Fast Deterministic Selection

Andrei Alexandrescu

The D Language Foundation, Washington, USA

Abstract

The selection problem, in forms such as finding the median or choosing the k top ranked items in a dataset, is a core task in computing with numerous applications in fields as diverse as statistics, databases, Machine Learning, finance, biology, and graphics. The selection algorithm Median of Medians, although a landmark theoretical achievement, is seldom used in practice because it is slower than simple approaches based on sampling. The main contribution of this paper is a fast linear-time deterministic selection algorithm MedianOfNinthers based on a refined definition of MedianOfMedians. A complementary algorithm MedianOfExtrema is also proposed. These algorithms work together to solve the selection problem in guaranteed linear time, faster than state-of-the-art baselines, and without resorting to randomization, heuristics, or fallback approaches for pathological cases. We demonstrate results on uniformly distributed random numbers, typical low-entropy artificial datasets, and real-world data. Measurements are open-sourced alongside the implementation at `https://github.com/andralex/MedianOfNinthers`.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases Selection Problem, Quickselect, Median of Medians, Algorithm Engineering, Algorithmic Libraries

Digital Object Identifier 10.4230/LIPIcs.SEA.2017.24

1 Introduction

The selection problem is widely researched and has numerous applications. Selection is finding the kth smallest element (also known as the kth order statistic): given an array A of length $|A| = n$, a non-strict order $\leq$ over elements of A, and an index k, the task is to find the element that would be in slot $A[k]$ if A were sorted. A variant that is the focus of this paper is *partition-based selection*: the algorithm must also permute elements of A such that $A[i] \leq A[k]\ \forall i, 0 \leq i < k$, and $A[k] \leq A[i]\ \forall i, k \leq i < n$.

Quickselect, originally called Find by its creator C.A.R. Hoare [17], is the selection algorithm most used in practice [29, 8, 25]. Like Quicksort [16], Quickselect relies on a separate routine Partition to divide the array into elements less than or equal to, and greater than or equal to, a specifically chosen element called the pivot. Unlike Quicksort which recurses on both subarrays left and right of the pivot, Quickselect only recurses on the side known to contain the kth smallest element.

The pivot choosing strategy is crucial for Quickselect because it conditions its performance between $O(n)$ and $O(n^2)$. Commonly used heuristics for pivot choosing – e.g. the median of 1–9 elements [30, 14, 3, 8] – work well on average but have high variance and do not offer worst-case guarantees. The "Median of Medians" pivot selection algorithm [4] solves the selection problem in guaranteed linear time. However, MedianOfMedians is seldom used in practice because it is intensive in element comparisons and especially swaps. Musser's Introselect algorithm [25] proceeds with a heuristics-informed Quickselect that monitors its own performance and only switches to MedianOfMedians if progress is slow. Contemporary implementations of selection (such as GNU C++ STL [13] and NumPy [28])

© Andrei Alexandrescu;
licensed under Creative Commons License CC-BY
16th International Symposium on Experimental Algorithms (SEA 2017).
Editors: Costas S. Iliopoulos, Solon P. Pissis, Simon J. Puglisi, and Rajeev Raman; Article No. 24; pp. 24:1–24:19

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

use QUICKSELECT or INTROSELECT in conjunction with simple pivot choosing heuristics. However, all current implementations are prone to high variance in run times even if we discount rare pathological cases. If heuristics provide poor pivot choices for the first 1–4 partition passes (when there is most data to process), the total run time increases strongly above its average [18, 9]. A selection algorithm that combines the principled nature and theoretical guarantees of MEDIANOFMEDIANS with good practical performance on average has remained elusive.

We seek to improve the state of the art in deterministic selection. First, we improve the definition of MEDIANOFMEDIANS to reduce comparisons and swaps. Second, we introduce *sampling* to improve average performance without compromising algorithm's linear asymptotic behavior. The resulting MEDIANOFNINTHERS algorithm is principled, practical, and competitive. Third, we introduce *adaptation* based on the observation that MEDIANOFMEDIANS is specialized to find the median, but may be used with any order statistic. That makes its performance degrade considerably when selecting order statistics away from the median. (These situations occur naturally even when searching for the median due to the way QUICKSELECT works.) We devise a simple and efficient partitioning algorithm MEDIANOFEXTREMA for searching for order statistics close to either end of the searched array. A driver algorithm QUICKSELECTADAPTIVE chooses dynamically the most appropriate partitioning method to find the median in linear time without resorting to randomization, heuristics, or fallback approaches for pathological cases. Most importantly, QUICKSELECTADAPTIVE does not compromise on efficiency – it was measured to be faster than a number of baselines, notably including GNU C++ `std::nth_element`. We open-sourced the implementation of QUICKSELECTADAPTIVE along with the benchmarks and datasets used in this paper [1].

The paper uses the customary pseudocode and algebraic notations. Divisions of integrals yield the floor as in many programming languages, e.g. $n/3$ or $\frac{n}{3}$ are $\lfloor \frac{n}{3} \rfloor$. We make the floor notation explicit when at least one operand is not integral. Arrays are zero-based. The length of array A is written as $|A|$. We denote with $A[a : b]$ (if $a < b$) a "slice" of A starting with $A[a]$ and ending with (and including) $A[b - 1]$. The slice is empty if $a = b$. Elements of the array are not copied – the slice is a bounded view into the given array.

2 Related Work

Hoare created the QUICKSELECT algorithm in 1961 [17], which still is the preferred choice of practical implementations, in conjunction with various pivot choosing strategies. Martínez et al. [22] analyze the behavior of QUICKSELECT with small data sets and propose stopping QUICKSELECT's recursion early and using sorting as an alternative policy below a cutoff, essentially a simple multi-strategy QUICKSELECT. The same authors [23] propose several adaptive sampling strategies for QUICKSELECT that take into account the index searched.

Blum, Floyd, Pratt, Rivest, and Tarjan created the seminal MEDIANOFMEDIANS algorithm [4], also known as BFPRT from its authors' initials. Subsequent work provided variants and improved on its theoretical bounds [12, 33, 34, 21, 5, 11]. Chen and Dumitrescu [6] propose REPEATEDSTEP (discussed in detail in §3), a variant of MEDIANOFMEDIANS that groups 3 or 4 elements (the original uses groups of 5 or more elements) and prove its linearity. Battiato et al. [2] describe Sicilian Median Selection, an algorithm for computing an approximate median that may be considered the transitive closure of REPEATEDSTEP.

Floyd and Rivest created the randomized SELECT algorithm [12] in 1975. Although further improved and benchmarked with favorable results by Kiwiel [19], at the time of this writing we found no implementation available online and no evidence of industry adoption.

Algorithm 1: QUICKSELECT.

Data: PARTITION, A, k with $|A| > 0, 0 \le k < |A|$
Result: Puts kth smallest element of A in $A[k]$ and partitions A around it.

```
1  while true do
2      p ← Partition(A);
3      if p = k then
4          return;
5      end
6      if p > k then
7          A ← A[0 : p];
8      else
9          k ← k − p − 1;
10         A ← A[p + 1 : |A|];
11     end
12 end
```

Algorithm 2: HOAREPARTITION.

Data: A, p with $|A| > 0, 0 \le p < |A|$
Result: p', the new position of $A[p]$; A partitioned at $A[p']$

```
1  Swap(A[p], A[0]);
2  a ← 1;
3  b ← |A| − 1;
4  loop: while true do
5      while true do
6          if a > b then break loop;
7          if A[a] ≥ A[0] then break;
8          a ← a + 1;
9      end
10     while A[0] < A[b] do b ← b − 1;
11     if a ≥ b then break;
12     Swap(A[a], A[b]);
13     a ← a + 1;
14     b ← b − 1;
15 end
16 Swap(A[0], A[a − 1]);
17 return a − 1;
```

3 Background: Quickselect, MedianOfMedians, and RepeatedStep

QUICKSELECT [8, 20] takes as parameters the input array A and the sought order statistic k. To facilitate exposition, our variant (Algorithm 1) also takes as parameter the partitioning primitive PARTITION, in a higher-order function fashion. PARTITION(A) chooses and returns an index $p \in \{0, 1, \ldots, |A| - 1\}$ called *pivot*, and also partitions A around $A[p]$. QUICKSELECT uses the pivot to either reduce the portion of the array searched from the left when $p < k$, reduce it from the right when $p > k$, or end the search when $p = k$.

