International Journal of Applied and Physical Sciences
volume 5 issue 1 pp. 28-35 doi: https://dx.doi.org/10.20469/ijaps.5.50004-1

Bidirectional Enhanced Selection Sort Algorithm Technique

Ramcis N. Vilchez*
Technological Institute of the Philippines (TIP), Quezon City, Philippines

Abstract: Sorting algorithm refers to the arranging of numerical or alphabetical or character data in statistical order (ascending or descending). Sorting algorithm plays a vital role in searching and the field of data science. Most of the sorting algorithms with $O(n^2)$ time complexity are very efficient for a small list of elements. However, for large data, these algorithms are very inefficient. This study presented a remedy for the noted deficiencies of $O(n^2)$ sort algorithm for large data. Among the $O(n^2)$ algorithms, selection sort was the subject of the study considering its simplicity. Although selection sort is regarded as the most straightforward algorithm, it is also considered the second worst algorithm in terms of time complexity for large data. Several enhancements were conducted to address the inefficiencies of selection sort. However, the procedures presented in all the enhancements can still lead to some unnecessary comparisons, and iterations that cause poor sorting performance. The modified selection sort algorithm utilizes a Bidirectional Enhanced Selection Sort Algorithm Technique to reduce the number of comparisons and iterations that causes sorting delays. The modified algorithm was tested using varied data to validate the performance. The result was compared with the other $O(n^2)$ algorithm. The results show that the modified algorithm has a significant run time complexity improvement compared with the other $O(n^2)$ algorithms. This study has a significant contribution to the field of data structures in computer science and the field of data science.

Keywords: *Sorting, selection sort, algorithm, bidirectional sorting*

Received: 13 November 2018; ***Accepted:*** 22 February 2019; ***Published:*** 08 March 2019

I. INTRODUCTION

A. *Background of the Study*

Sorting is significant in programming as it is in our daily life. I cannot imagine life without sorting, searching for a transcript of records for instance in a warehouse of a school registrar that existed for a century can be very difficult without sorting. The various applications of sorting will never be obsolete, even with the rapid development of technology, sorting is still very relevant and significant. The concept of data warehousing and data mining is something new and innovative, and yet these concepts are still dependent with sorting algorithm.

Sorting algorithm refers to the arranging of numerical or alphabetical or character data in statistical order (either in increasing order or decreasing order) or lexicographical order (alphabetical value like addressee key) [1].

Sorting algorithm performance varies on which type of data being sorted, not easier to say that which one algorithm is better than another. Here, the performance of the different algorithm is according to the data being sorted [2]. Examples of some common sorting algorithms are the exchange or bubble sort, the selection sort, the insertion sort, and the quick sort.

Among the sort algorithm, selection sort is the simplest and very straightforward. It resembles human instinct in arranging items in particular order. However, selection sort is considered the second worst algorithm in terms of time complexity [3, 4, 5].

The selection sort works by searching for the minimum value in the list and interchanging it with the first element. Then it looks for the second minimum value excluding the first element which was already found during the first pass and interchanging it with the second

*Correspondence concerning this article should be addressed to Ramcis N. Vilchez, Technological Institute of the Philippines (TIP), Quezon City, Philippines. E-mail: ramcis_vilchez@umindanao.edu.ph

element. In every step, the list is shrunk by one element at the beginning of the list. This processing is continued until the list becomes of size one when the list becomes trivially sorted.

In each step, to look for the maximum value, the selection sort starts from the beginning of the list. It starts assuming the first element to be the maximum and tests every element in the list whether the current maximum is the maximum. If it finds a greater value, it considers that value to be the new maximum.

B. *Problem Statement*

The selection sort can be the most popular sort algorithm because of its simple and straightforward steps that resemble human instinct in arranging items. However, the procedure involved in the selection sort algorithm causes the following identified problems:

1. Unnecessary comparisons and swapping that leads to huge running time.

2. Unnecessary iterations due to unnecessary comparisons and swapping.

C. *Objectives of the Study*

This study aims to modify the selection sort algorithm to improve the time complexity. Specifically, it seeks to do the following:

1. Eliminate the unnecessary comparisons, and iterations using bidirectional Enhanced Selection Sort Algorithm.

2. Compare the results of the modified selection sort with the classical selection sort and other enhanced selection sort algorithms using varied data to determine improvement in terms of execution time.

