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 17, 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.

Example: $[29, 10, 14, 37, 13] \rightarrow [10, 13, 14, 29, 37]$ (see report for detailed steps).
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.

Example: $[29, 10, 14, 37, 13] \rightarrow [10, 13, 14, 29, 37]$.
Complexity:

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

Example: $[29, 10, 14, 37, 13] \rightarrow [10, 13, 14, 29, 37]$.
Complexity:

- Time: $O(n \log n)$ comparisons, $O(n^2)$ shifts.
- 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.

Example: $[29, 10, 14, 37, 13] \rightarrow [10, 13, 14, 29, 37]$.
Complexity:

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

Example: $[29, 10, 14, 37, 13] \rightarrow [10, 13, 14, 29, 37]$.
Complexity:

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

Example: $[29, 10, 14, 37, 13] \rightarrow [10, 13, 14, 29, 37]$.
Complexity:

- Time: $O(n^{1.3})$ to $O(n^2)$.
- 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.

Example: $[29, 10, 14, 37, 13] \rightarrow [10, 13, 14, 29, 37]$.
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.

Example: $[29, 10, 14, 37, 13] \rightarrow [10, 13, 14, 29, 37]$.
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.

Example: $[29, 10, 14, 37, 13] \rightarrow [10, 13, 14, 29, 37]$.
Complexity:

- Time: $O(n \log n)$ (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: $[4, 2, 1, 4, 3] \rightarrow [1, 2, 3, 4, 4]$.
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.

Example: $[29, 10, 14, 37, 13] \rightarrow [10, 13, 14, 29, 37]$.
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.

Example: $[29, 10, 14, 37, 13] \rightarrow [10, 13, 14, 29, 37]$.
Complexity:

- Time: $O(n)$ (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	59	49995000	533	449985000	1478	1249975000	5917	4999950000	53291	44999850000	147670	124999750000
Insertion Sort	37	25011610	335	223992559	931	622987199	3799	2510381366	33744	22507251073	93414	62466706546
Binary Insertion Sort	37	25277702	335	224215059	943	624737014	3760	2504803555	33776	22490177340	94755	62553473311
Bubble Sort	83	37835154	532	338031834	1498	943235704	5972	3757413825	54780	33777058529	153244	93807598204
Shaker Sort	136	33269808	1299	298458888	3758	833815974	15788	3332679278	145558	29973511759	404764	83311139074
Shell Sort	1	262508	3	924139	7	1896576	16	4439866	55	14477986	105	29725967
Heap Sort	1	258440	4	869506	8	1524604	17	3249682	57	10691060	102	18548026
Merge Sort	1	120475	4	408776	7	718164	15	1536178	50	5083919	86	8836885
Quick Sort	2	148914	3	472742	6	902634	13	1810852	42	6078837	73	10766162
Counting Sort	0	9999	0	29999	0	49999	0	99999	2	299999	4	499999
Radix Sort	0	49999	0	179999	1	299999	2	599999	8	2099999	15	3499999
Flash Sort	0	44407	0	142390	1	234217	3	470453	9	1298805	18	2361430

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	530	449985000	1476	1249975000	5901	4999950000	53224	44999850000	147689	124999750000
Insertion Sort	0	70999	0	155305	0	573323	1	970435	4	2970493	7	4834901
Binary Insertion Sort	0	199445	1	590700	2	1121492	5	2156481	17	7145043	34	14515375
Bubble Sort	0	9999	0	29999	0	49999	0	99999	0	299999	0	499999
Shaker Sort	0	75555	0	221017	0	335352	2	838312	7	2555501	13	4523603
Shell Sort	0	142104	1	453530	2	835276	4	1793236	14	5780685	23	9745256
Heap Sort	1	273716	3	910168	6	1596376	12	3401612	42	11124616	73	19228244
Merge Sort	0	86524	2	277384	4	486461	8	1020799	27	3368542	45	5611708
Quick Sort	3	2142081	40	33647996	43	34641257	329	272967415	1728	1417360608	6229	5184794748
Counting Sort	0	9999	0	29999	0	49999	0	99999	2	299999	3	499999
Radix Sort	0	49999	0	179999	1	299999	2	599999	8	2099999	15	3499999
Flash Sort	0	55488	0	166490	1	277489	2	554989	6	1664989	10	2774989

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	530	449985000	1473	1249975000	5894	4999950000	53137	44999850000	147574	124999750000
Insertion Sort	0	9999	0	29999	0	49999	0	99999	0	299999	1	499999
Binary Insertion Sort	0	133616	1	447232	1	784464	4	1668928	13	5475712	25	9475712
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	273912	3	910200	6	1596608	13	3401708	42	11122320	73	19211400
Merge Sort	0	69008	2	227728	4	401952	8	853904	26	2797264	44	4783216
Quick Sort	1	113631	1	387248	3	684481	8	1468946	21	4875732	36	8475732
Counting Sort	0	9999	0	29999	0	49999	0	99999	2	299999	3	499999
Radix Sort	0	49999	0	179999	1	299999	2	599999	8	2099999	15	3499999
Flash Sort	0	55494	0	166494	1	277494	2	554994	6	1664994	10	2774994

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	66	49995000	597	449985000	1658	1249975000	6629	4999950000	59686	44999850000	166294	124999750000
Insertion Sort	75	49995000	669	449985000	1874	1249975000	7496	4999950000	67115	44999850000	186561	124999750000
Binary Insertion Sort	74	50118630	672	450402247	1910	1250709480	7426	5001518945	68108	45005025731	190432	125008725731
Bubble Sort	47	37507497	402	337522497	1106	937537497	4426	3750074997	40591	33750224997	112917	93750374997
Shaker Sort	151	49995000	1373	449985000	3834	1249975000	16098	4999950000	154074	44999850000	434947	124999750000
Shell Sort	0	172578	1	567016	2	1047305	4	2244585	15	7300919	26	12428778
Heap Sort	1	243392	3	828424	6	1447784	12	3094868	41	10252492	72	17836900
Merge Sort	0	64608	2	219504	4	382512	8	815024	26	2678448	44	4692496
Quick Sort	90	30001000	615	270003000	1773	750005000	7152	3000010000	65435	27000030000	183094	75000050000
Counting Sort	0	9999	0	29999	0	49999	0	99999	2	299999	3	499999
Radix Sort	0	49999	0	179999	1	299999	2	599999	8	2099999	15	3499999
Flash Sort	0	48747	0	146247	0	243747	2	487497	5	1462497	9	2437497

