Enhancement of Traditional Merge Sort

Raj Patel[a], Rushi Patel*[b]
[a]MTech Scholar
School of Computer Science and Engineering
Vellore Institute of Technology, Vellore – 632014, Tamil Nadu, India
[b]MTech Scholar
School of Computer Science and Engineering
Vellore Institute of Technology, Vellore – 632014, Tamil Nadu, India
E-mail: patel.raj2022@vitstudent.ac.in, patel.rushi2022@vitstudent.ac.in

Abstract

We all understand that sorting is fundamental computer science problem. Data sorting is used in many of applications and it plays main role for deciding performance of software, speed of any algorithm, and power consumption of any algorithm. In computer and mathematics many sorting methods are developed over decades. For sort the large data set merge sort performs better comparing to other sorting algorithms. If we want to minimize the execution time of sorting algorithm then these sorting algorithms must be efficient. In this paper, Improve the classical merge sort algorithm and to introduce a new approach that performs faster. The algorithm which we introduce here is more efficient than the traditional Merge sort and it takes less time with complexity O(nlogn). Here we also tested proposed algorithm for alphanumeric values ad string also.

Keywords: External Sorting, Recursive Sort, Merge Sort, Improved Merge sort, Divide and Conquer

Introduction

When we reorder the element in decreasing or increasing order computationally is called sorting. Mathematics and computer science both teach sorting algorithms. Everyone is aware that every search engine essentially uses a sorting algorithm to display the most relevant results for the keyword that the user has entered. The tables and file system in both cases may be very vast and contain thousands of data sources that need to be swiftly sorted.

Over six decades of research have resulted in over a hundred algorithms for solving the sorting problem, many of which are minor or major modifications of tens of conventional algorithms. Bubble (sinking sort), Selection, Insertion all are works on $O(N^2)$ time complexity which affect on their performance while we use these for large data set instead of this if we use quick sort or merge sort for large data then it takes less time compare to above sorting algorithm.

Here, we work to improve these traditional sorting algorithms while attempting to make them less difficult. In this paper we will discuss about the conventional comparison based merge sort and enhancement of this conventional merge sort.

Based on their time complexity and space complexity our proposed merge sort is same as classical merge sort but it runs faster than classical merge sort.

We looked at classical merge sort in the first part of this article. In the second section, we had discussed on proposed merge sort with that also see the comparison the classical merge sort with proposed approach. We covered comparison of proposed merge sort and traditional merge sort for string and alphanumeric values.

Classical Merge Sort

Divide and conquer is the basis for the traditional merge sort sorting method. In each recursion array is split in half, equally until there is no space or only one element left. Now it merge this all the parts recursively in sorted manner and at last we get the fully sorted array.

Performance analysis of classical merge sort:

In merge sort we do many iteration to get sorted sequence. In which we merge 1 size array in the first iteration, merge two size array in second iteration likewise in i^{th} iteration we merge two 2^{i-1} size array. The total number of iteration is therefore [log2n]. In merge operation we combine two sorted array that takes O (n) time. So, The average computing time is O(nlogn).

Table 1. Classical Merge Sort Time Complexity

Best Case	O (nlogn)
Average Case	O (nlogn)
Worst Case	O (nlogn)

Algorithm:

mergesort (array,left,right):

1. Initiate
2. Initialize the array A_array and l_pointer,r_pointer,m_pointer variable (l_pointer=left element index, r_pointer=right element index, m_pointer=middle element index)
3. if l_pointer > r_pointer
4. return
5. m_pointer = (l_pointer +r_pointer)/2
6. mergesort(A_array, l_pointer, m_pointer)
7. mergesort(A, m_pointer, r_pointer)
8. merge(A_array, l_pointer, m_pointer, r_pointer)
9. Terminate

