Selection from Read-Only Memory with Limited Workspace*

Amr Elmasry[1], Daniel Dahl Juhl[2], Jyrki Katajainen[2], and Srinivasa Rao Satti[3,**]

[1] Department of Computer Engineering and Systems, Alexandria University
Alexandria 21544, Egypt
[2] Department of Computer Science, University of Copenhagen
Universitetsparken 5, 2100 Copenhagen East, Denmark
[3] School of Computer Science and Engineering, Seoul National University
599 Gwanakro, Gwanak-Gu, Seoul 151-744, Korea

Abstract. In the classic selection problem the task is to find the kth smallest of N elements. We study the complexity of this problem on a space-bounded random-access machine: The input is given in a read-only array and the capacity of workspace is limited. We prove that the linear-time prune-and-search algorithm—presented in most textbooks on algorithms—can be adjusted to use $O(N)$ bits instead of $\Theta(N)$ words of extra space. Prior to our work, the best known algorithm by Frederickson could perform the task with $O(N)$ bits of extra space in $O(N \log^* N)$ time. In particular, our result separates the space-restricted random-access model and the multi-pass streaming model (since we can bypass the $\Omega(N \log^* N)$ lower bound known for the latter model). We also generalize our algorithm for the case when the size of the workspace is $O(S)$ bits, $\lg^3 N \leq S \leq N$. The running time of our generalized algorithm is $O(N \lg^*(N/S) + N \lg N/\lg S)$, slightly improving over the bound $O(N \lg^*(N \lg N/S) + N \lg N/\lg S)$ of Frederickson's algorithm. Of independent interest, the wavelet stack—a structure we used for repeated pruning—may also be useful in other applications.

1 Introduction

Let $x_1, x_2, \ldots, x_N$ be the set of input elements. In the selection problem we want to find the kth smallest of these elements. Without of loss of generality, we can assume that the elements are distinct (since in the case of equal elements the indices can be used to distinguish the elements). That is, the output will be a single index m with a guarantee that $k-1$ elements are smaller than x_m and $N-k$ elements are larger than x_m.

** Research partly supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science and Technology (grant number 2012-0008241).

In the unrestricted random-access machine the asymptotic complexity of the selection problem was settled to be $\Theta(N)$ by Blum et al. [3] in their celebrated article from 1973. Here we study the problem in the *space-restricted random-access model*: The input is given in a read-only array and only a limited amount of additional workspace is available. We focus on the case where the amount of available space is $O(N)$ bits. Surprisingly, although the selection problem with space restriction has been studied in several papers [8, 13, 15, 19], its exact complexity is still not fully resolved—even after our study.

To get a feeling of the difficulties encountered when designing algorithms for this model of computation, let us consider an algorithm that solves the selection problem using $O(\lg N)$ extra bits (a constant number of machine words). As in many other algorithms, we maintain two indices that specify the so-called *filters*; the elements whose values fall within the range of the two filters are still possible candidates for the kth smallest element. We say that the elements within the range of the filters are *alive*. Before proceeding, the input is scanned once to make the minimum and the maximum elements the initial filters.

When the procedure is called, the elements are in a contiguous segment of memory and only those elements that are alive will be considered. Assume that the size of the segment is M. First, we divide the segment into two zones: the first $\lfloor M/2 \rfloor$ elements form the first zone and the remaining elements the second zone. Second, we check which of the zones contains the majority of alive elements (this idea is from [15]); we say that this zone is *heavy*. Third, we select the median of the heavy zone recursively. Let x_ℓ and x_r be the filters and x_m the median found. After the recursive call, we scan through the elements in the current segment to determine whether the kth smallest element is in the interval $(x_\ell \,..\, x_m)$, is equal to x_m, or is in the interval $(x_m \,..\, x_r)$. If x_m is the kth smallest element, we return x_m as output. In the two other cases, we update the filters and set k to $k - j$ if j smaller alive elements were eliminated. Because of the median finding in the heavy zone, at least one fourth of the alive elements will be removed from further consideration. Finally, we recursively find the kth smallest of the remaining alive elements in the whole segment.

