Theory and Implementation of Online Multiselection Algorithms

Jérémy Barbay[1⋆], Ankur Gupta[2⋆⋆], Seungbum Jo[3⋆⋆⋆], S. Srinivasa Rao[3⋆⋆⋆], and Jonathan Sorenson[2]

[1] Departamento de Ciencias de la Computación (DCC), Universidad de Chile
[2] Department of Computer Science and Software Engineering, Butler University
[3] School of Computer Science and Engineering, Seoul National University

Abstract. We introduce a new online algorithm for the *multiselection problem* which performs a sequence of selection queries on a given unsorted array. We show that our online algorithm is 1-competitive in terms of data comparisons. In particular, we match the bounds (up to lower order terms) from the optimal offline algorithm proposed by Kaligosi et al.[ICALP 2005].
We provide experimental results comparing online and offline algorithms. These experiments show that our online algorithms require fewer comparisons than the best-known offline algorithms. Interestingly, our experiments suggest that our optimal online algorithm (when used to sort the array) requires fewer comparisons than both quicksort and mergesort.

1 Introduction

Let A be an unsorted array of n elements drawn from an ordered universe. The *multiselection* problem asks for elements of rank r_i from the sequence $R = r_1, r_2, \ldots, r_q$ on A. 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 unique queries, where $S_q = \{s_i\}$ denotes the queries ordered by rank. We define $\Delta_i^S = s_{i+1} - s_i$, where $s_0 = 0$ and $s_{q+1} = n$. Then,

$$\mathcal{B}(S_q) = \log n! - \sum_{i=0}^{q} \log\left(\Delta_i^S!\right) = \sum_{i=0}^{q} \Delta_i^S \log \frac{n}{\Delta_i^S} - O(n).^4$$

As mentioned by Kaligosi et al. [10], intuitively $\mathcal{B}(S_q)$ follows from the fact that any comparison-based multiselection algorithm identifies the Δ_1^S smallest elements, Δ_2^S next smallest elements, and so on. Hence, one could sort the original array A using $\sum_i \Delta_i^S \log \Delta_i^S + O(n)$ additional comparisons.

⋆ Supported in part by PROYECTO Fondecyt Regular no 1120054.
⋆⋆ Supported in part by the Arete Initiative at the University of Chicago.
⋆⋆⋆ Supported in part 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).

[4] We use $\log_b a$ to refer to the base b logarithm of a. By default, we let $b = 2$.

The *online multiselection* problem asks for elements of rank $r_1, r_2, \ldots, r_q$, where the sequence R is given one element at a time (in any order).

Motivation. Online multiselection is equivalent to generalized partial sorting [9]. Variants of this problem have been studied under the names *partial quicksort*, *multiple quickselect*, *interval sort*, and *chunksort*. Several applications, such as computing optimal prefix-free codes [3] and convex hulls [11], repeatedly compute medians over different ranges within an array. Online multiselection (where queries arrive one at a time) may be a key ingredient to improved results for these types of problems, whereas offline algorithms will not suffice. Most recently, Cardinal et al. [5] generalized the problem to *partial order production*, and they use multiselection as a subroutine after an initial preprocessing phase.

Previous Work. Several papers [6, 12, 9] have analyzed the offline multiselection problem, but these approaches must all know the queries in advance. Kaligosi et al. [10] described an algorithm performing $\mathcal{B}(S_q)+o(\mathcal{B}(S_q))+O(n)$ comparisons.

Our Results. For the *multiselection* problem in internal memory, we describe the *first* online algorithm that supports a sequence R of q selection queries using $\mathcal{B}(S_q) + o(\mathcal{B}(S_q))$ comparisons. Our algorithm is 1-competitive in the number of comparisons performed. We match the bounds above while supporting search, insert, and delete operations, achieve similar results in the external memory model [1]. We invite readers to see [2] (or the upcoming journal version) for more details on these results.

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 at least x. The median can be computed in $O(n)$ comparisons [8, 4, 13, 7], in particular, less than $3n$ comparisons [7].

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 2.2, we show that the simple algorithm has a constant competitive ratio. Section 3 describes modifications to the simple algorithm, and shows that the modified algorithm is optimal up to lower order terms. We describe the experimental results in Section 4.

