Abstract

Sorting and selection are fundamental algorithms, described and analyzed in detail in the literature. A polymorphic sort function is provided by many run-time libraries, but few libraries provide a selection function. Consequently, there are many different implementations of selection used in practice; many exhibit poor performance. System sort functions are not without performance issues; most existing quicksort implementations can be easily driven to quadratic performance; most also exhibit performance anomalies with some common input sequences. Although quicksort can sort data in-place, other sorting implementations may require substantial additional memory. During implementation of a high-performance polymorphic in-place selection function based on quicksort-like divide-and-conquer techniques, some remaining performance issues in a well-known and widely-used implementation of quicksort were analyzed and addressed. The resulting in-place polymorphic selection function can be used for sorting as well as selection, and has performance for sorting comparable to or better than other quicksort implementations; in particular, it cannot be driven to quadratic sorting or selection behavior. Analysis techniques and algorithm design provided insights which yielded improvements applicable to other well-known sorting implementations and which may be applicable to other software projects. Improved quality of sampling, especially compared to median-of-3 using the first, middle, and last elements, is one example of such an improvement.

Sorting and selection without tears

Bruce Lilly
bruce.lilly@gmail.com

June 3, 2017

1 Introduction

The selection problem is well known to have linear bounds. [1] [2] [3] [4] Yet one often encounters naïve implementations; ones that perform a full sort to obtain a single order statistic [5] [6] or worse. [7] [8] Lent & Mahmoud [4] describe an algorithm for obtaining multiple order statistics which determines a single order statistic with linear complexity and which can determine an arbitrarily large number of order statistics with no more complexity than a full sort. This algorithm, suitably modified, is chosen as the basis for the final high-performance in-place polymorphic selection and sorting algorithm described in this paper. The internal operation of the algorithm is the same as quicksort, with one small detail; whereas quicksort recursively processes multiple (usually two) regions produced by partitioning at each stage, multiple quickselect processes only regions containing desired order statistics. Multiple quickselect as described and analyzed by Lent & Mahmoud [4] uses an array of elements of known size plus indices to sub-array endpoints, and an array plus indices for order statistic ranks. Because the function which is described in the present work will have a polymorphic *qsort*-like interface, the public interface to the implementation described in this paper uses a pointer to the array base, the number of elements in the array, the array element size, and a pointer to a comparison function (as for *qsort*), plus a pointer to the base of an array of *size_t* elements for the desired order statistics and the number of order statistics. In the implementation, if the array of order statistics is absent (i.e. has a *NULL* base pointer) or is of zero size, a full sort is performed. Therefore, the implementation (hereafter referred to as *quickselect*) is suitable for both selection and sorting.

Insight: Simplify maintenance by appropriately combining functionality.

As the basic operations for multiple quickselect are the same as for quicksort, several well-known implementations of quicksort were examined for applicability to this implementation of multiple quickselect. One such quicksort implementation is described by Bentley & McIlroy. [9] Testing and analysis revealed a few shortcomings of that implementation, several of which also afflict other sorting implementations. The tests and analyses, the shortcomings they uncovered, and improvements to address those shortcomings are described in later sections of this paper.

Both quicksort and quickselect use the same basic operations: sample the input array, select a pivot element from the samples, partition the array around the pivot by swapping elements (dividing the array into three sections: those elements comparing less than the pivot, those comparing equal to the pivot, and those comparing greater than the pivot), repeating the process as necessary on the less-than and/or greater-than regions. Selection processes only regions which might contain the desired order statistics; sorting can be considered a simplification which avoids testing for desired order statistics, unconditionally processing both regions resulting from a partition. Analogous operations apply if more than one pivot element is selected.

After a brief description of the testing framework used, the structure of this paper proceeds with a top-down analysis of shortcomings commonly encountered in quicksort implementations and solutions to those problems, culminating in a design meeting the goal of a high-performance, in-place, *qsort*-compatible, polymorphic sorting and multiple selection function. Performance data are then presented to compare the resulting function to a few other sorting implementations.

2 Description of tests and test framework

A test framework was built which could run several sorting and selection implementations with various input sequences, collecting timing and comparison count information. For internally-implemented functions, counts of swapping and other internal operations were also maintained. It is similar to the test framework described by Bentley & McIlroy [9] with several additional test sequences:

- inverse organ-pipe (... 2 1 0 1 2 ...)
- median-of-3-killer [10]
- dual-pivot-killer
- wort-case input (for number of element swaps) for sorting networks
- rotated sequence [11]
- all permutations of N distinct values (for very small N)
- all combinations of N zeros and ones (for small N)
- all-equal-elements
- many-equal elements (several variants; equal elements on the left, in the middle, on the right, shuffled)
- random sequences (full-range integers, integers limited to the range [0,N), integers restricted to the range $[0,\sqrt{N})$, randomly shuffled distinct integers, approximately normally distributed random integers [12] , $\pm 6\sigma$ histogram data for approximately normal distribution)
- McIlroy's antiqsort adversary [13]
- prepared sequences read from a text file

Although McIlroy's adversary constructs a sequence on-the-fly, the sequence produced is simply a permutation of distinct-valued elements, and any sequence congruent to that permutation will elicit the same behavior from a sorting function adhering to the constraints given by McIlroy. [13] The constructed sequence can be saved to a file for reuse.

Data types generated were similar to those described by Bentley & McIlroy [9] . The relationship between various data type sizes may differ between 32-bit and 64-bit systems; this difference played an important role in uncovering one of the performance issues with Bentley & McIlroy's qsort, as described later in this paper. Unlike Bentley & McIlroy's test framework, the one used by the author did not rely on a trusted sort for verification; correctness of sorting was verified by a linear pass over the array. Testing capabilities included correctness, timing (elapsed user, system, and wall clock time), comparison and swap counts, including a breakdown of less-than, equal-to and greater-than comparison results, and partition size analysis.

3 The sorrows of sorting and selection

Many implementations of *qsort* can be driven to quadratic performance via McIlroy's adversary; McIlroy [13] and Valois [11] have published results. Other input sequences may be effective; Musser's median-of-3-killer [10] is one example.

Worse than quadratic sorting time is a sort that never terminates, or which does so only by crashing, for example by overrunning the program stack. Bentley & McIlroy's qsort implementation as originally written uses two recursive calls, with no attempt to order the calls based on region size, as stated in their paper. [9] It is quite easy to get that implementation to overrun its program stack. At least one modification [14] eliminates the tail recursion, resulting in the ability to handle larger worst-case inputs before overrunning the program stack. A more robust method always processes the smaller region first ensuring minimal stack use.

Selection using a variant of quickselect is subject to the same potential problems noted above for quicksort.

The goal of the present work is to implement a high-performance polymorphic function useful for in-place multiple order statistic selection as well as for in-place sorting. Finding an order statistic requires the ability to determine element rank, which in turn requires knowing where the entire array starts. A recursive call to *qsort* for processing of a sub-array generally loses that information, as the base pointer passed to a recursively-called instance of *qsort* is not necessarily the same base which was initially presented to the calling instance.

An internal interface which maintains the original array base pointer so that ranks can be determined, with a wrapper function to provide the standard *qsort* interface avoids losing rank information. One such approach is the one used in Musser's introsort [10] which uses two indices – one for the start of the range to be processed, and the other for the element past the end of the range to be processed; the two indices bracket a portion of the array. Lent & Mahmoud's [4] multiple quickselect operates similarly. The invariant

array base pointer allows rank of any element to be determined, and the ranks for the array section being processed are bounded by the two indices.

Quadratic behavior in quicksort and quickselect arises when an unfavorable partition, i.e. one which has a much larger large region than the small region, is processed, reducing the problem size only slightly instead of (ideally) by a factor of two, and when successive partitions are predominantly lopsided in a similar manner. An occasional lopsided partition is not itself a problem, but when successive iterations make only $O(1)$ reductions in the problem size instead of an $O(N)$ reduction, quadratic behavior results. Individual partitions can be quite lopsided with surprisingly limited effect on overall performance.

Known methods of pivot selection with strong guarantees of pivot rank are all of complexity $O(N)$. Because partitioning has $O(N)$ complexity, pivot selection which also has $O(N)$ complexity would increase cost by some factor, so while it is theoretically feasible to use such a method to guarantee that all pivots produce reasonable partitions, the overall run time for sorting typical inputs would increase by some factor. McIlroy [13] stated:

> No matter how hard implementers try, they cannot (without great sacrifice of speed) defend against all inputs.

It is possible to restrict the use of such a relatively costly method to be used only in case of emergency, i.e. when the partition resulting from a low-cost pivot selection turns out to be too lopsided. When a partition is particularly lopsided, the small region can be processed normally, at small cost, and the pivot and elements comparing equal to it are in-place and require no further processing, but rather than process the large region by again selecting a pivot which may result in another small reduction in problem size, the large region can be partitioned using an alternate pivot selection method which provides a better guarantee of pivot rank. Such a method, which is invoked for the large region only when a partition is particularly lopsided, will be referred to as a "break-glass" mechanism, analogous to the familiar "In case of emergency, break glass" legends sometimes seen near emergency alarm stations. Quadratic behavior can be avoided with surprisingly rare use of the break-glass mechanism; even if it is only invoked when the large region has more than 93.75% (i.e. $^{15}\!/_{16}$) of the array elements, quadratic behavior can be effectively eliminated. The key is that a proportion of the array size (i.e. failure to achieve $O(N)$ reduction) triggers the mechanism, which ensures its use if the initial pivot choice results in only an $O(1)$ reduction in problem size. The break-glass mechanism differs from Musser's [10] and Valois' [11] introsort variants in two important ways:

- Break-glass is used when a partition is exceptionally poor; Musser's introsort waits until recursion depth grows, by which time the expected average-case run time has already been exceeded by some factor. Valois' introsort examines reduction in problem size, but requires different versions for sorting and for selection.
- Break-glass switches pivot selection mechanisms, but continues to use cache-friendly quicksort or quickselect; Musser's introsort switches to heapsort, which has rather poor locality of access and does not readily lend itself to an efficient solution of the selection problem, especially for a variable number of order statistics. Valois' introsort uses a random number generator to pick a few samples, which are shuffled; it can use heapsort or can continue to shuffle the array. In either case, performance of Valois' introsort against McIlroy's adversary [13] was reported [11] as rather poor; 4 to 6 times worse than heapsort.

The break-glass mechanism defends against all inputs without sacrificing speed. It is practical in this implementation partly because of the goal of supporting selection, which is used for guaranteed-rank pivot selection. Improvements in quality and quantity of samples used for pivot selection (described in later sections of this paper) facilitate determination of when a partition is unexpectedly poor.

4 Pivot selection with guaranteed rank limits

Blum et al. [2] describe a median-of-medians method which can limit the rank to the range 30% to 70% asymptotically using medians of sets of 5 elements and a recursive call to find the median of medians. Median-of-5 requires a minimum of 6 comparisons for distinct-valued elements. In median-of-medians, medians are obtained for sets of elements, then the median of those medians is found. If selection (i.e. finding the median) has some cost kM for finding the median of M medians, and the cost of each of the M medians is 6 comparisons, then the overall cost of finding the median-of-medians for sets of 5 elements using 6 comparisons per set is $(1.2 + 0.2k)N$[1].

[1] In this analysis, a separate median selection algorithm with linear cost is used, rather than recursion.

Sets of 3 elements can be used; the rank guarantee is ⅓ to ⅔ asymptotically, which is a tighter bound than for sets of 5. The cost of obtaining the median of a set of 3 elements is on average ⅔ comparisons giving an overall cost for median of medians using sets of three elements of $(0.889 + 0.333k)N$. If the cost k of selection is less than 2.333[1], the tighter bound on pivot rank provided by median-of-medians using sets of 3 elements can obtained at lower cost than the looser guarantee from median-of-medians using sets of 5 elements. A simplification of median-of-medians ignores "leftover" elements if the array size is not an exact multiple of the set size, with a slight increase in the range of pivot rank.

Another factor favoring use of sets of 3 elements is that medians of sets of 3 elements are also used by other pivot selection methods, whereas median-of-5 would require additional code.

5 Repivoting decision

Consider an input sequence of N distinct values. Partitioning will divide the input into three regions; one contains only the pivot and the other two regions combined have $N - 1$ elements. For convenience, let $n = N - 1$, then the sizes of the latter two regions can be expressed as $\frac{n}{d}$ and $\frac{n \times (d-1)}{d}$. Let the costs of pivot selection and partitioning be linear in n, call the constant factors a and b. The total complexity of sorting an array of size N is

$$\begin{aligned} cN \log_2 N = (a+b)n + \frac{cn}{d} \log_2 \frac{cn}{d} \\ + \frac{cn(d-1)}{d} \log_2 \frac{cn(d-1)}{d} \end{aligned} \tag{1}$$

Or in words, sorting proceeds by pivot selection and partitioning, followed by recursion on the two regions resulting from the partition. Each sorting operation is expected to have complexity $O(N \log_2 N)$ with constant c. The objective is to determine how c is affected by lopsided partitions. Comparisons are assumed to dominate the cost for the *qsort* polymorphic sorting function.

The logarithm of a fraction is the difference between the logarithm of the numerator and the logarithm of the denominator:

$$\begin{aligned} cN \log_2 N = (a+b)n + \frac{cn}{d}(\log_2 n - \log_2 d) \\ + \frac{cn(d-1)}{d}(\log_2 (d-1) + \log_2 n - \log_2 d) \end{aligned} \tag{2}$$

Rearranging terms:

$$\begin{aligned} cN \log_2 N = (a+b)n + cn \log_2 n - cn \log_2 d \\ + \frac{cn(d-1)}{d} \log_2 (d-1) \end{aligned} \tag{3}$$

For large N, $N \approx n$ and

$$c \approx \frac{a+b}{\log_2 d - \frac{(d-1)}{d} \log_2 (d-1)} \tag{4}$$

If pivot selection has negligible cost in terms of n because the number of samples is a tiny fraction of N (for sufficiently large N), $b \approx 0$. Partitioning requires n comparisons to the pivot element, so $a \approx 1$. If the two regions resulting from partitioning have equal size, $d = 2$ and $c = a + b$. For an input sequence and pivot selection method which always results in the same split d, there is a constant factor for complexity given by equation (4) ($a + b = 1$). For example, a split with approximately a third of the elements in one region and two thirds in the other, $d = 3$, yields a complexity of $\approx 1.089 N \log_2 N$. A much more lopsided partition, with 15/16 of the elements in one region produces a complexity of $\approx 2.965 N \log_2 N$.[2]

Median-of-medians pivot selection using sets of 3 elements costs on average $(0.889 + 0.333k)N$ comparisons for a median selection cost of kM for M medians. Given a lopsided partition resulting from fast pivot selection, and the presumption that continuing to partition with fast pivot selection will continue to produce lopsided partitions, it is possible to determine when it might be advantageous to re-pivot. If median-of-medians pivot selection results in an ideal partition, $b = 0.889 + 0.333k$ and $c = a + b = 1.889 + 0.333k$. The worst asymptotic split for median-of-medians with sets of three elements is 2:1, so the worst case would be $1.089 \times (1.889 + 0.33k) = 2.057 + 0.363k$. If the number of comparisons per element for median selection, $k = 2.333$, then repivoting is advantageous at $d \geq 13$.

The above analysis somewhat overestimates the effect of an unfavorable split, in part because the split can consist only of integral numbers of elements, and also because extreme ratios are not possible for small

[1] If repeated comparisons are avoided during repartitioning (described in a later section), the cost of selection may rise to 3.333.

[2] d need not be an integer, but is treated so here for simplicity; d can be any finite number greater than 1.0.

sub-arrays; note that the final step of the derivation specifies large N. It is conjectured that no actual input sequence can produced the specified condition, i.e. lopsided partitions at every stage of processing; pivot selection (discussed in a later section) might, and partitioning does rearrange array elements. However, the analysis provides an idea of where to consider repivoting, and of the effect of lopsided partitions on sorting complexity.

The analysis above considered only the case of distinct input values. The decision to repivot and repartition the large non-pivot region should be dependent only on the size of that region as it relates to the size of the sub-array which was partitioned. Any elements comparing equal to the pivot and therefore having been partitioned into the region containing the pivot are already in-place and should not be reprocessed. The smaller of the non-pivot regions will be processed normally – the work done in partitioning is not discarded, even if the small region produced is tiny. So it is only the size of the larger of the non-pivot regions to the whole which is important, and repivoting is used only if the size of that region represents an inadequate reduction in the problem size from the size of the sub-array from which it was partitioned. As mentioned earlier, an occasional lopsided partition is produced when processing inputs which are not particularly adverse. To prevent excessive repivoting, it may be desirable to ignore some small number of lopsided partitions. More details are provided in a later section on the topic of performance tuning.

The effect of lopsided partitions is somewhat different for order statistic selection. For a single order statistic, if the desired element is always in the large region, and that region contains $\frac{N \times (d-1)}{d}$ elements, the cost of selection is the sum of the pivot selection and partitioning costs for the original array and all of the large sub-arrays resulting, approximating (for large N):

$$(a+b)N\left(1+\sum_{k=1}^{\infty}\left(\frac{d-1}{d}\right)^{k}\right) \approx (a+b)dN \tag{5}$$

While a 2:1 split increases sorting complexity by about 9%, it increases selection complexity by 50% (from $2N$ to $3N$ for unit partitioning cost and negligible pivot selection cost). Good pivot selection (to ensure an even partition split) to maintain efficiency is therefore more important for order statistic selection than for sorting. Whereas median-of-medians and similar methods of obtaining a pivot which guarantees a reasonable partition split would incur a significant increase in cost for sorting if used for every partition, they prevent the constant term for selection complexity from rising rapidly for lopsided partitions. Fast pivot selection has much lower cost of course, and the break-glass mechanism can be used for selection as well as for sorting to limit the effect of lopsided partitions. Because the constant term for selection rises much more quickly for lopsided partitions than the constant term for sorting, the ratio of worst-case to average performance is expected to be greater for selection than for sorting.

