Sorting Data with One Swap Operation

Rama Sushil[1], Sushil Kumar[2] & Anuj Kumar[3]

[1&3]Department of IT, SGRRITS, Dehradun, Uttarakhand, India
[2]Wadia Institute of Himalayan Geology, Dehradun, Uttarakhand, India
E-mail : ramasushil@yahoo.co.in[1], sushil_rohella@yhoo.co.in[2], anuj@rediffmail.com[3]

Abstract – **Sorting is a method that arranges the list of elements in ascending or descending order. It is frequently used in a large variety of important applications used by schools, hospitals, banks and in many other organizations. There are many sorting methods having their own Time and Space Complexity. This paper presents a novel sorting method named as SOS i.e. Swap Once and Sort. This method provides the correct position to an element by only one swap operation. It is based on the concept that, in ascending order list, all the elements after a particular element, will be greater than that element and vice versa for descending order list. Its algorithm is developed and then it is implemented in 'C'. Its C program is run for random data set of size 1000-30000 with the increment of 5000. For comparison purpose two parameters used are number of swap operations and CPU time taken to sort the given list. Comparison is performed with three existing popular sorting methods Bubble sort, Selection sort and Insertion sort. SOS takes lesser CPU time then all these three sorting techniques. SOS requires lesser number of swap operations in comparison to Bubble sort, Selection sort. Insertion sort uses shifting of data rather than swapping concept.**

Keywords – Bubble sort, Correct position, Insertion sort, Sorting, Selection sort, Swap.

I. INTRODUCTION

A sorting technique is an algorithm that puts elements of a list in a certain order. Efficient sorting is important for optimizing the use of other algorithms (such as search, merge sort and Binary Search algorithms) that require sorted lists to work correctly. A good algorithm is that which gives satisfactory result for every range of data set [1]. Sorting is the fundamental problem of computer science and remained burning issue for research over the last several years due to time complexity [2]. Sorting is often used in a large variety of critical applications and is a fundamental task that is used by most computers. Sorting algorithm falls into two basic categories: comparison based and non-comparison based using swapping or shifting of data. The comparison based sorting algorithm works on the basis of comparing the elements. Comparison based important algorithms are: quick sort, merge sort, heap sort, bubble sort, and insertion sort. A non-comparison based algorithm sorts an array without consideration of pair wise data elements. Radix sort is a non-comparison based algorithm.

Some existing algorithms are very fast but complex to implement, while some are not fast but easy to implement. Moreover some are better option for small size data while some for larger size data. Some sorting algorithms are suitable for floating point numbers, some are good for specific range, some are better for large dataset, and some are useful for data set having non-distinct values. There are two groups of sorting algorithms one having complexity $O(n^2)$ which include bubble, insertion, selection and other with complexity O(nlogn) which includes heap, merge and quick sort techniques in average case.

In comparison based sorting techniques, 'comparison' and 'swapping' operations are to be performed. But 'comparison' operation is the key operation to be considered for 'time complexity' calculation of the sorting algorithm on the basis of expression representing total number of comparison operations while ignoring the "swapping" operations. Although practically it is observed that swapping operation effects the running time and increases the CPU work load.

In this paper we are introducing a simple and efficient novel sorting technique named *"SOS: Swap Once and Sort"*. This technique places the particular data element at correct position by only one swapping operation. Practically it takes lesser running time than selection and bubble sort therefore is an efficient sorting technique for large data set.

II. SORTING ALGORITHMS

Sorting of data (numeric or Character) always remained in the focus for researchers. Sorting algorithms are still being optimized or even newly invented. In case of mobile systems and information retrieval, efficient sorting is a major concern. Following we describe some sorting algorithms that have used in the context of this study and are regularly taught to IT students. Following we include the critical discussion of Bubblesort, Heapsort, Insertionsort, Mergesort, Quicksort, Selectionsort, Shellsort and Shakersort [3]:

Bubblesort belongs to the family of comparison sorting. It works by repeatedly iterating through the list to be sorted, comparing two items at a time and swapping them if they are in the wrong order. The worst-case complexity is $O(n^2)$ and the best case is O(n). Its space complexity is O(n).

