LSA: A Novel State-Of-The-Art Sorting Algorithm for Efficient Arrangement of Large Data

Ariful Islam Shiplu*
Dhaka University of Engineering & Technology, Gazipur
Bangladesh
shipluarifulislam@gmail.com

Md. Mostafizer Rahman*†‡
Dhaka University of Engineering & Technology, Gazipur
Gazipur, Bangladesh
mostafiz26@gmail.com

Yutaka Watanobe
The University of Aizu
Aizuwakamatsu, Japan
yutaka@u-aizu.ac.jp

ABSTRACT

Over the years, data generation from various sources (social media, business, medical, education, programming, images, videos, etc.) has increased exponentially due to technological development, application, and daily usage. Organizing these large amounts of data efficiently is not a trivial task. Therefore, an efficient sorting algorithm can be helpful in processing and arranging these large data. To address this issue, we propose a novel sorting algorithm called the Layered Sorting Algorithm (LSA) that organizes data using a layering approach. The LSA aims to improve sorting efficiency by exploiting the inherent structure and characteristics of the data. The algorithm divides the input data into layers, where the data of each layer has the same length (or digits). Single-length data is on layer 1, double-length data is on layer 2, and so on. Within each layer, a specific sorting technique is applied to efficiently arrange the elements. In this paper, we conducted experiments with more than one million random integer data, which have a maximum length of 10. The experimental results show that LSA obtained better results in terms of time complexity and comparisons compared to existing state-of-the-art sorting algorithms. The results demonstrate that LSA achieves the best case and average case when sorting data. Moreover, LSA reduces a significant number of comparisons than the original algorithms when sorting data as well as reduces time complexity. Furthermore, the proposed LSA can be used with existing sorting algorithms to achieve better performance for sorting large-scale data.

CCS CONCEPTS

• **General and reference** → **Layered sorting algorithm**; *Big data*; Layered concept; Sorting Algorithm; 2D Array.

ACM Reference Format:
Ariful Islam Shiplu, Md. Mostafizer Rahman, and Yutaka Watanobe. 2023. LSA: A Novel State-Of-The-Art Sorting Algorithm for Efficient Arrangement of Large Data. In *2023 4th Asia Service Sciences and Software Engineering Conference (ASSE 2023), October 27–29, 2023, Aizu-Wakamatsu City, Japan.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3634814.3634829

*Both authors contributed equally to this research.
†Corresponding author
‡Also with The University of Aizu, Japan.

ASSE 2023, October 27–29, 2023, Aizu-Wakamatsu City, Japan

ACM ISBN 979-8-4007-0853-4/23/10…$15.00
https://doi.org/10.1145/3634814.3634829

1 INTRODUCTION

Sorting is the process of rearranging a list of data into the proper order because managing things in a predetermined sequence is more effective than managing randomly generated elements[1]. The initial input is a set of N randomly ordered items, and the desired result is a sorted list in which the elements are arranged in ascending or descending order [7]. The idea of sorting has been a fundamental problem in computer science and mathematics for centuries. "The Art of Computer Programming: Sorting and Searching" by Donald E. Knuth is a highly regarded book that delves into the realm of sorting and searching algorithms. This extensive work offers a thorough exploration of a wide range of sorting algorithms, including their analysis and practical implementations. With its comprehensive coverage, the book serves as a valuable resource for gaining deep insights into the field of sorting algorithms and their associated techniques [19].

Over the last few decades, sorting algorithms have earned significant importance and have been widely applied across various domains. One prominent application is in e-commerce platforms and online marketplaces, where sorting algorithms are utilized to arrange and display products based on criteria like relevance, popularity, price, or customer ratings. These algorithms are also critical components of recommendation systems, as they classify items according to user preferences, behavior, or similarity, enabling personalized recommendations that enhance customer satisfaction and engagement. Sorting algorithms play a crucial role in social media platforms, where they are employed to arrange posts, comments, or search results based on relevance and timeliness. This sorting functionality allows users to experience an optimized and personalized user experience, as content is organized according to factors such as consistency, relevance, or interaction with other users. Programming platforms use sorting algorithms to rank their problems and programmers according to different criteria [24, 27]. Logistics and supply chain management heavily rely on sorting algorithms to optimize the movement and scheduling of goods. These algorithms assist in efficiently organizing shipments based on delivery routes, time periods, or priority levels. Moreover, in data visualization, sorting algorithms are employed to arrange data points or elements, enabling the creation of meaningful visualizations that depict patterns, trends, or comparisons in complex data sets. Financial systems utilize sorting algorithms for various tasks, including portfolio management, risk assessment, and fraud detection. By leveraging sorting algorithms, finance data can be

effectively analyzed and organized, enabling informed decision-making and the identification of irregularities. In database systems, sorting algorithms play a fundamental role in efficiently organizing and retrieving data. They are utilized for indexing, query optimization, and sorting large result sets. Overall, sorting algorithms have found widespread applications across domains, providing efficiency, organization, and enhanced user experiences in various fields such as e-commerce, recommendation systems, social media, logistics, data visualization, finance, and database management systems [12].

