Enhancing Worst Sorting Algorithms

Md. Khairullah

Department of Computer Science and Engineering
Shahjalal University of Science and Technology
Sylhet, Bangladesh
khairullah-cse@sust.edu

Abstract

Sorting is a commonly used operation in computer science. In addition to its main job, sorting is often required to facilitate some other operation such as searching, merging and normalization. There are many sorting algorithm that are being used in practical life as well as in computation. A sorting algorithm consists of comparison, swap, and assignment operations. Bubble sort, selection sort and insertion sort are algorithms which are easy to comprehend but have the worst time complexity of $O(n^2)$. In this paper enhancement of the selection sort and the bubble sort by eliminating some useless comparisons is presented. A stack is used to store the locations of the past or local maximums, which can be used in later iterations of these algorithms. The insertion sort is improved by reducing shift operations with the aid of a double sized temporary array. The new algorithms are discussed, analyzed, tested and executed for benchmarking along with representing results. A significant improvement of well above 200% is achieved.

Keywords: *Bubble sort, Selection sort, Insertion sort, Comparisons, Shifts, Stack, Complexity*

1. Introduction

One of the basic problems of computer science is *sorting* a list of items. It refers to the arranging of numerical or alphabetical or character data in statistical order (either in increasing order or decreasing order) or in lexicographical order (alphabetical value like addressee key) [1-3]. There are a number of solutions to this problem, known as sorting algorithms. There are several elementary and advanced sorting algorithms. Some sorting algorithms are simple and spontaneous, such as the bubble sort. Others, such as the quick sort are enormously complex, but produce super-fast results. Some sorting algorithm work on less number of elements, some are suitable for floating point numbers, some are good for specific range, some sorting algorithms are used for huge number of data, and some are used if the list has repeated values. Other factors to be considered in choosing a sorting algorithm include the programming effort, the number of words of main memory available, the size of disk or tape units and the extent to which the list is already ordered [4]. That means all sorting Algorithms are problem specific, meaning they work well on some specific problem and do not work well for all the problems. However, there is a direct correlation between the complexity of an algorithm and its relative effectiveness [5]. Many different sorting algorithms have been developed and improved to make sorting fast.

The classical *bubble sort, selection sort,* and *insertion sort* are very simple algorithms, often included in the text books to introduce algorithms and sorting, having runtime complexity of $O(n^2)$ making them impractical to use. The *selection sort* has a slight better running time than the simplest *bubble sort* algorithm and worse than the *insertion sort*. It

yields around 60% performance improvement over the bubble sort, but the insertion sort is over twice as fast as the bubble sort and is just as easy to implement as the selection sort [6]. Though other better techniques such as divide and conquer [7] exist, still there are scopes for fine-tuning these algorithms. Philosophy of these simple algorithms most closely matches human intuition [8]. They can be classified as an in-place [9] and a comparison sort [10].

In the classical selection sort algorithm to sort a list with *n* data elements *n-1* passes are carried out. Each pass finds out the largest or the smallest data item. To do this the entire list is checked form the beginning until the portion of the list, which was sorted in the earlier passes. But, instead of looking from the start of the list in each pass, some information regarding the locations of the local maximum or minimum gathered and stored in the earlier iterations can be used to omit some useless search operations and the overall runtime can be reduced. The enhanced selection sort algorithm is based on this idea. The bubble sort also places the largest element in the proper location in each pass. As the bubble sort and selection sort are closely analogous, the enhancement of the bubble sort is done with the same method. In the classical insertion sort a sorted portion is maintained and in each pass of the algorithm a data item from the unsorted portion is inserted into the sorted portion from a certain side such that with the additional item it remains sorted. Considering just one side to insert leads many shift operations, which can be reduced if both sides of the sorted list is considered to insert a data item. The enhanced insertion sort incorporates this strategy.

This paper is organized in the following order. In Section 2 some previous works relating to improvement of these sorting algorithms are discussed briefly. Section 3 introduces the classical versions of each algorithm followed by detailed discussion and analysis of the proposed enhanced algorithms. Section 5 summarizes and concludes this paper after discussion of the results in Section 4 that shows superiority of the new algorithms.

2. Related Works

A bidirectional variant of selection sort, sometimes called cocktail sort or shaker sort or dual selection sort is an algorithm, which finds both the minimum and maximum values in the list in every pass [11-14]. Selection sort is enhanced by reduction of swap operations in [15]. Bingo sort takes advantage of having many duplicates in a list. After finding the largest item, another pass is performed to move any other items equal to the maximum item to their final place [16]. Exact sort is another new variation, which locates the elements to their sorted positions at once [17]. To find the exact positions of elements it counts the smaller elements than the element, which is indented to be located. It changes elements positions just once, to directly their destination. It is highly advantageous if changing positions of elements is costly in a system. It makes too many comparisons to find positions of elements and thus is highly disadvantageous if comparing two elements is costly, which is the case in common.

