Enhanced Insertion Sort by Threshold Swapping

Basima Elshqeirat[1], Muhyidean Altarawneh[2], Ahmad Aloqaily[3]
Department of Computer Science, the University of Jordan, P.O Box 11942, Amman, Jordan[1, 2]
Computer Science Department, Maharishi International University, Lowa Fairfield, IA 52557, U.S.A[2]
Department of Computer Science and its applications, the Hashemite University, P.O. Box 150459, Zarqa 13115, Jordan[3]

***Abstract*—Sorting is an essential operation that takes place in arranging data in a specific order, such as ascending or descending with numeric and alphabetic data. There are various sorting algorithms for each situation. For applications that have incremental data and require an adaptive sorting algorithm, the insertion sort algorithm is the most suitable choice, because it can deal with each element without the need to sort the whole dataset. Moreover, the Insertion sort algorithm can be the most popular sorting algorithm because of its simple and straightforward steps. Hence, the insertion sort algorithm performance decreases when it comes to large datasets. In this paper, an algorithm is designed to empirically improve the performance of the insertion sort algorithm, especially for large datasets. The new proposed approach is stable, adaptive and very simple to translate into programming code. Moreover, this proposed solution can be easily modified to obtain in-place variations of such an algorithm by maintaining their main features. From our experimental results, it turns out that the proposed algorithm is very competitive with the classic insertion sort algorithm. After applying the proposed algorithm and comparing it with the classic insertion sort, the time taken to sort a specific dataset was reduced by 23%, regardless of the dataset's size. Furthermore, the performance of the enhanced algorithm will increase along with the size of the dataset. This algorithm does not require additional resources nor the need to sort the whole dataset every time a new element is added.**

***Keywords*—Sorting; design of algorithm; insertion sort; enhanced insertion sort; threshold swapping**

I. Introduction

Sorting is considered as one of the fundamental operations and extensively studied problems in computer science. It is one of the most frequent tasks needed mainly due to its direct applications in almost all areas of computing. The various applications of sorting will never be obsolete, even with the rapid development of technology, sorting is still very relevant and significant [1]. Formally any sorting algorithm will basically consist of finding a permutation or swapping of elements of a dataset (typically as an array) such that they are organized in an ascending (or descending) or lexicographical order (alphabetical value like addressee key). A large number of efficient sorting algorithms have been proposed over the last ten years with different features [2].

In this paper, we will consider the insertion sort (IS) algorithm, which is one of the popular and well-known sorting algorithms. It is simply building a sorted array or list by sorting elements one by one. The IS algorithm begins at the first element of the array and inserts each element encountered into its correct position (index), after determining and locating a suitable position. This process is repeated for the next element until it reaches the last element in the dataset. Fig. 1 illustrates a classical procedure of the insertion sort algorithm where A is an array of elements. The main side effect of the sorting procedure is overwriting the value stored immediately after the sorted sequence in the array.

The complexity of the insertion sort algorithm depends on the initial array. If the array is already sorted by examining each element, then the best case would be $O(n)$ where n is the array's size. However, the worst case would be $O(n^2)$, as each value has to be swapped through the whole dataset, which makes the complexity increase exponentially as the dataset size increases. The average case would be under $O(n^2)$, since most values will be sorted to the beginning of the dataset, which is highly expected in large datasets.

Note that the insertion sort algorithm is less efficient when it comes to huge datasets than advanced algorithms, such as heap sort, quick sort, or merge sort. The main insertion sort procedure has an iterative operation, which takes one element with each repetition and compares it with the other elements to find its correct place in the array. Sorting is typically done in-place, by iterating through the array and increasing the sorted array behind it [1].

