Nordic Journal of Computing

PRACTICAL IN-PLACE MERGESORT*

JYRKI KATAJAINEN
Department of Computer Science, University of Copenhagen
Universitetsparken 1, DK-2100 Copenhagen East, DENMARK
Electronic mail: `jyrki@diku.dk`

TOMI PASANEN†
Turku Centre for Computer Science,
Lemminkäisenkatu 14A, FIN-20520 Turku, FINLAND
Electronic mail: `Tomi.Pasanen@utu.fi`

JUKKA TEUHOLA‡
Department of Computer Science, University of Turku
Lemminkäisenkatu 14 A, FIN-20520 Turku, FINLAND
Electronic mail: `Jukka.Teuhola@utu.fi`

Abstract. Two in-place variants of the classical mergesort algorithm are analysed in detail. The first, straightforward variant performs at most $N \log_2 N + O(N)$ comparisons and $3N \log_2 N + O(N)$ moves to sort N elements. The second, more advanced variant requires at most $N \log_2 N + O(N)$ comparisons and $\varepsilon N \log_2 N$ moves, for any fixed $\varepsilon > 0$ and any $N > N(\varepsilon)$. In theory, the second one is superior to advanced versions of heapsort. In practice, due to the overhead in the index manipulation, our fastest in-place mergesort behaves still about 50 per cent slower than the bottom-up heapsort. However, our implementations are practical compared to mergesort algorithms based on in-place merging.

Key words: sorting, mergesort, in-place algorithms

CR Classification: F.2.2

1. Introduction

Assume that we are given an array A of N elements that are to be sorted. Mergesort is a classical sorting routine that can iteratively be described as follows. Initially, each element is thought to form a sorted subsequence of size one. These subsequences are merged pairwise, and in this way we can (almost) halve the number of sorted subsequences. This process is then repeated until we have only one sorted sequence left, containing all the elements.

---

* Presented at the *7th Finnish Symposium on Computer Science*, Liperi, Finland, January 1994.

† Supported by the Academy of Finland under contract No. 1021129.
‡ Supported by the Academy of Finland under contract No. 1071337.

Received February 1996.

The heart of the construction is the merge routine, which combines two sorted sequences into one. The drawback of the straight 2-way merge algorithm [9], which repeatedly moves the smaller of the minima of the remaining subsequences to an output area, is that it requires extra storage space for its operation. If the sequences to be merged are of size m and n, respectively, then a trivial implementation of the 2-way merge requires at most $m+n-1$ comparisons, $m+n$ element moves, and $m+n$ extra storage locations.

Assuming that we have an extra array B of size N available, the mergesort algorithm can be technically realized so that the elements of A are merged back and forth from A to B. With the 2-way merge routine, the number of comparisons performed will be $N \log_2 N + O(N)$ and the number of element moves also $N \log_2 N + O(N)$. If K-way merge is applied, the number of moves can be reduced to $(2/\lceil \log_2 K \rceil) N \log_2 N + O(N)$ without affecting the number of comparisons (cf. Section 3).

In this paper we explore the question as to how the mergesort algorithm can be turned into an efficient *in-place* algorithm. More precisely, we assume that only one extra storage location (in addition to the array A) is available for storing the data elements and $O(1)$ storage locations are available for storing indices of the input array A. The only operations permitted on the data elements are comparisons and moves. In the basic algorithm, the indices are manipulated only by addition and subtraction. (The algorithm could be implemented, without loss of efficiency, even if addition by one and subtraction by one were the only arithmetic operations allowed. These details are, however, left for the interested reader.) In the fine-tuned version, we also need division (or shift) as in advanced heapsort variations [3].

In the theoretical analysis of the in-place algorithms to be presented, we calculate the number of element comparisons and element moves (or assignments) made in the worst case. The ultimate goal would be a sorting algorithm that performs $N \log_2 N + O(N)$ comparisons and $O(N)$ moves (cf. Munro and Raman [11]). The performance of our algorithms is well characterized by these two quantities, since the work required by the index manipulations is closely related to the number of the above operations. As to the practical efficiency of various in-place algorithms, the index manipulation is of course of importance (cf. Section 4).

Many algorithms for in-place merging have been proposed, the present champion being the one by Huang and Langston [7]. Their algorithm merges two sequences of size m and n in $O(m+n)$ time or, more precisely, it performs at most $1.5(m+n) + O(\sqrt{m+n}\log_2(m+n))$ comparisons and $5(m+n) + O(\sqrt{m+n}\log_2(m+n))$ moves[1]. When this merge routine is used for implementing mergesort, N elements will be sorted in-place with at most $1.5N \log_2 N + O(N)$ comparisons and $5N \log_2 N + O(N)$ moves.

