Asymptotically efficient in-place merging ⋆

Viliam Geffert [1]

Department of Computer Science, P. J. Šafárik University, Jesenná 5, SK-04154 Košice, Slovakia
geffert@kosice.upjs.sk

Jyrki Katajainen [2]

Department of Computing, University of Copenhagen, Universitetsparken 1, DK-2100 Copenhagen East, Denmark
jyrki@diku.dk

Tomi Pasanen

Turku Centre for Computer Science, Lemminkäisenkatu 14A, FIN-20520 Turku, Finland
tomi@cs.utu.fi

Abstract

Two linear-time algorithms for *in-place* merging are presented. Both algorithms perform at most $m(t+1)+n/2^t+o(m)$ comparisons, where m and n are the sizes of the input sequences, $m \le n$, and $t = \lfloor \log_2(n/m) \rfloor$. The first algorithm is for *unstable* merging and it carries out no more than $3(n+m)+o(m)$ element moves. The second algorithm is for *stable* merging and it accomplishes at most $5n+12m+o(m)$ moves.

Key words: In-place algorithms, merging, sorting

⋆ A preliminary and weaker version of this work appeared in *Proceedings of the 20th Symposium on Mathematical Foundations of Computer Science*, Lecture Notes in Computer Science **969**, Springer-Verlag, Berlin/Heidelberg (1995), 211–220.

[1] Supported by the Slovak Grant Agency for Science under contract 1/4376/97 (Project "Combinational Structures and Complexity of Algorithms").
[2] Partially supported by the Danish Natural Science Research Council under contracts 9400952 (Project "Computational Algorithmics") and 9701414 (Project "Experimental Algorithmics").

1 Introduction

Let $X = (x_1, x_2, \ldots, x_m)$ be a *sequence* of m *elements*, each consisting of a *key* and some information associated with that key. Assume that the keys are drawn from some totally ordered universe. The total order on the keys induces naturally a total order on the elements: x_i is smaller than or equal to x_j, denoted by $x_i \leq x_j$, if and only if the same relation holds for their keys; and similarly for the relations $<$, $=$, etc. For convenience, we hereafter ignore the difference between the elements and their keys. The sequence X is said to be *sorted* if $x_i \leq x_{i+1}$ for all $i \in \{1, 2, \ldots, m-1\}$. Given two sorted sequences of m and n elements, the classical *merging problem* is to rearrange these to form one sorted sequence of $m + n$ elements. In this merging the primitive operations permitted for the elements are comparisons and moves.

In this paper we study the computational complexity of the *in-place merging problem*, where the input sequences are given in two consecutive subarrays $\mathbf{A}[1\,..\,m]$ and $\mathbf{A}[m{+}1\,..\,m{+}n]$ of some array $\mathbf{A}$, and the output is produced into the same array $\mathbf{A}[1\,..\,m{+}n]$. We can assume that the size of the first sequence is less than or equal to that of the second sequence, i.e., $m \leq n$. For $m > n$, the problem can be solved symmetrically by seeing the last entry of $\mathbf{A}$ as the beginning of the input. In addition to array $\mathbf{A}$, one more storage location is available for storing elements. This is needed, e.g., when exchanging the places of two elements in $\mathbf{A}$. We assume also that only $O(1)$ storage locations are available for storing array indices, counters, etc. The normal arithmetic operations are used in the manipulation of these.

We measure the efficiency of merging algorithms by two quantities: the number of pairwise element comparisons and the number of element moves carried out in the worst case, both expressed as a function of m and n. For the algorithms to be considered, the overhead used for the index manipulations is closely related to these quantities. For example, the standard textbook solution for solving the merging problem requires at most $n + m - 1$ element comparisons and $n{+}m$ element moves, but a separate output array of size $n{+}m$ is needed. It is easy to reduce the size of the extra workspace to m (see [7, Exercise 5.2.4-10] or [8]): we can simply move the shorter sequence into the workspace and then merge back to the original array. Now, however, the number of element moves is $n + 2m$ in the worst case.

If the input sequences are approximately of the same size, $n + m - 1$ is also the best possible bound for the number of comparisons [14]. However, if one of the sequences is considerably shorter, the merging problem can be solved with at most $m \log_2(n/m) + 2m$ comparisons. More precisely, $m(t{+}1) + \lfloor n/2^t \rfloor$ comparisons are sufficient [5, Theorem 1] and $mt + \lfloor n/2^t \rfloor$ comparisons are known to be necessary [5, Theorem 2], where $t = \lfloor \log_2(n/m) \rfloor$. The main goal in the present paper is to develop in-place merging algorithms for which the number of comparisons performed is close to this information-theoretic lower bound.

As to the number of moves performed when solving the in-place merging problem, $n+2m$ is the only non-trivial lower bound known by us. This bound can be proved by considering a particular merging instance where the elements are to be moved according to the following permutation:

$$\begin{pmatrix} 1 & 2 & \cdots & m-1 & m & m+1 & m+2 & \cdots & 2m-1 & 2m & 2m+1 & \cdots & m+n \\ m+1 & m+2 & \cdots & 2m-1 & m+n & 1 & 2 & \cdots & m-1 & m & 2m & \cdots & m+n-1 \end{pmatrix}.$$

Thus every element has to be moved into a new location. Moreover, this permutation has m cycles. By [1, Lemma 2.1], at least $n + 2m$ moves are to be done when permuting the elements in-place.

A merging algorithm is said to be *stable* if it retains the original order of X-elements; that of Y-elements; and, in the case of equal elements, outputs X-elements before Y-elements. Many algorithms for in-place merging has been proposed [2,3,8,9,16]. All these algorithms can be made stable [4,12,13,15], but the resulting algorithms are complicated. The first publication showing that merging is possible in a linear time without a workspace, i.e. in-place, was due to Kronrod [8]. About ten years later, Trabb Pardo [16] presented the first stable in-place merging algorithm. Later Salowe and Steiger [13] observed an easy-to-correct error in the algorithm of Kronrod and made some simplifications to stable merging. According to the analysis of Pasanen [12], the algorithms developed by Huang and Langston [3,4] have the lowest complexity with respect to the number of moves if a linear number of comparisons is approved. The only algorithm performing asymptotically the optimal number of comparisons is that by Mannila and Ukkonen [9], as well as its stable variants introduced by Pasanen [12] and Symvonis [15].

For most in-place merging algorithms presented in the literature, the constant factors in their complexity (that is, the number of comparisons and moves performed) are not analysed properly. The only attempts into this direction, we are aware of, appear in [4,12]. Huang and Langston [3,4] proved that their unstable algorithm guarantees "a worst-case grand total of something less than" $3.5(n+m)+o(n)$ and that their stable algorithm assures "a worst-case key-comparison and record-exchange grand total bounded above by" $7(n+m)+o(n)$. A more careful analysis shows that the unstable algorithm given in [3] performs at most $1.125(n+m)+o(n)$ element comparisons and $2(n+m)+o(n)$ element exchanges. Note, however, that each element exchange requires three moves. Therefore, the number of moves is bounded by $6(n+m)+o(n)$. Katajainen et al. [6] observed that the number of moves can be reduced to $5(n+m)+o(n)$. For the stable algorithm given in [4], the number of comparisons is the same as that required by the unstable algorithm, but the number of exchanges is increased to $5.5(n + m) + o(n)$, which means $16.5(n + m) + o(n)$ moves.

Using the algorithm of Mannila and Ukkonen [9] as our starting point, we show that in-place merging is possible with $m(t+1)+n/2^t+o(m)$ comparisons ($t = \lfloor \log_2(n/m) \rfloor$) and $3(n+m)+o(m)$ moves (Section 3), and that stable in-place merging can also be accomplished with $m(t+1)+n/2^t+o(m)$ comparisons if the number of moves is increased to $5n + 12m + o(m)$ (Section 4). These results are based on the following observations: (1) the block factor $\lceil m^{1/2} \rceil$ used in the original algorithm is not optimal, but a larger factor gives a better performance; (2) the hole technique (cf. Section 2), already in heavy use in [6], can be used to reduce the number of moves.

2 Basic programming techniques

In this section we review the basic techniques used in the development of our in-place merging algorithms. We begin with some general programming techniques and thereafter we introduce those related especially to in-place merging.

Throughout this paper we use the following notation. A *block* W is any collection of consecutive elements in array $\mathbf{A}$. Any subarray of $\mathbf{A}$ occupying some consecutive positions is called a *zone*. We shall denote the number of elements in block W by $|W|$. For two adjacent blocks U and V, UV is the block consisting of all U-elements followed by all V-elements. If $W = UV$ for blocks U, V, and W, then we write $V = U^{-1}W$ and $U = WV^{-1}$.

