Optimal Stable Merging

ANTONIOS SYMVONIS

Basser Department of Computer Science, University of Sydney, Sydney, NSW 2006, Australia,
symvonis@cs.su.oz.au

This paper shows how to stably merge two sequences A and B of sizes m and n, $m \leq n$, respectively, with $O(m+n)$ assignments, $O(m\log(n/m+1))$ comparisons and using only a constant amount of additional space. This result matches all known lower bounds and closes an open problem posed by Dudzinski and Dydek in 1981. Our algorithm is based on the unstable algorithm of Mannila and Ukkonen. All techniques we use have appeared in the literature in more complicated forms but were never combined together. They are powerful enough to make stable all the existing linear in-place unstable algorithms we are aware of. We also present a stable algorithm that requires a linear number of comparisons and assignments which we consider to be the simplest algorithm for in-place merging.

Received November 6 1994

1. INTRODUCTION

In this paper, we consider the problem of *merging*. We are given two sequences A and B of m and n elements, respectively, such that A and B are sorted in increasing order according to a key value of their elements. Without loss of generality, in the rest of the paper we assume that $m \leq n$. The result of merging A and B will be a sequence S of $N = m + n$ elements, consisting of the elements of A and B, such that S is sorted in increasing order according to the same key value of its elements. An element of sequence A is called an *A-element*. Similarly, we define *B-elements*.

We assume that initially A and B occupy segments $L[0 \ldots m-1]$ and $L[m \ldots N-1]$ of an array L, respectively, and that after the merging S occupies the entire array $L[0 \ldots N-1]$.

In addition, we may want our merging algorithm to be *stable*. We say that the merging is stable if the internal ordering of sequences A and B is maintained in the resulting sequence S, i.e. elements with the same key value appear in sequence S in the same order in which they appear in sequences A and B before the merging. (In the case that elements with the same key appear in both sequences, the elements in sequence A are considered to be ''smaller'' than the elements with the same key in sequence B.)

The performance of any merging algorithm is judged according to the number of computation steps it performs during the merging as well as the amount of extra space used (in addition to the space for the array $L[0 \ldots N-1]$). If the merging algorithm uses only a constant amount of extra space, we say that the merging is performed *in place*.

It is relatively easy to derive lower bounds on the time performance of any merging algorithm. The number of assignments will be $\Omega(N)$ since there are instances of the merging problem in which, after the merging, each element is in a location of array L that is different than the one it occupied before the merging. The number of comparisons between keys of individual elements that a merging algorithm needs to perform is $\Omega(m\log(n/m))$. To see that, observe that there are $\binom{m+n}{n}$ different ways to distribute the elements of sequence B into locations of array $L[0 \ldots N-1]$. It follows that $\lceil\log\binom{m+n}{n}\rceil$ comparisons are necessary to distinguish among the $\binom{m+n}{n}$ possible orderings for S. The fact that, for $m \leq n$, $\lceil\log\binom{m+n}{n}\rceil = \Theta(m\log(n/m))$ suggests that $\Omega(m\log(n/m))$ comparisons are required for the merging of two sequences A and B with m and n elements, respectively (see [1], pp. 198–206, for a detailed analysis).

The obvious way of merging two sorted sequences A and B of m and n elements, respectively, into a sequence S of $N = m + n$ elements requires $O(N)$ time, which is optimal, but also uses $O(N)$ additional space (see [2], p. 114). Kronrod [3] derived a method of merging two sorted sequences of a total of N elements in $O(N)$ time using only a constant amount of additional space. In doing so, he introduced the important notion of the *internal buffer* which is used in almost all subsequent algorithms for the merging problem. (Actually, as Salowe and Steiger [4] pointed out, Kronrod's algorithm contained an error that went unnoticed for about 15 years.) A description of Kronrod's algorithm appears in [4] and as the answer to exercise 5.2.4.18 in Knuth's book ([1], p. 169, 623).

Unfortunately, Kronrod's merging algorithm was not stable. Horvath [5] managed to derive a stable algorithm with the same asymptotic complexity which, however, had the undesired characteristic of modifying the keys of the elements during the merging. Pardo [6] overcame this obstacle and finally derived an asymptotically optimal algorithm which did not use key modification.

Even though asymptotically optimal, because of their complex structure and the large constant of proportionality, both of the algorithms of Horvath and Pardo are considered impractical. In the effort to derive practical merging algorithms a rich literature has evolved. Several non-optimal stable algorithms were developed that compromise by using either more time [7–10] or more additional space

[8,9,11,12]. Dvorak and Durian [13], Mannila and Ukkonen [14] and Huang and Langston [15] derived linear time algorithms for unstable in-place merging. The algorithm of Mannila and Ukkonen differs from previously developed algorithms in using an internal buffer of length $\lceil\sqrt{m}\rceil$ instead of $\lceil\sqrt{N}\rceil$. Huang and Langston presented a surprisingly straightforward and practical method for unstable merging that uses (in a more creative way than previously developed algorithms) an internal buffer of size $\lceil\sqrt{N}\rceil$. In a later paper [16], they managed to make their algorithm stable. They also used their technique to attack the related problems of *duplicate-key extraction* [17].

Even though there are algorithms that perform stable merging by using only a constant amount of extra space in linear time, no one succeeds in matching the lower bounds on both the number of comparisons ($\Omega(m\log(n/m))$) and the number of element assignments ($\Omega(N)$). The algorithms of Horvath [5], Pardo [6] and Huang and Langston [16] perform $O(N)$ comparisons and element assignments. SPLITMERGE [11] matches the lower bounds but uses $O(m)$ extra space. The algorithm of Mannila and Ukkonen matches all the lower bounds (number of comparisons/ assignments and extra space) but is unstable. To achieve that, it uses the binary merge algorithm of Hwang and Lin [18].

This paper serves two purposes. First it presents a collection of useful techniques used in merging and, second, it presents an optimal stable in-place merging algorithm. We show how to make stable the algorithm of Mannila and Ukkonen while maintaining the same asymptotic complexity on the number of comparisons, assignments and the extra space. This yields the first optimal stable merging algorithm with respect to all known lower bounds. Surprisingly enough, the method we use to make the algorithm stable is very simple. It can in fact be used to stabilize all the unstable merging algorithms known to the author.