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 queries on array $\mathtt{A}$. We call an element $\mathtt{A}[i]$ at position i in array $\mathtt{A}$ a *pivot* if $\mathtt{A}[1 \ldots i-1] < \mathtt{A}[i] \leq \mathtt{A}[i+1 \ldots n]$.

Bitvector. We maintain a bitvector $\mathtt{V}$ of length n where $\mathtt{V}[i] = \mathbf{1}$ if and only if $\mathtt{A}[i]$ is a pivot. During *preprocessing*, we create $\mathtt{V}$ and set each bit to $\mathbf{0}$. We 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. The operation $\mathtt{A}.select(s)$ returns the sth smallest element of $\mathtt{A}$ (i.e., $\mathtt{A}[s]$ 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 $a < s$ and $b > s$, such that $\mathtt{V}[a] = \mathtt{V}[b] = \mathbf{1}$ but $\mathtt{V}[a+1 \ldots b-1]$ are all $\mathbf{0}$. Perform quickselect [8] 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.

As queries arrive, our algorithm performs the same steps that quicksort would perform, although not necessarily in the same order. As a result, our recursive subproblems mimic those from quicksort. We can show that comparisons needed to perform q select queries on an array of n items is $O(n \log q)$. We can improve this result to $O(\mathcal{B}(S_q))$.[5] We do not prove this bound directly, since our main result is an improvement over this bound. Now, we define terminology for this improved analysis.

2.1 Terminology

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 $s_0 = 1$ 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, 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 and Recursion Depth. The pivots chosen by the algorithm form a binary tree structure, defined as the *pivot tree* T of the algorithm over time.[6] 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.

Intervals. Each pivot was used to partition an interval in $\mathtt{A}$. Let $I(p_i)$ denote the interval partitioned by pivot 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 pivot 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.

Gaps. Define the query gap $\Delta_i^{S_t} = s_{i+1} - s_i$ and similarly the pivot gap $\Delta_i^{P_t} = p_{i+1} - p_i$. By telescoping we have $\sum_i \Delta_i^{S_t} = \sum_j \Delta_j^{P_t} = n - 1$.

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

[5] $\mathcal{B}(S_q) = n \log q$ when the q queries are evenly spaced over the input array $\mathtt{A}$.

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

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. □

2.2 Analysis of the Simple Algorithm

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

We call a pivot selection method *c-balanced* for some constant c with $1/2 \leq c < 1$ if, for all pairs (p_i, p_j) where p_i is an ancestor of p_j in the pivot tree, then $|I(p_j)| \leq |I(p_i)| \cdot c^{d(p_j)-d(p_i)+O(1)}$. If the median is always chosen as the pivot, we have $c = 1/2$ and the $O(1)$ term is zero. The pivot selection method of Kaligosi et al. [10, Lemma 8] is c-balanced with $c = 15/16$.

Lemma 1 (Entropy Lemma). *If the pivot selection method is c-balanced, then* $\mathcal{B}(P_t) = \mathcal{B}(S_t) + O(n)$.

Proof. We sketch the proof and defer the full details to the journal version of the paper. (Those results also appear in [2].) 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 $p_r = s'$. The lemma follows from the claim that $\mathcal{B}(P_\Delta) = O(\Delta)$, since

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

We now sketch the proof of our claim, which proves the lemma. There must be a unique pivot p_m in P_Δ of minimal recursion depth. We split the gap Δ at p_m. Since we use a c-balanced pivot selection method, we can bound the total information content of the left-hand side by $O(\sum_{i=l}^{m-1} \Delta_i)$ and the right-hand side by $O(\sum_{i=m}^{r-1} \Delta_i)$, leading to the claim. The result follows. □

3 Optimal Online Multiselection

In this section we prove the main result of our paper, Theorem 1.

Theorem 1 (Optimal Online Multiselection). *Given an unsorted array* `A` *of n elements, we provide an algorithm that supports a sequence R of q online selection queries using* $\mathcal{B}(S_q)(1 + o(1)) + O(n)$ *comparisons.*

Our bounds match those of the offline algorithm of Kaligosi et al. [10]. In other words, our solution is 1-competitive. We explain our proof in three main steps. In Section 3.1, we explain our algorithm and describe how it is different from Kaligosi et al. [10]. We then bound the number of comparisons resulting from merging by $\mathcal{B}(S_q)(1+o(1))+O(n)$ in Section 3.2. In Section 3.3, we bound the complexity of pivot finding and partitioning by $o(\mathcal{B}(S_q))+O(n)$.

