An Efficient Sorting Algorithm for Positive Values

[1]D Ravi Babu, [2]R Shiva Shankar, [3]V Priyadarshini, [4]P Neelima
[1,2,3,4]Dept. of CSE, S.R.K.R Engg. College, Affiliated to Andhra University, Bhimavaram, AP, India

Abstract
In this paper we present a sorting algorithm for positive values, which uses the methodology of indexing of the array and insert that number into proper index of the array without performing any element comparisons and swapping. This algorithm efficiently to give a much better performance than the existing sorting algorithms of the $O(n^2)$ class, for large array size with same length of digits of input data for positive values.

Keywords
Array-IndexedSorting Algorithm

I. Introduction

In computer science a sorting algorithm is an algorithm that puts elements of a list in a certain order. The most-used orders are numerical order and lexicographical order. Efficient sorting is important for optimizing the use of other algorithms (such as search and merge algorithms) that require sorted lists to work correctly; it is also often useful for canonicalizing data and for producing human-readable output. More formally, the output must satisfy two conditions:

- The output is in non decreasing order (each element is no smaller than the previous element according to the desired total order);
- The output is a permutation (reordering) of the input.

There are mainly two types of comparison based sorting algorithms (i) $O(n^2)$ and (ii) O(nlogn). In general $O(n^2)$ sorting algorithms run slower than O(nlogn) algorithms, but still their importance can't be ignored. Since the $O(n^2)$ algorithms are non recursive in nature, they require much less space on the RAM. Another importance of the algorithms is that they can be used in conjunction with algorithms and the final algorithm can be applied to match a suitable situation, for example divide a big array into smaller arrays. Another application of $O(n^2)$ sorting algorithms is in sorting small arrays. Since O(nlogn) sorting algorithms are recursive in nature their use is not recommended for sorting small arrays as they 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.

When the size of the array to be sorted approaches or exceeds the available primary memory, so that (much slower) disk or swap space must be employed, the memory usage pattern of a sorting algorithm becomes important, and an algorithm that might have been fairly efficient when the array fit easily in RAM may become impractical. In this scenario, the total number of comparisons becomes (relatively) less important, and the number of times sections of memory must be copied or swapped to and from the disk can dominate the performance characteristics of an algorithm. Thus, the number of passes and the localization of comparisons can be more important than the raw number of comparisons, since comparisons of nearby elements to one another happen at system speed (or, with caching, even at CPU speed), which, compared to disk speed, is virtually instantaneous.

In this paper we device a sorting algorithm for positive integers including zero value, in which we use to store input array as sorted order by using indices of numbers required is lesser the overhead is reduced thus decreasing running time.

The paper is organized as follows: section III, gives the basic algorithm and an example to illustrate the working of the algorithm. Section IV, gives the running cost of the algorithm. Section V, gives a detailed space complexity analysis of the algorithm. Section VI, gives the comparison of this algorithm with existing sorting algorithms. section VII, concludes and gives an overview of the future work.

II. Related Work

A number of sorting methods have been proposed so far. Here is the brief introduction of some of the techniques.

Bubble sort is the simplest way to sort a list but also the slowest one. The basic methodology behind this oldest sort is to compare two neighboring elements and also to swap them if they are in a wrong order. For the list of 100 items, bubble sort makes 10000 comparisons to sort the list. In the best case where the list is already sorted the algorithm has O(n) behavior, while in best average and worst case it shows $O(n^2)$ [2]. Selection sort is inefficient on larger list but works well on small list. The algorithm works by selecting the minimum value in the list, and swapping it with 1st element of the list. It has $O(n^2)$ [behavior and in certain situations, it has a prominent efficiency than some other complex algorithms [3]. Another renowned and simple technique of sorting is insertion sort. Insertion sort, an efficient sorting technique, selects one element in every pass and inserts it to the original location in a new list. It is highly efficient on small lists and is very simple and easy to implement. The worst case complexity of insertion sort is also $O(n^2)$. The insertion sort algorithm is a very slow algorithm when list is very large [4]. Cocktail sort is also known as bidirectional bubble sort, as the basic idea behind this sort is of bubble sort. The only difference with the bubble sort is that it sorts the list in both directions in one pass. The complexity of this sorting algorithm is $O(n^2)$ for both worst and average case. But has O(n) when the list is already ordered [5]. A significantly faster and well known technique is Quick Sort. The Algorithm selects an element as a pivot and rearranges the list in such a way that all elements in the list which are greater than pivot element comes after it. Then recursively sorting the smaller elements list as well as larger elements list. The algorithm has O(nlogn) behavior in both average and best cases, while gives $O(n^2)$ in the worst case performance [6]. Heap sort begins by building a heapout of the data set, and then removing the largest item and placing it at the end of the partially sorted array. After removing the largest item, it reconstructs the heap, removes the largest remaining item, and places it in the next open position from the end of the partially sorted array. This is repeated until there are no items left in the heap and the sorted array is full. The algorithm has O(nlogn) behavior in both average, best cases, and worst case performance [7]. It is also known as the generalized form of the insertion sort as the elements by this sort takes longer jumps to get their original positions. The worst case complexity of the algorithm is $O(n^2)$ [8]. Another profound method in sorting world is merge sort. It works on Divide and Conquer principle. The algorithm works

