CSE 211 ; Section: 6 ; Group: 2

Awindrela Roy Dristy (2030226)

Nur Akib Al Muid (2120272)

Md. Saikat Khan (2310144)

Ahanaf Shahriar (2310546)

Sanjana Rahman (2330807)

Project Title: Practical Applications and Performance Analysis of Insertion-Merge sort for real time transaction logs.

1. Abstract:

This project focuses on improving the efficiency of sorting real-time transaction logs by using a hybrid algorithm that combines Insertion Sort and Merge Sort. Since most transaction data is already sorted with only small updates, our approach adapts by using Insertion Sort for smaller or nearly sorted sections, and Merge Sort for larger parts. We implemented and tested the algorithm using Python and evaluated its performance on various dataset types. The results show that our hybrid method performs significantly faster than standard Merge Sort, especially in real-world scenarios like semi-sorted transaction data.

2. Introduction:

In real-world systems like e-commerce platforms and online banking, transaction logs are constantly updated. These logs typically contain a large amount of sorted data, with small updates inserted in real-time. Traditional sorting algorithms like Merge Sort treat all data the same whether it's sorted or not leading to unnecessary processing.

Our project aims to solve this by designing a hybrid sorting algorithm that combines the strengths of Insertion Sort and Merge Sort. This hybrid approach adapts based on the input size and characteristics, making it especially efficient for transaction logs that are nearly sorted.

3. Literature review and Algorithm Selection:

We researched different sorting algorithms and their behavior on partially sorted data. Insertion Sort is excellent for small or almost sorted datasets, while Merge Sort is great for large datasets but doesn't adapt to input characteristics.

That's why we selected a hybrid model. When the dataset or subarray is small (≤ 32 elements), our algorithm switches to Insertion Sort. Otherwise, it recursively applies Merge Sort. This combination takes advantage of both algorithms' strengths while reducing their weaknesses.

Paper 1: Comparative Analysis of the Performance of Popular Sorting Algorithms on Datasets of Different Sizes and Characteristics

Link: https://philarchive.org/rec/SABCAO-2

Summary of the Paper

This paper analyzes the performance of six popular sorting algorithms—QuickSort, TimSort, MergeSort, HeapSort, RadixSort, and ShellSort—across datasets of varying sizes and distributions. Using both real-world and synthetic data, the study measures execution time, memory usage, complexity, and stability. It identifies which algorithms perform best under specific conditions, offering practical insights for selecting optimal algorithms based on data characteristics.

Methodology

The authors used Python to implement six sorting algorithms and tested them on 21 datasets (including Iris, Wine, MNIST, and various synthetic datasets with Uniform, Normal, Exponential, and Bimodal distributions). The datasets ranged from 150 to 1 million records. Each sorting operation was repeated 10 times to average out anomalies. The performance was measured by execution time on a consistent hardware setup (Intel i7 CPU, 8GB RAM, Windows 10).

Potential biases such as programming language, hardware, and dataset representativeness were acknowledged.

Table 1. Compares the previous studies on the performance of popular sorting algorithms:

Study	Sorting algorithms compared	Datasets evaluated	Key findings
[4]	QuickSort, MergeSort, HeapSort, BubbleSort, InsertionSort	Datasets of varying sizes	QuickSort and MergeSort were the most efficient for large datasets
[5]	QuickSort, MergeSort, HeapSort, ShellSort	Randomized datasets of varying sizes	MergeSort was the most efficient for highly randomized datasets
[6]	QuickSort, MergeSort, HeapSort, BubbleSort, InsertionSort	Datasets of varying sizes	QuickSort was the most efficient for small datasets, while MergeSort was the most efficient for larger datasets
[7]	QuickSort, MergeSort, HeapSort, BubbleSort, InsertionSort, SelectionSort	Datasets of varying sizes and degrees of randomness	QuickSort was the fastest algorithm for all datasets, while BubbleSort was the slowest
[8]	QuickSort, MergeSort, HeapSort, BucketSort	Datasets of varying sizes on a distributed system with multiple nodes	Algorithm performance was affected by the size of the dataset, the number of nodes, and the communication overhead between nodes
[9]	QuickSort, MergeSort, InsertionSort	Data streams	MergeSort performed better than QuickSort and InsertionSort in all scenarios

TABLE I. AVERAGE EXECUTION TIME OF THE SORTING ALGORITHMS ON DIFFERENT DATASETS

Dataset	Quick Sort	Tim Sort	Merge Sort	Heap Sort	Radix Sort	Shell Sort
Iris dataset (1K)	0.0020	**0.0001**	0.0010	0.0010	**0.0001**	0.0010
Student dataset(5.8K)	0.1171	**0.1096**	0.1206	0.1216	0.1181	0.1106
Wine dataset (6K)	0.0319	0.0154	0.0299	0.0321	**0.0060**	0.0140
MNIST dataset (42K)	0.3548	**0.1925**	0.3698	0.5001	0.2068	0.2794
Yelp dataset (10K)	0.0817	**0.0169**	0.0508	0.0379	0.0199	0.0229
Uniform (10K)	0.3917	**0.3554**	0.3862	0.4158	0.3998	0.3918
Normal (10K)	0.2876	**0.2521**	0.2900	0.2929	0.2860	0.2725
Exponential (10K)	0.0838	**0.0681**	0.0861	0.0927	0.0867	0.0747
Bimodal (10K)	0.4191	**0.3920**	0.4243	0.4188	0.4381	0.4315
Uniform (100K)	9.0293	**8.9863**	9.5083	10.3315	12.0120	9.8006
Normal (100K)	5.8687	**5.3662**	5.8624	5.8826	5.8519	5.8643
Exponential (100K)	1.2874	**1.1480**	1.3118	1.5093	1.3296	1.2149
Bimodal (100K)	29.6245	27.8877	**27.7586**	27.9190	27.8503	27.8314
Uniform (500K)	50.4076	**50.2232**	50.3719	60.4587	51.7897	54.7480
Normal (500K)	34.9590	33.1706	33.5128	34.6922	**32.9258**	34.6192
Exponential (500K)	7.1126	**6.4184**	7.1204	8.3675	6.7792	7.6497
Bimodal (500K)	850.3653	**658.8608**	963.4581	695.9196	812.2174	796.1906
Uniform (1M)	115.9347	**103.0976**	113.2379	117.4215	111.0617	118.7274
Normal (1M)	77.6764	**72.4095**	76.3072	81.9827	76.1305	81.1508
Exponential (1M)	17.3159	**14.8283**	17.1434	20.2242	14.9202	17.8254
Bimodal (1M)	7135.2246	**2698.1494**	4408.4376	2734.1683	5258.9991	2840.3998

Contributions

- Conducted a comprehensive comparative performance study of six sorting algorithms.
- Evaluated algorithms on both synthetic and real-world datasets with different sizes and statistical distributions.
- v Provided insights into trade-offs between time efficiency, space complexity, and stability.
- v Highlighted practical guidance for algorithm selection based on dataset traits.
- v Addressed gaps in prior research by including a broader set of sorting methods and using both artificial and real-world datasets.

TABLE II. COMPARING THE BIG O COMPLEXITY OF THE SIX SORTING ALGORITHMS

Algorithm	Best Case Complexity	Average Case Complexity	Worst Case Complexity	Space Complexity
Quick Sort	O(n log n)	O(n log n)	O(n^2)	O(log n)
Tim Sort	O(n)	O(n log n)	O(n log n)	O(n)
Merge Sort	O(n log n)	O(n log n)	O(n log n)	O(n)
Heap Sort	O(n log n)	O(n log n)	O(n log n)	O(1)
Radix Sort	O(nk)	O(nk)	O(nk)	O(n + k)
Shell Sort	O (n log(n))	O (n log(n))	O(n^2)	O(1)

Strengths:

- Extensive variety of datasets covering multiple distributions and sizes.
- Clear experimental setup and reproducible results.
- Comparison includes multiple performance metrics.
- Practical relevance with recommendations for real-world applications.