D. *Significance of the Study*

The result of this study can be utilized for all sorting applications. Further, the concept presented in this study will be a very significant contribution to the field of data structures in computer science.

E. *Scope and Delimitations*

This study focuses on finding a remedy on the identified problems of the selection sort particularly on run time complexity by modifying the selection sort algorithm. The modified algorithm will then be tested using varied data to validate the performance. The result will also be compared with the other available classical and modified selection sort algorithms to validate running time complexity.

II. THEORETICAL FRAMEWORK

A. *Review of Related Literature*

The inefficient performance of selection sort on huge data leads to the development of several enhancements to improve the runtime complexity. These enhancements have a significant improvement in the runtime complexity of the classical selection sort. However, the procedures presented in all these enhancements can still lead to some unnecessary comparisons, and iterations that cause poor sorting performance.

1) Modified double-ended selection sorting: The study of [6] of Haryana Institute of Technology, Haryana, introduces the idea of double selection sort. The sort starts from two elements and searches the entire list until it finds the minimum value and maximum value. The sort will exchange value of the first element with the minimum value and the last element with the maximum value. It will then select the second and the last element and searchers for the second minimum and maximum element. The process will continue until the list is sorted. This sorting works better than the classical selection sort. However, unnecessary iteration and comparison are still possible.

2) Improving the performance of selection sort using a modified double-ended selection sorting: The idea of [6] is also promising since it uses two elements for both smallest and largest elements in the list and compare each other and placing them in their respective places in the front and rear locations. The researcher claimed that about 25% to 35% improvement in terms of time complexity was noted.

3) Upgraded selection sort: The study of [7] upgraded the selection sort by searching the smallest and largest items simultaneously and placing them on their right locations. This study was able to improve the number of iterations of the classical selection from n-1 to n/2. However, the time complexity remains the same.

4) Improved selection sort algorithm: The concept presented in a study of [8, 9, 10] utilizes a queue to store the locations of all values that are the same as the maximum value. This idea is only useful if the given unsorted list is with duplications. However, if the list is already distinct, then the modified selection sort presented in this study is as bad as the classical selection sort for large data.

5) Minimizing the execution time of selection sort algorithm: The study of [11] on selection sort enhancement presented the concept of dividing the array into two by getting the mean after finding the smallest and largest elements. The elements that are smaller or equal to the mean are placed at the front. While all the elements larger than the mean are placed at the rear portion, this concept

has a significant improvement in the time complexity. The author claimed that for an average case scenario, the time complexity of the modified selection sort is now O(n) from O(n2) of that of the classical selection sort algorithm.

6) New approach for dynamic bubble sort improvement: The approach presented in the study of [12] utilizes a stack to store the previous largest element to eliminate the unnecessary comparisons in the classical bubble sort algorithm. The succeeding iterations begin the search of the largest from the location of the previous largest element and not from the beginning of the array. With this approach, significant improvement in terms of time complexity was noted. For an average case scenario, the time complexity is O(n2/4) compared to the classical bubble sort algorithm that has an O(n2).

7) Insertion sort with its enhancement: An insertion sort enhancement approach presented in the study of [13] uses a bidirectional technique. For the first iteration, the first and the last element of the array is compared. If the first element is bigger than the last element, then the two elements are swapped. The location of the element from the left end and the element from the right end of the array are stored in the variables which are increased (left end) and decreased (right end) as the algorithm progresses. In the second iteration, two adjacent elements from the left of the array are taken and are compared. Insertion of elements is done if required according to the order. Then the similar process is carried as in Insertion sort. This approach is more efficient than the classical insertion sort algorithm.

8) Enhanced insertion sort algorithm: Another insertion sort technique presented in the study conducted by [14] uses a bidirectional approach in sorting the list. Both sides of the array will be sorted accordingly depending on the sort order. If the algorithm sorts ascendingly, the small elements are inserted into the front portion. While the large elements are inserted in the rear portion, this approach has improved the time complexity of the insertion sort from O(n2) to O(n1.5) for an average case scenario.

