Bidirectional Expansion – Insertion Algorithm for Sorting

Rupesh Srivastava
Department of Computer Science and Engineering, Motilal Nehru National Institute of Technology, Allahabad, 211004.
srirupesh@gmail.com

Tarun Tiwari
Department of Computer Science and Engineering, Motilal Nehru National Institute of Technology, Allahabad, 211004.
ttonhunt@gmail.com

Sweetesh Singh
Department of Computer Science and Engineering, Motilal Nehru National Institute of Technology, Allahabad, 211004.
sweetesh.mnnit@gmail.com

***Abstract*—** In this paper we present a sorting algorithm, which uses the methodology of insertion sort efficiently to give a much better performance than the existing sorting algorithms of the $O(n^2)$ class. We prove the correctness of the algorithm and give a detailed time complexity analysis of the algorithm. We also describe various applications of the algorithms.

Keywords-Algorithms; Sorting; Complexity; Comparison Sort.

I. 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. Since $O(n^2)$ algorithms are non recursive in nature, they require much less space on the RAM. Another importance of the $O(n^2)$ algorithms is that they can be used in conjunction with the $O(n \log n)$ 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. Another application of sorting $O(n^2)$ algorithms is in sorting small arrays. Since sorting algorithms are recursive in nature their use is not recommended for sorting small arrays as the perform poorly.

It is known that among $O(n^2)$ sorting algorithms, selection sort and insertion sort are the best performing algorithms in general data distributions. For some data distributions insertion performs better than selection and vice – versa. Other algorithms are suited for very limited and particular data distributions.

In this paper we device a sorting algorithm, in which sorting starts from the middle of the array and expands both ways i.e. left and right. Since the number of iterations required is half than that in case of insertion sort, the overhead is reduced thus decreasing running time.

The paper is organized as follows: section – 2 gives the basic algorithm and an example to illustrate the working of the algorithm. Section – 3 proves the correctness of the algorithm using the method of loop invariant, section – 4 gives a detailed time complexity analysis of the algorithm and the running time comparison graphs, section – 5 concludes and gives an overview of the future work and finally section – 6 gives important references..

II. THE ALGORITHM

The bidirectional expansion – insertion sort algorithm (BEIS) is described below:

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

```
BEIS ( A[ ], n )
1. mid ← (n/2), p ← mid + 2, q ← mid – 1
2. for i ← 1 to mid
3.         do if (A[i] > A[n – i + 1])
4.                   exchange A[i] ↔ A[n – i + 1]
5. do while (q ≥ 1)
6.         t ← A[p]
7.         do if (t >A[mid])
8.                   j ← p – 1
9.                   do while (A[j] >t]
10.                            A[j+1] ← A[j]
11.                            j ← j – 1
12.                  A[j+1] ← t
13.        else
14.                  for i ← p down to (mid+1)
15.                            A[i] ← A[i – 1]
16.                  j ← mid – 1
17.                  do while (A[j]>t)
18.                            A[j+1] ←A[j]
```

```
19.                         j ← j – 1
20.                 A[j+1] ← t
21.         p ← p+1
22.         t ← A[q]
23.         do if (t>A[mid+1])
24.                 for i ← q to mid
25.                         A[i] ← A[i+1]
26.                 j ← mid+2
27.                 do while (A[j]< t)
28.                         A[j–1] ←A[j]
29.                         j ← j+1
30.                 A[j–1] ← t
31.         else
32.                 j ← q+1
33.                 do while (A[j]< t)
34.                         A[j–1] ← A[j]
35.                         j ← j+1
36.                 A[j–1] ← t
37.         q ← q – 1
38. return A[ ]
```

The procedures from lines 9 – 12, 17 – 20, 27 – 30 and 33 – 36 simply represent the insertion procedure, while those from lines 14 – 15, and 24 – 25 are simply shifting procedures. To simplify the code we have define the following:

- Lines 9–12: Insert A[p] into the sub array A(mid+1)….p]. Let's call it INSERTP(A[p],mid+1,p).
- Lines 17–20: Insert A[p] into the sub array A[q….mid]. Let's call it INSERTP(A[p],q,mid).
- Lines 27–30: Insert A[q] into the sub array A[(mid+1)….p]. Let's call it INSERTQ(A[q],mid+1,p).
- Lines 33–36: Insert A[q] into the sub array A[q….mid]. Let's call it INSERTQ(A[q],q,mid).
- Lines 14–15: Shift the elements A[mid….(p–1)] one position right each. Let's call it SHIFTR(mid,p–1).
- Lines 14–15: Shift the elements A[(q+1)….(mid–1)] one position left each. Let's call it SHIFTL(q+1,mid–1).

