September, 1985

STABLE UNMERGING IN LINEAR TIME AND CONSTANT SPACE

J.S. Salowe and W.L. Steiger

DCS-TR-162

Department of Computer Science
Hill Center for the Mathematical Sciences
Busch Campus
Rutgers University
New Brunswick, NJ 08903

ABSTRACT

In the Summer 1985 issue of SIGACT news, N. Santoro inquired about the space-time complexity of unmerging. Suppose that two sorted lists A and B, each of size n/2, are merged to produce the list L. The problem is to separate L into A and B in sorted order. An optimal algorithm is presented which unmerges in time O(n) using O(1) extra space, and which is stable.

INTRODUCTION

In [5], N. Santoro introduced the unmerging problem: two sorted lists, A and B, of size n/2 undergo a stable merge to produce the sorted list L. The problem is to separate L into the two constituent sublists A and B, each in sorted order. If we also require A and B to be in the original order, the unmerge is stable. The fundamental question posed by this problem concerns the space-time complexity of unmerging. "Time" takes into account comparisons, movement of data and pointer manipulations; "space" involves any extra storage used beyond the n locations in which the input list L is presented

These questions have an added interest since merging is so well understood. Indeed, M.A. Kronrod [2] developed an algorithm to merge A and B in linear time and using constant extra space. It is unstable because equal valued A's or equal valued B's might become interchanged in the merged list. Luis Trabb Pardo [3] improved Kronrod's algorithm so it merges A and B in a stable fashion. Both algorithms may be shown to be optimal [1] up to the values of constants of proportionality in the space-time bounds.

Santoro [5] sketched a linear time, $O(n^{1/2})$ extra space unmerging algorithm. In [4], Reed, Salowe and Steiger improved this by describing stable merge and unmerge algorithms which, given a positive integer k, require time $O(n \log_k n)$ and extra space O(k); thus, for any $\epsilon > 0$, unmerging may be done in linear time if $O(n^\epsilon)$ extra space is available. In view of the tight bounds on the complexity of merging, it is tempting to seek an unmerging algorithm that somehow reverses the operations in the optimal merge, but which retains its performance characteristics. This paper develops an optimal unmerging algorithm which is analogous to Kronrod's algorithm.

THE ALGORITHM

Formally, the input to the unmerging algorithm is a list L of items. Each item has two fields associated with it, the type and the key. The type of item i, TYPE(i), can be either A or B, and it is used to determine from which of the sublists the item originated. The key of item i, K(i), is the result of evaluating a certain function applied to the i-th item, and it is used to order the list. Item i precedes item j in an ordering if and only if $K(i) \leq K(j)$. Both functions can be evaluated in constant time for any item.

The input list initially consists of n items sorted by key, n/2 items of each type. Assume that the keys of the same type are distinct; later this requirement will be relaxed. The output from the unmerging algorithm is a sorted group of n/2 A's followed by a sorted group of n/2 B's. The unmerging algorithm is divided up into three parts: (i), the blocking phase, (ii), the block rearrangement phase and (iii), the finish-up phase.

The Blocking Phase

Call a contiguous group of items a block and call the first item in a block the mark. Denote the block beginning at index i and ending at index j by L[i:j], and let $k = \lceil \sqrt{n} \rceil$. The general idea of the first phase of the algorithm is to group k items of the same type using space inside the list for temporary storage. Pardo [3] called this space the internal buffer, a concept originated by Kronrod [2]. We use L[1:2k] as the internal buffer. The first k places, L[1:k], are used to collect groups of A's, and the second k places, L[k+1:2k], are used to collect the B's. The action of grouping k consecutive items of the same type into sorted blocks and then arranging the blocks so that items of the same type are in order is refered to as k-blocking by type.

The algorithm *k-block-by-type* performs the task of k-blocking the list L[2k+1:n]. The algorithm iterates over each item in this list. Call the pointer which keeps track of the current item being examined the index. During each iteration of the pointer index, the item at L[index] is examined. Depending on its type, the item contained in L[index] is placed into the next available space of the appropriate buffer, and the item displaced from the buffer is placed in L[index]. If all k locations in the A buffer are filled after an item is placed, the buffer is emptied. The same rule applies to the B buffer. Let the pointer next-block-pos hold the address of the location immediately following the last k-block placed. Initially, next-block-pos is set to 2k+1. The k items in the filled buffer are exchanged - one item at a time - with the corresponding k items beginning at the position referenced by next-block-pos, and next-block-pos is incremented by k. It is important to note that each item placed back into the buffer after this operation is performed actually originated in one of the two buffers, though in general from a different location.