9) Enhanced bidirectional selection algorithm: An enhancement of Selection sort algorithm by is called Enhanced Bidirectional Selection Sort. This algorithm will select two values, the smallest from the front and largest from the rear and placing them in their respective locations. The smallest will be placed in the first location while the largest in the last location of another array thus, reducing the number of passes by half the total number of elements as compared with classical selection sort. The said maximum and minimum will then be deleted from the original list thus reducing the comparison by the factor of two.

10) Bi-directional mid selection sort: The study of [15] called Bi-directional mid selection sort algorithm is based on bidirectional. However, in this enhancement of selection sort algorithm, it sorted the data by selecting (maximum and minimum) elements by starting to look from the middle to both sides in the selected list by reducing the size of the list from n to 2 with the decrement of two in size. It has the two loops outer and inner loop. The outer loop manages the size of the list to be processed for searching the maximum, and minimum number and the inner loop is for finding the smallest and various number from the list selected by the outer loop.

11) Both ended sorting algorithm: Another enhancement is both ended sorting algorithm by [16], which claimed to be faster than the bubble and other algorithms. This algorithm compares both ends of the grid from the right end as well as from the left end. This enhancement is based on the bubble sort algorithm that compares one element from the front end with one element of the rear end. If the front element is greater than the rear end, then it will swap the front element with the rear element. In the second iteration, two consecutive elements from the front end and rear end of the array are compared. Replacing elements is done if required according to the order. Here four variables are taken which stores the position of two rights elements and two left elements which are to be sorted. This process continues until the list is sorted.

12) Bidirectional selection sort: The study of [17], introduced the concept of bidirectional selection sort. Its main idea is that successive elements are selected on both sides of the array and are placed in their proper position. In this technique, the sorting is done in a single pass in two ways. That is to find the minimum element from the list and interchange it with the first element. At the same time, it will look for the maximum element and interchange it with the last element. Bidirectional Selection sort algorithm performs better than the classical since it reduces the number of swaps. However, there are still grey areas in this algorithm, since it cannot detect an already sorted list. It will continue to execute and finish the iterations even with the existence of an already sorted list. Thus, unnecessary comparisons, swaps, and iterations are still possible in this improved algorithm.

13) Optimized Selection Sort Algorithm (OSSA): Another Selection sort enhancement from the study of [18] called OSSA is also based on the bidirectional selection sort. However, instead of finishing the iterations from the beginning to the last element, the iteration not found on the same page ends if it reaches the middle of the array. This concept saves some iteration time as compared with

the classical selection sort and bidirectional selection sort.

B. Concept of the Study

The proposed modified selection sort algorithm will be utilizing a stack to store the previous maximums or minimums. The locations of the values are stored in the list instead of storing the actual values.

