, hitting its $O(n^2)$ worst-case due to poor pivot choices on sorted data. Selection Sort (59–148,907 ms) remains consistently slow.
- **Scalability:** Merge Sort (0–43 ms) and Heap Sort (1–72 ms) scale at $O(n \log n)$, while Shell Sort (0–17 ms) performs well due to minimal gap adjustments.
- **Trends:** Binary Insertion Sort (0–40 ms) is slower than Insertion Sort due to unnecessary binary search overhead on already sorted data.

Comparisons:

- **Least Comparisons:** Insertion Sort, Bubble Sort, and Shaker Sort (9,999–499,999) each perform exactly $n - 1$ comparisons, terminating after one pass. Counting Sort (9,999–499,999) is similarly low.
- **Most Comparisons:** Quick Sort and Selection Sort (49,995,000–124,999,750,000) tie at $n(n-1)/2$, confirming Quick Sort's worst-case behavior.
- **Trends:** Merge Sort (69,008–4,783,216) uses fewer comparisons than Heap Sort (244,460–17,837,785), both at $O(n \log n)$. Binary Insertion Sort (237,235–17,451,427) incurs extra comparisons due to binary searches.

3.3.4 Reversed Sorted Data

Running Times:

- **Fastest Algorithms:** Counting Sort (0–4 ms), Radix Sort (0–14 ms), and Flash Sort (0–9 ms) remain the fastest, unaffected by data order. Shell Sort (0–26 ms) performs well due to its gap sequence optimization.
- **Slowest Algorithms:** Shaker Sort (152–435,102 ms) and Insertion Sort (63–161,019 ms) are the slowest, hitting their $O(n^2)$ worst-case with maximum shifts or passes. Bubble Sort (146–426,192 ms) is slightly faster than Shaker Sort.
- **Scalability:** Merge Sort (0–43 ms) and Heap Sort (1–72 ms) maintain $O(n \log n)$ efficiency. Quick Sort (101–265,961 ms) degrades to $O(n^2)$, similar to sorted data.
- **Trends:** Binary Insertion Sort (42–108,312 ms) outperforms Insertion Sort due to fewer comparisons, though both are $O(n^2)$.

Comparisons:

- **Least Comparisons:** Counting Sort (9,999–499,999), Radix Sort (33,748–1,687,498), and Flash Sort (33,748–1,687,498) lead among non-comparative sorts. Shell Sort (172,578–12,428,778) is notably low for a comparison-based algorithm.
- **Most Comparisons:** Selection Sort, Insertion Sort, Bubble Sort, Shaker Sort, and Quick Sort (49,995,000–124,999,750,000) all hit $n(n-1)/2$ in their worst case.
- **Trends:** Binary Insertion Sort (225,453–16,927,177) reduces comparisons to $O(n \log n)$ but retains $O(n^2)$ time. Merge Sort (64,608–4,692,496) uses fewer comparisons than Heap Sort (226,682–16,977,997).

3.3.5 Overall Comment

Across all data orders and sizes, **Counting Sort**, **Radix Sort**, and **Flash Sort** consistently outperform others, with running times of 0–17 ms and minimal "comparisons" (up to 1.7M at 500K), due to their non-comparative, linear or near-linear complexities. **Selection Sort** and **Shaker Sort** are the slowest overall (up to 150,543 ms and 435,102 ms at 500K), with Selection Sort performing the most comparisons (124,999,750,000) due to its fixed $O(n^2)$ behavior.

- **Stable Performers:** Merge Sort (0–86 ms) and Heap Sort (1–104 ms) offer reliable $O(n \log n)$ performance, with Merge Sort using fewer comparisons (up to 8.8M vs. 17.8M).
- **Adaptive Algorithms:** Insertion Sort, Bubble Sort, and Shaker Sort shine on nearly sorted or sorted data (0–15 ms), achieving $O(n)$ best-case, but struggle on randomized or reversed data (up to 161,019 ms).
- **Notable Anomaly:** Quick Sort (1–382,325 ms) excels on randomized data (1–71 ms) but degrades to $O(n^2)$ on nearly sorted, sorted, and reversed data due to poor pivot selection (e.g., first/last element), a critical limitation in its implementation.
- **Efficiency Grouping:** Non-comparative sorts lead in speed and scalability, followed by $O(n \log n)$ sorts (Merge, Heap, Quick on random data), while $O(n^2)$ sorts (Selection, Bubble, Shaker, Insertion) lag, with Shell Sort as a standout exception.

4 Project Organization and Programming Notes

The project is organized into several core components, each encapsulated within dedicated header and source files:

- **algorithms.hpp/cpp**: Contains the implementations of all 12 sorting algorithms evaluated in this project.
- **Command.hpp/cpp**: Implements the command-line interface and includes functions for benchmarking the algorithms, providing users with flexible interaction options.
- **DataGenerator.hpp/cpp**: Manages the generation of test data with various distributions, such as randomized, nearly sorted, sorted, and reversed sorted, to assess algorithm performance under different conditions.
- **HelperFunction.hpp/cpp**: Offers utility functions for file input/output operations and results formatting, ensuring consistent and user-friendly output.

The project encompasses a total of 12 sorting algorithms, divided into two categories based on their approach:

- **Comparison-based algorithms**: Bubble Sort, Selection Sort, Insertion Sort, Quick Sort, Shaker Sort, Shell Sort, Heap Sort, Binary Insertion Sort, and Merge Sort.
- **Non-comparison-based algorithms**: Radix Sort, Counting Sort, and Flash Sort.

The command-line interface provides a robust set of commands to support diverse testing and evaluation scenarios:

- **Command1**: Executes a specified sorting algorithm on data loaded from an input file.
- **Command2**: Runs a specified sorting algorithm on generated data with a user-defined distribution.
- **Command3**: Benchmarks a single algorithm across multiple data distributions to analyze its performance comprehensively.
- **Command4**: Compares the performance of two algorithms on the same input file.
- **Command5**: Compares the performance of two algorithms on generated data.
- **CommandBenchmarkAll**: Conducts an extensive benchmark of all algorithms across various input sizes and data distributions.

Two critical data structures underpin the project's flexibility and efficiency:

- **AlgorithmInfo Structure**: Pairs each algorithm's name with a function pointer to its implementation, enabling dynamic invocation based on user input.
- **Algorithm Registry**: A collection of AlgorithmInfo instances that allows the program to select and execute algorithms by name at runtime.

Performance evaluation is based on two key metrics:

- **Execution Time**: Measured with millisecond precision using the chrono library, providing an accurate representation of runtime performance across different scenarios.
- **Comparison Operations**: Recorded by incrementing a counter within each algorithm's implementation for every comparison performed, offering insight into algorithmic efficiency.

5 References

References

[1] Overleaf. (n.d.). *Overleaf Documentation.* Retrieved from https://www.overleaf.com/learn

[2] GeeksforGeeks. (n.d.). *Sorting Algorithms.* Retrieved from https://www.geeksforgeeks.org/sorting-algorithms/