2015

Near-Optimal Online Multiselection in Internal and External Memory

Jonathan P. Sorenson

Jérémy Barbay

Ankur Gupta

S. Srinivasa Rao

Near-Optimal Online Multiselection in Internal and External Memory

Jérémy Barbay*
Universidad de Chile
jbarbay@dcc.uchile.cl

Ankur Gupta†
Butler University
agupta@butler.edu

S. Srinivasa Rao‡
Seoul National University
ssrao@cse.snu.ac.kr

Jon Sorenson†
Butler University
jsorenso@butler.edu

Abstract

We introduce an online version of the *multiselection* problem, in which q selection queries are requested on an unsorted array of n elements. We provide the first online algorithm that is 1-competitive with offline algorithm proposed by Kaligosi et al.[ICALP 2005] in terms of comparison complexity. Our algorithm also supports online *search* queries efficiently.

We then extend our algorithm to the dynamic setting, while retaining online functionality, by supporting arbitrary *insertions* and *deletions* on the array. Assuming that the insertion of an element is immediately preceded by a search for that element, we show that our dynamic online algorithm performs an optimal number of comparisons, up to lower order terms and an additive $O(n)$ term.

For the external memory model, we describe the first online multiselection algorithm that is $O(1)$-competitive. This result improves upon the work of Sibeyn [Journal of Algorithms 2006] when $q > m$, where m is the number of blocks that can be stored in main memory. We also extend it to support searches, insertions, and deletions of elements efficiently.

1 Introduction

The *multiselection* problem asks for the elements of rank $Q = q_1, q_2, \ldots, q_q$ on an unsorted array A drawn from an ordered universe of elements. We define $\mathcal{B}(S_q)$ as the information-theoretic lower bound on the number of comparisons required in the comparison model to answer q queries, where $S_q = s_i$ denotes the queries ordered by rank. We define $\Delta_i = s_{i+1} - s_i$, where $s_0 = 0$ and $s_{q+1} = n$. Then,

$$\mathcal{B}(S_q) = \log n! - \sum_{i=0}^{q} \log(\Delta_i!) \in \sum_{i=0}^{q} \Delta_i \log \frac{n}{\Delta_i} - O(n).^1$$

Several papers have analyzed this problem. Dobkin and Munro [DM81] gave a deterministic bound using $3\mathcal{B}(S_q) + O(n)$ comparisons. Prodinger [Pro95] proved the expected comparisons with random pivoting is $2\mathcal{B}(S_q)\ln 2 + O(n)$. Most recently, Kaligosi et al. [KMMS05] showed a randomized algorithm performing $\mathcal{B}(S_q) + O(n)$ expected comparisons, along with a deterministic algorithm performing $\mathcal{B}(S_q) + o(\mathcal{B}(S_q) + O(n)$ comparisons. Jiménez and Martínez [JM10] later improved the number of comparisons in the expected case to $\mathcal{B}(S_q) + n + o(n)$ when $q \in o(\sqrt{n})$. Most recently, Cardinal et al. [CFJ+09] generalized the problem to a *partial order production*, of which multiselection is a special case. Cardinal et al. use the multiselection algorithm as a subroutine after an initial preprocessing phase.

Kaligosi et al. [KMMS05] provide an elegant result in the deterministic case based on tying the number of comparisons required for merging two sorted sequences to the information content of those sequences. This simple observation drives an approach where manipulating these runs both finds pivots that are "good enough" and partitions with near-optimal comparisons. The weakness of the approaches in internal memory is that they must know all of the queries a priori.

*Supported by Project Regular Fondecyt number 1120054.
†Supported in part by the Butler Holcomb Awards grant.
‡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).

[1] We use the notation $\log_b a$ to refer to the base b logarithm of a. By default, we let $b = 2$. We also define $\ln a$ as the base e logarithm of a.

In the external memory model with parameters M and B, we use N to denote the number of elements in $\mathtt{A}$. We also define $n = N/B$ and $m = M/B$ in external memory. Sibeyn [Sib06] solves external multiselection using $n + nq/m^{1-\epsilon}$ I/Os, where ϵ is any positive constant. The first term comes from creating a static index structure using n I/Os, and the reminder comes from the q searches in that index. In addition, his results also require the condition that $B \in \Omega(\log_m n)$. When $q = m$, Sibeyn's multiselection algorithm requires $O(nm^\epsilon)$ I/Os, whereas the optimum is $\Theta(n)$ I/Os. In fact his bounds are $\omega(\mathcal{B}_m(S_q))$, for any $q \geq m$, where $\mathcal{B}_m(S_q)$ is the lower bound on the number of I/Os required (see Section 6.1 for the definition).

1.1 Our Results

For the *multiselection* problem in internal memory, we describe the first online algorithm that supports a set Q of q selection, search, insert, and delete operations, of which q' are search, insert, and delete, using $\mathcal{B}(S_q) + o(\mathcal{B}(S_q) + O(n + q' \log n)$ comparisons.[2] Thus our algorithm is 1-competitive with the offline algorithm of Kaligosi et al. [KMMS05] in the number of comparisons performed. We also show a randomized result achieving 1-competitive behavior with respect to Kaligosi et al. [KMMS05], while only using $O((\log n)^{O(1)})$ sampled elements instead of $O(n^{3/4})$.

For the external memory model [AV88], we describe an online multiselection algorithm that supports a set Q of q selection queries on an unsorted array stored on disk in n blocks, using $O(\mathcal{B}_m(S_q))$ I/Os, where $\mathcal{B}_m(S_q)$ is a lower bound on the number of I/Os required to support the given queries. This result improves upon the work of Sibeyn [Journal of Algorithms 2006] when $q > m$, where m is the number of blocks that can be stored in main memory. We also extend it to support insertions and deletions of elements using $O(\mathcal{B}_m(S_q)) + (N/m))$ I/Os.

1.2 Preliminaries

Given an unsorted array $\mathtt{A}$ of length n, the *median* is the element x of $\mathtt{A}$ such that exactly $\lceil n/2 \rceil$ elements in $\mathtt{A}$ are greater than or equal to x. It is well-known that the median can be computed in $O(n)$ comparisons, and many [Hoa61, BFP+73, SPP76] have analyzed the exact constants involved. Dor and Zwick [DZ99] provide the best known constant, yielding a $2.942n + o(n)$ comparisons.

In the external memory model, we consider only two memory levels: the internal memory of size M, and the (unbounded) disk memory, which operates by reading and writing data in blocks of size B. We refer to the number of items of the input by N. For convenience, we define $n = N/B$ and $m = M/B$ as the number of blocks of input and memory, respectively. We make the reasonable assumption that $1 \leq B \leq M/2$. In this model, we assume that each I/O read or write is charged one unit of time, and that an internal memory operation is charged no units of time. To achieve the optimal sorting bound of $SortIO(N) \in \Theta(n \log_m n)$ in this setting, it is necessary to make the *tall cache* assumption [BF03]: $M \in \Omega(B^{1+\epsilon})$, for some constant $\epsilon > 0$, and we will make this assumption for the remainder of the paper.