by dividing the unsorted list into two, sorting the two sub lists recursively by applying the merge sort again. In the end merging the sub lists. It has O(nlogn) behavior in both average and worst case and O(logn) performance in the best case [9]. Each key is first figuratively dropped into one level of buckets corresponding to the value of the rightmost digit. Each bucket preserves the original order of the keys as the keys are dropped into. There is a one-to-one correspondence between the number of buckets and the number of values that can be represented by a digit. Then, the process repeats with the next neighbouring digit until there are no more digits to process. It has O(kN) behavior in worst case performance [10].

III. The Algorithm

The Array Indexed Sorting algorithm (AIS) is described below:

• Input : An unsorted array R[ ] of size n.
• Output : A sorted array S[ ] of size n.

```
ArrayIndexedSort( R[ ], n )
1. x← 0, TinyValue ← 0, HugeValue ← 0
2. for i ← 0 to n-1
3. begin
4. if (TinyValue>=R[i]) do TinyValue=R[i]
5. if (HugeValue<=R[i]) do HugeValue=R[i]
6. end
7. for j ← TinyValue to HugeValue
8. begin
9. for k ← 0 to n-1
10.begin
11.do if(R[k]==j)
12. S[x] ← R[k]
13.x++
12.end
13.end
14.for i ← 0 to n-1
15.R[i] ← S[i]
```

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

n=10,
R[10]={6,7,3,13,2,3,0,1,2,8}

Input data array R and lines 2-6 which is used to find the MinValue and HugeValue from the Input array

R[0]=6
R[1]=7
R[2]=3
R[3]=13←HugeValue
R[4]=2
R[5]=3
R[6]=0←TinyValue
R[7]=1
R[8]=2
R[9]=8

Thus,
TinyValue=0, HugeValue=13
From the line 7 to 9
j←0 to 13, K←0 to 9

Table 1:

For Index:	Numbers found:	Sorted Array:
j=0	R[6]=0	S[0]=0
j=1	R[7]=1	S[1]=1
j=2	R[4]=2	S[2]=2
	R[8]=2	S[3]=2
j=3	R[2]=3	S[4]=3
	R[5]=3	S[5]=3
j=4	NIL	
j=5	NIL	
j=6	R[0]=6	S[6]=6
j=7	R[1]=7	S[7]=7
j=8	R[9]=8	S[8]=8
j=9	NIL	
j=10	NIL	
j=11	NIL	
j=12	NIL	
j=13	R[3]=13	S[9]=13

Line 11, which is used to find the numbers in the input array corresponding to the index. Lines 12 and 13 are used to store the numbers as sorting order as maintain the array S, by the sequence numbers found in the input array R for the index j.

Finally line 15, which is used to copy the sorted array S to the array R.

IV. Running Cost Analysis

This algorithm is suitable for large arrays. The main structure of the algorithm depicts that there is an outer main loop within which there lies another loop. The outer loop will run number of times of difference of HugeValue and MinValue, inner loop will make its way n- 1 times in both cases, By keen observing it the worst case running cost of algorithm is calculated to be O(n2). The behavior of the algorithm in the best case will be O(n), depicting that the length of digits of elements in the list are same (i.e. if the one element in the array is "345" then length of digits of that element is "3") and array size would be large(array size with in multiples of thousands). Similarly the average case of the running cost will be $O(n^2)$ depending upon the elements in the list.

