BIT 30 (1990), 220–234

STABLE IN SITU SORTING AND MINIMUM DATA MOVEMENT

J. IAN MUNRO[1], VENKATESH RAMAN[1] and JEFFREY S. SALOWE[2]

[1] *Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada N2L 3G1*

[2] *Department of Computer Science, University of Virginia, Charlottesville, Virginia, USA 22903*

Abstract.

In this paper, we describe an algorithm to stably sort an array of n elements using only a linear number of data movements and constant extra space, albeit in quadratic time. It was not known previously whether such an algorithm existed. When the input contains only a constant number of distinct values, we present a sequence of *in situ* stable sorting algorithms making $O(n \lg^{(k+1)} n + kn)$ comparisons ($\lg^{(k)}$ means lg iterated k times and $\lg^*$ the number of times the logarithm must be taken to give a result ≤ 0) and $O(kn)$ data movements for any fixed value k, culminating in one that makes $O(n \lg^* n)$ comparisons and data movements. Stable versions of quicksort follow from these algorithms.

CR categories: E.5, F.2.2.

1. Introduction and motivation.

Sorting is a fundamental problem in computation. Indeed Knuth [5] has suggested that it can be viewed as a paradigm for most computing problems. Initial research on sorting centred on devising fast algorithms and, in particular, minimizing the number of comparisons needed. Once optimal $O(n \lg n)$ time algorithms were available, research shifted to other issues such as in-place sorting, stable sorting (i.e. keeping equal elements in their initial order) [10] and more recently to sorting in as few data movements as possible [9].

In this paper, we confront all of these issues simultaneously. We ask whether one can stably sort n elements using a constant number of pointers, counters and data locations (i.e. $O(\lg n)$ bits plus an extra data location or two) and only $O(n)$ data movements. Initially, it was not clear (at least to us) that it is possible to perform this task, regardless of the number of comparisons. Our main observation may in fact be

* Research supported by Natural Sciences and Engineering Research Council of Canada grant No.A-8237 and the Information Technology Research Centre of Ontario.

** Supported in part by a Research Initiation Grant from the Virginia Engineering Foundation.

Received December 1988. Revised July 1989.

that it is possible to stably sort n elements in situ with $O(n)$ data movements. We present an algorithm that makes a quadratic number of comparisons for the general case and several much faster techniques for the case in which the number of distinct elements is bounded by a constant. Stable versions of quicksort ([4], [2]) follow as a consequence of the latter.

The problem was raised to gracefully correct a flaw in the well known in-place merging algorithm of Kronrod [7]. In the block sorting phase of Kronrod's algorithm, approximately $\sqrt{n}$ blocks, each of size roughly $\sqrt{n}$, are sorted. Each data movement in the block sorting phase actually moves an entire block, so it costs $O(\sqrt{n})$. To maintain the linear time bound for Kronrod's algorithm, the sorting algorithm used in the block sorting phase can make a quadratic number of comparisons ($O((\sqrt{n})^2) = O(n)$), but only a linear number of data movements. Furthermore, as pointed out in [11] and [12], if the block sorting algorithm is not stable, Kronrod's algorithm may fail to merge the lists if some key is duplicated many times.

The term *in situ*, or in-place, is generally used to describe a technique that uses only a constant number of extra data locations and a small number of pointers or indices. Baase [1] insists that this number of indices is a constant, although many would consider quicksort [4], with the $O(\lg n)$ indices required for recursive calls, to be an in situ sorting algorithm. We will use the term primarily in Baase's sense, but avoid confusion over this subtle issue by quoting our results explicitly in terms of the amount of extra storage required. There are two common in situ unstable sorting algorithms that make a linear number of data movements: selection sort [5] and the in situ permutation sort which follows from [6]. Selection sort considers each location i ($i = 1..n$) in turn, and finds the element that should go in that spot by scanning locations i through n to find the smallest. This element is then interchanged with the one in location i. In contrast, the in situ permutation sort considers each value, finds the correct position for it, and continues by tracing the cycle structure of the permutation required to sort. Knuth [6] gives an algorithm which permutes an array in situ according to a given permutation. (In [6], Knuth attributes the procedure to J. C. Gower and cites the additional work of Macleod [8] and Windley [14].) The permutation required to sort can be easily computed on the fly, when all elements are distinct. Figure 1a presents pseudocode for this method. (# denotes the cardinality of a set). This version fails if two copies of the same value are encountered, as it will continue to interchange them indefinitely. The problem is easily overcome (Figure 1b) by placing an element at the end of the sequence of its sibling values that are already properly positioned. Selection sort and both versions of in situ permutation sort make a quadratic number of comparisons. Recently Munro and Raman [9] have given an algorithm to sort n distinct elements using linear data movements. The method, though unstable, is in situ and makes $O(n \lg n)$ comparisons on the average.