merge (A_array, l_pointer, m_pointer, r_pointer):

```
1.  N1_size = m_pointer -l_pointer +1
2.  N2_size = r_pointer -m_pointer
3.  Declare two array L_array and M_array of size N1_size and N2_size respectively
4.  For ( i=0 to N1_size ):
5.      L_array[i]=A_array[l_pointer +i]
6.  End for
7.  For ( i=0 to N2_size):
8.      M_array[i]=A_array[m_pointer +1+i]
9.  End for
10. i=0,j=0
11. k_pointer =l_pointer
12. while ( i<=N1_size && j<=N2_size ):
13.     if ( L_array[i] <= M_array[j] ):
14.         A_array[k_pointer]=L_array[i]
15.         i++
16.     End if
17.      else:
18.         A_array[k_pointer]=M_array[j]
19.         j++
20.     End else
21.     k_pointer ++
22. End while
23. while ( i <= N1_size ):
24.     A_array[k_pointer] = L_array[i]
25.     i++
```

26. k_pointer ++
27. End while
28. While (j <= N2_size):
29. A_array[k_pointer] = M_array[j]
30. j++
31. k_pointer ++
32. End While

Proposed Approach for Merge Sort

Divide and conquer is the basis for the traditional merge sort sorting method. In each recursion array is split in half, equally until there is no space or only one element left. So basically, we do same procedure until we divide the array but, While Merging two sorted element we maintain four pointer, two pointer i & j for starting index of the two sorted array and two pointer p & q for ending index of the two sorted array.

Now while merging two sorted array first compare L[i] and M[j] (L and M is sorted array) if L[i] < M[j] then copy L[i] to appropriate position at left side of the original array and increment i by 1 otherwise copy M[j] to left side of the original array and increment j by 1 and also in this iteration compare L[p] and M[q] , if L[p]>M[q] then copy L[p] to right side of the original array and decrement p by 1 otherwise copy M[q] to right side of the original array and decrement q by 1 and so on until i<=p and j<=q.

Algorithm:

mergesort (array,left,right):

1. Initiate
2. Initialize the array A_array and l_pointer,r_pointer,m_pointer variable (l_pointer=left element index, r_pointer=right element index, m_pointer=middle element index)
3. if l_pointer > r_pointer

4. return
5. m_pointer = (l_pointer +r_pointer)/2
6. mergesort(A_array, l_pointer, m_pointer)
7. mergesort(A, m_pointer, r_pointer)
8. merge(A_array, l_pointer, m_pointer, r_pointer)
9. Terminate

Merge (A_array, l_pointer, m_pointer, r_pointer):

1. N1_size = m_pointer -l_pointer +1
2. N2_size = r_pointer -m_pointer
3. Declare two array L_array and M_array of size N1_size and N2_size respectively
4. For (i=0 to N1_size):
5. L_array[i]=A_array[l_pointer +i]
6. End for
7. For (i=0 to N2_size):
8. M_array[i]=A_array[m_pointer +1+i]
9. End for
10. i=0,j=0
11. k_pointer =l_pointer
12. p=N1_size-1, q=N2_size-1
13. r=r_pointer
14. while (i<=p && j<=q):
15. if (L_array[i] <= M_array[j]):
16. A_array[k_pointer]=L_array[i]
17. i++
18. End if

```
19.      else:
20.          A_array[k_pointer]=M_array[j]
21.          j++
22.      End else
23.      k_pointer ++
24.      if ( L_array[p] >= M_array[q] ):
25.          A_array[r]=L_array[p]
26.          p—
27.      End if
28.      else
29.          A_array[r] = M_array[q]
30.          q—
31.      End else
32.      r—
33.  End while
34.  while ( i <= p ):
35.      A_array[k_pointer] = L_array[i]
36.      i++
37.      k_pointer ++
38.  End while
39.  While ( j <= q ):
40.      A_array[k_pointer] = M_array[j]
41.      j++
42.      k_pointer ++
43.  End While
```

