Optimal In-Place Sorting of Vectors and Records

Gianni Franceschini and Roberto Grossi

Dipartimento di Informatica, Università di Pisa
Largo Bruno Pontecorvo 3, 56127 Pisa, Italy
{francesc,grossi}@di.unipi.it

Abstract. We study the problem of determining the complexity of optimal comparison-based in-place sorting when the key length, k, is not a constant. We present the first algorithm for lexicographically sorting n keys in $O(nk+n\log n)$ time using $O(1)$ auxiliary data locations, which is *simultaneously* optimal in time *and* space.

1 Introduction

We study the computational complexity of the classical problem of comparison-based sorting by considering the case in which the keys are of non-constant length, k. We aim at minimizing simultaneously the time and space bounds under the assumption that the keys are vectors $x \in \Sigma^k$ of k scalar components over a totally ordered, possibly unbounded set Σ. Denoting the ith scalar component of vector x by $x(i)$ for $1 \leq i \leq k$, we indicate the vector's *chunks* by $x(i,j)$, which are the contiguous portions of x consisting of $x(i)$, $x(i+1)$, …, $x(j)$, where $1 \leq i \leq j \leq k$. The lexicographic (or alphabetic) order, $x \leq y$, is defined in terms of the scalar components: either $x(1) < y(1)$ or recursively $x(2,k) \leq y(2,k)$ for $x(1) = y(1)$. The model easily extends to k-field records in $\Sigma_1 \times \Sigma_2 \times \cdots \times \Sigma_k$, but we prefer to keep the notation simple.

We are given a set $\mathscr{V} \subseteq \Sigma^k$ of n vectors stored in n *vectorial locations*, one vector of $\mathscr{V}$ per location. We permit two kinds of operations on the vector locations: (1) exchange any two vectors in $O(k)$ time; (2) access the ith scalar component of any two vectors for comparison purposes in $O(1)$ time. Hence, determining the lexicographic order of any two vectors from scratch takes $O(k)$ time. We are also given a number of *auxiliary locations*, each location storing one integer of $O(\log n)$ bits. We employ the standard repertoire of RAM instructions on the auxiliary locations, with $O(1)$ time per operation.

The model resulting from the above rules naturally extends the comparison model to keys of non-constant length. (We obtain the comparison model by fixing $k = 1$.) We are interested in exploring algorithms using the minimal number of auxiliary locations, referring to the model using just $O(1)$ auxiliary locations as the *in-place model for vectors*. This model is useful for studying, in an abstract way, the complexity of in-place sorting and searching for a variety of keys: k-length strings, k-field records, k-dimensional points, k-digit numbers, etc.

One significant example is how to perform in-place searching on a set $\mathscr{V}$ of n vectors. With sophisticated techniques for proving upper and lower bounds on the complexity of searching $\mathscr{V}$ in lexicographic order, Andersson, Hagerup, Håstad and Petersson have proved in [1] that it requires

$$\Theta\left(\frac{k\log\log n}{\log\log(4+\frac{k\log\log n}{\log n})}+k+\log n\right)$$

time. This bound is worse than $\Theta(k+\log n)$, obtained by searching $\mathscr{V}$ *plus* $O(n)$ auxiliary locations (e.g., Manber and Myers [18]). Using permutations other than those resulting from sorting is a way to reach optimality: Franceschini and Grossi [10] have shown that for any set $\mathscr{V}$ of n vectors in lexicographic order, there exists a permutation of them allowing for $\Theta(k+\log n)$ search time using $O(1)$ auxiliary data locations.

In-place sorting is an even more intriguing example in this scenario. Any optimal in-place sorting algorithm for constant-sized keys can be turned into an $O(nk\log n)$-time in-place algorithm for vectors, losing optimality in this way. The lower bound of $\Omega(nk+n\log n)$ time easily derives from decision trees [14]. If the number of comparison is to be minimized, the best up-to-date result for in-place sorting is $n\log n+O(nk\log^* n)$ scalar comparisons and $n\log n+O(nk)$ vector exchanges by Munro and Raman [20]. Since each vector exchange takes $O(k)$ time, the time complexity sums up to $O(nk^2+nk\log n)$. For the same reason, the multikey Quicksort analyzed by Bentley and Sedgewick [4] yields a non-optimal algorithm of cost $O(nk\log n)$ when adapted to run in the in-place model for vectors, since it requires $O(n\log n)$ vector exchanges. The original version of the algorithm takes $O(nk+n\log n)$ time since it can exploit $O(n)$ auxiliary locations to store the pointers to the vectors. It exchanges the pointers rather than the vectors, following the *address table sorting* suggested in Knuth [14, p.74]. Recently, Franceschini and Geffert [9] have devised an optimal in-place algorithm for constant-sized keys with $O(n)$ data moves. Subsequent results by Franceschini [7, 8] have shown how to achieve cache-obliviousness or stableness for in-place sorting. However, the $O(k)$-time cost of each vector comparison makes these methods non-optimal in our setting. The bit encoding for vectors in Franceschini and Grossi [10] cannot help either, as it assumes that vectors are initially sorted while this is actually the major goal in this paper.

The above discussion highlights the fact that the known algorithms, to the best of our knowledge, are unable to simultaneously achieve time optimality and space optimality for sorting vectors (in place). Our main result is that of obtaining the first optimal bounds for sorting an arbitrary set of n vectors in place, taking $\Theta(nk+n\log n)$ time and using $O(1)$ auxiliary locations. An implication of our result is that we can provide optimal in-place preprocessing for efficient in-place searching [1, 10–12, 15] when the vectors are initially arranged in any arbitrary order, with a preprocessing cost of $O(nk+n\log n)$ time. Another implication is that sorting bulky records can be done optimally in place by exchanging them directly without using the $O(n)$ auxiliary locations required by Knuth's address table sorting.

