A Bi-partitioned Insertion Algorithm for Sorting

Tarun Tiwari, Sweetesh Singh, Rupesh Srivastava and Neerav Kumar
Department of Computer Science and Engineering,
Motilal Nehru National Institute of Technology, Allahabad, 211004.
ttonhunt@gmail.com; sweetesh.mnnit@gmail.com; srirupesh@gmail.com; neerav.mumbai@gmail.com;

Abstract

It is known that insertion sort performs better on almost sorted arrays than other $O(n^2)$ sorting algorithms. In this paper we develop a bi-partitioned insertion algorithm for sorting, which out beats all other $O(n^2)$ sorting algorithms in general cases. The graphs of total time taken by different sorting algorithms confirm the superiority of our algorithm over other existing similar algorithms. We also prove the correctness of the algorithm and give a detailed time complexity analysis of the algorithm.

1. Introduction

There are mainly two types of comparison based sorting algorithms (i) $O(n^2)$ and (ii) $O(n \log n)$. In general $O(n^2)$ sorting algorithms run slower than $O(n \log n)$ algorithms, but still their importance can not be ignored. The importance of the $O(n^2)$ algorithms is that they can be used in conjunction with $O(n \log n)$ algorithms and the final algorithm can be applied to match a suitable situation, for example divide a big array into smaller arrays, sort these arrays with $O(n^2)$ sorting algorithm and then merge them to produce an entirely sorted array.

In this paper we first develop a sorting algorithm in which we partition the data set into two equal partitions such that all the elements in one partitions are smaller than all the elements of the other partitions and both the partitions are sorted. We further reduce the time complexity of the algorithm by reducing the number of iterations required to half. This algorithm is further improved by using the SELECT procedure due to [1], thus removing the redundant data movements. We further suggest a
scheme for adopting the above methodology to a multi-partitioned insertion algorithm for sorting.

978-1-4244-4520-2/09/$25.00 ©2009 IEEE

This paper is organized as follows: section-2 gives the basic algorithm and various improvements, section-3 illustrates the correctness of the algorithm, section-4 gives the detailed complexity analysis and the running comparison graphs, section-5 discusses the adoption of the bi-partition insertion sort to multi-partition sort, section-6 concludes and gives an overview of future works and finally section-7 gives the important references.

2. The Algorithm

The bi-partition insertion sort algorithm (BISA) is described below:

- Input : An unsorted array A[ ] of size n.
- Output : A sorted array A[ ] of size n.

```
BISA( A[ ], n )
1. for  i ← 1 to ⌈n/2⌉
2.    k ← i, j ← n – i + 1, x ←A[k]
3    do while (x < A[k – 1]) and (k>1)
4.       A[k]←A[k – 1]
5.       k ← k – 1
6.    A[k]←x, y ←A[j]
7.    do while (y > A[j + 1]) and (j< n)
8.       A[j] ←A[j + 1]
9.       j ← j+1
10.   A[j]←y
11.   do if A[i] >A[n – i + 1]
12.      exchange A[i]↔A[n – i +1]
13.      k ← i, j ← n – i +1, x ←A[k]
14.      do while (x <A[k-1]) and (k>1)
15.         A[k] ←A[k – 1]
16.         k ← k – 1
17.      A[k] ←x, y←A[j]
18.      do while (A[j]>A[j+1]) and (j<n)
19.         A[j] ←A[j+1]
20.         j ← j+1
21.      A[j]←y
22. return A[ ]
```

The working of the above algorithm is better understood by working it out on a small array as shown below:

] 2 8 11 7 | 3 6 10 5 [
2] 8 11 7 | 3 6 10 [5
2] 8 11 7 | 3 6 5 [10
2] 5 11 7 | 3 6 8 [10
2 5] 11 7 | 3 6 [8 10
2 5] 6 7 | 3 11 [8 10
2 5] 6 7 | 3 8 [10 11
2 5 6] 7 | 3 [8 10 11
2 5 6] 3 | 7 [8 10 11
2 3 5] 6 | 7 [8 10 11
2 3 5 6] | [7 8 10 11

In the above example the ‘]’ marks the position of left partition, ‘[‘ marks the position of the right partition and ‘|’ marks the middle of the array. On careful investigation of the algorithm we find that lines 3 – 5, 6 – 8, 13 – 15 and 16 – 18 represent the basic insertion procedure. The algorithm gives a better running time than the other $O(n^2)$ sorting algorithms because:

1. The number of iterations is reduced to half, thus reducing the overhead.
2. Every time insertion is done on a sub – array, which is almost sorted, hence we get a better performance.