The paper is organized as follows. In the next section the details of several techniques used in merging are reviewed. These include block exchange algorithms, a binary-like search algorithm, a simple (stable but not in-place) merging algorithm, and the use of the internal buffer. Their performance analysis is presented and for the case of operations related to the internal buffer and the simple merging algorithm, the method of binary-like searching is employed to reduce the number of required comparisons. In Section 3, the algorithm of Mannila and Ukkonen [14] is presented. In Section 4, we present two methods that can be used for making stable an unstable algorithm and then we show how to modify the algorithm of Mannila and Ukkonen so that it is stable and optimal. In Section 5, we show how to perform the merging optimally in the case that we are not able to build the necessary internal buffers and thus to apply our general merging algorithm. In Section 6, a simple in-place and stable merging algorithm is presented. It requires linear time but does not match the lower bound on the number of comparisons. However, it is the simplest to describe stable algorithm to date. We conclude in Section 7.

2. BASIC TECHNIQUES

This section presents some general techniques that are widely used in merging. They will be used extensively in the remainder of this paper.

2.1. Block exchanges

A *block exchange* of two consecutive blocks U and V of sizes l_1 and l_2, respectively, that occupy segment $L[c \ldots c+l_1+l_2-1]$ results in the movement of block U which occupies segment $L[c \ldots c+l_1-1]$ to segment $L[c+l_2 \ldots c+l_1+l_2-1]$, and in the movement of block V initially in segment $L[c+l_1 \ldots c+l_1+l_2-1]$ to segment $L[c \ldots c+l_2-1]$ (see Figure 1). The above operation can be considered to be a *circular shift to the right* of length l_2.

A trivial way to implement the block exchange operation by using only constant extra space, is to consider it as a sequence of l_2 one-element circular shifts to the right (or, l_1 one-element circular shifts to the left if $l_1 < l_2$). Each one-element circular shift requires l_1+l_2+1 assignments for its implementation. This results to a total of min $(l_1, l_2)\cdot(l_1+l_2+1)$ assignments.

There is a more elegant algorithm which is considered to be part of the folklore.

```
BLOCK_EXCHANGE $(c, l_1, l_2)$
    INVERSE $(c, c+l_1-1)$
    INVERSE $(c+l_1, c+l_1+l_2-1)$
    INVERSE $(c, c+l_1+l_2-1)$
```

Procedure *INVERSE* $(i,j), i \leq j$, performs an inversion of the elements in segment $L[i \ldots j]$. It swaps $L[i]$ with $L[j]$, $L[i+1]$ with $L[j-1]$, $L[i+2]$ with $L[j-2]$, and so on. In general, the element at position $k, i \leq k \leq j$, is swapped with the element at position $i+j-k$. It is easy to verify that about l_1+l_2 swaps are performed or, since swapping two elements requires exactly three assignments, $3(l_1+l_2)$ assignments. The proof of correctness is left to the reader.

While the number of assignments required by procedure *BLOCK_EXCHANGE* (c, l_1, l_2) is linear in the number of elements participating in the block exchange, reducing their number is something desirable. A more complicated method developed by Dudzinski and Dydek [7] performs the block exchange by using only $l_1+l_2+\gcd(l_1,l_2)$ assignments, where $\gcd(l_1,l_2)$ denotes the greatest common divisor of l_1 and l_2. We refer the reader to [7] for a description and analysis of this method.

In the above discussion, it was assumed that only

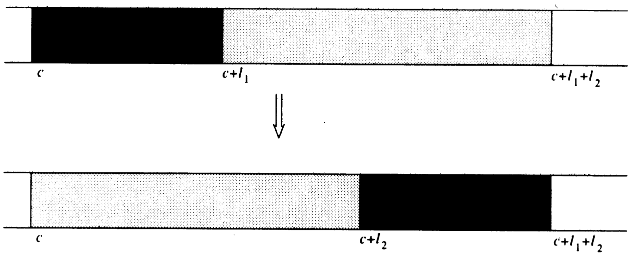

FIGURE 1. A block exchange.

consecutive blocks are exchanged. It is not difficult to exchange non-consecutive blocks. Say that we want to perform a block-exchange between blocks $U = L[c \ldots c + l_1 - 1]$ and $V = L[d \ldots d + l_2 - 1]$ that are separated by a block $M = L[c + l_1 \ldots d - 1], d > c + l_1$. The easiest way to derive the desired result is by applying *BLOCK_EXCHANGE* three times: first *BLOCK_EXCHANGE* $(c + l_1, d - c - l_1, l_2)$, then *BLOCK_EXCHANGE* (c, l_1, l_2), and finally, *BLOCK_EXCHANGE* $(c + l_1, l_2, d - c - l_1)$. However, this might dramatically affect the number of required assignments since the elements of block M are also moved. To avoid this, we have to generalize *BLOCK_EXCHANGE*. Fortunately, this is not too difficult since the only thing we have to do is to adjust all array references to take into account the existence of block M.

2.2. Binary-like search

We present the technique (which we call *binary-like search*) that Hwang and Lin used in their binary-merging algorithm [18], [1], pp. 204–206. It is through the binary-like search technique that we succeed in reducing the number of comparisons of the merging algorithm from $O(N)$ to $O(m \log(n/m + 1))$.

Assume that we are given a sorted sequence of N elements and element x to be inserted in that sequence. Binary-like search finds the element (its location) after which x is to be inserted[1] in $O(m + \log(N/m))$ comparisons for any $m \leq N$.

The algorithm proceeds as follows: The sequence is split into blocks of size $\lceil N/m \rceil$. By comparing the last element of the blocks, starting from left to right, we locate the block in which x is to be inserted. Then, in that block, we perform a binary search to locate the exact position for the insertion. We need $O(\log(N/m))$ comparisons for the binary search and $O(m)$ comparisons to locate the block that x is to be inserted. Thus, the $O(m + \log(N/m))$ bound on the number of comparisons. It is interesting to observe that in the case $m = 1$ binary-like search reduces to binary search, while in the case $m = N$ it reduces to linear search.

