JAMBURA JOURNAL OF INFORMATICS
Volume 5, No. 2 | October 2023, pp. 141-151
ISSN 2685-4244, doi: 10.37905/jji.v5i2.21995

Halstead's complexity measure of a merge sort and modified merge sort algorithms

Ghaniyyat Bolanle Balogun, Muhideen Abdulraheem, Peter Ogirima Sadiku, Olawale Debo Taofeek, Adebisi Sodiq Adewale

Department of Computer Science, University of Ilorin, Kwara State, Nigeria

Article history:
Received September 11, 2023
Revised November 7, 2023
Accepted November 8, 2023
Published November 17, 2023

Keywords:
Algorithms
Complexity measure
Halstead metrics
Modified merge sort algorithm

ABSTRACT. Complexity measuring tools in computer science are deployed to measure and compare different characteristics of algorithms to find the best one to solve a particular problem or that suits a specific situation. Also, this is used to measure the complexity of a software program without running the program itself. Given this, Halstead's complexity metrics are deployed to compare the efficiency of two external sorting methods: the Merge Sort and the Modified Merge Sort Algorithms. The methodology used in achieving this lies in extracting operators and operands from the C_sharp (C#) implemented program of the two algorithms. Six Halstead metrics are evaluated using these operators and operands as parameters. The results show that the modified merge sort algorithm is much more efficient than the conventional Merge sort as it has a lower Program Volume, Program Difficulty, and Program Effort even though the advantage of a higher Intelligence content goes to the merge sort algorithm.

Corresponding Author:
Ghaniyyat Bolanle Balogun,
Department of Computer Science, University of Ilorin,
Ilorin South, Kwara State, Nigeria.
Email: balogun.gb@unilorin.edu.ng

INTRODUCTION

Information technology systems are classified as some of the most complex artifacts that mankind produces (Sharma et al., 2016). In computer programming, as in other aspects of life, there are different ways of solving a problem. These different ways may imply different times, computational power, or any other metric you choose, so we need to compare the efficiency of different approaches to pick the right one.

Assessing complexity can significantly contribute to attaining the various quality attributes associated with a system. The avoidable complexity can be identified and reduced based on the assessment. It holds the key to the success of the system being developed. Various evaluation methods exist which have specific objectives and basis, and all contribute to enhancing product quality (Maushumi & Uzzal, 2019)

Software Complexity influences inward connections. The higher the multifaceted nature, the bigger the deformities. Programming complexity for any product or program is complex to discover without utilizing any measurements. Search algorithm complexity has been mostly evaluated mathematically or by computing the computer execution time. Neither of the two approaches is good enough for practical and realistic purposes, especially when more than one algorithm exists for solving a given problem or class of problems (Hasan et al., 2023).

The work on developing new sorting methods is still ongoing. The concept of big numbers emerged due to the increased development of huge data. Traditional sorting procedures can be used to sort thousands of records, either sorted or unsorted. In some circumstances, the intricacies can be overlooked because of the minor change in execution time. However, suppose the data is huge, and the execution or processing time of billions or trillions of records is significant. In that case, we cannot disregard the complexity of the problem, so an optimal sorting strategy is required (Shabaz & Kumar, 2019). Many writers have attempted to increase the performance of sorting algorithms in data structures, according to Balogun (2019). To overcome the issues with the merge sort algorithm, numerous alternatives have been investigated.

Merge Sort Algorithm

The DAC (Divide and Conquer) concept is used in the Merge sort algorithm, which is an external sorting strategy. For example, it divides a list of records into two smaller units, compares each element to an adjacent list, and then recursively sorts the two pieces or units of data sets, merging and sorting all of the entries in the list. A merge sort, in theory, splits the disorder list into n elements subunits or lists, comparing every aspect of the list until every single element is observed sorted (Irfan et al., 2018). Merge sorting is also known as a divide and conquer method of sorting features, and it is based on this method (Varshney & Yadav, 2014).