Now the shorter form of the algorithm can be stated as follows:

```
BEIS ( A[ ], n )
1. mid ← (n/2), p ← mid + 2, q ← mid – 1
2. for i ← 1 to mid
3.         do if (A[i] > A[n – i + 1])
4.                 exchange A[i] ↔ A[n – i + 1]
5. do while (q ≥ 1)
6.         t ← A[p]
7.         do if (t >A[mid])
8.                 j ← p – 1
9.                 INSERT(t, q, mid)
10.        else
11.                SHIFTR(mid, p–1)
12.                j ← mid – 1
13.                INSERTP(t, mid+1, p)
14.        p ← p+1
15.        t ← A[q]
16.        do if (t>A[mid+1])
17.                SHIFTL(q+1, mid–1)
18.                j ← mid+2
19.                INSERTQ(t, mid+1, p)
20.        else
21.                j ← q+1
22.                INSERTQ(t, q, mid)
23.        q ← q – 1
24. return A[ ]
```

The working of the above algorithm can be understood by the following example. Consider the following input array:

7 10 6 4 11 1 8 3 15 9

After lines 2 – 4 the array is as follows:

7 10 3 4 1 11 8 6 15 9

Now we show the array status after lines 14 and 23 respectively, for each iteration of the while loop.

```
7 10 3 4 [ 1 8 11 ] 6 15 9
7 10 3 [ 1 4 8 11 ] 6 15 9
7 10 3 [ 1 4 6 8 11 ] 15 9
7 10 [ 1 3 4 6 8 11 ] 15 9
7 10 [ 1 3 4 6 8 11 15 ] 9
7 [ 1 3 4 6 8 10 11 15 ] 9
7 [ 1 3 4 6 8 9 10 11 15 ]
[ 1 3 4 6 7 8 9 10 11 15 ]
```

Where [ ] represents the sorted sub part of the array A[ ]. The time required to sort is lesser than any other existing algorithm because:

1. The number of iterations has been reduced to half, thus reducing the overhead and the running time.

2. Number of swaps required is also less as compared to insertion sort, which further reduces the running time.

III. 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 BEIS ( ). We would use its shorter form for reference.

Loop invariant: Before and after each iteration the sub array A[(q+1)…….(p – 1)] is sorted. pagination anywhere in the paper. Do not number text heads-the template will do that for you.

- **Initialization:**

In this part lines 2 – 4 ensures that the sub array A[mid, mid+1) is sorted. This is the initial condition and we add elements on the left and right of this sub array, maintaining the loop invariant.

- **Maintenance:**

Consider q = (mid – k) and p = (mid+k+1), then from the loop invariant A[(mid–k+1)……(mid+k)] is sorted. In the lines 6–13, t = A[p] is inserted correctly getting the sorted array A[(m–k+1)…….(mid+k+1)]. In the line 14, 'p' is incremented, so p = (mid+k+2). Now, in the lines 15–22, t = A[q] is inserted correctly, getting sorted array A[(m–k)…….(mid+K+1)]. In the line 23 q is decremented, so q = (mid – k – 1). Hence at the end of this iteration sub array A[(m – k+1)…….(mid+k+1)] is sorted. Hence the loop invariant holds. Hence the loop invariant is maintained for each iteration of the while loop.

- **Termination:**

The loop terminates when q becomes 0. At the last iteration of the while loop, when q = 1, we have A[2……(n–1)] as sorted. From the reasoning of the maintenance part, we can infer that at the end of this iteration the A[1……n] is sorted. Hence loop invariant holds at termination of the algorithm also.

Hence, we prove that the algorithm BEIS ( ) correctly sorts the input array.

IV. Complexity Of The Algorithm

We compute the time complexity of BEIS ( ) 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 $f(n)$

BEIS (A[ ], n)	*cost*	*times*
1. mid ← (n/2), p←mid+2, q←mid–1	c_1	1
2. **for** i ← 1 **to** mid		
3. **do if** (A[i] > A[n – i + 1])	c_2	$(n/2)+1$
4. **exchange** A[i] ↔ A[n – i + 1]		
5. **do while** (q ≥ 1)	c_3	$(n/2)+1$
6. t ← A[p]	c_4	$(n/2)$
7. **do if** (t >A[mid])	c_5	$(n/2)$
8. j ← p – 1		
9. INSERT(t, q, mid)	c_6	$\sum_{j=2}^{n/2} T_j$
10 **else**	c_7	$(n/2)$
11. SHIFTR(mid, p–1)	c_8	$\sum_{j=2}^{n/2} T_j$
12. j ← mid – 1		
13. INSERTP(t, mid+1, p)		
14. p ← p+1	c_9	$(n/2)$
15. t ← A[q]	c_{10}	$(n/2)$
16. **do if** (t>A[mid+1])	c_{11}	$(n/2)$
17. SHIFTL(q+1, mid–1)	c_{12}	$\sum_{j=2}^{n/2} T_j$
18. j ← mid+2		
19. INSERTQ(t, mid+1, p)		
20. **else**	c_{13}	$(n/2)$
21. j ← q+1		
22. INSERTQ(t, q, mid)	c_{14}	$\sum_{j=2}^{n/2} T_j$
23. q ← q – 1	c_{15}	$(n/2)$
24. **return** A[ ]	c_{16}	1