Algorithms such as PageRank use sorting techniques to determine search results according to their importance, and provide user information that is most relevant [5]. For the purpose of organizing and preparing massive datasets, sorting algorithms are instrumental to data analytics and mining applications. For example, they are often used as an intermediate step for different algorithms like clustering or classification and association rule mining [16, 25]. Applications for sorting algorithms include genome sequencing, analysis of DNA sequences and protein structure prediction. It is possible to make accurate analyses and comparisons of the genetic sequences by using filtering methods, which allows you to organize and extract biology data [21]. For tasks such as image filtering, noise reduction, and feature extraction, sorting algorithms are used in image and signal processing applications. For further processing, sorting algorithms help to group pixels or signal samples on the basis of their characteristics [14]. Furthermore, sorting algorithms have been utilized in computational geometry for problems such as convex hull construction, line segment intersection, and geometric searching. Research by de Berg and collaborators showcases the application of sorting algorithms in computational geometry problems and their impact on geometric algorithms' efficiency [9–11]. Sorting algorithms are useful for large language models to present their results [23].

Efficiency is a key aspect of sorting algorithm development. Researchers work towards minimizing the computational resources required by sorting algorithms. This includes reducing memory usage, minimizing the number of comparisons or swaps performed, and optimizing the overall computational steps involved in the sorting process. The goal is to develop algorithms that are both time and memory-efficient, enabling faster and more efficient data processing. It is important to take into account several factors when comparing different sorting algorithms, for example, the time complexity of an algorithm determines how long that algorithm can run [6, 15, 26]. The optimized bubble sort (BS) algorithm has achieved a significant improvement by reducing the number of iterations by half compared to the traditional BS. The main idea behind the optimized BS is (*i*) to compare the top element of an array with its successor, swapping the two if they are not in the proper order, and (*ii*) to compare the bottom element of an array with its predecessor, swapping the two if they are not in the proper order [22]. The Magnetic BS is an improvement to the BS algorithm that performs much better when there are duplicates in the list. The number of distinct values in the list to be sorted determines the Magnetic BS's run time [3]. P. McIlroy's merge sort has guaranteed $O(nlogn)$ worst-case performance and is almost optimally adaptive to data with residual order but requires $O(n)$ additional memory [20]. Katajainen et al. present an optimized version of Merge Sort (MS) called "Practical In-Place Mergesort" which does not require any more memory to merge two separate sorting subarrays [18]. An ancient but little-known modification of Heapsort due to Floyd uses $O(nlogn) + O(n)$ comparisons, but needs almost that many swaps, four times (4×) as many as Quicksort [4]. Floyd introduced a more efficient variant of Heap Sort known as "Treesort." Treesort replaces the original Heap Sort algorithm's exact heap data structure with a linear tree structure, effectively reducing the overhead associated with the original algorithm [13].

Merge Sort, despite its efficiency, has drawbacks such as the need for additional space during merging and its non-in-place nature. Merge Sort requires additional memory to hold temporary arrays during the merging process. The heart of the construction is the merge routine, which combines two sorted sequences into one [18]. Bubble Sort's main limitations lie in its inefficiency for large datasets and its ineffectiveness for partially sorted arrays [8]. Insertion Sort is not ideal for sorting extensive datasets or situations where efficiency is crucial. Its quadratic time complexity of $O(n^2)$ makes it impractical for sorting large arrays or datasets with a substantial number of elements. Heap Sort has certain limitations, including the need for extra space to accommodate the heap data structure and its lack of stability. Selection Sort suffers from inefficiency for large datasets and partially sorted arrays [8]. The limitations of Quick Sort (QS) involve its time complexity in specific scenarios and its reliance on careful pivot selection. In certain cases, QS may exhibit a worst-case time complexity, and its performance is affected by the choice of pivot [17]. Radix Sort (RS) requires additional memory for auxiliary arrays and its space complexity is influenced by the number of digits or bits in the input data. This means that sorting larger numbers or data with a greater number of bits can result in higher memory usage during the sorting process [2]. However, these algorithms may not fully exploit the inherent structure and characteristics of the data being sorted, leading to suboptimal performance in certain scenarios.