We keep information about which of the two subproblems is called and which of the two zones is heavy in a recursion stack. The boundaries of the segment of the caller can be computed from the boundaries of the callee using these additional bits. The filters are passed from the caller to the callee, and vice versa. That is, the workspace is $O(\lg N)$ bits.

In the analysis of the running time, we use A to denote the number of alive elements and M the size of the contiguous segment where these elements reside. Now the worst-case running time can be described using the recurrence:

$$T(A, M) \leq \begin{cases} c_1 \cdot M & \text{if } A < 10 \\ T\left(A', \lceil M/2 \rceil\right) + T\left(A'', M\right) + c_2 \cdot M & \text{if } A \geq 10\,, \end{cases}$$

where c_1 and c_2 are positive constants, and A' and A'' denote the number of alive elements of the subproblems in the first and second recursive calls, respectively. We know that $\lceil A/2 \rceil \leq A' \leq \lceil M/2 \rceil$ and $A'' \leq \lfloor 3A/4 \rfloor$. We encourage the reader to solve this recurrence. (An answer can be found on the last page of this paper.)

Table 1. The best known algorithms for selecting the kth smallest of N elements in the space-restricted random-access model; N is the size of the read-only input. The first three algorithms work for a larger possible range of workspace, but we give their running times only for these specific values.

Inventors	Workspace in bits	Running time
Munro and Raman [15]	$\Theta(\lg N)$	$O(N^{1+\varepsilon})$
Raman and Ramnath [19]	$\Theta(\lg^2 N)$	$O(N \lg^2 N)$
Frederickson [8]	$\Theta(\lg^3 N)$	$O(N \lg N / \lg\lg N)$
Frederickson [8]	$\Theta(N)$	$O(N \log^* N)$
Blum et al. [3]	$\Theta(N \lg N)$	$\Theta(N)$
This paper	$O(N)$	$\Theta(N)$

This algorithm highlights several aspects that are important for algorithms designed for the space-bounded random-access machine. Since we cannot move and modify elements, we have to scan the read-only array several times. The elements that are alive are scattered over a large area, so we have to scan over already eliminated elements several times. Due to the limited memory resources, we cannot store information and we have to recompute some information that has been computed before. Also, because of the limited workspace we have to resort to some bit tricks to save space.

Asymptotically, the algorithm described above is not the fastest known when the amount of workspace is $O(\lg N)$ bits. The performance of the best known selection algorithms is summarized in Table 1. In the current paper we improve the known results when the amount of extra space is $O(N)$ bits by giving a new implementation for the algorithm of Blum et al. [3]. For the general case of $O(S)$ bits of workspace, the best known algorithm is that of Frederickson [8]. The running time of Frederickson's algorithm is $O(N \lg^*(N \lg N/S) + N \lg N/\lg S)$ when $S = \Omega(\lg^3 N)$. We generalize our algorithm to use $O(S)$ bits of workspace, and improve the running time to $O(N \lg^*(N/S) + N \lg N/\lg S)$.

In the literature two different models of computation have been considered: the multi-pass streaming model [4, 8, 13] and the space-restricted random-access model (that is used in this paper) [8, 15, 19]. The essential difference is that in the streaming model the read-only input must be accessed sequentially, but multiple scans of the entire input are allowed. In addition to the running time, the number of scans performed would be another optimization target. Chan [4] proved that Frederickson's algorithm is asymptotically optimal for the selection problem in the multi-pass streaming model. He questioned whether this lower bound would also hold in the space-restricted random-access model. We answer this question by bypassing this bound on the space-bounded random-access machine.