6 Element swapping

Performance of Bentley & McIlroy's qsort is poorer than expected for sawtooth inputs and all-equal inputs. The root cause of that was traced to one of the macros and its related code, viz. SWAPINIT. The poor performance results when an attempt is made to swap an element with itself. Function *swapfunc* then copies the element to a temporary variable, copies the element to itself, then copies the temporary variable back to the element. Kiwiel [15] refers to these as "vacuous swaps" and presents several partitioning algorithms which avoid them. Kiwiel's Algorithm L is one of those, and provides reasonable performance with a moderate increase in code[1].

Bentley & McIlroy [9] noted that their implementation

> depends more heavily on macros than we might like

Replacing the SWAPINIT macro and associated macros and type codes with an inline function improves performance and maintainability. It also permits accounting for advances since 1992; in that 32-bit era, *long* and plain *int* types were generally the same size, whereas on many 64-bit architectures, plain integers may be smaller than long integers. Bentley & McIlroy's qsort supported swapping in increments of *long* and *char* only; in the present implementation four sizes are supported, including *char*, *short*, *int*, and *double*[2]. Rather than pass an integer code indicating the type to be used for swapping, the implementation described in this paper passes a pointer to one of four swap functions, one each for the four supported sizes for swapping. The wrapper function which provides the *qsort* interface determines the appropriate size once

[1] Pointer inequality tests could be added to avoid self-swapping with simpler code, but the tests in inner loops add too much overhead.

[2] Sizes *long* and *void* $*$ could also be used, but on all 32- and 64-bit architectures tested, these sizes overlap with the other four.

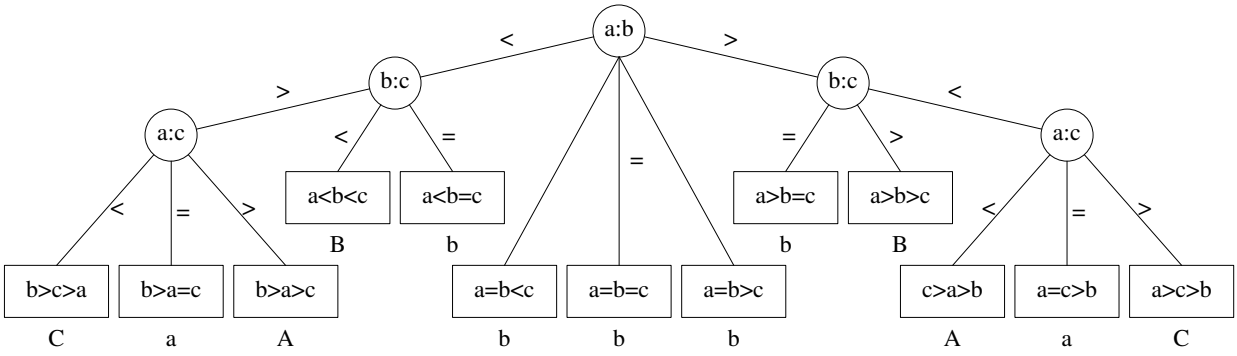

Figure 1: Ternary median-of-3 decision tree

based on array alignment and element size, and a pointer to the appropriate swapping function is passed to the internal quicksort/quickselect implementation.

One conceptual feature of element swapping in Bentley & McIlroy's qsort is retained; the *vecswap* function (actually yet another macro in Bentley & McIlroy's implementation) which moves a minimal number of elements to effectively reorder blocks of elements.

7 Partitions

Bentley & McIlroy considered various partitioning schemes, arriving at one which starts with swapping the selected pivot element to the first position[1], then scans from both ends of the unknown order region placing elements comparing equal to the pivot at both ends of the array before finally moving those regions to the canonical middle location by efficient block moves.

Several partitioning variations [15] [16] were investigated in the present work. The split-end scheme described by Bentley & McIlroy [9] as modified by Kiwiel's algorithm L [15] works best. Its performance advantage is a result of four factors:

- Removing elements which compare equal to the pivot from further processing by putting those elements in place adjacent to the pivot, subsequently processing only the less-than and greater-than regions of elements.
- The use of two pointers to process the unknown region from both ends, then placing two out-of-place elements in their correct regions with a single swap.
- The split-end equals regions permit putting an element comparing equal to the pivot (when scanning from the upper end of the unknowns) into the upper equals region with a single swap, rather than the two swaps (or equivalent 3-way exchange) that would be required with an equals region only on the left. Partitioning schemes that do not separate elements comparing equal to a pivot fare even worse; they entail additional comparisons which result in additional swaps.
- Swapping an element with itself is avoided by Kiwiel's algorithm L [15] .

Those factors reduce the number of element swaps, and the use of a single pivot reduces the number of comparisons and swaps compared to multi-pivot schemes[2].

Insight: Explore options; test, measure, and compare

8 Ternary median of three

Median-of-3 is used extensively in pivot selection; by itself and as a component of more complex pivot selection methods. The implementation in Bentley & McIlroy's qsort fails to take advantage of elements which compare equal. If any two of the three elements in a median-of-3 set compare equal, either one can be taken as the median; the value of the third element is irrelevant. Modification of the median-of-3 function

[1] Except when it does not, as explained later.

[2] Multi-pivot quicksort is usually compared to a strawman "classic" quicksort which uses a more costly partitioning scheme than Bentley & McIlroy's; multi-pivot quicksort requires additional passes (with additional comparisons and swaps) to process elements comparing equal to one or more of the pivots, or requires more complex swaps.

to return early on an equal comparison which determines the result reduces the number of comparisons required for pivot selection when some elements compare equal. For example, an array of size 41 of all-equal elements requires 44 rather than 52 comparisons to sort if equal comparison results are used.

In Figure 1, note that 6 of the 13 possible conditions relating the order of the three elements result in fixed medians; 2 conditions each for the three elements. These are indicated in the figure by upper-case letters below the conditions, specifying the median element. The remaining 7 conditions allow for some flexibility because of equal comparison results; either of two elements comparing equal could be returned as the median. The example in the figure biases the returned results toward the middle element where that is possible, otherwise toward the first element where that is possible. These biases could be altered by changing the order of comparisons and the return value used for equal comparisons. Because the number of possible conditions is not divisible by the number of elements it is not possible to produce an even distribution of results; there must always be some bias. When the result of median-of-3 is used to swap the median element to some position the bias can be used to reduce data movement inefficiency. With the bias as shown in Figure 1, more than half, specifically 7⁄13 or about 54% of the conditions result in the middle element being selected as the median. Compared to the ⅓ or 33% for the binary median-of-three decision tree, this bias can reduce the amount of swapping that would be required in the absence of bias (such as with the binary implementation) or if a poor choice of bias is used.

Three of the 13 conditions are handled with a single comparison, 4 result from exactly two comparisons, and the remaining 6 conditions require three comparisons. For equally likely conditions, the average number of comparisons is $\frac{3+4\times2+6\times3}{13} = \frac{29}{13} \approx 2.231$ vs. $\frac{8}{3} \approx 2.667$ for a binary decision tree. Compare Figure 1 to the binary decision tree shown as Program 5 in Bentley & McIlroy. [9]

9 Pivot element selection

Bentley & McIlroy use three pivot selection methods, in separate ranges based on sub-array size:

1. A single sampled element, used only for a sub-array with 7 elements.
2. The (binary) median of three elements, used when the sub-array contains 8 through 40 elements.
3. A pseudo-median of nine elements in three groups of three elements, used when there are more than 40 elements.

The third method, as well as the second, uses (binary) median-of-3.

As noted earlier in this paper, the limited rank guarantee provided by these methods permits quadratic behavior with adverse input sequences. While the break-glass mechanism presented earlier is sufficient to prevent quadratic behavior, it is desirable to have a pivot selection method which provides a better guarantee of pivot rank than the pseudo-median of nine elements, so that the break-glass mechanism is rarely needed. The use of only nine elements for the pseudo-median limits the effectiveness for large arrays; the small sample has an increased risk of finding a pseudo-median which is not close to the true median.

A good candidate for an improved pivot selection method for large arrays seems to be the remedian described by Rousseeuw & Bassett, [17] computed on a sample of the array elements. The remedian with base 3 is computed by selecting sets of 3 elements, taking the median of each set of 3, then then repeating the process on those medians, until a single median remains. If the sample consists of 9 elements, this is identical to Tukey's ninther. By increasing the sample size as the array size increases, the guarantee on the range of the pivot rank can be improved; Rousseeuw & Bassett [17] give the range of 1-based rank as

$$\text{the smallest possible rank is exactly } \lfloor b/2 \rfloor^k, \text{ whereas the largest possible rank is } n - \lfloor b/2 \rfloor^k + 1$$

where $b = 3$ and $k = \log_3 n$ for a sample of size n. Rousseeuw & Bassett give a rather complex algorithm for computing the remedian when n is not a power of 3; as a sample of the array elements will be used, the complexity can be avoided by making the sample size exactly a power of 3. McGeoch & Tygar [18] and Martínez & Roura [19] have determined that sample size roughly proportional to the square root of the number of elements is optimal. McGeoch & Tygar suggested a table of sample sizes, and that approach is adopted in the present work, in part to avoid costly division by 3 in loops. A table also permits adjustment to the sampling, as the optimum array size is not always exactly proportional to the square of the number of samples, particularly at small sizes. McGeoch & Tygar also note that using a sample size proportional to the square root of the number of elements limits worst-case sorting performance to $O(N^{1.5})$ which is a considerable improvement over $O(N^2)$.

Remedian can be implemented in-place (i.e. in $O(1)$ space) by swapping the median of each set to one position in that set. [20] This does, of course, have a higher cost than an implementation that uses storage

(e.g. for pointer variables) outside of the array being processed because of the inherent cost of swapping. Moreover, swapping may introduce or exacerbate disorder. Judicious use of bias in the ternary median-of-3 can reduce the amount of data movement required to implement such an in-place remedian of samples.

It is also possible to implement remedian recursively, using the stack to hold intermediate results, without any array data movement. That approach is used in the implementation described in this paper, in order to avoid the cost and disorder associated with the in-place implementation of remedian.

One drawback of all of the pivot selection methods using element comparisons is that the same comparisons may be repeated during partitioning. This redundancy is only addressed in one special instance, described in a later section, in part because the cost of doing so generally would outweigh the benefit.

10 Sampling for pivot selection

In discussions of sampling in this paper, there are two distinct characteristics considered; the quantity of samples used for pivot selection, and the quality of choice of those samples from the array elements. Quantity of samples has already been briefly mentioned. This section focuses mainly on the quality of sample selection, and the section following it discusses maintaining quality as quantity changes.

Bentley & McIlroy [9] use three methods of sampling, tied to the three types of pivot selection methods used:

> Our final code therefore chooses the middle element of smaller arrays, the median of the first, middle and last elements of a mid-sized array, and the pseudo-median of nine evenly spaced elements of a large array.

They go on to state:

> This scheme performs well on many kinds of nonrandom inputs, such as increasing and decreasing sequences.

However, Bentley & McIlroy's paper [9] noted worst-case performance with the test sequences they used as occurring with reverse-sorted (i.e. a decreasing sequence of) doubles:

> The number of comparisons used by Program 7 exceeded the warning threshold, 1.2n lg n, in fewer than one percent of the test cases with long-size keys and fewer than two percent overall. The number never exceeded 1.5n lg n. The most consistently adverse tests were reversed shuffles of doubles.

As noted earlier in the present paper, there are size differences related to machine word size which affect performance. Performance with reverse-sorted arrays of several element sizes was measured in the present study; results for 32-bit plain integers on a 64-bit machine are shown in Figure 2[1]. The same results are obtained for arrays of any type of element whose size differs from *sizeof(long)*, including doubles on a 32-bit machine. Figure 2 includes a dashed horizontal line at $1.5N \log_2 N$ comparisons mentioned in Bentley & McIlroy's paper. [9]

Although performance was measured in the present study at each value of array size N from 3 through 4096 for the data for Figure 2, points which lie on or very near a straight line between a pair of enclosing points are not individually plotted with symbols. That reduces clutter in the plots for sorted inputs, but is less effective for the plot for the organ-pipe input sequence because there is little pattern to the comparison count vs. array size for that input sequence. The same method of clutter reduction is applied to other figures in this paper.

The peaks of high numbers of comparisons for reversed input (performance for rotated input sequences was not reported by Bentley & McIlroy) shown in Figure 2 occur at values of array size N satisfying $N = 40 \times 2^k + 1, \quad k = 1, 2...$ where 40 is the cutoff point between median-of-3 and Tukey's ninther for pivot selection. Bentley & McIlroy [9] noted

> The disorder induced by swapping the partition element to the beginning is costly when the input is ordered in reverse or near-reverse.

and it is this disorder coupled with the use of the first and last array elements in computing median-of-3 pivot elements which is responsible for the peaks in the comparison counts evident in Figure 2. The peaks

[1] Most performance graphs in this paper report scaled comparison counts rather than execution time in order to avoid hardware obsolescence issues; also timing tends to be variable due to the effects of paging and other system activity, whereas comparison counts are generally repeatable.

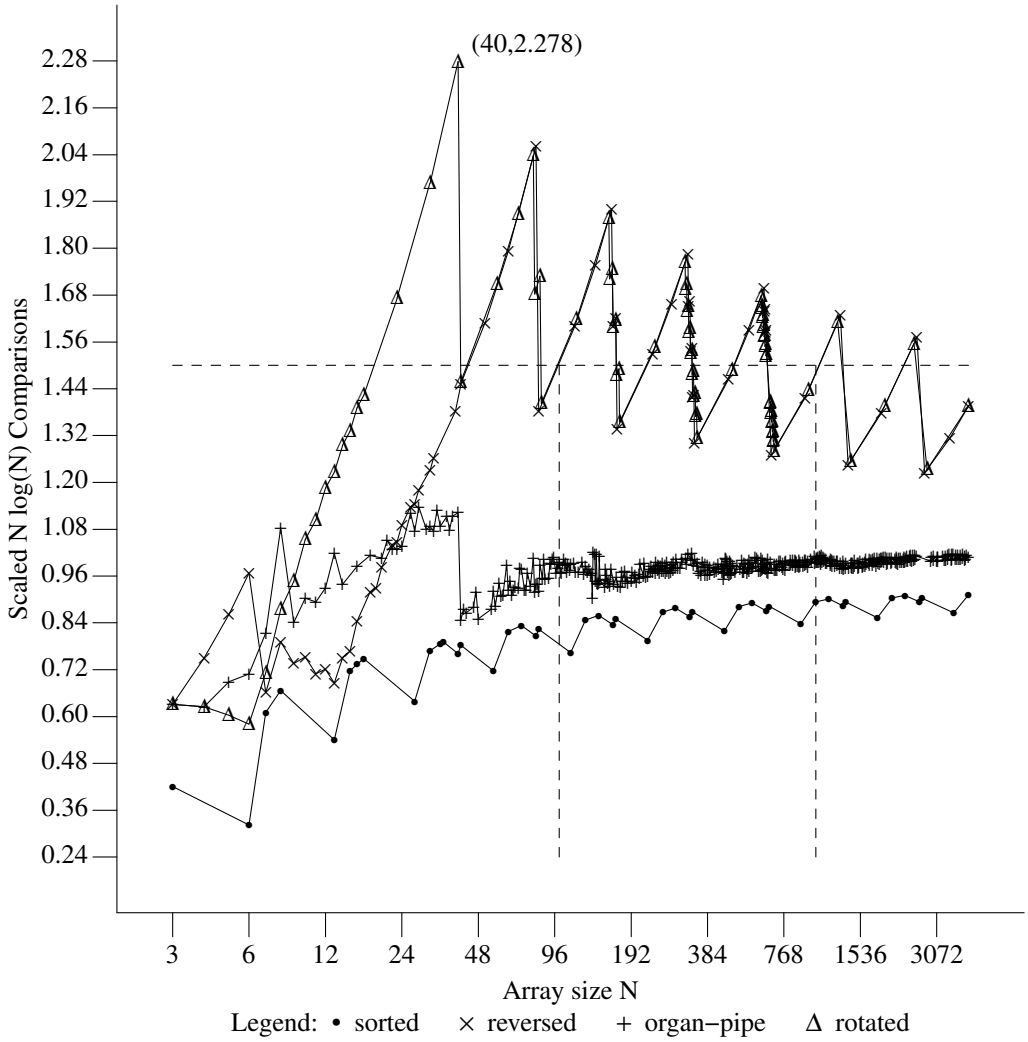

Figure 2: Comparisons for Bentley & McIlroy qsort

initial reverse-sorted array

19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1

swap pivot to start

10	18	17	16	15	14	13	12	11	19	9	8	7	6	5	4	3	2	1

partition

10	1	17	16	15	14	13	12	11	19	9	8	7	6	5	4	3	2	18

...

10	1	2	3	4	5	6	7	8	9	19	11	12	13	14	15	16	17	18

swap blocks (pivot) to canonicalize

9	1	2	3	4	5	6	7	8	10	19	11	12	13	14	15	16	17	18

Figure 3: Partitioning of reverse-sorted input in Bentley & McIlroy qsort

above the dashed horizontal line at $1.5N \log_2 N$ evident in Figure 2 would not have been noted at the array sizes tested by Bentley & McIlroy, which were 100 and 1023–1025 (dashed vertical lines in Figure 2).