We shall show that mergesort can be implemented in-place more efficiently,

[1] Actually, in [12] it was shown that the number of moves is bounded by $6(m+n) + O(\sqrt{m+n}\log_2(m+n))$ but by using the "hole" technique, to be described in Section 2, an implementation requiring at most $5(m+n)+O(\sqrt{m+n}\log_2(m+n))$ moves is obtained.

if we use the standard merge routine as a starting point, instead of the known in-place merge algorithms. The basic observation (see [10, Lemma 3] or [9, Exercise 5.2.4-10]) is that the 2-way merge can be easily modified such that it will merge two sequences of size m and n by using only $\min(m, n)$ extra space. One way to obtain this is as follows. For the sake of simplicity, assume that the sequences X and Y to be merged are consecutive subsequences of the same array, as they are in mergesort, and that X is not longer than Y. Now, move first the elements of X to a work area W, and then merge W and Y in the usual manner to the area covered by X and Y.

Our mergesort variant is based on the *partitioning principle* that has turned out to be useful in many contexts (see, e.g. [4, 10, 13]). First, we sort about one half of the elements by using the second half of the input array as a work area. Second, we sort half of the second half in the same way. Third, we merge the two sequences of sizes about $N/2$ and $N/4$ by using the last $N/4$ positions of the input array as a work area. Then we repeatedly sort half of the remaining elements and merge this block together with the big sorted block obtained so far. It is not difficult to see that this algorithm runs in $O(N \log_2 N)$ time. A detailed analysis, to be given in Section 2, shows that the algorithm can be easily implemented such that it will perform at most $N \log_2 N + O(N)$ comparisons and $3N \log_2 N + O(N)$ moves.

In Section 3 we describe an advanced version of the algorithm and show that it performs at most $N \log_2 N + O(N)$ comparisons and $\varepsilon N \log_2 N$ moves, for any fixed $\varepsilon > 0$ and any $N > N(\varepsilon)$. (Observe that ε has to be a fixed constant, since the amount of extra space required by the algorithm is exponential on $1/\varepsilon$.) These figures should be compared to the corresponding bounds known, for example, for heapsort. Floyd's improvement [6] of the standard heapsort [19] makes $2N \log_2 N + O(N)$ comparisons and $N \log_2 N + O(N)$ moves, and the bottom-up heapsort [5, 18] $1.5N \log_2 N + O(N)$ comparisons and $1.5N \log_2 N + O(N)$ moves, in the worst case. More advanced heapsort variations exist (see, e.g. [2, 3]), but these assume that the general shift operation is a constant time operation. (Observe that already bottom-up heapsort uses the division operation.) If the elements to be sorted are all distinct, the number of moves performed by the heapsort algorithms is at least $0.5N \log_2 N - O(N)$ [16]. So both the number of comparisons and moves made by heapsort are in the worst case larger than those made by our in-place mergesort.

In practical experiments our fastest in-place mergesort program was about 50 per cent slower than the bottom-up heapsort program. Therefore, the number of element comparisons and element moves does not tell the whole truth. However, if the efficiency is compared to earlier in-place mergesort implementations, e.g., that utilizing the in-place merging algorithm of Huang and Langston [7], our implementation is considerably faster. The experimental results are reported in Section 4.

It should be observed that a mergesort version similar to ours appears already in the seminal paper by Kronrod [10] (see also [15]). The main contribution in this paper is the detailed analysis of the algorithm and the evaluation of its practical efficiency. Observe also that an in-place sorting algorithm performing $N \log_2 N + O(N)$ comparisons and $\varepsilon N \log_2 N$ moves has been developed independently by Reinhardt [14]. He uses similar techniques as we do, but his main results concern minimizing the number of comparisons, that is, the constant in the linear term. He also applies quicksort-like partitioning, which in our opinion does not belong to a "pure" mergesort algorithm. Without exaggerating, it can be said that our algorithm is simpler than that given in [14].

2. Straightforward in-place mergesort

In this section we describe an in-place sorting algorithm that uses merging as its basic tool. This algorithm is a variant of the second in-place sorting algorithm proposed by Kronrod [10]. We also analyse the efficiency of the algorithm. A more fine-tuned version will be given in the next section.

Assume that the elements to be sorted are given in an array A of size N. We use the notation $\langle i, j \rangle$ to denote the *block* $A[i..j]$ of elements. The case $i > j$ represents an empty block. By $X - Y$ we mean a block, which is left over X when the elements of Y are removed from X, and by $X + Y$ a block which includes all elements of X and Y. Two blocks P and Q are said to be *equivalent*, denoted $P \equiv Q$, if they contain precisely the same elements of A, possibly in a different order. The *size* $|P|$ of a non-empty block $P = \langle i, j \rangle$ is $j - i + 1$. The size of an empty block is 0.

