

# Radix Sorting With No Extra Space<sup>\*</sup>

Gianni Franceschini<sup>1</sup>, S. Muthukrishnan<sup>2</sup>, and Mihai Pătrașcu<sup>3</sup>

<sup>1</sup> Dept of Computer Science, Univ. of Pisa; francesc@di.unipi.it

<sup>2</sup> Google Inc., NY; muthu@google.com

<sup>3</sup> MIT, Boston; mip@mit.edu

**Abstract.** It is well known that  $n$  integers in the range  $[1, n^c]$  can be sorted in  $O(n)$  time in the RAM model using radix sorting. More generally, integers in any range  $[1, U]$  can be sorted in  $O(n\sqrt{\log \log n})$  time [5]. However, these algorithms use  $O(n)$  words of extra memory. Is this necessary?

We present a simple, stable, integer sorting algorithm for words of size  $O(\log n)$ , which works in  $O(n)$  time and uses only  $O(1)$  words of extra memory on a RAM model. This is the integer sorting case most useful in practice. We extend this result with same bounds to the case when the keys are read-only, which is of theoretical interest. Another interesting question is the case of arbitrary  $c$ . Here we present a black-box transformation from any RAM sorting algorithm to a sorting algorithm which uses only  $O(1)$  extra space and has the same running time. This settles the complexity of in-place sorting in terms of the complexity of sorting.

## 1 Introduction

Given  $n$  integer keys  $S[1 \dots n]$  each in the range  $[1, n]$ , they can be sorted in  $O(n)$  time using  $O(n)$  space by *bucket sorting*. This can be extended to the case when the keys are in the range  $[1, n^c]$  for some positive constant  $c$  by *radix sorting* that uses repeated bucket sorting with  $O(n)$  ranged keys. The crucial point is to do each bucket sorting *stably*, that is, if positions  $i$  and  $j$ ,  $i < j$ , had the same key  $k$ , then the copy of  $k$  from position  $i$  appears before that from position  $j$  in the final sorted order. Radix sorting takes  $O(cn)$  time and  $O(n)$  space. More generally, RAM sorting with integers in the range  $[1, U]$  is a much-studied problem. Currently, the best known bound is the randomized algorithm in [5] that takes  $O(n\sqrt{\log \log n})$  time, and the deterministic algorithm in [1] that takes  $O(n \log \log n)$  time. These algorithms also use  $O(n)$  words of extra memory in addition to the input.

We ask a basic question: *do we need  $O(n)$  auxiliary space for integer sorting?* The ultimate goal would be to design *in-place* algorithms for integer sorting that uses only  $O(1)$  extra words. This question has been explored in depth for comparison-based sorting, and after a series of papers, we now know that in-place, stable comparison-based sorting can be done in  $O(n \log n)$  time [10]. Some very nice algorithmic techniques have been developed in this quest. However, no such results are known for the integer sorting case. Integer sorting is used as a subroutine in a number of algorithms that deal with trees and graphs,

---

<sup>\*</sup> A full version of this paper is available as arXiv:0706.4107.

including, in particular, sorting the transitions of a finite state machine. Indeed, the problem arose in that context for us. In these applications, it is useful if one can sort in-place in  $O(n)$  time. From a theoretical perspective, it is likewise interesting to know if the progress in RAM sorting, including [3, 1, 5], really needs extra space.

Our results are in-place algorithms for integer sorting. Taken together, these results solve much of the issues with space efficiency of integer sorting problems. In particular, our contributions are threefold.

*A practical algorithm.* In Section 2, we present a stable integer sorting algorithm for  $O(\log n)$  sized words that takes  $O(n)$  time and uses only  $O(1)$  extra words.

This algorithm is a simple and practical replacement to radix sort. In the numerous applications where radix sorting is used, this algorithm can be used to improve the space usage from  $O(n)$  to only  $O(1)$  extra words. We have implemented the algorithm with positive results.

One key idea of the algorithm is to compress a portion of the input, modifying the keys. The space thus made free is used as extra space for sorting the remainder of the input.

*Read-only keys.* It is theoretically interesting if integer sorting can be performed in-place *without modifying the keys*. The algorithm above does not satisfy this constraint. In Section 3, we present a more sophisticated algorithm that still takes linear time and uses only  $O(1)$  extra words without modifying the keys. In contrast to the previous algorithm, we cannot create space for ourselves by compressing keys. Instead, we introduce a new technique of *pseudo pointers* which we believe will find applications in other succinct data structure problems. The technique is based on keeping a set of distinct keys as a pool of preset read-only pointers in order to maintain linked lists as in bucket sorting.

As a theoretical exercise, the full version of this paper also considers the case when this sorting has to be done stably. We present an algorithm with identical performance that is also stable. Similar to the other in-place stable sorting algorithms e.g., comparison-based sorting [10], this algorithm is quite detailed and needs very careful management of keys as they are permuted. The resulting algorithm is likely not of practical value, but it is still fundamentally important to know that bucket and radix sorting can indeed be solved stably in  $O(n)$  time with only  $O(1)$  words of extra space. For example, even though comparison-based sorting has been well studied at least since 60's, it was not until much later that optimal, stable in-place comparison-based sorting was developed [10].

