AJBAR Vol 1(1), 2022: 55-64, ISSN: 2811-2881

Arid Zone Journal of Basic and Applied Research

Faculty of Science, Borno State University
Maiduguri, Nigeria

Journal homepage: https://www.azjournalbar.com

Research Article

Performance Comparison of three Sorting Algorithms Using Shared Data and Concurrency Mechanisms in Java

[1*]Abbas M. Rabiu, [2]Etemi J. Garba, [3]Benson Y. Baha, [2]Yusuf M. Malgwi and [1]Mu'azu Dauda
[1] Department of Computer Science, Federal University Dutse, Nigeria
[2] Department of Computer Science, Modibbo Adama University of Technology Yola, Nigeria
[3]Department of Information Technology, Modibbo Adama University of Technology, Yola, Nigeria
*Corresponding Author: mambas86@fud.edu.ng

ARTICLE INFO:

Keyword:
Algorithms,
Atomicity,
Running time,
Performance,
Sorting

ABSTRACT

Sorting large data sets or database is a problem commonly found in Computer Science and to find a solution to this problem, several quick-sorting algorithms were developed while some need to be improved upon to make them more efficient. Sorting algorithms can be developed using both shared and non-shared data based on programmers' choices. This paper aimed to develop three distinct sorting algorithms that involved shared data using three concurrency mechanisms in Java to measure their running times and to compare their performances. The main method used is the practical measurement of running times using benchmarking carried out on a machine with eight (8) processing cores. System current Time Millis () was used to measure time during the experiment. The results show that SD-Parallel quick sort implementation using Atomicity outperforms the other two algorithms. It was also shown that creating more threads leads to overhead when algorithms are developed using shared data. It was concluded that the running times of algorithms that involved shared data increase with the increase in the array size when proper synchronization is performed It was further revealed that SD-Parallel quick sort implemented using Acyclic-Barrier emerged as the second-best algorithm. Other concurrency mechanisms namely: Phaser, and Double-Atomicity provided by other Java JDK versions have the capacity of building more efficient framework that can be used to improve the performance of these algorithms due to their dynamic functionality. These and other concurrency mechanism provided by Java deserves further investigation as they are also capable of building more efficient algorithm

Corresponding author: Abbas M. Rabi'u, Email: mambas86@fud.edu.ng
Department of Computer Science, Federal University Dutse

INTRODUCTION

Sorting a large data set or database is a problem commonly found in Computer Science and to find a solution to this problem, several quick-sorting algorithms were developed while some need to be improved upon to lower their running times and to increase their speeds to make them more efficient (Rabiu *et al.,* 2018; Sengupta *et al.,* 2007). Performance comparison between the "Grouping Comparison Sort (GCS)" and some conventional algorithms namely: Quicksort, Insertion sort, Selection sort, Merge sort and Bubble sort were carried out by considering their execution time (Khalid *et al.,* 2018). A similar experiment was conducted where five different versions of sorting algorithms namely; Selection, Bubble, Quick, Merge, and Insertion sorts were compared (Naeem *et al.,* 2016). Furthermore, five different sorting algorithms namely: selecting sort, bubble sort merge sort, insertion sort, and quick sort were compared by summarizing their time and space complexities (Yash & Anuj, 2020). Another study was carried out by taking advantage of the complexity and performance of the algorithm into consideration. Additionally, a new complexity analysis approach was proposed for the "DHS algorithm" by considering the relation between the input size and the identified domain of the inputs (Hazem, 2019). An experimental analysis was carried out to measure and compare the performance of sorting algorithms that belong to the On(log n) class by measuring their execution times using a parallel approach (Mubashir, 2020). Algorithm visualization was studied "using high-level dynamic visualization of software that uses user interface techniques to portray and monitor the computational steps of algorithms". The experimental approach was the main method used (Jamil *et al.,* 2020). A comprehensive review of some selected machine algorithms was carried out by Yahaya *et al.* (2020), these algorithms are used to predict cardiovascular disease and their performances were compared. Comparison of bubble sorts and selection sort algorithms was carried out using benchmarking methods (Muhammad *et al.,* 2017). Another comparative study on sorting algorithms using an experimental approach was carried out by (Suleiman, 2013). A recent study compares Insertion sort, merge sorts and quick sorts were compared by considering their time complexities. Data of different sizes were used to implement them in Java (Dinesh, 2021).