To address this gap, we propose a novel sorting algorithm called a Layered Sorting Algorithm (LSA) that takes an ideal approach to data sorting. The LSA introduces the concept of layering the data based on their length[1] count. Each data is placed into a corresponding layer according to its digit count. For example, data 22, 78, and 69 have the 2 digits or length 2 and will be placed into Layer-2. The motivation behind developing the LSA lies in the observation that numbers with fewer digits tend to have a smaller range of values, making them quicker to sort. The algorithm is intended to reduce the number of comparisons and swaps needed in a sorting process, resulting in improved efficiency and effectiveness by exploiting this characteristic. This study will provide an in-depth exploration of the LSA, including its design, implementation, and evaluation in terms of time to execute, number of comparisons, and swaps on different data sets. Moreover, we compared LSA's performance with traditional sorting algorithms. Unnecessary comparisons and swaps in each layer can be reduced with this layer-based splitting strategy, which will improve overall performance. Furthermore, we discuss the implications and possible use of an LSA in particular with regard to situations where there are variations in data on digits. The key contributions of this paper are as follows:

[1]The terms **length** and **digit** are used interchangeably and have the same meaning.

- Development of LSA: This paper presents the design and implementation of LSA, which uses the layering approach to represent data. Then, sorting techniques are applied to each layer to sort data. LSA optimizes the sorting process and reduces unnecessary comparisons and interchanges.
- Exploring Performance and Efficiency: The paper performs a comprehensive evaluation of LSA and compares its performance with traditional sorting algorithms, such as quicksort, merge sort, and heap sort. Evaluation metrics include execution time, number of comparisons, and number of swaps. The results provide deep insights into the efficiency and effectiveness of LSA in sorting data.
- Application of LSA: The research paper highlights potential applications of LSA in scenarios where data has variations in the number of digits/lengths. The scope of application includes numerical data analysis, financial data processing, and database management where efficient data sorting is critical.
- Research Opportunities: The introduction of LSA opens new avenues for the development of sorting algorithms by highlighting the importance of considering additional data properties, such as the number of digits. This work contributes to the broader field of sorting algorithms by encouraging researchers to explore alternative sorting strategies that exploit specific properties of the data being sorted.

The results and contributions of this work advance the field of sorting algorithms by presenting a novel approach to data organization and sorting. The LSA offers potential improvements in efficiency, adaptability, and performance, and represents a promising alternative to traditional sorting algorithms. The remainder of the paper is structured as follows: Section 2 presents the proposed LSA and its pseudocode. Section 3 explores the experimental results. Section 4 discusses the results and the limitations of the LSA. Finally, Section 5 concludes with future remarks.

2 PROPOSED LAYERED SORTING ALGORITHM

The goal of the proposed LSA is to sort the large number of data in an efficient way in terms of comparisons and execution time. To achieve this goal, a layering approach is introduced in LSA, which can help to reduce the number of comparisons as well as the execution time for sorting the data. Figure 1 shows the workflow of the proposed approach.

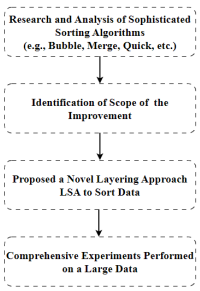


Figure 1: The workflow of the proposed approach

Algorithm 1 shows the pseudocode of our proposed LSA. LSA takes a number of N random data as input and produces a number of N sorted data as output. To store the data, a two-dimensional array (Array2D) is created. Next, LSA calculates the length ($\mathcal{L}$) of the data and stores the data in the $Array2D$ based on the length of the data. For example, data 5785, 3697, and 4750 have a length of 4 and will be placed in row 4 of the $Array2D$. A row of the $Array2D$ is called a layer. The $Push\_back()$ function is used to insert the data into the corresponding row of $Array2D$. This process is repeated for all input data. Next, the data from each layer is read and sent to the $Sorting\_alogorithm()$ function to sort the data in each layer. It is noted that the $Sorting\_algorithm()$ function can leverage any state-of-the-art sorting algorithm to sort data. Finally, the sorted data stored in the $Array2D$ is printed.

