Asian Journal of Information Technology 5(7): 678-680, 2006

Binary Insertion Sort: A Modified Way of Sorting

Md. Mosaddik Hasan, Md. Sazzad Hossain, Shib Nath Datta and Md. Abu Yousuf
Department of Computer Science and Engineering, Mawlana Bhashani Science and Technology University, Santosh, Tangail-1902, Bangladesh

Abstract: This study presents a technique for sorting data in an efficient way. The number of comparisons of the proposed algorithm is less then the existing algorithm and it does not require extra memory space. So this algorithm is very suitable for sorting large number of data item. For general case when we need to sort all the data that means when all the data is newly inserted then our algorithm is better than any other existing algorithm and when some data is inserted to a large amount of sorted data (e.g. voter management system or central database system in a country) then our algorithm is very much better than the existing algorithms.

Key words: Binary search, insertion sort, quick sort, merge sort, heap sort and divide and conquer

INTRODUCTION

Sorting is a very essential tool in computing. Many computing system especially database-related systems need to sort data frequently. The need of an improved sorting algorithm is very crucial when the volume of data to be sorted is very large. Voter management system or central database system in a country, for example may require sorting with respect to a key item. In this case an efficient sorting technique can save much processing time.

Insertion sort[1,2] finds the correct position of the considered data by searching linearly with each of the sorted data. In the proposed algorithm we used divide and conquer method (Binary Search[1]) to find the position of the considered data which decreases number of comparison many times. Experimental result shows that the proposed algorithm requires less number of comparisons than the existing algorithms such as Quick Sort, Merge Sort, Heap Sort, Radix sort etc.

FORMULATING ALGORITHM

To sort a data set we combined Insertion Sort and Binary Search algorithm. The approach of the proposed algorithm is quite similar to Insertion Sort. Insertion Sort works the way many people sort a hand of playing cards. Insertion Sort starts with an empty left hand and the cards face down on the Table. Then Insertion Sort removes one card at a time from the Table and inserts it into the correct position in the left hand. To find the correct position for a card Insertion Sort compare it with each of the cards already in the hand. But our proposed algorithm finds the correct position applying Binary Search algorithm. The proposed algorithm is transcribed in Fig. 1.

Implementation of proposed algorithm: Let us consider the following data set as an example of sorting using proposed algorithm.

Data set = {5, 2, 4, 6, 1, 3}. The step-by-step sorting procedure using proposed algorithm is depicted in Fig 2.

Complexity analysis: The number of data movements, however, can be varied depending on the distribution of

```
Alogrithm
Bi_insertion_sort(n,a)
{
 a [0] = -∞;
 for k := 2 to n;
   {
       Temp: =a[k], Beg: =1, End: =k-1;
       While (Beg<=End)
          {
            Mid :=(Beg+End)/2;
            if (Temp<a [Mid] and Temp>=a [Mid-1] or Temp=a [Mid]) then
                  {
                    p:=mid;
                    break;
                  }
            else if (Temp>a[Mid] and Temp<=a[Mid+1]) then
                  {
                    p:=Mid+1;
                    break;
                  }
            else if(Temp>a[Mid]) then
                    Beg:=Mid+1;
                 else End:=Mid-1;
          }
       for i:=k-1 to p
          a[i+1]:=a[i];
        a[p]:=Temp;
   }
}
```

Fig. 1: The sorting algorithm

Corresponding Author: Md. Mosaddik Hasan, Department of Computer Science and Engineering, Mawlana Bhashani Science and Technology University, Santosh, Tangail-1902, Bangladesh

Steps:

(a)

0	1	2	3	4	5	6
—∞	5	2	4	6	1	3

Beg = 1, End = 1, Mid = 1, Temp =a [2] = 2, Temp<a [Mid] and Temp>a [Mid-1]
So, position = Mid = 1.

(b)

0	1	2	3	4	5	6
—∞	2	5	4	6	1	3

Beg = 1, End = 2, Mid = 1, Temp =a [3] = 4, Temp>a [Mid] and Temp<a [Mid+1]
So, position = Mid + 1 = 2.

(c)

0	1	2	3	4	5	6
—∞	2	4	5	6	1	3

Beg = 1, End = 3, Mid = 2, Temp =a [4] = 6, Temp>a [Mid] and Temp! <a [Mid+1],
Beg = Mid +1=3, Mid = 3, Temp>a [Mid] and Temp = a[Mid+1]
So, position = Mid + 1 = 4.