As should be obvious, we rely heavily on the random-access capabilities. The kernel of our construction is the wavelet stack— a new data structure that allows us to eliminate elements while being able to sequentially scan the alive elements and jump over the eliminated ones. In the meantime, such structure only requires a constant number of bits per element (instead of the usual $\lceil \lg N \rceil$ bits required for storing indices). The wavelet stack is by no means restricted to this particular

application; our hope is that it would be generally useful for prune-and-search algorithms in the space-bounded setting. A wavelet stack comprises several layers of bit vectors, each supporting *rank* and *select* queries in $O(1)$ worst-case time [6, 11, 14, 18]. Using the *rank* and *select* facilities, we can navigate between the layers of the stack and perform successor queries efficiently.

2 Basic Tools: Bit Vectors with *rank* and *select* Support

In this section we briefly describe the set of basic tools used by us. The reader is advised to check the original references if our descriptions are too short and more details are needed.

A *bit vector* is an array of bits (0's and 1's). In our case a bit vector V is a data structure that has a fixed size and supports the operations:

$V.access(i)$**:** Return the bit at index i, also denoted as $V[i]$.
$V.rank(i)$**:** Return the number of 1-bits among the bits $V[0], V[1], \ldots, V[i]$.
$V.select(j)$**:** Return the index of the jth 1-bit, i.e. when the return value is i, $V[i] = 1$ and $rank(i) = j$.

Let w denote the size of the machine word in bits. It is a routine matter [12, Section 7.1.3] to store a bit vector of size N such that it occupies $\lceil N/w \rceil$ words and any consecutive substring of at most w bits—not only a single bit—can be accessed in $O(1)$ worst-case time.

There exist several space-efficient solutions to support the two other operations in $O(1)$ worst-case time. Jacobson [11] showed how to support *rank* and *select* in $O(\lg N)$ bit probes using only $o(N)$ bits in addition to the bit vector itself. Clark and Munro [6, 14] showed how to support the queries in $O(1)$ worst-case time on a random-access machine with word size $\Theta(\lg N)$; Raman et al. [18] improved the space bound to $O(N \lg \lg N / \lg N)$ bits, which was shown by Golynski [9] to be optimal provided that the bit vector is stored in plain form. The basic idea in all the mentioned solutions is to divide the input into blocks, store the *rank* and *select* values for some specific positions, and compute the *rank* and *select* values for the remaining positions on the fly using the stored values, values in some precomputed tables, and bits in the original bit vector.

Note that the only requirements on the bit vectors we use are that operations must have $O(1)$ worst-case cost, a space usage must be $O(N)$ bits, and the construction of the supporting structures must take linear worst-case time. For these requirements, Chazelle [5] described a simple solution to support *rank* operations. After breaking the bit vector into words, for the first bit of each word a *landmark* is computed which is the number of 1-bits preceding this position. Let the words be $B_0, B_1, \ldots, B_{\lceil N/w \rceil - 1}$ and the landmarks $L_0, L_1, \ldots, L_{\lceil N/w \rceil - 1}$. To compute $rank(i)$, we locate the corresponding word B_j, i.e. $j = \lfloor i/w \rfloor$, and the offset f inside this word, i.e. $f = i - w \times \lfloor i/w \rfloor$. Then we mask the bits up to index f in B_j and calculate the number of 1-bits in the masked part; let this number be r. As the end result, we output $L_j + r$ as $rank(i)$. The only difficult part is to calculate the number of 1-bits in a word, but fortunately this can

be done by using the population-count function that is a hardware primitive in most modern processors. As far as we know, no equally simple data structure is known to support *select* operations.

3 Expanding the Toolkit: Wavelet Stacks

The computation of a recursive algorithm can be described as a tree. A path from the root of the tree (corresponding to the original problem) to the present node (corresponding to the subproblem being currently solved) summarizes the decisions made by the algorithm when exploring the tree. In a prune-and-search algorithm, where we repeatedly eliminate some of the answer candidates, the set of alive elements can be compactly represented using a bit vector. The computational history of the decisions made by such an algorithm can be conveniently described using a stack of bit vectors. We call this kind of data structure a wavelet stack because of its resemblance to wavelet trees [10] (for a survey, see [16]). In this section we describe the data structure in details, and in the next section we show how it can be used to solve the selection problem. We expect this data structure to be useful in other applications as well.