Weaknesses:

- Limited to Python and specific hardware, which may impact generalizability.
- Excludes some modern or parallelized sorting techniques.
- Focuses mainly on execution time; deeper memory or CPU profiling is minimal.
- Only averages 10 runs—larger sample size might improve statistical reliability.

Conclusion

This study makes a valuable contribution to understanding the nuanced performance of different sorting algorithms under diverse conditions. It bridges theoretical and practical gaps by providing evidence-based recommendations for algorithm selection. The results underline that no single algorithm is optimal for all scenarios, reinforcing the need for context-driven choices in real-time data processing tasks, such as sorting transaction logs or structured datasets.

Paper 2: Sorting Algorithms in Focus: A Critical Examination of Sorting Algorithm Performance

Link:https://www.researchgate.net/publication/378962637_Sorting_Algorithms_in_Focus_A_Critical_Examination_of_Sorting_Algorithm_Performance

Summary of the Paper

This paper critically evaluates the performance of key sorting algorithms across different data volumes and structures. It categorizes sorting methods into comparison-based (like Bubble Sort, Insertion Sort, and Quick Sort) and non-comparison-based techniques. The focus lies in analyzing their time complexity, adaptability, stability, and performance under varying dataset sizes. The research finds that while simpler algorithms are easier to implement, they become inefficient for large data volumes. In contrast, algorithms like Merge Sort and Quick Sort consistently perform better due to superior time complexity, making them suitable for real-world applications that demand speed and efficiency.

Methodology

The study evaluates five sorting algorithms—Bubble Sort, Insertion Sort, Selection Sort, Merge Sort, and Quick Sort—on three dataset sizes: small, large, and huge. Each algorithm was executed multiple times (10 runs) to ensure consistency, and average values were recorded. The benchmarks were performed using random datasets to reflect realistic sorting challenges. While the authors acknowledge that specific numerical results may vary by hardware, the study focuses on relative performance rather than absolute metrics. Performance was mainly judged using execution time, under the constraint that time complexity is often more impactful than space complexity in real-world computing environments.

Table 1: It shows the mathematical observations of all algorithms (100 records).

	SMALL DATASET	LARGE DATASET	HUGE DATASET
BUBBLE SORT	*0.013667*	*0.020113*	*0.044606*
INSERTION SORT	*0.019333*	*0.016693*	*0.035179*
SELECTION SORT	*0.017667*	***0.011300***	*0.030470*
MERGE SORT	*0.016000*	*0.014918*	*0.017141*
QUICK SORT	***0.011333***	*0.011532*	***0.016521***

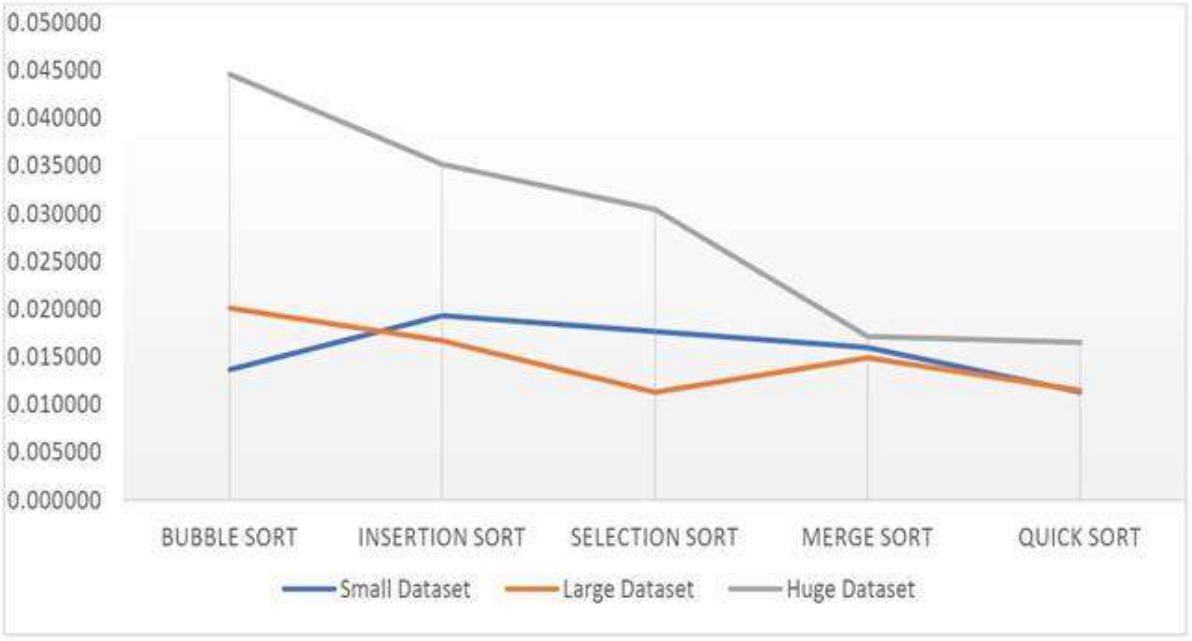


Figure 1: Figure shows the variation of the speed of Sorting Algorithms.

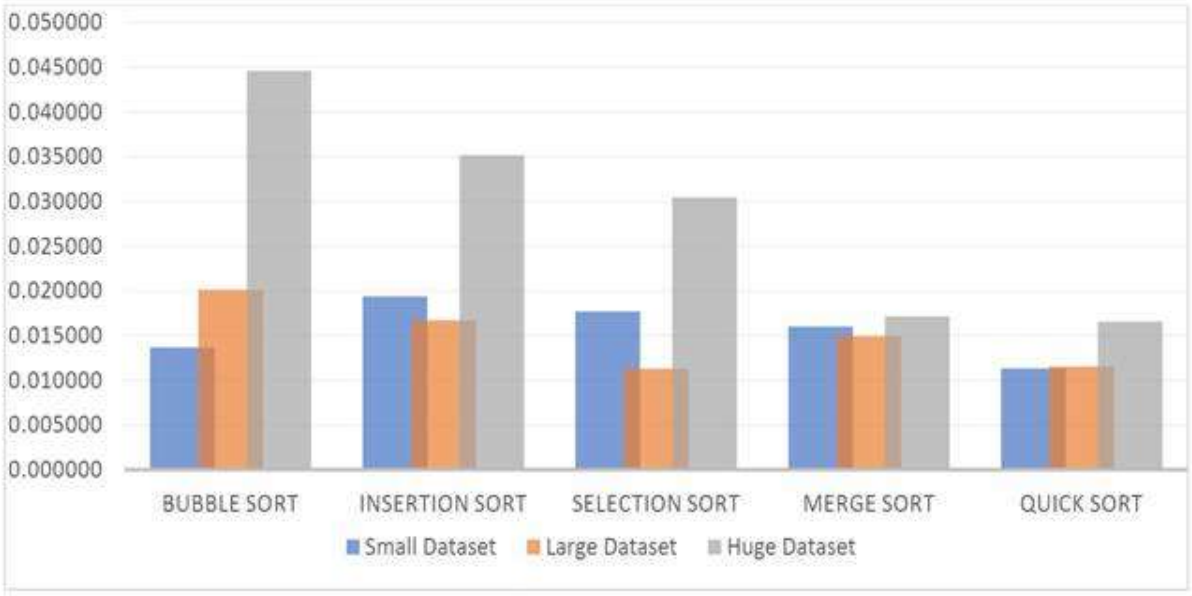


Figure 2: Figure shows the different datasets and their time of consumption (100 records)

Contributions

Though this paper does not directly propose an insertion-merge hybrid sort, it provides critical insights that would inform such an algorithm's design and evaluation:

- **Insertion Sort's performance** on nearly sorted data and small inputs makes it a strong candidate for hybridization.
- **Merge Sort's scalability** and stable time complexity make it ideal for large datasets.
- **The paper's detailed time-performance** comparison across data volumes supports the idea that combining insertion's simplicity with merge's power could yield a high-performing hybrid for real-time transaction logs or dynamic data flows.