The algorithm repeats this process over the entire list L[2k+1:n], exchanging either the filled A buffer or the filled B buffer with k contiguous, original buffer elements starting at position next-block-pos. After swapping L[n] into the buffer, there may be incomplete blocks left in the buffers, and these items must be placed back into the list. Therefore, the last step in algorithm *k-block-by-type* exchanges the partial block of extra A's followed by the partial block of extra B's in the buffers with the remainder of the list.

After this final step, the items in the list L[2k+1:n] are k-blocked by type except for the two blocks to the far right of the list which are generally of size less than k. In addition, the items that originated in locations L[1:2k] ended up in L[1:2k], though the ordering may have changed. These items have the smallest 2k keys in the list. Each block of size k in L[2k+1:n] is internally sorted, as are the marks of blocks of the same type. Figure 1 gives an example of how the algorithm k-blocks a small list. A formal specification of the algorithm is presented in Figure 2.

[figures 1 and 2 go here]

There are two undefined procedures in the specification, *swap* and *swap-block*. Procedure *swap* uses one space of temporary storage to interchange two items. Procedure *swap-block* uses one space of temporary storage and two pointers to interchange two equal-sized blocks by swapping corresponding items. Therefore its time complexity is linear in the block size.

Algorithm *k-block-by-type* uses O(n) time and O(1) extra space. In terms of space, there are nine extra storage locations used. In terms of time, each item in the list L[2k+1:n] is moved twice, once into the buffer, and once more as its block moves out of the buffer. Associated with each movement is a bounded number of tests and a corresponding movement of buffer elements. Each of these items moves at most k/2 times making the time complexity $O(k^2) = O(n)$.

Block Rearrangement Phase

After being k-blocked by type, the list L[2k+1:n] is a sequence of blocks each containing items of one type. Items in each block increase, as do the marks of successive blocks of the same type. We now reorder the blocks so that the A blocks precede the B blocks and the blocks of the same type are sorted by their marks.

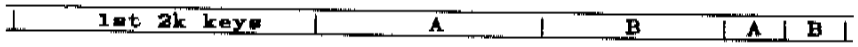
Each block in L[2k+1:n] is of size k, except possibly the last two blocks. This phase operates on the size k blocks. First, it finds the A block with the smallest mark and

exchanges it, if necessary, with the block in the first k spaces of the list L[2k+1:n]. The exchange is a block-swap and takes time O(k) and space O(1). The algorithm then finds the A block with the second smallest mark and exchanges it, if necessary, with the block in the second k spaces of the list, and continues until all A blocks of size k are contiguous and precede all B blocks. The B blocks of size k are exchanged in the same way until they are all increasing. At this point, the blocks of size k in L[2k+1:n] are unmerged.

The Algorithm *group-blocks*, is presented in detail in Figure 3. It requires extra space O(1). The time complexity is O(n) because the block size, k, is $\lceil\sqrt{n}\rceil$. Therefore the $\sqrt{n}$ marks may be sorted in time O(n) and at most $\sqrt{n}$ blocks of size $\sqrt{n}$ need to be swapped.

Finish-up Phase

At this point, the list has the following structure. The smallest 2k elements are in the first 2k positions in the list but are unsorted. Next there is a large block of sorted A items followed by a large block of sorted B items. Finally there may be a small A block followed by a small B block. Three more steps are needed to unmerge the list.



First, the 2k elements at the front of the list must be reordered. As in the previous section, the algorithm finds the smallest A item and exchanges it, if necessary, with the item in L[1]. Next, it finds the second smallest A and swaps it into L[2]. This process continues until all the A's in L[1:2k] form a sorted block. In the same way, a sorted block of B's is formed. These operations use constant extra space and time $O(k^2) = O(n)$.