The Insertion sort algorithm is the optimal algorithm when it comes to incremental, instantly, and dynamically initiated data, which is due to its adaptive behavior. This paper proposes an enhanced algorithm to reduce the execution time of the insertion sort algorithm by changing the behavior of the algorithm, more specifically on large datasets. This proposed algorithm called Enhanced Insertion Sort algorithm (EIS), which aims to enhance how the elements are relocated from the first part of the dataset, rather than waiting to find its correct position by comparing and swapping. Instead, a simple question is asked during the algorithm's execution; is the particular element less than the determined threshold? If yes, then the algorithm applies by traversing the elements that are under the threshold to find the correct position of this particular element. This algorithm will be explained in detail in Section III. The structure of the paper is as follows. Section II presents a brief of related works that are proposed to handle and improve the insertion sort algorithm. Section III describes the proposed EIS algorithm with an explanation of its complexity cost, Pseudo-code, implementation code and finally simple comparisons between EIS and other IS algorithms. Section IV shows the experimental results of our proposed EIS algorithm. Finally, the conclusion of the paper presented in Section V.

```
i ← 1
while i < length(A)
    j ← i
    while j > 0 and A[j-1] > A[1]
        swap A[1] and A[j-1]
        j ← j - 1
    end while
    i ← i + 1
end while
```

Fig. 1. Pseudo-Code of a Typical IS Algorithm.

II. Related Work

The IS algorithm is considered as one of the best and most flexible sorting methods despite its quadratic worst-case time complexity, mostly due to its stability, good performances, simplicity, in-place and online nature. It is a simple iterative sorting procedure that incrementally builds the final sorted array [2]. There were several suggestions to improve insertion sort and some of them were even implemented, as seen in [2-4]. Insertion sort can be simplified by using an external element, known as a sentinel value [5]. Bidirectional approaches were proposed in [6], where it consists of two steps, the first step compares both the first and last elements, and then swaps them if the first element is larger. The second step takes two adjacent elements from the beginning of the array and then compares them as well. Abbasi and Dahiya [7] proposed a bidirectional approach to minimize the shifting process, which supposes that there are two sorted parts on the left and right. This approach reduces the shifting process, rather than an element that may shift through the whole array. Patel, et al. [8] presented an approach of inserting elements from the middle of a dataset and applying a bidirectional sorting, using arrays as structured data. Paira, et al. [9] proposed an approach that applies a dual scan from both directions, which locates the position from both sides. Sodhi, et al. [10] presents a binary insertion sort that achieves a time complexity of $O(n^{1.585})$ for some average cases by reducing the number of comparisons. This approach starts with the middle element, which reduces the number of swaps needed. Then it determines the position of the suitable location for each element. Afterward, it chooses one direction, either left or right, then adds or appends it to another array. Khairullah [11] presented an approach that keeps track of both directions. It also starts from the middle element's location and compares it according to the element in the middle. Some approaches were not bidirectional, such as [12], which implements an algorithm to simply arrange a worst-case insertion sort that reverses the values.

However, bidirectional methods are efficient when compared with other classical insertion sort algorithms, but these kinds of approaches require the complete dataset to be sorted before knowing its size. In this case, these approaches cannot be implemented in applications that have incoming incremental data.

III. Methodology

The methodology section demonstrates a detailed explanation of the proposed EIS algorithm method in subsection III.A, then it presents an analysis of the algorithm's complexity in subsection III.B. Further, subsections III.C and III.D handling the details of Pseudo-code and the implementation code of the EIS algorithm. Finally, subsection III.E illustrates a simple comparison between the EIS and IS algorithms.

A. EIS Algorithm Method

In the EIS algorithm, the enhancement occurs when the algorithm behaves differently, more specifically when a value of the selected element is lower than a given threshold. The threshold is defined as the index of selected elements from the sorted part of the array, A, in the particular step during the EIS algorithm. Note that the Threshold= $\lfloor i/3 \rfloor$, where i is defined as the index of the particular element which is select to be sorted and $0<i\leq n$. Please note that if i element is selected now to be sort, then this means that all elements in the array A from A[0] to A[i-1], are fully sorted according to the original insertion sort algorithm behavior. Moreover, the threshold is determined by ⅓ of the dataset size, which changes dynamically as the algorithm sorts the elements one by one and i will increase by 1 in each iteration. The ratio, ⅓, of the dataset was chosen after executing several experiments, which concluded that it is the best case in terms of time complexity. It is clear that there is no specific way to determine the optimal ratio for the threshold unless you try out a bunch of different values and test the performance. This will be further discussed in section VI.