2 A Simple Online Algorithm

Let $\mathtt{A}$ be an input array of n unsorted items. We describe a simple version of our algorithm for handling selection and search queries on array $\mathtt{A}$. We say that an element in array $\mathtt{A}$ at position i is a *pivot* if $\mathtt{A}[1 \ldots i-1] < \mathtt{A}[i] \leq \mathtt{A}[i+1 \ldots n]$.

Bit Vector. Throughout all the algorithms in the paper, we maintain a bitvector $\mathtt{V}$ of length n where $\mathtt{V}[i] = \mathbf{1}$ if and only if it is a pivot.

Preprocessing. Create a bitvector $\mathtt{V}$ and set each bit to $\mathbf{0}$. Find the minimum and maximum elements in array $\mathtt{A}$, swap them into $\mathtt{A}[1]$ and $\mathtt{A}[n]$ respectively, and set $\mathtt{V}[1] = \mathtt{V}[n] = \mathbf{1}$.

Selection. We define the operation $\mathtt{A}.select(s)$ to refer to the selection query s, which returns $\mathtt{A}[s]$ if $\mathtt{A}$ were sorted. To compute this result, if $\mathtt{V}[s] = \mathbf{1}$ then return $\mathtt{A}[s]$ and we are done. If $\mathtt{V}[s] = \mathbf{0}$, find $a < s$, $b > s$, such that $\mathtt{V}[a] = \mathtt{V}[b] = \mathbf{1}$ but $\mathtt{V}[a+1 \ldots b-1]$ are all $\mathbf{0}$. Perform quickselect [Hoa61] on $\mathtt{A}[a+1 \ldots b-1]$, marking pivots found along the way in $\mathtt{V}$. This gives us $\mathtt{A}[s]$, with $\mathtt{V}[s] = \mathbf{1}$, as desired.

Search. We define the operation $\mathtt{A}.search(p)$ returns the position j, which satisfies $p = \mathtt{A}[j]$ if $\mathtt{A}$ were sorted; if $p \notin \mathtt{A}$, then j is the number of items in $\mathtt{A}$ smaller than p.[3] Perform a binary search on $\mathtt{A}$ *as if* $\mathtt{A}$ *were sorted.* Let i be the location in $\mathtt{A}$ we find from the search; if along the way we discovered endpoints for the subarray we are searching that were out of order, stop the search and let i be the midpoint. If

[2] For the dynamic result, we assume that the insertion of an element is immediately preceded by a search for that element. In that case, we show that our dynamic online algorithm performs an optimal number of comparisons, up to lower order terms and an additive $O(n)$ term.

[3] The *search* operation is essentially the same as *rank* on the set of elements stored in the array A. We call it *search* to avoid confusion with the *rank* operation defined on bitvectors in Section 5.

$\mathtt{A}[i] = p$ and $\mathtt{V}[i] = \mathbf{1}$ return i and we are done. Otherwise, we have just identified the unsorted interval in $\mathtt{A}$ that contains p if it is present. Perform a selection query on this interval; choose which side of a pivot on which to recurse based on the *value* of p (instead of an array position as would be done in a normal selection query). As above, we mark pivots in $\mathtt{V}$ as we go; at the end of the recursion we will discover the needed value j.

As queries arrive, our algorithm performs the same steps that quicksort would perform, although not necessarily in the same order. If we receive enough queries, we will, over time, perform a quicksort on array $\mathtt{A}$. This also means that our recursive subproblems mimic those from quicksort.

We have assumed, up to this point, that the last item in an interval is used as the pivot, and a simple linear-time partition algorithm is used. We explore using different pivot and partitioning strategies to obtain various complexity results for online selection and searching. The time to perform q select and search queries on an array of n items can easily be shown to be $O(n \log q + q \log n)$. We do not prove this bound directly, since our main result is an improvement over this bound. Now, we define terminology for this alternate analysis.

2.1 Terminology

For now we assume that all queries are selection queries, since search queries are selection queries with a binary search preprocessing phase taking $O(\log n)$ comparisons. We explicitly bound the binary search cost in our remaining results.

Query and Pivot Sets. Let Q denote a sequence of q selection queries, ordered by time of arrival. Let $S_t = \{s_1, s_2, \ldots, s_t\}$ denote the first t queries from Q, sorted by position. We also include $s_0 = 1$ and $s_{t+1} = n$ in S_t for convenience of notation, since the minimum and maximum are found during preprocessing. Let $P_t = \{p_i\}$ denote the set of k pivots found by the algorithm when processing S_t, again sorted by position. Note that $p_1 = 1$, $p_k = n$, $\mathtt{V}[p_i] = \mathbf{1}$ for all i, and $S_t \subseteq P_t$.

Pivot Tree, Recursion Depth, and Intervals. The pivots chosen by the algorithm form a binary tree structure, defined as the *pivot tree* T of the algorithm over time.[4] Pivot p_i is the parent of pivot p_j if, after p_i was used to partition an interval, p_j was the pivot used to partition either the right or left half of that interval. The root pivot is the pivot used to partition $\mathtt{A}[2..n-1]$ due to preprocessing. The *recursion depth*, $d(p_i)$, of a pivot p_i is the length of the path in the pivot tree from p_i to the root pivot. All leaves in the pivot tree are also selection queries, but it may be the case that a query is not a leaf. Each pivot was used to partition an interval in $\mathtt{A}$. Let $I(p_i)$ denote the interval partitioned by p_i (which may be empty), and let $|I(p_i)|$ denote its length. Intervals form a binary tree induced by their pivots. If p_i is an ancestor of p_j then $I(p_j) \subset I(p_i)$. The recursion depth of an array element is the recursion depth of the smallest interval containing that element, which in turn is the recursion depth of its pivot.

[4]Intuitively, a pivot tree corresponds to a *recursion tree*, since each node represents one recursive call made during the quickselect algorithm [Hoa61].

Gaps and Entropy. Define the query gap $\Delta_i^{S_t} = s_{i+1} - s_i$ and similarly the pivot gap $\Delta_i^{P_t} = p_{i+1} - p_i$. Observe that each pivot gap is contained in a smallest interval $I(p)$. One endpoint of this gap is the pivot p of interval $I(p)$, and the other matches one of the endpoints of interval $I(p)$. By telescoping we have $\sum_i \Delta_i^{S_t} = \sum_j \Delta_j^{P_t} = n - 1$.