It is informative to work through an example to see exactly how and why this disorder occurs and how it interacts with sampling the first and last sub-array positions for median-of-3 pivot selection. Such an example is shown in Figure 3. At the end of partitioning, **both** regions resulting from the partition have an extreme-valued element at the first position. Both regions resemble a rotated sequence, which Valois [11] found to result in poor performance. Figure 2 confirms that a rotated sequence exhibits even worse performance than reverse-sorted input. If median-of-3 is used to select pivots for those sub-arrays using the first, middle, and last elements, the extreme value in the first position causes the second-most extreme value (in the last position) to be selected as the pivot. That results in only $O(1)$ problem size reduction, which of course leads to poor performance; once a rotated sequence is processed by the combination of first, middle, last median-of-3 pivot selection and pivot swapping, recursive calls also operate on rotated sequences resulting in continued very small reductions in problem size. All partitioning methods which initially swap the pivot element to one end of the array share this characteristic, and if pivot selection uses median-of-3 with samples at the array endpoints, the same poor performance with reverse sorted and rotated input sequences can be expected. There are two interacting factors responsible for the poor performance:

1. The initial swap of the pivot element to one end of the array, which moves an extreme-valued element initially there to near the middle of the array, where it ends up at one side of the opposite region to be subsequently processed. The initial swap coupled with the final swap required to canonicalize the partition results in an extreme-valued element being placed in the first position of one region. Initial pivot swaps turn reverse-sorted sequences into rotated sequences.

2. Use of samples at the first, middle, and last elements for median-of-3 pivot selection, i.e. poor quality of sampling. First, middle, last sampling for median-of-3 pivot selection results in very lopsided partitions of rotated sequences.

The poor performance caused by this combination can be remedied for reverse-sorted input sequences by addressing one or both of those factors. Quality of sampling is the more important of the two factors; merely avoiding disorder introduced by pivot swapping does not help if the input is a rotated sequence.

Arrays of *double* types were affected in Bentley & McIlroy's qsort tests because of an optimization its authors made for *long* integer types (and types of the same size); macro PVINIT places the pivot in a separate location rather than swapping to the first position, but only for types of the same size as *long*.. On 32-bit architectures such as were available in 1992, doubles were of a different size than long integers; therefore the PVINIT macro was ineffective for doubles, which were found by Bentley & McIlroy to exhibit the worst performance for reverse-sorted inputs. On 64-bit architectures commonly available in 2016, sorting doubles does not exhibit the same behavior noted by Bentley & McIlroy; the behavior instead shows up for other data types, such as the integers used to generate Figure 2.

The second contributing factor to the poor performance of reverse-sorted inputs, use of the array endpoints in median-of-3 for pivot selection, can be avoided by improving the choice of samples of the input array used to select a pivot element.

Use of the array endpoints with median-of-3 is detrimental when finding a pivot for partitioning organ-pipe inputs and rotated inputs[1] as well as when disorder is introduced as shown for reverse-sorted inputs. Interestingly, Bentley & McIlroy began work on their implementation as the result of a reported problem in an earlier implementation of qsort — when presented with an organ-pipe input sequence:

> They found that it took n^2 comparisons to sort an 'organ-pipe' array of $2n$ integers: 123..*nn*..321.

(Try an example to see why using the array endpoints and middle element is a problem before reading on).

Bitonic input sequences like an organ-pipe sequence arise in many instances. For example, consider an array where values represent the number of elements in histogram bins for some approximately normally distributed data. The values are low at the ends and high in the middle, just like an organ-pipe sequence. Such a sequence has extreme-valued elements at the ends of the array; median-of-3 pivot selection using those endpoints results in selecting a poor pivot element, just as with the rotated sequence resulting from partitioning a reversed sequence.

The ideal case for organ-pipe inputs would include elements at the ¼ and ¾ positions when using median-of-3 to select a pivot, because it is at these locations that the median(s) of an organ-pipe sequence lie. Conversely, the endpoints and middle element contain extreme values.

[1] The anomalous performance for rotated input sequences persists even for *long* data types with Bentley & McIlroy's PVINIT trick, and is exhibited by many existing quicksort implementations. Correcting that anomaly **requires** improving quality of sampling.

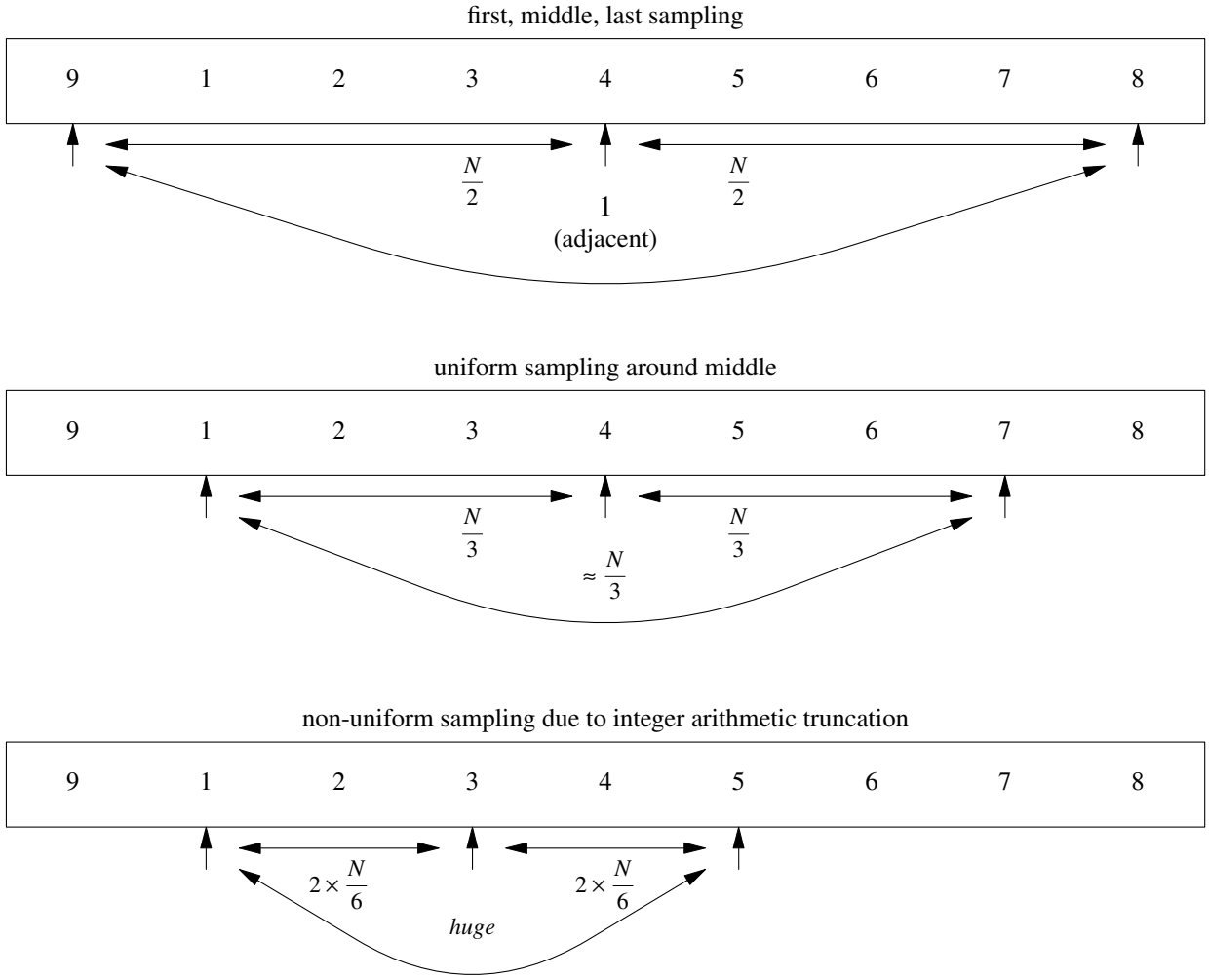

Figure 4: Median-of-3 sampling at ends and middle, uniformly spaced about middle, and non-uniform due to truncation

Quality (as distinct from quantity) of sampling for pivot selection in quicksort is rarely discussed; many published papers and textbooks mentioning median-of-3 pivot selection specify the array middle and ends with no rationale given for choosing the ends. And many implementations which in fact use the array endpoints for median-of-3 for pivot selection can be improved by using positions other than the array ends.

Sampling at ¼, middle, and ¾ positions for median-of-3 pivot selection produces the exact median for sorted, reversed, and organ-pipe input sequences, and the median is always selected if such sequences are rotated by an integral multiple of ¼ of the array size. The samples are not uniformly spaced (considering the array as a circular buffer), however, and the nonuniformity leads to widely varying results if those input sequences are rotated by a variable amount. Uniform sampling reduces such variation, and is achieved for 3 samples by sampling at ⅙, middle, and ⅚ positions; this is not optimal for unrotated organ-pipe inputs but performance is not much worse than optimal.

Consider the left sub-array at the bottom of Figure 3, which is an increasing sequence rotated one position to the right, as shown in Figure 4. Sampling at the endpoints and middle, as is often done in quicksort implementations, produces a poor pivot element with a rotated or organ-pipe input, resulting in a very lopsided partition, and poor performance as can be seen in Figure 2. Uniform sampling around the middle works as well for already-sorted input sequences (because of the sample at the middle) and has minimum variation in performance for rotations of increasing, decreasing, and bitonic input sequences (because of the uniform sample spacing). The implementation of qsort in plan9 [21] computes ⅙ of the sub-array size, uses that as the index of the first sample, then doubles it to get the spacing for the remaining two samples. Unfortunately, integer division by 6 results in a rather large error for small sub-array sizes, as shown in the bottom diagram of Figure 4. The present work computes ⅓ spacing and applies that from the middle element, which produces more nearly uniform sampling at all sub-array sizes.

With modified sampling, performance of Bentley & McIlroy's qsort operating on reverse-sorted, rotated,

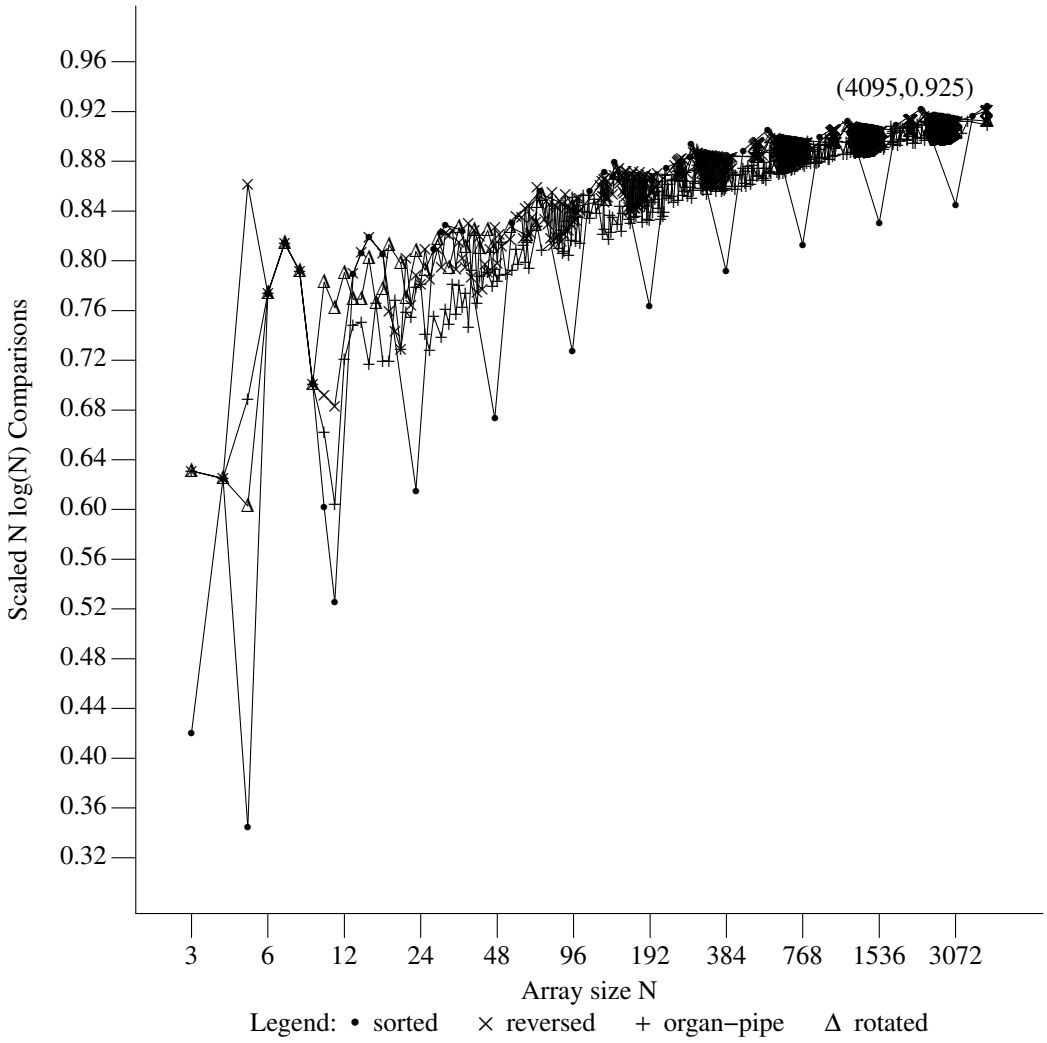

Figure 5: Comparisons for modified Bentley & McIlroy qsort

and organ-pipe inputs is markedly improved, at small cost and with little impact on sorting of other commonly arising input sequences, as shown in Figure 5. Using improved sampling and avoiding single-sample pivot selection by using dedicated sort for small sub-arrays (discussed in some detail in a later section) reduces the impact of the disorder caused by swapping the pivot to one end of the array. Although counterintuitive, disorder can be somewhat reduced in the case of Bentley & McIlroy's split-end partition by swapping the pivot to the end of the array farthest from the original pivot element location (but not if the pivot element is already in an end position). The anomalous high comparison count regions ($\approx 1N \log_2 N$ or higher) for organ-pipe and reversed input sequences below array size 41 seen in Figure 2 have been eliminated, as has the poor performance with reverse-sorted and rotated input sequences at larger sizes[1]. Performance for these input sequences is markedly improved; nearly the same as for already-sorted input.

Insight: Size-specific optimizations may change when size of basic types change.

The curse it is cast
The slow one now
Will later be fast [22]

Insight: Be wary of clever macros.

Insight: Poor performance should be especially carefully and thoroughly analyzed.

Insight: Graphical display of performance measures eases identification of patterns.

The third sampling method used by Bentley & McIlroy, use of the middle array element as the sole sample, also yields poor results with organ-pipe input sequences, for the same reason that use of the array endpoints does; the endpoints and the middle are where the extreme values of an organ-pipe sequence are located. In Bentley & McIlroy's qsort, the middle element is sampled for pivot selection only for arrays of size 7; insertion sort is used for smaller arrays and at least three elements are sampled starting with an array of eight elements. A single sample can be expected to pose a problem; no matter which element is selected, it will be a poor choice for several possible input sequences. The present work avoids using a single sample, as discussed in later sections.

11 Improved sampling

The three sampling methods used by Bentley & McIlroy have one thing in common: the number of samples used is a power of 3, and that is also characteristic of the larger sample sizes used in the present work for remedian of samples. The smallest number, $3^0 = 1$, presents a quandary: any single sample will be the worst possible choice for some rotation of any input sequence. Use of a single sample can be avoided except in one case: when there are only 2 elements, which is not enough for 3 samples. Because such small arrays are not partitioned for divide-and-conquer sorting, that case only arises for order statistic selection under specific conditions, as discussed in a later section of this paper.

When 3 samples are used for pivot selection, uniformly-spaced samples including the middle element appears to be the best choice, as discussed when examining the poor performance obtained by using the array endpoints.

For more than 3 samples, it is still desirable to include samples at uniform spacing centered in the array[2].

The method for computing the remedian presented by Rousseeuw & Bassett [17] computes the remedian sequentially using samples taken in order, as does the method presented by Battiato et al. [20] However, that results in non-uniform sample spacing at the lowest level; the samples are bunched together in one region of the array. A different method is used in the present work, taking samples at uniform spacing for each of the lowest-level medians, as shown in Figure 6 using an 81-element array with an organ-pipe sequence and with 9 uniformly spaced samples. The sequential method selects a pivot element which yields a rather lopsided partition. Taking elements at uniform spacing (shown in rows in the diagram) selects a pivot which results in a more balanced partition. Sequential computation of the remedian can result in worse partitions than the one shown, for example if the input sequence is rotated. Uniformly spaced samples at the lowest level as shown at the bottom of Figure 6 is about as lopsided as that method will produce; a rotation by 2 elements in either direction will yield the true median.

An adaptive sampling method was adopted for the present work:

[1] Pivot element swapping is not deferred, avoiding the extra comparisons which such deferral would entail.

[2] Although Bentley & McIlroy described their implementation of sampling for Tukey's ninther as "evenly spaced", the first and last samples are actually farther apart (considering wrap-around) than the other samples' spacing.

