PROPOSAL OF A TWO WAY SORTING ALGORITHM AND PERFORMANCE COMPARISON WITH EXISTING ALGORITHMS

Eshan Kapur[1], Parveen Kumar[2] and Sahil Gupta[3]

Department of Computer Science,
Lovely Professional University,
Phagwara, Punjab
eshankapur@gmail.com[1]
parveen.it@gmail.com[2]
sahil.gupta2688@gmail.com[3]

ABSTRACT

An algorithm is any well-defined procedure or set of instructions, that takes some input in the form of some values, processes them and gives some values as output. Sorting involves rearranging information into either ascending or descending order. Sorting is considered as a fundamental operation in computer science as it is used as an intermediate step in many operations. A new sorting algorithm namely 'An End-to-End Bi-directional Sorting (EEBS) Algorithm' is proposed to address the shortcomings of the current popular sorting algorithms. The goal of this research is to perform an extensive empirical analysis of the newly developed algorithm and present its functionality. The results of the analysis proved that EEBS is much more efficient than the other algorithms having O(n^2) complexity, like bubble, selection and insertion sort.

KEYWORDS

End-to-End Bi-directional sort (EEBS), algorithms, selection sort, bubble sort, insertion sort.

1. INTRODUCTION

Algorithms have a vital and key role in solving the computational problems, informally an algorithm is a well-defined computational procedure that takes input and produces output. Algorithm is a tool or a sequence of steps to solve the computational problems[1]. The existence of algorithms goes way back as they were in existence even before the existence of computers.
There are various methodologies and techniques based on which various different kinds of algorithms are designed. Out of all these problem solving algorithms, let us talk about the sorting algorithms. In case of sorting, it is required to arrange a sequence of numbers into a given order, generally non-decreasing. The sorting problem is countered with quite often in practice and it acts as a fertile ground for the introduction of many standardized design techniques and analysis tools. The formal definition of the sorting problem is as follows:

Input: A sequence having n numbers in some random order (a1, a2, a3, ….. an)

Output: A permutation (a'1, a'2, a'3, ….. a'n) of the input sequence such that a’1 ≤ a’2 ≤ a’3 ≤ ….. a’n

For instance, if the given input of numbers is (59, 41, 31, 41, 26, 58), then the output sequence returned by a sorting algorithm will be (26, 31, 41, 41, 58, 59)[7].

In practical, some records in the data which are to be sorted according to their keys. An n records sequence (R1, R2,...,Rn), whose corresponding sequence of keywords is (K1, K2,...,Kn) is required to be sorted to identify a permutation p1, p2,..., pn of the current subscript sequence 1,2, ...,n, so that the appropriate keywords meet the decreasing or increasing relationship, that is (Kp1 ≤ Kp2 ≤... ≤ Kpn) in order to get an ordered record sequence by their keywords. Such process is known as sort[3].

Sorting is considered as a fundamental operation in computer science as it is used as an intermediate step in many programs. Due to this, the number of sorting algorithms available at our disposal is pretty large in number. Out of these available algorithms, which one is considered as the best one for a particular application further has its dependency on various other factors which include:

- The size of the array or sequence to be sorted,
- The extent up to which the given input sequence is already sorted,
- The probable constraints on the given input values,
- The system architecture on which the sorting operation will be performed,
- The type of storage devices to be used: main memory or disks[7].

Almost all the available sorting algorithms can be categorized into two categories based on their difficulty. The complexity of an algorithm and its relative effectiveness are directly correlated. A standardized notation i.e. Big O(n), is used to describe the complexity of an algorithm. In this notation, the O represents the complexity of the algorithm and n represents the size of the input data values. The two groups of sorting algorithms are $O(n^2)$, which includes the bubble, insertion, selection sort and O(nlogn) which includes the merge, heap & quick sort.

Two categories of sort algorithms were classified according to the records whether stored in the main memory or secondary memory. One category is the internal sort which stores the records in the main memory. Another is the external sort which stores the records in the hard disk because of the records’ large space occupation[6].

The paper is organized into following sections. In section 2, a discussion about the importance of sorting is made, along with some of the widely used algorithms, like selection, bubble and insertion sort. In section 3, the working of the proposed algorithm is discussed. The complexity of the algorithm is also discussed in this section. Next section i.e. 4, discusses the comparative analysis of the proposed algorithm with the existing ones. The last section gives a conclusions about the performance and hence, the results achieved by the proposed algorithm.

2. SORTING ALGORITHMS

From the last few decades, the discipline of computer science has emerged as a discipline whose understanding and in-depth knowledge has become crucial for the successful completion of many engineering projects. There exist few concepts and methods which are of great importance in various engineering projects. Sorting and searching of elements in a given set of record is one of them. In general, sorting is considered as a process of rearranging a given set of objects in a

specific order. The major significance of sorting is that it finds its application to facilitate the later search for members of the sorted set[7].
Hence sorting is an almost universally performed and hence, considered as a fundamental activity. The usefulness and significance of sorting is depicted from the day to day application of sorting in real-life objects.

For instance, objects are sorted in:

- Telephone Directories
- Income Tax Files
- Tables of Contents
- Libraries
- Dictionaries

Above mentioned are a few objects in which searching and retrieval of records is done very often. Even in our daily life, children are confronted with problems like sorting their things, like toys, in some order. Hence, they learn the problem of sorting even before learning anything about computing or arithmetic. This shows the relevance and essentiality of sorting, especially in the field of data processing. There are quite a few techniques in computer science that directly or indirectly does not have some connection with sorting algorithms.

The structure of the data to be processed, acts as a main factor for choosing the type of the algorithm to be used. Hence, keeping this fact into consideration, the sorting methods can be classified into two categories. These are:

- Sorting of array
- Sorting of sequential files

The two types are distinguished as internal and external sorting. In case of internal sorting arrays are used whose storage is generally done in the high speed, fast, random-access internal memory. But in case of external sorting, which involves files, are kept on the more spacious but comparatively slower external stores.

This distinction is important and is pretty obvious by considering the fact that sorting numbered objects as in array, structures them as individual objects which are accessible to the user. Structuring the objects as a file means that out of each segment only the object on the top end is accessible.

2.1. Existing Sorting Algorithms

2.1.1 Bubble Sort

Bubble sort is a basic sorting algorithm that performs the sorting operation by iteratively comparing the adjacent pair of the given data items and swaps the items if their order is reversed. There is a repetition of the passes through the list is sorted and no more swaps are needed. It is a simple algorithm but it lacks in efficiency when the given input data set is large.

