

Comparative Analysis of five Sorting Algorithms on the basis of Best Case, Average Case, and Worst Case

¹Mohsin Khan, ²Samina Shaheen, ³Furqan Aziz Qureshi

¹Computing & Technology Department, IQRA University Islamabad Campus (IUIIC)

²Computing & Technology Department, ABASYN University Islamabad Campus (AIUC)

³Center for Emerging Science, Engineering & Technology, Islamabad

Email: mohsin_btn@yahoo.com, samnpg13@gmail.com, furqan.aziz89@gmail.com

ABSTRACT

Sorting is one of the fundamental issues in computer science. Sorting problem gain more popularity, as efficient sorting is more important to optimize other algorithms e.g. searching algorithms. A number of sorting algorithms has been proposed with different constraints e.g. number of iterations (inner loop, outer loop), complexity, and CPU consuming problem. This paper presents a comparison of different sorting algorithms (Sort, Optimized Sort, Selection Sort, Quick Sort, and Merge Sort) with different data sets (small data, medium data, and large data), with Best Case, Average Case, and worst case constraint. All six algorithms are analyzed, implemented, tested, compared and concluded that which algorithm is best for small, average, and large data sets, with all three constraints (best case, average case, and worst case).

Key words: Bubble Sort, Selection Sort, Quick Sort, Merge Sort, Optimized Bubble Sort, Enhanced selection Sort, Complexity

1. INTRODUCTION

An algorithm is a well-defined step by step procedure to solve computational problems. Algorithm takes an input, and provides output. We can also say that algorithm is a tool or a sequence of steps to solve computational problems [1]. To design an algorithm various techniques and methodologies are applied. Sorting algorithms are design to order the data set into ascending or descending orders. Sorting algorithms are very important, as these algorithms are used to design other algorithms e.g. searching algorithms. It is used in many importing applications, since the performance of sorting algorithm is very important issue [2]. Due to this importance a number of sorting algorithms has been presented each with different constraints. Since we can formally define sorting as:

Input: A sequence of data set having n numbers of random order data $A = (a_1, a_2, a_3 \dots, a_{(n-1)}, a_n)$

Output: A permutation of the input sequence $A' = (a'_1, a'_2, a'_3 \dots, a'_{(n-1)}, a'_n)$, such that $a'_1 \leq a'_2 \leq a'_3 \leq \dots \leq a'_{(n-1)} \leq a'_n$



Fig 1.1: Algorithms Diagram

For example if the given sequence is (59, 41, 31, 41, 26, 58); then the output sequence of sorting algorithm will be (26, 31, 41, 41, 58, 59)[3].

As discussed above sorting is an intermediate and an important part of managing data. Due to their importance a number of sorting algorithms are proposed e.g. bubble sort, selection sort, insertion sort, quick sort, merge sort etc. Some of sorting algorithms are very simple e.g. bubble sort; while other are complex e.g. quick sort and merge sort. Many researchers proposed some enhancement in the above algorithms e.g. enhanced bubble sort, optimized bubble sort, enhanced selection sort, and enhanced quick sort etc. Remember that all sorting algorithms work for a specific environment. Some sorting algorithms works best on small data sets, some works good in a large data set, while others work good in a medium data set. The performance of a sorting algorithm is described by their complexity. Complexity of the algorithm is

described by a standard notation big O as $O(n)$; Where Big O represents the complexity of the algorithm, while n represents the number of elements in the database.

In this paper, a comparison of most used sorting algorithms Bubble sort, Selection sort, Insertion sort, Quick sort, and merge sort are described on the basis of their CPU time, Complexity, and number Inner & Outer loop iterations. The remaining paper is organized as: section 2 describes the criteria for comparison, section 3 describes the algorithms used in this simulation, section 4 describes the results of simulation, and section 5 describes the conclusion and future work.

2. Criteria of comparison:

- a. **Speed of sorting:** Many algorithms have same complexity but not have same speed. The speed will be different in best case, average case, and worst case. Since judge the algorithm on factors best case, average case, and worst case. Speed is also considered when it takes values from external storage devices or main memory [4][5].
- b. **Memory:** some algorithms take more memory to sort, as they create a new array or variables to store sorted elements.
- c. **Complexity:** the effectiveness of an algorithm is directly proportional to its complexity[6]. Sorting algorithms are divided into two main category according to their complexity: the $O(n^2)$ category and $O(n \log n)$ category.

3. Sorting Algorithms:

As described above sorting algorithms are divided into two main categories. The bubble sort, insertion sort, and selection sort falls in $O(n^2)$ category, while quick and merge sort falls in $O(n \log n)$ category. The description of these algorithms is described below.