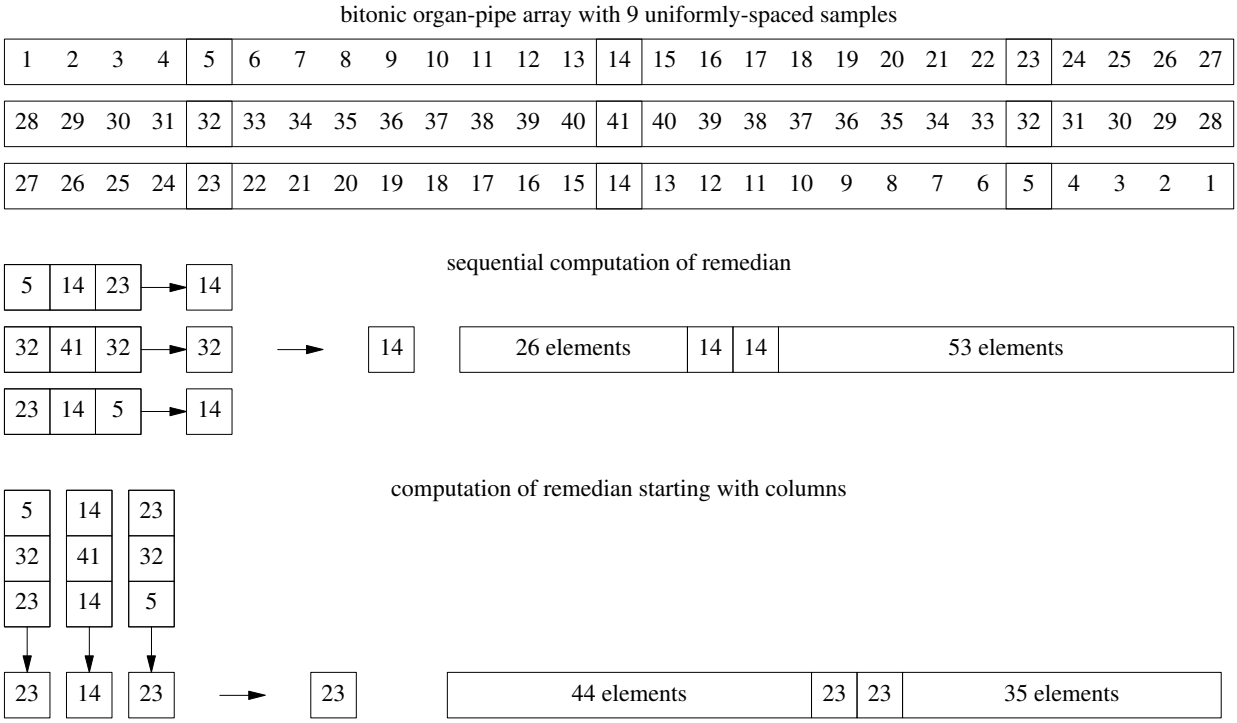

Figure 6: Two methods of computing the remedian

- The number of samples is always a power of 3, and is proportional to the square root of the number of array elements when sorting. Sorting of arrays of fewer than 9 elements is performed by dedicated sorting, obviating sampling and pivot selection for sorting very small arrays. Order statistic selection uses 3 samples down to an array size of 3 elements.
- The element at the second position is used for arrays of 2 elements when selecting order statistics.
- Three elements with uniform spacing and centered in the array are used as samples for arrays of 3 through 68 elements.
- Larger arrays use a selection of a power of three samples evenly spaced and arranged as 3 rows of samples.

Such a sampling method was used to produce Figure 5, using no more than 9 samples (same as Bentley & McIlroy's qsort), which was used for array sizes 69 and larger.

The improvement in performance that results from the improved quality of sampling permits a higher threshold for use of median-of-3 pivot selection; Bentley & McIlroy [9] used empirically-determined cutoffs where median-of-3 was used at arrays of size 8 or more with 9 samples used for arrays of 41 or more elements, whereas median-of-3 pivot selection is used for sorting in the present work for arrays beginning with size 9 and 9 samples are used for arrays starting at 69 elements. Using median-of-3 and ninther for pivot selection when sorting small arrays entails a disproportionate increase in the number of comparisons used. Because partitioning of large arrays results in many small arrays, the savings in reducing the number of comparisons required to process small arrays is multiplied when sorting large arrays. Improved sampling and sorting for small sub-arrays leads to improved performance for all array sizes.

When the remedian of samples is used for pivot element selection, the probability of the selected pivot being close to the median of the array increases as the number of samples increases. [17] [20] For large arrays of random element values the pivot is very likely to be quite close to the median; the partition is well balanced. For smaller arrays and for sub-arrays produced by partitioning larger ones, the selected pivot may produce a quite lopsided partition. Adverse inputs can also result in lopsided partitions. Arrays of randomly shuffled distinct values were generated for array size 9 through 43046721 elements increasing by a factor of 3 for each size and were sorted by *quickselect* using the sampling and pivot selection methods previously described. Recall that sample size is $\propto \sqrt{N}$; sample size varied from 3 through 6561 for the array sizes used. Ten arrays of each size were generated and sorted, and the maximum ratio of the larger region resulting from partitioning to the number of other elements (the smaller region plus the pivot) at each range of sub-array sizes corresponding to the sampling sizes was recorded for each partitioned sub-array during

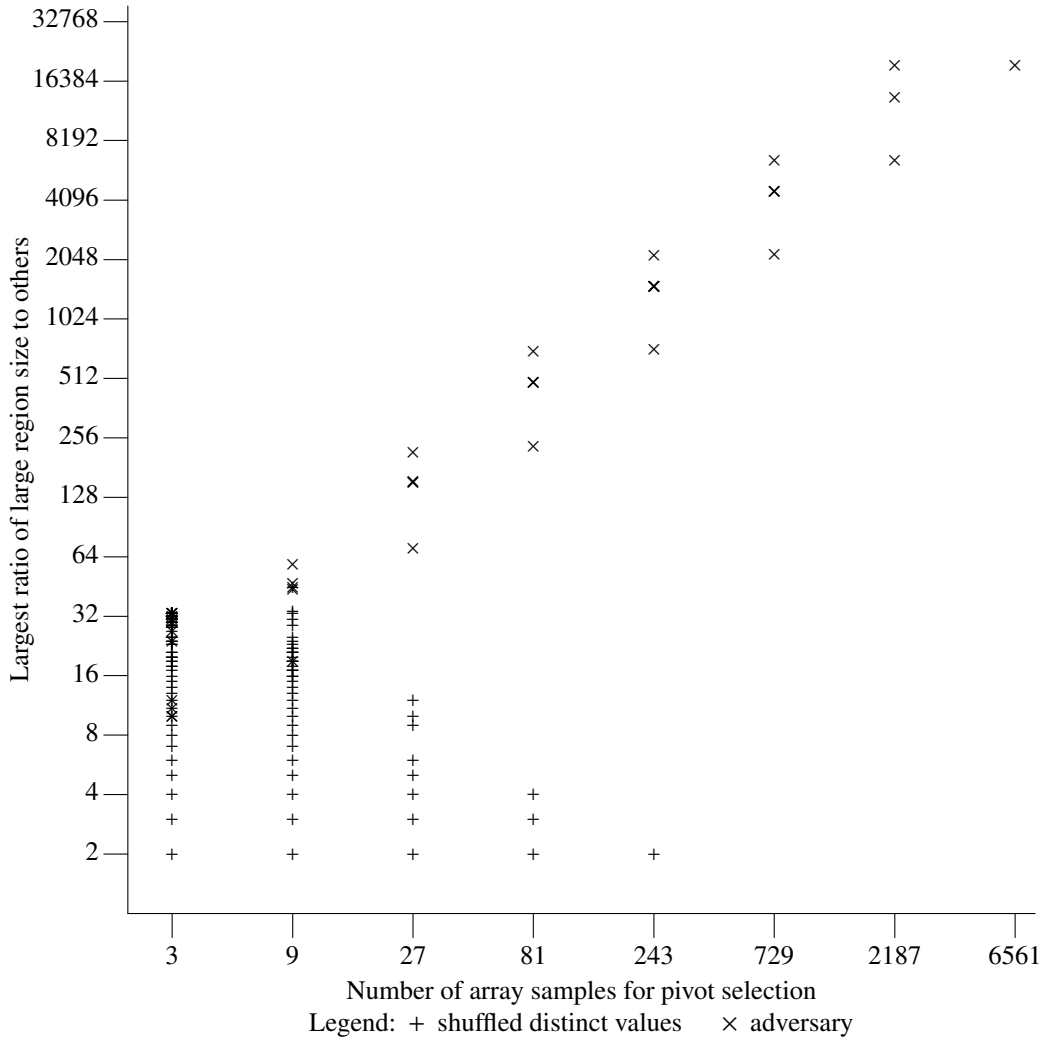

Figure 7: Partition lopsidedness vs. pivot sample quantity

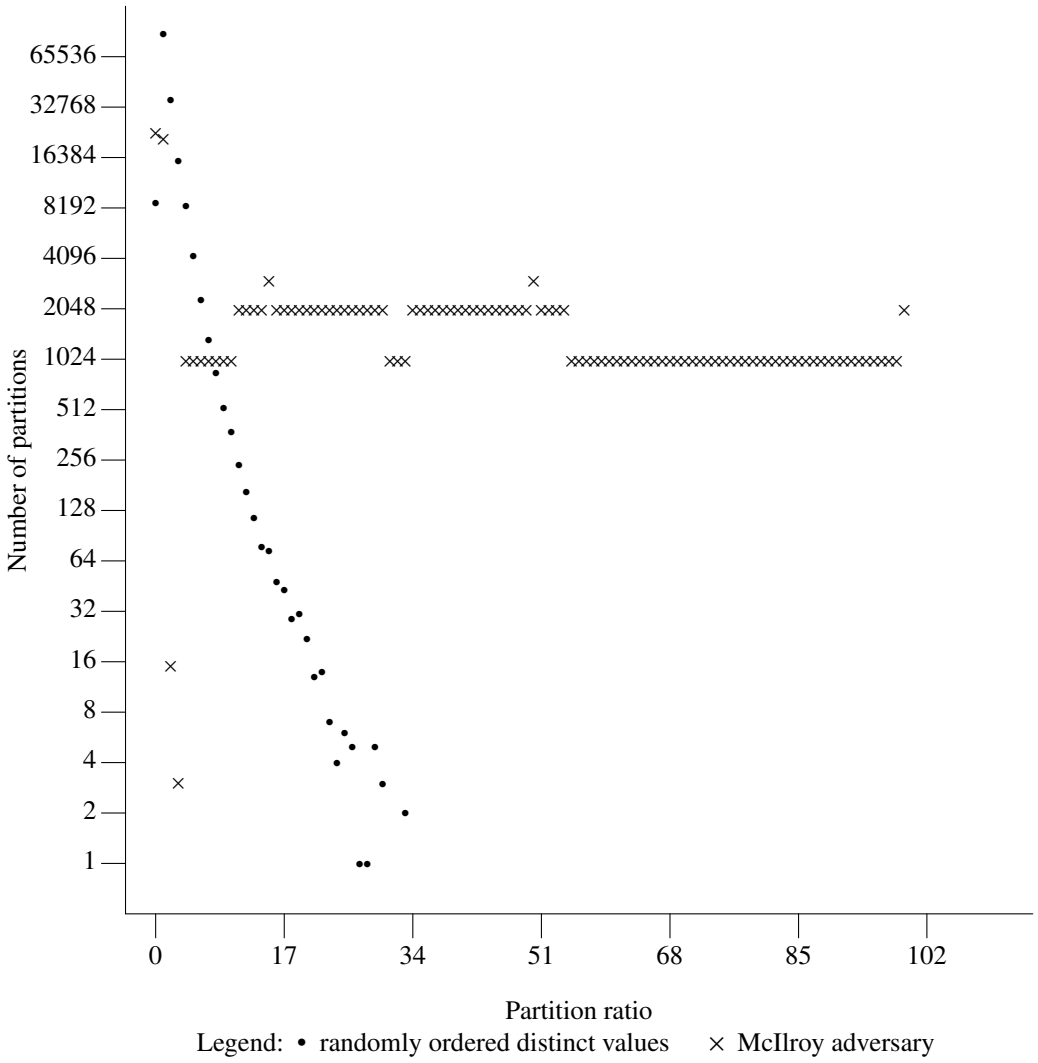

Figure 8: Partitions vs. lopsidedness

sorting. Arrays of the same sizes were sorted with adverse inputs generated by McIlroy's adversary and the corresponding ratios were recorded. Figure 7 shows the largest ratios observed. At modest sub-array size (and therefore small sample size) there is considerable overlap; for example at sub-array sizes through 68 elements (3 samples), the maximum possible ratio occurs when median-of-3 pivot selection results in a single element in the small region and 66 in the large region, giving a ratio of 33:1. Such a ratio as well as smaller ones was observed for sub-arrays resulting from sorting both random and adverse inputs. As sub-array size and sample quantity increase, the ratio for random input first increased (at 9 samples) then decreased quickly, while the largest observed ratio continued to increase for adverse inputs. At 729 samples and beyond, no partition resulting from random input was observed with a ratio as high as 2:1. This sharp divergence in lopsidedness between random and adverse inputs makes detection of adverse sequences at large sub-array sizes easy; non-adverse inputs rarely result in lopsided partitions at large sub-array sizes (with sampling proportional to $\sqrt{N}$). Adaptive detection of adverse inputs at large sub-array sizes is therefore possible, and is discussed in more detail in a later section.

Most partitions resulting from pivot selection using remedian of samples with sample size increasing as $\sqrt{N}$ are well balanced. Figure 8 shows the number of partitions with ratios of large region to remaining elements produced when sorting initial arrays of randomly shuffled distinct valued elements and when sorting adverse inputs. The horizontal axis represents ranges of ratios corresponding to integer division, e.g. 5 represents all ratios in the interval $[5.0, 6.0)$. Array size was 1000 elements and 1000 arrays for each sequence type were generated and sorted. Although it is possible to have very large ratios, none greater than 33:1 were observed for randomly shuffled values in this experiment. There is a clear difference between the characteristics for the two sequence types; while both produce a large proportion of reasonably balanced partitions, and both show a rapid decline as ratios initially increase, that decline stops at a size-dependent

Average performance over all permutations of distinct values

array	quicksort		insertion		network	
size	comp.	swap	comp.	swap	comp.	swap
2	1	0	1	0	1	0
3	5	2	3	2	3	1
4	7	3	5	3	5	3
5	10	4	8	5	9	3
6	14	5	11	7	12	5
7	18	7	15	11	16	6
8	22	8	19	14	19	8
9	26	10	24	18	25	13
10	30	11	30	22	29	11

Best- and worst-case performance

	insertion sort			network
array	best	worst		worst
size	comp.	comp.	swap	swap
2	1	1	1	1
3	2	3	3	2
4	3	6	6	5
5	4	10	10	5
6	5	15	15	10
7	6	21	21	13
8	7	28	28	19
9	8	36	36	24
10	9	45	45	23

Table 1: Small-array sorting performance characteristics

ratio for adverse inputs[1].

12 Small sub-arrays

Quicksort's overhead of sampling, pivot selection, partitioning, and recursion can be relatively expensive for small arrays. It is common to use insertion sort with quicksort for small sub-arrays. Insertion sort has a best-case comparison complexity of $N - 1$ comparisons (and no swaps) for already-sorted input and worst-case complexity (both comparisons and swaps) of $^{N(N-1)}/_{2}$ for reversed input.

It is possible to do even better with a few caveats. First, the worst-case for a candidate sorting algorithm should be no worse than the quadratic worst case for quicksort or insertion sort. Second, the average complexity for operations (comparisons and swaps) should no higher than for insertion sort. Third, overhead should be no higher than for insertion sort's two loops. A decrease in comparison or swap complexity compared to insertion sort is desirable. Performance advantage compared to divide-and-conquer for arrays larger than 6 elements is a bonus.

Optimal sorting networks [23] [24] [25] have very low overhead, exhibit low comparison complexity, and use relatively few swaps. However, due to the data-oblivious nature of sorting networks, the best-case, average, and worst-case comparison complexity are all the same; unlike insertion sort, "easy" inputs such as already-sorted sequences use as many comparisons as any other input sequence.

The use of sorting networks as an alternative to insertion sort for small sub-arrays was evaluated for sorting networks through 10 inputs. Table 1 summarizes performance. All permutations of distinct inputs were sorted using divide-and-conquer alone (with median-of-3 pivot selection for sizes 3 and greater), insertion sort, and sorting networks. The average number of comparisons and swaps at each sub-array size are shown in the upper part of Table 1. Each method exhibits the same number of comparisons and swaps for a sub-array of size 2; the sorting network has the lowest overhead. For sub-array sizes 3 and 4, both insertion sort and sorting networks use on average fewer comparisons and swaps than divide-and-conquer with median-of-3 pivot selection. At 5 or more elements, fewer comparisons are used by insertion sort and

[1]The number of partitions for large ratios with adverse input is equal to an integral multiple of the number of runs (1000) because the input is always the same. The same number of runs was used to make the vertical scaling commensurate with the results for randomly shuffled distinct input values.

sorting networks, but insertion sort swaps more elements on average than the other methods. At 8 elements, the sorting network uses the fewest average comparisons of the three methods, and only marginally more swaps than divide-and-conquer. By 9 elements, there is little to be gained by using insertion sort or a sorting network; both of those methods swap more elements on average than divide-and-conquer, and use only slightly fewer comparisons on average.