*Arbitrary word length.* Another question of fundamental theoretical interest is whether the recently discovered integer sorting algorithms that work with long keys and sort in  $o(n \log n)$  time, such as [3, 1, 5], need any auxiliary space. In Section 4, we present a black-box transformation from any RAM sorting algorithm to an sorting algorithm which uses only  $O(1)$  extra space, and retains the same time bounds. As a result, the running time bounds of [1, 5] can now be matched with only  $O(1)$  extra space. This transformation relies on a fairly natural technique of compressing a portion of the input to make space for simulating space-inefficient RAM sorting algorithms.

*Definitions.* Formally, we are given a sequence  $S$  of  $n$  elements. The problem is to sort  $S$  according to the integer keys, under the following assumptions:

- (i) Each element has an integer key within the interval  $[1, U]$ .
- (ii) The following unit-cost operations are allowed on  $S$ : (a) indirect address of any position of  $S$ ; (b) read-only access to the key of any element; (c) exchange of the positions of any two elements.
- (iii) The following unit-cost operations are allowed on integer values of  $O(\log U)$  bits: addition, subtraction, bitwise AND/OR and unrestricted bit shift.
- (iv) Only  $O(1)$  auxiliary words of memory are allowed; each word had  $\log U$  bits.

For the sake of presentation, we will refer to the elements' keys as if they were the input elements. For example, for any two elements  $x, y$ , instead of writing that the key of  $x$  is less than the key of  $y$  we will simply write  $x < y$ . We also need a precise definition of the *rank* of an element in a sequence when multiple occurrences of keys are allowed: the *rank* of an element  $x_i$  in a sequence  $x_1 \dots x_t$  is the cardinality of the multiset  $\{x_j \mid x_j < x_i \text{ or } (x_j = x_i \text{ and } j \leq i)\}$ .

## 2 Stable Sorting for Modifiable Keys

We now describe our simple algorithm for (stable) radix sort without additional memory.

*Gaining space.* The first observation is that numbers in sorted order have less entropy than in arbitrary order. In particular,  $n$  numbers from a universe of  $u$  have binary entropy  $n \log u$  when the order is unspecified, but only  $\log \binom{u}{n} = n \log u - \Theta(n \log n)$  in sorted order. This suggests that we can “compress” sorted numbers to gain more space:

**Lemma 1.** *A list of  $n$  integers in sorted order can be represented as: (a) an array  $A[1 \dots n]$  with the integers in order; (b) an array of  $n$  integers, such that the last  $\Theta(n \log n)$  bits of the array are zero. Furthermore, there exist in-place  $O(n)$  time algorithms for switching between representations (a) and (b).*

*Proof.* One can imagine many representations (b) for which the lemma is true. We note nonetheless that some care is needed, as some obvious representations will in fact not lead to in-place encoding. Take for instance the appealing approach of replacing  $A[i]$  by  $A[i] - S[i - 1]$ , which makes numbers tend to be small (the average value is  $\frac{u}{n}$ ). Then, one can try to encode the difference using a code optimized for smaller integers, for example one that represents a value  $x$  using  $\log x + O(\log \log x)$  bits. However, the obvious encoding algorithm will not be in-place: even though the scheme is guaranteed to save space over the entire array, it is possible for many large values to cluster at the beginning, leading to a rather large prefix being in fact expanded. This makes it hard to construct the encoding in the same space as the original numbers, since we need to shift a lot of data to the right before we start seeing a space saving.

As it will turn out, the practical performance of our radix sort is rather insensitive to the exact space saving achieved here. Thus, we aim for a representation which makes in-place encoding particularly easy to implement, sacrificing constant factors in the space saving.

First consider the most significant bit of all integers. Observe that if we only remember the minimum  $i$  such that  $A[i] \geq u/2$ , we know all most significant bits (they are zero up to  $i$  and one after that). We will encode the last  $n/3$  values in the array more compactly, and use the most significant bits of  $A[1 \dots \frac{2}{3}n]$  to store a stream of  $\frac{2}{3}n$  bits needed by the encoding.