All of these algorithms are unstable as equal-valued items may change their relative positions in a haphazard fashion. It seems difficult to construct a stable version of selection sort without drastic modifications. The technique we will

```
insitu_permsort_nodups(L, n)
for k = 1 to n
  repeat
    l ← #{j : L[j] < L[k], k + 1 ≤ j ≤ n}
    swap(L[k], L[k + l])
  until l = 0
endfor
```

Figure 1a. In Situ Permutation Sort assuming all elements are distinct.

```
insitu_permsort(L, n)
for k = 1 to n
  repeat
    l ← #{j : L[j] < L[k], k + 1 ≤ j ≤ n}
    if l ≠ 0 then
      while (L[k] = L[k + l]) do l = l + 1
      swap(L[k], L[k + l])
    endif
  until l = 0
endfor
```

Figure 1b. In Situ Permutation Sort permitting repeated values

present in Section 4 can be viewed as a stable version of the in situ permutation sort. We will observe that a procedure similar to the in situ permutation can make the sort stable, if we first ensure that any element occurring in the interval of its sibling values lies in its correct final location of the stably rearranged list. It is this invariant we establish and exploit to derive the general result in the Section 4. It turns out that establishing this invariant is equivalent to stably sorting an array containing only two distinct keys. In Section 3 we give algorithms for this special case. Apart from achieving the run time needed for our general result, we give a spectrum of faster techniques for sorting an array consisting of any fixed constant number of distinct keys. By applying the algorithms in this section to the partition step of quicksort, we obtain stable versions of quicksort. The results in Section 3 use a version of the in situ permutation sort that is made stable by using n bits of extra memory. We begin the development of our algorithm in the next section, with a description of that stable, though "non-insitu", version of the in situ permutation sort.

Throughout this paper all logarithms are to base 2, $\lg^{(k)}$ means lg iterated k times, and lg* denotes the number of times the logarithm may be taken before the quantity becomes at most 0.

2. A stable permutation sort.

A scan of the pseudocode for the in situ permutation sort reveals two causes of instability. The most obvious is that as an element is moved into its correct position, it is interchanged with another, the initial position of which is immediately lost. This is easily corrected by retaining a single index. A more serious problem is that when an element is discovered to be in a location appropriate for it or one of its sibling values, it is not clear whether this is a chance occurrence or whether it was moved there on a previously traversed cycle. While the latter suggests the element is in its proper final position, the former does not. Given this difficulty, we begin the development of our methods with an approach to sort n elements stably using a linear number of data movements and quadratic time, but requiring n bits of extra storage.

LEMMA 1. *An array of n elements can be stably sorted in quadratic time using $O(n)$ data movements, constant number of indices and n extra bits.*

PROOF: Let $L[1:n]$ be the input array and $M[1:n]$ be an array containing n bits of extra space. If $L[i]$ is known to be in its correct final location, then $M[i]$ is set to 1 and we say the element is marked. Otherwise $M[i]$ is 0 and $L[i]$ is said to be unmarked. The vector M is initialized to 0.

Our basic approach is to scan the array until we find an unmarked element ($M[i] = 0$). As $L[i]$ is not known to be in its final position, the elements in the cycle of $L[i]$ are permuted into their proper locations and the scan continues.

There are two issues of concern: determining the permutation and actually performing it as it is being determined. First consider the problem of determining where unmarked element x at location i (i.e. $L[i] = x$) should be moved, given that the invariant on M not only holds, but that all elements not in their final positions are in their initial positions. Let l_x denote the number of locations containing values less than x, and e_x the number equal to x. Then $L[i]$ should be moved to some position in the interval $L[l_x + 1 : l_x + e_x]$. We call this interval I_x. Consider those elements that equal x and currently precede the location i, but are not known to be in their proper places. Call this number of elements q_i. These elements should precede $L[i]$ in I_x when the stable sort is completed. As the marked elements in I_x are already in their final positions, these q_i elements occupy the first q_i unmarked elements of I_x. $L[i]$, then, is to be moved into the $(q_i + 1)$st unmarked location in I_x.

A scan of L suffices to determine l_x, e_x and q_i. A rescan of part of I_x suffices to find the proper destination of the value x. The function *destination*1(i, k) in Figure 2 performs this task. The problem of permuting the elements in place necessitates two parameters of this function. We address this issue next.