The running time of QUICKSELECT is linear if PARTITION(A) returns in linear time a pivot p ranked within a fixed fraction $0 < f < 1$ from either extremum of A. Most pivot selection schemes use *heuristics* by choosing a pivot unlikely to be close to an extremum in conjunction with HOAREPARTITION, which partitions the array in $O(n)$. (Many variants of Hoare's PARTITION algorithm [15] exist; Algorithm 2 is closer to the implementation we used, than to the original definition.) Heuristics cannot provide a good worst-case run time guarantee for QUICKSELECT, but perform well on average.

MEDIANOFMEDIANS, prevalently implemented as shown in BFPRTBASELINE (Algorithm 3) [10, 7, 26, 32], spends more time to guarantee good pivot choices. The algorithm first computes medians of groups of 5 elements for a total of $\frac{|A|}{5}$ groups. The rote routine MEDIAN5(A, a, b, c, d, e) swaps the median of $A[a], \ldots, A[e]$ into $A[c]$. Computing the median of these medians yields a pivot p with a useful property. There are $\frac{|A|}{10}$ elements less than or equal to $A[p]$, but each of those is the median of 5 distinct elements, so $A[p]$ is not smaller than at least $\frac{3|A|}{10}$ elements. By symmetry, $A[p]$ is not greater than at least $\frac{3|A|}{10}$ elements.

Algorithm 3: BFPRTBaseline.

```
   Data: A
   Result: Pivot 0 ≤ p < |A|; A partitioned at A[p]
 1 if |A| < 5 then
 2 |  return HoarePartition(A, |A|/2);
 3 end
 4 i ← 0;  j ← 0;
 5 while i + 4 < |A| do
 6 |  Median5(A, i, i + 1, i + 2, i + 3, i + 4);
 7 |  Swap(A[i + 2], A[j]);
 8 |  i ← i + 5;
 9 |  j ← j + 1;
10 end
11 Quickselect(BFPRTBaseline, A[0 : j], j/2);
12 return HoarePartition(A, j/2);
```

Algorithm 4: RepeatedStep.

```
   Data: A
   Result: Pivot 0 ≤ p < |A|; A partitioned at A[p]
 1 if |A| < 9 then
 2 |  return HoarePartition(A, |A|/2);
 3 end
 4 i ← 0;  j ← 0;
 5 while i + 2 < |A| do
 6 |  Median3(A, i, i + 1, i + 2);
 7 |  Swap(A[i + 1], A[j]);
 8 |  i ← i + 3;
 9 |  j ← j + 1;
10 end
11 i ← 0;  m ← 0;
12 while i + 2 < j do
13 |  Median3(A, i, i + 1, i + 2);
14 |  Swap(A[i + 1], A[m]);
15 |  i ← i + 3;
16 |  m ← m + 1;
17 end
18 Quickselect(RepeatedStep, A[0 : m], m/2);
19 return HoarePartition(A, m/2);
```

Selection is initiated by invoking Quickselect(BFPRTBaseline, A, k). To prove linearity, let us look at the worst-case number of comparisons depending on $n = |A|$:

$$C(n) \leq C\left(\frac{n}{5}\right) + C\left(\frac{7n}{10}\right) + \frac{6n}{5} + n \tag{1}$$

where the first term accounts for the median of medians computation, the second is the time taken by Quickselect after partitioning, the third is the cost of computing the medians of five, and the last is the cost of HoarePartition. Consequently $C(n) \leq 22n$.

The number of swaps is also of interest. BFPRTBaseline uses a common optimization [10, 7, 26, 32] – it reuses the first quintile of A for storing the medians. $S(n)$ satisfies:

$$S(n) \leq S\left(\frac{n}{5}\right) + S\left(\frac{7n}{10}\right) + \frac{7n}{5} + \frac{n - \frac{n}{10}}{2}\,. \tag{2}$$

The terms correspond to those for $C(n)$. Consequently $S(n) \leq \frac{37n}{2}$. However, neither bound is tight, meaning they only have advisory value; given that generating worst-case data for MedianOfMedians remains an open problem, we use empirical benchmarks (§ 7) to compare the performance of all algorithms discussed.

Recently Chen and Dumitrescu [6] proposed linear-time MedianOfMedians variants that use groups of 3 or 4 elements, disproving long-standing conjectures to the contrary. Algorithm 4 shows the pseudocode of their RepeatedStep algorithm with a group size of 3.

Key to the algorithm is that the median of medians step is repeated, thus choosing the pivot as the median of medians of medians of three (sic). This degrades the pivot's quality, placing it within $\frac{2n}{9}$ elements from either extremum of A. However, the median of medians computation only needs to recurse on $\frac{n}{9}$ elements. Intuitively, trading off some pivot quality for faster processing in MedianOfMedians is a good idea for competing with imprecise but fast pivot heuristics. $C(n)$ for Quickselect(RepeatedStep, A, k) satisfies ($n = |A|$):

$$C(n) \leq C\left(\frac{n}{9}\right) + C\left(\frac{7n}{9}\right) + \frac{3n}{3} + \frac{3n}{9} + n \tag{3}$$

where the first term is the cost of computing the median of medians of medians, the second is the worst-case time spent processing the remaining elements, and the last three terms account respectively for computing the medians of 3, computing the medians of 3 medians of 3, and the partitioning. Consequently $C(n) \le 21n$. For $S(n)$, each median of three uses at most 1 swap, to which we add 1 for swapping the median to the front of the array. Partitioning costs at most $\frac{n}{2}$ swaps in general, but the first $\frac{n}{18}$ elements are not swapped:

$$S(n) \le S\left(\frac{n}{9}\right) + S\left(\frac{7n}{9}\right) + \frac{2n}{3} + \frac{2n}{9} + \frac{n - \frac{n}{18}}{2} \tag{4}$$

which solves to $S(n) \le \frac{49n}{4}$. These bounds prove linearity but are not necessarily tight.

4 Partitioning During Pivot Computation

We now set out to improve these algorithms. One starting observation is that heuristics-based partition uses a $O(1)$ step (picking a pivot) followed by a linear pass (invoking HoarePartition). In contrast, BFPRTBaseline and RepeatedStep make *two* linear passes: one for finding the pivot, and one for the partitioning step (also using HoarePartition). Therefore, algorithms in the MedianOfMedians family are at a speed disadvantage.

This motivates one key insight: we aim to *integrate* the two steps, i.e. make the comparisons and swaps performed during pivot computation also count toward partitioning. RepeatedStep organizes the array in groups of 3 and computes the median of each group; then it repeats the same procedure for the medians of three. That imparts a non-trivial implicit structure onto the input array in addition to computing the pivot. However, that structure is not used by HoarePartition. Ideally, that structure should be embedded in the array in a form favorable to the subsequent partitioning step.

Our approach (MedianOfNinthersBasic shown in Algorithm 5) is to make the small groups *non-contiguous* and choose their placement in a manner that is advantageous for the partitioning step, so as to avoid comparing and swapping elements more than once. To that end, we place the subarray of medians in the very middle of A, more precisely in the 5th 9^{th}ile of the array. (Recall that RepeatedStep computes a subarray of medians of medians with $\frac{|A|}{9}$ elements and recurses against it to compute its median.)

