Algorithms and Data Structures

Daniel Sleator Editor

Practical In-Place Merging

BING-CHAO HUANG and MICHAEL A. LANGSTON

ABSTRACT: We present a novel, yet straightforward linear-time algorithm for merging two sorted lists in a fixed amount of additional space. Constant of proportionality estimates and empirical testing reveal that this procedure is reasonably competitive with merge routines free to squander unbounded additional memory, making it particularly attractive whenever space is a critical resource.

1. INTRODUCTION

Merging is a fundamental operation in computer programming. Given two sorted lists whose lengths sum to n, the obvious methods for merging them in $O(n)$ steps require a linear amount of extra memory as well. On the other hand, it is easy to merge *in place* using only a constant amount of additional space by heap-sorting, but at a cost of $O(n \log n)$ time.

Happily, an algorithm by Kronrod [8] solves this dilemma by merging in *both* linear time and constant extra space. Kronrod's seminal notions of "block rearrangements" and "internal buffering," which employ $O(\sqrt{n})$ blocks each of size $O(\sqrt{n})$, have spawned continued study of this problem, including the work of Horvath [3], Mannila and Ukkonen [9] and Pardo [10]. Unfortunately, all of these schemes have been fairly complicated. More importantly, none has been shown to be very effective in practice, primarily because of their large linear-time constants of proportionalities.

Herein we present a fast and surprisingly simple algorithm which merges in linear time and constant extra space. We also use $O(\sqrt{n})$ blocks, each of size $O(\sqrt{n})$. In general, this approach allows the user to employ one block as an internal buffer to aid in rearranging or otherwise manipulating the other blocks in constant extra space. Since only the contents of the buffer and the relative order of the blocks need ever be out of sequence, linear time is sufficient to sort both the buffer and the blocks (each sort involves $O(\sqrt{n})$ keys) and thereby complete the merge. As we shall show, two major factors which make our scheme so much more straightforward and practical than previously reported attempts to solve this problem are these: 1) we rearrange blocks before we initiate a merging phase and 2) we efficiently pass the internal buffer across the list so as to minimize unnecessary record movement.

Given today's low cost of computer memory components, the greatest practical significance of our result may well lie in the processing of large external files. For example, notice that in-place merging can be viewed as an efficient means for increasing the effective size of internal memory when merging or merge-sorting external files. This may result in a substantial reduction in the overall elapsed time for file processing whenever the extra space can be utilized for more buffers to increase parallelism or larger buffers to reduce the number of I/O transfers needed.

In the next section, we present an overview of the main ideas behind this new $O(n)$ time and $O(1)$ space merging technique, with simplifying assumptions made on the input to facilitate discussion. Section 3, which may be skipped on a first reading, contains the $O(\sqrt{n})$ time and $O(1)$ space implementation details necessary to handle arbitrary inputs. In Section 4, we formally and empirically evaluate this new merge. We demonstrate that, for large n, its average run time exceeds that of a standard merge, free to exploit $O(n)$ temporary extra memory cells, by less than a factor of two. The final section of this presentation consists of a collection of observations and remarks pertinent to this topic.

This research is supported in part by the National Science Foundation under grants ECS-8403859 and MIP-8603879. A preliminary version of a paper describing the work reported here was presented at the ACM-IEEE/CS Fall Joint Computer Conference held at Dallas, Texas, in October 1987.

2. AN OVERVIEW OF THE MAIN ALGORITHM

Let L denote a list of n records containing two sublists to be merged, each in nondecreasing order. For the sake of discussion, let us suppose that n is a perfect square. Figure 1a illustrates such a list with $n = 25$. Only record keys are listed, denoted by capital letters. Subscripts are added to keep track of duplicate keys as the algorithm progresses. Let us also assume that we have been able to permute the elements of L so that $\sqrt{n}$ largest-keyed records are at the front of the list (their relative order there is immaterial), followed by the remainders of the two sublists, each of which we assume, for simplicity, now contains an integral multiple of $\sqrt{n}$ records in nondecreasing order. Figure 1b shows our example list in this format.

