ELSEVIER

Theoretical Computer Science 165 (1996) 311–323

Theoretical Computer Science

Selection from read-only memory and sorting with minimum data movement [1]

J. Ian Munro [a,*], Venkatesh Raman [b]

[a] *Department of Computer Science, University of Waterloo, Waterloo, Ont., Canada N2L 3G1*

[b] *Institute of Mathematical Sciences, C.I.T. Campus, Madras 600113, India*

Received July 1991; revised December 1993

Communicated by M.S. Paterson

Abstract

Selecting an element of given rank, for example the median, is a fundamental problem in data organization and the computational complexity of comparison based problems. Here, we consider the scenario in which the data resides in an array of read-only memory and hence the elements cannot be moved within the array. Under this model, we develop efficient selection algorithms using very little extra space ($o(\log n)$ extra storage cells). These include an $O(n^{1+\varepsilon})$ worst case algorithm and an $O(n \log\log n)$ average case algorithm, both using a constant number of extra storage cells or indices. Our algorithms complement the upper bounds for the time–space tradeoffs obtained by Munro and Paterson [9] and Frederickson [4] who developed algorithms for selection in the same model when $\Omega((\log n)^2)$ extra storage cells are available.

We apply our selection algorithms to obtain sorting algorithms that perform the minimum number of data moves on any given array. We also derive upper bounds for time–space tradeoffs for sorting with minimum data movement.

1. Introduction

Finding an element of a given rank in an unordered array is a fundamental problem in computer science. In 1961, Hoare [5] presented an algorithm that performs $\Theta(n)$ comparisons on the average. In their seminal 1973 paper, Blum et al. [1] showed that this bound can be attained in the worst case as well. At essentially the same time, Floyd and Rivest [3] obtained a very simple method with excellent average case behavior. The best worst-case algorithm for finding the median remained the (1976) $3n + o(n)$ comparisons algorithm of Schönhage et al. [14] until recently, when Dor and Zwick [2] gave a $2.95n + o(n)$ algorithm. All these algorithms perform $\Theta(n)$ data moves, or exchanges. Here we ask how efficiently we can find the median (or element of

[1] This work was supported by NSERC of Canada under Grant No. A-8237 and ITRC of Ontario. A preliminary version of this paper appeared in [11].

* Corresponding author. E-mail: imunro@uwaterloo.ca and vraman@imsc.ernet.in.

SSDI 0304-3975(95)00225-1

any rank) without performing swaps or exchanges among the data. Such a technique is essential, for example, when the data resides in read-only memory and we are unwilling to use $\Theta(n)$ extra storage space. Our specific interest is in the mathematical question of how quickly we can find the median in read-only memory, using a constant number of extra storage cells. We hope that the answer to this question will shed light on a time–space tradeoff for selection. As the basic unit of extra storage, we use a $\lg n$ bit cell. Using such cells as indices sidesteps our need for any more than one extra location to store data items temporarily. Our measure of time is the number of comparisons, and of space, the number of indices used by an algorithm.

Munro and Paterson [9] and Frederickson [4] considered this selection problem and developed algorithms for selection on machines with read-only registers, assuming $\Omega((\log n)^2)$ extra storage cells are available. In [9] access to the input is sequential and time is measured by the number of passes. The complexity of these and other algorithms in terms of comparisons was noted in [4]. In the next section, we develop efficient algorithms when only $O((\log n/\log\log n)^{1/2})$ storage cells are available. Our algorithms improve the only known upper bound of $\Theta(n^2/s)$ comparisons when $s = o((\log n)^2)$ [9] storage cells are available. If s is a fixed constant, for example, our algorithm performs $O(n^{1+\varepsilon})$ comparisons for any $\varepsilon \in (0,1)$, while the Munro and Paterson method takes $\Theta(n^2)$. We develop an algorithm using s extra storage cells that performs $O(n\log(\log n/\log s))$ comparisons on the average, assuming all input permutations are equally likely. This improves the previous upper bound of $\Theta(n\log n/\log s)$ [9].