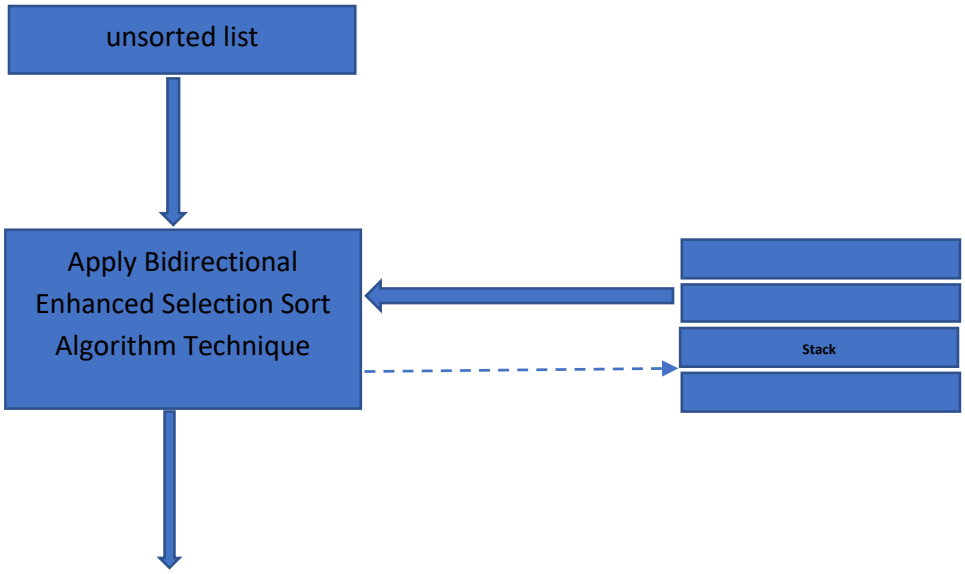

Fig. 1. Modified selection sort algorithm

III. OPERATIONAL FRAMEWORK

A. Methods

The goal of this study is to identify limitations or problems of selection sort algorithm and to find a remedy to these problems to improve the performance of selection sort. Let us first examine how selection sort works and determine its limitations or problems.

1) Classical selection sort algorithm: The classical selection sort algorithm below works by searching for the maximum value in the list and interchanging it with the last element. Then it looks for the second maximum value excluding the last element which was already found during the first pass and interchanging it with the second to the last element. In every step, the list is shrunk by one element at the end of the list. This processing is continued until the list becomes of size one when the list becomes trivially sorted.

In each step, to look for the maximum value, the selection sort starts from the beginning of the list. It starts assuming the first element to be the maximum and tests every element in the list whether the current maximum is really the maximum. If it finds a greater value, it considers that value to be the new maximum.

This procedure is very easy to comprehend but has a lot of flaws that make it the second worst sorting algorithm for large items.

2) Algorithm: Selection Sort (array[], length) Here L is the unsorted input list and length is the length of the array. After completion of the algorithm array will become sorted. Variable max keeps the location of the maximum value.

Step 1. Repeat steps 2 to 5 until length=1

Step 2. Set max=0

Step 3. Repeat for count=1 to length

If (L[count]>L[max])

Set max=count End if

Step 4. Interchange data at location length-1 and max

Step 5. Set length=length-1

The procedure involved in the selection sort algorithm as illustrated above causes the following identified problems:

1. Unnecessary comparisons and swapping that leads to huge running time.

2. Unnecessary iterations due to unnecessary comparisons and swapping.

3) Proposed enhancement of selection sort: To eliminate unnecessary comparisons, swaps, and iterations, a bidirectional Enhanced Selection Sort Algorithm will be used to memorize the location of the previous maximum (from left to right) and previous minimum (from right to left) when the new maximum and minimum are found. A stack is used to store the locations of the past or local

maximums and minimums, which can be used in later iterations. It is guaranteed that no value in the list is larger or smaller than the former maximum and minimum value before the location of the former maximum and former minimum. Thus, there is no need to go through this range. In the next iteration, it is now safe to start looking for the next maximum and minimum from the location of the current maximum and minimum. This concept saves searching time, and it works better than the other enhancements in the selection sort algorithm. This enhancement will solve problem number 1 in the enumerated identified problems above but cannot solve number 2 and 3.

To illustrate the proposed enhancement, for example, list 7, 15, 5, 11, 50, 10, 98, 67, 80, 19, 30 is to be sorted. In this list, the maximum is 98, and it will be interchanged with the last value of the list, which is 30. On the other side, starting from the location of the second to the last item of the list, the minimum is 5, and it will be interchanged with the first value of the list which is 7. If the classical selection sort approach is followed, the list will be like 7, 15, 5, 11, 50, 10, 30, 67, 80, 19, 98 after the first pass. But the fact that before finding 98, the maximum value was 50 should be noticed, and likewise, before finding 5, the minimum was 10. So, it is guaranteed that there is no value greater than 50 before the location of 50 and likewise, no value lesser than 10 before the location of 10 in the list. So instead of starting from the beginning of the list, the next pass can start from the location of 50 for the maximum and 10 for the minimum, removing some unnecessary searches. It is also observed that before finding 98 the maximum value was 50 and before finding 5, the minimum was 10. So, there is no value greater than 50 in the location range between 50 and the immediate past of the location of 98. Likewise, on finding the minimum, there is no value lesser than 10 in the location range between 10 and the immediate past of the location of 5.