Second, the large A block must be shifted to the left so that it is adjacent to the initial A block. It is known that block permutation of two blocks can be done in time O(n) and extra space O(1)[1]. Finally, the small A block must be moved to the left so that it is adjacent to the other A's. This amounts to another block permutation.

---

[1] A block U can be reversed in place in time O(|U|) by swapping the pair of end elements, then the pair 1 in from the end, etc. Let U' be U reversed. Adjacent blocks U and V may be permuted by forming U', then V', and then reversing (U'V') (see [3]).

Combining the analysis of all three phases we may state

Theorem 1: A list of size n with distinct keys of each type can be unmerged in time O(n) and extra space O(1).

STABILITY

We now consider the general case where different items may have the same key. It is quite surprising how much more difficult this case is. As with Kronrod's merge [2], the current unmerge algorithm would now be unstable. After the blocking phase L[1:2k] is a permutation of the original elements in L[1:2k]. When they are blocked in phase 3, items of the same type with equal key values may be reversed from their original, premerged order.

A more serious difficulty may arise if after the blocking phase, marks of blocks in L[2k+1:n] have equal keys. If L[2k+1:n] is the following sequence of blocks

$$B_1\ A_1\ A_2\ B_2\ A_3\ B_3,$$

and B_1 and B_2 have marks with equal keys, then the block rearrangement algorithm *group-blocks* would reorganize L[2k+1:n] into

$$A_1\ A_2\ A_3\ B_2\ B_1\ B_3.$$

The rearrangement is unstable because B_1 and B_2 are reversed. In addition if B_2 has at least two distinct keys, the rearranged list is not even sorted; the last item in B_2 has a larger key than that of the mark of B_1. The possibility that block rearrangement may not work without distinct keys seems to have been overlooked by Kronrod [2] and we have to conclude that without modification, his method can fail on lists with some duplicate keys.

We now sketch an analog of *group-blocks* for phase 2. It rearranges the blocks of size k in L[2k+1:n] so that items really are unmerged, and in a stable fashion. The trick is to swap blocks of size k except that marks are not moved. This device permits one to decide if a block is in its final rearranged position or if not, where it should end up. Once blocks are rearranged, their marks may be correctly placed.

Specifically, for a given A block (the j^{th}) in L[2k+1:n], by counting the number α of marks of A blocks that precede block j, the final position may be deduced to be $2k+1+\alpha k$. Similarly if β B blocks in L[2k+1:n] have marks that precede that of a given B block, its final position is $2k+1+n_A k+\beta k$, where n_A is the number of A blocks of size k in L[2k+1:n].

Therefore the algorithm *stable-group-block* counts n_A in time O(k). For each block of size k in L[2k+1:n] compute the final position. If the block is already in place, iterate to the next block. Suppose the block in L[jk+1:(j+1)k] belongs in L[mk+1:(m+1)k]. We swap these blocks leaving the marks L[jk+1] and L[mk+1] alone.

A new block now resides in L[jk+1:(j+1)k] but its mark is in L[mk+1]. If this block is in correct position, iterate to the next block L[(j+1)k+1:(j+2)k]. Otherwise swap it into its correct position but leaving marks untouched. Eventually each of the O(k) blocks of size k in L[2k+1:n] will be rearranged except for their marks. A detailed description of the algorithm appears in Figure 3.

[figure 3 goes here]

Each block moves at most twice, once from its current position, once to its final position. The O(k) block swaps required will need constant extra space and O(n) time. The rearrangement is stable because phase 1 does not reverse items of the same type. Therefore the count of preceding marks of the same type will always indicate the correct final position of a block, even if marks have equal keys.

To complete the rearrangement, marks of the blocks of size k in L[2k+1:n] must now be moved so that they are once again at the beginning of their correct blocks. This may be accomplished via a bubble sort of the at most k-2 marks, A marks interchanging with B marks that precede them. Correctness and stability of this phase are assured because marks of the same type were originally in correct order and are never exchanged in the sorting. The complexity is $O(k^2)$.

If the input list L had its first 2k items with distinct keys, the new block rearrangement algorithm would guarantee a stable unmerge with optimal space and time bounds. We now show how to further modify the algorithm to cope with equal keys in L[1:2k].