In Section 3, we use our selection methods to obtain sorting algorithms that perform the exact minimum number of data moves on any given array. To do so, we first derive a lower bound on the number of data moves required. We also derive upper bounds for time–space tradeoffs on sorting with minimal data movement. Throughout the paper, we assume that all the values are distinct. To simplify notation, we ignore various necessary integer round-ups or round-downs; they do not affect the asymptotic analysis.

2. Selection in read-only memory

If n extra storage cells are available, we can obviously use them as references to the data and apply any $O(n)$ selection algorithm. On the other hand, if only a constant number of extra storage cells are available, a $\Theta(n^2)$ algorithm is trivial. Beating it is not.

Our selection techniques use several passes or *phases*. In each phase we have a pair of elements called the *left* and *right filters*, which are of known ranks in the entire set. They are, respectively, too small and too large to be the desired results. The *candidates* to be the answer are those falling between the filters in value. Let r denote the number of candidates and k, the rank among the candidates of the element we are to find (i.e. we want the kth smallest). Then our algorithms, like [4, 9], take the following

approach:

Basic Algorithm. 1. Choose an element (u) from the candidates (details specified later).

2. Compare all candidates with u counting the number less than u.

3. Based on k and the number of candidates less than u, the filters and the desired rank are updated. More specifically, if the number of candidates less than u is $k-1$, then u is the element of desired rank. Otherwise, if k is smaller than the number of candidates less than u, then u becomes the right filter (i.e. recurse on those candidates less than u for the element of desired rank). Otherwise, u becomes the left filter and k is updated (i.e. recurse on those above u and find an appropriate ranked element).

The number of phases performed depends on the number of candidates eliminated at each phase in Step 3, which in turn depends on the choice of u in Step 1. Furthermore, Step 2 performs $\Theta(n)$ comparisons, since each element is compared with the filters and the candidates are compared with u. Step 3 involves no comparisons, so the key step in our algorithms is the proper choice of u in Step 1, and so, the method of finding u. We begin by a simple randomized $O(n \log n)$ average case algorithm to prove the following theorem.

Theorem 1. *The kth smallest element in a read-only array of n elements can be found using $O(1)$ indices and $O(n \log n)$ comparisons on the average, independent of the order of the given data.*

Proof. The key idea is to choose u to be a random candidate at Step 1 of the basic algorithm in each phase. More specifically, suppose at the beginning of a phase, r of the n elements are candidates. Choose a random integer p between 1 and r, and let u (in Step 1) be the value of the pth candidate from the left in the array. As all possible choices for p are equally likely, the expected number of comparisons required to find the kth smallest element among r candidates in an array of n elements, $C(r,n)$, satisfies the following recurrence. We start with the set maximum and minimum as filters, and so $r = n-2$.

$$C(r,n) \leqslant O(n) + \frac{1}{r}\left(\sum_{j=1}^{k-1} C(r-j,n) + \sum_{j=k+1}^{r} C(j-1,n)\right),$$

$$C(1,n) = 0.$$

This recurrence is analogous to the one that occurs in the average case analysis of Quicksort, and so, $C(r,n) = O(n \log r)$. As the randomization is built into the algorithm, the $O(n \log r)$ bound is independent of the order of the input, and the theorem follows. □

If all permutations of the input are equally likely, we can reduce the number of phases in the selection algorithm to $O(\log \log n)$ by choosing the value of u more carefully, in a manner similar to Floyd and Rivest [3]. The choice of u is based on a

sampling strategy that requires an extra $O(n)$ comparisons on the average in Step 1, but reduces the expected number of candidates remaining after each phase to $O(r^{3/4})$. The method permits a small degree of latitude to benefit from extra storage, and so we state:

Theorem 2. *The kth smallest element in a read-only array of n elements can be found using* $O(s)$ *indices and* $O(n \log(\log n/\log s))$ *comparisons on the average, if all the permutations of the given input are equally likely.*

The most interesting aspect is the method of choosing u. We develop a family of selection algorithms using $O(n^{1+\varepsilon})$ comparisons in the worst case, and apply one of these to the sampling strategy of the average case method. We first describe these worst case methods, before returning to the proof of Theorem 2. A_s $(s \geqslant 1)$ will denote our worst case selection method. On n elements of which r are candidates, it has $O(2^s s! n r^{1/s})$ runtime and uses $O(s)$ indices. A_s will make use of A_{s-1}. A_1 denotes the $O(nr)$ method resulting from an arbitrary choice of u in the basic algorithm. We first describe the $O(n^{3/2})$ algorithm, A_2.

