A Bidirectional Approach to Insertion Sort

Md. Khairullah, Muhammed Adnan Choudhury*
Department of Computer Science and Engineering, Shah Jalal University of Science & Technology, Sylhet
*Department of Computer Science, Stamford University Bangladesh, Dhaka
khairul_cse@yahoo.com, m_a_choudhury@yahoo.com

Abstract

Insertion sort [1] is a comparison sort [2] algorithm. Though It has a large time complexity of $O(n^2)$, it has some advantages. It is simple to implement, stable, in place and online algorithm [3]. It is more efficient in practice than most other simple $O(n^2)$ algorithms such as selection sort or bubble sort. Bidirectional approaches to the bubble sort and selection sort, named the cocktail sort and the shaker sort respectively, has significant improvement on them. In this paper, a similar approach has been tried with the insertion sort algorithm. The proposed algorithm is then tried on random data set of different sizes and its performance is analyzed.

Keywords: *Sorting, Bidirectional sorting, Complexity, Insertion Sort.*

INTRODUCTION

Sorting refers to the arranging of numerical or alphabetical or character data in ascending or descending order [4]. Sorting is an important issue in the literature of computer science, as various types of real world data need to be sorted every now and then. Efficient sorting techniques (in terms of processing time) are necessary when we have to sort large number of data.

Extensive research works are being conducted to find out better techniques. Divide and conquer [4] methods are very popular because their complexities are of order O (log n) or O (n log n). Techniques of order $O(n^2)$ are also being studied to find improvements to existing algorithms.

The bubble sort and the selection sort algorithms are easy to understand and implement but the problem with them is their large running time. Bidirectional approaches have been tried for both of them and significant improvements have been noticed. The bidirectional forms of them are known as the cocktail sort and shaker sort [3] respectively.

In this paper a similar approach has been tried for the insertion sort. In this method, construction of the sorted list is performed in both sides instead of just in one.

We will make the list sorted in both of the sides. For this purpose, the first and last elements of the list have to be placed in a sorted manner. After this, the second items are compared from the beginning and the ending and insert the smaller item in the first side and the

larger item in the last side of the list. This process is continued until the middle item is reached; when we will compare and place all items to the first side or the last side according to the result of the comparison. Till now, we have a tendency to place the smaller items are placed in the first half and the larger items in the last half. Items in the both halves are themselves sorted with a high probability of smaller items in the first half and larger items in the last half. As both of the lists are sorted, it will be confirmed that the whole list is sorted if the last item of the first half is smaller than the first item of the last half. If this is not true, both items are simply interchanged to ensure their proper place. However, this interchange can disrupt the sorted state of both the sub-lists. To confirm the sorted ness, both the items are placed properly on their corresponding sub-lists. The sorted ness of the whole list is again tested by the above-mentioned procedure and if the test succeeds, we will stop. If the test fails, this process of test-interchange-proper-placing will continue.

PROPOSED ALGORITHM

Suppose we have the data in the array A(1:N)

1. Interchange A[1] and A[N] if A[1]>A[N]
2. Repeat steps a, b and c for i=2 to N/2
    a) Set MAX=the larger of A[i] and A[N-i] and MIN= the smaller.
    b) Insert MAX into the sub list A(1:i-1) to its proper place to keep the sub list sorted.
    c) Insert MIN into the sub list A((N-i+1):N) to its proper place to keep the sub list sorted.
3. Compare the last element of the first half and the first element in the second half.
    a) If they are in their proper place, every element in A is in their proper place. Terminate the execution.
    b) If these two elements are not in their proper place, interchange them to their proper sub list and place them in their proper place in their sub list. Continue step 3.

Example:

Here is an example of the execution process of this algorithm. The data, which are just placed in their proper place according to the algorithm, are represented in bold and italic font. The sorted sub lists constructed so far are shown underlined.

