International Journal of Information, Control and Computer Sciences
ISSN: 2517-9942
Vol:2, No:10, 2008

A Novel In-Place Sorting Algorithm with $O(n \log z)$ Comparisons and $O(n \log z)$ Moves

Hanan Ahmed-Hosni Mahmoud, and Nadia Al-Ghreimil

***Abstract*—**In-place sorting algorithms play an important role in many fields such as very large database systems, data warehouses, data mining, etc. Such algorithms maximize the size of data that can be processed in main memory without input/output operations. In this paper, a novel in-place sorting algorithm is presented. The algorithm comprises two phases; rearranging the input unsorted array in place, resulting segments that are ordered relative to each other but whose elements are yet to be sorted. The first phase requires linear time, while, in the second phase, elements of each segment are sorted in-place in the order of $z \log (z)$, where z is the size of the segment, and $O(1)$ auxiliary storage. The algorithm performs, in the worst case, for an array of size n, an $O(n \log z)$ element comparisons and $O(n \log z)$ element moves. Further, no auxiliary arithmetic operations with indices are required. Besides these theoretical achievements of this algorithm, it is of practical interest, because of its simplicity. Experimental results also show that it outperforms other in-place sorting algorithms. Finally, the analysis of time and space complexity, and required number of moves are presented, along with the auxiliary storage requirements of the proposed algorithm.

***Keywords*—**Auxiliary storage sorting, in-place sorting, sorting.

I. Introduction

SORTING is one of the most fundamental problems in the field of computer science[1], [2]. Comparison-based algorithms perform, in the average case, at least $\log n! \approx n \bullet \log n - n \bullet \log e \approx n \bullet \log n - 1.443n$ comparisons to sort an array of n elements [2], [3]. The lower bound for element moves is $n \log n$. The merge sort Algorithm performs very closely to the optimum, with $n \log n$ comparisons [2], [4]. However, the merge sort is *not* an *in-place* algorithm, it needs an additional n-element array for sorting n elements [5]–[7]. In-place algorithms play an important role, because they maximize the size of data that can be processed in the main memory without input/output operations. In-place sorting algorithms, performing $O(n \bullet \log n)$ comparisons and, $O(1)$ auxiliary storage are of great importance.

The binary-search version of insert sort requires $\log n!+n$ comparisons, and only $O(1)$ index variables of $\log n$ bits each, for pointing to the input array [8]–[10]. However, the algorithm performs $O(n^2)$ element moves, which makes it very slow, especially as n increases. The heap sort was the first in-place sorting algorithm with time complexity bounded by $O(n \bullet \log n)$ in the worst case with $O(1)$ storage requirements but only performs $n \bullet \log n + O(n)$ element moves [10]–[14]. In-place variants of a k-way merge sort with at most $n \bullet \log n + O(n)$ comparisons, $O(1)$ auxiliary storage, and $\varepsilon \bullet n \bullet \log n + O(n)$ moves are presented, instead of merging only 2 blocks, k sorted blocks are merged together at the same time. Here, k denotes an arbitrarily large, but fixed, integer constant, and $\varepsilon>0$ an arbitrarily small, but fixed, real constant [2], [15], [16].

The k-way variant has been generalized to a $(\log n/\log \log n)$-way in-place merge sort. This algorithm uses $n \bullet \log n + O(n \bullet \log n)$ comparisons, $O(1)$ auxiliary storage, and only $O(n \bullet \log n/\log \log n)$ element moves. This algorithm is of theoretical interest, as the first comparisons-based sorting algorithm to break the bound of $(n \bullet \log n)$ for the number of moves [17]–[21].

In this paper, a new in-place sorting algorithm that reduces, simultaneously, the number of comparisons, element moves, and required auxiliary storage will be presented. Our algorithm operates, in the worst case, with at most $n \log z$ element comparisons, $n \log z$ element moves, and $O(1)$ auxiliary storage. Also there are no auxiliary arithmetic operations with indices. Besides these theoretical achievements of this algorithm, it is of practical interest, because of its simplicity. Experimental results will also show that it outperforms other in-place sorting algorithms [21].