Now assume that we have m sorted elements that we wish to insert (or better, compute the elements after which they must be inserted) in a sorted array of N elements. Clearly we can complete this task with $O(m^2 + m \log(N/m))$ comparisons by simply performing a binary-like search for each one of the m elements. However, we have not exploited the fact that the m elements are sorted. If we take it into account, it is not difficult to insert all of them into the sorted array of N elements with $O(m + m \log(N/m))$ comparisons. To see that, simply observe that the ith element will be inserted in a position to the right of the $(i - 1)$th element. This implies that when we try to locate the block in which the ith element must be inserted, we can start our search from the block in which the $(i - 1)$th element was inserted. The result then follows from the fact that the array in which we insert the elements was partitioned into m blocks.

2.3. A simple merging algorithm

In this section, we briefly present a simple merging algorithm that we will use as a building block in our optimal stable merging algorithm. The algorithm stably merges two sorted sequences A and B of m and n elements respectively, $m \leq n$. It uses extra space enough to store m elements. Assume that A and B occupy segments $L[0 \ldots m - 1]$ and $L[m \ldots n + m - 1]$, respectively, and that $T[0 \ldots m - 1]$ is the additional space that we will use to do the merging.

At the first step, the algorithm copies the elements of sequence A into $T[0 \ldots m - 1]$. Then the merging proceeds as follows: Assume that we have already merged a elements of sequence A with b elements of sequence B and that:

- the resulting merged sequence occupies segment $L[0 \ldots a + b - 1]$,
- the remaining elements of sequence B occupy segment $L[m + b \ldots m + n - 1]$, and
- the remaining elements of sequence A occupy segment $T[a \ldots m - 1]$.

Notice that the number of positions in array L which are not occupied by either the already merged sequence or the remaining unmerged elements of sequence B, is equal to the number of the remaining unmerged elements of sequence A which are stored in array T. This property is an invariant of the algorithm.

At the next step, the elements in positions $T[a]$ (an A-element) and $L[m + b]$ (a B-element) are compared. The smallest element is moved to position $L[a + b]$. Notice that the invariant is maintained no matter which element was moved to that position. The algorithm terminates when one of the two sequences is exhausted. If sequence A is exhausted first, then array L contains the merged sequence. If sequence B is exhausted first, then we have to move the remaining elements of the A sequence from T to the empty positions in array L. These positions are in the rightmost part of L and the invariant guarantees that their number is equal to the number of the remaining A-elements.

It is trivial to see that the algorithm performs $O(m + n)$ comparisons and $O(m + n)$ assignments. However, an easy modification of the algorithm can reduce the number of comparisons to $O(m + \log(n/m))$. Instead of comparing at each step the smallest A and B elements, we identify (from the remaining B sequence) the B-element after which the smallest A-element (which is still unmerged) will be inserted. Then we move all the unmerged B elements which are smaller than the A-element at the end of merged sequence. We do that with a block exchange operation. Then we move the A-element at the end of the already merged sequence. In order to identify the positions after which we will insert the A-elements, we use the binary-like

[1] We are interested in computing only the location after which x must be inserted since this is the only operation which will contribute to the number of comparisons between elements. Moving the elements around (in order to create a free position for x) contributes only to the number of assignments.

search method which was developed in the previous section. The analysis is identical and guarantees that the merge will be completed after $O(m + \log(n/m))$ comparisons and $O(m + n)$ assignments.

2.4. The internal buffer

Kronrod [3] introduced the notion of the *internal buffer* in his effort to derive a merging algorithm that uses constant extra space. An internal buffer is simply a segment of the input array which is used for buffering purposes. To make the use of the buffer clear, we demonstrate it with an example. Suppose that array $L[0 \ldots N - 1]$ contains two sorted sequences that are to be merged, the first (A) occupying positions $L[0 \ldots N/2 - 1]$ and the second (B) occupying positions $L[N/2 \ldots N - 1]$. Furthermore, assume that we know an algorithm that can merge the two sequences by using only extra space of $\alpha < N/2$ elements. For the purposes of this example assume that $\alpha = \lceil\sqrt{N}\rceil$. Obviously, we can merge the two sorted $A' = L[\lceil\sqrt{N}\rceil \ldots N/2 - 1]$ and $B = L[N/2 \ldots N - 1]$ by using the $\lceil\sqrt{N}\rceil$ leftmost positions of array L as the extra space required by our algorithm. This is our internal buffer. If we manage to ensure that at the end of the merging of A' with B, the buffer contains the elements that were initially in positions $L[0 \ldots \lceil\sqrt{N}\rceil - 1]$ (but possibly permuted), then sorting[2] and then distributing these elements into the sorted array $L[\lceil\sqrt{N}\rceil \ldots N - 1]$ completes the merging.

Notice that, in order to be able to complete the merging, we have to make sure that we do not destroy the original contents of the internal buffer. To achieve that, whenever we want to store an element into a position of the buffer, we make sure that the element stored previously in that position is moved to another position of the array. Thus, we allow the modification of the buffer by either swaps or circular shifts of its elements.

After merging the sequences (A' and B in the above example) with the use of the internal buffer, the elements of the buffer are not in their original order. Sorting the buffer and then distributing it into the already merged sequence will produce the final merged sequence. Notice that, in the case that the buffer does not consist of distinct elements, the merging is not stable. This is because the sorting of the buffer is not enough to restore the initial order of elements with the same key value. However, when all buffer elements are distinct the distribution of the buffer in the already sorted sequence can be done in a stable fashion.

In the rest of this paper, the buffers will always occupy the left part of the array. By realizing the importance of having a buffer of distinct elements for stable merging, we show how to extract a buffer of distinct elements.

Assuming that a buffer of size b is needed, we will move b distinct elements into the left part of the array. We call this operation *buffer extraction*. The reverse operation, *buffer distribution*, is also important. In the next section, we see how we can implement these operations.

2.4.1. *Buffer extraction and distribution*

For simplicity, we assume that the input array contains only one sorted sequence with at least b distinct elements. We will form an internal buffer of b distinct elements placed in segment $L[0 \ldots b - 1]$.

