# Optimal Stable Merging

Antonios Symvonis
Basser Department of Computer Science
University of Sydney
Sydney, N.S.W. 2006
Australia
symvonis@cs.su.oz.au

**Abstract**

In this paper we show 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 [2] 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.

Proc. ICCI'94, 124-143 © 1994 Int. Conf. on Computing and Information

# 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 $C$ of $N = m + n$ elements, consisting of the elements of $A$ and $B$, such that $C$ 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 $C$ 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 $C$. (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 needs to perform the merging as well as the amount of extra space used. 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 every 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 $\Omega(m \log(n/m))$. (See [12, pages 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 $C$ of $N = m + n$ elements (see [7, page 114]) requires $O(N)$ time, which is optimal, but also uses $O(N)$ additional space. Kronrod [13] 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. Unfortunately, Kronrod's merging algorithm was not stable. Horvath [10] 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 [15] overcame this obstacle and finally derived an asymptotically optimal algorithm which didn't 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. Several non-optimal stable algorithms that compromise by using either more time [2, 3, 5, 18] or more additional space [1, 3, 5, 4] were developed. Dvorak and Durian [6], Mannila and Ukkonen [14] and Huang and Langston [8] 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 $\sqrt{m}$ instead of $\sqrt{N}$. 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 $\sqrt{N}$. In a later paper [9], they managed to make their algorithm stable.

None of the known stable in-place merging algorithms succeeds to match 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 [10], Pardo [15] and Huang and Langston [9] perform $O(N)$ comparisons and element assignments. SPLITMERGE [1] 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 merging algorithm of Hwang and Lin [11].

In this paper, 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. Actually, it can be used to make stable almost all of the existing unstable algorithms.

The paper is organized as follows: In the next section we review several techniques that are used in merging. These include block exchange algorithms, binary-like searching and the use of the internal buffer. We present their performance analysis and for the case of operations related to the internal buffer we employ the method of binary-like searching to reduce the number of required comparisons. In Section 3, we present the algorithm of Mannila and Ukkonen [14]. In Section 4, we present our stable optimal merging algorithm. 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 there are less than $2\sqrt{m}$ distinct elements. In this case, we are not able to build the necessary internal buffers and thus to apply our general merging algorithm. We conclude in Section 6.

# 2 Basic techniques

## 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 to the movement of block $U$ to segment $L[c + l_2 \ldots c + l_1 + l_2 - 1]$, and to the movement of block $V$ to segment $L[c \ldots c + l_2 - 1]$. (The case where the two blocks are not consecutive can be treated similarly.)

There is a clever algorithm that is considered to be part of the folklore that performs the block exchange using about $l_1 + l_2$ swaps ($3(l_1 + l_2)$ assignments).

*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* $(i, j)$, $i < j$, performs an inversion of the elements in segment $L[i \ldots j]$ by executing exactly $\lfloor (j - i + 1)/2 \rfloor$ swaps.

A more complicated method developed by Dudzinski and Dydek [2] 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 $l_2$. It is based on the following theorem:

**Theorem 1 [Dudzinsky, Dydek, [2]]** *Given an array $L[0 \ldots s-1]$, a circular shift to the right of size $m$ is decomposed to exactly $\gcd(m, s-m)$ cycles where $\gcd()$ denotes the greatest common divisor function. Furthermore, locations $0 \ldots \gcd(m, s-m) - 1$ belong to different cycles.*

## 2.2 Binary-like searching

We present the technique (which we call binary-like searching) that Hwang and Lin used in their binary-merging algorithm [11][12, pages 204-206]. It is through the binary-like searching technique that we succeed in reducing the number of comparisons of the merging algorithm from $O(N)$ to $O(m \log(n/m))$.

We are given a sorted sequence of $N$ elements and element $x$ to be inserted in that sequence. Binary-like searching finds the location in which $x$ is to be inserted 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 which $m = 1$ binary-like searching reduces to binary searching while in the case which $m = N$ it reduces to linear searching.