We will analyze the complexity of our algorithms based on the number of element comparisons. The lower bound on the number of comparisons required to answer the selection queries in S_t is obtained by taking the number of comparisons to sort the entire array, and then subtracting the comparisons needed to sort the query gaps. We use $\mathcal{B}(S_t)$ to denote this lower bound.

$$\mathcal{B}(S_t) \quad = \quad \sum_{i=0}^{t} \left(\Delta_i^{S_t}\right) \log\left(n / \left(\Delta_i^{S_t}\right)\right) - O(n).$$

Note that $\mathcal{B}(S_q) \leq n \log q$: this upper bound is met when the queries are evenly spaced over the input array $\mathtt{A}$. We can show that the simple algorithm performs $O(\mathcal{B}(S_q) + q \log n)$ for a sequence Q of q select and search queries on an array of n elements. We will also make use of the following fact.

Fact 1. *For all $\epsilon > 0$, there exists a constant c_ϵ such that for all $x \geq 4$, $\log\log\log x < \epsilon \log x + c_\epsilon$.*

Proof. Since $\lim_{x\to\infty}(\log\log\log x)/(\log x) = 0$, there exists a k_ϵ such that for all $x \geq k_\epsilon$, we know that $(\log\log\log x)/(\log x) < \epsilon$. Also, in the interval $[4, k_\epsilon]$, the continuous function $\log\log\log x - \epsilon \log x$ is bounded. Let $c_\epsilon = \log\log\log k_\epsilon - 2\epsilon$, which is a constant. □

3 Analysis of the Simple Algorithm

In this section we analyze the simple online multiselect algorithm of Section 2.

3.1 A Lemma on Sorting Entropy

Pivot Selection Methods. We say that a pivot selection method is *good* for the constant c with $1/2 \leq c < 1$ if, for all pairs of pivots p_i and p_j where p_i is an ancestor of p_j in the pivot tree, then

$$|I(p_j)| \leq |I(p_i)| \cdot c^{d(p_j)-d(p_i)+O(1)}.$$

Note that if the median is always chosen as the pivot, we have $c = 1/2$ and the $O(1)$ term is in fact zero. The pivot selection method of Kaligosi et al. [KMMS05, Lemma 8] is *good* with $c = 15/16$.

Lemma 1. *If the pivot selection method is good as defined above, then $\mathcal{B}(P_t) \in \mathcal{B}(S_t) + O(n)$.*

Proof. We sketch the proof and defer the full details to the full version of the paper. Consider any two consecutive selection queries s and s', and let $\Delta = s' - s$ be the gap between them. Let $P_\Delta = (p_l, p_{l+1}, \ldots, p_r)$ be the pivots in this gap, where $p_l = s$ and $p_r = s'$. Note that $\mathcal{B}(P_t) = (n \log n - \sum_{j=0}^{k} \Delta_j^{P_t} \log \Delta_j^{P_t})$. We define $\mathcal{B}(S_t)$ similarly. The lemma follows from the claim that $\mathcal{B}(P_\Delta) \in O(\Delta)$, since

$$\begin{aligned}
\mathcal{B}(P_t) - \mathcal{B}(S_t) &= \sum_{i=0}^{t} \Delta_i^{S_t} \log \Delta_i^{S_t} - \sum_{j=0}^{k} \Delta_j^{P_t} \log \Delta_j^{P_t} \\
&= \sum_{i=0}^{t} \mathcal{B}(P_{\Delta_i^{S_t}}) \\
&= \sum_{i=0}^{t} O\left(\Delta_i^{S_t}\right) \in O(n).
\end{aligned}$$

We now sketch the proof of our claim.

There must be a unique pivot p_m in P_Δ of minimal recursion depth. We split the gap Δ at p_m. We define For brevity, we define $D_l = \sum_{i=0}^{m-1} \Delta_i$ and $D_r = \sum_{i=m}^{r-1} \Delta_i$, giving $\Delta = D_l + D_r$.

We consider the proof on the right-hand side D_r, and proof for D_l is similar. Since we use a good pivot selection method, we can bound the total information content of the right-hand side by $O(D_r)$. This leads to the claim, and the proof follows. □

Theorem 1 (Online Multiselection)**.** *Given an array of n elements, on which we have performed a sequence Q of q online selection and search queries, of which q' are search, we provide*

- *a randomized online algorithm that performs the queries using $\mathcal{B}(S_q) + O(n + q' \log n)$ expected number of comparisons, and*
- *a deterministic online algorithm that performs the queries using at most $4\mathcal{B}(S_q)+O(n+q' \log n)$ comparisons.*

Proof. For the randomized algorithm, we use the randomized pivot selection algorithm of Kaligosi et al. [KMMS05, Section 3, Lemma 2].) This algorithm gives a good pivot selection method with $c = 1/2+o(1)$, and the time to choose the pivot is $O(\Delta^{3/4})$ on an interval of length Δ, which is subsumed in the $O(n)$ term in the running time. Each element in an interval participates in one comparison per partition operation. Thus, the total number of comparisons is expected to be the sum of the recursion depths of all elements in the array. This total is easily shown to be $\mathcal{B}(P_q)$, and by Lemma 1, the proof is complete.

For the deterministic algorithm, we use the median of each interval as the pivot; the median-finding algorithm of Dor and Zwick [DZ99] gives this to us in under 3Δ comparisons. We add another comparison for the partitioning, to give a count of comparisons per array element of four times the recursion depth. This is at most $4\mathcal{B}(P_q)$, which is no more than $4\mathcal{B}(S_q) + O(n)$ from Lemma 1, and the result follows. □

In Section 3.2, we describe how to get a good pivot selection method with just $6(\log n)^3(\log \Delta)^2$ samples, instead of $O(\Delta^{3/4})$ samples.

3.2 Reducing the Samples Used by the Randomized Algorithm

Our pivot-choosing method is simple and randomized. We choose $2m$ elements at random from an interval of size Δ, sort them (or use a median-finding algorithm) to find the median, and use that for our pivot. We wish to set values of m and t such that two events happen:

- At least $2t$ elements are chosen in an interval of size $2\Delta/\log \Delta$ about the median of the interval.
- Between $m - t$ and $m + t$ elements are chosen less than the median.
- Between $m - t$ and $m + t$ elements are chosen larger than the median.

If we can show that all events happen with probability $1 - O(1/n^2)$, then we end up with the median of our $2m$ elements being a pivot at position $1/2(1 + O(1/\log \Delta))$, which is a good pivot.

Note that the last two events are mirror images of one another, and so have the same probability of occurring.

First Event. This is the simpler of the two to estimate. A randomly chosen element fails to land in the middle interval with probability $1 - 2/\log\Delta = \exp[-2/\log\Delta(1 + o(1))]$. If we choose at least $(1.1)\log\Delta\log n$ elements, all fail to land in this middle interval with probability $(1 - 2/\log\Delta)^{(1.1)\log\Delta\log n} = \exp[-(2.2)\log n(1 + o(1))] \in O(1/n^2)$. Since we need $2t$ elements in the interval, it suffices for $2m \geq (2.2)t\log\Delta\log n$, or $m \geq (1.1)t\log\Delta\log n$.