In our sorting algorithm, illustrated in Fig. 1, we maintain an invariant that array A consists of two consecutive blocks, P and Q, where P contains some elements in no specific order, whereas the elements of Q are in sorted order. An initial situation is created by choosing $P = \langle 1, \lceil N/2 \rceil \rangle$ and sorting the rest of A to Q by using P as the work area. The size of Q is increased gradually as follows. First, we divide P into two blocks, P_1 and P_2, so that $|P_1| = |P_2|$ or $|P_1| = |P_2| - 1$. Second, we sort P_1 by using the standard mergesort routine. Block P_2 is used as a work area. As a result we get two blocks P_1' and P_2', where $P_1' \equiv P_1$ and $P_2' \equiv P_2$, and the elements of P_1' are in sorted order. Third, we merge the blocks P_1' and Q. The output is produced "over" P_2' and Q. As a result of this merge we get two consecutive blocks, P_2'' and Q', where $P_2'' \equiv P_2' \equiv P_2$ and $Q' \equiv P_1' + Q$, and the elements of Q' appear in sorted order. Now we assign $P \leftarrow P_2''$ and $Q \leftarrow Q'$, and repeat the process, until $|P| = 1$ in which case the single element is moved into its right place in Q, pushing the smaller elements one step left.

For the purpose of the analysis, two parts of the algorithm can be readily separated: the *sort part* consists of the calls to the mergesort routine and the *merge part* consists of the calls to the merge routine (outside mergesort). Let $S(m)$ denote the time required by the sort part after P has reached the size m, and let $M(m, n)$ denote the time required by the merge part after P has reached the size m and Q the size n. The time requirements of both

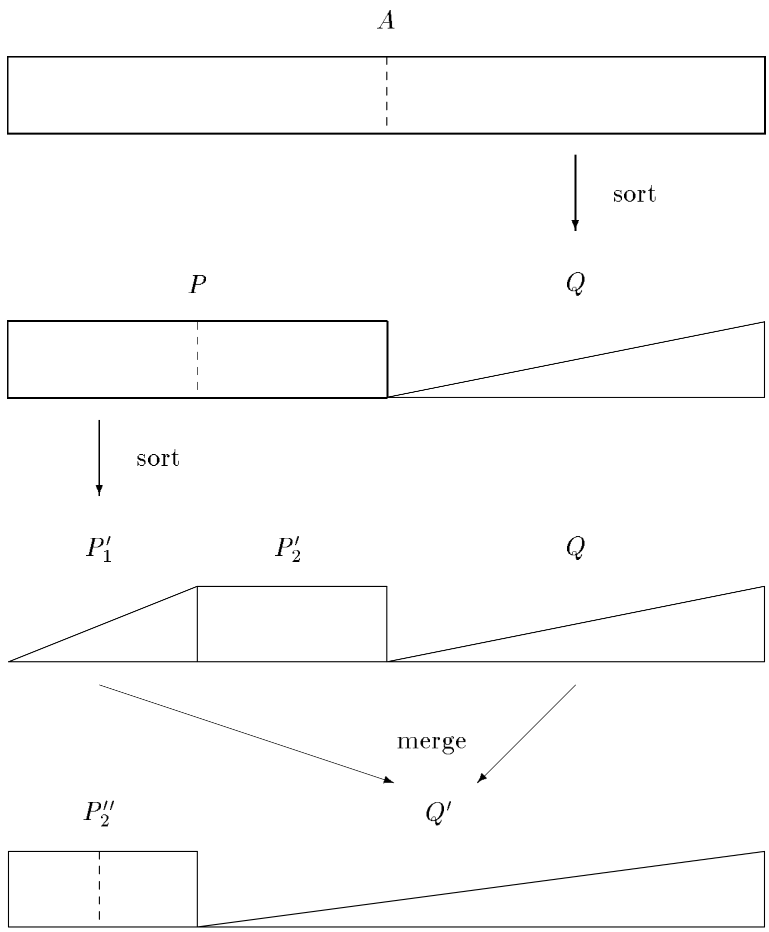


Fig. 1: Illustration of the steps in the basic algorithm. A horizontal rectangle represents an unordered block, and a triangle a sorted block.

parts can be expressed recursively with the following recurrence relations:

$$\begin{aligned} S(m) &= O(m \log_2 m) + S(\lceil m/2 \rceil), \text{ for } m > 1, \\ S(1) &= O(1)\,; \end{aligned}$$

$$\begin{aligned} M(m,n) &= O(m+n) + M(\lceil m/2 \rceil, \lfloor m/2 \rfloor + n), \text{ for } m > 1, \\ M(1,n) &= O(1+n)\,; \end{aligned}$$

It is easy to see that $S(m) = O(m \log_2 m)$ and also $M(\lceil m/2 \rceil, \lfloor m/2 \rfloor) = O(m \log_2 m)$. Since the total running time of the algorithm is proportional to $S(\lceil N/2 \rceil) + M(\lceil N/2 \rceil, \lfloor N/2 \rfloor)$, the algorithm runs in $O(N \log_2 N)$ time.