Let $x_1, x_2, \ldots, x_N$ be a sequence of N elements given in a read-only array. Assume that we want to find some specific subset of these elements using prune-and-search elimination. A prune-and-search algorithm is a recursive procedure that may call itself several times. Hence, we need a recursion stack to keep track of the subproblems being solved. In addition to a recursion stack (with constant-size activation records), we maintain a stack of bit vectors to mark the elements in the current configuration; a 1-bit indicates that the corresponding element is still alive. Subsequently, we can avoid scanning the pruned elements.

In an abstract form, our stack H of bit vectors—called a *wavelet stack*—is a data structure that can efficiently answer two types of queries:

$H.alive?(i)$**:** Return whether the element x_i is alive at the current configuration.
$H.index(j)$**:** Return the index of the jth alive element, i.e. the index of the element corresponding to the jth 1-bit at the top-most bit vector.

To fully understand these operations, we have to consider a concrete implementation of a wavelet stack (see Fig. 1). A wavelet stack is a hierarchy of bit vectors. The bottom-most level stores one bit per element, so at the beginning all elements are potential answers. If we have z 1-bits at level h, the bit vector at level $h + 1$ is of size z. Therefore, the bit vectors become smaller and smaller as we eliminate more elements from further consideration.

The two operations have a nice symmetry: *alive*? traverses up from the bottom to the top of the stack, and *index* traverses down from the top to the bottom of the stack. To implement *alive*?(i), we start from the bottom-most level and compute $rank(i)$. This gives us the index to access the bit vector above. Continuing upwards and relying on *rank*, we either reach a level where the bit corresponding to the index value is 0 indicating that the element x_i is not alive any more, or we reach the top-most level where the bit value is 1 indicating that

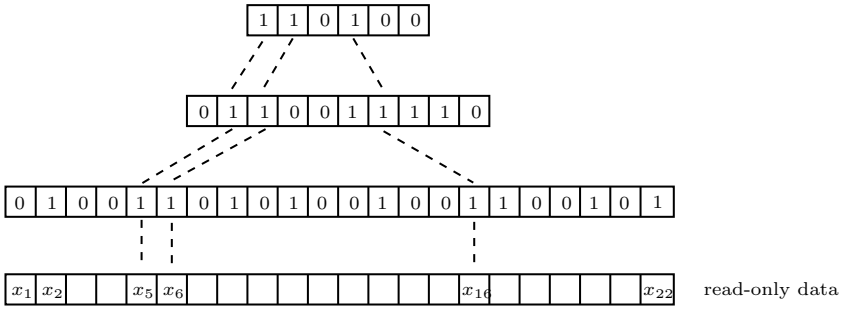

Fig. 1. A wavelet stack for an array of 22 elements. Only elements x_5, x_6, and x_{16} are alive at this point.

x_i is still alive. To implement *index*(j), we start from the top-most level and compute *select*(j). Then we use the returned index at the level below. This way, we can proceed down using *select* until we reach the bottom-most level. Using the last returned index, we can access the desired element whenever needed.

We can summarize the space requirements of the data structure and the time performance of the operations as follows:

Theorem 1. *Assume that we have built a wavelet stack of height h for a read-only array of N elements. Furthermore, assume that at each level we have succeeded in eliminating a constant fraction of the elements.*

1. *The data structure requires $O(N)$ bits in total.*
2. *The total time used in the construction of the data structure is $\Theta(N)$.*
3. *In the worst case, both alive? and index operations take $O(h)$ time.*

Proof. Since the number of bits needed at each level is only a constant fraction of that needed at the level below, for a constant $c < 1$, $\sum_{i=0}^{h-1} c^i N = O(N)$ is an upper bound on the number of bits used. Since the length of the bit vectors is not known beforehand and since their sizes may vary, we can allocate a header storing references to a big bit vector that contains the bits stored at all levels together. This header will only require $O(\lg^2 N)$ bits.