Nowadays applications in all segments of computing, including some embedded systems are getting more complex, because of the increased range of functionality they offer. This complexity requires platforms with increased performance that satisfies such growing computational demands. This need has driven the adoption of multi-core processors in embedded systems since they allow performance to be increased at reasonable energy consumption (Hazem, 2017). "The majority of applications used today were written to run on only a single processor, failing to use the capability of multi-core processors (Rabiu *et al.,* 2021)". Although software development firms are capable of developing software that utilizes the multi-core processing machine to the fullest capacity, the problem being faced by these firms is how to upgrade legacy software programs that have been developed many years ago to multi-core software programs (Dempsey, 2014). According to Göetz *et al.* (2006) overheads can also result in degradation of the performance of algorithms and concurrent software in general. The literature reviewed so far focused on developing sorting algorithms using non-shared data without the need for synchronization. However, in this paper, three different sorting algorithms were developed by implementing three distinct frameworks using shared data to measure their running times and to compare their

performances on an octa-core machine. Table 1 and Figure 1 present a description of some popular and known sorting algorithms based on their time complexity, size of the input data, and the memory required.

Table 1: Description of some popular sorting algorithms based on their complexities

S/No.	Name	Concurrent Mechanism	Complexity
1.	SD-quick-sort	Sequential framework	This is a sequential algorithm that sorts an item sequentially one after another and in sequence. Its time complexity grows in the order of n squared. (On^2). More information about the complexity of sequential quick sorts can be found in Ahmed and Zirra (2013); Dinesh (2021).
2.	SD-Parallel quick sort	Atomicity	This is a parallel quick sorting algorithm implemented using concurrency mechanisms called Atomicity. The complexity of Parallel Quick sorts growth in the order of n times log of n (i.e. $O(nlogn)$, (Rabiu *et. al.,* 2018: Ahmed & Zirra, 2013)
3.	SD-Parallel quicksort	Acyclic-Barrier	This is another parallel implementation of quicksort based on Acyclic-Barrier. Similarly, its complexity grows in the order of n times log of n (i.e. $O(nlogn)$, (Ginanjar, 2021; Rabiu *et. al.,* 2021).
4.	QuickSortParalNaive	Naïve	This is another parallel implementation of quicksort based on the Naïve framework. When this faction is called, a new thread is created for each recursive call. It has the complexity of $O(nlogn)$, (Rabiu *et al.,* 2020; Dinesh, 2021).
5.	QuickSortForkJoin	Fork-join	This works based on the divide and conquer approach to problem-solving. Like other popular algorithms, its time complexity also grows in the $O(nlogn)$, (Rabiu *et al.,* 2020**;** Ginanjar, 2021; Niraj and Rajesh, 2019).

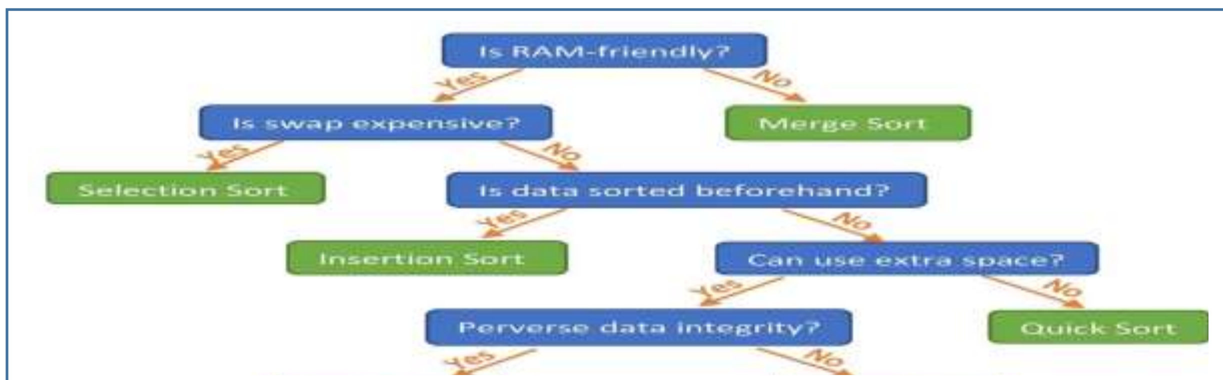

Figure 1: Description of some popular sorting algorithms (Anonymous, 2021)