In below example (Figure 1), we took eight random value and demonstrate proposed merge sort. In which very first pass we divide the list up to it contain single element and we know single element is already sorted. Now in the merge procedure we have to merge two sorted list L and M, for this we merge the list from both the side left and right. Like as we can see in the above example sorted list { 1 , 3 , 5 , 7 } and { 2 , 6 , 8 , 9 }. Now assign pointer i is point to " 1 " in the first list and pointer j is point to " 2 " in the second list same as for right side pointer p is point to " 7 " in the first list and pointer q is point to " 9 " in the second list. So basically I and j pointer is point minimum of the both the list and pointer p and q is point to maximum of the both the list. Now compare L[i] and M[j] , if L[i] <= M[j] then we copy L[i] element to original array and increment i by one else copy M[j] element to original array and increment j by one now in the same iteration also compare L[p] and M[q], if L[p] >= M[q] then copy L[p] element to original array and decrement p by one else copy M[q] element to original array and decrement q by one. As we can see in the above example, iteration 1, element 1 and 9 (yellow shaded) are fixed at its appropriate position in the original list. Same as in iteration 2, element { 1 , 2 , 8 , 9 } is at appropriate position in the original list we have to continue the iteration until i<=p or j<=q.

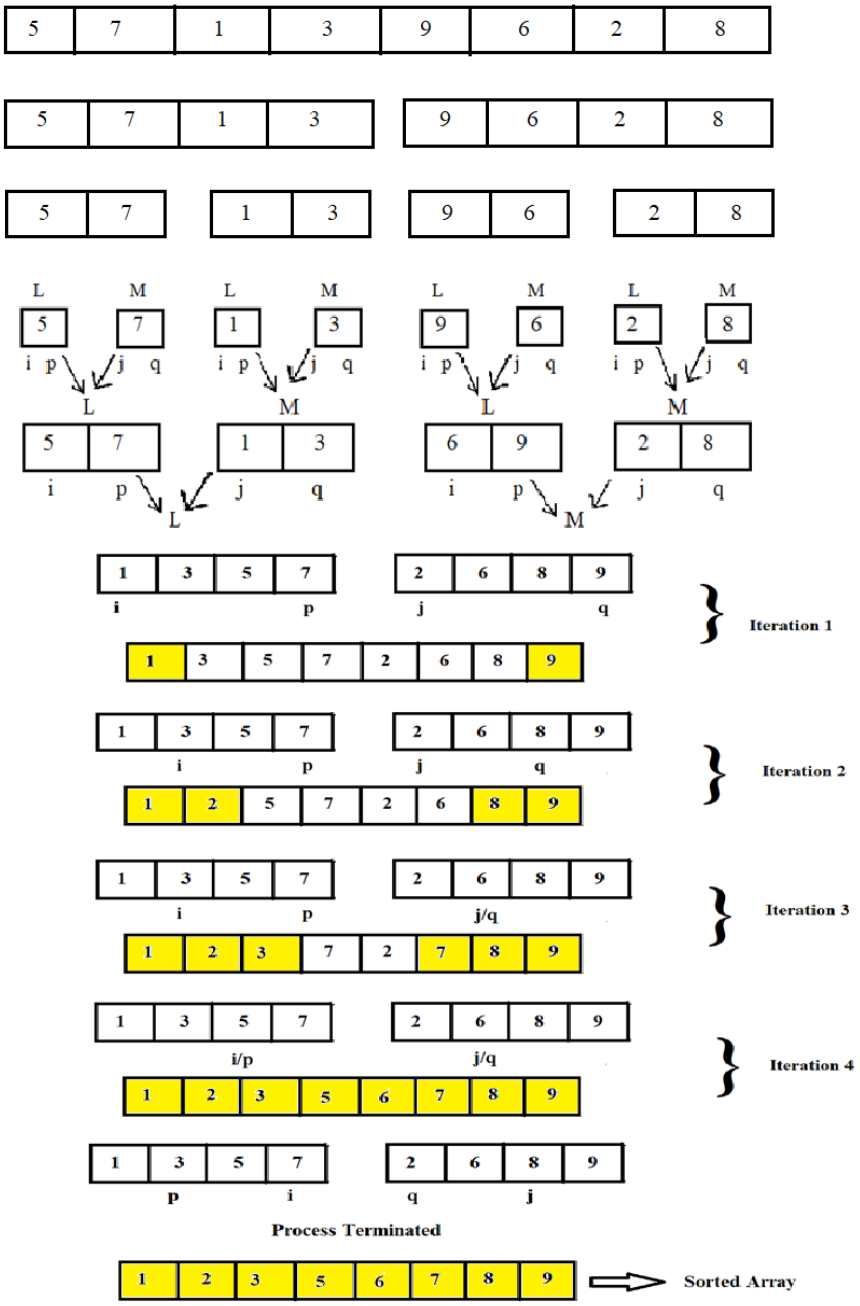

Figure 1. Flow diagram of Proposed Merge Sort

Performance analysis of proposed merge sort

In merge sort we do many iteration to get sorted sequence. In which we merge 1 size array in the first iteration, merge two size array in second iteration likewise in i^{th} iteration we merge two 2^{i-1} size array. The total number of iteration is therefore [log2n]. Here in proposed merge sort if there are n numbers in the list then we have to merge two sorted n/2 size list and as per the proposed algorithm we merge these both the sorted list by both side (left and right) so it

takes less time compare to classical merge sort. As proposed in merge, we may merge two sorted part in linear time, implying that each iteration takes O(n). So overall computing time is O(nlogn) because there are [log2n] passes .

Table 2. Proposed Merge Sort Time Complexity

Best Case	O (nlogn)
Average Case	O (nlogn)
Worst Case	O (nlogn)

Result and Discussion

Experimental System:

CPU: Intel® Core™ i5-7200U CPU @ 2.50GHz

RAM: 12GB

Solid State Drive: 512GB

No. Of Core: 2

Logical Processor: 4

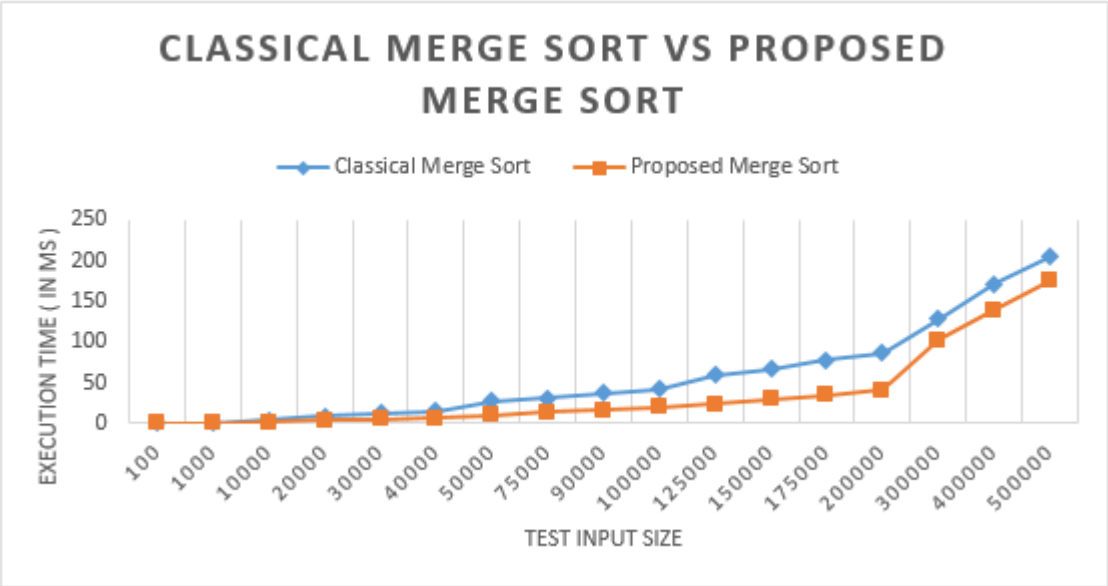

Figure 2. The proposed merge sort is compared to the traditional merge sort

In above graph (Figure 2) , blue line shows execution time of the classical merge sort on random integer value array of size 100 to 500000 and orange line shows execution of the proposed merge sort on random integer value of size 100 to 500000. After observing the above graph we can say that proposed merge sort takes lesser time compare to conventional merge sort.

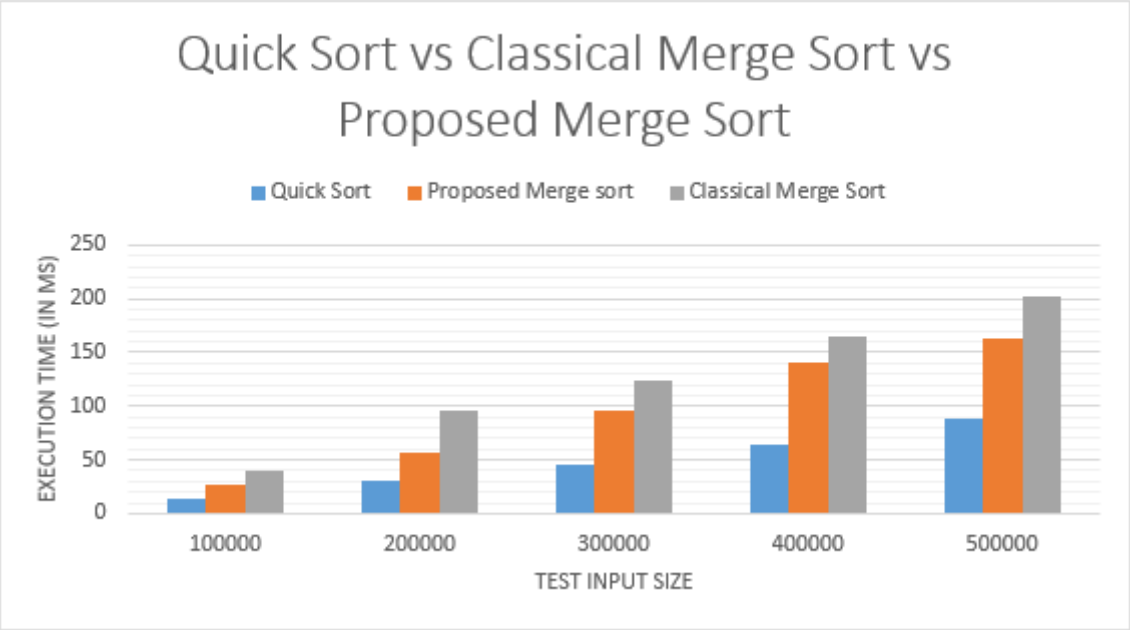

Figure 3. Comparison of the suggested merge sort, the traditional merge sort, and quick sort

In above graph (Figure 3), we have compared classical quick sort, classical merge sort, and proposed merge sort on random integer value array of size 100000 to 500000. In above graph blue bar describes execution time of quick sort, orange bar shows running time of the proposed merge sort and grey bar shows running time of the classical merge sort. After observing the abpve graph, we can say that quick sort is very fast compare to classical merge sort and proposed merge sort also but the proposed merge sort takes lesser time compare to conventional merge sort..

In below graph (Figure 4), we have compared classical merge sort and proposed merge sort on random string array of size 100 to 75000. In the graph blue light shows execution time of the

classical merge sort and orange line shows execution time of the proposed merge sort. Here we can say that proposed algorithm is not that much faster than classical algorithm for string.