In the following sections, the proposed algorithm, and the computational requirement analysis will be presented in details. The proposed algorithm and the required data structures will be presented in section 2. Analysis of the time complexity will be carried out in section 3. Analysis of required number of moves will be discussed in section 4. Auxiliary storage requirements will be discussed in section 5. Issues on stability of the proposed algorithm will be detailed in section 6. Experimental results are presented in section 7. Conclusions and references will follow.

Manuscript received September, 2006.

Hanan Ahmed-Hosni Mahmoud is with the Information Technology Department, College of Computer and Information Sciences, King Saud University, Riyadh, Saudi-Arabia on leave from the department of Computer and System Engineering, Faculty of Engineering, University of Alexandria (e-mail: hanan2010us@yahoo.com).

Nadia Al-Ghreimil, is with the Information Technology Department, College of Computer and Information Sciences, King Saud University, Riyadh, Saudi-Arabia (phone/fax: +966-1-4781479, e-mail: ghreimil@ksu.edu.sa).

II. The Proposed In-Place Sorting Algorithm

Using n elements, divide the elements into some segments $\sigma_0, \sigma_1, \ldots, \sigma_f$, of variable lengths L_i, such that all elements in the segment σ_k satisfy $a_k \le a \le a_{k+1}$. L_i can be obtained by scanning the n elements once. The sorted array is obtained by forming the sequence $a_1, \ldots, a_f$, where a_k denotes the segment σ_k in sorted order. To sort σ_k, use any in-place sorting algorithm with worst case time complexity of n log n where n is the size of the segment to be sorted. Also the chosen algorithm to sort the segments should require at most n log n moves. The data structure and the proposed algorithm are presented below.

A. Data Structures

1. **ELM** is an array of n elements. This is the array of the elements to be sorted. The elements can be of any type as long as lexical ordering between the elements can be established.
2. **count**, **count1** are two small integer arrays of m elements, where m denotes the number of segments or buckets. The elements of these two arrays will hold the number of elements in each segment, i.e. element **count(j)** will hold the count of the elements of segment j.
3. **where** is a small array of m integers, the element **where(j)** of this array will hold the supposed start position of the elements of segment j.
4. **last** is a small array of m integers, the element **last(j)** holds the supposed last position of the elements of the segment j.

B. The New Proposed Algorithm

Algorithm In-place Sorting

```
{
//phase 1: in place formation of segments or buckets//
for j= 1 to m
   {
   count(j) = 0;
   count1(j) = 0;
   where(j)= 0;
   last(j)= 0;
   }
for i = 1 to n
   {
   read ELM(i);
   buck = determine( ELM(i));
   //determine buck which is the bucket of element ELM(i)//
   count(buck)++;
   }
for j = 1 to m
   {
   count1(j) = count(j);
   where(j) = count(j-1) +1;
   last(j) = where(j) +count(j);
   }
i=1; temp =ELM(i);
Do until (count(1)=0 and count(2)=0 and ……. count(m)=0)
{
read temp;
buck = determine( temp)
if ( temp is not equal to ELM(where(buck))
   {
   i1= where(buck);
   Temp= ELM(i1);
   move ELM(i) to position (where(buck)) in the array ELM;
   count(buck)--;
   where(buck)++;
   i=i1;
   }
else
   i++;
temp = ELM(i);
}

//phase 2: in place sorting of each segment //
str = 1;
for j = 1 to m
   {
   for (k = str to str + count (j))
      {
      xsort(ELM(i), K}
      str = str + count(j);
      }
   //xsort is any in-place sorting algorithm of O(n log n), in
   time complexity and number of required moves//
   }
```

III. Analysis of Time Complexity

A. Time Complexity of Phase 1

1. Counting elements in each bucket requires one scan for the array ELM; therefore, time complexity for this step is of the order n.
   Time complexity of Phase1_step1 Θ n. (1)
2. Moving the elements to their appropriate place according to their bucket requires one scan for the array ELM, therefore, time complexity for this step is of the order n.
   Time complexity of Phase1_step2 Θ n. (2)

From (1) and (2):
Time complexity of Phase1 Θ n. (3)

B. Time Complexity of Phase 2

Assume that there are m buckets, on the average, each bucket is of size $z = n/m$.