We can aggregate the constants into singular constants, without the loss of generality, thus making the derivation more comprehensible. Hence we compute $f(n)$ by multiplying the costs with corresponding times and then taking a summation of all the terms. So we have,

$$f(n) = \alpha\left(\sum_{j=1}^{n/2} T_j\right) + \gamma\left(\frac{n}{2}\right) + \delta$$

The best case of the algorithm occurs when the input array is already sorted. In this case $T_j = 1$. So the resulting expression $f(n)$ is given as

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

for some constants *a* and *b*. Hence $f(n)$ is a linear function of *n*. Now the worst case occurs when $T_j = j$ in which case,

$$\sum_{j=2}^{n/2} T_j = (2+4+6+.....+(n/2))$$

or, $$\sum_{j=2}^{n/2} T_j = \left(\frac{n(n+4)}{16}\right)$$

With the above results we now have the time function as

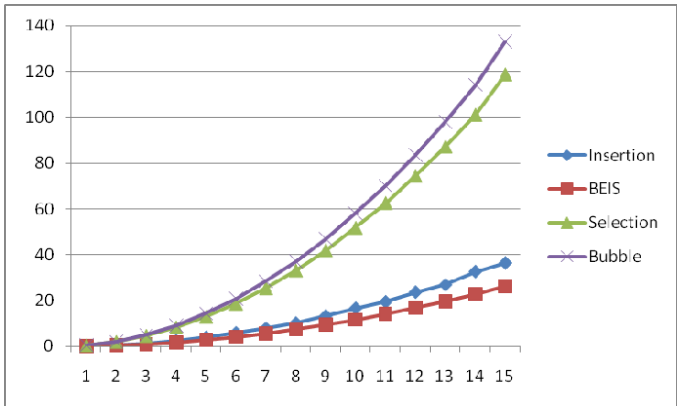
$$f(n) = an^2 + bn + c$$

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

$$f(n) = \Omega(n^2)$$
$$f(n) = \mathrm{O}(n^2)$$

Thus we prove that the algorithm BEIS ( ) belongs to $O(n^2)$ the 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 its 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.



It is clear from the above graph that BEIS runs faster than the existing sorting algorithms of the same complexity class. Hence we can conclude that BEIS ( ) is the best performing amongst selection, bubble, insertion and BEIS.

V. Conclusion And Future Works

In this paper we presented our algorithm which gave a better running time than the existing sorting algorithms of the same complexity class.

The present algorithm can be efficiently used to sort small arrays, typically of size lying from 10 – 50 elements. The algorithm can be used in conjunction with the $O(n \log n)$ sorting algorithms efficiently. It can also be used to sort small arrays efficiently. 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. As a matter of fact, we have yet not found the worst case of the algorithm.

The future work includes the study of the performance of the algorithm in conjunction with merge sort when applied in external sorting applications. Also we are currently working on reducing the number of swaps required by balancing the insertion procedure. The challenge is to propose a method which introduces a minimal overhead.

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.

[5] Data Structures with Abstract Data Types and Pascal by Stubbs and Webre, pub. Brooks/Cole.

[6] M. D. McIlroy, **A killer adversary for quicksort**, *Software--Practice and Experience* **29** (1999) 341-344.

[7] P. M. McIlroy, K. Bostic and M. D. McIlroy, **Engineering radix sort**, *Computing Systems* 6 (1993) 5-27.

[8] Shell,D.L. (1959). "**A high-speed sorting procedure**". Communications of the ACM **2** (7): 30–32.

[9] Owen Astrachan. **Bubble Sort: An Archaeological Algorithmic Analysis**. SIGCSE 2003.

[10] M. A. Bender, M. Farach-Colton and M. A. Mosteiro. "**Insertion Sort is O(n log n)**". In *Proceedings of the 3rd International Conference on Fun with Algorithms (FUN)*, pages 16-23, 2004.