3.1 Algorithm Description

We briefly describe the deterministic algorithm from Kaligosi et al. [10]. Their result is based on tying the number of comparisons required for merging two sorted sequences to the information content of those sequences. This simple observation drives their underlying approach that both finds pivots that are "good enough" and partitions using near-optimal comparisons.

In particular, they create *runs*, which are sorted sequences from `A` of length roughly $\ell = \log(\mathcal{B}/n)$. Then, they compute the median μ of the medians of these sequences, and partition the runs based on μ. 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 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 algorithm of Kaligosi et al. [10]:

- Since the value of $\mathcal{B}(S_q)$ is not known in advance (because R is provided *online*), we cannot preset a value for ℓ, as done in Kaligosi et al. [10]. Instead, we locally set $\ell = 1 + \lfloor \log(d(p)+1) \rfloor$ in the interval $I(p)$. Since we use only balanced pivots, $d(p) = O(\log n)$. We keep track of the recursion depth of pivots, from which it is easy to compute the recursion depth of an interval.
- We use a bitvector `W` to identify the endpoints of runs within each interval.
- The queries from R are processed *online*. We support online queries using the bitvector `V` from Section 2. Recall that a search query incurs $O(\log n)$ additional comparisons to find its corresponding interval.

To perform the operation `A`.*select*(s), we first use bitvector `V` to identify the interval I containing s. If $|I| \leq 4\ell^2$, we sort the interval I (making all elements of I pivots) and answer the query s. The cost for this case is bounded by Lemma 5. Otherwise, we compute the value of ℓ for the current interval, and proceed as in Kaligosi et al. [10] to answer the query s.

We can borrow much of the analysis done in [10], but it depends heavily on the use of ℓ, which we do not know in advance. In the rest of Section 3, we modify their techniques to handle this complication.

3.2 Merging

Kaligosi et al. [10, 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 its proof uses the value of ℓ, its validity does not depend the size of ℓ. The proof merely requires that there are at least $4\ell^2$ items in each interval, which also holds for our algorithm. Lemmas 9 and 10 (from Kaligosi et al. [10]) together will bound the number of comparisons by $\mathcal{B}(S_q)(1+o(1))+O(n)$ if we can prove Lemma 2, which bounds the information content of runs in intervals that are not yet partitioned.

Lemma 2. *Let a run r be a sorted sequence of elements from $\mathtt{A}$ in a gap $\Delta_i^{P_t}$, where $|r|$ is its length. Then, $\sum_{i=0}^{k}\sum_{r\in\Delta_i^{P_t}}|r|\log|r| = o(\mathcal{B}(S_t))+O(n)$.*

Proof. In a gap 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, $\sum_{i=0}^{k}\sum_{r\in\Delta_i^{P_t}}|r|\log|r| \le \sum_i \Delta_i\log\log\log(n/\Delta_i)$. Recall that $\mathcal{B}(S_t) = \mathcal{B}(P_t)+O(n) = \sum_i\Delta_i\log(n/\Delta_i)+O(n)$. Fact 1 completes the proof. □

3.3 Pivot Finding and Partitioning

Now we prove that the cost of computing medians and performing partitions 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 3 and 4. We bound the comparison cost for all nodes with fewer elements in Lemma 5.

Let d be the current depth of the pivot tree T (defined in Section 2.1), and let the root of T have depth $d=0$. Each node v in T 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.

Recall that $\ell = 1+\lfloor\log(d+1)\rfloor$, and a *run* is a sorted sequence of elements in $\mathtt{A}$. 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 [7]. Let r_v^s be the number of short runs at node v, and let r_v^l be the number of *long runs* (runs of length at least ℓ).

Lemma 3. *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 in 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 3 by level. The overall cost at level d is at most $2^d\beta\ell + 2^d\ell\log\ell + (n/\ell)\beta + (n/\ell)\log(2\ell)$ comparisons. Summing over all the levels, we can bound the total cost of all such nodes in the pivot tree to obtain the following lemma.