3.2 Graphs

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

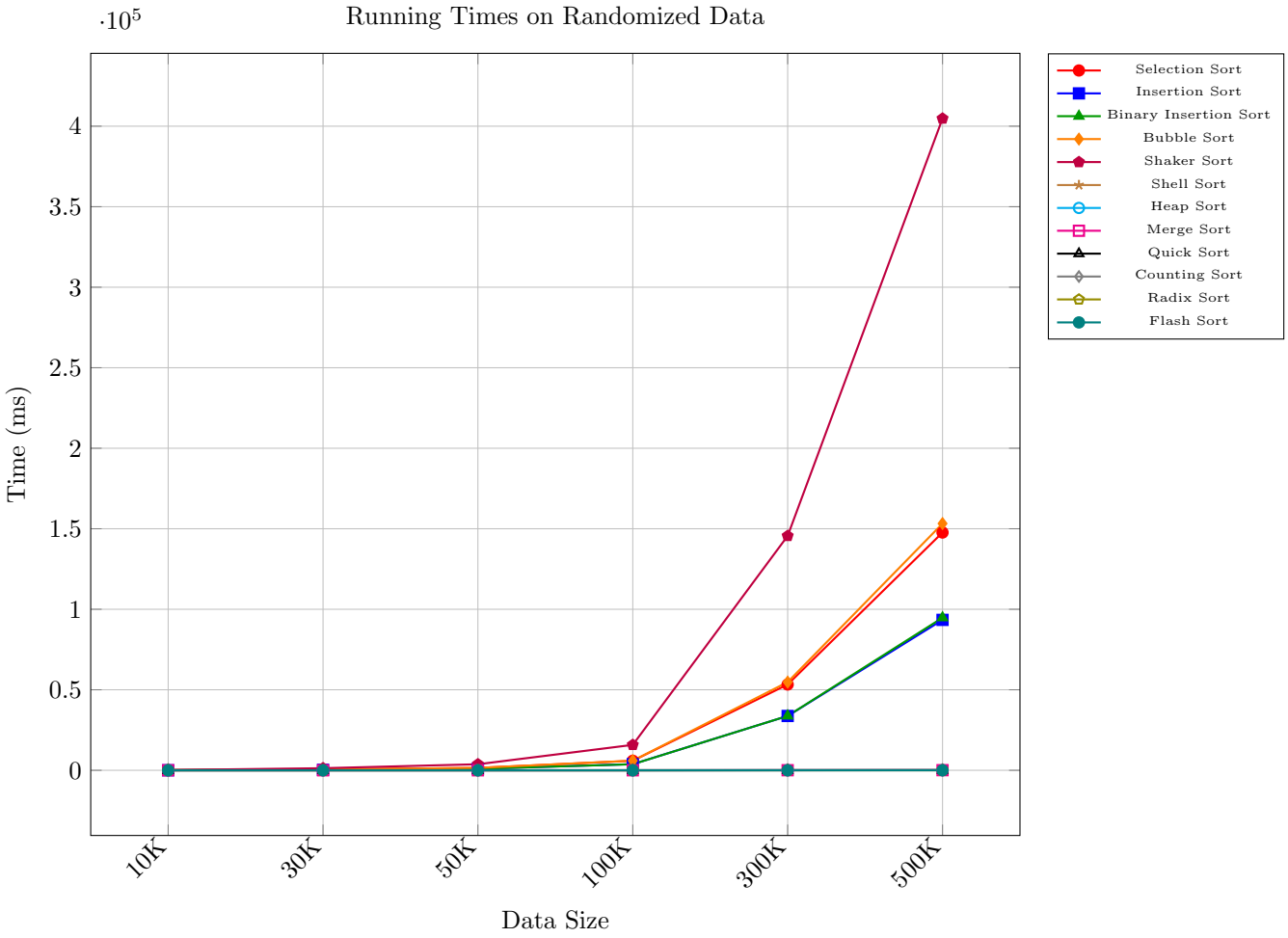

Figure 1: Running times of sorting algorithms on randomized data.