3.1 Bubble sort:the basic sorting algorithm is bubble sort. It compares two adjacent elements and do swap operation if there is miss order found with repeated steps. This is also called a comparison sorting algorithm [7]. The original bubble sort does

more iteration even if the data set is sorted. The complexity of bubble sort is $O(n^2)$ for all cases e.g. best case, worst case, and average case. For this bubble sort is used only for small number of data set, for large and average data sets this algorithm is impractical. The main advantage of bubble sort is its simplicity, and ease of implementation; but the code inefficient as do more iterations [8]. In bubble sort algorithm the number of iterations is same for all cases, since this algorithm is not practical. This algorithm is enhanced in Enhanced Bubble Sort algorithm, discussed next. The bubble sort algorithm is described below:

```
Procedurebubble_Sort (Array, size)
1. vari, j, temporary
2 for i=1 to size
  a. for j = 1 to size
    b. if Array(j+1) > Array(j) then do
  i. assign temporary = Array[j+1]
    ii. assign Array[j+1] = Array[j]
    iii. assign Array[j] = temporary
    iv. end if statement
  c. end for j loop
3. end for I loop
```

Original bubble sort algorithm is enhanced to reduce the number of repetitions. The complexity of enhanced bubble sort algorithm is same as bubble sort for average and worst case, but in case of best case its complexity becomes $O(n)$, which is better than original bubble sort. Also the inner loop iteration is minimized as we run the inner loop to size-1 times. This algorithm is described below:

```
ProcedureE_bubbleSort (Array, size)
1. vari, j, temporary
2 for i= 1 to size-1 step_size is 1
  a. for j = 1 to istep_size is 1
    b. if Array(j-1) > Array(j) then do
  i. temporary = Array[j-1];
    ii. Array[j-1] = Array[j];
    iii. Array[j] = temporary;
    iv. end if statement
  c. end for j loop
3. end for I loop
```

3.2 Enhanced Bubble Sort:Bubble sort is enhanced in [9]. This algorithm works as to find maximum and minimum element in the array, and then swap minimum element with the first element and maximum element with the last element in repeated steps. Remember that at every step decrease the array size by two. The complexity of

Enhanced Bubble Sort algorithm is $O(n \log n)$ for all cases e.g. best case, average case and worst case. The number of swaps is $n/2$ for all cases but bubble sort uses $0, n/2$, and n swaps for each case e.g. best case, average case and worst case respectively. The Enhanced Bubble Sort Algorithm is described below:

Procedure EnhancedBubbleSort (Array, size, firstindex, lastindex)	
1. if size > 1 then	
A. variables temp = 0,	
B. variables maxcounter = lastindex	
C. variable mincounter = firstindex	
D. variable maximum = Array(lastindex)	
E. variable minimum = Array(firstindex)	
a. for a= firstindex to lastindex step 1	
I. if Array(a) ≥ max then do	
i. assign maximum = Array(a)	
ii. assign maxcounter = a	
II. end of if	
III. if Array(a) < minimum then do	
i. assign minimum = Array(a)	
ii. assign mincounter = a	
IV. end of if	
b. end for (a loop)	
F. if (firstindex==maxcounter) AND (lastindex==mincounter) then	
a. assign Array(firstindex) = minimum	
b. assign Array(lastindex) = maximum	
G. else	
a. if (firstindex==maxcounter) AND (lastindex ≠ mincounter) then do	
I. assign temp := array(lastindex)	
II. assign Array(lastindex) = maximum	
III. assign Array(firstindex) = minimum	
IV. assign Array(mincounter) = temp	
b. else	
I. if (firstindex ≠ maxcounter) AND (lastindex==mincounter) then do	
i. assign temp = Array(firstindex)	
ii. assign Array (firstindex) = minimum	
iii. assign Array (lastindex) = maximum	
iv. assign Array (maxcounter) = temp	
II. else	
i. assign temp = Array(firstindex)	
ii. assign Array(firstindex) = minimum	
iii. assign Array(mincounter) = temp	
iv. assign temp =	

	v. assign Array(lastindex) = maximum
	vi. assign Array(maxcounter) = temp
	III. end of if
	c. end of if
	H. end of if
	I. assign firstindex = firstindex + 1
	J. assign lastindex = lastindex - 1
	K. assign size = size - 2
	L. return EnhancedBubbleSort (Array,size,firstindex,lastindex)
3. else	
	a. return array
4. end of if	

3.3 Selection Sort: this algorithm is the improved version of bubble sort and thus one of the simplest types of sorting algorithms. Selection Sort is a comparison sorting algorithm. This algorithm search the minimum element in the unsorted array or list and swap it to the first unsorted array location, with repeated iteration steps. This algorithm takes more time in finding minimum element in the array list. The complexity of selection sort algorithm is $O(n^2)$ for all best, average and worst cases [10][11]. This algorithm is used for small list, but replaced by insertion sort [8]. Selection sort uses more comparisons but less amount of data moving, since if data set has less key but large data shairing then Selection is sort is best [12].

