An Enhancement of Major Sorting Algorithms

Jehad Alnihoud and Rami Mansi
Department of Computer Science, Al al-Bayt University, Jordan

Abstract: *One of the fundamental issues in computer science is ordering a list of items. Although there is a huge number of sorting algorithms, sorting problem has attracted a great deal of research; because efficient sorting is important to optimize the use of other algorithms. This paper presents two new sorting algorithms, enhanced selection sort and enhanced bubble Sort algorithms. Enhanced selection sort is an enhancement on selection sort by making it slightly faster and stable sorting algorithm. Enhanced bubble sort is an enhancement on both bubble sort and selection sort algorithms with O(nlgn) complexity instead of O(n^2) for bubble sort and selection sort algorithms. The two new algorithms are analyzed, implemented, tested, and compared and the results were promising.*

Keywords: *Enhanced selection sort, enhanced bubble sort, selection sort, bubble sort, number of swaps, time complexity.*

Received May 27, 2008; accepted September 1, 2008

1. Introduction

Information growth rapidly in our world and to search for this information, it should be ordered in some sensible order. Many years ago, it was estimated that more than half the time on many commercial computers was spent in sorting. Fortunately this is no longer true, since sophisticated methods have been devised for organizing data, methods which do not require that the data be kept in any special order [9].

Many algorithms are very well known for sorting the unordered lists. Most important of them are Heap sort, Bubble sort, Insertion sort and shell sort [17]. As stated in [5], sorting has been considered as a fundamental problem in the study of algorithms, that due to many reasons:

- The need to sort information is inherent in many applications.
- Algorithms often use sorting as a key subroutine.
- In algorithm design there are many essential techniques represented in the body of sorting algorithms.
- Many engineering issues come to the fore when implementing sorting algorithms.

Efficient sorting is important to optimize the use of other algorithms that require sorted lists to work correctly; it is also often in producing human-readable output. Formally, the output should satisfy two major conditions:

- The output is in non-decreasing order.
- The output is a permutation, or reordering, of the input.

Since the early beginning of computing, the sorting problem has attracted many researchers, perhaps due to the complexity of solving it efficiently. Bubble sort was analyzed as early as 1956 [2].

Many researchers considered sorting as a solved problem. Even so, useful new sorting algorithms are still being invented, for example, library sort was first published in 2004. Sorting algorithms are prevalent in introductory computer science classes, where the abundance of algorithms for the problem provides a gentle introduction to a variety of core algorithm concepts [1, 19]. In [1], they classified sorting algorithms by:

- Computational complexity (worst, average and best behavior) of element comparisons in terms of list size (n). For typical sorting algorithms good behavior is O(n log n) and bad behavior is $\Omega(n^2)$. Ideal behavior for a sort is O(n). Sort algorithms which only use an abstract key comparison operation always need $\Omega(n \log n)$ comparisons in the worst case.
- Number of swaps (for in-place algorithms).
- Stability: stable sorting algorithms maintain the relative order of records with equal keys (values). That is, a sorting algorithm is stable if whenever there are two records R and S with the same key and with R appearing before S in the original list, R will appear before S in the sorted list.
- Usage of memory and other computer resources. Some sorting algorithms are "in place", such that only O(1) or O(log n) memory is needed beyond the items being sorted, while others need to create auxiliary locations for data to be temporarily stored.
- Recursion: some algorithms are either recursive or non recursive, while others may be both (e.g., merge sort).
- Whether or not they are a comparison sort. A comparison sort examines the data only by

comparing two elements with a comparison operator.

In this paper, two new sorting algorithms are presented. These new algorithms may consider as selection sort as well as bubble sort algorithms. The study shows that the proposed algorithms are more efficient, theoretically, analytically, and practically as compared to the original sorting algorithms. Section 2 presents the concept of enhanced Selection Sort (SS) algorithm and its pseudocode. Furthermore, the implementation, analysis, and comparison with selection sort are highlighted. Section 3 introduces enhanced bubble sort algorithm and its pseudocode, implementation, analysis, and comparison with bubble sort. Also, a real-world case study for the proposed algorithms is presented in section 4. Finally, conclusions were presented in section 5.