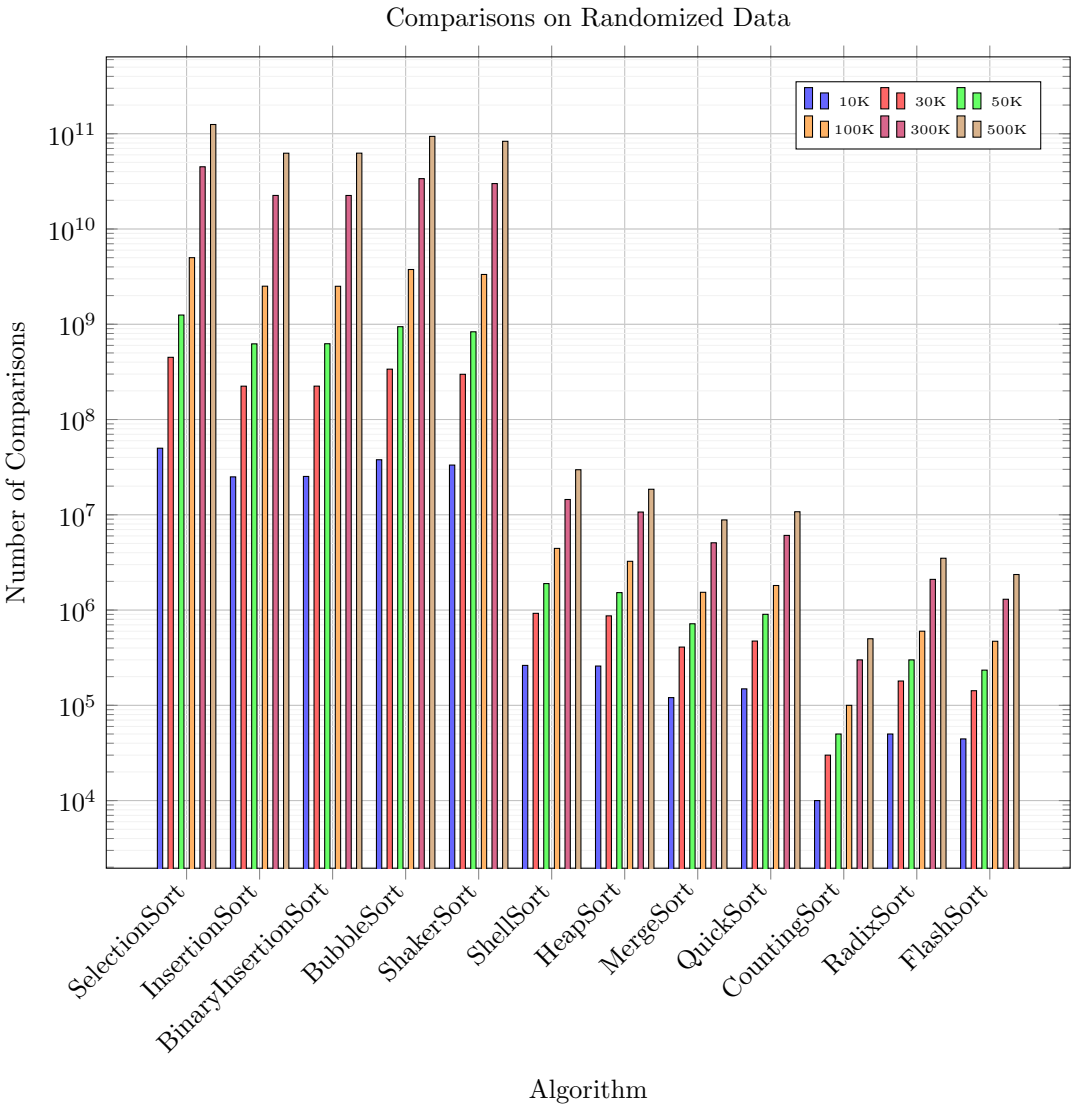

Figure 2: Number of comparisons of sorting algorithms on randomized data.