METHODOLOGY

Array Data Structure

The data structure specified for developing the sorting algorithm in this paper is the array data structure. The data structure contains positive integer data types, ranging from 1 to 10 million elements that were used to carry out sorting in this paper. Several test runs were used to enable us to test different array sizes and to also reduce the effects of background programs on the measured time.

Loop Creation

Having defined our data structure, the next step after specifying the data size of the array is creating a loop that takes two parameters namely: The "seed" for generating random values, and the required "array size". After each iteration, the size of the "seed" remains the same but the array size is varied.

Hardware Specification

The following hardware specifications were used for experimentation and benchmarking in this paper. An octa-core computer with 2.5 GHz CPU cores was used for the benchmarking. The system runs Windows 10, 64 bits (OS).

Software Specifications

All concurrency mechanisms used to develop frameworks and algorithms in this paper were provided by Java. Some tools are thread pools used to create and manage threads, some frameworks used to synchronize threads, and to carry out tasks executions. Others are locks, atomic operations, counting semaphores as well as some condition variables.

Running times Measurements

System.currentTimeMillis() method was used to measure running times during benchmarking in this paper. System.currentTimeMillis() method returns a measured time in milliseconds. It gives a difference between the current time and the mid-January, 1970 UTC. System.currentTimeMillis() is faster than System.nanoTime() which makes it a better choice to measure time in this paper.

Data Analysis tool

Graph-Pad prism version 9.3.1 was used to plot all the graphs in this paper. With this tool, a logarithmic scale, linear scale, and natural log scale can be selected depending on the user's choice and the nature of the data to be analyzed. In this paper, a logarithmic scale was used to plot all the graphs due to the large size of our data.

RESULTS

Framework Testing Using Algorithm with Share Data on an Octa-core Machine

In this section, three frameworks were developed to implement algorithms that involved shared data. Comparing algorithms developed using different frameworks that involve shared data enables us to test different implementations of the concurrency mechanism. It will also enable us to see the effects of synchronizations of threads among different processing cores to optimize the performances of these algorithms. Table 2 and Figure 2 described the running times of SD-quick-sort implementation using shared data tested on an 8-core processor machine

Table 2: Depicting the running times of SD-Quick-sort sequential on an octa-core machine

Number of elements	Running times (ms)
1000	0.00
10000	1.00
100000	1.20
1000000	7.10
10000000	3.00

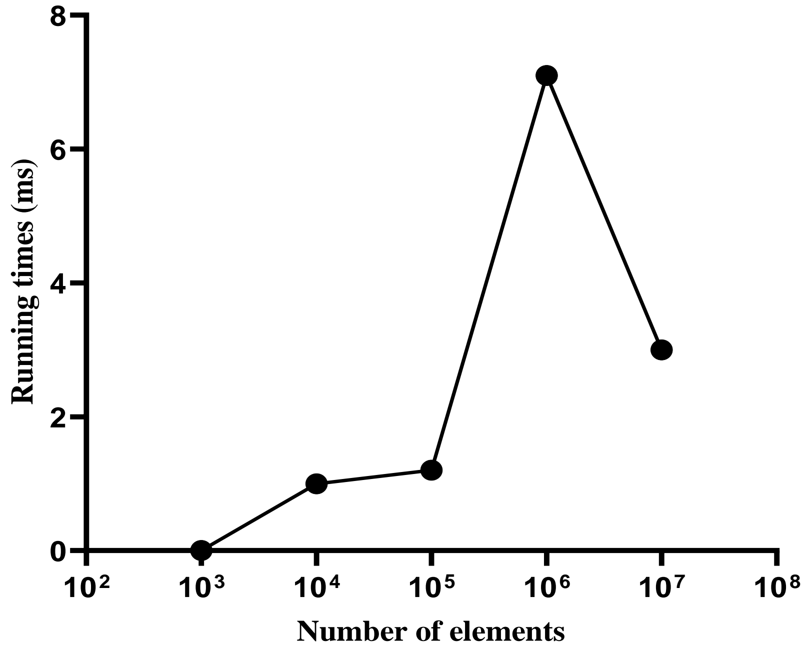

Figure 2: Describing the running times of SD Quick sort sequential on an octa-core machine