2. Enhanced Selection Sort

2.1. Concept

Inserting an array of elements and sorting these elements in the same array (in-place) by finding the maximum element and exchanging it with the last element, and then decreasing the size of the array by one for next call. In fact, the Enhanced Selection Sort (ESS) algorithm is an enhancement to the SS algorithm in decreasing number of swap operations, making the algorithm to be data dependent, and in making it stable. The differences between ESS and SS algorithms are discussed in section 2.5.

2.2. Procedures

The procedures of the algorithms can be described as follows:

- Inserting all elements of the array.
- Calling the "Enhanced Selection Sort" function with passing the array and its size as parameters.
- Finding the maximum element in the array and swapping it with the last index of the same array.
- Decreasing the size of the array by one.
- Calling the "Enhanced Selection Sort" function recursively. The size of the array is decremented by one after each call of the "Enhanced Selection Sort" function. Operationally, the (*size*) after the first call became (*size-1*), and after the second call became (*size-2*), and so on.

2.3. Pseudocode

In simple pseudocode, enhanced selection sort algorithm might be expressed as:

function enhanced selection sort (array , size)

```
1 if size > 1 then
2   var index, temp, max
3     index := size-1
4     max := array(index)
5   for a:= 0 to size-2 do
6       if array(a) ≥ max then
7            max := array(a)
8            index := a
9       end if
10    end for
11    if index ≠ size-1 then
12        temp := array(size-1)
13        array(size-1) := max
14        array(index) := temp
15    end if
16  size := size-1
17  return Enhanced Selection Sort (array , size)
18 else
19      return array
20 end if
```

2.4. Analysis

For loop, in line 5 iterates *n* times in the first call then *n* keeps decreasing by one. We may say that:

$$T(n) = \begin{cases} 0 & n = 0 \\ n + T(n-1) & n > 0 \end{cases} \tag{1}$$

$$\begin{aligned} T(n) &= n + T(n-1) \\ &= n + n-1 + T(n-2) \\ &= n + n-1 + n-2 + T(n-3) \\ &= n + n-1 + n-2 + n-3 + T(n-4) \\ &= \dots \\ &= n + n-1 + n-2 + n-3 + \dots + \\ &\quad (n-k+1) + T(n-k) \end{aligned} \tag{2}$$

$$= \sum_{i=n-k+1}^{n} i + T(n-k) \quad \text{for } n \geq k$$

To terminate the recursion, we should have $n - k = 0$ => $k = n$:

$$\sum_{i=1}^{n} i + T(0) = \sum_{i=1}^{n} i + 0 = n\frac{n+1}{2} \tag{3}$$

So, $$T(n) = n\frac{n+1}{2} = O(n^2) \tag{4}$$

ESS algorithm is easy to analyze compared to other sorting algorithms since the loop does not depend on the data in the array. Selecting the highest element requires scanning all *n* elements (this takes *n* - 1 comparisons) and then swapping it into the last position. Then, finding the next highest element requires scanning the remaining *n* - 2 elements and so on, for $(n-1)+(n-2)+...+2+1 = n(n-1)/2 = O(n^2)$ comparisons.

The number of swaps in the proposed algorithm may elaborate as follows:

a) In the best-case scenario; if the input array is already sorted (ascending order), then there is no need to make swapping, since each element is in the correct place.
b) In the average-case scenario; if the input array is sorted in reverse (descending order), then the

total number of swapping operations is: floor of (n/2). Since swapping the maximum value with the last element means that the maximum and the minimum values are in the correct places. For example, if we have a descending sorted array as follows:

5	4	3	2	1

Then the algorithm will swap the first element (max) with the last element, as follows:

1	4	3	2	5

Since the last element before swapping was the minimum value, it is after swapping got in the correct place and cannot be the maximum value in any of the next comparisons.

In the next comparison, the second element now is the maximum value and it will be swapped with the fourth (size-1)th element.

1	2	3	4	5

The array now is sorted and the algorithm required two swap operations to sort the input array of five elements. That means, the swapping operations that are needed to sort a descending sorted array is floor of (n/2).

c) In the worst case; if the input array is unsorted neither ascending nor descending, then the required swapping operations are approximately *n* operations.