The functionality of the proposed algorithm is the same as the insertion sort process, but it asks a question before it starts comparing and swapping the selected i element in A[i] where the index=i; is the value of the i element being compared less than the value of the element in threshold index? If yes, then.

The EIS algorithm searches for the correct index to move the i element and place it. Note that EIS will start searching for the correct index for element i from the segment part (block) that contains elements that have values that are less than the value of the threshold index. When the suitable index is spotted, it swaps the selected element to the specific index and then shifts all other elements to the right. This operation reduces the number of comparisons and swapping of elements and this reduction will increase if the size of the array is also increased. In case that the value of the element is higher than the value of the threshold, then EIS behaves like the original insertion sort process and the original IS procedure will be done for this particular selected element in index i.

Fig. 2 demonstrates an illustration example of the procedure of the proposed Enhanced Insertion Sort (EIS). The threshold is dynamically changing based on the size of the traversed elements while the algorithm sorts the elements on by one. In each step, the algorithm examines whether the value of the selected element i is lower than the value of the threshold index or not. In steps 6 and 9, the algorithm begins to search beyond the threshold, and then it replaces the elements to a suitable index, and then shifted all the elements to the correct index. To illustrate that if you look at step 9, rather than

comparing the element in A[*i*]=1, where *i*=9, with all the other elements from A[0] to A[*i*-1], it only performs three comparisons with elements A[2] = 5, A[1] = 4 and A[0] = 3 wherein this particular step the threshold = 3. Note that the number of the comparisons in step 9 will be 9 if we use the original LS, but when using our proposed EIS, the number of the comparisons will reduce to 3, as shown in Table I.

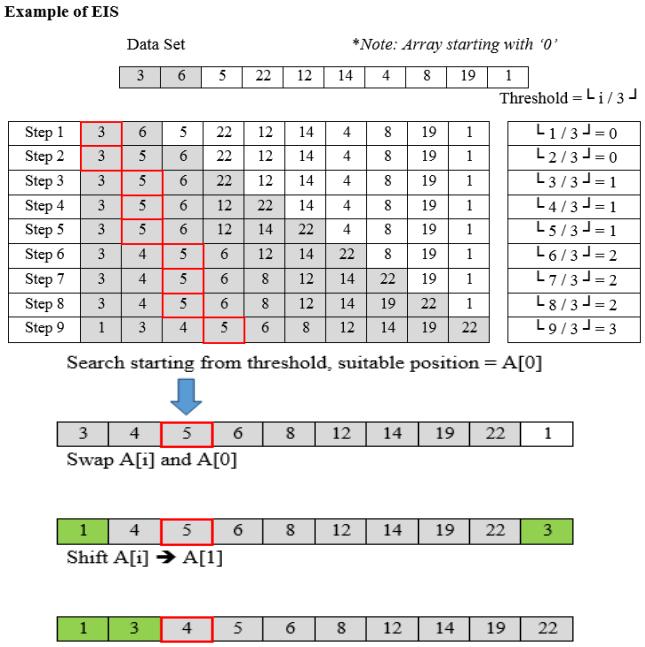

Fig. 2. The Procedure of the Enhanced Insertion Sort (EIS).

B. The Complexity of the EIS Algorithm

As shown in Fig. 3, the cost of the execution is reduced to $n^{2/3}$, as the algorithm relies heavily on the threshold procedure to reduce the search space. Given the following ➔ $2(n^{2/3}) + 3n^2 + 4n$. It will remain as $O(n^2)$. However, the results empirically show more efficient behavior. The Pseudo-code of the proposed Enhanced insertion sort algorithm is shown in Fig. 3.

C. Source Code of EIS Algorithm

Fig. 4 shows the java source code of the proposed Enhanced insertion sort algorithm. Furthermore, the full implementation of the EIS algorithm can be downloaded and run from the Github website:https://github.com/muhyidean/EnhancedInsertionSort-ThresholdSwapping.