We build the buffer by adding one element at a time. Initially it consists of the element $L[0]$. Assume that after adding j elements to the buffer, $1 \leq j < b$, the buffer occupies the segment $L[i \ldots i + j - 1]$, $i + j - 1 < N - 1$, the elements of the buffer are sorted in increasing order, and that $L[i + j - 1]$ was the last element added to the buffer. We add another element to the buffer as follows: Let $L[k], k \geq i + j$, be the element with the smallest index in segment $L[i + j \ldots N - 1]$ that satisfies the relation $L[i + j - 1] < L[k]$. We add element $L[k]$ into the buffer by exchanging the contents of segments $L[i \ldots i + j - 1]$ and $L[i + j \ldots k - 1]$. This can be done in $O(k - j)$ steps by a call to our *BLOCK_EXCHANGE* routine. Observe that, after the expansion of the buffer with a new element, the elements of the buffer are still distinct and sorted in increasing order. We continue this process until a buffer of b elements is created. A final block exchange will move the buffer to the beginning of the array.

Note that, in the case where duplicate elements exist, only the leftmost of them might be included in the buffer. This property simplifies the distribution of the buffer into the sorted sequence in a stable manner. Also note that, in the case that there are not b distinct elements, the above algorithm creates a buffer of maximum possible size.

The time complexity of the above procedure is $O(b^2 + N)$. To see that, simply observe that each element that does not belong to the buffer is moved exactly once to the left, while each buffer element moves at most b times to the right.

It is a simple task to modify the buffer extraction algorithm to work when the input array contains two sorted sequences A and B of m and n elements, respectively. The algorithm will essentially remain the same but some attention is needed when extracting elements from sequence B. We have to make sure that an element with the same key value was not extracted previously from sequence A.

By recalling our initial goal, i.e. to obtain a linear time stable merging algorithm, we conclude that the maximum size of the buffer that we can afford is $O(\sqrt{N})$.

In the buffer extraction algorithm described above, the next element to be added to the buffer is located by a linear scan of the two sequences. As a result, $O(N)$ comparisons are performed. We will show that the number of comparisons can be reduced in some settings.

THEOREM 1. *A buffer of $\lceil\sqrt{m}\rceil$ distinct elements can be*

[2] We can use any kind of in-place sorting method, even a quadratic one, if the contribution of the sorting does not affect the final time complexity. To achieve the results stated in this paper, we use buffers of size $O(\sqrt{m})$. Thus, sorting by a quadratic in-place method, say *insertion sort*, will contribute to the final complexity at most $O(m)$ element comparisons and assignments.

extracted from an array containing two sorted sequences A and B of m and n elements, $m < n$, respectively, in linear time and with $O(m + \sqrt{m}\,log(n/m))$ element comparisons.

Proof. We prove the theorem by demonstrating an algorithm that achieves the stated performance. The algorithm is similar to the buffer extraction algorithm given before. The only difference is in the way that we locate the elements that we add to the buffer. We scan sequence A, element by element. This will contribute m comparisons. Sequence B is scanned by the binary-like searching method. If the last element of sequence B that was added to the buffer is at position i, then we perform the binary-like searching starting from position i and by using blocks of size n/m. Note that at most $O(\sqrt{m})$ distinct elements from sequence B have to be rejected from inclusion in the buffer because they were already included in the buffer from sequence A. This results in a possible overhead of $O(\sqrt{m}\log(n/m))$ comparisons. Putting everything together, we see that the buffer will be extracted after $O(m + n)$ steps with $O(m+ \sqrt{m}\log(n/m))$ comparisons. Q.E.D.

It should be evident that distributing (or equivalently merging) a buffer of size b within a sorting sequence of size N can be performed in time $O(b^2 + N)$ by exactly the reverse procedure of buffer extraction. Actually, the distribution can be performed in a stable way by insisting that each buffer element ''is smaller than'' elements of the sequence with the same key value. When we are concerned with the number of comparisons, we can prove the following theorem:

THEOREM 2. *A buffer of $\lceil\sqrt{m}\rceil$ distinct elements can be distributed into an array of N elements in linear time and with $O(m + \sqrt{m}\,log(N/m))$ element comparisons.*

Proof. An algorithm that satisfies the stated performance is one that operates in a reverse way to the corresponding algorithm for buffer extraction. Q.E.D.

3. THE MERGING ALGORITHM OF MANNILA AND UKKONEN

In this section we review the merging algorithm of Mannila and Ukkonen [14]. Their algorithm uses a constant amount of additional space and performs $O(N)$ assignments and $O(m\log(n/m))$ comparisons. Its only drawback is that it is unstable. In the next section we show how to make it stable.

We have to merge sequence A consisting of m elements, with sequence B consisting of n elements, $m \le n$, which occupy the m leftmost and the n rightmost positions of array $L[0 \ldots m + n - 1]$, respectively. The first $\sqrt{m}$ elements of sequence A will be used as an internal buffer. Without loss of generality, we assume that $\sqrt{m}$ is an integer.

The algorithm splits sequence A into $\sqrt{m}$ blocks of fixed length, i.e., $length(A_i) = \sqrt{m}, 0 \le i \le \sqrt{m} - 1$. A_0 will be used (and referred to) as the internal buffer. Sequence B is partitioned into $\sqrt{m} - 1$ blocks $B_1, B_2, \ldots, B_{\sqrt{m}-1}$ of variable length. It is partitioned in such a way that the concatenation of the merged sequences

$$MERGE(A_1, B_1), MERGE(A_2, B_2), \ldots,$$
$$MERGE(A_i, B_i), \ldots, MERGE(A_{\sqrt{m}-1}, B_{\sqrt{m}-1})$$

results in a sorted sequence. In order to achieve that, appropriate splitting points for sequence B must be located. Let $first(X)$ ($last(X)$) denote the position of the first (last) element of a sequence X. We need to specify the values $first(B_i)$ and $last(B_i)$, $1 \le i \le \sqrt{m} - 1$. An appropriate choice is the following:

$$first(B_1) = m$$
$$first(B_i) = last(B_{i-1}) + 1, 1 < i \le \sqrt{m} - 1$$
$$last(B_i) = \begin{cases} m - 1 & \text{if } L[last(A_i)] < L[m] \\ m + n - 1 & \text{if } L[last(A_i)] > L[m + n - 1] \\ j & \text{such that } j \ge m \text{ and} \\ & L[j] < L[last(A_i)] \le L[j + 1] \end{cases}$$

There is the possibility that several of the blocks from sequence B might be empty. A block X is empty when $last(X) = first(X) - 1$.