Next we shall give a detailed description of the subroutines used, in order to analyse the number of comparisons and moves performed. We start with the merge routine, which is needed in the sort part. The basic task is to merge repeatedly two consecutive blocks, X and Y, of about the same size to another block W in the work area. Each time an element e from X or Y is moved to W, the element sitting in W should be moved to the place of e. Usually this is done by one swap, but since a swap requires 3 moves (assignments), we choose another way of implementing the merges. We take the first element of W and store it separately. This creates a *hole* in W. Then the next element e of X or Y is moved to the hole, and a new hole is created by moving the element of W beside the previous hole into the place of e. This is repeated until all the elements of X and Y are moved to W. Finally, the first stored element of W is moved to the hole in X or Y.

Implemented this way, the merging of two blocks of size m and n requires $m+n-1$ comparisons and only $2(m+n)+1$ moves, not $3(m+n)$ as when implemented with swaps. To sort n' elements, $n' \log_2 n' + O(n')$ comparisons and $2n' \log_2 n' + O(n')$ moves are required. In the sort part, sorting is needed for blocks of size $\lfloor \frac{N}{2} \rfloor$, $\lfloor \lceil \frac{N}{2} \rceil / 2 \rfloor$, etc. It is not difficult to see that in the sort part, the number of comparisons is bounded by $N \log_2 N + O(N)$ and the number of moves by $2N \log_2 N + O(N)$.

Actually, we could divide the work a bit differently, so that the elements in the block $\langle \lceil N/3 \rceil + 1, N \rangle$ are mergesorted first, by using the block $\langle 1, \lceil N/3 \rceil \rangle$ as the work area. However, it can be easily verified that this choice has no effect on the constants of the above formulas. Hence, we stick to the more uniform approach of dividing blocks always at half.

In the merge part of our algorithm, the two blocks to be merged are not of the same size. Therefore, it is here better to use the *binary merge* routine, instead of the normal (unary 2-way) one. The binary merge algorithm was described in [8]. The basic idea is as follows. Let X and Y be the blocks to be merged. Let their sizes be m and n, respectively. For the sake of simplicity, assume that $m \le n$. Now let $t = \lfloor \log_2(n/m) \rfloor$ and compare x_1, the first element of X, to y_{2^t}, the 2^tth element of Y. If $x_1 < y_{2^t}$, the proper place of x_1 is searched for by applying binary search. Let us assume that $y_{k-1} \le x_1 < y_k$, $k \le 2^t$. Now the Y-block $\langle 1, k-1 \rangle$ is moved to the output area followed by x_1, and the merging process is repeated for X-block $\langle 2, |X| \rangle$ and Y-block $\langle k, |Y| \rangle$. If $x_1 \ge y_{2^t}$, then Y-block $\langle 1, 2^t \rangle$ is moved to the output area and the merge process is repeated for X-block $\langle 1, |X| \rangle$ and Y-block $\langle 2^t+1, |Y| \rangle$.

Hwang and Lin [8] proved that the binary merge routine merges two blocks of size m and n with $\lceil \log_2 \binom{m+n}{m} \rceil + \min\{m, n\}$ comparisons, which is $O(m \log_2(n/m))$. The number of moves will be $2(m+n)+1$, if implemented as carefully as in the merges of the sort part. Now, in the merge part, the binary merge routine is called at most $\log_2 N + 2$ times. First, the sizes of the merged blocks are $\lfloor \frac{N}{2} \rfloor$ and $\lfloor \lceil \frac{N}{2} \rceil / 2 \rfloor$; second, $\lfloor \frac{N}{2} \rfloor + \lfloor \lceil \frac{N}{2} \rceil / 2 \rfloor$ and $\lfloor \lceil \lceil \frac{N}{2} \rceil / 2 \rceil / 2 \rfloor$, etc. The number of comparisons is then of the order

$$O\left(\frac{N}{4} \log_2 2 + \frac{N}{8} \log_2 6 + \cdots + \frac{N}{2^i} \log_2 (2^i - 2) + \cdots \right) ,$$

which is

$$O\left(N\sum_{i=1}^{\log_2 N} i/2^i\right) = O(N)\,.$$

The number of moves is bounded above by

$$2((N/2+N/4)+(N/2+N/4+N/8)+\cdots)+O(N)\,,$$

which gives $2N\log_2 N + O(N)$.

In the merge part, most work is done when small blocks are merged with the big sorted block. We say that a block is *tiny* if its size is smaller than $\sqrt{N}$, otherwise a block is called *huge*. Both the number of tiny and huge blocks is less than or equal to $\frac{1}{2}\log_2 N + 2$. To save work in the merge part, the huge and tiny blocks can be merged separately. That is, when the first tiny block is encountered, this is no more merged to the merge result of the huge blocks, but another sorted block of size at most $2\sqrt{N}$ is formed by merging all tiny blocks together. Finally, the two sorted blocks (X and Y) are merged together by moving the elements of X into their proper places in the following way. First, find the proper place within Y for the minimum element of X. Then interchange block X with the front part of Y, and repeat the procedure for the next smallest element of X and the rest of Y. (This is analogous to the BLOCK_MERGE_BACKWARDS operation of Trabb Pardo [17].) It is easy to see that this final merge only requires $O(N)$ time, since each element of X is moved over a subblock of Y at most $2\sqrt{N}$ times, and each element of Y is touched only once. This trick reduces the number of moves performed in the merge part of our algorithm to $N\log_2 N + O(N)$.

