No Sorting? Better Searching! *

Gianni Franceschini†
Dipartimento di Informatica
Università di Pisa
francesc@di.unipi.it

Roberto Grossi‡
Dipartimento di Informatica
Università di Pisa
grossi@di.unipi.it

March 7, 2005

Abstract

Questions about order versus disorder in systems and models have been fascinating scientists over the years. In Computer Science, order is intimately related to sorting, commonly meant as the task of arranging keys in increasing or decreasing order with respect to an underlying total order relation. The sorted organization is amenable for searching a set of n keys, since each search requires $\Theta(\log n)$ comparisons in the worst case, which is optimal if the cost of a single comparison can be considered a constant. Nevertheless, we prove that disorder implicitly provides more information than order does. For the general case of searching an array of multi-dimensional keys, whose comparison cost is proportional to their length (and hence cannot be considered a constant), we demonstrate that "suitable" disorder gives better bounds than those derivable by using the natural lexicographic order.

We start out from previous work done by Andersson, Hagerup, Håstad and Petersson [*SIAM Journal on Computing*, 30(2), 2001], who proved that

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

character comparisons (or probes) are the tight complexity for searching a plain sorted array of n keys, each of length k, arranged in lexicographic order. We describe a novel *permutation* of the n keys that is different from the sorted order. When keys are kept "unsorted" in the array according to this permutation, the complexity of searching drops to

$$\Theta\left(k + \log n\right)$$

character comparisons (or probes) in the worst case, which is *optimal* among all possible permutations, up to a constant factor. Consequently, disorder carries more information than order does; this fact was not observable before, since the latter two bounds are $\Theta(\log n)$ when $k = O(1)$. More implications are commented in the paper, including searching in the bit probe model.

*The results in this paper have been presented at the 45th Annual IEEE Symposium on Foundations of Computer Science (FOCS), 2004.

†Work done while visiting EECS Department, Berkeley U., USA. Supported in part by the Italian Ministry of Research and Education.

‡Supported in part by the Italian Ministry of Research and Education.

1 Introduction

"Imagine how hard it would be to use a dictionary
if its words were not alphabetized!"
—D.E. Knuth, The Art of Computer Programming III (1998)

As noted at the beginning of Knuth's book [14], sorting n keys with respect to an underlying total order provides a basic data organization for optimal searching with $\Theta(\log n)$ time and comparisons in the worst case. This fact is corroborated by our common sense and everyday practice; sorting and searching are strictly related companions in designing and analyzing many algorithms for the comparison model. In this paper we demonstrate that "suitable" disorder provides better bounds for searching than those derivable by using the sorted order. In particular, we prove that sorting is *not* the best general way to organize multi-dimensional keys (e.g. vectors) in an array for *optimal* searching.

We are given n keys, where each key is a sequence of k symbols drawn from an arbitrary, possibly unbounded, totally ordered alphabet. The keys can be maintained in any permutation, and the ith key in the permutation can be selected in constant time. Given any such key, the jth symbol in it can be accessed in constant time. Conceptually, it is more useful to think of the permutation as a $k \times n$ matrix $\mathscr{A}$ in which the ith column contains the ith key, say x; hence, x itself can be considered as an array $x \equiv x[1 \ldots k]$ of k entries. The keys underlie the lexicographic (alphabetic) order, namely, for any two keys x, y in $\mathscr{A}$, we have $x \leq y$ if and only if either $x = y$ or there exists $j < k$ such that $x[1 \ldots j] = y[1 \ldots j]$ and $x[j+1] < y[j+1]$. Since this model extends the comparison model to keys of arbitrary length, we can only compare the individual symbols of the keys (without hashing or bit manipulation). We measure the time complexity by accounting interchangeably for the number of character comparisons or probes, as they are linearly proportional to each other in our case. Hence, comparing any two keys x and y requires $O(k)$ time.

We study the fundamental problem of establishing whether a given key of length k appears in $\mathscr{A}$ as one of its stored keys. Setting $k = 1$ gives the classical searching problem. After sorting $\mathscr{A}$, we can run the binary search on $\mathscr{A}$ with optimal cost of $\Theta(\log n)$. In general, when $k \geq 1$, previous work focused on keys sorted under the lexicographic order.

The problem was introduced by Hirschberg [12], with upper bounds of $O(k + n)$ and $O(k \log n)$ in the worst case. The former is obtained by scanning $\mathscr{A}$ while the latter is a simple binary search on $\mathscr{A}$. The worst-case lower bound of $\Omega(k + \log n)$ follows quite easily. The logarithmic term in n comes from the decision tree for searching in a set of n keys, while the linear term in k comes from the need of reading all the k symbols in the search key.

The first nontrivial upper bound was $O(k \log n / \log k)$ by Hirschberg [13]. Kosaraju [15] later improved it to $O(k\sqrt{\log n} + \log n)$. With sophisticated techniques for proving upper and lower bounds on the complexity of *searching the sorted array* without exploiting any preprocessed auxiliary information, Andersson, Hagerup, Håstad and Petersson closed the gap by proving in [1, 3, 2] that the cost is

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

character comparisons in the worst case. Note that the bound is $\Theta(\log n)$ when $k = 1$, which is an established fact in algorithmics. However, when $k \geq \log n$, the cost is larger than the previous lower bound of $\Omega(k + \log n)$.