2.5. Comparison with SS Algorithm

SS algorithm, works by selecting the smallest unsorted item remaining in the array, and then swapping it with the item in the next position to be filled. The selection sort has a complexity of $O(n^2)$ [8, 11]. In simple pseudocode, selection sort algorithm might be expressed as:

```
Function SelectionSort(array, size)
1  var i, j
2  var min, temp
3  for i := 0 to size-2 do
4      min := i
5      for j := i+1 to size-1 do
6          if array(j) < array(min)
7              min := j
8          end if
9      end for j
10     temp := array(i)
11     array(i) := array(min)
12     array(min) := temp
13 end for i
```

The main advantage enhanced selection sort over selection sort algorithms is: selection sort always performs O(*n*) swaps while enhanced selection sort depends on the state of the input array. In other words, if the input array is already sorted, the ESS does not perform any swap operation, but selection sort performs *n* swap operations. Writing in memory is more expensive in time than reading, since EBS performs less number of swaps (read/write) then it is more efficient than selection sort when dealing with an array stored in a secondary memory or in EEPROM (electrically erasable programmable read only memory). However, there are many similarities between ESS and SS algorithms, as shown in Table 1.

Table 1. ESS vs SS algorithms.

Criteria	Enhanced Selection Sort	Selection Sort
Best Case	$O(n^2)$	$O(n^2)$
Average Case	$O(n^2)$	$O(n^2)$
Worst Case	$O(n^2)$	$O(n^2)$
Memory	O(1)	O(1)
Stability	Yes	Yes
Number of Swaps	Depends on data: 0, *n*/2, or *n*	Always *n*

To prove that ESS algorithm is relatively faster than SS algorithm, we implement each of them using C++, and measure the execution time of both programs with the same input data, and using the same computer. The built-in function (clock ()) in C++ is used to get the elapsed time of the two algorithms.

```
#include<iostream.h>
#include<ctime>
#include <cstdlib>
int sort(int[], int);
void main()
{       . . . .
    clock_t Start, Time;
    Start = clock();
    // the function call goes here
    Time = (clock() - Start);
    cout<<"Execution Time :  "<<Time<<" ms."<<endl;
}
```

Since the execution time of a program is measured in milliseconds using this function; we should measure execution time of sorting algorithms with a huge size of input array. Table 2 shows the differences between execution times of ESS and SS with using an array of (9000) elements in the best, average, and worst cases.

Table 2 shows that Enhanced Bubble Sort (EBS) is relatively faster than selection sort in all cases. That because the number of comparisons and swap operations are less. In SS, the number of swaps is always (8999), which is (*n-1*), but in ESS, it is (*n*) in the worst-case, (*n*/2) in the average-case, and (0) in the best-case. If the array is stored in a secondary memory; then the SS will operate in relatively low performance as compared with ESS.

3. Enhanced Bubble Sort

The history of Bubble sort algorithm may elaborate as follows:

In 1963 FORTRAN textbook [13] states the following code to what so called “Jump-down” sort.

```
void JumpDownSort(Vector a, int n){
for(int j=n-1; j>o; j--)
  for(int k=0; k<j;k++)
      if (a[j] < a[k])
          Swap(a,k,j);}
```

Table 2. Execution time for ESS vs SS algorithms.

Case	Criteria	Enhanced Selection Sort	Selection Sort
Best	Number of comparisons	40495500	40504499
	Number of swaps	0	8999
	Elapsed time	125 ms	171 ms
Average	Number of comparisons	40495500	40504499
	Number of swaps	4500	8999
	Elapsed time	133 ms	203 ms
Worst	Number of comparisons	40495500	40504499
	Number of swaps	8999	8999
	Elapsed time	156 ms	203 ms

In another early 1962 book [10] “Jump-down” version appears with no name. In another two early works [3, 7] the “jump-down” sort is referred to as selection sort. Then bubble sort is also covered and referred as sorting by repeated comparison and exchanging, respectively.

3.1. Concept and Procedures of EBS

The proposed algorithm is considered as an enhancement to the original Bubble sort algorithm and it works as follows:

Inserting an array of elements and sorting these elements in the same array (in place) by finding the minimum and the maximum elements and exchanging the minimum with the first element and the maximum with the last element, and then decreasing the size of the array by two for next call.