Second (and third) Event. We need a bound on the sum of the first k binomial coefficients.

We use the following lemma to bound the summation of binomial coefficients:

Lemma 2 ([LPV03])**.** *Let $0 \leq k < m$ and define $c := \binom{2m}{k+1}/\binom{2m}{m}$. Then*

$$\sum_{i=0}^{k}\binom{2m}{i} < \frac{c}{2}\cdot 2^{2m}.$$

Proof. Write $k + 1 = m - t$. Define

$$A := \sum_{i=0}^{m-t-1}\binom{2m}{i}$$

$$B := \sum_{i=m-t}^{m}\binom{2m}{i}$$

By the definition of c we have

$$\binom{2m}{m-t} = c\binom{2m}{m}$$

and, because the growth rate of one binomial coefficient to the next slows as we approach $\binom{2m}{m}$, we have

$$\binom{2m}{m-t-1} < c\binom{2m}{m-1}$$

and thus

$$\binom{2m}{m-t-j} < c\binom{2m}{m-j}$$

for $0 \leq j \leq m - t$.

Thus it follows that the sum of any t consecutive binomial coefficients is less than c times the sum of the next t coefficients as long as we stay on the left-hand side of Pascal's triangle. Thus $A < cB + c^2B + c^3B + \cdots < \frac{c}{1-c}B$. We also have $A + B \leq 2^{2m-1}$. Combining these we have

$$A < \frac{c}{1-c}B \leq \frac{c}{c-1}\left(2^{2m-1} - A\right).$$

Solving for A completes the proof. □

We then bound

$$\frac{\binom{2m}{m-t}}{\binom{2m}{m}} \leq e^{-t^2/(m+t)}.$$

This can be derived from Stirling's formula and Taylor series estimates for the exponential and logarithm functions. We then obtain that

Lemma 3. *Let $0 \leq t < m$. Then*

$$\sum_{i=0}^{m-t-1}\binom{2m}{i} < 2^{2m-1}\cdot e^{-t^2/(m+t)}.$$

Since choosing an element from an interval at random and observing if it falls before or after the median is an event of probability $1/2$, the event of choosing $2m$ elements and having less than $m-t$ fall below the median occurs with probability at most

$$2^{-2m}\sum_{i=0}^{m-t-1}\binom{2m}{i}.$$

By Lemma 3, this is bounded by $(1/2)\exp[-t^2/(m + t)]$. Thus, the probability there are between $m - t$ and $m + t$ elements below the median is at least $1 - \exp[-t^2/(m+t)]$ by the symmetry of Pascal's triangle. To obtain $1 - O(1/n^2)$ we need $t^2/(m + t) > 2\log n$, or $t \geq \sqrt{2m\log n}(1 + o(1)$.

Using our lower bound for m in terms of t above, we conclude that $m = 6(\log n)^3(\log\Delta)^2$ and $t = 4(\log n)^2\log\Delta$ meet our needs.

Theorem 2. *Given a list of elements of length $\Delta < n$, with Δ at least $6(\log n)^3(\log\Delta)^2$, with probability at least $1 - O(1/n^2)$, if we sample $6(\log n)^3(\log\Delta)^2$ of the Δ elements uniformly at random, then median of the sample falls in position $\Delta/2 \pm \Delta/\log\Delta$ in the original list.*

4 Optimal Online Multiselection

In this section we prove the following theorem.

Theorem 3 (Optimal Online Multiselection)**.** *Given an unsorted array `A` of n elements, we provide a deterministic algorithm that supports a sequence Q of q online selection and search queries, of which q' are search, using $\mathcal{B}(S_q)(1 + o(1)) + O(n + q'\log n)$ comparisons in the worst case.*

Note that our bounds match those of the offline algorithm of Kaligosi et al. [KMMS05] when $q' = 0$ (i.e., there are no search queries). In other words,

we provide the first 1-competitive online multiselection algorithm. We explain our proof with three main steps. In Section 4.1, we explain our algorithm and describe how it is different from the algorithm in [KMMS05]. We then bound the number of comparisons from merging by $\mathcal{B}(S_q)(1+o(1))+O(n)$ in Section 4.2, and in Section 4.3, we bound the number of comparisons from pivot finding and partitioning by $o(\mathcal{B}(S_q))+O(n)$.

4.1 Algorithm Description

We briefly describe the deterministic algorithm from Kaligosi et al. [KMMS05]. They begin by creating *runs*, which are sorted sequences from `A` of length roughly $\ell = \log(\mathcal{B}/n)$. Then, they compute the median m of the median of these sequences and partition the runs based on m. After partitioning, they recurse on the two sets of runs, sending *select* queries to the appropriate side of the recursion. To maintain the invariant on run length on the recursions, they merge short like-sized runs optimally until all but ℓ of the runs are again of length between ℓ and 2ℓ.

We make the following modifications to the deterministic algorithm of Kaligosi et al. [KMMS05]:

- The queries are processed online, that is, one at a time, from Q without knowing which queries will follow. To do this, we maintain the bitvector `V` as described in Section 2.
- We admit search queries in addition to selection queries; in the analysis we treat them as selection queries, paying $O(q' \log n)$ comparisons to account for binary search.
- Since we don't know all of Q at the start, we cannot know the value of $\mathcal{B}(S_q)$ in advance. Therefore, we cannot preset a value for ℓ as in Kaligosi et al. [KMMS05]. Instead, we set ℓ locally in an interval $I(p)$ to $1+\lfloor \lg(d(p)+1) \rfloor$. Thus, ℓ starts at 1 at the root of the pivot tree T, and since we use only good pivots, $d(p) \in O(\lg n)$. (Also, $\ell \in \log\log n + O(1)$ in the worst case.) We keep track of the recursion depth of pivots, from which it is easy to compute the recursion depth of an interval. Also observe that ℓ can increase by at most one when moving down one recursion level during a selection.
- We use a second bitvector `R` to identify the endpoints of runs within each interval that has not yet been partitioned.

The selection algorithm to perform a selection query is as follows:

- As described earlier in this paper, we use bitvector `V` to identify the interval from which to begin processing. The minimum and maximum are found in preprocessing.
- If the current interval has length less than $4\ell^2$, we sort the interval to complete the query (setting all elements as pivots). The cost for this case is bounded by Lemma 7.
- As in Kaligosi et al. [KMMS05], we compute the value of ℓ for the current interval, merge runs so that there is at most one of each length $< \ell$, and then use medians of those runs to compute a median-of-medians to use as a pivot. We then partition each run using binary search.