D. Comparisons between the EIS and other IS Algorithms

Table I shows a detailed comparison between the insertion sort (IS) and our proposed algorithm (EIS) using the same dataset in Fig. 2 as an example. As shown in Table I, it is clear that the number of comparisons for the dataset example using the proposed EIS is better than using IS, where the number of comparisons for the IS algorithm is 45; while the number of comparisons for EIS is 34. As a particular example for specific iteration in step 9, when *i* = 9, it is clear that the number of the comparisons will be 9 if we use the original IS, but when using our proposed EIS, the number of the comparisons will reduce to 3, which is the one third (⅓) threshold value as shown in Table I.

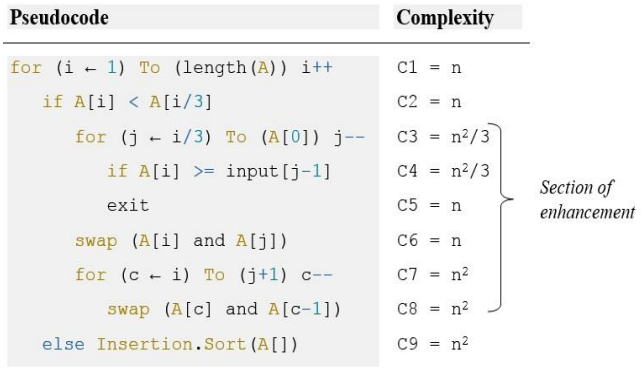

Fig. 3. Pseudo-Code of the EIS Algorithm.

```
1.  public static int[] EIS(int[] list){
2.      int t, j,k;
3.      int threshold  = 3;
4.      for (int i = 1; i < list.length; i++) {
            // If less than i/threshold
5.          if (list[i] < list[i/ threshold] ){
                // determine the Search position
6.              for( j = i/ threshold ; j > 0 ; j--){
7.                  if(list[i] >= list[j-1]){
8.                      break;
9.                  }
10.             }
                // Swap items
11.             t = list[j];
12.             list[j] = list[i];
13.             list[i] = t;
                // Shifting
14.             for(int c = i ; c > j+1 ; c--){
15.                 t = list[c];
16.                 list[c] = list[c-1];
17.                 list[c-1] = t;
18.             }
19.         }
20.         else{
21.             t = list[i];
22.             k = i -1;
23.             while (k>=0 && list[k]>t){
24.                 list[k+1]=list[k];
25.                 k=k-1;
26.             }
27.             list[k+1]= t;
28.         }
29.     }
30.     return list;
31. }
```

Fig. 4. Java Code of the EIS Algorithm.

TABLE I. COMPARISONS BETWEEN THE EIS AND IS ALGORITHMS

Dataset ➔	3	6	5	22	12	14	4	8	19	1

	Number of Comparisons based on the EIS algorithm	Number of Comparisons based on the IS algorithm	LIST AFTER SORTING	ETI Element To Insert	Number of Sorted items	VALUE OF (i)
	1	1	3,6	6	0	**i = 1**
	2	2	3,5,6	5	2	**i =2**
	3	3	3,5,6,22	22	3	**i =3**
	4	4	3,5,6,12,22	12	4	**i =4**
	5	5	3,5,6,12,14,22	14	5	**i =5**
	1	6	3,4,5,6,12,14,22	4	6	**i =6**
	7	7	3,4,5,6,8,12,14,22	8	7	**i =7**
	8	8	3,4,5,6,8,12,14,19,22	19	8	**i =8**
	3	9	1,3,4,5,6,8,12,14,19,22	1	9	**i =9**
Total	**34**	**45**	** The number of comparisons was reduced using EIS*			

IV. RESULTS

A. The Implementation Procedure

The proposed EIS algorithm was implemented using the Java programming language. In order to evaluate the performance of the proposed EIS algorithm. Three algorithms, the classical IS algorithm, the proposed EIS algorithm using ⅓ threshold, and ¼ threshold are utilized and the results are compared. The performance evaluations were performed on a computer machine with a 2.8 GHz Intel Core i5 processor with 4 GB 1600 MHz DDR3 memory on a windows platform. An experimental test has been done on empirical data (integer numbers) that are generated randomly using Java.