The construction of a bit vector, including the supporting structures, can be done in linear time. The construction time of the wavelet stack can also be expressed as a geometric series, and is thus $\Theta(N)$. Since the structure has h levels, and the *rank* and *select* operations take $O(1)$ worst-case time at each level, the $O(h)$ time bound for *alive*? and *index* follows. □

4 Selection with $O(N)$ Bits

In this section we show how to adapt the prune-and-search algorithm of Blum et al. [3] (alternatively see [7, Section 9.3]) such that it only requires $O(N)$ bits of space—instead of $\Theta(N)$ words—but still runs in $\Theta(N)$ worst-case time.

The basic idea in the algorithm is to search for an element from the set of possible answers and use it to make the set of candidates smaller. This is done repeatedly until the final answer is found. In the variant considered here we use a wavelet stack to keep track of the decisions made by the algorithm. The kth smallest among M alive elements is found as follows.

1. A new bit vector V is pushed onto the top of the wavelet stack; the size of the bit vector equals the number of the currently alive elements M.
2. Divide the set of M elements into groups of five, so that only the last group may have less than 5 elements. Find the median of each of the $\lceil M/5 \rceil$ groups. After processing a group, mark its median as alive and the other elements of the group as not alive in the top-most bit vector V.
3. Recursively compute the median x of these medians.
4. Set the bits of V to indicate that all the M elements are again alive.
5. Scan through the alive elements and determine how many of them are smaller than x and how many are larger. Let these numbers be σ and τ, respectively. If $k = \sigma + 1$, i.e. if x is the kth smallest element, stop and return x as an answer. If $k \leq \sigma$, mark the the elements smaller than x as the only alive elements in V and recursively compute the kth smallest of these elements. Otherwise, if $k > \sigma + 1$, mark the elements larger than x as the only alive elements in V and set k to $k - \sigma - 1$ before the recursive call.
6. After the last recursive call, before returning the answer further to the caller, pop the top-most bit vector V of the wavelet stack.

This algorithm is a perfect example of recursion in action. There are two recursive calls, one at Step 3 and another at Step 5. The analysis of this algorithm is almost identical to that of the original. The key point is that, even though the input is in a read-only array, we do not waste time in browsing the elements that have already been eliminated as we rely on the *rank-select* facilities provided for the bit vectors. The only overhead is when we want to access an element, we have to traverse down the wavelet stack.

Suppose that we resolve subproblems smaller than 100 without recursive calls, e.g. by having a constant-size array of indices to the elements and applying mergesort indirectly via this array. Let $T(h, M)$ denote the worst-case running time of the algorithm for a subproblem having size M when the height of the wavelet stack is h. An upper bound on the worst-case running time can be expressed using a recurrence relation:

$$T(h, M) \leq \begin{cases} c_1 \cdot h & \text{if } M < 100 \\ c_2 \cdot h \cdot M + T\left(h + 1, M'\right) + T\left(h + 1, M''\right) & \text{if } M \geq 100\,, \end{cases}$$

where c_1 and c_2 are some constants, and M' and M'' denote the sizes of the subproblems in the first and second recursive calls, respectively. We know that $M' = \lceil M/5 \rceil$ and that $M'' \leq 7M/10 + 6$ [7, Section 9.3]. For every $M \geq 100$, $7M/10 + 6 \leq 22M/30$ and $\lceil M/5 \rceil \leq 7M/30$. Using these facts, we can solve the recurrence by considering the corresponding recursion tree. The sum of the sizes of the subproblems at the ith level of the recursion tree, $i \geq 1$, is $(29/30)^{i-1}N$. In

accordance, the cost accompanying the ith level is $O(i \cdot (29/30)^{i-1}N)$. Summing the costs for all the levels, it follows that $T(1, N) = O(N)$. The intuition behind the result is that, in spite of the overhead caused by the traversals in the wavelet stack, which increases as a linear function of h, the size of the subproblems decreases exponentially with h.