One of the most efficient sorting algorithms is merge sort. It operates on the divide-and-conquer premise. Merge sort continuously cuts down a list into numerous sub-lists until each sub-list contains only one entry, then merges those sub-lists into a sorted list. 2020 (Interviewbit). The Fusion Merge Sorting Step Every recursive algorithm relies on a base case and the ability to mix results from several base cases. The merge sort is no exception. The merge step is the most critical aspect of the merge sort algorithm. The merge step solves the problem of combining two sorted lists (arrays) into a single large sorted list (array). The technique keeps track of three-pointers: one for each of the two arrays and one for the final sorted array's current index (Programiz, 2020).

Merge Sort is quite fast and has a time complexity of O(n*log n). It is also a stable sort, which means the "equal" elements are ordered in the same order in the sorted list. The total time for the merge sort function will become n(log n + 1), which gives us a time complexity of O(n*log n). Worst Case Time Complexity [Big-O]: O(n*log n) Best Case Time Complexity [Big-omega]: O(n*log n) Average Time Complexity [Big-theta]: O(n*log n) Space Complexity: O(n).

The time complexity of Merge Sort is O(n*Log n) in all the 3 cases (worst, average, and best) as merge sort always divides the array into two halves and takes linear time to merge two halves. It requires an equal amount of additional space as the unsorted array. It is the most effective method for sorting Linked Lists (Programiz, 2020). To sort a file of n records, the external merge sort algorithm described above requires logn passes. As a result, each record must be read and written to disk logn time. By noting that merge sort is not required for tiny runs, the number of passes can be greatly decreased (Open DSA, 2019). Looked at the time and space complexity of five different sorting algorithms: bubble sort, choosing sort, insertion sort, merge sort, and rapid sort (Rajagopal & Thilakavalli, 2016). Based on the experiments, several findings were produced from aspects of the input sequence. They found that insertion sort or selection sort performs well when the data is minimal, and insertion sort or bubble sort performs well when the sequence is in the ordered form. Their research resulted in a study of sorting algorithms and their attributes.

The quick sort and merge sort algorithms have been widely used for sorting, according to (Taiwo et al., 2020). However, determining which is the most efficient has always been a contentious issue because most of the existing literature has compared these algorithms using machine-dependent factors such as computational complexity. Still, few have used machine-independent factors such as internal/external sorting, algorithm c, and algorithm d. Their research attempted to contribute to this conversation by considering both machine-dependent and independent aspects. Their implementation was done in the MATLAB programming environment, with the internal system clock set to keep track of the sorting time.

The Merge Step of Merge Sort

Every recursive algorithm relies on a base case and the ability to mix results from several base cases. The merge sort is no exception. The merge step is the most critical aspect of the merge sort algorithm. The merge step solves the problem of combining two sorted lists (arrays) into a single large sorted list (array). The technique keeps track of three-pointers: one for each of the two arrays and one for the final sorted array's current index (Programiz, 2020). Merge Sort is a fast algorithm with an O(n*log n) time complexity. It's also a stable sort, meaning the "equal" entries in the sorted list are in the same order.

Worst Case Time Complexity [Big-O]: O(n*log n)
Best Case Time Complexity [Big-omega]: O(n*log n)
Average Time Complexity [Big-theta]: O(n*log n)
Space Complexity: O(n)

Merge Sort has an O(n*Log n) time complexity in all three cases (worst, average, and best) because it splits the array into two halves and merges them in linear time. It takes up the same amount of space as the unsorted array. It is the most effective method for sorting Linked Lists (Programiz, 2020). To sort a file of n records, the external merge sort algorithm described above requires logn passes. As a result, each record must be read and written to disk logn time. By noting that merge sort is not required for tiny runs, the number of passes can be greatly decreased (Open DSA, 2019).

Sample Algorithm for the Mergesort on an array [r] (GeeksforGeeksmergesort, 2020)

```
MergeSort(arr[], l,  r)
If r > l
Step 1 Find the middle point to divide the array into two halves:
          middle m = (l+r)/2
 Step 2 Call mergeSort for first half:
          Call mergeSort(arr, l, m)
  Step 3 Call mergeSort for second half:
          Call mergeSort(arr, m+1, r)
  Step 4 Merge the two halves sorted in step 2 and 3:
          Call merge(arr, l, m, r)
```

Modified Merge sort

In the workings of the modified merge sort algorithm, the sample array is broken down into 'n' large parts. Each part, starting from the first part, '1', is sorted using the quick sort algorithm. This is repeated for all the other parts in 'n'. When all the parts in the array [arr 1, n] have been sorted, they are combined using the merge sort algorithm. This modified merge sort algorithm works on the idea of preventing the merge sort from breaking down the array into the smallest unit before sorting can take place (Balogun, 2021).