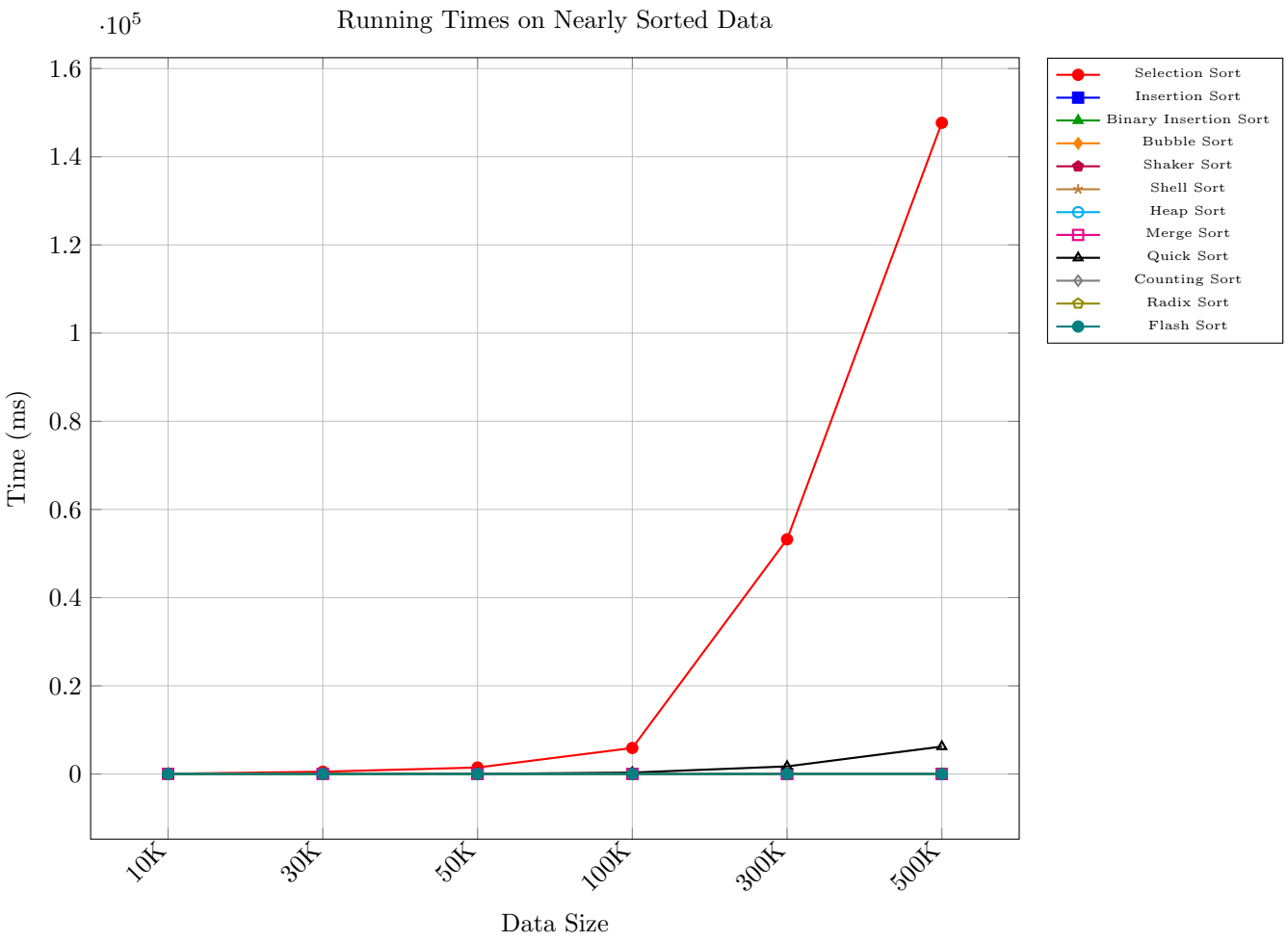

Figure 3: Running times of sorting algorithms on nearly sorted data.

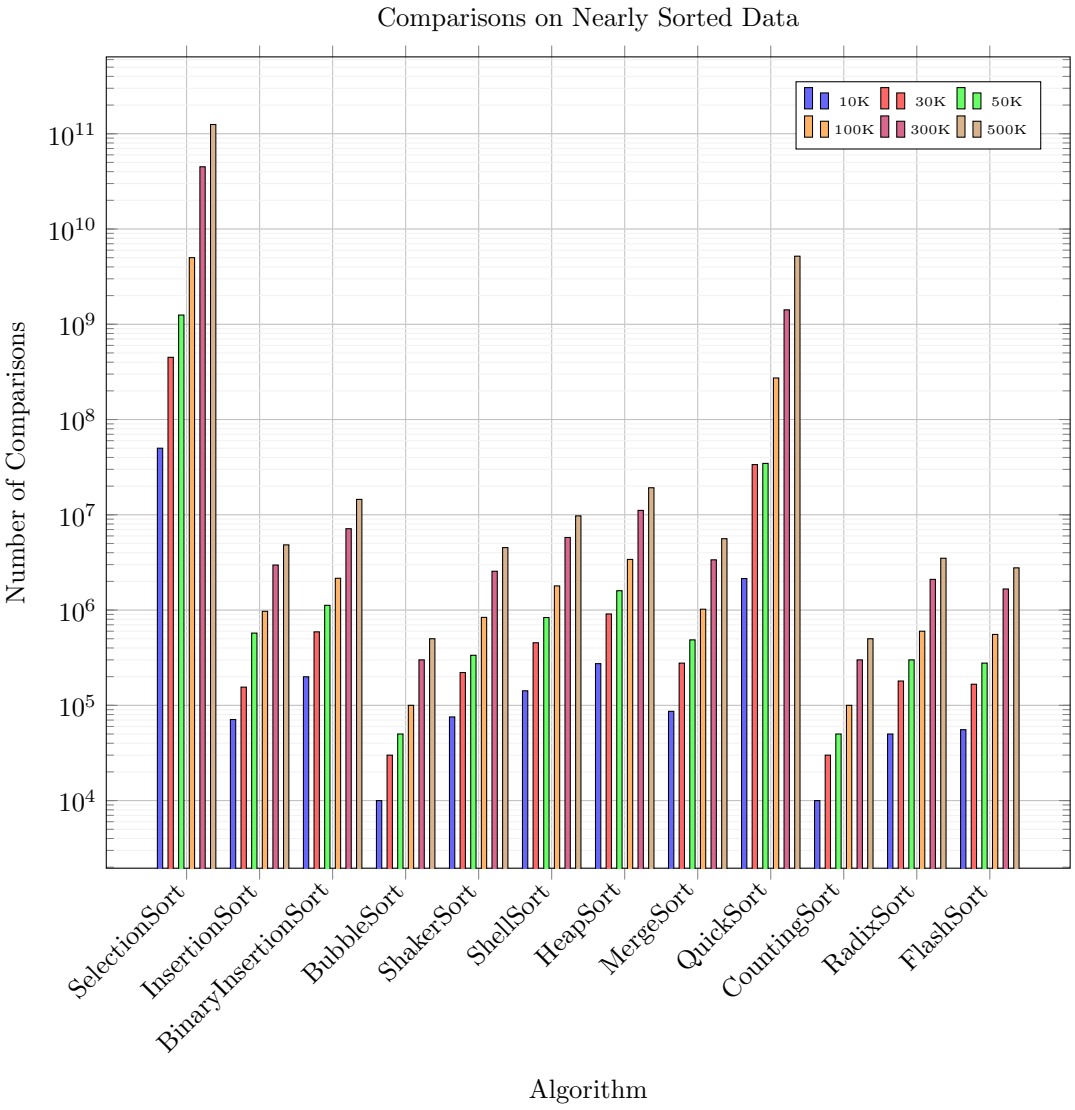

Figure 4: Number of comparisons of sorting algorithms on nearly sorted data.