We now break a number  $x$  into  $\text{hi}(x)$ , containing the upper  $\lfloor \log_2(n/3) \rfloor$  bits, and  $\text{lo}(x)$ , with the low  $\log u - \lfloor \log_2(n/3) \rfloor$  bits. For all values in  $A[\frac{2}{3}n + 1 \dots n]$ , we can throw away  $\text{hi}(A[i])$  as follows. First we add  $\text{hi}(A[\frac{2}{3}n + 1])$  zeros to the bit stream, followed by a one; then for every  $i = \frac{2}{3}n + 2, \dots, n$  we add  $\text{hi}(A[i]) - \text{hi}(A[i - 1])$  zeros, followed by a one. In total, the stream contains exactly  $n/3$  ones (one per element), and exactly  $\text{hi}(A[n]) \leq n/3$  zeros. Now we simply compact  $\text{lo}(A[\frac{2}{3}n + 1]), \dots, \text{lo}(A[n])$  in one pass, gaining  $\frac{n}{3} \lfloor \log_2(n/3) \rfloor$  free bits.  $\square$

*An unstable algorithm.* Even just this compression observation is enough to give a simple algorithm, whose only disadvantage is that it is unstable. The algorithm has the following structure:

1. sort the subsequence  $S[1 \dots (n/\log n)]$  using the optimal in-place mergesort in [10].
2. compress  $S[1 \dots (n/\log n)]$  by Lemma 1, generating  $\Omega(n)$  bits of free space.
3. radix sort  $S[(n/\log n) + 1 \dots n]$  using the free space.
4. uncompress  $S[1 \dots (n/\log n)]$ .
5. merge the two sorted sequences  $S[1 \dots (n/\log n)]$  and  $S[(n/\log n) + 1 \dots n]$  by using the in-place, linear time merge in [10].

The only problematic step is 3. The implementation of this step is based on the cycle leader permuting approach where a sequence  $A$  is re-arranged by following the cycles of a permutation  $\pi$ . First  $A[1]$  is sent in its final position  $\pi(1)$ . Then, the element that was in  $\pi(1)$  is sent to its final position  $\pi(\pi(1))$ . The process proceeds in this way until the cycle is closed, that is until the element that is moved in position 1 is found. At this point, the elements starting from  $A[2]$  are scanned until a new cycle leader  $A[i]$  (i.e. its cycle has not been walked through) is found,  $A[i]$ 's cycle is followed in its turn, and so forth.

To sort, we use  $2n^\epsilon$  counters  $c_1, \dots, c_{n^\epsilon}$  and  $d_1, \dots, d_{n^\epsilon}$ . They are stored in the auxiliary words obtained in step 2. Each  $d_j$  is initialized to 0. With a first scan of the elements, we store in any  $c_i$  the number of occurrences of key  $i$ . Then, for each  $i = 2 \dots n^\epsilon$ , we set  $c_i = c_{i-1} + 1$  and finally we set  $c_1 = 1$  (in the end, for any  $i$  we have that  $c_i = \sum_{j < i} c_j + 1$ ). Now we have all the information for the cycle leader process. Letting  $j = (n/\log n) + 1$ , we proceed as follows:

- (i) let  $i$  be the key of  $S[j]$ ;
- (ii) if  $c_i \leq j < c_{i+1}$  then  $S[j]$  is already in its final position, hence we increment  $j$  by 1 and go to step (i);
- (iii) otherwise, we exchange  $S[j]$  with  $S[c_i + d_i]$ , we increment  $d_i$  by 1 and we go to step (i).

Note that this algorithm is inherently unstable, because we cannot differentiate elements which should fall between  $c_i$  and  $c_{i+1} - 1$ , given the free space we have.

*Stability through recursion.* To achieve stability, we need more than  $n$  free bits, which we can achieve by bootstrapping with our own sorting algorithm, instead of merge sort. There is also an important practical advantage to the new stable approach: the elements are permuted much more conservatively, resulting in better cache performance.

1. recursively sort a constant fraction of the array, say  $S[1 \dots n/2]$ .
2. compress  $S[1 \dots n/2]$  by Lemma 1, generating  $\Omega(n \log n)$  bits of free space.
3. for a small enough constant  $\gamma$ , break the remaining  $n/2$  elements into chunks of  $\gamma n$  numbers. Each chunk is sorted by a classic radix sort algorithm which uses the available space.
4. uncompress  $S[1 \dots n/2]$ .
5. we now have  $1 + 1/\gamma = O(1)$  sorted subarrays. We merge them in linear time using the stable in-place algorithm of [10].

We note that the recursion can in fact be implemented bottom up, so there is no need for a stack of superconstant space. For the base case, we can use bubble sort when we are down to  $n \leq \sqrt{n_0}$  elements, where  $n_0$  is the original size of the array at the top level of the recursion.

Steps 2 and 4 are known to take  $O(n)$  time. For step 3, note that radix sort in base  $R$  applied to  $N$  numbers requires  $N + R$  additional words of space, and takes time  $O(N \log_R u)$ . Since we have a free space of  $\Omega(n \log n)$  bits or  $\Omega(n)$  words, we can set  $N = R = \gamma n$ , for a small enough constant  $\gamma$ . As we always have  $n = \Omega(\sqrt{n_0}) = u^{\Omega(1)}$ , radix sort will take linear time.

The running time is described by the recursion  $T(n) = T(n/2) + O(n)$ , yielding  $T(n) = O(n)$ .