Using the same model as that adopted in previous work, we describe a novel *permutation* of the n keys in $\mathscr{A}$ that is different from the sorted order but that can be obtained efficiently from it.

When keys are stored according to this "unsorted" order in the array, the worst-case complexity of searching in the array drops to

$$\Theta\left(k + \log n\right)$$

character comparisons, which is asymptotically *optimal* among all possible permutations of the n keys in the array included the lexicographically sorted one.

As a result, we provide a (hopefully!) unexpected insight on the relation between sorting and searching, which can be restated in terms of the basic question of the informative power of order versus disorder. We show that sorted arrays are *not* the best data organization suitable for searching k-dimensional keys. In this sense, sorting is an optimal placement of keys only for searching an array when $k = O(1)$.

Our algorithms are also suitable for more powerful queries in optimal time, such as computing the rank of a search key among those stored in $\mathscr{A}$, still with a cost of $\Theta(k + \log n)$. We can identify its predecessor or successor within the same bounds. We can also list the keys in $\mathscr{A}$ belonging to a given input interval $[a, b]$ for any two keys $a \leq b$. We attain an output-sensitive cost of $\Theta(k + \log n + \#retrieved)$, where $\#retrieved$ denotes the number of keys in $\mathscr{A}$ that belong to the interval $[a, b]$. Since the latter bounds cannot be achieved with the sorted array alone, this strengthens the fact that our permutation is more powerful than the sorted one. Furthermore, our permutation is efficiently computable and reversible: we can obtain our permutation from the sorted one and vice versa in $O(nk)$ time.

We point out that our optimal bounds can be achieved with other data structures keeping the keys sorted and exploiting additional information in *extra* space (e.g. the longest common prefix information in suffix arrays by Manber and Myers [16], applied to $\mathscr{A}$ in sorted order, or the extra fields in ternary search trees by Bentley and Sedgewick [4]). The current and previous work touched in this paper does not make use of this extra space. The result of Andersson et al. deals with an array alone, without the possibility of exploiting any additional preprocessing information (e.g. no pointers or integers) apart from $O(1)$ values, namely, the address of the array and its size $n \times k$. In this sense, we obtain the first *optimal* implicit organization for k-dimensional keys, since sorted arrays are optimal just for $k = O(1)$. To appreciate the connection, let us recall that Munro and Suwanda [18] define an implicit data structure as a permutation of the keys plus just $O(1)$ auxiliary cells each of $O(\log n)$ bits. Closely related is the problem of arranging n records with k fields into a $k \times n$ array so that searches can be performed quickly for any given field value. Searches under this model can be supported in $O(\log n)$ time by Fiat et al. [7], where the "O" includes a factor of $k \log k$. This method cannot be extended to solve our problem optimally.

We remark that our bounds hold also for the dictionary problem in the bit probe model [5, 19]. In this model, the keys are binary strings and the complexity accounts for the number of bits probed in the binary array $\mathscr{A}$ storing them. We can store these keys permuted in nk bits (i.e. a $k \times n$ binary matrix), so that membership requires $\Theta(k)$ bit probes in the worst case, which is optimal. Note that our bound gives an alternative perspective to Yao's result [19] on achieving an optimal search for keys that can be permuted in an array without using extra space (i.e. without storing the name of a suitable hashing function). It also relates to the issues on extra space studied for perfect hashing by Fredman, Komlós and Szemerédi [11], non-oblivious hashing by Fiat, Naor, Schmidt and Siegel [9, 10], and implicit probe search by Fiat and Naor [6, 8].

The paper is organized as follows. In Section 2, we show how to encode bits with keys of non-constant length and, hence, how to implicitly represent the extra information for optimal searching. In Section 3, we describe how to exploit the encoding by devising a search algorithm that probes fews bits during its execution. We draw our conclusions in Section 4.

2 Permuting for Encoding Bits of Information

In this section we provide a novel tool for encoding and decoding bits with keys of non-constant length k. What is easily done in the implicit data organizations presented in the previous literature [17], namely, encoding a bit by swapping or not two consecutive keys, is non-trivial in our case. (We recall that the technique in [17] assumes by default that, between two certain consecutive keys, the smallest comes first; so, when the largest is first met, a 1 is encoded, otherwise a 0 is.) First, the keys are not necessarily distinct under our model. Second, decoding just one bit requires an $O(k)$-time comparison of two keys when done naively. Starting out from the sorted sequence of a given sequence of r keys, in non-decreasing order, we show how to encode bits according to a new scheme, which we call *ditch*. We employ two incarnations of this scheme:

- *small* ditches, obtained by permuting $r = O(\log n)$ keys;
- *large* ditches, obtained by permuting $r = O(n)$ keys.

We recall that the n input keys are maintained in $\mathscr{A}$, which is conceptually a $k \times n$ matrix to be permuted column-wise, and initially sorted in non-decreasing order. Here, the columns are the n keys, where a_j is the key in the jth column of $\mathscr{A}$. The ith symbol $a_j[i]$ in that key is in entry $\mathscr{A}[i, j]$. Hence, the ith row in the matrix contains the ith symbols of all the keys. For any two keys x and y, we introduce the notation $lcp(x, y) = \max(\{0\} \cup \{1 \leq \ell \leq k : x[1 \ldots \ell] = y[1 \ldots \ell]\})$ to denote the length of their longest common prefix, widely adopted in string searching [16]. Indeed, if we can infer $\ell = lcp(x, y)$, we can compare x and y in constant time (either $x = y$ or it suffices to compare the first mismatching symbols, $x[\ell + 1]$ and $y[\ell + 1]$).