Function Selection_Sort (Array, Size)
1. take Variable i and j
2. take variable minimum and temp
3. for i=0 to Size - 2
a. assign min = i
b. for j=i+1 to Size - 1
i. if Array(j) < Array(minimum)
ii. assign minimum = j
iii. End of if statement
c. End of j loop
4. assign temp = Array(i)
5. assign Array(i) = Array(minimum)
6. assign Array(minimum) = temp
7. End of i loop
8. End of function

3.4 Enhanced Selection Sort: this algorithm is proposed by JehadAlnihoud and Rami Mansi in paper [9]. The Enhanced Selection Sort algorithm works as by finding the maximum element in the array and interchange it with the last element of the array. The size of the array is decreased at each

iteration of the loop, which reduces the number of swaps as compare to Selection Sort algorithm. The Selection Sort performs $O(n)$ number of swaps while Enhanced Selection sort swapping is dependent on the number of array elements. In the best case it does not perform any swap, while selection sort performs $O(n)$ number of swaps. Enhanced Selection Sort algorithm is described below:

Function Enhanced_Selection_Sort(Array, Size)
<p>A. if size > 1 then</p> <ol style="list-style-type: none"> 1. variable index, temp, maximum 2. assign index = size-1 3. assign max = array(index) 4. for i = 0 to size-2 <ol style="list-style-type: none"> a. if Array(i) ≥ maximum then <p>i. assign maximum = Array(i)</p> <ol style="list-style-type: none"> ii. assign index = i b. end if <p>5. end i loop</p> <p>6. if index ≠ size-1 then</p> <ol style="list-style-type: none"> a. assign temp = Array(size-1) b. assign Array(size-1) = maximum c. assign Array(index) = temp <p>7. end if</p> <p>8. assign size = size-1</p> <p>9. return Enhanced_Selection_Sort (array , size)</p> <p>B. else</p> <ol style="list-style-type: none"> 1. return array <p>C. end if</p>

3.5 Insertion Sort: Insertion sort algorithm is a simple and mostly used algorithm for small and mostly sorted arrays lists. This algorithm uses two same size array lists: one is sorted and one is unsorted. In each step of sorting a minimum element is found at unsorted list and placed it to the sorted list at its proper location [11]. This algorithm uses two arrays since more memory is used, also more time consuming to find minimum element and copy it to new sorted array. The complexity of this algorithm is $O(n)$ for best case and $O(n^2)$ for average and worst cases [10]. This algorithm is relatively used for small and average data sets in place of bubble and selection sort algorithms. Insertion Sort algorithm is twice faster than bubble sort algorithm [8]. The insertion sort algorithm is described below:

Procedure insertion_sort(array, length)
<ol style="list-style-type: none"> 1. variable i, j ,temporary 2. for i = 1 to length step 1 <ol style="list-style-type: none"> A. Assign j = i; B. while (j > 0 && array[j - 1] > array[j]) do <ol style="list-style-type: none"> a. Assign temporary = array[j] b. Assign array[j] = array[j - 1] c. Assign array[j - 1] = temporary d. Assign j=j-1 C. end of while loop 3. end of for loop

3.6 Quick Sort: Quick sort is a divide and conqueror algorithm uses recursion to sort the data [13]. It is also called a comparison sort developed by Tony Hoar [6][14]. It uses a programmer selected pivot to sort the data list. Quick Sort Algorithm is divided into two parts: the first part is a procedure QUICK to use the reduction steps of the algorithm, while second part uses the QUICK procedure to sort the elements in the array list [13]. The complexity of quick sort in best and average cases is $O(n \log n)$ and $O(n^2)$ in worst case [10]. The average case time complexity is $O(n \log n)$ which is a bit costly if used for large data sets, also in worst case complexity is $O(n^2)$ which make it impractical for large worst data sets [15]. The Quick Sort algorithm is described below:

Procedure quickSort(array[],left, right)
<ol style="list-style-type: none"> 1. variable i,j, pivot 2. assign i=left 3. assign j=right 4. assign pivot = array[(i+j)/2] // partition 5. while i <= j do <ol style="list-style-type: none"> A. while array[i] < pivot do <ol style="list-style-type: none"> a. assign i=i+1 B. end of while C. while array[j] > pivot do <ol style="list-style-type: none"> a. assign j=j-1 b. end of while D. if i <= j then do <ol style="list-style-type: none"> a. variable temp b. assign temp = array[i] c. assign array[i] = array[j] d. assign array[j] = temp e. assign i=i+1 f. assign j=j-1 E. end of if 6. end of while // recursion 7. if left < j then do <ol style="list-style-type: none"> A. quickSort(input, left, j) 8. if i < right then do <ol style="list-style-type: none"> A. quickSort(input, i, right)