1. The complexity of xsort is z log z, therefore:
   Time complexity of xsort Θ z log z. (4)
2. Total time complexity for sorting m buckets using xsort is (m*z log z), therefore:
   Time complexity of total xsort Θ m*z log z (5)

From (4) and (5), it can be concluded that the time complexity for Phase 2 is m*z log z, therefore:
Time complexity of Phase 2 Θ m*z log z (6)

Time complexity of Phase 2 $\Theta\ m*n/m \log z$ (7)

Time complexity of Phase 2 $\Theta\ n \log z$ (8)

C. Total Time Complexity

From (3) and (8), total time complexity for The proposed in-place sorting algorithm can be computed as follows:

Time complexity of Sort $\Theta\ n + n \log z$ (9)

Since z is an integer that is much greater than 1,
therefore $\log z$ is greater than 1,
therefore $n \log z > n$ (10)

Time complexity of The proposed in-place sorting algorithm $\Theta\ n \log z$ (11)

IV. Analysis of Number of Moves

A. Phase 1

1. Counting elements in each bucket requires no moves.
2. Moving elements to their appropriate buckets requires n moves.

These results are detailed in the following equation:

Total number of moves for Phase 1 Θ n (12)

B. Phase 2

The number of moves, required by phase two, depends on the used sorting algorithm xsort. However, according to our assumptions, the maximum number of moves for sorting each bucket is equal to $z \log z$. This sums up to $n \log z$ for the m buckets which is presented in the following equation:

Total number of moves in Phase 2 Θ n log z (13)

C. Total Moves

Maximum number of moves for The proposed in-place sorting algorithm can be calculated from (12) and (13) as follows:

Total number of moves for The proposed in-place sorting algorithm Θ n + n log z (14)

But since $n \log z > n$ as given in (10), we can conclude that

Total number of moves for The proposed in-place sorting algorithm Θ n log z (15)

As discussed before, the time complexity analysis and the analysis of the number of moves required by The proposed in-place sorting algorithm depends – to a great extent – on the time complexity and the number of moves required by the xsort algorithm. Further studies are already in progress for choosing an appropriate algorithm for xsort. These studies are going to be published in the near future.

V. Analysis of Space Complexity

There is no extra auxiliary storage required for The proposed in-place sorting algorithm. Also, there are no indices or any other hidden data structures required for the execution of the algorithm. These issues are discussed below.

A. Phase 1

There is a requirement for m integer variables in Phase 1, since m is a small constant, we can assume that the auxiliary storage required for Phase 1 is of $O(1)$, this is summarized in (16) as follows,

Auxiliary storage for Phase1 Θ (1) (16)

B. Phase 2

The computation of the auxiliary storage requirements for Phase 2 depends on the choice of xsort algorithm. Since one of the conditions for choosing xsort is that it has to be an in-place algorithm, this implies that xsort and consequently Phase 2 require auxiliary storage in $O(1)$, this is summarized in (17).

Auxiliary storage for phase2 Θ (1) (17)

C. Total Space Complexity

The auxiliary storage requirements for The proposed in-place sorting algorithm can be derived from (16) and (17) as follows:

Auxiliary storage for The proposed in-place sorting algorithm Θ (1) (18)

It can be concluded, from the previous discussion, that The proposed in-place sorting algorithm is indeed an in-place algorithm.

VI. Analysis of the Stability

Stability of sorting algorithms is required in many applications, for some others stability is not be that important. Generally speaking, stability of a sorting algorithm is established by a penalty in the algorithm performance (time or space complexity).

The proposed in-place sorting algorithm is an *unstable* sorting algorithm. A counter example showing the instability of the proposed in-place sorting algorithm follows.

Assume that the unsorted array **ELM** is as shown in Fig.1.

x
x
a
z
a
a
x

Fig. 1 The array ELM before sorting

Applying phase 1 of the proposed in-place sorting algorithm will result in properly placed segments. As one can see in Fig. 2, the x's have been placed in the buckets without preserving the original order. Also, there is no way to tell what the original ordering was.

In future work, further study of the stability of the algorithm and how to establish a stable version of it will be carried out.