SD-Parallel Quicksort Using Atomicity on an Octa-core Machine

Here, a parallel quicksort namely: SD-Parallel quick sort was implemented using the Atomicity framework. The results are described in Table 3 and Figure three respectively.

Table 3: Describing the running times of SD-Parallel quick sort using atomicity an octa-core

Number of elements	Running times (ms)
1000	0.00
10000	9.50
100000	11.2
1000000	0.20
10000000	0.40

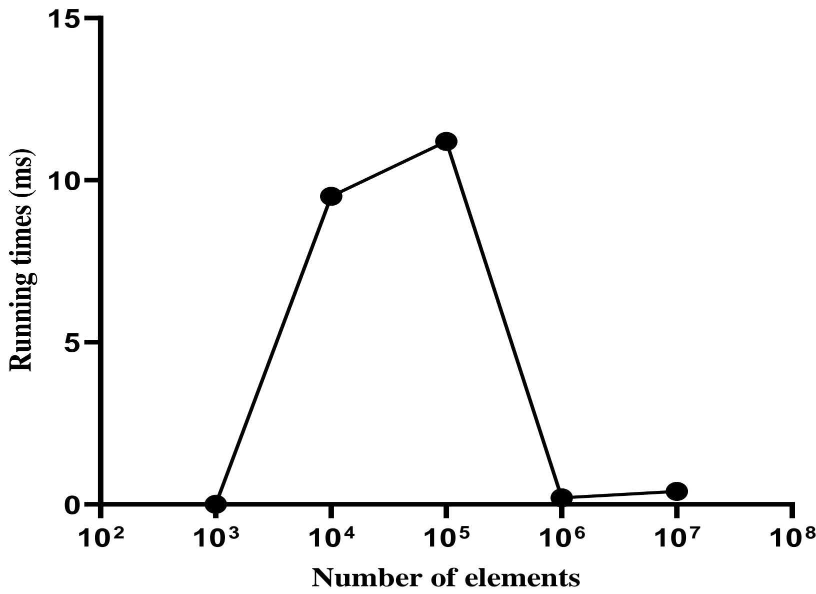

Figure 3: Describing the running times of SD-Parallel quick sort using atomicity an octa-core

Test Runs of SD-Parallel Quicksort Using Acyclic-Barrier

Here, another parallel quicksort was implemented and tested using the Acyclic-Barrier framework. The results are presented in Table 4 and Figure 4 respectively.

Table 4: Describing the running times of SD-Parallel quicksort using Acyclic-Barrier

Number of elements	Running times (ms)
1000	0.10
10000	10.80
100000	11.10
1000000	0.60
10000000	0.20

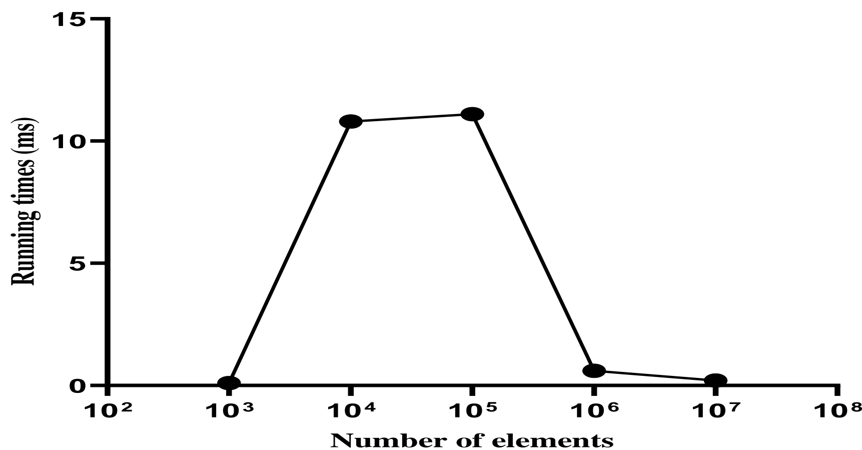

Figure 4: Showing the running times of SD-Parallel quicksort using Acyclic-Barrier

Comparison of all the Three Implementations of Algorithms on an Octa-core Machine

As contained in Table 5 and Figure 5, the running times of all the three sorting algorithms were compared on an octa-core machine.