Insertion sort performance varies greatly from its best case (already sorted input) to its worst case (reversed input). The disparity can be seen at the far left (small array size) in Figure 2. Sorting networks always use the same number of comparisons regardless of the input data, but the number of swaps is data-dependent. Best-case comparisons and worst-case comparisons and swaps for insertion sort and worst-case swaps for sorting networks are shown in the lower part of Table 1. Sorting networks use the same number of swaps as insertion sort (viz. zero) for the best case (already sorted input), and always use fewer swaps in the worst case (which is a different input sequence for the two methods) above size 2; the variation in performance is smaller for sorting networks.

For the simplest possible comparisons and swaps, run time was lower for sorting networks than for insertion sort except at a size of 3 elements, where insertion sort was slightly faster. The magnitude of the differences in complexities and runtime varied considerably; at 4 elements insertion sort has less than a 1% lower comparison complexity than the sorting network, uses about 16% more swaps, and uses about 5% more run time. At 5 elements, insertion sort uses about 14% fewer comparisons than the sorting network, about 60% more swaps, but less than 1% more run time.

Beyond 8 elements, the lack of data adaption in sorting networks led to anomalies in performance for already-sorted inputs. Therefore, a hybrid arrangement is used for sorting small sub-arrays; insertion sort is used for sub-arrays of 3 or 5 elements, and sorting networks are used for the remaining sizes from 2 through 8 elements. Sub-arrays of 9 or more elements use divide-and-conquer with sample quantity and spacing as discussed in a previous section of this paper, the method also used for order statistic selection for small sub-arrays. Note in Figures 2 and 5 that the number of comparisons for small arrays using sorting networks is independent of the input sequence, whereas there are differences with insertion sort as used through array size 6 in Bentley & McIlroy's implementation, and at sizes 3 and 5 in the present implementation.

13 Selection

A given array size can present an enormous variation in inputs for sorting; for distinct elements, the number of permutations increases as the factorial of array size. Allowing for repeated values, the variation is greater still. Despite the size of the sorting problem posed by the input variation, sorting is reasonably well understood, and the methods described so far suffice to provide efficient solutions which are well-behaved for all possible inputs (performance data are presented in a later section of this paper).

The selection problem is in many ways more difficult than sorting. Although the principle of [multiple] selection is quite similar to quicksort [4] , the problem is more complex; in addition to the large variation of possible inputs, the number and distribution of order statistics sought may also vary greatly. As noted earlier in this paper, [single order statistic] selection performance is much more sensitive to a lopsided partition than is sorting. Whereas sorting by quicksort involves unconditionally processing both regions resulting from each partitioning, selection requires examining the desired order statistic ranks against the partitioned regions' ranks to determine whether or not to process each region, resulting in somewhat higher overhead for selection; the overall effect of the increased overhead depends on the costs of comparisons and swaps. For complex comparisons and expensive swaps, the overhead of examining ranks in order to reduce comparison complexity from linearithmic to linear may favor selection, whereas for simple comparisons and cheap swaps it may be more efficient to sort than to select for some number of order statistics, depending on the distribution of the order statistic ranks.

A practical implementation of multiple selection cannot reasonably account for all possible variations of number and distribution of order statistics. It is possible to manage efficient multiple selection for a few categories of order statistic distribution. Distribution of order statistics is relatively unimportant for very small numbers of order statistics, and at the other extreme as the number of order statistics approaches the number of elements, multiple selection becomes equivalent to sorting as both regions resulting from each partition require processing [1]. If the desired order statistic ranks are tightly grouped (e.g. selection of the m largest (or smallest, or middle, etc.) elements), performance is close to that for selection of a single order statistic even if many order statistics are sought; most partitions result in only one of the two regions requiring further processing. On the other hand, if the desired ranks are distributed throughout the range of

[1] Because multiple selection results in partitioning the array segments between each selected order statistic, selection for alternate element ranks (slightly fewer than half of the array elements) results in a sorted array.

elements, a relatively small number of specified order statistic ranks will result in performance like sorting; most regions resulting from partitioning will require processing.

Experimentation with multiple selection revealed that three categories of order statistic rank distribution could be used to manage reasonably efficient selection:

- order statistic ranks distributed across the array element ranks result in processing most regions resulting from partitioning; for relatively few order statistics, sorting may be more efficient than explicit multiple selection. Breakpoints for sampling quantity used for sorting are suitable for sampling for pivot selection for such a distribution. At large array sizes, sorting is generally more efficient if the number of (widely distributed) order statistics is 10% or more of the array size. Smaller arrays can be efficiently processed by multiple selection for a somewhat greater proportion of desired order statistics.

- order statistics grouped near the middle of the array ranks will result in processing both regions resulting from the initial partitions (assuming the partitions are not too lopsided), but after a few partitions, the regions near the ends of the original array can be ignored as there are no desired order statistic ranks in them. Conversely, the regions near the middle of the original array will likely be processed by sorting, as all sub-array element ranks correspond to desired order statistics. Sample quantity for pivot selection can be higher than for sorting, in order to ensure well-balanced partitions. Because repeated partitioning results in many small sub-arrays, and because ignoring sub-arrays with no desired order statistic ranks avoids the work which would be associated with those sub-arrays (recursively), the regions away from the middle require no processing. For small arrays, the savings is small, and selection of a modest portion of order statistic ranks near the middle results in a performance disadvantage compared to a full sort of the array. At large array sizes, multiple selection may be more efficient than sorting even if more than 90% of the array's ranks (centered) are desired order statistics, a very different situation than for widely distributed order statistic ranks.

- order statistic ranks grouped near one or both ends of the array share some characteristics with grouping near the middle; there are relatively large regions of the original array which will require no processing. In this case however, there is less benefit from well-balanced initial partitions, so the number of samples used for pivot selection can be reduced.

In each case, when the number of desired order statistics favors multiple selection rather than sorting, the thresholds for repivoting are lower than for sorting because of the greater sensitivity of selection to lopsided partitions.

The distribution of the desired order statistic ranks for multiple selection is determined by examining the desired ranks which fall within three bands of element ranks. The distribution is used to determine whether to sort or select based on the ratio of the number of desired order statistics to the number of array elements for the order statistic rank distribution. Because of relatively large variation in selection vs. sorting performance for small arrays, a table is used for array sizes from 6 through 261 elements (arrays of 5 or fewer elements are always more efficiently processed by sorting, except when selecting the median of medians if the extents of the regions are required in order to avoid repeated comparisons when repartitioning after an initial lopsided partition). Larger arrays use the 10% and 90% ratios for widely distributed and for grouped ranks mentioned above. When selection (rather than sorting) is used, separate tables of breakpoints are used for pivot selection sample quantity as described above. In total, three such tables are used; the table for sorting is also used for widely distributed order statistic rank selection, and two additional tables are used for ranks grouped near the middle and for ranks predominantly away from the middle (near one or both ends of the array). Whereas sample quantity for sorting and for widely distributed ranks is proportional to the square root of the array size, sample quantity increases more quickly (proportional to the ⅔ power of array size) for selection of ranks grouped near the array middle. Sample quantity increases more slowly (proportional to the cube root of array size) for order statistic ranks away from the middle of the array. The values in the tables were determined experimentally for practical array sizes (up to a few tens of millions of array elements) and fit to the power functions mentioned.

An aggressive repivoting factor table is used for selection. The repivot factors are based on the number of samples used for pivot selection (divergence of partitioning ratios for random and adverse inputs is the same as shown in Figures 7 and 8).

14 Synergy

Several of the improvements made work together synergistically:

- A wrapper function providing the *qsort* interface permits one-time determination of an appropriate swapping size, avoiding recomputation for recursive calls to the internal sorting function. The wrapper also permits the internal function to keep track of element rank for order statistic selection.
- Median-of-3 is used for pivot selection from 3 samples, and is reused for pivot selection via remedian of samples and via median-of-medians of sets of 3 elements for break-glass pivot selection. Ternary median-of-3 benefits all uses for inputs with many equal elements.
- Use of dedicated sorting for small sub-arrays through size 8 avoids choosing between a single sample for pivot selection and use of median-of-3 pivot selection for very small arrays when sorting.
- Improved quality of sampling (uniform spacing) obviates deferral of pivot swapping at the start of partitioning to reduce introduction of disorder, avoiding the additional comparisons that deferral (or a copy of the pivot) would entail. Improved quality of sampling is necessary to avoid anomalies when sorting rotated input sequences.
- Remedian of samples for pivot selection with an increasing number of samples used as sub-array size increases leads to a progressive improvement in the balance of the resulting partitions for non-adverse inputs and limits worst-case imbalance. That improvement leads directly to near-optimum recursion depth and indirectly facilitates early detection and correction of adverse inputs.
- Implementation of order statistic selection and use of median-of-medians for break-glass pivot selection permits saving ⅓ of the comparisons required for partitioning after repivoting a large region. Selection of medians is facilitated by moving the medians-of-3 to a contiguous region at the middle of the array, using bias in ternary median-of-3 to minimize data movement.

15 Costs and benefits

Some features considered in this paper might require significant additions to code, increasing object file size and maintenance cost. It is prudent to consider the benefits provided against these costs.

- Sampling improvements require a small amount of code, certainly a bit more than simply selecting a middle element, but the performance improvements are substantial. Improved quality of sampling accounts for much of the improved sorting performance of the present work for non-adversarial input sequences. Use of tables of sampling breakpoints is a space vs. time tradeoff, increasing object file size in order to avoid repeated computation of breakpoints.
- Improved median-of-3, taking equality comparisons into account adds no cost, but provides substantial performance improvement for some common input sequences.
- Use of Kiwiel's algorithm L [15] avoids self-swapping to improve performance at the cost of a modest increase in code size.
- Replacement of SWAPINIT and related macros with inline swap functions simplifies maintenance and provides performance improvement. There is a noticeable increase in object file size, but that is because of the additional code which supports efficient swapping in units of size in addition to *char* and *long*.
- Remedian of samples for fast pivot selection is slightly more complicated than Tukey's ninther, but provides performance improvement for large arrays. The code increase is minimal and subsumes computation of ninther; it uses the same basic median-of-3 used by other pivot selection methods.
- An internal stack would add some code, however would also facilitate selection, which might otherwise have to implemented and maintained separately. In addition to being one of the project goals, selection is required for median-of-medians pivot selection, which is used to avoid quadratic sorting behavior. An alternative chosen for the implementation described in this paper is a qsort-compatible wrapper function with an internal implementation that uses a range of indices to bound the sub-array being processed. Small sub-arrays are processed recursively; large ones iteratively, avoiding the overhead of maintaining an internal stack.
- The "break-glass" mechanism consists of examination of the sizes of the regions resulting from partitioning, determination whether or not to repartition a large region, and implementation of a guaranteed-rank pivot selection mechanism (median-of-medians). All of these require some additional

code, parts of which reuse code already used; but without some mechanism, poor sorting behavior for adverse inputs is unavoidable[1]. The break-glass mechanism does not interfere with selection. Performance against adverse input sequences is compared to introsort in a later section of this paper.

- Median-of-medians, which is used to select an improved pivot in conjunction with the break-glass mechanism can function well using sets of 3 elements, which reuses median-of-3 code, otherwise extensively used (in median-of-3 pivot selection and remedian pivot selection).

- Sorting networks rather than insertion sort for some small sub-arrays reduces the average number of comparisons and swaps used for sorting. Their use permits avoiding median-of-3 pivot selection and its additional comparisons for sorting sub-arrays through 68 elements. There is considerable additional code, as the sorting network is a bit different for each size. Because the saving for small sub-arrays accrue for each partitioning of larger arrays, the benefit extends to all array sizes. Use of sorting networks for small sub-arrays significantly contributes to improved performance.

- *Quickselect* operates by partitioning after selection of a pivot element. When the pivot element is selected by median-of-medians during break-glass processing, one third of the array is partitioned as a side-effect of selecting the median of the medians, and need not be reprocessed when partitioning the remainder of the array [18] . Saving about ⅓ of the comparisons required for repartitioning requires some additional code to skip over already-partitioned elements[2]. This further requires some transfer of information about the extents of the partial partitioning via added function call parameters. The additional overhead provides some benefit to a rarely-used part of the whole. The need for information about the extent of the equal-to-pivot region for avoiding recomparison precludes the use of sorting for small sub-arrays when repartitioning with median-of-medians; divide-and-conquer is used down to sub-arrays of 2 elements. Because it provides a small but measurable benefit[3], use of the partial partitioning information to avoid recomparisons is available as a compile-time option in the implementation described in this paper. That option was used for performance graphs and results reported in this paper. An alternate means of avoiding recomparison was proposed by Martínez & Roura [19] : select the samples exclusively from both ends of the array; that would work poorly for input sequences with structure such as organ-pipe sequences and would complicate remedian of samples and median-of-medians.

16 Program and related text analysis

Reading the code of Bentley & McIlroy's qsort, in the course of investigating and resolving the issues discussed in this paper, both as published and as found used in various places highlighted a number of issues that are addressed in the implementations described in this paper. One issue is the terse variable naming chosen by Bentley & McIlroy, which differs from the names given in the standard qsort declaration. It is easier to match code to the specification when variable names, etc. are common in code and specification. Therefore, the implementation described in this paper uses an array pointer *base*, element count *nmemb*, element size *size*, and comparison function *compar*.

Insight: Match code to specification where possible.

17 Assembling the product

The final polymorphic in-place selection and sorting function incorporates components based on analysis, algorithm design, and testing described earlier in this paper.

- An internal function using indices to bracket the sub-array being processed, with an invariant array base pointer so that element ranks may be determined. A wrapper function provides the standard *qsort* interface. When performing (multiple) selection, regions not containing desired order statistic ranks are ignored. Smallest regions are processed first to prevent overrunning the program stack when sorting.

[1] $O(N^{1.5})$ for $\sqrt{N}$ samples is better than ($O(N^2)$, but is much worse than $O(Nlog_2(N))$.

[2] An alternative is swapping the already-partitioned elements to their appropriate regions to avoid recomparison, but the cost of the swaps is prohibitive.

[3] Avoiding recomparisons during repartitioning saves about 2% of comparisons overall when sorting adverse input, and about 10% for order statistic selection of one or two statistics from adverse input.

Sorting sampling table excerpt

Samples	min_nmemb
3	3
9	69
27	617
81	5554
243	49989
729	449904
2187	4049135
6561	36442217

Table 2: Sorting sampling table excerpt

- Partitioning based on the efficient split-end method used by Bentley & McIlroy, as modified by Kiwiel to avoid self-swapping, and with swapping of the pivot element to the farthest array end to reduce disorder.
- A break-glass mechanism to obtain an improved pivot in the event of extremely lopsided partitions, improving performance and preventing quadratic worst-case performance. The mechanism defends against adverse inputs without sacrificing speed.
- A pivot selection algorithm for use with the break-glass mechanism using median-of-medians with sets of 3 elements to provide a guaranteed relatively narrow range of pivot rank, computed at reasonable cost. Optionally, skipping over the partially partitioned elements resulting from median selection, avoiding recomparisons of those elements.
- A fast pivot selection method using remedian with base 3 over a sample of array elements; sample size is a power of 3, increasing slowly as array size increases. The use of more samples for larger arrays provides an improved pivot rank guarantee and statistical efficiency of the pseudo-median.
- Improved sampling of elements for median-of-3 pivot selection and the other pivot selection methods, avoiding array endpoints. Improved sampling results in improved performance for commonly-occurring input sequences, such as organ-pipe sequences, rotated sequences, and reverse-sorted sequences.
- Improved median-of-3 taking advantage of comparison results indicating equal-valued elements.
- Inline functions are used for swapping, eliminating some macros and special-case code; functions support swapping by a variety of available sizes.
- Sorting networks for small sub-array sorting. General performance is maintained by using insertion sort for a few sub-array sizes where insertion sort retains an advantage over sorting networks.
- Tables for sampling breakpoints and for repivoting region ratio thresholds to avoid expensive computation and to accommodate variations in characteristics which are not readily computable.

18 Performance tuning

Performance in sorting and selection is affected by table values used for sample quantity and repivoting decisions. Martínez & Roura [19] relate optimum sample size $s = 2k + 1$ for selection to sub-array size N by the relationship $k = \sqrt{\frac{N}{\beta}} + o(\sqrt{N})$. For sorting, they determined that optimal sample size was proportional to $\sqrt{N}$, which differs somewhat at small sample (and sub-array) sizes. Multiple quickselect was not considered by Martínez & Roura. In the present implementation, sample sizes are powers of 3. Arrays of randomly shuffled distinct input values of various sizes were sorted with varying breakpoints. Within each range of sub-array sizes for a given number of samples there is considerable variation in comparison complexity as sub-array size varies. The minimum variation and the lowest overall comparison complexity was obtained with breakpoints set at the nearest integer value to ${}^{(\ln(2)+1)}/_{2} \times s^2$ for s samples.

Multiple selection uses different breakpoints based on the distribution of desired order statistics, as previously presented in this paper.

Having determined breakpoints for sampling, the next step was to manage repivoting of lopsided sub-arrays resulting from partitioning. The basis for repivoting decisions has been briefly discussed. Pivot rank

Selection middle sampling table excerpt

Samples	min_nmemb
3	3
9	27
27	140
81	729
243	3788
729	19683
2187	102276
6561	531441

Selection separated sampling table excerpt

Samples	min_nmemb
3	3
9	102
27	2756
81	74402
243	2008847
729	54238868
2187	1464449448
6561	39540135107

Table 3: Selection sampling tables excerpts

guarantees provided by median-of-3 and remedian pivot selection methods limit the need for repivoting. Also, when sorting (but not for order statistic selection) large regions with fewer than 10 elements are never repivoted, because the next partition is guaranteed to produce two regions which will both be directly sorted by sorting networks or insertion sort (size at most 8 elements). When selecting order statistics, large regions with fewer than 9 elements are never repivoted, because for 8 elements there would be only 2 sets of medians of 3, and the resulting pivot rank is not guaranteed to be any better than the less costly median of 3 samples for regular pivot selection. For 131072 element arrays of randomly shuffled sequences of distinct integers with repivoting effectively disabled, the sorting complexity is about $1.0061N \log_2 N$ comparisons and adverse inputs can still lead to poor performance, although the improved pivot rank guarantees provided by remedian of samples pivot selection, with the increased number of samples limit the effect compared to Bentley & McIlroy's qsort, and processing small regions first ensures that *quickselect* cannot overrun its program stack.