Other than the cocktail sort, another bi-directional approach is presented in [15]. A new approach is presented in [18] that works on the principle that rather than swapping two variables using third variable, a shift and replace procedure should be followed, which takes less time as compared to swapping. In [19] Oyelami's Sort is presented that combines the techniques of bidirectional bubble sort with a modified diminishing increment sorting.

The inner loop of insert sort can he simplified by using a sentinel value as discussed in [20]. A bidirectional approach like the binary insertion sort is presented in [21], which achieves time complexity of $O(n^{1.585})$ for some average cases. When people run insertion sort in the physical world, they leave gaps between items to accelerate insertions. Gaps help in computers as well. This idea of gapped insertion sort discussed in [22] has insertion times of O(log n) with high probability, yielding a total running time of O(n log n) with high

probability. An end-to-end bi-directional sorting algorithm is proposed to address the shortcomings of the bubble sort, selection sort, and insertions sort algorithms [23].

3. The Proposed Enhanced Sorting Algorithms

In the subsequent discussion, sorting is assumed to be in the ascending order for standardization. The classical algorithms are briefly described before the discussion of the proposed algorithms for reference and comparison.

3.1. Selection Sort

3.1.1. Classical Selection Sort Algorithm: The classical selection sort algorithm looks for the maximum value in a list and interchanges the last element with it. Then it looks for the second maximum value in the list. To do this it looks for the maximum value in the list excluding the last element, which was found in the previous step. It interchanges this value with the last element but one. In every step the list is shrunk by one element at the end of the list. These 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 classical 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. The classical selection sort algorithm is listed below.

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

1. Repeat steps 2 to 5 until **length=1**
2. Set **max=0**
3. Repeat for **count=1** to **length**
   If (**a[count]>a[max]**)
   Set **max=count**
   End if
4. Interchange data at location **length-1** and **max**
5. Set **length=length-1**

3.1.2. Concept of Enhances Selection Sort (ESSA): The steps involved in the classical selection sort algorithm lead to huge unnecessary comparisons and large running time. But the location of the previous maximum can be memorized when a new maximum is found and it can be exploited later on. It is guaranteed that no value in the list is larger than the former maximum value before the location of the former maximum. Next iteration can start from the location of the former maximum while looking for the maximum. However, this approach can be further improved. It is also guaranteed that in the range between the former maximum and the just found current maximum no value is greater than the former maximum. Therefore, it is also wastage of time to look for a value greater than the former maximum in this range. Then, in the next iteration it is safe for to look for the maximum from the location of the current maximum and can put the former maximum element at the location just before the current maximum value by interchanging appropriately to minimize the search space.

One important thing to note here that in this process multiple local maximums can be discovered in a single pass and all of them must be remembered to use in later passes of the algorithm. A stack can be used to accomplish this. When a new maximum value is found, there should be an interchange between the old maximum value and the value located just before the current maximum value and this new location of the old maximum value should be pushed on the stack. When the end of the list is reached, the top item in the stack is considered to be the location of the former maximum value. Then the latest maximum value is swapped with the last item and the location of the current maximum value is considered as the starting point in the next iteration to look for the maximum. The next pass starts by assigning the value at the location stored at top in the stack to the current maximum. The local maximums stored in the stack as indices are sorted in ascending order according to their discovery time. The most recently discovered local maximum is always the largest and the initial local maximum is the smallest among them. Figure-1 illustrates the operations and the relationships among them in the proposed algorithm.

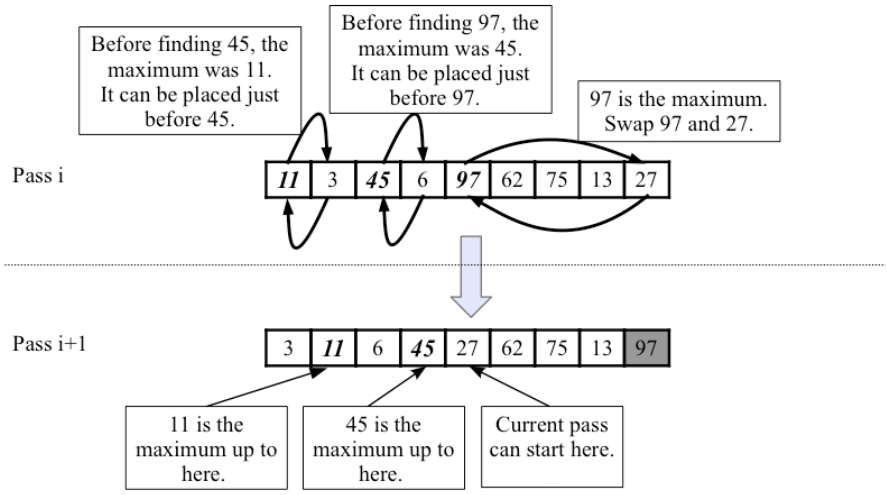

Figure 1. Concept of ESSA is to Memorize the Locations of the Past Maximum and Start from there in the Next Pass