Sample Algorithm for the Modified Merge sort on an array [r] (Balogun, 2019).

```
Step 1 – Split the array [r] into [n] parts
Step 2 – Select part [1]
Step 3 − Choose the highest index value as pivot in part [1]
Step 4 − Take two variables to point left and right of the list, excluding the pivot
Step 5 − left points to the low index
Step 6 − right points to the high
Step 7 − while the value at left is less than the pivot, move right
Step 8 − while the value at right is greater than the pivot move left
Step 9 − if both step 5 and step 6 are not matched, swap left and right
```

Step 10 − if left ≥ right, the point where they met is the new pivot
Step 11– if all items in [1] are sorted select next part [1+p]
Step 12 – go to Step 3
Step 13 – If all parts [arr 1, n] are sorted then go to next step else go to step 11
Step 14 Find the middle point to divide the parts [arr 1, n] into two halves: middle m = (l+n)/2
Step 15 Call mergeSort for first half: Call mergeSort(arr, l, m)
Step 16 Call mergeSort for second half: Call mergeSort(arr, m+1, n)
Step 17 Merge the two halves sorted in step 15 and 16: Call merge(arr, l, m, n)

Methods

The methodology used in comparing the two external sorting methods (Merge sort and Modified Merge sort) lies in implementing the two algorithms using C. The operators and operands in each C program are extracted, and the result is tabulated. These operators and operands are then substituted in Halstead's metrics to obtain their values.

Operators and Operands

n1 = Number of distinct operators.
n2 = Number of distinct operands.
N1 = Total number of distinct operators.
N2 = Total number of distinct operands.

Halstead Metrics

I. Halstead Program Length–The number of distinct operators and the number of distinct operands. N = N1+N2

II. Halstead Vocabulary–The number of unique operators and unique operand occurrences. n = n1+n2

III. Program Volume–Proportional to program size, represents the size, in bits, of space necessary for storing the program. This parameter is dependent on specific algorithm implementation. The properties V, N, and the number of lines in the code are shown to be linearly connected and equally valid for measuring relative program size.
V = Size*(log2 Vocabulary) = N*log2(n)

IV. Program Difficulty–This parameter shows how difficult it is to handle the program.
D = (n1/2)*(N2/n2), D = 1/L/
As the volume of program implementation increases, the program level decreases, and the difficulty increases. Thus, programming practices such as redundant operands or the failure to use higher-level control constructs will increase the volume and difficulty.

V. Programming Effort–Measures the mental activity needed to translate the existing algorithm into implementation in the specified program language.
E = V/L = D*V = Difficulty*Volume

VI. Intelligence Content–Determines the amount of intelligence presented (stated) in the program. This parameter measures program complexity independently of the program language in which it was implemented.
I = V/D

Merge Sort

Table 1 illustrates the number of operators and operands for merge sort, and Table 2 illustrates the number of operators and operands for modified merge sort, respectively.

Table 1. Number of operators and operands for merge sort

Operators	Number of Occurence	Operand	Number of Occurence
{}	14	STAThread	1
[]	29	K	11
Void	3	Y	2
Char	2	Result	3
=	33	Filename	3
;	51	Text	3
Do	1	Null	1
DialogResult	2	fileChooser	1
String	4	DialogResult.OK	1
()	38	File.ReadAllLines	1
OpenFileDialog	2	System.Environment.Exit	1
New	5	0	8
FileChooser.FileName	1	N	7
If	3	text.Length	1
==	4	X	4
Else	2	J	9
Int	22	FileChooser.ShowDialog	1
Foreach	1	S	2
int.parse	1	Watch	3
Stopwatch	2	Sort	4
For	3	I	21
<	8	Time	1
\n	3	1000	1
Execution	1	Y	2
Float	1	myArray	4
watch.ElapsedMilliseconds	1	myArray.Length	1
/	3	1	1
Try	1	2	1
char.parse	1	L	8
Catch	1	R	4
While	4	leftArray	9
||	1	rightArray	9
Return	1	System.exception	1
-	1	Merge	2
+	1	OriginalArray	5
,	6	leftArray.Length	1
&&	1	rightArray.Length	1
<=	1		
++	12		
n1 =39	N1 = 271	n2 = 37	N2 = 139