Lemma 4. *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 $\ell' \leq \log(\mathcal{B}(P_t)/n)$, 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 $\ell = \lfloor\log(d+1)\rfloor + 1$, we can easily bound the first term of the summation 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 $\log(\mathcal{B}(P_t)/n) < \ell' \leq \log\log n + O(1)$, we 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 $\Delta_v \geq 4\ell^2$ could be $\omega(\mathcal{B}(S_t))$.

We bound the overall cost as follows, using the result of Lemma 3. 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 $\sum_v(\Delta_v/\ell)\log(2\ell) \leq \sum_v \Delta_v\log(2\ell) = o(\mathcal{B}(P_t)) + O(n)$, 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. □

We now bound the comparison cost for all nodes v where $\Delta_v \leq 4\ell^2$.

Lemma 5. *For nodes v in the pivot tree T where $\Delta_v \leq 4\ell^2$, the total cost in comparisons for all operations is at most $o(\mathcal{B}(S_t)) + O(n)$.*

Proof. 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 be such that $z = (\log\log n)^2\log\log\log n + O(1)$. We sort the elements of any node v with $\Delta_v \leq 4\ell^2$ elements using $O(z)$ comparisons, since $\ell \leq \log\log n + O(1)$. 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, $t \geq n/z$. Using Jensen's inequality, we have $\mathcal{B}(P_t) \geq (n/z)\log(n/z)$, which represents the cost of sorting n/z adjacent queries. Thus, $tz = o(\mathcal{B}(P_t))$. Using Lemma 1, we know that $\mathcal{B}(P_t) = \mathcal{B}(S_t) + O(n)$, which proves the lemma. □

4 Experimental Results

In this section, we present the experimental evaluation of the online and offline multiselection algorithms. Section 4.1 describes the experimental setup. Our results are described in Section 4.2.

4.1 Experimental Setup

Our input array consists of a random permutation of the (distinct) elements from $[1, 2^{18}]$. (We also ran some experiments for larger n up to 2^20, and results were similar.) Our queries are generated using the indicated distribution for each experiment. We allow repetitions of queries, except in the evenly-spaced case. We only report comparisons with elements of the input array, averaged over 10 random experiments. In particular, we do *not* count comparisons between indices in the input array. Finally, we compute the *Entropy* of a query sequence S_q (defined in Section 1) by $\lfloor \log n! - \sum_{i=0}^{q} \log\left(\Delta_i^S!\right) \rfloor$ using double precision arithmetic on a 64-bit machine.

Now, we briefly describe the algorithms we considered for choosing the pivot in an unsorted interval I. The *First Element* and *Random* methods choose the corresponding element as the pivot. The *Medof3* method uses the median of the first, middle, and last elements of I as the pivot. The *Median* (using MedofMed) uses Blum et al.'s linear-time algorithm [4] as the pivot. The *MedofMed* method is the first step of Blum et al.'s algorithm [4] that computes the median of every five elements, and then uses the median of those medians as the pivot.

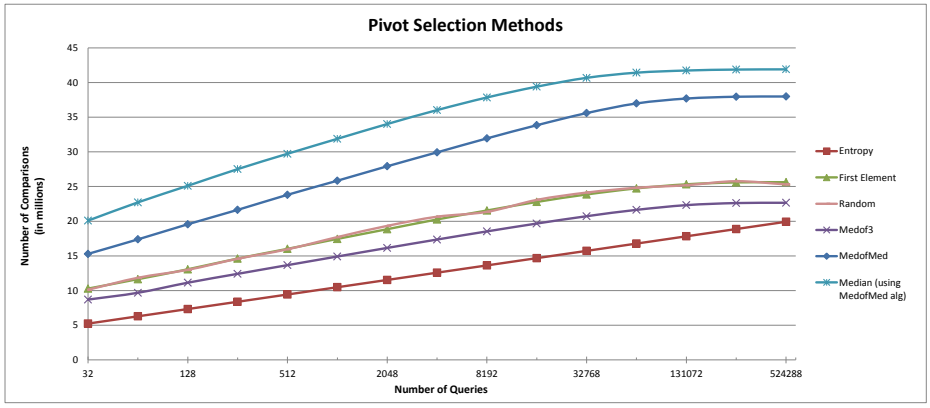

Fig. 1. Performance of various pivot selection methods on random input sequences.

