Computing and Informatics, Vol. 25, 2006, 333–350

LINEAR-TIME IN-PLACE SELECTION WITH $\varepsilon \cdot N$ ELEMENT MOVES[*]

Viliam Geffert

Department of Computer Science
P. J. Šafárik University
Jesenná 5, 040 01 Košice, Slovakia
e-mail: geffert@upjs.sk

Ján Kollár

Department of Computers and Informatics
Technical University of Košice
Letná 9, 042 00 Košice, Slovakia
e-mail: Jan.Kollar@tuke.sk

Manuscript received 11 April 2004; revised 27 March 2006
Communicated by Marián Vajteršic

Abstract. We present a new in-place selection algorithm that finds the k^{th} smallest element in an array of n elements in linear time, using only $\varepsilon \cdot n$ element moves. Here $\varepsilon > 0$ denotes an arbitrarily small, but fixed, real constant. As a consequence, partitioning the array in-place into segments of elements with ranks smaller than, equal to, and larger than k can be performed with $(1 + \varepsilon) \cdot n$ element moves. Minimizing the sum of comparisons and moves, we get a selection algorithm using $C(n) < 10.236n$ comparisons and $M(n) < 0.644n$ moves. The algorithm can be further optimized, by tuning up for the given cost ratio between a single move and a single comparison. As an example, we present an algorithm with $C(n) + 10 \cdot M(n) \leq 13.634n$.

Keywords: Algorithms, in-place selection, sorting

[*] This work was supported by the Slovak Grant Agency for Science (VEGA) under the contract "Combinatorial Structures and Complexity of Algorithms".

1 INTRODUCTION

The problem of finding the k^{th} smallest element in a multiset of n elements, drawn from a totally-ordered universe, has been the subject of intense investigation. For a long time, it was believed to be as difficult as sorting. Surprisingly, it was shown to be solvable in linear time by Blum et al. in 1973 [2]. This result was improved in 1976 by Schönhage, Paterson, and Pippenger [13], giving an upper bound of $3n + o(n)$ comparisons in the worst case. Despite many attempts, this had not been improved until 1994, when Dor and Zwick [4, 5] gave a $2.95n+o(n)$ algorithm. They also raised slightly the lower bound, from $2n - o(n)$ comparisons in [1], to $(2 + 2^{-40}) \cdot n - o(n)$.

All these algorithms use some additional space, the amount of which grows in n. This is needed for storing array indices, counters, etc. For example, a straightforward implementation of the algorithm by Schönhage et al. [13] requires additional linear space.

This triggered the search for an efficient *in-place* selection algorithm, i.e., an algorithm using only a constant amount of auxiliary variables, apart from the array storing the n elements. The in-place feature remains important even in the face of dramatically reduced memory costs, since in-place algorithms maximize the size of the file that can be processed without an access to a secondary storage; moreover, such algorithms do not use a dynamic memory allocation. Logical elegance is also an important feature of such algorithms. From a theoretical point of view, it is interesting to know whether we can replace some traditional storage requirements by storing information implicitly into ordering of elements in such a way that we can avoid recomputing.

The first in-place selection algorithm running in linear time was developed by Lai and Wood in 1988 [9]. This algorithm uses fewer than $6.83n + o(n)$ comparisons and $18.69n + o(n)$ moves. The most sophisticated versions known so far were presented by Carlsson and Sundström in 1995 [3]. The first one is only $\varepsilon \cdot n$ comparisons away from the best known upper bound without space restriction, it uses $(2.95 + \varepsilon) \cdot n$ comparisons, but this is paid by a fairly large number of moves, $O((1/\varepsilon)^2 \cdot n)$. The second one uses $3.75n + o(n)$ comparisons, but the number of moves is reduced to $9n + o(n)$.

In the algorithms above, the authors attempted to minimize the number of element comparisons. The number of element moves (assignments) was only a second-order criterion. However, in a typical realistic application, the cost of moving an element is much larger than the cost of a comparison. The relative order of an element is usually induced by a relative order of some *key*, forming a small portion of data within a large record. Thus, transportation-efficient in-place algorithms are (at least) as important as comparison-efficient versions.

For the number of element moves, we do not have any lower bound corresponding to $(2+2^{-40}) \cdot n - o(n)$ for the number of comparisons. One can easily design a simple $O(n^2)$ in-place algorithm that finds the k^{th} smallest element with no element moves at all. That is, in-place selection can be performed even if the elements reside in a *read-only* memory. The best known algorithm of this kind was developed by Munro and Raman [11], using a constant number of index variables, no element moves, and $O(n^{1+\varepsilon})$ comparisons, for any constant $\varepsilon > 0$. However, the constant associated with the $O(n^{1+\varepsilon})$ term grows exponentially in $1/\varepsilon$.

The in-place selection algorithm we shall present here uses only $\varepsilon \cdot n$ element moves, while keeping the number of comparisons and other auxiliary operations bounded by $O(n)$. Here $\varepsilon > 0$ denotes an arbitrarily small, but fixed, real constant. Moreover, the constant in the $O(n)$ term grows only logarithmically in $1/\varepsilon$. Partitioning the array in-place into segments of elements with ranks smaller than, equal to, and larger than k can be performed with $(1+\varepsilon)\cdot n$ moves. The number of index variables is bounded by $O((1/\varepsilon)^2 \cdot \log(1/\varepsilon))$.

Minimizing the sum of comparisons and moves, we get a selection algorithm using $C(n) < 10.236n$ comparisons and $M(n) < 0.644n$ moves. The algorithm can be further optimized, by tuning up for the given cost ratio between a single move and a single comparison. This improves the total computational cost in applications where the key of an element forms a small portion in a large record of data. As an example, we present an algorithm with $C(n) + 10 \cdot M(n) \le 13.634n$.

Our algorithm is based on ideas presented in [2] and [9]. Actually, we can use any $O(n)$ in-place selection algorithm as a starting point, even with some large constant factors. To speed up, we select a properly chosen sample of $\varepsilon \cdot n$ elements that represents the distribution of elements in the original sequence. The sample is very similar to that used by Munro and Raman in [10], where they minimize the number of moves in sorting algorithms. By finding two elements y, x of carefully chosen ranks in this sample, we get two elements such that the k^{th} smallest element a, in the original sequence, satisfies $y \le a \le x$, with the number of elements between y and x so small that a can be located by selection from $\varepsilon \cdot n$ candidates.

2 PRELIMINARIES

We first need to introduce some notation. Let $\mathcal{A} = \{a_1, \ldots, a_n\}$ denote the multiset of n elements. An element $a \in \{a_1, \ldots, a_n\}$ *is of rank* k, for some $k \in \{1, \ldots, n\}$, if the number of elements smaller than or equal to a in $\{a_1, \ldots, a_n\}$ is at least $L_a \ge k$, and the number of elements greater than or equal to a is at least $R_a \ge n - k + 1$. The *upper rank* of a, denoted by $a^{\succ}$, is the largest $k \in \{1, \ldots, n\}$ such that a is of rank k. Similarly, the *lower rank* $a_{\prec}$ is the smallest $k \in \{1, \ldots, n\}$ such that a is of rank k.

If all elements in the multiset $\mathcal{A} = \{a_1, \ldots, a_n\}$ are different from a (excluding the element a itself), the rank of a is unique, with $L_a + R_a = n + 1$ and $a_{\prec} = a^{\succ}$. However, since several elements in $\mathcal{A}$ may be equal to a, we only have $L_a + R_a \ge n+1$ and $a_{\prec} \le a^{\succ}$. Clearly, a is of rank k for each $k \in \{a_{\prec}, \ldots, a^{\succ}\}$.