Table 5: Performance comparison of all the three algorithms on an octa-core machine

Number of elements	SD-Quick-sort Sequential Running times (ms)	SD-Quick-sort Using Atomicity Running times (ms)	SD-Quick-sort Using Acyclic-Barrier Running times (ms)
1000	0.00	0.00	0.10
10000	1.00	9.50	10.80
100000	1.20	11.20	11.10
1000000	7.10	0.20	0.60
10000000	3.00	0.400	0.20

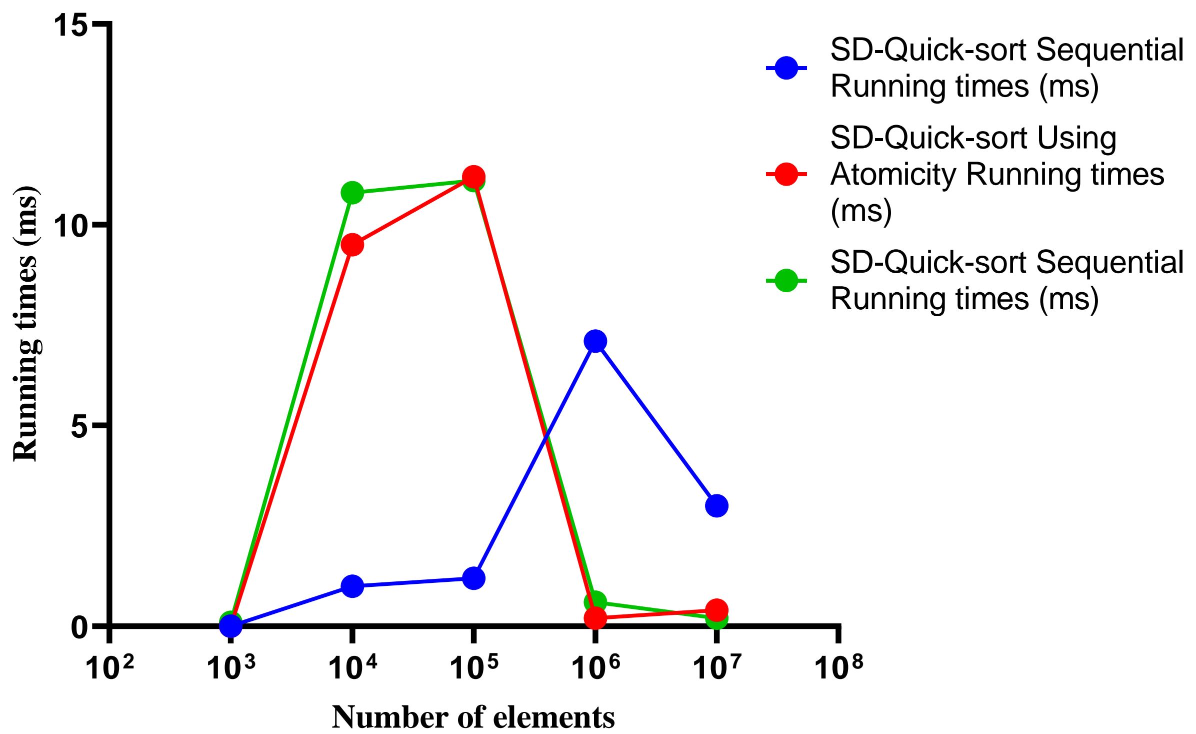

Figure 5: Describing the performance comparison of all the three algorithms on an octa-core machine