Consequently, it is apparent that in the next iteration it is wastage of time to look for values greater than 50 and lesser than 10 before the current location of 98 and 5 respectively. Therefore, the next iteration can start from the current location of the value 98 for the maximum and 5 for the minimum, reducing unnecessary comparisons. And 50, the former maximum, can be safely placed at the immediate past location of 98 by interchanging with the current value 10. Same with 10 the former minimum can be safely placed at the immediate past location of 5. This strategy leads to the list having the content 5, 11, 7, 10, 15, 19, 30, 67, 80, 50, 98 after the first iteration and now it possesses more degree of sorting, compared to the list generated by the classical selection sort approach.

The second iteration finds the second largest item and looks for a larger value than 50, starting from 30. It finds 67 to be the maximum and consider 50 to be the former maximum. Then 80 is found to be the new maximum and 67 to be the new former maximum. On the other side in finding for the minimum, the second iteration finds the second smallest item and looks for a lesser value than 10, starting from the location of 7. It finds 7 to be the minimum and consider 10 to be the former maximum. By following the same strategy, after this iteration, the updated list is 5, 7, 11, 10, 15, 19, 30, 67, 50, 80, 98. In the third iteration, a larger value than 67 is looked for starting from the location of 50, which was the old location of the maximum 80 in the first iteration. Likewise, a smaller value than 10 is looked for starting at position number 11. After the third iteration, the list will be 5, 7, 10, 11, 15, 19, 30, 50, 67, 80, 98. After the 3rd iteration, the list is already on its sorted order and after the 4th iteration, there is no swap detected. Thus, the iteration will stop. The proposed enhanced algorithm will be utilizing a flag to detect an occurrence of a swap. If there was no swap detected during each pass the iteration will stop with the assumption that the list is already sorted. To further elaborate how the proposed enhancement work, consider the illustration below.

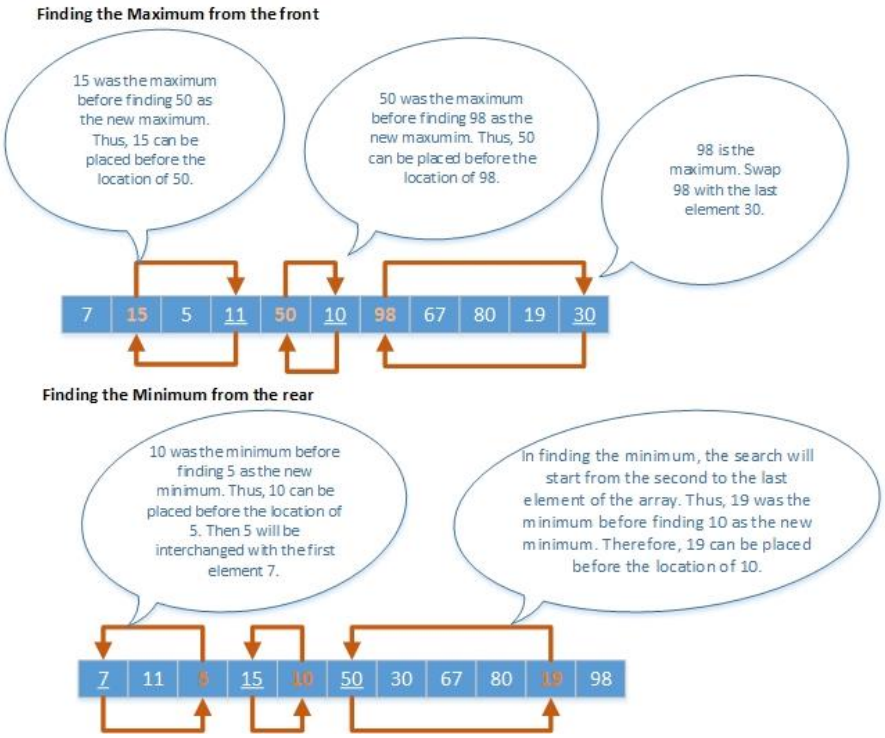

Fig. 2. Bidirectional enhanced selection sort algorithm

4) Modified Selection Sort Algorithm (MOSSA): Here L is the unsorted input list, and length/n is the length of an array. After completion of the algorithm, the array will become sorted. Variable max keeps the location of the current maximum, while variable min keeps the location of the current minimum.