3.7 Enhanced Quick Sort: Quick Sort is enhanced by Rami Massi in paper [16]. Enhanced Quick Sort divide original array in three temporary arrays: the positive array (pos_array), negative array (neg_array), and frequent array (freq_array). This algorithm uses three procedures: Scan, Move, and Sort. As compare to quick sort, enhanced quick sort is faster in case of large lists. the complexity of enhanced quick sort is $O(n)$ in all cases e.g. best case, average case, and worst case. The algorithms of scan, move and sort procedure of enhanced quick sort are described below:

Procedure Scan(array, size)
<p>1 if size > 1 then</p> <p>A. variable var, maximum, minimum, No_of_Pos, No_of_Neg</p> <p>B. assign max=array(0)</p> <p>C. assign min:=array(0)</p> <p>D. assign No_of_Pos=0</p> <p>E. assign N_of_Neg=0</p> <p>F. for a= 0 to size-1 step 1</p> <p style="padding-left: 20px;">a. if array(var) > maximum then do</p> <p style="padding-left: 40px;">i. assign max = array(var)</p> <p style="padding-left: 20px;">b. else</p> <p style="padding-left: 40px;">i. assign minimum = array(var)</p> <p style="padding-left: 20px;">c. end of if</p> <p style="padding-left: 20px;">d. if array(a) ≥ 0 then do</p> <p style="padding-left: 40px;">i. assign No_of_Pos= No_of_Pos+1</p> <p style="padding-left: 20px;">e. Else</p> <p style="padding-left: 40px;">i. Assign No_of_Neg = No_of_Neg+1</p> <p style="padding-left: 20px;">f. end of if</p> <p>G. end of for</p> <p>H. if minimum ≠ maximum then do</p> <p style="padding-left: 20px;">a. Move(array, size, No_of_Pos, No_of_Neg, maximum, minimum)</p> <p>I. end of if</p> <p>2. end of if</p>

Procedure Move(array, size, No_of_Pos, No_of_Neg, maximum, minimum)
<p>1. variable b,c,d,i</p> <p>2. assign i=0</p> <p>3. create a new array: Frequent_Array[size] and initialize by the value (minimum-1)</p> <p>4. if No_of_Pos > 0 then do</p> <p style="padding-left: 20px;">A. create a new array: Positive_Array[maximum+1]</p> <p style="padding-left: 20px;">B. for b=0 to maximum step 1</p> <p style="padding-left: 40px;">a. assign Positive_Array(b) = minimum-1</p> <p style="padding-left: 20px;">C. end of for</p> <p>5. end of if</p> <p>6. if No_of_Neg > 0 then do</p> <p style="padding-left: 20px;">A. create a new array: Negative_Array[minimum +1]</p> <p style="padding-left: 20px;">B. for c= 0 to min +1 step 1</p> <p style="padding-left: 40px;">a. assign Negative_Array(c)= minimum-1</p> <p style="padding-left: 20px;">C. end of for</p> <p>7. end of if</p> <p>8. for d= 0 to size-1 step 1</p> <p style="padding-left: 20px;">A. if array(d) ≥ 0 then do</p> <p style="padding-left: 40px;">a. if</p>

<p style="padding-left: 40px;">Positive_Array(array(d))==minimum-1 then do</p> <p style="padding-left: 60px;">i. assign Positive_Array(array(d))=array(d)</p> <p style="padding-left: 20px;">b. Else</p> <p style="padding-left: 40px;">i. Frequent_Array(i):=array(d)</p> <p style="padding-left: 40px;">ii. Assign i=i+1</p> <p style="padding-left: 20px;">c. end of if</p> <p>B. Else</p> <p style="padding-left: 20px;">a. if Negative_Array(array(d))==minimum-1 then do</p> <p style="padding-left: 40px;">i. Negative_Array(array(d))=array(d)</p> <p style="padding-left: 20px;">b. Else</p> <p style="padding-left: 40px;">i. Assign Frequent_Array(i)=array(d)</p> <p style="padding-left: 40px;">ii. Assign i= i+1</p> <p style="padding-left: 20px;">C. end of if</p> <p style="padding-left: 40px;">a. end of if</p> <p>9. end of for</p> <p>10. Sort(array, Negative_Array, Positive_Array, Frequent_Array,..., No_of_Neg, No_of_Pos, maximum, minimum, i)</p>