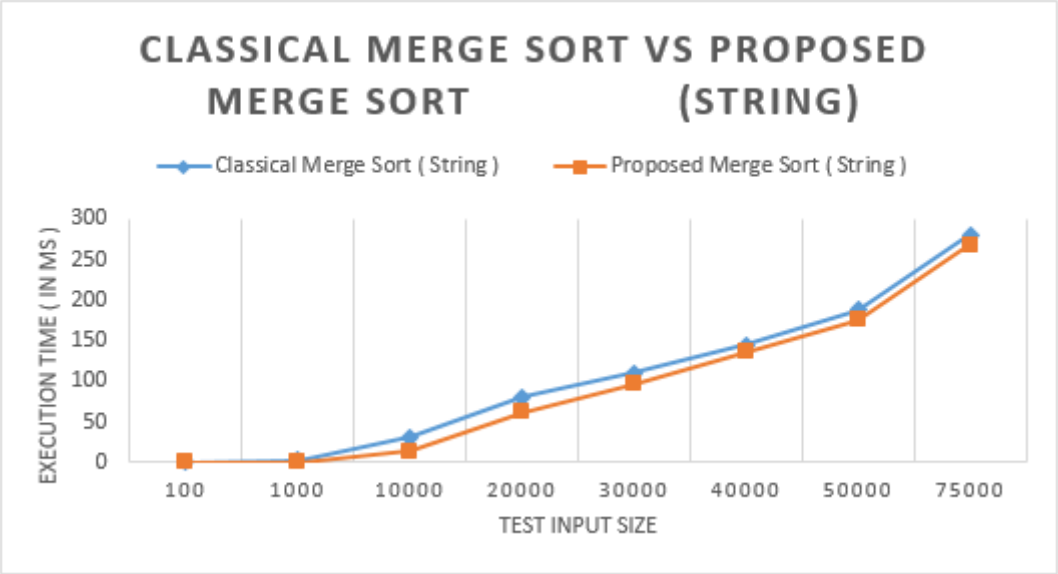

Figure 4. The suggested merge sort for strings is compared to the traditional merge sort

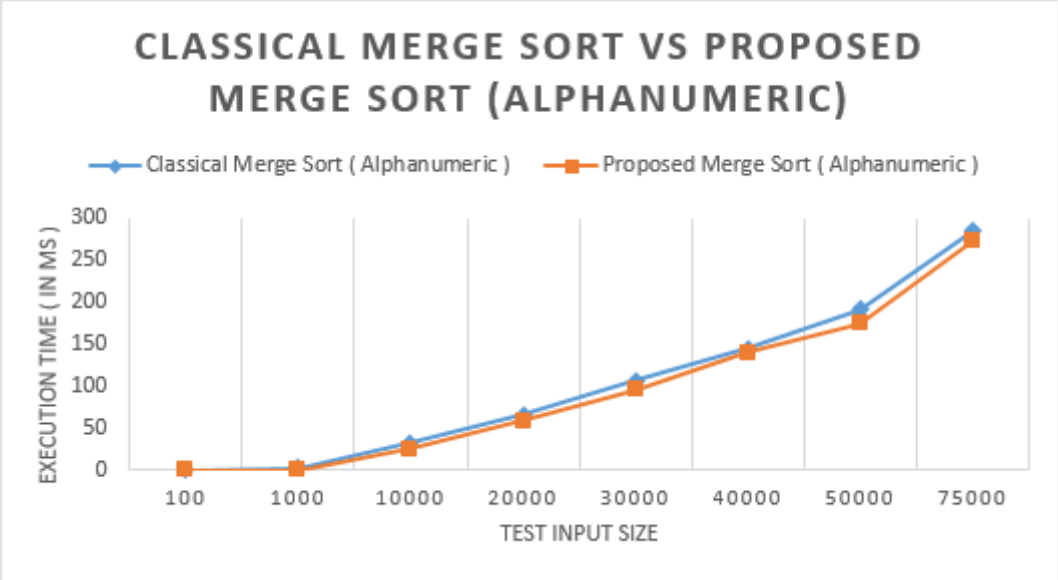

Figure 5. The suggested merge sort for alphanumeric is compared to the traditional merge sort

In above graph (Figure 5), we have compared classical merge sort and proposed merge sort on random alphanumeric array of size 100 to 75000. In the graph blue light shows execution time of the classical merge sort and orange line shows execution time of the proposed merge sort.