## 2.3 The internal buffer

Kronrod [13] 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. However, there is a restriction on how we use the internal buffer. 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. We achieve that by allowing the modification of the buffer by either swaps or circular shifts of its elements.

After merging the sequences 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. Realizing the importance of having a buffer of distinct elements for stable merging, we will show how to extract a buffer.

Assuming that a buffer of size $b$ is needed, we will move $b$ distinct elements in the left part of the array. We call this operation *buffer extraction*. The reverse operation, *buffer distribution*, is also important.

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 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 that there do not exist $b$ 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 above described buffer extraction algorithm, the next element to be added to the buffer is located by a linear scanning of the two sequences. As a result, $O(N)$ comparisons are performed. By using the binary-like searching to locate the elements to be added at the buffer, we can prove the theorem:

**Theorem 2** *A buffer of $\sqrt{m}$ distinct elements can be 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.*

Distributing a buffer of size $b$ within a sorted sequence of size $N$ can be performed in time $O(b^2 + N)$ by exactly the reverse procedure of buffer extraction. When we are concerned with the number of comparisons, we can prove the following theorem:

**Theorem 3** *A buffer of $\sqrt{m}$ 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.*

# 3 The merging algorithm of Mannila and Ukkonen

In this section we will review the in-place algorithm of Mannila and Ukkonen [14]. The algorithm 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 $A$ occupying $L[0 \ldots m-1]$, with sequence $B$ occupying $L[n \ldots m+n-1]$, $m < n$. The first $\sqrt{m}$ elements of sequence $A$ will be used as an internal buffer. We assume that $\sqrt{m}$ is an integer.

The algorithm splits each of the sequences $A$ and $B$ into $\sqrt{m}-1$ blocks. The blocks of sequence $A$ are of fixed length, i.e., $length(A_i) = \sqrt{m},\ 1 \leq i \leq \sqrt{m}-1$, while the blocks of sequence $B$ are of variable length. Sequence $B$ is partitioned in such a way that the concatenation of the merged sequences

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

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 $LAST(B_i)$, $1 \leq i \leq \sqrt{m}-1$. An appropriate choice is the following:

$$FIRST(B_1) = m \qquad FIRST(B_i) = LAST(B_{i-1}+1),\ 1 < i \leq \sqrt{m}-1$$

$$LAST(B_i) = \begin{cases} m-1 & \text{if } LAST(A_i) < L[m] \\ m+n-1 & \text{if } LAST(A_i) > L[m+n-1] \\ j & \text{such that } L[j] < LAST(A_i) \leq L[j+1] \end{cases}$$

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 thus, $O(\log(n/m))$ steps are enough to locate each one of them. As we will see, it is not necessary to compute

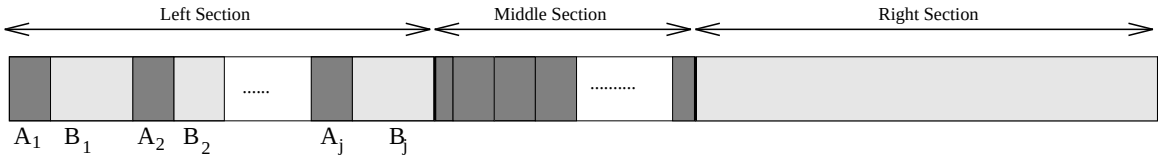


Figure 1: During the block rearrangement in the merging algorithm of Mannila and Ukkonen the array is divided into 3 sections.

all of them at once and thus, we do not need any extra space to store them.

Moving the blocks to positions suitable for the local merges 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 1). The left section contains $j$, $j < \sqrt{m} - 1$, pairs of blocks in their final order (ready for the local merges). 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. We denote the blocks of the middle section by $M_i$, $1 \leq i \leq \sqrt{m} - 1 - j$. The right section of the array consists of the remaining blocks of the $B$ sequence.

The algorithm locates in the middle section the next block to be merged. This will be the block with the smallest $LAST()$ element, say $A_{j+1}$. By using $LAST(A_{j+1})$ we can identify from the right section the corresponding block $B_{j+1}$. 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.