Procedure Sort(array, Negative_Array, ..., Positive_Array, Frequent_Array,..., No_of_Neg, No_of_Pos, maximum, minimum, i)
<p>1. variable index,x,y</p> <p>2. assign index=0</p> <p>3. if No_of_Neg > 0 then do</p> <p style="padding-left: 20px;">A. for x= minimum downto 0 do</p> <p style="padding-left: 40px;">a. if Negative_Array(x) ≠ minimum-1 then do</p> <p style="padding-left: 60px;">i. assign array(index)= Negative_Array(x)</p> <p style="padding-left: 60px;">ii. assign index= index+1</p> <p style="padding-left: 60px;">iii. for y= 0 to i step 1</p> <p style="padding-left: 60px;">iv. if Frequent_Array(y)==array(index-1) then</p> <p style="padding-left: 80px;">v. assign array(index)= Frequent_Array(y)</p> <p style="padding-left: 80px;">vi. assign index= index+1</p> <p style="padding-left: 60px;">vii. end of if</p> <p style="padding-left: 60px;">viii. end of for</p> <p style="padding-left: 40px;">b. end of if</p> <p style="padding-left: 20px;">B. end of for</p> <p>4. end of if</p> <p>5. if No_of_Pos > 0 then do</p> <p style="padding-left: 20px;">A. for x= 0 to maximum do</p> <p style="padding-left: 40px;">a. if Positive_Array(x) ≠ minimum-1 then do</p> <p style="padding-left: 60px;">b. assign array(index)= Positive_Array(x)</p> <p style="padding-left: 60px;">c. assign index= index+1</p> <p style="padding-left: 60px;">d. for y= 0 to i step 1</p> <p style="padding-left: 60px;">e. if Frequent_Array(y)==array(index-1) then do</p> <p style="padding-left: 80px;">f. assign array(index)=Frequent_Array(y)</p> <p style="padding-left: 80px;">g. assign index= index+1</p> <p style="padding-left: 60px;">h. end of if</p> <p style="padding-left: 60px;">i. end of for</p> <p style="padding-left: 60px;">j. end of if</p> <p style="padding-left: 40px;">B. end of for</p> <p>6. end of if</p>

3.8 Merge Sort: this is also a divide and conqueror algorithm, with advantage of ease of merging lists with new sorted lists. The worst case complexity of merge sort is $O(n \log n)$, since could be used for large and worst data sets.

Merge sort uses the following three steps to sort a array list [8]:

1. **Divide:**if size of array is greater than 1, then split it into two equal half size sub arrays.
2. **Conquer:** sort both sub arrays by recursion.
3. **Merging:**Combine both sorted sub arrays into original size array. This will gives a complete sorted array.

Merge sort is more suitable for large and worst case, but it uses more memory as compare to other divide and conqueror algorithms. Merge sort algorithm is described below:

Procedure mergesort(a, b, low, high)
1. variable pivot
2. iflow<high then do
A. Assign pivot=(low+high)/2
B. mergesort(a,b,low,pivot)
C. mergesort(a,b,pivot+1,high)
D. merge(a,b,low,pivot,high)
3. end of if

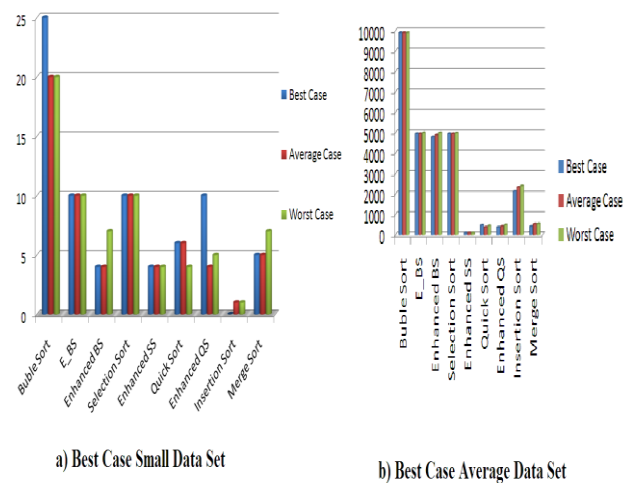
Procedure merge(a,b,low,pivot,high)
1. Variavle h,i,j,k;
2. assign h=low
3. assign i=low
4. assign j=pivot+1;
5. while((h<=pivot)&&(j<=high)) do
A. if(a[h]<=a[j]) then do
a. assign b[i]=a[h]
b. assign h=h+1
B. else
a. assign b[i]=a[j]
b. assign j=j+1
C. ed of if
D. assign i=i+1
7. end of while
8. if(h>pivot) then do
A. for k=j to high step 1
a. assign b[i]=a[k]
b. assign i=i+1
9. else
A. for(k=h to pivot step 1
a. assign b[i]=a[k]
b. assign i=i+1
B. end of for
10. end of if
11. for k=low to high step 1
A. assign a[k]=b[k]

The complexity of all above sorting algorithms is described in table 1.1.

4. Results:The simulation results for Comparisons of Sorting Algorithms are run on Asus EEEPC1001p, Simulation parameters for this simulation are described in table 1.2.