We now view L as a series of $\sqrt{n}$ blocks, each of size $\sqrt{n}$. We will use the leading block as an internal buffer to aid in the merge. Our first step is to sort the $\sqrt{n} - 1$ rightmost blocks so that their tails (rightmost elements) form a nondecreasing key sequence. (A good choice for the sort is the straight selection method as described in Knuth [7]. This technique is simple and will in this setting require only $O(n)$ key comparisons and record exchanges.) Records within a block retain their original relative order. See Figure 1c.

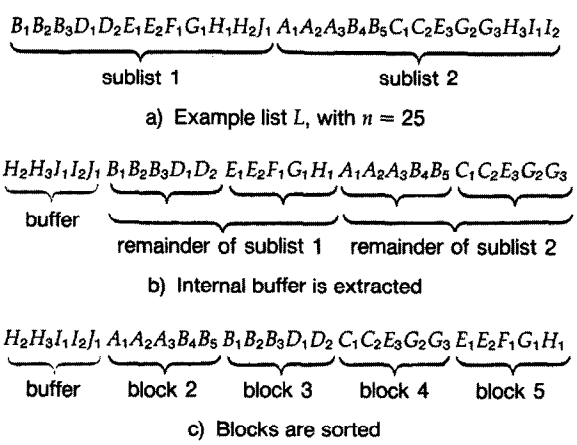

FIGURE 1. Initial Rearrangement of Blocks

Next, we locate two series of elements to be merged. The first series begins with the head (leftmost element) of block 2 and terminates with the tail of block i, $i \geq 2$, where block i is the first block such that tail $(i) >$ head $(i + 1)$. The second series consists solely of the elements of block $i + 1$. See Figure 2a, where $i = 3$. We now use the buffer to merge these two series. That is, we repeatedly compare the leftmost unmerged element in the first series to the leftmost unmerged element in the second, swapping the smaller with the leftmost buffer element. Ties are broken in favor of the first series. In general, the buffer may be broken into two pieces as we merge. See Figure 2b. We halt this process when the tail of block i has been moved to its final position. At this point, the buffer must be in one piece, although not on a block boundary. Block $i + 1$ must contain one or more unmerged elements (as a result of the first step in which blocks were sorted by their tails). See Figure 2c.

We now locate the next two series of elements to be merged. This time, the first begins with the leftmost unmerged element of block $i + 1$ and terminates as before for some $j \geq i$. The second consists solely of the elements of block $j + 1$. See Figure 3a, where $j = 4$. We resume the merge until the tail of block j has been moved. See Figure 3b.

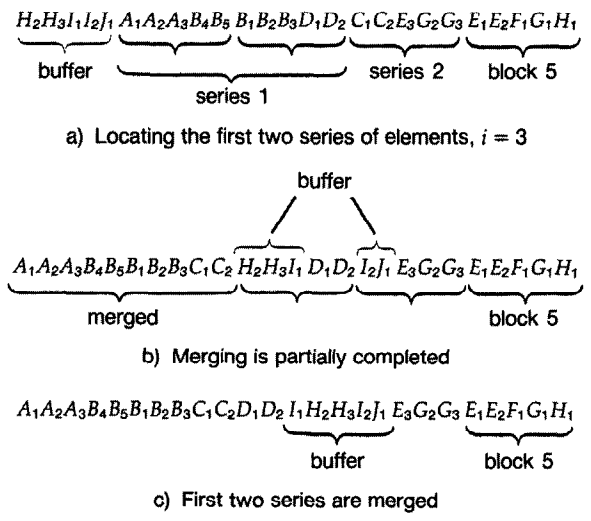

FIGURE 2. Merging the First Two Series of Elements

We continue this process of locating series of elements and merging them until we reach a point where only one such series exists, which we merely shift left, leaving the buffer in the last block. See Figure 3c. A sort of the buffer completes the merge of L. See Figure 3d.