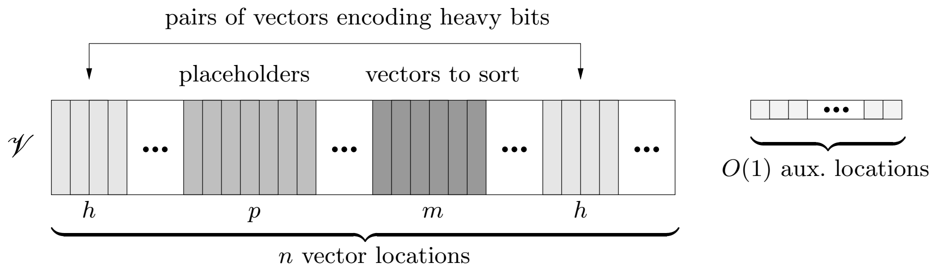

Fig. 1. An instance of $GVSP\{m, p, h\}$. Each of the n vector locations in $\mathscr{V}$ contains one vector. Each of the $O(1)$ auxiliary locations contains one integer of $O(\log n)$ bits.

2 High-Level Description

We present our in-place sorting algorithm for vectors in a top-down fashion. We describe how to reduce the original problem to a sequence of simpler sorting problems to solve. In our description, we identify the n input vectors in $\mathscr{V}$ with their vectorial locations. At the beginning of the computation, $\mathscr{V}[i]$ represents the ith vectorial location, for $1 \leq i \leq n$, and contains the ith input vector. At the end of the computation, $\mathscr{V}[i]$ contains the vector of rank i after the sorting (ties for equal vectors are broken arbitrarily). During the intermediate steps, we solve several instances of a *general vector sorting problem*, denoted $GVSP\{m, p, h\}$ (see Figure 1). Given n vectors in $\mathscr{V}$, we refer to $GVSP\{m, p, h\}$ as the problem of sorting a subset of m contiguous vectors in $\mathscr{V}$, using

- $O(1)$ auxiliary locations,
- p vectors as *placeholders* taken from a contiguous subsequence of p locations in $\mathscr{V}$,
- h *heavy* bits suitably encoded by h pairs of vectors taken from two contiguous subsequences, each consisting of h locations in $\mathscr{V}$,

under the requirement that $m + p + 2h \leq n$ and that the four subsequences of h, p, m, and h vector locations, respectively, are pairwise disjoint as shown in Figure 1. Placeholders and heavy bits are defined in the rest of the section.

The general notation of $GVSP\{m, p, h\}$ is useful for expressing the various sorting instances that we get by reducing our initial problem, $GVSP\{n, 0, 0\}$, to simpler problems (with suitable values of m, p and h). Some basic instances occur just a constant number of times in the reduction and are easy to solve.

Lemma 1. *Any instance of $GVSP\{O(n/\log n), 0, 0\}$ takes $O(nk)$ time.*

Proof. We employ the in-place mergesort of [21] and pay a slowdown of $O(k)$ in the time complexity, since we run it on $O(n/\log n)$ vectors, each of length k. The cost is $O(k \times (n/\log n) \log(n/\log n)) = O(nk)$ time.

We now present the high-level structure of our reduction. In the following, for any two vectors x and y, we denote the length of their longest common prefix by $lcp(x, y) = \max\bigl(\{0\} \cup \{1 \leq \ell \leq k : x(1, \ell) = y(1, \ell)\}\bigr)$.

Heavy bits (Section 3). To begin with, we reduce an instance of $GVSP\{n, 0, 0\}$ to a suitable instance of $GVSP\{n-o(n), 0, O(n/\log^2 n)\}$ plus a constant number of instances of $GVSP\{O(n/\log n), 0, 0\}$. We partition in place the sequence $\mathscr{V}$ into contiguous subsequences $\mathscr{L}$, $\mathscr{M}$, and $\mathscr{R}$, such that for each $x \in \mathscr{L}$, $y \in \mathscr{M}$ and $z \in \mathscr{R}$, we have $x \leq y \leq z$. Moreover, the number of vectors in $\mathscr{L}$ equals that of $\mathscr{R}$, namely, $|\mathscr{L}| = |\mathscr{R}| = O(n/\log n)$. Assuming that $\max \mathscr{L} \neq \min \mathscr{R}$ (otherwise sorting is trivial), we consider the pairs $P = \{\langle \mathscr{L}[i], \mathscr{R}[i] \rangle$, for $1 \leq i \leq |\mathscr{L}|\}$. Note that for every pair $\langle x, y \rangle \in P$, vectors x and y are distinct ($x < y$) and their first mismatching scalar component is at position $lcp(x, y) + 1$. Based on this observation we identify a subset H of the pairs in P satisfying two constraints:

1. $|H| = \Omega(n/\log^2 n)$.
2. There exists an interval $[l, r] \subseteq [1, k]$ of size $\max\{1, k/\log n\}$, such that $lcp(x, y) + 1 \in [l, r]$ for every pair $\langle x, y \rangle \in H$.

