

# Doubly Inserted Sort: A Partially Insertion Based Dual Scanned Sorting Algorithm

Smita Paira, Anisha Agarwal, Sk. Safikul Alam  
and Sourabh Chandra

**Abstract** Computer science has made a tremendous impact in practical as well as real life. In recent days, it has faced a lot of challenges and to overcome those, many researchers have put forward their brains to develop various efficient algorithms. Among those, searching and sorting are the most fundamental to keep track of the database. In this paper, a new sorting algorithm has been developed. It is an iterative approach with two different concepts and can perform better than the recursive Divide and Conquer sorting algorithms, having a worst case time complexity of  $O(n)$ . It consumes less space in the stack and can perform better for large data.

**Keywords** Sorting • Insertion sort • Bubble sort • Selection sort • Divide and conquer • Internal sorting • External sorting • Iterative • Recursive • Stack • Quick sort • Merge sort • Heap sort

## 1 Introduction

Sorting is a basic operation of computer science than places the elements in a particular order. It may be in terms of length or alphabetically [1]. Based on the space requirement in the main memory, sorting is of two types namely, Internal Sorting and External Sorting [2]. The type of sorting that can arrange small number of objects, enough to be placed in the main memory, is called Internal Sorting. But if the number of elements is very large then some of them require an external storage for sorting. Such type of Sorting is called External Sorting.

The Internal Sorting is based on two approaches namely, Iterative approach and Recursive approach. The iterative approaches require repeated passes on some portion of the array which scans the entire array in the worst case. As a result, the worst case time complexity is usually  $O(n^2)$ . The recursive approaches, in terms of

---

S. Paira (✉) · A. Agarwal · Sk.S. Alam · S. Chandra  
Department of CSE, Calcutta Institute of Technology, Howrah 711316, India

divide and Conquer, basically uses inductive steps and are quite faster with a worst case time complexity of  $O(n \log_2 n)$ . However, such approaches put extra burden to manage and maintain the stack memory and becomes complicated for large arrays [3]. The Fig. 1 represents the classification of Sorting.

In this paper, a new sorting algorithm has been developed. This algorithm scans the array from both ends just like the Max Min Sorting algorithm [4] and considers two elements at the same time. In other words, the two elements are picked up and are compared. If the lower index contains the larger element then the elements are interchanged. Finally the targeted elements are inserted in the appropriate positions by applying the conventional Insertion Sort algorithm on both ends of the array.