*A self-contained algorithm.* Unfortunately, all algorithms so far use in-place stable merging algorithm as in [10]. We want to remove this dependence, and obtain a simple and practical sorting algorithm. By creating free space through compression at the right times, we can instead use a simple merging implementation that needs additional space. We first observe the following:

**Lemma 2.** *Let  $k \geq 2$  and  $\alpha > 0$  be arbitrary constants. Given  $k$  sorted lists of  $n/k$  elements, and  $\alpha n$  words of free space, we can merge the lists in  $O(n)$  time.*

*Proof.* We divide space into blocks of  $\alpha n/(k+1)$  words. Initially, we have  $k+1$  free blocks. We start merging the lists, writing the output in these blocks. Whenever we are out of free blocks, we look for additional blocks which have become free in the original sorted lists. In each list, the merging pointer may be inside some block, making it yet unavailable. However, we can only have  $k$  such partially consumed blocks, accounting for less than  $k \frac{\alpha n}{k+1}$  wasted words of space. Since in total there are  $\alpha n$  free words, there must always be at least one block which is available, and we can direct further output into it.

At the end, we have the merging of the lists, but the output appears in a nontrivial order of the blocks. Since there are  $(k+1)(1+1/\alpha) = O(1)$  blocks in total, we can remember this order using constant additional space. Then, we can permute the blocks in linear time, obtaining the true sorted order.  $\square$

Since we need additional space for merging, we can never work with the entire array at the same time. However, we can now use a classic sorting idea, which is often used in introductory algorithms courses to illustrate recursion (see, e.g. [2]). To sort  $n$  numbers, one can first sort the first  $\frac{2}{3}n$  numbers (recursively), then the last  $\frac{2}{3}n$  numbers, and then the first  $\frac{2}{3}n$  numbers again. Though normally this algorithm gives a running time of  $\omega(n^2)$ , it works efficiently in our case because we do not need recursion:

1. sort  $S[1 \dots n/3]$  recursively.
2. compress  $S[1 \dots \frac{n}{3}]$ , and sort  $S[\frac{n}{3} + 1 \dots n]$  as before: first radix sort chunks of  $\gamma n$  numbers, and then merge all chunks by Lemma 2 using the available space. Finally, uncompress  $S[1 \dots \frac{n}{3}]$ .
3. compress  $S[\frac{2n}{3} + 1 \dots n]$ , which is now sorted. Using Lemma 2, merge  $S[1 \dots \frac{n}{3}]$  with  $S[\frac{n}{3} + 1 \dots \frac{2n}{3}]$ . Finally uncompress.
4. once again, compress  $S[1 \dots \frac{n}{3}]$ , merge  $S[\frac{n}{3} + 1 \dots \frac{2n}{3}]$  with  $S[\frac{2n}{3} + 1 \dots n]$ , and uncompress.

Note that steps 2–4 are linear time. Then, we have the recursion  $T(n) = T(n/3) + O(n)$ , solving to  $T(n) = O(n)$ . Finally, we note that stability of the algorithm follows immediately from stability of classic radix sort and stability of merging.

*Practical experience.* The algorithm is surprisingly effective in practice. It can be implemented in about 150 lines of C code. Experiments with sorting 1-10 million 32-bit numbers on a Pentium machine indicate the algorithm is roughly 2.5 times slower than radix sort with additional memory, and slightly faster than quicksort (which is not even stable).

### 3 Unstable Sorting for Read-only Keys

#### 3.1. Simulating auxiliary bits

With the bit stealing technique [9], a bit of information is encoded in the relative order of a pair of elements with different keys: the pair is maintained in increasing order to encode a 0 and vice versa. The obvious drawback of this technique is that the cost of accessing a word of  $w$  encoded bits is  $O(w)$  in the worst case (no word-level parallelism). However, if we modify an encoded word with a series of  $l$  increments (or decrements) by 1, the total cost of the entire series is  $O(l)$  (see [2]).

To find pairs of distinct elements, we go from  $S$  to a sequence  $Z'Y'XY''Z''$  with two properties. (i) For any  $z' \in Z'$ ,  $y' \in Y'$ ,  $x \in X$ ,  $y'' \in Y''$  and  $z'' \in Z''$  we have that  $z' < y' < x < y'' < z''$ . (ii) Let  $m = \alpha \lceil n / \log n \rceil$ , for a suitable constant  $\alpha$ .  $Y'$  is composed by the element  $y'_m$  with rank  $m$  plus all the other elements equal to  $y'_m$ .  $Y''$  is composed by the element  $y''_m$  with rank  $n - m + 1$  plus all the other elements equal to  $y''_m$ . To obtain the new sequence we use the in-place, linear time selection and partitioning algorithms in [6, 7]. If  $X$  is empty, the task left is to sort  $Z'$  and  $Z''$ , which can be accomplished with any optimal, in-place mergesort (e.g. [10]). Let us denote  $Z'Y'$  with  $M'$  and  $Y''Z''$  with  $M''$ . The

$m$  pairs of distinct elements  $(M'[1], M''[1]), (M'[2], M''[2]), \dots, (M'[m], M''[m])$  will be used to encode information.