2.1 Comparisons in merging

The *binary-merge routine* of Hwang and Lin [5] was designed to save some comparisons in merging. We shall now describe how the comparisons are organized. Assume that two sorted sequences X and Y to be merged are of size m and n, respectively, with $m \leq n$. The main idea is to granulate the Y-sequence (implicitly) into blocks of size 2^t, where $t = \lfloor \log_2(n/m) \rfloor$, so that the number of blocks in the Y-sequence is approximately equal to the number of X-elements.

Let X' denote the X-elements and Y' the Y-elements that have not been merged yet. We call the first element x_c of X' the *current* X-*element* and the first element y_c of Y' the *current* Y-*element*. First, we find the leftmost

Y-element $y' \geq x_c$ that is an integer multiple of 2^t positions apart from y_c. This is accomplished by a sequential search, comparing x_c with every 2^tth element of Y'. Second, by a binary search over the $2^t - 1$ elements to the left of y', we determine k, the number of Y'-elements smaller than x_c. Finally, we transport the first k elements of Y' to the output, followed by x_c. Thus, the next X-element to the right becomes the new x_c-element. This is repeated until the X-sequence or Y-sequence is exhausted, after which the rest of the remaining sequence is appended onto the end of the output.

From the description above, it is clear that the number of element comparisons performed is at most $m(t+1) + \lfloor n/2^t \rfloor$. We have used t comparisons for each X-element in the binary search. One more comparison was spent to compare x_c with y'. The remaining comparisons (sequential search) are charged to Y-elements transported to the output, one comparison per at least 2^t elements. Hwang and Lin [5] also proved that $mt + \lfloor n/2^t \rfloor$ is a lower bound for the number of comparisons performed by any merging algorithm, in-place or not.

The situation is not so simple for an in-place algorithm, i.e., if the sorted output should be stored within the same array $\mathbf{A}[1\,..\,m+n]$ as subarrays containing the input sequences X and Y. Therefore, the main problem to be tackled is the organization of element movements. In the subsequent sections, we shall describe some techniques organizing the transportation of elements in the basic algorithm.

2.2 Internal buffering

Most in-place algorithms, like those for in-place merging [2–4,8,9,12,13,15,16], rely on the *partitioning technique* in which the input is partitioned into blocks of the same size. During the computation, elements of some blocks get temporarily out of order; such a block is used as a workspace, called an *internal buffer*. Whenever some element is moved into an internal buffer, the corresponding buffer element is saved in the place of the element just moved.

2.3 Block swapping

In *block swapping*, we are given two (not necessarily adjacent) blocks U and V of the same size and we want to move the U-elements into the place of V-elements and vice versa. It is easy to accomplish this task by exchanging the ith element of U and V for each $i \in \{1, \ldots, |U|\}$. Since each exchange requires three element moves, the total number of moves performed is $3|U|$. Dudziński and Dydek [1] proved that this procedure is optimal.

The number of moves can be reduced to $2|U|$ if we can use the *floating hole* technique to be described next. Assume that there is an empty location e in front of, say, U and we can end up with the empty location at the rear of the same zone. More precisely, we are given eU and V, and we want to replace them by Ve and U, respectively, where the content of e can be destroyed. We simply move the ith element of V into the ith position of the block eU and then the ith element of U into the place released in V, for $i = 1, \ldots, |U|$.

Similarly, the rotation of three blocks Z_0, Z_1, and Z_2 of equal size requires $4|Z_0|$ moves. By the floating hole technique, we can replace eZ_0, Z_1, and Z_2 by Z_1e, Z_2, and Z_0, respectively, using only $3|Z_0|$ moves. This requires, for $i = 1, \ldots, |Z_0|$, to move the ith element of Z_1 into the ith position of eZ_0, then the ith element of Z_2 to the ith position of Z_1, followed by the move of the ith element of Z_0 to the ith position in Z_2. It can be easily seen that this technique works also in the reverse direction, beginning with the hole at the end of one block and ending up in its leftmost position.

2.4 Block interchanging

In *block interchanging* we are given two *adjacent* blocks, U and V, not necessarily of the same size, and we want to produce block VU within the zone originally occupied by UV. Dudziński and Dydek [1] showed how this can be carried out with $|U| + |V| + \gcd(|U|, |V|) \leq |U| + |V| + \min(|U|, |V|)$ element moves, where $\gcd(a, b)$ denotes the greatest common divisor of positive integers a and b. They also proved that the number of moves required cannot be reduced below this amount. This is caused by the fact that each element of UV is rotated $|V|$ positions to the right and such permutation has $\gcd(|U|, |V|)$ cycles.

Sometimes the order of elements in one of the blocks is not important. By using the hole technique, it is relatively easy to see that the block interchanging requires $2|U| + 1$ moves, if the order of U-elements is to be retained, and $2|V| + 1$ moves, if the order of V-elements is to be retained.

2.5 Moves in merging

Let us consider the element movements done in a merge routine relying on internal buffering. Assume that two sorted blocks U and V in the zones A, B, respectively, should be merged together and placed in the zone C of length $|U| + |V|$. The W-elements, originally in C, should be transported to the zones A and B, but the order of W-elements can be mixed up. Here the hole technique can be applied again.

First, the leftmost W-element is put aside to create a hole. Then the smallest U- or V-element is moved into the hole in C and the next W-element to the right of this hole is moved into the place released by the U- or V-element just moved. This is repeated until zone C is full and both the U- and V-elements are exhausted. Finally, the first W-element is put into the hole in A or B. The number of moves required by this procedure is clearly $2(|U| + |V|) + 1$.

2.6 Block swapping and merging in "parallel"

The situation is more complicated in our in-place algorithms, since the U- and V-elements to be merged are moved into a zone C whose elements *should not* be mixed up but rather transported, keeping their original order, to a new zone D of length $|C|$. Now the zone D contains some buffer elements that can be mixed up and moved into the zones A and B, originally occupied by the blocks U and V. A natural idea is to perform these two actions "sequentially", i.e., the swap of the blocks C and D first, the merging second. This would require $4(|U| + |V|) + 2$ moves. However, this number can be reduced by combining the actions into a single "parallel" process.

We start by removing the first element in the *escape zone* D, to create a hole. Then the smallest W-element escapes from zone C to zone D. Now we can move the smallest of the U- or V-elements into the C zone. Finally, the next buffer element of the escape zone D is moved into the place released in A or B, creating a hole in D again. This is repeated until all W-elements escape from C to D and all U- and V-elements are merged together in C. The buffer elements from D are scattered in the zones A and B. It can be easily seen that this "parallel" process reduces the number of moves to $3(|U| + |V|) + 1$.

2.7 Merging by repeated block interchanges

We shall now describe a simple in-place implementation of the binary-merge routine of Section 2.1. This procedure will be used in some special cases. Let X' and Y' denote the X-elements and Y-elements, respectively, that have not been merged yet. To put the first element x_c of X' into its final location, we first determine the length of the prefix Y_1 containing all Y'-elements smaller than x_c. This is accomplished by the binary-merge strategy of Section 2.1. Then X' and Y_1 are interchanged, the next X-element to the right of x_c becomes a new x_c-element, and the block $Y_1^{-1}Y'$ becomes a new Y'-sequence. This is repeated until X' or Y' is exhausted.

Clearly, the number of comparisons performed is given by Section 2.1, i.e., at most $m(t+1)+n/2^t$, where $t = \lfloor \log_2(n/m) \rfloor$. To put x_c into its final location, we perform $|Y_1| + |X'| + \gcd(|Y_1|, |X'|)$ moves, by Section 2.4. Since each Y-element takes part in at most one block interchange and $\gcd(|Y_1|, |X'|) \leq |X'| = m - i + 1$ at the ith iteration, the total number of moves is at most $n + 2 \cdot \sum_{i=1}^{m}(m - i + 1) = n + m^2 + m$. If the sequence X is much shorter than Y, e.g., if $m \leq n^{1/2}$, this gives at most $2n + m$ moves. One should also observe that this procedure merges the sequences stably.

3 Semi-stable in-place merging

Mannila and Ukkonen [9] showed how the binary-merge algorithm can be made in-place. We use their algorithm as a starting point, but in our description some implementation details are different from those in [9]. The changes are done in order to get an asymptotically faster algorithm. The basic technique used is internal buffering where the input sequences X and Y are partitioned into blocks of size s (to be determined later). One should notice that the algorithm is *semi-stable*, i.e., it retains the original order of Y-elements and, in the case of equal elements, outputs X-elements before Y-elements. This will be needed when designing a stable variant of the algorithm in Section 4.

