IN-PLACE SORTING WITH FEWER MOVES*

JYRKI KATAJAINEN†
Department of Computing, University of Copenhagen
Universitetsparken 1, DK-2100 Copenhagen East, Denmark

TOMI A. PASANEN
Turku Centre for Computer Science
Lemminkäisenkatu 14A, FIN-20520 Turku, Finland

Abstract. It is shown that an array of n elements can be sorted using $O(1)$ extra space, $O(n \log n / \log \log n)$ element moves, and $n \log_2 n + O(n \log \log n)$ comparisons. This is the first in-place sorting algorithm requiring $o(n \log n)$ moves in the worst case while guaranteeing $O(n \log n)$ comparisons, but due to the constant factors involved the algorithm is predominantly of theoretical interest.

Keywords. In-place algorithms, sorting, merging, mergesort, multiway merge.

1. Introduction

In **array sorting** we are given an array of n elements, each consisting of a key and some information associated with that key, and the task is to reorder these in ascending order according to their keys. To carry out the sorting we assume that the only operations allowed for the elements are *key comparisons* and *element moves*. Moreover, we want to perform the sorting **in-place**, i.e., we assume that besides the input array there is a constant number of memory locations available for storing elements and a constant number of machine words, each consisting of $O(\log n)$ bits, for storing counters, pointers, and indices. Normal *logical* and *arithmetical operations*, including the unrestricted shift to both directions, are assumed to be allowed when manipulating these words.

The performance of the algorithms is measured by counting the number of element moves, key comparisons, and word-manipulation operations performed in the worst case. If x and y denote the respective number of trivial and non-trivial cycles in the permutation required to sort an array of n elements, then $n - x + y$ element moves are necessary to sort the array [8]. In the worst case, $x = 0$ and $y = \lfloor n/2 \rfloor$, implying that

* Technical Report 98/22, Department of Computing, University of Copenhagen, Copenhagen, Denmark (1998).

† Supported partially by the Danish Natural Science Research Council under contract 9701414 (project Experimental Algorithmics).

$\lfloor 3n/2 \rfloor$ moves might be needed. By the standard information-theoretic argument, any comparison-based sorting algorithm must carry out at least $n \log_2 n - n \log_2 e + (1/2) \log_2 n + O(1)$ key comparisons [5, Section 5.3.1]. For every algorithm discussed in this paper the number of word-manipulation operations will always be bounded from above by $O(M(n) + C(n))$ if the number of moves and comparisons performed is $M(n)$ and $C(n)$, respectively. Therefore, the cost of the word manipulation will be omitted in the subsequent analysis.

Several in-place sorting algorithms are known that are efficient with respect to the number of moves performed. When stating the performance of known algorithms, we let ε denote a *fixed* positive constant not greater than 1. Munro and Raman [8] showed that the exact optimum number of moves, $n - x + y$, is obtainable by an algorithm that performs $O(2^{(1/\varepsilon)}(1/\varepsilon)!\, n^{2+\varepsilon})$ comparisons. Both the selection-sort [5, Section 5.2.3] and permutation-sort (see, e.g., [8]) algorithms perform $O(n)$ moves and $O(n^2)$ comparisons. This was improved to $O(n/\varepsilon)$ moves and $O(n^{1+\varepsilon}/\varepsilon)$ comparisons by Munro and Raman [7]. The variants of the mergesort algorithm introduced by Reinhardt [10] and Katajainen et al. [4] require $\varepsilon\, n \log_2 n$ moves and $n \log_2 n + O(n)$ comparisons.

If an auxiliary array of n words is available, it is easy to modify any efficient in-place sorting algorithm to perform $O(n)$ element moves and $n \log_2 n + O(n)$ key comparisons (cf., [5, p. 74]). It is even possible to reduce the amount of extra memory to $O(n^\varepsilon/\varepsilon)$ so that the number of moves is $O(n/\varepsilon)$ and that of comparisons $O(n \log n)$, as shown by Munro and Raman [7]. Furthermore, they proved that an array of n elements can be sorted in-place by performing $O(n)$ moves and $O(n \log n)$ comparisons on an average; in the worst case the number of comparisons is of order n^2.