Start End

x_1	-	-	-	a	a	a	a
x_2	x_2	x_2	x_2	x_2	-	a	a
a	a	a	a	a	a	a	a
z	x_1	x_1	x_1	x_1	x_1	x_1	x_1
a	a	a	x_3	x_3	x_3	x_3	x_3
a	a	a	a	a	x_2	x_2	x_2
x_3	x_3	z	z	z	z	z	z

Note: the subscripts on the x's are only for clarification of their relative position to each other. And highlighted cells indicate the elements that are put into their appropriate position during an iteration

Fig. 2 The changes in the array ELM when applying phase 1

VII. Experimental Results

To obtain experimental results, 3 of the better-known sorting algorithms were chosen and implemented: mergesort, in-place mergesort (k-way mergesort), and quicksort. A comparison of the properties of these basic sorting algorithms and our proposed in-place sorting algorithm is given in Table I below.

Each of the 3 basic sorting algorithms was used in turn as the xsort for Phase 2 of the proposed in-place sorting algorithm. This lead to the implementation of 3 versions of The proposed in-place sorting algorithm.

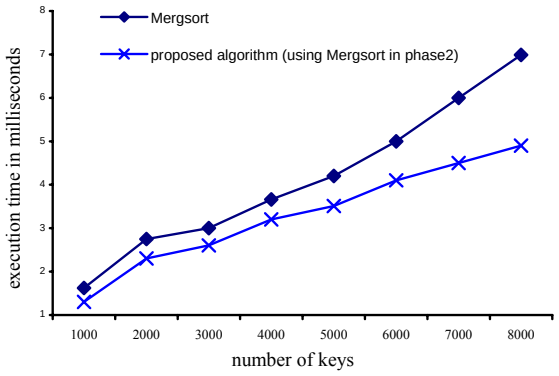
The 3 basic sorting algorithms and the 3 versions of The proposed in-place sorting algorithm were tested on different sizes of datasets where the number of keys to be sorted ranged from 1000 to 220000. The keys were 20 characters long. The execution time was measured for the 6 algorithms and recorded.

The experimental results show that each version of The proposed in-place sorting algorithm achieves better performance than its corresponding xsort algorithm alone. I.e., the proposed in-place sorting algorithm with merge sort algorithm as xsort performs better than merge sort alone and so forth (see Fig. 3). Complete results are presented in Fig. 4-6.

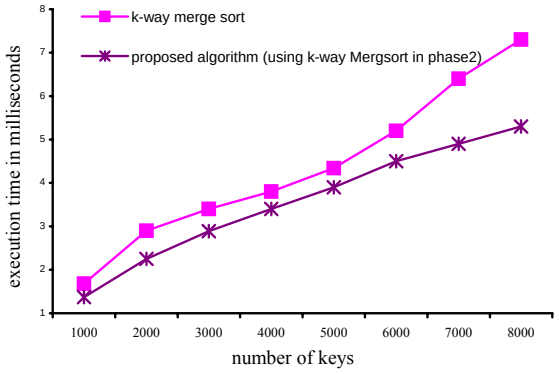
Finally, the best performance was achieved by the proposed in-place sorting algorithm with quicksort as the xsort algorithm.

TABLE I
LIST OF THE PROPERTIES OF SOME WELL KNOWN SORTING ALGORITHMS AND THE PROPOSED IN-PLACE SORTING ALGORITHM

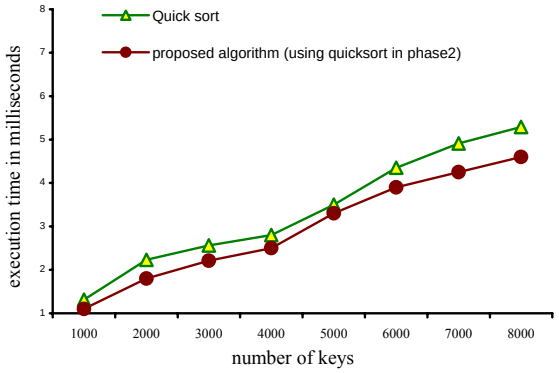
Sorting Algorithm	Average Time	Worst Time	Memory	Stable
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n)$	Yes
In-Place merge sort	$O(n \log n)$	$O(n \log n)$	$O(1)$	Yes
Quicksort	$O(n \log n)$	$O(n)$	$O(\log n)$	No
The proposed in-place sorting algoritm	$O(n \log z)$	$O(n \log z)$	$O(1)$	No