Note that the boundaries of the blocks from sequence B can be located by a procedure similar to that used in the buffer extraction, and so, $O(\log(n/m))$ steps are enough to locate each one of them. As we will see next, it is not necessary to compute all of them at once, and so we do not need any extra space to store them.

Moving the blocks to positions suitable for the local merging is not an easy task. Finding a clever way to perform it was a key to the success of the algorithm of Mannila and Ukkonen.

We assume that at any time the array is divided into three sections (see Figure 2). The left section contains j pairs of blocks in their final order (ready for the local merging), $j < \sqrt{m} - 1$. The middle section consists of the remaining blocks of the A sequence. The blocks are permuted and it is possible that one of them is divided into two parts. Its left part occupies the right end of the middle section and the rest occupies the left part of the middle section. The right section of the array consists of the remaining (unmerged) part of the B sequence.

The algorithm locates in the middle section the next block to be merged. This will be the block, say A_{j+1}, with the smallest ''*first*'' element or, in the case that there are several blocks with the same smallest ''*first*'' element, any one of them which contains only identical elements. By using $L[last(A_{j+1})]$ we can identify from the right section the corresponding block B_{i+1}. Then we have to transfer these two blocks in the left section of the array in such a way that the middle section maintains its properties.

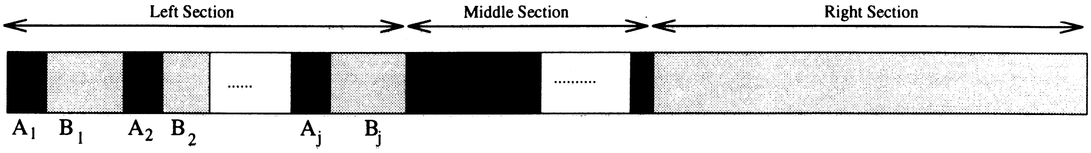

FIGURE 2. During the block rearrangement in the merging algorithm of Mannila and Ukkonen the array is divided into three sections.

There are two cases to consider depending on whether block A_{j+1} is the divided block of the middle section or not.

In the case where A_{j+1} is the divided block, we simply move (by *BLOCK_EXCHANGE*) the first part of the divided block and block B_{j+1} (they are adjacent) immediately after the rest of divided block. To restore the divided block we exchange its two parts. Observe that the middle section retains its structure (Figure 3).

The case where A_{j+1} is not the divided block is easy to handle as well. By two block exchanges between the divided block and A_{j+1} we can make A_{j+1} be the divided block. Now, the block setting is the one described in the previous case (Figure 4).

Locating the blocks from the middle section requires a total of $O(m)$ comparisons. This is because in order to locate the jth block, where $1 \leq j \leq \sqrt{m}$, we have to examine the first (and possibly the last) element of each block in the middle section. Since at any time there are at most $\sqrt{m}$ blocks, the required number of comparisons is $O(m)$. Each local merge can be performed immediately after the blocks to be merged are moved next to each other. This is very important for achieving an in-place merging algorithm. Otherwise, we would have to use $\Omega(\sqrt{m})$ extra space to store the boundaries of the blocks. The merging of the paired blocks is done by the simple merging algorithm described in Sections 2.3 and 2.4. This algorithm needs extra space for $\sqrt{m}$ elements. The internal buffer ($L[0 \ldots \sqrt{m}-1]$) is exactly that size. Merging each pair of blocks requires $O(m+\sqrt{m}\log(n/m))$ comparisons, and $O(m\log(n/m+1))$ comparisons are enough for all local merges [14]. Sorting the buffer with a quadratic method, say selection sort, requires at most $O(m)$ comparisons and assignments. The task of distributing the buffer into the merged sequence does not affect the final complexity if performed by the algorithm described in Section 2.4.1.

4. AN OPTIMAL STABLE MERGING ALGORITHM

To the best of our knowledge, all the known in-place unstable merging algorithms fail to be stable because of two common problems. The first of them concerns their ability to extract buffers composed of distinct elements. After performing all local merges with the use of the internal buffer, the elements of the buffer are permuted. If the elements are distinct, then sorting the buffer will restore the initial order. However, if there are elements with the same key value, then sorting will not necessarily restore the initial order of these elements. Thus the use of internal buffers consisting of distinct elements is imperative if the desired outcome is a stable merging algorithm. In Section 5, we show how to do the merging in the case where it is impossible to create distinct-element internal buffers of the appropriate size.

The second problem arises from the fact that almost all of the merging algorithms are based on local merges of blocks from the two sequences. During the merging, the next block that will participate in the local merge must be located. This is usually done by searching for the block with the smallest first (or last) element. In the case that in