We first describe the ditch in Section 2.1, along with some properties common to small and large ditches. We then show how to encode the information on the inner structure of a large ditch in Section 2.2. We describe in Section 2.3 how to preprocess $\mathscr{A}$ by laying out a large ditch in it.

2.1 The ditch: A basic tool for encoding and decoding bits

We start out from a sequence of r keys, $x_1, \ldots, x_r$, initially in sorted order. We say that i and j are *twin positions* (for $1 \leq i < j \leq r$) if and only if they are specular, namely, $i + j = r + 1$. In other words, the number of keys up to position i equals that of keys from position j to the end of the sequence. Let us "dig" into the sequence by comparing incrementally the keys at consecutive twin positions, starting from the two extreme positions of the sequence. We trace this process with an integer d, called *digging depth*, according to the procedure below:

```
DIGGING(x_1, ..., x_r):
1: d ← 1, i ← 1, j ← r
2: WHILE i < j DO
3:     WHILE d ≤ k AND x_i[d] = x_j[d] DO
4:         d ← d + 1
5:     i ← i + 1, j ← j − 1
```

The pseudocode determines a *ditch*, illustrated by the example in Figure 1, which we characterize formally by its useful properties. The relevant one is that of encoding bits so that they can be quickly decoded. We first note an important property for DIGGING.

Lemma 1 *The inner loop of procedure* DIGGING *computes* $d = lcp(x_i, x_j) + 1$ *for any twin positions* i *and* j. *The total cost of* DIGGING *for a sorted sequence of* r *keys is* $O(k + r)$ *time, where the keys are of length* k.

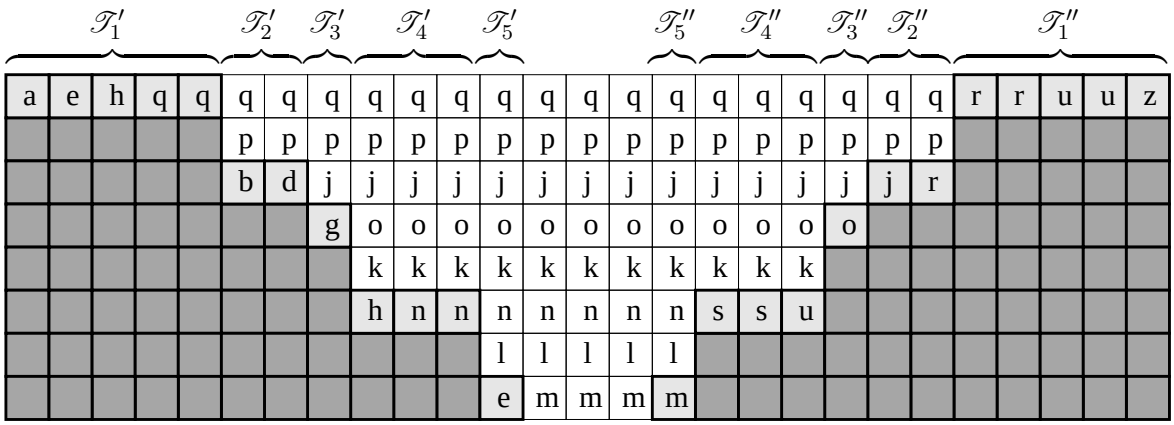

Figure 1: A ditch whose keys in twin positions are not yet swapped. The mismatching symbols, $x_i[d] \neq x_j[d]$, yielding the digging depth d for any twin positions i and j are shown in light gray.

Proof: Since $lcp(x_{i'}, x_{j'}) \leq lcp(x_i, x_j)$ for any $i' < i < j < j'$, the digging depth for twin positions i and j is surely at least d, before the execution of the inner loop in DIGGING. After that loop, we consequently update d so that $d = lcp(x_i, x_j) + 1$. The monotonicity of the digging depths accounts for the total cost of digging. Hence, the incremental cost of comparing keys in twin positions $i = i' + 1$ and $j = j' - 1$, after having done it with i' and j', is proportional to $lcp(x_i, x_j) - lcp(x_{i'}, x_{j'}) + O(1)$. The total cost is a telescopic sum that evaluates to the maximum *lcp* value, $O(k)$, plus a cost proportional to the number r of keys in the sequence. This gives the final bound on the cost of digging. □

With reference to Figure 1, we can interpret Lemma 1 as saying that the digging cost is proportional to the perimeter, $O(k + r)$, of the ditch rather than its area, $O(k \times r)$. Since the keys involved in digging are initially in sorted order, we can easily verify that the following holds.

Fact 1 *For any pair of twin positions i and j with $i < j$, if the digging depth satisfies $d \leq k$, then $x_i[d] < x_j[d]$.*

We exploit Fact 1 and the knowledge of d for encoding and decoding bits. We adopt a simple, but effective, rule for encoding a bit with a pair of twin positions i and j: swap keys x_i and x_j to encode 1, so that x_j is in position i and x_i is in position j; otherwise, leave x_i and x_j at their own positions i and j, respectively, thus encoding 0.

Decoding is simple and takes constant time for any given pair of twin positions i and j, provided that their digging depth d is given. Namely, let z_i and z_j be the keys in these positions (note that either $z_i = x_i$ and $z_j = x_j$, or $z_i = x_j$ and $z_j = x_i$). We decode the ith bit by simply comparing $z_i[d]$ and $z_j[d]$: that bit is 1 if and only if $z_i[d] > z_j[d]$, as a consequence of Fact 1.