Under constraints 1–2, we can use the vectors in H for implicitly representing $O(n/\log^2 n)$ bits, called *heavy bits*, so that decoding one heavy bit requires $O(1 + k/\log n)$ time while encoding it takes $O(k)$ time. Let us see why do we need these bits. When designing an optimal in-place algorithm, the constraint on using just $O(1)$ auxiliary locations, namely, $O(\log n)$ extra bits of information, is rather stringent. Fortunately, permutations of the keys encode themselves further bits of informations. Potentially, we are plenty of $\log h!$ bits by permuting h distinct keys. Based on this idea, bit stealing [19] is a basic technique for implicitly encoding up to h bits of information by pairwise permuting h pairs of keys. In its original design, the technique encodes a bit with each distinct pair of keys x and y, such that $x < y$. The bit is of value 0 if x occurs *before* y in the permutation; it's of value 1 if x occurs *after* y in the permutation. The main drawback of this technique in our setting is that we need $O(k)$ time for encoding and decoding one bit since x and y are vectors. As we shall see, we will require an amortized number of $O(1)$ encoded bits and $O(\log n)$ decoded bits per vector, so that we have to decrease the cost of decoding to $O(1 + k/\log n)$ to stay within the claimed bounds.

At this stage, sorting $\mathscr{V}$ reduces to sorting $\mathscr{M}$ as an instance of $GVSP\{n - o(n), 0, O(n/\log^2 n)\}$. After that, it also reduces to sorting $\mathscr{L}$ and $\mathscr{R}$ as instances of $GVSP\{O(n/\log n), 0, 0\}$ solved by Lemma 1.

Buffering and session sorting (Sect. 4). We solve $GVSP\{n-o(n), 0, O(n/\log^2 n)\}$ reducing to $O(\log n)$ instances of $GVSP\{O(n/\log n), O(n/\log n), O(n/\log^2 n)\}$ and to $O(1)$ instances of $GVSP\{O(n/\log n), 0, 0\}$ solved by Lemma 1. We logically divide $\mathscr{M}$ into contiguous subsequences $\mathscr{M}_1, \ldots, \mathscr{M}_{s-1}, \mathscr{M}_s$, where $|\mathscr{M}_2| = \cdots = |\mathscr{M}_{s-1}| \leq |\mathscr{M}_s| = O(n/\log n)$ and $s = O(\log n)$. Moreover, $|\mathscr{M}_1| = O(n/\log n)$ has a sufficiently large multiplicative constant, so that $\mathscr{M}_1$ can host enough vectors playing the role of placeholders. With reference to Figure 1, we sort the $m = O(n/\log n)$ vectors in each individual $\mathscr{M}_i$, $i \neq 1$, using the $p = O(n/\log n)$ placeholders in $\mathscr{M}_1$ and the $h = O(n/\log^2 n)$ heavy bits encoded by the pairs in $H \subseteq \mathscr{L} \times \mathscr{R}$.

Let us first give some motivation for using the placeholders while sorting. Having just n vector locations, we cannot rely on a temporary area of vector locations for efficiently permuting the vectors with a few moves. We therefore exploit a virtual form of temporary area using the internal buffering technique of Kronrod [16]. We designate the vectors in $\mathscr{M}_1$ as placeholders for "free memory" since we do not care to sort them at this stage. Hence, they can be scrambled up without interplaying with the sorting process that is running on a given $\mathscr{M}_i$, $i \neq 1$. When we need to move a vector of $\mathscr{M}_i$ to the temporary area, we simulate this fact by exchanging the vector with a suitable placeholder of $\mathscr{M}_1$. At the same time, we should guarantee that this exchange is somehow reversible, allowing us to put the placeholders back to the "free memory" in $\mathscr{M}_1$ without perturbing the sorting obtained for $\mathscr{M}_i$, $i \neq 1$.

Assuming to have obtained each of $\mathscr{M}_2, \ldots, \mathscr{M}_{s-1}, \mathscr{M}_s$ in lexicographic order, we still have to merge them using the heavy bits in H and the placeholders in $\mathscr{M}_1$. It turns out that this task is non-trivial to be performed. Just to have a rough idea, let us imagine to run the 2-way in-place mergesort for $O(\log s) = O(\log \log n)$ passes on them. This would definitively give a non-optimal time cost for the vectors since the number of vector exchanges would be $\omega(n)$, losing optimality in this way. We introduce a useful generalization of the technique in [7, 16], thus obtaining what we call *session sorting*. Let us assume that the vectors are distinct (we shall disregard this assumption in Section 4).

The main goal of session sorting is that of rearranging all the vectors in $\mathscr{M}_2, \ldots, \mathscr{M}_{s-1}, \mathscr{M}_s$, so that they are not too far from their final destination. If any such vector has rank r among all the other vectors in $\mathscr{M}_2\mathscr{M}_3 \cdots \mathscr{M}_{s-1}\mathscr{M}_s$, and occupies a position $g > r$ after session sorting, we guarantee that $g - r \leq |\mathscr{M}_i|$. (Note that we do not claim anything regarding the case $g \leq r$.) Using this strong property, we show that the sequence of 2-way in-place operations for merging $\mathscr{M}_i$ and $\mathscr{M}_{i+1}$ for $i = 2, 3, \ldots, s-1$ (in this order) yields the sorted sequence. (We remark that this is not generally true if we do not apply session sorting.) As a result, the entire sequence $\mathscr{M}_2\mathscr{M}_3 \cdots \mathscr{M}_{s-1}\mathscr{M}_s$ is in lexicographic order with a linear number of moves.

What remains to do is sorting $\mathscr{L}$, $\mathscr{M}_1$, and $\mathscr{R}$ individually as instances of $GVSP\{O(n/\log n), 0, 0\}$ by Lemma 1. Merging them in place with the rest of sorted vectors is a standard task giving $\mathscr{V}$ in sorted order. Hence, we are left with an instance of $GVSP\{O(n/\log n), O(n/\log n), O(n/\log^2 n)\}$, which corresponds to sorting a given $\mathscr{M}_i$, $i \neq 1$, using the placeholders initially hosted in $\mathscr{M}_1$ and the heavy bits encoded by the pairs in H.