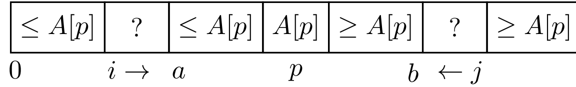
Also, instead of executing two loops, we execute a single pass that ensures the same postcondition. This is done with Tukey's Ninther routine [31, 3], which takes 9 array elements, computes the medians of 3 disjoint groups of 3, and yields the median of those 3 medians. Specifically, $\text{Ninther}(A, i_1, i_2, i_3, i_4, i_5, i_6, i_7, i_8, i_9)$ computes the index of the median of $A[i_1]$, $A[i_2]$, and $A[i_3]$ into i'_1, the index of the median of $A[i_4]$, $A[i_5]$, and $A[i_6]$ into i'_2, and the index of the median of $A[i_7]$, $A[i_8]$, and $A[i_9]$ into i'_3. Then it swaps $A[i_5]$ with the median of $A[i'_1]$, $A[i'_2]$, and $A[i'_3]$. After this operation, $A[i_5]$ is no less than at least 3 elements and no greater than at least 3 other elements among $A[i_1], \ldots, A[i_9]$. We use Ninther against groups that pick 4 elements from the front of A, one from the mid 9^{th}ile (which receives the median), and 4 from the right of that 9^{th}ile.

These changes have important theoretical and practical advantages. First, Ninther computes the same medians of 3 medians of 3 as the first two loops in RepeatedStep using the same number of comparisons (12 per group of 9) but with 0–1 swaps instead of 0–3. Second, a single pass is better than the two successive loops in RepeatedStep. Third, after recursing to Quickselect against $A\left[\frac{4|A|}{9} : \frac{5|A|}{9}\right]$, the mid 9^{th}ile is already partitioned properly around the pivot; there is no need to visit it again. That way, the medians computation step contributes one ninth of the final result at no additional cost.

Algorithm 5: MedianOfNinthersBasic.

```
Data: A
Result: Pivot p, 0 ≤ p < |A|; A partitioned at p
1  if |A| < 9 then
2  |   return HoarePartition(A, |A|/2);
3  end
4  f ← |A|/9;
5  for i ← 4f through 5f − 1 do
6  |   l ← i − 4f;
7  |   r ← i + 5f;
8  |   Ninther(A, l, l + 1, l + 2, l + 3, i,
   |     r, r + 1, r + 2, r + 3);
9  end
10 Quickselect(RepeatedStepBasic,
     A[4f : 5f], f/2);
11 return
     ExpandPartition(A, 4f, 4f + f/2, 5f − 1);
```

The pivot computation leaves the array well suited for partitioning by visiting 9^{th}iles 1–4 and 6–9 (subarrays $A\left[0 : \frac{4|A|}{9}\right]$ and $A\left[\frac{5|A|}{9} : |A|\right]$). This work is carried by ExpandPartition (not shown, available online [1]), a modified HoarePartition algorithm that takes into account the already-partitioned subarray around the pivot. The call ExpandPartition(A, a, p, b) proceeds by the following scheme, starting with $i = 0$ and $j = |A| - 1$ and moving them toward a and b, respectively:



The procedure swaps as many elements as possible between $A[i : a]$ and $A[b + 1 : j + 1]$ because that is the most efficient use of swapping – one swap puts two elements in their final slots. There may be some asymmetry (one of i and j reaches its limit before the other) so the pivot position may shift to the left or right. ExpandPartition returns the final position of the pivot, which BFPRTImproved forwards to the caller. ExpandPartition(A, a, b) performs $a + |A| - b$ comparisons and at most $\max(a, |A| - b)$ swaps.

The number of comparisons $C(n)$ satisfies the recurrence:

$$C(n) \le C\left(\frac{n}{9}\right) + C\left(\frac{7n}{9}\right) + n + \frac{n}{3} + \frac{8n}{9}\,. \tag{5}$$

The only difference from the corresponding bound of RepeatedStep is the last term, which is slightly smaller in this case because we don't revisit the middle 9^{th}ile during partitioning. The recurrence solves to $C(n) \le 20n$.

For $S(n)$, each Ninther contributes at most 1 swap per 9 elements. In the worst case ExpandPartition needs to swap $\frac{4n}{9}$ elements from the left side with $\frac{4n}{9}$ elements from the right. However, the swaps don't sum because one swap operation takes care of two elements wherever possible. So the number of swaps performed by ExpandPartition is at most $\frac{4n}{9}$.

$$S(n) \le S\left(\frac{n}{9}\right) + S\left(\frac{7n}{9}\right) + \frac{n}{9} + \frac{4n}{9}\,. \tag{6}$$

Consequently $S(n) \le 5n$, a sizeable improvement over RepeatedStep. Integrating pivot searching with partitioning is crucial for improving efficiency. MedianOfNinthersBasic not only has much better performance, but also allows further optimizations to build upon it.

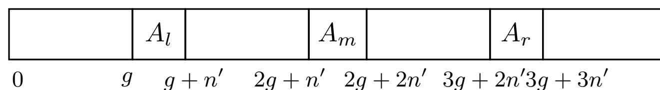
5 Sampling Without Compromising Linearity

We are now ready to introduce MedianOfNinthers (Algorithm 6). It uses a hyperparameter $0 < \phi \le 1$ from which it derives a subsample size $n' = \left\lfloor \frac{\phi|A|}{3} \right\rfloor$ and a gap length $g =$

Algorithm 6: MedianOfNinthers.

```
   Data: A
   Result: Pivot p, 0 ≤ p < |A|; A partitioned at p
 1 n ← |A|;
 2 n' ← ⌊φn/3⌋;
 3 if n' < 3 then
 4 |  return HoarePartition(A, |A|/2);
 5 end
 6 g ← (n − 3n')/4;
 7 A_m ← A[2g + n' : 2g + 2n'];
 8 l ← g;
 9 m ← 2g + n';
10 r ← 3g + 2n';
11 for i ← 0 through n'/3 − 1 do
12 |  Ninther(A, l, m, r, l + 1, m + n'/3, r + 1,
   |    l + 2, m + 2n'/3, r + 2, m, m + n'/3,
   |    m + 2n'/3);
13 |  m ← m + 1;
14 |  l ← l + 3;
15 |  r ← r + 3;
16 end
17 Quickselect(MedianOfNinthers, A_m, n'/2);
18 return ExpandPartition(A,
      2g + n', 2g + n' + n'/2, 2g + 2n');
```

$\frac{|A|-3n'}{4}$, after which it chooses three equidistant disjoint subarrays from A as follows: $A_l = A[g : g+n']$, $A_m = A[2g+n' : 2g+2n']$, and $A_r = A[3g+2n' : 3g+3n']$. The three subarrays have length n' each, and the gaps around them have length g each (except for the last gap which is longer by $(|A| - 3n') \mod 4$), as illustrated below.



The plan is to save time by computing the pivot solely by looking at A_l, A_m, and A_r instead of visiting the entire array. The approach is essentially to perform the same algorithm as MedianOfNinthersBasic against the conceptual concatenation of A_l, A_m, and A_r.

The challenge is choosing an appropriate iteration schedule for picking the 9 elements to pass to Ninther. We do so by pairing successive triples of adjacent elements in A_l with the triple $A_m[i]$, $A_m[i+n'/3]$, $A_m[i+2n'/3]$ (having i range in $\{0, 1, \ldots, n'/3 - 1\}$) and with successive triples of adjacent elements in A_r. The result of each ninther is swapped to $A_m[2i]$, i.e. to the second tertile of A_m, which in turn is in the middle of A. After the loop, the second tertile of A_m contains the medians needed for recursion.

Let us prove linearity of Quickselect(MedianOfNinthers, A, k) by computing an upper bound of the number of comparisons $C(n)$ from the recurrence:

$$C(n) \le C\left(\frac{\phi n}{9}\right) + C\left(n - \frac{2\phi n}{9}\right) + \frac{12\phi n}{9} + \frac{n(9-\phi)}{9}. \tag{7}$$

The first term accounts for the recursive call to Quickselect, which processes that many elements. The second term accounts for processing the remainder of the array (as reasoned for RepeatedStep, in the worst case $\frac{2\phi n}{9}$ elements are eliminated in one partitioning step), the third is the cost of the first loop (12 comparisons for each Ninther call, of which there are $\frac{\phi n}{9}$), and the last is the cost of ExpandPartition. Consequently $C(n) \le \frac{(11\phi+9)n}{\phi}$. As expected, the number of comparisons goes up as ϕ goes down. For swaps we obtain (with the same term positions):

$$S(n) \le S\left(\frac{\phi n}{9}\right) + S\left(n - \frac{2\phi n}{9}\right) + \frac{\phi n}{9} + \frac{(9-\phi)n}{18} \tag{8}$$

resulting in the bound $S(n) \le \frac{(\phi+9)n}{2\phi}$. Although this result is theoretically unremarkable, it is attractive engineering-wise. It means we can fine-tune the tradeoff between the time spent computing the pivot and the quality of the pivot, without ever losing the linearity of the

algorithm. An entire spectrum opens up between the constant sample size used by heuristics and the full scan performed by all MedianOfMedians variations discussed so far.