Since the choice of the constant  $\alpha$  does not affect the asymptotic complexity of the algorithm, we have reduced our problem to a problem in which we are allowed to use a special *bit memory* with  $O(n/\log n)$  bits where each bit can be accessed and modified in constant time but without word-level parallelism.

### 3.2. Simulating auxiliary memory for permuting

With the internal buffering technique [8], some of the elements are used as placeholders in order to simulate a working area and permute the other elements at lower cost. In our unstable sorting algorithm we use the basic idea of internal buffering in the following way. Using the selection and partitioning algorithms in [6, 7], we pass from the original sequence  $S$  to  $ABC$  with two properties. (i) For any  $a \in A$ ,  $b \in B$  and  $c \in C$ , we have that  $a < b < c$ . (ii)  $B$  is composed of the element  $b'$  with rank  $\lceil n/2 \rceil$  plus all the other elements equal to  $b'$ . We can use  $BC$  as an auxiliary memory in the following way. The element in the first position of  $BC$  is the *separator element* and will not be moved. The elements in the other positions of  $BC$  are placeholders and will be exchanged with (instead of being overwritten by) elements from  $A$  in any way the computation on  $A$  (in our case the sorting of  $A$ ) may require. The “emptiness” of any location  $i$  of the simulated working area in  $BC$  can be tested in  $O(1)$  time by comparing the separator element  $BC[1]$  with  $BC[i]$ : if  $BC[1] \leq BC[i]$  the  $i$ th location is “empty” (that is, it contains a placeholder), otherwise it contains one of the elements in  $A$ .

Let us suppose we can sort the elements in  $A$  in  $O(|A|)$  time using  $BC$  as working area. After  $A$  is sorted we use the partitioning algorithm in [6] to separate the elements equal to the separator element ( $BC[1]$ ) from the elements greater than it (the computation on  $A$  may have altered the original order in  $BC$ ). Then we just re-apply the same process to  $C$ , that is we divide it into  $A'B'C'$ , we sort  $A'$  using  $B'C'$  as working area and so forth. Clearly, this process requires  $O(n)$  time and when it terminates the elements are sorted. Obviously, we can divide  $A$  into  $p = O(1)$  equally sized subsequences  $A_1, A_2 \dots A_p$ , then sort each one of them using  $BC$  as working area and finally fuse them using the in-place, linear time merging algorithm in [10]. Since the choice of the constant  $p$  does not affect the asymptotic complexity of the whole process, we have reduced our problem to a new problem, in which we are allowed to use a special *exchange memory* of  $O(n)$  locations, where each location can contain *input elements only* (no integers or any other kind of data). Any element can be moved to and from any location of the exchange memory in  $O(1)$  time.

### 3.3. The reduced problem

By blending together the basic techniques seen above, we can focus on a reduced problem in which assumption (iv) is replaced by:

- (iv) *Only  $O(1)$  words of normal auxiliary memory and two kinds of special auxiliary memory are allowed:*
  - (a) *A random access bit memory  $\mathcal{B}$  with  $O(n/\log n)$  bits, where each bit can be accessed in  $O(1)$  time (no word-level parallelism).*

- (b) A random access exchange memory  $\mathcal{E}$  with  $O(n)$  locations, where each location can contain only elements from  $S$  and they can be moved to and from any location of  $\mathcal{E}$  in  $O(1)$  time.

If we can solve the reduced problem in  $O(n)$  time we can also solve the original problem with the same asymptotic complexity. However, the resulting algorithm will be unstable because of the use of the internal buffering technique with a large pool of placeholder elements.

### 3.4. The naive approach

Despite the two special auxiliary memories, solving the reduced problem is not easy. Let us consider the following naive approach. We proceed as in the normal bucket sorting: one bucket for each one of the  $n^\epsilon$  range values. Each bucket is a linked list: the input elements of each bucket are maintained in  $\mathcal{E}$  while its auxiliary data (e.g. the pointers of the list) are maintained in  $\mathcal{B}$ . In order to amortize the cost of updating the auxiliary data (each pointer requires a word of  $\Theta(\log n)$  bits and  $\mathcal{B}$  does not have word-level parallelism), each bucket is a linked list of *slabs* of  $\Theta(\log^2 n)$  elements each ( $\mathcal{B}$  has only  $O(n/\log n)$  bits). At any time each bucket has a partially full *head slab* which is where any new element of the bucket is stored. Hence, for each bucket we need to store in  $\mathcal{B}$  a word of  $O(\log \log n)$  bits with the position in the head slab of the last element added. The algorithm proceeds as usual: each element in  $S$  is sent to its bucket in  $O(1)$  time and is inserted in the bucket's head slab. With no word-level parallelism in  $\mathcal{B}$  the insertion in the head slab requires  $O(\log \log n)$  time. Therefore, we have an  $O(n \log \log n)$  time solution for the reduced problem and, consequently, an unstable  $O(n \log \log n)$  time solution for the original problem.