Munro and Raman [7] stated as an open problem whether there exists an in-place sorting algorithm that performs $O(n)$ moves and $O(n \log n)$ comparisons in the worst case. In this paper we describe an algorithm that is asymptotically superior to the earlier worst-case algorithms but it is still unable to reach the above-mentioned ultimate goal. Our algorithm sorts an array of n elements using $O(1)$ extra space, $O(n \log n / \log \log n)$ moves, and $n \log_2 n + O(n \log \log n)$ comparisons. This result is proved in two stages: in Section 2 we recall a simplified version of the in-place mergesort algorithm of Katajainen et al. [4], on which our algorithm is based, and in Section 3 we show how the key subroutine needed, the multiway mergesort algorithm with a work zone, can be implemented efficiently.

2. In-place mergesort

Assume that the array being sorted is $A[0\,..\,n-1]$ and $n \geq 2$. We call any subarray occupying some consecutive positions of A a **zone**, and a collection of elements stored in a zone a **sequence**. Let 2^k be the largest power of 2 smaller than n, i.e., $2^k < n \leq 2^{k+1}$. We divide the array A into $k+2$ zones:

A_0 is the zone $A[0..0]$, i.e., it consists of the single element $A[0]$, A_i is the zone $A[2^{i-1} \mathinner{..} 2^i-1]$ for $i \in \{1, 2, \ldots, k\}$, and A_{k+1} is the zone $A[2^k \mathinner{..} n-1]$. We let s_i denote the size of A_i, i.e., $s_0 = 1$, $s_i = 2^{i-1}$ for $i \in \{1, 2, \ldots, k\}$, and $s_{k+1} = n-2^k$. Now the array A is sorted in two phases.

In the **sorting phase**, for each $i = k+1, k, \ldots, 2$, the sequence in A_i is sorted by d-way mergesort which utilizes $A[0 \mathinner{..} s_i - 1]$ as a work zone. The parameter d is to be determined later. Each sorting is carried out by repeated d-way merges by moving the elements back and forth between the two zones until A_i contains all its original elements in sorted order. In particular, each time an element is moved from one location to another some other element is put in the place of the element just moved so that no elements are lost.

In the **merging phase**, the sorted sequences created are merged together. The sequences in A_0 and A_1 are merged first and then, for each $i = 2, 3, \ldots, k+1$, the just merged sequence in $A[0 \mathinner{..} 2^{i-1}-1]$ is merged with the sequence in A_i. These 2-way merges are carried out in-place by using any efficient in-place merging algorithm, e.g., the fast algorithm given in [3].

The in-place mergesort algorithms described in [4, 10] are similar to the foregoing algorithm; they just required that the parameter d is a constant. The d-way mergesort algorithm can be implemented such that it sorts a sequence of size m, when a work zone of size m is available, using $O(d)$ extra space, $2m \log_d m + O(m)$ moves, and $m \log_2 m + O(m \log d)$ comparisons [4]. Since $\sum_{i=2}^{k+1} s_i = n-2$, the number of moves performed in the sorting phase is bounded by $2n \log_d n + O(n)$ and that of comparisons by $n \log_2 n + O(n \log d)$. The cost of a single 2-way merge, even when carried out in-place, is linear in relation to the sum of the sizes of the sequences being merged (see, e.g., [3]). Hence, in the worst case the number of moves and comparisons performed in the merging phase is proportional to $\sum_{i=1}^{k} 2s_i + \sum_{i=0}^{k+1} s_i$, which is $O(n)$. That is, the computational costs are dominated by those of the sorting phase.

3. New in-place mergesort

3.1. Algorithm and its analysis

Let $A[0 \mathinner{..} n-1]$ be the array being sorted, $n \geq 2^{16}$, and d a power of 2 such that $\log_2 n / \log_2 \log_2 n \leq d < 2 \log_2 n / \log_2 \log_2 n$. Before the actual sorting, we divide the array $A[0 \mathinner{..} n-1]$ into two zones: the **encoding zone** $A[0 \mathinner{..} 2e-1]$ and the **mergesort zone** $A[2e \mathinner{..} n-1]$, where $e = d\lceil \log_2 n \rceil$. The encoding zone is used for storing d indices implicitly by means of the original elements as described, for example, in [6]. These indices are needed for the implementation of the d-way mergesort algorithm used as a subroutine in the algorithm described in Section 2.