Theorem 3. *The kth smallest element in a read-only array of n elements can be found using* $O(1)$ *indices and* $O(n^{3/2})$ *comparisons in the worst case.*

Proof. Let r be the number of candidates at the beginning of a phase. Initially, $r = n - 2$, and the minimum and the maximum are the filters. In each phase we first identify a block, B, of at most $m = n/\sqrt{r}$ elements containing $c = \sqrt{r}$ candidates. This is accomplished as follows. View the array as approximately $\sqrt{r}$ sections of approximately $n/\sqrt{r}$ elements each. Since there is a total of r candidates in the array, at least one of these sections must contain at least $c = \sqrt{r}$ candidates. By scanning the array, we find B, the prefix of the leftmost section containing at least c candidates ending with its cth candidate. Next we find the median of the candidates in B using algorithm A_1. Since only $\sqrt{r}$ of at most $n/\sqrt{r}$ elements are candidates, this step takes $O(n)$ time. This median is the element chosen as u in Step 1 of the basic algorithm. All candidates are then compared with u. Depending on the number of candidates less than u and the value of the desired rank, the filters are updated as in Steps 2 and 3 of the basic algorithm. This process is repeated on the remaining candidates until the desired element is found.

Approximately $\sqrt{r}/2$ candidates are guaranteed to be less than and at least that many are certain to be more than u. Hence, depending on the partition in which the kth smallest item remains, at least $\sqrt{r}/2$ candidates are eliminated. Thus in each phase of the algorithm, which takes $O(n)$ comparisons, we discard at least $\sqrt{r}/2$ candidates. Therefore, if $P(r)$ is the number of phases of the algorithm in the worst case, $P(r)$ satisfies the following recurrence:

$$P(1) = 0, \quad \text{and} \quad P(r) \leqslant 1 + P(r - \sqrt{r}/2) \text{ otherwise.}$$

It follows that $P(r) \leqslant 4\sqrt{r}$. As each phase takes $O(n)$ comparisons, and $r = n - 2$ initially, the entire algorithm takes $O(n^{3/2})$ comparisons. □

To generalize this algorithm to $s > 1$, we use A_{s-1} rather than A_1 to find the sample median in each phase. This permits us to use a larger sample of candidates, and so discard more candidates in each phase.

Theorem 4. *The kth smallest element in a read-only array of n elements can be found using* $O(2^s s! n^{1+1/s})$ *comparisons and* $O(s)$ *indices in the worst case, where* $s \geqslant 1$ *is any fixed parameter.*

Proof. We describe the algorithm A_s, where $s > 1$ is a fixed parameter. As before, r is the number of candidates at the beginning of a phase. Initially $r = n - 2$, and the minimum and the maximum values are the filters. Again, we first identify a block, B, of at most $m = nr^{-1/s}$ elements containing $c = r^{1-1/s}$ candidates. This is accomplished by dividing the list into roughly $r^{1/s}$ sections of about $nr^{-1/s}$ elements each. At least one of these must contain at least $c = r^{1-1/s}$ candidates. B is the prefix containing c candidates of the leftmost such section. Next we find the median of the candidates in B by algorithm A_{s-1}. This median is the element, u, chosen in Step 1 of the basic algorithm. All candidates are then compared with u, and depending on the number of candidates less than u and the value of the desired rank, the filters are updated as in Steps 2 and 3 of our basic algorithm. The phases are then repeated until the desired element is found. At least $r^{1-1/s}/2$ candidates must be less than, and approximately that many are certain to be greater than u. So after each phase, at least $r^{1-1/s}/2$ candidates are eliminated.