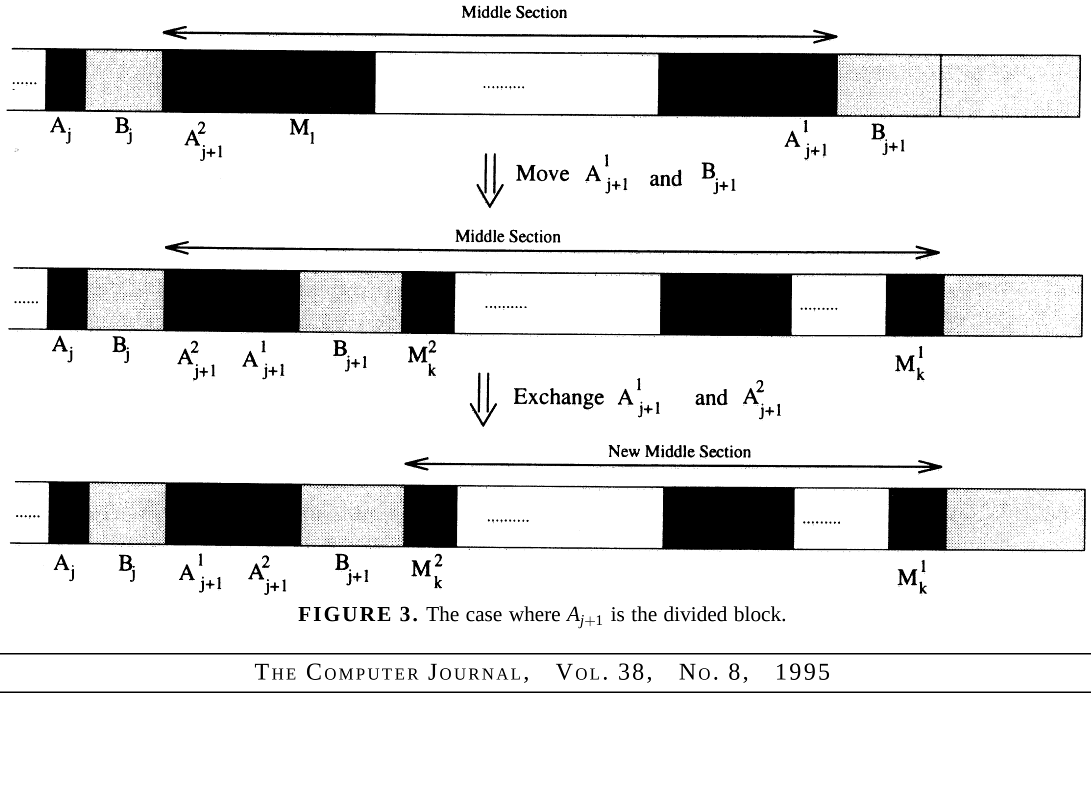
Middle Section
......
....
A_j B_j A^2_{j+1} M_1 A^1_{j+1} B_{j+1}
Move A^1_{j+1} and B_{j+1}
Middle Section
......
....
....
A_j B_j A^2_{j+1} A^1_{j+1} B_{j+1} M^2_k M^1_k
Exchange A^1_{j+1} and A^2_{j+1}
New Middle Section
......
....
....
A_j B_j A^1_{j+1} A^2_{j+1} B_{j+1} M^2_k M^1_k

FIGURE 3. The case where A_{j+1} is the divided block.

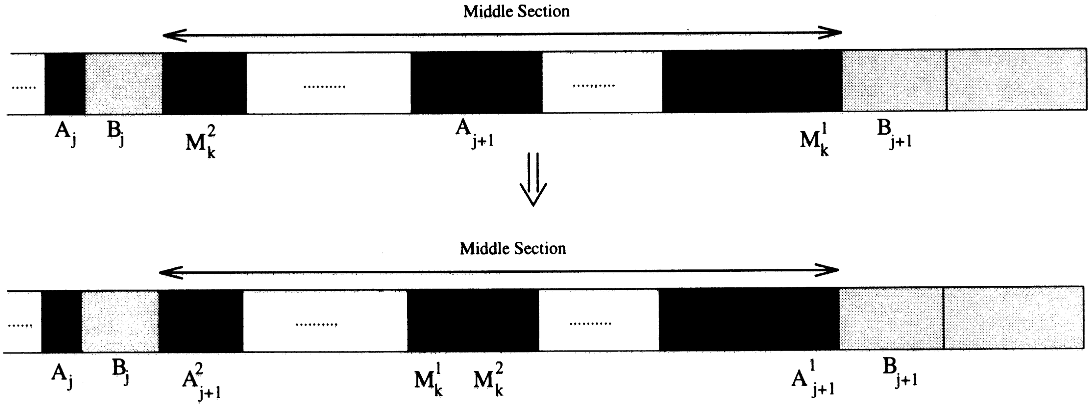

FIGURE 4. The case where A_{j+1} is not the divided block.

a sequence there are more elements with the same key value than the size of the block, there might be more than one block with the same first (or last) element. Since during the merging the blocks are permuted, it is possible to pick the blocks in the wrong order. (The error in the algorithm of Kronrod [3] is related to the above problem.) A surprisingly easy method that can be used to overcome this problem is to number the blocks. This will ensure that we use the blocks in the correct order. However, since we are allowed to use only a constant amount of extra memory, we must record the block number in a different way. We will show two methods to do it.

4.1. Method I: creating a ''peak''

Assume that there are k blocks that we want to number. In this method we mark the ith block, $1 \leq i \leq k$, by substituting the ith element of the block by an element with key value larger than all block elements. To make the marking possible, we create a sorted sequence of at least k elements by partially merging the right ends of the initial sequences A and B. The length of the new sorted sequence is sufficient to guarantee that its smallest element is larger than all the elements in the blocks we have to mark. Having created that sequence, we exchange the ith element of the ith block, $1 \leq i \leq k$, with the ith element of the sequence. During the merging, when we locate a block, we swap back the elements to restore it and then proceed with the merging. Notice that the ordering of the sequence with the ''large'' elements is not destroyed. Salowe and Steiger [4] used a similar but much more complicated method in their stable in-place linear time merging method. In their paper they also treat the problem that might occur when during the creation of the sequence with the ''large'' elements, one of the original sequences is exhausted. (On the contrary, the merge is trivially simple.)

4.2. Method II: movement imitation

The second method we present uses an additional buffer of size equal to the number of blocks we need to mark. The elements of the buffer must be distinct and the buffer sorted. We put the buffer elements into 1-to-1 correspondence with the blocks. The ith smallest buffer element corresponds to the ith block. Then, during the merging we maintain this correspondence by imitating the movement of the blocks by the elements of the buffer. So, when we want to locate the ith block, we compute the position of the ith smallest element of the buffer. The ith block will be in the same relative position with respect to the other blocks. The above method is a simplified version of the method used by Pardo [6].