The worst case as well as average case complexity of bubble sort is $O(n^2)$, where n represents the total number of items in the given array to be sorted. While comparing with other sorting algorithms, results show that there are many of such algorithms which perform better in their worst and/or their average case. Due to this reason, it is not considered as a practical sorting algorithm when n is substantially large. There is one major advantage that bubble sort has over the other better performing algorithms is that it has the ability to detect whether the given list is trivially sorted or not. Hence, it has the best performance i.e. complexity of O(n) in case of an already sorted list.

The algorithm for bubble sort having DATA as an array with N elements.is as follows[8]:

BUBBLE (DATA, N)

```
  i.  Repeat Steps ii and iii for K=1 to N-1
 ii.  Set PTR = 1
iii.  Repeat while PTR<= N-K

        a.  If DATA[PTR] > DATA[PTR+1], then
        b.  Swap DATA[PTR] and DATA[PTR+1]
        c.  Set PTR = PTR+1
 iv.  Exit.
```

2.1.2 Selection Sort

Selection sort is noted for its simplicity and even performs better than many complicated algorithms in certain situations especially where auxiliary memory is limited as it is an in-place comparison sort. It starts by selecting the minimum value in the given list, as the name suggests. After selecting the value, it swaps with the value in the first position of the list. In this way, the given list is divided into two parts consisting of the sub-list of items already sorted from left to right and the other sub-list of items remaining to be sorted. It repeats this step iteratively and places each element as its proper position.

It has O(n^2) time complexity in best, worst and average case, which makes it somewhat inefficient when applied on large lists. In this case, it even has worse performance than somewhat similar insertion sort.

The algorithm for selection sort having DATA as an array with N elements is as follows[8]:
SELECTION SORT (DATA, N)

```
i. Repeat Steps ii & iii for K=1 to N-1
        ii. Set MIN = DATA[K] and LOC = K
        iii. Repeat for J= K+1 to N
        iv. If MIN > DATA[J] then
                a. MIN= DATA [J]
                b. LOC = DATA [J]
                c. LOC = J
        v. Set TEMP = DATA[K]
        vi. DATA [K] = DATA[LOC]
        vii. DATA[LOC] = TEMP
viii. Exit
```

2.1.3 Insertion Sort

Insertion sort is a simple algorithm which considers the given elements of array, one at a time. It inserts each element in its proper place by comparing that element with the elements placed to the left of it. Hence, the array is bifurcated into two sublists, a sorted one and unsorted one. At the beginning of sorting process, the sorted sublist contains first element of the array and unsorted list contains the rest of the array. At every step, algorithm takes first element in the unsorted part and inserts it to the right place of the sorted one. The algorithm stops when no element is left in the unsorted sublist.

The running time of insertion sort in its best case is linear i.e., O(n). In worst case, much effort is required because for each iteration done by the inner loop, it has to scan and then shift all of the sorted subsection of the given array before next element could be inserted. Hence, in this case the running time of insertion sort is quadratic i.e., $O(n^2)$. The average case is same as that of worst case due to which insertion sort becomes impractical for sorting large arrays but for sorting small arrays, it is one of the best performing algorithms.

The algorithm for insertion sort having DATA as an array with N elements.is as follows[8]:
INSERTION (DATA, N)

```
i. Set A[0] = -∞
ii. Repeat Steps iii to v for K = 2 to N
        iii. Set TEMP = DATA[K] and PTR = K-1
    v.  Repeat while TEMP < DATA[PTR]

                a. Set DATA[PTR+1] = A[PTR]
                b. Set PTR =PTR – 1
        v. Set DATA[PTR+1] = TEMP
vi. Exit.
```

3. AN END-TO-END BIDIRECTIONAL SORTING ALGORITHM

3.1 Introduction

There had been various authors who had made continuous efforts for increasing the efficiency and performance of the sorting process. To mention a few, there is a new friends sort algorithm[2] and OSSA[4] which are based on the selection sort. The proposed algorithm is based on bubble sort as its working in the second step of the operation is somewhat, similar to the bubble sort.
The proposed algorithm works in two steps:

1. In first step, the first and the last element of the array is compared. If the first element is larger than the last element, then swapping of the elements is required. The position of the element from front end and element from the rear end of the array are stored in variables which are increased (front end) and decreased (rear end) as the algorithm progresses.

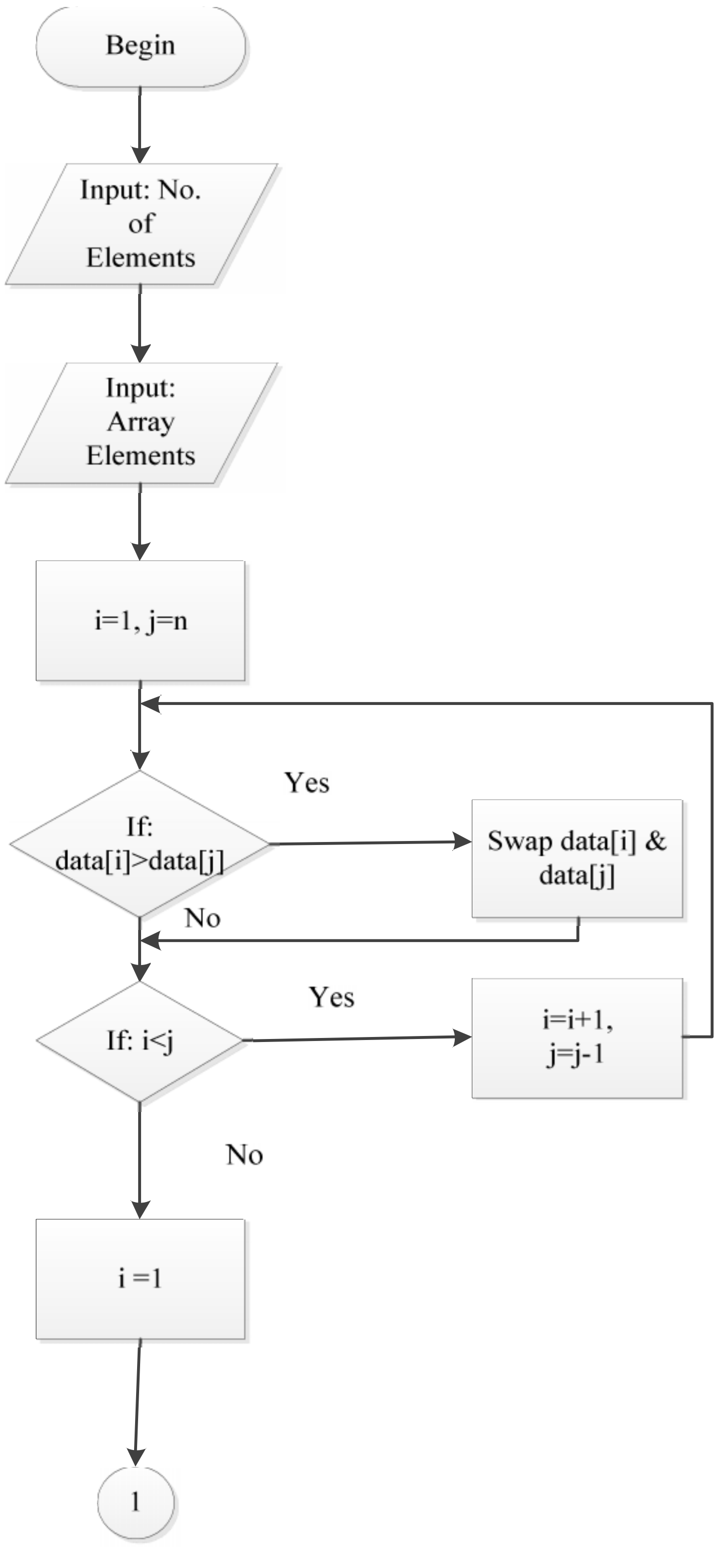
2. In the second step, two adjacent elements from the front and rear end of the array are taken and are compared. Swapping of elements is done if required according to the order. Four variables are taken which stores the position of two front elements and two rear elements to be sorted.