The *selection problem* is to find an element of rank k, given the value of k and n elements $a_1, \ldots, a_n$, stored in some array $\mathcal{A}$. The array is not sorted. We can assume that

$$1 \le k \le \lceil n/2 \rceil, \qquad (1)$$

for, if $k > n/2$, the problem can be solved symmetrically by seeing the last entry of $\mathcal{A}$ as the beginning of the input and by searching for the $(n-k+1)^{\mathrm{st}}$ largest element.

The elements are atomic, that is, they can only be moved or compared with the operations $\leq$ and $=$. To allow the elements move, some constant number of elements can be put aside, to extra locations. (We shall actually use only a single extra location in our selection algorithm, or for two-way partitioning. For efficient three-way partitioning, we need two extra locations.) We can also use a constant amount of index variables, of $1 + \lfloor \log_2 n \rfloor$ bits each. The standard arithmetic operations and comparisons are used in manipulation of these.

The in-place selection problem can be solved in linear time, using $c_{\mathrm{c}} \cdot n + o(n)$ comparisons and $c_{\mathrm{m}} \cdot n + o(n)$ moves, where c_{c} and c_{m} are some constants. Here we can use either a conceptually simpler algorithm with $c_{\mathrm{c}} \leq 6.83$ and $c_{\mathrm{m}} \leq 18.69$ [9], or a more efficient, but much more complicated, version with $c_{\mathrm{c}} = 3.75$ and $c_{\mathrm{m}} = 9.00$ [3].

3 SELECTION ALGORITHM

Now we are ready to present our algorithm for the selection problem. The algorithm depends on two built-in integer constant parameters $q, \ell \geq 1$. (They can be fixed quite arbitrarily; however, the number of element comparisons varies, and index variables depend on the choice.) We shall use also a third built-in constant, defined by

$$d = q \cdot (\ell + 1) + \ell = q \cdot \ell + q + \ell.$$

Phase 1. Divide the input array into sequences of length d. The last one will be of length d', satisfying $1 \leq d' \leq d$. Sort each segment and select, in each of them, q evenly distributed sample elements so that there are ℓ nonsample elements between each two adjacent sample elements, and also before the first and after the last sample element, within the segment. That is, pick up the $(\ell+1) \cdot i^{\mathrm{th}}$ element of each segment, for $i = 1, \ldots, q$. (See Figure 1.)

If n, the length of the input array, is not an integer multiple of d, the last segment is of length $d' < d$. If, moreover, $d' \leq 2\ell$, no sample elements are picked up in the last segment. Otherwise, for $d' > 2\ell$, pick up the $(\ell+1) \cdot i^{\mathrm{th}}$ element, for $i = 1, \ldots, \lfloor (d' - \ell)/(\ell+1) \rfloor$, so that even here each sample element is enclosed in between ℓ nonsample elements. The last sample element is followed by ℓ' nonsample elements, with $\ell' \in \{\ell, \ldots, 2\ell\}$. (See Figure 1.)

Move all sample elements into one contiguous place, in the leftmost locations of the array $\mathcal{A}$, by swapping them with any nonsample elements.

Before passing further, let us consider the computational cost for the first phase. To sort a segment of length d, we use the Ford-Johnson's algorithm [6]. It is known [7] that this algorithm sorts d elements by using $f(d) = \sum_{i=1}^{d} h_i$ comparisons, where $h_i = \lceil \log_2(3i/4) \rceil$. This sum is bounded by $f(d) \leq d \cdot \log_2 d - (2 - \log_2 3 + 2^{\alpha} - \alpha) \cdot d + O(\log d)$, where $\alpha = \lceil \log_2(3d/4) \rceil - \log_2(3d/4)$, which is a real value ranging between

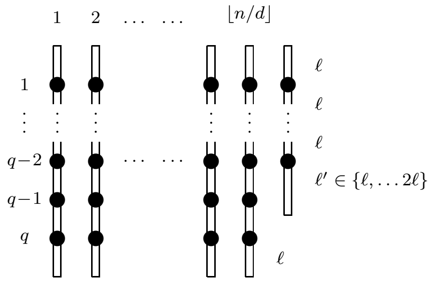

Fig. 1. Division of $\mathcal{A}$ into sorted segments. The segments are represented by vertical bars, sample elements by bullets.

zero and one. (For argument, see Answers to Exercises 14 and 15 of Section 5.3.1 in [8].) This allows us to derive an upper bound

$$f(d) \le d \cdot \log_2 d - 1.328d + O(\log d). \tag{2}$$

Since we sort $\lceil n/d \rceil$ segments of length at most d, the number of comparisons in the first phase is bounded by

$$C_1 \le \lceil n/d \rceil \cdot f(d) \le n \cdot f(d)/d + f(d).$$

Let us now consider the number of moves. The sorting itself does not require any moves. All segments are of constant lengths, and hence, instead of moving elements, we only rearrange pointers in a constant number of index variables. However, all sample elements are gathered into one contiguous place, in the leftmost locations of $\mathcal{A}$. These moves are organized as follows.

During the first phase, the configuration of the array $\mathcal{A}$ is of the form $SeNU$, where S represents the block of sample elements collected so far, e is the hole where the next sample element should go, N the block of discarded nonsample elements, and U the sequence of segments not yet processed. Initially, S and N are empty. The hole is created when the first segment is being processed. Consider first the general case, when the first segment is done.

First, the permutation of d elements, forming the leftmost segment of U, is determined by the Ford-Johnson's algorithm. Then, one after another, the q sample elements of this segment are collected to the left. Namely, each sample element is moved into the hole, and the next element on the right of this hole is moved to the place released by the sample element just moved. Thus, the current segment vanishes, contributing the sample and nonsample elements to S and N, respectively. This is repeated until all segments are done.

Processing of the first segment in $\mathcal{A}$ is slightly different, it requires one additional move. This is used to put the first nonsample element aside, to create the hole e. This element returns back to $\mathcal{A}$ when the last segment has been processed, to fill up the hole after the last sample element.

The number of moves is thus equal to $1 + 2n'$, where n' denotes the size of the sample. Since there are $\lceil n/d \rceil$ segments, each contributing at most q sample elements, the size of the sample is

$$n' \leq \lceil n/d \rceil \cdot q \leq n \cdot q/d + q. \tag{3}$$

Thus, the number of moves is at most

$$M_1 = 1 + 2n' \leq n \cdot 2 \cdot q/d + O(q).$$

Phase 2. There are two cases to consider. If

$$k \geq n \cdot \ell/d + (2\ell + 3), \tag{4}$$

then, using any other linear-time in-place selection algorithm, search the sample for two elements x, y, of ranks

$$\begin{aligned} k_x &= \lceil k/(\ell+1) \rceil, \\ k_y &= n' - \lceil (n-k+1)/(\ell+1) \rceil + 1, \end{aligned} \tag{5}$$

respectively, where k is the original rank of wanted element and n the original problem size. As will be shown later, the ranks given in (5) guarantee that the k^{th} smallest element a, in the original array $\mathcal{A}$, satisfies $y \leq a \leq x$, with the number of elements in the sorted sequence between y and x so small that a can be located by selection from $\varepsilon \cdot n$ candidates.

If (4) does not hold, i.e., for $k < n \cdot \ell/d + (2\ell + 3)$, we find only the element x in the sample, of the same rank $k_x = \lceil k/(\ell+1) \rceil$. In this case, k is so small that the wanted element a can be found among elements satisfying $a \leq x$, again from $\varepsilon \cdot n$ candidates.