We can borrow much of the analysis done in [KMMS05]. We cannot use their work wholesale, because we don't know $\mathcal{B}$ in advance. For this reason, we cannot define ℓ as they have, and their algorithm depends heavily on its use. To finish the proof of our theorem, we show how to modify their techniques to handle this complication.

4.2 Merging

Kaligosi et al. [KMMS05, Lemmas 5—10] count the comparisons resulting from merging. Lemmas 5, 6, and 7 do not depend on the value of ℓ and so we can use them in our analysis. Lemma 8 shows that the median-of-medians built on runs is a good pivot selection method. Although the proof clearly uses the value of ℓ, its validity does not depend on how large ℓ is; only that there are at least $4\ell^2$ items in the interval, which also holds for our algorithm. Lemmas 9 and 10 together will bound the number of comparisons by $\mathcal{B}(S_q)(1+o(1))+O(n)$ if we can prove Lemma 4, which bounds the information content of runs in intervals that are not yet partitioned.

Lemma 4. *Let a run r be a sorted sequence of elements from `A` in a gap $\Delta_i^{P_t}$, where $|r|$ is its length. Then,*

$$\sum_{i=0}^{k} \sum_{r \in \Delta_i^{P_t}} |r| \lg |r| \in o(\mathcal{B}(S_t)) + O(n).$$

Proof. In a gap of size Δ, $\ell \in O(\log d)$ where d the recursion depth of the elements in the gap. This gives $\sum_{r\in\Delta} |r| \log |r| \le \Delta \log(2l) \in O(\Delta \log\log d)$, since each run has size at most 2ℓ. Because we use a good pivot selection method, we know that the recursion depth of every element in the gap is $O(\log(n/\Delta))$. Thus, $\sum_{i=0}^{k} \sum_{r\in\Delta_i^{P_t}} |r| \log |r| \le$

$\sum_i \Delta_i \log\log\log(n/\Delta_i)$. Recall that $\mathcal{B}(S_t) = \mathcal{B}(P_t) + O(n) \subset \sum_i \Delta_i \log(n/\Delta_i) + O(n)$. Using Fact 1, the proof is complete. □

4.3 Pivot Finding and Partitioning

Now we prove that the cost of computing medians and performing partition requires at most $o(\mathcal{B}(S_q))+O(n)$ comparisons. The algorithm computes the median m of medians of each run at a node v in the pivot tree T. Then, it partitions each run based on m. We bound the number of comparisons at each node v with more than $4\ell^2$ elements in Lemmas 5 and 6. We bound the comparison cost for all nodes with fewer elements in Lemma 7.

Terminology. Let d be the current depth of the pivot tree T (defined in Section 2.1), and let the root of T have depth $d = 0$. In tree T, each node v is associated with some interval $I(p_v)$ corresponding to some pivot p_v. We define $\Delta_v = |I(p_v)|$ as the number of elements at node v in T.

Recall that $\ell = 1 + \lfloor \log(d+1) \rfloor$. Recall that a *run* is a sorted sequence of elements from `A`. We define a *short run* as a run of length less than ℓ. Let βn be the number of comparisons required to compute the exact median for n elements, where β is a constant less than three [DZ99]. Let r_v^s be the number of short runs at node v, and let r_v^l be the number of long runs.

Lemma 5. *The number of comparisons required to find the median m of medians and partition all runs at m for any node v in the pivot tree T is at most $\beta(\ell-1) + \ell\log\ell + \beta(\Delta_v/\ell) + (\Delta_v/\ell)\log(2\ell)$.*

Proof. We compute the cost (in comparisons) for computing the median of medians. For the $r_v^s \le \ell - 1$ short runs, we need at most $\beta(\ell-1)$ comparisons per node. For the $r_v^l \le \Delta_v/\ell$ long runs, we need at most $\beta(\Delta_v/\ell)$.

Now we compute the cost for partitioning each run based on m. We perform binary search on each run. For short runs, this requires at most $\sum_{i=1}^{\ell-1} \log i \le \ell \log \ell$ comparisons per node. For long runs, we need at most $(\Delta_v/\ell)\log(2\ell)$ comparisons per node. □

Since our value of ℓ changes at each level of the recursion tree, we will sum the above costs by level. The overall cost in comparisons at level d is at most

$$2^d \beta\ell + 2^d \ell \log \ell + (n/\ell)\beta + (n/\ell)\log(2\ell).$$

We can now prove the following lemma.

Lemma 6. *The number of comparisons required to find the median of medians and partition over all nodes v in the pivot tree T with at least $4\ell^2$ elements is within $o(\mathcal{B}(S_t)) + O(n)$.*

Proof. For all levels of the pivot tree up to level $\ell' \le \log(\mathcal{B}(P_t)/n)$, the cost is at most

$$\sum_{d=1}^{\log(\mathcal{B}(P_t)/n)} 2^d \ell(\beta + \log\ell) + (n/\ell)(\beta + \log(2\ell)).$$

Since $\ell = \lfloor \log(d+1) \rfloor + 1$, the first term of the summation is bounded by $(\mathcal{B}(P_t)/n)\log\log(\mathcal{B}(P_t)/n) = o(\mathcal{B}(P_t))$. The second term can be easily upper-bounded by

$$n\log(\mathcal{B}(P_t)/n)(\log\log\log(\mathcal{B}(P_t)/n)/\log\log(\mathcal{B}(P_t)/n))$$

which is $o(\mathcal{B}(P_t))$. Using Lemma 1, the above two bounds are $o(\mathcal{B}(S_t)) + O(n)$.

For each level ℓ' with $\log(\mathcal{B}(P_t)/n) < \ell' \le \log\log n + O(1)$, we need to bound the remaining cost. It is easy to bound each node v's cost by $o(\Delta_v)$, but this is not sufficient—though we have shown that the total number of *comparisons* for merging is $\mathcal{B}(S_t) + O(n)$, the number of *elements* in nodes with $\Delta_v \ge 4\ell^2$ could be $\omega(\mathcal{B}(S_t))$.

We bound the overall cost as follows, using the result of Lemma 5. Since node v has $\Delta_v > 4\ell^2$ elements, we can rewrite the bounds as $O(\Delta_v/\ell\log(2\ell))$. Recall that $\ell \in \log d + O(1) \subset \log(O(\log(n/\Delta_v))) \subset \log\log(n/\Delta_v) + O(1)$, since we use a good pivot selection method. Summing over all nodes, we get $\sum_v (\Delta_v/\ell)\log(2\ell) \le \sum_v \Delta_v \log(2\ell) \in o(\mathcal{B}(P_t)) + O(n)$, using Fact 1 and recalling that $\mathcal{B}(P_t) = \sum_v \Delta_v \log(n/\Delta_v)$. Finally, using Lemma 1, we arrive at the claimed bound for queries. □

Now we show that the comparison cost for all nodes v where $\Delta_v \le 4\ell^2$ is at most $o(\mathcal{B}(S_t)) + O(n)$.