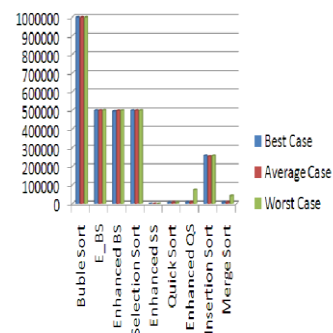
Parameters	Constraints
System	Asus Eee PC 1001p, dual core, 1.67 processor, 1 GB Ram
Compiler	Visual C++ 2008
Constraints	Best Case, Average Case, Worst Case (Small Data set, Medium data set, Large data set)
Best Case	Array of 5 elements
Average data set	Array of 100 elements
Large data set	Array of 1000 elements

Table 1.2: Implementation Parameters



a) Best Case Small Data Set

b) Best Case Average Data Set



c) Best Case Large Data Set

Fig 1.2: Inner Loop Number of Iterations

The results are described as number of inner loop iterations, number of Outer loop Iterations, and CPU processing time. The number of inner loop iterations in all cases and all data sets are described in table 1.3. Enhanced Selection sort gives best results for inner loop iteration in all cases, while Bubble Sort gives worst results for number of inner loop iterations. E_Bubble Sort and Enhanced Bubble Sort give double performance in case of

inner loop iteration as compare to bubble sort algorithm; while Optimized bubble sort performance is good as compare to enhanced bubble sort. Merge Sort, Quick sort and Enhanced Quick Sort gives good results for all cases, while quick sort results are good as compare to enhanced quick sort algorithm. Insertion sort results are one half of enhanced bubble sort, optimized bubble sort, and selection sort. Graphical representations of all these results are also shown for each case (best case, average case, and worst case) in fig 1.2.

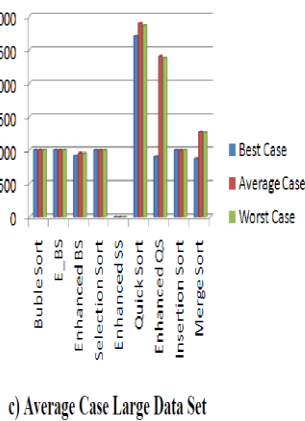
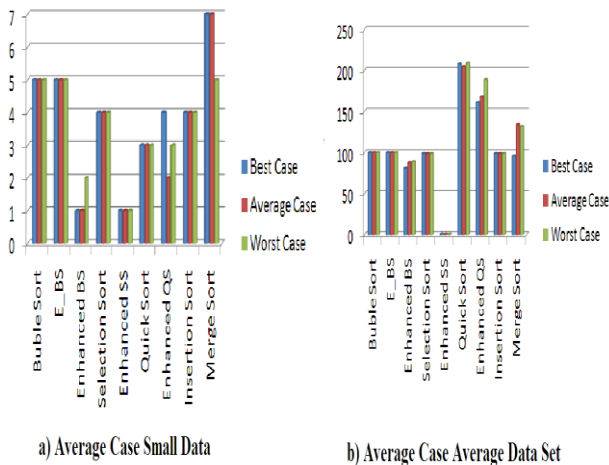


Fig 1.3: Outer Loop Number of Iterations

The number of outer loop iterations in all cases and on all data sets are described in table 1.4. Enhanced Selection sort and Enhanced Quick Sort gives best result in all cases for all data sets. The performance of Quick sort and Enhanced Quick sort is good for all cases all data sets.

Merge sort performance is good for large data sets in worst case than Quick Sort and Enhanced Quick sort. The insertion sort performance is better than

bubble sort in all cases. Overall Enhanced Selection Sort performance in outer loop is best. The graphs of number of outer loop iterations are shown in figure 1.3.

According to operating system architecture we cannot predict the exact CPU processing time of any algorithm. As some system uses parallel processing and multithreading processing. Although CPU Processing time of each algorithm in each cases are shown in table 1.5. we use ASUS EEE PC 1001p mini notebook for implementation of each algorithm. Our results shows that enhanced selection sort and enhanced quick sort take less amount of time for all cases and data sets. Original bubble sort in average takes more time than all other algorithms. the graphs of CPU Processing time is shown in figure 1.4.