Depending on whether k satisfies the condition (4), we select one or two elements from the sample of size n'. In either case, this can be done with $C_2 \leq 2c_{\mathrm{c}} \cdot n' + o(n')$ comparisons and $M_2 \leq 2c_{\mathrm{m}} \cdot n' + o(n')$ moves, for some constants c_{c} and c_{m}. Using (3), we get

$$\begin{aligned} C_2 &\leq n \cdot 2 \cdot c_{\mathrm{c}} \cdot q/d + o(n), \\ M_2 &\leq n \cdot 2 \cdot c_{\mathrm{m}} \cdot q/d + o(n). \end{aligned}$$

Before passing to the third phase, we shall analyze how x and y rank among the elements in the sample, as well as among the elements in the original sequence. This requires to calculate a lower bound for n'. Recall that there are $\lfloor n/d \rfloor$ segments of the full length d, each contributing exactly q sample elements. If n is not an integer multiple of d, there is also a shorter segment of length $d' = n - \lfloor n/d \rfloor \cdot d$ at the end. This segment contributes either $q' = 0$ or $q' = \lfloor (d' - \ell)/(\ell+1) \rfloor = \lfloor (n - \lfloor n/d \rfloor \cdot d - \ell)/(\ell+1) \rfloor$ sample elements, depending on whether $d' \leq 2\ell$ or $d' > 2\ell$, respectively. But then, using $q \cdot (\ell+1) - d = -\ell$, the size of the sample can be bounded from below by

$$\begin{aligned}
n' &= \lfloor n/d \rfloor \cdot q + \max\{0, \lfloor (n - \lfloor n/d \rfloor \cdot d - \ell)/(\ell+1) \rfloor\} \\
&\geq \lfloor n/d \rfloor \cdot q \cdot (\ell+1)/(\ell+1) + [(n - \lfloor n/d \rfloor \cdot d - \ell)/(\ell+1) - 1] \\
&= n/(\ell+1) + \lfloor n/d \rfloor \cdot [q \cdot (\ell+1) - d]/(\ell+1) - \ell/(\ell+1) - 1 \\
&\geq n/(\ell+1) - n/d \cdot \ell/(\ell+1) - (2\ell+1)/(\ell+1).
\end{aligned}$$

Now, using $(2\ell+1)/(\ell+1) < 2$, we get

$$n' > n \cdot (1 - \ell/d)/(\ell+1) - 2. \tag{6}$$

Now we can show that $1 \leq k_x \leq n'$, i.e., that the rank k_x, defined by (5), is meaningful for the sample and hence the element x does exist, for each sufficiently large n. Using (5), (1), $\ell \geq 1$, $n \geq (7\ell+7)/(1-2\ell/d)$, and (6), in that order, we get

$$\begin{aligned}
k_x &= \lceil k/(\ell+1) \rceil \leq \lceil \lceil n/2 \rceil/(\ell+1) \rceil \\
&\leq [n/(2\ell+2) + 1/(\ell+1)] + 1 \leq n/(2\ell+2) + 7/2 - 2 \\
&= n/(2\ell+2) + (7\ell+7)/(1-2\ell/d) \cdot (1-2\ell/d)/(2\ell+2) - 2 \\
&\leq n/(2\ell+2) + n \cdot (1-2\ell/d)/(2\ell+2) - 2 = n \cdot (1-\ell/d)/(\ell+1) - 2 < n'.
\end{aligned}$$

On the other hand, it is trivial to see that

$$k_x = \lceil k/(\ell+1) \rceil \geq 1,$$

since $k \geq 1$ and $\ell \geq 1$. Summing up, $1 \leq k_x \leq n'$.

Recall that the element y of rank k_y has been selected only if k satisfies (4), that is, only if $k \geq n \cdot \ell/d + 2 \cdot (\ell+1) + 1$. Using this, together with (5) and (6), we get

$$\begin{aligned}
k_y &= n' - \lceil (n-k+1)/(\ell+1) \rceil + 1 \geq n' - (n-k+1)/(\ell+1) \\
&= k/(\ell+1) + n' - n/(\ell+1) - 1/(\ell+1) \\
&> [n \cdot \ell/d + 2 \cdot (\ell+1) + 1]/(\ell+1) + [n \cdot (1-\ell/d)/(\ell+1) - 2] \\
&\quad - n/(\ell+1) - 1/(\ell+1) \\
&= 0.
\end{aligned}$$

That is, k_y is an integer greater than zero, and hence $k_y \geq 1$. For completeness, using (5) and (1) for sufficiently large n satisfying $n \geq 2 \cdot (\ell+1)$, we have

$$\begin{aligned}
k_y &= n' - \lceil (n-k+1)/(\ell+1) \rceil + 1 \leq n' - (n - \lceil n/2 \rceil + 1)/(\ell+1) + 1 \\
&= n' - (\lfloor n/2 \rfloor + 1)/(\ell+1) + 1 \leq n' - (\lfloor \ell+1 \rfloor + 1)/(\ell+1) + 1 \leq n'.
\end{aligned}$$

Summing up, $1 \leq k_y \leq n'$, for each k satisfying (4).

Now we can derive tight bounds for the ranks of x and y in the *original* sequence. The number of elements smaller than or equal to x, in the sample, is at least k_x (including x itself). With each element $a \leq x$ in the sample, we can cluster a separate group of ℓ nonsample elements smaller than or equal to a, within the same segment which the element a came from. (See Fig. 1.) Therefore, there are at least $k_x \cdot (\ell+1)$ elements smaller than or equal to x in $\mathcal{A}$. Thus, the upper rank of x in $\mathcal{A}$ is at least

$$x^{\succ} \;\geq\; k_x \cdot (\ell+1) = \lceil k/(\ell+1) \rceil \cdot (\ell+1) \geq k,$$

by (5). Similarly, the sample contains at least $n' - k_x + 1$ elements greater than or equal to x. (Again, including x itself.) But then the original sequence $\mathcal{A}$ must contain at least $(n' - k_x + 1) \cdot (\ell+1)$ elements greater than or equal to x, since, with each sample element $a \geq x$, we can cluster a separate group of ℓ nonsample elements greater than or equal to a. Thus, the lower rank of x in $\mathcal{A}$ is, by (6) and (5), at most

$$\begin{aligned} x_{\prec} &\leq n - (n' - k_x + 1) \cdot (\ell+1) + 1 \\ &< n - [n \cdot (1 - \ell/d)/(\ell+1) - 2] \cdot (\ell+1) + [k/(\ell+1) + 1] \cdot (\ell+1) \\ &\quad - (\ell+1) + 1 \\ &= k + n \cdot \ell/d + (2\ell + 3). \end{aligned}$$

By the same argument, using (5) and (6), we get the bounds for the ranks of y:

$$\begin{aligned} y_{\prec} &\leq n - (n' - k_y + 1) \cdot (\ell+1) + 1 \\ &= n - n' \cdot (\ell+1) + [n' - \lceil (n-k+1)/(\ell+1) \rceil + 1] \cdot (\ell+1) - (\ell+1) + 1 \\ &= n - \lceil (n-k+1)/(\ell+1) \rceil \cdot (\ell+1) + 1 \\ &\leq k, \end{aligned}$$

and

$$\begin{aligned} y^{\succ} &\geq k_y \cdot (\ell+1) > [(n \cdot (1 - \ell/d)/(\ell+1) - 2) \\ &\quad - \lceil (n-k+1)/(\ell+1) \rceil + 1] \cdot (\ell+1) \\ &> n \cdot (1 - \ell/d) - 2 \cdot (\ell+1) - (n - k + 1) \\ &= k - n \cdot \ell/d - (2\ell + 3). \end{aligned}$$

Note that $y^{\succ} > 0$, for k satisfying (4).

This implies that $\mathcal{A}$ can be rearranged into a sorted sequence so that x is placed somewhere between positions k and $k + n \cdot \ell/d + (2\ell + 3)$. (Recall that we have a potential freedom in sorting $\mathcal{A}$, since some elements may be equal.) At the same time, provided that k satisfies (4) and hence the element y has also been selected, y can be placed somewhere between positions $k - n \cdot \ell/d - (2\ell + 3)$ and k.