Algorithm 1 Pseudocode of layered sorting algorithm (LSA)

1: **Define:** Data size N, and maximum length of a data $\mathcal{L} = 10$
2: **Input:** Random N data
3: **Output:** Sorted N data
4: **Create** a 2D vector of integers $Array2D$
5: **for** $i = 0; i < N$ **do**
6: $X = input[i]$ //input data taken randomly
7: $P = length\_of\_string(X)$ // calculate the length of the input
8: **if** $P > \mathcal{L}$ **then**
9: $Break$
10: **else**
11: $Push\_back(X)$ // push back X to $Array2D$ in $P - 1$ layer/row
12: **end if**
13: **end for**
14: **for** $i = 0; i < \mathcal{L}$ **do**
15: $Sorting\_algorithm(Array2D[i])$ // apply sorting algorithm to each row of the $Array2D$
16: **end for**
17: **Print** sorted data with $Print(Array2D)$

The proposed LSA has a wide range of scalability and adaptability for sorting data. Most state-of-the-art sorting algorithms can be used with the LSA approach to improve their sorting performance in various metrics. According to the LSA pseudocode, all data from each layer is passed through the $Sorting\_algorithm()$ function to sort the data. Depending on the type of data, any appropriate sorting algorithm (e.g., merge, selection, heap sort, etc.) can be used in the $Sorting\_algorithm()$ function to sort the data. Adapting LSA with various state-of-the-art algorithms for sorting data is an uncomplicated task. Thus the scalability of the LSA is high.

Figure 2 visualizes the data layering approach using the proposed LSA algorithm. Basically, the layers are created based on the length of the data in $Array2D$. For example, a dataset contains data {19, 1594, 159, 7, 45676, 12, 5, 139, 5713, 34654, 2134, 87546, 21, 875, 9, 58, 978, 1}. Next, we calculate the length of the data for layering. In this case, data 7, 5, 9, and 1 have a single length and are placed on layer 1. Similarly, data 19, 12, 21, and 58 have a length of 2 and are placed on layer 2, and so on. So for the example data set, there

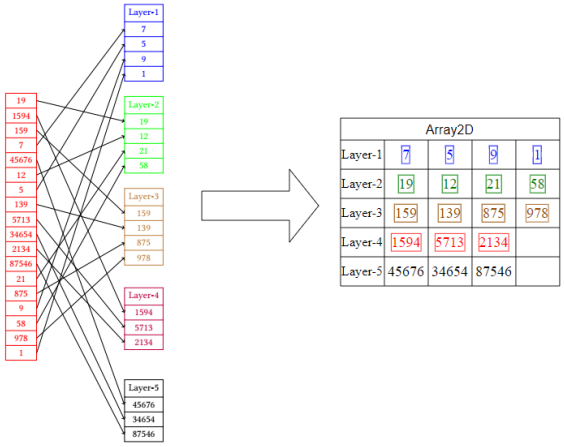


Figure 2: Data layering approach of LSA.

are five layers in total. Once the layering of the data is complete, sorting algorithms are applied to each layer, which significantly reduces the number of comparisons and the execution time for sorting the data. This layering approach is particularly useful for sorting algorithms that perform linear comparisons.

To demonstrate the efficacy of the proposed LSA, we applied LSA to sort this data `{19, 159, 7, 45676, 12, 5, 139, 5713, 34654, 2134}`, although, the data size is small. It can be seen that LSA requires only 5 comparisons to sort the data, while other state-of-the-art algorithms (e.g., BS, QS, selection sort (SS), MS, heap sort (HS), and insertion sort (IS)) require more comparisons, as shown in Table 1. LSA reduced the comparisons with a larger margin than the state-of-the-art algorithms. We expect the performance of the proposed LSA to be more significant on large datasets.

Table 1: Calculated the number of comparisons without LSA and with LSA

Sl.	Approach	BS	IS	SS	QS	HS	MS
1	Without LSA	45	17	45	23	18	24
2	With LSA	5	5	5	5	5	5

3 EXPERIMENTAL RESULTS

We conducted experiments with LSA and other algorithms based on a large number of randomly generated data. All experiments are performed on a Windows 11-based operating system with processor: AMD Ryzen 5 5600G UP4.4GZ 6 Core 12 Thread and RAM: 8GB DDR4. Initially, we conducted experiments with the fundamental sorting algorithms (e.g., bubble sort, selection sort, insertion sort, merge sort, etc.). Table 2 shows the data comparisons of various sorting algorithms on different sets of data. Similarly, Table 3 shows the execution time of different sorting algorithms to sort the given data. The obtained results show that merge sort, heap sort outperforms other sorting algorithms in terms of both comparisons and execution time.

Table 4 shows the number of comparisons to sort the data when sorting algorithms are leveraged with LSA. It can be seen that the comparisons are reduced significantly compared to the solely/independently used algorithms (e.g., BS, IS, SS, etc.), as shown in Table 2.