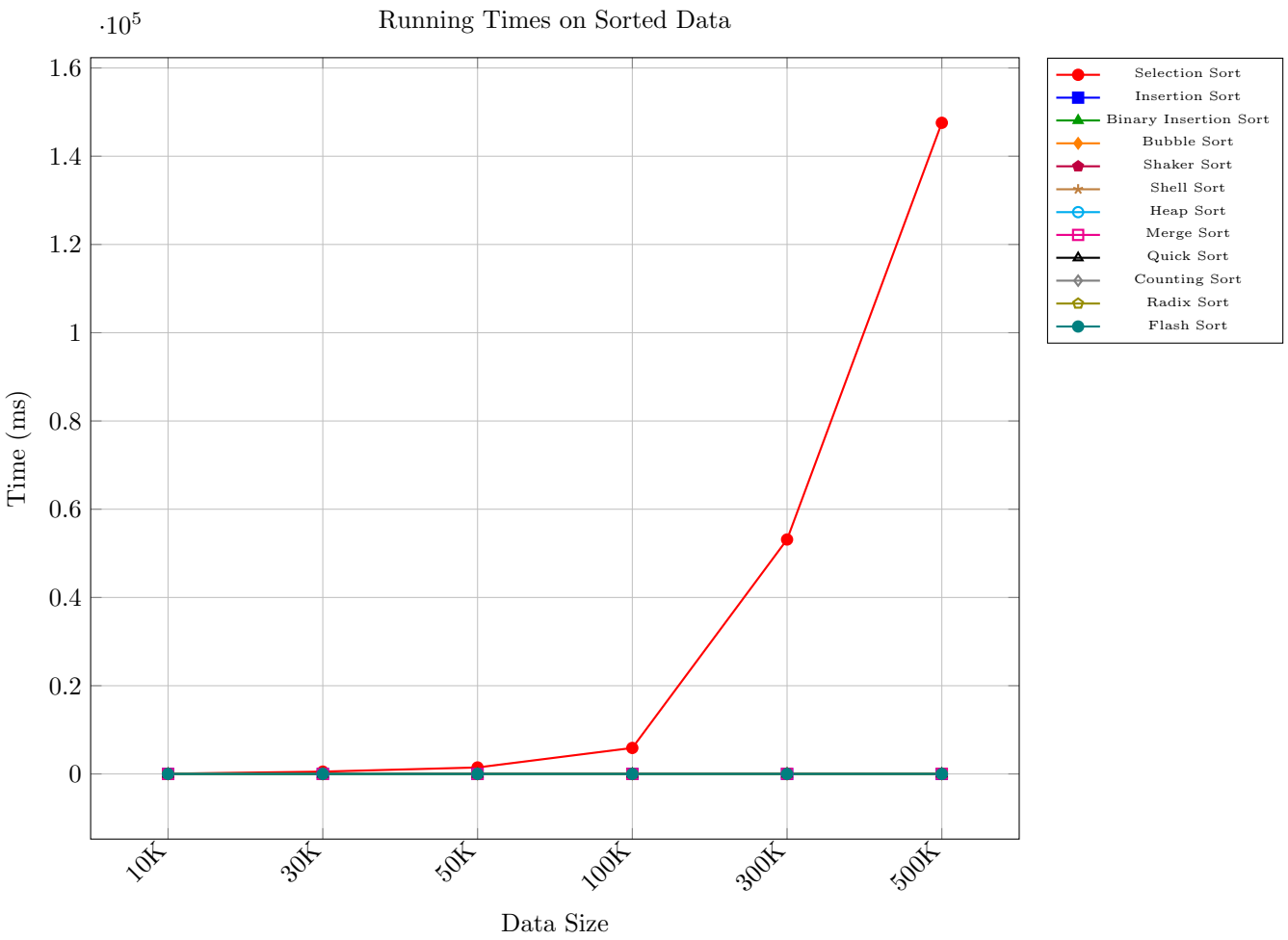

Figure 5: Running times of sorting algorithms on sorted data.