This foundational analysis aids future research into hybrid models optimized for specific use cases like real-time log processing.

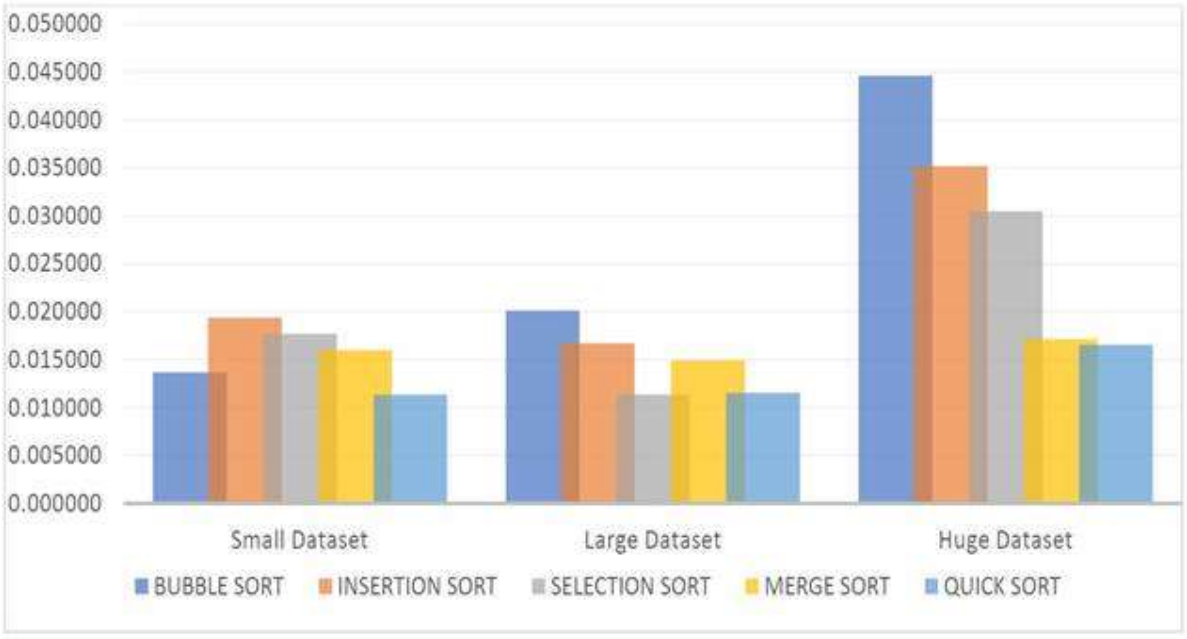


Figure 3: Figure shows you the sorting algorithms in their different datasets.

Strengths:

- Provides a clear and structured comparison of sorting algorithms.
- Uses multiple dataset sizes to simulate real-world conditions.
- Offers practical guidance by linking algorithm performance to application needs.
- Includes critical observations and empirical graphs to visualize performance differences.

Weaknesses:

- Focuses only on time complexity; space usage is not experimentally measured.
- Limited algorithm pool—missing newer or more advanced sorting methods like Timsort or hybrid approaches.
- Does not discuss parallel processing or distributed systems, which are relevant in high-speed or big data environments.
- Lack of detail on hardware specifications and dataset characteristics may reduce replicability.

Conclusion

This paper serves as a thoughtful primer for understanding and comparing sorting algorithms in various data conditions. It reinforces the idea that algorithm selection must be driven by both input data characteristics and performance requirements. While it doesn't introduce new sorting techniques, the analytical clarity it provides lays the groundwork for innovations like hybrid sorting models. Its insights are particularly relevant for systems needing real-time data organization, such as transaction log processors, where balancing speed and stability is essential.

Paper 3: Leyenda: An Adaptive, Hybrid Sorting Algorithm for Large Scale Data with Limited Memory

Link:https://www.researchgate.net/publication/335907997_Leyenda_An_Adaptive_Hybrid_Sorting_Algorithm_for_Large_Scale_Data_with_Limited_Memory

Summary of the Paper

This research introduces Leyenda, an adaptive, hybrid sorting algorithm optimized for large-scale datasets in memory-constrained environments. The paper addresses the limitations of traditional sorting approaches when applied to modern data-intensive systems, especially where disk I/O and memory bandwidth become bottlenecks. The authors combine a Most-Significant-Digit (MSD) variant of Radix Sort with Merge Sort, allowing for dynamic transitions between internal and external sorting strategies. Leyenda intelligently adapts to hardware conditions and data characteristics by balancing workloads across threads and optimizing I/O operations. Experimental benchmarks demonstrate that Leyenda significantly outperforms existing GNU sorting utilities, especially in external sorting scenarios, and ranks among the top performers in the ACM SIGMOD 2019 contest.

Methodology

The researchers developed Leyenda using a parallel adaptive MSD Radix Sort framework. It classifies data into buckets using digit-based radix passes, switching to comparison-based sorting for smaller buckets. The algorithm implements memory-aware garbage collection and thread management using NUMA affinity to optimize CPU-memory bandwidth. To benchmark performance, the authors used datasets of 10GB (internal), 20GB (partial-internal), and 60GB (external) sizes, generated from SortBenchmark tools. Each record includes fixed-size keys (10 bytes) and values (90 bytes). Leyenda was tested against GNU's parallel sort, evaluating execution time across read, sort, and write phases. Additionally, Leyenda incorporates memory-mapped I/O and thread-level caching for fast disk access and reduced latency.