If keys can be changed it is easy to stabilize the algorithm. Suppose that in L[1:2k] the minimum key value has been found to be m and the maximum, M. Write δ for the minimum positive difference between successive items in L[1:2k] and let R = M - m. Scan L[1:2k] to detect sequences $u_1,\ldots,u_p$ with equal keys and let $u_{p+1} > u_p$ be the next element. Subtract $R + .5\delta/j$ from the j-th element of this set of sequences. This produces sequences $v_1 < \ldots < v_p < u_{p+1}$ of distinct elements all less than m. Now L[1:2k] has 2k distinct elements so that after k blocking by type, L[1:2k] may be sorted into increasing order in

time $O(k^2)$ and space O(1). Then, to the i-th element of L[1:2k] less than m, add R + $.5\delta/i$. Now, for the first step of the finish-up phase, block the A's in L[1:2k] by bubble sorting: A's move to the left by swapping with B's immediately preceding them or with strictly larger A's immediately preceding them; B's move to the right by swapping with strictly smaller B's immediately following them or with any A that immediately follows. The rest of the finish-up phase of the original algorithm will now give a stable unmerging of L. The time and space bounds have not been changed by this modification.

However when $k = \sqrt{n}$ is large, numerical instability will cause this procedure to break down. Therefore we show how to alter the present algorithm without key modifications to produce a stable unmerge, in much the same way that Pardo [3] modified Kronrod's optimal merge.

The idea is to preprocess L so that the first 2k elements are distinct. Assume that there are enough distinct key values in L to carry out the following steps: If an element in L[1:2k] has the same key value (say x) as the preceding element of the same type, search L[2k+1:n] for the first item of this type with key value greater than x and in a position not equal to a multiple of k. Swap the elements and continue scanning from this point in L[1:2k]. When the next repetition is encountered, resolve it in the same way by a suitable swap, and continue until L[1:2k] contains only distinct elements of each type.

The *k-block-by-type* algorithm will now produce type blocks in L[2k+1:n]. No mark in one of these blocks was swapped from the original buffer region because of the condition that the swap not be to a multiple of k. Therefore the second phase of the current algorithm will produce a stable unmerge of L[2k+1:n] except that there are some original buffer items present.

L[1:2k] may now be sorted in place in time O(n). It will be the distinct-key original buffer elements in order, followed by sorted items that originated in L[2k+1:n]. This second group can be swapped with the original buffer elements now situated in L[2k+1:n], element by element, in time O(n) and constant space. Specifically, find the first element in L[1:2k] with key greater than that of L[2k+1] and swap it with the first element of the same type in L[2k+1:n] whose key is smaller that that of the preceding item, etc. If the two parts of L[1:2k] are now merged in a stable way, phase 3 of the original algorithm will complete a stable unmerge.

The final non-trivial detail arises if there are not enough distinct keys of each type to create a buffer of distinct keys. In this case L must have a fairly rigid structure in which

there are only O(k) sorted blocks having variable size but within which, all items are of the same type. With such a structure, it is easy to *k-block-by-type* without using a buffer. Starting with A's, find a_1 and a_k, the first and k^{th} items. If they have no B's between them they are already k-blocked and we go on to a_{k+1} and a_{2k}.

Otherwise, let $p \leq k$ point to the last A with a B immediately preceding it. Move $a_p a_{p+1} \ldots a_k$ to the left across any B's until they abut a_{p-1}. Those B's never move again. Now let $q < p$ point to the last A with a B immediately preceding it and move $a_q a_{q+1} \ldots a_k$ to the left across any B's until they abut a_{q-1}. Those B's won't move again. Eventually $a_1, \ldots, a_k$ will be consecutive and form a k-block, having moved $j_1 \leq k-1$ groups of A's and the i_1 B's that were between a_1 and a_k, one move for each B. Therefore the cost of forming the first block is $O(kj_1+i_1)$.

These items remain stationary because they precede $a_{k+1}, \ldots, a_{2k}$. Continuing with the second block, we move groups of A's to the left across any intervening B's; the cost is $O(kj_2+i_2)$, i_2 representing the number of B's that were between a_{k+1} and a_{2k}. Since $j_1 + \ldots + j_k = O(k)$ and $i_1 + \ldots + i_k = n/2$, the cost of k-blocking the A's is O(n).