Since the size of the subproblems is the same as for the original algorithm, the total size of all bit vectors constructed is $O(N)$ too. In addition to the wavelet stack, we need a recursion stack to keep track of the type of the subproblems being solved. However, the depth of recursion is only logarithmic so the recursion stack never uses more than $O(\lg^2 N)$ bits.

The performance of the algorithm is summarized in the following theorem:

Theorem 2. *The kth smallest of N elements in a read-only array can be found in $\Theta(N)$ time using $O(N)$ extra bits in the worst case.*

5 General Solution with $O(S)$ Bits

In this section we extend our algorithm to handle the more general case of using a workspace of $O(S)$ bits, where $\lg^3 N \leq S \leq N$. The main idea is to use Frederickson's algorithm [8] to prune the elements and stop its execution when the number of alive elements is at most S. To finish the selection process, we resume pruning using an $O(N)$-time algorithm that we shall present next.

First we mention the following lemma about Frederickson's algorithm, and omit its proof from this version of the paper. We also refer the reader to [8].

Lemma 1. *By applying a trimmed execution of Frederickson's algorithm, we can prune the elements until the number of alive elements is at most S in $O(N \lg^*(N/S))$ worst-case time, assuming $S = \Omega(\lg^3 N)$.*

If $S \leq \sqrt{N \lg N}$, we simply use Frederickson's algorithm all the way. The resulting running time is $O(N \lg^* ((N \lg N)/S) + N \lg N / \lg S) = O(N \lg^*(N/S) + N \lg N / \lg S)$ as claimed. From now on we assume that $S > \sqrt{N \lg N}$. We apply a trimmed execution of Frederickson's algorithm as specified by Lemma 1. The outcome is two filters that guard the—at most—S candidates. Consequently, we are left with the task of selecting the designated element among those candidates.

Using a wavelet stack and a bit vector supporting *rank* and *select* queries, we can finish the pruning in $O(N)$ time. We create and maintain a wavelet stack H—an element hierarchy where each bit corresponds to an element among those whose values fall between the filters. Since there are at most S such elements, the wavelet stack H uses $O(S)$ bits. While our algorithm is in action, the wavelet stack is to be updated to indicate the elements that are currently surviving the pruning phases. We divide the input sequence (consisting of N elements) into S buckets, where the uth bucket consists of the elements from the input sequence with indices from the range $[u \cdot \lceil N/S \rceil \,..\, (u+1) \cdot \lceil N/S \rceil - 1]$, for $0 \leq u \leq S - 1$ (except possibly the last bucket). In addition, we create the *count vector* C—a static bit vector that indicates the number of candidates originally contained in

each bucket after the execution of Frederickson's algorithm. The count vector C should efficiently support *rank* and *select* queries. We store these counts encoded in unary, using a 0-bit to mark the border between every two consecutive buckets. Since a total of at most S candidates need to be stored, the count vector C contains at most S ones; and since we have exactly S buckets, C contains $S-1$ zeros. The count vector thus uses $O(S)$ bits as well.