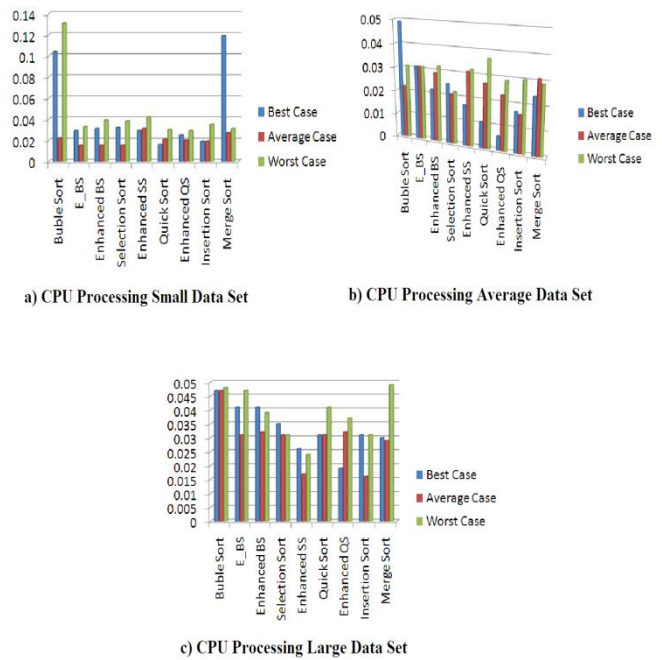


Fig 1.4: CPU Time for all data sets

5. Conclusion

In this paper the comparison of five basic and 4 enhanced sorting algorithms are described. The same number of elements is used for each data set e.g. five elements for small data set, 100 elements for medium data set and 1000 elements for large data set. As discussed in the introduction sections that each sorting has their own advantages and disadvantages in terms of CPU processing time,

Complexity, number of Inner/Outer loops iteration, and the most popular the best case, average case, and worst case performance for small, medium and large data sets. According to the literature review and simulation results it is obtained that for large data sets and worst conditions Enhanced Quick Sort is best choice, Quick sort and Merge sort are may also used for large data sets. For small data sets and worst condition enhanced selection Sort and Quick

Sort are good choice. Bubble sort is good only for small and best case, while Enhanced Bubble Sort is good for both small & average data sets and Worst conditions. Insertion Sort is good for small worst condition as compare to bubble sort.

Keep this paper in mind, a new sorting algorithm will be design for worst case large data sets in the near future.

Sorting Algorithms Complexity									
Algorithms	Best Case			Average Case			Worst Case		
	Small Data Set()	Average Data Set	Large Data Set	Small Data Set()	Average Data Set	Large Data Set	Small Data Set()	Average Data Set	Large Data Set
Bubble Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n^2)$
E BS	$O(n)$	$O(n)$	$O(n)$	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n^2)$
Enhanced BS	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n^2)$
Enhanced SS									
Insertion Sort	$O(n)$	$O(n)$	$O(n)$	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n^2)$
Quick Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$			
Enhanced QS	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$

Table 1.1: Complexity of sorting algorithms [8][9][10][15][16]

Inner Loop Iterations									
Algorithms	Best Case			Average Case			Worst Case		
	Small Data Set()	Average Data Set	Large Data Set	Small Data Set()	Average Data Set	Large Data Set	Small Data Set()	Average Data Set	Large Data Set
Bubble Sort	25	9900	999000	20	9900	999000	20	9900	999000
Enhanced BS	10	4950	499500	10	4950	499500	10	49500	500541
Optimized BS	4	4789	495584	4	4884	498510	7	4995	499475
Selection Sort	10	4950	499500	10	4950	499500	10	4981	499889
Enhanced SS	4	99	999	4	99	999	4	99	999
Quick Sort	6	479	6982	6	376	7517	4	453	7424
Enhanced QS	10	375	8143	4	420	8696	5	496	74210
Insertion Sort	0	2141	258994	1	2331	254697	1	2399	258944
Merge Sort	5	414	8735	5	538	8701	7	545	42650

Table 1.3: Inner Loop Number of Iterations

Outer Loop Iterations									
Algorithms	Best Case			Average Case			Worst Case		
	Small Data Set()	Average Data Set	Large Data Set	Small Data Set()	Average Data Set	Large Data Set	Small Data Set()	Average Data Set	Large Data Set
Bubble Sort	5	100	1000	5	100	1000	5	100	1000
E_BS	5	100	1000	5	100	1000	5	100	1000
Enhanced BS	1	81	911	1	88	955	2	89	951
Selection Sort	4	99	999	4	99	999	4	99	999
Enhanced SS	1	1	1	1	1	1	1	1	1
Quick Sort	3	208	2702	3	205	2897	3	209	2862
Enhanced QS	4	161	899	2	168	2400	3	189	2379
Insertion Sort	4	99	999	4	99	999	4	99	999
Merge Sort	7	96	871	7	134	1271	5	132	1265

Table 1.4: Outer Loop Number of Iteration

CPU Timing									
Algorithms	Best Case			Average Case			Worst Case		
	Small Data Set()	Average Data Set	Large Data Set	Small Data Set()	Average Data Set	Large Data Set	Small Data Set()	Average Data Set	Large Data Set
Bubble Sort	0.104	0.104	0.047	0.022	0.022	0.047	0.131	0.031	0.048
E_BS	0.029	0.031	0.041	0.015	0.031	0.031	0.033	0.031	0.047
Enhanced BS	0.031	0.022	0.041	0.015	0.029	0.032	0.039	0.032	0.039
Selection Sort	0.032	0.025	0.035	0.015	0.021	0.031	0.038	0.022	0.031
Enhanced SS	0.029	0.017	0.026	0.031	0.031	0.017	0.042	0.032	0.024
Quick Sort	0.016	0.011	0.031	0.021	0.027	0.031	0.030	0.037	0.041
Enhanced QS	0.025	0.006	0.019	0.02	0.023	0.032	0.029	0.029	0.037
Insertion Sort	0.019	0.017	0.031	0.019	0.016	0.016	0.035	0.030	0.031
Merge Sort	0.119	0.024	0.030	0.027	0.031	0.029	0.031	0.029	0.049