Lemma 7. *For nodes v in the pivot tree T where $\Delta_v \le 4\ell^2$, the total cost in comparisons for all operations is at most $o(\mathcal{B}(S_t)) + O(n)$.*

Proof. We observe that nodes with no more than $4\ell^2$ elements do not incur any cost in comparisons for median finding and partitioning, unless there is (at least) one associated query within the node. Hence, we focus on nodes with at least one query.

Let $z = (\log\log n)^2 \log\log\log n + O(1)$. We sort the elements of any node v with $\Delta_v \le 4\ell^2$ elements using $O(z)$ comparisons, since $\ell \in \log\log n + O(1)$. We set each element as a pivot. The total comparison cost over all such nodes is no more than $O(tz)$, where t is the number of queries we have answered so far. If $t < n/z$, then the above cost is $O(n)$.

Otherwise, $t \geq n/z$. Then, we know that $\mathcal{B}(P_t) \geq (n/z)\log(n/z)$, by Jensen's inequality. (In words, this represents the sort cost of n/z adjacent queries.) Thus, $tz \in o(\mathcal{B}(P_t))$. Using Lemma 1, we know that $\mathcal{B}(P_t) \in \mathcal{B}(S_t) + O(n)$, thus proving the lemma. □

5 Optimal Online Dynamic Multiselection

In this section, we extend our results for the case of the static array by allowing insertions and deletions in the array, while supporting the selection queries. We are originally given the unsorted list $\mathtt{A}$. To support *insert* and *delete* efficiently, we maintain newly-inserted elements in a separate data structure, and mark deleted elements in $\mathtt{A}$. These *insert* and *delete* operations are occasionally merged to make the array $\mathtt{A}$ up-to-date. Let $\mathtt{A}'$ denote the current array with length n'. We support two additional operations:

- *insert*(a), which inserts a into $\mathtt{A}'$, and;
- *delete*(i), which deletes the ith sorted entry from $\mathtt{A}'$.

5.1 Preliminaries

Our solution uses the *dynamic bitvector* of Hon et al. [HSS03]. This structure supports the following operations on a dynamic bitvector $\mathtt{V}$. The $rank_b(i)$ operation tells the number of b bits up to the ith position in $\mathtt{V}$. The $select_b(i)$ operation gives the position in $\mathtt{V}$ of the ith b bit. The $insert_b(i)$ operation inserts bit b in the ith position. The *delete*(i) operation deletes the bit in the ith position. The *flip*(i) operation flips the bit in the ith position.

Note that one can determine the ith bit of $\mathtt{V}$ by computing $rank_{\mathbf{1}}(i) - rank_{\mathbf{1}}(i-1)$. (For convenience, we assume that $rank_b(-1) = 0$.) The result of Hon et al. [HSS03, Theorem 1] can be re-stated as follows, for the case of maintaining a dynamic bit vector (the result of [HSS03] is stated for a more general case).

Lemma 8 ([HSS03])**.** *Given a bitvector $\mathtt{V}$ of length n, there exists a data structure that takes $n+o(n)$ bits and supports $rank_b$ and $select_b$ in $O(\log_t n)$ time, and insert, delete and flip in $O(t)$ time, for any t where $(\log n)^{O(1)} \leq t \leq n$. This structure assumes access to a precomputed table of size n^ϵ, for any fixed $\epsilon > 0$.*

The elements in the array $\mathtt{A}$ swapped during the queries and *insert* and *delete* operations, to create new pivots, and the positions of these pivots are maintained as before using the bitvector $\mathtt{V}$. In addition, we also maintain two bitvectors, each of length n': (i) an *insert bitvector* $\mathtt{I}$ such that $\mathtt{I}[i] = \mathbf{1}$ if and only if $\mathtt{A}'[i]$ is newly inserted, and (ii) a *delete bitvector* $\mathtt{D}$ such that if $\mathtt{D}[i] = \mathbf{1}$, the ith element in $\mathtt{A}$ has been deleted. If a newly inserted item is deleted, it is removed from $\mathtt{I}$ directly. Both $\mathtt{I}$ and $\mathtt{D}$ are implemented as instances of the data structure of Lemma 8.

We maintain the values of the newly inserted elements in a balanced binary search tree T. The inorder traversal of the nodes of T corresponds to the increasing order of their positions in $\mathtt{A}'$. We support the following operations on this tree: (i) given an index i, return the element corresponding to the ith node in the inorder traversal of T, and (ii) insert/delete an element at a given inorder position. By maintaining the subtree sizes of the nodes in T, these operations can be performed in $O(\log n)$ time without having to perform any comparisons between the elements.

Our preprocessing steps are the same as in the static case. In addition, bitvectors $\mathtt{I}$ and $\mathtt{D}$ are each initialized to n $\mathbf{0}$s. The tree T is initially empty.

After performing $|\mathtt{A}|$ *insert* and *delete* operations, we merge all the elements in T with the array $\mathtt{A}$, modify the bitvector $\mathtt{B}$ appropriately, and reset the bitvectors $\mathtt{I}$ and $\mathtt{D}$ (with all zeroes). This increases the amortized cost of the *insert* and *delete* operations by $O(1)$, without requiring additional comparisons.

5.2 Dynamic Online Multiselection

We now describe how to support $\mathtt{A}'.insert(a)$, $\mathtt{A}'.delete(i)$, $\mathtt{A}'.select(i)$, and $\mathtt{A}'.search(a)$ operations.

$\mathtt{A}'.insert(a)$. First, we search for the appropriate unsorted interval $[\ell, r]$ containing a using a binary search on the original (unsorted) array $\mathtt{A}$. Now perform $\mathtt{A}.search(a)$ on interval $[\ell, r]$ (choosing which subinterval to expand based on the insertion key a) until a's exact position j in $\mathtt{A}$ is determined. The original array $\mathtt{A}$ must have chosen as pivots the elements immediately to its left and right (positions $j-1$ and j in array $\mathtt{A}$); hence, one never needs to consider newly-inserted pivots when choosing subintervals. Insert a in sorted order in T among at position $\mathtt{I}.select_{\mathbf{1}}(j)$ among all the newly-inserted elements. Calculate $j' = \mathtt{I}.select_{\mathbf{0}}(j)$, and set a's position to $j'' = j' - \mathtt{D}.rank_{\mathbf{1}}(j')$. Finally, we update our bitvectors by performing $\mathtt{I}.insert_{\mathbf{1}}(j'')$ and $\mathtt{D}.insert_{\mathbf{0}}(j'')$. Note that, apart from the *search* operation, all other operations in the insertion procedure do not perform any comparisons between the elements.

$\mathtt{A}'.delete(i)$. Compute $i' = \mathtt{D}.select_{\mathbf{0}}(i)$. If i' is newly-inserted (i.e., $\mathtt{I}[i'] = \mathbf{1}$), then remove the node (element) with inorder number $\mathtt{I}.rank_{\mathbf{1}}(i')$ from T.