We can extend the property above to b contiguous (and nested) pairs of twin positions. We say that two pairs i, j and i', j' are *contiguous* (where $i' < i < j < j'$) if $i = i' + 1$ and $j = j' - 1$. Suppose that we do not know their digging depths. Hence, we cannot apply directly the constant-time decoding of individual bits described above. Fortunately, we can exploit the computational pattern of procedure DIGGING to circumvent this drawback as shown below.

Lemma 2 *We can decode the b bits encoded by b consecutive pairs of twin positions in $O(b + k)$ time.*

Proof: Decoding all the b bits is like digging. What changes in decoding is that the keys are pairwise permuted in twin positions. Interestingly the inner (i.e. matching) characters of the ditch do not change. So the matched characters in the inner while loop at lines 3–4 of procedure DIGGING are likewise matched in decoding. As a result, decoding computes on the fly all the digging depths d of the first b twin positions. What changes are the mismatching characters. A mismatch causing a loop exit in DIGGING might find that the keys in the current twin positions are swapped for encoding purposes. Since we know at this point d, we can decode the current bit by looking at the mismatch according to Fact 1. The analysis for the time complexity of decoding goes along the same lines as that of Lemma 1. □

We remark that our use of ditches will be twofold. For a *small* ditch, with $r = O(\log n)$, we will scan all the r keys for decoding $b = r = O(\log n)$ bits in $O(k + \log n)$ time by Lemma 2. For a *large* ditch, with $r = O(n)$, we need to record and encode the positional information of the ditch's components to avoid a full scan of the r keys, which is not that efficient. Hence, we focus on how to encode the inner structure of large ditches.

2.2 Encoding large ditches

We recall that a large ditch contains $r = O(n)$ keys that are initially sorted, and need to be permuted suitably for encoding purposes. However, we should encode information also for the encoder itself, the large ditch. Hence, we need a two-level approach. First, we use few keys to encode the information representing the large ditch, and define its structure deriving from the digging process. We discuss this topic in this section. Second, we use the rest of the keys in the large ditch to encode an implicit search data structure, which is detailed in Section 3.

Here we focus on encoding the large ditch. Pictorially, the border of the ditch is the concatenation of two specular stairs, one descending and one ascending, shown in ligth gray in Figure 1. The characters implicitly matched inside a ditch during the digging, are equal to the homologous characters of the median key. Let us fix any row d, such that d is one of the digging depths computed by procedure DIGGING. In that row, there can be further characters (in the dark gray part of Figure 1, outside the border of the ditch) that are equal to the median's: however, they can only extend beyond one side of the ditch rather than both sides, since otherwise the ditch would be deeper.

As noted in the proof of Lemma 1, the digging depth d is monotonically non-decreasing. Thus we can split the positions in the sequence into *twin intervals*, as shown in Figure 1. A pair of twin intervals, $\mathscr{T}', \mathscr{T}''$, for a given digging depth d collects all the twin positions with the same digging depth d. These intervals are well defined since their twin positions are contiguous, with $|\mathscr{T}'| = |\mathscr{T}''|$. For reference purposes, we can number the twin intervals from left to right. Note that we cannot have more than $2k$ twin intervals.

The twin intervals represent the inner structure of a large ditch. Consequently, we represent a ditch by the positional information of each pair of twin intervals in it, say $\mathscr{T}', \mathscr{T}''$:

- The leftmost and rightmost position in $\mathscr{T}'$, in $O(\log n)$ bits. We can infer that of $\mathscr{T}''$ from $\mathscr{T}'$ as they are specular with respect to the median position in the ditch.
- Their digging depth d, in $O(\log k)$ bits. However, we do not need to store d explicitly. It can be recovered in $O(k)$ time by computing $d = lcp(x_{i'}, x_{i''}) + 1$ on the fly, for any two keys with $i' \in \mathscr{T}'$ and $i'' \in \mathscr{T}''$.

As a result, we have to encode just $O(\log n)$ bits per pair of twin intervals. Since there are $O(k)$

of them in a ditch, we need to encode a total of $O(k \log n)$ bits to represent a ditch implicitly. We will single out $O(k \log n)$ keys from the sequence to this end.

The importance of using twin intervals is made clear by providing a glimpse on our searching scheme. In addition to the $O(k \log n)$ keys, previously mentioned for encoding a large ditch, let us single out the leftmost key in each twin interval, from left to right. We can search in these keys in $O(k + \log n)$ time and identify a unique twin interval, say $\mathscr{T}$, in which we will continue the search. In other words, the ditch is able to route the search optimally towards a twin interval $\mathscr{T}$. What makes a difference with respect to the original search problem is that searching inside $\mathscr{T}$ is an easier task as we can rely on the implicitly encoded bits in it. The pair of twin intervals, $\mathscr{T}$, $\hat{\mathscr{T}}$, encodes b bits, where $b = |\mathscr{T}| = |\hat{\mathscr{T}}|$. We go through the convention of assigning $b/2$ bits to each of them; say, the bits encoded by odd positions to $\mathscr{T}$ and those encoded by even positions to $\hat{\mathscr{T}}$ (assuming that $\mathscr{T}$ is to the left of $\hat{\mathscr{T}}$). When the search is routed to a twin interval $\mathscr{T}$, the digging depth d of $\mathscr{T}$ can be easily retrieved. Using d, any encoded bit associated with $\mathscr{T}$ can be subsequently decoded with a character comparison in $O(1)$ time.