3.1 Initiation

The array $\mathbf{A}[1\,..\,m{+}n]$ with the input sequences X and Y of respective sizes m and n is divided into blocks of equal length s so that a block boundary is placed between the last element of X and the first element of Y. This yields the first X-block of length $s_x = m \bmod s$; similarly, the last Y-block is of length $s_y = n \bmod s$. The exact value of s depends on m. It will be specified so that the number of comparisons and moves is minimized. For the time being, assume only that $m \geq 2s$.

The largest $2s$ elements of the X-sequence are used in internal buffers. The second last X-block containing s buffer elements forms the *free block*, and its start position is stored in a variable f. If, during the computation, there is no free block available, we let $f = 0$. The last X-block forms the *escape block*; if $s_x \neq 0$, the position $e_c = m - s_x + 1$ is the *current escape position*; if $s_x = 0$, $e_c = m - s + 1$. We create a hole here by putting the X-element at this position aside. The first X-element x_c becomes the *current X-element* and the first Y-element y_c the *current Y-element*. The first X-block is the *current X-block* and the first Y-block is the *current Y-block*. The first X-block also becomes the *output block* and $o_c = 1$ pointing to x_c the *current output position*. Observe that $e_c \bmod s = o_c \bmod s$. Clearly, the initial positions for x_c, y_c, o_c, e_c, and f are determined in $O(1)$ time.

Usually, the output block, the current X-block, the current Y-block, the escape block, and the optional free block are all disjoint, and the merging proceeds as described in Section 3.2. However, after the initiation the output block overlays the current X-block and therefore the computation starts in the special mode of Section 3.10.

3.2 Standard mode

We can have the $2s$ buffer elements in four places: to the left of x_c in the current X-block, to the left of y_c in the current Y-block, to the right of the hole e_c in the escape block, and, optionally, we can have an entire free block of s buffer elements, beginning at position f. The sequence Y', consisting of the Y-elements not yet processed, occupies the contiguous Y*-zone* from the position of y_c in the current Y-block to the rightmost position $m + n$. The X- and Y-elements merged already form the contiguous *output zone* in the beginning of $\mathbf{A}$, ending at position $o_c - 1$. Hence, the next element to be output will go to position o_c in the output block. The sequence X', consisting of those X-elements that have not been merged yet, is scattered in the blocks between the output zone and the Y-zone. The permutation of X-blocks is allowed. So the current X-block containing the current X-element x_c, the escape block, and the optional free block can reside anywhere within this X*-zone*. Figure 1 illustrates these concepts.

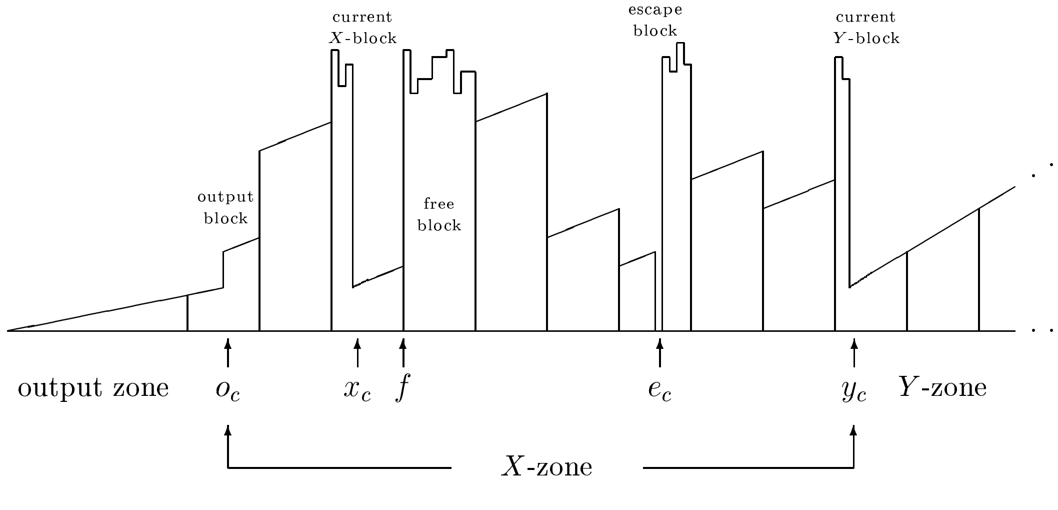

Fig. 1. Illustration of the normal situation.

The output block spans across the current output position o_c so its left part belongs to the output zone while the right part is in the X-zone. As the output grows to the right, the X-elements lying to the right of o_c are moved from the output block to the corresponding positions in the escape block, i.e., to the right of e_c. The positions of o_c and e_c are synchronized, i.e., we have always $o_c \bmod s = e_c \bmod s$. Hence, the relative order of the escaping elements is preserved within the blocks. Moreover, o_c and e_c reach their respective block boundaries at the same time.

Now we are ready for merging. The basic idea is simple; using the current X-element x_c, we first determine the prefix of Y' that contains all Y'-elements smaller than x_c and then we move those to the output zone, followed by x_c. Since the Y-zone is contiguous, we can use the binary-merge routine of Section 2.1. Hence, the number of comparisons is bounded by $m(t+1) + n/2^t$, where $t = \lfloor \log_2(n/m) \rfloor$. The move of X- and Y-elements corresponds to the strategy presented in Section 2.6.

Step A. The X-element at position o_c in the output block escapes to the hole positioned at e_c.
Step B. The smaller of x_c and y_c is moved to its final position o_c.
Step C. A new hole is created at position $e_c + 1$ by moving its buffer element to the place released by x_c or y_c and all necessary index variables are incremented.

This gives three moves per each element transported into its final location. Now there are various special cases that should be detected and handled with care. All exceptions are checked up on after the execution of *Step B*, in the order of their appearance, unless stated otherwise. Most of the exception handling routines skip *Step C*.

3.3 Escape block becomes full

If the rightmost element of the output block is moved to the last position of the escape block, the new hole cannot be created at the position $e_c + 1$ in *Step C*. Instead, a new hole is created at the beginning of the free block that becomes the new escape block. This is accomplished by assigning f to e_c and clearing f. The move of the subsequent buffer element from the new position of e_c to the place released by x_c or y_c does not increase the number of moves; it replaces *Step C* for the last element just output.

There does exist an entire free block available via f, i.e., we have $f \neq 0$. Assume, for example, that the current X-element has just been transported to the output zone. Then we have $r_x \in \{1, \ldots, s\}$ buffer elements in the current X-block (including the hole at the position of x_c), $r_y \in \{0, \ldots, s-1\}$ buffer elements in the current Y-block, since y_c still resides in it, but no buffer elements in the escape zone. Assuming that no free block is available, we have at most $2s - 1$ buffer elements in total. But this is a contradiction, since the number of buffer elements is always equal to $2s$. The same holds if y_c has been transported.

3.4 Current Y-block becomes empty

We check next if the last element of the current Y-block has been transported to the output zone. Then the new y_c element lies at the beginning of the next block to the right. This releases an entire block of buffer elements. Thus, the old Y-block becomes a free block; its position is assigned to f.

We do not have to check if $f = 0$, i.e., we can be sure that f does not already point to another free block. Thus, no buffer elements can be "lost". Suppose on the contrary that we have two blocks of size s containing buffer elements only, namely, the free block and the old Y-block. Since we have only $2s$ buffer elements, including the hole, this may happen only if the escape block is full. But, having done the exception handling of Section 3.3 first, we know that the escape block is not full, which is a contradiction.

3.5 Current X-block becomes empty

Again we dispose the old X-block as free, similarly to the Y-block case. But since the blocks are mixed up in the X-zone, we need to scan sequentially the remaining X-blocks to determine which block should become the new current X-block. The block with the smallest element at its leftmost position is the next to be processed. In the case of equal first elements, the elements at the rightmost block positions are used as the secondary key.

The free block is ignored in this scanning. Moreover, the elements to the left of e_c in the escape block (if it is not empty) together with the elements to the right of o_c in the output block are viewed as a single logical block. In a practical implementation, we can start with the leftmost escape-block element and the rightmost output-block element as a starting key and search the rest of the X-zone for a block with a smaller key. If the logical block composed of the left part of the escape block and the right part of the output block should be processed next, the program control will be switched to the mode described in Section 3.6.

If the escape block is empty, then both e_c and o_c point to the beginning of their respective blocks. Then the escape block is skipped and the output block is handled as an ordinary X-block, so we may even find out that the new X-block should be located at the same position as the output block. This special mode is explained in Section 3.10.

It is easy to see that looking through the X-blocks consumes $O((m/s)^2)$ additional comparisons: there are $O(m/s)$ blocks in the X-zone and such search is activated only if the current X-block is exhausted, i.e., at most $O(m/s)$ times.

3.6 Current X-block overlays escape block

If the X-block that should be processed next is the escape block with its logical right part still placed in the output block, then both the current X-block and the escape block are located within the same physical block. Here we have always x_c positioned to the left of e_c and the buffer elements are both to the left of x_c and to the right of e_c.