The scheme given above can easily be used to compute the locations on a cycle of the permutation to be performed. Some care, however, must be taken in actually moving the elements. We seem stuck with either moving each element (as displaced)

```
stable_permsort(L, M, n)
for k = 1 to n
  if M[k] = 0 then
    i ← k
    repeat
      ito ← destination1(i,k)
      swap(L[ito], L[k])
      M[ito] ← 1
      i ← ito
    until (i = k)
  endif
endfor
destination1(i,k)
  l ← #{j : L[j] < L[k], 1 ≤ j ≤ n}
  q ← #{j : L[j] = L[k], 1 ≤ j ≤ i − 1, j ≠ k, M[j] = 0}
  destination1 ← j such that M[j] = 0 and #{s : M[s] = 0, l + 1 ≤ s < j} = q
```

Figure 2. A stable permutation sort with n bits of extra storage

to a temporary location, and so effectively leaving a null value in the table, or swapping elements, and so having one element in a location which is neither its initial nor its final position. As in the pseudocode of Figure 1 we somewhat arbitrarily opt for the latter approach. The invariant we maintain, on each iteration of the repeat loop in the *stable_permsort* procedure of Figure 2, is that all elements are either marked ($M[j] = 1$) as in their final locations or unmarked ($M[j] = 0$) indicating they are in their initial positions, except for element at position k. Position k is the lowest numbered location in its cycle, and is used as a temporary location.

Initially, of course, $i = k$; the final location of $L[i]$ is found and the value swapped to the proper location. This leads to the general step in which location k is unmarked and contains the element originally in position i. The only twist caused in *destination1* by i and k differing is that location k must be ignored in the scan to determine q.

The main loop continues by placing the element initially in location i (i.e. $L[k]$) in its proper place $L[ito]$, and marking it. The procedure continues till the cycle is completed before continuing the scan for the first location in another cycle.

It is clear that the algorithm makes a linear number of data movements as each swap places one additional item in its correct location. It is equally clear that the algorithm makes a quadratic number of comparisons, as each final address computation takes a linear number of comparisons. Finally, the sort is stable as final addresses are calculated so that equal valued keys are in the same relative order after the sort as at the beginning of the sort. ■

3. Stably sorting two distinct keys.

The basic building block of this section is the following lemma in which we assume each key value of L is one of two values (A or B). It is adequate for the general result proven in the next section, and is the basis of the stronger results on stably sorting a fixed number of distinct key values proven later in this section.

LEMMA 2. *An array containing n elements, but only two distinct key values, can be stably sorted in $O(n^{3/2})$ time using $O(n)$ data movements and $O(1)$ indices.*

PROOF: Our scheme has two cases, depending on the number of A's and B's. If there are fewer than $\lceil\sqrt{n}\rceil$ items of either key value, then Step 1, below, can be adapted to sort L in linear time. Therefore, we concentrate on the case where both A and B appear at least $\lceil\sqrt{n}\rceil$ times. The five phase algorithm is described below. In the first phase, we form an *internal buffer* to encode an adequate portion of the array M of the preceding section. This internal buffer consists of $\lceil\sqrt{n}\rceil$ A's and $\lceil\sqrt{n}\rceil$ B's. These $2\lceil\sqrt{n}\rceil$ items will be used to encode $\lceil\sqrt{n}\rceil$ bits of M. The first phase makes the first $\lceil\sqrt{n}\rceil$ items A's and the second $\lceil\sqrt{n}\rceil$ B's. In the second phase, the remainder of L is divided into blocks of size $\lceil\sqrt{n}\rceil$, and each block is sorted using the algorithm of Lemma 1. The third phase prepares L for rearrangement by regrouping items with the same key into segments of size $\lceil\sqrt{n}\rceil$, and the fourth phase rearranges the segments using Lemma 1. The final phase of the algorithm restores the buffer and completes the sort by coalescing the A's and B's of the buffer and those of the undersized blocks with the rest of the list. All phases except the second are performed in linear time.

In our algorithm, we make use of the trick of block permuting or interchanging two consecutive blocks, UV by performing three block reversals ($VU = (U^R V^R)^R$). This algorithm is referred to as *block_permute* in the pseudocode description of the algorithms.

Step 1: *Forming the Internal Buffer.*

The internal buffer consists of $2\lceil\sqrt{n}\rceil$ items; our task here is to initialize it to consist of a contiguous block of the first $\lceil\sqrt{n}\rceil$ A's followed by a contiguous block of the first $\lceil\sqrt{n}\rceil$ B's. In order to do this, start at the position of the $\lceil\sqrt{n}\rceil$th A which we assume, without loss of generality, occurs after the $\lceil\sqrt{n}\rceil$th B. Locate the preceding A, and block permute the $\lceil\sqrt{n}\rceil$th A with any intervening B's, so that the $\lceil\sqrt{n}\rceil$th A abuts the preceding A in L. Continue in the same fashion with these two A's, and stop when the first $\lceil\sqrt{n}\rceil$ A's are contiguous. If they do not occupy the first $\lceil\sqrt{n}\rceil$ positions of L, perform a block permutation so that they do.