To sum up, the number of comparisons performed in the whole algorithm is bounded by $N\log_2 N + O(N)$ and the number of moves by $3N\log_2 N + O(N)$.

3. Advanced in-place mergesort

In this section we show how the efficiency of our in-place mergesort algorithm can be improved. Our purpose is first to reduce the number of moves in the sort part of the algorithm and then those performed in the merge part.

To reduce the number of moves required in the sort part, we use K-way merge (K being an arbitrary constant) instead of 2-way merge as done earlier. By doing this, the number of merging levels will come down to $\lceil \log_K N \rceil + O(1)$. Choosing, e.g., $K = 4$ will readily reduce the number of moves by a factor of two. More generally, the number of moves will be $(2/\lceil \log_2 K \rceil) N \log_2 N + O(N)$. However, the number of comparisons can become larger if the K-way merge is not implemented carefully. Next we show how the number of comparisons can be kept almost unchanged.

Let the K blocks to be merged be $X_1, \ldots, X_K$. To decide, which of the blocks contains the smallest element, $K-1$ comparisons are required. This is done by building a *selection tree* (cf. [9]) of depth $\lceil \log_2 K \rceil$ above the K elements. The smallest element is moved to the output area. After this, the selection tree is updated by inserting in the tree the element following the smallest one of the same block. To do this, only $\lceil \log_2 K \rceil$ comparisons are needed. Note that, to avoid element moves, only pointers to elements are stored in the tree.

Since, during the algorithm, only $O(N/K)$ K-way merges are performed, K-merging will cause $O(N)$ extra comparisons. For all elements, except the smallest, $\lceil \log_2 K \rceil$ comparisons are done per element at every merging level. But since the number of levels is now $\lceil \log_K N \rceil + O(1)$, the number of comparisons performed is $N \log_2 N + O(N)$ in total.

Let us next consider, how the number of moves can be reduced in the merge part of the algorithm. Actually, one could say that the merges in this part are done in the worst possible order. If the smallest block could be merged with the second smallest, and the result of this with the third smallest, and so on, the total work required for the merges would be $O(N)$. However, this cannot be done, because we need a work area for our merges.

As observed already in the previous section, most work in the merge part is done when small blocks are merged with the union of previous blocks. Now we will elaborate this idea further on. Let us call a block *small*, if its size is less than $N/\log_2 N$; otherwise call it *big*. In the same way as earlier we sort and merge the big blocks (forming block Y) until the first small block is encountered. Because the size of the work area (including the first small block) is now at most $2N/\log_2 N$, we can sort it by the straightforward in-place mergesort in a linear time giving block X, see Fig. 2(a).

When merging the blocks X and Y we need a work area of size $|X|$. This is formed by searching the $|X|$ smallest elements of X and Y giving blocks x and y (actually we must only ensure that $|X - x| \leq |x| + |y|$), see Fig. 2(b). The final steps of the algorithm are: interchange blocks $X - x$ and x, merge blocks $X - x$ and $Y - y$ and sort block $x + y$ by the straightforward in-place mergesort, see Fig. 2(c,d,e). As we easily see, the final steps of the algorithm need only a linear time. Alternatively, X and Y could have been merged, as proposed by Kronrod [10], by using an in-place merging routine, the use of which we have wanted to avoid.

To summarize, in the merge part of the algorithm we make $O(N)$ comparisons and $2N \log_2 \log_2 N + O(N)$ moves, since the number of merges is $\log_2 \log_2 N + O(1)$. Thus, the computational costs of the algorithm are dominated by those of the sort part, which is shown to require $N \log_2 N + O(N)$ comparisons and $(2/\lceil \log_2 K \rceil) N \log_2 N + O(N) = 2N \log_K N + O(N)$ moves, where $K \geq 2$ is a constant. In other words, the number of comparisons performed by the algorithm is bounded above by $N \log_2 N + O(N)$ and the number of moves by $\varepsilon N \log_2 N$, for any fixed $\varepsilon > 0$, if $N > N(\varepsilon)$ (e.g., choose $K = 16^{1/\varepsilon}$ and let N be big enough).