V. Space Complexity of Algorithm

The space complexity of an algorithm is the number of elements which the program needs to store during its execution. This number is calculated by the size of input which is provided to a program. According to program structure, the outer loop will run number of times of difference of HugeValue and MinValue, inner loop will make its way n-1 times in both cases. By keen observing complexity will be $O(n^2)$ and it requires an extra array same size as input array in both cases.

VI. Comparison of Array- Indexed Sort With Existing Sorting Algorithm

To check the efficiency of algorithm, it was compared with other sorting techniques. By taking lists of various sizes, items were generated using a uniform random number generator with values ranging from 1 to 3000. Running times were noted down. The plots are given below.

A. Comparison with Bubble Sort

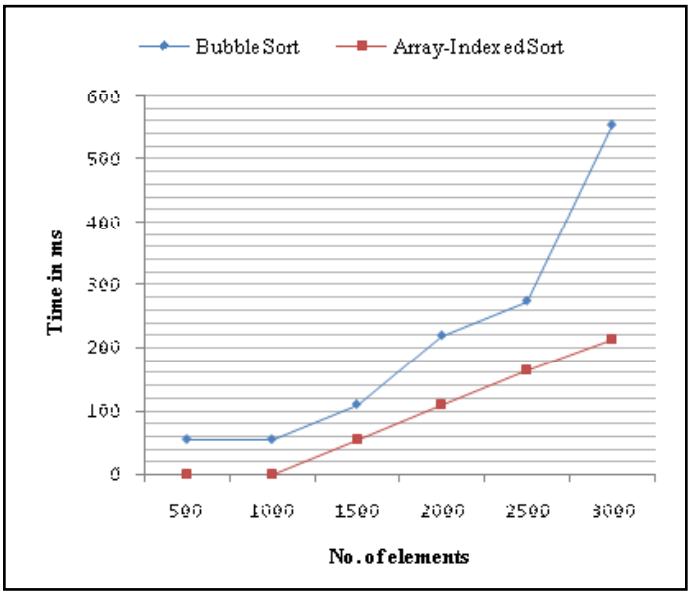

Fig. 1: Bubble Sort v/s Array-Indexed Sort

In the above graph, at x-axis we have placed number of elements in the list to be sorted and at y-axis we have placed the time taken by program for execution in milliseconds. It can be seen clearly that up to 3000 elements the difference is not much but when the size starts increasing, execution time of bubble sort also increases rapidly while Array-Indexed sort shows much more better performance that is obvious from the graph.

B. Comparison with Insertion Sort

Below graph depicts the performance difference of Insertion and the Array-Indexed sort. Along x-axis is the number of tems in the input list while along y-axis is the execution times.

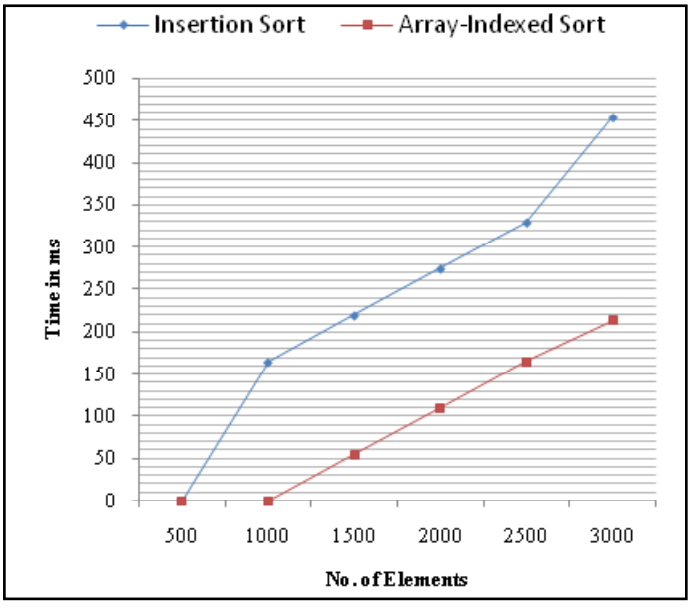

Fig. 2: Insertion Sort v/s Array-Indexed Sort

Above graph depicts the performance difference of Selection and the Array-Indexed sort. Along x-axis is the number of items in the input list while along y is the execution times.