(a)



(b)



(c)

Fig. 3 Comparing The proposed in-place sorting algorithm and its xsort: (a) Mergesort, (b) k-way Mergesort, and (c) quicksort

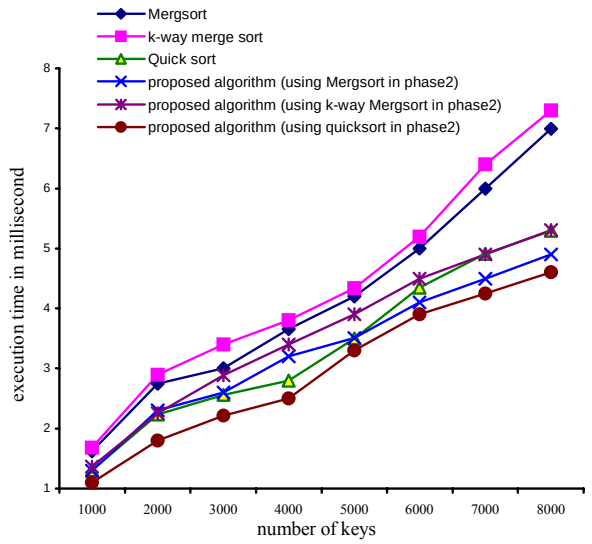


Fig. 4 The running times of sorting algorithms in milliseconds for a number of keys to be sorted ranging from 1000 to 8000 keys each key is 20 characters long

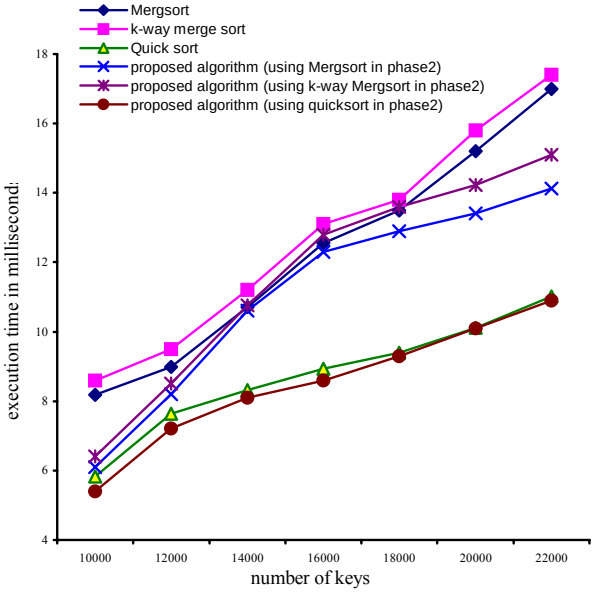


Fig. 5 The running times of sorting algorithms in milliseconds for a number of elements to be sorted ranging from 10000 to 22000 keys each key is 20 characters long

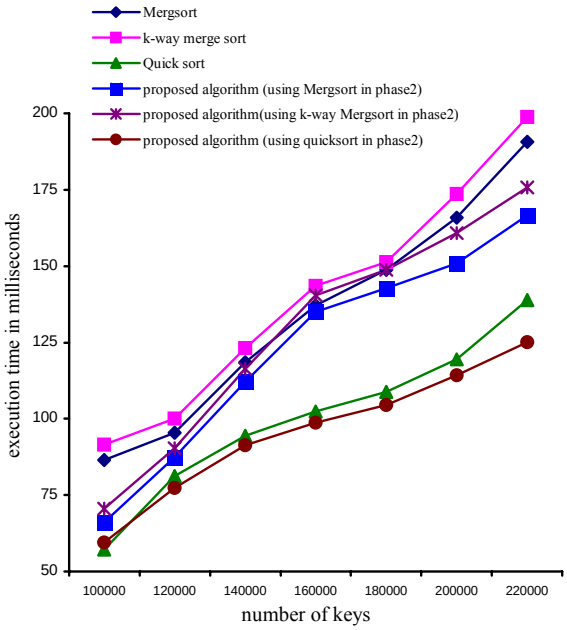


Fig. 6 The running times of sorting algorithms in milliseconds for a number of elements to be sorted ranging from 100000 to 220000 keys each key is 20 characters long

VIII. Conclusion

In this paper, a new sorting algorithm was presented. This algorithm is an in-place sorting algorithm with $O(n \log z)$ comparisons and $O(n \log z)$ moves and $O(1)$ auxiliary storage. Comparison of the new proposed algorithm and some well-known algorithms has proven that the new algorithm outperforms the other algorithms. Further, it was proved that no auxiliary arithmetic operations with indices were required. Besides, this algorithm is of practical interest because of its simplicity. Experimental results have shown that it outperformed other in-place sorting algorithms.