3.2 Algorithm

EEBS (arr, n)

```
i, j, n, temp, exc ← 1,i ← 1,j ← n
while(i<j)
{
    if(arr[i]>arr[j])
    {
        temp ← arr[i]
        arr[i] ← arr[j]
        arr[j] ← temp
    }
    i ← i+1
    j ← j-1
}
for (i ← 1 to n/2 && exc)
{
    exc ← 0
    for (j ← i to n-i)
    {
        if(arr[j]>arr[j+1])
        {
            temp ← arr[j]
            arr[j] ← arr[j+1]
            arr[j+1] ← temp
            exc ← 1
        }
        if(arr[n-j]>arr[n-j+1])
        {
            temp ← arr[n-j]
            arr[n-j] ← arr[n-j+1]
            arr[n-j+1] ← temp
            exc ← 1
        }
    }
}
```

3.3 Flowchart



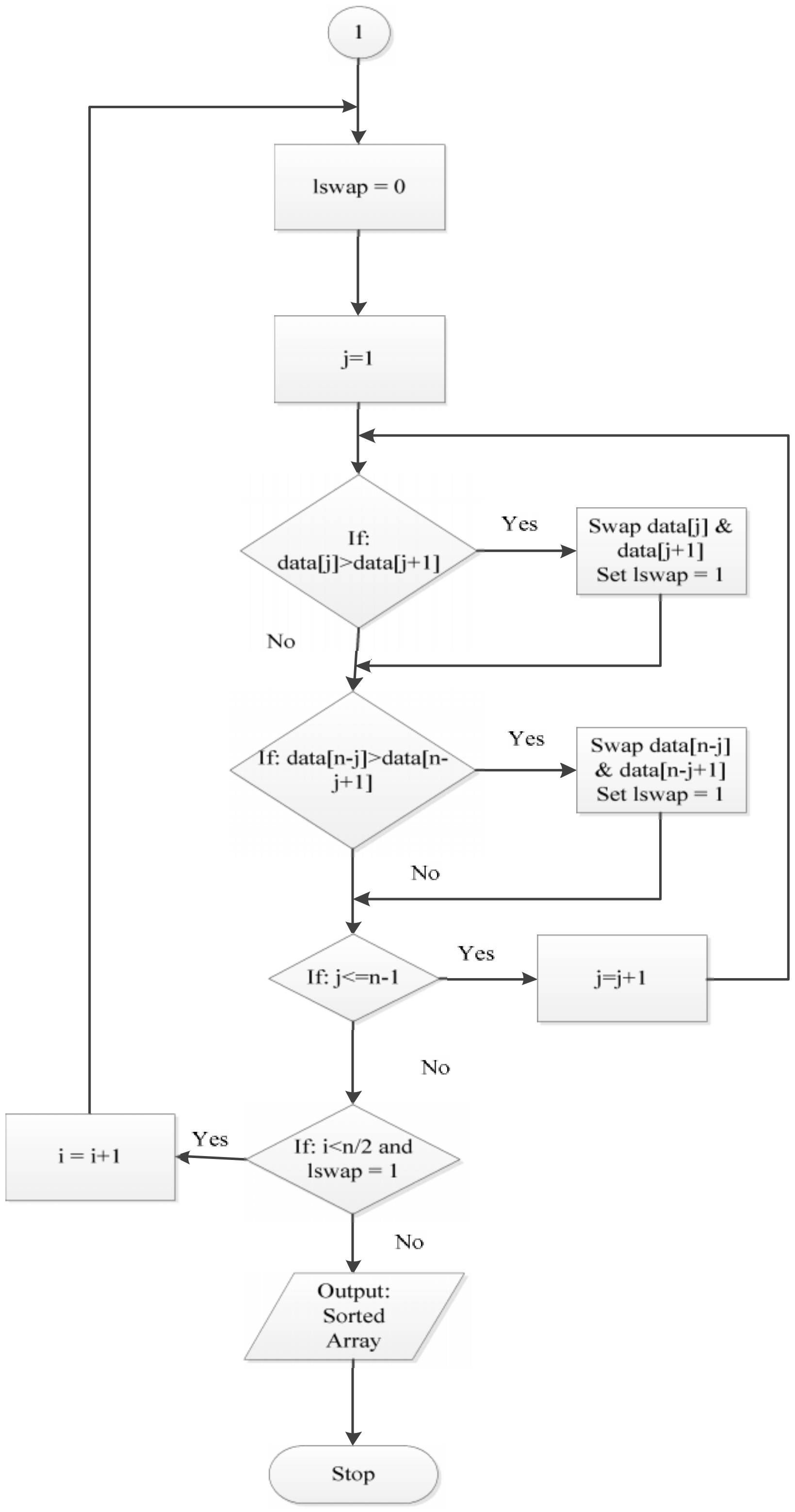


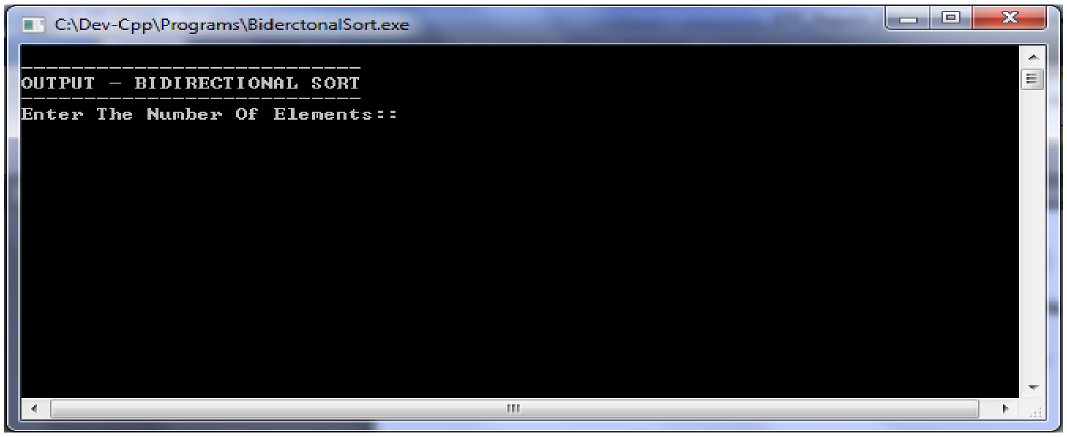
Figure 1. Flowchart of EEBS

3.4 Implementation of proposed Algorithm

The proposed algorithm has been implemented in C++ programming language. The tool used for the implementation is Dev-C++ version 4.9.9.2 by Bloodshed Software. It is a free of cost integrated development environment (IDE) and distributed under the GNU General Public License for programming in C and C++. The IDE is written in Delphi.