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 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 2).

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 3).

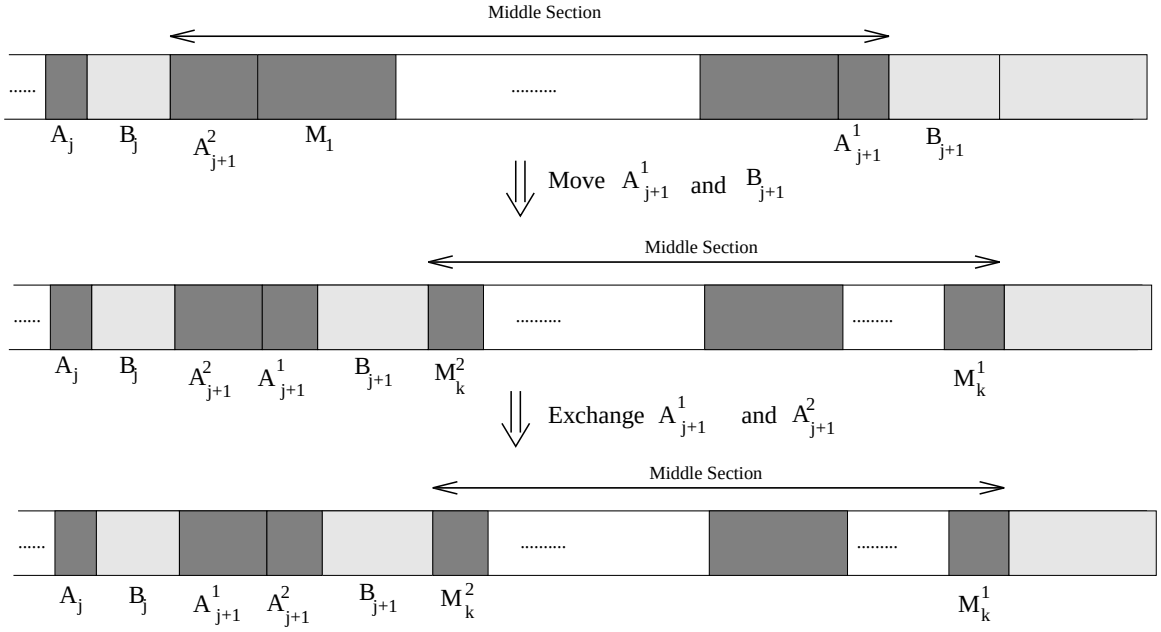


Figure 2: The case where $A_{j+1}$ is the divided block.

Locating the blocks from the middle section requires a total of $O(m)$ compar-

isons. Each local merge can be performed with $O(m + \sqrt{m}\log(n/m))$ comparisons and $O(m\log(n/m\ +1))$ comparisons are enough for all local merges. The task of distributing the buffer into the merged sequence is off the same complexity.

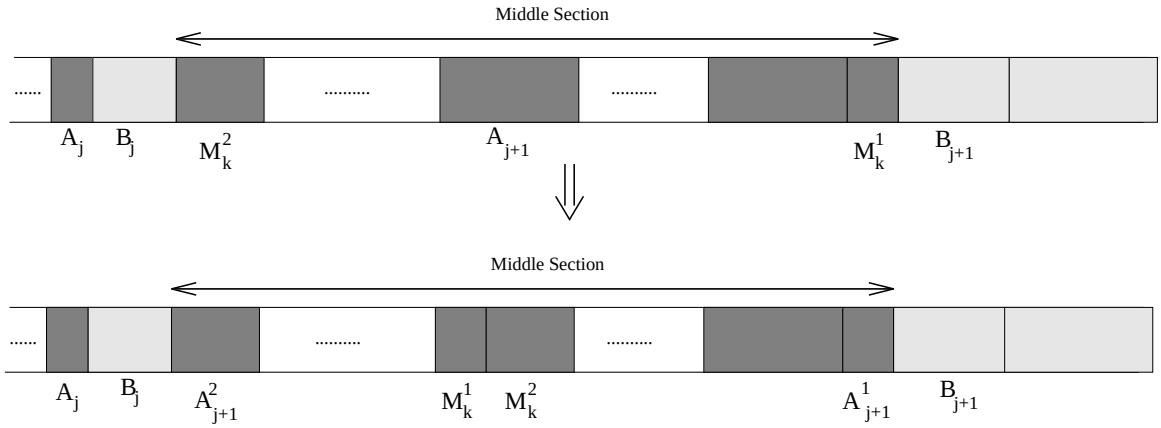


Figure 3: The case where $A_{j+1}$ is not the divided block.