1. Set Min=n-2
2. Repeat steps 3 to 21 while Min<>Max
3. Repeat steps 4 to 11 until length=1
4. if stack is empty push 0 in the stack
5. Pop stack and put in max
6. Set count=max+1
7. Repeat steps 8 to 9 while count<length
8.if(L[count]>L[max])
a. Push count-1 on stack
b. Interchange data at location count-1 and max
c. Set max=count
9. Set count=count+1
10. Interchange data at location length-1 and max
11. Set length=length-1
12. Set i= 0 to n
13. Repeat steps 14 to 21 while i<n
14. if stack is empty push 0 in the stack
15. Pop stack and put in Min
16. Set countmin=Min-1
17. Repeat steps 18 and 19 until countmin< i
18. if(L[countmin]<L[Min])
a. Push countmin+1 on stack
b. Interchange data at location countmin+1 and Min
d. Set min=countmin
19. Set countmin= countmin - 1
20. Interchange data at location i and min
21. Set i=i+1

IV. RESULTS

A. Sorting Performance Comparison

TABLE 1
THE NUMBER OF COMPARISONS FOR DIFFERENT SORT ALGORITHM USING A RANDOM DATA SET

Sort/Number of Elements	10	20	50	100	200
Selection Sort	45	190	1225	4950	19900
Insertion Sort	45	190	1391	5399	20473
Exchange Sort	55	210	1410	5335	20300
BESSA	41	110	495	341	1020

TABLE 2
THE NUMBER OF ITERATIONS FOR DIFFERENT SORT ALGORITHM USING A RANDOM DATA SET

Sort/Number of Elements	10	20	50	100	200
Selection Sort	9	19	49	99	199
Insertion Sort	9	19	49	99	199
Exchange Sort	9	19	49	99	199
BESSA	5	10	24	4	95

B. Time Complexity Comparison

TABLE 3
THE NUMBER OF ITERATIONS FOR DIFFERENT SORT ALGORITHM USING A RANDOM DATA SET

	Worst Case	Average Case	Best Case
Selection Sort	O(n2)	O(n2)	O(n2)
Insertion Sort	O(n2)	O(n2)	O(n)
Exchange Sort	O(n2)	O(n2)	O(n)
BESSA	O(log n)	O(n)	O(n2)/2

V. CONCLUSION AND RECOMMENDATIONS

A. Conclusion

Significant improvement with regards to the time complexity was noted based on the test results. The improvement can be attributed to the elimination of the identified problems from the procedure of classical selection sort. The reduction of unnecessary comparisons using the enhanced bidirectional selection sort technique plays a vital role in the significant improvement of the modified selection sort algorithm. Additionally, the utilization of a flag to determine the already sorted list in the early iteration significantly reduces the number of iterations. Lastly, the employment of a distinct function reduces the number of comparisons and iterations thus, decreases the sorting processing time from O(n2) to O(log n) for best time complexity.

B. Recommendations

The technique presented in this paper plays a vital role in the significant improvement of the modified selection sort algorithm. However, a further enhancement to simplify the complication of the modified algorithm is recommended to improve performance.

REFERENCES

[1] G. Franceschini and V. Geffert, "An in-place sorting with o (n log n) comparisons and o (n) moves," in 44th Annual IEEE Symposium on Foundations of Computer Science*, Cambridge, MA,*, 2003.

[2] Y. Han, "Deterministic sorting in o (n log log n) time and linear space." *Proceedings of the Thiry-fourth Annual ACM Symposium on Theory of Computing*, New York, NY, 2002.

[3] M. Khairullah, "Enhancing worst sorting algorithms," *International Journal of Advanced Science and Technology*, vol. 56, pp. 13–26, 2013.