The following screenshots shows the output of the execution of the code implemented. These are as follows:

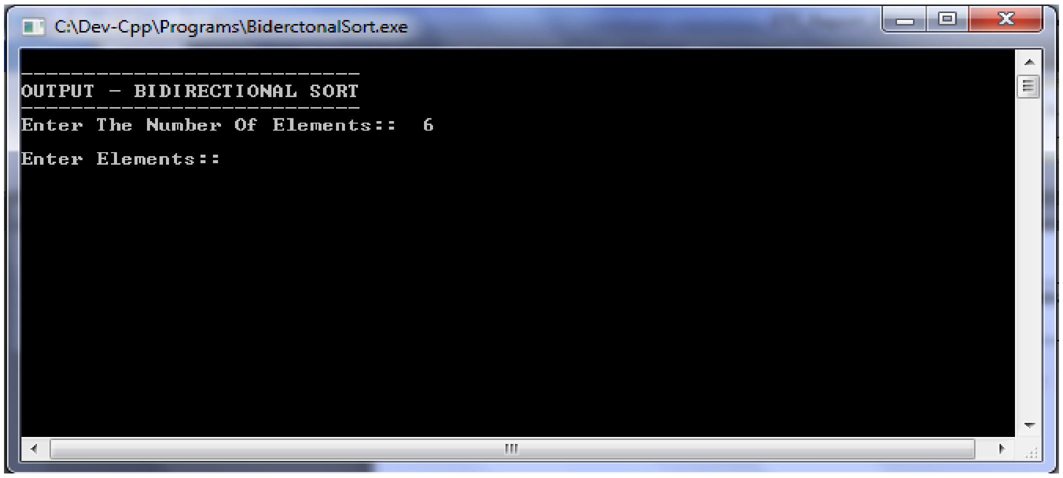
i. First of all, the output console asks the user for entering the number of elements that are to be sorted. The entered number is stored in a variable n which acts as the size of the array to be sorted.



```
----------------------------
OUTPUT - BIDIRECTIONAL SORT
----------------------------
Enter The Number Of Elements::
```

Figure 2a. Execution Output Console

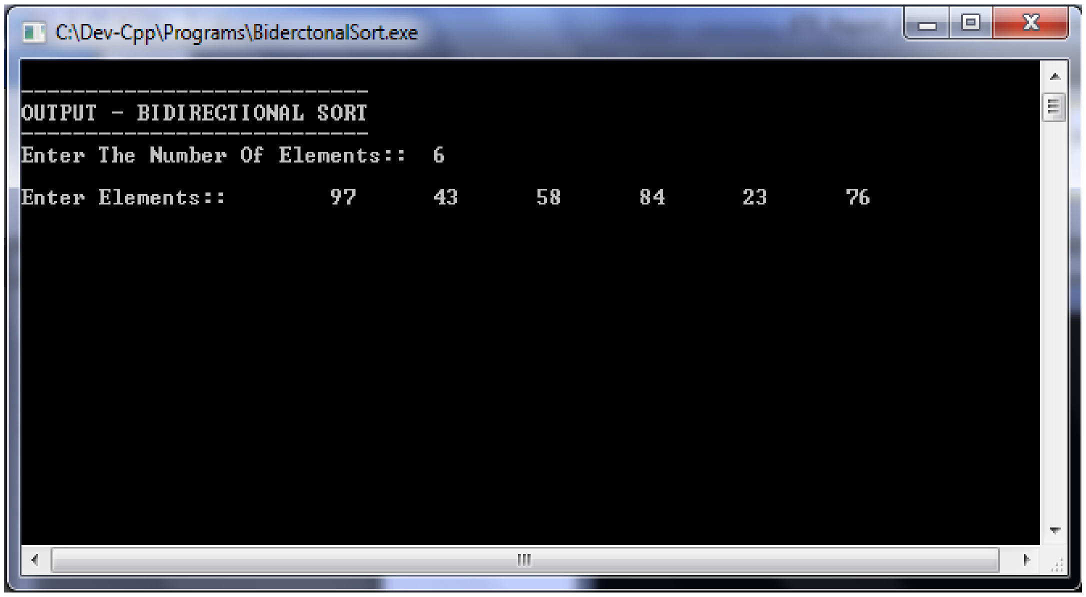
ii. Let us suppose that the user enters the number of elements to be sorted as 6. The console window shows the current state of the execution. After giving the input of the number of elements, the user is required to input the elements of the array. These are the elements on which the sorting operations will be performed.



```
----------------------------
OUTPUT - BIDIRECTIONAL SORT
----------------------------
Enter The Number Of Elements::  6

Enter Elements::
```

Figure 2b. Execution Output Console

iii. The elements entered by the user are shown below. As per the input given, the algorithm will