3.1.3. Example: Let the list 11, 3, 45, 6, 97, 62, 75, 13, 27 is to be sorted. In this list the maximum is 97 and it will be interchanged with the last value of the list, which is 27. If the classical selection sort approach is followed, the list will be like 11, 3, 45, 6, **27**, 62, 75, 13, **97** after the first pass. But the fact that before finding 97, the maximum value was 45 should be noticed. So it is guaranteed that there is no value greater than 45 before the location of 45 in the list. So instead of starting from the beginning of the list, the next pass can start from the location of 45, removing some unnecessary searches. It is also observed that before finding 97 the maximum value was 45. So, there is no value greater than 45 in the location range between 45 and the immediate past of the location of 97. Consequently, it is apparent that in the next iteration it is wastage of time to look for values greater than 45 before the current location of 97. Therefore, the next iteration can start from the current location of the value 97, reducing unnecessary comparisons. And 45, the former maximum, can be safely placed at the

immediate past location of 97 by interchanging with the current value 6. This strategy leads to the list having the content 3, 11, 6, 45, 27, 62, 75, 13, 97 after the first iteration and now it possess 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 larger value than 45, starting from 27. It finds 62 to be the maximum and consider 45 to be the former maximum. Then, 75 is found to be the new maximum and 62 to be the new former maximum. By following the same strategy, after this iteration the updated list is 3, 11, 6, 27, 45, 62, 13, 75, 97. In the third iteration, a larger value than 62 is looked for starting from the location of 13, which was the old location of the maximum 75 in the second iteration. After the third iteration the list will be 3, 11, 6, 27, 45, 13, 62, 75, 97. It is clearly seen that in the next iteration, the search should start from the location of 13 because there is no value greater than 45 before the location of 45. The benefit of memorizing the location of the former maximum value when a new maximum value is found is obvious.

Table-1 represents the content of the list and the stack in each pass or iteration of the outer loop of the above example with ESSA. Notice that the locations of the values are stored in the list instead of storing the actual values. A 0 based indexing is assumed. The values to be interchanged are shown in bold font and the former or local maximums are shown with an underline. The portion of the list sorted so far is represented by italic font.

Table 1. Example of ESSA Working Procedure for a Random List

Iteration	List	Stack	Max.	Former Max. Value	Starting point of search
0	11, 3, 45, 6, 97, 62, 75, 13, 27	-	-	-	1
1	3, 11, 6, 45, **27**, 62, 75, 13, ***97***	1, 3	97	45	4
2	3, 11, 6, 27, 45, 62, **13**, ***75***, *97*	1, 4, 5	75	62	6
3	3, 11, 6, 27, 45, **13**, ***62***, *75*, *97*	1, 4	62	45	5
4	3, 11, 6, 27, **13**, ***45***, *62*, *75*, *97*	1	45	11	2
5	3, 6, 11, **13**, ***27***, *45*, *62*, *75*, *97*	2	27	13	3
6	3, 6, 11, *13*, *27*, *45*, *62*, *75*, *97*	2	13	11	3
7	3, 6, *11*, *13*, *27*, *45*, *62*, *75*, *97*	-	11	-	1
8	3, *6*, *11*, *13*, *27*, *45*, *62*, *75*, *97*	0	6	-	1

3.1.4. ESSA pseudo-code:

Algorithm: *ESSA*(***a*[]**, ***length***)
Here ***a*** is the unsorted input list and **length** is the length of array. After completion of the algorithm array will become sorted. Variable **max** keeps the location of the current maximum.

1. Repeat steps 2 to 9 until **length=1**
2. If stack is empty
   Push 0 in the stack
   End if
3. Pop stack and put in **max**
4. Set **count=max+1**
5. Repeat steps 6 and 7 until **count<length**
6. If (**a[count]>a[max]**)

a. Push **count-1** on stack
b. Interchange data at location **count-1** and **max**
c. Set **max=count**

End if

7. Set **count=count+1**
8. Interchange data at location **length-1** and **max**
9. Set **length=length-1**

3.1.5. Analysis:

a) Time complexity*:* Let the input list consists of n items. Like other sorting algorithms number of comparisons is used as a measure of computations required for sorting. It is observed in the above algorithm that the outer loop (statement 1) is always executed but the inner loop (statement 5) is problem instance dependent. So the complexity of this algorithm is not deterministic and three special situations need to be dealt for complexity analysis.

Best case**:** in every pass statement 5 will be executed just once and hence the comparison in statement 6 will be accomplished also once and hence, T(n) = n-1 = O(n). The best-case complexity of the classical selection sort is $O(n^2)$.

Worst case**:** in the iteration no *k*, when *k-1* items are already sorted and *n-k+1* items are still unsorted, *n-k* comparisons are required to find the largest element. Thus in this case T(n) = (n-1) + (n-2) +(n-3) + ……. + 3 + 2 + 1 = n(n-1)/2 = $(n^2-n)/2 = O(n^2)$.

