Journal of Discrete Algorithms 36 (2016) 3–17

Contents lists available at ScienceDirect

Journal of Discrete Algorithms

www.elsevier.com/locate/jda

Near-optimal online multiselection in internal and external memory [☆]

Jérémy Barbay [a,1], Ankur Gupta [b,2], Srinivasa Rao Satti [c,*,3], Jon Sorenson [b]

[a] *Departamento de Ciencias de la Computación (DCC), Universidad de Chile, Chile*
[b] *Department of Computer Science and Software Engineering, Butler University, United States*
[c] *School of Computer Science and Engineering, Seoul National University, Republic of Korea*

A R T I C L E   I N F O

Article history:
Available online 10 November 2015

Keywords:
Selection
Sorting
Multiselection
Online algorithms
Deferred data structures
Dynamic
External memory

A B S T R A C T

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 the offline algorithm proposed by Kaligosi et al. [14] 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, 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 [20] when $q = \omega(m^{1-\epsilon})$ for any fixed positive constant ϵ, 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 elements of rank r_i from the sequence $R = \{r_1, r_2, \ldots, r_q\}$ on an unsorted array $\mathbb{A}$ of size n 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 selection queries, where $S_q = \{s_1, s_2, \ldots, s_q\}$ denotes the queries ordered by rank. This lower bound can be obtained by taking the number of comparisons needed to sort the entire array, and then subtracting the comparisons needed to sort the query gaps. (Please see Section 2.2 for more details on this bound.) The *online multiselection* problem asks for elements of rank r_i, where the sequence R is given one element at a time. The lower bound $\mathcal{B}(S_q)$ also applies to *search* queries in the offline model, as well as to both types of queries in the online model.

[☆] A preliminary version of these results have appeared in the proceedings of ESA-2013 [2] and WALCOM-2015 [3].
[*] Corresponding author.
E-mail addresses: jeremy@barbay.cl (J. Barbay), agupta@butler.edu (A. Gupta), ssrao@cse.snu.ac.kr (S.R. Satti), jsorenso@butler.edu (J. Sorenson).
[1] Supported in part by the Millennium Nucleus "Information and Coordination in Networks" ICM/FIC RC130003.
[2] Supported in part by the Areté Initiative at the University of Chicago and the Butler Holcomb Awards grant.
[3] Supported by the 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).

http://dx.doi.org/10.1016/j.jda.2015.11.001
1570-8667/© 2015 Elsevier B.V. All rights reserved.

The *dynamic online multiselection* problem supports *select*, *search*, *insert* and *delete* operations, described below:

- *select*(s), returns the position of the sth smallest element in $\mathtt{A}$;
- *search*(p), returns the position of the largest element $y \le p$ from $\mathtt{A}$;
- *insert*(p), inserts p into $\mathtt{A}$; and
- *delete*(s), deletes the sth smallest element from $\mathtt{A}$.

1.1. Previous work

Offline Multiselection. Several papers have analyzed the offline multiselection problem, but all of these approaches required the queries to be known in advance. Dobkin and Munro [8] gave a deterministic algorithm for this problem using $3\mathcal{B}(S_q) + O(n)$ comparisons. Prodinger [18] proved the expected comparisons with random pivot selection is at most $2\mathcal{B}(S_q)\ln 2 + O(n)$. More recently, Kaligosi et al. [14] described a randomized algorithm that uses $\mathcal{B}(S_q) + O(n)$ expected comparisons, along with a deterministic algorithm that performs $\mathcal{B}(S_q) + o(\mathcal{B}(S_q) + O(n)$ comparisons. Jiménez and Martínez [13] later improved the bound on the number of comparisons in the expected case to $\mathcal{B}(S_q) + n + o(n)$, when $q = o(\sqrt{n})$. Cardinal et al. [7] generalized the problem to *partial order production* (of which multiselection is a special case), and they used multiselection as a subroutine after an initial preprocessing phase.

In the external memory model [1] with internal memory M and block size 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 [20] solved external multiselection using $n + nq/m^{1-\epsilon}$ I/Os, where ϵ is any fixed positive constant. The first term comes from creating a static index structure using n I/Os, and the remaining $nq/m^{1-\epsilon}$ comes from answering q searches using that index. In addition, this result requires the condition that $B = \Omega(\log_m n)$.[4] When $q = m$, Sibeyn's multiselection algorithm takes $O(nm^{\epsilon})$ I/Os, whereas the optimum is $\Theta(n)$ I/Os. In fact, his bounds are $\omega(\mathcal{B}_m(S_q))$, for any $q = \omega(m^{1-\epsilon})$, where $\mathcal{B}_m(S_q)$ is the lower bound on the number of I/Os required. (See Section 6.1 for the definition of $\mathcal{B}_m(S_q)$.)

Online Multiselection. Motwani and Raghavan [17] introduced the static online multiselection problem, where selection and search queries arrive one at a time, as a "Deferred Data Structure" for sorting. (In other words, the input array is sorted over time, as queries are answered.) Barbay et al. [2] described a simpler solution with an improved analysis that matched the offline results of Kaligosi et al. [14]. Ching et al. [21] extended Motwani and Raghavan's solution [17] to support insertion and deletion, with optimal amortized complexity in the worst case over instances with a fixed number q of queries. Our solution is simpler, and our analysis finer, in the worst case over instances where the query positions are fixed. To the best of our knowledge, there are no existing dynamic results for the multiselection problem in the external memory model.

1.2. Our results

For the *dynamic online multiselection* problem in internal memory, we describe the *first* algorithm that supports a sequence R 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.[5] For the *online multiselection* problem (when $q' = 0$), our algorithm is 1-competitive with the offline algorithm of Kaligosi et al. [14] in the number of comparisons performed. In addition, we obtain a randomized result that matches (i.e., is 1-competitive with) the performance of Kaligosi et al. [14], while only using $O((\log n)^{O(1)})$ sampled elements instead of $O(n^{3/4})$ elements.

For the external memory model [1], we describe an *external online multiselection* algorithm on an unsorted array $\mathtt{A}$ of size N, 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 [20] when $q = \omega(m^{1-\epsilon})$ for any fixed positive constant ϵ. We also extend it to support search, insert, and delete operations using $O(\mathcal{B}_m(S_q) + q \log_B N)$ I/Os.