The detailed procedures of the algorithm can be summarized as follows:

1. *Inserting all elements of the array.*
2. *Defining and initializing two variables, (firstindex = 0) and (lastindex = size-1).*
3. *Calling the "Enhanced Bubble Sort" function with passing the array, its size, firstindex, and lastindex as parameters of the function.*
4. *In the "Enhanced Bubble Sort" function, the operation now is to find the maximum and the minimum elements and saving the index value of the max of the array in the variable maxcounter, and the index value of the min in the variable mincounter.*
5. *Put the max in the lastindex and min in the firstindex of the array without losing the last values of the first index and the last index of the original array.*
6. *Decreasing the value of lastindex by one and increasing the value of firstindex by one. Operationally, the size of the array after the first call became (size-2), and after the second call it actually became (size-4), and so on.*
7. *Calling the "Enhanced Bubble Sort " array recursively while the size of the array is greater than one (size>1). Then returning the sorted array.*

3.2. Pseudocode

The pseudocode of EBS algorithm might be expressed as:

```
function EnhancedBubbleSort (array, size, firstindex, lastindex)
if size > 1 then
    var temp := 0, maxcounter := lastindex
    var mincounter := firstindex
    var max := array(lastindex),min := array(firstindex)
    for a:= firstindex to lastindex do
        if array(a) ≥ max then
            max := array(a)
            maxcounter := a
        end if
        if array(a) < min then
            min := array(a)
            mincounter := a
        end if
    end for
    if firstindex==maxcounter AND
    astindex==mincounter then
        array(firstindex):= min
        array(lastindex) := max
     else
     if firstindex==maxcounter AND
         lastindex ≠ mincounter then
         temp := array(lastindex)
         array(lastindex)  := max
         array(firstindex) := min
         array(mincounter) := temp
      else
    if firstindex ≠ maxcounter AND
        lastindex==mincounter then
        temp := array(firstindex)
        array(firstindex):= min
        array(lastindex) := max
        array(maxcounter):= temp
    else
       temp := array(firstindex)
       array(firstindex):= min
```

```
34       array(mincounter):= temp
35       temp := array(lastindex)
36       array(lastindex):= max
37      array(maxcounter):= temp
38   end if
39   end if
40   end if
41   firstindex := firstindex + 1
42   lastindex := lastindex - 1
43   size := size – 2
44   return EnhancedBubbleSort
      (array,size,firstindex,lastindex)
45  else   return array
46  end if
```

3.3. Analysis

The first call of the function loop iterates *n* times, as shown in line 5, and in the second call, the loop iterates *n*-2 times, and so on. We may analyze the algorithm as follows:

$$
\begin{aligned}
T(n) &= 1, && \text{for } n < 2\\
T(n) &= n + T(n-2), && \text{for } n \geq 2\\
&= n + (n-2) + T(n-4)\\
&= 2n - 2 + T(n-4)\\
&= 2n - 2 + (n-4) + T(n-6)\\
&= 3n - 6 + T(n-6)\\
&= 4n - 12 + T(n-8)\\
&= 5n - 20 + T(n-10)\\
&= \ldots\\
&= in - (i^2 - i) + T(n - 2i)\\
&= kn - (k^2 - k) + T(n - 2k)
\end{aligned}
$$

Assume that:

$n = 2^k$, *taking* lg_2 *on both sides:*
$lg\, n = k\, lg_2$
$lg\, n = k * 1$
$lg\, n = k$
Therefore,
$kn - (k^2 - k) + T(n - 2k) =$
$nlgn - ((lgn)^2 - lgn) + T(n - 2lgn)$
$T(n) = O(nlgn)$

3.4. Comparison with Bubble Sort Algorithm

Bubble Sort (BS) repeatedly stepping through the array to be sorted, comparing two items at a time and swapping them if necessarily. Passing through the list is repeated until no swaps are needed, which indicates that the list is already sorted. The algorithm gets its name from the way smaller elements "bubble" to the top of the list. Since it uses comparisons only to operate on elements, it is a comparison sort [2, 11]. In simple pseudocode, bubble sort algorithm might be expressed as:

```
function bubbleSort(array, array_size)
1   var i, j, temp;
2   for i:=(array_size-1)downto 0 step-1
3       for j := 1 to i do
4           if array(j-1) > array(j) then
5               temp := array[j-1];
6               array[j-1] := array[j];
7               array[j] := temp;
8           end if
9       end for
10  end for
11  end function
```

The positions of the elements in bubble sort play an important role in determining its performance. Large elements at the beginning of the list are quickly swapped, while small elements at the beginning move to the top extremely slowly. This has led to these types of elements being named rabbits and turtles, respectively [2, 20]. BS algorithm makes *n* comparisons at the first time, and then it makes *n*-1 comparisons, and so on [19]. This yields to $n + (n-1) + (n-2) + \ldots + 2 + 1$ which is equal to $n(n+1)/2$ which is $O(n^2)$.