# 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 capability to extract buffers composed of distinct elements. We will show how to do the merging for this case in Section 5. The second problem is caused from the fact that almost all of them 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 minimum first (or last) element. In the case that in 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 Konrod [13] was similar 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 $i^{th}$ block, $1 \leq i \leq k$, by substituting the $i^{th}$ element of the block by an element with key value larger that 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 two sequences. The length of the new sorted sequence is sufficiently large to guarantee that its smallest element is larger that all the elements in the blocks we have to mark. Having created that sequence, we exchange the $i^{th}$ element of the $i^{th}$ block, $1 \leq i \leq k$, with the $i^{th}$ element of the sequence. During the merging, when we locate a block, we swap back the elements to restore it and then to proceed with the merging. Notice that the ordering of the sequence with the "large" elements is not destroyed. Salowe and Steiger [16] 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. (Actually, this is not a problem.

It is something that we must hope to occur.)

## 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 a one-to-one correspondence with the blocks. The $i^{th}$ smallest buffer element corresponds to the $i^{th}$ 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 $i^{th}$ block, we compute the position of the $i^{th}$ smallest element of the buffer. The $i^{th}$ 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 [15].

## 4.3 The optimal algorithm

Having available all the techniques developed in the previous sections, it is easy to describe an optimal in-place and stable merging algorithm.

We start by extracting two buffers, each of $\sqrt{m}$ distinct elements. We assume 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 and the second one to ensure that the blocks of sequence $A$ are merged in the correct order. We do that by using the movement imitation method. After splitting the remaining elements of sequence $A$ into blocks of size $\sqrt{m}$ the array looks like:

$$< \text{buffer\_1} > < \text{buffer\_2} > \; A_{small} A_1 A_2 \ldots < \text{remaining elements of sequence B} >$$

where $A_{small}$ is a non-full block that we ignore in the next step.

We proceed by executing a stable version of the algorithm of Mannila and Ukkonen. After it, the array looks like:

$$< \text{buffer\_1} > < \text{buffer\_2} > \; A_{small} < \text{stably sorted sequence of elements} >$$

We complete the stable merging by distributing from the left and in a stable fashion i) $A_{small}$, ii) buffer_2, and iii) buffer_1.

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

# 5 Merging in the presence of at most $\lambda$ 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 $\lambda < 2\sqrt{m}$ distinct elements. The algorithm is based on Kronrod's algorithm [13] and the ideas of Pardo [15] and it was also presented in [16]. Our presentation is on the lines of [16] with appropriate modifications of the block sizes in order to achieve the desired performance.

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 $\lceil (m+n)/\lambda \rceil$. Now, the array looks like:

$$< \text{buffer} > A_{small} A_1 A_2 \cdots B_1 B_2 \cdots B_{small}$$

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 (from left to right) correspondence 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.

Then, 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 in the sorted sequence of all blocks is location $i$, then after the block sorting that element 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 $\overline{X}$ (the opposite of type $X$) that follows $L[l]$ (Figure 4). 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 set $l$ 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 of 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.

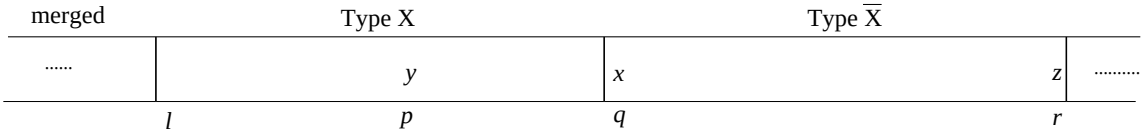


Figure 4: Merging in the case where it is not possible to form the required buffers.