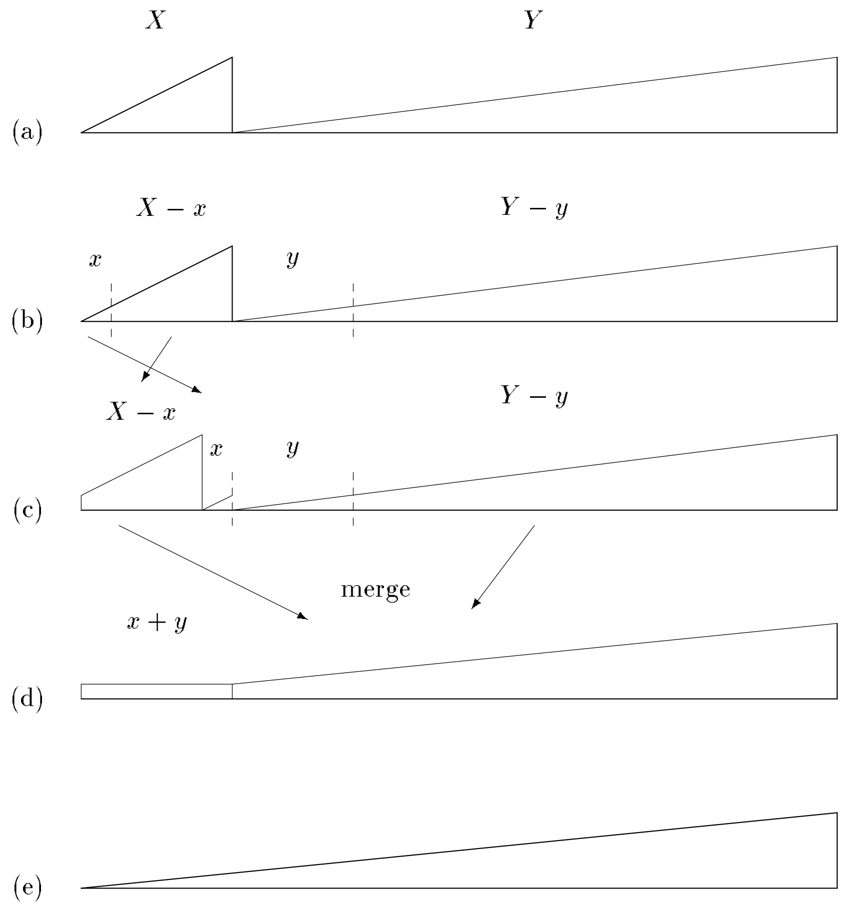


Fig. 2: Illustration of the advanced algorithm. A horizontal rectangle represents an unordered block.

4. Experimental results

To evaluate the practical efficiency of the in-place mergesort algorithm presented in the previous section, we programmed it with the C language. For the sake of comparisons, we programmed also the bottom-up heapsort algorithm [18] and the standard mergesort algorithm using linear extra space [9].

Our very first implementation of the in-place mergesort algorithm used a general k-way merge routine. The program turned out to be unacceptably slow, so we wrote specialized programs for k equal to 2,4, and 8. The program with $k = 4$ behaved best in the following experiments. After some profiling, it was observed that the selection tree was a bottleneck in the in-place 4-mergesort program. Therefore we replaced the selection tree by an array of size 4. The resulting program is denoted in-place 4*-mergesort. Now, to find the smallest of any four elements, three comparisons are required, while only two are necessary if the selection tree is in use. The increase in the number of comparisons can be clearly seen in Fig. 3(a). The theoretical analysis of the 4*-mergesort algorithm is left for the reader. Even if this implementation is theoretically inferior to the implementation of Section 3, it was the fastest in-place mergesort algorithm in our experiments.

The experimental results reported in Fig. 3 and Fig. 4 were carried out on Solbourne S-4000, which has a Sparc processor. The programs were compiled by the Gnu-C compiler with no optimizations. The input elements were pseudo-random integers and the amount of elements in different experiments were 5 000, 10 000, 20 000, 40 000, 60 000, 80 000, and 100 000. The measured quantities in Fig. 3 and Fig. 4 are the average values for 10 and