Average case: probabilistically an item is greater than half of the items in the list and smaller than the other half in the list. So, in iteration *k*, when *k-1* items are already sorted and *n-k+1* items are still unsorted, (*n-k*)/2 comparisons are required to find the current maximum. This leads to T(n) = ((n-1) + (n-2) + (n-3) + ……. + 3 + 2 + 1)/2 = n(n-1)/4 = $(n^2-n)/4 = O(n^2)$.

b) Space complexity*:* It is obvious that the enhanced selection sort algorithm uses a stack. In the worst case, the size of the stack can be as large as the size of the data array to be sorted. So the space complexity of the proposed algorithm is O(n). But the classical selection sort algorithm does not use such stack. Hence, the enhanced algorithm's memory requirement is roughly double of that of the classical algorithm. But classical computers always biases for increased performance with the cost of some extra memory in case of time-space tradeoff scenarios [24].

3.2. Bubble Sort

3.2.1. Classical bubble sort algorithm (EBSA): The bubble sort works by iterating down an array to be sorted from the first element to the last, comparing each pair of elements and switching their positions if necessary. This ensures the placement of the largest item in its proper location at the end of the list. This process is repeated as many times as necessary, until no swaps are needed, which indicates that the list is sorted. Since the worst case scenario is that the array is in reverse order, and that the first element in sorted array is the last element in the starting array, the most exchanges that will be necessary is equal to the length of the array.

Algorithm: *Bubble Sort* (***a*[]**, ***length***)

Here ***a*** is the unsorted input list and **length** is the length of array. After completion of the algorithm array will become sorted. Variable **max** keeps the location of the maximum value.

1. Repeat step 2 for **i = 0 to length-2**
2. Repeat step 3 for **j= 0 to length-i-2**
3. If (**a[j]>a[j+1]**)
   Interchange a[j] and a[j+1]
   End if

3.2.2. Concept of Enhanced Bubble Sort: The enhanced bubble sort exploits exactly the same idea of the enhanced selection sort algorithm presented earlier. In the classical bubble sort, in a single iteration a data item is shifted to the end until a larger item is found and in this case no swap operation takes place and this indicates that the element located just before the larger item is the largest till that location and it is guaranteed no data item beyond this can reach the end in the next pass. So the next pass can start from the location where no swap operation occurs. Figure 2 shows the operations and the relationships among them in the proposed algorithm.

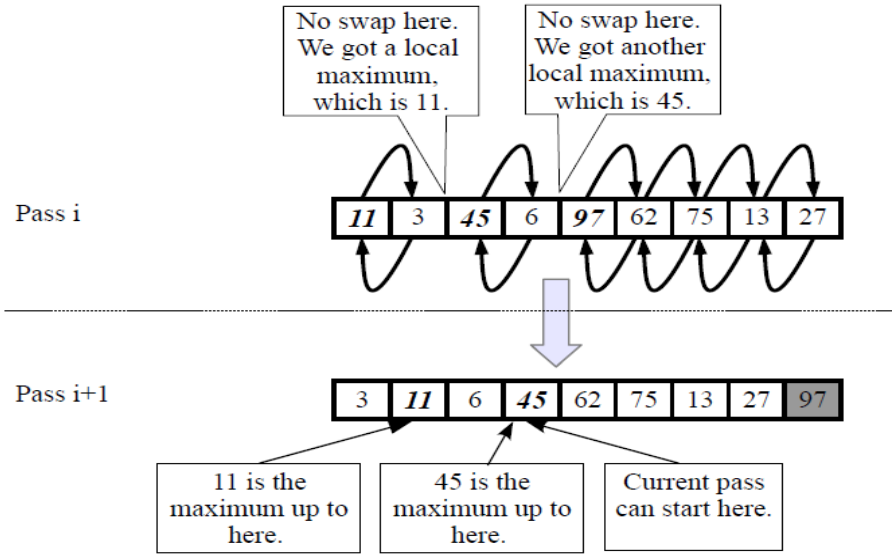

Figure 2. Concept of EBSA is to Memorize the Locations of the Local Maximum and Start from there in the Next Pass

3.2.3. Example: The same input list is used here to illustrate the enhanced bubble sort algorithm in Table 2. Found local maximums are marked with underlines. The portion that is sorted till the current pass is italicized and the item just placed in its proper place is marked with a bold font. The bubble sort has no direct specification about maximum values and also omitted in this example.