Note that in each block exchange there are involved at least two distinct elements. Since there are $\lambda$ 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. So, 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 looks like:

$$< \text{buffer} > A_{small} < \text{merged full blocks} > B_{small}$$

We complete the stable merging by i) distributing in a stable fashion $A_{small}$ from the left, ii) distributing in a stable fashion $B_{small}$ from the right, and iii) 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-full blocks and the buffer will not change the asymptotic performance of the algorithm.

# 6 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 [2] in 1981. It is amazing that 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 [16] 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 make stable all of the unstable in-place merging algorithms that we are aware of. We were able to use them to make stable the algorithms of Kronrod [13], Dvorak and Durian [6], and Huang and Langston [8], as well as to simplify the algorithms presented by Salowe and Steiger [16]. Note that, any of the two methods presented in Section 4 can be used in making these merging algorithms stable. In the algorithms of this paper, we used the method of movement imitation because we think it is more elegant.

Given the result of this paper, no further improvement in the asymptotic complexity of in-place stable merging can be expected. However, it is interesting to note that internal buffers are used in almost all algorithms that followed their introduction by Kronrod in [13]. It would be nice to know if linear in-place merging is possible without the use of an internal buffer.

# References


[1] S. Carlsson, "SPLITMERGE-A Fast Stable Merging Algorithm", *Information Processing Letters* 22 (1986), 189-192.

[2] K. Dudzinski and A. Dydek, "On a Stable Storage Merging Algorithm", *Information Processing Letters* 12 (1981), 5-8.

[3] S. Dvorak and B. Durian, Towards an Efficient Merging", *Lecture Notes in Computer Science* 133 (1986), 290-298, Springer-Verlag.

[4] S. Dvorak and B. Durian, "Stable Linear Time Sublinear Space Merging", *The Computer Journal* 30 (1987), 372-375.

[5] S. Dvorak and B. Durian, "Merging by Decomposition Revisited", *The Computer Journal* 31 (1988), 553-556.

[6] S. Dvorak and B. Durian, "Unstable Linear Time $O(1)$ Space Merging", *The Computer Journal* 31 (1988), 279-282.

[7] E. Horowitz and S. Sahni, *Fundamentals of Computer Algorithms*, Computer Science Press, Maryland, 1978.

[8] B-C Huang and M.A. Langston, "Practical In-Place Merging", *Communications of the ACM* 31 (1988), 348-352.

[9] B-C Huang and M.A. Langston, "Fast Stable Merging and Sorting in Constant Extra Space", *The Computer Journal* 35 (1992), 643-650.

[10] E. C. Horvath, "Stable Sorting in Asymptotically Optimal Time and Extra Space", *Journal of the ACM* 25 (1978), 177-199.

[11] F.K.Hwang and S. Lin, "A Simple Algorithm for Merging Two Disjoint Linearly Ordered Sets", *SIAM Journal on Computing* 1 (1972), 31-39.

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

[13] M. A. Kronrod, "An Optimal Ordering Algorithm without a Field Operation", *Dokladi Akad. Nauk SSSR* 186 (1969), 1256-1258.

[14] H. Mannila and E. Ukkonen, "A simple Linear-Time Algorithm for In Situ Merging", *Information Processing Letters* 18 (1984), 203-208.

[15] L. T. Pardo, "Stable Sorting and Merging with Optimal Space and Time Bounds", *SIAM Journal on Computing* 6 (1977), 351-372.

[16] J. Salowe and W. Steiger, "Simplified Stable Merging Tasks", *Journal of Algorithms* 8 (1987), 557-571.

[17] A. Symvonis, "Optimal stable Merging", Technical Report 466, Basser Department of Computer Science, University of Sydney, Australia, May 1993.

[18] J. K. Wong, "Some Simple In-Place Merging Algorithms", *Bit* 21 (1982), 157-166.