Moreover, Figure 3 shows the one-to-one comparison between the state-of-the-art algorithms and LSA for sorting one (01) million random data. It can be seen that LSA with the X sorting algorithm reduces the number of comparisons than the X sorting algorithm alone in all cases. Where X can be any state-of-the-sorting algorithm. In addition, the Bubble, Insertion, and Selection Sort algorithms and their LSA (Figures 3a, 3b, and 3e) required higher comparisons to sort the data than Heap, Merge, and Quick Sort. Moreover, Figure 4 shows the relative performances of two groups of algorithms, namely algorithms with *High* comparisons and algorithms with *Low* comparisons. Interestingly, Figure 4a shows that the Bubble and Selection Sort algorithms require almost identical comparisons to sort data. The overall experimental results show that LSA with Quick Sort needs the least number of comparisons to sort one million random data, as shown in Figure 4b.

Similarly, we calculated the execution time with LSA to sort data, as shown in Table 5. It can be seen that sorting algorithms with LSA required less execution time to sort data compared to other algorithms (e.g., bubble sort, insertion sort, and selection sort), as shown in Table 5. Moreover, we evaluated the comparison results of how the proposed LSA performed in sorting data. We found that the proposed LSA required approximately 5× fewer comparisons than the original bubble sort, insertion sort, and selection sort. Also, LSA required approximately 1.2× fewer comparisons than the merge and heap sort. Similarly, it requires approximately 2× fewer comparisons than the quick sort. Thus, we can express these performances mathematically as follows: $\delta_{bubble} \approx \Delta_{bubble}/5$, $\delta_{insertion} \approx \Delta_{insertion}/5$, $\delta_{selection} \approx \Delta_{selection}/5$, $\delta_{merge} \approx \Delta_{merge}/1.2$, $\delta_{heap} \approx \Delta_{heap}/1.2$, and $\delta_{quick} \approx \Delta_{quick}/2$, where δ_{xx} denotes the number of comparisons to sort data with LSA and Δ_{xx} denotes the number of comparisons to sort data without LSA. Based on the experimental results, we can say that the performance of the sorting algorithms is improved significantly when they are applied with LSA.

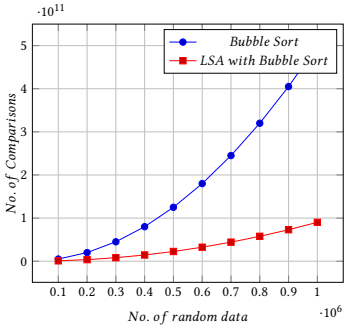
Figure 5 shows that state-of-the-art sorting algorithms occupied less memory space rather than the proposed LSA to sort data. This happened due to the layered approach of LSA. However, the proposed LSA obtained better results in terms of comparisons and execution time to sort large numbers of data.

4 DISCUSSION

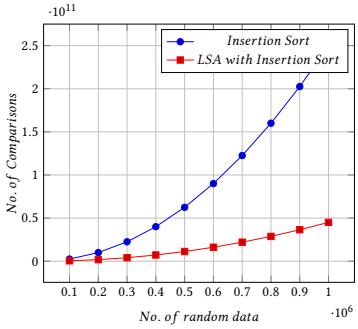
The proposed LSA introduced a novel layering approach for large data sorting. Experimental results show that LSA achieves better results in terms of comparisons and execution time compared to state-of-the-art data sorting algorithms. LSA achieved better results, but we observed the suboptimal performance of LSA in terms of memory space occupation. As can be seen in Figure 5, LSA requires more memory for sorting data than the state-of-the-art algorithms. This is due to the layering approach, which costs additional memory. For example, if we have 100,000 data divided into 7 layers, this may require additional time and memory for layering. Next, sorting

Table 2: Calculated the number of comparisons to sort the data using fundamental algorithms

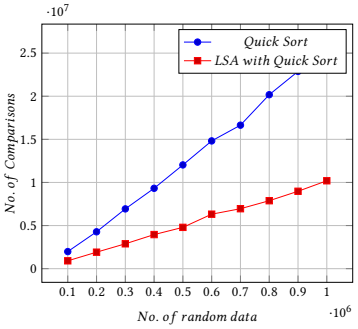
Sl.	# Sample	BS	IS	SS	QS	HS	MS
1	10	45	11	45	29	24	23
2	100	4950	2342	4950	661	694	552
3	1000	499500	249049	499500	11551	11718	8733
4	10000	49995000	24988001	49995000	156602	167028	121067
5	100000	4999950000	2501459800	4999950000	1981879	2167738	1545367
6	1000000	499999500000	250032617396	499999500000	25869502	26669586	18717415