Insertionsort is a naive algorithm that belongs to the family of comparison sorting. In general insertionsort has average time complexity of $O(n^2)$ but is known to be efficient on data sets which are already substantially sorted. It's time complexity in best case is linear O(n). Furthermore insertion sort is an in-place algorithm with O(n) space complexity.

Heapsort is a comparison-based sorting algorithm and part of the selection sort family. Although somewhat slower in practice on most machines than a good implementation of Quicksort, it has the advantage of a worst-case time complexity of O(nlog(n)).

Mergesort was invented by John von Neumann and belongs to the family of comparison-based sorting. Mergesort has an average and worst-case performance of O(nlog(n)). Unfortunately, Mergesort requires three times the memory of in-place algorithms such as Insertionsort.

Quicksort belongs to the family of exchange sorting. It's complexity in average case is O(nlog(n)), while in worst case it requires $O(n^2)$ comparisons. It is one of the most efficient algorithms and is used for many sorting tasks. Its space complexity depends on factors such as selection of right Pivot element, etc. On average, its recursion depth is of O(log (n)) and therefore space complexity is O(log n) as well.

Selectionsort searches for the minimum value, exchanges it with the value in the first position and repeats the first two steps for the remaining list. It belongs to the family of in-place comparison sorting. Its average case time complexity is $O(n^2)$, that is why it is inefficient for large. Selectionsort typically outperforms bubble sort but is generally outperformed by Insertionsort.

More Generalized form of Insertionsort is termed as Shellsort [4]. It is named after its inventor, Donald Shell. It belongs to the family of in-place sorting but is regarded to be unstable. Its worst case time complexity is $O(n^2)$, but can be improved to O(n log (n)). Shellsort improves Insertionsort 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 Shellsort is a plain Insertionsort, but by then, the list of data is guaranteed to be almost sorted.

A variant of Shellsort is Shakersort. It compares each adjacent pair of items in a list in turn, swapping them if necessary, and alternately passes through the list from the beginning to the end then from the end to the beginning. This process stops when a pass does no swaps. Its time complexity is $O(n^2)$ in worst case, while O(n) in best case.

III. MATERIAL AND METHOD

In this section we discuss SOS by explaining the steps used for sorting by giving its pseudocode in 'c', an example and its implementation details with results obtained.

A. Swap once and Sort: SoS

It is an in-place sorting technique. An in-place algorithm is an algorithm which overwrites its input with its output. This sorting technique is applicable on distinct and non-distinct data set. Following is the pseudo code for SOS.

```
SOS (list, n)
  {  i=0;  //function start
    while (i <= n-1)    //first while loop,
       {    count = 0;   j = i+1 ;
                 while (j < n)
                        {   If (list[i] > list[j])  then
                              Count++ ;    J++ ;
                        }
               k = 0;
           If (count  > 0)   then
                    {    while (k <= n-1)
                           {
                        If  (list[i] != list[i+count+k])
```

```
{swap the list[i] and list[i+count+k]; Break; }
                    else
                        k++ ;
                }
            }
        else
            i++ ;
    }   // end of first while loop
} // end of function
```

Note: variable 'k' increment if the pivot element and swapped element is same it will increase the index till when it does not find the different element or end of array.

B. Pseudocode in 'c'

Let us consider a set of data to be sorted is in the 'list' of size 'n'. Name of the function is SOS with arguments 'list' and 'n'.

In general, procedure of applying SOS for a distinct data set, we select the i^{th} indexed element as a pivot element. Than we count the number of smaller elements coming after the pivot element. Suppose total number of smaller elements is 'count' then we swap the pivot i.e. i^{th} indexed element with the $[count+i]^{th}$ indexed element. That position will be the correct position of that pivot element. After this swapping, that pivot element will not be involved in another swapping operation.

For the data set of non-distinct elements i.e. in data set there is repletion of some elements. In this case the procedure remains same but if the element which is ready to swap with pivot element is equal to the pivot element then swap operation is not performed but we move on to the next element and check if that element is not equal to pivot element then perform swapping and so on.