This simple strategy can be improved by dividing the head slab of a bucket into *second level slabs* of  $\Theta(\log \log n)$  elements each. As for the first level slabs, there is a partially full, second level head slab. For any bucket we maintain two words in  $\mathcal{B}$ : the first one has  $O(\log \log \log n)$  bits and stores the position of the last element inserted in the second level head slab; the second one has  $O(\log \log n)$  bits and stores the position of the last full slab of second level contained in the first level head slab. Clearly, this gives us an  $O(n \log \log \log n)$  time solution for the reduced problem and the corresponding unstable solution for the original problem. By generalizing this approach to the extreme, we end up with  $O(\log^* n)$  levels of slabs, an  $O(n \log^* n)$  time solution for the reduced problem and the related unstable solution for the original problem.

### 3.5. The pseudo pointers

Unlike bit stealing and internal buffering which were known earlier, the pseudo pointers technique has been specifically designed for improving the space complexity in integer sorting problems. Basically, in this technique a set of elements with distinct keys is used as a pool of pre-set, read-only pointers in order to simulate efficiently traversable and updatable linked lists. Let us show how to use this basic idea in a particular procedure that will be at the core of our optimal solution for the reduced problem.

Let  $d$  be the number of distinct keys in  $S$ . We are given two sets of  $d$  input elements with distinct keys: the sets  $\mathcal{G}$  and  $\mathcal{P}$  of *guides* and *pseudo pointers*,

respectively. The guides are given us *in sorted order* while the pseudo pointers form a sequence in arbitrary order. Finally, we are given a multiset  $\mathcal{I}$  of  $d$  input elements (i.e. two elements of  $\mathcal{I}$  can have equal keys). The procedure uses the guides, the pseudo pointers and the exchange memory to sort the  $d$  input elements of  $\mathcal{I}$  in  $O(d)$  time.

We use three groups of contiguous locations in the exchange memory  $\mathcal{E}$ . The first group  $H$  has  $n^\epsilon$  locations (one for each possible value of the keys). The second group  $L$  has  $n^\epsilon$  slots of two adjacent locations each. The last group  $R$  has  $d$  locations, the elements of  $\mathcal{I}$  will end up here in sorted order.  $H$ ,  $L$  and  $R$  are initially empty. We have two main steps.

*First.* For each  $s \in \mathcal{I}$ , we proceed as follows. Let  $p$  be the leftmost pseudo pointer still in  $\mathcal{P}$ . If the  $s$ th location of  $H$  is empty, we move  $p$  from  $\mathcal{P}$  to  $H[s]$  and then we move  $s$  from  $\mathcal{I}$  to the first location of  $L[p]$  (i.e. the first location of the  $p$ th slot of  $L$ ) leaving the second location of  $L[p]$  empty. Otherwise, if  $H[s]$  contains an element  $p'$  (a pseudo pointer) we move  $s$  from  $\mathcal{I}$  to the first location of  $L[p]$ , then we move  $p'$  from  $H[s]$  to the second location of  $L[p]$  and finally we move  $p$  from  $\mathcal{P}$  to  $H[s]$ .

*Second.* We scan the guides in  $\mathcal{G}$  from the smallest to the largest one. For a guide  $g \in \mathcal{G}$  we proceed as follows. If the  $g$ th location of  $H$  is empty then there does not exist any element equal to  $g$  among the ones to be sorted (and initially in  $\mathcal{I}$ ) and hence we move to the next guide. Otherwise, if  $H[G]$  contains a pseudo pointer  $p$ , there is at least one element equal to  $g$  among the ones to be sorted and this element is currently stored in the first location of the  $p$ th slot of  $L$ . Hence, we move that element from the first location of  $L[p]$  to the leftmost empty location of  $R$ . After that, if the second location of  $L[p]$  contains a pseudo pointer  $p'$ , there is another element equal to  $g$  and we proceed in the same fashion. Otherwise, if the second location of  $L[p]$  is empty then there are no more elements equal to  $g$  among the ones to be sorted and therefore we can focus on the next guide element.

Basically, the procedure is bucket sorting where the auxiliary data of the list associated to each bucket (i.e. the links among elements in the list) is *implemented by pseudo pointers in  $\mathcal{P}$*  instead of storing it explicitly in the bit memory (which lacks of word-level parallelism and is inefficient in access). It is worth noting that the buckets' lists implemented with pseudo pointers are spread over an area that is larger than the one we would obtain with explicit pointers (that is because each pseudo pointer has a key of  $\log n^\epsilon$  bits while an explicit pointer would have only  $\log d$  bits).

### 3.6. The optimal solution

We can now describe the algorithm, which has three main steps.

*First.* Let us assume that for any element  $s \in S$  there is at least another element with the same key. (Otherwise, we can easily reduce to this case in linear time: we isolate the  $O(n^\epsilon)$  elements that do not respect the property, we sort them with the in-place mergesort in [10] and finally we merge them after the other  $O(n)$  elements are sorted.) With this assumption, we extract from  $S$  two sets  $\mathcal{G}$  and  $\mathcal{P}$  of  $d$  input elements with distinct keys (this can be easily achieved in  $O(n)$  time using only the exchange memory  $\mathcal{E}$ ). Finally we sort  $\mathcal{G}$  with the optimal in-place mergesort in [10].