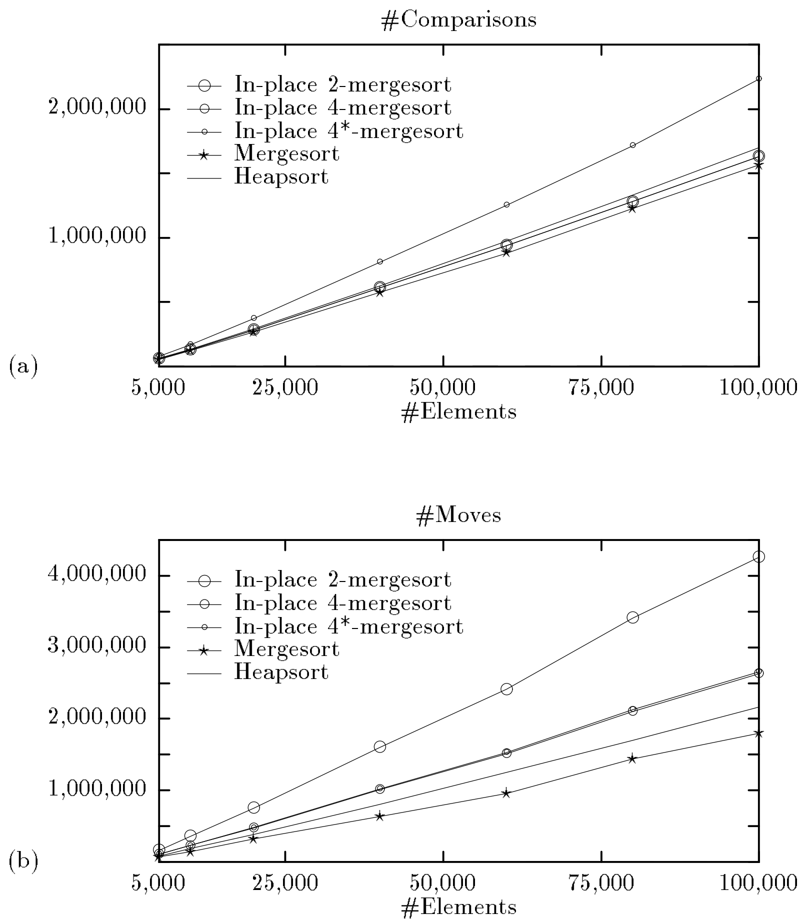
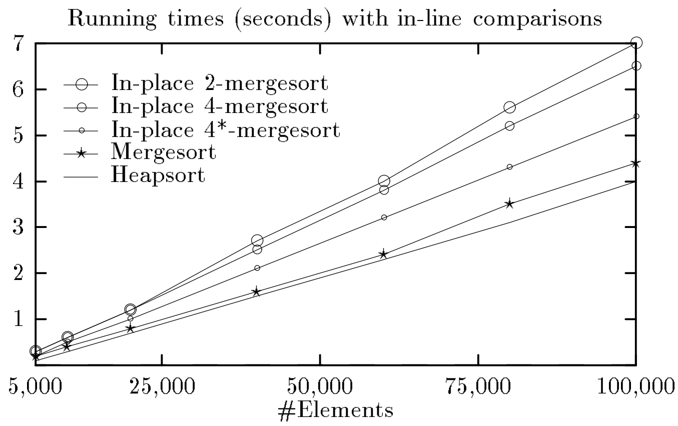


Fig. 3: The number of (a) comparisons and (b) moves performed by various sorting programs for randomly generated data.

50 runs, respectively.

Fig. 3(a) gives the number of comparisons and Fig. 3(b) the number of moves performed by the various sorting programs. We studied the actual running times of the programs in two models: MIX model [9] and `qsort` model [1]. The MIX model reflects the fact that the elements to be sorted are integers, which is known beforehand. Therefore, the cost of comparisons, moves, and index overhead should be about the same. Moves should, however, be more expensive if we were sorting big records. The `qsort` model takes into account the fact that a general sorting program should be polymorphic, as the `qsort` sorting routine in a C library. Since each comparison involves a function call, comparisons are more expensive than moves and in-

(a)



(b)

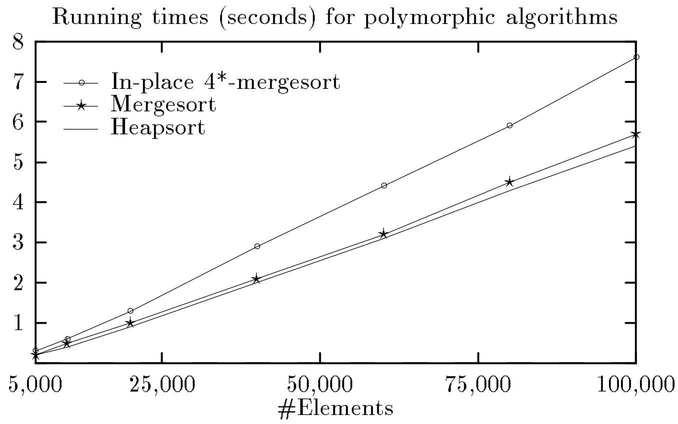


Fig. 4: The running times of various sorting programs for randomly generated data under (a) the MIX model and (b) the `qsort` model.

dex overhead. Fig. 4(a) gives the running times for the programs with in-line comparisons and Fig. 4(b) for the programs with the `qsort` interface.

One should observe that the average case for mergesort is only a little bit better than the worst case while for bottom-up heapsort this is not true; the experiments reflect this because pseudo-random integers were used as input. According to our experiments bottom-up heapsort is still the fastest in-place sorting method guaranteeing the $O(N \log_2 N)$ worst-case performance. In-place mergesort is 25-50 per cent slower than bottom-up heapsort depending on the cost model. We compared also our in-place mergesort implementation to those based on in-place merging, which had been programmed by the second author as part of his M.Sc. project [12]. These earlier mergesort programs were however much slower than those reported in the present work.