We compared the performance of these pivoting methods for random arrays in Figure 1 for our simple online algorithm described in Section 2. We performed similar experiments for different algorithms. The results from Figure 1 are representative of all of our findings. One can clearly see that *Medof3* uses the fewest

comparisons and *Median* requires significantly more comparisons. The performance of other pivoting methods fall in between these two extremes. For the rest of the paper, we show results only for the *Medof3* pivoting method.

4.2 Results

Now, we briefly describe the algorithms we considered for multiselection. All algorithms use *Medof3* as the pivoting strategy (where applicable). The *Quicksort* algorithm is the standard quicksort, augmented by q array lookups (which require no comparisons). The *Mergesort* algorithm is the standard recursive mergesort, augmented by q array lookups (which require no comparisons). The *Simple Online* algorithm is described in Section 2. The *Optimal Online* algorithm is described in Section 3.1, where we set ℓ based on the recursion depth of the corresponding interval. The performance of the online algorithms is independent of the order of the queries. (We defer the experiments supporting this claim until the journal version of this paper.)

The *Dobkin-Munro* algorithm is described in [6]. The *Kaligosi (sorted)* algorithm is described in [10], which assumes that queries are given in sorted order. The *Kaligosi (unsorted)* algorithm first sorts the unsorted queries, and then performs the *Kaligosi (sorted)* algorithm. In some cases, sorting queries is tantamount to sorting the array. Since this algorithm is offline, one can assume that the algorithm will detect this case and revert to *Quicksort* or *Mergesort* instead.

We show our results in Figures 2 and 3. For Figure 2, the queries (in the first graph) are evenly distributed across the input array. This query distribution results in a worst-case entropy, and hence is a difficult case for multiselection algorithms. The second graph in Figure 2 has uniformly distributed queries. For Figure 3, we display results for a normal query distribution with mean $\mu = n/2$ and standard deviation $\sigma = n/8$. The second graph in Figure 3 is an exponential query distribution with $\lambda = 16/n$.

For all query distributions, our online algorithms (*Simple Online* and *Optimal Online*) outperform their offline counterparts (respectively, *Quicksort* and *Kaligosi*). The *Dobkin-Munro* algorithm requires more comparisons than *Quicksort* for any reasonably large number of queries (based on query distribution). In other words, it is usually better to sort than to use *Dobkin-Munro*. The *Kaligosi* algorithm performs quite well in terms of comparisons, but is relatively slow. The *Simple Online* algorithm converges to *Quicksort* as queries increases, highlighting that the online algorithm performs the same work as the *Quicksort*, as intuition (and the analysis) suggests.

The *Optimal Online* algorithm outperforms *Mergesort*, *Quicksort*, and *Kaligosi* (sorted and unsorted), and is even better than *Entropy* when the number of queries is large. Having an algorithm perform fewer comparisons than *Entropy* isn't a contradiction, since *Entropy* is a worst-case lower bound for an arbitrary input. Hence, the number of comparisons for an algorithm could be less than *Entropy* for a given (specific) input. Even though our algorithm is similar to *Kaligosi*, we can clearly see the value of online computation when comparing