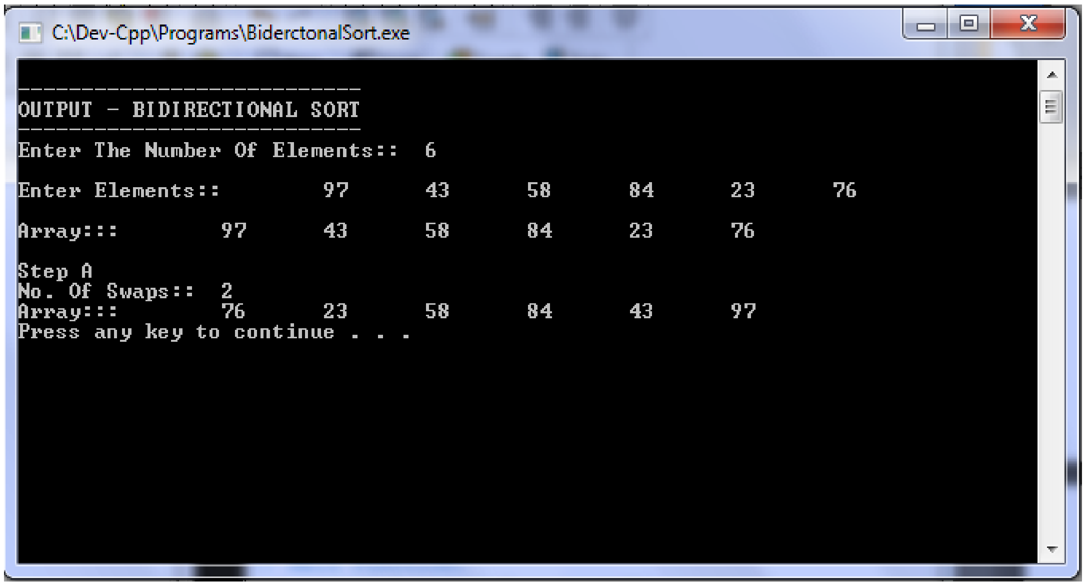


```
C:\Dev-Cpp\Programs\BiderctonalSort.exe

----------------------------
OUTPUT - BIDIRECTIONAL SORT
----------------------------
Enter The Number Of Elements::  6

Enter Elements::        97      43      58      84      23      76
```

perform an average case execution.

Figure 2c. Execution Output Console

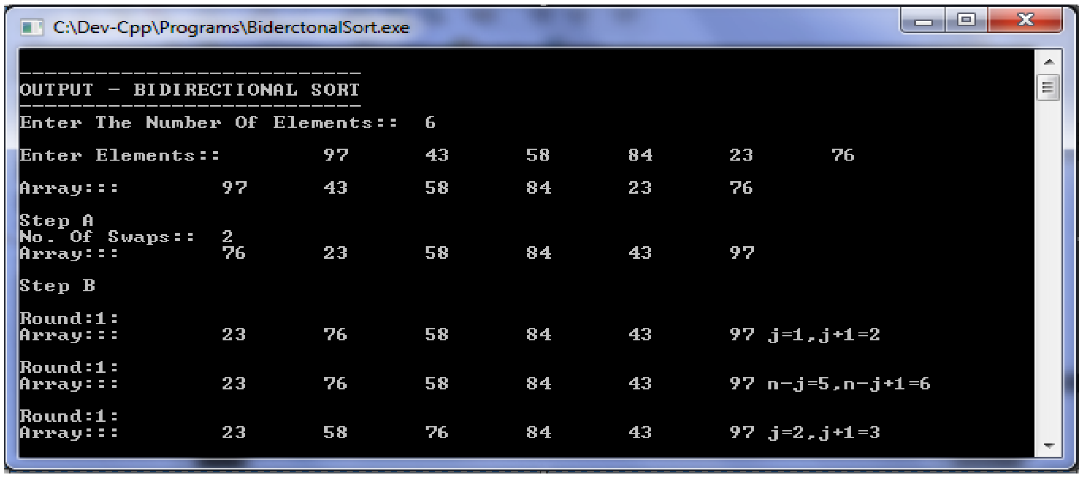
iv. As per the working of the algorithm discussed above, the algorithm works in a two steps. The state of the array after the execution of Step A is as follows.



```
C:\Dev-Cpp\Programs\BiderctonalSort.exe

----------------------------
OUTPUT - BIDIRECTIONAL SORT
----------------------------
Enter The Number Of Elements::  6

Enter Elements::        97      43      58      84      23      76

Array:::        97      43      58      84      23      76

Step A
No. Of Swaps::  2
Array:::        76      23      58      84      43      97
Press any key to continue . . .
```

Figure 2d. Execution Output Console

v. Although the above shown output console displays the state of the array after Step A only, but the real execution of the code is continuous for both the steps. The above output shows the intermediate state of the array, which has intentionally been shown for understanding of the flow of the code. The actual output is as follows.