1.3. Preliminaries

Given an unsorted array $\mathtt{A}$ of length n, the *median* of $\mathtt{A}$ is the element x such that $\lceil n/2 \rceil$ elements in $\mathtt{A}$ are no greater than x. It is well-known that the median can be computed in $O(n)$ comparisons, and many [11,5,19] have analyzed the exact constants involved. Dor and Zwick [9] provided the best known constant, yielding $2.942n + o(n)$ comparisons.

With a linear time median-finding algorithm, one can obtain a linear time algorithm to select the element of a specified rank r in $\mathtt{A}$. Dobkin and Munro [8] considered the extension of this selection problem to the multiselection problem, and gave an algorithm that requires an asymptotically optimal number of comparisons. As mentioned earlier, Kaligosi et al. [14]

[4] 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.

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

reduced the number of comparisons to $\mathcal{B} + o(\mathcal{B}) + O(n)$, where $\mathcal{B}$ is a lower bound on the number of comparisons required to support the queries on an unsorted array of length n.

Since our online algorithm (presented in Section 4) is a modification of the deterministic algorithm from Kaligosi et al. [14], we briefly describe it here. They begin by creating *runs*, which are sorted sequences from $\mathtt{A}$ of length roughly $\ell = \log(\mathcal{B}/n)$. More precisely, there are at most ℓ runs of size less than ℓ, no two of which share the same length. All the remaining runs are of length between ℓ and 2ℓ. Then, they compute the median m of the medians of these runs and partition them based on m. After partitioning, they recurse on the two sets of runs, sending *select* queries to the appropriate side of the recursion. In each recursive subproblem, they merge short runs of the same size optimally until all but ℓ of the runs are again of length between ℓ and 2ℓ.

External Memory Model. In the external memory model [1], 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 $1 \le B \le M/2$, or in words, internal memory consists of at least two blocks. In this model, we only measure the number of blocks transferred between disk and internal memory. The *cache-oblivious model* measures performance the same way, but the algorithm has no knowledge of M or B.

To achieve the optimal sorting bound of $\mathit{SortIO}(N) = \Theta(n \log_m n)$ in this setting, it is necessary to make the *tall cache* assumption [6]: $M = \Omega(B^{1+\epsilon})$, for some constant $\epsilon > 0$, and we will make this assumption for the remainder of this article. In keeping with the spirit of the external memory model, our external memory results do not include the RAM complexity; typically, such algorithms have undesirable bounds. For a better RAM complexity, we invite the reader to adapt the results from Sections 2–4.

1.4. Outline

In the next section, we present a simple algorithm for the online multiselection problem, and introduce some terminology to describe its analysis. In Section 3, we show that the simple algorithm has a constant competitive ratio. Section 4 describes modifications to the simple algorithm, and shows that the modified algorithm is optimal up to lower order terms. Our main results are contained in this section, and they form the foundation of the remaining results in the paper. Section 5 extends these results to support online updates to array $\mathtt{A}$. Finally, in Section 6, we describe how to adapt our optimal algorithm to perform well in the external memory model.

2. A simple online algorithm

In this section, we will describe a simple version of our online algorithm that supports selection and search queries on an array $\mathtt{A}$ of n (unsorted) elements. For simplicity, in this article we assume that all the elements in $\mathtt{A}$ are distinct. (Standard techniques can be used to remove this restriction.) We will refer to an element in position i from $\mathtt{A}$ as a *pivot* if $\mathtt{A}[1 \dots i-1] < \mathtt{A}[i] < \mathtt{A}[i+1 \dots n]$. Usually, one must rearrange the array $\mathtt{A}$ to satisfy the previous restriction. We refer to such a rearrangement as *partitioning*. Throughout the article, we will explore various pivot selection and partitioning techniques. Hence, the algorithms presented in this section do not specify how this partitioning is done. We will address its implementation as we need it.

2.1. Algorithm description and correctness

Auxiliary Data Structures and Preprocessing. Throughout all the algorithms in this article, we will create and maintain a bitvector $\mathtt{V}$ of length n, where $\mathtt{V}[i] = \mathbf{1}$ if and only if it is a pivot, and $\mathtt{V}[i] = \mathbf{0}$ otherwise. We will initialize $\mathtt{V}$ and set each bit to $\mathbf{0}$. Finally, we will find the minimum and maximum elements in array $\mathtt{A}$, swap them into $\mathtt{A}[1]$ and $\mathtt{A}[n]$ respectively, and set $\mathtt{V}[1] = \mathtt{V}[n] = \mathbf{1}$. Both the search and select algorithms use a subroutine called *partition*, which will change throughout the article. In the simple algorithm described below, *partition* will use the last element in an interval as the pivot, and will partition accordingly.

Selection. The operation $\mathtt{A}.\mathit{select}(s)$ returns the sth smallest element from $\mathtt{A}$. (In other words, it returns $\mathtt{A}[s]$ as if $\mathtt{A}$ were sorted.) The pseudocode for $\mathtt{A}.\mathit{select}(s)$ is given in Algorithm 1, but we briefly describe it here.

If $\mathtt{V}[s] = \mathbf{1}$, return $\mathtt{A}[s]$, and we are done. Otherwise, find the smallest unsorted interval containing the sth element. (The smallest unsorted interval is $\mathtt{A}[a+1 \dots b-1]$, where $\mathtt{V}[a] = \mathtt{V}[b] = \mathbf{1}$ but $\mathtt{V}[a+1 \dots b-1]$ are all $\mathbf{0}$.) Finally, we use a variant of the quickselect [11] algorithm (Algorithm 2) on $\mathtt{A}[a+1 \dots b-1]$, marking pivots found along the way in $\mathtt{V}$.