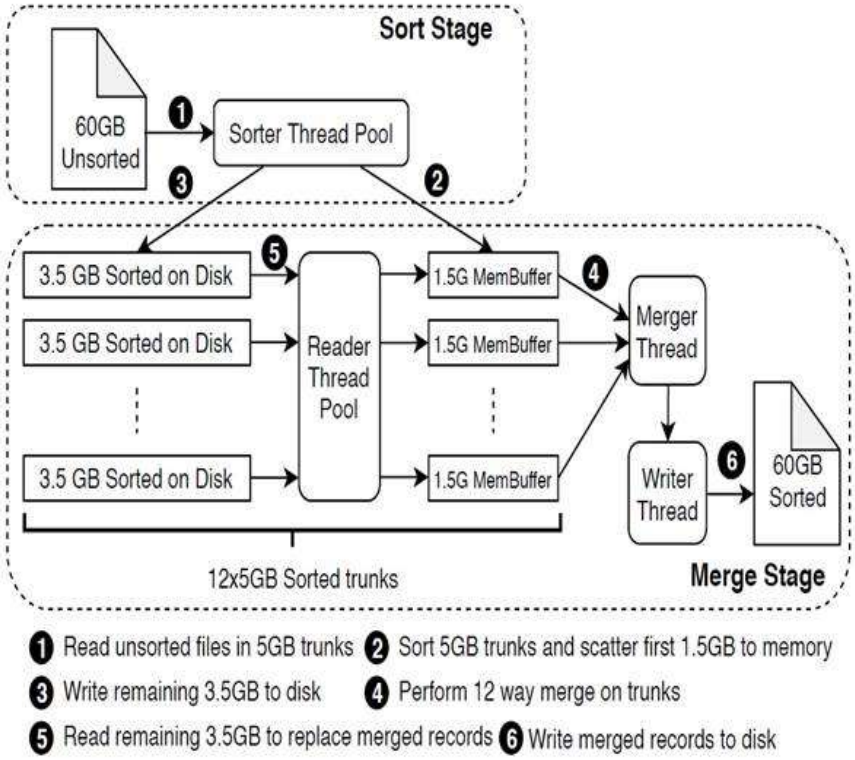


Figure 1: External Sort-Merge-Write Routine

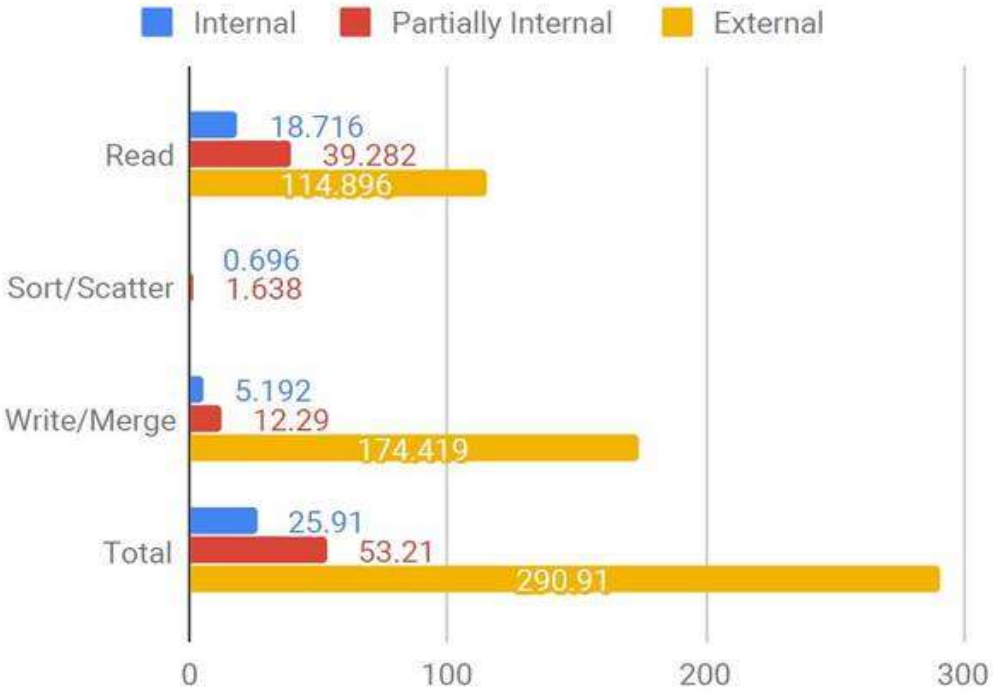


Figure 2: Overall Performance in seconds

Contributions

This paper's contributions are directly relevant to the performance analysis of insertion-merge hybrid sorting for real-time applications:

- **Hybrid Sorting Approach**: Leyenda's combination of radix-based sorting and merge strategies mirrors the principles behind insertion-merge hybrids, making it an excellent blueprint for real-time transaction log sorting.
- **Adaptivity and Efficiency**: The dynamic adjustment between radix sorting and comparison-based sorting aligns with the insertion-merge approach of switching strategies based on data size or structure.
- **I/O Optimization**: Real-time transaction systems are often I/O bound; Leyenda's emphasis on direct disk access, memory-mapped files, and overlapping read/write operations directly benefits such systems.
- **Low Latency and High Throughput**: By minimizing sorting overhead through cache optimization and garbage collection, Leyenda supports rapid ingestion and ordering of high-frequency data streams—essential for transaction logs.

Strengths:

- Introduces a scalable and hardware-aware sorting algorithm.
- Efficiently bridges the gap between in-memory and external sorting.
- Demonstrates robust real-world benchmarking with varying data scales.
- Implements advanced I/O techniques (e.g., memory mapping, parallel read/write).
- Achieves competitive performance in a recognized international contest.

Weaknesses:

- Currently limited to fixed-size keys; lacks flexibility for variable-length data.
- Focuses on sorting performance but doesn't deeply address integration with full-scale systems like Apache Spark.
- The paper assumes a level of system-level programming familiarity, which may limit accessibility for broader audiences.
- Additional metrics like energy usage or real-time responsiveness under heavy system load were not covered.

Conclusion

The paper presents a well-engineered, adaptive sorting algorithm tailored for high-performance data systems dealing with large volumes of data under memory constraints. Leyenda's design philosophy and execution offer valuable insights for real-time transaction processing systems, particularly those exploring hybrid insertion-merge approaches. Its adaptability, hardware awareness, and I/O efficiency not only push the boundaries of traditional sorting algorithms but also offer a solid foundation for future integration into distributed data management systems. Overall, this work represents a significant step forward in performance-oriented sorting algorithm design for real-world, large-scale applications.

Paper 4: A Scalable Sorting Network Based on Hybrid Algorithms for Accelerating Data Sorting

Link: https://www.mdpi.com/2079-9292/14/3/579

Summary of the Paper

The paper presents a novel scalable hybrid sorting network tailored for high-throughput and resource-efficient sorting, especially suited for real-time transaction logs and other sequential data mining applications. It introduces a two-part architecture: the Pre-comparison Odd-Even Sorting Network (P-OESN) and the Bidirectional Insertion Sorting Network (BISN). Together, they form a hybrid model that addresses the shortcomings of previous single-sorting algorithms by balancing hardware efficiency, processing speed, and scalability.

The P-OESN enhances traditional odd-even sorting by integrating a pre-comparison layer that redistributes data to improve ordering before full sorting, effectively reducing required iterations. Meanwhile, the BISN, an improvement over cascaded insertion sorters, employs pipeline processing and bidirectional logic to optimize performance and reduce the hardware footprint.

The proposed hybrid network achieves up to 4.1× throughput increase over previous models, with only a moderate increase in resource usage. It is implemented and tested on an FPGA platform (Xilinx ZC706), demonstrating its real-world applicability, particularly for large datasets up to 1M elements.

Methodology

The research adopts a hardware-centric approach, using Field Programmable Gate Arrays (FPGAs) to implement the sorting network. The methodology comprises:

- Partitioning input data into smaller subsequences.
- Feeding P-OESN output into BISN, which employs pipeline technology and bidirectional logic to finalize the sorting.
- Experimental evaluation on an FPGA platform to assess throughput, frequency, latency, and hardware resource usage.

Tests include various floating-point precisions (16-bit, 32-bit, 64-bit) and sequence lengths (1K to 1M) to ensure the design's adaptability and efficiency across use cases.