We discuss how to preprocess and search each $\mathscr{T}$ to this end in Section 3. Here we describe the rest of the preprocessing of the input array $\mathscr{A}$.

2.3 Preprocessing array $\mathscr{A}$

We now have a better picture of how to permute the keys by preprocessing the input array $\mathscr{A}$. We recall that it is a two-dimensional $k \times n$ matrix of keys, initially in lexicographic order. We divide $\mathscr{A}$ into four zones with the (exotic) name of zones A, B, C and D.

Zones A and B contain overall $O(k \log n)$ distinct keys from $\mathscr{A}$ while the rest of the keys form the large ditch in zones C and D. If we have less than $O(k \log n)$ distinct keys in $\mathscr{A}$, we are in a special case that we can easily handle using [2], for example. Hence, let us assume that we have a sufficient number of distinct keys. We move them to zones A and B, maintaining the initial sorted order, and dividing them into k blocks each of $\Theta(\log n)$ keys. We move the leftmost key of each block, in left-to-right order, to zone A. We obtain a two-level structure, zone A and zone B, in which the $O(k)$ keys of zone A form a small search directory for identifying a block in zone B, which hosts the rest of these keys. In this way, we can support searching in these two zones. (Alternatively, we can employ the techniques from [2] to combine zones A and B.)

However, the main purpose of the keys in zone B is for encoding the large ditch in zones C and D. Specifically, the ith block in zone B implicitly encodes the $O(\log n)$ bits representing the positional information for the ith pair of twin intervals in the large ditch (see Section 2.2). These bits are encoded using a small ditch built within the ith block itself. We use Lemma 2 to this end, with $b = O(\log n)$. Hence, decoding the information for the ith pair of twin intervals requires $O(k + \log n)$ time by accessing the ith block in zone B. We repeat this task for $i = 1, 2, \ldots, k$ to complete the preprocessing of zones A and B.

Zones C and D contain the rest of the keys and properly form the large ditch. Analogously to zone A, we move the leftmost key from each twin interval to zone C (note that the keys in this zone are still in lexicographic order). Searching zone C is like searching zone A. The remaining keys are in zone D (and they are still sorted at this stage but will be permuted later on). We refer the reader to Section 3 for the rest of the preprocessing, which deals with searching inside a twin interval.

3 Searching with a Few Bits of Information

We now can give a description of our search, which uses Hirschberg's search [12] as a basic routine taking $O(k+r)$ time for r keys. However, we need to adapt this search to a ditch, since the keys in twin position can be pairwise permuted to encode bits. It is not difficult to combine our on-the-fly decoding based on digging (Section 2.1) and Hirschberg's search, attaining $O(k+r)$ total search time for a ditch with r keys. This is an immediate corollary to Lemma 2. We will refer to Hirschberg's search thus modified as *scanning search*.

3.1 Restricting the range for searching

Given a search key x, we first check whether x occurs in zones A or B. For this, we apply the scanning search to the $r = O(k)$ keys that form the directory in zone A, in $O(k)$ time. We then identify a block in zone B with $r = \Theta(\log n)$ keys. We also apply here the scanning search in $O(k + \log n)$ time.

If we do not find the key in zones A and B, we search it in the large ditch represented by zones C and D. First we check if the search key equals the median key in the ditch in $O(k)$ time. If not, we proceed to search in the directory of zone C. Like zone A, it takes $O(k)$ time.

If the key is not in zone C, we end up in one position, say, of the ith key in zone C. By our preprocessing of Section 2.3, we infer that we have to access the ith twin interval, say $\mathscr{T}$, containing x in zone D (if any; if x occurs more than once, we take its leftmost occurrence). At this point, we need the positional information of $\mathscr{T}$ (see Sections 2.2– 2.3). Assume without loss of generality that $\mathscr{T}$ belongs to the ith pair of twin intervals. We decode $\mathscr{T}$'s information by scanning $b = O(\log n)$ keys in the ith block of zone B, in $O(k + \log n)$ time by Lemma 2. We also compute, on the fly, the digging depth of $\mathscr{T}$ in $O(k)$ time. We recall that this depth, say d, can be computed as $d = lcp(y, z) + 1$ for any chosen pair of keys y in $\mathscr{T}$ and z in $\mathscr{T}$'s twin interval.

As a result we have reduced, in $O(k + \log n)$ time, the problem of searching a key x in the array $\mathscr{A}$ into the problem of searching x in a suitable twin interval, $\mathscr{T}$, of zone D in $\mathscr{A}$. We therefore describe how to preprocess a twin interval $\mathscr{T}$ to this end, exploiting the fact that we can encode implicitly up to $|\mathscr{T}|/2$ bits in it.

Lemma 3 *The keys in the input array $\mathscr{A}$ can be permuted to form a ditch so that searching in $\mathscr{A}$ reduces, in $O(k + \log n)$ time, to searching in one of its twin intervals.*

3.2 Searching within a twin interval

We describe how to preprocess a twin interval $\mathscr{T}$ for searching purposes, assuming that we can encode up to $|\mathscr{T}|/2$ bits by permuting keys. If $|\mathscr{T}| = O(\log n)$, we run the scanning search in $O(k + \log n)$ and we are done. We assume hereafter that $|\mathscr{T}| = \Omega(\log n)$.