2.1. Analysis of redundant data movements:

The running time of the algorithm is degraded due to the redundant data movements which increase the time required unnecessarily. Consider the following example:

] 11 3 10 5 | 8 12 6 **9** [
] **9** 3 10 5 | 8 12 6 11 [
9] 3 10 5 | 8 12 6 [11
3] **9** 10 5 | 8 12 6 [11
3] 6 10 5 | 8 12 **9** [11
3 6] 10 5 | 8 12 [**9** 11
3 6] 10 5 | 8 **9** [11 12
3 6] **9** 5 | 8 10 [11 12
3 6 **9**] 5 | 8 [10 11 12
3 6 5] **9** | 8 [10 11 12
3 5 6] 8 | **9** [10 11 12
3 5 6 8] | [**9** 10 11 12

In the above example we observe the movement of element ‘**9**’. It is changing it partition from right to left and vice – versa needlessly as it has to remain in right partition when the algorithm terminates. We can eliminate such redundant data movements and also reduce number of comparisons, if we know the middle element of the sorted array – the benchmark element.

2.2. Improved bi – partition insertion sort (IBISA)

As discussed above, if we get the benchmark element, the performance may be further improved. For this purpose we use the SELECT procedure due to Blum, Floyd, Pratt, Rivest and Tarjan [2], we get the benchmark in $O(n)$ time bound. Hence the improved version of BISA( ), IBISA( ) follows:

- Input : Unsorted array A[ ] of size n.
- Output : Sorted array A[ ] of size n.

```
IBISA (A[ ], n)
1. B ← SELECT (A[ ], ⌈n/2⌉)
2. Assign arrays BL[ ⌊n/2⌋ ] and BR[ ⌈n/2⌉ ]
3. l ← 1, r ← ⌈n/2⌉, f ← l, b ← r
4. for i ← 1 to ⌈n/2⌉
5.   do if A[i] < B
6.      do if l = f
7.           BL[f] ← A[i], f ← f+1
8.           continue
9.      else
10.          BL[f] ← A[i], f ← f+1
11.          k ← f – 1, x ←BL[k]
12.          do while (x < BL[k–1]) and (k>1)
13.              BL[k] ← BL[k–1]
14.              k ← k–1
15.          BL[k] ←x
16. else if A[i] > B
17.       do if b = r
18.            BR[b] = A[i], b ← b – 1
19.            continue
20.       else
21.            BR[b] = A[i], b ← b – 1
22.            j ← b+1, y ←BR[j]
23.            do while (BR[j] > BR[j+1]) and (j< n)
24.                BR[j] ← BR[j+1]
25.                j ← j+1
26.          BR[j] ←y
27. do if A[n–i+1] < B
28.     do if l = f
29.          BL[f] ← A[i], f ← f+1
30.          continue
31.     else
32.          BL[f] ← A[i], f ← f+1
33.          k ← f – 1, x ←BL[k]
34.          do while (x < BL[k–1]) and (k>1)
35.              BL[k] ← BL[k–1]
36.              k ← k–1
37.          BL[k] ←x
```

```
38. else if A[n–i+1] > B
39.       do if b = r
40.            BR[b] = A[i], b ← b – 1
41.            continue
42.       else
43.            BR[b] = A[i], b ← b – 1
44.            j ← b+1, y ←BR[j]
45.           do while (y > BR[j+1]) and (j< n)
46.               BR[j] ← BR[j+1]
47.               j ← j+1
48.           BR[j] ←y
49. Assign array BM[ ⌊n/2⌋ – f + b + 1]
50. for i ← 1 to (⌊n/2⌋ – f + b + 1)
51.       do BM[i] ← B
52. k ← 1
53. for i ← 1 to (f – 1)
54.       do A[k] ← BL[i], k ← k – 1
55. for i ← 1 to (⌊n/2⌋ – f + b + 1)
56.       do A[k] ← BM[i], k ← k – 1
57. for i ← b+1 to ⌈n/2⌉
58.       do A[k] ← BR[i]
59. return A[ ]
```