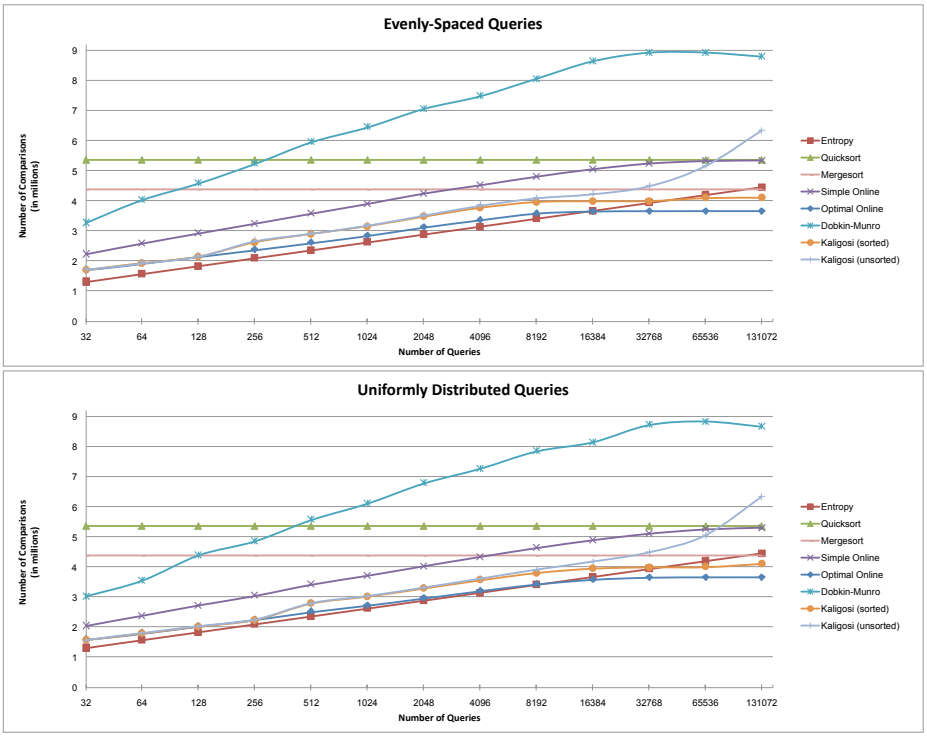

Fig. 2. Performance of multiselection algorithms on random input sequences using median of three pivot selection method, when the queries are distributed as indicated.

these two results. The primary reason for our improved results is due to the fact that *Kaligosi* will pre-process runs, even for intervals that do not contain any queries. For the *Optimal Online* algorithm, since run lengths are based on the recursion depth, the algorithm will not spend comparisons generating long runs unless queries are in those intervals.

In fact, these results suggest that using the *Optimal Online* algorithm with $n/2$ queries (e.g., each odd position) can sort an array in fewer comparisons than *Mergesort*. The reason for this is that the runs computed at the beginning of the algorithm save a lot of comparisons in future recursive rounds. We are currently running experiments on tuning the length ℓ of the run to see if we can further improve this performance.

Finally, we provide similar results for a decreasing input array, since this is a best-case scenario for *Mergesort*. Notice that both *Mergesort*, *Optimal Online*, and *Kaligosi* are better than *Entropy* as queries increase. However, both multiselection algorithms outperform *Mergesort*. The sudden dip in the curve corresponding to the *Kaligosi (sorted)* algorithm after $65,536$ queries corresponds to a discrete increase in the calculated value of ℓ (from 4 to 5). This sort of stair-stepping behavior is expected to continue as n increases.