Algorithm 1: A.*select*(s)

```
Input: array A and bitvector V of size $n$; $1 \le s \le n$
Output: $s$th element (in sorted order) from A
  if V[$s$] == 1 then return A[$s$];
  // Interval boundaries of $s$th
     element.
  $a = b = s$;
  while V[$a$] $\neq$ 1 do $a = a - 1$;
  while V[$b$] $\neq$ 1 do $b = b + 1$;
  $s$ = A.qselect($a + 1, b - 1, s$);
return A[$s$];
```

Algorithm 2: A.*qselect*(l, r, s)

```
Input: array A and bitvector V of size $n$;
       $1 \le l, r, s \le n$
Output: $s$th element (in sorted order) from A
  if $l \ge r$ then V[$s$] = 1; return $s$;
  // A.partition partitions using a
  // pivot, and returns its position.
  $pivotpos$ = A.partition($l, r$);
  V[$pivotpos$] = 1;
if $s == pivotpos$ then return $pivotpos$;
else if $s < pivotpos$ then
    return A.qselect($l, pivotpos - 1, s$);
else return A.qselect($pivotpos + 1, r, s$);
```

Search. The operation A.*search*(p) returns the number of elements from A with value $\le p$.[6] The pseudocode for A.*search*(p) is given in Algorithm 3, but we describe it briefly here.

We begin by performing a binary search on A *as if* A *were sorted.*[7] Let j be the location in A that was found from the search. (If the endpoints of the interval being compared were out of order, stop the search prematurely and let j instead be the midpoint of that interval.) If A[j] $= p$ and V[j] $= \mathbf{1}$, return j and we are done. Otherwise, we have identified the unsorted interval in A that might contain p (if $p \in$ A). Finally, use a variant of the quickselect [11] algorithm (Algorithm 4) on this interval, marking pivots found along the way in V. (We decide which side to recurse on, based on the *value* of p, instead of an array position.) This gives us the value *rank*, where A[*rank*] $= p$, as desired.

Algorithm 3: A.*search*(p)

```
Input: array A and bitvector V of size $n$; query $p$
Output: number of elements from A with value $\le p$
  if $p <$ A[1] then return 0;
  if $p \ge$ A[$n$] then return $n$;
  // Find interval that may contain $p$.
  $a = 2; b = n$;
  while $a + 1 < b$ do
      $mid = (a + b)/2$;
      if $p <$ A[$mid$] then $b = mid - 1$;
      else if $p >$ A[$mid$] then $a = mid + 1$;
      else $a = b = mid$;
  while V[$a$] $\neq$ 1 do $a = a - 1$;
  while V[$b$] $\neq$ 1 do $b = b + 1$;
if $p ==$ A[$a$] then return $a$;
if $p ==$ A[$b$] then return $b$;
return A.qsearch($a + 1, b - 1, p$);
```

Algorithm 4: A.*qsearch*(l, r, p)

```
Input: array A and bitvector V of size $n$;
       $1 \le l, r \le n$; $p$ is the search value
Output: number of elements from A with value $\le p$
  $rank = l$;
  if $l \ge r$ then
      if $p >$ A[$rank$] then $rank = rank + 1$;
      V[$rank$] = 1;
      return $rank$;
  // A.partition partitions using a
  // pivot, and returns its position.
  $pivotpos$ = A.partition($l, r$);
  V[$pivotpos$] = 1;
if $p ==$ A[$pivotpos$] then return $pivotpos$;
else if $p <$ A[$pivotpos$] then
    return A.qsearch($l, pivotpos - 1, p$);
else return A.qsearch($pivotpos + 1, r, p$);
```

Proof of correctness. The selection and search procedures described above maintain the following invariant throughout both the algorithms.

Invariant. For any index a where $1 \le a \le n$, if V[a] $= \mathbf{1}$, then A[a] is a pivot. (In other words, A[a] is in its correct position in sorted order in A if V[a] $= \mathbf{1}$.)

By construction, the invariant holds after the preprocessing step. Also, each time the selection algorithm partitions, it creates a pivot that satisfies the invariant. The correctness of the selection algorithm follows immediately from these observations, since A.*select*(s) only returns when V[s] $= \mathbf{1}$.

[6] 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.

[7] This idea is similar to one described by Biedl et al. [4], in which they also search an unsorted array. However, they do not maintain interval boundaries, which is a crucial aspect of our implementation (and analysis).

To prove the correctness of the search procedure, it is enough to prove the claim that the binary search on the (unsorted) array A always returns a position i that is contained in an interval $[a, b]$ such that $\mathtt{V}[a] = \mathtt{V}[b] = \mathbf{1}$ and $\mathtt{A}[a] \le p \le \mathtt{A}[b]$, and for any $a < c < b$, $\mathtt{V}[c] = \mathbf{0}$. From this index i, the search algorithm scans the bitvector V both ways to find the interval $[a, b]$. The correctness of the remaining part of the algorithm follows along the same lines as the correctness proof for the select algorithm, since the procedure A.*qselect* is essentially the same as A.*qsearch*, only that A.*qselect* recurses based on a position s whereas A.*qsearch* recurses based on the value p.

The claim follows from the following observations: Suppose the search value p belongs to an interval $[a, b]$ such that $\mathtt{V}[a] = \mathtt{V}[b] = \mathbf{1}$ and $\mathtt{A}[a] \le p \le \mathtt{A}[b]$, and for any $a < c < b$, $\mathtt{V}[c] = \mathbf{0}$. Suppose the binary search algorithm probes a location i during the search. Then if $i < a$, the binary search will continue to the right of i since, by the invariant, $A[i] < A[a] \le p$. Similarly, if $i > b$, then the search will continue to the left of i since $p \le A[b] < A[i]$ (again, by the invariant). Finally, when $a \le i \le b$, the search may proceed either to the left or to the right of i. It is possible that the search may probe a location outside the interval $[a, b]$ after probing some locations inside the interval, but it will ultimately terminate inside the interval.

Analysis for a Simple Pivot Method. 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 over time, we will have performed a quicksort on array A, since our recursive subproblems mimic those from quicksort. Furthermore, we have assumed, up to this point, that the last element in an interval is used as the pivot, and a simple linear-time partitioning algorithm is used. In the rest of this article, we will explore different pivot and partitioning strategies, which produce various complexity results for online multiselection (and multisearch). For example, if we employ a linear-time median-finding algorithm, we can easily modify the (selection) algorithm from [17] to achieve the following proposition:

Proposition 1. *The number of comparisons used by the simple online algorithm to perform q select and search queries on an (unsorted) array* A *with n elements is at most* $O(n \log q + q \log n)$.

The proof of Proposition 1 is an easy consequence of a more detailed and precise analysis to follow. Next, we define terminology for our improved analysis.

2.2. Terminology

For now, we assume that all queries are selection queries, since search queries are basically selection queries with a binary search preprocessing phase taking $O(\log n)$ comparisons. In all future results in this article, we will explicitly bound the binary search cost incurred.