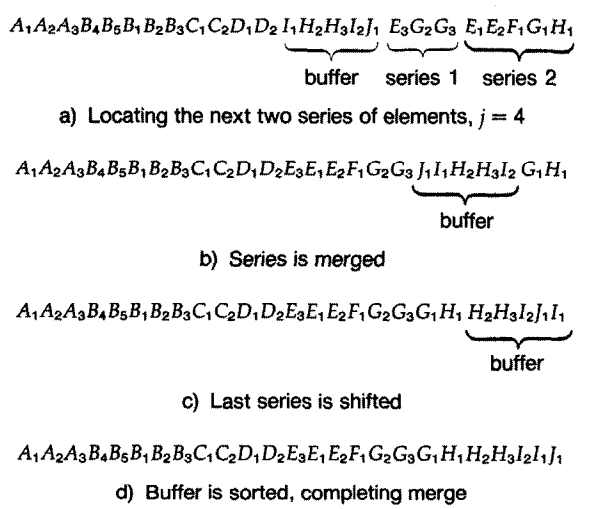

FIGURE 3. Completing the Merge of L

$O(1)$ space suffices for this procedure, since the buffer was internal to the list, and since only a handful of additional pointers and counters are necessary. $O(n)$ time suffices as well, since the block sorting, the series merging and the buffer sorting each require at most linear time.

This completes our overview of the central features of this fast merging strategy, illustrating the method we use to rearrange L by blocks and to merge the elements of these blocks with the aid of the internal buffer. In the next section, we provide the interested reader with $O(\sqrt{n})$ time implementation details for dealing with lists and sublists of arbitrary sizes and for efficiently handling the initial extraction of the internal buffer.

3. IMPLEMENTATION DETAILS

For the special case in which one of the sublists to be merged has $l < \sqrt{n}$ elements, it is a simple matter to perform the merge efficiently without the general procedure outlined in the last section. For example, supposing the shorter sublist begins in location 1, we could employ the "block merge forward" scheme suggested in [10]. We first search the longer sublist for an insertion point for record 1. That is, we find p such that key(p) < key(1) ≤ key (p + 1). Then elements indexed from 1 through p can be circularly shifted $p - l$ places to the right, so that the rightmost element of the shorter sublist occupies position p. (Such a shift is often termed a "rotation," and can be efficiently achieved with three sublist reversals.) The first element of the shorter sublist and all elements to its left are now in their final position. We complete the merge by iterating this operation over the remainder of the list until one of the sublists is exhausted. Linear time is sufficient for this method since we have insured that elements of the shorter sublist are moved no more than $\sqrt{n}$ times, and elements of the longer sublist are moved only once. (If the larger sublist occurs first, we can, of course, use an analogous "block merge backwards" scheme.)

In what follows, therefore, we assume that each sublist has at least $s = \lfloor\sqrt{n}\rfloor$ elements. Note that $(s + 1)^2 > n$ guarantees that the total number of s-sized blocks is no more than $s + 2$. We begin by comparing the right ends of the sublists, identifying s largest-keyed records which are to become the internal buffer. Let A and B, with respective sizes s_1 and s_2, denote these two portions of L, where $s_1 + s_2 = s$. Let C denote $s - s_1 = s_2$ elements immediately to the left of A. Let D denote the shortest (possibly empty) collection of records to the immediate left of B which, if D and B were deleted from the second sublist, would leave that sublist with an integral multiple of s records. Thus we can view L as a list of s-sized blocks, except possibly for the first block of size t_1, where $0 < t_1 \leq s$, and for the last block of size t_2, where $0 < t_2 < 2s$. See Figure 4a.

