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

Abstract

Sorting is commonly meant as the task of arranging keys in increasing or decreasing order (or small variations of this order). Given n keys underlying a total order, the best organization in an array is maintaining them in sorted order. Searching requires $\Theta(\log n)$ comparisons in the worst case, which is optimal.

We demonstrate that this basic fact in data structures does not hold for the general case of multi-dimensional keys, whose comparison cost is proportional to their length. In two papers [1, 3] and the full version in [2], Andersson, Hagerup, Håstad and Petersson study the complexity of searching a sorted array of n keys, each of length k, arranged in lexicographic (or alphabetic) order for an arbitrary, possibly unbounded, ordered alphabet. They give sophisticated arguments for proving a tight bound in the worst case for this basic data organization, up to a constant factor, obtaining

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

character comparisons (or probes). Note that the bound is $\Theta(\log n)$ when $k=1$, which is the case that is well known in algorithmics.

We describe a novel permutation *of the n keys that is different from the sorted order, and sorting is just the starting point for describing our preprocessing. When keys are stored according to this "unsorted" order in the array, the complexity of searching drops to*

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

character comparisons (or probes) in the worst case, which is optimal *among all possible permutations of the n keys in the array, up to a constant factor. Again, the bound is $\Theta(\log n)$ when $k=1$.*

Jointly with the aforementioned result of Andersson et al., our finding provably shows that keeping k-dimensional keys sorted in an array is not *the best data organization for searching. This fact was not observable before by just considering $k=O(1)$ as sorting is an optimal organization in this case. More implications of our result are commented in the introduction.*

1. Introduction

"Imagine how hard it would be to use a dictionary if its words were not alphabetized!" As noted at the beginning of Knuth's book [14], sorting n keys in an array 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. Does this viewpoint completely cover the intrinsic complexity of this basic problem? We show in this paper that sorting is *not* the best general way to organize multi-dimensional keys 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, 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 as well. 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 $x\equiv x[1\ldots k]$. 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 $x[j+1]<y[j+1]$. In this model, we can only compare the individual symbols of the keys (e.g., no hashing or bit manipulation involving these symbols). We measure the time complexity by accounting

---

[*] 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.

interchangeably for the number of comparisons or character 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 $k = 1$ we obtain the classical searching problem for an array $\mathscr{A}$. After sorting it, we can run binary searching with optimal cost $\Theta(\log n)$. For arbitrary values of k, 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 preprocessing, Andersson, Hagerup, Håstad and Petersson proved in [1, 3, 2] that it requires

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

character comparisons in the worst case. Note that the bound is $\Theta(\log n)$ when $k = 1$, which is a well known fact in algorithmics.

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 (used only as the starting point for defining our preprocessing). When keys are stored according to this "unsorted" order in the array, we show that the worst-case complexity of searching in the array is

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

character comparisons, which is asymptotically *optimal* among all possible permutations of the n keys in the array including those given by sorting. Also this bound is $\Theta(\log n)$ when $k = 1$.

As a result, we provide a (hopefully!) unexpected insight on the relation between sorting and searching. 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 for searching an array only when $k = O(1)$.

Additionally, we can compute the rank of a search key among those stored in $\mathscr{A}$, with a cost of $\Theta(k + \log n)$, and 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 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 data organization is more powerful than putting the keys in sorted order. As a side remark, we can obtain a sorted array from our permutation and vice versa in $O(nk)$ time; hence, our permutation is efficiently reversible.

We point out that the bounds above 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 $\mathscr{A}$ or the extra fields in ternary search trees by Bentley and Sedgewick [4]). The result of Andersson et al. deals with an array alone, without any additional preprocessing and 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 see the connection, let's recall that Munro and Suwanda [18] define an implicit data structure as a permutation of the keys plus just $O(1)$ cells. 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. [6], where the "O" includes a factor of $k \log k$. This method cannot be extended to solve our problem optimally.

Interestingly, our bounds hold also for the dictionary in the bit probe model [5, 19]. In this model, the keys are distinct binary strings and the complexity accounts for the number of bits probed in the array $\mathscr{A}$ storing them. We can store these keys permuted in nk bits, 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., for 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 [8, 9], and implicit probe search by Fiat and Naor [10, 7].

Our presentation is organized as follows. In Section 2, we show how to encode bits with keys of unbounded 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.

2. Permuting for Encoding Bits of Information

In this section we provide a novel tool for encoding and decoding bits with keys of unbounded 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. 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 from the sorted sequence of r keys in non-decreasing order, we show how to encode bits according to a new scheme, which we call *ditch*. We obtain two incarnations: *small* ditches, obtained by permuting $r = O(\log n)$ keys; *large* ditches 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 $lcp(x, y) = \max\big(\{0\} \cup \{1 \leq \ell \leq k \; : \; x[1 \ldots \ell] = y[1 \ldots \ell]\}\big)$ to denote the length of their longest common prefix.