In addition, the above bounds imply that $y \leq x$. If $x < y$, we would have $x^{\succ} < y_{\prec}$, which contradicts $y_{\prec} \leq k \leq x^{\succ}$.

It should also be clear that, if $x_{\prec} \leq k$, then the desired element of rank k in $\mathcal{A}$ is x, since then we have $k \in \{x_{\prec}, \ldots, x^{\succ}\}$. Similarly, if $k \leq y^{\succ}$, y is of rank k, since then $k \in \{y_{\prec}, \ldots, y^{\succ}\}$. Otherwise, we have $y^{\succ} < k < x_{\prec}$. Then the wanted element of rank k can be located by selection from elements satisfying $y < a < x$. The number of such elements in $\mathcal{A}$ is bounded by

$$n'' = x_{\prec} - y^{\succ} - 1 \leq n \cdot 2 \cdot \ell/d + O(\ell). \tag{7}$$

Roughly speaking, from each segment of length d, only 2ℓ elements need be included in n''.

If (4) does not hold, i.e., for $k < n \cdot \ell/d + (2\ell+3)$, there is no element y. There are two subcases to consider – either $x_{\prec} \leq k$, and hence the wanted element of rank k is x, as in the standard case, or $k < x_{\prec}$; but then the wanted element satisfies $a < x$. Using the fact that (4) does not hold, the number of such elements can be bounded by

$$n'' = x_{\prec} - 1 \leq k + n \cdot \ell/d + (2\ell + 3) - 1 \leq n \cdot 2 \cdot \ell/d + O(\ell).$$

Thus, the upper bound on the number of remaining candidates, given by (7), does not depend on the truth of (4).

Phase 3. There are again two cases to consider, depending on the truth of the condition (4). However, these two cases are quite similar, so we give a complete description only for the "standard" case of k satisfying (4). For $k < n \cdot \ell/d + (2\ell+3)$, with no element y, we shall just point out the differences.

By comparing all elements in $\mathcal{A}$ with x and y, determine the exact values of $x_{\prec}$, $y^{\succ}$, and n''. These values depend on the number of elements greater than or equal to x, smaller than or equal to y, and on the number of remaining elements, respectively.

At the same time, partition the array $\mathcal{A}$ into two blocks, the first one with elements satisfying $y < a < x$, the second one with $a \leq y$ or $a \geq x$. (The former difference between sample and nonsample elements is no longer recognized.)

Then, if $x_{\prec} \leq k$, return x as the desired element of rank k. If $k \leq y^{\succ}$, return y. Otherwise, i.e., if $y^{\succ} < k < x_{\prec}$, find an element of rank $k'' = k - y^{\succ}$ in the block of n'' elements satisfying $y < a < x$, using any linear-time in-place selection algorithm. Return this element as the required element of rank k in the array $\mathcal{A}$.

If $k < n \cdot \ell/d + (2\ell + 3)$, there is no element y. Here we compute only the values of $x_{\prec}$ and n'', by comparing all elements with x. As in the standard case, the array $\mathcal{A}$ is partitioned into two blocks, consisting this time of elements satisfying $a < x$ and $a \geq x$, respectively. Similarly, if $x_{\prec} \leq k$, we return x as the wanted element. Otherwise, we find an element of rank $k'' = k$, among the n'' elements satisfying $a < x$.

To establish the computational cost of this phase, some implementation details must be given.

Here the configuration of the array $\mathcal{A}$ is of the form $KeED$, where K represents the block of elements satisfying $y < a < x$, kept for further inspection, e is the hole, E the block of elements not yet examined, and D the block of elements that have been discarded, with $a \leq y$ or $a \geq x$. Initially, K is empty, D contains one element, namely y, and x is put aside, to create the hole e. Such initiation requires four moves, with no comparisons. The initial values of counters are set to $x_{\prec} = n$, $y^{\succ} = 1$, and $n'' = n - 2$.

Let a denote the rightmost element of E. This element is first compared with x and, if $a < x$, also with y. Then all necessary indices and counters are updated: If $a \geq x$ or $a \leq y$, the boundary between E and D is shifted one position to the left. If $y < a < x$, the element a is moved to the hole e and, after that, the leftmost element of E is moved to the place just released. This is repeated until E becomes empty. Then the element x is returned to the hole e in $\mathcal{A}$.

If $k < n \cdot \ell/d + (2\ell + 3)$ and there is no element y, the block K contains the elements satisfying $a < x$, while D contains the elements with $a \geq x$. Because of the missing element y, the block D is initially empty. Also the initial values of counters are slightly different, namely, $x_{\prec} = n$ and $n'' = n - 1$. The element moves are organized in the same way as in the standard case, however, the elements need not be compared with y.

Summing up, there are $n - 2$ comparisons of a's with x (or $n - 1$, because of the missing y), but only $x_{\prec} - 1 < k + n \cdot \ell/d + O(\ell)$ comparisons with y (if any). The number of moves is bounded by $2n'' + 5$.

Finally, if the wanted element does not coincide with x or y, an element of rank k'' is selected among the n'' elements in K, which requires $c_{\mathrm{c}} \cdot n'' + o(n'')$ comparisons and $c_{\mathrm{m}} \cdot n'' + o(n'')$ moves.

Thus, using (7), the number of comparisons in the third phase is

$$C_3 \leq n + x_{\prec} + c_{\mathrm{c}} \cdot n'' + o(n'') < n + k + n \cdot (2c_{\mathrm{c}} + 1) \cdot \ell/d + o(n).$$

By (7), the number of moves is bounded by

$$M_3 \leq 2n'' + 5 + c_{\mathrm{m}} \cdot n'' + o(n'') < n \cdot 2 \cdot (c_{\mathrm{m}} + 2) \cdot \ell/d + o(n).$$

By summing all computational costs, we get the following bounds for the total number of comparisons and moves, respectively:

$$\begin{array}{rcl} C(n) & \leq & n \cdot [f(d)/d + 1] + k + n \cdot [2c_{\mathrm{c}} \cdot q/d + (2c_{\mathrm{c}} + 1) \cdot \ell/d] + o(n), \\ M(n) & \leq & n \cdot 2 \cdot [(c_{\mathrm{m}} + 1) \cdot q/d + (c_{\mathrm{m}} + 2) \cdot \ell/d] + o(n), \end{array}$$

where $f(d)/d \leq \log_2 d - 1.328 + o(1)$, by (2), and $k \leq \lceil n/2 \rceil$, by (1).

The number of index variables is dominated by the Ford-Johnson's sorting algorithm [6] (see also [8]), used to obtain a permutation of d elements in a segment. This algorithm needs a recursion stack of $O(\log d)$ nested levels, and a constant number of array pointers per each element and each level. (We leave details to the reader.) This gives $O(d \cdot \log d) \leq O(q \cdot \ell \cdot \log(q \cdot \ell))$ index variables.

Taking sufficiently large $q = \ell$, we obtain

Theorem 1. For each fixed real $\varepsilon > 0$, there is an in-place algorithm that, given the value of k and array of n elements, finds an element of rank k, using at most

$\varepsilon \cdot n$ element moves and $O(\log(1/\varepsilon) \cdot n)$ comparisons. The number of index variables is bounded by $O((1/\varepsilon)^2 \cdot \log(1/\varepsilon))$.

It should be pointed out that, by using the notation $o(n)$ and $O(1)$, we have implicitly introduced the assumption that n is "sufficiently large", above some threshold constant n_0. Moreover, in order to prove the correctness of the algorithm, we have used the assumptions that $n \geq (7\ell + 7)/(1 - 2\ell/d) = 7d \cdot (\ell + 1)/(q \cdot \ell - \ell + q)$, and that $n \geq 2 \cdot (\ell + 1)$. If $q \geq 2$, these can be replaced by a simpler assumption that $n \geq 7d$. For $q \geq 8$, we can use $n \geq d$. Shorter inputs can then be handled in a different way, using any other linear-time selection algorithm, which costs $c_c \cdot n + o(n) \leq O(d)$ comparisons. No element moves are required here, since we can use $n \leq O(d) \leq O((1/\varepsilon)^2)$ auxiliary indices pointing to the elements. Thus, Theorem 1 holds for each $n \geq 1$.