The time complexity of this procedure is linear. Each of the $\lceil\sqrt{n}\rceil$ A's is block permuted at most $\lceil\sqrt{n}\rceil$ times, and each B, at most once.

Step 2: *Sorting blocks*

The internal buffer is used to encode array M described in the proof of Lemma 1.

Conceptually, the ith A and the ith B form a pair occupying positions i and $\lceil\sqrt{n}\rceil + i$. If an A occupies position i, it is interpreted in the same way as $M[i] = 0$. If the B occupies position i, it is interpreted as if $M[i] = 1$. For j going from 2 to $(\lfloor\sqrt{n}\rfloor - 1)$, we apply Lemma 1 to the block $L[j\lceil\sqrt{n}\rceil + 1 : (j+1)\lceil\sqrt{n}\rceil]$. After each such block is sorted, the $\lceil\sqrt{n}\rceil$ B's in the buffer precede the $\lceil\sqrt{n}\rceil$ A's in the buffer, and the buffer can be restored by permuting the block of B's with the block of A's. There may be one block at the end of L of size less than $\lceil\sqrt{n}\rceil$, and it is sorted in the same fashion. Sorting each block takes $O(n)$ comparisons and $(O(\lceil\sqrt{n}\rceil)$ data movements, so the entire step takes $O(n^{3/2})$ comparisons and $O(n)$ data movements.

Step 3: *Regrouping items*

At this point, each block is sorted. We now rearrange the array so that each full block (except perhaps the last one) contains items of only one type.

Consider the portion consisting of all L but the internal buffer, i.e. $L[2\lceil\sqrt{n}\rceil + 1 : n]$, and suppose there is a $\lceil\sqrt{n}\rceil$th A in this sublist and it appears in position t of block j_1 of this sublist. If j_1 is 1, then the first $\lceil\sqrt{n}\rceil$ A's of the sublist are already contiguous, and we proceed to the next sublist. Otherwise, block permute the A's in positions $(j_1 + 1)\lceil\sqrt{n}\rceil + 1$ through $(j_1 + 1)\lceil\sqrt{n}\rceil + t$ with the B's in the preceding block. (Note that the $(j_1 + 2)$nd block of L is the j_1th block of the sublist $L[2\lceil\sqrt{n}\rceil + 1 : n]$.) Repeat this process (of block permuting the contiguous A's with B's in the previous block), until the $\lceil\sqrt{n}\rceil$ A's are contiguous. They will now fully occupy the first block within $L[2\lceil\sqrt{n}\rceil + 1 : n]$ that previously contained any A's. This procedure takes $O(j_1\lceil\sqrt{n}\rceil)$ time. At this time, each block of the sublist $L[1 : (j_1 + 1)\lceil\sqrt{n}\rceil]$, for some $j_1 \geq 2$, contains items of only one type. Now block permute the first t B's of the $(j_1 + 2)$nd block of L with the A's in the block so that the block is sorted.

Now, in the same fashion, consider the sublist $L[(j_1 + 1)\lceil\sqrt{n}\rceil + 1 : n)$, and suppose that the $\lceil\sqrt{n}\rceil$th A appears in the j_2th block of this sublist. These A's can be coalesced to occupy a block of the sublist, in $O(j_2\lceil\sqrt{n}\rceil)$ time. If this process is repeated s times until all A's (except a small group at the end of L) are grouped together into blocks of size $\lceil\sqrt{n}\rceil$, $O(\lceil\sqrt{n}\rceil\Sigma_{l=1}^{s} j_l) = O(n)$ time is taken.

Step 4: *Rearranging blocks*

Now keys having the same value are grouped in units of size $\lceil\sqrt{n}\rceil$, except possibly one regular and one undersized block at the end. Hence, Lemma 1 can be applied to sort the $O(\lceil\sqrt{n}\rceil)$ full sized single valued blocks in $L[2\lceil\sqrt{n}\rceil + 1 : n]$, based on their first elements. As the swap operation is actually a block swap, it takes $O(\lceil\sqrt{n}\rceil)$ time. Consequently, Step 4 takes $O(n)$ comparisons and $O(n)$ data movements.