[4] J. Lim, H. Gilbert, K. Han, J. T. Kim, and S. Kim, "Panelizing algorithms for free-form concrete panels considering esthetic surfaces," *International Journal of Technology and Engineering Studies*, vol. 1, no. 3, pp. 81–86, 2015. doi: https://doi.org/10.20469/ijtes.40003-3

[5] S. H. S. N. Ugtakhbayar, B. Usukhbayar and J. Nyamjav, "Detecting TCP based attacks using data mining algorithms," *International Journal of Technology and Engineering Studies*, vol. 2, no. 1, pp. 1–4, 2016. doi: https://doi.org/10.20469/ijtes.2.40001-1

[6] S. Lakra and Divya, "Improving the performance of selection sort using a modified double-ended selection sorting," *International Journal of Application or Innovation in Engineering & Management (IJAIEM)*, vol. 2, no. 5, pp. 364–370, 2013.

[7] T. C. S. Chand and R. Parveen, "Upgraded selection sort," *International Journal on Computer Science and Engineering*, vol. 3, no. 4, pp. 1633–1637, 2011.

[8] J. Hayfron-Acquah, O. Appiah, and K. River-

son, "Improved selection sort algorithm," *International Journal of Computer Applications*, vol. 110, no. 5, pp. 29–33, 2015. doi: https://doi.org/10.5120/19314-0774

[9] E. Uma, A. Kannan *et al.*, "Self-aware message validating algorithm for preventing XML based injection attacks," *International Journal of Technology and Engineering Studies*, vol. 2, no. 3, pp. 60–69, 2016. doi: https://doi.org/10.20469/ijtes.2.40001-3

[10] A. H. Al-Saeedi and O. Altun, "Binary Mean-Variance Mapping Optimization Algorithm (BMVMO)," *Journal of Applied and Physical Sciences*, vol. 2, no. 2, pp. 42–47, 2016. doi: https://doi.org/10.20474/japs-2.2.3

[11] M. Kumar, M. Malhotra, and D. Ahuja, "Minimizing the execution time of selection sort algorithm," *International Journal of Engineering and Computer Science*, vol. 6, no. 4, pp. 21–27. doi: https://doi.org/10.18535/ijecs/v6i4.37

[12] E. A. K. Thabit and F. Bahareth, "New approach for dynamic bubble sort improvement," *Research Notes in Information Science (RNIS)*, vol. 14, pp. 245–252, 2013.

[13] M. P. K. Chhatwani, "Insertion sort with its enhancement," *International Journal of Computer Science and Mobile Computing*, vol. 3, no. 3, pp. 801–806, 2014.

[14] A. S. Mohammed, Ş. E. Amrahov, and F. V. Çelebi, "Bidirectional conditional insertion sort algorithm; an efficient progress on the classical insertion sort," *Future Generation Computer Systems*, vol. 71, pp. 102–112, 2017. doi: https://doi.org/10.1016/j.future.2017.01.034

[15] M. F. Umar, E. U. Munir, S. A. Shad, and M. W. Nisar, "Enhancement of selection, bubble and insertion sorting algorithm," *Research Journal of Applied Sciences, Engineering and Technology*, vol. 8, no. 2, pp. 263–271, 2014.

[16] A. Brijwal, A. Goel, A. Papola, and J. Gupta, "Both ended sorting algorithm & performance comparison with existing algorithm," *International Journal of IT, Engineering and Applied Sciences Research (IJIEASR)*, vol. 3, no. 6, pp. 4–9, 2014.

[17] P. S. V. R. M. Patelia, S. D. Vyas and N. S. Patel, "An enhanced selection sort algorithm," *International Journal of Advanced Technology in Engineering and Science*, vol. 3, no. 1, pp. 153–157, 2015.

[18] S. Jadoon, S. F. Solehria, S. Rehman, and H. Jan, "Design and analysis of optimized selection sort algorithm," *International Journal of Electric & Computer Sciences (IJECS-IJENS)*, vol. 11, no. 01, pp. 16–22, 2011.