Query and Pivot Sets. Let R 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 R, sorted by position. We also include $s_0 = 0$ and $s_{t+1} = n$ in S_t for convenience of notation, since we find the minimum and maximum elements from A 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 $S_t \subseteq P_t$, $p_1 = 1$, $p_k = n$, and $\mathtt{V}[p_i] = \mathbf{1}$ for all $1 \le i \le k$.

Pivot Tree, Recursion Depth, and Intervals. The pivots chosen by the algorithm form a binary tree structure over time, defined as the *pivot tree* T.[8] 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 the "first" interval $\mathtt{A}[2..n-1]$, which was created in the preprocessing phase. 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 T are also selection queries, but it may be the case that a selection query is not a leaf. Each pivot in the pivot tree was used to partition an interval in 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 the interval's corresponding pivot.

Gaps and Entropy. We define the query gap $\Delta_i^{S_t} = s_{i+1} - s_i$ and 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 incurred. The worst-case lower bound to sort an array of n elements is $\log n! = n \log n - n/\ln 2 + (\log n)/2 + O(1)$ comparisons. (See [15, Section 5.3.1].)

We obtain the lower bound $B(S_t)$ on the number of comparisons needed to answer the selection queries in S_t by taking the number of comparisons to sort the entire array, and subtracting the comparisons needed to sort the query gaps. Using Stirling's approximation, we have

8 Intuitively, a pivot tree corresponds to a *recursion tree*, since each node represents one recursive call made during the quickselect algorithm [11].

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

Note that $\mathcal{B}(S_q) \le n\log q$: this upper bound is met when the queries are evenly spaced over the input array A. We can strengthen Proposition 1 to show the improved bound in Proposition 2. As before, Proposition 2 is subsumed by later results, so we do not prove it here.

Proposition 2. *The number of comparisons used by the simple online algorithm to perform a sequence R of q select and search queries on an (unsorted) array A of n elements is at most $O(\mathcal{B}(S_q) + q\log n)$.*

We define the notation ${}^{y}x = x \uparrow\uparrow y = \underbrace{x^{x^{\cdot^{\cdot^{\cdot^{x}}}}}}_{y\ times}$. We also let $\log^{(i)} x = \underbrace{\log\log\log\ldots\log x}_{i\ times}$. Now we can define the following fact, which we will use throughout the article.

Fact 1. *For all $\epsilon > 0$, there exists a constant c_ϵ such that for all integers $i > j > 0$ and $x \ge 2 \uparrow\uparrow (i-1)$, $\log^{(i)} x < \epsilon \log^{(j)} x + c_\epsilon$.*

Proof. It suffices to show that $\log^{(2)} x < \epsilon \log^{(1)} x + c_\epsilon$, since one can utilize the following two observations to produce the claimed result for any $i > j > 0$ and $x \ge 2 \uparrow\uparrow (i-1)$:

1. $\log^{(i+1)} x < \log^{(i)} x$, and
2. $\log^{(i)} x < \epsilon \log^{(j)} x + c_\epsilon \implies \log^{(i+1)} x < \epsilon \log^{(j+1)} x + c_\epsilon$.

Now we prove the result for $\log^{(2)} x < \epsilon \log^{(1)} x + c_\epsilon$. Let $x \ge 2 \uparrow\uparrow 1 = 2$. Since $\lim_{x\to\infty}(\log\log x)/(\log x) = 0$, there exists a $k_\epsilon > 2$ such that for all $x \ge k_\epsilon$, we know that $(\log\log x)/(\log x) < \epsilon$. Also, in the interval $[2, k_\epsilon]$, the continuous function $\log\log x - \epsilon\log x$ is bounded. Choosing $c_\epsilon = \log\log k_\epsilon$, which is a positive constant, finishes the proof. □

3. Analysis of the simple algorithm

In this section, we improve the analysis written in Proposition 2. The final result is stated in Theorem 1 with a fixed constant of 4 in the deterministic case. To achieve the main result of the article, we need a different pivot selection and partitioning method, which we describe in detail in Section 4.

3.1. A lemma on sorting entropy

Pivot Selection Methods. We say that a pivot selection method is *good* for the constant c with $1/2 \le c < 1$ 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

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

For example, 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. [14, Lemma 8] is *good* with $c = 15/16$.

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

Proof. Consider any two consecutive (in terms of rank) 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 $p_r = s'$. Recall that S_t is the set of t queries being processed, and P_t is the set of k pivots generated by processing set S_t. The lemma follows from the claim that $\mathcal{B}(P_\Delta) = O(\Delta)$, since

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

$$= \sum_{i=0}^{t} \left(\Delta_i^{S_t} \log \Delta_i^{S_t} - \sum_{p_j \in P_{\Delta_i^{S_t}}} \Delta_j^{P_t} \log \Delta_j^{P_t} \right)$$
$$= \sum_{i=0}^{t} \mathcal{B}\left(P_{\Delta_i^{S_t}}\right) = \sum_{i=0}^{t} O\left(\Delta_i^{S_t}\right) = O(n).$$

We now proceed to prove our claim.

There must be a unique pivot in P_Δ of minimal recursion depth. Any pair of pivots with the same recursion depth must have a common ancestor, and this ancestor must lie between the pair. Moreover, this ancestor is in P_Δ, and it has smaller recursion depth than the pair. Let p_m denote the pivot of minimum depth. (Note that $p_m = s$ or $p_m = s'$ are possible.) As before, define the gaps $\Delta_i = p_{i+1} - p_i$ for $l \le i < r$. We split the gap Δ at p_m. We address the right side first, and the argument for the left side is similar.

The sequence $d(p_m), d(p_{m+1}), \ldots, d(p_{r-1})$ must be strictly increasing. Otherwise, one of these pivots would be a leaf in the pivot tree, and hence a query, which is a contradiction.

Now consider $I(p_{m+1})$. This interval must have p_m as its left endpoint, due to its smaller recursion depth. Its right endpoint must have recursion depth shallower than p_{m+1}, and hence it contains all pivots up to and including p_r. For $m+1 < i < r$, this means that $I(p_i) \subset I(p_{m+1})$ and $\Delta_i = p_{i+1} - p_i < |I(p_i)|$. Additionally, $\Delta \le |I(p_{m-1})| + |I(p_{m+1})|$.