Step 5: *Finishing up*

After completing the block sort, restore the buffer and permute the B's in the buffer so that they abut the B's in the rest of the array. Then permute the A's in the last two blocks so that they abut the A's in the rest of the array. At this point, the array is sorted.

```
sort2(L, n)
if #A's is less than ⌈√n⌉ then
  few_keys(L)
else if #B's is less than ⌈√n⌉ then
  few_keys(L)
  block_permute the B's so that they appear at the end of L.
else
  form_buffer(L)
  for i = 2 to ⌊√n⌋ − 1 do
    stable_permsort(L[i⌈√n⌉ + 1 : (i + 1)⌈√n⌉], buffer, ⌈√n⌉)
    block_permute(L[1 : ⌈√n⌉], L[⌈√n⌉ + 1 : 2⌈√n⌉])
  endfor
  apply stable_permsort to undersized block at the end and restore buffer
  regroup_list(L)
  stable_permsort(L[2⌈√n⌉ + 1 : n], buffer, ⌊√n⌋)
    (each item is a block of ⌈√n⌉ items)
  finish_up(L)
endif
```

Figure 3. Stably sorting an array of two distinct values

Step 2 takes $O(n^{3/2})$ comparisons while the other steps require only a linear number of comparisons. Furthermore, each step makes $O(n)$ data movements and uses $O(1)$ indices, so the algorithm runs in the bounds stated in the lemma. Stability is assured because the relative order of equal-valued keys before each phase is the same as the relative order after each phase. A pseudocode description of the overall structure of the algorithm is presented in Figure 3. In the description, *form_buffer* is the algorithm described in Step 1, *regroup_list* is the algorithm referred to in Step 3, and *finish_up* is the algorithm mentioned in Step 5. Procedure *few_keys* extracts to the front of the list all the items of a key whose key value occurs fewer than $\lceil \sqrt{n} \rceil$ times; and it is alluded to in the text. ■

Our original motivation in proving this lemma was as a building block for sorting an arbitrary number of distinct keys. For that purpose, the lemma is adequate as other problems keep the running time of the algorithm (in the next section) quadratic. The specific result, however, can be improved as we adapt it to handle any list containing a constant number of key values and reduce the run time. Note first that the method is immediately adaptable to a file containing any fixed constant number of distinct keys. Next observe that as Step 2 is the only nonlinear portion of the algorithm, we could apply our method recursively in that step. Going k levels deep in the recursion, we can get a stable sorting algorithm that makes $O(kn)$ data movements and $O(n^{1+1/2^{k+1}} + kn)$ comparisons for any k. The storage costs implicit in recursion can be removed by starting with blocks of size roughly $n^{1/2^{k+1}}$ and proceeding in a bottom-up fashion. Setting k to $(\lg \lg n)$ above, we get

COROLLARY 1. *An array of n elements consisting of a constant number of distinct key values can be stably sorted using a constant number of indices, $O(n \lg \lg n)$ comparisons and $O(n \lg \lg n)$ data movements.*

We can, in fact, do substantially better than this bound by reducing the number of data movements to linear and the number of comparisons to $O(n \lg^{(k)} n)$ for any constant $k \geq 1$. Recall that $\lg^{(k)}$ is the *logarithm* function iterated k times.

The first step is to refine Lemma 1 to take advantage of the fact that there are only a constant number of distinct keys. This refinement applied to Lemma 2 would immediately reduce the number of comparisons to $O(n \lg n)$. A slight modification to the scheme of Lemma 2 together with recursion improves the number of comparisons to $O(n \lg^{(k)} n)$. For the remainder of this section, assume that the list contains c distinct key values, $1, 2, \ldots, c$.

As the number of distinct key values is a constant, we can store the l_x's, for each distinct value x, in c storage locations. Suppose further that we have two storage locations available for each position i. The first is a single bit containing the value of $M[i]$ from Lemma 1. The second storage location is $O(\lg n)$ bits, and contains the value q_i, the number of items with key value $L[i]$ to the left of i in the original, unsorted list. The q_i's can be computed in linear time in one pass over the list using a constant number of counters. Given M, the l_x's and the q_i's, we can place each element in its final position in linear time using an algorithm similar to the one in the proof of Lemma 1. We have thus proved the following lemma:

LEMMA 3. *An array of size n containing a constant number of distinct key values can be stably sorted in linear time using $O(n)$ indices.*

If we attempt to encode the information at each position into an internal buffer as in Lemma 2, then just writing the q_i's requires $O(n \lg n)$ bit assignments which will result in $O(n \lg n)$ data movements. We therefore encode information for a sparse subset of the positions to prove the following lemma:

LEMMA 4. *An array of size n containing a constant number of distinct key values can be stably sorted using $O(n \lg n)$ comparisons, $O(n)$ data movements and $O(n)$ bits of extra storage.*

PROOF: We adapt the ideas of Lemma 3 to use only $O(n)$ bits.