We now show how to handle the rightmost block which may have an unusual size. We swap B and C, bringing the buffer into one block. Then we use the buffer to merge C and D, giving block E of size t_2. See Figure 4b. If both C and D were nonempty, then the tail of E comes from either C or D, and is as large as any nonbuffer element in L. Thus E is in its correct position and does not need to be considered later when nonbuffer blocks are sorted by their tails. If C and D (or both) were empty, then although E's tail may be smaller than one or more nonbuffer elements, it is easy to see that E still need not be moved before the main algorithm commences. Therefore, in any event, E remains

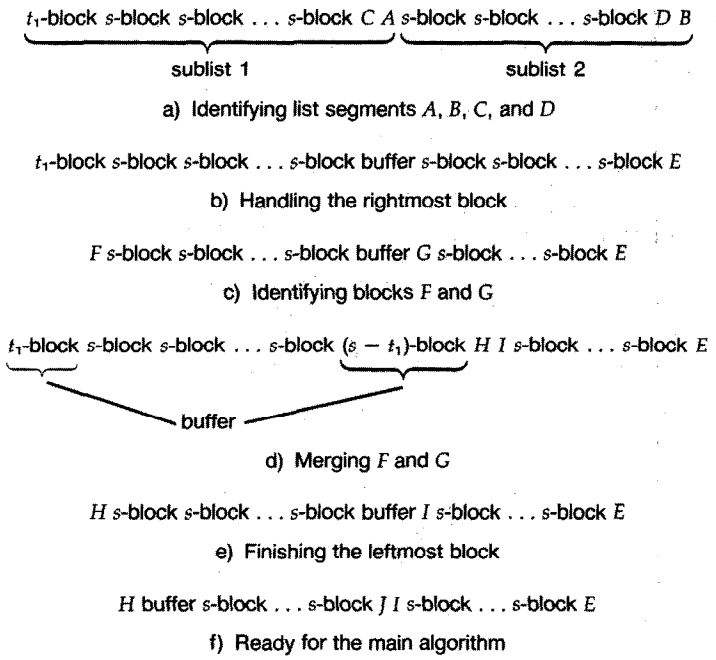

FIGURE 4. Implementation Details

where it is and its size can cause no difficulty for the merging which begins after these blocks are sorted.

Next, we take care of the leftmost block, but only if it has an unusual size (that is, if $t_1 < s$). Let F denote this block. Let G denote the first block of the second sublist. See Figure 4c. We use the rightmost t_1 buffer elements to merge F and G, giving H and I. See Figure 4d. By virtue of this merge of the smallest blocks in L, we can now move H to its final position by swapping it with the t_1 buffer elements there. This also restores the buffer. See Figure 4e.

We now swap the buffer with J, the first s-sized block of L, which is either the first block of L or the one following H. See Figure 4f. We are thus ready to perform the main algorithm as previously described. (Notice that blocks J, I and E might have to be moved to provide the two sorted sublists that we assumed at this point, for the ease of discussion, in Section 2. Naturally, we do not actually move them, since E is to remain where it is and since I and J are next sorted by their tails anyway.)

Thus we have outlined an efficient method for handling arbitrary list and sublist sizes and for extracting the buffer. These implementation details are of time complexity $O(\sqrt{n})$, thereby dominated by the linear time required for the main algorithm.

4. ANALYSIS AND COMPUTATIONAL EXPERIENCE

4.1 Constant of Proportionality Estimates

Key comparisons and record exchanges are generally regarded as by far the most time consuming merging operations used. Both require storage-to-storage instructions for most architectures. Moreover, it is possible to count them independently from the code of any particular implementation. Therefore, their total gives a useful estimate of the size of the linear-time constant of proportionality for the merge routine we have described.

We now proceed to derive a worst-case upper bound on their sum. (We focus only on the main algorithm, since extracting the buffer and other algorithmic details described in Section 3 need only $O(\sqrt{n})$ time. Similarly, the final sort of the buffer can be accomplished in-place with a heap sort in $O(\sqrt{n} \log n)$ time.) Recall that $s = \lfloor\sqrt{n}\rfloor$ and that there are no more than $s + 2$ blocks. Using a selection sort to sort no more than $s + 1$ blocks of size s requires at most $s(s + 1)/2$ comparisons and at most s^2 exchanges [7], summing to $1.5s^2 + .5s \leq 1.5n + .5s$. When merging series of elements, the number of comparisons is bounded above by the number of exchanges. Exchanges occur only between buffer and nonbuffer elements; nonbuffer elements are moved only once. Thus the comparison plus exchange total for the merging phase is at most $2(n - s)$.