C. Example

Now, explaining the SOS with an example data of size 10. Let us take an array named *list[10]* as following:

15 12 10 16 26 5 20 7 11 24

First we select 0^{th} index element '15' as a pivot element. Total number of smaller elements than the pivot element is counted. Using the concept, *"the right position of any element is after the number of lesser elements",* for ascending order list. In the *list[10]* there are **'5'** smaller elements than pivot element '15'. Now, swapping the pivot element with the next 5^{th} element in the list[10] i.e. swapping with the element at index value 5 in the *list[10], where index is ranging from 0 to 9*. It means 15 will be swapped with 5 and *list[10]* becomes as following:

5 12 10 16 26 15 20 7 11 24

We have completed the first iteration and placed the pivot element 15 at its correct and final position. Correct position is the position where it would be in the sorted list and 15 is placed at its final position by just one swapping. Element 15 is shaded with grey color as an indication of final position of the particular element as it would be in the sorted list.

Next we select the 0^{th} index element as pivot element which is '5'. It is found that there is no element smaller than '5'in the *list[10]*. It means that the pivot element is already at its final position. So, by now two elements are at their final and correct position shown in below list colored grey. Now, we will move at next element 12 in the above list and following the same steps as done for the first pivot element '15'. At this step '12' is exchanged with its next 3^{rd} element i.e. 26 and *list[10]* elements are as following:

5 26 10 16 12 15 20 7 11 24

Similar steps will be repeated till all the elements get their final and correct position as would be in sorted list. Following table shows all the steps of SOS method for sorting the above data of size 10.

5	12	10	16	26	5	20	7	11	24
5	12	10	16	26	15	20	7	11	24
5	12	10	16	26	15	20	7	11	24
5	26	10	16	12	15	20	7	11	24
5	24	10	16	12	15	20	7	11	26
5	11	10	16	12	15	20	7	24	26
5	16	10	11	12	15	20	7	24	26
5	20	10	11	12	15	16	7	24	26
5	7	10	11	12	15	16	20	24	23
5	7	10	11	12	15	16	20	24	23
5	7	10	11	12	15	16	20	24	23

D. Implementation and Comparative Study

We have the run the 'C' code using the compiler "C free" of SOS, bubble sort, selection sort and insertion sort for the study of worst case running time of each separately for same data set, following table shows the running time taken by SOS, bubble sort, selection sort and insertion sort.

Running time in milliseconds

Data Size	SOS	InsertionSort	SelectionSort	BubbleSort
1000	0.000	0.000	0.000	0.000
5000	0.078	0.078	0.140	0.156
10000	0.343	0.343	0.593	0.625
15000	0.750	0.765	1.328	1.421
20000	1.421	1.432	2.468	2.687
25000	2.187	2.203	3.843	4.187
30000	2.953	3.171	4.101	4.480

Above table data shows that SOS and Insertion sort are sorting the data of size 1000-10000 in same time, while SOS sorts the data faster for data size 15000-30000. With all different data sets SOS is faster than selection sort and bubble sort. Above data is plotted in graphical form shown below in figure 1.

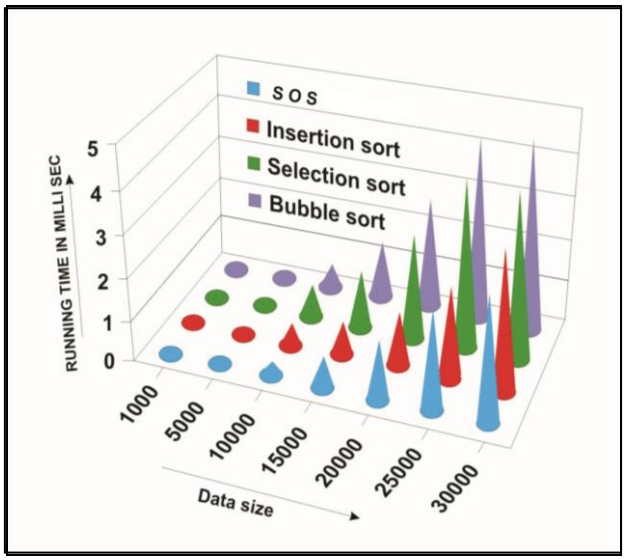

Fig. 1: Running time vs. Data size

We have compared the SOS with selection sort and bubble sort on the basis of another parameter also that is Number of swapping operations taking place to sort the given data. Insertion sort is not the part of it as instead of swapping, shifting operation is used in it.