*Second.* Let  $S'$  be the sequence with the  $O(n)$  input elements left after the first step. Using the procedure in § 3.5 (clearly, the elements in the sets  $\mathcal{G}$  and  $\mathcal{P}$  computed in the first step will be the guides and pseudo pointers used in the procedure), we sort each block  $B_i$  of  $S'$  with  $d$  contiguous elements. After that, let us focus on the first  $t = \Theta(\log \log n)$  consecutive blocks  $B_1, B_2, \dots, B_t$ . We distribute the elements of these blocks into  $\leq t$  groups  $G_1, G_2, \dots$  in the following way. Each group  $G_j$  can contain between  $d$  and  $2d$  elements and is allocated in the exchange memory  $\mathcal{E}$ . The largest element in a group is its *pivot*. The number of elements in a group is stored in a word of  $\Theta(\log d)$  bits allocated in the bit memory  $\mathcal{B}$ . Initially there is only one group and is empty. In the  $i$ th step of the distribution we scan the elements of the  $i$ th block  $B_i$ . As long as the elements of  $B_i$  are less than or equal to the pivot of the first group we move them into it. If, during the process, the group becomes full, we select its median element and partition the group into two new groups (using the selection and partitioning algorithms in [6, 7]). When, during the scan, the elements of  $B_i$  become greater than the pivot of the first group, we move to the second group and continue in the same fashion. It is important to notice that the number of elements in a group (stored in a word of  $\Theta(\log d)$  bits in the bit memory  $\mathcal{B}$ ) is updated by increments by 1 (and hence the total cost of updating the number of elements in any group is linear in the final number of elements in that group, see [2]). Finally, when all the elements of the first  $t = \Theta(\log \log n)$  consecutive blocks  $B_1, B_2, \dots, B_t$  have been distributed into groups, we sort each group using the procedure in § 3.5 (when a group has more than  $d$  elements, we sort them in two batches and then merge them with the in-place, linear time merging in [10]). The whole process is repeated for the second  $t = \Theta(\log \log n)$  consecutive blocks, and so forth.

*Third.* After the second step, the sequence  $S'$  (which contains all the elements of  $S$  with the exclusion of the guides and pseudo pointers, see the first step) is composed by contiguous subsequences  $S'_1, S'_2, \dots$  which are *sorted* and contain  $\Theta(d \log \log n)$  elements each (where  $d$  is the number of distinct elements in  $S$ ). Hence, if we see  $S'$  as composed by contiguous runs of elements with the same key, we can conclude that the number of runs of  $S'$  is  $O(n / \log \log n)$ . Therefore  $S'$  can be sorted in  $O(n)$  time using the naive approach described in § 3.4 with only the following simple modification. As long as we are inserting the elements of a single run in a bucket, we maintain the position of the last element inserted in the head slab of the bucket in a word of auxiliary memory (we can use  $O(1)$  of them) instead of accessing the inefficient bit memory  $\mathcal{B}$  at any single insertion. When the current run is finally exhausted, we copy the position in the bit memory. Finally, we sort  $\mathcal{P}$  and we merge  $\mathcal{P}$ ,  $\mathcal{A}$  and  $S'$  (once again, using the sorting and merging algorithms in [10]).

### 3.7. Discussion: Stability and Read-only Keys

Let us focus on the reasons why the algorithm of this section is not stable. The major cause of instability is the use of the basic internal buffering technique in conjunction with large ( $\omega(\text{polylog}(n))$ ) pools of placeholder elements. This is clearly visible even in the first iteration of the process in § 3.2: after being used to permute  $A$  into sorted order, the placeholder elements in  $BC$  are left permuted in a completely arbitrary way and their initial order is lost.

## 4 Reducing Space in any RAM Sorting Algorithm

In this section, we consider the case of sorting integers of  $w = \omega(\log n)$  bits. We show a black box transformation from any sorting algorithm on the RAM to a stable sorting algorithm with the same time bounds which only uses  $O(1)$  words of additional space. Our reduction needs to modify keys. Furthermore, it requires randomization for large values of  $w$ .

We first remark that an algorithm that runs in time  $t(n)$  can only use  $O(t(n))$  words of space in most realistic models of computation. In models where the algorithm is allowed to write  $t(n)$  arbitrary words in a larger memory space, the space can also be reduced to  $O(t(n))$  by introducing randomization, and storing the memory cells in a hash table.