The ideal approach is that of using the search adopted for Manber and Myers' suffix array [16], which we refer to as MM-SEARCH in the rest of the paper. Originally designed for an array of suffixes, it finely works also for arbitrary keys of unbounded length in lexicographic order as is our case. However, MM-SEARCH and its variations devised thereafter access $\Omega(\log n \log k)$ extra bits and $O(k)$ characters of the keys. The problem comes with the extra bits, since decoding them would require $\Omega(\log n \log k)$ time (we need to perform at least a comparison to establish the value of an encoded bit), which gives a sub-optimal bound for our problem. We propose a variant that is tailored for the encoded bits in twin intervals, in that it is less demanding for the number of extra bits accessed during the process.

We first review in a nutshell how MM-SEARCH works for a sequence of r keys each of length s (we use s in place of k since we will apply MM-SEARCH to keys of length different from k). It performs a variation of the classical binary search according to three cases. For the current search interval, $[L \ldots R]$, it maintains the invariant that $x_L \leq x \leq x_R$, and the *lcp* values between the search key x and the keys x_L, x_R at the extreme positions L and R, respectively. Let $M = (L + R)/2$ be the middle point of the interval. MM-SEARCH infers the outcome of the comparison of x versus x_M (the key in position M) while preserving the invariant on searching into a smaller interval, either $[L \ldots M]$ (if $x \leq x_M$) or $[M \ldots R]$ (if $x_M < x$). Namely, let us suppose that $lcp(x_L, x) \geq lcp(x, x_R)$ without loss of generality, so that $m = lcp(x_L, x)$ is the number of initial symbols in x that have been successfully matched so far. MM-SEARCH makes use of a precomputed value, $lcp(x_L, x_M)$, which is independent of the search key x:

1. Case $m < lcp(x_L, x_M)$. Set $L = M$.
2. Case $m = lcp(x_L, x_M)$. Compute $lcp(x, x_M)$ by comparing x and x_M from position $m+1$ on. Set $m = lcp(x, x_M)$ to the value thus computed. Access symbols in positions $m + 1$ (if any) of x and x_M. If $x[m + 1] > x_M[m + 1]$, set $L = M$; else, set $R = M$.
3. Case $m > lcp(x_L, x_M)$. Set $R = M$.

Another precomputed value, $lcp(x_M, x_R)$, is employed for the symmetric case, $lcp(x_L, x) < lcp(x, x_R)$. Since we are not allowed to use more than $O(1)$ auxiliary cells of memory, these precomputed values must be encoded somewhere in the twin interval. Hence, MM-SEARCH needs to read these $O(\log s)$ encoded bits anyway, and there are $O(\log r)$ steps in the binary search. Hence, it cannot read less than $O(\log r \log s)$ decoded bits, which gives a sub-optimal search in our case. We can draw an analogous conclusion when MM-SEARCH maintains its invariant. It requires to maintain the values of $lcp(x_L, x)$ and $lcp(x, x_R)$ after dealing with each of the three cases (and their symmetric cases). The first two cases are not a problem. After the first case, $lcp(x_L, x) = lcp(x, x_M)$, which is also the current value of m. So, there are no extra bits to decode. After the second case, $lcp(x_L, x)$ or $lcp(x, x_R)$ has been just computed as well since it becomes equal to m. Again, no bits to decode. However, after the third case, MM-SEARCH needs to know $lcp(x_L, x_M)$ since it becomes the new value of $lcp(x, x_R)$. As previously remarked, this leads to a sub-optimal search in our case.

We make MM-SEARCH more parsimonious by introducing two modifications:

- First, we encode the precomputed *lcp* information in unary using a ternary string. For a non-negative integer $\ell \leq s$ (i.e. an *lcp* value), we define its "unary-ternary" representation as $a_0, a_1, \ldots, a_s$, where $a_0 = \cdots = a_{\ell-1} = 0$, $a_\ell = 1$, and $a_{\ell+1} = \cdots = a_s = 2$. Each such digit can be encoded by two bits. The clear advantage is that comparing an integer g to ℓ, where $0 \leq g \leq s$, just requires decoding an individual digit, a_g. The price to pay is that we require now $O(s)$ bits to represent an *lcp* value, instead of $O(\log s)$ bits.
- Second, we change the invariant maintained by the MM-SEARCH on interval $[L \ldots R]$. Since it requires the value of $lcp(x_L, x_M)$ to handle the third case (resp., $lcp(x_M, x_R)$ in the symmetric case), we have to decode the value of $lcp(x_L, x_M)$ from its "unary-ternary" representation, which gives a sub-optimal solution as previously mentioned. Instead, we make a simple but effective observation. We *only* keep m, which is the *lcp* value between the search key x and the key in $\{x_L, x_R\}$ that maximizes that *lcp*, say x_L. The actual value of $lcp(x, x_R)$ in the third case above is *not* functional to the search: we only need it to compare against m.