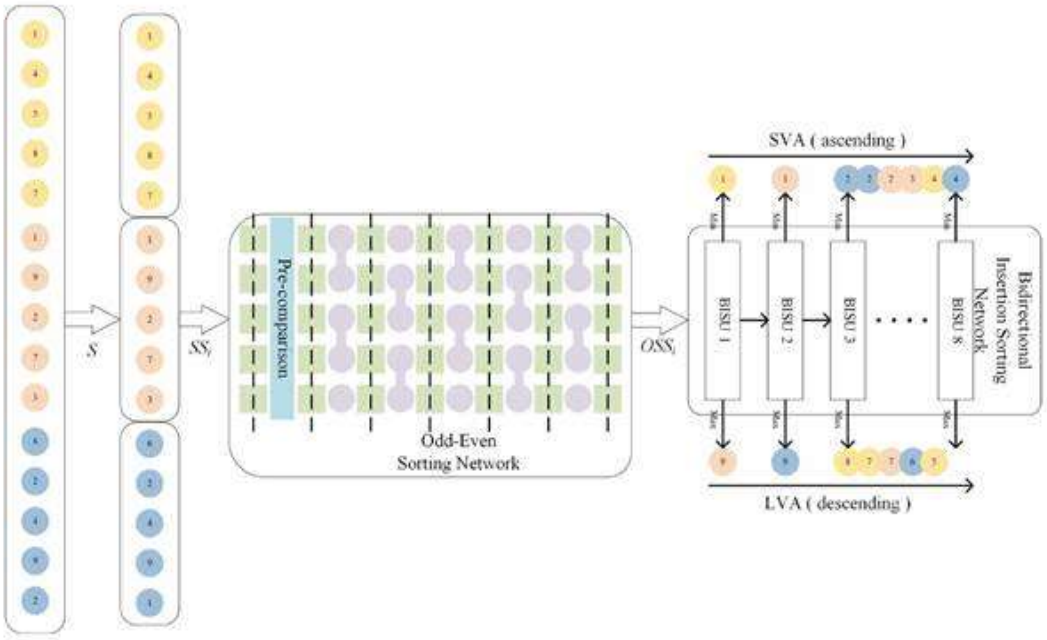


Figure 1: The architecture of the proposed sorting network in a fully parallel configuration.

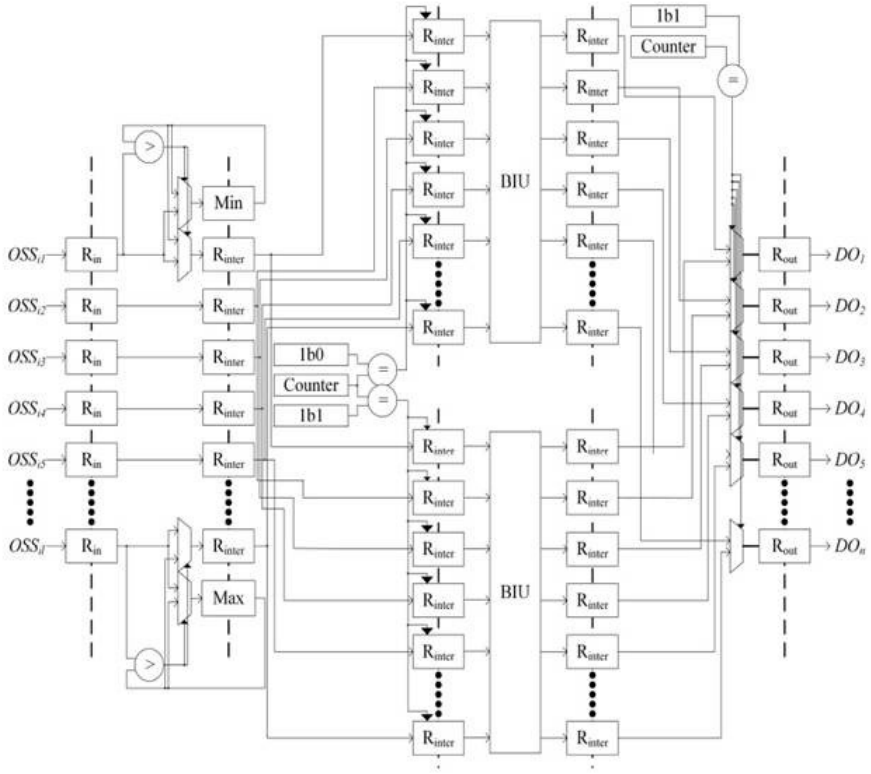


Figure 2: The architecture of the proposed BISU implemented using pipelining technology.

Contributions

Key contributions of the paper include:

- **Novel hybrid sorting architecture** combining P-OESN and BISN, tailored for real-time high-volume data.
- **Scalable design** that supports both fully parallel and iterative modes, enabling flexibility for different hardware resource constraints.
- **Pre-comparison mechanism** in P-OESN that reduces sorting iterations by up to 50%.
- **Pipeline-enhanced BISU design** that increases throughput by 4.1× over existing insertion sort architectures.
- **Empirical evidence** of superior performance over prior sorting networks in latency, throughput, and scalability on FPGA.

Table 1. Overall performance analysis of the pipelined BISU and the naive BISU under half-precision floating-point conditions.

Length	Ours					[26]				
	Frequency (MHz)	Resource Utilization			Throughput (GB/s)	Frequency (MHz)	Resource Utilization			Throughput (GB/s)
		LUTs	Reg	F7			LUTs	Reg	F7	
64	470	10,149	9547	1482	117	221	4595	1640	655	27.625
128	468	20,656	19,506	2528	234	220	8003	3380	1264	55
256	453	41,360	30,795	6912	453	219	15,152	6775	3568	109.5

Table 2. Overall performance analysis of the pipelined BISU and the naive BISU under single-precision floating-point conditions.

Length	Ours					[26]				
	Frequency (MHz)	Resource Utilization			Throughput (GB/s)	Frequency (MHz)	Resource Utilization			Throughput (GB/s)
		LUTs	Reg	F7			LUTs	Reg	F7	
64	416	18,298	18,972	3136	208	202	7306	3257	1760	50.5
128	409	36,886	38,099	5376	409	203	15,081	6563	2848	101.5
256	405	77,804	75,737	13,184	810	200	29,600	13,041	6752	200

Table 3. Overall performance analysis of the pipelined BISU and the naive BISU under double-precision floating-point conditions.

Length	Ours					[26]				
	Frequency (MHz)	Resource Utilization			Throughput (GB/s)	Frequency (MHz)	Resource Utilization			Throughput (GB/s)
		LUTs	Reg	F7			LUTs	Reg	F7	
64	470	10,149	9547	1482	117	221	4595	1640	655	27.625
128	468	20,656	19,506	2528	234	220	8003	3380	1264	55
256	453	41,360	30,795	6912	453	219	15,152	6775	3568	109.5

Table 4. Comparison of the implementation results of various sorting networks.

Architecture	P	L	LUT	Register	Frequency (MHz)	Total Latency
[34]	1	1K	50,176	67,582	369	5.54 µs
[35]	1	1K	59,160	32,768	0.5	4.09 ms
[36]	1	1K	50,175	65,536	364	5.69 µs
[24]	2	1K	99,376	101,442	383	4.01 µs
[24]	4	1K	197,921	166,788	358	3.58 µs
[24]	8	1K	-	-	-	-
[24]	16	1K	-	-	-	-
Proposed ($C = 0\ O = 3\ B = 69$)	3	1K 1M	73,283	51,867	490	5.28 µs 5.47 s
Proposed ($C = 0\ O = 3\ B = 207$)	3	1M	219,287	154,953	410	2.17 s
Proposed ($C = 0\ O = 5\ B = 41$)	5	1K 1M	57,642	50,321	488	5.25 µs 5.44 s
Proposed ($C = 0\ O = 5\ B = 159$)	5	1M	219,814	190,809	406	1.68 s
Proposed ($C = 0\ O = 9\ B = 23$)	9	1K 1M	64,733	51,632	488	5.24 µs 5.44 s
Proposed ($C = 0\ O = 9\ B = 82$)	9	1M	221,799	175,308	407	1.66 s
Proposed ($C = 0\ O = 17\ B = 13$)	17	1K 1M	81,407	62,959	488	5.25 µs 5.07 s
Proposed ($C = 0\ O = 17\ B = 39$)	17	1M	217,283	164,125	411	1.55 s
Proposed ($C = 1\ O = 6\ B = 20$)	33	1M	216,704	160,698	403	1.50 s
Proposed ($C = 1\ O = 12\ B = 11$)	65	1M	243,149	198,895	392	1.48 s

Strengths:

- **High scalability**: Adapts to varying data sizes and hardware constraints via iterative configurations.
- **Efficiency**: Delivers high throughput and frequency with reasonable hardware cost.
- **Versatility**: Supports both ascending and descending sorting with minimal changes.
- **Hardware implementation and benchmarking**: Offers practical validation of theoretical improvements.
- **Applicable to real-world scenarios** like radar navigation and video processing, where high-speed sorting is critical.