Once the position of x_c is properly initiated, all actions are performed in the standard way of Section 3.2. That is, the elements are removed from the x_c- or y_c-positions and appended at the position of e_c. Since e_c moves to the right "faster" than x_c, this special case returns automatically to the standard mode as soon as e_c reaches a block boundary. Then the escape block separates from the current X-block as described in Section 3.3.

3.7 Output block overlays escape block

Next we check if the output zone, crossing a block boundary, does not bump into any "special" block within the X-zone. It is easy to show that this may happen only if e_c points to the beginning of the escape block that is empty, using the fact that the positions of o_c and e_c are synchronized and that the special handling of Section 3.3 is performed first.

Now consider that the output block overlays the escape block, i.e., they are both located within the same physical block. In this mode, we have always $o_c = e_c$. The element movement corresponds now to the more efficient scheme of Section 2.5 (cf. Section 3.2).

Step B'. The smaller of x_c and y_c is moved to the hole positioned at $o_c = e_c$.
Step C'. A new hole is created at the position $o_c + 1 = e_c + 1$ by moving its buffer element to the place released by x_c or y_c.

Step A is eliminated since $o_c = e_c$. This mode is terminated as soon as o_c and e_c reach a block boundary. Then the escape zone may jump into another block by Section 3.3. We need also a new (slightly modified) version of the routine described in Section 3.5. When the current X-block becomes empty, it becomes free as usual, but the combined output/escape block is skipped out while searching for the next X-block.

3.8 Output block overlays free block

This may happen only if all index variables reach block boundaries at the same time. For example, the escape block is full and hence, by Section 3.3, the free block becomes first a new escape block. Then, by Section 3.5, the empty X-block may be disposed as a new free block. If this block is next to the right of the old output block, the new output and free blocks are overlaid.

In any case, by the same argument as in Section 3.7, we have that e_c points to the beginning of an empty escape block. Therefore, we can easily swap the free block with the escape block by swapping the pointers stored in f and e_c, since both these blocks contain the buffer elements only. Second, one move suffices to transport the hole from one block to another. Note that this additional move is for free, we actually save some moves; the next s transports to the output zone will require only $2s$ moves, instead of $3s$ as in the standard case. Thus, the program control is switched to the mode described in Section 3.7.

3.9 Output block overlays current X-block

Again, by the argument of Section 3.7, the output position o_c can be located at the beginning of the current X-block only if e_c points to the beginning of an empty escape block. There are now two cases to consider.

(A) If x_c is the leftmost element of the current X-block, we have exactly s buffer elements in the remaining "special" blocks, i.e., in the current Y-block and/or the optional free block. Suppose that $f = 0$, i.e., there is no free block available. Then the current Y-block must be empty, containing exactly s buffer elements. But this is a contradiction, since the exception handling of Section 3.4 has been performed earlier. Therefore, we have two empty blocks, i.e., the escape and free blocks. The program control is switched immediately to the special mode to be described in Section 3.10.

(B) If x_c is not the leftmost element of the X-block, the total number of buffer elements in the X-block and the escape block is strictly above s, thus, no free block is available. Hence, we can dispose the empty escape block as free, update f, and create a hole at the position $e_c = o_c$ by moving a single buffer element from o_c to the position of the updated f. This additional move is for free, not increasing the total number of moves, because we can charge it as (nonexistent) *Step A* for the next element that will be transported to the output zone. Since x_c is not placed at the beginning of the block, we can guarantee that at least one transport to the output will use only two moves in the next future.

This special mode may be viewed as if *three blocks* were overlaid, namely, the output, escape, and current X-blocks. The buffer elements are between the hole at $e_c = o_c$ and the current X-element x_c. The elements are moved according to *Step* B' and *Step* C' of Section 3.7. However, there is a different exception handling here.

(1) If the rightmost X-element of this combined block has been transported to the output zone, then the X-block separates from the output/escape block, since we search for the next X-block to be processed. But here, unlike in Section 3.5, the combined output/escape block is not disposed as free, moreover, it is skipped out during the search. The program control is switched to the mode of Section 3.7; the output and escape blocks are overlaid still.

(2) Now consider that this combined block becomes full, i.e., the output position o_c may "bump" into x_c. This may happen if and only if the last element of the current Y-block is transported to the output zone. We have disposed one block as free at the very beginning of this mode, therefore, the total number of buffer elements in the Y-block and the combined output/escape/X-block must be equal to s. We do not dispose the empty Y-block as free (we already have one) but rather it becomes a new escape block. The hole, located at the position of the "old" y_c in the "new" escape block, jumps within the same block to a new position e_c so that $e_c \bmod s = o_c \bmod s$. This move replaces *Step* C' for the last element just merged. Hence, it does not increase the total number of moves. Then we follow the instructions of Section 3.10.

3.10 Output zone bumps into the current X-element

The program control can jump to this special mode from several different places (Sections 3.1, 3.5, 3.9(A), or 3.9(B.2)). In any case, we have two empty blocks containing all buffer elements (including the hole), namely, the free and escape blocks. The output and X-blocks are overlaid, moreover, there is no room in between, i.e., the output position o_c is pointing to the current X-element x_c. The position of the hole in the escape block is synchronized with o_c, i.e., we have $e_c \bmod s = o_c \bmod s$.

As long as X-elements are smaller than or equal to the current Y-element, they can be transported to the output zone. This needs no actual transportation, just the positions of o_c and x_c are moved synchronously to the right. To keep e_c synchronized with o_c, we move the hole along the escape block in parallel, which gives one move per element. There are two ways out of this loop.

(1) If o_c and x_c reach the block boundary, we simply search for the next X-block to be processed; the current configuration is the same as if, in

the standard mode, o_c, e, x_c, and y_c reached the block boundaries at the same time (with the old X-block disposed as free, by Section 3.5). Thus, unless something "exceptional" happens, the program control returns to the standard mode. (The possible exceptions are those discussed in Sections 3.7–3.9, and 3.12.) The single move required to place the hole at the beginning of the escape block is for free, it substitutes *Step C* for the last element merged.

(2) If the current Y-element y_c must be transported to the output zone, some rearrangement is necessary. Recall that the hole position e_c of the escape block is synchronized with o_c pointing to x_c, i.e., we have $e_c \bmod s = o_c \bmod s$. First, the current X-element x_c is moved from position o_c to position e_c. Now we can transport y_c to the output position o_c. Finally, a new hole is created[3] at the position $e_c + 1$, moving its buffer element to the place released by y_c.

The result is that the current X-block, overlaid by the output block, jumps and overlays the escape block. Thus, the control is switched to the mode of Section 3.6. Recall that here we have a "queue"; the elements are removed from the x_c-position and appended at the position of e_c, within the same block. Now the "queue" of Section 3.6 contains a single element only, namely, the current X-element.

Clearly, this rearrangement needs only three moves. Since one more element has been transported to the output zone, the number of moves is the same as in the standard case.

3.11 Y-elements are exhausted

If the last Y-element has been transported to the output zone, the remaining X-elements are appended in the ascending order. The movement of X-elements is exactly the same as if y_c had pointed to $+\infty$, a dummy element larger than any other element, placed as the $(n+1)$st Y-element at the very end of the Y-zone. However, we do not compare the X-elements with this dummy Y-element. In particular, the last Y-block of length $s_y \leq s$ is merged in a uniform way.

[3] Unless the position $e_c + 1$ itself is across the block boundary. If x_c is moved to the rightmost position of the escape block, the escape block jumps immediately and the free block becomes a new escape block. This nested exception thus returns the program control to the standard mode; all "special" blocks now reside in pairwise disjoint regions, with $f = 0$. However, the output zone has just crossed a block boundary, and therefore we jump to the point where the standard routine checks the exceptions of Sections 3.7–3.12.

Finally, we have to check whether the last "non-buffer" X-element has been transported to the output zone. Now the array $\mathbf{A}$ is of the form $Z_1B_1Y_1$, where Z_1 denotes the merged output with the largest non-buffer X-element at the end, B_1 represents the largest $2s$ X-elements used in the internal buffers, and Y_1 is the sequence of Y-elements not yet processed.

Fix $s = \lceil c \cdot m^{2/3}/(\log_2 m)^{1/3} \rceil$, where c is a constant subject to further optimization. Now, using in-place mergesort (see, e.g., [6]), we sort the buffer B_1 and then merge B_1 with Y_1. That is, we create a new buffer, consisting of the largest $2s_2$ elements of B_1, with $s_2 = \lceil s^{1/2} \rceil$, and then rerun the entire algorithm, starting from the leftmost position of B_1. After this $\mathbf{A}$ is of the form $Z_1Z_2B_2Y_2$, where B_2 represents the largest $2s_2$ elements of B_1 used as the new buffer and Y_2 is again a sequence of Y-elements not merged yet. The sequence B_2 is sorted and then merged with Y_2, by using the block interchanging procedure of Section 2.7, consuming $|Y_2| + O(|B_2|^2) = |Y_2| + O(m^{2/3}/(\log_2 m)^{1/3})$ moves.