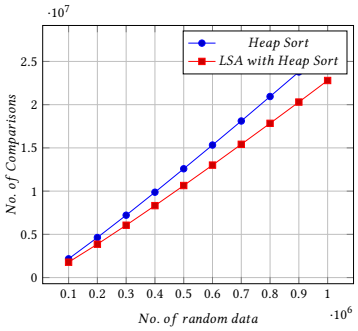
(a) Bubble Sort Vs LSA with Bubble Sort



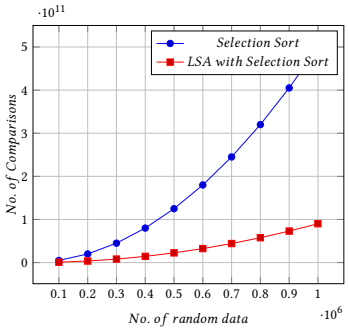
(b) Insertion Sort Vs LSA with Insertion Sort



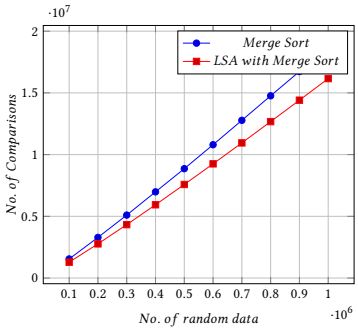
(c) Quick Sort Vs LSA with Quick Sort



(d) Heap Sort Vs LSA with Heap Sort



(e) Selection Sort Vs LSA with Selection Sort



(f) Merge Sort Vs LSA with Merge Sort

Figure 3: Comparative comparisons between state-of-the-art sorting algorithms and the proposed LSA for sorting data

Table 3: Calculated the execution time in milliseconds (ms) to sort the data using fundamental algorithms

Sl.	# Sample	BS	IS	SS	QS	HS	MS
1	10000	109	31	48	0	0	0
2	100000	11770	3313	4658	15	16	15
3	1000000	-	453947	-	110	219	203

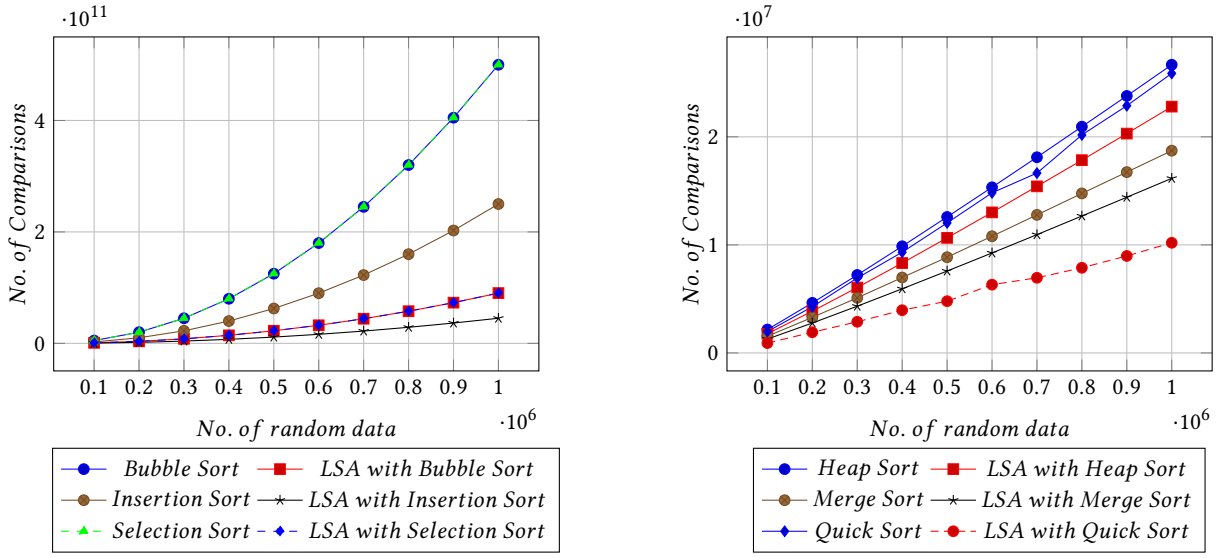
algorithms are applied to each layer to sort the data. In particular, LSA with merge sorting requires more memory because merge sort creates sublists in each layer that require more memory. This can be considered as one of the limitations of the proposed LSA that needs to be addressed.