DISCUSSION

From Table 2 and Figure 2, it can be seen that the running times of SD-quick sort increase with the increase in the number of elements. While sorting 10,000 array elements, a running time of 1ms was obtained. At the end of the sorting process, when the array size reached 10,000,000 elements, a running time of 3ms was obtained. Therefore, the running times of algorithms with shared data increase with the increase in the array size with proper synchronization put in place. This is similar to the findings in (Rabiu *et al.,* 2021; Ahmed and Zirra, 2013). From the running times obtained in Figure 2 and Figure 3, it can be seen that there was a performance problem when the number of elements is less than 50,000. The decrease in the performance recorded using an octa-core processing machine is significantly higher with the set threshold value. As there were more threads created for a small number of elements, the performance of these algorithms is negatively affected. Thus, it was revealed that creating more threads leads to more overheads. Another possible reason for this poor performance is that some part of the codes was run sequentially as defined by Amdahl's law. Figure 4 and Table 4, it can be observed that the running times of this SD-Quick-sort using the Acyclic-Barrier framework increases with the increase in the array size until when the running times reached 11.10ms before the performance started improving. A running time of 0.2ms was recorded at the end of the sorting process when the number of elements reached 10,000,000. The reason for this poor performance is that, as the array exceeded the set threshold value, the effects of overheads become noticeably higher.

From Figure 5 and Table 5, a performance comparison of all the three algorithms was carried out. It can be observed that their running time rises slowly but it finally settled down as the size of the array increases. It is clearly shown that the set threshold of 50,000 sizes does not suit SD-Parallel quicksort implementation. It can be observed that when the array sizes reach 10,000 elements; SD quick-sort immerged as the best sorting algorithm. This is because Sequential SD-quick sort does well on smaller array sizes. This is following the findings made by Suleiman (2013). However, when the array increases to 100,000 elements, SD-quick sorts implementation using Atomicity becomes the best algorithm throughout the sorting process until the size of the array reached 10,000,000 elements. This is also following the findings made by Dinesh (2021). This good performance is a result of better synchronization mechanisms with which Atomicity is built up. Similarly, as observed when testing these algorithms with a high processing machine (octa-core), it is revealed that a threshold of 50,000 elements does not suit SD-Parallel quick sort implementations. Therefore, a larger threshold value could have been better with this implementation with a little increase in the number of threads. Finally, it was further revealed that SD-Parallel quick sort implementation using Acyclic-Barrier outperforms emerged as the second-best algorithm. This is because of the dynamic nature of the Acyclic-Barrier framework.

CONCLUSION

This study measured and compared the performance of three distinct algorithms using shared data. Each of the algorithms was developed using a different concurrency framework in Java. It was revealed that SD-Parallel Sequential does well on a small number of elements. It was further revealed that the SD-Quick sort developed using Atomicity is the best algorithm when sorting a large number of elements. SD-Quick sort using Acyclic-Barrier emerged as the second-best when measured on a large number of elements. This paper used a limited number of concurrency mechanisms namely: Sequential, Atomicity, and Barrier-Acyclic to develop an algorithm using shared data. Other mechanisms, such as Phaser, and Double Atomicity provided by other JDK versions can be further tested as they are also capable of building a more efficient framework due to their dynamic functionality. These and other

recent concurrency tools provided by Java deserve further investigation to determine their performance.

REFERENCES

Ahmed, M. & Zirra, P. (2013). A comparative analysis of sorting algorithms on integer and character arrays. *International Journal of Engineering and Science 2*(2), 25-30.

Ananymous, (2021). Most popular sorting algorithms [jpg]. Available at: https://www.google.com/search?q=most+popular+sorting+algorithms&tbm=isch&chips=q:most+popular+sorting+algorithms,online_chips:quicksort:g7QfdGMisjU%3D&rlz=1C1IBEF_enNG937NG937&hl=en&sa=X&ved=2ahUKEwjD6LvBtuH1AhW4iv0HHWTSA88Q4lYoBnoECAEQJw&biw=1263&bih=689#imgrc=Ki6YdWGEgGlStM&imgdii=IyQ4JCQ3HIEitM

Dempsey, P. (2014). Monsters incapacitated multi-core processors. Journal of *Engineering & Technology 3*(2)*,* 38-41.