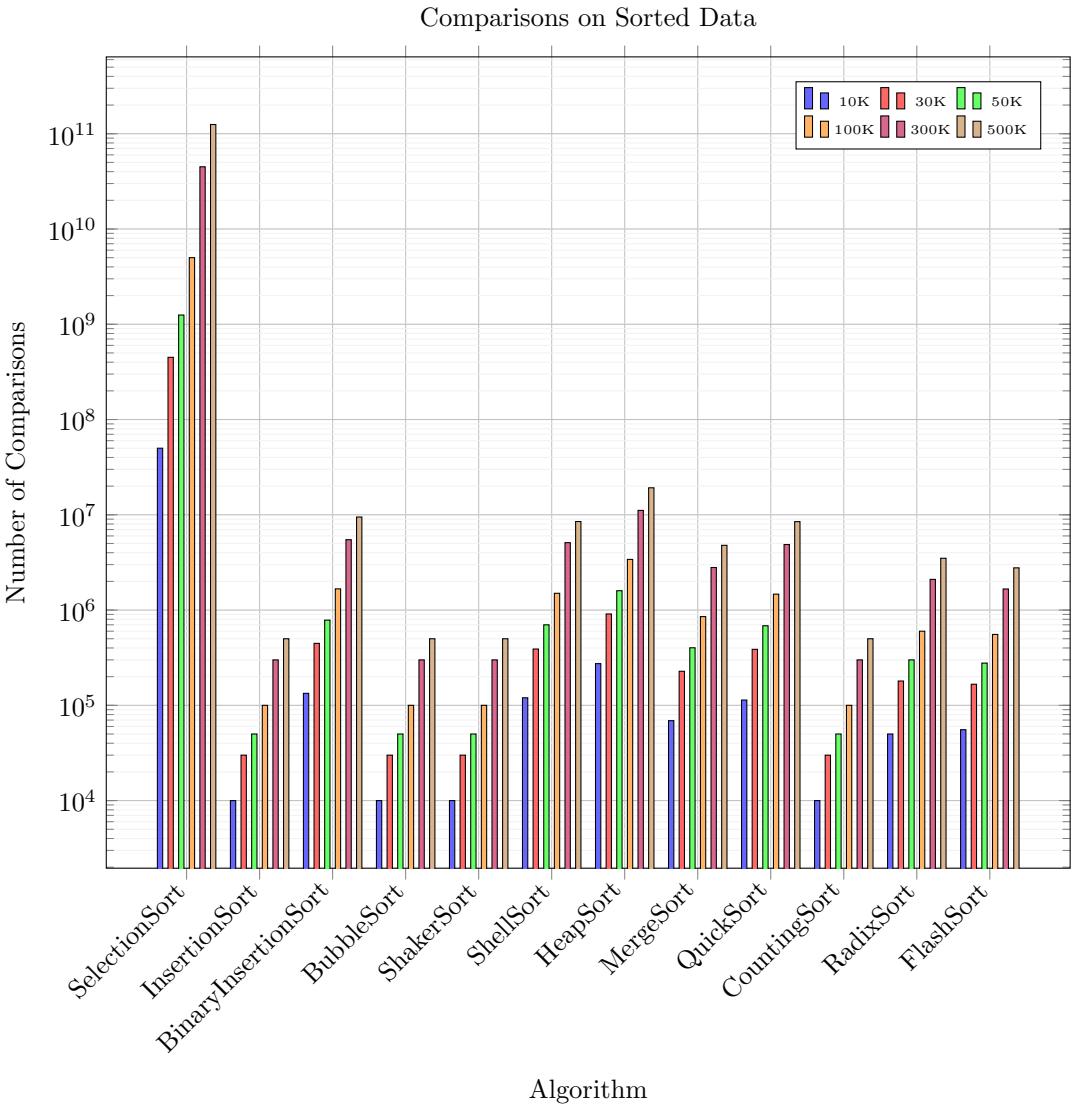

Figure 6: Number of comparisons of sorting algorithms on sorted data.

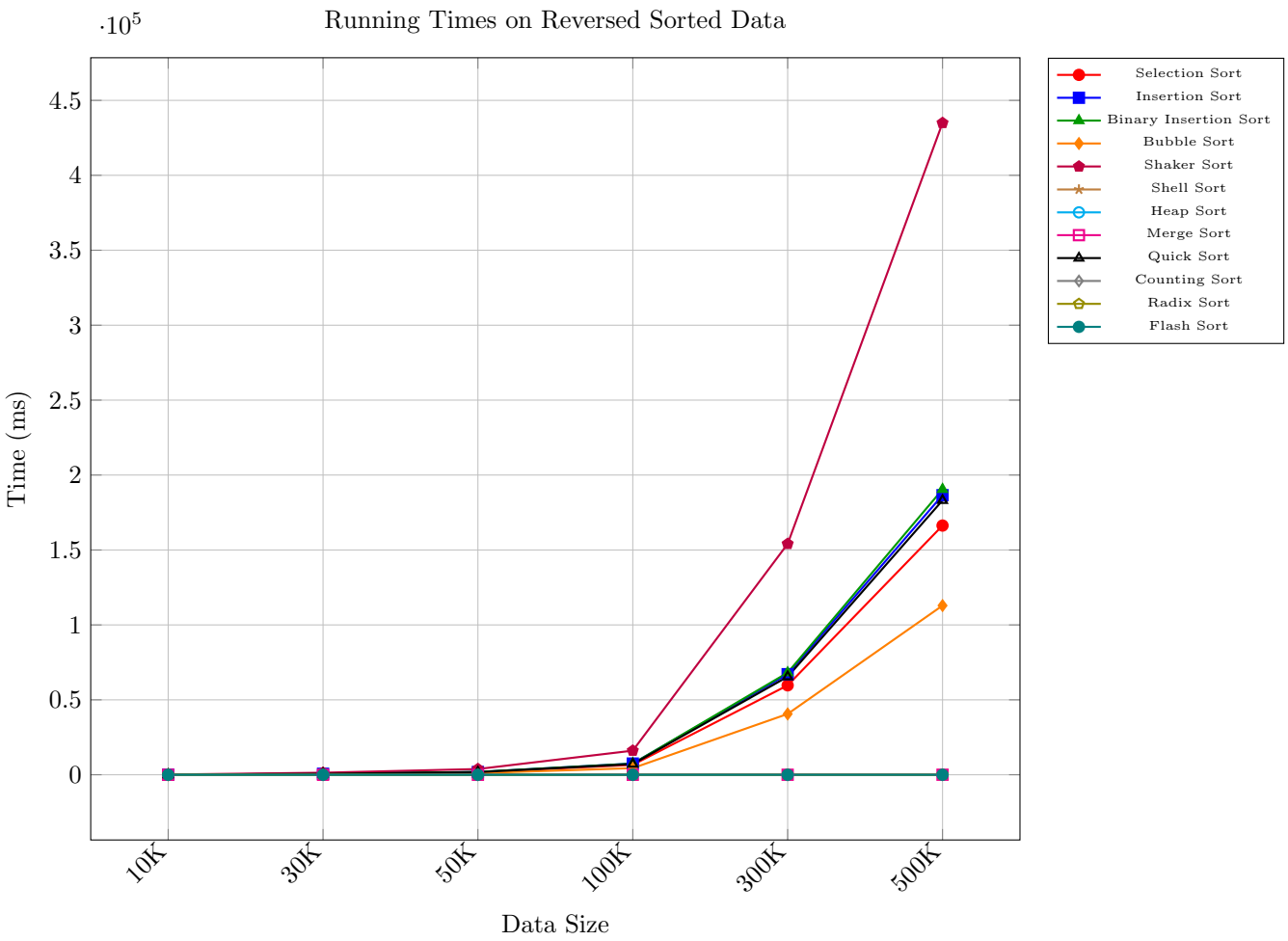

Figure 7: Running times of sorting algorithms on reversed sorted data.