Sorting each $\mathscr{M}_i$ individually (Section 5). We describe this stage in general terms. For a given $i \neq 1$, let $\mathscr{M}' = \mathscr{M}_i$ and $\mathscr{M}_B = \mathscr{M}_1$, for the instance of $GVSP\{|\mathscr{M}'|, |\mathscr{M}_B|, |H|\}$ that we are going to solve with the heavy bits in H (see Figure 1). Using $\mathscr{M}_B$ as a "free memory" area, we simulate the sorting of the $m' = |\mathscr{M}'|$ vectors by inserting them into a suitable structure that is incrementally built inside $\mathscr{M}_B$. Each insertion of a vector $x \in \mathscr{M}'$ into the internal structure of $\mathscr{M}_B$ exchanges x with a placeholder. After each such exchange we permute some of the vectors inside $\mathscr{M}_B$, so as to dynamically maintain a set of

$O(m'/\log^2 m')$ pivot vectors in the internal structure. The pivots have buckets associated inside $\mathscr{M}_B$ for the purpose of distributing the non-pivot vectors inserted up to that point, like in distribution sort. Each bucket contain $\Theta(\log^2 m')$ vectors that are kept *unsorted* to minimize the number of vector exchanges needed to maintain the internal structure of $\mathscr{M}_B$.

The pivots inside $\mathscr{M}_B$ are kept searchable by a suitable blend of the techniques in [10, 13, 18], requiring to decode $O(\log n)$ heavy bits per inserted vector (which is fine since decoding takes $O(1 + k/\log n)$ time). In particular, we logically divide each vector x into a concatenation of $O(\log m') = O(\log n)$ equally sized chunks. We only store the *lcp* information for the chunks considered as "meta-characters," thus obtaining an approximation of the *lcp* information for the vectors. After that the distribution completes by inserting all the vectors of $\mathscr{M}'$ into the internal structure of $\mathscr{M}_B$, we sort the buckets individually by using a constant number of recursive iterations of session sorting whose parameters are suitably adapted to the buckets' size. The base case of the recursion consists in solving $GVSP\{O(\sqrt{\log m'}), O(\sqrt{\log m'}), 0\}$, for which we design an optimal ad-hoc algorithm. After completing the individual sorting of the buckets, which still reside in $\mathscr{M}_B$, we exchange them with the placeholders that were temporarily moved to $\mathscr{M}'$. We place back the sorted buckets and their pivots to $\mathscr{M}'$ according to their relative order, which means that the m' vectors in $\mathscr{M}'$ are in lexicographic order. Since this stage is quite full of technicalities, we give more details in the full paper.

Known tools. We use a few optimal algorithmic tools for atomic keys: in-place stable mergesort and in-place merge [21]; in-place selection for order statistics [17]. We apply these algorithms to vectors in a straightforward way by paying a slowdown of $O(k)$ per elementary step in their time complexity. We also use Hirschberg's linear scanning method [11] for searching in place a set of n vectors of length k, with the simple bound of $O(k + n)$ time. We go through the convention that the last lowercase letters—$\ldots, x, y, w, z$—denote vectors and the middle ones—$\ldots, i, j, k, l, \ldots$—are auxiliary indices or parameters.

3 Heavy Bits

We detail how to reduce the problem of sorting in place n vectors—an instance of $GVSP\{n, 0, 0\}$—to an instance of $GVSP\{n - o(n), 0, O(n/\log^2 n)\}$ plus a constant number of instances of $GVSP\{O(n/\log n), 0, 0\}$. (The notation for $GVSP\{m, p, h\}$ is defined in Section 2 and illustrated in Figure 1.) We recall that we partition in place the sequence $\mathscr{V}$ into $\mathscr{L}$, $\mathscr{M}$, and $\mathscr{R}$, where $|\mathscr{L}| = |\mathscr{R}| = p = O(n/\log n)$. We obtain this partition by performing order statistics in place [17] so as to identify the pth and the $(n - p + 1)$st elements of $\mathscr{V}$ in $O(nk)$ time. In the rest of the paper we assume that $w_L \neq w_R$; otherwise, $\mathscr{M}$ is made up of all equal vectors and sorting is trivially solved by applying Lemma 1 to $\mathscr{L}$ and $\mathscr{R}$.

Let us consider the set of pairs of vectors thus obtained, $P = \{\langle \mathscr{L}[i], \mathscr{R}[i] \rangle : 1 \leq i \leq p\} \subseteq \mathscr{L} \times \mathscr{R}$. Let us conceptually divide each of these vectors into chunks of $k/\ell = O(1 + k/\log n)$ scalar components, where $\ell = \min\{k, \log n\}$. We index these chunks from 1 to ℓ, in the order of their appearance inside the vector. We assign an integer label j to each pair $\langle \mathscr{L}[i], \mathscr{R}[i] \rangle$, where $1 \le j \le \ell$ and $1 \le i \le p$. Since $\mathscr{L}[i] < \mathscr{R}[i]$ by construction, label j is the index of the chunk containing the first mismatching position for $\mathscr{L}[i]$ and $\mathscr{R}[i]$; that is, it satisfies $(j-1)\,k/\ell \le lcp(\mathscr{L}[i], \mathscr{R}[i]) < j\,k/\ell$. By the pigeon principle, there must exist a value of j for which at least $p/\ell = \Omega(n/\log^2 n)$ pairs in P are labeled j. We can identify that value by running at most ℓ in-place scans of $\mathscr{L}$ and $\mathscr{R}$, with an overall cost of $O(\ell \times pk) = O(nk)$ time. With a further scan of $\mathscr{L}$ and $\mathscr{R}$, we single out $h = \Theta(p/\ell) = \Theta(n/\log^2 n)$ pairs in P that have label j, moving them in place at the beginning of $\mathscr{L}$ and $\mathscr{R}$, respectively. Consequently, we identify these vectors in the first h locations in $\mathscr{L}$ and $\mathscr{R}$ by a set of pairs, denoted H:

– $H \subseteq P$ and $|H| = h = \Theta(n/\log^2 n)$;
– $H = \{\langle \mathscr{L}[i], \mathscr{R}[i] \rangle \,:\, 1 \le i \le h\}$ after the preprocessing;
– there exists $j \in [1, \ell]$ such that $(j-1)\,k/\ell \le lcp(x, y) < j\,k/\ell$ for every pair $\langle x, y \rangle \in H$.

We steal bits in H using the knowledge of j as follows. For $1 \le i \le h$, we encode the ith bit of value 1 by exchanging $\mathscr{L}[i]$ and $\mathscr{R}[i]$ in $O(k)$ time; namely, $\mathscr{L}[i]$ occupies now position i inside $\mathscr{R}$ and $\mathscr{R}[i]$ does it inside $\mathscr{L}$. If the bit is 0, we leave them at their position (no exchange). In order to decode the ith bit, we only compare their jth chunk to find their mismatching position in the interval $[(j-1)\,k/\ell+1, j\,k/\ell]$. In this way, we can establish whether or not the two vectors have been exchanged during encoding (and so we decode either 0 or 1). Decoding performs at most k/ℓ scalar comparisons and thus takes $O(1 + k/\log n)$ time. The non-constant cost of bit stealing motivates our choice of referring to these bits as *heavy*.

Lemma 2. *We can encode $h = \Theta(n/\log^2 n)$* heavy bits *by the pairwise permutation of vectors in $H \subseteq \mathscr{L} \times \mathscr{R}$. Encoding one bit requires $O(k)$ time while decoding it requires $O(1+k/\log n)$ time. Preprocessing requires $O(nk)$ time using $O(1)$ auxiliary locations.*

We keep $\mathscr{L}$ and $\mathscr{R}$ unsorted for encoding bits until the end of the algorithm. At that point, we can in-place sort $\mathscr{L}$ and $\mathscr{R}$ by Lemma 1, in $O(nk)$ time. Consequently we are left with the problem of sorting $\mathscr{M}$.

Lemma 3. *There exists an $O(nk)$-time reduction from $GVSP\{n, 0, 0\}$ to $GVSP\{n - o(n), 0, O(n/\log^2 n)\}$, using $O(1)$ auxiliary locations.*

4 Buffering and Session Sorting

In this section, we detail how to sort the vectors in $\mathscr{M}$, which is an instance of $GVSP\{n - o(n), 0, O(n/\log^2 n)\}$. We logically divide $\mathscr{M}$ into contiguous subsequences $\mathscr{M}_1, \ldots, \mathscr{M}_{s-1}, \mathscr{M}_s$, called *blocks*, where $|\mathscr{M}_2| = \cdots = |\mathscr{M}_{s-1}| \le |\mathscr{M}_s| = O(n/\log n)$ and $s = O(\log n)$. In the following, we assume without loss of generality that $|\mathscr{M}_s| = |\mathscr{M}_{s-1}|$ (if not, we treat $\mathscr{M}_s$ differently, applying Lemma 1 to it). We remark that only a constant number of blocks can be sorted with the bounds of Lemma 1. Hence we should proceed otherwise. We designate the $O(n/\log n)$ vectors in $\mathscr{M}_1$, for a sufficiently large multiplicative constant, to act as placeholders [16]. In this way we obtain $O(\log n)$ instances of $GVSP\{O(n/\log n), O(n/\log n), O(n/\log^2 n)\}$, plus a constant number of instances of $GVSP\{O(n/\log n), 0, 0\}$ solved by Lemma 1.

We are still missing a crucial part of the reduction performed at this stage, namely, how to obtain *all* the vectors in $\mathscr{M}_2\mathscr{M}_3 \cdots \mathscr{M}_{s-1}\mathscr{M}_s$ in lexicographic order. We introduce the *right-bounded* permutations, since they rearrange the vectors so that each vector cannot occupy a position beyond a bounded distance to the right of its final position in the sorted sequence. As we will prove, the net effect of the right-bounded permutation is that we can simulate the in-place merging scheme by the following scheme: IN-PLACE-MERGE$(\mathscr{M}_2, \mathscr{M}_3)$; IN-PLACE-MERGE$(\mathscr{M}_3, \mathscr{M}_4)$; $\ldots$; IN-PLACE-MERGE$(\mathscr{M}_{s-1}, \mathscr{M}_s)$. We describe this permutation in general terms as it is of independent interest.

4.1 Right-bounded permutations

We are given three positive integers m, p, q, such that q divides p and p divides m, satisfying

$$\left(\frac{m}{p} - 1\right) \times (q-1) \le p. \tag{1}$$

Given a sequence $\mathscr{B}$ of m vectors, we logically divide it into m/q sub-blocks of q vectors each, denoted $\mathscr{S}_1, \ldots, \mathscr{S}_{m/q}$. The sub-blocks are grouped into blocks of p/q sub-blocks each, thus logically dividing $\mathscr{B}$ into m/p blocks of p vectors each, denoted $\mathscr{B}_1, \ldots, \mathscr{B}_{m/p}$. A right-bounded permutation is the arrangement of the vectors in $\mathscr{B}$ resulting from steps P1–P2, with steps P3–P4 yielding the sequence in lexicographic order:

P1. For $j = 1, \ldots, m/p$, sort each block $\mathscr{B}_j$ individually.
P2. Sort *stably* the m/q sub-blocks $\mathscr{S}_1, \ldots, \mathscr{S}_{m/q}$ according to their first vector (i.e., comparisons are driven by the minimum vector in each sub-block, and the rest of the vectors are considered as "satellite data").
P3. For $j = 1, \ldots, m/p$, sort each block $\mathscr{B}_j$ individually (note that the content of the blocks changed!).
P4. For $j = 1, \ldots, m/p - 1$, merge the vectors contained in blocks $\mathscr{B}_j$ and $\mathscr{B}_{j+1}$.