Based on the data shown in Figures 7 and 8, tables were generated to control repivoting for each range of sub-array size. Two thresholds are used to trigger repivoting: if the ratio of the number of elements in the large region to the remaining elements is at least as large as *factor1*, repivoting takes place. Otherwise, two occurrences of a ratio at least as large as *factor2* will result in repivoting. Three sets of parameters for sorting are shown in Table 4, with a brief performance summary. For the "aggressive" set of parameters, arrays with size through 68 elements use 3 samples: a region with a ratio of 18:1 or larger is immediately repivoted. On the second occurrence of a ratio of 8:1 or more, repivoting takes place (counters are reset after a repivot, and for processing the smaller of the regions resulting from a partition). Likewise for other sub-array sizes; for sub-arrays using 243 or more samples (sub-arrays with at least 49989 elements), a ratio of 2:1 [1] or more always triggers a repivot – Figure 7 shows that that is exceedingly rare. The performance summary for the "aggressive" parameter set indicates $1.007N \log_2 N$ comparisons on average for a 131072 element array of randomly shuffled distinct values, which is 0.091% higher than the scaled comparison factor without repivoting. That is not a great sacrifice in performance. Against McIlroy's adversary, the absolute worst-case scaled comparison factor is $1.4161N \log_2 N$ comparisons at an array size of 321 elements, and the scaled comparison complexity factor at an array size of 16 Mi elements drops to $1.1464N \log_2 N$ comparisons, about 14% higher than for random input[2]. The "relaxed" sorting repivoting parameter set was used for the performance graphs in this paper. The "relaxed" parameter set gives excellent performance, with worst-case

[1] Recall that median-of-medians with sets of 3 elements will produce a ratio of just under 2:1.

[2] With the optional avoidance of recomparisons during partitioning after a repivot, performance numbers are a bit better: $1.0069N \log_2 N$, 0.075% penalty, $1.3967N \log_2 N$ worst-case adverse comparisons (at 28 elements), and $1.1248N \log_2 N$ at 16 Mi elements for the "aggressive" parameter set, for example.

Aggressive repivot tuning parameters

Samples	factor1	factor2
3	18	8
9	16	10
27	60	33
81	6	3
243	2	2
729+	2	2

Relaxed repivot tuning parameters

Samples	factor1	factor2
3	19	9
9	16	10
27	60	33
81	6	3
243	2	2
729+	2	2

Loose repivot tuning parameters

Samples	factor1	factor2
3	34	16
9	90	19
27	86	41
81	6	3
243	3	2
729+	2	2

With repeated comparisons in repartition

Parameters	random shuffle @128Ki		adversary worst	@	@
	scaled	penalty	scaled	size	16Mi
aggressive	1.0070	0.091%	1.4161	321	1.1464
relaxed	1.0068	0.067%	1.4527	21	1.1463
loose	1.0061	-0.001%	1.9791	123	1.1427

No repeated comparisons in repartition

Parameters	random shuffle @128Ki		adversary worst	@	@
	scaled	penalty	scaled	size	16Mi
aggressive	1.0069	0.075%	1.3967	28	1.1248
relaxed	1.0067	0.056%	1.4527	21	1.1247
loose	1.0061	-0.003%	1.9615	123	1.1218

Table 4: Sorting repivot parameter settings and performance characteristics

Aggressive repivot tuning parameters

Samples	factor1	factor2
3	7	2
9	8	2
27	13	2
81	6	2
243	3	2
729+	2	2

Table 5: Selection repivot parameter settings

Processor	RAM	Operating System	Compiler
AMD Athlon LE-1620	4 GB	Linux 3.12.67	gcc 4.8.1
Intel Celeron SL54Q	512 MB	NetBSD 7.0.2	gcc 6.2.0
Intel Core 2 Duo E6550	2 GB	OpenIndiana	gcc 4.8.5
Intel Core 2 Quad Q6600	4 GB	NetBSD 7.1	gcc 6.3.0
Intel i7-5500U	6 GB	Linux 4.11.2	gcc 7.1.1
Intel Pentium III	512 MB	NetBSD 7.0.2	gcc 6.2.0
Intel Xeon 3520	24 GB	NetBSD 7.1	gcc 6.3.0

Table 6: Test hardware and software environment

adverse input generated by McIlroy's adversary scaled to less than $1.5N \log_2 N$[1] with negligible effect on random input sequences. The "loose" parameter set provides worst-case sorting complexity for adverse input sequences better than the best-case for introsort above introsort's insertion sort cutoff and makes a small improvement in sorting random input sequences by repivoting a few occasional lopsided partitions.

Because selection performance is more sensitive to lopsided partitions, a very aggressive repivoting table is used for sub-arrays for which selection rather than sorting is used.

19 Performance results

Comparisons reported below were made with gcc version 7.1.1 using −Ofast optimization, running on a machine with an Intel® Core™ i7-5500U CPU @ 2.40GHz and with 6 GiB main RAM memory. Performance was tested on a variety of machines, with various operating systems, compilers, and processors as shown in Table 6, with generally consistent results.

Attention has been paid to making sure that *quickselect* operates reasonably efficiently on large arrays with arbitrary sequences of element values and with complex comparisons. Bentley & McIlroy's [9] qsort behaves quadratically and crashes easily when presented with adverse inputs at modest array sizes. *Quickselect* maintains linearithmic complexity even against McIlroy's antiqsort adversary [13] . Whereas Bentley & McIlroy's [9] qsort implementation becomes less efficient (in terms of scaled $N \log_2 N$ comparisons) as array size increases, *quickselect* maintains (and even increases) efficiency at large array sizes, as shown in Figure 10. Low comparison count is most important when the comparison function is slow, such as when comparing large data (e.g. long character strings) or when multiple keys need to be compared to resolve partial matches. The comparison complexity data for *quickselect* with randomly shuffled distinct inputs shown in Figure 10 has a least squares fit to $0.985802N \log_2 N + 0.776356N - 7.12398\ {}^{N}/_{\log_2 N}$. Figure 9 is a graph of comparison counts for sorting limited-range random input vs. array size. Limited-range random input for array size N consists of integers in the range [0,N). As such there is a high probability of some repeated values. It is intended to simulate real-world random input, which rarely consists of distinct values. The modified version of Bentley & McIlroy's qsort uses one-time determination of swapping function, ternary median-of-3, modified sampling, Kiwiel's Algorithm L modifications to prevent self-swapping during partitioning, previously discussed cutoff values (9 for median-of-3 and 69 for ninther), swaps in any appropriate basic type size, and processes the smaller partitioned region first, recursively, then processes the large region iteratively. It differs from *quickselect* in limiting the number of samples for pivot selection to at most 9, by lack of provision for order statistic selection, and by lack of protection against quadratic behavior. Some of these differences exhibit effects on the number of comparisons and swaps, as explained in a later paragraph.

In tests, dual-pivot quicksort [16] performed poorly in terms of number of comparisons, number of swaps, and run time compared to the modified version of Bentley & McIlroy's qsort and compared to *quickselect*. With the simplest data types, viz. plain integers, the run time difference was least. With practical data types requiring non-trivial comparisons and/or swaps of modest amounts of data, the dual-pivot quicksort exhibited much longer run times than *quickselect*. It has been conjectured [26] that performance for dual-pivot quicksort may be less bad than would be expected from comparison and swap count for trivial data types due to cache effects. Performance of dual-pivot quicksort can be improved by applying some of the same techniques used to improve Bentley & McIlroy's qsort, viz. increased sample size for pivot selection as array size increases and improvements in swapping, however those improvements (incorporated in Figure 12) are insufficient to achieve competitive performance even for trivial data types. Kushagra

[1] The same maximum – albeit in this case including against McIlroy's adversary – claimed for Bentley & McIlroy's qsort implementation (but actually performing as in Figure 2).

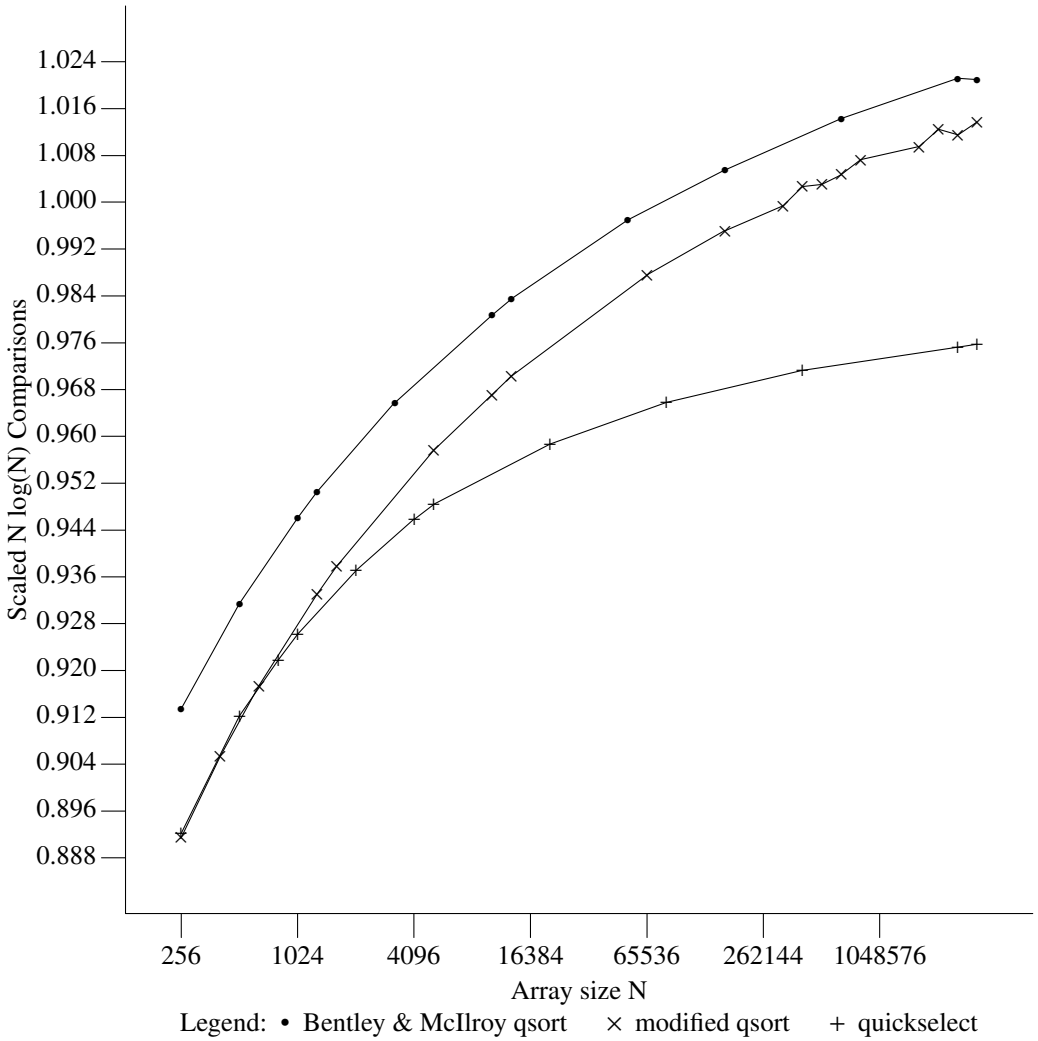

Figure 9: Comparisons for Bentley & McIlroy qsort and *quickselect*, limited-range random input