For brevity, we define $D_l = \sum_{i=l}^{m-1} \Delta_i$ and $D_r = \sum_{i=m}^{r-1} \Delta_i$. Observe that $\Delta = D_l + D_r$, and further, $D_l \le |I(p_{m-1})|$ and $D_r \le |I(p_{m+1})|$. We also define $\alpha_i = D_r/\Delta_i$ for $m \le i < r$. Then we can write

$$D_r \log D_r - \sum_{i=m}^{r-1} \Delta_i \log \Delta_i = \sum_{i=m}^{r-1} \Delta_i \log(D_r/\Delta_i) = D_r \sum_{i=m}^{r-1} \frac{\log \alpha_i}{\alpha_i}.$$

This quantity can be bounded from above with a lower bound on α_i. Write $D_r = b \cdot |I(p_{m+1})|$ for a constant b with $0 < b \le 1$. So we have

$$\alpha_i = \frac{D_r}{\Delta_i} > \frac{D_r}{|I(p_i)|} = \frac{b \cdot |I(p_{m+1})|}{|I(p_i)|}.$$

Since we are using a *good* pivot selection method, we get the bound

$$|I(p_i)| \le |I(p_{m+1})| \cdot c^{d(p_i) - d(p_{m+1}) + O(1)}.$$

Plugging in gives us $\alpha_i > b \cdot c^{-d(p_i)+d(p_{m+1})+O(1)} \ge b \cdot c^{m+1-i+O(1)}$. The last inequality uses the fact that the recursion depths must be strictly increasing. Then

$$\sum_{i=m}^{r-1} \frac{\log \alpha_i}{\alpha_i} \le \sum_{j=0}^{r-1-m} \frac{\log(bc^{j+O(1)})}{bc^{j+O(1)}} = O(1).$$

And thus

$$D_r \log D_r - \sum_{i=m}^{r-1} \Delta_i \log \Delta_i = O(D_r).$$

A similar argument on the left side gives

$$D_l \log D_l - \sum_{i=0}^{m-1} \Delta_i \log \Delta_i = O(D_l).$$

Finally, we show that $\Delta \log \Delta - D_r \log D_r - D_l \log D_l = O(\Delta)$ to complete the proof. Since we have chosen a good pivot, for some constant $0 < \alpha < 1$, let $D_r = \alpha\Delta$ and $D_l = (1-\alpha)\Delta$. Then we have

$$\begin{aligned}\Delta \log \Delta - D_r \log D_r - D_l \log D_l &= \Delta \log \Delta - \alpha\Delta \log \alpha\Delta - (1-\alpha)\Delta \log(1-\alpha)\Delta \\ &= \Delta \log \Delta - (\alpha\Delta \log \Delta + \alpha\Delta \log \alpha) - ((1-\alpha)\Delta \log \Delta + (1-\alpha)\Delta \log(1-\alpha)) \\ &= O(\Delta),\end{aligned}$$

which finishes the proof. □

Theorem 1 *(Online multiselection). Given an (unsorted) array* $\mathbb{A}$ *of* n *elements and a sequence* R *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. [14, 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). 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 [9] gives this to us in under 3Δ comparisons. We add another comparison per element for the partitioning step. Overall, for each array element, we require comparisons equal to four times the element's 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, which improves upon the work of Kaligosi et al. [14].

3.2. Reducing the number of 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 median as our pivot. We wish to set values for m and t such that three 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. The last two events are mirror images of one another, and so have the same probability of occurring.

First Event. This event is the simplest of the three 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 $(1 - 2/\log\Delta)^{(1.1)\log\Delta\log n} = \exp[-(2.2)\log n(1 + o(1))] = O(1/n^2)$. 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 to bound the sum of the first k binomial coefficients to achieve our result. The following bound and proof are attributed to Lovász [16].

Lemma 2. *(See [16].) 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 $0 \le j \le m-t$.

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. Hence $A < cB + c^2B + c^3B + \cdots < \frac{c}{1-c}B$. We also know that $A + B \le 2^{2m-1}$. Combining these we get

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

Solving for A completes the proof. □

We then bound

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

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 $0 \le t < m$. 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 with probability $1/2$, the event of choosing $2m$ elements and having less than $m-t$ fall below the median occurs with probability at most

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

This is bounded by $(1/2)\exp[-t^2/(m+t)]$ by Lemma 3. 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 $t \ge \sqrt{2m\log n}(1+o(1))$. (Here we assume that $t = o(m)$.)

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. *For any integer n, suppose we are given an array of Δ elements, such that $\Delta/(\log\Delta)^2 \ge 6(\log n)^3$ and $\Delta \le n$. 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 the median of the sample falls in position $\Delta/2 \pm \Delta/\log\Delta$ in the original array.*

4. Optimal online multiselection

In this section, we prove our central theorem, Theorem 3, which forms the foundation for the remaining results in this article. Our bounds match those of the offline algorithm of Kaligosi et al. [14] when $q' = 0$ (i.e., when there are no search queries). In other words, we provide the first 1-competitive online multiselection algorithm. The main difference between the simple algorithm from Theorem 1 and the optimal online algorithm from Theorem 3 is that the latter uses *runs* to find a good pivot choice and partition efficiently. Kaligosi [14] also finds these runs, but their algorithm is offline.

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

We explain the proof of Theorem 3 in three main steps. In Section 4.1, we explain our algorithm and describe how it is different from the algorithm in [14]. 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 $o(\mathcal{B}(S_q)) + O(n)$. Combining these results, we arrive at the result of Theorem 3.

4.1. Algorithm description

We briefly describe the deterministic algorithm from Kaligosi et al. [14] to facilitate in understanding the key differences between our new online algorithm and their offline algorithm. They begin by creating *runs*, which are sorted sequences from $\mathtt{A}$ of length roughly $\ell = \log(\mathcal{B}/n)$. More precisely, there are at most ℓ runs of size less than ℓ, no two of which share the same length. All the remaining runs are of length between ℓ and 2ℓ. Then, they compute the median m of the medians of these runs 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. In each recursive subproblem, they merge short runs of the same size 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. [14]:

- The queries are processed online, that is, one at a time, from R without knowing which queries will follow. To do this, we maintain the bitvector $\mathtt{V}$ as described in Section 2.
- We support q' search queries in addition to selection queries; in the analysis, we treat them as selection queries and pay an extra $O(q' \log n)$ comparisons to account for the binary search to find the (unsorted) interval corresponding to each query (as in Algorithm 3).
- Since we don't know all of R 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. [14]. 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, $d(p) = O(\lg n)$. (Also, $\ell = \log\log n + O(1)$ 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 $\mathtt{W}$ 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 article, we use bitvector $\mathtt{V}$ 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. [14], we compute the value of ℓ for the current interval, merge runs so that there is at most one run of each length less than ℓ, and then use medians of those runs to compute a median-of-medians to use as a pivot. We then partition each run by using binary search on the median-of-medians.