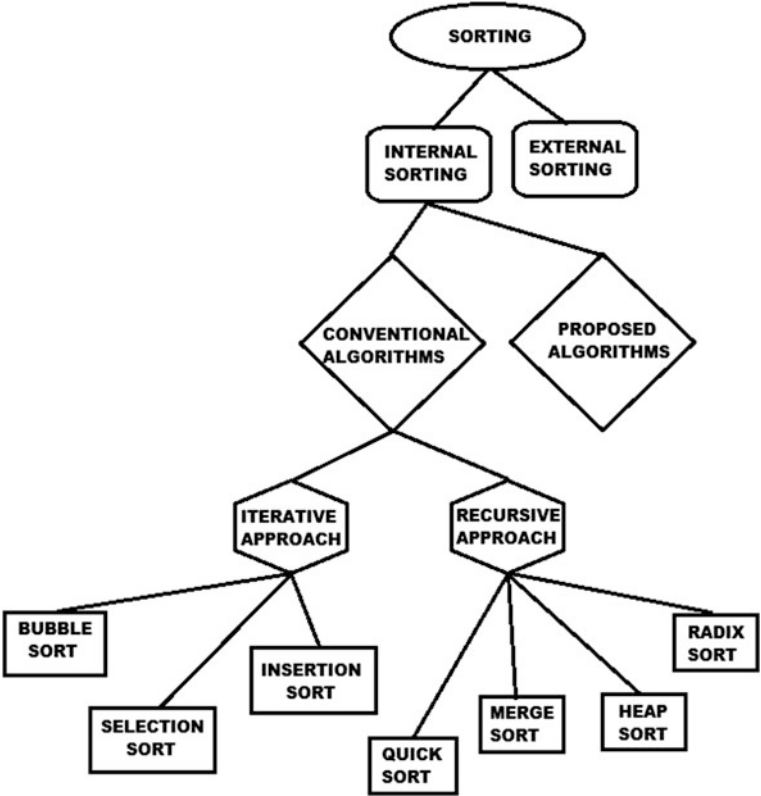


Fig. 1

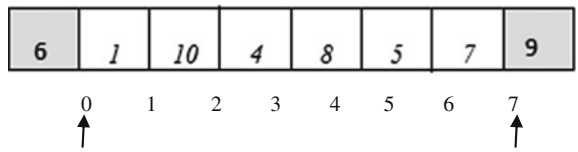
Since the algorithm requires only  $n/2$  steps for sorting an array of size  $n$  in the worst case and best case, it is much more efficient than the conventional iterative and Divide and Conquer sorting approaches. It also has a space complexity of  $O(1)$  and thus provides less overhead on the stack memory, compared to the recursive algorithms. The Doubly Inserted Sort provides fast performance for larger array and has various real life applications.

## 2 Problem Under Consideration

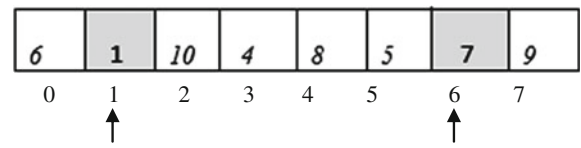
Let us consider an array of  $n$  elements, say  $A []$ . The Doubly Inserted Sort first compares the elements at the first (index 0) and last (index  $n - 1$ ) indices. If they are in proper position, then it proceeds for the next (index 1) and previous (index  $n - 2$ ) elements respectively and so on. If the smaller index element is greater than the larger index element then they are swapped and finally the partial insertion mechanism is applied on both the targeted elements. In the next section, the flow of mechanism of the newly proposed algorithm has been illustrated on an array  $A []$  of size 8.

## 3 Flow of Algorithm

Let the array  $A[8]$  be defined as:



*Iteration 1:* Elements 6 (index 0) and 9 (index 7) are compared. As  $6 < 9$ , they do not need swapping. So, after 1st iteration, the array remains as it is.



*Iteration 2:* Consider elements 1 (index 1) and 7 (index 6). Since,  $1 < 7$ , they do not require to exchange. Now, 1 is compared with 6. As  $1 < 6$ , swap A [0] and A [1]. Similarly, compare 7 and 9. As  $7 < 9$ , they are not interchanged. Thus, after 2nd iteration, the array will look like:

1	6	10	4	8	5	7	9
0	1	2	3	4	5	6	7

↑
↑

*Iteration 3:* Consider elements A [2] and A [5]. As  $10 > 5$ , A [2] and A [5] are swapped. Compare A [2] i.e. 5 with its predecessor. If it is smaller than its immediate predecessor, then swap and continue until the condition fails. Similarly, A [5] i.e. 10 is compared with its successor. If it is greater than its immediate successor, then swap and continue. Otherwise stop. Thus, the array after 3rd iteration looks like:

1	5	6	4	8	7	9	10
0	1	2	3	4	5	6	7

↑
↑

*Iteration 4:* Consider elements A [3] and A [4]. As  $4 < 8$ , they are not swapped. Compare A [3] i.e. 4 with its predecessor. If it is smaller than its immediate predecessor, then interchange and continue until the condition fails. Similarly, A [4] i.e. 8 is compared with its successor. If it is greater than its immediate successor, then swap and continue. Otherwise stop. Thus, the array after 4th iteration looks like:

1	4	5	6	7	8	9	10
0	1	2	3	4	5	6	7

Finally after 4th iteration, the array is sorted.

## 4 Step by Step Solution of the Newly Proposed Algorithm Based on C Code

Algorithm: New\_sort (A)

Where A [] = an array of elements

n = number of elements

Step 1: Repeat for i=0 to n-1

a) Read A [i]

Step 2: Repeat for i=0 to n/2 and j=n-1 to n/2 through Step 7

Step 3: If A [i] > A [j]

a) Swap A [i] and A [j]

Step 4: If i! = 0

a) Repeat for k=i-1 to 0 and l=j+1 to n-1 through Step 6

Step 5: If A [k] < A [k+1] and A [l] > A [l-1]

Break

Step 6: i) If A [k] > A [k+1]

Swap A [k] and A [k+1]

ii) If A [l] < A [l-1]

Swap A [l] and A [l-1]

Step 7: If i=j

i) Initialise

a) k = i-1

b) l = j+1

ii) If A [k] > A [k+1] or A [l] < A [l-1]

a) Goto Step 4

Step 8: Repeat for i=0 to n-1

Output A [i]

Step 9: Exit

## 5 Time Complexity

Let us consider an array A [] of size n.