The overall structure of our in-place sorting algorithm is the following. First, suitable elements are gathered into the encoding zone; Section 3.2 gives the details. Second, the remaining sequence in the mergesort zone is sorted by the algorithm of Section 2 but now the d-way mergesort algorithm used in its sorting phase is implemented as described in Section 3.3. Third, the elements in the encoding zone are sorted by using any efficient in-place sorting algorithm. Fourth, the sorted sequences in the two zones are merged by using any efficient in-place merging algorithm. This completes the sorting of the whole array $A[0\,..\,n-1]$.

In Section 3.2 we show that the creation of the encoding zone can be done in-place with $O(n)$ moves and $O(n)$ comparisons. In Section 3.3 we show that any subsequence of size m can be sorted using $O(1)$ extra space, $4m\log_d m + O(m)$ moves, and $m\log_2 m + O(m\log d)$ comparisons when a work zone of size m and an encoding zone of size $2e$ are available. This implies that the sorting phase of the algorithm given in Section 2 requires at most $4n\log_d n + O(n)$ moves and $n\log_2 n + O(n\log d)$ comparisons. Recall that the merging phase of the algorithm of Section 2 requires only $O(n)$ moves and $O(n)$ comparisons. Since the size of the encoding zone is only $O((\log n)^2/\log\log n)$, its sorting takes $o(n)$ moves and comparisons. The in-place merging of the sorted sequences in the two zones requires $O(n)$ moves and $o(n)$ comparisons [3].

To summarize, the number of moves performed is at most $4n\log_d n + O(n)$ and that of comparisons $n\log_2 n + O(n\log d)$. For $\log_2 n/\log_2\log_2 n \le d < 2\log_2 n/\log_2\log_2 n$ and $n \ge 2^{16}$, $(1/2)\log_2\log_2 n \le \log_2 d < 2\log_2\log_2 n$. Therefore, the number of moves performed is bounded above by $8n\log_2 n/\log_2\log_2 n + O(n)$ and that of comparisons by $n\log_2 n + O(n\log\log n)$.

3.2. Creation of the encoding zone

In the encoding zone we want to store d indices, each being an integer drawn from the range $\{0,\ldots,n-1\}$. To present such an integer we need $\lceil\log_2 n\rceil$ bits. Two elements with distinct keys can encode one bit. For example, if the key of x is smaller than that of y, by storing the elements in the order $x\,y$ may denote a 0-bit and the opposite order a 1-bit. Hence, $e = d\lceil\log_2 n\rceil$ pairs of elements with distinct keys can encode e indices. Observe that to read the value of such an encoded index requires $O(\log n)$ comparisons and to update the value of an index requires $O(\log n)$ moves.

The pairs of elements needed can be found as follows. First, the element with the median key in the input array $A[0\,..\,n-1]$ is searched for by using any efficient in-place selection algorithm (see, e.g., [6]). Second, a 3-way partitioning of the array around the element with the median key is performed (see, e.g., [2]). Let $A_<$, $A_=$, and $A_>$ denote the three sequences created. If the size of both $A_<$ and $A_>$ is less than e, we sort both of them by using any efficient in-place sorting algorithm and we are done. Hence, assume that either the size of $A_<$ or the size of $A_>$ is larger than or equal to e. Since $A_=$ contains the elements whose key is equal to the median key and n is so large compared to e, the keys of the first e elements and those of the last e elements in $A[0\,..\,n-1]$ must be pairwise distinct. The elements in the zones $A[0\,..\,e-1]$ and $A[n-e\,..\,n-1]$ are moved interleaved into the zone

$A[0\,..\,2e{-}1]$, starting from the rear, after which the creation of the encoding zone is finished.

The computational costs of this procedure are dominated by those of the median finding and partitioning. Both of these routines require $O(n)$ moves and $O(n)$ comparisons, which means that the creation of the encoding zone is done within the same resource bounds.