In EBS algorithm, decreasing the number of comparisons by two each call led the complexity to be $O(nlgn)$ which is very much better than $O(n^2)$. The difference between EBS and BS may not be clear with a small size of the input array, but with a large size it is very clear that EBS is faster than bubble sort. The main differences between EBS and BS are:

- In the average-case; BS performs *n*/2 swapping operations, in the best-case it performs 0 operations, and in the worst-case it performs *n* swapping operations, while EBS performs (*n*/2) swapping operations in all cases.
- Since EBS needs to check the locations of the found minimum and maximum elements before performing swapping to avoid losing data; it has a larger size of code than Bubble sort.
- In all cases, EBS makes $O(nlgn)$ comparisons and Bubble sort makes $O(n^2)$ comparisons to sort *n* elements of the input array.

Table 3 shows the main differences between EBC and Bubble sort algorithms:

Table 3. EBS vs BS algorithms.

Criteria	**Enhanced Bubble Sort**	**Bubble Sort**
Best Case	$O(nlgn)$	$O(n^2)$
Average Case	$O(nlgn)$	$O(n^2)$
Worst Case	$O(nlgn)$	$O(n^2)$
Memory	O(1)	O(1)
Stability	Yes	Yes
Number of Swaps	Always *n*/2	Depends on data: 0, *n*/2, or *n*

To be certain about these results we should compute the execution time of the implementation program of each algorithm. Table 4 shows the differences between execution time of EBS and BS algorithms using C++ with (9000) elements as the size of the input array.

From Table 4 we may conclude that EBS is faster than bubble sort especially when *n* is large. This is an important advantage of EBS over bubble sort. At the same time, EBS always performs *n*/2 swap operations and it has a larger code size as compared to bubble sort.

Table 4. Execution time for EBS vs BS algorithms.

Case	Criteria	Enhanced Bubble Sort	Bubble Sort
Best	Number of comparisons	20254500	40504500
	Number of swaps	4500	0
	Elapsed time	93 ms	187 ms
Average	Number of comparisons	20254500	40504500
	Number of swaps	4500	4500
	Elapsed time	109 ms	437 ms
Worst	Number of comparisons	20254500	40504500
	Number of swaps	4500	9000
	Elapsed time	140 ms	453 ms

To support the previous results we may compare the proposed algorithms with some recent advanced sorting algorithms, such as cocktail sort [12], shell sort [18], and enhanced shell sort [17].

Cocktail sort as stated in [12], also known as bidirectional BS and cocktail shaker sort, is a variation of BS and selection sort that is both a stable sorting algorithm and a comparison sort. The algorithm differs from BS in that it sorts in both directions each pass through the list. This sorting algorithm is only marginally more difficult than BS to implement, and solves the problem with so-called turtles in BS.

In the first stage of the cocktail sort, it loops through the array from bottom to top, as in BS. During the loop, adjacent items are compared. If at any point the value on the left is greater than the value on the right, the items are swapped. At the end of the first iteration, the largest number will reside at the end of the set.

In the second stage, it loops through the array in the opposite direction; starting from the item just before the most recently sorted item, and moving back towards the start of the list. Again, adjacent items are swapped if required. The cocktail sort also fits in the category of exchange sorts due to the manner in which elements are moved inside the array during the sorting process.

As illustrated in [12], both space and time complexities of the Cocktail sort are the same as that of the BS for exactly the same reasons. That is, time complexity is O(n^2), and space complexity for in-place sorting is O(1). EBS is more efficient and faster than both, bubble sort and Cocktail sort, since it takes O(*nlgn*) time complexity while BS and cocktail sort take O(n^2) to sort *n* elements.

Shell sort [18] which is an enhanced version of insertion sort, reduces the number of swaps of the elements being sorted to minimize the complexity and time as compared to the insertion sort. Shell sort improves the insertion sort by comparing elements separated by a gap of several positions. This lets an element take bigger steps toward its expected position. Multiple passes over the data are taken with smaller and smaller gap sizes. The last step of Shell sort is a plain insertion sort, but by then, the array of data is guaranteed to be almost sorted.