4 SOME VARIANTS

In several applications, we select an element z of a given rank k in order to *partition* the array $\mathcal{A}$ around the element z, into blocks $\mathcal{A}_<$, $\mathcal{A}_=$, and $\mathcal{A}_>$, with elements smaller than, equal to, and greater than z, respectively. Once we have found z, using the selection algorithm above, this task can clearly be performed in linear time. Since our primary objective is to minimize the number of moves, some technical details deserve an explanation.

Phase 4. Using $n - 1$ comparisons, with no moves, we first determine the exact value of $z_\prec$, which gives the space required by $\mathcal{A}_<$.

The array $\mathcal{A}$ is then arranged into five blocks $\mathcal{A}_< E_1 \mathcal{A}_= E_2 \mathcal{A}_>$, where $\mathcal{A}_<$, $\mathcal{A}_=$, and $\mathcal{A}_>$ are the blocks defined above, E_1 and E_2 represent two blocks of elements not yet processed. The elements in E_1 and E_2 are compared with z and added to $\mathcal{A}_<$, $\mathcal{A}_=$, or $\mathcal{A}_>$, depending on their relative order. Initially, $\mathcal{A}_<$, $\mathcal{A}_=$, and $\mathcal{A}_>$ are empty. The blocks $\mathcal{A}_<$ and $\mathcal{A}_=$ grow to the right, while $\mathcal{A}_>$ grows to the left. Thus, the "heads" of $\mathcal{A}_<$ and $\mathcal{A}_=$, that is, the positions where the next element should go, are one position to the right of $\mathcal{A}_<$ and $\mathcal{A}_=$, respectively, while the head of $\mathcal{A}_>$ is one position to the left of $\mathcal{A}_>$. Initially, their head positions are 1, $z_\prec$, and n, respectively.

To minimize element movement, we put *two* elements aside, one of them being z. Initially, the holes are created at the positions 1 and $z_\prec$. We maintain the invariant that exactly two of the blocks $\mathcal{A}_<$, $\mathcal{A}_=$, or $\mathcal{A}_>$ have holes in their head positions. The element a in the head position of the block without a hole is called the current element. This element is compared with z, which costs two comparisons, and moved (if necessary) to the proper head position of $\mathcal{A}_<$, $\mathcal{A}_=$, or $\mathcal{A}_>$. This requires only one move (if any). Now the destination block of the element a is without a hole. Thus, we can use the element residing in the new head position of this block as the new current element. This is repeated until one of the following conditions comes true:

i. The block E_2 has collapsed into a single hole. If this happens, the algorithm moves the element z to the hole in E_2, and the second element is put aside to the hole in E_1. Then the algorithm terminates; all elements greater than or equal to z, including z itself, are in their final destinations, occupying locations $z_{\prec}, \ldots, n$ in $\mathcal{A}$. As a consequence, all remaining elements in E_1, including the second element that was put aside, must be strictly smaller than z, and hence they may become a part of $\mathcal{A}_<$ without being compared with z.

ii. The block E_1 has become empty. Thus, all elements smaller than z have been put in their final locations $1, \ldots, z_{\prec} - 1$, and hence all remaining elements, in E_2 or put aside, must be greater than or equal to z. Moreover, the two holes are now in the head positions of $\mathcal{A}_=$ and $\mathcal{A}_>$.

    The algorithm first puts the element z to the hole in the head position of $\mathcal{A}_=$. From this moment forward, the algorithm performs a two-way partitioning, using a *single* hole in the head position of $\mathcal{A}_=$ or $\mathcal{A}_>$, with the current element always at the opposite end of E_2. That is, we keep on moving elements to the proper positions, this time to the head positions of $\mathcal{A}_=$ or $\mathcal{A}_>$, until E_2 collapses into a single hole. Then the second element put aside is moved to the hole in E_2, without being compared with z, and the algorithm terminates.

Clearly, the respective number of comparisons and moves can be bounded by

$$\begin{aligned} C_4 &\leq 3n - 5, \\ M_4 &\leq n + 3. \end{aligned}$$

Adding the above resource requirements to those of Theorem 1, we get:

Theorem 2. For each fixed real $\varepsilon > 0$, there is an in-place algorithm that, given the value of k and array $\mathcal{A}$ of n elements, divides $\mathcal{A}$ into three blocks with elements smaller than, equal to, and greater than the kth smallest element, using at most $(1 + \varepsilon) \cdot n$ element moves and $O(\log(1/\varepsilon) \cdot n)$ comparisons.

One can easily see that the number of comparisons in the fourth phase can be reduced to

$$C'_4 \leq n - 1,$$

if a *two-way* partitioning is required, e.g., partition $\mathcal{A}$ into blocks $\mathcal{A}_<$ and $\mathcal{A}_\geq$, with elements smaller than z, and greater than or equal to z, respectively. This follows from the fact that $\mathcal{A}_=$ and $\mathcal{A}_>$ are unified into a single block $\mathcal{A}_\geq$, growing from the right end to the left, so we do not need to compute the space required by $\mathcal{A}_<$, nor to distinguish between elements equal to and greater than z. In addition, only one hole is required here and hence a single extra location for putting elements aside is sufficient.

We shall now return to the selection problem, and present an upper bound for the sum of comparisons and moves. First, we shall modify the second phase of the algorithm, which gives slightly better results.

Phase 2, modified. After collecting a sample of n' elements in the first phase, select from the sample an element x of rank k_x. This needs $c_\mathrm{c} \cdot n' + o(n')$ comparisons and $c_\mathrm{m} \cdot n' + o(n')$ moves.

If k is so small that it does not satisfy the condition (4), the algorithm goes immediately to the third phase.

Otherwise, for $k \geq n \cdot \ell/d + (2\ell+3)$, we have also to find an element y of rank k_y in the sample. This is done as follows. First, using the two-way partition procedure of Phase 4, divide the sample into two blocks, with elements smaller than x, and greater than or equal to x. This needs only $n' - 1$ comparisons and $n' + 2$ moves. It should be clear that the partitioning procedure returns also a value of $x_\ll$, the lower rank of x *among the elements of the sample*.

Now there are two cases to consider. If $x_\ll \leq k_y$, then an element y of rank k_y must satisfy $y \geq x$; but we have already shown that $y \leq x$. Thus, we could use $x = y$ as the desired element of rank k_y; but then $x_\prec = y_\prec \leq k \leq x^\succ$, i.e., x is of rank k in the original sequence. Thus, if $x_\ll \leq k_y$, return x as the element of rank k, skipping the third phase of the algorithm.

Consider now $k_y < x_\ll$. Then y of rank k_y can be found by selection in the block of $x_\ll - 1 \leq k_x - 1$ sample elements smaller than x; but then, by (5), (1), and (6),

$$\begin{aligned} x_\ll - 1 &\leq k_x - 1 \leq k/(\ell+1) \leq (n/2+1)/(\ell+1) \\ &= \tfrac{1}{2} \cdot n \cdot (1 - \ell/d)/(\ell+1) \cdot d/(d-\ell) + 1/(\ell+1) < \tfrac{1}{2} \cdot (n'+2) \cdot d/(d-\ell) \\ &\quad + 1 \\ &< \tfrac{1}{2} \cdot n' \cdot d/(d-\ell) + 3. \end{aligned}$$

That is, only about a half of the sample is examined in order to find y, which requires only $c_\mathrm{c} \cdot n'/2 \cdot d/(d-\ell) + o(n')$ comparisons and $c_\mathrm{m} \cdot n'/2 \cdot d/(d-\ell) + o(n')$ moves.