3.3. Multiway mergesort with a work zone and an encoding zone

In this section we show how a sequence of size m can be sorted efficiently by the d-way mergesort algorithm when a work zone of size m and an encoding zone of size $2d\lceil\log_2 n\rceil$ are available. Here n is an integer such that $n \geq 2m$, and $d = \Theta(\log n/\log\log n)$. For the sake of clarity, we assume that $B[0\,..\,m-1]$ is the array to be sorted, $W[0\,..\,m-1]$ the work zone, and $A[0\,..\,2d\lceil\log_2 n\rceil-1]$ the encoding zone. In reality, all these zones are parts of the original array $A[0\,..\,n-1]$ which is being sorted by the algorithm of Section 3.1. The sorting of the array B is now carried out as follows.

Initially, each element in $B[0\,..\,m-1]$ is considered to form a sorted sequence of length one. In one **pass**, the collection of sorted sequences is divided into the **groups** of d consecutive sequences, except the last group that can contain fewer than d sequences; the sequences of each group are merged by moving the elements in sorted order from the original zone to the work zone. The roles of the two zones are interchanged and the process is repeated until only one sorted sequence remains. In the last pass, the elements are moved to the original zone if they are not there already. As pointed out in the previous section, the work zone contains also some elements but it is trivial to organize the moves so that these elements are not lost, even though their order may change (i.e., the sorting algorithm is *not* stable).

We make one substantial change to this fairly standard procedure. When the size of the sorted sequences becomes larger than $\ell = \lceil(\log_2 n)^2\rceil$, these are divided into **blocks** of size ℓ, except the last block that can be smaller than the others. If the size of a sorted sequence is not larger than ℓ, it is seen as a single block. We call the elements that are still to be merged **active**. In each sequence the first block that still contains active elements is called the **leading block**. In the algorithm we maintain an invariant that the location of the leading block is fixed; the leading block is kept in the zone originally occupied by the first block of the sequence. Hence, the position of the leading block of the ith sequence can be calculated by using d, e, i, the pass number, and the index of the group being merged.

To carry out a merge of d sequences, the active element with the smallest key from each of the d sequences under consideration is kept in a **selection tree** as proposed in [5, Sections 5.2.3 and 5.4.1]. This tree is used when seeking for the element with the minimum key among the active elements. After finding this element, it is moved to the work zone. The tree must also be updated by removing a reference to the element just moved and adding a new reference to the next element, if any, in the same sequence since this element becomes a candidate as the new overall minimum.

We number the nodes of the selection tree from 1 to $2d - 1$. Like in a heap, node 1 is the **root** of the tree, node $\lfloor i/2 \rfloor$ is the **parent** of node i, if $i > 1$, and nodes $2i$ and $2i+1$ are the **children** of node i, if those exists. Nodes from d to $2d - 1$ will be called **leaves** and the other nodes **branches**. For $i \in \{d, \ldots, 2d - 1\}$, node i is said to be the jth leaf if $j = i - d + 1$. Each node of the tree stores an $O(\log \log n)$-bit integer. Since d is $O(\log n / \log \log n)$, the whole selection tree can be stored in a few words of $O(\log n)$ bits each. Observe that the position of the parent or the children of a node can be calculated by using a constant number of shifts and other arithmetical operations.

In our data structure, illustrated in Fig. 1, we maintain three kinds of indices: **implicit indices**, **small indices**, and **large indices**. Every index indicates a position in the array B (A) or a node in the selection tree. Therefore, the indices are visualized as pointers in Fig. 1. The parent and children of a node in the selection tree are indicated by implicit indices. The ith leaf of the selection tree has an implicit index to the beginning of the leading block of the ith sequence being merged. Moreover, the ith leaf stores explicitly an offset to the first active element inside the leading block; an offset is a small index whose presentation uses $O(\log \log n)$ bits. Each branch of the selection tree stores a small index to the leaf containing the active element with the smallest key in the leaves of the subtree rooted by this particular branch. Finally, the encoding zone stores d large indices, i.e., indices whose representation requires $O(\log n)$ bits; the ith of these indices indicates the next full block inside the ith sequence that still contains active elements. If no such full block exists, the index has the value zero.

In the beginning of each d-way merge, the implicit indices from the selection tree to the sequences under consideration are initialized simply by updating the group index. The offsets at the leaves of the selection tree are initialized to zero indicating that the first element in each leading block is the active element with the smallest key in the corresponding sequence. The large indices for each sequence are initialized to point to the beginning of the second block, if there is any. Since the location of the leading block is fixed, the implicit indices from the selection tree to the sequences are valid all the time.