Modified Merge Sort

Table 2. Number of operators and operands for modified merge sort

Operators	Number of Occurrences	Operands	Number of Occurrences
{}	19	STAThread	1
[]	36	Response	5
Void	4	Y	3
String	4	Result	5
Char	2	Filename	4

Operators	Number of Occurrences	Operands	Number of Occurrences
=	39	Text	4
;	63	Null	1
Do	2	fileChooser	3
DialogResult	2	fileChooser.ShowDialog	1
OpenFileDialog	2	fileChooser.FileName	1
New	5	DialogResult.OK	1
()	46	System.Environment.Exit	1
If	4	0	10
==	3	N	5
file.ReadAllLines	1	text.length	1
Else	2	X	5
Int	34	J	9
Foreach	1	S	2
++	14	L	7
Int.parse	1	1	5
+	4	2	1
/	2	X1	6
For	4	X2	6
<	8	I	26
Stopwatch	2	X1.Length	2
-	4	X2.Length	2
Float	1	Watch	6
Try	1	Quicksort	5
Char.parse	1	Merge	2
Catch	1	x.Length	1
While	4	Watch.elapsedMilliseconds	1
||	1	1000	1
>=	1	Console.readline	1
Return	2	System.Exception	1
<=	2	A	14
&&	1	Start	8
		End	8
		Pindex	8
		Partition	2
		Pivot	2
		Temp	2
		B	5
		OriginalArray	5
		leftArray	5
		rightArray	5
		K	6
		R	2
n1 = 36	N1 = 323	n2 = 47	N2 = 207

Results and Discussion

Merge sort

(1) Program Length (N) = N1 + N2 = 271 + 139 = 410
(2) Program Vocabulary (n)= n1 + n2 = 39 + 37 = 76
(3) Program Volume (V) = N log2 n =410log276 = 2561.65 bits
(4) Program Difficulty (D) = n1/2 * N2/n2 = 39/2 * 139/37 = 14097/74 = 190.5
(5) Programming Effort (E) = D * V = 190.5 * 2561.65 = 487994.325
(6) Intelligence Content (I) = V/D = 2561.65/190.5 = 13.48

Modified Mergesort

(1)	Program Length (N)	= N1 + N2 = 323 + 207 = 530
(2)	Program Vocabulary (n)	= n1 + n2 = 36 + 47 = 80
(3)	Program Volume (V)	= N log2 n = 530log280 =511.12 bits
(4)	Program Difficulty (D)	= n1/2 * N2/n2 = 36/2 * 207/47 = 7452/94 = 79.2766
(5)	Programming Effort (E)	= D * V = 79.2766 * 511.12 = 40519.85
(6)	Intelligence Content (I)	= V/D = 511.12/79.2766 =6.45

Table 3, Table 4, Table 5, and Table 6 represent Program length, Program vocabulary, Program volume, and Program difficulty. Figures 1 to 6 show the merge sort has a smaller program length and program vocabulary when compared to the modified merge sort.