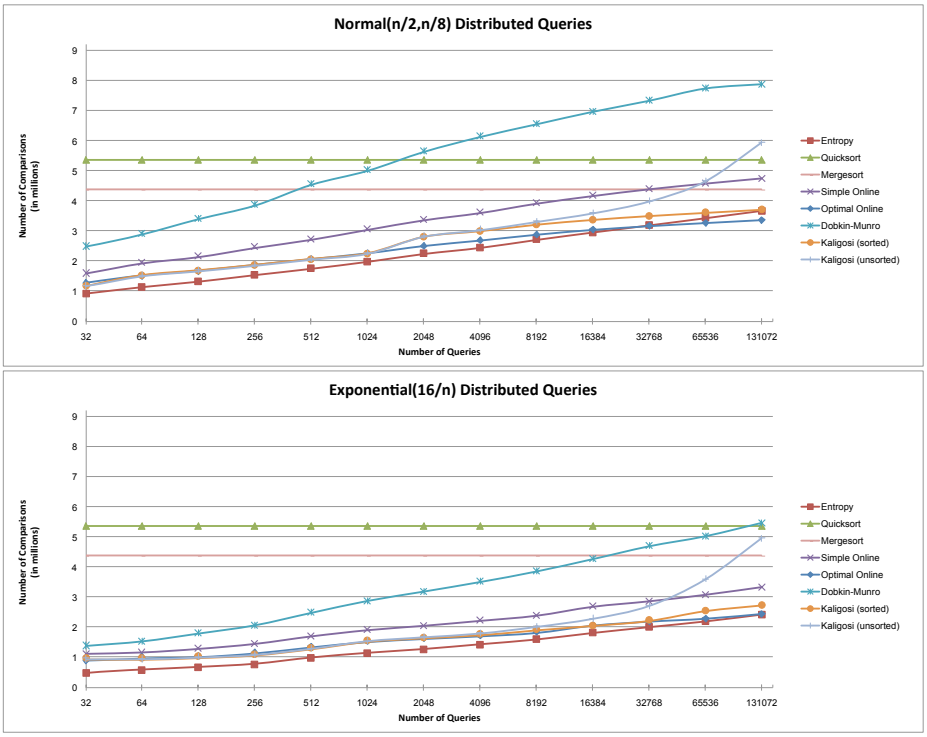

Fig. 3. Performance of multiselection algorithms on random input sequences using median of three pivot selection method, when the queries are distributed as indicated.

References

1. Alok Aggarwal and Jeffrey Scott Vitter. The input/output complexity of sorting and related problems. *Commun. ACM*, 31(9):1116–1127, 1988.
2. Jérémy Barbay, Ankur Gupta, S. Srinivasa Rao, and Jonathan Sorenson. Competitive online selection in main and external memory. *CoRR*, abs/1206.5336, 2012.
3. A. A. Belal and A. Elmasry. Distribution-sensitive construction of minimum-redundancy prefix codes. In *STACS*, pages 92–103, 2006.
4. M. Blum, R. W. Floyd, V. R. Pratt, R. L. Rivest, and R. E. Tarjan. Time bounds for selection. *J. Comput. Syst. Sci.*, 7(4):448–461, 1973.
5. J. Cardinal, S. Fiorini, G. Joret, R. M. Jungers, and J. I. Munro. An efficient algorithm for partial order production. In *STOC*, pages 93–100, 2009.
6. D. P. Dobkin and J. I. Munro. Optimal time minimal space selection algorithms. *J. ACM*, 28(3):454–461, 1981.
7. D. Dor and U. Zwick. Selecting the median. *SICOMP*, 28(5):1722–1758, 1999.
8. C. A. R. Hoare. Algorithm 65: find. *Commun. ACM*, 4(7):321–322, 1961.
9. R. M. Jiménez and C. Martínez. Interval sorting. In *ICALP*, pages 238–249, 2010.
10. K. Kaligosi, K. Mehlhorn, J. I. Munro, and P. Sanders. Towards optimal multiple selection. In *ICALP*, pages 103–114, 2005.
11. David G Kirkpatrick and Raimund Seidel. The ultimate planar convex hull algorithm. *SIAM J. Comput.*, 15(1):287–299, 1986.

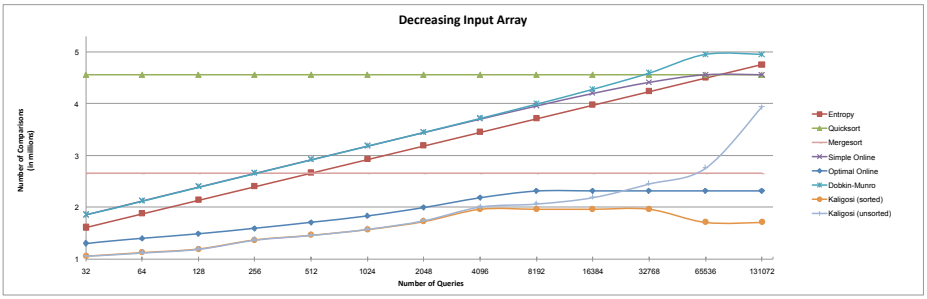

Fig. 4. Performance of multiselection algorithms on a decreasing input sequence using median of three pivot selection method, when the queries are distributed as indicated.

12. Helmut Prodinger. Multiple quickselect - Hoare's find algorithm for several elements. *Inf. Process. Lett.*, 56(3):123–129, 1995.
13. A. Schönhage, M. Paterson, and N. Pippenger. Finding the median. *J. Comput. Syst. Sci.*, 13(2):184–199, 1976.