The initialization of the small indices in the branches of the selection tree is done in a bottom-up manner. For each branch at every level, the elements with the smallest key within the subtrees rooted by the children of that particular branch are accessed and the small index is assigned to point to the leaf that contained an element with the smaller key. Clearly, at most d comparisons are necessary during this initialization. In all passes this initialization is done $O(m/d)$ times so the overall cost caused by these is linear.

After the construction of the selection tree, it is used to find the active element with the smallest key. By using the small index stored at the root,

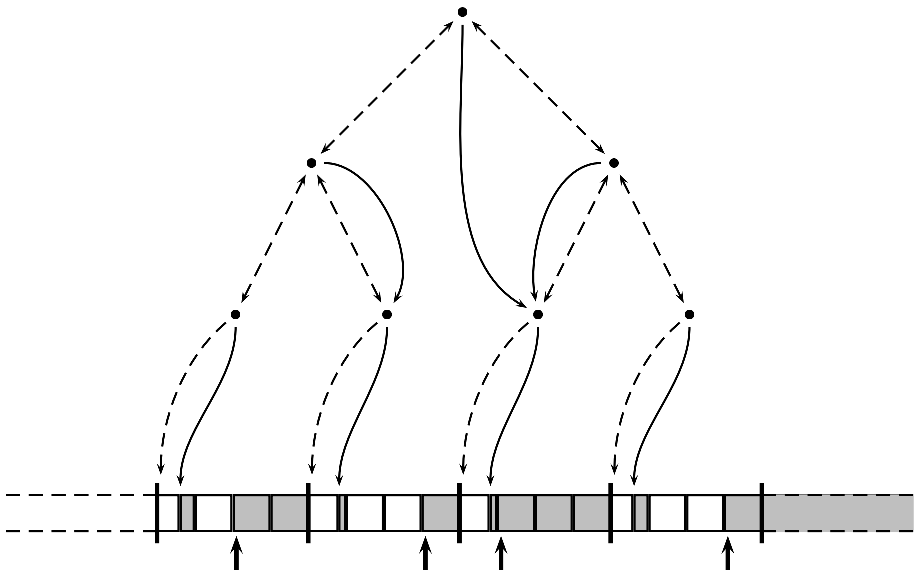

FIGURE 1. The overall data structure when $d = 4$ and each sequence consists of 4 blocks. The implicit indices are visualized with dashed arrows, small indices with sold arrows, and large indices with bold arrows. The darkened zones in the array contain still elements to be merged.

the implicit index of the leaf pointed to by the root, and the offset stored at the leaf, the active element to be moved to the work zone is easily located. After the element is moved to the work zone (and an element from there to the location occupied by the element just moved), the offset at the leaf is incremented by one and the small indices in each branch on the path to the root are updated if necessary. This updating requires $\log_2 d$ comparisons. The number of comparisons performed in one pass is at most $m \log_2 d$ and that over all $\log_d m + O(1)$ passes $m \log_2 d \log_d m + O(m \log d)$, which is $m \log_2 m + O(m \log d)$.

Let us now calculate the number of moves performed. In one pass there may be two reasons why an element in the array B is moved: 1) a leading block becomes empty and the next block containing active elements, if there is any, should be moved to the zone occupied by the previous leading block; 2) an element from a leading block is moved to the work zone. Since the order of the elements in the work zone can be changed, by using the *hole technique* as described in [3], the elements in two blocks of size ℓ can be swapped with $2\ell + O(1)$ moves. There are at most $\lceil m/\ell \rceil$ blocks so the number of moves caused by these block swaps is $2m + O(m/\ell)$ per a pass. By maintaining a hole at the current output position in the work zone, the active element with the minimum key can be moved into this hole and the element next to this hole into the location occupied by the element just moved, which creates a new hole at the next output position. This organization guarantees that the number of moves caused by outputting is $2m + O(1)$ per a pass. Therefore, at most $4m + O(m/\ell)$ moves are carried out in each pass, the total number of moves over all $\log_d m + O(1)$ passes being at most $4m \log_d m + O(m)$.