Hence, we are guaranteed a worst-case grand total of something less than $3.5n$. This is a very reasonable quantity indeed, especially since *any* merge must, in the worst case, use at least $n - 1$ key comparisons and n record moves. (Since L is not in a completely random form when the blocks are sorted, we can take advantage of its partial order to speed up this step, reducing our figure from $3.5n$ down to $3.125n$. We omit the details from this presentation since they are cumbersome and not particularly interesting in their own right. Curious readers are welcome to request additional information from the authors.)

4.2 Observed Behavior

To provide a more accurate estimate of the practical potential of our new merge procedure, we next describe the results of a series of experiments aimed at comparing its average performance to that of a common, heavily-used merge free to take advantage of unbounded temporary extra storage. Suppose L contains two sublists, the first of length l_1, the second of length l_2. This fast merge works as follows. If $l_1 \leq l_2$, then it moves the first sublist to temporary storage and merges (forward) into L's original position. Otherwise, it moves the second sublist to temporary storage and merges (backward) into L's original position.

TABLE I. Observed Behavior of In-Place Strategy, A, Relative to Memory-Dependent Procedure, B.

List Size, n	Algorithm A's Average Run-Time, AVG_A, in Milliseconds	Algorithm B's Average Run-Time, AVG_B, in Milliseconds	AVG_A/AVG_B
50	0.1991	0.0602	3.307
100	0.3620	0.1205	3.004
500	1.583	0.6259	2.529
1,000	2.806	1.301	2.157
5,000	13.44	6.734	1.996
10,000	26.32	13.35	1.972
50,000	127.8	66.8	1.913
100,000	238.8	131.4	1.817
500,000	1152	651.2	1.769
1,000,000	2287	1309	1.747

There are several reasons for conducting an empirical comparison of these two merge algorithms. Optimistically, we observe that, on the average, our merge should perform even better than the worst-case key comparison plus record exchange total derived in the last section would suggest. Pessimistically, however, we note that the lower-order (nonlinear) time complexity terms may be of significance, especially for small n. Furthermore, the space-squandering procedure can accomplish the merge without true record exchanges, since it only needs to move records one at a time, not swap pairs of them, as it executes.

We wrote both merge routines in Pascal for an IBM 3090-200, and timed them over large collections of lists, with relative sublist sizes and key values uniformly chosen at random. Table I summarizes the behavior we observed. Let A denote our in-place strategy. Let B denote the fast procedure we have just described which requires $O(n)$ extra space. Average run-times reported

(AVG_A and AVG_B) reflect, for each list size n, the mean execution time each merge incurred during one hundred experiments. (These figures, of course, account for the time spent merging only; they do not include the great amount of time devoted to overhead operations such as generating and sorting sublists.)

Quite naturally, other factors such as record length, computer architecture, percentage of duplicates in a list and so on can affect algorithmic behavior. Therefore, the ratios we compute can only be viewed as estimates. We conclude that the expected penalty for merging large lists in place is less than a doubling of program execution times. This penalty can be insignificant, especially in applications such as external file processing, in which overall run-times are dictated by the ways in which main memory can be used to assist in the effective minimization and concurrency of I/O operations. We emphasize that we have implemented algorithm A just as we have described it herein, refraining from the temptation to "fine tune" it to make it faster. We suggest that more streamlined implementations can merge at a considerably accelerated pace.

5. REMARKS

We have presented a relatively straightforward and efficient merging procedure which simultaneously optimizes both time and space (to within a constant factor). For the sake of speed and simplicity, our algorithm is not stable. That is, records with equal keys are not assured of retaining their original relative order (refer back to Figure 3d). We have, however, been able to employ similar methods to devise a stable in-place merge for this problem [4]. Unfortunately, this algorithm is more complicated and runs somewhat slower. Even so, it is much faster than the fastest previously published stable in-place optimal time and space merge [10]. We have also been able to develop a stable in-place $O(n \log n)$ sort [4], which bypasses the obvious merge-sort implementation, and is several times faster than any reported to date.