The bubble, insertion, quick, merge, heap, and selection sort algorithms used these equations $n * (n-1)/2$, $(n^2 - n)/4$, $1.39 * n * log(n)$, $n * log(n) - (n-1)$, $n * logn + time\ to\ heap$, and $n * (n-1)/2$, respectively, to calculate the comparisons for sorting the data. Figure 3 shows the one-to-one comparison between LSA and Sophisticated sorting algorithms. It can be seen that LSA has significantly reduced the number of comparisons for the Bubble, Insertion, Selection, and Quick Sort algorithms compared to Heap and Merge Sort to sort one million random data. Figure 4b shows that LSA with quick sort required the lowest number of comparisons to sort data than others. However, the efficiency of the proposed LSA outperforms the Sophisticated sorting algorithms in terms of comparisons.

In addition, LSA with heap and quick sort have about the same time complexity as their original algorithms but require fewer comparisons to sort data. LSA requires more memory than bubble and insertion sort but significantly improves the time complexity to $O(n^2/2)$, which is half of the original algorithms. LSA is particularly

Table 4: Calculated the number of comparisons to sort the data using LSA

Sl.	# Sample	BS	IS	SS	QS	HS	MS
1	10	5	4	5	6	4	5
2	100	886	457	915	340	340	307
3	1000	89742	45565	90383	4405	7847	6191
4	10000	9017072	4503773	9030075	66693	128611	95043
5	100000	902379555	451732626	902447923	911575	1779767	1283884
6	1000000	90177914885	45014642223	90178703260	10194389	22795694	16158644



(a) Algorithms with high comparisons

(b) Algorithms with low comparisons

Figure 4: Relative comparisons between two groups such as High comparisons and Low comparisons of sorting algorithms

Table 5: Calculated the execution time in milliseconds (ms) to sort the data using LSA

Sl.	# Sample	BS	IS	SS	QS	HS	MS
1	10000	62	15	31	0	0	0
2	100000	6704	1860	2470	15	31	31
3	1000000	-	184127	246335	187	375	422

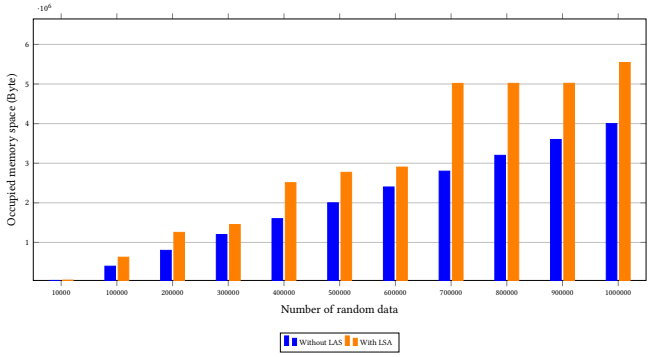


Figure 5: Comparison of occupied memory space with LSA and without LSA

useful in dealing with a large number of data, which reduces comparisons and time. On the contrary, if, by chance, all random data happens to have the same length, the effectiveness of LSA would be diminished as all the data would be confined to a single layer. Nonetheless, encountering such a scenario is infrequent, especially when dealing with a substantial volume of data. To the best of our knowledge, LSA can be very effective in sorting big data with its novel layering approach.

5 CONCLUSION

In this paper, we have proposed a new state-of-the-art sorting algorithm called LSA. LSA uses a layering approach to organize the data into layers based on the length of the data. Next, the sorting algorithm is applied to each layer to sort the data. The pseudocode of LSA is also presented. We have conducted extensive experiments on one million random data. The experimental results show that LSA requires fewer comparisons and execution time compared

to state-of-the-art algorithms (Bubble, Selection, Insertion, Merge, Heap, and Quick Sort) to sort a large number of data. The results also show that LSA with the Quick Sort algorithm requires the lowest data comparisons to sort one million data compared to other algorithms. These results demonstrate the effectiveness of the proposed LSA for sorting data. However, LSA achieved suboptimal results in occupying memory space while sorting data. In the future, further improvement of LSA will be considered to reduce excessive memory occupation for sorting data.

REFERENCES

[1] Pooja Adhikari and A Pooja. 2007. Review On Sorting Algorithms-A comparative study on two sorting algorithms. *Mississippi State, Mississippi* 4 (2007).

[2] Alfred V Aho and John E Hopcroft. 1974. *The design and analysis of computer algorithms*. Pearson Education India.