Table 2. Example of Enhanced Bubble Sort Working Procedure for a Random List

Iteration	List	Stack	Starting point of search
0	11, 3, 45, 6, 97, 62, 75, 13, 27	-	0
1	3, 11, 6, 45, 62, 75, 13, 27, ***97***	1, 3	3
2	3, 11, 6, 45, 62, 13, 27, ***75***, *97*	1, 3, 4	4
3	3, 11, 6, 45, 13, 27, ***62***, *75*, *97*	1, 3	3
4	3, 11, 6, 13, 27, ***45***, *62*, *75*, *97*	1	1
5	3, 6, 11, 13, ***27***, *45*, *62*, *75*, *97*	2, 3	3
6	3, 6, 11, ***13***, *27*, *45*, *62*, *75*, *97*	2	2
7	3, 6, ***11***, *13*, *27*, *45*, *62*, *75*, *97*	-	0
8	3, ***6***, *11*, *13*, *27*, *45*, *62*, *75*, *97*	0	0

3.2.4. Pseudo-code of EBSA:

Algorithm: *EBSA* (***a*[]**, ***length***)
Here ***a*** is the unsorted input list and **length** is the length of array. After completion of the algorithm array will become sorted.

1. Set **top = -1**
2. Repeat steps 3 to 5 for **i = 0 to n-2**
3. If (**top<0**)
   Push 0 on stack
   End if
4. Pop stack and put in **val**
5. Repeat step 6 for **j = val to n-i-2**
6. If (**a[j]>a[j+1]**)
   Interchange **a[j]** and **a[j+1]**
   Else
   Push **j** on stack
   End if

3.2.5. Analysis: The same analysis for the enhanced selection sort algorithm is applicable for the enhanced bubble sort algorithm. That is the best-case complexity is O(n) instead $O(n^2)$ of the classical bubble sort algorithm. The average case and the worst-case complexity remains $O(n^2)$, the same as the classical version.

Again here, the stack can grow as large as the array to be sorted. So the space complexity is O(n).

3.3. Insertion Sort

3.3.1. Classical Insertion Sort Algorithm: An insertion sort works by separating an array into two sections, a sorted section and an unsorted section. Initially the entire array is unsorted. The sorted section is then considered to be empty. The algorithm considers the elements in the unsorted portion one at a time, inserting each item in its suitable place among those already considered (keeping them sorted). To insert an item in a certain location all

other items up to that location needs to be shifted to the right. However, memory writes are costlier than memory read and many write operations are required only to place a single item in its proper place.

The proper location of a new item in the sorted portion can be searched either by a linear search or by a binary search. Certainly, the binary insertion sort, the version using binary search, has better performance than the other one. Anyway, for simplicity the linear search version of the insertion sort is listed below.

Algorithm: *Insertion Sort* (***a*[]**, ***length***)
Here ***a*** is the unsorted input list and **length** is the length of array. After completion of the algorithm array will become sorted.

1. Repeat steps 2 to 5 for **i=1 to n-1**
2. Set **temp = a[i]**
3. Set **j=i-1**
4. Repeat steps 4a and 4b while (**temp<a[j] and j>=0**)
    4a. Set **a[j+1] = a[j]**
    4b. Set **j = j-1**
5. Set **a[j+1] = temp**

3.3.2 Concept of Enhanced Insertion Sort Algorithm (EISA): As mentioned earlier, many costly write operations are necessary to insert a single item in the classical insertion sort. However, a linear array has two sides and both of them can be considered to insert an element in it. It is natural that there will be unequal number of required shifts to insert an item along the left or the right side. So, it would be cost effective in terms of memory writes to insert an item along the side which demands less number of shifts. But this has a major drawback. This cannot be implemented in place. That is the same array containing the list to be sorted cannot be used to accomplish this strategy. As data can be shifted left or right, there must be sufficient space in the left to hold the shifted data. This requires equal number of cells of the original array both in the left and the right sides. A temporary array with a double length of the original array can come to aid in this situation. The first item of the original array should be copied in the middle of the temporary array. Subsequent items would be either appended to the left or to the right or to be inserted along the left side or the right side of the temporary array according to the need.

It is guaranteed that the temporary list is capable to hold the whole sorted array in all situations. In the worst cases, the sorted array will be entirely in the right or entirely in the left from the middle position of the temporary array. However, the partial content of the temporary array that represents the sorted array must be copied back to the original array at the end to give output to the user. Therefore starting and ending locations of the sorted array in the temporary list must be maintained and updated in each pass. Figure 3 shows the working principle of the proposed algorithm for two consecutive iterations.

3.3.3. Example: Again the previous input list is used here to describe the enhanced insertion sort in table-3. The enhanced sort is not in place. A temporary array is used. Initially the list is empty and the first item is copied in the middle. Then each element of the original array is considered one after another to insert into the temporary list. Then number of shifts required from the left side and from the right side is computed. The side with the minimum shifts is chosen. For a tie situation the left side is chosen to break it. Data item just inserted is shown in bold in the temporary list.