We can now iterate efficiently through the alive candidates. Let $i-1$ be the rank of the element that has just been considered in our iterative scan within the alive elements. First, we find the index j of the next element to be considered within the wavelet stack. For that we set

$$j = H.index(i),$$

which is the index of the element we are looking for with respect to those falling between the two filters inherited from Fredrickson's algorithm. Since the difference between the index of a bit, in the count vector C, and its rank is exactly the index of the bucket its corresponding element belongs to, we can compute the index u (first bucket has index 0) of the bucket containing this element as

$$u = C.select(j) - j.$$

We then calculate the index t that corresponds to the 0-bit resembling the border between the uth and $(u-1)$th buckets. This is done by setting $t = \bar{C}.select(u)$, where $\bar{C}$ is the complement vector of C. Since the total number of elements, among those surviving Frederickson's algorithm, stored in the u preceding buckets to the current one can be computed by a *rank* query for t within C, we determine the position p of the sought element among those candidates within the surviving elements of the uth bucket as:

$$p = j - C.rank(t).$$

If the element that has just been reported was also from bucket u, we continue scanning the elements of the uth bucket from where we stopped. Otherwise, we jump to the beginning of the uth bucket, i.e. to the element whose index is $u \cdot \lceil N/S \rceil$ in the input array. We sequentially scan the elements of the located bucket, discard the ones falling outside the filters, and count the others until locating the pth element among them; this is the element we are looking for.

We can now proceed as in the $O(N)$-bit solution. Starting with the elements surviving Frederickson's algorithm, we recursively determine the median-of-medians and use it to perform the pruning. During this process we keep the wavelet stack up to date as before. The pruning process continues until only one bucket remains, at such point only $O(N/S)$ elements are alive. Since this procedure of the algorithm is employed only when $S = \Omega(\sqrt{N \lg N})$, the indices of the alive elements can fit in the allowable workspace, each in $O(\lg N)$ bits, and we continue the selection in linear time.

Since we are operating on buckets, we might have to spend $\Theta(N/S)$ time for scanning per bucket. However, we note that initially there is at most S candidates

and accordingly at most S buckets. Since we prune a constant fraction of the candidates in each iteration, we also reduce the bound on the number of the remaining buckets (those having at least one alive element each) by the same constant fraction. Noting that we skip the buckets that have no alive elements, the work done per pass to iterate over the buckets that have at least one alive element can be bounded, as elaborated in the next lemma.

Lemma 2. *Given a read-only input array X, where $|X| = N$, and two filters f_1 and f_2, such that the element to be selected lies in $I = \{e | f_1 \leq e \leq f_2, e \in X\}$ and $|I| \leq S$; that is, at most S elements lie between the filters. If $S = \Omega(\sqrt{N \lg N})$, we can solve the selection problem in $O(N)$ time.*

Proof. In each pruning iteration we spend time proportional to the number of buckets remaining, while scanning the elements in these buckets and comparing them with the filters. The number of alive elements after we apply the ith pruning iteration of the median-of-medians algorithm is $O(S/c^i)$, for some constant $c > 1$. Obviously, the number of buckets that have alive elements cannot exceed the number of elements. It follows that, throughout all the passes of the algorithm, the number of scanned buckets is at most $O(\sum_{i \geq 0} S/c^i) = O(S)$. Accordingly, the overall work done in scanning these buckets is $O(N)$. Once we have $O(N/S)$ elements remaining, as $S = \Omega(\sqrt{N \lg N})$, we can continue the selection process in the working memory in $O(N/S)$ time. □

The main result of this paper is summarized in the upcoming theorem.

Theorem 3. *Given a read-only array of N elements and a workspace of $O(S)$ bits such that $\lg^3 N \leq S \leq N$, it is possible to solve the selection problem in $O(N \lg^*(N/S) + N \lg N / \lg S)$ worst-case time in the space-restricted random-access model.*

Theorem 3 implies that, in the read-only space-limited setting, Chan's lower bound [4] for the selection problem in the multi-pass streaming model does not apply to the space-restricted random-access model. This, in turn, indicates that the space-restricted random-access model is more powerful than the multi-pass streaming model.

6 Conclusions

We showed that, given an array of N elements in a read-only memory, the kth smallest element can be found in $\Theta(N)$ worst-case time using $O(N)$ bits of extra space. We also generalized our algorithm to run in $O(N \lg^*(N/S) + N \lg N / \lg S)$ time using a workspace of $O(S)$ bits, $\lg^3 N \leq S \leq N$. Our main purpose was to show that the lower bound proved by Chan [4] for the multi-pass streaming model can be bypassed in the space-restricted random-access model.