6 Adaptation: MedianOfExtrema

Finding the kth order statistic in A is most difficult for $k = \frac{|A|}{2}$. However, sometimes k may be closer to one side of A than to its middle. Skewed values of k are possible not only when requested by the caller (e.g. fetch the 1000 best-ranked items from a large input), but also while computing the median proper. Recall that Quickselect (Algorithm 1) reduces $|A|$ progressively in a loop, which changes the relationship between $|A|$ and k. A few iterations bring the median search to an endgame of chasing a k close to 0 or $|A|$.

Quickselect(A, k) runs faster for skewed values of k than for $k = \frac{|A|}{2}$ because any pivot choosing method has a higher likelihood of finding a good pivot (one that allows eliminating a large fraction of the searched array). This puts elaborate pivot computing methods at a disadvantage compared to simple heuristics, so such situations are worth addressing. To that end we define a specialized algorithm MedianOfExtrema with two variants, MedianOfMinima for order statistics close to 0, and MedianOfMaxima for order statistics close to $|A|$. In the following we limit the discussion to MedianOfMinima. The corresponding variant MedianOfMaxima is defined analogously.

For small values of k relative to $|A|$, the partition function should *not* find a pivot to the left of k because that would only eliminate a small portion of the input. Martínez et al. discuss this risk for their related proportional-of-3 strategy [23]. So we require that MedianOfMinima must find a pivot not smaller than k.

MedianOfMinima (Algorithm 7) is based on the following intuition. MedianOfMedians computes medians of small groups and takes their median to find a pivot approximating the median of A. In this case, we pursue an order statistic skewed to the left, so instead of the median of each group, we compute its *minimum*; then, we obtain the pivot by computing the median of those groupwise minima. By construction, the pivot's rank will be shifted to the left of the true median. This is an easier task, too, because computing the minimum of a group is simpler and computationally cheaper than computing the group's median. We place these minima at A's front so they don't need to be swapped again.

The outer loop in Algorithm 7 organizes A such that its first $2k$ slots contain the minima of groups of size $\gamma = \frac{|A|}{2k}$ elements. Specifically, $A[0]$ receives the minimum over $A[0]$ and the first group of $\gamma - 1$ elements of $A\,[2k : |A|]$; $A[1]$ receives the minimum over $A[1]$ and the second group over $\gamma - 1$ elements of $A\,[2k : |A|]$; and so on through $A[2k-1]$, which receives the minimum over $A[2k-1]$ and the $2k^{\text{th}}$ group of $\gamma - 1$ elements of $A\,[2k : |A|]$. This permutation ensures that for each element in $A\,[0 : 2k]$, there are at least $\gamma - 1$ additional elements in $A\,[2k : |A|]$ greater than or equal to it.

The next steps compute the kth order statistic over $A\,[0 : 2k]$ (i.e. the upper median of the subarray of minima, as the name of the algorithm suggests), and uses the obtained $A[k]$ as pivot to expand the obtained partition to the entire array A. The recursive call replaces Quickselect with QuickselectAdaptive. The latter (fully specified in the next section) chooses to use either MedianOfMinima, MedianOfNinthers, or MedianOfMaxima depending on the ratio of k to $|A|$.

Let us assess the quality of the pivot p obtained by calling MedianOfMinima(A, k), i.e. the length of the subarray we can eliminate from the search after one call to MedianOfMinima. Obviously $p \geq k$ because the recursive call to QuickselectAdaptive places k elements to the left of $A[k]$ that are no greater than it. In addition, each of the k

Algorithm 7: MedianOfMinima.

```
   Data: A, 0 < k < |A|/6
   Result: Pivot p, k ≤ p ≤ |A|/2; A partitioned at A[p]
 1 if |A| = 1 then
 2 |  return 0;
 3 end
 4 γ ← |A|/2k;
 5 k ← G;
 6 for i ← 0 through 2k − 1 do
 7 |  m ← 2k + i(γ − 1);
 8 |  for j ← m + 1 through m + γ − 1 do
 9 |  |  if A[j] < A[j - 1] then
10 |  |  |  m ← j;
11 |  |  end
12 |  end
13 |  if A[m] < A[i] then
14 |  |  Swap(A[i], A[m]);
15 |  end
16 end
17 QuickselectAdaptive(A[0 : 2k], k);
18 return ExpandPartition(A, 0, k, 2k);
```

elements in subarray $A\,[k : 2k]$ is greater than or equal to the pivot; but by construction, for each of these elements there are $\gamma - 1$ others greater than or equal to the pivot in the subarray $A\,[2k : |A|]$. It follows that at least $k\left\lfloor \frac{|A|}{2k} \right\rfloor$ elements of A are greater than or equal to the pivot. Through algebraic manipulation we get $k\left\lfloor \frac{|A|}{2k} \right\rfloor \geq \frac{|A|-2k+1}{2} \geq \frac{|A|}{2} - k$ so the call MedianOfMinima(A, k) yields a pivot that allows the elimination of at least $\frac{|A|}{2} - k$ elements from the search.

These elements are eliminated from the search at the next iteration of QuickselectAdaptive, and we want to make sure the cost of the computation stays within linear bounds. The cost of eliminating these elements is $n - k$ for the minima computations plus a recursion on $2k$ elements. We conservatively require by the Master Theorem [27] that the recursion on $2k$ elements eliminates more than $2k$ elements, so $\frac{n}{2} - k > 2k$, which results in the requirement $k < \frac{n}{6}$. Conversely, for the MedianOfMaxima the threshold is $k > \frac{5n}{6}$. These thresholds work well in practice and are used in the implementation and experiments.

6.1 Choosing Strategy Dynamically: QuickselectAdaptive

In order to implement adaptation, we need to dynamically choose the partitioning algorithm from among MedianOfMinima, MedianOfNinthers, and MedianOfMaxima. A good place to decide strategy is the Quickselect routine itself, which has access to the information needed and drives the selection process. Before each partitioning step, the appropriate partitioning algorithm is chosen depending on the relationship between $|A|$ and k. After partitioning, both A and k are modified and a new decision is made, until the search is over. QuickselectAdaptive (Algorithm 8) embodies this idea.

7 Experiments and Results

For the implementation [1] we choose the sampling constant for MedianOfNinthers $\phi = \frac{1.0}{64.0}$ for arrays up to 2^{17} elements and $\phi = \frac{1.0}{1024.0}$ for larger arrays. Performance is not highly dependent on ϕ, for example there are no dramatic changes when halving or doubling ϕ. However, sampling is needed; with $\phi = 1$, the algorithm falls behind the best baseline. The data sets used are:

- *random*: uniformly-distributed random floating-point numbers.
- *random01*: $\frac{n}{2}$ zeros and $\frac{n}{2}$ ones, shuffled randomly. This puts to test algorithms' ability to cope with many duplicates.

Algorithm 8: QUICKSELECTADAPTIVE.

Data: A, k with $0 \leq k < |A|$
Result: Puts kth smallest element of A in $A[k]$ and partitions A around it.

```
1  while true do
2    if |A| ≤ 16 then
3      p ← HoarePartition(A, k);
4    else if 6k < |A| then
5      p ← MedianOfMinima(A, k);
6    else if 6k > 5|A| then
7      p ← MedianOfMaxima(A, k);
8    else
9      p ← MedianOfNinthers(A);
10   end
11   if p = k then return;
12   if p > k then
13     A ← A[0 : p];
14   else
15     i ← k − p − 1;
16     A ← A[p + 1 : |A|];
17   end
18 end
```

- *m3killer*: Musser's "median-of-3-killer sequence" [25].
- *organpipe*: numbers $0, 1, 2, \ldots, \frac{n}{2} - 1, \frac{n}{2} - 1, \ldots, 1, 0$.
- *sorted*: numbers $0, 1, 2, \ldots, n - 1$.
- *rotated*: numbers $1, 2, \ldots, n - 1, 0$.
- *googlebooks*: We complement artificial data sets with a real-world task – compute the median frequency of 1-grams (words) in different languages in the Google Ngrams corpus [24]. These data sets consist of between $5.4M$ and $20M$ 1-grams along with their frequencies. Words have been grouped per year with summing of frequencies. The part-of-speech annotations have been kept. The languages processed are English (eng), Russian (rus), French (fre), German (ger), Italian (ita), and Spanish (spa).