If the procedure is now repeated on the B's, L will have been *k-blocked-by-type* in linear time and constant space. Now the new block rearrangement phase produces a stable unmerge of L and we have

Theorem 2: If L is the stable merge of lists A and B of size n/2 each, it can be unmerged into A|B in the original order using linear time and constant extra space.

FINAL REMARKS

Several ingredients of Kronrod's optimal merge feature in our unmerge algorithm. Starting with L = A|B, (1), the blocks of size k are rearranged so that their marks are in order. This uses at most k *swap-block*s so it takes time O(n) and constant extra space.

Next, (2), adjacent blocks are merged using a buffer. Because each block consists of sorted items of a single type, and because the marks are in order, at most k items that precede block j can have key values larger than that of the mark of block j. Therefore if blocks 2 and 3 are merged using block 1 as an internal buffer, the second k items will be in correct, merged order. Blocks 3 and 4 may then be merged in the same way, etc. The merge is completed when (3), the buffer, L[1:k] is sorted.

It is interesting to observe that in our algorithm the analogue of (2) is *k-block-by-type*. It unmerges items into sorted blocks of a single type. Analogous to (1), our algorithm rearranges blocks. The unmerge may be viewed as a reverse of Kronrod's algorithm in the sense that our versions of (1) and (2) are performed in the opposite order.

Acknowledgement: We thank Chinh Hoang for many helpful comments that improved the paper.

Keywords: merge, unmerge, stable, optimal time, optimal space

References

[1] Horvath, Edward C.
Efficient Minimum Extra Space Stable Sorting.
In *Proceedings of the 6th Annual Symposium of the Theory of Computing (SIGACT 6)*, pages 194-215. Association for Computing machinery, New York, 1974.

[2] Kronrod, M.A.
Optimal Ordering Algorithm Without Operational Field.
Soviet Math. Doklady 10:744-746, 1969.

[3] Pardo, Luis Trabb.
Stable Sorting and Merging with Optimal Space and Time Bounds.
SIAM J. Computing 6:351-372, 1977.

[4] Reed, B., Salowe, J., and Steiger, W.
A Note on Unmerging.
Technical Report, Department of Computer Science, Rutgers University, 1985.

[5] Santoro, N.
On the Unmerging Problem.
SIGACT News 17-1:5, 1985.

INPUT LIST: n = 24, k = 5

$\underline{A_1\ B_1\ A_2\ B_2\ B_3}\ \underline{A_3\ A_4\ B_4\ A_5\ B_5}\ A_6\ A_7\ B_6\ A_8\ B_7\ B_8\ B_9\ A_9\ B_{10}\ B_{11}\ A_{10}\ A_{11}\ B_{12}\ A_{12}$

index

B BUFFER FULL:

$\underline{A_6\ A_7\ A_8\ A_9\ B_3}\ \underline{B_6\ B_7\ B_8\ B_9\ B_{10}}\ A_1\ B_1\ A_3\ A_2\ A_4\ B_4\ A_5\ B_2\ B_5\ B_{11}\ A_{10}\ A_{11}\ B_{12}\ A_{12}$

index

SWAP BLOCK:

$\underline{A_6\ A_7\ A_8\ A_9\ B_3}\ \underline{A_1\ B_1\ A_3\ A_2\ A_4}\ \underline{B_6\ B_7\ B_8\ B_9\ B_{10}}\ B_4\ A_5\ B_2\ B_5\ B_{11}\ A_{10}\ A_{11}\ B_{12}\ A_{12}$

B block

index

A BUFFER FULL:

$\underline{A_6\ A_7\ A_8\ A_9\ A_{10}}\ \underline{B_{11}\ B_1\ A_3\ A_2\ A_4}\ \underline{B_6\ B_7\ B_8\ B_9\ B_{10}}\ B_4\ A_5\ B_2\ B_5\ A_1\ B_3\ A_{11}\ B_{12}\ A_{12}$

B block

index

SWAP BLOCK:

$\underline{B_4\ A_5\ B_2\ B_5\ A_1}\ \underline{B_{11}\ B_1\ A_3\ A_2\ A_4}\ \underline{B_6\ B_7\ B_8\ B_9\ B_{10}}\ \underline{A_6\ A_7\ A_8\ A_9\ A_{10}}\ B_3\ A_{11}\ B_{12}\ A_{12}$