Number of swapping operations in worst case

Data Size (n)	SOS	Selection sort	Bubble sort
1000	500	499499	499500
5000	2500	12497499	12502500
10000	5000	49994999	50005000
15000	7500	112492499	112507500
20000	10000	199989999	20001000
25000	12500	312487499	312512500
30000	15000	449984999	450015000

Above data is plotted in graphical form shown below in figure 2.

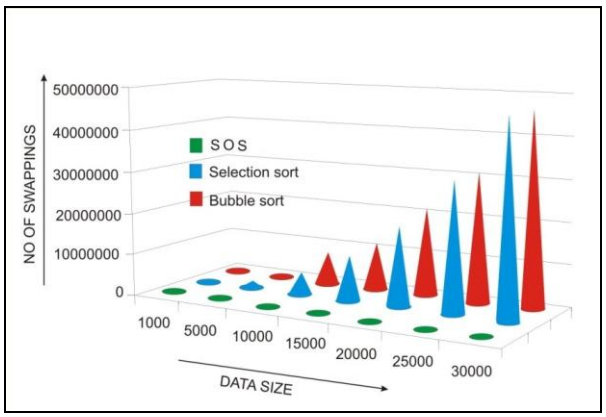

Fig. 2: Number of swapping operations vs. Data size

IV. DISCUSSION OF RESULTS

In fact running time taken by any particular program depends on processor and configuration of the system. We have used "C free" compiler to observe running time by including the header file 'time.h'. Runing time is calculated by the statement run_time= ((t2-t1) / (double) CLOCKS_PER_SEC), where t1 and t2 is the initial and ending run time respectively. We have tested the SOS for worst case. Results show that SOS and Insertion sort are sorting the data of size 1000-10000 in same time, while SOS sorts the data faster for data size 15000-30000. With all different data sets SOS is faster than Insertion sort, selection sort and bubble sort.

There is huge difference in number of swapping operations used for sorting same data set in worst case by SOS, bubble sort, selection sort and Insertion. There are much lesser swapping operations required for SOS in comparison to bubble sort, selection sort. Insertion sort uses shifting operation rather than swapping to sort

the given set of data. SOS uses only n\2 swapping operations in the worst case.

V. CONCLUSION AND FUTURE WORK

SOS has the unique concept to sort the data. Concept of SOS is simple. It is more efficient for small data set in comparison to insertion sort, selection sort and bubble sort. SOS uses maximum n/2 swapping operations to sort the given data of size 'n'. It places the element at its correct position, i.e. position where that element will be in sorted list, after one swapping operation only. In future we intend to compare it with other existing techniques by observing the running time and number of comparison operations of each.

VI. REFERENCES

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein (2003)."Introduction to Algorithms", MIT Press, Cambridge, MA, 2nd edition.

[2] Alfred V, Aho J, Horroroft, Jeffrey DU (2002). Data Structures and Algorithms (India: Pearson Education Asia).

[3] Seymour Lipschutz (2009), "Data Structure with C", Schaum Series, Tata McGraw-Hill Education.

[4] Brejov´a, B. (2001), 'Analyzing variants of Shell Sort', Information Processing Letters 79(5), 223–227.

[5] Lafore, R. (2002), Data Structures and Algorithms in Java, 2nd edn, SAMS Publishing, Indianapolis, Indiana, USA.

[6] Robert S (1998). Algorithms in C. Addison-Wesley Publishing Company, Inc.

[7] Knuth E., The Art of Computer Programming Sorting and Searching, Addison Wesley, 1998.

◈◈◈