This general line of research has the potential to change the way practitioners *think* about performing everyday file processing. For example, we have found that the merge routine we have presented here is straightforward enough to be implemented by students in an upper-divisional undergraduate computer algorithms class (at Washington State University) with only a preliminary version of this paper to use as a guide. In fact, we believe that this would be the case for our optimal time and space duplicate-key extraction, selection, set and multiset operations as well [5, 6].

We observe that obvious, traditional methods for merging in linear time require that $n/2$ additional memory cells be available. Even though internal storage is often rather cavalierly viewed as an inexpensive resource, it may well be that in many environments our algorithm's modest run-time premium is more than offset by its avoidance of a 50 percent storage overhead penalty. In particular, this may be the case when managing critical resources such as cache memory or other relatively small, high-speed memory components. Not only is there the immediate space savings, but the strategy we employ to pass the buffer across the list keeps active blocks near one another, thereby preserving locality of reference and potentially reducing the likelihood of paging problems. (Of course, virtual memory behavior depends on several other factors as well, and can sometimes be surprising. See, for example, Alanko, Erkio and Haikala [1]).

Finally, we note that other researchers are beginning to report some initial progress related to this general topic (see, for example, Dvorak and Durian [2]). As block rearrangement strategies become better known, we believe further investigations will help to determine their practical potential for sorting, merging and related file-processing operations.

Acknowledgements. The authors wish to thank Donald Knuth for suggesting this general line of investigation, and Michael Morford for his assistance in performing the run-time experiments whose results we have reported. We also thank the anonymous referees whose comments helped to improve the readability of this paper.

REFERENCES

1. Alanko, T.O., Erkio, H.H., and Haikala, I.J. Virtual memory behavior of some sorting algorithms. *IEEE Trans. on Software Engr. 10* (1984), 422–431.
2. Dvorak, S., and Durian, B. Towards an efficient merging. *Lecture Notes in Computer Science* 233 (1986), 290–298.
3. Horvath, E.C. Stable sorting in asymptotically optimal time and extra space. *J. of the ACM* 25 (1978), 177–199.
4. Huang, B.C., and Langston, M.A. Fast stable merging and sorting in constant extra space, Computer Science Department Technical Report CS-87-170, Washington State University, 1987.
5. Huang, B.C., and Langston, M.A. Stable duplicate-key extraction with optimal time and space bounds. *Acta Informatica*, to appear.
6. Huang, B.C., and Langston, M.A. Stable set and multiset operations in optimal time and space, Computer Science Department Technical Report CS-87-166, Washington State University, 1987.
7. Knuth, D.E. *The Art of Computer Programming, Vol. 3: Sorting and Searching.* Addison-Wesley, Reading, Mass., 1973.
8. Kronrod, M.A. An optimal ordering algorithm without a field of operation (in Russian), *Dok. Akad. Nauk SSSR* 186 (1969), 1256–1258.
9. Mannila, H., and Ukkonen, E. A simple linear-time algorithm for in situ merging. *Information Processing Letters* 18 (1984), 203–208.
10. Pardo, L.T. Stable sorting and merging with optimal space and time bounds. *SIAM J. Comput.* 6 (1977), 351–372.

CR Categories and Subject Descriptors: D.4.3 **[Operating Systems]**: File Systems Management—*Maintenance*; F.2.2 **[Analysis of Algorithms and Problem Complexity]**: Nonnumerical Algorithms and Problems—*Sorting and Searching*; H.2.4 **[Database Management]**: Systems—*Transaction Processing*

General Terms: Algorithms, Theory

Additional Key Words and Phrases: Block rearrangements, file processing, internal buffering, in-place algorithms, optimal time and space merging

Received 8/86; revised 5/87; accepted 6/87

Authors' Present Address: Department of Computer Science, Washington State University, Pullman, WA 99164-1210