In the read-only setting the selection problem has been studied since 1980 [13]. In contrast to sorting, the exact complexity of selection is still open. The space-time trade-off for sorting is known to be $\Theta(N^2/S + N \lg S)$ [2, 17], where S

is the asymptotic target for the size of the workspace in bits, $\lg N \leq S \leq N/\lg N$. The optimal bound for sorting can even be realized using a natural priority-queue-based algorithm [1].

References

1. Asano, T., Elmasry, A., Katajainen, J.: Priority queues and sorting for read-only data. In: Chan, T-H.H., Lau, L.C., Trevisan, L. (eds.) TAMC 2013. LNCS, vol. 7876, pp. 32–41. Springer, Heidelberg (2013)
2. Beame, P.: A general sequential time-space tradeoff for finding unique elements. SIAM J. Comput. 20(2), 270–277 (1991)
3. Blum, M., Floyd, R.W., Pratt, V., Rivest, R.L., Tarjan, R.E.: Time bounds for selection. J. Comput. System Sci. 7(4), 448–461 (1973)
4. Chan, T.M.: Comparison-based time-space lower bounds for selection. ACM Trans. Algorithms 6(2), 26:1–26:16 (2010)
5. Chazelle, B.: A functional approach to data structures and its use in multidimensional searching. SIAM J. Comput. 17(3), 427–462 (1988)
6. Clark, D.: Compact Pat Trees. Ph.D. thesis, Department of Computer Science, University of Waterloo, Waterloo (1996)
7. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms. The MIT Press, Cambridge, 3rd edn. (2009)
8. Frederickson, G.N.: Upper bounds for time-space trade-offs in sorting and selection. J. Comput. System Sci. 34(1), 19–26 (1987)
9. Golynski, A.: Optimal lower bounds for rank and select indexes. Theoret. Comput. Sci. 387(3), 348–359 (2007)
10. Grossi, R., Gupta, A., Vitter, J.S.: High-order entropy-compressed text indexes. In: SODA 2003. pp. 841–850. SIAM, Philadelphia (2003)
11. Jacobson, G.: Space-efficient static trees and graphs. In: FOCS 1989. pp. 549–554. IEEE Computer Society, Los Alamitos (1989)
12. Knuth, D.E.: Combinatorial Algorithms: Part 1, The Art of Computer Programming, vol. 4A. Addison-Wesley, Boston (2011)
13. Munro, J.I., Paterson, M.S.: Selection and sorting with limited storage. Theoret. Comput. Sci. 12(3), 315–323 (1980)
14. Munro, J.I.: Tables. In: Chandru, V., Vinay, V. (eds.) FSTTCS 1996. LNCS, vol. 1180, pp. 37–42. Springer, Heidelberg (1996)
15. Munro, J.I., Raman, V.: Selection from read-only memory and sorting with minimum data movement. Theoret. Comput. Sci. 165(2), 311–323 (1996)
16. Navarro, G.: Wavelet trees for all. In: Kärkkäinen, J., Stoye, J. (eds.) CPM 2012. LNCS, vol. 7354, pp. 2–26. Springer, Heidelberg (2012)
17. Pagter, J., Rauhe, T.: Optimal time-space trade-offs for sorting. In: FOCS 1998. pp. 264–268. IEEE Computer Society, Los Alamitos (1998)
18. Raman, R., Raman, V., Satti, S.R.: Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. ACM Trans. Algorithms 3(4), Article 43 (2007)
19. Raman, V., Ramnath, S.: Improved upper bounds for time-space trade-offs for selection. Nordic J. Comput. 6(2), 162–180 (1999)

Answer: By substitution one can show that $\Omega(N^2) < T(N-2, N) < O(N^3)$.