Table 3. Program length

Mergesort	Modified Mergesort
410	530

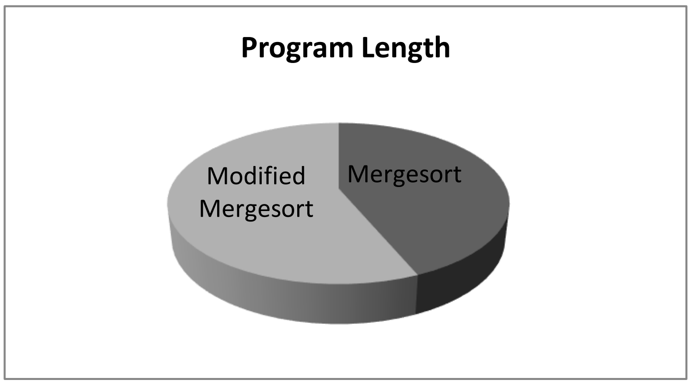

Figure 1. Program length

Table 4. Program vocabulary

Mergesort	Modified Mergesort
76	80

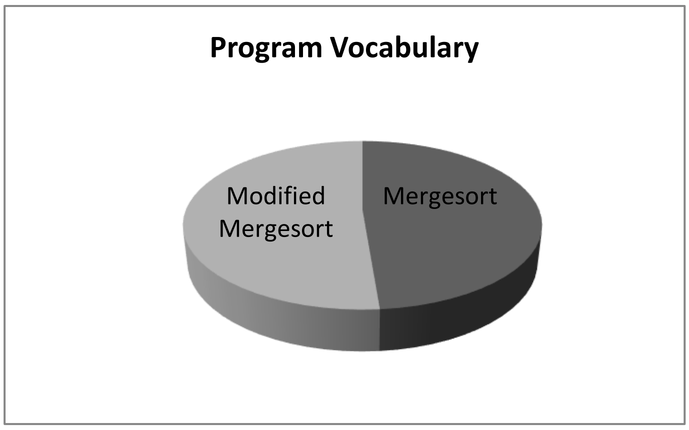

Figure 2. Program length

Table 5. Program volume

Mergesort	Modified Mergesort
2561.65	511.12

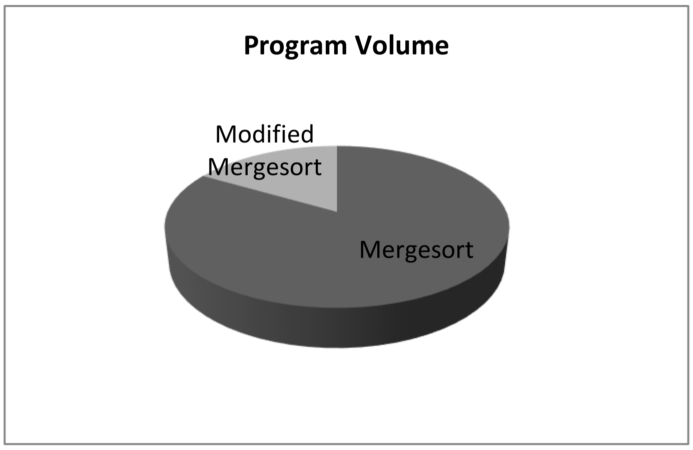

Figure 3. Program volume

Table 6. Program difficulty

Mergesort	Modified Mergesort
190.5	79.2766

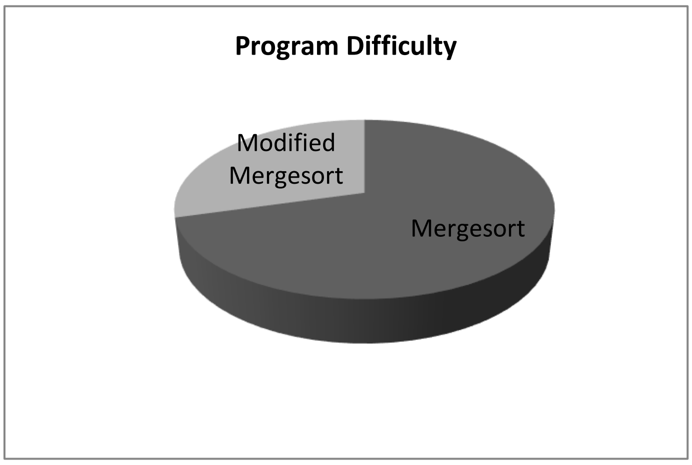

Figure 4. Program length

Table 7. Programming effort

Mergesort	Modified Mergesort
487994.325	40519.85

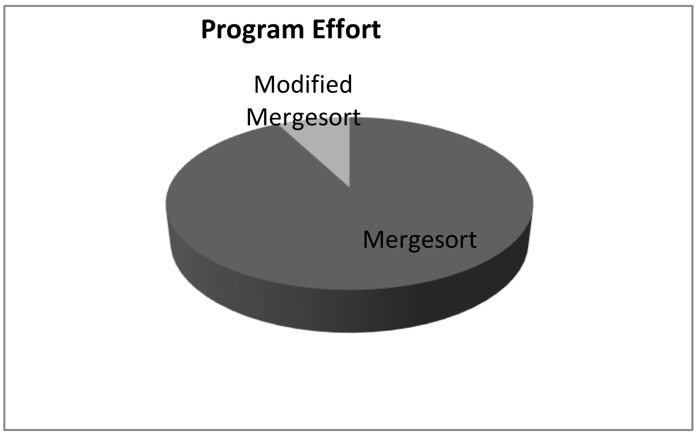

Figure 5. Program length

Table 8. Intelligence content

Mergesort	Modified Mergesort
13.48	6.45

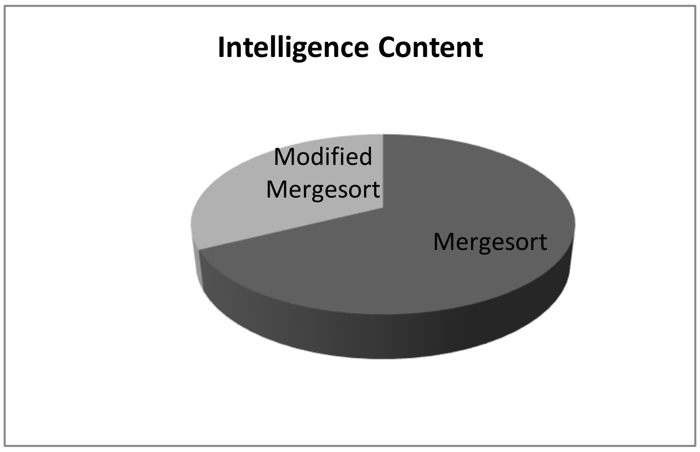

Figure 6. Program length

However, the bit size of the program volume, the metric used in measuring space efficiency, indicates that the modified mergesort algorithm is far more efficient than the mergesort algorithm. The Halstead metrics result also shows that the program difficulty and effort required for mergesort implementation far outweigh that of the modified mergesort. This implies that it is much more difficult with the more considerable effort required in using the mergesort algorithm when compared to the use of the modified merge sort algorithm. Thus implying that the modified merge sort is more efficient than the mergesort algorithm. However, the measure of the intelligence indicates that the merge sort is about twice as intelligent as the modified merge sort.

CONCLUSION