Perform $\mathtt{I}.delete(i')$ and $\mathtt{D}.delete(i')$. If instead i' is an older entry, perform $\mathtt{D}.\mathit{flip}(i')$. In other words, we mark position i' in $\mathtt{A}$ as deleted even though the corresponding element may not be in its proper place.[5]

$\mathtt{A}'.select(i)$. If $\mathtt{I}[i] = \mathbf{1}$, return the element corresponding to the node with inorder number $\mathtt{I}.rank_{\mathbf{1}}(i)$ in T. Otherwise, compute $i' = \mathtt{I}.rank_{\mathbf{0}}(i) - \mathtt{D}.rank_{\mathbf{1}}(i)$, and return $\mathtt{A}.select(i')$.

$\mathtt{A}'.search(a)$. Search for the unsorted interval $[\ell, r]$ containing a using a binary search on the original (unsorted) array $\mathtt{A}$. Then perform $\mathtt{A}.search(a)$ on interval $[\ell, r]$ until a's exact position j is found. If a appears in $\mathtt{A}$ (which we discover through *search*), we need to check whether it has been deleted. We compute $j' = \mathtt{I}.select_{\mathbf{0}}(j)$ and $j'' = j' - \mathtt{D}.rank_{\mathbf{1}}(j')$. If $\mathtt{D}[j'] = \mathbf{0}$, return j''. Otherwise, it is possible that the item has been newly-inserted. Compute $p = \mathtt{I}.rank_{\mathbf{1}}(j')$, which is the number of newly-inserted elements that are less than or equal to a. If $T[p] = a$, then return j''; otherwise, return failure.

We show that the above algorithm achieves the following performance.

Theorem 4 (Online Dynamic Multiselection)**.** *Given a dynamic array $\mathtt{A}'$ of n original elements, there exists a dynamic online data structure that can support $q \in O(n)$ select, search, insert, and delete operations, of which q' are search, insert, and delete, we provide a deterministic online algorithm that uses at most $\mathcal{B}(S_q)(1+o(1))+O(n+q'\log n)$ comparisons.*

Proof. Let $\mathtt{A}'$ denote the current array of length n', after a sequence of queries and insertions. Let Q be the sequence of q selection operations performed (either directly or indirectly through other operations) on $\mathtt{A}'$, ordered by time of arrival. Let S_q be the queries of Q, ordered by position. We now analyze the number of comparisons performed by a sequence of queries and *insert* and *delete* operations.

We consider the case when the number of *insert* and *delete* operations is less than n. In other words, we are between two rebuildings of our dynamic data structure. If q' is the number of *search*, *insert*, and *delete* operations in the sequence, then we perform $O(q'\log n')$ comparisons to perform the required searches. Note that our algorithm does not perform any comparisons for $delete(i)$ operations, until some other query is in the same interval as i. The deleted element will participate in the other costs (merging, pivot-finding, and partitioning) for these other queries, but its contribution can be bounded by $O(\log n)$, which we have as a credit.

Since a *delete* operation does not perform any additional comparisons beyond those needed to perform a *search*, we assume that all the updates are insertions in the rest of this section. Since each inserted element becomes a pivot immediately, it does not contribute to the comparison cost of any other *select* operation. Also, note that in the algorithm of Theorem 3, no pivot is part of a run and hence cannot effect the choice of any future pivot.

Since Q is essentially a set of q selection queries, we can bound its total comparison cost for selection queries by Theorem 3, which gives a bound of $\mathcal{B}(S_q)(1+o(1))+O(n)$. This proves the theorem. □

6 External Multiselection

Suppose we are given an unsorted array $\mathtt{A}$ of length N stored in $n = N/B$ blocks in the external memory. Sorting $\mathtt{A}$ in the external memory model requires $\Theta(n\log_m n)$ I/Os. The techniques we use in internal memory are not immediately applicable to the external memory model. In the extreme case where we have $q = N$ queries, the internal memory solution would require $O(n\log_2(n/m))$ I/Os. This compares poorly to the optimal $O(n\log_m n)$ I/Os performed by the external mergesort algorithm.

6.1 A Lower Bound for Multiselect in External Memory

As in the case of internal memory, the lower bound on the number of I/Os required to perform a given set of selection queries can be obtained by subtracting the number of I/Os required to sort the elements between the 'query gaps' from the sorting bound. More specifically, let $S_t = \{s_i\}$ be the first t queries from a query set Q, sorted by position, and for $1 \le i \le t$, let $\Delta_i^{S_t} := s_{i+1} - s_i$ be the query gaps, as defined in Section 2.1. Then the lower bound on the number of I/Os required to support the queries in S_t is given by $\mathcal{B}_m(S_t) \in n\log_m n - \sum_{i=0}^{t}(\Delta_i^{S_t}/B)\log_m(\Delta_i^{S_t}/B) - O(n)$, where we assume that $\log_m\left(\Delta_i^{S_t}/B\right) = 0$ when $\Delta_i^{S_t} < mB = M$ in the above definition. Note that $\mathcal{B}_m(S_t) \in \Omega(n)$ for all $t \ge 1$.

6.2 Partitioning in External Memory

The main difference between our algorithms for internal and external memory is the partitioning procedure. In the internal memory algorithm, we parti-

[5] If a user wants to delete an item with value a, one could simply search for it first to discover its rank, and then delete it using this function.

tion the values according to a single pivot, recursing on the half that contains the answer. In the external memory algorithm, we modify this binary partition to a d-way partition, for some $d \in \Theta(m)$, by finding a sample of d "roughly equidistant elements." The next lemma describe how to find such a sample, and then partition the range of values into $d+1$ subranges with respect to the sample.

As in [AV88], we assume that $B \in \Omega(\log_m n)$—which allows us to store a pointer to a memory block of the input using a constant number of blocks. This is similar to the word-size assumption for the trans-dichotomous word RAM model [FW93]. In addition, the algorithm of Sibeyn [Sib06] only works under this assumption, though this is not explicitly mentioned.

Lemma 9. *Given an unsorted array* `A` *containing N elements in external memory and an integer parameter $d < m/2$, one can perform a d-way partition in $O(n+d)$ I/Os, such that the size of each partition is in the range $[n/(2d), 3n/(2d)]$.*

Proof. Let $s = \lfloor\sqrt{m/4}\rfloor$. We perform the s-way partition described in [AV88] to obtain $s+1$ super-partitions. We reapply the s-way partitioning method to each super-partition to obtain $d < m/2$ partitions in total.