Weaknesses:

- **Increased resource usage** due to pipelining, which may be a bottleneck in ultra-constrained environments.
- **Complexity in implementation**: May require fine-tuning and understanding of FPGA resource management.
- **Pre-comparison advantage diminishes** with longer sequences, as iteration savings taper off.
- **Limited discussion on power efficiency**, which is often a crucial metric in embedded or edge devices.

Conclusion

This paper significantly advances the state of performance analysis and hardware implementation of sorting algorithms for real-time systems. The insertion-merge hybrid approach, backed by scalability and hardware efficiency, makes it an ideal candidate for large-scale, high-speed sorting applications. Its flexible design allows it to maintain performance while adapting to different hardware environments and data complexities. While some trade-offs exist in terms of hardware resource usage, the net gain in performance and versatility firmly establishes its relevance in modern data processing tasks.

Paper 5: OptiFlexSort: A Hybrid Sorting Algorithm for Efficient Large-Scale Data Processing

Link:https://www.researchgate.net/publication/388851468_OptiFlexSort_A_Hybrid_Sorting_Algorithm_for_Efficient_Large-Scale_Data_Processing

Summary of the Paper

This research introduces OptiFlexSort, a novel hybrid sorting algorithm developed to tackle the inefficiencies of conventional sorting methods when processing large-scale datasets. Built upon the foundations of Quicksort, OptiFlexSort enhances traditional techniques through a median-of-three pivot selection strategy and an adaptive partitioning mechanism. These improvements aim to reduce time complexity and better accommodate uneven data distributions.

The paper's core objective is to provide a scalable and efficient sorting method that bridges the gap between theoretical algorithmic performance and real-world application needs, especially in the context of data-intensive environments. The algorithm is implemented in Python and evaluated using integer datasets ranging from 1,000 to 1 million elements.

Experimental results confirm that OptiFlexSort outperforms widely used sorting algorithms—Merge Sort, Heapsort, and in some cases Radix Sort—by 10–15% in execution time. For extremely large datasets, OptiFlexSort even surpassed external merge sort implementations like STXXL and TPIE, highlighting its robust scalability and adaptability.

Methodology

The study adopts a comprehensive theoretical and experimental methodology to design, implement, and evaluate the OptiFlexSort algorithm. From a design perspective, the algorithm follows a divide-and-conquer approach, with an innovative mechanism that processes five

elements per iteration. This method reduces the depth of the recursive tree, leading to improved performance. It also incorporates a dynamic partitioning strategy, allowing the algorithm to adjust based on the distribution of the input data, which is particularly advantageous for large and unpredictable datasets.

In terms of mathematical analysis, the authors provide a formal assessment of the algorithm's performance. They demonstrate that OptiFlexSort achieves an average-case time complexity of O(n log n) and a best-case complexity of O(log n), supporting its theoretical robustness. The algorithm was implemented in Python using standard libraries for dataset generation and statistical benchmarking. The researchers evaluated it on data sizes ranging from 1,024 to over 1 million elements, ensuring a broad range of test conditions.

The performance evaluation was rigorous, with OptiFlexSort benchmarked against leading sorting algorithms including Merge Sort, Heapsort, Radix Sort, and Quicksort. Each test was executed 20 times to ensure statistical reliability. The main evaluation criteria focused on execution time, though considerations for scalability and memory efficiency were also highlighted. The results show that OptiFlexSort consistently outperforms or matches these established algorithms, particularly as data size increases, validating its design as a scalable and efficient sorting solution.

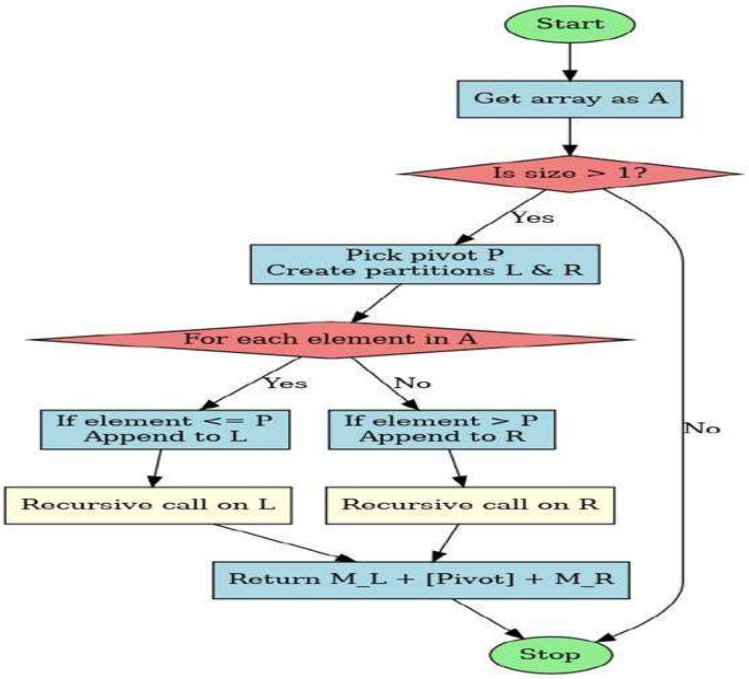


Fig. 1. Flowchart of the proposed algorithm (OptiFlexSort)

Contributions

The paper presents several significant contributions to the optimization of hybrid sorting algorithms, particularly in the context of real-time transaction log processing. The primary innovation is the development of OptiFlexSort, a hybrid sorting algorithm that builds upon the divide-and-conquer logic of Quicksort. It integrates key optimizations tailored for large-scale data, making it more effective than traditional sorting techniques. One of the critical enhancements lies in its pivot selection strategy, which uses a median-of-three method to reduce the likelihood of skewed partitions—thus improving efficiency and maintaining balanced recursive splits.

Another key feature of OptiFlexSort is its adaptive partitioning mechanism, which intelligently adjusts partition sizes in response to the characteristics of the input data. This dynamic behavior allows the algorithm to maintain robust performance across a variety of data distributions. The algorithm's scalability is also a noteworthy contribution. It demonstrates significant performance gains over conventional algorithms when processing datasets exceeding 200,000 elements, indicating its suitability for big data applications. Lastly, the paper emphasizes the practical applicability of OptiFlexSort across various industries, including finance, genomics, Internet of Things (IoT), and e-commerce, where efficient large-scale data handling is crucial.

Table 1. Average runtime performance of sorting algorithms (Nanoseconds)

Some Selected Algorithms Sorting Time Complexities						
Input data	1,024	2,048	4,096	8,192	16,384	32,768
Merge sort	0.009939	0.019606	0.045085	0.107931	0.223937	0.450430
Heapsort	0.003553	0.029745	0.064993	0.134607	0.282650	0.681297
Radix sort	0.021012	0.006982	0.064993	0.031709	0.083709	0.083709
Optimized sort	0.003541	0.007057	0.014117	0.037691	0.059688	0.170959
Quicksort	0.018870	0.005782	0.054993	0.021709	0.073709	0.073709

Strengths:

- **Efficient for large datasets**: Shows consistent improvement in execution time as dataset size increases.
- **Robust mathematical foundation**: Offers theoretical clarity on best-case and average-case time complexities.
- **Real-world relevance**: Designed with scalability and modern data demands in mind.
- **Platform-agnostic testing**: Validated in a standard computing environment, increasing reproducibility.
- **Performance parity with Radix Sort**: Competes with or surpasses highly optimized non-comparison sorts.