### 5.1 *Best Case*

If the array is already sorted, then the inner loop of Step 4, in the above section, does not execute. Only Step 2 executes. Thus, the number of iterations required =  $1 + 1 + 1 + \dots$  + up to  $n/2$  times

$$= n/2$$

As a result, the best case time complexity is  $O(n)$ .

### 5.2 *Average Case*

In average case, if  $n$  is even and except the first iteration, both Step 2 and Step 4 of the previous section, execute because the first iteration does not require the inner loop to execute. As a result, the number of iterations required

$$\begin{aligned} &= 0 + 1 + 2 + 3 + \dots + (n - 2)/2 \\ &= ((n - 2) * (n - 4))/8 \\ &= (n^2 - 6n + 8)/8 \end{aligned}$$

If  $n$  is odd, the algorithm requires an extra iteration of the inner loop due to Step 7 of the previous section. As a result, the total number of iterations for odd number of elements is  $((n^2 - 6n + 8)/8) + 1$ .

Thus, the average case time complexity is  $O(n^2)$ .

### 5.3 *Worst Case*

If the array is in descending order, then also the inner loop of Step 4, in the above section, does not execute. Only Step 2 executes. Thus, the number of iterations required

$$\begin{aligned} &= 1 + 1 + 1 + \dots + \text{up to } n/2 \text{ times} \\ &= n/2 \end{aligned}$$

As a result, the worst case time complexity is  $O(n)$ .

Thus, in worst case and best case, the Doubly Inserted Sort requires less number of steps and is much more efficient than the Divide and Conquer Sorting algorithms.

## 6 A Brief Comparison Study Among the Conventional Iterative and Recursive Algorithms with the Doubly Inserted Sort

The iterative sorting algorithms are simpler to implement compared to the recursive ones but in the worst case, they scan almost the entire list. As a result, their time complexity is usually  $O(n^2)$ . Just like the Bubble Sort, Selection Sort and Insertion Sort [5–7], the newly proposed algorithm is an in place sorting algorithm and has a space complexity of  $O(1)$ . Among the iterative methods, insertion sort is highly efficient for small data, especially for substantially sorted array but its efficiency decreases with the array size [8, 9]. The Doubly Inserted Sort is a modified version of the Insertion Sort, in the fact that it scans two data at the same time from both end of the list. It picks up two elements from both ends, compares them first and then apply insertion sort on both of them. With a worst case time complexity of  $O(n)$ , it performs faster than the iterative sorting algorithms.

Compared to the Insertion Sort, Merge Sort performs better for larger array [10]. However, it has one major drawback. It requires an extra auxiliary array for storing the elements and has a space complexity of  $O(n)$  [11]. On the other hand, Quick Sort is better than the Merge Sort as it is an in place sorting algorithm. But, due to the unbalanced partition of the array in the worst case, it has a time complexity of  $O(n^2)$  [12]. Heap sort is better than the Quick Sort as it has a time complexity of  $O(n \log n)$ .

**Table 1** Comparison table for different types of sorting algorithms

Sorting algorithm	Time complexity			Space complexity
	Best case	Average case	Worst case	
Doubly inserted Sort	$O(n)$	$O(n^2)$	$O(n)$	$O(1)$
Bubble sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
Insertion sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Quick sort	$O(n \log_2 n)$	$O(n \log_2 n)$	$O(n^2)$	$O(\log_2 n)$
Merge sort	$O(n \log_2 n)$	$O(n \log_2 n)$	$O(n \log_2 n)$	$O(n)$
Heap sort	$O(n \log_2 n)$	$O(n \log_2 n)$	$O(n \log_2 n)$	$O(1)$
Radix sort	$O(n^2)$	$O(n \log_2 n)$	$O(n \log_2 n)$	$O(m + n)$ or $O(mn)$ , where $m$ is the space required to hold the radices

**Table 2** Table showing the execution times of different types of sorting algorithms

Sorting algorithms	Execution time (s)
Doubly inserted sort	0.989011008
bubble sort	3.461538560
Insertion sort	2.417582336
Selection sort	3.461538560
Quick sort	2.098652741
Merge sort	1.043956032
Heap sort	1.043956032

$\log_2 n$ ). But it is very unstable [13]. Radix Sort also has a time complexity of  $O(n \log_2 n)$  but it is less flexible as it depends on its inner operations [14]. The Doubly Inserted Sort is more efficient than the above Divide and Conquer sorting algorithms as it has a time complexity and space complexity of  $O(n)$  and  $O(1)$  respectively in the worst case. It also reduces the complexities of recursion and performs better in case of large data. It reduces overheads in terms of computation, management and cost. The complexities of different sorting algorithm have been compared in Table 1.