Here we can say that proposed algorithm is not that much faster than classical algorithm for alphanumeric.

Conclusion

As per the above observations and results, we can conclude that classical merge sort is slower compare to proposed merge sort but still it is slower than quick sort and also it does not works efficiently for the alphanumeric and string data.

Acknowledgement

Dr. Saravanan R, a professor at the Vellore Institute of Technology in Tamil Nadu, who worked as our research guide and gave us the opportunity to carry out this study, deserves our sincere gratitude. His passion, vision, genuineness, and enthusiasm have significantly motivated us. He has shared expertise about how to do out research and clearly communicate the study's findings.

We are incredibly appreciative of all the love, prayers, care, and sacrifices made by our parents to raise us and prepare us for the future.

References

Abdulla, M. (2016). Selection Sort with Improved Asymptotic Time Bounds. *The International Journal Of Engineering And Science (IJES)*, *5*(5), 125-130.

Alyasseri, Z. A. A., Al-Attar, K., & Nasser, M. (2014). Parallelize Bubble Sort Algorithm Using OpenMP. *arXiv preprint arXiv:1407.6603*.

Appiah, O., & Martey, E. M. (2015). Magnetic Bubble Sort Algorithm. *International Journal of Computer Applications*, *122*(21).

Chand, S., Chaudhary, T., & Parveen, R. (2011). Upgraded selection sort. *International Journal on Computer Science and Engineering*, *3*(4), 1633-1637.

Chhatwani, M. P. K. (2014). Insertion sort with its enhancement. *International Journal of Computer Science and Mobile Computing*, *3*(3), 801-806.

Edjlal, R., Edjlal, A., & Moradi, T. (2011, March). A sort implementation comparing with bubble sort and selection sort. In *2011 3rd International Conference on Computer Research and Development* (Vol. 4, pp. 380-381). IEEE.

Goyani, M., Chharchhodawala, M., & Mendapara, B. (2013). Min-Max Selection Sort Algorithm–

Improved Version of Selection Sort. *Int. J. Adv. Res. Comput. Sci. Softw. Eng.*, 6.

Hayfron-Acquah, J. B., Appiah, O., & Riverson, K. (2015). Improved selection sort algorithm. *International Journal of Computer Applications*, *110*(5).

Jadoon, S., Solehria, S. F., & Qayum, M. (2011). Optimized selection sort algorithm is faster than insertion sort algorithm: a comparative study. *International Journal of Electrical & Computer Sciences IJECS-IJENS*, *11*(02), 19-24.

Kumar, A., Dutt, A., & Saini, G. (2014). Merge Sort Algorithm. *International Journal of Research*, *1*(11), 16-21.

Li, L., & Rui, S. (2012). Experiment analysis on the bubble sort algorithm and its improved algorithms. In *Information Engineering and Applications* (pp. 24-31). Springer, London.

Lobo, J., & Kuwelkar, S. (2020, July). Performance analysis of merge sort algorithms. In *2020 International Conference on Electronics and Sustainable Communication Systems (ICESC)* (pp. 110-115). IEEE.

Paira, S., Chandra, S., & Alam, S. S. (2016). Enhanced Merge Sort-a new approach to the merging process. *Procedia Computer Science*, *93*, 982-987.

Sodhi, T. S., Kaur, S., & Kaur, S. (2013). Enhanced insertion sort algorithm. *International journal of Computer applications*, *64*(21).

Tables

Figures