Here we have created three partitions – BL[ ] contains data less than B, BM[ ] contains data equal to B and BR[ ] contains data greater than B. The space requirements may be further reduced by calculating the number of elements smaller than, equal to and greater than the benchmark element and then allocating exact spaces for the partitions BL[ ], BM[ ] and BR[ ]. Also we can make this algorithm in–place, but it increases the overhead in which case it is favorable to use BISA.

3. Correctness of the algorithm

Theorem – The necessary and sufficient condition for an array a[max] to be sorted is for any indices p and q, $p \le q \leftrightarrow A[p] \le A[q]$ where $p, q \in [1, n]$. We now prove that after completion of algorithm, resultant array satisfies the above condition.

Proof: - We would use loop invariant method to prove correctness of BISA ( ), as it is simpler to understand than IBISA ( ).

Loop invariant: After each iteration, the two partitions are sorted and the largest element of left partition (LP) is smaller than the smallest element of the right partitions (RP).

- **Initialization**

In the first iteration i = 1, the LP contains only A[1] and RP contains only A[n]. If A[1] > A[n] they are swapped in lines 9 – 10. Hence, both the partitions are trivially sorted and loop invariant holds.

- **Maintenance**

Consider i = k, then LP contains A[1…….(k–1)] and RP contains A[(n–k+2)……..n] as can be inferred from the algorithm. Following are the observations:

$$A[k-1] \le A[(n-k+2)],$$

and $\forall$ i, j such that $A[i] \in$ LP and $A[j] \in$ RP

$$A[i] \le A[j]$$

Hence loop invariant is maintained before the iteration. Now following cases may occur with the data elements A[k] and A[n–k+1]

1. $A[k] \le A[n-k+1]$ in which case LP contains A[1……..…k] in sorted order and RP contains A[(n–k+1)……n] in sorted order, hence the loop invariant is maintained and control moves to next iteration.

2. $A[k] \ge A[n-k+1]$ in which case they are swapped in the lines 9 – 10, and inserted into LP and RP respectively. From the above line of reasoning we can infer that the loop invariant holds in this case too.

- **Termination**

The loop terminates when $i = \lceil n/2 \rceil$. It can be simply argued that at this instance BISA ( ) returns two partitions of size $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$ such that LP contains A[1…… $\lfloor n/2 \rfloor$] and RP contains A[$\lceil n/2 \rceil$.....n] with the condition that LP and RP are sorted and $A[\lfloor n/2 \rfloor] \le A[\lceil n/2 \rceil]$. Hence the loop invariant is maintained at termination. So all the elements in A[1……..n] are sorted and hence we prove that the algorithm BISA ( ) works. The same argument can be formatted to show that IBISA ( ) works too.

4. Complexity analysis

We compute the time complexity of BISA ( ) using the cost time analysis. Since the number of constants involved would be large, we would skip some of the details of the analysis but it would not affect the flow of the analysis.

The complexity is given by the time function $T(n)$.