In Step 1 of any phase, at most $2n$ comparisons are used to identify B. In Step 3, at most $2n$ comparisons are made to identify the candidates, and the r candidates are compared with u, taking another $r - 1$ comparisons. Thus fewer than $5n$ comparisons are made in the non-recursive steps of a phase. The number of comparisons spent in the worst case, $C(r,n,s)$, satisfies the following recurrence for $r \leqslant n$.

$$C(r,n,s) \leqslant 5n + C(r^{1-1/s}, nr^{-1/s}, s-1) + C(r - r^{1-1/s}/2, n, s),$$

$$C(1,n,s) \leqslant 2n.$$

We show by simultaneous induction on r, s and n, that $C(r,n,s) \leqslant 5ng(s)r^{1/s}$, where $g(s)$ denotes $2^{s+1}s! - 2$. The basis cases are easily verified. Then

$$\begin{aligned} C(r,n,s) &\leqslant 5n + 5g(s-1)nr^{-1/s}(r^{1-1/s})^{1/(s-1)} + C(r - r^{1-1/s}/2, n, s) \\ &\leqslant 5n + 5g(s-1)n + 5g(s)n(r - r^{1-1/s}/2)^{1/s} \\ &= 5n\left(1 + g(s-1) + g(s)r^{1/s}\left(1 - \frac{1}{2r^{1/s}}\right)^{1/s}\right) \\ &= 5n\left[1 + g(s-1) + g(s)r^{1/s}\left(1 - \frac{1}{s2r^{1/s}} - \frac{1(s-1)}{2(s2r^{1/s})^2} - \frac{(s-1)(2s-1)}{3!(s2r^{1/s})^3} - \cdots\right)\right] \end{aligned}$$

$$\leqslant 5n\left(1 + g(s-1) + g(s)r^{1/s} - \frac{g(s)}{2s}\right)$$
$$\leqslant 5ng(s)r^{1/s}$$

since

$$1 + g(s-1) - g(s)/2s = 1 + (2^s(s-1)! - 2) - (2^s(s-1)! - 1/s)$$
$$= (1/s) - 1 \leqslant 0.$$

The storage, $S(r,n,s)$, used by the algorithm is the maximum number of indices used in any phase. As r decreases after each phase, the maximum storage is used in the first phase. Five storage cells suffice for indices and counters in the main (non-recursive) steps of the algorithm. Then $S(r,n,s)$ satisfies

$$S(r,n,s) \leqslant 5 + S(r^{1-1/s}, nr^{-1/s}, s-1) \quad \text{and} \quad S(r,n,1) \leqslant 5.$$

It follows that $S(r,n,s) \leqslant 5s$. □

Setting $s = 1/\varepsilon$, $\varepsilon \in (0,1)$, in Theorem 4, we have:

Corollary 1. *The kth smallest element in a read-only array of size n can be found using* $O(n^{1+\varepsilon})$ *comparisons and* $O(1)$ *indices in the worst case, where* ε *is any arbitrary, but fixed, positive constant less than* 1.

The value of s that minimizes the number of comparisons made by the algorithm of Theorem 4 turns out to be at most $(2\lg n/\lg\lg n)^{1/2}$. Thus by setting $s = (2\lg n/\lg\lg n)^{1/2}$ in Theorem 4, and simplifying, we obtain the following corollary.

Corollary 2. *The kth smallest element in a read-only array of size n can be found using* $O(n^{1+O(\sqrt{\log\log n/\log n})})$ *comparisons and* $O((\log n/\log\log n)^{1/2})$ *indices in the worst case.*

Notice that this number of comparisons is $n\log^{\omega(1)} n$ but $n^{1+o(1)}$.

Proof of Theorem 2. We use the Floyd–Rivest approach to develop the following $O(n\log(\log n/\log s))$ average case method using $O(s)$ storage cells assuming the input is given in random order.

Algorithm. As before, at the beginning of a phase r candidates remain, of which we are to find the kth smallest. Consider the *first* $c = r^{3/4}$ candidates, and let m denote the position of the last of these. Select among these candidates, the items u and v of ranks $\max(kc/r - \sqrt{r},\ 1)$ and $\min(kc/r + \sqrt{r},\ m)$ respectively, using the algorithm A_3. This takes $O(mc^{1/3})$ comparisons.