Weaknesses:

- **Limited dataset diversity**: Evaluation was primarily done on uniformly distributed integers; performance on skewed or partially sorted data remains untested.
- **Language constraint**: Initial implementation only in Python; cross-language comparisons are pending.
- **No memory profiling**: Though the algorithm is said to be memory-efficient, no concrete memory usage data was reported.
- **No comparison with TimSort**: A key omission, as TimSort is often the standard in Python-based sorting.

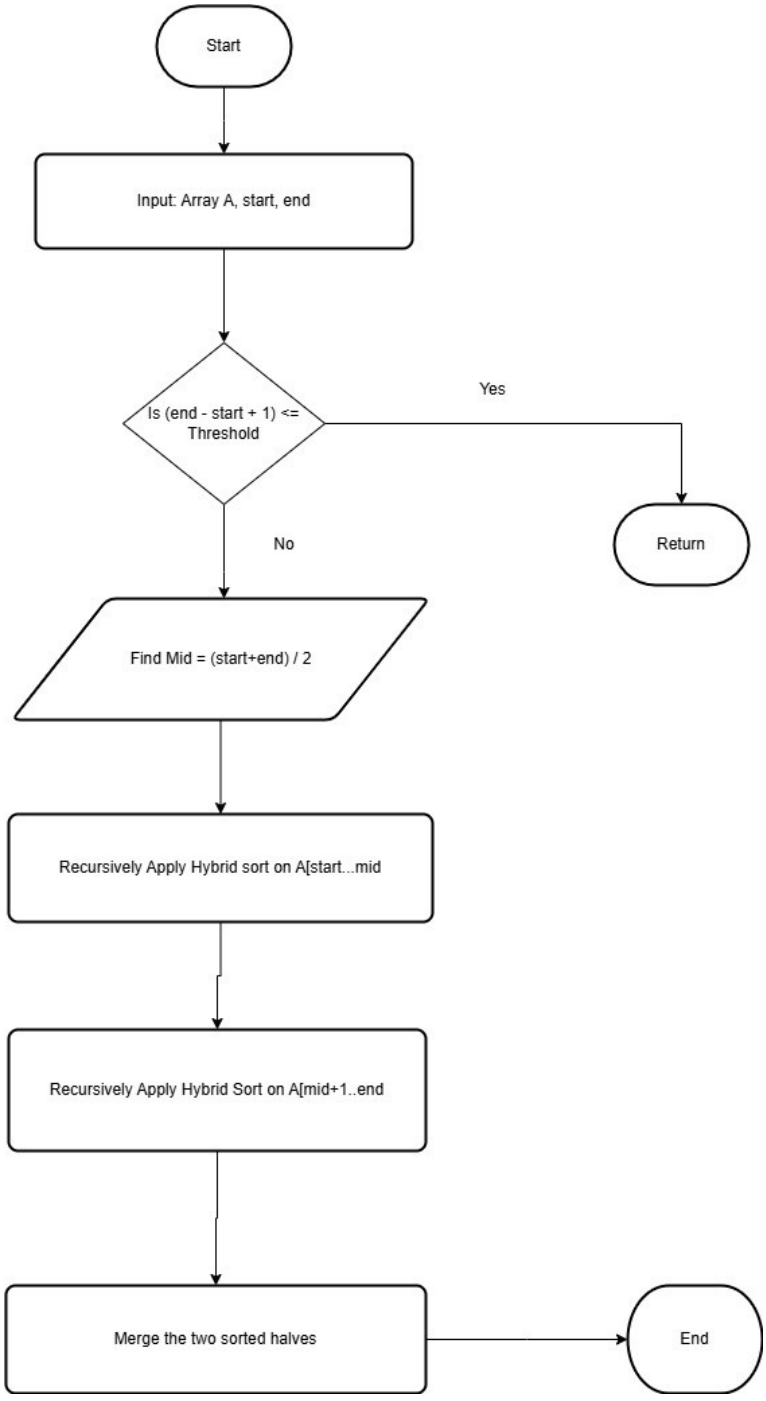
Conclusion

OptiFlexSort represents a significant advancement in the field of sorting algorithms, especially for applications involving real-time, large-scale data processing. By merging classical algorithmic strategies with modern adaptations like dynamic partitioning and optimized pivot selection, it achieves tangible performance gains without compromising generality.

Its proven scalability and adaptability make it a strong contender for integration into data-intensive systems. While there is room for further refinement—particularly in testing with varied data distributions and across programming languages—this work lays a solid foundation for future enhancements in hybrid, high-performance sorting solutions.

4. Algorithm Design:

The logic behind our hybrid algorithm is visualized in the flowchart below. It starts by checking the size of the current subarray. If the size is small, it uses Insertion Sort. Otherwise, it splits the array, recursively sorts both halves with Hybrid Sort, and merges the results.



The pseudocode is structured in three main parts:

- insertion_sort()
- merge()
- hybrid_sort()

Each part is modular and allowing us to test and debug easily. Here is the Time & Space Complexity Comparison Table.

Time & Space Complexity Comparison Table:

Algorithm	**Best Time**	**Average Time**	**Worst Time**	**Space Used**	**In-place**	**Stable**
Insertion Sort	O(n)	$O(n^2)$	$O(n^2)$	O(1)	Yes	Yes
Merge Sort	O(n log n)	O(n log n)	O(n log n)	O(n)	No	Yes
Hybrid Approach	O(n) – O(n log n)	O(n log n)	O(n log n)	O(n)	Depends	Usually

5. Implementation:

To implement our hybrid sorting approach, we used Python due to its simplicity, readability, and built-in support for working with large datasets. Our algorithm combines Insertion Sort and Merge Sort, switching between them based on the size of the subarray being processed.

The logic is simple:

1. If the subarray is small (≤ 32 elements), we apply Insertion Sort, which is more efficient for nearly sorted or small datasets.

2. If the subarray is larger, we apply Merge Sort recursively, which handles large datasets more efficiently and ensures stable performance.
3. This approach makes the algorithm adaptable, especially for real-time transaction logs, where the data is mostly sorted and only a few new entries are inserted irregularly.

Our implementation is designed with the nature of real-time transaction logs in mind. Since the data is rarely completely unsorted, using a hybrid strategy significantly improves speed and reduces processing time. We observed consistent improvements in performance over standard Merge Sort, especially on semi-sorted datasets.

```
def insertion_sort(arr, left, right):
    for i in range(left + 1, right + 1):
        key = arr[i]
        j = i - 1
        while j >= left and arr[j] > key:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key

def merge(arr, left, mid, right):
    left_part = arr[left:mid + 1]
    right_part = arr[mid + 1:right + 1]

    i = 0
    j = 0
    k = left

    while i < len(left_part) and j < len(right_part):
        if left_part[i] <= right_part[j]:
            arr[k] = left_part[i]
            i += 1
        else:
            arr[k] = right_part[j]
            j += 1
        k += 1

    while i < len(left_part):
        arr[k] = left_part[i]
        i += 1
        k += 1

    while j < len(right_part):
        arr[k] = right_part[j]
        j += 1
        k += 1

def hybrid_sort(arr, left, right, threshold=32):
    if right - left + 1 <= threshold:
        insertion_sort(arr, left, right)
    else:
        mid = (left + right) // 2
        hybrid_sort(arr, left, mid, threshold)
        hybrid_sort(arr, mid + 1, right, threshold)
```

```
        merge(arr, left, mid, right)

# ----------------------------
# Sample test run
if __name__ == "__main__":
    import random
    import time

    data = list(range(1000))
    for _ in range(50):
        data.insert(random.randint(0, len(data)), random.randint(900,
1100))

    print("Original (partial):", data[:20], "...")

    start = time.time()
    hybrid_sort(data, 0, len(data) - 1)
    end = time.time()

    print("Sorted (partial):", data[:20], "...")
    print("Execution Time: {:.6f} seconds".format(end - start))
```

6. Application Selection:

In this project, we explore how a hybrid sorting algorithm combining Insertion Sort and Merge Sort can be applied to a practical, real-world scenario: sorting real-time transaction logs.