		cost	times
BISA(A[ ], n)			
1.	**for** i←1 **to** $\lceil n/2 \rceil$	c_1	$(n/2)+1$
2.	k←i, j←n–i+1, x ←A[k]	c_2	$(n/2)$
3	do while (x < A[k–1]) & (k>1)	c_3	$\sum_{k=1}^{n/2} t_k$
4.	A[k] ←A[k–1]	c_4	$\sum_{k=1}^{n/2} (t_k - 1)$
5.	k ← k–1	c_5	$\sum_{k=1}^{n/2} (t_k - 1)$
6.	A[k] ←x, y ←A[j]	c_6	$(n/2)$
7.	do while (y > A[j+1]) & (j< n)	c_7	$\sum_{k=1}^{n/2} t_k$
8.	A[j]←A[j + 1]	c_8	$\sum_{k=1}^{n/2} (t_k - 1)$
9.	j←j+1	c_9	$\sum_{k=1}^{n/2} (t_k - 1)$
10.	A[j]←y	c_{10}	$(n/2)$
11.	do if A[i] >A[n – i + 1]	c_{11}	$(n/2)$
12.	exchange A[i]↔A[n – i+1]	c_{12}	$(n/2)$
13.	k←i, j←n–i+1, x←A[k]	c_{13}	$(n/2)$
14.	do while (x <A[k–1]) &(k>1)	c_{14}	$\sum_{k=1}^{n/2} t_k$
15.	A[k]↔A[k – 1]	c_{15}	$\sum_{k=1}^{n/2} (t_k - 1)$
16.	k ← k–1	c_{16}	$\sum_{k=1}^{n/2} (t_k - 1)$
17.	A[k]←x, y←A[j]	c_{17}	$(n/2)$
18.	do while (y>A[j+1]) & (j<n)	c_{18}	$\sum_{k=1}^{n/2} t_k$
19.	A[j]←A[j+1]	c_{19}	$\sum_{k=1}^{n/2} (t_k - 1)$
20.	j ← j+1	c_{20}	$\sum_{k=1}^{n/2} (t_k - 1)$
21.	A[j]←y	c_{21}	$(n/2)$
22.	return A[ ]	c_{22}	1

We give the expression for the time function $T(n)$ by aggregating the constants into singular constants which would make the analysis understandable. Hence we compute $T(n)$ by multiplying the costs with corresponding times and then taking a summation of all the terms. So we have,

$$T(n) = \alpha\left(\sum_{k=1}^{n/2} t_k\right) + \beta\left(\sum_{k=}^{n/2} (t_k - 1)\right) + \gamma\left(\frac{n}{2}\right) + \delta$$

The best case of the algorithm occurs when the element to be inserted in LP is greater than all the elements of LP and the element to be inserted in RP is smaller than all the elements of RP. In this case, $t_k = 1$ in every iteration, so

$$T(n) = \alpha\left(\frac{n}{2}\right) + \gamma\left(\frac{n}{2}\right) + \delta = an + b$$

for some constants a and b. Hence $T(n)$ is a linear function of n. The worst case of the algorithm occurs when the element to be inserted in LP has to be compared with each element of the LP and a similar thing happens with RP. In this case, $t_k = k$ in each iteration, so we have,

$$\sum_{k=1}^{n/2} t_k = \sum_{k=1}^{n/2} k = \frac{n(n+2)}{8} \quad \text{and,}$$

$$\sum_{k=1}^{n/2} (t_k - 1) = \sum_{k=1}^{n/1} (k-1) = \frac{n(n-2)}{8}$$

With the above results we now have the time function as,

$$T(n) = \alpha\left(\frac{n(n+2)}{2}\right) + \beta\left(\frac{n(n-2)}{8}\right) + \gamma\left(\frac{n}{2}\right) + \delta$$

or $$T(n) = \frac{n^2}{8}(\alpha + \beta) + \frac{n}{8}(2\alpha - 2\beta + 4\gamma) + \delta$$

$$= an^2 + bn + c$$

for some constants a, b and c. Hence we see that $T(n)$ varies as quadratic function of n. Thus we have

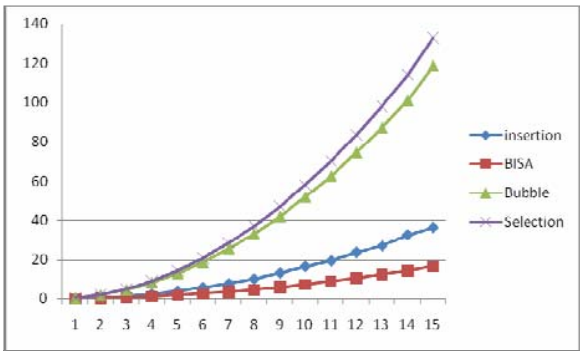
$$T(n) = \Omega(n) \text{ and,}$$

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

Thus we prove that the algorithm BISA ( ) belongs to the $O(n^2)$ complexity sorting algorithm class.