We make a slight modification to Step 2 of our basic algorithm and compare all candidates with u and v, counting the number less than u, greater than v, and between u and v. Depending on the number of candidates in each interval and k, the filters are updated. We repeat the phase to select the desired element *only* if u and v form the new filters (i.e. the desired element falls between u and v) *and* the number of candidates remaining is greater than s and less than $3r^{3/4}$. If at most s candidates remain, we apply any standard linear time selection algorithm using our $O(s)$ indices. If, on the other hand, u and v fail to be filters, or there are too many candidates, we revert to A_4 to complete the selection.

Analysis. Let m be, as above, the random variable denoting the number of items to be examined to select the first $c = r^{3/4}$ candidates. Then, as the given sequence of elements is assumed to be in random order, it follows that

$$\text{Probability}(m = j) = \frac{\binom{r}{c}\binom{j-1}{c-1}c!\binom{n-r}{j-c}(j-c)!(n-j)!}{n!}$$

from which it follows (see [6, p. 85]) that the expected value of m,

$$E(m) = \frac{\binom{r}{c}c!}{n!}\sum_{j=c}^{n-r+c}\binom{j-1}{c-1}\binom{n-r}{j-c}(j-c)!(n-j)!j$$

$$= \frac{c(n+1)}{r+1}$$

which is approximately $nr^{-1/4}$ as $c = r^{3/4}$. Hence the expected number of comparisons used to select u and v in Step 1 is $O(n)$, as $n/r^{-1/4}(r^{3/4})^{1/3} = n$.

As all permutations are equally likely, the sample chosen in Step 1 (the first s candidates) is random. It follows from the analysis of the Floyd–Rivest algorithm that with probability $1 - O(1/r^{1/4})$, the new filters are u and v and the number of remaining candidates is at most $3r^{3/4}$. Our sample size is more conservative than that of Floyd and Rivest. So one can verify the result using Chebychev's inequality, since the expected number of candidates between u and v is roughly $2r^{3/4}$ and the variance of this number is at most $r^{1+1/4}$. See [13] for details.

If the number of candidates remaining is more than $3r^{3/4}$, or if u and v are not proper filter values, then we revert to using algorithm A_4, which performs $O(nr^{1/4})$ comparisons. The probability of this event is $O(r^{-1/4})$, and so, the expected number of comparisons required is $O(n)$.

Thus the expected number of comparisons made in each phase is $O(n)$. Furthermore, as we enter the next phase only when the number of remaining candidates is at least $s + 1$ and at most $3r^{3/4}$, the number of phases is $O(\log\log n - \log\log s)$ in the worst case. Thus we obtain an $O(n\log(\log n/\log s))$ average case algorithm for selection, using $O(s)$ storage cells. □

3. Sorting with minimum data movement

While the selection algorithms developed in the last section are interesting in their own right, they can also be applied to sorting. In [12] they are used to reduce the number of comparisons necessary for stable in-place sorting with a linear number of data moves. Here we will use them to sort using the absolute minimum number of data moves.

All well-known $O(n \log n)$ sorting algorithms perform $\Theta(n \log n)$ data or index moves as well. Clearly, n moves are required to sort a list of n elements in the worst case. Recently, the authors [10, 13] developed an algorithm that sorts using $O(n \log n)$ comparisons on the average, and $O(1)$ extra space and $O(n)$ data moves in the worst case. For the worst case, however, the best known result is a generalization of Heapsort that makes $O(n)$ moves using $O(1)$ extra indices and $O(n^{1+\varepsilon})$ comparisons.

To better understand the role of data moves in sorting, we first restrict the number of data moves to the exact *minimum* for the particular input, rather than the "generic $O(n)$". We derive the minimum number of data moves required to sort any given list, and then apply our selection algorithms of Section 2 to obtain sorting algorithms minimizing data moves.

We make the standard assumption that each input value is atomic, i.e. individual bits cannot be accessed. By a *data move*, we mean copying a record from one location to another, perhaps temporary, location. A simple exchange or swap involves three data moves. We do not count any implicit move required to make a comparison or the cost of index manipulation. We also require that the input be rearranged in the given array.