To verify that the same data is examined for each execution, a random dataset is generated and copied to three different arrays. Then the data is used to apply each algorithm and the time it takes (in milliseconds) to complete the sorting process was recorded. This is to assure that the algorithm's performance works for all types of data organizations and sizes. Twenty random datasets were utilized and the average execution times are reported. The sizes of the datasets that were utilized were 10000, 50000, 100000, and 500000.

B. Experimental Results and Discussion

Table II shows the overall performance results of the employed algorithms on different utilized datasets. As shown in Table II, the performance results, exposed by the classical IS algorithm, are stable on all utilized datasets. These results are expected since the computational complexity of the IS algorithm is practically the same. Further, the reported results in Table II emphasize that the proposed EIS algorithm using thresholds of ¼ and ⅓ reported enhanced performance results as compared to the classical IS algorithm. In fact, when the threshold equals ⅓ the results are superior. Consequently, a threshold of ⅓ states an appropriate threshold for employing the EIS algorithm.

The results are also showing, as the size of the dataset increases, the deviations of the performance results are also improved. Fig. 5 to 7 illustrates exceptional performance results of the proposed EIS especially when the threshold equals ⅓ and when the dataset's size is larger. As Fig. 5 Shows, the performance results of the EIS algorithm are much improved in terms of computational complexity time. Although, the complexity of the EIS algorithm, as reported in section III.B, states that the EIS has an $O(n^2)$ computational time, but the results empirically demonstrate better performance.

Fig. 6 and 7 demonstrate also similar performance results. When the size of a dataset is larger, the EIS algorithm performs better. Overall, the EIS algorithm, with varying datasets sizes and with a range of threshold values, performs empirically better than the classical IS algorithm. To further compare the performance of the employed algorithms, the average, maximum and minimum execution times for each dataset's size are reported and compared. As Table III shows the classical IS reported results are the lowest in terms of average execution time. When the threshold equals ¼, the EIS algorithm has a higher maximum value and a lower minimum value than using ⅓. This indicates that when the threshold equals ⅓ the reported results are more efficient. The lowest average execution times are highlighted to determine the most efficient performance. To conclude, The EIS algorithm, with a threshold of ⅓, demonstrates the best average performance results as highlighted in Table III. The improvements of the proposed EIS over the IS algorithms are also compared in terms of the average execution time and reported. Table IV shows that the proposed EIS algorithm outperforms the classical IS algorithm. The performance improvements of the proposed EIS algorithm in terms of execution time on average was 23%. The main reason for the significant improvement in performance is that the threshold procedure of the EIS algorithm reduces the number of comparisons and swapping needed to complete the sorting procedure.

TABLE II. THE PERFORMANCE OF THE EMPLOYED ALGORITHMS

Dataset size / Algorithms	Execution time (in milliseconds) on 20 Datasets that are randomly generated																			
10^4	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**
Ins	36	45	38	34	71	48	60	39	57	69	67	36	41	44	74	42	39	112	42	38
EIS ⅓	34	32	54	43	49	34	41	42	33	33	48	35	35	36	44	42	37	52	31	34
EIS ¼	35	48	43	30	53	36	31	34	38	42	49	37	33	32	35	42	53	55	33	35
$(10^4)*5$	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**
Ins	769	827	770	774	792	813	766	865	809	833	780	774	766	794	763	809	811	773	784	782
EIS ⅓	773	625	565	601	597	609	556	567	632	559	574	582	606	597	763	571	565	578	568	603
EIS ¼	516	748	742	749	532	534	512	756	576	600	742	529	541	742	783	775	506	740	781	725
10^5	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**
Ins	3035	3074	3032	3051	3031	3034	3021	3041	3062	3050	3026	3037	3038	3063	3033	3055	3040	3023	3041	3047
EIS ⅓	2139	2134	2131	2150	3014	2146	2125	2161	2141	2134	2959	2151	2966	2149	2127	2158	2136	2133	2150	2963
EIS ¼	2972	2033	2973	2033	2019	2964	2010	3000	2023	2094	2967	2997	2204	3037	2969	2037	2969	2015	2962	2963
$(10^5)*5$	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**
Ins	75242	77496	75306	75761	76568	76665	76140	76173	75406	75483	75114	75408	75092	75518	75378	75821	76094	75198	75792	76444
EIS ⅓	53348	54684	53315	53697	54092	75198	55537	59393	55480	53693	53173	53633	73721	55514	53463	58709	53628	73674	73531	59116
EIS ¼	73333	52147	50568	50678	55212	73765	73352	51036	73816	73268	73196	73345	73087	50763	54505	73449	73583	73273	50997	50684