In summary, we only need the "unary-ternary" representation $a_0, a_1, \ldots, a_s$ of $lcp(x_L, x_M)$ (resp., $lcp(x_M, x_R)$), without explicitly computing its actual value. We rephrase the three cases of MM-SEARCH by examining a_m according to our modifications:

1. Case $a_m = 0$: Inferring that $m < lcp(x_L, x_M)$, set $L = M$ and do not change m (since x_L still maximizes the *lcp* value).
2. Case $a_m = 1$: Inferring that $m = lcp(x_L, x_M)$, proceed as in the original MM-SEARCH . The new value of $m = lcp(x, x_M)$ indicates that x_M is the best match (which becomes either x_L or x_R in the next search step, as previously mentioned).
3. Case $a_m = 2$: Inferring that $m > lcp(x_L, x_M)$, set $R = M$ and do not change m (since x_L still maximizes the *lcp* value).

As it should be clear, we do not need the actual value of $lcp(x_L, x_M)$, but only the outcome of its comparison with m (i.e. a_m, which are two bits). Moreover, when $m \neq lcp(x_L, x_M)$, the value of m is unchanged and x_L still maximizes the *lcp* value. Therefore, we can circumvent the drawback of decoding the entire value of $lcp(x_L, x_M)$ as required in MM-SEARCH and we can state the following lemma:

Lemma 4 *We can modify* MM-SEARCH *for r keys each of length s, so as to access $O(1)$ bits of precomputed lcp values per step during the binary search but still performing $O(s + \log r)$ character comparisons for the whole process. The total number of required bits to encode is $O(rs)$.*

We are now ready to describe the preprocessing of a twin interval $\mathscr{T}$ of length $b = |\mathscr{T}|$. We first partition $\mathscr{T}$ into blocks each of size $\Theta(\log b)$. We take the leftmost key in each block (from left to right) as the *leading* key of the block. So, we have overall $O(b/\log b)$ leading keys in $\mathscr{T}$. Finding a key inside a block can be done with the scanning search, in $O(k + \log n)$ time (since $b \leq n$).

In order to identify the suitable block of $\mathscr{T}$, we first need to search among the leading keys. We build an implicit suffix array on them for running the modified MM-SEARCH as follows. We consider each leading key, of length k, as composed by $s = \log b$ macro-characters. Each macro-character is made up of $\Theta(1 + k/\log b)$ symbols of the original key. So the comparison of any two macro-characters requires $O(1 + k/\log b)$ time. We encode the $O(s) = O(\log b)$ bits needed by the our modification of MM-SEARCH for each leading key, in its corresponding block of $\mathscr{T}$.

We recall that we measure the cost by accounting interchangeably for the number of comparisons and character probes, as they are linearly proportional to each other in our case.

Theorem 1 *For any (multi)set $\mathcal{S}$ of n keys each of length k, there exists a permutation $\Pi_{\mathcal{S}}$ of $\mathcal{S}$ that can be searched with $\Theta(k + \log n)$ comparisons and $O(1)$ auxiliary space. Starting from $\mathcal{S}$ in lexicographical order, $\Pi_{\mathcal{S}}$ can be constructed in $O(nk)$ time and $O(1)$ auxiliary space.*

Proof: We have shown so far how to reduce the search in $\mathscr{A}$ to the search in a twin interval, in $O(k + \log n)$ time by Lemma 3. We then continue the search among the leading keys of the twin interval. By Lemma 4, since $r = O(b/\log b)$ and $s = \log b$, we can perform a search with $O(\log b)$ macro-character comparisons, decoding $O(\log b)$ bits. This gives a total of $O(k + \log b)$ time as each macro-character comparison requires $O(1 + k/\log b)$ time. We then search inside a block in $O(k + \log b)$ time by applying the scanning search on $r = O(\log b)$ keys. We get the final bound since $b \leq n$. The preprocessing steps require a constant number of scans of the input array, $\mathscr{A}$, thus giving $O(nk)$ time. □

3.3 Answering rank and interval queries

A quick review of the search algorithm yielding the cost stated in Theorem 1, identifies four main search steps:

1. Zone A: here we have the keys in sorted order.
2. Zone B: here the blocks are relatively sorted, while inside each block we have a permutation.
3. Zone C: see Zone B.
4. Zone D: the left-to-right sequence of twin intervals is relatively sorted. Internally, each twin interval is permuted but the original order can be retrieved on the fly, since the keys in twin positions are the only ones being (pairwise) permuted.

Using the above scheme, we show how to extend the repertoire of supported operations:

- Successor and predecessor: given any key x, find the smallest key $y \in \mathscr{A}$ such that $x < y$, or the largest $y \in \mathscr{A}$ such that $x > y$.
- Rank: given any key x, report the number of keys $y \in \mathscr{A}$, such that $y \leq x$.
- One-dimensional range query: given any two keys a and b, where $a \leq b$, report all the keys $y \in \mathscr{A}$, such that $a \leq y \leq b$.

These operations can be implemented with minor modifications in each of zones A–D. We can then merge the four outcomes thus found.

Corollary 1 *Reporting the rank of a search key among those stored in $\mathscr{A}$, and computing its predecessor or successor, has a cost of $\Theta(k + \log n)$. Performing a one-dimensional range query in $\mathscr{A}$ has an output-sensitive cost of $\Theta(k+\log n+\#retrieved)$, where $\#retrieved$ denotes the number of keys in $\mathscr{A}$ that belong to the query interval.*

3.4 Searching in the bit probe model

We have seen so far how to attain the optimal search bound in the comparison model. When keys are strings whose characters are drawn from a constant-sized alphabet, we can opt for a different model. We consider here the case of binary strings, but it can be generalized to σ-ary strings for any $\sigma \geq 2$. We show how obtain an optimal bound for searching (membership) in the bit probe model [5, 19], as a corollary of Theorem 1. In this model, the keys are binary strings and the complexity accounts for the number of bits probed in the array $\mathscr{A}$ storing them. Array $\mathscr{A}$ is therefore a $k \times n$ *binary* matrix, where we store the keys permuted, so that membership requires $\Theta(k)$ bit probes in the worst case, which is optimal (i.e. we can drop the $\log n$ term resulting from the comparison model, as previously discussed). The occupied space is that of the keys, namely, nk bits.