To sum up, the number of comparisons and moves performed in the main part of the algorithm is bounded by $m(t+1) + n/2^t$ (where $t = \lfloor \log_2(n/m) \rfloor$) and $3(n+m)$, respectively, since we need three moves per each element transported to the output and $3|Z_1|+3|Z_2|+|Y_2| \leq 3(n+m)$. Both the number of additional comparisons and moves (the selections of the next smallest X-block in both phases, sorting the buffers B_1 and B_2, and $O(|B_2|^2)$ factor) are bounded by $O((m/s)^2 + (s/s_2)^2 + s\log_2 s + s_2\log_2 s_2 + {s_2}^2) = O((m/s)^2 + s\log_2 s) = O((m\log_2 m)^{2/3})$. The number of auxiliary operations (e.g., arithmetic for the index manipulation) is bounded by $O(n+m)$.

As pointed out in Section 3.1, the algorithm does not work if $m < 2s$. By an easy calculation we get that $m \geq 2s$ for each $m \geq 6$, provided that the block size is fixed to $s = \lceil m^{2/3}/(\log_2 m)^{1/3} \rceil$. If, for example, we claim additionally that at most one half of the X-elements may be used in internal buffering, i.e. $m \geq 4s$, we get the restriction $m \geq 20$. For $m < 20$, merging can be carried out efficiently by using the algorithm described in Section 2.7, which gives at most $n + O(1)$ moves. (The actual minimal value of m depends on the chosen constant c, mentioned above.) The $O(1)$ factor can be further optimized since we can remember the positions of all X-elements in index variables. We leave the details to the reader. Hence, we have proved the following:

Theorem 1 *Two sorted sequences of size m and n ($m \leq n$) can be merged in-place in $O(n+m)$ time by performing $m(t+1) + n/2^t + O((m\log_2 m)^{2/3})$ comparisons ($t = \lfloor \log_2(n/m) \rfloor$) and $3(n+m) + O((m\log_2 m)^{2/3})$ moves in the worst case.*

4 Stable in-place merging

In the algorithm presented in Section 3, the stability can be lost in two places. First, the order of buffer elements may change. Since some buffer elements may be equal, their original order cannot be recovered. Second, the order of X-blocks may change. Since all elements in some blocks might be equal, the order of these homogeneous blocks cannot be recovered. Pasanen [12] and Symvonis [15] have shown how these problems can be resolved in the algorithm by Mannila and Ukkonen [9]. The basic technique used by us is similar to that used by Pasanen. However, our goal is again the asymptotic efficiency of the algorithm. Our purpose in this section is to prove:

Theorem 2 *In the worst case, two sorted sequences of size m and n ($m \leq n$) can be stably merged in-place in $O(n+m)$ time by performing $m(t+1)+n/2^t+O(m^{6/7})$ comparisons ($t = \lfloor \log_2(n/m) \rfloor$) and $5n + 12m + O(m^{6/7})$ moves.*

If $m \leq n^{1/2}$, the merging can be carried out as described in Section 2.7. Hence, assume that $m > n^{1/2}$. Using the block size $s = \lceil m^{4/7} \rceil$, we shall have $O(\log_2 s) = O(\log_2 m) = O(\log_2 n) = O(\log_2(n+m))$ and $m/s \leq s$.

4.1 Selection of buffer elements

To avoid the problem with equal buffer elements, we pick up from the X-sequence only distinct elements. This idea was already used in the stable merging algorithm by Trabb Parbo [16]. Now the computation starts by counting whether ℓ, the number of distinct elements in X, is larger than or equal to $r = \lceil m/s \rceil + 2s$. This is easily carried out in $O(r \log_2 m) = O(s \log_2 s)$ time by performing r binary searches.

If there are enough distinct elements, i.e. $\ell \geq r$, two buffers are created. The first *little buffer* contains $\lceil m/s \rceil$ elements and is placed in front of array $\mathbf{A}$. The second *merging buffer* should have $2s$ elements, which are gathered behind all the other X-elements just in front of the Y-sequence. If $\ell < r$, only one buffer of size ℓ is created in the beginning of array $\mathbf{A}$.

A little buffer B_1 of size $k_1 \in \{\ell, \lceil m/s \rceil\}$ is created in front of the X-sequence as follows. Assume that $X = X_1X_2$, where X_1 denotes the prefix containing precisely k_1 distinct elements. The X_1-elements are maintained in order HB_1T, where H denotes the block of X_1-elements not yet processed and T the block of X_1-elements processed so far. We begin with $H = X_1$ and empty B_1T. Let x_{R} be the last element in H. Using binary search, we determine the position of x_{L}, the first element in H that is equal to x_{R}. Hence, we have $H = H'x_{\mathrm{L}}E$, where E denotes the block of X-elements all equal to x_{L}. Now the blocks E

and B_1 are interchanged; the order of B_1-elements may be mixed up. Then $X_1 = H' x_{\mathrm{L}} B_1' ET$, where H', $x_{\mathrm{L}} B_1'$, and ET represent the new values for H, B_1, and T, respectively. This is repeated until block H becomes empty. At last, we sort the buffer by in-place mergesort. Another buffer of size $k_2 = 2s$ can be created at the rear of the X-sequence in a similar manner, processing a suffix of X.

When locating the buffer elements, $O(k \log_2 m)$ comparisons are performed, where $k = k_1 + k_2$. Since the buffer elements can be mixed up, the number of moves transporting the first buffer forward is $\sum_{i=1}^{k_1}(2h_i + 1) = 2|X_1| - k_1$, where h_i denotes the size of block E at the ith block interchange. In sorting, $O(k_1 \log_2 k_1)$ comparisons and moves are done. Similarly, $2|X_2| - k_2$ moves are sufficient for the second buffer. Thus, the number of comparisons and moves performed during the creation of the buffers is bounded by $O(k \log_2 m) = O(s \log_2 s)$ and $2m + O(k \log_2 k) = 2m + O(s \log_2 s)$, respectively.

4.2 Buffer embedding

After merging, the configuration of array $\mathbf{A}$ will be $B_1 Z B_2$, in which B_1 is the little buffer of size $k_1 \in \{\ell, \lceil m/s \rceil\}$, Z a sorted sequence of length at most $n + m$, and B_2 is the optional merging buffer of size $k_2 = 2s$.

Let us first embed buffer B_2 into Z. This is done by the *two-level block-interchanging technique* described below. First, B_2 is divided into subblocks of size $\lceil s^{1/2} \rceil$. Using a slightly modified routine of Section 2.7, we merge these subblocks with Z so that the first element of each subblock falls in its final location. The merging proceeds from right to left, handling the subblocks as some unbreakable large "elements". Second, each time the current subblock reaches its final location, the first-level merging is temporarily interrupted by a call of the *second-level* routine of Section 2.7, that merges the current subblock with elements lying on its right.

By an analysis similar to that in Section 2.7, we get that the first-level routine performs at most $O(s^{1/2} \log_2 s)$ comparisons and $(n+m-h) + O(s^{3/2})$ moves, where h is the final position for the first element of B_2. We use the fact that $O(\log_2(n+m)) = O(\log_2 s)$ and that the ith subblock of B_2 takes part in at most $2s^{1/2} - i + 1$ block interchanges. The second level represents $2s^{1/2}$ calls of the procedure described in Section 2.7, performing in total $2s^{1/2} \cdot O(s^{1/2} \log_2 s)$ comparisons and $\sum_{i=1}^{2s^{1/2}} (d_i + \lceil s^{1/2} \rceil^2 + \lceil s^{1/2} \rceil)$ moves, where d_i are the distances between the subblocks after the first-level merging, $\sum_{i=1}^{2s^{1/2}} d_i = n + m - h$. Therefore, we have $O(s \log_2 s)$ comparisons and $2n + 2m - 2h + O(s^{3/2})$ moves for buffer B_2.

The first buffer B_1 of size $k_1 = \lceil m/s \rceil$ is so small that it can be embedded by a *single-level* procedure of Section 2.7, using $O((m/s)\log_2 s)$ comparisons and $h + O((m/s)^2)$ moves. Summing up, the embedding of both buffers requires $O(s \log_2 s)$ comparisons, since $m/s \leq s$, and at most $2n+2m+O((m/s)^2+s^{3/2})$ moves.