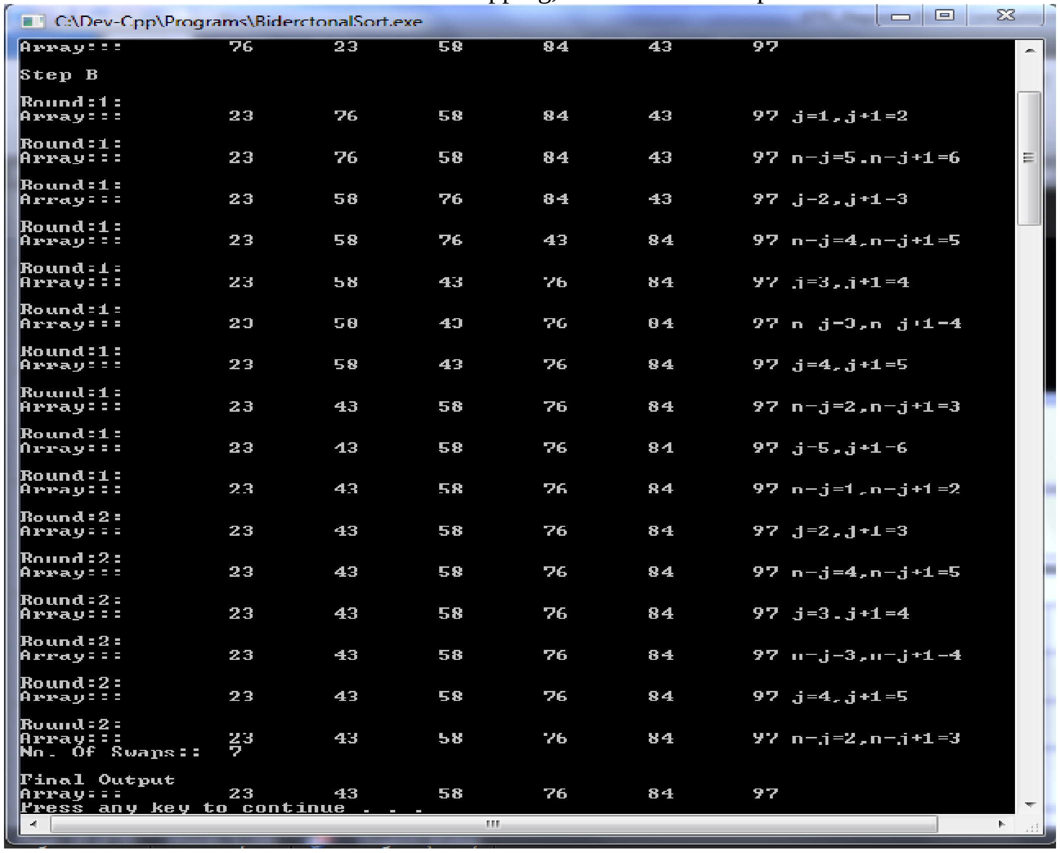


C:\Dev-Cpp\Programs\BiderctonalSort.exe

```
----------------------------
OUTPUT - BIDIRECTIONAL SORT
----------------------------
Enter The Number Of Elements::  6

Enter Elements::        97      43      58      84      23      76

Array:::        97      43      58      84      23      76

Step A
No. Of Swaps::  2
Array:::        76      23      58      84      43      97

Step B

Round:1:
Array:::        23      76      58      84      43      97 j=1,j+1=2

Round:1:
Array:::        23      76      58      84      43      97 n-j=5,n-j+1=6

Round:1:
Array:::        23      58      76      84      43      97 j=2,j+1=3
```

Figure 2e. Execution Output Console

vi. The following console window shows the execution of Step B of the algorithm. Front and Rear pointers are shown along with the array. These pointers points to the index of the array element which is under consideration for swapping, based on the comparisons.



C:\Dev-Cpp\Programs\BiderctonalSort.exe

```
Array:::        76      23      58      84      43      97

Step B

Round:1:
Array:::        23      76      58      84      43      97 j=1,j+1=2

Round:1:
Array:::        23      76      58      84      43      97 n-j=5,n-j+1=6

Round:1:
Array:::        23      58      76      84      43      97 j=2,j+1=3

Round:1:
Array:::        23      58      76      43      84      97 n-j=4,n-j+1=5

Round:1:
Array:::        23      58      43      76      84      97 j=3,j+1=4

Round:1:
Array:::        23      58      43      76      84      97 n-j=3,n-j+1=4

Round:1:
Array:::        23      58      43      76      84      97 j=4,j+1=5

Round:1:
Array:::        23      43      58      76      84      97 n-j=2,n-j+1=3

Round:1:
Array:::        23      43      58      76      84      97 j=5,j+1=6

Round:1:
Array:::        23      43      58      76      84      97 n-j=1,n-j+1=2

Round:2:
Array:::        23      43      58      76      84      97 j=2,j+1=3

Round:2:
Array:::        23      43      58      76      84      97 n-j=4,n-j+1=5

Round:2:
Array:::        23      43      58      76      84      97 j=3.j+1=4

Round:2:
Array:::        23      43      58      76      84      97 n-j=3,n-j+1=4

Round:2:
Array:::        23      43      58      76      84      97 j=4,j+1=5

Round:2:
Array:::        23      43      58      76      84      97 n-j=2,n-j+1=3
No. Of Swaps::  7

Final Output
Array:::        23      43      58      76      84      97
Press any key to continue . . .
```

Figure 2f. Execution Output Console

3.5 Complexity Analysis

The general working and the implementation details of the proposed algorithm is already discussed in detail. Another important aspect about the algorithm is its complexity analysis. The following table shows the detailed evaluation of each step of the algorithm and computes the complexity of the proposed algorithm. It takes the cost which is incurred by the system to execute each step and the no. of times that particular step will be execution in a single complete run of the algorithm[5].

Table 1. Complexity Analysis of algorithm

Step No.	Iteration	Cost	Times
1.	i, j, n, temp, exc ← 1	C1	1
2.	i ← 1		
3.	j ← n		
4.	while(i<j)	C2	n/2 +1
5.	if(arr[i]>arr[j])	C3	n/2
6.	temp ← arr[i]	C4	$\sum_{k=1}^{n/2} t$
7.	arr[i] ← arr[j]		
8.	arr[j] ← temp		
9.	i ← i+1	C5	n/2
10.	j ← j-1		
11.	for i ← 1 to n/2 && exc	C6	1
12.	exc ← 0	C7	n/2
13.	for j ← i to n-i		
14.	if(arr[j]>arr[j+1])	C8	$\sum_{k=0}^{n/2-1} (2k+1).t$
15.	temp ← arr[j]		
16.	arr[j] ← arr[j+1]		
17.	arr[j+1] ← temp		
18.	exc ← 1		
19.	if(arr[n-j]>arr[n-j+1])		
20.	temp ← arr[n-j]		
21.	arr[n-j] ← arr[n-j+1]		
22.	arr[n-j+1] ← temp		
23.	exc ← 1		

Note: t=1 when if statement is true, else t=0.

C4 $\sum_{k=1}^{n/2} t$ = this evaluates to a constant value based on the value of t (either 1 or 0).

C8 $\sum_{k=0}^{n/2-1} (2k+1).t$

Suppose for t=1 we have

$$C8 \sum_{k=0}^{n/2-1} (2k+1) = C8\ 2\sum_{k=0}^{n/2-1} k + C8 \sum_{k=0}^{n/2-1} 1$$

$= C8\ 2(((n/2-1)*((n/2-1)+1))/2) + C8\ (n/2-1)$
[By Applying $1+2+3+\ldots n = (n(n+1)/2)$]
$= C8\ ((n^2-2n)/4) + (n/2-1)$
Now calculating for all the steps,
$T(n) = C1(1) + C2(n/2 +1) + C3(n/2) + C4 + C5(n/2) + C6(1) + C7(n/2) + C8\ (((n^2-2n)/4) + (n/2-1))$
$= n^2/2(C8) + (n/2)(C2 + C3 + C5 + C7) + (C1 + C2 + C4 + C6)$
Let $a = C8/2$
$b = (C2 + C3 + C5 + C7)/2$
$c = (C1 + C2 + C4 + C6)$
$T(n) = an^2 + bn + c$
Therefore, by ignoring the constants a, b, c and the lower terms of n, and taking only the dominant term, i.e. n^2 the running time of the algorithm is,

$$T(n) = O(n^2)$$

4. Comparison with Other Algorithms

In this section the performance of the proposed algorithm is evaluated by comparing it with other existing algorithms. Let the given set of elements are 97, 43, 58, 84, 23, 76.