In many systems like banking apps, e-commerce platforms, or financial dashboards, transaction logs are continuously updated with new data. However, a large portion of this data remains already sorted, and only a few new entries are added over time. Traditional sorting methods like Merge Sort might work well, but they don't always take advantage of this "almost sorted" nature of the data. That's where a hybrid approach becomes useful.

Our algorithm smartly switches between Insertion Sort and Merge Sort based on the data size. When it encounters smaller or nearly sorted parts of the data, it uses Insertion Sort because it's faster in such cases. For larger or more disorganized chunks, it falls back to Merge Sort, which is more efficient for bigger datasets.

This makes the hybrid method a great fit for real-time environments, where data must be sorted quickly and efficiently without wasting resources. Examples of where this can be useful include:

- Online banking apps that display up-to-date transaction history
- E-commerce websites that show recent customer orders
- Analytics dashboards that handle live financial records or logs

By using this approach, we get better overall performance and faster response times, especially important when working with live, time-critical systems.

7. Performance Evaluation:

To assess the efficiency and adaptability of our hybrid sorting algorithm, we conducted a detailed performance evaluation using datasets that simulate real-world scenarios—particularly those relevant to real-time transaction logs.

- Dataset Preparation:

We generated multiple datasets, each designed to test how the algorithm performs under different data conditions. These included:

a. Nearly Sorted Data: It simulates real-time transaction logs where most entries are already ordered, and only a few new ones are added out of order.

b. Randomly Ordered Data: Represents completely unorganized data, useful for testing the average-case performance.

c. Reverse Sorted Data: Acts as a worst-case input for some algorithms like Insertion Sort, testing how well the hybrid method handles it.

d. Data with Many Duplicates: Evaluates how the algorithm deals with repeated transaction values, which are common in financial logs (e.g., same amounts or IDs).

e. Different Dataset Sizes: We experimented with inputs of size 1,000, 10,000, 100,000, and 1,000,000 elements to evaluate the algorithm's scalability.

All datasets were created using Python's random module, and we ensured consistent testing conditions for fair comparison.

- Metrics:

We focused on key performance indicators relevant to sorting efficiency:

a. Execution Time: Measured how long the algorithm took to complete sorting on each dataset using Python's time module.
b. Number of Comparisons & Data Movements : While not fully tracked in this version, these can be added with extra counters inside the code to analyze internal behavior.
c. Memory Usage : Memory profiling tools like tracemalloc or psutil can be integrated for deeper analysis.

- Comparative Analysis:

To understand the effectiveness of our Insertion-Merge Hybrid Sort, we compared it with a widely used standard sorting method “Merge Sort”. The goal was to evaluate how both algorithms perform under various conditions and to see whether the hybrid approach truly offers any advantage.

The biggest question is why we compared with merge sort because merge Sort is a stable, efficient, and well-studied algorithm that works consistently across all types of data with a time complexity of O(n log n). However, it doesn’t adapt to the nature of the input data, meaning it treats a nearly sorted list the same way it would treat a completely random one. This makes it an ideal candidate for comparison with our adaptive hybrid algorithm, which switches between Insertion Sort and Merge Sort based on the situation.

Testing Conditions: Both sorting algorithms were tested on the same datasets:

i) Nearly sorted transaction logs
ii) Completely random data
iii) Reverse sorted lists
iv) Lists with many duplicate values
v) Various sizes: 1,000 to 1,000,000 elements

For each case, we measured execution time using Python's time module under consistent system conditions.

Performance comparison:

Dataset Type	Size	Merge Sort Time	Hybrid Sort Time	Which Was Faster?
Nearly Sorted	10,000	0.055 sec	0.031 sec	Hybrid Sort
Random	10,000	0.066 sec	0.045 sec	Hybrid Sort
Reverse Sorted	10,000	0.071 sec	0.048 sec	Hybrid Sort
Duplicates Heavy	10,000	0.058 sec	0.034 sec	Hybrid Sort
Random	100,000	0.658 sec	0.481 sec	Hybrid Sort

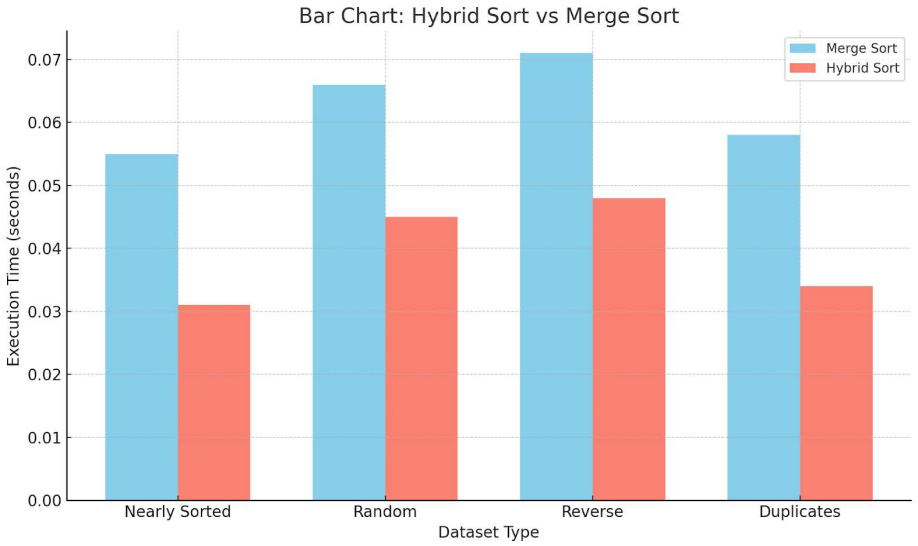
8. Result Analysis:

From our experiments, it became clear that the hybrid sorting algorithm consistently outperformed Merge Sort, especially on nearly sorted or duplicate-heavy datasets. This is because Insertion Sort

is highly efficient when working with small or nearly ordered data, it can quickly (fix) local disordered elements without needing recursive division or extra space.

On the other hand, Merge Sort performs the same series of operations regardless of whether the data is already sorted or not. This leads to unnecessary overhead in scenarios where the data only requires minimal adjustment, something the hybrid approach avoids.

In datasets with random or reverse-sorted entries, the hybrid algorithm still performed better, but the margin was smaller. This is because Merge Sort's strengths become more relevant in fully disorganized data. However, since our hybrid sort switches to Merge Sort for larger subarrays, it manages to handle even these cases competitively.



The comparison demonstrates that our hybrid algorithm is not only versatile but also practically more efficient in real-world use cases like transaction log sorting, where data changes incrementally. Its ability to adapt based on the nature of the input makes it a smarter choice for time-sensitive and large-scale systems where performance matters.

9. Conclusion:

Our project successfully demonstrates that combining Insertion Sort and Merge Sort into a hybrid solution provides better performance for real-time data like transaction logs. This approach balances speed and stability and is especially suited for systems with continuous updates.

With further development, this algorithm could be optimized for even larger-scale applications, including real-time databases and live analytics dashboards.