We can borrow much of the analysis done in [14]. 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 and analysis to handle this complication.

4.2. Merging

Kaligosi et al. [14, 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 (from Kaligosi et al. [14]) 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 of length $|r|$ be a sorted sequence of elements from $\mathtt{A}$ in a gap $\Delta_i^{P_t}$. Then, for the pivot set P_t with k pivots,*

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

Proof. In a gap δ of size Δ, $\ell = O(\log d)$ where d the recursion depth of the elements in the gap. This gives $\sum_{r \in \delta} |r| \log |r| \le \Delta \log(2l) = O(\Delta \log\log d)$, 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, $\sum_{i=0}^{k} \sum_{r \in \Delta_i^{P_t}} |r| \log |r| \le \sum_{i=0}^{k} |\Delta_i^{P_t}| \log\log\log(n/|\Delta_i^{P_t}|)$. Recall that $\mathcal{B}(S_t) = \mathcal{B}(P_t) + O(n) = \sum_{i=0}^{k} |\Delta_i^{P_t}| \log(n/|\Delta_i^{P_t}|) + O(n)$. 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 incurred 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 2.2), and let the root of T have depth $d = 0$. In tree T, each node v is associated with an 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 $\ell = 1 + \lfloor \log(d+1) \rfloor$, and that a *run* is a sorted sequence of elements from $\mathbb{A}$. We define a *short run* as a run of length less than ℓ, and a *long run* as having a length of ℓ or more. Let βn be the number of comparisons required to compute the exact median for n (unsorted) elements, where β is a constant less than three [9]. 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 finding the median m 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 costs from Lemma 5 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 with at least* $4\ell^2$ *elements in the pivot tree T is at most* $o(\mathcal{B}(S_t)) + O(n)$.

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

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

Since $\ell = \lfloor \log(d+1) \rfloor + 1$, the first term of the summation is bounded by

$$O\left((\mathcal{B}(P_t)/n) \log\log(\mathcal{B}(P_t)/n) \cdot \log\log\log(\mathcal{B}(P_t)/n)\right) = o\left(\mathcal{B}(P_t)\right).$$

The second term can be easily upper-bounded by

$$O\left(\frac{n \log(\mathcal{B}(P_t)/n) \cdot \log\log\log(\mathcal{B}(P_t)/n)}{\log\log(\mathcal{B}(P_t)/n)}\right) = o\left(\mathcal{B}(P_t)\right).$$

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

For each level d with $\log(\mathcal{B}(P_t)/n) < d \le \log\log n + O(1)$, 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 $\ell = \log d + O(1) = \log(O(\log(n/\Delta_v))) = \log\log(n/\Delta_v) + O(1)$, since we use a good pivot selection method. Summing over all nodes, we get $\sum_v (\Delta_v/\ell)\log(2\ell) \le \sum_v \Delta_v \log(2\ell) = o(\mathcal{B}(P_t)) + O(n)$, 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 $\ell \le \log\log n + O(1)$. 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, $t \ge n/z$. Then, we know that $\mathcal{B}(P_t) \ge (n/z)\log(n/z)$, by Jensen's inequality. (In words, this represents the sort cost of n/z adjacent queries.) Thus, $tz = o(\mathcal{B}(P_t))$. Using Lemma 1, we know that $\mathcal{B}(P_t) = \mathcal{B}(S_t) + O(n)$, 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 still supporting selection queries. We are originally given the (unsorted) array A. To support *insert* and *delete* efficiently, we maintain newly-inserted elements in a separate data structure, and mark deleted elements in A. These *insert* and *delete* operations are occasionally merged to make the array A up-to-date. Let A′ denote the current array with length n'. We support the following update operations:

- *insert*(a), which inserts a into A′, and;
- *delete*(i), which deletes the ith sorted entry from A′.

5.1. Preliminaries

Our solution uses the *dynamic bitvector* of Hon et al. [12], which supports the following operations on a dynamic bitvector V:

- The $rank_b(i)$ operation returns the number of b bits up to the ith position in V.
- The $select_b(i)$ operation returns the position in 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.

One can determine the ith bit of V by computing $rank_{\mathbf{1}}(i) - rank_{\mathbf{1}}(i-1)$. (For convenience, we assume that $rank_b(-1) = 0$.) For the case of maintaining a dynamic bit vector, the result of Hon et al. [12, Theorem 1] can be re-stated as follows:

Lemma 8. *(See [12].) Given a bitvector* 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 $(\log n)^{O(1)} \le t \le n$. This structure assumes access to a precomputed table of size n^ϵ, for any fixed positive constant ϵ.*

All the pivots (and their positions) generated during *select*, *search*, *insert*, and *delete* operations on array A are maintained using a bitvector V as in Section 4. In addition, we also maintain two bitvectors, each of length n': (i) an *insert bitvector* I such that $\mathtt{I}[i] = \mathbf{1}$ if and only if $\mathtt{A}'[i]$ is newly inserted, and (ii) a *delete bitvector* D such that if $\mathtt{D}[i] = \mathbf{1}$, the ith element in A has been deleted. If a newly inserted item is deleted, it is removed from I directly. Both I and 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 A′. We support the following operations on tree T: (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 I and D are each initialized with n **0**s. The tree T is initially empty.

After performing n *insert* and *delete* operations, where n is the size of array A, we merge all the elements in T with the array A, modify the bitvector V appropriately, and reset the bitvectors I and D (with all zeroes). This increases the amortized time of the *insert* and *delete* operations by $O(1)$, without requiring additional comparisons.

5.2. Dynamic online multiselection

We now describe how to support A′.*insert*(a), A′.*delete*(i), A′.*select*(i), and A′.*search*(a) operations.