4.1 EEBS

Table 2. Working of End-to-End Bidirectional Sort

Step No.	Elements						Pointer
1.	**97**	43	58	84	23	**76**	x=1, y=6
2.	76	**43**	58	84	**23**	97	x=2, y=5
3.	76	23	**58**	**84**	43	97	x=3, y=4
4.	**76**	**23**	58	84	**43**	**97**	p=1, q=2 x=6, y=5
5.	23	**76**	**58**	**84**	**43**	97	p=2, q=3 x=5, y=4
6.	23	58	**76**	**43**	84	97	p=3, q=4 x=4, y=3
7.	23	**58**	**43**	**76**	**84**	97	p=4, q=5 x=3, y=2
8.	**23**	**43**	58	76	**84**	**97**	p=5, q=6 x=2, y=1
9.	23	**43**	**58**	**76**	**84**	97	p=2, q=3 x=5, y=4
10.	23	43	**58**	**76**	84	97	p=3, q=4 x=4, y=3
11.	23	**43**	**58**	**76**	**84**	97	p=4, q=5 x=3, y=2
Sorted:	23	43	58	76	84	97	

The tables shown below evaluate the same set of data on three different algorithms i.e. Bubble Sort, Selection Sort & Insertion Sort.

4.2 Bubble Sort

Table 3. Working of Bubble Sort

Step No.	Elements					
1.	**97**	**43**	58	84	23	76
2.	43	**97**	**58**	84	23	76
3.	43	58	**97**	**84**	23	76
4.	43	58	84	**97**	**23**	76
5.	43	58	84	23	**97**	**76**
6.	**43**	**58**	84	23	76	97
7.	43	**58**	**84**	23	76	97
8.	43	58	**84**	**23**	76	97
9.	43	58	23	**84**	**76**	97
10.	**43**	**58**	23	76	84	97
11.	43	**58**	**23**	76	84	97
12.	43	23	**58**	**76**	84	97
13.	**43**	**58**	23	76	84	97
14.	43	**58**	**23**	76	84	97
15.	**43**	**23**	58	76	84	97
16.	23	43	58	76	84	97
Sorted:	23	43	58	76	84	97

4.3 Selection Sort

Table 4. Working of Selection Sort

Step No.	Elements					
1.	**M=97**	43	58	84	23	76
2.	97	**M=43**	58	84	23	76
3.	97	M=43	**58**	84	23	76
4.	97	M=43	58	**84**	23	76
5.	97	43	58	84	**M=23**	76
6.	97	43	58	84	M=23	**76**
7.	23	43	58	84	97	76
8.	23	**M=43**	58	84	97	76
9.	23	M=43	**58**	84	97	76
10.	23	M=43	58	**84**	97	76
11.	23	M=43	58	84	**97**	76
12.	23	M=43	58	84	97	**76**
13.	23	43	**M=58**	84	97	76
14.	23	43	M=58	**84**	97	76
15.	23	43	M=58	84	**97**	76
16.	23	43	M=58	84	97	**76**
17.	23	43	58	**M=84**	97	76
18.	23	43	58	84	**97**	76
19.	23	43	58	84	97	**M=76**
20.	23	43	58	76	**M=97**	84
21.	23	43	58	76	97	**M=84**
22.	23	43	58	76	84	97
Sorted:	23	43	58	76	84	97

4.4 Insertion Sort

Table 5. Working of Insertion Sort

Step No.	**Elements**					
1.	**97**	**43**	58	84	23	76
2.	43	**97**	**58**	84	23	76
3.	**43**	**58**	97	84	23	76
4.	43	58	**97**	**84**	23	76
5.	43	**58**	**84**	97	23	76
6.	43	58	84	**97**	**23**	76
7.	43	58	**84**	**23**	97	76
8.	43	**58**	**23**	84	97	76
9.	**43**	**23**	58	84	97	76
10.	23	43	58	84	**97**	**76**
11.	23	43	58	**84**	**76**	97
12.	23	43	**58**	**76**	84	97
Sorted:	23	43	58	76	84	97

The above given comparisons are restricted to only a random set of six numbers. For further evaluations, different values of the random set of numbers is taken into consideration.

4.5 Average Case Analysis

Average case in a sorting algorithm refers to the situation when the given set of numbers is in a random manner. These results are achieved by using the rand() function in C++ defined in the stdlib header file. The results obtained may vary for some extent when the input of the random numbers change, depending upon the extent to which the numbers are already sorted. n in the table shown below refers to the no. of elements to be sorted.

Table 6. No. of swaps/steps in average case

	n=10	**n=50**	**n=100**
Bubble Sort	21	675	2414
Selection Sort	22	521	2013
Insertion Sort	28	701	2511
EEBS	16	299	1356

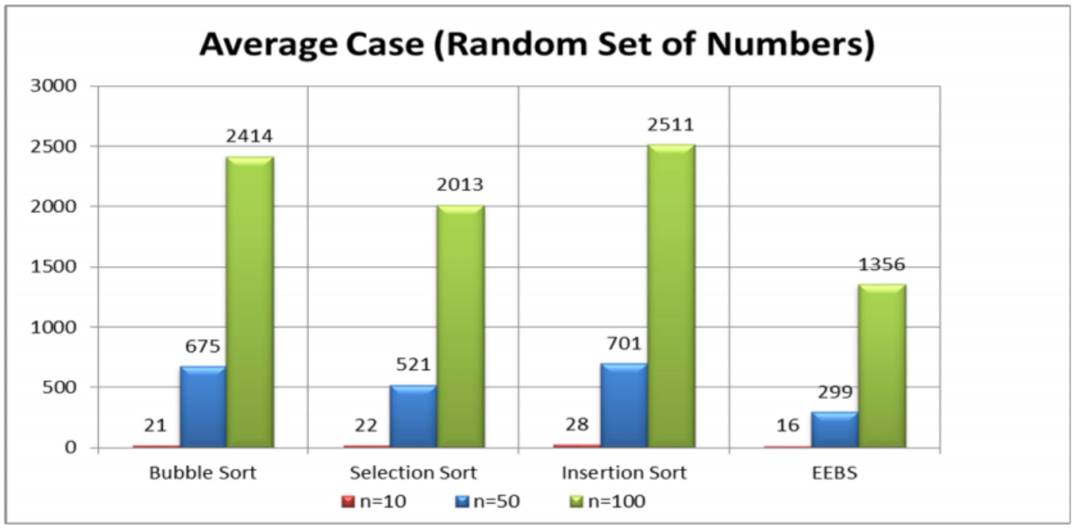


Figure 3. Average Case Analysis Results

4.6 Worst Case Analysis

Worst case in a sorting algorithm refers to the situation when the given set of numbers is in decreasing order while the required set has to be in increasing order. These results are achieved by using the rand() function in C++ defined in the stdlib header file to get a random set of numbers and then changing them in decreasing order. After that, these four algorithms are applied to change them in increasing order. The results obtained may vary for some extent when the input of the random numbers change, depending upon the data set. n in the table shown below refers to the no. of elements to be sorted.