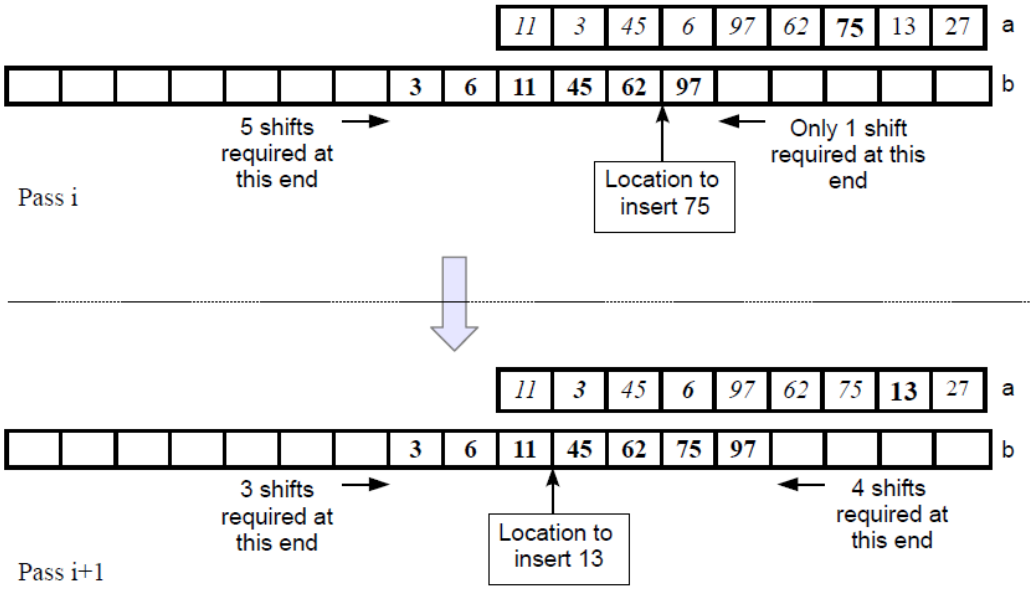

Figure 3. Concept of EISA is to insert into the Side giving Minimum Data Shifts

Table 3: Example of Enhanced Insertion Sort Working Procedure for a Random List

i	Data	Temporary List	Shifts		Loc
			L	R	
0	11	-, -, -, -, -, -, -, -, -, -, -, -, -, -, -, -, -	0	0	9
1	3	-, -, -, -, -, -, -, -, -, **11**, -, -, -, -, -, -, -, -	0	1	8
2	45	-, -, -, -, -, -, -, -, **3**, 11, -, -, -, -, -, -, -, -	2	0	10
3	6	-, -, -, -, -, -, -, -, 3, 11, **45**, -, -, -, -, -, -, -	1	2	8
4	97	-, -, -, -, -, -, -, 3, **6**, 11, 45, -, -, -, -, -, -, -	4	0	11
5	62	-, -, -, -, -, -, -, 3, 6, 11, 45, **97**, -, -, -, -, -, -	4	1	11
6	75	-, -, -, -, -, -, -, 3, 6, 11, 45, **62**, 97, -, -, -, -, -	5	1	12
7	13	-, -, -, -, -, -, -, 3, 6, 11, 45, 62, **75**, 97, -, -, -, -	3	4	9
8	27	-, -, -, -, -, -, 3, 6, 11, **13**, 45, 62, 75, 97, -, -, -, -	4	4	9
-	-	-, -, -, -, -, 3, 6, 11, 13, **27**, 45, 62, 75, 97, -, -, -, -	-	-	-

3.3.4. Pseudo-code of EISA:

Algorithm: *EISA* (***a*[]**, ***length***)
Here ***a*** is the unsorted input list and **length** is the length of array and b is a temporary array of size 2*length. After completion of the algorithm array will become sorted.

1. Set **left = length**
2. Set **right = length**
3. Set **b[left] = a[0]**
4. Repeat steps 5 to 9 for **i = 1 to length-1**
5. If (**a[i]>=b[right]**)
   5a. Set **right = right+1**

5b. Set **b[right]=a[i]**
5c. Go to step 4
End if
6. If(**a[i]<=b[left]**)
5a. Set **left = left-1**
5b. Set **b[left] = a[i];**
5c. Go to step 4
End if
7. **Set loc = right**
8. Repeat while (**a[i]<b[loc]**)
Set **loc = loc-1**;
9. If (**right-loc<loc-left**)
9a. Set **j=right+1**
9b. Repeat steps 9bx and 9by while (**j>loc+1**)
9bx. Set **b[j]=b[j-1]**
9by. Set **j=j-1**
9c. Set **right=right+1**
9d. Set **b[loc+1]=a[i]**

Else
9a. Set **j=left-1**
9b. Repeat step 9bx and 9by while (**j<loc**)
9bx. Set **b[j]=b[j+1]**
9by. Set **j=j+1**
9c. Set **left = left-1**
9d. Set **b[loc] = a[i]**
End if
10. Repeat steps 10a and 10b for **i = 0 to n-1**
10a. Set **a[i]=b[left]**
10b. Set **left = left+1**

3.3.5. Analysis: The enhancement of the proposed algorithm is by reducing shift operation, which does not necessarily reduce any comparisons. Hence, the time complexity of the enhanced insertion sort is exactly same as the classical version, which are O(n), $O(n^2)$, and $O(n^2)$ for the best, average, and the worst case respectively.