Table 1.5: CPU Processing Time Results

Bibliography

- [1] Rupesh Srivastava & Pooja Varinder Kumar Bansal, "Indexed Array Algorithm for Sorting," International Conference on Advances in Computing, Control, and Telecommunication Technologies, 2009.
- [2] J. L. Bentley and R. Sedgewick, "Fast Algorithms for Sorting and Searching Strings," ACM-SIAM SODA '97, pp. 360-369, 1997.
- [3] Thomas H. Cormen, Ronald L. Rivest, Clifford Stein Charles E. Leiserson, "Introduction To Algorithms," vol. 3rd Ed, pp. 147-150, 2009.
- [4] Hina Gull, Abdul Wahab Muzaffar Sardar Zafar Iqbal, "A New Friends Sort Algorithm," in IEEE Second International Conference on Computer Science and Information Technology, 2009.
- [5] Ping Yu, Yan Gan You Yang, "Experimental Study on the Five Sort Algorithms," in International Conference on Mechanic Automation and Control Engineering (MACE), 2011.
- [6] C.A.R. Hoare, "Algorithm 64: Quick sort," Comm. ACM , p. 321, July 1961.
- [7] Levitin A, "Introduction to the Design and Analysis of Algorithms," Addison Wesley, 2007.
- [8] Pankaj Sareen, "Comparison of Sorting Algorithms (On the Basis of Average Case)," International Journal of Advanced Research in Computer Science and Software Engineering, vol. 3, no. 3, pp. 522-532, March 2013.
- [9] Jehad Alnihoud and Rami Mansi, "An Enhancement of Major Sorting Algorithms," The International Arab Journal of Information

- Technology, vol. 7, no. 1, pp. 55-62, January 2010.
- [10] Dr. P. B. Zirra Ahmed M. Aliyu, "A Comparative Analysis Of Sorting Algorithms On Integer And Character Arrays," The International Journal Of Engineering And Science (IJES), vol. 2, no. 7, pp. 25-30, July 2013.
- [11] Parveen Kumar and Sahil Gupta Eshan Kapur, "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, pp. 61-78, June 2012.
- [12] Salman Faiz Solehria, Prof. Dr. Salim ur Rehman, Prof. Hamid Jan Sultanullah Jadoon, "Design and Analysis of Optimized Selection Sort Algorithm," International Journal of Electric & Computer Sciences IJECS-IJENS, vol. 11, no. 1, pp. 16-21, February 2011.
- [13] Mr. Imran Uddin, Mr. Simarjeet Singh Bhatia Ms. Nidhi Chhajed, "A Comparison Based Analysis of Four Different Types of Sorting Algorithms in Data Structures with Their Performances," International Journal of Advanced Research in Computer Science and Software Engineering , vol. 3, no. 2, pp. 373-381, February 2013.
- [14] C. A. R. Hoare, "Quicksort," Computer Journal, vol. 5, no. 4, pp. 10-15, 1962.
- [15] Deepti Grover Sonal Beniwal, "Comparison Of Various Sorting Algorithms: A review," International Journal of Emerging Research in Management & Technology, vol. 2, no. 5, pp. 83-86, May 2013.
- [16] Rami Mansi, "Enhanced Quicksort Algorithm," The International Arab Journal of Information Technology, vol. 7, no. 2, pp. 161-166, April 2010.

Author Profile:

Mohsin Khan

He is a student of MS (Telecommunication and Networking) at IQRA University Islamabad. He did his BS (Telecommunication and Networking) from COMSATS Institute of Information Technology, Abbottabad in 2011. He is also working as Lecturer at Lahore Garrison University. His research area includes OFDMA, Vehicular Ad-Hoc Networks (VANET), Network Security, and Programming Algorithms.

SaminaShaheen

She is a student of MS (Computer Science) at ABASYN University Islamabad Campus. She did her BSCS from University of Punjab in 2005, and MBA HRM from AIOU University at 2011. She is currently working as Lecturer Computer Science at Education Department Govt. of Punjab. Her research area includes Mobile Ad-Hoc Networks (MANET), Network Security, and Programming Algorithms.

Furqan Aziz Qureshi

He is working as Lab Engineer at Center for Emerging Sciences, Engineering and Technology, Islamabad. He got his BS Degree in Computer Engineering from COMSATS Institute of Information Technology, Abbottabad in 2011. His research area includes Digital Systems, Embedded Systems, Programming Techniques and Algorithms.