We first describe the ditch in Section 2.1 and some properties common to small and large ditches. We then show how to encode a large ditch itself in Section 2.2. We describe how to preprocess $\mathscr{A}$ by encoding a large ditch in it.

2.1. The ditch: A basic tool for encoding 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 $i + j = r + 1$; that is, the number of keys up to position i equals that of keys from position j to the end. Let's "dig" into the sequence by comparing incrementally the keys at twin positions. 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 twin positions* i *and* j*. The total cost of* DIGGING *for* r *sorted keys of length* k *is* $O(k + r)$ *time.*

Proof: Since $lcp(x_{i'}, x_{j'}) \leq lcp(x_i, x_j)$ for $i' < i < j < j'$, the digging depth for i and j is surely at least d, before the execution of the inner loop. After that loop, we consequently have $d = lcp(x_i, x_j) + 1$. 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 the number r of keys in the sequence. This gives the final bound on the cost of digging. □

Since the keys involved in digging are sorted, 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 for encoding and decoding bits. We adopt a simple 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, when their digging depth d is known. 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 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 $j = j' - 1$. If we do not know their digging depths, we cannot apply directly the constant-time decoding of individual bits described above. We can circumvent this drawback as stated 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 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. A mismatch causing a loop exit 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 a *large* ditch, with $r = O(n)$, we need to encode the positional information of the ditch's components to avoid a full scan of the r keys.

2.2. Encoding large ditches

In this section, we focus on *large* ditches where the $r = O(n)$ sorted keys are to be permuted suitably. Consequently, we have to encode the information representing a large ditch itself, and define the structure deriving from the digging process. Pictorially, the ditch is the concatenation of two specular stairs, one descending and one ascending, as shown in Figure 1. The characters implicitly removed on a row during the digging, are equal to the homologous characters of the median key. Let's fix any row d such that d is one of the digging depths computed by procedure DIGGING. There can be further characters (not shown in the figure) outside the ditch in row d that are equal to the median's; however, they can only extend beyond one side of the ditch rather than both sides (otherwise the ditch would be larger).

As noted in the proof of Lemma 1, the digging depth d is monotonically non-decreasing. Thus we can split the sequence of digging depths into *twin intervals*, as shown in Figure 1. They collect all the twin positions giving rise to the same digging depth d. These positions are contiguous and form two twin intervals $\mathscr{T}', \mathscr{T}''$, with $|\mathscr{T}'| = |\mathscr{T}''|$, whose digging depth is d. Using the observation above, we can scan the twin intervals from left to right and number them for reference purposes. Note that we cannot have more than $2k$ twin intervals.

Using the features above, we represent a ditch by recording somewhere 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 $i'' \in \mathscr{T}''$.

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 have seen so far how to encode bits. However, the ditch is not yet able to route the search optimally towards a twin interval. Another goal of the ditch is indeed reducing the problem of searching a key in $\mathscr{A}$ to that of searching among the keys of one selected twin interval. For this, we need to single out the leftmost key in $\mathscr{T}$, for each twin interval $\mathscr{T}$ from left to right.

It is worth noting that searching inside a twin interval is easier than the original search in $\mathscr{A}$ as we can rely on the implicitly encoded bits in the interval. Indeed, each pair of twin intervals $\mathscr{T}', \mathscr{T}''$ encodes b bits, where $b = |\mathscr{T}'| = |\mathscr{T}''|$. Hence, we can assign $b/2$ bits to each of them; say, the bits encoded by odd positions to $\mathscr{T}'$ and those encoded by even positions to $\mathscr{T}''$. We defer the details on how to exploit this to Section 3.

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 sorted 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 $\mathscr{A}$ to form the large ditch (in zones C and D). If the number of distinct keys in $\mathscr{A}$ is smaller than that, we are in a special case which we can easily handle using [2], for example. Otherwise, starting from the sorted sequence of these distinct keys, we divide them into k blocks of $\Theta(\log n)$ keys each. We move the leftmost key of each block, in left-to-right order, to zone A. We thus form a small directory of $O(k)$ keys for identifying a block in zone B, which hosts the rest of these keys. The ith block implicitly encodes $O(\log n)$ bits with a small ditch. The encoded integers are the positional information for the ith pair of twin intervals in the large ditch (of zones C and D). We use Lemma 2 for the small ditch in the block, with $b = O(\log n)$. Hence, decoding the information for a pair of twin intervals requires $O(k + \log n)$ time by accessing zone B. The purpose of zone A is to make zone B searchable. (Alternatively, we can employ techniques from [2] to combine zones A and B.)