The situation is different if $\ell < \lceil m/s \rceil + 2s$, i.e., if there are not enough distinct elements in the X-sequence. Then we have only buffer B_1 of size $k_1 = \ell$. If $\ell > m^{1/2}$, B_1 is merged by the two-level block-interchanging procedure used for B_2, which gives again $O(s \log_2 s)$ comparisons and $2n+2m+O(s^{3/2})$ moves, since $\ell < \lceil m/s \rceil + 2s \leq O(s)$. However, if $\ell \leq m^{1/2}$, B_1 will be merged by the single-level procedure of Section 2.7, which reduces the number of moves to $n + m + \ell^2 + \ell \leq 2n + m + O(s)$, using $\ell^2 \leq m \leq n$ and $\ell \leq O(s)$.

4.3 Standard mode

If the X-sequence contains at least $\lceil m/s \rceil + 2s$ distinct elements, the merge of X and Y proceeds as follows. First, the little buffer B_1 and the merging buffer B_2 are created as described in Section 4.1. After this the configuration of array $\mathbf{A}$ is of the form $B_1 X' B_2 Y$, where X' denotes the remaining X-elements. The sequences $X'B_2$ and Y' are divided into blocks of length s as described in Section 3.1. Observe that the buffer elements of the escape and free blocks are all distinct, since these blocks are formed from B_2. To remember the original order of homogeneous X'-blocks, we swap the leftmost element in the ith X'-block with the ith element of the little buffer, for each but the first X'-block (which is the current X-block). Now we can merge X' and Y by the algorithm of Section 3, using the following modifications:

(1) The search for the next smallest X-block is simplified (cf. Section 3.5), since all X-blocks, except the current one, contain distinct elements in their leftmost positions. As the next smallest X-block is found, the exchanged elements are swapped back (for this particular block only). To determine a proper swapping position in the little buffer, we need an additional counter for the number of X-blocks processed so far. This trick returns the swapped X'-elements back to their respective positions *before* they are compared with any Y-elements. Moreover, all elements of the little buffer are safely back in B_1 when the X-sequence is exhausted. This requires only $O(1)$ additional moves per each X-block processed, i.e., $O(m/s)$ moves in total.
(2) The merging goes ahead until *both* sequences are exhausted, i.e., the buffer elements of B_2 end up at the very end of array $\mathbf{A}$ (cf. Section 3.12). Here we restore the original order of B_2, using in-place mergesort. The buffer embedding is done in a different way, as described in Section 4.2.

Let us now analyse the computational costs of the above procedure. The swapping of the B_1-elements with the X'-elements will require $O(m/s)$ moves. Adding this to the resource requirements of Section 3.12, we have $m(t+1) + n/2^t + O((m/s)^2 + s\log_2 s)$ comparisons ($t = \lfloor \log_2(n/m) \rfloor$) and $3(n+m) + O(s\log_2 s + m/s)$ moves, plus the additional administration needed for the creation and destruction of the buffers. By Section 4.1, the initial buffer creation costs $O(s\log_2 s)$ comparisons and $2m + O(s\log_2 s)$ moves, while the final buffer embedding, by Section 4.2, $O(s\log_2 s)$ comparisons and $2(n+m) + O((m/s)^2 + s^{3/2})$ moves. To sum up, if there are enough distinct X-elements to create both buffers, $m(t+1) + n/2^t + O((m/s)^2 + s\log_2 s) \leq m(t+1) + n/2^t + O(m^{6/7})$ comparisons and $5n + 7m + O((m/s)^2 + s^{3/2}) \leq 5n + 7m + O(m^{6/7})$ moves are sufficient, for $s = \lceil m^{4/7} \rceil$.

4.4 X-sequence has few distinct elements

If ℓ, the number of distinct X-elements, is less than $\lceil m/s \rceil + 2s$, we do not have enough distinct elements to create the buffers needed in the standard mode. Still, we can create the little buffer B_1 of size ℓ, by the routine of Section 4.1. Hence, we can keep track of the relative order of ℓ blocks, by swapping the leftmost elements in these blocks with the corresponding elements in B_1, as described in Section 4.3. Dividing the sequences X' and Y into blocks of size $q = \lceil m/\ell \rceil$ will allow us to control disordered blocks in an area of about m elements. In the first stage, we merge these blocks of size q, using a buffer of ℓ blocks. In this merging a Y-block is considered to be smaller than an X-block if and only if the *last* element of the Y-block is *strictly smaller* than the *first* element of the X-block. The technical details of the block merging are postponed to Section 4.5.

The block merging is stable; it does not mix up homogeneous X- or Y-blocks. It also preserves the relative order of elements inside the blocks. Hence, after block merging there does not exist a pair of elements $y \geq x$ so that Y-element y is placed (any number of positions) to the left of X-element x. However, we may still find some inverted pairs of type $x > y$. More precisely, such elements may be found in places where an X-block is followed by a Y-block. That is, some X-elements should be moved forward and some Y-elements backward (see Figure 2). In the second stage, such inverted elements are transported into their final locations; the method for this local merging is described in Section 4.6. Finally, the buffer elements of B_1 are embedded, by the technique of Section 4.2.

Recall that the buffer of size ℓ is initially sorted and placed in the beginning of array $\mathbf{A}$. The sequences X' and Y are divided into blocks of size $q = \lceil m/\ell \rceil$, except for the last X'-block and the first Y-block of size 1 (one element), and the first X'-block and the last Y-block of size $(|X'|–1) \bmod q$ and $(n–1) \bmod q$, respectively. For block merging, the key of each X-block is its *first* element and the key of each Y-block is its *last* element. We can think of the first X'-block X_L as if it had begun with $-\infty$, i.e., with a dummy element smaller than any other element, placed in front of the first X'-element. Similarly, the last Y-block Y_R is viewed as if it had ended by $+\infty$, a dummy element larger than any other element. Thus, the first and last short blocks are already in their final locations and we shall only merge *full* blocks of X'' and Y'', where $X'' = X_{\mathrm{L}}{}^{-1}X'$ and $Y'' = YY_{\mathrm{R}}{}^{-1}$. Scanning the sequences X'' and Y'' with two cursors, we determine the smallest ℓ blocks, i.e., we find two boundaries such that $X'' = X_1X_2$ and $Y'' = Y_1Y_2$, where X_1 contains a blocks, Y_1 b blocks, and $a+b = \ell$. (Do not count the first element of Y'', representing a "block" by itself.) This needs only ℓ comparisons. To remember the original order of the homogeneous blocks in Y_1 and to guarantee the stability of block merging, we swap the leftmost elements in Y_1-blocks with the corresponding elements of buffer B_1, as done in Section 4.3.

After this initiation, we merge X'' and Y'' in two phases. In the first phase, we merge X_2 with Y_2 and place the result in the zone originally occupied by Y_1Y_2. The Y_1-blocks are transported to the zone released by X_2-blocks. We start by putting the first Y_1-element aside, to create a hole. During the computation, the configuration of the zone originally occupied by $X''Y''$ is of the form $X_1HX_2'ZeRY_2'$, where X_2' and Y_2' denote the X_2- and Y_2-blocks not merged yet, Z represents the blocks merged already, and e is the hole. The Y_1-blocks, used in internal buffering, are scattered in front of X_2' and Y_2', i.e., in H and R. While the blocks of R get mixed up quite arbitrarily, the blocks of H are organized in a data structure similar to a d-heap, where $d = \lceil \ell^{1/2} \rceil$, used and analysed, e.g., in [10]. That is, the blocks in H are divided into at most $\ell^{1/2}$ *superblocks*, each consisting of $\lceil \ell^{1/2} \rceil$ blocks. The largest block in each superblock is stored in its rightmost position, called the *top of a superblock*. The rightmost superblock need not be complete; it can be shorter, having no top. We do not care about the order of the blocks in this incomplete superblock. Since in the second phase the heap structure will move to the left, the leftmost superblock can also be incomplete but it has a top as all complete superblocks. Note, especially, that the positions of the tops are fixed even though the data structure moves.

The moves of blocks are organized as follows. Let Z_0 be the leftmost block in R and Z_1 be the smallest block of X_2' and Y_2' not merged yet. If Z_1 is the first block of Y_2', or the first block of X_2', but not placed at the superblock boundary of H, we are done by swapping Z_1 and Z_0. The hole e floats one block to the right, therefore, this can be done with $2q$ moves, by Section 2.3. If Z_1 is the first block of X_2' that is in a superblock top position, we search sequentially for Z_2, the largest Y_1-block in the rightmost (incomplete) superblock of H. This requires at most $O(\ell^{1/2})$ comparisons. If Z_2 is smaller than Z_0, we swap Z_1 and Z_0. Otherwise, Z_2, Z_1, and Z_0 are to be rotated, we move Z_j ($j \in \{0, 1, 2\}$) into the place of $Z_{(j-1) \bmod 3}$. This is performed in order to organize the worn-out Y_1-blocks of R into a heap-like structure H, i.e., to have the largest block in the top position for each superblock. By Section 2.3, this rotation requires $3q$ moves, since the hole floats to the right. This is repeated until the X_2' sequence is exhausted, except for the rightmost block. Recall that this block contains the largest X_2-element only. Since the floating hole is always placed between the merged output Z and Y_2', we shift all smaller Y_2'-elements one position to the left and insert the largest X_2-element, using binary search. This completes the merging of X_2 and Y_2; the hole is now in front of the merged output.