Dinesh B. (2021). Comparison between quicksort, mergeSort, and insertion sort. *Global Sceintific Journal, 9*(4), 2320-9186. Available at: https://www.globalscientificjournal.com/researchpaper/Comparison_between_Quicksort_MergeSort_and_Insertion_Sort.pdf

Göetz, B., Peierls, T., Bloch, J., Bowbeer, J., David, H., & Lea, D. (2006). *Java concurrency in practice.* Addison: Wesley Professional, 1-10.

Hazem, A. A. (2017). Integrating data flow and non-data flow real-time application models on multi-core platforms [Doctorate Thesis], *Faculty of Engineering, Computer Engineering University of Oporto,* 1-144.

Hazem, M.B. (2019). Complexity analysis and performance of double hashing sort algorithm. Journal of the Egyptian Mathematical Society 27(3), 1-10.

Jamil, A., Jamil, A., Maslan, Z., Oliinyk, A., Azwan, A. Rahman, A. I. & Zikri, A. B. (2020). The development of a system for algorithms visualization using sim java. *ARPN Journal of Engineering and Applied Sciences,15*(24), 3024-3033.

Khalid S.A., Ibrahim, M.A., Abdallah, M.I. AlTurani & Nabeel I.Z. (2018). Review on sorting algorithms: A comparative study. *International Journal of Computer Science and Security, 7* (3). 110-120.

Mubashir, A., Harsha, N., Wajid A., Aamir, H., Nosheen, K.M. & Khalid, P. (2020). Experimental analysis of On(log n) class parallels sorting algorithms. *International Journal of Computer Science and Network Security, 20*(1), 139-140.

Muhammad, R. A., Harith Z., Farouk S. & Dauda B. (2017). *Comparison of bubble sort and selection sort with their enhanced versions.* Department of Electrical Engineering University of Lahore, Pakistan. 1-10.

Naeem, A., Muhammad, I., & Furqan, R. (2016). Sorting algorithms: A comparative study. *International Journal of Computer Science and Information Security*, 14(12), 930-940. Available at: https://sites.google.com/site/ijcsis/.

Niraj, K. M. and Rajesh S. (2019). Performance comparison of sorting algorithms based on complexity. *International Journal of Computer Science and Information Technology Research 2*(2)*,* pp. 394-398.

Rabiu, A.M., Garko, A.B. & Abdullahi, A.M. (2018). Effects of multi-core processors on linear and binary sorting algorithms. *Dutse Journal of Pure and Applied Sciences 4*(2). Available at http://fud.edu.ng/journals.php, 130-140.

Rabiu, A.M., Garko, A.B., Abdullahi, A.M, Umar, H.A., & Babagana, M. (2018). Performance evaluation of three quick-sorting algorithms on single and multi-core processors. *Dutse Journal of Pure and Applied Sciences*

4(2). Available at http://fud.edu.ng/journals.php, 254-263.

Rabiu, A.M., Garba, E.G., Baha, B.Y. & Mukhtar, M.I. (2020). Optimizing frameworks for building a more efficient concurrent application in Java. *Islamic University Multidisciplinary Journal (IUMJ)* **7**(2): 348-355. Available online: https://www.iuiu.ac.ug/iumj/ArticleDetails.aspx?jid=14&did=254

Rabiu, A.M., Garba, E.G., Baha, B.Y. & Mukhtar, M.I. (2021). Comparative Analysis Between selection sort and merge sort algorithms. *Nigerian Journal of Basic and Applied Sciences (NJBAS), 29*(1), 48-53.

Sengupta et al. (2007). *Algorithms in java* (3rd ed.). New York: Wesley. Retrieved from: http://www.quora.com/

Suleiman, A. K. (2013). Review on sorting algorithms: A comparative study. *International Journal of Computer Science and Security (IJCSS), 3*(7):120-126.

Yahaya, L., Hassan, I. & Rabiu, A.M. (2020). A survey of the performance of some selected machine learning algorithms for cardiovascular disease predictions. *Journal of Science and Technology 4*(1), 165-180.

Yash, C. & Anuj, D. (2020). Different Sorting Algorithms comparison based upon Time Complexity. *International Journal of Research and Analytical Reviews (IJRAR) 7*(3).