Zones C and D contain the rest of the keys and properly form the large ditch. We move the leftmost key from each twin interval of the ditch to zone C (note that the keys in this zone are in lexicographic order). The remaining keys remain in zone D (they are still sorted at this stage but will be permuted later on). The $O(k)$ keys in zone C constitute a small search directory to locate a twin interval in zone D. We refer the reader to Section 3 for the preprocessing within a twin interval and the adopted searching method.

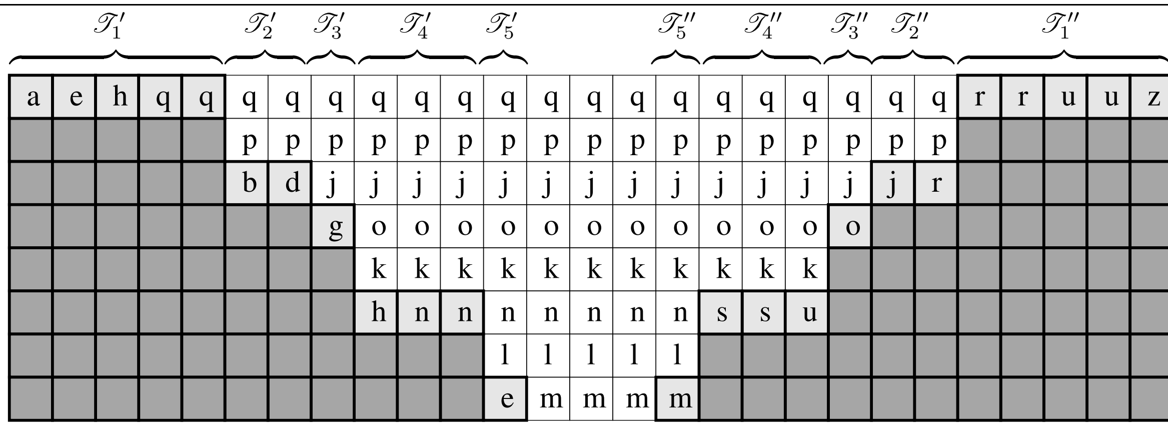

Figure 1. A ditch (ligth gray) encoding all 0s; hence, no two keys in twin positions are swapped.

3. Searching with Few Bits of Information

We now can give a description of our search, which uses Hirschberg's search [12] as a basic routine. The latter takes $O(k + r)$ time for r keys and we refer to it as *scanning search*. If the r keys encode bits with our digging methods, it's not difficult to combine our on-the-fly decoding based on digging and the scanning search, attaining $O(k + r)$ total time. This is an immediate corollary to Lemma 2.

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 apply here the (combined) scanning search in $O(k + \log n)$ time.

If we do not find the key, 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. Again, we run the scanning search on the $r = O(k)$ keys in that zone, in $O(k)$ time.

If the key is not in zone C, we end up in one position, say, that of the ith key in zone C. We can identify the unique twin interval containing x in zone D (if any; if x occurs more than once, we take its leftmost occurrence). Indeed, we can infer that we have to access the ith twin interval, say $\mathscr{T}$. We go to zone B and decode the positional information of $\mathscr{T}$ by scanning $b = O(\log n)$ keys in the corresponding block of that zone, in $O(k + \log n)$ time by Lemma 2. Afterward we compute 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 key x in the array $\mathscr{A}$ to the problem of searching x into twin interval $\mathscr{T}$ of zone D. We therefore focus on such interval $\mathscr{T}$ for preprocessing and searching by exploiting the fact that we can encode implicitly up to $b/2$ bits in it, where $b = |\mathscr{T}|$. In the full version, we show that answering rank and interval searches are a minor variation of our algorithms.

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

3.2. Searching within a twin interval

We describe how to preprocess and search a twin interval $\mathscr{T}$, assuming that we can encode up to $b/2$ bits by permuting keys, where $b = |\mathscr{T}|$. If $|\mathscr{T}| = O(\log n)$, we run the (combined) 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 of Manber and Myers [16], which we refer to as MM-SEARCH . Originally designed for an array of suffixes, it finely works also for arbitrary keys of unbounded length in lexicographic order. However, MM-SEARCH and its variations devised thereafter access too many bits. We propose a variant that is tailored for our twin intervals in that it is less demanding in the number of bits to be decoded.

We first review in a nutshell how MM-SEARCH works for a set of r keys of length s each. It performs a variation of the classical binary search according to three cases. For the current search interval, $[L \dots R]$, we denote the middle point by M. The invariant maintains the lcp value between the search key x and the keys x_L, x_R at positions L and R, respectively. We know by induction that

$x_L \leq x \leq x_R$. We want to infer 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 $[L \ldots M]$ or $[M \ldots R]$. For the sake of discussion, suppose that $lcp(x_L, x) \geq lcp(x, x_R)$, and let $m = lcp(x_L, x)$ be the number of initial symbols in x that have been successfully matched so far.

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 $m = lcp(x, x_M)$ 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 $R = M$.