The shell sort is a “Diminishing Increment Sort”, better known as a comb sort [4] to the unwashed programming masses. The algorithm makes multiple passes through the list, and each time sorts a number of equally sized sets using the insertion sort [12]. The size of the set to be sorted gets larger with each pass through the list, until the set consists of the entire list. This sets the insertion sort up for an almost-best case run each iteration with a complexity that approaches O(*n*). Donald L. shell [18] invented a formula to calculate the value of ‘h’. This work focuses to identify some improvement in the conventional Shell sort algorithm.

As stated in [15, 18], the original implementation of Shell sort performs O(n^2) comparisons and exchanges in the worst case. A minor change given in V. Pratt's book [13] which improved the bound to O($n \log^2 n$). This is worse than the optimal comparison sorts, which are O($n \log n$).

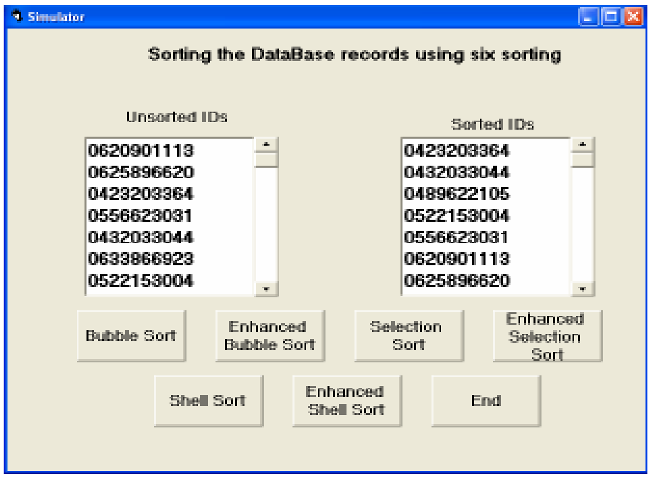
ESS algorithm [17] is an improvement in the Shell sort algorithm to calculate the value of ‘h’. It has been observed that by applying this algorithm, number of swaps can be reduced up to 60 percent as compared to the shell sort algorithm, but it is still a quadratic sorting algorithm. It is clear that EBS is faster than shell sort and its variants since all of them makes a quadratic time while EBS makes a O(*nlgn*) time to sort *n* elements.

4. Case Study

This section presents a real-world case study to sort students of Al al-Bayt University in Jordan by the university number (students IDs) in ascending order. In this case study, BS, SS, shell sort, enhanced shell sort, ESS, and EBS algorithms are applied with 12500 as total number of students.

Table 5. Execution time for the six algorithms.

Algorithm	Elapsed Time
Bubble sort	506 ms
Enhanced bubble sort	151 ms
Selection sort	346 ms
Enhanced selection sort	307 ms
Shell sort	322 ms
Enhanced shell sort	249 ms


Simulator

Sorting the DataBase records using six sorting

Unsorted IDs

```
0620901113
0625896620
0423203364
0556623031
0432033044
0633866923
0522153004
```

Sorted IDs

```
0423203364
0432033044
0489622105
0522153004
0556623031
0620901113
0625896620
```

Figure 1. The interface of the sorting application.

The simulator was built using Visual C++ to deal with the database and to sort its records. The interface of this simulator is shown in Figure 1.

The elapsed time for sorting the database is measured using the (clock()) function of C++ and recorded for each algorithm, as shown in Table 5. Figure 2 shows a comparison of the elapsed time in milliseconds of the BS, EBS, SS, and ESS.

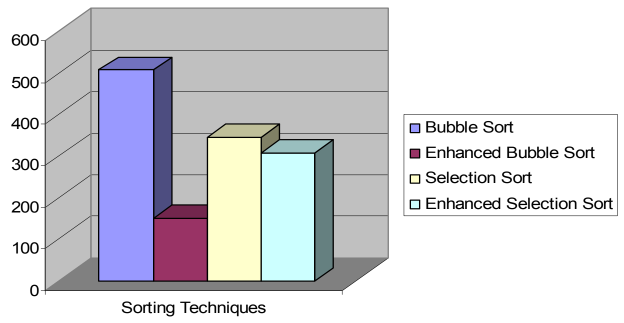


Figure 2. Comparison of sorting techniques.