Here, a temporary array of double length of the list to be sorted is used. So the space complexity is 2n or O(n).

4. Results and Discussions

Figure 4 illustrates the comparison of performance of the enhanced and classical version along with the cocktail sort or shaker sort or the so-called dual selection sort in terms of the time required to sort a list with random data, which represents the average case. The comparison with the insertion sort shows the degree of improvement by the enhancement of the selection sort. The bingo sort and exact sort are avoided in this discussion as they are limited to some special situations and may perform worse than the classical selection sort in other contexts. The measurements are taken on a 2.4 GHz Intel Core 2 Duo processor with 2 GB 1067 MHz DDR3 memory machine with OS X version 10.6.8 platform. It depicts the

improved performance in terms of execution time. It shows a performance improvement of around 220-230%for random or reversely sorted list of any size to be sorted.

Similarly, the same strategy of memorizing the locations of the local maximum to reduce useless comparisons can be applied to the classical bubble sort algorithm. Consequently, the sorting time is roughly halved, as clearly seen in Figure-5. No other algorithm is compared in this context as bubble sort is the worst among the discussed trio and only the enhancement with respect to the classical algorithm is the main concern.

Figure 6 represents the performance gain by the claimed improvement. Both the linear search and binary search versions are included in the comparison. The binary insertion sort uses the least time to find the proper location of the data item to be inserted and spends most of the time to shift other data items. Hence, the enhancement effect of the proposed algorithm on the binary insertion sort is larger than the linear search version, which is evident from the result. Moreover, the enhancement of the linear search version makes it almost equal efficient as the binary insertion sort.

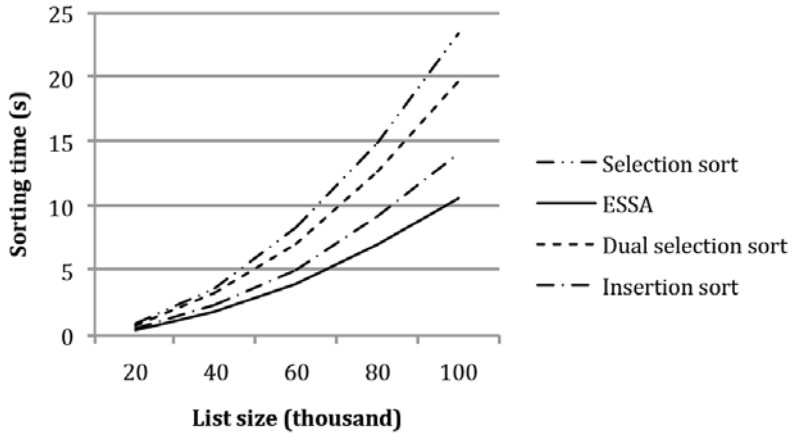

Figure 4. Comparison of Performance with Respect to Sorting Time for a Random List

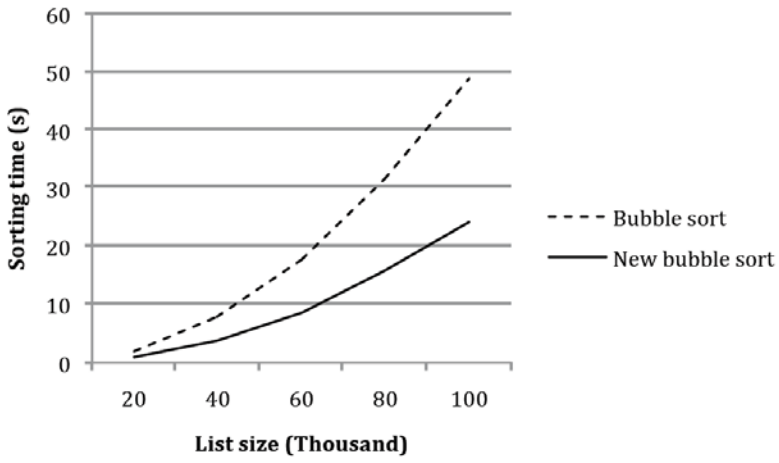

Figure 5. Improving the Bubble Sort with the Same Technique

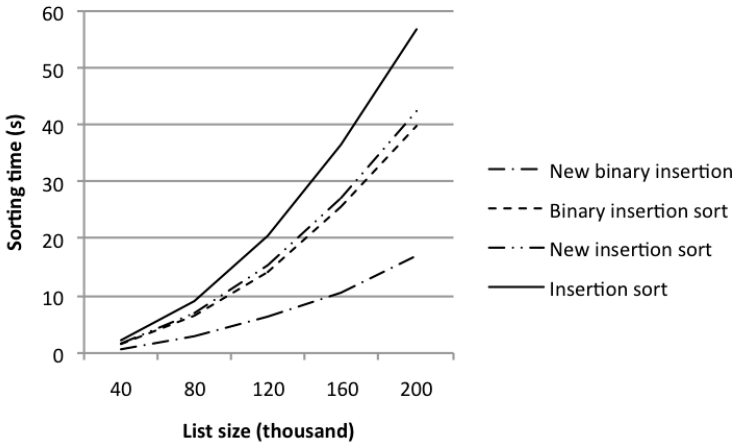

Figure 6. Performance of Different Insertion Sort Algorithms

5. Conclusion

The proposed enhancements have a significant improvement over the classical algorithms. This substantial improvement is achieved by avoiding unnecessary comparisons, which are considered the performance bottleneck in searching or sorting and data shifts, which requires costly memory writes. It shows that avoiding or minimizing either certain comparisons or certain swaps or data movements can enhance typical inefficient algorithms.