Lemma 4. *For each vector $\mathscr{B}[i]$, $1 \le i \le m$, let g_i be the number of vectors $\mathscr{B}[j] > \mathscr{B}[i]$ such that $1 \le j < i$* right after *steps P1–P2. Then*

$$g_i \le \left(\frac{m}{p} - 1\right) \times (q-1). \tag{2}$$

Proof. Let us consider the arrangement of the vectors in $\mathscr{B}$ right after steps P1–P2. In order to prove equation (2), we need to consider the intermediate arrangement of the vectors in $\mathscr{B}$ *after* step P1 and *before* step P2. Recall that we logically divide $\mathscr{B}$ into blocks and sub-blocks, indexing the blocks from 1 to m/p. We assign a unique type to each block based on its index, namely, block $\mathscr{B}_t$ is assigned *type* t, where $1 \leq t \leq m/p$, since it is the tth block in $\mathscr{B}$. For the intermediate arrangement above, we say that a vector has type t if it belongs to $\mathscr{B}_t$ (recall that $\mathscr{B}_t$ is sorted). We can assign type t to the sub-blocks of each $\mathscr{B}_t$ in the same manner, since each sub-block contains vectors of the same type t by construction. Hence the type of a sub-block is well defined. We refer to the first vector of each sub-block, which is also the minimum in it, as the *header* of the sub-block.

Let us now resume the arrangement of the vectors in $\mathscr{B}$ *right after* steps P1–P2. Consider a generic vector $\mathscr{B}[i]$ belonging to a sub-block, say $\mathscr{S}'$ of type t', and let g_i be defined as above. We give an upper bound to g_i so as equation (2) holds. Specifically, we count the maximum number of vectors contributing to g_i. Let us discuss them by their type. By the stability of the sorting process in step P2, we know that the vectors of type t' have maintained the relative order they had in the intermediate arrangement (after step P1 and before step P2) and so they cannot contribute to g_i.

Let x be the header of the sub-block $\mathscr{S}'$ containing $\mathscr{B}[i]$. Let us evaluate the contribution to g_i for the vectors of type $t'' \neq t'$. Consider all sub-blocks of type t'': we claim that at most one of them, say $\mathscr{S}''$, can contain vectors contributing to g_i. Precisely, $\mathscr{S}''$ is the sub-block of type t'' having the largest header less than or equal to x. Let $y \leq x$ be the header of $\mathscr{S}''$ and z be one of such contributing vectors in $\mathscr{S}''$. Sub-block $\mathscr{S}''$ is laid out before $\mathscr{S}'$ by construction but $z > \mathscr{B}[i]$ by definition of g_i. Note that there can be at most $q-1$ such vectors z in $\mathscr{S}''$. For any other sub-block of type t'', we show that its vectors cannot contribute to g_i. Since the block of type t'' is sorted after step P1, there are two possibilities for its sub-blocks $\mathscr{S}''' \neq \mathscr{S}''$: (a) $\mathscr{S}'''$ contains all vectors that are less than or equal to $y \leq x$ (i.e., $\mathscr{S}'''$ is laid out before $\mathscr{S}''$); they do not contribute to g_i by transitivity since $x \leq \mathscr{B}[i]$. (b) $\mathscr{S}'''$ contains all vectors that are greater than or equal to $z > \mathscr{B}[i] \geq x$ (i.e., $\mathscr{S}'''$ is laid out after $\mathscr{S}''$); they do not contribute because the header of $\mathscr{S}'''$ is strictly larger than x by transitivity and so $\mathscr{S}'''$ is laid out after $\mathscr{S}'$. Summing up, the total contribution to g_i for the vectors of type $t'' \neq t'$ is at most $q-1$ (a subset of the vectors in $\mathscr{S}''$). Since there are $\frac{m}{p} - 1$ different types other than t', we obtain the upper bound for equation (2).

Theorem 1. *After steps P1–P4, the sequence $\mathscr{B}$ is sorted.*

Proof. We proceed by induction on the length of prefixes of blocks in $\mathscr{B}$. The base case is obvious, as we know that $\mathscr{B}_1$ is sorted by step P3. Let us assume that the jth prefix of blocks $\mathscr{B}_1\mathscr{B}_2\cdots\mathscr{B}_j$ is sorted by induction, for $j \geq 1$. After step P3, the upper bound in equation (2) still holds for any vector v in block $\mathscr{B}_{j+1}$ (modulo the inner permutation due to the sorting of $\mathscr{B}_{j+1}$). Indeed, the number of vectors $z > v$ that are laid out before v cannot increase; those inside $\mathscr{B}_{j+1}$ disappear after sorting it and so the upper bound in equation (2) is anyway valid. By equation (1), we derive that p, the size of each block, is larger than the upper bound of equation (2). As a result, the number of vectors $z > v$

that belong to the jth prefix of blocks cannot exceed p. Hence, they should be contained in the last locations of block $\mathscr{B}_j$ since $p = |\mathscr{B}_j|$ and $\mathscr{B}_1\mathscr{B}_2 \cdots \mathscr{B}_j$ is sorted by induction. This allows us to conclude that after merging $\mathscr{B}_j$ and $\mathscr{B}_{j+1}$, the $(j+1)$st prefix of blocks $\mathscr{B}_1\mathscr{B}_2 \cdots \mathscr{B}_{j+1}$ is sorted, thus proving the statement of the theorem.