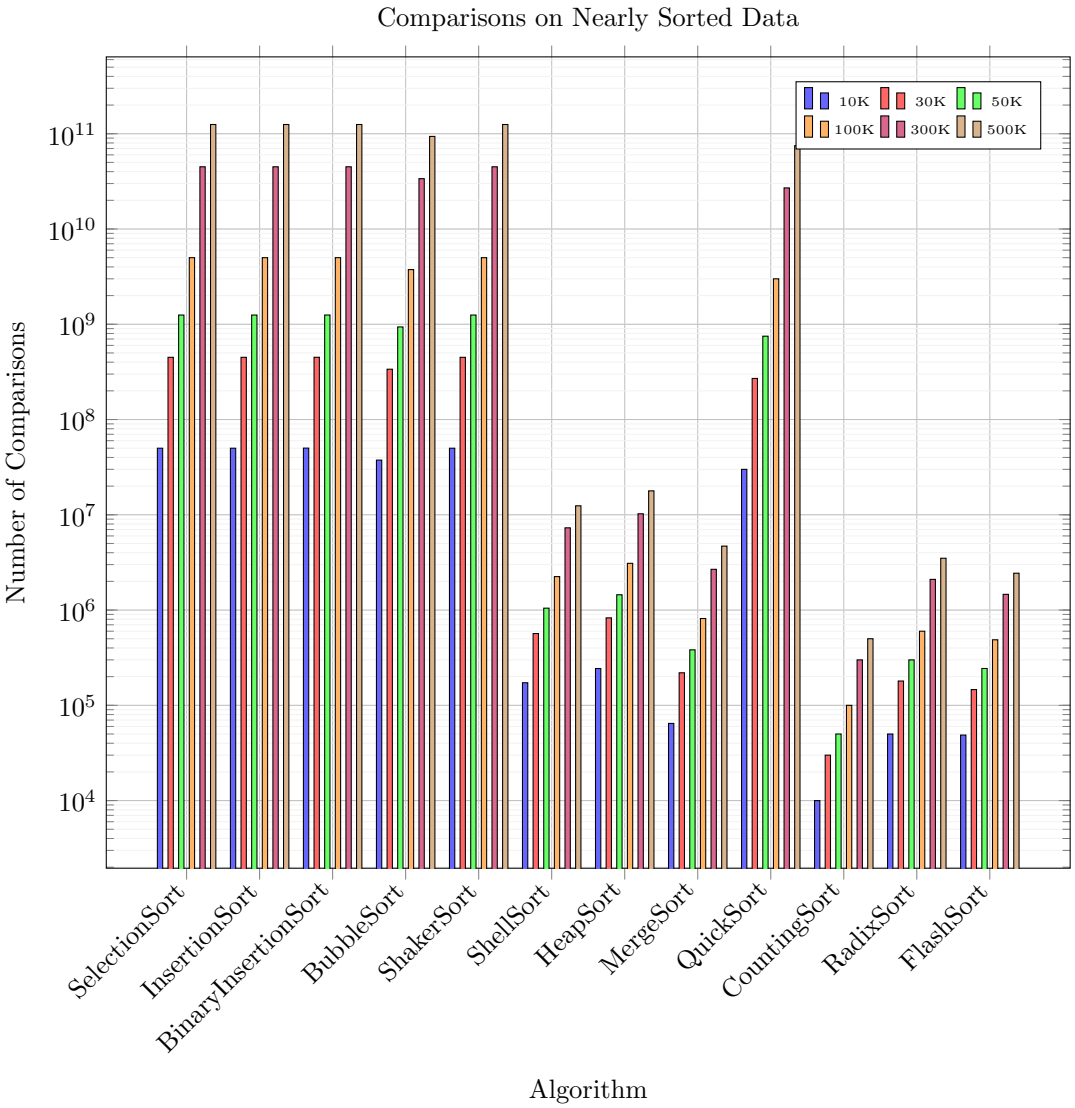

Figure 8: Number of comparisons of sorting algorithms on nearly sorted data.

3.3 Comments

3.3.1 Randomized Data

Running Times:

- **Fastest Algorithms:** Counting Sort, Radix Sort, and Flash Sort consistently show the lowest running times (0–18 ms across all sizes), with Counting Sort often at 0 ms for smaller sizes due to its linear complexity $O(n + k)$. These non-comparison-based algorithms excel on random data where comparison overhead is avoided.
- **Slowest Algorithms:** Shaker Sort (136–404,764 ms) and Selection Sort (59–147,670 ms) are the slowest, reflecting their $O(n^2)$ worst-case complexity. Shaker Sort's bidirectional bubbling increases overhead compared to Bubble Sort (83–153,244 ms).
- **Scalability:** Algorithms like Merge Sort (1–86 ms), Heap Sort (1–102 ms), and Quick Sort (1–52 ms) scale well, maintaining low times even at 500K elements due to their $O(n \log n)$ complexity. Quadratic algorithms (e.g., Insertion Sort: 37–93,414 ms) show exponential growth in time as data size increases.
- **Trends:** Shell Sort (1–105 ms) outperforms other $O(n^2)$ algorithms due to its gap-based optimization, approaching $O(n^{1.3})$.

Comparisons:

- **Least Comparisons:** Counting Sort (9,999–499,999) uses the fewest comparisons, as it's non-comparative, followed by Radix Sort (49,999–3,499,999) and Flash Sort (44,407–2,361,430), which leverage distribution or digit-based sorting.
- **Most Comparisons:** Selection Sort (49,995,000–124,999,750,000) consistently performs the maximum number of comparisons ($n(n-1)/2$), followed by Binary Insertion Sort and Insertion Sort (around 25M–62B), though Binary Insertion reduces comparisons slightly via binary search.
- **Anomalies:** Quick Sort (151,633–11,184,327) uses fewer comparisons than Merge Sort (120,475–8,836,885) on average, reflecting its efficient partitioning, though it's still $O(n \log n)$.

3.3.2 Nearly Sorted Data

Running Times:

- **Fastest Algorithms:** Bubble Sort, Insertion Sort, Counting Sort, Radix Sort, and Flash Sort show near-zero times (0–15 ms) for most sizes, leveraging the data's partial order. Insertion Sort (0–7 ms) excels with its $O(n)$ best-case performance.
- **Slowest Algorithms:** Quick Sort (40–98,641 ms) performs poorly, likely due to poor pivot choices (e.g., first/last element) on nearly sorted data, degrading to $O(n^2)$. Selection Sort (59–147,689 ms) remains slow, unaffected by data order.
- **Scalability:** Merge Sort (0–45 ms) and Heap Sort (1–73 ms) scale linearly, while Shaker Sort (0–13 ms) benefits from bidirectionality on nearly sorted data, outperforming Bubble Sort in comparisons but not time.
- **Trends:** Binary Insertion Sort (0–34 ms) is slower than Insertion Sort due to shift overhead despite fewer comparisons.