In the second phase, we merge X_1 with Y_1-blocks stored in H. The configuration is of the form $X_1'HeZ$, where X_1' denotes the X_1-blocks not merged yet, Z is the merged output, and e is the hole. Now we proceed from right to left. Let Z_0 be the rightmost block in H and Z_1 the largest block not merged yet. To locate Z_1 we scan along the tops of the superblocks in H, along the blocks in the rightmost incomplete superblock, and compare the maximum with the rightmost X_1'-block, using $O(\ell^{1/2})$ comparisons. If Z_1 is a Y_1-block, we swap its leftmost element with the corresponding element in little buffer B_1, to restore their original contents (cf. Section 4.3). Then, if $Z_1 = Z_0$, we shift this block one position to the right, to let the hole float to the left. If Z_1 lies in the rightmost superblock, we swap Z_1 and Z_0. Otherwise we search for Z_2, the second largest block inside the superblock containing Z_1 by using $O(\ell^{1/2})$ comparisons. If Z_2 is smaller than Z_0, we swap Z_1 and Z_0, otherwise we have to move Z_j ($j \in \{0, 1, 2\}$) into the place of $Z_{(j-1) \bmod 3}$. Since the hole floats to the left, the block transportation can be done with $2q$ or $3q$ moves, respectively, by Section 2.3. If Z_1 is an X_1'-block, we shall move it into the place of Z_0, extending the structure of H more to the left. That is, if Z_1 is in a superblock top position, we swap Z_1 and Z_0. Otherwise, let Z_2 be the largest block in the leftmost (shorter) superblock, i.e., Z_2 is the leftmost block in the top position. If Z_2 is larger than Z_0, we swap Z_1 and Z_0, otherwise we move Z_j into the place of $Z_{(j-1) \bmod 3}$. This is repeated until the structure H is exhausted, i.e., all Y_1-blocks are merged. Recall that the hole was created by putting the smallest Y_1-element aside. Now the floating hole is placed between X_1' and the merged output, so we shift all larger X_1'-elements one position to the right and insert the smallest Y_1-element, using binary search.

It is easy to see that fixing the size of X_1 and Y_1 requires only ℓ comparisons. When swapping the B_1-elements with the leftmost elements in Y_1-blocks, required to remember the original permutation of Y_1, $O(\ell)$ moves are carried out. The merging of X_2 and Y_2 requires two moves per each merged element, i.e., at most $2n$ moves, plus $\ell^{1/2}q \leq m/\ell^{1/2} + \ell^{1/2}$ moves spent to organize the large Y_1-blocks in the superblock top positions of H. Building H requires also $O(\ell)$ comparisons; this represents $O(\ell^{1/2})$ sequential searches in the superblocks of size $O(\ell^{1/2})$. Since Y_2' is contiguous, the merging of X_2 and Y_2 can use the binary scheme of Section 2.1, i.e., the number of comparisons in the main part of the first phase is bounded by $\tilde{m}(\tilde{t}+1) + \tilde{n}/2^{\tilde{t}}$, ($\tilde{t} = \lfloor \log_2(\tilde{n}/\tilde{m}) \rfloor$), where $\tilde{m} \leq \ell$ and $\tilde{n} \leq \ell \cdot n/m$. This gives $O(\ell \log_2 s)$ comparisons, using $m > n^{1/2}$ and $O(\log_2 m) = O(\log_2 s)$. The merging of X_1 with Y_1-blocks in the second phase needs $O(\ell^{3/2})$ comparisons, since we search for the maximal block and update the structure of H $O(\ell)$ times, performing each time $O(\ell^{1/2})$ comparisons. The number of moves is bounded by $3m$ in this phase. To summarize, the number of comparisons required in block merging is $O(s^{3/2})$, using $\ell \leq O(s)$, and the number of moves at most $2n + 4m + O(s)$ if $\ell \leq m^{1/2}$, but only at most $2n + 3m + m^{3/4} + O(s)$ if $\ell > m^{1/2}$.

4.6 Local merging

After block merging the array consists of subsequences $X_1Y_1X_2Y_2 \ldots X_kY_k$, where each X_i is a collection of some X-blocks, i.e., $X_i = X_{i,1}X_{i,2} \ldots X_{i,h_i}$, and each Y_i a collection of Y-blocks, i.e., $Y_i = Y_{i,1}Y_{i,2} \ldots Y_{i,g_i}$, for some $h_i, g_i > 0$ (see Figure 2). Let x_i^L and x_i^R denote the *first* element of block $X_{i,1}$ and X_{i,h_i}, and y_i^L, y_i^R the *last* element of $Y_{i,1}$, Y_{i,g_i}, respectively. Due to the choice of the keys for block merging, we have

$$y_{i-1}^R < x_i^L \leq x_i^R \leq y_i^L \leq y_i^R \,, \tag{1}$$

for each $i \in \{1, \ldots, k\}$ (take $y_0^R = -\infty$), and hence also $y_1^R < y_2^R < \ldots < y_k^R$ and $x_1^L < x_2^L < \ldots < x_k^L$. Since the X-sequence contains only ℓ distinct elements, we have $k \leq \ell$.

Recall that our task is to merge X' and Y, where X' denotes the X-elements remaining after the extraction of buffer elements. Now let C_i be the sequence of all X'-elements satisfying $y_{i-1}^R < x \leq y_i^L$ and D_i the sequence of all X'-elements satisfying $y_i^L < x \leq y_i^R$. Then the original X'-sequence can be divided into pieces $X' = C_1D_1 \ldots C_kD_k$. (Recall that $y_k^R = +\infty$ so we have no elements $x > y_k^R$.) These pieces have pairwise disjoint sets of elements, i.e., $\sum_{i=1}^{k}(c_i+d_i) \leq \ell$, where c_i and d_i denote the number of distinct elements in C_i and D_i, respectively. Observe that the merged output of X' and Y can be expressed as $Z_1^C Z_1^D \ldots Z_k^C Z_k^D$, where Z_i^C is obtained by merging C_i with $Y_{i,1}$ and Z_i^D by merging D_i and $Y_{i,2} \ldots Y_{i,g_i}$. After block merging, however, D_i is a short segment placed in the middle of the block X_{i,h_i}, since the X-blocks are

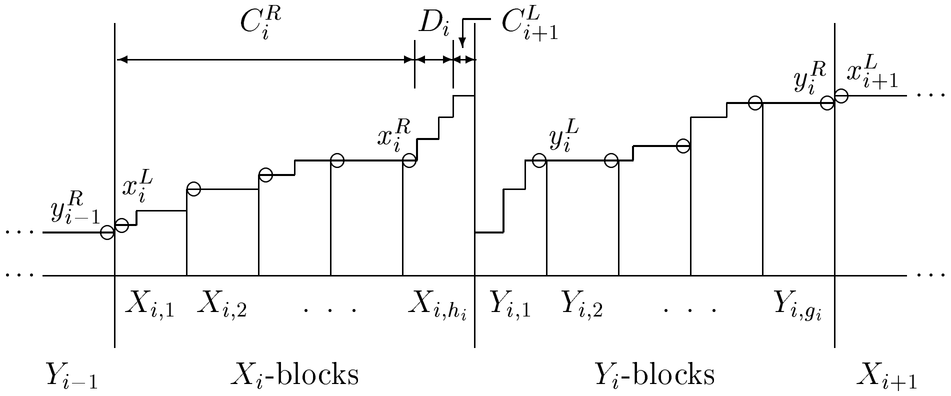

Fig. 2. Local merging after block merging. The key of each block is marked by a circle.

not mixed up and hence all X-elements of D_i must reside between x_i^R and x_{i+1}^L, by (1). Thus, for each i, $X_i = X_{i,1}X_{i,2}\ldots X_{i,h_i}$ consists of three parts, namely, $X_i = C_i^R D_i C_{i+1}^L$, where C_i^R denotes the right segment of C_i and C_{i+1}^L the left segment of C_{i+1}. This implies that sequence C_i is not necessarily contiguous; it can be divided into two pieces C_i^L and C_i^R, separated by Y_{i-1}.