The mergesort has always been a popular and efficient external sorting method. However, a significant disadvantage of this algorithm lies in breaking down the data to its smallest part before sorting can be implemented. This led to its modification in the modified merge sort algorithm. This, therefore, compared the efficiency of the two sorting algorithms using Halstead's metrics. The result shows that the modified merge sort algorithm is more efficient regarding program volume, programming difficulty, and effort than the merge sort. But, the mergesort carries the day in terms of the intelligence content. The findings of the comparison between the standard merge sort and the modified merge sort algorithms using Halstead's metrics revealed that the necessity to break down data into its most minor parts in the merge sort algorithm is a significant weakness, primarily due to its high memory requirements, making it less suitable for systems with limited memory resources. The modified merge sort outperformed the standard merge sort regarding program volume, programming difficulty, and programming effort. However, the usual merge sort retained an advantage regarding intelligence content, indicating its potential in more complex or versatile applications. Further research opportunities lie in empirical testing to validate these findings, defining more precise criteria for algorithm evaluation, exploring their performance in specific application contexts with varying data types and sizes, including time and space complexity measurements, and discussing practical implications to aid decision-making in choosing the appropriate sorting algorithm for specific needs and resource constraints.

REFERENCES

Balogun, G. B. (2019). A modified linear search algorithm. *African Journal of Computer Science & ICT*, *12*(2), 43-54.