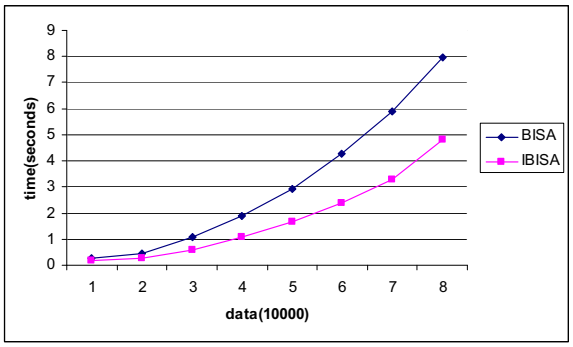
We now give the running time comparison chart, where our algorithm is compared with selection, bubble and insertion sort. Here insertion is showing a lower time requirement because of the structure of data given as input. One important thing to note about our algorithm is that it's running time not affected by the variance of the data set given as the input, whereas for

all other algorithms, the running time depends greatly on the structure of data given as input.



The scale on Y – axis is x1 seconds, and scale on X – axis is x1000 elements in input data.

The improvements we did in the BISA ( ) resulted in IBISA ( ) which was expected to perform better. In fact it did perform extraordinarily giving a speedup of 1.8 – 1.9 against an expected speedup of 2. The following graph confirms our claim.



Hence we can conclude that IBISA ( ) is the best performing amongst selection, bubble, insertion, BISA and IBISA.

5. Adoption of bi–partition methodology to develop multi–partition insertion sort

In this section we suggest an adoption of bi–partition methodology to develop multi–partition insertion sort algorithm (MISA). We will use the procedure SELECT to find the $n/4, n/2$ and the $3n/4$ smallest elements of the array and create four equal quarter partitions – A[1……n/4], A[n/4…..n/2], A[n/2……3n/4] and A[3n/4……n]. Then we would insert the elements according to the partitions they belong to. The above scheme can also be implemented for 6 partitions, 8 partitions, 10 partitions etc. One important point to note is that the number of partitions can exceed only up to a certain limit, above which its performance is degraded.

Implementation of above scheme is still under testing and analysis and we are getting the expected speedups. We are yet to discover the optimal number of partitions.

6. Conclusion and future work

The important thing about this algorithm is that its best case running time is linear and for generally distributed data sets it is the best candidate amongst the $O(n^2)$ sorting algorithms. Also it has been noticed that the number of swaps required are of the order of $O(n)$. As a matter of fact, we have yet not found the worst case of the algorithm. The findings of section-6 also recommend this algorithm to be used in conjunction with the $O(n \log n)$ sorting algorithms.

The primary objective of this paper is not to improve the insertion sort algorithm but it suggest a methodology in which we have to maintain two or more sorted partitions and hence when we use insertion to sort the partitions we get an algorithm of $O(n^2)$ complexity whereas if we use heap or tree to keep the partitions sorted we get an $O(n \log n)$ sorting algorithm. We are currently working on this line of thought and have obtained positive results with heaps. The use of trees is yet to be implemented.

One more area on which we are working is that once we use SELECT we have the data items sorted in small chunks, say 5 or 7. We could utilize this to further reduce the running time of the algorithm.

7. References

[1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. The Design and Analysis of Computer Algorithms. Addison Wesley, 1974.

[2] Donald E. Knuth. Sorting and Searching, volume 3 of The Art of Computer Programming. Addison Wesley, 1981.

[3] Horowitz E. and S. Sahni. Fundamentals of Computer Algorithms. Woodland Hills, Cal. : Computer Science Press, 1978.

[4]Thomas H. Cormen, Charles E. Lieserson, Ronald L. Rivest and Clifford Stein. Introduction to Algorithms. MIT Press.