A′.*insert*(a). First, perform A.*search*(a) (Algorithm 3), which the reader may recall searches for the appropriate unsorted interval $[\ell, r]$ containing a using a binary search on the original (unsorted) array A; it then performs A.*qsearch*(ℓ, r, a) on interval $[\ell, r]$ 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 at position $\mathtt{I}.\mathit{select}_{\mathbf{1}}(j)$ among all the newly-inserted elements. Calculate $j' = \mathtt{I}.\mathit{select}_{\mathbf{0}}(j)$, and set a's position to $j'' = j' - \mathtt{D}.\mathit{rank}_{\mathbf{1}}(j')$. Finally, we update our bitvectors by performing $\mathtt{I}.\mathit{insert}_{\mathbf{1}}(j'')$ and $\mathtt{D}.\mathit{insert}_{\mathbf{0}}(j'')$. Note that, apart from the *search* operation, no other operation in the insertion procedure performs comparisons between the elements.

$\mathtt{A}'.\mathit{delete}(i)$. Compute $i' = \mathtt{D}.\mathit{select}_{\mathbf{0}}(i)$. If i' is newly-inserted (i.e., $\mathtt{I}[i'] = \mathbf{1}$), then remove the node (element) with inorder number $\mathtt{I}.\mathit{rank}_{\mathbf{1}}(i')$ from T. Perform $\mathtt{I}.\mathit{delete}(i')$ and $\mathtt{D}.\mathit{delete}(i')$. 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.[9]

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

$\mathtt{A}'.\mathit{search}(a)$. Search using $\mathtt{A}.\mathit{search}(a)$ (Algorithm 3), to find the appropriate unsorted interval $[\ell, r]$ containing a; followed by $\mathtt{A}.\mathit{qsearch}(\ell, r, a)$ on interval $[\ell, r]$ until a's exact position j in $\mathtt{A}$ is determined. If a appears in $\mathtt{A}$ (which we discover through $\mathtt{A}.\mathit{search}$), we need to check whether it has been deleted. We compute $j' = \mathtt{I}.\mathit{select}_{\mathbf{0}}(j)$ and $j'' = j' - \mathtt{D}.\mathit{rank}_{\mathbf{1}}(j')$. If $\mathtt{D}[j'] = \mathbf{0}$, return j''. Otherwise, it is possible that the item has been newly-inserted. Compute $p = \mathtt{I}.\mathit{rank}_{\mathbf{1}}(j')$, which is the number of newly-inserted elements that are less than or equal to a. If a is the element corresponding to the pth node in the inorder traversal of T, then return j''; otherwise, return failure.

We now analyze the comparison cost for the above algorithm in Theorem 4 and the running time in Corollary 9.

Theorem 4 *(Online dynamic multiselection). Given an (unsorted) array $\mathtt{A}'$ of n elements, we provide a dynamic online algorithm that can support $q = O(n)$ select, search, insert, and delete operations, of which r are search, insert, and delete, using at most $\mathcal{B}(S_q)(1 + o(1)) + O(n + r\log n)$ comparisons.*

Proof. Let n' denote the current length of the dynamic array $\mathtt{A}'$, 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 re-buildings of our dynamic data structure. Recall that each of the r *search*, *insert*, and *delete* operations in the sequence will perform a constant number of search operations. To execute these searches, we require $O(r\log n')$ comparisons. 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 can safely assume that all the updates are insertions. Since each inserted element becomes a pivot immediately, it does not contribute to the comparison cost of any other *select* operation. Similarly, from Theorem 3, no pivot is part of a run and hence cannot affect the choice of any future pivot.

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

Next, we modify Theorem 4 to account for the time costs of the dynamic bitvector from Lemma 8, and we obtain the following result.

Corollary 9. *Given a dynamic array $\mathtt{A}'$ of n original elements, there exists a dynamic online data structure that can support $q = O(n)$ select, search, insert, and delete operations, of which r are search, insert and delete, and u of which are insert and delete. We provide an online algorithm supporting those operations in $O(\mathcal{B}(S_q) + q\log_t n + r\log n + ut)$ time, for any t where $(\log n)^{O(1)} \le t \le n$.*

6. External online multiselection

Suppose we are given an (unsorted) array $\mathtt{A}$ of length N stored in $n = N/B$ blocks in the external memory. Recall that 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 $O(n\log_m n)$ I/Os used by the optimal mergesort algorithm for external memory.

[9] 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.

6.1. *A lower bound for multiselect in external memory*

As in the internal memory case, 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 sequence R, 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 2.2. Then the lower bound on the number of I/Os required to support the queries in S_t is given by

$$\mathcal{B}_m(S_t) = n \log_m n - \sum_{i=0}^{t} \left(\Delta_i^{S_t}/B\right) \log_m \left(\Delta_i^{S_t}/B\right) - O(n),$$

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 $\mathcal{B}_m(S_t) = \Omega(n)$ 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 partitioned the array elements 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 $d = \Theta(m)$, by finding a sample of d "roughly equidistant elements." The next lemma describes how to find such a sample, and then partition the array elements into $d+1$ subranges with respect to the sample.

As in [1], we assume that $B = \Omega(\log_m n)$—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 transdichotomous word RAM model [10]. In addition, Sibeyn's algorithm [20] works only under this assumption, though this fact is not explicitly mentioned in that paper.

Lemma 10. *Given an (unsorted) array* $\mathtt{A}$ *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 [1] 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. *Algorithm 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 *(External static online multiselection). Given an (unsorted) array* $\mathtt{A}$ *occupying* n *blocks in external memory, we provide a deterministic algorithm that supports a sequence* R *of* q *online selection queries using* $O(\mathcal{B}_m(S_q))$ *I/Os under the condition that* $B = \Omega(\log_m n)$.

Proof. Our algorithm uses the same approach as the simple internal memory algorithm of Section 3, 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 10. This process requires $O(\delta_v + d)$ I/Os, where $\delta_v = \Delta_v/B$.

We also maintain the bitvector $\mathtt{V}$ of length N, as described before. For each $\mathtt{A}.select(i)$ query, we access position $\mathtt{V}[i]$. If $\mathtt{V}[i] = \mathbf{1}$, we return $\mathtt{A}[i]$; otherwise we 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 queries, instead of taking $O(\log N)$ time performing a binary search on the blocks of $\mathtt{V}$, we build a B-tree T maintaining all pivots from $\mathtt{A}$. (During preprocessing, we insert $\mathtt{A}[1]$ and $\mathtt{A}[n]$ into T.) The B-tree T can support *search* queries in $O(\log_B N)$ I/Os instead of $O(\log N)$ I/Os. Then, we modify the proof of Theorem 5 to obtain the following result.