*Small word size.* We first deal with the case  $w = \text{polylog}(n)$ . The algorithm has the following structure:

1. sort  $S[1 \dots n/\log n]$  using in-place stable merge sort [10]. Compress these elements by Lemma 1 gaining  $\Omega(n)$  bits of space.
2. since  $t(n) = O(n \log n)$ , the RAM sorting algorithm uses at most  $O(t(n) \cdot w) = O(n \text{polylog}(n))$  bits of space. Then we can break the array into chunks of  $n/\log^c n$  elements, and sort each one using the available space.
3. merge the  $\log^c n$  sorted subarrays.
4. uncompress  $S[1 \dots n/\log n]$  and merge with the rest of the array by stable in-place merging [10].

Steps 1 and 4 take linear time. Step 2 requires  $\log^c n \cdot t(n/\log^c n) = O(t(n))$  because  $t(n)$  is convex and bounded in  $[n, n \log n]$ . We note that step 2 can always be made stable, since we can afford a label of  $O(\log n)$  bits per value.

It remains to show that step 3 can be implemented in  $O(n)$  time. In fact, this is a combination of the merging technique from Lemma 2 with an atomic heap [4]. The atomic heap can maintain a priority queue over  $\text{polylog}(n)$  elements with constant time per insert and extract-min. Thus, we can merge  $\log^c n$  lists with constant time per element. The atomic heap can be made stable by adding a label of  $c \log \log n$  bits for each element in the heap, which we have space for. The merging of Lemma 2 requires that we keep track of  $O(k/\alpha)$  subarrays, where  $k = \log^c n$  was the number of lists and  $\alpha = 1/\text{polylog}(n)$  is fraction of additional space we have available. Fortunately, this is only  $\text{polylog}(n)$  values to record, which we can afford.

*Large word size.* For word size  $w \geq \log^{1+\varepsilon} n$ , the randomized algorithm of [1] can sort in  $O(n)$  time. Since this is the best bound one can hope for, it suffices to make this particular algorithm in-place, rather than give a black-box transformation. We use the same algorithm from above. The only challenge is to make step 2 work: sort  $n$  keys with  $O(n \text{polylog}(n))$  space, even if the keys have  $w > \text{polylog}(n)$  bits.

We may assume  $w \geq \log^3 n$ , which simplifies the algorithm of [1] to two stages. In the first stage, a signature of  $O(\log^2 n)$  bits is generated for each input value (through hashing), and these signatures are sorted in linear time. Since we

are working with  $O(\log^2 n)$ -bit keys regardless of the original  $w$ , this part needs  $O(n \text{ polylog}(n))$  bits of space, and it can be handled as above.

From the sorted signatures, an additional pass extracts a subkey of  $w/\log n$  bits from each input value. Then, these subkeys are sorted in linear time. Finally, the order of the original keys is determined from the sorted subkeys and the sorted signatures.

To reduce the space in this stage, we first note that the algorithm for extracting subkeys does not require additional space. We can then isolate the subkey from the rest of the key, using shifts, and group subkeys and the remainder of each key in separate arrays, taking linear time. This way, by extracting the subkeys instead of copying them we require no extra space. We now note that the algorithm in [1] for sorting the subkeys also does not require additional space. At the end, we recompose the keys by applying the inverse permutation to the subkeys, and shifting them back into the keys.

Finally, sorting the original keys only requires knowledge of the signatures and *order* information about the subkeys. Thus, it requires  $O(n \text{ polylog}(n))$  bits of space, which we have. At the end, we find the sorted order of the original keys and we can implement the permutation in linear time.

*Acknowledgements.* Our sincere thanks to Michael Riley and Mehryar Mohri of Google, NY who, motivated by manipulating transitions of large finite state machines, asked if bucket sorting can be done in-place in linear time.

## References

1. Arne Andersson, Torben Hagerup, Stefan Nilsson, and Rajeev Raman. Sorting in linear time? *Journal of Computer and System Sciences*, 57(1):74–93, August 1998.
2. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 2001.
3. M. L. Fredman and D. E. Willard. Surpassing the information theoretic bound with fusion trees. *J. Comput. System Sci.*, 47:424–436, 1993.
4. M. L. Fredman and D. E. Willard. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. *J. Comput. System Sci.*, 48(3):533–551, 1994.
5. Yijie Han and Mikkel Thorup. Integer sorting in  $O(n\sqrt{\log \log n})$  expected time and linear space. In *FOCS*, pages 135–144. IEEE Computer Society, 2002.
6. Jyrki Katajainen and Tomi Pasanen. Stable minimum space partitioning in linear time. *BIT*, 32(4):580–585, 1992.
7. Jyrki Katajainen and Tomi Pasanen. Sorting multisets stably in minimum space. *Acta Informatica*, 31(4):301–313, 1994.
8. M. A. Kronrod. Optimal ordering algorithm without operational field. *Soviet Math. Dokl.*, 10:744–746, 1969.
9. J. Ian Munro. An implicit data structure supporting insertion, deletion, and search in  $O(\log^2 n)$  time. *Journal of Computer and System Sciences*, 33(1):66–74, 1986.
10. Jeffrey Salowe and William Steiger. Simplified stable merging tasks. *Journal of Algorithms*, 8(4):557–571, December 1987.