Summing up, the modified version of the second phase uses $C_2' \leq n' \cdot c_\mathrm{c} \cdot [1 + 1/2 \cdot d/(d-\ell)] + n' + o(n')$ comparisons and $M_2' \leq n' \cdot c_\mathrm{m} \cdot [1 + 1/2 \cdot d/(d-\ell)] + n' + o(n')$ moves. By (3), we get

$$\begin{aligned} C_2' &\leq n \cdot [c_\mathrm{c} + 1 + c_\mathrm{c}/2 \cdot d/(d-\ell)] \cdot q/d + o(n), \\ M_2' &\leq n \cdot [c_\mathrm{m} + 1 + c_\mathrm{m}/2 \cdot d/(d-\ell)] \cdot q/d + o(n). \end{aligned}$$

By summing the respective costs over all three phases and using $k \leq n/2 + 1$, by (1), we get the total number of comparisons and moves for the modified in-place selection:

$$\begin{aligned} C'(n) &\leq n \cdot [f(d)/d + (c_\mathrm{c} + 1 + c_\mathrm{c}/2 \cdot d/(d-\ell)) \cdot q/d + 3/2 + (2c_\mathrm{c} + 1) \cdot \ell/d] \\ &\quad + o(n), \\ M'(n) &\leq n \cdot [(c_\mathrm{m} + 3 + c_\mathrm{m}/2 \cdot d/(d-\ell)) \cdot q/d + 2 \cdot (c_\mathrm{m} + 2) \cdot \ell/d] + o(n). \end{aligned}$$

Tuning up. In the modified algorithm above, we can use $c_\mathrm{c} = 6.83$ and $c_\mathrm{m} = 18.69$, i.e., the search for elements of ranks k_x, k_y, and k'' in short blocks is performed by the selection algorithm of Lai and Wood [9]. By choosing $q^{(1)} = 43$ and $\ell^{(1)} = 30$

(the reasoning for such odd choices will be explained later), which gives $d^{(1)} = 1363$ and $f(d^{(1)}) = 12271$, we get an algorithm $A^{(1)}$ with minimized $C'(n) + M'(n)$, using fewer than $11.183n + o(n)$ comparisons and $1.897n + o(n)$ moves.[1]

Note that this yields an algorithm with new constants $c_{\mathrm{c}}^{(1)} = 11.183$ and $c_{\mathrm{m}}^{(1)} = 1.897$, i.e., with $c_{\mathrm{c}}^{(1)} + c_{\mathrm{m}}^{(1)} < c_{\mathrm{c}} + c_{\mathrm{m}}$. Now we can construct another algorithm $A^{(2)}$, that uses $A^{(1)}$ to search for elements of ranks k_x, k_y, and k'' in short blocks. In order to minimize $C'(n) + M'(n)$ for $A^{(2)}$, we choose $q^{(2)} = 20$ and $\ell^{(2)} = 15$, which implies that $d^{(2)} = 335$ and $f(d^{(2)}) = 2344$. This gives a selection algorithm $A^{(2)}$ using $10.621n + o(n)$ comparisons and $0.701n + o(n)$ moves, i.e., an algorithm with constants $c_{\mathrm{c}}^{(2)} = 10.621$ and $c_{\mathrm{m}}^{(2)} = 0.701$, satisfying $c_{\mathrm{c}}^{(2)} + c_{\mathrm{m}}^{(2)} < c_{\mathrm{c}}^{(1)} + c_{\mathrm{m}}^{(1)}$.

It is obvious that this process can be iterated, which yields a sequence of selection algorithms $A^{(1)}, A^{(2)}, A^{(3)}, \ldots$ with parameters $q^{(i)}$, $\ell^{(i)}$, and $d^{(i)} = q^{(i)} \cdot \ell^{(i)} + q^{(i)} + \ell^{(i)}$, for $i = 1, 2, 3, \ldots$, using $c_{\mathrm{c}}^{(i)} \cdot n + o(n)$ comparisons and $c_{\mathrm{m}}^{(i)} \cdot n + o(n)$ moves. We are free to fix the parameters $q^{(i)}$ and $\ell^{(i)}$ quite arbitrarily; however, we shall choose them so that we minimize $c_{\mathrm{c}}^{(i)} + c_{\mathrm{m}}^{(i)}$, where $c_{\mathrm{c}}^{(i)}$ and $c_{\mathrm{m}}^{(i)}$ are defined by

$$\begin{aligned} c_{\mathrm{c}}^{(i)} &= f(d^{(i)})/d^{(i)} + (c_{\mathrm{c}}^{(i-1)} + 1 + c_{\mathrm{c}}^{(i-1)}/2 \cdot d^{(i)}/(d^{(i)} - \ell^{(i)})) \cdot q^{(i)}/d^{(i)} + 3/2 \\ &\quad + (2c_{\mathrm{c}}^{(i-1)} + 1) \cdot \ell^{(i)}/d^{(i)}, \\ c_{\mathrm{m}}^{(i)} &= (c_{\mathrm{m}}^{(i-1)} + 3 + c_{\mathrm{m}}^{(i-1)}/2 \cdot d^{(i)}/(d^{(i)} - \ell^{(i)})) \cdot q^{(i)}/d^{(i)} + 2 \cdot (c_{\mathrm{m}}^{(i-1)} + 2) \cdot \ell^{(i)}/d^{(i)}. \end{aligned} \tag{8}$$

Here $c_{\mathrm{c}}^{(0)} = c_{\mathrm{c}} = 6.83$ and $c_{\mathrm{m}}^{(0)} = c_{\mathrm{m}} = 18.69$.

It turns out that by choosing $q^{(i)} = 15$ and $\ell^{(i)} = 10$, for each $i \geq 3$, which implies that $d^{(i)} = 175$ and $f(d^{(i)}) = 1064$, we obtain the best sum of comparisons and moves. Using these values in (8) and simplifying, we get, for each $i \geq 3$, that

$$\begin{aligned} c_{\mathrm{c}}^{(i)} &= \tfrac{27}{110} \cdot c_{\mathrm{c}}^{(i-1)} + \tfrac{2703}{350}, \\ c_{\mathrm{m}}^{(i)} &= \tfrac{27}{110} \cdot c_{\mathrm{m}}^{(i-1)} + \tfrac{17}{35}. \end{aligned}$$

It can be easily verified that both $c_{\mathrm{c}}^{(0)}, c_{\mathrm{c}}^{(1)}, c_{\mathrm{c}}^{(2)}, \ldots$ and $c_{\mathrm{m}}^{(0)}, c_{\mathrm{m}}^{(1)}, c_{\mathrm{m}}^{(2)}, \ldots$ are convergent sequences. They approach their respective fixed points $c_{\mathrm{c}}^{(\infty)}$ and $c_{\mathrm{m}}^{(\infty)}$, defined by

$$\begin{aligned} c_{\mathrm{c}}^{(\infty)} &= \tfrac{27}{110} \cdot c_{\mathrm{c}}^{(\infty)} + \tfrac{2703}{350}, \\ c_{\mathrm{m}}^{(\infty)} &= \tfrac{27}{110} \cdot c_{\mathrm{m}}^{(\infty)} + \tfrac{17}{35}. \end{aligned}$$

Thus, the sequence $c_{\mathrm{c}}^{(0)}, c_{\mathrm{c}}^{(1)}, c_{\mathrm{c}}^{(2)}, \ldots$ is approaching the limit $c_{\mathrm{c}}^{(\infty)} = \frac{2703}{350}/(1 - \frac{27}{110}) = \frac{29733}{2905} < 10.236$, while the sequence $c_{\mathrm{m}}^{(0)}, c_{\mathrm{m}}^{(1)}, c_{\mathrm{m}}^{(2)}, \ldots$ the limit $c_{\mathrm{m}}^{(\infty)} = \frac{17}{35}/(1 - \frac{27}{110}) = \frac{374}{581} < 0.644$. By calculating the first few values, one can easily verify that already $c_{\mathrm{c}}^{(7)} < 10.236$ and $c_{\mathrm{m}}^{(7)} < 0.644$.