3.1. A lower bound on data moves

The sorting process consists of discovering a permutation using comparisons and applying it to the input. Consider the cycles of the permutation required to sort a list of n items. Elements in trivial cycles (those of length 1) are already in their final locations and need not be moved. Every element in a non-trivial cycle must be moved at least once. Furthermore, at least one element of each non-trivial cycle must be copied to a location that is not its final destination. That is, for every non-trivial cycle, an extra data move is required. The following lemma follows from these observations.

Lemma 1. *Let x and y denote the number of trivial and non-trivial cycles respectively, in the permutation required to sort a list of n items. Then $n - x + y$ data moves are necessary to sort the list.*

Because the number of non-trivial cycles in a permutation is at most $n/2$, it follows that this minimum is at most $\lfloor 3n/2 \rfloor$. There are two well known sorting algorithms that perform $O(n)$ data moves, albeit performing $\Theta(n^2)$ comparisons: Selection Sort [7], and Permutation Sort which follows from Knuth's [8] in-place permutation technique. Selection Sort finds the element that should go into each location, for $i = 1$ to n, by

finding the minimum of the unplaced values. It performs a swap, and hence three data moves, to place each item (except the last) of a non-trivial cycle, in its final destination. Thus it performs $3n-3x-3y$ data moves. Permutation Sort traces the cycle structure of the permutation by finding the destination of each element in turn. It takes $2n-2x-y$ data moves. The number of data moves performed by both the algorithms is within a constant factor of the optimum. Both perform the exact minimum number of moves if and only if each non-trivial cycle is a transposition (a cycle of length 2). In this rather special case we can improve the $\Theta(n^2)$ time.

Theorem 5. *If all the non-trivial cycles of a list are transpositions, then it can be sorted using* $O(1)$ *indices and the minimum number of moves in* $\Theta(n \log n)$ *comparisons in the worst case. Furthermore, this number of comparisons is necessary on average.*

Proof. The algorithm proceeds by properly positioning the median, quartiles, octiles, etc. In stage q $(1 \leqslant q \leqslant \lg n)$ it interchanges the elements in positions $(2p+1)n/2^q$ for $p = 0-2^{q-1}-1$ with those of the corresponding ranks.

At each stage, we view the array as being partitioned into blocks whose end points are the equally spaced elements that have already been properly placed. Consider the placement of element a, currently in the middle of block B_a. By binary search on the block ends, we determine block B_b which contains the element b with which a is to be exchanged. If a is the element of rank i among those in B_a that will ultimately move to B_b, then b is the ith element, as they are presently ordered, of those in B_b that will go to B_a. Linear scans of the two blocks suffice to find b, before exchanging the values. We observe that there are no added complications when $B_a = B_b$, including the degenerate case when $a = b$. Each stage requires $O(n)$ comparisons resulting in the total runtime of $O(n \log n)$. The lower bound follows from the observation that the logarithm of the number of transposition permutations is $\Theta(n \log n)$. □

3.2. Sorting by chasing cycles backward

The minimum number of data moves in sorting any array can be achieved by following the observations that led to the lower bound. For each non-trivial cycle, the first (the leftmost) element is copied into a temporary location. Then the element whose final destination is the original location of the first element is found and placed in its destination. Continuing in this fashion, we find, for each location emptied by a move, the element whose destination is that location and place it there until the temporary location is emptied. Then we continue on to the next cycle. Thus every element of each cycle is relocated backwards to its final destination. This approach requires a technique to identify the element whose final destination is a given location, without any data movement. This is trivial in $O(n^2)$ time, and leads to an $O(n^3)$ sorting algorithm. By applying the selection algorithms we developed in Section 2, we do better. The following corollary follows from Theorem 1.

Corollary 3. *An array of n items can be sorted using* $O(1)$ *indices, an expected* $O(n^2 \log n)$ *comparisons, and the minimum number of data moves.*

The assumption that all possible permutations are equally likely makes the analysis of a sorting algorithm based on our $O(n \log\log n)$ selection algorithm difficult. After several applications of the selection algorithm to sorting, the list order is no longer random. It may be that the knowledge gained about the order of the input in performing several selections can be used to make further selections faster, but such a proof is not obvious. Hence obtaining an $O(n^2 \log\log n)$ average case sorting algorithm using constant extra space and the minimum number of data moves remains open. For the worst case, however, we apply the selection algorithm of Theorem 4 n times to obtain:

Corollary 4. *An array of n items can be sorted using* $O(2^s s! n^{2+1/s})$ *comparisons,* $O(s)$ *indices and minimal data movement, where $s \geqslant 1$ is any fixed parameter.*

Similar bounds follow from Corollaries 1 and 2.