The artificial data sizes increase exponentially from $10,000$ to $10,000,000$ with step $\sqrt{10}$.

For baseline algorithms, we chose the pivot strategies most competitive and in prevalent industrial use: MEDIANOF3RANDOMIZED (which chooses the pivot as the median of three random array elements), NINTHER (Tukey's ninther deterministic), and GNUINTROSELECT, GNU's implementation of the C++ standard library function `std::nth_element`. (To avoid clutter, we did not plot other heuristics that performed worse, such as single random pivot, ninther randomized, and median of 3 and 5 elements.)

First, we benchmarked run times on a desktop computer (Intel Core i7 3.6 GHz) against arrays of 64-bit floating point data. The compiler used was `gcc` version 5.4.0 invoked with `-O4 -DNDEBUG`. We ran each experiment 102 times, eliminated the 2 longest measurements to account for outliers caused by cache warmup and other additive noise, and took the average of the remaining timings. For random data, the input was randomly shuffled before each run.

GNUINTROSELECT is our main baseline because it is an independent and mature implementation that has received extensive use and scrutiny. All speed benchmarks plotted are normalized such that GNUINTROSELECT has relative speed $y = 1.0$, so as to make it easier to compare algorithms across widely different input sizes. Larger numbers are better, e.g. $y = 2.0$ means twice the speed.

Figure 1 plots the run times of the algorithms tested for finding the median in arrays of uniformly-distributed floating point numbers. (All run times are given in Appendix A.) QUICKSELECTADAPTIVE is faster by a large margin for all data sizes.

For the *random01* data set (Figure 2), again the ranking puts QuickselectAdaptive first (albeit by a smaller margin), followed by GNUIntroselect. There is no noticeable difference between the performances of the two other algorithms.

The *m3killer* dataset (Figure 3) has GNUIntroselect as winner for most data sizes. The reason is that the median-of-3-killer pattern was intended to cause quadratic behavior to algorithms choosing the pivot as the median of $A[0]$, $A[|A|/2]$, and $A[|A|-1]$, but GNUIntroselect uses $A[1]$ instead of $A[0]$. Therefore, the first pivot chosen by GNUIntroselect is the median of $|A|/2+1$, 2, and $|A|$, which is exactly the upper median of the entire array. Therefore, all other algorithms compete against one single pass through a highly optimized implementation of HoarePartition.

The *organpipe* dataset (Figure 4) features Ninther and QuickselectAdaptive as best performers. Ninther makes good median choices because its sample positions are close to the actual median (which is at the 25th and 75th percentiles). QuickselectAdaptive's sampling strategy also finds the median with relative ease. The same pattern can be noted on the *sorted* dataset (Figure 5).

Figure 6 shows one interesting pathological case: GNUIntroselect is up to 30x slower than the other algorithms, and the gap grows with the size of the data set. This is surprising because the *rotated* data set (essentially a sorted sequence with a small value at the end) may plausibly occur in practice (e.g. a sorted array with one appended element).

Figure 7 compares performance for computing the median frequency of words in the Google Ngrams corpus [24]. QuickselectAdaptive outperforms all baselines.

7.1 Measuring comparisons, swaps, and variance of run times

Next, we tested the hypothesis (made in the introduction) that QuickselectAdaptive has lower variance than heuristics-based algorithms, by measuring and comparing the coefficient of variation $\frac{\sigma}{\mu}$ (standard deviation divided by mean) of run times. (Comparing σ values directly would not be appropriate because they characterize distributions with different averages.) We also measured the number of comparisons and swaps. The worst-case number of comparisons for computing the median has the lower bound $C(n) = (2+\epsilon)n$, where $\epsilon > 0$ is a constant [11]. On random data, the expected number of swaps by an optimal median selection algorithm is $S(n) = \frac{n}{4}$ (statistically half of the elements on either side of the median need to be swapped).

For the algorithms tested, Figure 8 shows comparisons per element $\frac{C(n)}{n}$, Figure 9 shows swaps per element $\frac{S(n)}{n}$, and Figure 12 shows the coefficient of variation $\frac{\sigma}{\mu}$ of the algorithms tested for 100 trial runs against the *random* dataset ($n = 10,000,000$). Data has been shuffled between runs.

The coefficient of variation $\frac{\sigma}{\mu}$ of run times of QuickselectAdaptive is one order of magnitude smaller than that of the baselines for medium and large data sets. The results for $C(n)$ and $S(n)$ indicate that QuickselectAdaptive makes a large reduction in the gap between theory and practice. Figure 10 and 11 reveal that the improvements also apply to real-world data. (Appendix A provides detailed numeric results.) Also, we speculate that further improvements will likely be difficult. $C(n)$ may still be improved significantly because $(2+\epsilon)n$ describes the worst, not average, case, but $S(n)$ is virtually at its theoretical optimum.

Acknowledgements. Thanks to Timon Gehr, Ivan Kazmenko, Scott Meyers, and Todd Millstein who reviewed drafts of this document. Teppo Niinimäki provided support code.

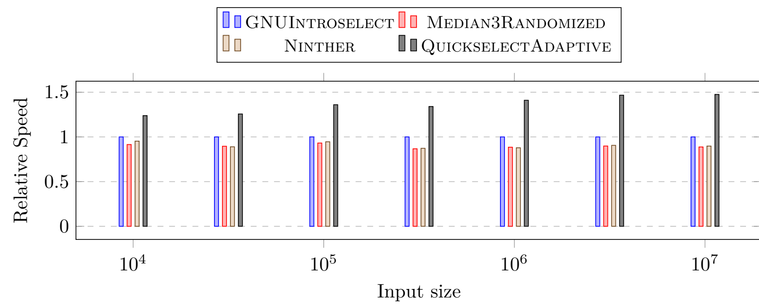


Figure 1 Speed relative to GNUIntroSelect (*random* dataset)

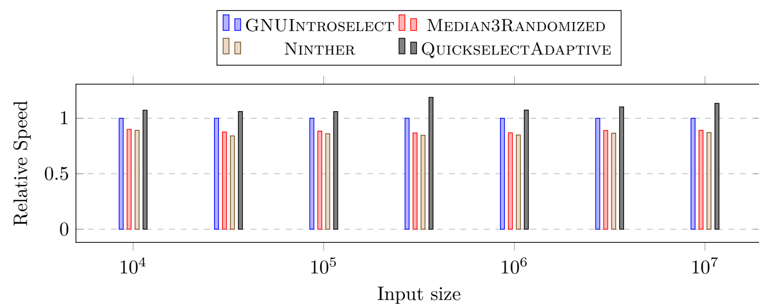


Figure 2 Speed relative to GNUIntroSelect (*random01* dataset).

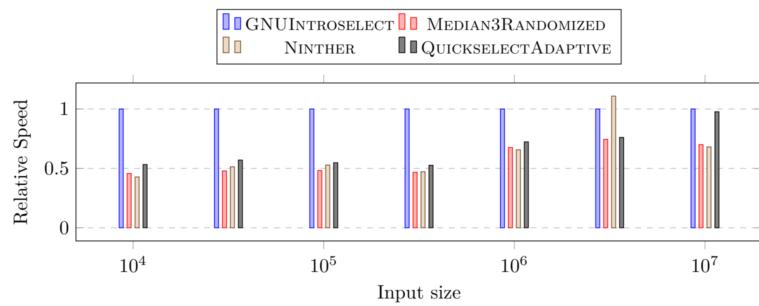


Figure 3 Speed relative to GNUIntroSelect (*m3killer* dataset).