4.3. The optimal algorithm

Having available all the techniques developed in the previous sections, we describe how to turn the algorithm of Mannila and Ukkonen [14] into a stable one and thus obtain an optimal in-place and stable merging algorithm.

We start by extracting two buffers, each of $\sqrt{m}$ distinct elements. We assume that there are enough distinct elements to create the buffers. We will see in the next section how to treat the case in which we are not able to form the buffers. We use the first buffer to perform the local merges as described in Section 2.3 and 2.4. The second buffer is used to ensure that the blocks of sequence A are merged in the correct order. For clarity reasons, in our presentation we choose to use the method of *movement imitation*. The method of *creating a ''peak''* can also be used.

After splitting the remaining elements of sequence A into blocks of size $\sqrt{m}$ the array has the following form:

$$\langle buffer\_1\rangle\langle buffer\_2\rangle A_{small}A_1A_2\ldots$$

$$\ldots\langle\text{remaining elements of sequence } B\rangle$$

where A_{small} is a non-full block (i.e. a block with less than $\sqrt{m}$ elements) that we ignore in the next step.

We proceed by executing a stable version of the algorithm of Mannila and Ukkonen. *buffer_1* is used for the local merges while *buffer_2* is used in implementing the movement imitation method. This make it possible to choose the blocks of the A sequence in their correct order. After the merging, the array has the following form:

$\langle \mathit{buffer\_1} \rangle \langle \mathit{buffer\_2} \rangle A_{small}$

⟨Stably sorted sequence of the remaining elements⟩

We sort the elements of each buffer and then we complete the stable merging by distributing from the left and in a stable fashion first A_{small}, then *buffer_2*, and finally *buffer_1*.

Based on the analysis in earlier sections, it follows that the above algorithm is stable and that it performs $O(m \log(n/m + 1))$ comparisons and $O(N)$ assignments; thus it is optimal.

5. MERGING IN THE PRESENCE OF AT MOST λ DISTINCT KEYS

The optimal stable merging algorithm presented in this paper assumes that there exist enough distinct elements for the extraction of two buffers each of size $\sqrt{m}$. In this section we present a way to do the merging when there are $\lambda < \alpha$ distinct elements. In the case where we use the method of *creating a ‘‘pe ak’’*, $\alpha = \sqrt{m}$. If we use the method of *movement imitation*, $\alpha = 2\sqrt{m}$. The algorithm was first presented in [4] and it was based on Kronrod's algorithm [3] and the ideas of Pardo [6]. Our presentation is on the lines of [4] with appropriate modifications of the block sizes in order to achieve the desired complexity with respect to the number of comparisons performed by the algorithm.

Assume that we have already extracted a buffer of maximum possible size $\lambda < 2\sqrt{m}$ elements which are sorted. We divide the remaining elements of each of the two sequences into blocks of size $\lceil (m+n)/\lambda \rceil$. The array now has the following form:

$$\langle \mathit{buffer} \rangle A_{small} A_1 A_2 \ldots B_1 B_2 \ldots B_{small}$$

where A_{small} and B_{small} are blocks with less than $\lceil (m+n)/\lambda \rceil$ elements. (We call them *non-full* blocks. All other blocks are *full* blocks.) We ignore these blocks for the moment. We will distribute them into the sorted sequence at the end.

We put each full block into a 1-to-1 correspondence (from left to right) with the buffer elements. Let k be the key value of the buffer element that corresponds to the last A block. Then, all B blocks correspond to buffer elements with key values greater than k. This provides us with an easy method to determine the sequence in which a block belongs.

Next, we stably sort the blocks based on the key value of their first element. While doing that, the movement of each block is imitated by the buffer elements. The stable sorting of the blocks can be done in place by a variant of selection sort. The method of the *movement imitation*, together with the fact that all buffer elements that correspond to blocks of sequence A (B) are smaller or equal to (larger than) k, allows us to perform the stable sorting of the blocks. It requires $O(N)$ assignments and $O(\lambda^2) = O(m)$ comparisons.

After the stable block sorting, all block elements possess an important property. If the final position of an element e in the sorted sequence of all blocks is location i, then after the block sorting e is located in segment $L[1 \ldots i + \lceil (m+n)/\lambda \rceil]$. This follows from the fact that the blocks are stably sorted based on the key value of their first element.

We proceed with the merging of the blocks from left to right. Suppose that we have merged all block elements up to location $l - 1$ and that the element at location l belongs to sequence X (X is either A or B). Let $x = L[q]$ be the first element of type $\bar{X}$ (the opposite of type X) that follows $L[l]$ (Figure 5). If $L[q-1] < L[q]$ ($\leq$ if $L[q]$ belongs to sequence B) then the elements up to location q are merged. In this case we update l to be position $q + 1$. In the case where $L[q-1] > L[q]$ ($\geq$ if $L[q]$ belongs to sequence A), we locate the first element in segment $L[l+1 \ldots q-1]$ that must be placed after x in the sorted sequence. Let it be element y at location p. We also compute the location r that contains the last element, say z, that belongs before y.

We expand the merged sequence of the array by exchanging segments $L[q \ldots r]$ and $L[p \ldots q-1]$. Now the segment $L[1 \ldots p + r - q]$ is merged. We update l to be position $p + r - q$ and we continue until all block elements are exhausted.

Note that each block exchange involves at least two distinct elements. Since there are λ distinct elements, we will perform at most $\lambda/2$ block exchanges. Also note that each exchange moves at most $\lceil (m+n)/\lambda \rceil$ elements to a position which is not their final position. Thus, the algorithm will perform $O(m+n) = O(N)$ assignments. We perform a comparison only when we need to locate elements x, y and z. It is easy to see that by using the ‘‘binary-like searching’’ we will need a total of $O(m + \lambda \log(N/m)) = O(m \log(n/m + 1))$ comparisons.