4.2 Session sorting

We apply the steps stated in Theorem 1 to sorting the vectors in $\mathscr{M}$ into sessions. We choose $\mathscr{M}_1$ of size $O(n/\log n)$ for the placeholders. We then fix $q = \log^3 n$, $p = qn/\log^4 n = \Theta(n/\log n)$, and we pick m as the largest multiple of p such that $m \leq |\mathscr{M}| - |\mathscr{M}_1|$. These values satisfy equation (1). We therefore obtain the logical division of $\mathscr{M}$ into blocks $\mathscr{M}_1$, …, $\mathscr{M}_{s-1}$, $\mathscr{M}_s$, as expected. We comment on how to apply steps P1–P4 to $\mathscr{M}_2$, …, $\mathscr{M}_s$ (assuming w.l.o.g. that $|\mathscr{M}_s| = |\mathscr{M}_{s-1}|$).

In steps P1 and P3, we have to solve a number of $m/p = O(\log n)$ instances of $\mathit{GVSP}\{O(n/\log n), O(n/\log n), O(n/\log^2 n)\}$ (see Section 5).

In step P2, we have just $m/q = O(n/\log^3 n)$ vectors to sort, which are the minimum in each sub-block. We refer to them as *headers* and to the rest of the vectors as *satellite data* (with $q - 1$ vectors each). We associate a unique implicit index in the range from 1 to m/q with the satellite data in each sub-block. We employ the heavy bits in H so as to form a sequence of m/q integers $h_1, h_2, \ldots, h_{m/q}$ of $\log n$ bits each, employed to encode a permutation of these indexes. Note that we have direct access to any h_j, $1 \leq j \leq m/q$, in $O(k \log n)$ time for encoding it and $O(k + \log n)$ time for decoding it by Lemma 2.

At the beginning of step P2, we set $h_j = j$ and exchange the jth header with the jth placeholder in $\mathscr{M}_1$, for $1 \leq j \leq m/q$. We then apply the in-place *stable* mergesort on the headers thus collected in $\mathscr{M}_1$. Each comparison cost is $O(k)$ time while each exchange requires $O(k \log n)$ time. Indeed, when exchanging two headers inside $\mathscr{M}_1$, say at position j' and j'', we have also to swap the values of $h_{j'}$ and $h_{j''}$, involving their decoding and encoding in H. Note that the satellite data is not exchanged but $h_{j'}$ and $h_{j''}$ are correctly updated to maintain the association of the headers with their satellite data in the sub-blocks. At the end of the mergesort, we exchange the jth header in $\mathscr{M}_1$ with the placeholder temporarily hosted in the h_jth sub-block. The total cost is $O((m/q) \log(m/q) \times (k \log n)) = O((n/\log^3 n) \log n \times (k \log n)) = o(nk)$.

We now have to permute the sub-blocks according to the values of $h_1, \ldots, h_{m/q}$ encoded in H. Specifically, the h_jth sub-block must occupy the jth position among the sub-blocks to reflect the stable sorting of their headers. We employ an additional sequence of integers $r_1, r_2, \ldots, r_{m/q}$ encoded in H, initializing $r_i = j$ if and only if $h_j = i$. We proceed incrementally for $j = 1, 2, \ldots, m/q - 1$ (in this order), preserving the invariant that we have correctly placed the first $j - 1$ sub-blocks, with $h_1, \ldots, h_{m/q}$ and $r_1, r_2, \ldots, r_{m/q}$ suitably updated to reflect the fact that one permutation is the inverse of the other (in particular, $h_{j'} = r_{j'} = j'$ for $1 \leq j' < j$, so the invariant is meaningful for the rest of the indexes). Note that some of the sub-blocks may have exchanged in order to place the first $j-1$ sub-blocks. Hence, when we refer to the jth and the h_jth sub-blocks, they are taken from the current arrangement of sub-blocks. If $j = h_j$, the current sub-block is already correctly placed and the invariant is trivially preserved. Otherwise, we exchange the jth and the h_jth sub-blocks by pairwise exchanging their ith vectors for $i = 1, 2, \ldots, q$. In order to preserve the invariant, we simultaneously swap the values of h_j and h_{r_j} and the values of r_j and r_{h_j}, respectively, re-encoding them in H. Since the exchange of sub-blocks requires the pairwise exchange of q vectors plus the encoding and decoding of $O(1)$ values among $h_1, \ldots, h_{m/q}$ and $r_1, r_2, \ldots, r_{m/q}$, the cost is $O(qk + k \log n)$. When $j = m/q - 1$, the last two sub-blocks are placed correctly and we have performed a total of $O(m/q)$ such exchanges. The final cost is $O(m/q \times (qk + k \log n)) = O(n/\log^3 n \times k \log^3 n) = O(nk)$. Hence, the total cost of step P2 is $O(nk)$.

Finally, in step P4, we use the in-place merging with comparison cost $O(k)$. As a result, we obtain a total cost of $O(m/p \times pk) = O(nk)$ for step P4 (and $\mathscr{M}$ is sorted).