As in Lemma 1, assign a single bit to each position to form array M. Using an additional $O(n)$ extra bits, store the following positional information. For every $\lceil \lg n \rceil$th position i, store $q_{i,1}, \ldots, q_{i,c}$, the number of elements to the left of position i in the original, unsorted list with key values $1, \ldots, c$ respectively. Also, as before, store l_x for each key value x in c storage locations.

Let x be the element at location i of the array which is not known to be in its final position. In order to compute the final address of $L[i]$, there are two cases. If i is

a multiple of $\lceil \lg n \rceil$, then the final address of $L[i]$ is $l_x + q_{i,x} + 1$. If i is not multiple of $\lceil \lg n \rceil$, let j be the largest multiple of $\lceil \lg n \rceil$ which is less than i. To compute the final address of $L[i]$, we need three quantities: (1) l_x, (2) $q_{j,x}$ and (3) s, the number of keys with value x from positions j to $i - 1$ which are not in their correct locations at the time $L[i]$ is examined. From this information, the final address of $L[i]$ is the sth position for which M value is 0 following position $l_x + q_{j,x} + 1$.

The values l_x for each item x and $q_{i,x}$ for each item x and for every $\lceil \lg n \rceil$th location i can be computed in linear time using a constant number of counters and a constant number of passes over the list. Once the values $q_{i,x}$ and l_x are computed, the final address computation needs only the computation of s which, in the worst case, takes $\lg n$ time. Given the ability to compute final addresses, the sorting strategy is precisely that of Lemma 1. Hence the entire algorithm takes $O(n \lg n)$ comparisons and linear data movements. ■

If we encode the $O(n)$ bits using an internal buffer of A's and B's and apply the above technique to the block sorting step (Step 2) of Lemma 2, then the following bounds can be proven. We can encode the positional information contained in the $O(n)$ bits in linear time. If i is a multiple of $\lceil \lg n \rceil$, the final address computation takes $O(\lg n)$ comparisons to decode. If i is not a multiple of $\lceil \lg n \rceil$, $q_{j,x}$ and s can both be obtained in $O(\lg n)$ comparisons. Therefore, the total number of comparisons made by the step will be $O(n \lg n)$ while the number of data movements will remain linear. As the other steps of Lemma 2 take linear time, we have proved the following theorem:

THEOREM 1. *An array of size n containing a constant number of distinct key values can be stably sorted using a constant number of indices, $O(n \lg n)$ comparisons and $O(n)$ data movements.*

We present some corollaries to the theorem resulting in an $O(n \lg^* n)$ in situ stable algorithm to sort the array. Recall that $\lg^*$ denotes the number of times the logarithm base 2 may be taken before the quantity is at most 0.

COROLLARY 2. *An array of n elements containing a constant number of distinct key values can be stably sorted in linear time using $O(\lg n)$ indices.*

PROOF: We proceed as in Lemma 2 except that instead of blocks of size $\lceil \sqrt{n} \rceil$, we form blocks of size $\lceil \lg n \rceil$. We use the $O(\lg n)$ indices for the block sorting step, thus eliminating the need for internal buffer and hence Step 1 of Lemma 2.

Each block is sorted using the $O(\lg n)$ indices. By Lemma 3, this step can be done in linear time. Observe that Steps 3 and 5 can still be done in linear time. In Step 4, we have $O(n/\lg n)$ blocks of size $\lceil \lg n \rceil$ to be sorted. The algorithm of Theorem 1 can be applied to sort the blocks in a linear number of comparisons and $O(n/(\lg n))$ block moves. As each block move costs $O(\lg n)$, Step 4 can be done in linear time. Consequently, the whole algorithm is linear. ■

COROLLARY 3. *An array of n elements contaning a constant number of distinct key values can be stably sorted using only a constant number of indices,* $O(n \lg^{(k+1)} n + kn)$ *comparisons and* $O(kn)$ *data movements for any fixed k.*

PROOF: In the proof of the previous corollary, we observed that except for the block sorting step, every step can be done in linear time using only a constant number of indices. For the block sorting step, recurse k levels deep and then sort the remaining blocks using the algorithm of Theorem 1. Each level of recursion takes linear time and therefore after k levels, the algorithm would make $O(kn)$ comparisons and data movements. Furthermore, after k levels of recursion we will have $O(n \lg^{(k)} n)$ blocks of size roughly $\lg^k n$ each. As the algorithm of Theorem 1 makes $O(n \lg n)$ comparisons and linear data movements to sort an array of n elements containing a constant number of distinct key values, the stated bounds in the corollary hold. Here again, the implicit storage for the recursion can be removed by building up from structures of size approximately $\lg^{(k)} n$. ■