Corollary 1. There exists an in-place algorithm that, given the value of k and array of n elements, finds an element of rank k, using at most $0.644n$ element moves and $10.236n$ comparisons.

[1] All real constants presented here are rounded up at the last displayed digit.

The above corollary hides some details deserving additional explanation. First, for each $i \geq 1$, the algorithm $A^{(i)}$ uses $A^{(i-1)}$ as a subprogram for selection of some elements in short blocks only if the input size exceeds some constant n_0. By theoretical analysis given above, the constant n_0 depends on $d^{(i)}$; however, it actually does not depend on i, since we have fixed $q^{(i)} = 15$, $\ell^{(i)} = 10$, and hence $d^{(i)} = 175$, for each $i \geq 3$. Shorter inputs are handled in a different way. (For more details, see remarks below Theorem 1.)

Second, the values of $c_{\mathrm{c}}^{(\infty)}$ and $c_{\mathrm{m}}^{(\infty)}$ do not depend on characteristics $c_{\mathrm{c}}^{(0)} = c_{\mathrm{c}}$ or $c_{\mathrm{m}}^{(0)} = c_{\mathrm{m}}$ of the "base" algorithm $A^{(0)}$, used as a starting point for the sequence $A^{(1)}, A^{(2)}, A^{(3)}, \ldots$. Replacing $A^{(0)}$ by a different selection algorithm affects significantly only the first few levels. Afterwards, the pairs $[q^{(i)}, \ell^{(i)}]$ become identical with $[15, 10]$, and the computational cost differences become negligible. As a consequence, we do not gain too much if we use as a starting point $A^{(0)}$ the best known algorithm, presented by Carlsson and Sundström in [3]. On the contrary, we are free to replace the Lai and Wood's algorithm [9] by any simpler but linear-time in-place selection algorithm that, though less efficient, is easier to implement. This simplifies the implementation of any algorithm in the sequence $A^{(1)}, A^{(2)}, A^{(3)}, \ldots$.

Compared to the best known upper bound for $C(n) + M(n)$, based on the Carlsson and Sundström's algorithm with $3.75n + o(n)$ comparisons and $9n + o(n)$ moves [3], the selection algorithm of Corollary 1 saves about $1.87n$ comparisons/moves. This is not too much at the first glance; however, we save much more if the cost of moving an element exceeds the cost of a comparison. This is typical in many applications, where the relative order of an element is induced by the relative order of some *key*, forming a small part in a large record of data.

As an example, in an application where a single element occupies ten machine words, one of them being used as a key, the total running time is not characterized by $C(n) + M(n)$ but, rather, by $C(n) + 10 \cdot M(n)$. This corresponds to $93.75n$ in the Carlsson and Sundström's algorithm, but only to $16.676n$ in the algorithm presented by Corollary 1. Moreover, we can tune up the values of $q^{(i)}$ and $\ell^{(i)}$, in order to minimize the new cost criterion $c_{\mathrm{c}}^{(\infty)} + 10 \cdot c_{\mathrm{m}}^{(\infty)}$: For each $i \geq 1$, let $q^{(i)} = 43$ and $\ell^{(i)} = 30$. Using these values in (8), together with $d^{(i)} = 1363$ and $f(d^{(i)}) = 12271$, we get that $c_{\mathrm{c}}^{(i)} = \frac{7749}{84506} \cdot c_{\mathrm{c}}^{(i-1)} + \frac{28777}{2726}$ and $c_{\mathrm{m}}^{(i)} = \frac{7749}{84506} \cdot c_{\mathrm{m}}^{(i-1)} + \frac{249}{1363}$. This gives $c_{\mathrm{c}}^{(\infty)} = \frac{28777}{2726}/(1 - \frac{7749}{84506}) = \frac{892087}{76757} < 11.623$ and $c_{\mathrm{m}}^{(\infty)} = \frac{249}{1363}/(1 - \frac{7749}{84506}) = \frac{15438}{76757} < 0.202$.

Corollary 2. There exists an in-place algorithm that, given the value of k and array of n elements, finds an element of rank k, using at most $0.202n$ element moves and $11.623n$ comparisons, i.e., with $C(n) + 10 \cdot M(n) \leq 13.634n$.

More specifically, one can easily verify, by a straightforward calculation, that these bounds are obtained already by the algorithm $A^{(6)}$ in the updated sequence.

In general, we are given a value of $r > 0$, the cost ratio between a single move and single comparison. Our objective is to tune up the values of q and ℓ so that they minimize $C(n) + r \cdot M(n)$.

First, by Corollary 1, we already have an algorithm with $C(n) + r \cdot M(n) \leq \mu \cdot n$, where $\mu = 10.236 + r \cdot 0.644$, corresponding to $[q^{(i)}, \ell^{(i)}] = [q, \ell] = [15, 10]$, for each

$i \geq 1$. (For the purpose of asymptotic analysis, the differences in $[q, \ell]$ for the first few levels are not essential.)

This rules out all pairs $[q, \ell]$ with $q \geq 2^{\mu-1}$ or $\ell \geq 2^{\mu-1}$. For each such pair and each $i \geq 1$, using (8) and the general lower bound for sorting d elements [8], we get:

$$\begin{aligned} c_{\mathrm{c}}^{(i)} &\geq f(d)/d + 3/2 \geq \lceil \log_2 d! \rceil/d + 3/2 \geq (d \cdot \log_2 d - d \cdot \log_2 e)/d + 3/2 \\ &\geq \log_2 d = \log_2(q \cdot \ell + q + \ell) > \log_2(2 \cdot 2^{\mu-1}) = \mu. \end{aligned}$$

But then $c_{\mathrm{c}}^{(\infty)} = \lim_{i\to\infty} c_{\mathrm{c}}^{(i)} \geq \mu$, and hence $c_{\mathrm{c}}^{(\infty)} + r \cdot c_{\mathrm{m}}^{(\infty)} > \mu$. Thus, the pair $[q, \ell]$ does not yield a better value of $c_{\mathrm{c}}^{(\infty)} + r \cdot c_{\mathrm{m}}^{(\infty)}$ than does the pair $[15, 10]$.

As a consequence, the optimal values of q and ℓ, resulting in the smallest value of $c_{\mathrm{c}}^{(\infty)} + r \cdot c_{\mathrm{m}}^{(\infty)}$, can be found by a straightforward brute-force enumeration, examining all possible pairs $[q, \ell]$ with $q < 2^{\mu-1}$ and $\ell < 2^{\mu-1}$, which is a finite number of cases.[2]

For each examined pair $[q, \ell]$, compute $d = q \cdot \ell + q + \ell$ and $f(d)$, and use all these values in (8). This gives, after some simplification, two recurrences in the form $c_{\mathrm{c}}^{(i)} = \alpha \cdot c_{\mathrm{c}}^{(i-1)} + \beta_{\mathrm{c}}$ and $c_{\mathrm{m}}^{(i)} = \alpha \cdot c_{\mathrm{m}}^{(i-1)} + \beta_{\mathrm{m}}$, where α, β_{c}, and β_{m} denote some positive real constants, with $\alpha < 1$. Thus, the fixed points are $c_{\mathrm{c}}^{(\infty)} = \beta_{\mathrm{c}}/(1-\alpha)$ and $c_{\mathrm{m}}^{(\infty)} = \beta_{\mathrm{m}}/(1-\alpha)$, which in turn gives a positive real constant $\mu_{q,\ell} = c_{\mathrm{c}}^{(\infty)} + r \cdot c_{\mathrm{m}}^{(\infty)}$. Thus, for each $[q, \ell]$, we get a different sequence of algorithms $A^{(1)}, A^{(2)}, A^{(3)}, \ldots$, approaching the upper bound $C(n) + r \cdot M(n) = \mu_{q,\ell} \cdot n$.