(d)

0	1	2	3	4	5	6
—∞	2	4	5	6	1	3

Beg = 1, End = 4, Mid = 2, Temp =a [5] = 1, Temp<a [Mid] and Temp! >a [Mid-1],
End = Mid -1= 1, Mid = 1, Temp<a [Mid] and Temp > a[Mid-1]
So, position = Mid = 1.

(e)

0	1	2	3	4	5	6
—∞	1	2	4	5	6	3

Beg = 1, End = 5, Mid = 3, Temp =a [6] = 3, Temp<a [Mid] and Temp>a [Mid-1],
So, position = Mid = 3.

(f)

0	1	2	3	4	5	6
—∞	1	2	3	4	5	6

Fig. 2: The sorting process

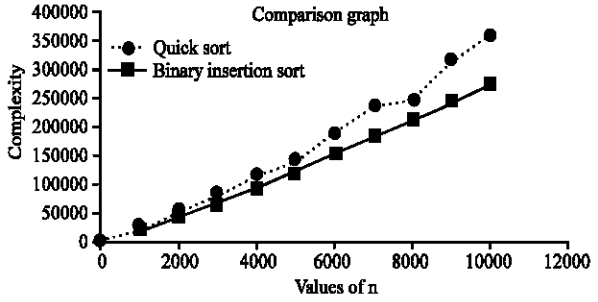

Fig. 3: Complexity of quick sort and current algorithm for 1000<n<10000

Number of Inputs		Number of comparisons	
m	n	Proposed algorithm	Quick sort
100	1000	695	10784
200	2000	1529	23704
300	3000	2416	37429
400	4000	3337	51678
500	5000	4282	66316
600	6000	5248	81264

Fig. 4: Comparison Table where m is a number of new data and n is the number of existing sorted data

data, which is not taken into account in determining complexity. To find the correct position of a desired data we have used Binary Search algorithm. We know that complexity of Binary Search algorithm is log n.
So, find position of-

31st data we used log 1 comparison
2nd data we used log 2 comparison
3rd data we used log 3 comparison
……
……
nth data we used log n comparison
So, total number of comparison is,
log 1+log 2+log 3 +…+log n
=log (1 x 2 x 3 x...x n)
=log (n!)
≈n log n

So, the complexity of the proposed algorithm is n log n for all cases.

Suppose, in voter management system we have to entry new data before every election. Consider that the system has n voters at this time we have to insert m voters into this system.

For 1st data we have to compare log (1+n) time
For 2nd data we have to compare log (2+n) time
For 3rd data we have to compare log (3+n) time
……
……
For mth data we have to compare log (m + n) time

So, the total number of comparison
= log (1+n) + log (2+n) + log (3+n) +…+
log (m + n)
= log {(1+n) (2+n) (3+n)… (m + n)}
= log (m+n!/n!)

But, Quick sort, Heap sort will compare n log n times. When m<n and m is very large then

log (m+n!/n!) << n log n.

So, in this case our algorithm is very much better.

Performance analysis: The average case complexity of Quick Sort algorithm is 1.4 n log n but worst-case complexity is O (n^2)[1,2,3]. The complexity of Merge Sort and Heap Sort is O (n log n) but they require extra memory space[1]. But our algorithm's all cases complexity is n log n and doesn't require extra memory. The complexity curve for some number of data items is illustrated in Fig 3. From the complexity curves we observe that our algorithm requires less number of comparisons than quick sort.

When small number of data is inserted into large number of sorted data, this algorithm sorts the data set much more efficiently than the existing methods, which is shown in the Fig. 4.

CONCLUSION

The most significant aspect of the current algorithm is that it requires minimum number of comparison than existing algorithm. It is especially very much better than existing algorithms for those systems, which have to insert new data frequently. It is also better than other algorithm where whole dataset is sorted. So, we hope that one can save much processing time using this algorithm for sorting.

REFERENCES

1. Lipschutz, S., 2002. Data Structures. McGraw Hill Book Company.
2. Horowitz, E. and S. Sahni, 1998. Fundamentals of Computer algorithms. Galgotia Publications Ltd., New Delhi.
3. Tomas, H. Cormen and E. Charles Leiserson, 2004. Introduction to Algorithms. Prentice-Hall of India Private Ltd., New Delhi.