B block A block

index

END OF ALGORITHM *k-block-by-type:*

$B_3\ B_4\ B_2\ B_5\ A_1\ B_1\ A_5\ A_3\ A_2\ A_4\ \underline{B_6\ B_7\ B_8\ B_9\ B_{10}}\ \underline{A_6\ A_7\ A_8\ A_9\ A_{10}}\ \underline{A_{11}\ A_{12}}\ \underline{B_{11}\ B_{12}}$

Internal Buffers B block A block A B

Figure 1: Trace of *k-block-by-type*

An example of Algorithm *k-block-by-type*. In the example, the B buffer is filled first, and it is swapped with the block of k elements beginning at position 2k+1. The A buffer is then completed, and its contents are exchanged with the k items immediately following the B block. Finally, the extra elements are blocked. Note that at the end of the algorithm, the smallest 2k items are in the first 2k positions, but not in their original order.

```
ALGORITHM k-block-by-type (L,n)

VAR
        k, A-buffer-index, B-buffer-index,
        next-block-pos, index

k ← ⌈√n⌉
A-buffer-index ← 1
B-buffer-index ← k + 1
next-block-pos, index ← 2k + 1

WHILE index ≤ n DO

        IF type (L[index]) = A THEN                          /*Place in A buffer

            swap (L[index], L[A-buffer-index])
            A-buffer-index ← A-buffer-index + 1
            IF A-buffer-index = k + 1 THEN
                                                             /*A buffer full.
                                                             /*Place block in list.
                swap-block (L[1:k], L[next-block-pos:next-block-pos + k - 1])
                next-block-pos ← next-block-pos + k
                A-buffer-index ← 1

        ELSE                                                 /*Place in B buffer

            swap (L[index], L[B-buffer-index])
            B-buffer-index ← B-buffer-index + 1
            IF B-buffer-index = 2k + 1 THEN
                                                             /*B buffer full
                                                             /*Place block in list
                swap-block (L[k+1:2k], L[next-block-pos:next-block-pos + k - 1])
                next-block-pos ← next-block-pos + k
                B-buffer-index ← k + 1

        index ← index + 1
END WHILE

IF A-buffer-index > 1 THEN                                   /*extra As
        swap-block (L[1:A-buffer-index - 1],
                    L[next-block-pos:next-block-pos + A-buffer-index - 2])
        next-block-pos ← next-block-pos + A-buffer-index - 1

IF B-buffer-index > k + 1 THEN                               /*extra Bs
        swap-block (L[k+1:B-buffer-index - 1],L[next-block-pos:n])
```

Figure 2: ALGORITHM *k-block-by-type*

```
ALGORITHM group-blocks (L,n)

VARS front, rear, nk, m

k ← ⌈√n⌉
front ← 2k + 1
nk ← # of size k blocks
rear ← (nk+2)k

WHILE there are still A blocks in the list L[front:rear]
        m ← smallest A mark
        swap-block (L[front:front+k-1], L[m:m+k-1])
        front ← front + k

WHILE the list L[front:rear] is not empty          /*only B blocks are left
        m ← smallest mark
        swap-block (L[front:front+k-1], L[m:m+k-1])
        front ← front + k
END


ALGORITHM stable-group-blocks(L,n)

VARS n_A, n_B, aoff, boff, block, from, to, nextto

n_A ← number of A blocks
n_B ← number of B blocks
boff ← kn_A + 2k + 1
aoff ← 2k + 1

FOR block = 1 to n_A DO
        from ← aoff + k(block - 1)
        IF type(L[from]) = A THEN
                to ← aoff + k(number of A marks that precede from)
        ELSE
                to ← boff + k(number of B marks that precede from)

        WHILE from ≠ to DO
                IF type(L[to]) = A THEN
                        nextto ← aoff + k( number of A marks that precede to)
                ELSE
                        nextto ← boff + k(number of B marks that precede to)

                swap-block (L[from+1:from+k-1],L[to+1:to+k-1])
                to ← nextto
        END WHILE
END FOR
```

Figure 3: ALGORITHMS *group-blocks* and *stable-group-blocks*