COROLLARY 4. *An array of n elements containing a constant number of distinct key values can be stably sorted using a constant number of indices and* $O(n \lg^* n)$ *comparisons and data movements.*

PROOF: Let $k = \lg^* n$ in Corollary 3. ■

COROLLARY 5. *Quicksort can be adapted to stably sort an array of n elements in* $O(n \lg n)$ *time on average using* $O(\lg n)$ *indices.*

PROOF: The standard version of quicksort uses $O(\lg n)$ pointers for the recursive calls. (The trick of sorting the smaller sublist first guarantees this bound.) In the partition step of quicksort, elements less than, equal to, and greater than the pivot element can be considered to have key values A, B and C respectively. Then the stable partitioning reduces to stably sorting the three key values. This can be done in linear time using another $O(\lg n)$ indices from Corollary 2. After the stable partitioning, the $O(\lg n)$ pointers can be used for the partitioning in the next recursive step. Thus applying this stable partitioning method at each recursive step, we obtain a stable version of quicksort that uses $O(\lg n)$ pointers. As the standard version of quicksort runs in $O(n \lg n)$ time on average, the corollary follows. ■

Recently, however, versions of quicksort ([2], [3], [13]) have been proposed that do not require the $O(\lg n)$ indices implicit in the recursive calls. Corollary 4 can be used with this approach to stably partition the array at each step.

COROLLARY 6. *Quicksort can be adapted to stably sort an array of n elements in* $O(n \lg n \lg^* n)$ *time on average using only a constant number of indices.*

4. Stably sorting many keys.

We now proceed to the most general result of the paper, that an array with an arbitrary number of distinct key values can be stably sorted in quadratic time using linear data movements, and a constant amount of extra space. This observation is based on an invariant; in some sense, the invariant is the "midpoint" of the algorithm in that it is not true initially, but can be established in situ by Lemma 2. After the invariant is established, a process similar to the in situ permutation algorithm can be used to complete the sort.

Consider a particular key x. Suppose there are l_x items with keys strictly smaller than x and that key x occurs e_x times. In the correctly rearranged array, items with key x occupy positions $l_x + 1$ through $l_x + e_x$. As before, call this interval I_x.

INVARIANT: Any key with value x appearing in I_x must be in its correct final position.

4.1. *Establishing the invariant.*

To establish the invariant, we process each I_x in increasing order by distinct value. After determining the bounds of I_x, perform the sorting algorithm used in Lemma 2 on I_x, treating an item as an "A" if it has key x and a "B" otherwise. Next, compute $q_{(l_x+1)}$, the number of items with key x to the left of the interval I_x. These items will occupy the first $q_{(l_x+1)}$ locations of I_x when sorting is complete. Hence, permute the first $q_{(l_x+1)}$ B's in I_x with the block of A's in I_x. At this point, the items with key x inside interval I_x are placed correctly.

The total amount of work done by this step is quadratic as it is dominated by the time taken to find the bounds on the I_x. Furthermore, only a linear number of data movements are made by the procedure. For each key, we can calculate I_x in linear time without any data movement. After computing I_x, the sorting algorithm of Lemma 2 takes $O(|I_x|^{3/2})$ time and makes $O(|I_x|)$ data movements. Since there are at most n different keys and the I_x's partition the list, the stated bounds hold.

4.2. *Sorting after the invariant is established.*

After the invariant is established, we can compute the correct address of any item in the sorted array in linear time with no data movement. To compute the final location of the element x at location i, the procedure is to first compute I_x by computing l_x and e_x. If the position i is within I_x, then its current address is also its final address. If i is outside I_x, then compute q_i, the number of items of key x currently to the left of position i outside I_x. The items counted in q_i are not yet placed correctly, but will be placed to the left of the final location of $L[i]$ in I_x.

```
stable_insitu_permsort(L, n)
1. Establish the invariant
   max ← max(L)
   x ← −∞
   repeat
      x ← min{L[i]: L[i] > x, 1 ≤ i ≤ n}
      l ← #{j: L[j] < x, 1 ≤ j ≤ n}
      e ← #{j: L[j] = x, 1 ≤ j ≤ n}
      sort2(L[l + 1: l + e], e)
      r ← #{j: L[j] = x, l < j ≤ l + e}
      q ← #{j: L[j] = x, 1 ≤ j ≤ l}
      block_permute(L[l + 1: l + r], L[l + r + 1: l + r + q])
   until x = max
2. Follow through the permutation cycles
   For k = 1 to n
      i ← destination2(k)
      while i ≠ k
         j ← destination2(i)
         swap(L[i], L[k])
         i ← j
      endwhile
   endfor
destination2(i)
      l ← #{j: L[j] < L[i], 1 ≤ j ≤ n}
      e ← #{j: L[j] = L[i], 1 ≤ j ≤ n}
      if l < i ≤ l + e then
         destination2 ← i
      else
         q ← #{j: L[j] = L[i], 1 ≤ j < i and (j ≤ l or l + e < j)}
         destination2 ← j such that L[j] ≠ L[i] and #{s: l + 1 ≤ s < j and
         L[s] ≠ L[i]} = q
```