Corollary 11. *Given an (unsorted) array* $\mathtt{A}$ *occupying* n *blocks in external memory, we provide a deterministic algorithm that supports a sequence* R *of* q *online selection and search queries using* $O(\mathcal{B}_m(S_q) + q\log_B N)$ *I/Os under the condition that* $B = \Omega(\log_m n)$.

Proof. The first term follows directly from the proof of Theorem 5. Now we justify the second term $O(q\log_B N)$.

As mentioned earlier, we build a B-tree T maintaining all pivots from $\mathtt{A}$. (During preprocessing, we insert $\mathtt{A}[1]$ and $\mathtt{A}[n]$ 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 $O(\min\{qm(\log_m N), N\} \cdot (\log_B N))$ 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 $\log_B(\min\{qm(\log_m N), N\}) = O(\log_B N)$ I/Os navigating to I_z using T. Our approach is to insert all $O(m\log_m N) = O((M/B)\log_m N) = O(M)$ pivots within I_z in a single batched manner. This process can be done easily 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 $q = O(\mathcal{B}_m(S_q)/\log_B N)$, then Corollary 11 requires only $O(\mathcal{B}_m(S_q))$ I/Os, matching the bounds from Theorem 5. Hence, our result is asymptotically optimal when $\mathcal{B}_m(S_q)/q = \log_B N$.

Combining the ideas from Corollary 11 and Theorem 4, we can dynamize the above algorithm. The proof follows from the fact that we can maintain the bit vectors $\mathtt{I}$ and $\mathtt{D}$ described in the multiselection algorithms of Section 5 using a B-tree in external memory.

Theorem 6 *(External dynamic online multiselection). Given an (unsorted) array* $\mathtt{A}$ *occupying* n *blocks in external memory, we provide a deterministic algorithm that supports a sequence* R *of* q *online select, search, insert, and delete operations using* $O(\mathcal{B}_m(S_q) + q\log_B N)$ *I/Os under the condition that* $B = \Omega(\log_m n)$.

References

[1] A. Aggarwal, J.S. Vitter, The input/output complexity of sorting and related problems, Commun. ACM 31 (9) (1988) 1116–1127.
[2] J. Barbay, A. Gupta, S. Jo, S.R. Satti, J.P. Sorenson, Theory and implementation of online multiselection algorithms, in: Proceedings of the European Symposium on Algorithms, 2013, pp. 109–120.
[3] J. Barbay, A. Gupta, S.R. Satti, J.P. Sorenson, Dynamic online multiselection in internal and external memory, in: M.S. Rahman, E. Tomita (Eds.), Proceedings of WALCOM: Algorithms and Computation, 9th International Workshop, WALCOM 2015, Dhaka, Bangladesh, February 26–28, 2015, in: Lecture Notes in Computer Science, vol. 8973, Springer, 2015, pp. 199–209.
[4] T.C. Biedl, T.M. Chan, E.D. Demaine, R. Fleischer, M.J. Golin, J.A. King, J.I. Munro, Fun-sort—or the chaos of unordered binary search, Discrete Appl. Math. 144 (3) (2004) 231–236.
[5] M. Blum, R.W. Floyd, V.R. Pratt, R.L. Rivest, R.E. Tarjan, Time bounds for selection, J. Comput. Syst. Sci. 7 (4) (1973) 448–461.
[6] G. Brodal, R. Fagerberg, On the limits of cache-obliviousness, in: Proceedings of the ACM Symposium on Theory of Computing, 2003, pp. 307–315.
[7] J. Cardinal, S. Fiorini, G. Joret, R.M. Jungers, J.I. Munro, An efficient algorithm for partial order production, in: Proceedings of the 41st Annual ACM Symposium on Theory of Computing, STOC '09, ACM, New York, NY, USA, 2009, pp. 93–100.
[8] D.P. Dobkin, J.I. Munro, Optimal time minimal space selection algorithms, J. ACM 28 (3) (1981) 454–461.
[9] D. Dor, U. Zwick, Selecting the median, SIAM J. Comput. 28 (5) (1999) 1722–1758.
[10] M.L. Fredman, D.E. Willard, Surpassing the information theoretic bound with fusion trees, J. Comput. Syst. Sci. 47 (3) (1993) 424–436.
[11] C.A.R. Hoare, Algorithm 65: find, Commun. ACM 4 (7) (1961) 321–322.
[12] W.-K. Hon, K. Sadakane, W.-K. Sung, Succinct data structures for searchable partial sums, in: Proceedings of the International Symposium on Algorithms and Computation, 2003, pp. 505–516.
[13] R.M. Jiménez, C. Martínez, Interval sorting, in: Proceedings of the International Colloquium on Automata, Languages, and Programming, 2010, pp. 238–249.
[14] K. Kaligosi, K. Mehlhorn, J.I. Munro, P. Sanders, Towards optimal multiple selection, in: ICALP, 2005, pp. 103–114.
[15] D.E. Knuth, The Art of Computer Programming, Volume III: Sorting and Searching, Addison–Wesley, 1973.
[16] L. Lovász, J. Pelikán, K. Vesztergombi, Discrete Mathematics: Elementary and Beyond, Springer, New York, NY, 2003.
[17] R. Motwani, P. Raghavan, Deferred data structuring: query-driven preprocessing for geometric search problems, in: Symposium on Computational Geometry, 1986, pp. 303–312.
[18] H. Prodinger, Multiple quickselect—Hoare's find algorithm for several elements, Inf. Process. Lett. 56 (3) (1995) 123–129.
[19] A. Schönhage, M. Paterson, N. Pippenger, Finding the median, J. Comput. Syst. Sci. 13 (2) (1976) 184–199.
[20] J.F. Sibeyn, External selection, J. Algorithms 58 (2) (2006) 104–117.
[21] M.H.M. Smid, Y.-T. Ching, K. Mehlhorn, Dynamic deferred data structuring, Inf. Process. Lett. 35 (1) (1990) 37–40.