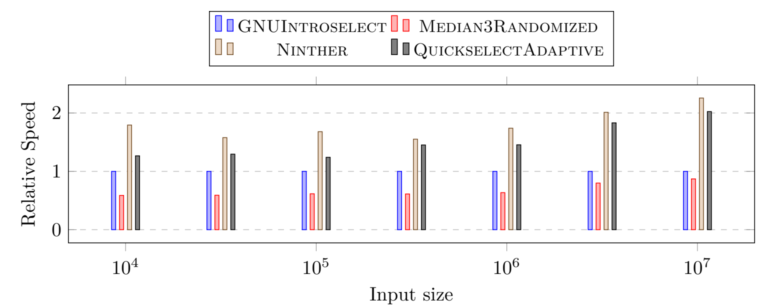


Figure 4 Speed relative to GNUIntroSelect (*organpipe* dataset).

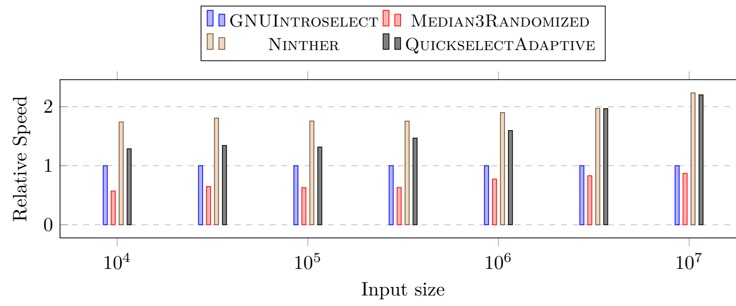


Figure 5 Speed relative to GNUIntroSelect (*sorted* dataset).

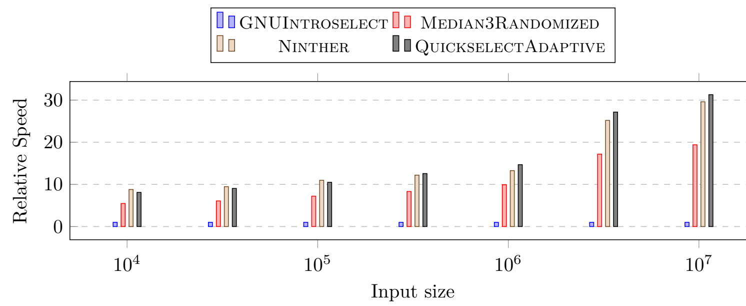


Figure 6 Speed relative to GNUIntroSelect (*rotated* dataset).

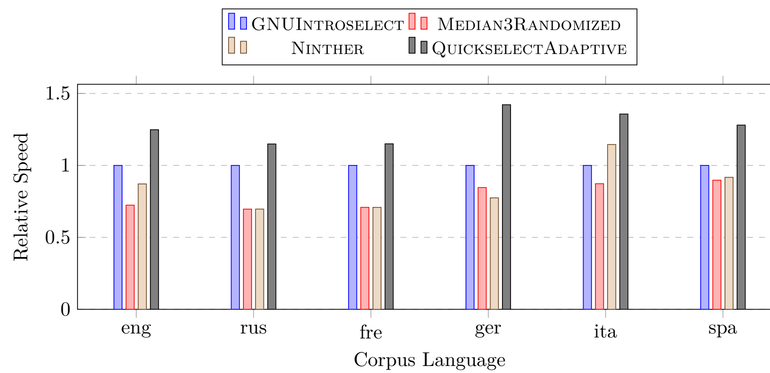


Figure 7 Speed relative to GNUIntroSelect (*googlebooks* dataset).

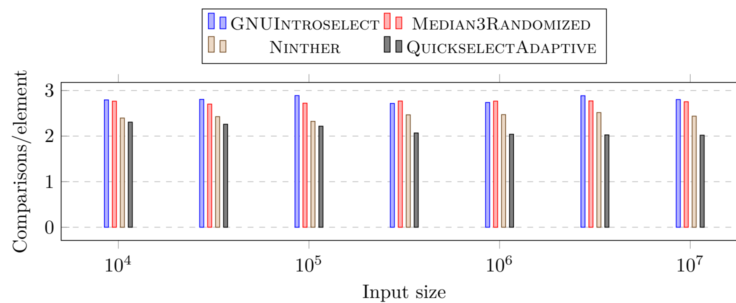


Figure 8 Comparisons per element (*random* dataset).

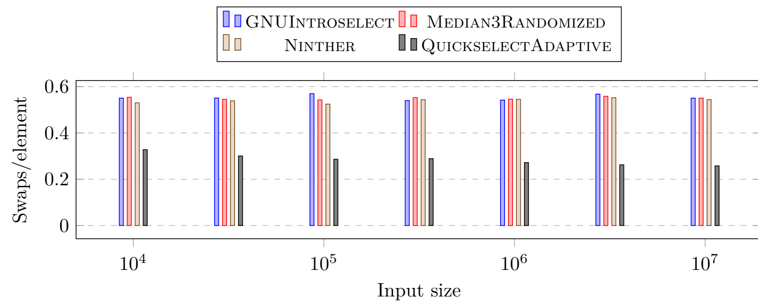


Figure 9 Swaps per element (*random* dataset).

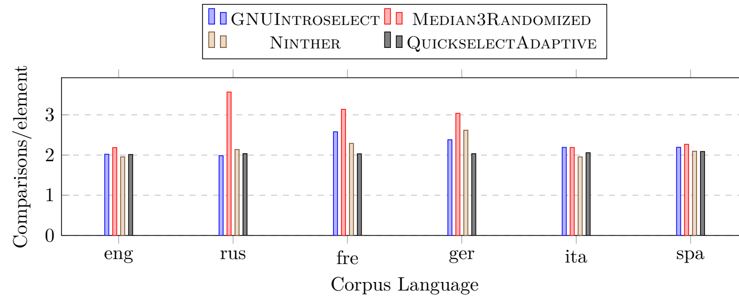


Figure 10 Comparisons per element (*googlebooks* dataset).

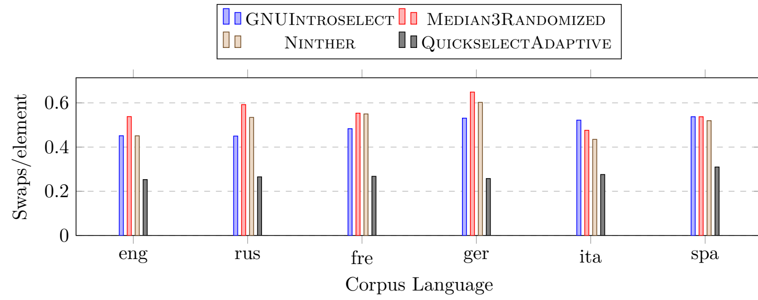


Figure 11 Swaps per element (*googlebooks* dataset).

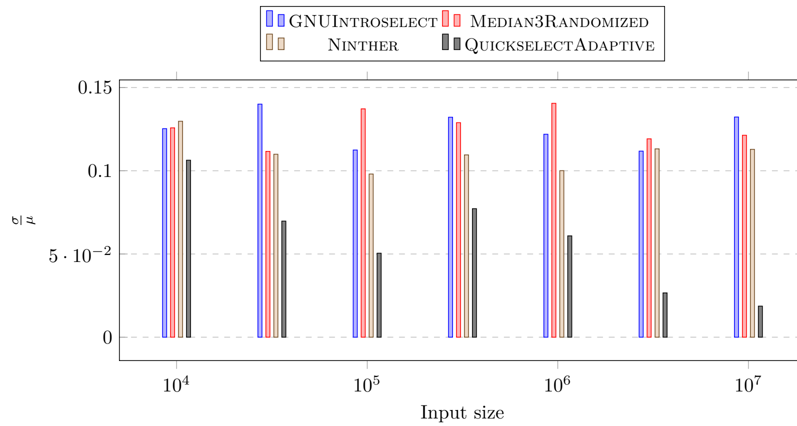


Figure 12 Coefficient of variation (*random* dataset).