Figure 4. Stable in situ permutation sort

Therefore, the final address of item $L[i]$ is the location of the $(q_i + 1)$st non-x item in I_x. A pseudocode description of the algorithm is given in Figure 4. In the pseudocode, *destination*2 returns the final address of the item at the position given by its argument.

Because of the invariant we have established, this sorting procedure has an important advantage over that of Lemma 1. That is, an element displaced is always different from the element that displaces it and so the key values of any two successive elements in a nontrivial cycle are distinct. We can take advantage of this observation to eliminate the second parameter present in *destination*1 of Figure 2.

Let $L[i]$ be the value whose final destination is to be determined. In this pseudocode, unlike in *stable_permsort*, an element is moved to position k *after* its final position is determined. Further, the invariant maintained before the call of routine *destination*2 with parameter i is that, the element that displaces $L[i]$ is in the position k. Therefore, the position k is always ignored (as $L[k] \neq L[i]$) while calculating the destination of $L[i]$ (more specifically, in the calculation of q). As this was the only purpose of the second parameter (k) in *destination*1, there is no need for it in *destination*2.

Since there are a linear number of final address calculations taking $O(n)$ time each, and each data movement puts at least one item in its final position, this procedure takes quadratic time and makes a linear number of data movements.

As a result, we have proved:

THEOREM 2. *Let L be an array of n items. L can be stably sorted in $O(n^2)$ time using $O(n)$ data movements and $O(1)$ indices.*

5. Conclusion.

In this paper we have shown that there exists an algorithm, albeit quadratic, to stably sort n elements in situ using a linear number of data movements.

In Section 3, we developed an algorithm for the restricted case of a constant number of keys using $O(n \lg^{(k+1)} n + kn)$ comparisons and $O(kn)$ data movements for any fixed k. Despite the improvement to the restricted problem, it seems difficult to improve on the quadratic time bound for the general problem. Although we are primarily concerned with the mathematical question of what can be done, it is hard to justify quadratic time to obtain stability, constant extra space and linear data movements. An interesting open problem is to devise a sorting algorithm which satifies the space, data movements and stability requirements but runs in subquadratic time.

REFERENCES

[1] S. Baase, *Computer Algorithms: Introduction to Design and Analysis*, Addison-Wesley (1988).
[2] B. Ďurian, *Quicksort without a stack*, Math. Foundations of Computer Science, *Lecture Notes in Computer Science* 223, *Springer Verlag* (1986) 283–289.
[3] D. Gries, *Constant-space quicksort*, Unpublished manuscript, March 1989.
[4] T. Hoare, *Quicksort*, CACM 4 (7) (1961) 321.
[5] D. E. Knuth, *The Art of Computer Programming. Volume III: Sorting and Searching*, Addison-Wesley (1973).
[6] D. E. Knuth, *Mathematical Analysis of Algorithms*, Proceedings, I.F.I.P. Congress, ed. C.V Frieman, North-Holland, Amsterdam (1972) 19–27.
[7] M. A. Kronrod, *Optimal ordering algorithm without operational field*, Soviet Math. Dokl. **10** (1969) 744–746.
[8] I. D. G. Macleod, *An algorithm for in-situ permutation*, Australian Computing Journal **2** (1970) 16–19.

[9] J. I. Munro and V. Raman, *Sorting with minimum data movement*, Proceedings of Workshop on Algorithms and Data Structures, Ottawa, *Lecture Notes in Computer Science* 382, *Springer Verlag* (August 1989) 552–562.
[10] L. Trabb Pardo, *Stable sorting and merging with optimal space and time bounds*, SIAM J. on Computing **6** (1977) 351–372.
[11] J. S. Salowe and W. L. Steiger, *Stable unmerging in linear time and constant space*, IPL **25** (1987) 285–294.
[12] J. S. Salowe and W. L. Steiger, *Simplified stable merging tasks*, *J. Algorithms* **8** (1987) 557–571.
[13] L. M. Wegner, *A generalized one way stackless quicksort*, BIT **27** (1987) 44–48.
[14] P. F. Windley, *Transposing matrices in a digital computer*, The Computer Journal **2** (1959) 47–48.