C. Comparison with Selection Sort

Fig. 3: Selection Sort v/s Array-Indexed Sort

VII. Conclusion and Future Work

In this paper we presented our algorithm which give a better running time than the existing sorting algorithms of the same complexity class. (Bubble sort, Selection sort, Insertion sort), for larger data, although it is less efficient than the most efficient algorithms like Merge sort and Quick sort. Overall this analysis shows that Array-Indexed Sort is very efficient for large number items with same length of digits of elements in a list. The future work includes sorting of negative integers and the study of the performance of the algorithm in conjunction with other best sorting algorithms.

References

[1] H. C. Thomas, E. L Charles, L. R Ronald, S. Clifford, "Introduction to Algorithms", Second Edition. MIT Press and McGraw- Hill, Section 1.1: Algorithms, pp. 5–6, 2001.

[2] K. Donald,"The Art of Computer Programming Sorting and Searching", Third Edition. Addison-Wesley, Section 5.2.2: Sorting by Exchanging. Vol. 3, pp. 106–110, 1997.

[3] L. Seymour,"Theory and Problems of Data Structures", Schaum's Outline Series: International Edition, McGraw-Hill. Section 9.4: Selection Sort. pp. 324–325, 1986.

[4] L. Seymour,"Theory and Problems of Data Structures", Schaum's Outline Series: International Edition, McGraw-Hill. Section 9.3: Insertion Sort. pp. 322–323, 1986.

[5] Cocktail_sort, (2009), Cocktail sort. [Online] Available: http://en. wikipedia.org/wiki/Cocktail_sort, Accessed January 5, 2011.

[6] Hoare, C. A. R.,"Analysis of the performance of the parallel quick sort method", BIT Numerical Mathematics Springer Netherlands, Vol. 25, No.1, pp, 321-322, 1985.

[7] Heap_sort, (2011), heap sort [Online] Available: http://en.wikipedia.org/wiki/HeapsortAccessed January 5, 2011.

[8] Shell _sort, (2011), Shell Sort. [Online] Available: http://en.wikipedia.org/wiki/Shell_sort,Accessed January 5,11,

[9] W.H. Butt, M. Y. Javed, (2008) "A New Relative Sort Algorithm based on mean value", IEEE Multi topic Conference.
[10] Merge _sort, (2011), Merge Sort. [Online] Available: http://en.wikipedia.org/wiki/Radix_sort, Accessed January 5,

D Ravi Babu received his B.Tech. degree in Computer Science & Engineering from Sagi Rama Krishnam Raju engineering College(Andhra University, Visakhapatnam), Bhimavaram, INDIA, in 2005, the M.Tech. degree in Computer Science and Technology from SRKR ENGG. College, Bhimavaram (Andhra University) ,INDIA, in 2009,. He was an Assistant professor, SRKR Engineering college, Andhra University, Visakhapatnam in 2005. His research interests include Data Mining, Privacy, and Computer Algorithms.

R Shiva Shankar received his B.Tech. Degree in Computer Science & Engineering from MVGR College of Engineering (JNTU HYDERABAD) INDIA, in 2006, the M.Tech. Degree in Computer Science and Technology from SRKR ENGG. College, Bhimavaram (Andhra University, Visakhapatnam), INDIA, in 2009. He was an Assistant professor, SRKR Engineering college, Andhra University, Visakhapatnam in 2008. His research interests include Data Mining, Image Processing.

V Priya Darshini received her B.Tech. degree in Computer Science & Engineering from Shri Vishnu Engineering College for Women(JNTU HYDERABAD) INDIA, in 2005, the M.Tech. degree in Computer Science and Technology from SRKR ENGG. College, Bhimavaram(Andhra University, Visakhapatnam) ,INDIA, in 2010,. She was an Assistant professor, SRKR Engineering College, Andhra University, in 2006. Her research interests include Computer Networks, Image Processing.

PNeelima received her B.Tech. Degree in Computer Science & Engineering from SRKR Engineering College(Andhra University, Visakhapatnam) INDIA, in 2006, the M.Tech. degree in Computer Science and Technology from SRKR ENGG. College, Bhimavaram(Andhra University, Visakhapatnam), INDIA, in 2010 She was an Assistant professor, SRKR Engineering college, Andhra University, in 2006. Her research interests include Computer Networks, Image Processing.