References

1. Andrei Alexandrescu. Median of ninthers: code, data, and benchmarks. `https://github.com/andralex/MedianOfNinthers`, 2017.
2. Sebastiano Battiato, Domenico Cantone, Dario Catalano, Gianluca Cincotti, and Micha Hofri. An efficient algorithm for the approximate median selection problem. In *Algorithms and Complexity*, pages 226–238. Springer, 2000.
3. Jon L Bentley and M Douglas McIlroy. Engineering a sort function. *Software: Practice and Experience*, 23(11):1249–1265, 1993.
4. Manuel Blum, Robert W. Floyd, Vaughan Pratt, Ronald L. Rivest, and Robert E. Tarjan. Time bounds for selection. *J. Comput. Syst. Sci.*, 7(4):448–461, August 1973. `doi:10.1016/S0022-0000(73)80033-9`.
5. Svante Carlsson and Mikael Sundström. *Algorithms and Computations: 6th International Symposium, ISAAC'95 Cairns, Australia, December 4–6, 1995 Proceedings*, chapter Linear-time in-place selection in less than 3n comparisons, pages 244–253. Springer Berlin Heidelberg, Berlin, Heidelberg, 1995. `doi:10.1007/BFb0015429`.
6. Ke Chen and Adrian Dumitrescu. Select with groups of 3 or 4. In *Algorithms and Data Structures: 14th International Symposium, WADS*, 2015.
7. Derrick Coetzee. An efficient implementation of Blum, Floyd, Pratt, Rivest, and Tarjan's worst-case linear selection algorithm, 2004. URL: `http://moonflare.com/code/select/select.pdf`.
8. Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.
9. Jean Daligault and Conrado Martínez. On the variance of quickselect. In *Proceedings of the Meeting on Analytic Algorithmics and Combinatorics*, pages 205–210. Society for Industrial and Applied Mathematics, 2006.
10. Kevin Dinkel and Andrew Zizzi. Fast median finding on digital images. In *AIAA Regional Student Paper Conference*, 2012.
11. Dorit Dor and Uri Zwick. Median selection requires $(2+\epsilon)N$ comparisons. *SIAM J. Discret. Math.*, 14(3):312–325, March 2001. `doi:10.1137/S0895480199353895`.
12. Robert W. Floyd and Ronald L. Rivest. Expected time bounds for selection. *Commun. ACM*, 18(3):165–172, March 1975. `doi:10.1145/360680.360691`.
13. GNU Team. Implementation of `std::nth_element`, 2016. [Online; accessed 27-Nov-2016]. URL: `https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a01347.html`.
14. Robin Griffin and K. A. Redish. Remark on algorithm 347 [m1]: An efficient algorithm for sorting with minimal storage. *Commun. ACM*, 13(1):54–, January 1970. `doi:10.1145/361953.361993`.
15. C. A. R. Hoare. Algorithm 63: Partition. *Commun. ACM*, 4(7):321–, July 1961. URL: `http://doi.acm.org/10.1145/366622.366642`, `doi:10.1145/366622.366642`.
16. C. A. R. Hoare. Algorithm 64: Quicksort. *Commun. ACM*, 4(7):321–, July 1961. `doi:10.1145/366622.366644`.
17. C. A. R. Hoare. Algorithm 65: Find. *Commun. ACM*, 4(7):321–322, July 1961. `doi:10.1145/366622.366647`.
18. Peter Kirschenhofer and Helmut Prodinger. Comparisons in Hoare's find algorithm. *Combinatorics, Probability and Computing*, 7:111–120, 3 1998. URL: `http://journals.cambridge.org/article_S0963548397003325`, `doi:null`.
19. Krzysztof C. Kiwiel. On Floyd and Rivest's SELECT Algorithm. *Theor. Comput. Sci.*, 347(1-2):214–238, November 2005. `doi:10.1016/j.tcs.2005.06.032`.
20. Donald E. Knuth. *The Art of Computer Programming, Volume 3: (2Nd Ed.) Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998.

21 Tony W. Lai and Derick Wood. *SWAT 88: 1st Scandinavian Workshop on Algorithm Theory Halmstad, Sweden, July 5–8, 1988 Proceedings*, chapter Implicit selection, pages 14–23. Springer Berlin Heidelberg, Berlin, Heidelberg, 1988. doi:10.1007/3-540-19487-8_2.

22 Conrado Martínez, Daniel Panario, and Alfredo Viola. *Mathematics and Computer Science II: Algorithms, Trees, Combinatorics and Probabilities*, chapter Analysis of Quickfind with Small Subfiles, pages 329–340. Birkhäuser Basel, Basel, 2002. doi:10.1007/978-3-0348-8211-8_20.

23 Conrado Martínez, Daniel Panario, and Alfredo Viola. Adaptive sampling strategies for quickselect. *ACM Trans. Algorithms*, 6(3):53:1–53:45, July 2010. doi:10.1145/1798596.1798606.

24 Jean-Baptiste Michel, Yuan Kui Shen, Aviva Presser Aiden, Adrian Veres, Matthew K Gray, Joseph P. Pickett, Dale Hoiberg, Dan Clancy, Peter Norvig, Jon Orwant, et al. Quantitative analysis of culture using millions of digitized books. *science*, 331(6014):176–182, 2011.

25 David R. Musser. Introspective sorting and selection algorithms. *Software – Practice & Experience*, 27(8):983–993, 1997.

26 Himangi Saraogi. Median of medians algorithm, 2013. URL: http://himangi774.blogspot.com/2013/09/median-of-medians.html.

27 Uwe Schöning. Mastering the master theorem. *Bulletin of the EATCS*, 71:165–166, 2000.

28 SciPy.org. Implementation of `argpartition`, 2017. [Online; accessed Feb 9, 2017]. URL: https://docs.scipy.org/doc/numpy/reference/generated/numpy.argpartition.html.

29 R. Sedgewick and K. Wayne. *Algorithms*. Pearson Education, 2011. URL: https://books.google.com/books?id=idUdqdDXqnAC.

30 Richard C. Singleton. Algorithm 347: An efficient algorithm for sorting with minimal storage [m1]. *Commun. ACM*, 12(3):185–186, March 1969. doi:10.1145/362875.362901.

31 J.W. Tukey. The ninther, a technique for low-effort robust (resistant) location in large samples. *Contributions to Survey Sampling and Applied Statistics in Honor of HO Hartley, Academic Press, New York*, pages 251–258, 1978.

32 Wikipedia. Median of medians, 2016. [Online; accessed 25-Feb-2016]. URL: https://en.wikipedia.org/wiki/Median_of_medians.

33 Andrew C. Yao and F. F. Yao. On the average-case complexity of selecting the k-th best. Technical report, Stanford University, Stanford, CA, USA, 1979.

34 Chee K. Yap. New upper bounds for selection. *Commun. ACM*, 19(9):501–508, September 1976. doi:10.1145/360336.360339.

A Additional Measurement Results

Table 1 Run times in milliseconds (*random* dataset).

Size	GNUIntroselect	BFPRT	Rnd3Pivot	Ninther	MedianOfNinthers
10,000	$7.45 \cdot 10^{-2}$	0.19	$8.14 \cdot 10^{-2}$	$7.81 \cdot 10^{-2}$	$6.01 \cdot 10^{-2}$
31,620	0.22	0.61	0.25	0.25	0.18
100,000	0.72	1.97	0.77	0.76	0.53
316,220	2.15	6.19	2.48	2.47	1.61
1,000,000	6.99	19.74	7.90	7.96	4.96
3,162,280	23.11	62.86	25.73	25.52	15.75
10,000,000	71.67	198.41	80.74	79.84	48.57

Table 2 Run times in milliseconds (*random01* dataset).

Size	GNUIntroselect	BFPRT	Rnd3Pivot	Ninther	MedianOfNinthers
$10,000$	$5.76{\cdot}10^{-2}$	0.14	$6.40{\cdot}10^{-2}$	$6.46{\cdot}10^{-2}$	$5.37{\cdot}10^{-2}$
$31,620$	0.17	0.44	0.20	0.21	0.16
$100,000$	0.55	1.35	0.62	0.64	0.52
$316,220$	1.71	4.19	1.97	2.02	1.44
$1,000,000$	5.50	13.31	6.33	6.47	5.12
$3,162,280$	17.92	43.54	20.13	20.70	16.26
$10,000,000$	57.47	142.78	64.48	65.96	50.66

Table 3 Run times in milliseconds (*m3killer* dataset).