A sample example of the execution times of the above sorting algorithms has been calculated, as shown in Table 2.

## 7 Conclusion

Sorting depends on various circumstances like low time complexity, small memory and simplicity [15]. Various algorithms have been developed so far to arrange a collection of data in some particular order. However, they have certain drawbacks. Some of them lose their efficiency in case of large data while the others may put additional overhead in terms of cost and memory management. The Doubly Inserted Sort requires an average of  $(n^2 - 6n + 8)/8$  steps to sort an array which is much less than  $(n^2 - n)/2$ , as is required by the conventional iterative sorting algorithms. The most significant advantage of this algorithm is that it just makes  $n/2$  passes in the worst case and reduces the unnecessary space and time complexities of the conventional recursive Divide and Conquer sorting algorithms. Its efficiency increases with the size of the list as it scans from both ends of the list and can be applied in database management, directories and various record keeping purposes.

## References

1. Langsam, A.T.: Data Structure Using C & C++, 2nd edn. ISBN13: 9788120311770
2. Aho, A.V., Hopcroft, J.E., Ullman, J.D.: Data Structures & Algorithms, 2nd edn. Chapter 8, pp. 366–425, ISBN13: 9780201000238



3. Paira, S., Chandra, S., Alam, S.S., Partha S.D.: A survey report on divide and conquer sorting algorithms. In: IEEE National Conference on Electrical, Electronics, and Computer Engineering (CALCON 2014), Print ISBN: 978-93-833-0383-0
4. Paira, S., Chandra, S., Alam, S.S. Patra, S.S.: Max min sorting algorithm—a new approach of sorting. *Int. J. Technol. Explor. Learn. (IJTEL)*, **3**(2), 405–408 (2014). ISSN: 2319-2135
5. Lipschutz, S.: *Data structure & Algorithm*, 2nd edn. Schaum's Outlines Tata McGraw Hill, New Delhi. ISBN13: 9780070991309
6. Adamson, I.T.: *Data Structures and Algorithms: A First Course*, pp. 79–85. ISBN13: 9783540760474
7. Divya, S.L.: Improving the performance of selection sort using a modified double-ended selection sorting. *Int. J. Appl. Innov. Eng. Manage. (IJAIEM)* **2**(5), 364–370 (2013). ISSN: 2319-4847
8. Sodhi, T.S., Kaur, S., Kaur, S.: Enhanced insertion sort algorithm. *Int. J. Comput. Appl. (IJCA)* **64**(21), 35–39 (2013). doi: [10.5120/10761-5724](https://doi.org/10.5120/10761-5724)
9. [www.answers.com/Q/What\\_are\\_the\\_advantages\\_and\\_disadvantages\\_of\\_insertion\\_sort](http://www.answers.com/Q/What_are_the_advantages_and_disadvantages_of_insertion_sort)
10. [www.answers.com/Q/What\\_are\\_advantages\\_and\\_disadvantages\\_of\\_merge\\_sort](http://www.answers.com/Q/What_are_advantages_and_disadvantages_of_merge_sort)
11. [www.quora.com/Why-is-quicksort-considered-to-be-better-than-merge-sort](http://www.quora.com/Why-is-quicksort-considered-to-be-better-than-merge-sort)
12. [programmers.stackexchange.com/questions/150615/why-is-quicksort-better-than-other-sorting-algorithms-in-practice](http://programmers.stackexchange.com/questions/150615/why-is-quicksort-better-than-other-sorting-algorithms-in-practice)
13. [stackoverflow.com/questions/8311090/why-not-use-heap-sort-always](http://stackoverflow.com/questions/8311090/why-not-use-heap-sort-always)
14. [www.cprogramming.com/tutorial/computersciencetheory/radix.html](http://www.cprogramming.com/tutorial/computersciencetheory/radix.html)
15. Baluja, G.S. *Data Structure Through C*, 4th edn. Chapter 10, pp. 541–550, ISBN13: 9788174462855

Emerging Research in Computing, Information,  
Communication and Applications

ERCICA 2015, Volume 1

Shetty, N.R.; Hamsavath, P.N.; N., P. (Eds.)

2015, XX, 580 p. 305 illus., 218 illus. in color.,

Hardcover

ISBN: 978-81-322-2549-2