3.3. Sorting by chasing cycles forward in groups

As noted in Section 2, given a location, it seems difficult ($O(n^{1+\varepsilon})$ time) to find the element that belongs in that location using $O(1)$ indices and no data moves. However, given an item, it is easy ($n-1$ comparisons) to find its destination. Thus a Permutation Sort type approach (which chases each cycle forward, pushing out all elements but one in each cycle) performs $O(n^2)$ comparisons, though it requires two moves for all but one element of each non-trivial cycle. If we think of the elements of each non-trivial cycle as being in blocks of two, we can prove the following theorem.

Theorem 6. *An array of n items can be sorted using* $O(n^2)$ *comparisons,* $O(1)$ *indices, and* $\lfloor 3n/2 \rfloor$ *data moves in the worst case.*

Proof. We place elements of a non-trivial cycle as follows. Let x be an element of a non-trivial cycle. Find the destination $d(x)$ of x, and let y be the element in that location. Next find the destination $d(y)$ of y. Then copy the element at $d(y)$ into a temporary location (t), move y to $d(y)$, and x into $d(x)$. Thus for every pair of elements in the cycle we perform 3 moves. The placement continues with the element in t and the next two. A second temporary location is implied here, but can be avoided. If the cycle is of odd length, the final element is simply moved to the original location of x, and hence, the $\lfloor 3n/2 \rfloor$ bound follows. □

Although $\lfloor 3n/2 \rfloor$ moves may not be necessary to sort the given input, this theorem does give a quadratic algorithm that uses the minimum number of moves required on some input. If we have longer cycles we can clearly adapt the method to make fewer moves at the cost of more indices. In lieu of more indices, we could recompute some

of their values as required. We focus on these time–space tradeoffs, starting with a method that uses $O(\log n)$ indices.

We scan the array from left to right dealing with each non-trivial cycle, placing the elements of that cycle into their proper locations. The *basic step* is to compute the location of the $(k+1)$-th element in the cycle from the location of the kth, each such step takes $n-1$ comparisons. We describe how to place the elements of a non-trivial cycle of length l, using $\lg l$ indices.

First we find the length of the cycle in $O(nl)$ time. This value will be known in the recursive calls. The first element of the cycle is put into a temporary location. Next compute the $(l/2+1)$-th position of the cycle using $l/2$ basic steps. Starting from that position, recursively find and place the elements in the next $l/2$ cycle positions into their final locations. Then recursively recompute the first $l/2$ positions of the cycle and place the elements in those positions into their final locations.

A recurrence for the number of basic steps (each taking $n-1$ comparisons) necessary to coordinate the shifting of a cycle of length l is

$$T(l) = l/2 + 2T(l/2).$$

This gives a $\Theta(l \log l)$ solution. A similar recurrence for the number of indices necessary to place a cycle of length l is

$$S(l) = 2 + S(l/2)$$

which gives a $\Theta(\log l)$ solution. Since the sum of the lengths of the non-trivial cycles can be, and very likely is, $\Theta(n)$, this leads to a $\Theta(n^2 \log n)$ comparison method.

Theorem 7. *An array of n items can be sorted using $O(n^2 \log n)$ comparisons, $O(\log n)$ indices and the minimum number of data moves in the worst case.*

In contrast with this $O(n^2 \log n)$ algorithm for sorting with minimal data movement using $O(\log n)$ indices, an $O(n \log n)$ algorithm using $O(\log n)$ indices for selection from read-only memory remains unknown.

If we have $\Omega(\log n)$ indices available, we can adapt this method to require fewer recomputations. That is, with $\Theta(cn^{1/c})$ indices (for some value $c \geqslant 1$), we first compute the positions $il^{1-1/c}$, for $i = 1$–$l^{1/c}$, of the cycle of length l. Then we place each block of $l^{1-1/c}$ elements (starting from the last block) recursively. One can verify that this can be done using $\Theta(cl)$ basic steps and $\Theta(cl^{1/c})$ indices. As each basic step takes $n-1$ comparisons and the sum of the lengths of the non-trivial cycles could be n, we have the following theorem.