[3] Obed Appiah and Ezekiel Mensah Martey. 2015. Magnetic bubble sort algorithm. *International Journal of Computer Applications* 122, 21 (2015).

[4] Jon L Bentley and M Douglas McIlroy. 1993. Engineering a sort function. *Software: Practice and Experience* 23, 11 (1993), 1249–1265.

[5] Sergey Brin. 1998. The PageRank citation ranking: bringing order to the web. *Proceedings of ASIS, 1998* 98 (1998), 161–172.

[6] Curtis R Cook and Do Jin Kim. 1980. Best sorting algorithm for nearly sorted lists. *Commun. ACM* 23, 11 (1980), 620–624.

[7] T Cormen, C Leiserson, R Rivest, and Clifford Stein. 2009. Book: introduction to algorithms.

[8] Thomas H Cormen, Ch E Leiserson, RL Rivest, and C Stein. 2009. Introduction to algorithms. Massachusetts. *London: MIT Press. Section* 25 (2009), 636–40.

[9] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. 2008. Delaunay triangulations: height interpolation. *Computational geometry: algorithms and applications* (2008), 191–218.

[10] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. 2008. Line Segment Intersection: Thematic Map Overlay. *Computational Geometry: Algorithms and Applications* (2008), 19–43.

[11] Mark de Berg, Marc van Kreveld, Mark Overmars, Otfried Schwarzkopf, Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwarzkopf. 1997. Orthogonal Range Searching: Querying a Database. *Computational Geometry: Algorithms and Applications* (1997), 93–117.

[12] Vladmir Estivill-Castro and Derick Wood. 1992. A survey of adaptive sorting algorithms. *ACM Computing Surveys (CSUR)* 24, 4 (1992), 441–476.

[13] Robert W Floyd. 1964. Algorithm 245: treesort. *Commun. ACM* 7, 12 (1964), 701.

[14] RC Gonzalez and RE Woods. 2017. Digital Image Processing (th Edition).

[15] Michael T Goodrich, Roberto Tamassia, and Michael H Goldwasser. 2014. *Data structures and algorithms in Java*. John wiley & sons.

[16] J Han, M Kamber, and J Pei. 2011. Cluster Analysis: Basic Concepts and Methods in Data Mining Concepts and Techniques.

[17] Charles AR Hoare. 1962. Quicksort. *The computer journal* 5, 1 (1962), 10–16.

[18] Jyrki Katajainen, Tomi Pasanen, and Jukka Teuhola. 1996. Practical in-place mergesort. *Nord. J. Comput.* 3, 1 (1996), 27–40.

[19] Donald E Knuth. 1973. Sorting and searching.(The art of computer programming, vol. 3) Addison-Wesley. *Reading, Mass* (1973).

[20] Peter McIlroy. 1993. Optimistic sorting and information theoretic complexity. In *Proceedings of the fourth annual ACM-SIAM Symposium on Discrete algorithms*. 467–474.

[21] DW Mount and Sequence Bioinformatics. 2001. Genome analysis. *Bioinformatics: Sequence and Genome Analysis. Cold Spring Harbor Laboratory Press, Cold Spring Harbor, NY* (2001), 479–532.

[22] Ramesh M Patelia, Shilpan D Vyas, and Parina S Vyas. 2015. An Analysis and Design of Optimized Bubble Sort Algorithm. *IJRIT International Journal of Research in Information Technology* 3, 1 (2015), 65–68.

[23] Md. Mostafizer Rahman and Yutaka Watanobe. 2023. ChatGPT for education and research: Opportunities, threats, and strategies. *Applied Sciences* 13, 9 (2023), 5783.

[24] Md Mostafizer Rahman, Yutaka Watanobe, Rage Uday Kiran, Truong Cong Thang, and Incheon Paik. 2021. Impact of practical skills on academic performance: A data-driven analysis. *IEEE Access* 9 (2021), 139975–139993.

[25] Md Mostafizer Rahman, Yutaka Watanobe, Taku Matsumoto, Rage Uday Kiran, and Keita Nakamura. 2022. Educational data mining to support programming learning using problem-solving data. *IEEE Access* 10 (2022), 26186–26202.

[26] Michael Sipser. 1996. Introduction to the Theory of Computation. *ACM Sigact News* 27, 1 (1996), 27–29.

[27] Yutaka Watanobe, Md Mostafizer Rahman, Taku Matsumoto, Uday Kiran Rage, and Penugonda Ravikumar. 2022. Online judge system: Requirements, architecture, and experiences. *International Journal of Software Engineering and Knowledge Engineering* 32, 06 (2022), 917–946.

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009