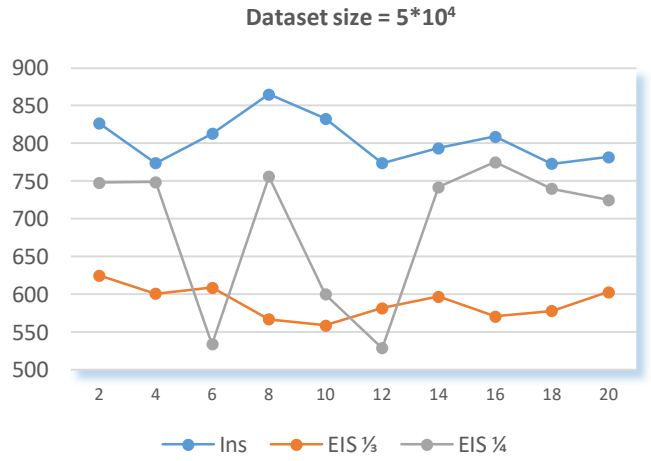

Fig. 5. The Performance of the Employed Algorithms on utilized Datasets of Size $5*10^4$.

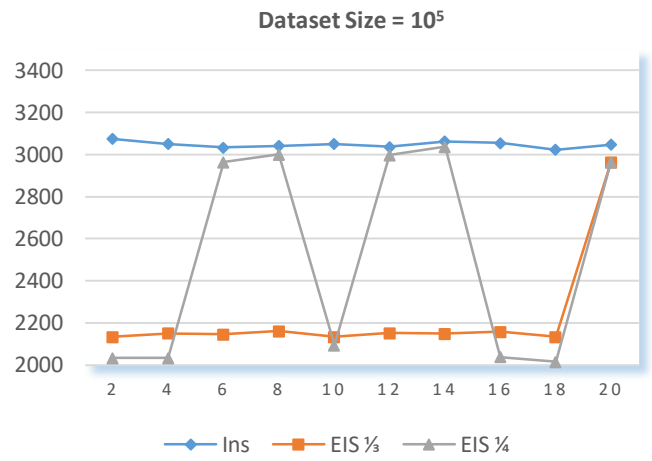

Fig. 6. The Performance of the Employed Algorithms on utilized Datasets of Size 10^5.

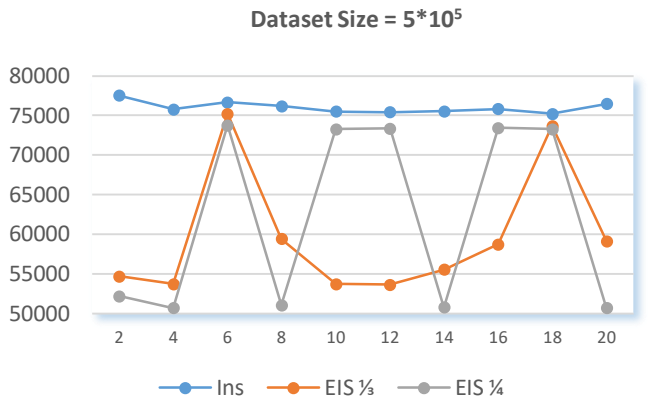

Fig. 7. The Performance of the Employed Algorithms on utilized Datasets of Size $5*10^5$.