Balogun, G. B., Olanrewaju, B. A., Awotunde, J. B., Oladipo, I. D., & AbdulRaheem, M. (2021). Evaluating the time efficiency of a modified merge sort algorithm. *Bulletin of the Science Association of Nigeria*. 32. 141-159.

Diego, L. Y. (2020). Time complexity: how to measure the efficiency of algorithms. Retrieved February 2022 from https://www.kdnuggets.com/2020/06/time-complexity-measure-efficiency-algorithms.html

GeeksforGeeks (2020). External sorting. Retrieved May 2020 from https://www.geeksforgeeks.org/external-sorting

GeeksforGeeks. Merge sort. Retrieved May 2020 from https://www.geeksforgeeks.org/merge-sort

Hariprasad, T., Vidhyagaran, G., K. Seenu, K., Thirumalai, C. (2017). Software complexity analysis using Halstead metrics. In *International Conference on Trends in Electronics and Informatics ICOEI* 2017, Tirunelveli, India. doi: 10.1109/ICOEI.2017.8300883

Hassan, A. A., Kvasnikov, A. A., Klyukin, D. V., Ivanov, A. A., Demakov, A. V., Mochalov, D. M., & Kuksenko, S.P. (2023). On modeling antennas using mom-based algorithms: Wire-grid versus surface triangulation. *Algorithms*, *16*(4), 200. doi: 10.3390/a16040200

InterviewBit. Merge sort algorithm. Retrieved May 2020 from https://www.interviewbit.com/tutorial/merge-sort-algorithm

Irfan, A., Haque, N., Imran, K., Ameen, M.C., & Malook, M.R. (2018). Performance comparison between merge and quick sort algorithms in data structure. *International Journal of Advanced Computer Science and Applications (IJACSA)*, *9*(11), 192-195. doi: 10.14569/IJACSA.2018.091127

Maushumi, L. & Uzzal S. (2019). Complexity assessment based on UML-activity diagram. *International Journal of Recent Technology and Engineering (IJRTE)*, *8*(2), 6117-6122.

Mohammad S. & Ashok K. (2019). A novel sorting technique for large-scale data. *Journal of Computer Networks and Communications*, *1*(1), 1-7.

Open DSA (2019). CS 3. Data structures and algorithms. External sorting. Retrieved May 2020 from https://opendsa-server.cs.vt.edu/ODSA/Books/CS3/html/ExternalSort.html

Programiz. (2020). Merge sort algorithm. Retrieved May 2020 from https://www.programiz.com/dsa/merge-sort

Rabiu, A. M., Garba, E. J., Baha, B. Y., & Mukhtar, M. I. (2021). Comparative analysis between selection sort and merge sort algorithms. *Nigerian Journal of Basic and Applied Sciences*, *29*(1), 43-48. doi: 10.4314/njbas.v29i1.5

Rajagopal, D., & Thilakavalli, K. (2016). Different sorting algorithm's comparison based upon the time complexity. *International Journal of U- and e-Service, Science and Technology*, *9*(8), 287–296. doi: 10.14257/ijunesst.2016.9.8.24

Senan, S., & Sevgen, S. (2017). Measuring software complexity using neural networks. *Journal of Electrical and Electronics Engineering*, *17*(2), 3503-3508.

Shabaz, M., & Kumar, A. (2019). SA sorting: A novel sorting technique for large-scale data. *Journal of Computer Networks and Communications*. doi: 10.1155/2019/3027578

Sharma, C. B. Panwar and R. Arya (Diego, 2022). High power pulsed current laser diode driver. In *International Conference on Electrical Power and Energy Systems (ICEPES)*, Bhopal, India, *1*(1), 120-126, doi: 10.1109/ICEPES.20A

Taiwo, O. E., Christianah, A. O., Oluwatobi, A. N., & Aderonke, K. A. (2020). Comparative study of two divide and conquer sorting algorithms: Quicksort and mergesort. *Procedia Computer Science*, 171, 2532-2540. doi: 10.1016/j.procs.2020.04.274

Yadav, R., & Varshney, K. (2014). Brief study about the variation of complexities in algorithmic merge sort. *International Journal of Advanced Research in Computer Science and Software Engineering*, *4*(2), 874-878.