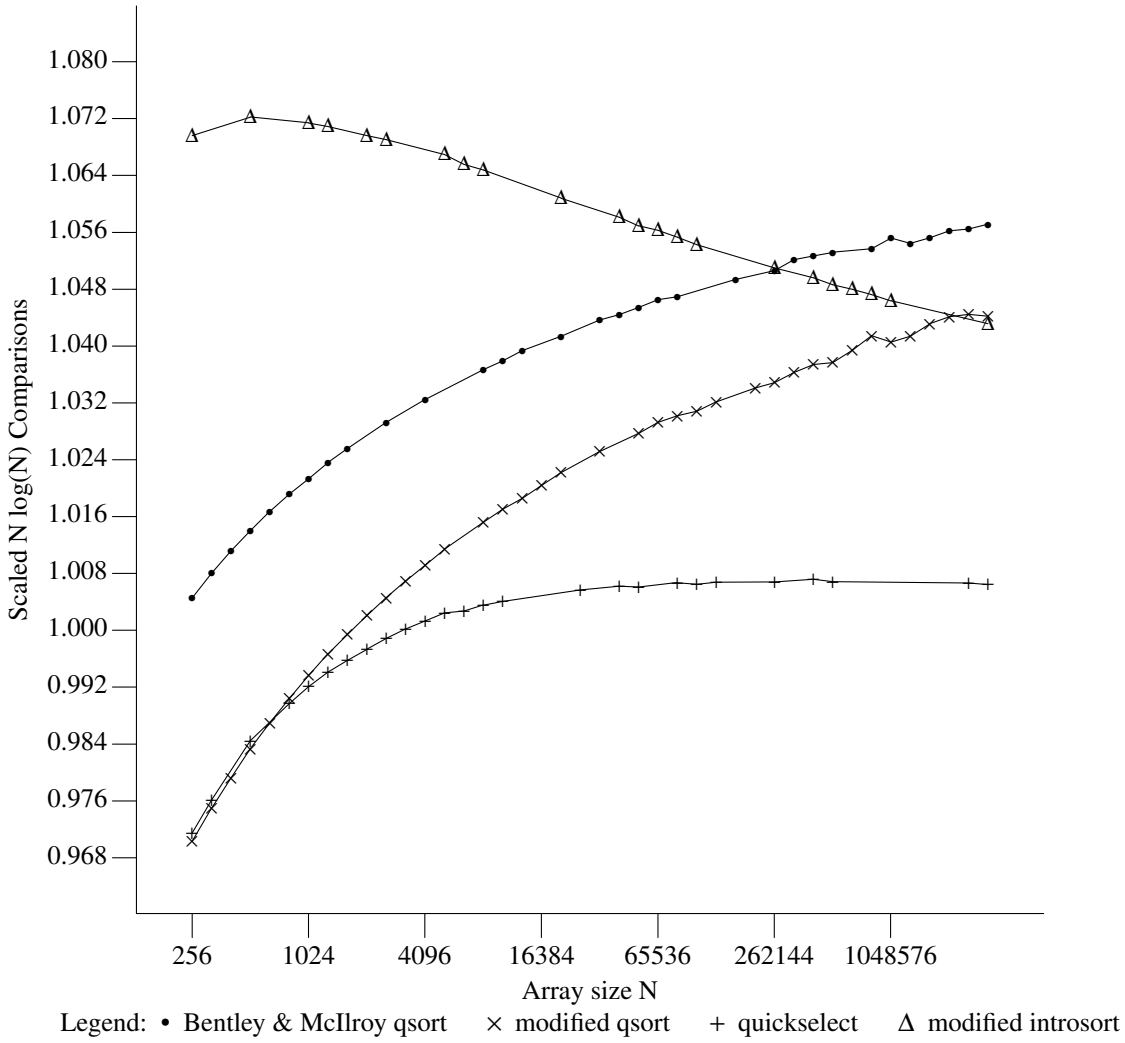

Figure 10: Comparisons for sorting implementations, shuffled input

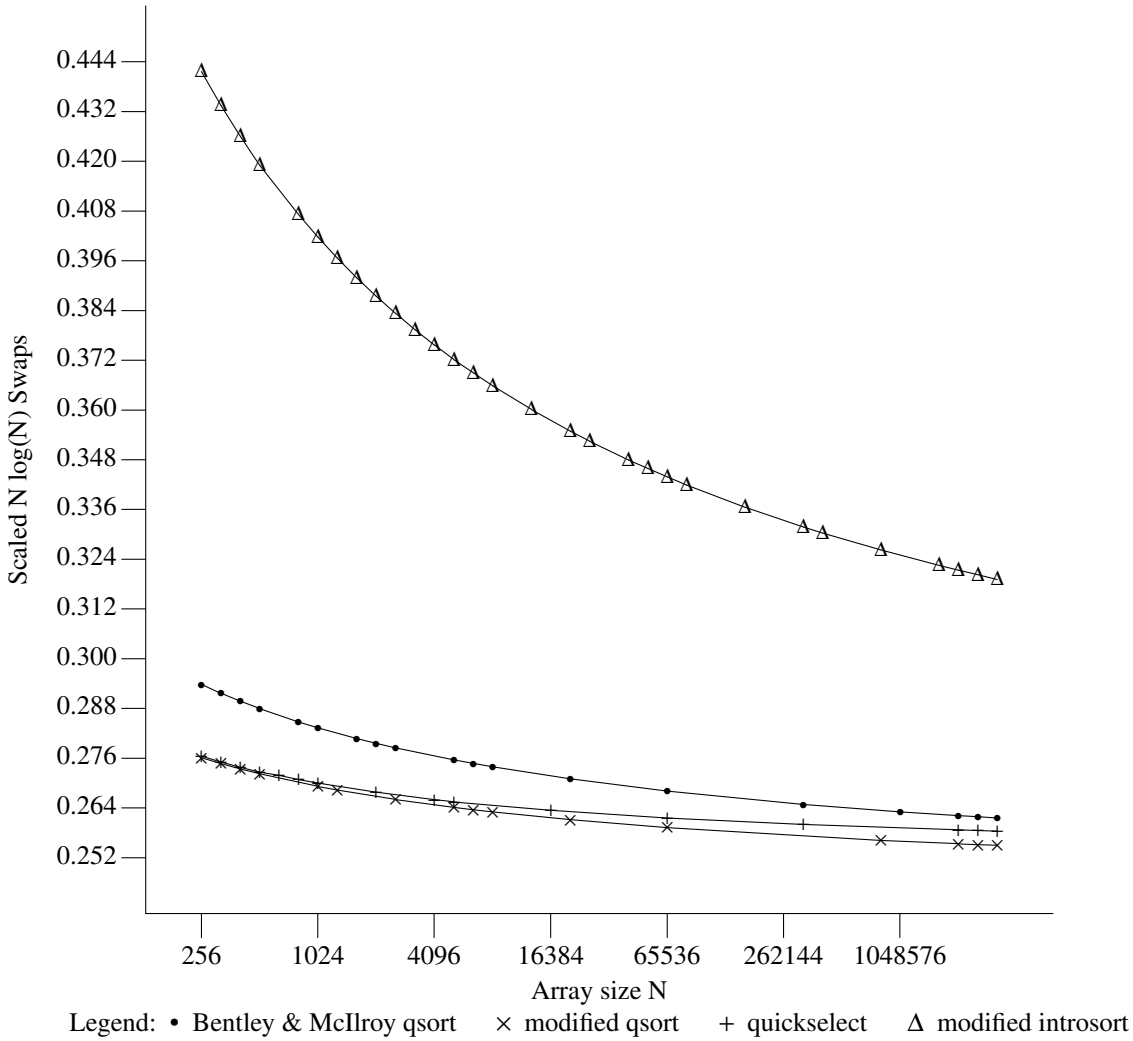

Figure 11: Swaps for sorting implementations, shuffled input

et al. [26] measured 7-8% performance improvement of some dual-pivot sort implementation over some unspecified 1-pivot quicksort using some version of median-of-3 pivot selection with distinct-valued input. The modifications to Bentley & McIlroy's qsort described above result in run time about 5% lower than the unmodified qsort as measured in the present study, which is itself about 5% faster than Yaroslavskiy's dual-pivot sort [16] with a similar polymorphic qsort-like interface, all measured sorting 100000 randomly shuffled distinct plain integers over 1000 trials. Simple input sequences (mod-3 sawtooth, random zeros and ones) that should sort quickly take significantly less time for the modified Bentley & McIlroy qsort than the original which in turn take significantly less time than Yaroslavskiy's dual-pivot sort: differences in each case amounted to more than 20% run time. Analysis by Wild et al. [27] indicates a disadvantage of dual-pivot schemes for selection. Multi-pivot partitioning schemes have several shortcomings for sorting and selection:

- Stack size required for a multi-pivot scheme is significantly larger than for a single-pivot partitioning scheme.
- Partitioning arrays with equal elements requires multiple passes (entailing additional comparisons and swaps), or complex and expensive swaps.
- Pivots must be sorted.
- Truly fast pivot selection methods based on median-of-3 provide an approximation to the sample median, which is ideal for partitioning around a single pivot, but not useful for selecting multiple pivots. The simplest practical methods for selecting reasonably spaced multiple pivots would seem to be sorting (or selecting desired-rank pivots from) a sample of elements.
- Quadratic performance remains possible; eliminating it without hampering selection would be more complex than the method described in this paper for the single-pivot partitioning scheme.

Because of the limitations listed above, the poor practical performance, and the lack of advantages for selection, multi-pivot schemes were rejected as impractical and too low in performance for the in-place polymorphic multiple selection and sorting implementation described in this paper.

An implementation of introsort was also compared to *quicvkselect* (for sorting, not selection). Like dual-pivot quicksort, introsort was only slightly slower in run time than *quickselect* for simple data types, but showed significantly worse performance for types which are non-trivial to compare or swap. Application to introsort of some of the techniques used to improve Bentley & McIlroy's qsort (Bentley & McIlroy's split-end partitioning modified per Kiwiel's algorithm L, with introsort modified to process only less-than and greater-than regions, improved swapping, ternary median-of-3), performance was made even better than *quickselect* for simple data types at moderate array sizes, but not for data types involving non-trivial comparison or swapping or for large arrays. Improvement in sampling quality and quantity yielded additional improvement to introsort for larger arrays, but because the number of comparisons and swaps is greater for introsort than for *quickselect*, introsort performance for data types with non-trivial comparisons or swaps is lower than for *quickselect*. The cause of the higher number of comparisons and swaps is the final insertion sort over the entire array; if that is changed to use insertion sort (or sorting networks) only at the end of the introsort loop on sub-arrays, the number of comparisons and swaps for introsort with all of the above modifications becomes slightly lower than for *quickselect* (because of occasional repivoting in the latter). A final insertion sort over the entire array includes all elements which were selected as pivots during partitioning, and all elements which compared equal to them, also any elements which ended up as the sole element of a partitioned region; those elements are not involved in any of the many smaller sorts when those are performed at the end of the loop. The single final insertion sort over the entire array was used for Figures 10 and 11 in order to illustrate the magnitude of the effect. With smaller sorts at the end of the introsort loop, the number of comparisons and swaps would be essentially the same as for *quickselect*. Introsort avoids quadratic behavior, but shows worse performance than *quickselect* for adverse input sequences (see Figure 15 later in this paper). And there is a separate introselect function for selection (of a single order statistic only). Introsort's interface (array base plus two indices) is workable for selection with or without an internal stack, but heapsort is not readily adaptable to multiple selection for a variable number of order statistics.

Figure 10 is a graph of comparison counts for sorting randomly shuffled distinct input values vs. array size, and Figure 11 shows the scaled number of swaps. Because of the much larger number of comparisons and swaps for dual-pivot quicksort compared to the efficient single-pivot sorts, the corresponding data are shown separately in Figure 12. The modified version of Bentley & McIlroy's qsort uses fewer comparisons and swaps than the unmodified version. The number of comparisons for *quickselect* is lower still, especially at large array sizes, at the expense of slightly more swaps than the modified qsort. At large array sizes,

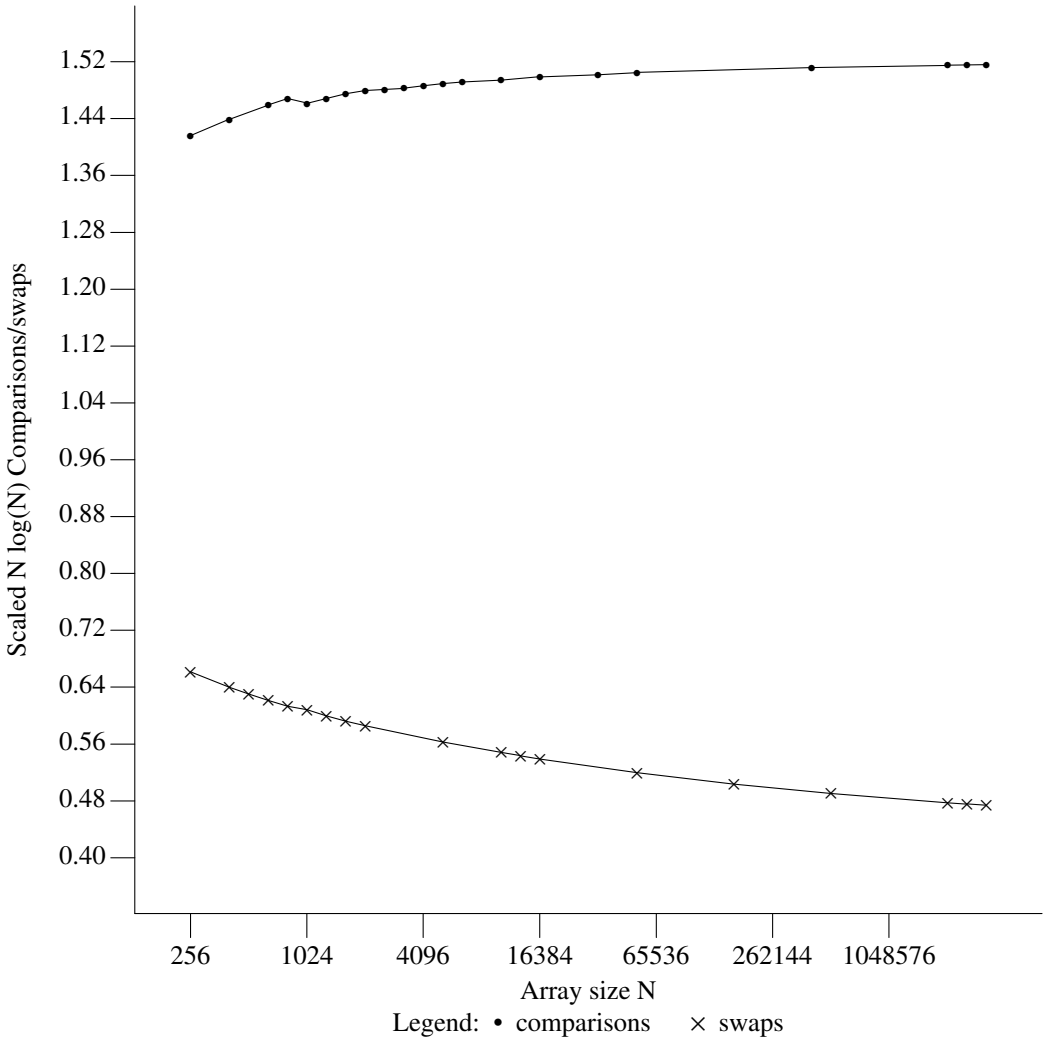

Figure 12: Comparisons and swaps for modified dual-pivot quicksort, shuffled input

Sequence	Bentley & McIlroy qsort		quickselect
	plain	long	
sorted	0.91789	0.91789	0.90198
reversed	1.18971	0.91789	0.93701
shuffled	1.04752	1.04812	1.00663
organ	1.02951	1.03575	0.94444
rotated	1.19044	1.19044	0.94245
binary	0.09032	0.09032	0.09048
equal	0.06021	0.06021	0.06028
random	1.04838	1.04730	1.00669
limited	1.00190	1.00181	0.96627

Table 7: Scaled $N \log_2 N$ Comparisons for Bentley & McIlroy qsort and *quickselect* input sequences

it trades a reduction of $\approx 0.044N \log_2 N$ comparisons for an increase (relative to the modified qsort) of $\approx 0.0039N \log_2 N$ swaps at large N; a net performance increase provided that swaps are no more than 11 times as costly as comparisons. For very large element sizes where swaps might be expected to be expensive, one can use an auxiliary array of pointers to elements, accessing element data for comparisons via the pointers and (inexpensively) swapping the pointers rather than the large elements. Both dual-pivot and introsort implementations incorporating improvements noted above use many more comparisons and swaps than *quickselect*. The reduction in scaled comparisons at large array sizes for the modified introsort and for *quickselect* results from the increased number of samples used for pivot selection for large arrays. That modification was not incorporated into the modified Bentley & McIlroy qsort, but was applied to the dual-pivot implementation, where it results in fewer comparisons than if not applied, but fails to prevent the scaled comparisons from continuing to rise, at least up to the largest array sizes tested.

Figures 10, 11, and 12 show several effects:

- The differences between the unmodified and modified Bentley & McIlroy qsort curves result from sorting networks for very small arrays and improved sampling, which in turn improves performance for large arrays which are partitioned into many small arrays and permits higher cutoff values for median-of-3 and ninther pivot selection methods.
- The differences between the modified qsort and *quickselect* curves results from increasing sample size as the array size increases, using remedian of samples for pivot selection when many samples are taken. The result is considerably fewer comparisons and slightly more swaps for *quickselect* for large arrays.
- The gap between introsort and *quickselect* results from the costly final insertion sort used by introsort.
- The very large gap between dual-pivot and the other curves (necessitating a separate graph to avoid compressing the detail of the others) results from effects of multiple pivots: sampling and pivot selection are more complicated than the fast pseudomedians that suffice for a single pivot, and partitioning is less efficient, requiring approximately 50% more comparisons and twice as many swaps as the efficient Bentley & McIlroy partitioning method. Cache effects lessen the run-time impact of the gap, but cannot overcome the disadvantages.

Bentley & McIlroy's qsort and *quickselect* were tested sorting 100000 element arrays of plain and long integers on a 64-bit machine, with various initial sequences. Each sequence was generated and sorted in 1000 runs, and wall-clock running times and comparison counts were collected. The following sequences were used:

- already-sorted sequence $0, 1, 2, 3..$
- reverse-sorted sequence $..3, 2, 1, 0$
- randomly shuffled distinct integers in the range 0-99999
- organ-pipe sequence .., 49998, 49999, 49999, 49998, ..
- rotated sequence $1, 2, .., 99999, 0$
- random binary-valued elements (i.e. ones and zeros)
- all-equal element values

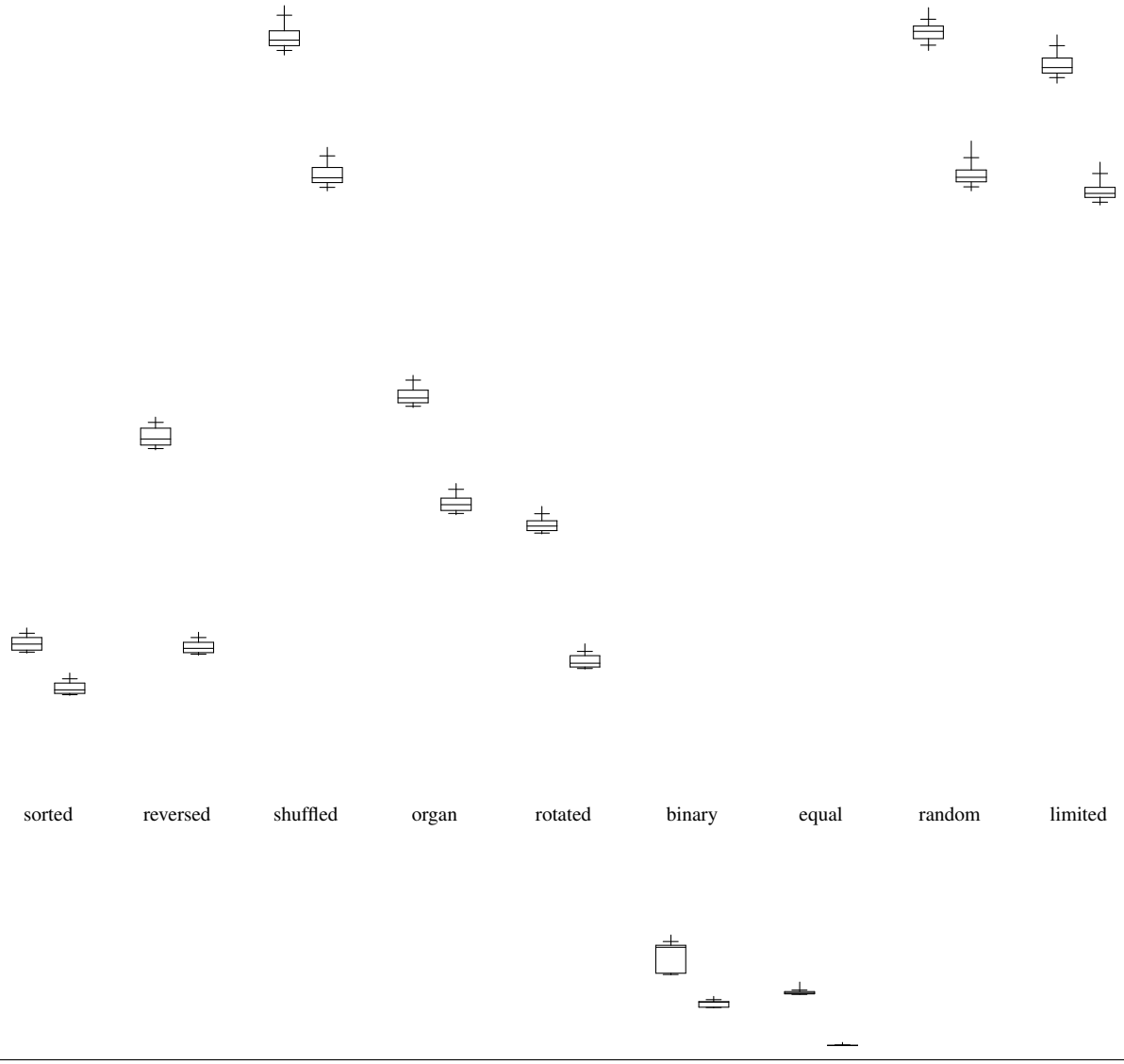

Figure 13: Run time for Bentley & McIlroy qsort and *quickselect* sorting plain integer input sequences