In order to maintain its invariant, MM-SEARCH requires to maintain the values of $lcp(x_L, x)$ and $lcp(x, x_R)$ after dealing with each of the three cases. The first two cases are not a problem. In the first case, $lcp(x_L, x) = lcp(x, x_M)$, which is also the current value of m. In the second case, $lcp(x_L, x)$ or $lcp(x, x_R)$ has been just computed as well since it becomes equal to m. However, in the third case, MM-SEARCH needs $lcp(x_L, x_M)$ since it becomes the new value of $lcp(x, x_R)$. This requires reading $O(\log s)$ bits somewhere. Unfortunately, it would give a sub-optimal search in our case.

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

- We encode the required 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.
- 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, we have to decode the value of $lcp(x_L, x_M)$ from its "unary-ternary" representation. This is equivalent to identifying the digit $a_{lcp(x_L,x_M)} = 1$ by accessing $\Omega(\log s)$ bits from that representation. Instead, we make a simple but effective observation. We keep *only* 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. In other words, the value of $lcp(x, x_R)$ in the third case above is not actually functional to the search as it is smaller than m.

In summary, we only need the "unary-ternary" representation $a_0, a_1, \ldots, a_s$ of $lcp(x_L, x_M)$, without explicitly computing the actual value of $lcp(x_L, x_M)$. 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 avoid to decode the entire value of $lcp(x_L, x_M)$.

Lemma 4 *We can modify* MM-SEARCH *for r keys of length s so as to access $O(1)$ bits of lcp values per step during the binary search. We represent each lcp value by $O(s)$ bits to work, for a total of $O(rs)$ bits storing the lcp information. The resulting* MM-SEARCH *requires a total of $O(\log r)$ bits of the latter kind, and $O(s + \log r)$ character comparisons.*

Proof: As previously mentioned, we store the lcp information needed by MM-SEARCH in "unary-ternary", thus requiring no more than $O(s)$ bits per lcp value. Since the number of the lcp values is bounded by the number of keys, this yields a total of $O(rs)$ bits. Our modification of MM-SEARCH accesses just $O(1)$ of these bits for each step. Since there are $O(\log r)$ searching steps, this gives a total of $O(\log r)$ accessed bits. Note that the number of character comparison remains unchanged, $O(s + \log r)$, as in the original MM-SEARCH . □

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 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 (combined) scanning search, in $O(k + \log n)$ time (since $b \leq n$).

In order to identify the suitable block of $\mathscr{T}$, we first search the leading keys. We consider each leading key, of length k, as composed by $\log b$ 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 $s = O(\log b)$ bits needed by the modified MM-SEARCH in Lemma 4 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 *Given a sequence of n keys of length k each, in lexicographic order, we can preprocess them by permuting in $O(nk)$ time, so that each search has a cost of $\Theta(k + \log n)$, which is optimal.*

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. We 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. We get the final bound since $b \leq n$. $\square$

3.3. Searching in the bit probe model

As a corollary of Theorem 1, we obtain an optimal search bound in the bit probe model [5, 19]. In this model, the keys are distinct 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. The occupied space is that of the keys, namely, nk bits.

Corollary 1 *In the bit probe model, n keys of k bits each 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, which is optimal.*

Proof: If $k > 1/2 \log n$, we use Theorem 1 adapted to the bit probe model. Otherwise, we access one of the all $O(\sqrt{n})$ possible answers that we encode in the permutation itself. This requires decoding $O(1)$ bits, and so accessing $O(1)$ keys. $\square$

4. Conclusions

In this paper, we have shown that a sorted array is not the best data organization for optimal searching n multidimensional 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. Nevertheless, we still like to consult dictionaries of alphabetized words!

Acknowledgements

We thank the anonymous Referees for their helpful comments on the presentation.

References

[1] A. Andersson, T. Hagerup, J. Håstad, and O. 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] 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, Oct. 2001.

[3] A. Andersson, J. Håstad, and O. 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] J. L. Bentley and R. 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] P. Elias and R. A. Flower. The complexity of some simple retrieval problems. *J. ACM*, 22(3):367–379, 1975.

[6] A. Fiat, J. I. Munro, M. Naor, A. A. Schäffer, J. P. Schmidt, and A. Siegel. An implicit data structure for searching a multikey table in logarithmic time. *Journal of Computer and System Sciences*, 43(3):406–424, Dec. 1991.

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

[8] A. Fiat, M. Naor, J. Schmidt, and A. 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.

[9] A. Fiat, M. Naor, J. P. Schmidt, and A. Siegel. Nonoblivious hashing. *Journal of the ACM*, 39(4):764–782, Oct. 1992.

[10] 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.

[11] M. L. Fredman, J. Komlós, and E. 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*. Addison–Wesley, Reading, Massachusetts, 1973.

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

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

[17] 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.

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

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