Suppose we have the data set A={20,18,29,12,34,23,78,12,56,10}

Applying step 1 of the algorithm, we have

A={***10***,18,29,12,34,23,78,12,56,***20***}

Then the candidate items are 18 and 56. As 18<56, 18 will be inserted in the first list and 56 at the last sub list. Then

A={10,***18***,29,12,34,23,78,12,20,***56***}

Then the candidate items are 29 and 12. As 12<29, 12 will be inserted in the first sub list and 29 at last sub list. Then

A={10,***12***,18,12,34,23,78,20,***29***,56}

Then the candidate items are 12 and 78. Therefore, 12 goes in the first sub list and 78 in the last. Then

A={10,***12***,12,18,34,23,20,29,56,***78***}

Then the candidate items are 34 and 23 and 23 goes in the first sub list and 34 in the last. Then

A={10,12,12,18,***23***,20,29,***34***,56,78}

Now we have two sorted sub list with higher probability of having smaller data in the first sub list compared with the second sub list. It needs to be showed that all data in the first sub list are smaller than all data in the last list. This can be proved by a simple comparison of the last item of the first list and the first item of the last list. We have 23 and 20 in these places and 23>20. This means that the whole list is not sorted. Therefore, we need to interchange 23 and 20 to the opposite list and place them in their list such that the two sub lists are again themselves sorted. Now we have

A={10,12,12,18,***20***,***23***,29,34,56,78}

Again testing the last item of the first sub-list and the first item of the last sub-list, we are assured that all data in the first list are smaller than all data in the second list. As both of the lists are themselves sorted hence the whole list is sorted.

RESULTS

Table 1 shows the sorting time for typical insertion sort and the proposed bidirectional insertion sort for the same data set of various sizes. The data sets consist of random data. The Experiment was run on a PC with Pentium III 733 MHz Processor and 128 MB RAM.

Dataset Size	Insertion Sort (Time in second)	Bidirectional Insertion Sort (Time in second)	Performance Improvement (%)
10000	0.721	0.540	25.10
20000	2.814	2.153	23.49
30000	6.459	4.887	24.34
40000	11.597	8.772	24.36
50000	18.877	15.111	19.95
60000	27.950	22.762	18.56
70000	42.091	36.522	13.23
80000	61.388	52.966	13.72
90000	82.839	73.385	11.41
100000	96.549	89.198	7.61
110000	135.004	115.726	14.28
120000	171.336	138.489	19.17

Table 1: Execution times for Random data set

Figure 1 shows graph of the execution time for data in Table1.

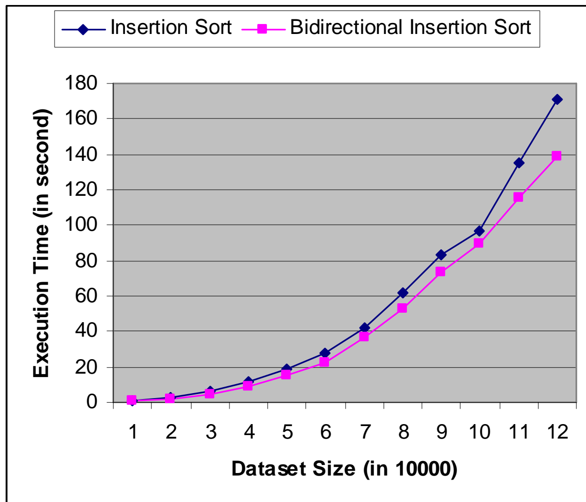

Figure 1: Graphical representation of execution time of the data presented in table 1.

Complexity Analysis:

Complexity of sorting algorithms is the required number of data comparisons. In step 2 of the algorithm, two sub lists of size n/2 have to be processed. Each sub list needs $n^2/8$ comparisons. Hence, step 2 needs $n^2/4$ comparisons. In a typical insertion sort it needs $n^2/2$ comparisons. So the initial processing needs almost half comparison than a typical insertion sort requires. But getting a completely sorted list is possible only after completion of step 3. Each iteration of step 3 needs n comparisons. But as there is a high probability that the data in the first sub-list is smaller than the data in the second list, the completion of step 3 will take considerably less time. So the affect of step 3 is small in the over all complexity and hence complexity of step 2 will dominate in the complexity of the algorithm.

CONCLUSION

Any sorting algorithm needs to place the minimum valued sample in the first position and the maximum valued sample in the last position, second minimum in the second position and the second largest in the place before the last position and so on. If both the ends can be built in a single step and both the lists grow towards the center, it is possible to build the whole sorted list in a shorter time. This performance improvement is reflected in the table presented in the result section.

References:

[1] Horowitz, E., Sahni, S., Rajasekaran, S. Fundamentals of computer algorithms. Galgotia Publications Pvt. Ltd. 2003-2004

[2] Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C. Introduction to algorithms. McGraw Hill Book Company, 2002-2003

[3] http://en.wikipedia.org, Sorting Algorithms

[4] Lipschutz, S. Theory and problems of data structures. McGraw Hill Book Company, 1999