After fixing the pair $[q, \ell]$ with the smallest value of $\mu_{q,\ell}$, we can calculate the exact characteristics of some first few members in the sequence $A^{(1)}, A^{(2)}, A^{(3)}, \ldots$, for this pair, until we obtain an algorithm $A^{(i_0)}$ with $c_{\mathrm{c}}^{(i_0)} + r \cdot c_{\mathrm{m}}^{(i_0)}$ "sufficiently close" to $\mu_{q,\ell} = c_{\mathrm{c}}^{(\infty)} + r \cdot c_{\mathrm{m}}^{(\infty)}$. After that, one can even try to do some fine-tuning of the first few levels, by fixing different pairs $[q^{(i)}, \ell^{(i)}]$ for different i's, so that it minimizes $c_{\mathrm{c}}^{(i)} + r \cdot c_{\mathrm{m}}^{(i)}$. This does not reduce the asymptotic characteristic $c_{\mathrm{c}}^{(\infty)} + r \cdot c_{\mathrm{m}}^{(\infty)}$, but can result in a smaller value of i_0.

5 CONCLUDING REMARKS

Theorem 1 has presented a selection algorithm using only $\varepsilon \cdot n$ element moves, where $\varepsilon > 0$ denotes an arbitrarily small, but fixed, real constant. The price we pay for reducing the number of moves is reasonably small, namely, $O(\log(1/\varepsilon) \cdot n)$ comparisons and $O((1/\varepsilon)^2 \cdot \log(1/\varepsilon))$ auxiliary index variables. That is, we do not have to remember too much information at a time. There is at use only a constant number of index variables and, since there are only $\varepsilon \cdot n$ moves available, we do not encode too much in the in-place element ordering. (This was the case of algorithms presented in [3].) We neither recompute too much information over and over again, since the total running time is linear. (This was the case of algorithms for read-only memory [11], using a superlinear number of comparisons.)

[2] This tedious work can be handed over to a simple computer program, as we actually did in the case of $r = 1$ and $r = 10$.

An unexpected, but useful, side effect feature of the sequence $A^{(1)}, A^{(2)}, A^{(3)}, \ldots$ is that, for each i_0, the algorithm $A^{(i_0)}$ takes advantage of sequential memory access to the given input array $\mathcal{A}$, if the computer uses a storage with some memory block pages stored in a faster cache memory, with at least two cache pages available. $A^{(i_0)}$ first scans the entire input array $\mathcal{A}$ from left to right, collecting a small block of n' elements at the left end of $\mathcal{A}$. This small block is also built from left to right. After processing n' elements in the small block, $A^{(i_0)}$ scans the entire input again, this time from right to left, collecting another block of n'' elements at the left end, after which it concentrates on this small block only. The access for processing two small blocks of size n' and n'' can be determined recursively. Practical experiments with several different applications (see Section 1.2 in [12]) have shown that, for a modern computer with a cache memory, the order in which memory locations are accessed has considerable influence on the running time.

The bottleneck of our algorithms is $n \cdot f(d)/d$ comparisons, spent to pick up q evenly distributed sample elements in each segment of length d. A possible improvement could replace the Ford-Johnson's sorting algorithm by an algorithm that, for a fixed q and d, selects an evenly distributed sample without sorting. A factory production of such partial orders, analogous to those used in [13] or [4, 5], would be sufficient.

So we leave as open problems the cost of selecting q evenly distributed sample elements in a segment of d elements, and the unit cost per each such segment processed, with continual input and output of elements.

Acknowledgement

The authors thank anonymous referees for pointing to the special case of a rank k that is very small.

REFERENCES

[1] Bent, S. W.—John, J. W.: Finding the Median Requires $2n$ Comparisons. In Proc. ACM Symp. Theory of Comput., pp. 213–16, 1985.

[2] Blum, M.—Floyd, R. W.—Pratt, V.—Rivest, R. L.—Tarjan, R. E.: Time Bounds for Selection. J. Comput. System Sci., Vol. 7, 1973, pp. 448–61.

[3] Carlsson, S.—Sundström, M.: Linear-Time in-Place Selection in Less Than $3n$ Comparisons. In Proc. Internat. Symp. Algorithms and Comput., Lect. Notes Comput. Sci., 1004, pp. 244–53. Springer-Verlag, 1995.

[4] Dor, D.: Selection Algorithms. Ph. D. Thesis, School Math. Sci., Tel-Aviv Univ., 1995.

[5] Dor, D.—Zwick, U.: Selecting the Median. SIAM J. Comput., Vol. 28, 1999, pp. 1722–58.

[6] Ford, L. R.—Johnson, S. M.: A Tournament Problem. Amer. Math. Monthly, Vol. 66, 1959, pp. 387–89.

[7] Hadian, A.: Optimality Properties of Various Procedures for Ranking n Different Numbers Using Only Binary Comparisons. Ph. D. Thesis, Dept. Statist., Univ. Minnesota, 1969. (Tech. Rep. 117.)

[8] Knuth, D. E.: The Art of Computer Programming, Vol. 3: Sorting and Searching. Addison-Wesley, 1973. (Second edition: 1998.)

[9] Lai, T. W.—Wood, D.: Implicit Selection. In Proc. Scand. Workshop on Algorithm Theory, Lect. Notes Comput. Sci., Vol. 318, 1988, pp. 14–23. Springer-Verlag.

[10] Munro, J. I.—Raman, V.: Sorting with Minimum Data Movement. J. Algorithms, Vol. 13, pp. 374–93, 1992.

[11] Munro, J. I.—Raman, V.: Selection from Read-Only Memory and Sorting with Minimum Data Movement. Theoret. Comput. Sci., Vol. 165, 1996, pp. 311–23.

[12] Pasanen, T.: In-Place Algorithms for Sorting Problems. Ph. D. Thesis, Dept. Comput. Sci., Univ. Turku, 1999. (TUCS Dissert. No. 15.)

[13] Schönhage, A.—Paterson, M.—Pippenger, N.: Finding the Median. J. Comput. System Sci., Vol. 13, 1976, 184–99.

Viliam Geffert was born in 1955. He finished his studies at P. J. Šafárik University, Košice, Slovakia, in 1979, and received his Ph. D. degree in computer science at Comenius University in Bratislava, in 1988. His present position is professor at the Department of Computer Science, P. J. Šafárik University, Košice. His main research interests are space bounded computations, formal languages and finite automata, and in-place sorting algorithms.

Ján Kollár (Assoc. Prof., M. Sc., Ph. D.) received his M. Sc. summa cum laude in 1978 and his Ph. D. in computing science in 1991. In 1978–1981 he was with the Institute of Electrical Machines in Košice. In 1982–1991 he was with the Institute of Computer Science at the P. J. Šafárik University in Košice. Since 1992 he has been with the Department of Computers and Informatics at the Technical University of Košice. In 1985 he spent 3 months in the Joint Institute of Nuclear Research in Dubna, Soviet Union. In 1990 he spent 2 months at the Department of Computer Science at Reading University, Great Britain. He was involved in the research projects dealing with the real-time systems, the design of (micro) programming languages, image processing and remote sensing, the dataflow systems, the implementation of programming languages. Currently he is working in the area of multi-paradigmatic languages, with respect of aspect paradigm. He is the author of $\mathcal{PFL}$ – a process funtional language.