Corollary 2 *In the bit probe model, n keys each of k bits can be permuted in $\mathscr{A}$, so that they occupy just nk bits and searching a key probes $O(k)$ bits in the worst case.*

Proof: Let n' be the number of distinct binary strings in $\mathscr{A}$, where $n' \leq n$. Note that $k \geq \log n'$, since there are at most 2^k distinct binary keys of length k. We move these distinct strings at the beginning of $\mathscr{A}$ and apply Theorem 1 to them. The techniques described in this paper are suitable for the bit probe model as well. Since their comparison cost asymptotically matches their character probe cost, we derive that the number of bit probes is $O(k + \log n') = O(k)$. □

4 Conclusions

In this paper, we have shown that a sorted array is not the best data organization for optimal searching n multi-dimensional keys. Our result sheds a further light on the relation between searching and sorting from a theoretical point of view. We describe a new organization of k-dimensional keys that is based upon a suitable permutation of them, and that allows us to search optimally in $O(k + \log n)$ time. These keys in sorted order cannot be optimally searched as a consequence of the results in [2] and in this paper. In this sense, disorder carries more information than order does.

References

[1] Arne Andersson, Torben Hagerup, Johan Håstad, and Ola Petersson. The complexity of searching a sorted array of strings. In ACM, editor, *Proceedings of the twenty-sixth annual ACM Symposium on the Theory of Computing: Montreal, Quebec, Canada, May 23–25, 1994*, pages 317–325, New York, NY, USA, 1994. ACM Press.

[2] Arne Andersson, Torben Hagerup, Johan Håstad, and Ola Petersson. Tight bounds for searching a sorted array of strings. *SIAM Journal on Computing*, 30(5):1552–1578, October 2001.

[3] Arne Andersson, Johan Håstad, and Ola Petersson. A tight lower bound for searching a sorted array. In ACM, editor, *Proceedings of the twenty-seventh annual ACM Symposium on the Theory of Computing*. ACM Press, 1995.

[4] Jon L. Bentley and Robert Sedgewick. Fast algorithms for sorting and searching strings. In ACM, editor, *Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, New Orleans, Louisiana, January 5–7, 1997*, pages 360–369, New York, NY 10036, USA, 1997. ACM Press.

[5] Peter Elias and Richard A. Flower. The complexity of some simple retrieval problems. *J. ACM*, 22(3):367–379, 1975.

[6] A. Fiat and M. Noar. Implicit $O(1)$ probe search. In ACM, editor, *Proceedings of the twenty-first annual ACM Symposium on Theory of Computing, Seattle, Washington, May 15–17, 1989*, pages 336–344, New York, NY, USA, 1989. ACM Press.

[7] Amos Fiat, J. Ian Munro, Moni Naor, Alejandro A. Schäffer, Jeanette P. Schmidt, and Alan Siegel. An implicit data structure for searching a multikey table in logarithmic time. *Journal of Computer and System Sciences*, 43(3):406–424, December 1991.

[8] Amos Fiat and Moni Naor. Implicit $O(1)$ probe search. *SIAM Journal on Computing*, 22(1):1–10, 1993.

[9] Amos Fiat, Moni Naor, Jeanette Schmidt, and Alan Siegel. Non-oblivious hashing. In ACM, editor, *Proceedings of the twentieth annual ACM Symposium on Theory of Computing, Chicago, Illinois, May 2–4, 1988*, pages 367–376, New York, NY, USA, 1988. ACM Press.

[10] Amos Fiat, Moni Naor, Jeanette P. Schmidt, and Alan Siegel. Nonoblivious hashing. *Journal of the ACM*, 39(4):764–782, October 1992.

[11] Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with $O(1)$ worst case access time. *J. ACM*, 31(3):538–544, 1984.

[12] D. S. Hirschberg. A lower worst-case complexity for searching a dictionary. In *Proc. 16th Annual Allerton Conference on Communication, Control, and Computing*, pages 50–53, 1978.

[13] D. S. Hirschberg. On the complexity of searching a set of vectors. *SIAM Journal on Computing*, 9(1):126–129, 1980.

[14] D. E. Knuth. *The Art of Computer Programming III: Sorting and Searching (2nd ed.)*. Addison–Wesley, Reading, Massachusetts, 1998.

[15] S. Rao Kosaraju. On a multidimensional search problem. In *Eleventh Annual ACM Symposium on Theory of Computing (STOC '79)*, pages 67–73, New York, April 1979. ACM.

[16] Udi Manber and Gene Myers. Suffix arrays: A new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, October 1993.

[17] 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.

[18] J. Ian Munro and Hendra Suwanda. Implicit data structures for fast search and update. *Journal of Computer and System Sciences*, 21(2):236–250, 1980.

[19] Andrew C. Yao. Should tables be sorted? *J. Assoc. Comput. Mach.*, 31:245–281, 1984.