From Figure 2, it is apparent that the ESS relatively increases the efficiency of the selection sort and EBS speeds up the bubble sort and enhances its efficiency.

5. Conclusions

In this paper, two new sorting algorithms are presented. ESS has O(n^2) complexity, but it is faster than SS, especially if the input array is stored in secondary memory, since it performs less number of swap operations.

SS can be specially implemented to be stable. One way of doing this is to artificially extend the key comparison, so that comparisons between two objects with other equal keys are decided using the order of the entries in the original data order as a tie-breaker. ESS is stable without the need to this special implementation.

EBS is definitely faster than BS, since BS performs O(n^2) operations but EBS performs O($nlgn$) operations to sort n elements. Furthermore, the proposed algorithms are compared with some recent sorting algorithms; shell sort and enhanced shell sort. These algorithms are applied on a real-world case study of sorting a database of (12500) records and the results showed that the EBS also increases the efficiency of both shell sort and enhanced shell sort algorithms.

References

[1] Aho A., Hopcroft J., and Ullman J., *The Design and Analysis of Computer Algorithms*, Addison Wesley, 1974.

[2] Astrachanm O., *Bubble Sort: An Archaeological Algorithmic Analysis*, Duk University, 2003.

[3] Bell D., "The Principles of Sorting," *Computer Journal of the Association for Computing Machinery*, vol. 1, no. 2, pp. 71-77, 1958.

[4] Box R. and Lacey S., "A Fast Easy Sort," *Computer Journal of Byte Magazine*, vol. 16, no. 4, pp. 315-315, 1991.

[5] Cormen T., Leiserson C., Rivest R., and Stein C., *Introduction to Algorithms*, McGraw Hill, 2001.

[6] Deitel H. and Deitel P., *C++ How to Program*, Prentice Hall, 2001.

[7] Friend E., "Sorting on Electronic Computer Systems," *Computer Journal of ACM*, vol. 3, no. 2, pp. 134-168, 1956.

[8] Knuth D., *The Art of Computer Programming*, Addison Wesley, 1998.

[9] Kruse R., and Ryba A., *Data Structures and Program Design in C++*, Prentice Hall, 1999.

[10] Ledley R., *Programming and Utilizing Digital Computers*, McGraw Hill, 1962.

[11] Levitin A., *Introduction to the Design and Analysis of Algorithms*, Addison Wesley, 2007.

[12] Nyhoff L., *An Introduction to Data Structures*, Nyhoff Publishers, Amsterdam, 2005.

[13] Organick E., *A FORTRAN Primer*, Addison Wesley, 1963.

[14] Pratt V., *Shellsort and Sorting Networks*, Garland Publishers, 1979.

[15] Sedgewick R., "Analysis of Shellsort and Related Algorithms," *in Proceedings of the 4th Annual European Symposium on Algorithms*, pp. 1-11, 1996.

[16] Seward H., "Information Sorting in the Application of Electronic Digital Computers to Business Operations," *Masters Thesis*, 1954.

[17] Shahzad B. and Afzal M., "Enhanced Shell Sorting Algorithm," *Computer Journal of Enformatika*, vol. 21, no. 6, pp. 66-70, 2007.

[18] Shell D., "A High Speed Sorting Procedure," *Computer Journal of Communications of the ACM*, vol. 2, no. 7, pp. 30-32, 1959.

[19] Thorup M., "Randomized Sorting in O(n log log n) Time and Linear Space Using Addition, Shift, and Bit Wise Boolean Operations," *Computer Journal of Algorithms*, vol. 42, no. 2, pp. 205-230, 2002.

[20] Weiss M., *Data Structures and Problem Solving Using Java*, Addison Wesley, 2002.

Jehad Alnihoud received his PhD of computer science from University Putra Malaysia in 2004. Currently, he is an assistant professor at the Faculty of Information Technology in Al al-Bayt University in Jordan. His research areas include image retrieval and indexing, image processing, algorithms, GIS, and computer graphics.

Rami Mansi obtained his BSc in information technology with a major in software engineering from Philadelphia University, 2006. His research interests include computer graphics, algorithms design and analysis, HCI, programming, and text processing.