Lemma 5. *There is an $O(nk)$-time reduction from $GVSP\{n-o(n), 0, O(n/\log^2 n)\}$ to a number of $O(\log n)$ instances of $GVSP\{O(n/\log n), O(n/\log n), O(n/\log^2 n)\}$, using $O(1)$ auxiliary locations.*

5 Sorting Each Block Individually

We have to solve an instance of $GVSP\{O(n/\log n), O(n/\log n), O(n/\log^2 n)\}$ (see Figure 1). We reformulate it as $GVSP\{m', O(m'), O(m'/\log m')\}$, where $m' = O(n/\log n)$ vectors in $\mathscr{M}'$ should be sorted using a sufficiently large number $O(m')$ of placeholders in $\mathscr{M}_B$. We need to encode $O(1)$ sequences of integers of $O(\log m') = O(\log n)$ heavy bits each in $H \subseteq \mathscr{L} \times \mathscr{R}$, totalizing $O(m'/\log m')$ heavy bits. We sort $\mathscr{M}'$ by repeatedly inserting its vectors in an internal structure maintained inside $\mathscr{M}_B$ to mimic a distribution sort into buckets of $O(\log^2 m')$ vectors each. Each bucket is sorted by applying a constant number of recursive calls to session sorting (Section 4.2). The base case is an instance of $GVSP\{O(\sqrt{\log m'}), O(\sqrt{\log m'}), 0\}$. We first rank the vectors by linking them in a sorted list without moving the vectors (we mimic the insertion sort in a list without moving vectors). The list pointers of $O(\log\log m')$ bits each, however, are not encoded with heavy bits in this case. Since we sort one bucket at a time and have $O(\sqrt{\log m'})$ such pointers, we can keep the $O(\sqrt{\log m'} \log\log m') = o(\log n)$ bits for all the pointers in one auxiliary location. We can access any such pointer in constant time, and we can append a new pointer to them within the same complexity by using RAM operations. We apply Hirschberg's linear scanning to add a new vector to the sorted list and mimic insertion sort. Hence, the cost per vector is $O(k + \sqrt{\log m'})$. After setting up the linked list that reflects the sorted order, we permute the vectors using the temporary buffer of $O(\sqrt{\log m'})$ placeholders. Thus the time complexity of $GVSP\{O(\sqrt{\log m'}), O(\sqrt{\log m'}), 0\}$ is bounded by $O(k\sqrt{\log n} + \log n)$. We summarize the resulting bounds, leaving several technical details to the full paper.

Lemma 6. *An instance of GVSP*$\{O(n/\log n), O(n/\log n), O(n/\log^2 n)\}$ *takes* $O(n + nk/\log n)$ *time using* $O(1)$ *auxiliary locations.*

Theorem 2. *An arbitrary set of* n *vectors of length* k *can be sorted in place optimally, taking* $O(nk + n\log n)$ *time and using* $O(1)$ *auxiliary locations.*

References

1. A. Andersson, T. Hagerup, J. Håstad, and O. Petersson. Tight bounds for searching a sorted array of strings. *SIAM Journal on Computing*, 30(5):1552–1578, 2001.
2. L. Arge, P. Ferragina, R. Grossi, and J.S. Vitter. On sorting strings in external memory. ACM STOC '97, 540–548, 1997.
3. M.A. Bender, E.D. Demaine, and M. Farach-Colton. Cache-oblivious B-trees. IEEE FOCS '00, 399–409, 2000.
4. J.L. Bentley and R. Sedgewick. Fast algorithms for sorting and searching strings. ACM-SIAM SODA '97, 360–369, 1997.
5. G.S. Brodal, R. Fagerberg, and R. Jacob. Cache oblivious search trees via binary trees of small height. ACM-SIAM SODA '02, 39–48. 2002.
6. T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms.* MIT Press, 2001.
7. G. Franceschini. Proximity mergesort: Optimal in-place sorting in the cache-oblivious model. ACM-SIAM SODA '04, 284–292, 2004.
8. G. Franceschini. Sorting stably, in-place, with $O(n \log n)$ comparisons and $O(n)$ moves. STACS '05, to appear, 2005.
9. G. Franceschini and V. Geffert. An In-Place Sorting with $O(n \log n)$ Comparisons and $O(n)$ Moves. IEEE FOCS '03, 242–250, 2003.
10. G. Franceschini and R. Grossi. No Sorting? better Searching! IEEE FOCS '04, 491–498, 2004.
11. D.S. Hirschberg. A lower worst-case complexity for searching a dictionary. Proc. 16th Allerton Conference on Comm., Control, and Computing, 50–53, 1978.
12. D.S. Hirschberg. On the complexity of searching a set of vectors. *SIAM J. Computing*, 9(1):126–129, 1980.
13. A. Itai, A.G. Konheim, and M. Rodeh. A sparse table implementation of priority queues. ICALP '81, 417–431, 1981.
14. D.E. Knuth. *The Art of Computer Programming III: Sorting and Searching.* Addison–Wesley, 1998.
15. S.R. Kosaraju. On a multidimensional search problem. ACM STOC '79, 67–73, 1979.
16. M.A. Kronrod. Optimal ordering algorithm without operational field. *Soviet Math. Dokl.*, 10:744–746, 1969.
17. T.W. Lai and D. Wood. Implicit selection. SWAT '88, 14–23, 1988.
18. U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.
19. J.I. 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.
20. J.I. Munro and V. Raman. Sorting multisets and vectors in-place. WADS '91, 473–480, 1991.
21. J. Salowe and W. Steiger. Simplified stable merging tasks. *Journal of Algorithms*, 8(4):557–571, 1987.
22. D.E. Willard. Maintaining dense sequential files in a dynamic environment. ACM STOC '82, 114–121, 1982.