References

[1] A. LaMarca and R. E. Ladner, "The influence of caches on the performance of heaps," *Journal of Experimental Algorithmics*, vol. 1, Article 4, 1996.

[2] D. Knuth, *The Art of Computer Programming: Volume 3 / Sorting and Searching,* Addison-Wesley Publishing Company, 1973.

[3] Y. Azar, A. Broder, A. Karlin, and E. Upfal, "Balanced allocations," in *Proceedings of 26th ACM Symposium on the Theory of Computing*, 1994, pp.593-602.

[4] Andrei Broder and Michael Mitzenmacher."Using multiple hash functions to improve IP lookups," in *Proceedings of IEEE INFOCOM*, 2001.

[5] Jan Van Lunteren, "Searching very large routing tables in wide embedded memory," in *Proceedings of IEEE Globecom*, November 2001.

[6] Ramesh C. Agarwal, "A super scalar sort algorithm for RISC processors," *SIGMOD Record* (ACM Special Interest Group on Management of Data), 25(2):240–246, June 1996.

[7] R. Anantha Krishna, A. Das, J. Gehrke, F. Korn, S. Muthukrishnan, and D. Shrivastava, "Efficient approximation of correlated sums on data streams," *TKDE*, 2003.

[8] A. Arasu and G. S. Manku, "Approximate counts and quantiles over sliding windows," *PODS*, 2004.

[9] N. Bandi, C. Sun, D. Agrawal, and A. El Abbadi, "Hardware acceleration in commercial databases: A case study of spatial operations," *VLDB*, 2004.
[10] Peter A. Boncz, Stefan Manegold, and Martin L. Kersten, "Database architecture optimized for the new bottleneck: Memory access," in *Proceedings of the Twenty-fifth International Conference on Very Large Databases*, 1999, pp. 54–65.
[11] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*. MIT Press, Cambridge, MA, 2nd edition, 2001.
[12] Abhinandan Das, Johannes Gehrke, and Mirek Riedewald, "Approximate join processing over data streams," in *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, ACM Press, 2003, pp.40-51.
[13] A. LaMarca and R. Ladner, "The influence of caches on the performance of sorting," in *Proc. of the ACM/SIAM SODA*, 1997, pp. 370–379.
[14] Stefan Manegold, Peter A. Boncz, and Martin L. Kersten, "What happens during a join? Dissecting CPU and memory optimization effects," in *Proceedings of 26th International Conference on Very Large Data Bases*, 2000, pp. 339–350.
[15] A. Andersson, T. Hagerup, J. H°astad, and O. Petersson, "Tight bounds for searching a sorted array of strings," *SIAM Journal on Computing*, 30(5):1552–1578, 2001.
[16] L. Arge, P. Ferragina, R. Grossi, and J.S. Vitter, "On sorting strings in external memory," *ACM STOC '97*, 1997, pp.540–548.
[17] M.A. Bender, E.D. Demaine, andM. Farach-Colton, "Cache-oblivious B-trees," *IEEE FOCS '00*, 2000, pp.399–409.
[18] J.L. Bentley and R. Sedgewick, "Fast algorithms for sorting and searching strings," *ACM-SIAM SODA '97*, 1997, pp.360–369.
[19] G. Franceschini, "Proximity mergesort: Optimal in-place sorting in the cacheoblivious model," *ACM-SIAM SODA '04*, 2004, pp.284–292.
[20] G. Franceschini. "Sorting stably, in-place, with O(n log n) comparisons and O(n) moves," *STACS '05*, 2005.
[21] G. Franceschini and V. Geffert. "An In-Place Sorting with O(n log n) Comparisons and O(n) Moves," *IEEE FOCS '03*, 2003, pp. 242–250.