Theorem 8. *An array of n items can be sorted using $O(cn^2)$ comparisons, $O(cn^{1/c})$ indices and the minimum number of data moves in the worst case, where c is any value such that $1 \leqslant c \leqslant \lg n$.*

4. Conclusions

We have investigated the problem of selection from read-only memory, using a very small number of indices. Our results include $O(n^{1+\varepsilon})$ worst case and $O(n \log\log n)$ average case algorithms that use a constant number of indices. Our algorithms establish upper bounds for a time–space tradeoff for selection complementing the tradeoffs of Frederickson [4]. There is still a gap in finding efficient selection algorithms when the extra space s satisfies $\omega(\sqrt{\log n/\log\log n}) < s < o((\log n)^2)$. The gap in the time–space product tradeoff is particularly large around $s = o(\log n)^2$, as Frederickson's algorithms perform $O(n \lg^* s + (n \log n/\log s))$ comparisons when $s = \Omega(\log^2 n)$. It would be interesting to see whether our upper bounds for time–space tradeoffs are optimal.

Following two different approaches, we investigated sorting with minimum, or near minimum, data movement. We feel the methods developed establish interesting upper bounds for time–space tradeoffs for sorting with the optimum number of data moves. As can be observed, the restriction on data movement and space increases significantly the time complexity of the algorithms for both selection and sorting. It would be an interesting exercise to develop a computational model that captures data moves along with comparisons and space and leads to the proof of lower bounds.

Acknowledgement

The authors thank the anonymous referee for the comments that improved the readability of the paper.

References

[1] M. Blum, R.W. Floyd, V.R. Pratt, R.L. Rivest and R.E. Tarjan, Time bounds for selection, *J. Comput. System Sci.* **7** (1973) 448–461.

[2] D. Dor and U. Zwick, Selecting the median, in: *Proc. 6th ACM–SIAM Symp. on Discrete Algorithms* (1995) 28–37.

[3] R.W. Floyd and R. Rivest, Expected time bounds for selection, *Comm. ACM* **18(3)** (1975) 165–172.

[4] G.N. Frederickson, Upper bounds for time–space trade-offs in sorting and selection, *J. Comput. System Sci.* **34** (1987) 19–26.

[5] C.A.R. Hoare, Algorithms 64 PARTITION and Algorithm 65 FIND, *Comm. ACM* **4** (1961) 321.

[6] N.L. Johnson and S. Kotz, *Urn Models And Their Application, An Approach to Modern Discrete Probability Theory* (Wiley, New York, 1977).

[7] D.E. Knuth, *The Art of Computer Programming. Volume III: Sorting and Searching* (Addison-Wesley, Reading, MA, 1973).

[8] D.E. Knuth, Mathematical analysis of algorithms, in: C.V. Frieman, ed., *Proc. I.F.I.P. Congress* (North-Holland, Amsterdam, 1972) 19–27.

[9] J.I. Munro and M.S. Paterson, Selection and sorting with limited storage, *Theoret. Comput. Sci.* **12** (1980) 315–323.

[10] J.I. Munro and V. Raman, Sorting with minimum data movement, *J. Algorithms* **13** (1992) 374–393.

[11] J.I. Munro and V. Raman, Selection from read-only memory and sorting with optimum data movement, in: *Proc. 12th FST & TCS Conf.*, Lecture Notes in Computer Science, Vol. 652 (Springer, Berlin, 1992) 380–391.

[12] I. Munro and V. Raman, Fast stable in-place sorting with $O(n)$ data moves, *Algorithmica*, to appear.
[13] V. Raman, Sorting in-place with minimum data movement, Ph.D. Thesis, Dept. of Computer Sci., Univ. Waterloo, CS-91-12, 1991.
[14] A. Schonhage, M. Paterson and N. Pippenger, Finding the median, *J. Comput. System Sci.* 13 (1976) 184–199