Finally, our algorithm scans the data, keeping one input block and $d+1$ output blocks in main memory. An output block is written to external memory when it is full, or when the scan is complete. The algorithm performs n I/O to read the input, and at most $(n + d + 1)$ I/Os to write the output into $d+1$ partitions, thus showing the result. □

6.3 Achieving $O(\mathcal{B}_m(S_q))$ I/Os

We now show that our lower bound is asymptotically tight, by describing an $O(1)$-competitive algorithm.

Theorem 5. *Given an unsorted array* `A` *occupying n blocks in external memory, we provide a deterministic algorithm that supports a sequence Q of q online selection queries using $O(\mathcal{B}_m(S_q)))$ I/Os under the condition that $B \in \Omega(\log_m n)$.*

Proof. Our algorithm uses the same approach as the internal memory algorithm, except that it chooses $d-1$ pivots at once. In other words, each node v of the pivot tree T containing Δ_v elements has a branching factor of d. We subdivide its Δ_v elements into d partitions using Lemma 9. This requires $O(\delta_v + d)$ I/Os, where $\delta_v = \Delta_v/B$.

We also maintain the bitvector `V` of length N, as described before. For each `A`.*select*(i) query, we access position `V`$[i]$. If `V`$[i] = \mathbf{1}$, return `A`$[i]$, else scan left and right from the ith position to find the endpoints of this interval I_i using $|I_i|/B$ I/Os. The analysis of the remaining terms follows directly from the internal memory algorithm, giving $O(\mathcal{B}_m(S_q)) + O(n) = O(\mathcal{B}_m(S_q))$ I/Os. □

To add search, instead of taking $O(\log N)$ time performing binary search on the blocks of `V`, we build a B-tree T maintaining all pivots from `A`. (During preprocessing, we insert `A`$[1]$ and `A`$[n]$ into T.) The B-tree T will be used to support *search* queries in $O(\log_B N)$ I/Os instead of $O(\log N)$ I/Os. We modify the proof of Theorem 5 to obtain the following:

Corollary 1. *Given an unsorted array* `A` *occupying n blocks in external memory, we provide a deterministic algorithm that supports a sequence Q of q online selection and search queries using $O(\mathcal{B}_m(S_q) + q\log_B N)$ I/Os under the condition that $B \in \Omega(\log_m n)$.*

Proof. The first two terms follow directly from the proof of Theorem 5. Now we explain the source of the last term.

We build a B-tree T maintaining all pivots from `A`. (During preprocessing, we insert `A`$[1]$ and `A`$[n]$ into T.) Naively, for q queries, we must insert $qm\log_m N$ new pivots into T. The B-tree construction for these pivots would require $O(\min\{qm(\log_m N), N\}(\log_B N))$ I/Os, which is prohibitive.

Instead, we notice that the pivots for an individual query z are all inserted in some unsorted interval $I_z = [l, r]$, where l and r are consecutive leaves of the pivot tree T (in left-to-right level order). For z, we may spend $\log_B(\min\{qm(\log_m N), N\}) \in O(\log_B N)$ I/Os navigating to I_z using T. Our approach is to insert all $O(m\log_m N) \in O((M/B)\log_m N) = O(M)$ pivots within I_z in a single batched manner. This process can easily be done in a bottom-up fashion by merging nodes in the tree T of an implicit B-tree T' for the $O(M)$ pivots using $O(m)$ I/Os.

Thus, we have $O(\min\{qm\log_m N, N\}$ pivots in T, and using the batched insertion process above, we can do this using only $O(\min\{qm(\log_m N)/B, N/B\}) = O(\min\{qm, n\})$ I/Os. We must also add $O(q\log_B N)$ I/Os to navigate to the correct interval for each query.

Overall, for q queries, the algorithm takes $O(\mathcal{B}_m(S_q)) + O(n) + O(q\log_B N) = O(\mathcal{B}_m(S_q) + q\log_B N)$ I/Os, matching the result. □

Note that if $q \in O(\mathcal{B}_m(S_q)/\log_B N)$, then Corollary 1 requires only $O(\mathcal{B}_m(S_q))$ I/Os, matching the bounds from Theorem 5. Hence, our result is asymptotically optimal when $\mathcal{B}_m(S_q)/q = \log_B N$.

References

[AV88] Alok Aggarwal and Jeffrey Scott Vitter. The input/output complexity of sorting and related problems. *Commun. ACM*, 31(9):1116–1127, 1988.

[BF03] G. Brodal and R. Fagerberg. On the limits of cache-obliviousness. In *Proceedings of the ACM Symposium on Theory of Computing*, pages –, 2003.

[BFP+73] Manuel Blum, Robert W. Floyd, Vaughan R. Pratt, Ronald L. Rivest, and Robert Endre Tarjan. Time bounds for selection. *J. Comput. Syst. Sci.*, 7(4):448–461, 1973.

[CFJ+09] Jean Cardinal, Samuel Fiorini, Gwenaël Joret, Raphaël M. Jungers, and J. Ian Munro. An efficient algorithm for partial order production. In *Proceedings of the 41st annual ACM symposium on Theory of computing*, STOC '09, pages 93–100, New York, NY, USA, 2009. ACM.

[DM81] David P. Dobkin and J. Ian Munro. Optimal time minimal space selection algorithms. *J. ACM*, 28(3):454–461, 1981.

[DZ99] Dorit Dor and Uri Zwick. Selecting the median. *SIAM J. Comput.*, 28(5):1722–1758, 1999.

[FW93] Michael L. Fredman and Dan E. Willard. Surpassing the information theoretic bound with fusion trees. *J. Comput. Syst. Sci.*, 47(3):424–436, 1993.

[Hoa61] C. A. R. Hoare. Algorithm 65: find. *Commun. ACM*, 4(7):321–322, 1961.

[HSS03] Wing-Kai Hon, Kunihiko Sadakane, and Wing-Kin Sung. Succinct data structures for searchable partial sums. In *Proceedings of the International Symposium on Algorithms and Computation*, pages 505–516, 2003.

[JM10] Rosa M. Jiménez and Conrado Martínez. Interval sorting. In *Proceedings of the International Colloquium on Automata, Languages, and Programming*, pages 238–249, 2010.

[KMMS05] Kanela Kaligosi, Kurt Mehlhorn, J. Ian Munro, and Peter Sanders. Towards optimal multiple selection. In *ICALP*, pages 103–114, 2005.

[LPV03] L. Lovász, J. Pelikán, and K. Vesztergombi. *Discrete Mathematics: Elementary and Beyond*. Springer-Verlag, 2003.

[Pro95] Helmut Prodinger. Multiple quickselect - Hoare's find algorithm for several elements. *Inf. Process. Lett.*, 56(3):123–129, 1995.

[Sib06] Jop F. Sibeyn. External selection. *J. Algorithms*, 58(2):104–117, 2006.

[SPP76] Arnold Schönhage, Mike Paterson, and Nicholas Pippenger. Finding the median. *J. Comput. Syst. Sci.*, 13(2):184–199, 1976.