Table 7. No. of swaps/steps in worst case

	n=10	**n=50**	**n=100**
Bubble Sort	45	1214	4903
Selection Sort	34	1088	3965
Insertion Sort	54	1263	4804
EEBS	5	25	50

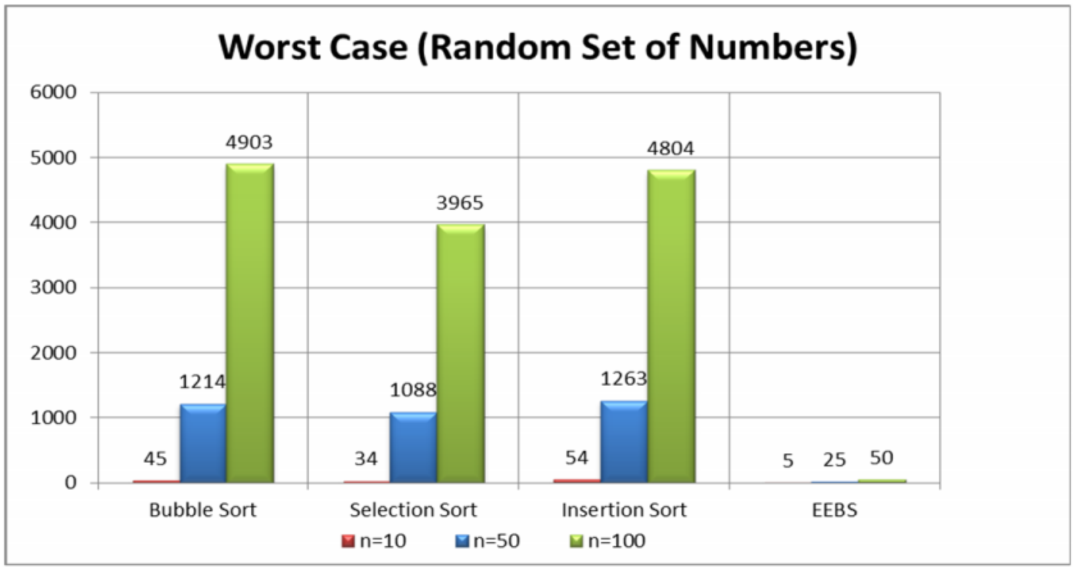


Figure 4. Worst Case Analysis Results

5. Conclusions And Future Scope

As the above shown results proves, EEBS works remarkably well in worst case scenarios. The first step of the algorithm places each element as its proper position with minimum number of comparisons and swaps. Even in the average case, the number of swaps/steps is less as compared to bubble, insertion and selection sort. This is due to the fact that even in the average case the first step in the algorithm places most of the elements at their proper positions, while the second step performs the remaining swap by working in both front as well as rear end of the given array. Although the complexity is same as the other algorithms being compared i.e. $O(n^2)$ but the performance is exceptionally well.

As the algorithm has already proved itself well in the case of integer values, this could be applied to other complex data types and its performance could be evaluated. One such data type could be characters or even strings. This could be helpful in sorting the character strings and be applied in, for example, contacts in mobile phones, words in dictionary etc.
The two dimensional matrices are being used to store and represent massive data in many fields, such as meteorology, engineering design and management. Efficient two-dimensional sorting algorithms are needed when these data are sorted by computers. The proposed algorithm could act as a base for the development of effective two dimensional sorting algorithms that could serve the purpose.

For further evaluation purposes, the algorithm can be compared with merge, cocktail or heap sort etc. Although these algorithm have their own specific implementation constraints, performance analysis with these or some other existing sorting algorithms will give be useful.

References

[1] Varinder Kumar Bansal, Rupesh Srivastava & Pooja, (2009) “Indexed Array Algorithm for Sorting”, International Conference on Advances in Computing, Control, and Telecommunication Technologies.
[2] Sardar Zafar Iqbal, Hina Gull, Abdul Wahab Muzaffar, (2009) “A New Friends Sort Algorithm”, IEEE Second International Conference on Computer Science and Information Technology.

[3] Wang Min (2010) “Analysis on 2-Element Insertion Sort Algorithm”, International Conference on Computer Design And Appliations (ICCDA).
[4] Sultanullah Jadoon , Salman Faiz Solehria, Prof. Dr. Salim ur Rehman, Prof. Hamid Jan , (2011) “Design and Analysis of Optimized Selection Sort Algorithm” International Journal of Electric & Computer Sciences IJECS-IJENS, Vol: 11 No: 01.
[5] Sultanullah Jadoon, Salman Faiz Solehria, Mubashir Qayum, (2011) “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.
[6] You Yang, Ping Yu, Yan Gan, (2011) “Experimental Study on the Five Sort Algorithms”, International Conference on Mechanic Automation and Control Engineering (MACE).
[7] Charles E. Leiserson, Thomas H. Cormen, Ronald L. Rivest, Clifford Stein (2009) “Introduction To Algorithms”, MIT Press, 3rd Ed. p.5-7, 147-150.
[8] Seymour Lpischutz, G A Vijayalakshmi Pai (2006) “Data Structures”, Tata McGraw-Hill Publishing Company Limited, p.4.11, 9.6, 9.8.

Authors Biography/Details

Mr. Eshan Kapur
M.Tech
Department of Computer Science
Lovely Professional University
Phagwara, Punjab

Mr. Parveen Kumar
Assistant Professor
Department of Computer Science
Lovely Professional University
Phagwara, Punjab

Mr. Sahil Gupta
M.Tech
Department of Computer Science
Lovely Professional University
Phagwara, Punjab