Now the array has the following form:

$$\langle \mathit{buffer} \rangle A_{small} \langle \text{merged full blocks} \rangle B_{small}$$

We complete the stable merging by firstly distributing in a stable fashion A_{small} from the left, then distributing in a stable fashion B_{small} from the right, and finally sorting the buffer and distributing it in a stable fashion from the left. The number of comparisons performed by the sorting of the buffer and the distribution of the non-

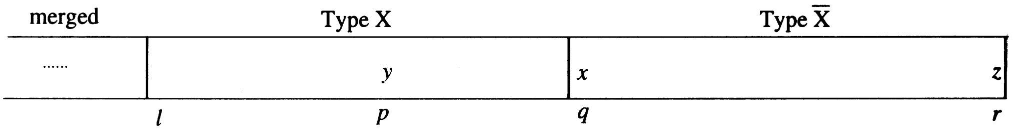

FIGURE 5. Merging in the case where it is not possible to form the required buffers.

full blocks and the buffer will not change the asymptotic performance of the algorithm.

Based on the algorithms presented in this and the previous section, we can state the following theorem:

Theorem 3. *Two sorted sequences (stored in an array) A and B of m and n elements, respectively, can be optimally merged in a stable and in-place fashion with $O(m+n)$ assignments and $O(m \log(n/m+1))$ comparisons.*

6. A SIMPLE STABLE MERGING ALGORITHM

Assume that we want to stably merge two sorted sequences A and B of a total of N elements. Without loss of generality assume that $\sqrt{N}$ is an integer. In this section, we present a stable merging algorithm that requires two buffers, one of size $2\sqrt{N}$ which we call the *merge-buffer*, and one of size $\sqrt{N}$ which is used to keep track of the order in which the sorted blocks of size $\sqrt{N}$ are created. We do that by using the *movement imitation* method. The algorithm merges in linear time but it is not optimal with respect to the number of comparisons. However, it particularly simple and worthy of inclusion.

The algorithm proceeds as follows: we stably merge the elements of the two sequences and we place the resulting sequence in the merge-buffer until one of the two sequences contributes exactly $\sqrt{N}$ elements. Since the buffer is of size $2\sqrt{N}$, this will happen before the buffer is filled. Assume that sequence X (X is either A or B), is the one that contributed the $\sqrt{N}$ elements and that the first unused element of sequence X is at position i. Then, segment $L[i-\sqrt{N}-1 \ldots i-1]$ consists of buffer elements. We exchange this segment with the first half of the buffer, and then we permute the two halves of the buffer. Now, segment $L[i-\sqrt{N}-1 \ldots i-1]$ contains a block of size $\sqrt{N}$ of the final sorted sequence. We use the method of *movement imitation* and the second buffer to record the number of that block. We continue in the same fashion until we exhaust both sequences.

Now, the array contains the two buffers and the sequence of the remaining elements sorted in blocks of size $\sqrt{N}$ that are permuted. By using the second buffer, we sort the blocks and thus, we restore the sorted sequence. We finish the merging by distributing the buffers in the sorted sequence in a stable manner.

The algorithm terminates in linear time. Note that it fails to be optimal with respect to the number of comparisons because its buffer size is $O(\sqrt{N})$ instead of $O(\sqrt{m})$. The creation of the blocks can be performed with only $O(m \log(n/m+1))$ comparisons. However, it is not possible to do the buffer extraction, the block sorting, and the buffer distribution with the same number of comparisons.

7. CONCLUSIONS

In this paper we considered stable in-place merging. We presented a merging algorithm based on the method of Mannila and Ukkonen that performs an optimal number of comparisons and assignments. This closes an open problem mentioned by Dudzinski and Dydek [7] in 1981. We also presented a linear, but non-optimal with respect to the number of comparisons, algorithm that is strikingly simple. Surprising, all techniques presented in this paper have already appeared in the literature, usually in more complicated forms, but they have not been put together to achieve the required result. Salowe and Steiger [4] made an effort to present simple stable algorithms, but even though the unstable versions of their algorithms were simple, the modifications required to make them stable were very complex. However, the methods presented in Section 4 are powerful enough to stabilize all of the unstable in-place merging algorithms known to the author. They are used to make stable the algorithms of Kronrod [3], Dvorak and Durian [13], and Huang and Langston [15], as well as to simplify the algorithms presented by Salowe and Steiger [4].

All algorithms that followed the introduction of the internal buffer by Kronrod in [3] use internal buffers extensively. One interesting challenge would be to establish whether linear in-place merging is possible without the use of an internal buffer.

REFERENCES

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

[2] Horowitz, E. and Sahni, S. (1978) *Fundamentals of Computer Algorithms*, Computer Science Press, MD.

[3] Kronrod, M. A. (1969) ‘An optimal ordering algorithm without a field operation’, *Dokladi Akad. Nauk SSSR*, **186,** 1256–1258.

[4] Salowe, J. and Steiger, W. (1987) ‘Simplified stable merging tasks’, *Journal of Algorithms*, **8,** 557–571.

[5] Horvath, E. C. (1978) ‘Stable sorting in asymptotically optimal time and extra space, *Journal of the ACM,* **25,** 177–199.

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

[7] Dudzinski, K. and Dydek, A. (1981) ‘On a stable storage merging algorithm’, *Information Processing Letters*, **12,** 5–8.

[8] Dvorak, S. and Durian, B. (1986) ‘Towards an Efficient Merging’, *Lecture Notes in Computer Science*, **133,** Springer-Verlag, 200–298.

[9] Dvorak, S. and Durian, B. (1988) ‘Merging by decomposition revisited’, *The Computer Journal*, **31,** 553–556.

[10] Wong, J. K. (1982) ‘Some simple in-place merging algorithms’, *Bit*, **21,** 157–166.

[11] Carlsson, S. (1986) ‘SPLITMERGE: a fast stable merging algorithm’, *Information Processing Letters*, **22,** 189–192.

[12] Dvorak, S. and Durian, B. (1987) ‘Stable linear time sublinear space merging’, *The Computer Journal*, **30,** 372–375.

[13] Dvorak, S. and Durian, B. (1988) ‘Unstable linear time $O(1)$ space merging’, *The Computer Journal*, **31,** 279–282.

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

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

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

[17] Huang, B.-C. and Langston, M. A. (1989) ‘Stable duplicate-key extraction with optimal time and space bounds’, *Acta Informatica*, **26,** 473–484.

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