We are now ready for merging. Assume that, after $i-1$ iterations, the configuration of the zone occupied by $X'Y$ is $ZC_i^L X_i Y_i \ldots X_k Y_k$, where Z denotes the already sorted part of all elements not larger than y_{i-1}^R. Clearly, $Z = Z_1^C Z_1^D \ldots Z_{i-1}^C Z_{i-1}^D$. By the foregoing argument, $X_i = C_i^R D_i C_{i+1}^L$, with $D_i C_{i+1}^L$ contained in the last block of X_i. Using binary search over X_{i,h_i}, we determine the boundary between C_i and D_i. Then $D_i C_{i+1}^L$ is merged with Y_i, using a routine similar to that of Section 2.7. But here, instead of single elements, the *plateaus of equal X-elements* are transported into their final locations, by repeated block interchanges. The boundaries between the blocks are determined by binary search. Since the number of distinct elements in D_i is d_i, the number of block interchanges is at most $d_i + 1$; the last interchange transports C_{i+1}^L to the right end (all C_{i+1}^L-elements are larger than y_i^R). Each Y_i-element is involved in at most one interchange, therefore, at most $|Y_i|+2(d_i+1)q$ moves are performed, by Section 2.4 (cf. also Section 2.7). The number of comparisons is bounded by $d_i \cdot O(\log_2 s)$, using $O(\log_2(n+m)) = O(\log_2 s)$. As an additional information, we remember $|C_{i+1}^L|$, the length of block transported to the right end (if any).

Since the block $Y_{i,1}$ is shifted to the left (all D_i-elements are larger than y_i^L), the configuration has been changed to $ZC_i^L C_i^R Y_{i,1} Z_i^D C_{i+1}^L X_{i+1} Y_{i+1} \ldots X_k Y_k$. Block $Y_{i,1}$ of length q begins behind C_i^R, at the boundary we have already computed. The frontier between C_i^L and C_i^R coincides with the former $Y_{i-1}X_i$-boundary, and the length of C_i^L is known from the previous iteration processing $X_{i-1}Y_{i-1}$. Hence, we can merge $Y_{i,1}$ and $C_i = C_i^L C_i^R$, using again the repeated block interchanges, but processing from right to left. Here the number of block interchanges is at most c_i, the number of distinct elements in C_i, which gives

$c_i \cdot O(\log_2 s)$ comparisons and at most $|C_i| + 2c_iq$ moves. Now we have configuration $ZZ_i^C Z_i^D C_{i+1}^L X_{i+1} Y_{i+1} \ldots X_k Y_k$. The process is repeated until the entire sequence is sorted.

Let us now consider the computational costs of the local merging. Observe that $\sum_{i=1}^k |Y_i| = n$, $\sum_{i=1}^k |C_i| \leq m$, and $\sum_{i=1}^k (c_i+d_i) \leq \ell$. Hence, the number of moves is bounded by $\sum_{i=1}^k (|Y_i| + 2(d_i+1)q + |C_i| + 2c_iq) \leq n + m + 2q\ell + 2qk \leq n+5m+O(s)$, using $q = \lceil m/\ell \rceil$ and $k \leq \ell \leq O(s)$. The number of comparisons is bounded by $\ell \cdot O(\log_2 s) \leq O(s \log_2 s)$.

The above strategy is based on the assumption that we know the boundaries between the X_i- and Y_i-collections. Such information cannot be stored in a constant memory, therefore, the block merging of Section 4.5 is interrupted by a call of the procedure described here each time a new collection X_iY_i has been transported to the output. That is, when the first block of X_{i+1} is selected, the routine of Section 4.5 passes all necessary segment boundaries for the ith collection, as well as the current position of the floating hole, as parameters to its local merging procedure. This holds for the first phase of the block merging, placing blocks to the right of position m. The second phase is similar, but since the block merging proceeds from right to left, the local merging proceeds in this direction as well. By symmetry, it can be easily seen that here we divide the Y-sequence into segments C_i and D_i, determined by values of x_i^L and x_i^R.

Now the computational costs for the case of few distinct X-elements can be derived from the preceding sections. By Section 4.1, the initial creation of the buffer takes $O(s \log_2 s)$ comparisons and $2m + O(s \log_2 s)$ moves. By Section 4.5, the block merging can be carried out with $O(s^{3/2})$ comparisons and $2n + 4m + O(s)$ moves if $\ell \leq m^{1/2}$, but with only $2n + 3m + m^{3/4} + O(s)$ moves if $\ell > m^{1/2}$. By Section 4.6, the local merging requires $O(s \log_2 s)$ comparisons and $n+5m+O(s)$ moves. By Section 4.2, the final buffer embedding takes $O(s \log_2 s)$ comparisons and $2n + m + O(s)$ moves if $\ell \leq m^{1/2}$, but $2n + 2m + O(s^{3/2})$ moves if $\ell > m^{1/2}$. Summing up, both for $\ell \leq m^{1/2}$ and $\ell > m^{1/2}$, the total number of comparisons is bounded by $O(s^{3/2}) = O(m^{6/7})$ and that of moves by $5n + 12m + m^{3/4} + O(s^{3/2}) \leq 5n + 12m + O(m^{6/7})$, since $s = \lceil m^{4/7} \rceil$.

5 Concluding remarks

In this paper we showed that in-place merging can be accomplished efficiently with almost optimal number of comparisons, even if the resulting algorithm is required to be stable. Both our semi-stable and stable algorithms run in linear time and perform $m(t+1) + n/2^t + o(m)$ element comparisons, where $t =$

$\lfloor \log_2(n/m) \rfloor$. For semi-stable merging the number of element moves is bounded by $3n+3m+o(m)$, whereas for stable merging the corresponding upper bound is $5n + 12m + o(m)$. We also proved that $n + 2m$ is a lower bound for the number of moves carried out by any in-place merging algorithm. There seems to be a trade-off between the number of moves and comparisons required, since the above lower bound is reachable with an algorithm performing a superlinear number of comparisons (cf. [11]). Note also that the block merging of Section 4.5 is an unstable merging algorithm performing $2n+3m+O(m^{1/2})$ moves, when implemented with block size 1 and superblock size $\lceil m^{1/2} \rceil$. The price we pay is $O(m^{3/2})$ additional comparisons. However, there is still a big gap between the upper and lower bounds for the number of moves when only a (sub)linear number of comparisons or linear time is permitted. A possible improvement of our upper and/or lower bounds is left as an open problem.

Acknowledgement

We would like to thank Jukka Teuhola for his comments on an earlier version of this paper, George Titan for the preliminary discussions that led to the conference version of this paper, and Jesper Larsson Träff for presenting the conference version at the 20$^{\text{th}}$ Symposium on Mathematical Foundations of Computer Science.

References

[1] Dudziński K., Dydek A., On stable minimum storage merging algorithm, *Information Processing Letters* **12** (1981), 5–8

[2] Dvořák S., Ďurian B., Towards an efficient merging, *Proceedings of the 12th Symposium on Mathematical Foundations of Computer Science*, Lecture Notes in Computer Science **233**, Springer-Verlag, Berlin/Heidelberg (1986), 290–298

[3] Huang B-C., Langston M. A., Practical in-place merging, *Communications of the ACM* **31** (1988), 348–352

[4] Huang B-C., Langston M. A., Fast stable merging and sorting in constant extra space, *The Computer Journal* **35** (1992), 643–650

[5] Hwang F. K., Lin S., A simple algorithm for merging two disjoint linearly ordered sets, *SIAM Journal on Computing* **1** (1972), 31–39

[6] Katajainen J., Pasanen T., Teuhola J., Practical in-place mergesort, *Nordic Journal of Computing* **3** (1996), 27–40

[7] Knuth D. E., *The Art of Computer Programming*, Vol. 3: *Sorting and Searching*, Addison-Wesley Publishing Company, Reading (1973)

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

[9] Mannila H., Ukkonen E., A simple linear-time algorithm for in situ merging, *Information Processing Letters* **18** (1984), 203–208

[10] Munro J. I., Raman V., Sorting with minimum data movement, *Journal of Algorithms* **13** (1992), 374–393

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

[12] Pasanen T., *Lajittelu minimitilassa*, M.Sc. Thesis T-93-3, Department of Computer Science, University of Turku, Turku (1993)

[13] Salowe J. S., Steiger W. L., Simplified stable merging tasks, *Journal of Algorithms* **8** (1987), 557–571

[14] Stockmeyer P. K., Yao F. F., On the optimality of linear merge, *SIAM Journal on Computing* **9** (1980), 85–90

[15] Symvonis A., Optimal stable merging, *The Computer Journal* **38** (1995), 681–690

[16] Trabb Pardo L., Stable sorting and merging with optimal space and time bounds, *SIAM Journal on Computing* **6** (1977), 351–372