Comparisons:

- **Least Comparisons:** Bubble Sort (9,999–499,999) and Insertion Sort (70,999–4,834,901) minimize comparisons due to early termination or minimal shifts. Counting Sort (9,999–499,999) remains constant.
- **Most Comparisons:** Quick Sort (15,225,649–32,458,243,635) exhibits quadratic behavior, confirming the anomaly in running time. Selection Sort (49,995,000–124,999,750,000) is again the highest.
- **Anomalies:** Shaker Sort (75,555–4,523,603) uses fewer comparisons than Bubble Sort on smaller sizes but scales worse due to bidirectional passes.

3.3.3 Sorted Data

Running Times:

- **Fastest Algorithms:** Insertion Sort, Bubble Sort, Shaker Sort, Counting Sort, Radix Sort, and Flash Sort (0–15 ms) dominate, with $O(n)$ best-case performance for adaptive algorithms and linear time for non-comparative ones.
- **Slowest Algorithms:** Quick Sort (158–409,288 ms) is the slowest due to its $O(n^2)$ worst-case on sorted data with a bad pivot. Selection Sort (59–147,574 ms) remains consistently slow.
- **Scalability:** Merge Sort (0–44 ms) and Heap Sort (1–73 ms) maintain $O(n \log n)$ efficiency, while Shell Sort (0–17 ms) benefits from minimal gaps on sorted data.
- **Trends:** Binary Insertion Sort (0–25 ms) is slower than Insertion Sort due to binary search overhead being unnecessary on fully sorted data.

Comparisons:

- **Least Comparisons:** Insertion Sort, Bubble Sort, and Shaker Sort (9,999–499,999) tie with Counting Sort, as they detect the sorted state early. Flash Sort (55,494–2,774,994) is also low.
- **Most Comparisons:** Quick Sort and Selection Sort both hit $n(n-1)/2$ (49,995,000–124,999,750,000), confirming Quick Sort's degradation.
- **Anomalies:** Binary Insertion Sort (133,616–9,475,712) uses more comparisons than Insertion Sort due to binary search, despite no advantage on sorted data.

3.3.4 Reversed Sorted Data

Running Times:

- **Fastest Algorithms:** Counting Sort, Radix Sort, and Flash Sort (0–15 ms) remain the fastest, unaffected by data order. Shell Sort (0–26 ms) also performs well due to its gap optimization.
- **Slowest Algorithms:** Shaker Sort (151–434,947 ms) and Insertion Sort (75–186,561 ms) are the slowest, hitting their $O(n^2)$ worst-case due to maximum shifts or passes.
- **Scalability:** Merge Sort (0–44 ms) and Heap Sort (1–72 ms) scale predictably at $O(n \log n)$, while Bubble Sort (47–112,917 ms) outperforms Shaker Sort due to fewer operations per pass.
- **Trends:** Quick Sort (116–319,322 ms) degrades to $O(n^2)$, similar to sorted data, due to poor pivot selection.

Comparisons:

- **Least Comparisons:** Counting Sort (9,999–499,999) and Radix Sort (49,999–3,499,999) lead, followed by Flash Sort (48,747–2,437,497). Bubble Sort (37,507,497–93,750,374,997) is lower than other $O(n^2)$ algorithms.
- **Most Comparisons:** Binary Insertion Sort (50,118,630–125,008,725,731) slightly edges out Insertion Sort and Shaker Sort (both 49,995,000–124,999,750,000), with Selection Sort tied at the maximum.
- **Anomalies:** Shell Sort (172,578–12,428,778) uses significantly fewer comparisons than other $O(n^2)$ algorithms, aligning with its $O(n^{1.3})$ behavior.

3.3.5 Overall Comment

Across all data orders and sizes, **Counting Sort**, **Radix Sort**, and **Flash Sort** consistently emerge as the fastest algorithms (0–18 ms), leveraging their non-comparative nature and linear or near-linear complexity ($O(n + k)$, $O(d(n + k))$, and $O(n)$, respectively). They also use the fewest comparisons (up to 3.5M at 500K), making them highly efficient for large datasets with known ranges or digit-based properties. Conversely, **Shaker Sort** and **Selection Sort** are the slowest overall (up to 434,947 ms and 166,294 ms at 500K), with Selection Sort performing the most comparisons (up to 124,999,750,000) due to its rigid $O(n^2)$ structure, unaffected by data order.

- **Stable Performers:** Merge Sort and Heap Sort maintain consistent $O(n \log n)$ performance (0–86 ms and 1–102 ms), making them reliable across all scenarios, though Merge Sort uses fewer comparisons (up to 8.8M vs. 19.2M for Heap Sort).
- **Adaptive Algorithms:** Insertion Sort, Bubble Sort, and Shaker Sort excel on nearly sorted or sorted data (0–15 ms), achieving $O(n)$ best-case, but falter on randomized or reversed data (up to 186,561 ms).
- **Unstable Outlier:** Quick Sort varies widely (1–409,288 ms), excelling on randomized data ($O(n \log n)$) but degrading to $O(n^2)$ on sorted or reversed data due to poor pivot choices, a notable anomaly in its performance profile.
- **Efficiency Grouping:** Non-comparative sorts (Counting, Radix, Flash) lead in speed and scalability, followed by $O(n \log n)$ sorts (Merge, Heap, Shell), while $O(n^2)$ sorts (Selection, Insertion, Bubble, Shaker) lag, with Shell Sort as an exception due to its optimization.

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/