5. Conclusions

The performance of the presented in-place algorithms is summarized in Table I. The first algorithm is the basic one where no optimization is done. The second one merges the tiny blocks (smaller than $\sqrt{N}$) and the huge blocks separately. The last one uses K-way merge in sorting as well as merges the small blocks (smaller than $N/\log_2 N$) and the big blocks separately.

TABLE I: Summary of the results.

		comparisons	moves
basic algorithm	sort	$N \log_2 N + O(N)$	$2N \log_2 N + O(N)$
	merge	$O(N)$	$2N \log_2 N + O(N)$
straightforward algorithm	sort	$N \log_2 N + O(N)$	$2N \log_2 N + O(N)$
	merge	$O(N)$	$N \log_2 N + O(N)$
advanced algorithm	sort	$N \log_2 N + O(N)$	$(2/\lceil \log_2 K \rceil) N \log_2 N + O(N)$
	merge	$O(N)$	$2N \log_2 \log_2 N + O(N)$

We have shown that it is relatively simple to device an in-place mergesort, the performance of which is of the order $O(N \log_2 N)$. With various elaborations, also the precise complexity can be brought close to the optimum $N \log_2 N - N \log_2 e + \frac{1}{2} \log_2 N + O(1)$, with respect to the number of comparisons [9]. A natural question is, whether the number of moves can be reduced from $\varepsilon N \log_2 N$, while keeping the number of comparisons the same.

Our starting point was a bit different from the usual approaches: We did not try to solve the in-place merging problem but went directly to solve the sorting problem. Since merging is stable by nature, it is fair to ask, whether it would be possible to develop a powerful stable in-place mergesort algorithm that is more efficient than those based on stable in-place merging.

References

[1] Bentley J.L. and McIlroy M.D., Engineering a sort function, *Software — Practice and Experience* **23** (1993) 1249–1265.

[2] Carlsson S., A variant of HEAPSORT with almost optimal number of comparisons, *Inform. Process. Lett.* **24** (1987) 247–250.

[3] Carlsson S., A note on HEAPSORT, *Comput. J.* **35** (1992) 410–411.

[4] Carlsson S., Katajainen J., and Teuhola J., In-place linear probing sort, *Proc. of the 9th Symposium on Theoretical Aspects of Computer Science*, Lecture Notes in Comput. Sci. **577**, Springer-Verlag (1992), pp. 581–587.

[5] Fleischer R., A tight lower bound for the worst case of Bottom-Up Heapsort, *Proc. of the 2nd International Symposium on Algorithms*, Lecture Notes in Comput. Sci. **557**, Springer-Verlag (1991), pp. 251–262.

[6] Floyd R.W., Treesort 3 (Algorithm 245), *Comm. ACM* **7** (1964) 701.

[7] Huang B.-C. and Langston M.A., Practical in-place merging, *Comm. ACM* **31** (1988) 348–352.

[8] Hwang F.K. and Lin S., A simple algorithm for merging two disjoint linearly ordered sets, *SIAM J. Comput.* **1** (1972) 31–39.

[9] Knuth D.E., *The Art of Computer Programming*, Vol. 3: *Sorting and Searching*, Addison-Wesley, 1973.

[10] Kronrod M.A., Optimal ordering algorithm without operational field, *Soviet Math. Dokl.* **10** (1969) 744–746.

[11] Munro J.I. and Raman V., Sorting with minimum data movement, *J. Algorithms* **13** (1992) 374-393.

[12] Pasanen T., *Lajittelu minimitilassa*, M.Sc. Thesis, Dept. of Computer Science, Univ. of Turku, Turku, Finland, 1993.

[13] Raman V., *Sorting in-place with minimum data movement*, Ph.D. Dissertation, Dept. of Computer Science, Univ. of Waterloo, Waterloo, Canada, 1991.

[14] Reinhardt K., 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, *Proc. of the 3rd International Symposium on Algorithms and Computation*, Lecture Notes in Comput. Sci. **650**, Springer-Verlag (1992), pp. 489–498.

[15] Salowe J. and Steiger W., Simplified stable merging tasks, *J. Algorithms* **8** (1987) 557–571.

[16] Schaffer R. and Sedgewick R., The analysis of Heapsort, *J. Algorithms* **15** (1993) 76–100.

[17] Trabb Pardo L., Stable sorting and merging with optimal space and time bounds, *SIAM J. Comput.* **6** (1977) 351-372.

[18] Wegener I., BOTTOM-UP-HEAPSORT, a new variant of HEAPSORT beating, on an average, QUICKSORT (if n is not very small), *Theoret. Comput. Sci.* **118** (1993) 81–98.

[19] Williams J.W.J., Heapsort (Algorithm 232), *Comm. ACM* **7** (1964) 347–348.