Some moves are also necessary in the encoding zone due to the updates of the large indices. When the block size is not larger than ℓ, the large indices are not needed at all. When the block size is larger than ℓ, the large indices are in use. However, in one pass each of the at most $\lceil m/\ell \rceil$ blocks is moved only once. Hence, the number of updates of the large indices is bounded by $O(m/(\log n)^2)$. The cost of each update is $O(\log n)$ which means that the overall cost caused by these index updates over all passes is sublinear because $n \geq 2m$.

This completes the description and the analysis of the d-way mergesort algorithm used as a subroutine in our in-place sorting algorithm. To sum up, the number of moves performed is bounded by $4m \log_d m + O(m)$ and that of comparisons by $m \log_2 m + O(m \log d)$.

4. Final remarks

We have showed that an array of n elements can be sorted using $O(1)$ extra space, $O(n \log n / \log \log n)$ element moves, and $n \log_2 n + O(n \log \log n)$ key comparisons. This performance is guaranteed in the worst case. The main idea in our algorithm was to utilize the word parallelism and store the selection tree in a few machine words. It seems difficult to develop this idea any further since a larger merging factor will automatically mean a larger selection tree, which cannot be stored in a constant number of words any more. On the other hand, the encoding technique could be used to store the selection tree implicitly, but the updates of the indices would force us to use more moves as well.

For two reasons our algorithm is primarily of theoretical interest: 1) the index manipulation needed is complicated and 2) in practice, $\log_2 \log_2 n$ is seldom larger than 5 or 6 so the constant factor in the leading term in the number of moves makes the algorithm impractical. This suggests that in-place algorithms should not only be designed with asymptotic analysis in mind. The earlier papers [4, 10] and the present paper all give a different implementation for the merging phase of the algorithm described in Section 2. It would be interesting to know which of the proposals leads to the fastest practical implementation.

Recently, some interesting non-comparison-based algorithms for sorting integers have been developed (for a survey, see [1]). All these algorithms require linear, or even more, extra space. The classical time-space trade-off results (see, e.g., [9]) assume a read-only memory whereas we allowed reordering of the input through element moves. It is natural to ask what is the fastest *in-place* algorithm for sorting (small) integers under our model of computation.

References

[1] A. ANDERSSON, Sorting and searching revisited, in *Proceedings of the 5th Scandinavian Workshop on Algorithm Theory*, Lecture Notes in Computer Science **1097**, Springer-Verlag, Berlin/Heidelberg, Germany (1996), 185–197.

[2] J. L. BENTLEY AND M. D. MCILROY, Engineering a sort function, *Software—Practice and Experience* **23** (1993), 1249–1265.

[3] V. GEFFERT, J. KATAJAINEN, AND T. PASANEN, Asymptotically efficient in-place merging, *Theoretical Computer Science*, to appear.

[4] J. KATAJAINEN, T. PASANEN, AND J. TEUHOLA, Practical in-place mergesort, *Nordic Journal of Computing* **3** (1996), 27–40.

[5] D. E. KNUTH, *The Art of Computer Programming, Volume 3/Sorting and Searching*, Addison-Wesley Publishing Company, Reading, Massachusetts (1973).

[6] T. W. LAI AND D. WOOD, Implicit selection, in *Proceedings of the 1st Scandinavian Workshop on Algorithm Theory*, Lecture Notes in Computer Science **318**, Springer-Verlag, Berlin/Heidelberg, Germany (1988), 14–23.

[7] J. I. MUNRO AND V. RAMAN, Sorting with minimum data movement, *Journal of Algorithms* **13** (1992), 374–393.

[8] J. I. MUNRO AND V. RAMAN, Selection from read-only memory and sorting with minimum data movement, *Theoretical Computer Science* **165** (1996), 311–323.

[9] J. PAGTER AND T. RAUHE, Optimal time-space trede-offs for sorting, Unpublished manuscript.

[10] K. REINHARDT, Sorting *in-place* with a *worst case* complexity of $n \log n - 1.3n + O(\log n)$ comparisons and $\varepsilon n \log n + O(1)$ transports, in *Proceedings of the 3rd International Symposium on Algorithms and Computation*, Lecture Notes in Computer Science **650**, Springer-Verlag, Berlin/Heidelberg, Germany (1992), 489–498.