This paper shows advantages of the proposed strategies with the three major simple and inefficient algorithms selection sort, bubble sort, and insertion sort, representing three broad classes selection, exchange and insertion based sorting algorithms. Other popular sorting algorithms such as the quick sort, the heap sort and shell sort from these categories have the potentiality to be improved by using the proposed approaches. These methods and the mentioned algorithms deserve future research.

References

[1] I. Flores, "Analysis of Internal Computer Sorting", ACM, vol. 7, no. 4, **(1960)**, pp. 389- 409.
[2] G. Franceschini and V. Geffert, "An In-Place Sorting with O(n log n) Comparisons and O(n) Moves", Proceedings of 44th Annual IEEE Symposium on Foundations of Computer Science, **(2003)**, pp. 242-250.
[3] D. Knuth, "The Art of Computer programming Sorting and Searching", 2nd edition, Addison-Wesley, vol. 3, **(1998)**.
[4] A. D. Mishra and D. Garg, "Selection of the best sorting algorithm", International Journal of Intelligent Information Processing, vol. 2, no. 2, **(2008)** July-December, pp. 363-368.
[5] C. A. R. Hoare, Algorithm 64: Quick sort. Comm. ACM, vol. 4, no. 7 **(1961)**, pp. 321.
[6] Selection Sort, http://www.algolist.net/Algorithms/Sorting/Selection_sort,
[7] E. Horowitz, S. Sahni and S. Rajasekaran, Computer Algorithms, Galgotia Publications.
[8] Selection Sort Algorithm, http://www.cs.ship.edu/~merlin/Sorting/selintro.htm,
[9] Selection Sort, http://en.wikipedia.org,
[10] T. H. Coreman, C. E. Leierson, R. L. Rivest and C. Stein, Introduction to Algorithms, 2nd edition, MIT Press.
[11] Cocktail Sort Algorithm or Shaker Sort Algorithm, http://www.codingunit.com/cocktail-sort-algorithm-or-shaker-sort-algorithm.
[12] 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.
[13] S. Jadoon, S. F. Solehria and M. Qayum, "Optimized Selection Sort Algorithm is faster than Insertion Sort Algorithm: a Comparative Study", International Journal of Electrical & Computer Sciences (IJECS-IJENS), vol. 11, no. 02, pp. 19-24.

[14] S. Chand, T. Chaudhary and R. Parveen, "Upgraded Selection Sort", International Journal on Computer Science and Engineering (IJCSE), ISSN: 0975-3397, vol. 3, no. 4, **(2011)**, pp. 1633-1637.
[15] J. Alnihoud and R. Mansi, "An Enhancement of Major Sorting Algorithms", International Arab Journal of Information Technology, vol. 7, no. 1, **(2010)**, pp. 55-62.
[16] Bingo Sort, http://xlinux.nist.gov/dads/HTML/bingosort.html.
[17] Exact-Sort, http://www.geocities.ws/p356spt/.
[18] V. Mansotra and K. Sourabh, "Implementing Bubble Sort Using a New Approach", Proceedings of the 5th National Conference; INDIACom, New Delhi, **(2011)**.
[19] O. O. Moses, "Improving the performance of bubble sort using a modified diminishing increment sorting", Scientific Research and Essay, vol. 4, no. 8, **(2009)**, pp. 740-744.
[20] H. W. Thimbleby, "Using Sentinels in Insert Sort", Software-Practice and Experience, vol. 19, no. 3, **(1989)**, pp. 303-307.
[21] T. S. Sodhi, S. Kaur and S. Kaur, "Enhanced Insertion Sort Algorithm", International Journal of Computer Applications, vol. 64, no. 21, **(2013)**, pp. 35-39.
[22] M. A. Bender, M. Farach-Colton and M. A. Mosteiro, "Insertion Sort is O(n log n)", Proceedings of the Third International Conference on Fun With Algorithms (FUN), **(2004)**, pp. 16-23.
[23] E. Kapur, P. Kumar and S. Gupta, "Proposal of a two way sorting algorithm and performance comparison with existing algorithms", International Journal of Computer Science, Engineering and Applications (IJCSEA), vol. 2, no. 3, **(2012)**, pp. 61-78.
[24] S. Lipschutz, "Theory and Problems of Data Structure", McGraw Hill Book Company.