TABLE III. AVERAGE, MAX AND MIN EXECUTION TIME OF THE IS AND EIS ALGORITHMS

Algorithms \ Dataset size	Overall Performance		
10^4	**Average**	**Max**	**Min**
IS	51.6	112	34
EIS ⅓	39.45	54	31
EIS ¼	39.7	55	30
$5*10^4$	**Average**	**Max**	**Min**
IS	792.7	865	763
EIS ⅓	604.55	763	556
EIS ¼	656.45	783	506
10^5	**Average**	**Max**	**Min**
IS	3041.7	3074	3021
EIS ⅓	2308.35	3014	2125
EIS ¼	2562.05	3037	2010
5*10^5	**Average**	**Max**	**Min**
IS	75804.95	77496	75092
EIS ⅓	58829.95	75198	53173
EIS ¼	63702.85	73816	50568

TABLE IV. THE OVERALL PERFORMANCE RESULTS OF THE EIS ALGORITHMS

Algorithms \ Dataset size	10^4	$5*10^4$	10^5	$5*10^5$
IS	51.6	792.7	3041.7	75804.95
EIS ⅓	39.45	604.55	2308.35	58829.95
EIS ¼	39.7	656.45	2562.05	63702.85

Improvement of EIS ⅓ over the IS algorithm

Improvement	**24%**	**24%**	**24%**	**22%**

C. Source Code of Implementation

Due to the space limitation, the source code of the proposed EIS algorithm is uploaded online to the GitHub website (https://github.com/muhyidean/EnhancedInsertionSort-ThresholdSwapping).

V. CONCLUSION

Insertion sort is the suitable sorting algorithm when it comes to incremental, instantly and dynamically initiated data. Yet, its complexity increases exponentially when the data size increases, making it inefficient. In this paper, an enhancement of the IS algorithm was proposed, named enchanted insertion sort, to improve the computational complexity of the IS algorithm by changing its behavior.

The proposed algorithm reduces the number of comparisons and swapping needed to complete the sorting procedure. After executing the algorithm and comparing it with the classical insertion sort algorithm, there was an improvement of 23% in terms of the execution time taken to complete the sorting process. The worst case of the complexity remains $O(n^2)$, but the reported results are empirically promising. The efficiency of the proposed EIS algorithm is attributable to the reduction of the number of comparisons during the sorting process.

REFERENCES

[1] M. Aliyu and P. Zirra, "A Comparative Analysis of Sorting Algorithms on Integer and Character Arrays," The International Jornal of Engineering and Science, pp. 25-30, 2013.

[2] A. Chadha, R. Misal, T. Mokashi, and A. Chadha, "Arc sort: Enhanced and time-efficient sorting algorithm," International Journal of Applied Information Systems vol. 7, pp. 31-36, 2014.

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, Introduction to algorithms: MIT press, 2009.

[4] S. Samee, A. Chitte, and Y. Tangde, "Analysis of Insertion Sort in Java," Advances in Computational Research, vol. 7, p. 182, 2015.

[5] O. Nevalainen, T. Raita, and H. Thimbleby, "An improved insert sort algorithm," Software: Practice and Experience, vol. 33, pp. 999-1001, 2003.

[6] M. P. K. Chhatwani, "Insertion sort with its enhancement," International Journal of Computer Science and Mobile Computing, vol. 3, pp. 801-806, 2014.

[7] H. Abbasi and M. Dahiya, "Services Marketing: Challenges and Strategies," International Journal on Recent and Innovation Trends in Computing and Communication, vol. 4, pp. 345-349, 2016.

[8] S. Patel, M. D. Singh, and C. Sharma, "Increasing Time Efficiency of Insertion Sort for the Worst Case Scenario," International Journal of Computer Applications, vol. 975, p. 8887.

[9] S. Paira, A. Agarwal, S. S. Alam, and S. Chandra, "Doubly Inserted Sort: A Partially Insertion Based Dual Scanned Sorting Algorithm," in Emerging Research in Computing, Information, Communication and Applications, ed: Springer, 2015, pp. 11-19.

[10] T. S. Sodhi, S. Kaur, and S. Kaur, "Enhanced insertion sort algorithm," International journal of Computer Applications, vol. 64, 2013.

[11] M. Khairullah, "Enhancing Worst Sorting Algorithms," International Journal of Advanced Science and Technology, vol. 56, pp. 13-26.

[12] P. S. Dutta, "An approach to improve the performance of insertion sort algorithm," International Journal of Computer Science & Engineering Technology (IJCSET), vol. 4, pp. 503-505, 2013.