Size	GNUIntroselect	BFPRT	Rnd3Pivot	Ninther	MedianOfNinthers
$10,000$	$8.58{\cdot}10^{-3}$	0.11	$1.88{\cdot}10^{-2}$	$2.01{\cdot}10^{-2}$	$1.61{\cdot}10^{-2}$
$31,620$	$2.79{\cdot}10^{-2}$	0.28	$5.83{\cdot}10^{-2}$	$5.43{\cdot}10^{-2}$	$4.89{\cdot}10^{-2}$
$100,000$	$8.65{\cdot}10^{-2}$	1.05	0.18	0.16	0.16
$316,220$	0.25	2.42	0.53	0.53	0.47
$1,000,000$	1.25	9.43	1.85	1.91	1.73
$3,162,280$	5.42	33.15	7.28	4.89	7.13
$10,000,000$	19.06	109.99	27.24	27.99	19.52

Table 4 Run times in milliseconds (*organpipe* dataset).

Size	GNUIntroselect	BFPRT	Rnd3Pivot	Ninther	MedianOfNinthers
$10,000$	$5.19{\cdot}10^{-3}$	$2.24{\cdot}10^{-2}$	$8.83{\cdot}10^{-3}$	$2.89{\cdot}10^{-3}$	$4.10{\cdot}10^{-3}$
$31,620$	$1.60{\cdot}10^{-2}$	$6.24{\cdot}10^{-2}$	$2.71{\cdot}10^{-2}$	$1.01{\cdot}10^{-2}$	$1.24{\cdot}10^{-2}$
$100,000$	$4.86{\cdot}10^{-2}$	0.29	$7.90{\cdot}10^{-2}$	$2.90{\cdot}10^{-2}$	$3.92{\cdot}10^{-2}$
$316,220$	0.15	0.74	0.24	$9.60{\cdot}10^{-2}$	0.10
$1,000,000$	0.47	2.61	0.74	0.27	0.32
$3,162,280$	2.33	10.00	2.92	1.16	1.27
$10,000,000$	8.37	37.28	9.61	3.71	4.13

Table 5 Run times in milliseconds (*sorted* dataset).

Size	GNUIntroselect	BFPRT	Rnd3Pivot	Ninther	MedianOfNinthers
$10,000$	$9.92{\cdot}10^{-3}$	$5.54{\cdot}10^{-2}$	$1.74{\cdot}10^{-2}$	$5.70{\cdot}10^{-3}$	$7.71{\cdot}10^{-3}$
$31,620$	$3.12{\cdot}10^{-2}$	0.14	$4.82{\cdot}10^{-2}$	$1.73{\cdot}10^{-2}$	$2.32{\cdot}10^{-2}$
$100,000$	$9.51{\cdot}10^{-2}$	0.79	0.15	$5.41{\cdot}10^{-2}$	$7.22{\cdot}10^{-2}$
$316,220$	0.30	1.60	0.47	0.17	0.20
$1,000,000$	1.40	5.84	1.81	0.74	0.88
$3,162,280$	4.93	20.28	5.94	2.50	2.51
$10,000,000$	17.10	81.60	19.67	7.66	7.77

Table 6 Run times in milliseconds (*rotated* dataset).

Size	GNUIntroselect	BFPRT	Rnd3Pivot	Ninther	MedianOfNinthers
10,000	$9.45{\cdot}10^{-2}$	$5.29{\cdot}10^{-2}$	$1.73{\cdot}10^{-2}$	$1.08{\cdot}10^{-2}$	$1.17{\cdot}10^{-2}$
31,620	0.32	0.15	$5.18{\cdot}10^{-2}$	$3.33{\cdot}10^{-2}$	$3.48{\cdot}10^{-2}$
100,000	1.12	0.59	0.16	0.10	0.11
316,220	3.92	1.57	0.47	0.32	0.31
1,000,000	16.91	5.90	1.70	1.27	1.15
3,162,280	98.91	20.29	5.75	3.93	3.64
10,000,000	366.41	82.09	18.89	12.38	11.71

Table 7 Run times in milliseconds (*Google Ngram* dataset).

Corpus	GNUIntroselect	BFPRT	Rnd3Pivot	Ninther	MedianOfNinthers
eng	117.81	396.29	162.80	135.19	94.40
fre	49.69	169.70	70.12	70.14	43.21
ger	77.98	222.23	92.16	100.67	54.86
ita	37.96	106.29	43.49	33.15	27.98
rus	64.83	224.04	93.15	93.12	56.44
spa	50.01	134.70	55.74	54.52	39.07

Table 8 Comparisons per element (*random* dataset).

Size	GNUIntroselect	BFPRT	Rnd3Pivot	Ninther	MedianOfNinthers
10,000	2.80	6.75	2.77	2.40	2.31
31,620	2.81	6.91	2.70	2.43	2.26
100,000	2.89	7.14	2.72	2.32	2.22
316,220	2.72	7.21	2.77	2.47	2.07
1,000,000	2.74	7.30	2.77	2.47	2.04
3,162,280	2.89	7.33	2.77	2.52	2.03
10,000,000	2.80	7.34	2.75	2.44	2.02

Table 9 Comparisons per element (*googlebooks* dataset).

Corpus	GNUIntroselect	BFPRT	Rnd3Pivot	Ninther	MedianOfNinthers
eng	2.02	7.29	2.18	1.95	2.01
fre	2.58	7.45	3.14	2.29	2.03
ger	2.38	7.54	3.04	2.62	2.03
ita	2.19	7.58	2.19	1.95	2.06
rus	1.98	7.44	3.57	2.14	2.03
spa	2.19	7.38	2.27	2.09	2.09

Table 10 Swaps per element (*random* dataset).

Size	GNUIntroselect	BFPRT	Rnd3Pivot	Ninther	MedianOfNinthers
$10,000$	0.55	3.28	0.55	0.53	0.33
$31,620$	0.55	3.35	0.55	0.54	0.30
$100,000$	0.57	3.46	0.54	0.52	0.29
$316,220$	0.54	3.50	0.55	0.54	0.29
$1,000,000$	0.54	3.54	0.55	0.55	0.27
$3,162,280$	0.57	3.54	0.56	0.55	0.26
$10,000,000$	0.55	3.55	0.55	0.54	0.26

Table 11 Swaps per element (*googlebooks* dataset).

Corpus	GNUIntroselect	BFPRT	Rnd3Pivot	Ninther	MedianOfNinthers
eng	0.45	3.43	0.54	0.45	0.25
fre	0.48	3.50	0.55	0.55	0.27
ger	0.53	3.49	0.65	0.60	0.26
ita	0.52	3.55	0.48	0.43	0.28
rus	0.45	3.50	0.59	0.53	0.27
spa	0.54	3.44	0.54	0.52	0.31

Table 12 Coefficient of variation of run time (*random* dataset).

Size	GNUIntroselect	BFPRT	Rnd3Pivot	Ninther	MedianOfNinthers
$10,000$	0.13	0.11	0.13	0.13	0.11
$31,620$	0.14	$8.63{\cdot}10^{-2}$	0.11	0.11	$6.97{\cdot}10^{-2}$
$100,000$	0.11	$2.57{\cdot}10^{-2}$	0.14	$9.80{\cdot}10^{-2}$	$5.05{\cdot}10^{-2}$
$316,220$	0.13	$5.13{\cdot}10^{-2}$	0.13	0.11	$7.72{\cdot}10^{-2}$
$1,000,000$	0.12	$1.56{\cdot}10^{-2}$	0.14	0.10	$6.09{\cdot}10^{-2}$
$3,162,280$	0.11	$1.61{\cdot}10^{-2}$	0.12	0.11	$2.66{\cdot}10^{-2}$
$10,000,000$	0.13	$1.57{\cdot}10^{-2}$	0.12	0.11	$1.87{\cdot}10^{-2}$