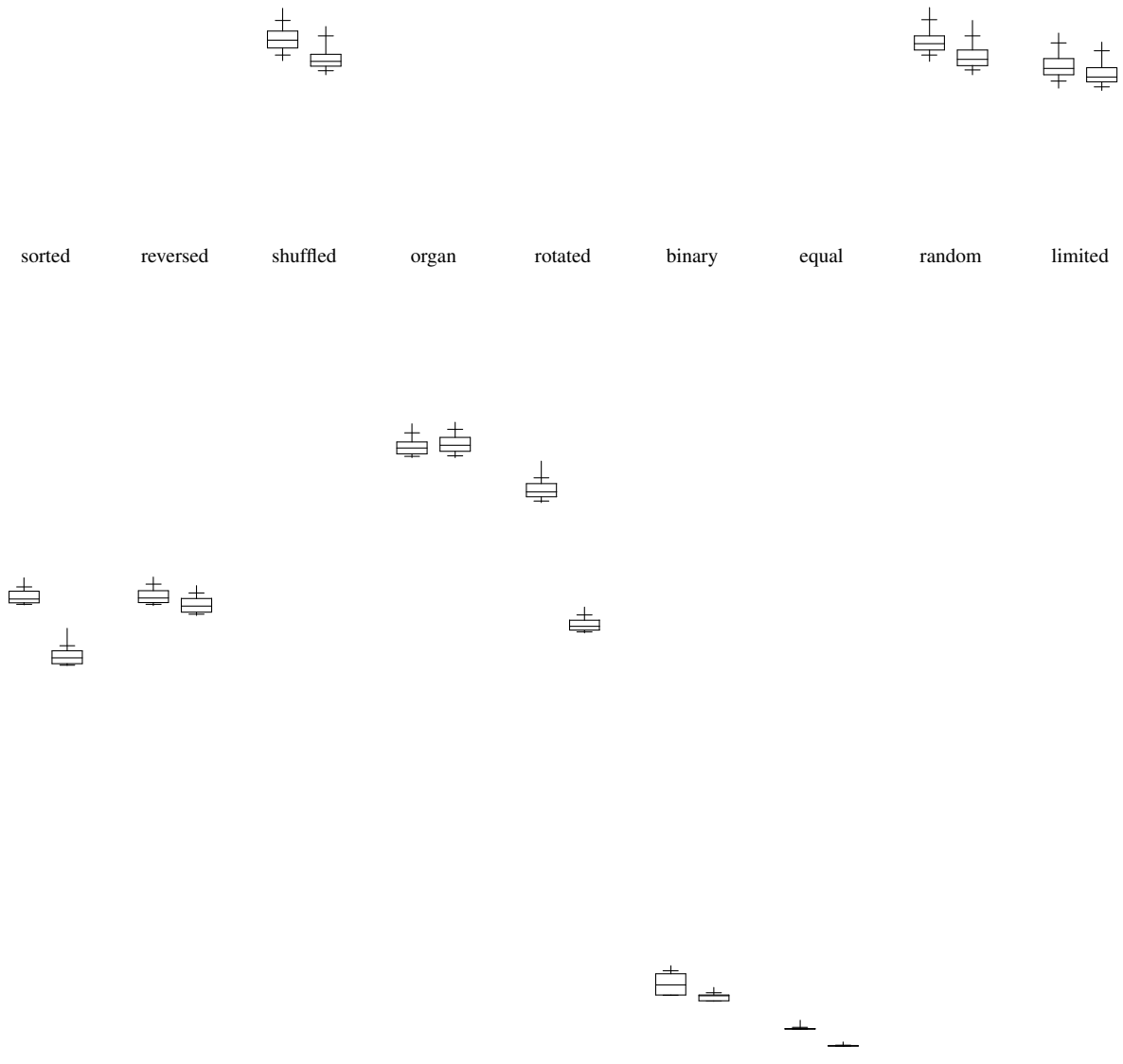

Figure 14: Run time for Bentley & McIlroy qsort and *quickselect* sorting long integer input sequences

- 64-bit random non-negative integers
- limited-range random integers in [0,100000)

Comparison counts were averaged over the runs and scaled to $N \log_2 N$ and are tabulated in Table 7. With three exceptions, *quickselect* shows lower comparison counts than Bentley & McIlroy's qsort.

Running time statistics are plotted as pairs of box-and-whisker plots in Figures 13 and 14. For each input sequence, Bentley & McIlroy's qsort is the left of the pair and *quickselect* is on the right. The central box of each plot extends from the first to third quartile and has a line at the median value. Whiskers extend to the 2 and 98 percentile values, with horizontal ticks at the 9 and 91 percentile values. In several cases, there is very little spread in the running times, so the box-and-whisker plots as a horizontal bar. The vertical axis is linear in run time; a baseline at zero time appears at the bottom. Run time can be compared for different input sequences as well as between sorting implementations. Bentley & McIlroy's qsort is optimized for long integers; its performance is lower for plain integers.

Quickselect is usually slightly faster on already-sorted inputs. The comparison count is usually lower for *quickselect*.

Quickselect is faster at most sizes and data types for decreasing input sequences and rotated sequences (sampling unaffected by pivot movement; see Figures 2 & 5). Comparison count is lower for *quickselect* except for decreasing sequences of long integers.

Quickselect is usually faster, sometimes slightly slower on random inputs (shuffled, full-range, and limited-range) at moderate array sizes, and is always faster for large arrays. The comparison count is significantly lower, so *quickselect* is expected to run faster when comparisons are costly.

Quickselect is usually faster but sometimes slower for organ-pipe inputs and the comparison count is lower.

Quickselect is usually faster for random zeros and ones (improvements in median-of-3, swapping). However, comparison count is marginally higher for *quickselect*.

Bentley & McIlroy [9] noted that

> many users sort precisely to bring together equal elements

Quickselect is much faster than Bentley & McIlroy's qsort for all-equal inputs (improvements in median-of-3, swapping). However, comparison count is marginally higher for *quickselect* when sorting all-equal inputs. The higher comparison count is a result of the larger number of array elements sampled for pivot selection by remedian of samples. It is more than compensated for by the improvements in median-of-3 and swapping. Performance of *quickselect* is also usually superior for input sequences with many (but not all) equal elements.

There are no discernible effects because of pivot movement and/or sampling nonuniformity (see Figures 2 & 5).

Quickselect tends to run significantly faster for data types which differ from *sizeof(long)* (improvements in swapping; reduced comparison count). *Quickselect* does not intentionally alter basic performance with data type size. Bentley & McIlroy's qsort alters performance for data types with alignment and size equal to long integers; compare Figures 13 and 14.

Performance improves relative to Bentley & McIlroy qsort with larger array sizes because of slower (or no) growth (or even reduction) in scaled comparisons (improved pivot rank through use of remedian-of-samples), as shown in Figure 10. The data for Table 7 and Figures 13 and 14 were collected using arrays of 100000 elements. This size represents a moderate difference between performance of *quickselect* and Bentley & McIlroy's qsort, and is roughly worst-case for scaled comparisons for *quickselect* with randomly shuffled distinct input values. The performance difference is less at small array sizes and becomes increasingly large at large array sizes (see Figure 10).

Quickselect is much faster (non-quadratic) vs. McIlroy's antiqsort adversary and other adverse input sequences (because of the break-glass mechanism and guaranteed-rank-range pivot selection). See Figure 15, which shows performance of original and modified Bentley & McIlroy qsort, *quickselect*, and a modified introsort vs. McIlroy's adversary. Bentley & McIlroy's qsort quickly becomes quadratic. The higher cutoff values for median-of-3 and ninther pivot selection in the modified version result in higher comparison counts for highly adverse input sequences at small array sizes, but lower comparison counts for very samll and for large arrays. Introsort's recursion depth limit and use of heapsort when that limit is reached results in large-array comparison count which is heapsort plus $O(N)$ times the recursion depth limit factor. The implementation of heapsort used with the tested implementation of introsort uses about $1.87N \log_2 N$ comparisons for an array of 16 Mi elements; introsort with a recursion depth limit of $2 \log_2 N$ uses about $3.87N \log_2 N$ comparisons to sort that size array. The number of swaps is dominated by heapsort, and approaches $1N \log_2 N$ for large arrays. Some details are dependent on the specific implementation of heapsort

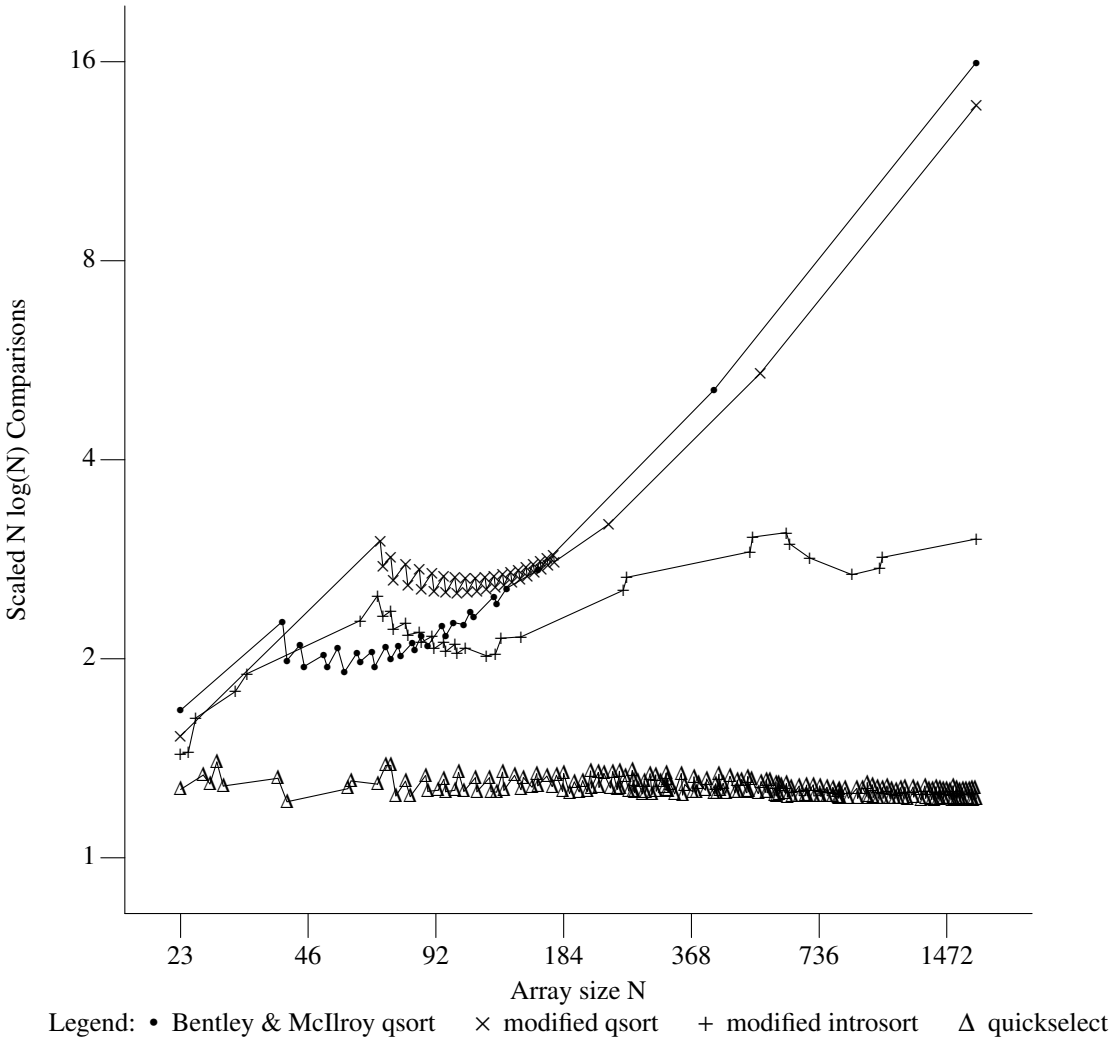

Figure 15: Adverse input comparisons for several sorting implementations

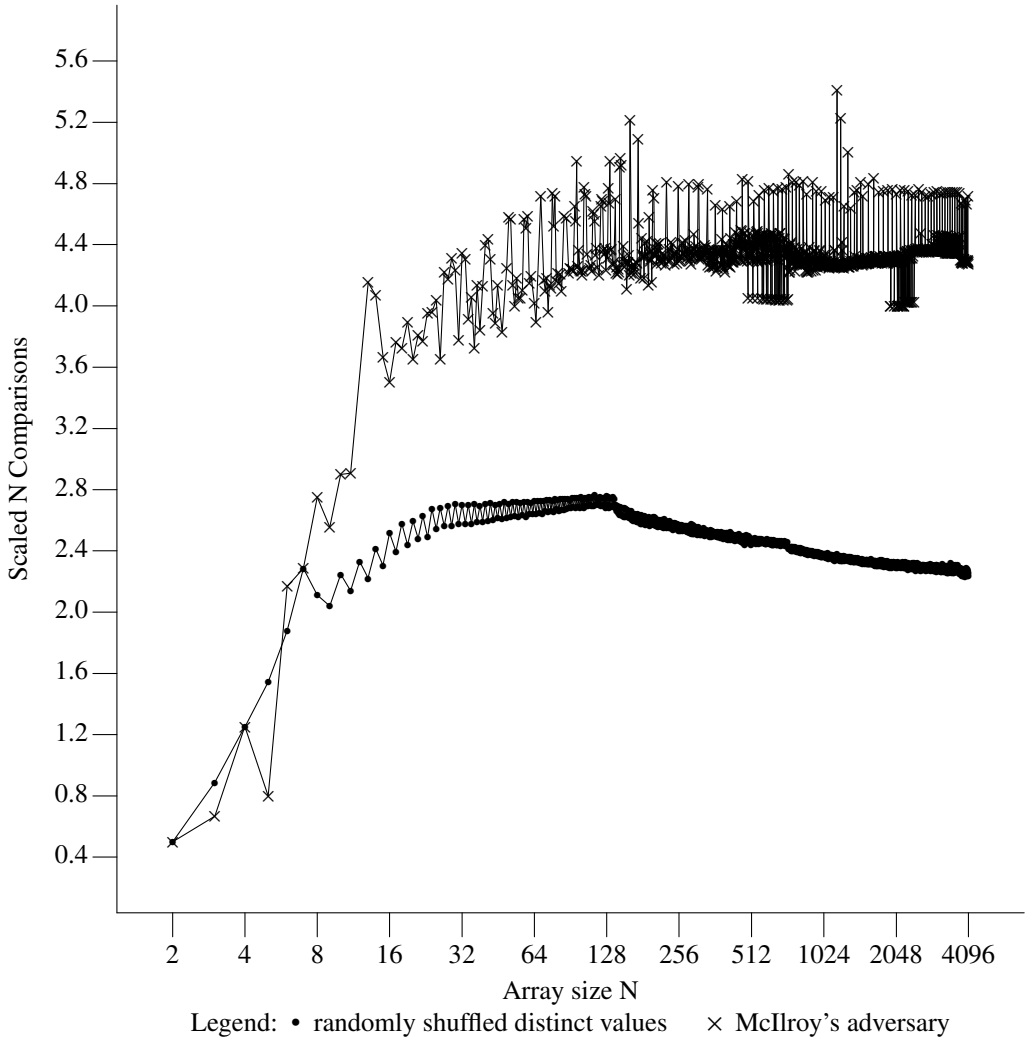

Figure 16: Scaled comparisons for *quickselect* median(s) selection

(or other "stopper" algorithm), but for a given recursion depth limit $k \log_2 N$, adverse input will result in at least $kN \log_2 N$ comparisons. Introsort typically uses $k = 2$ for the recursion depth limit, resulting in a minimum asymptotic complexity of $2N \log_2 N$ (typically considerably higher) for adverse input sequences[1]. The reduced number of comparisons and swaps used by *quickselect* results in significantly lower run time with adverse input (locality of access, poor in heapsort, is also a factor). The maximum asymptotic complexity for *quickselect* can be made lower than the minimum asymptotic complexity for introsort, for adverse inputs. It should be noted that McIlroy's adversary [13] produces results which are as much indicative of the adversary as of the quicksort variant against which it is used, as McIlroy has been careful to point out. Valois [11] describes it as "a strong adversary" and "unrealistic", but it has some limitations compared to possible worst-case input sequences. It uses repeated comparisons with an element as a heuristic to detect the pivot; this fails utterly for multi-pivot quicksort variants and is somewhat confounded by pivot selection using even a modest number of samples. Values for non-pivot elements compared during pivot selection may be "frozen", limiting the extent to which the adversary can force the pivot rank. The effect of this limitation can be seen in Figure 15, where the scaled number of comparisons drops after each increase in the number of samples used for pivot selection.

Of course *quickselect* can be used for efficient selection (in linear time for a single order statistic), while other sorting implementations considered cannot. As noted in the introduction, a full sort is an extraordinarily expensive way to obtain an order statistic. On random inputs, *quickselect* finds medians[2] using about $2.1N$ comparisons for mid-sized (131072 elements) arrays. Simple cases (e.g., all-equal) take slightly more than $1N$ comparisons to find the median. McIlroy's adversary can push the asymptotic median-finding comparison count up to around $4.68N$ with an absolute worst-case of $5.8055N$ for an array of 1162 elements[3]. Figure 16 shows comparisons scaled to array size for finding the median(s) of arrays of randomly shuffled distinct values and as pitted against McIlroy's adversary. The randomly shuffled input curve has a least squares fit to $1.41903N + 12.3249\ {}^{N}/_{\log_2 N} - 26.8567\ {}^{N}/_{(\log_2 N)^2}$.

20 Source files, documentation, testing framework

Source files and documentation for *quickselect* and the testing framework, tools for generating graphs, etc.may be found at `https://github.com/brucelilly/quickselect/`.

21 Future directions

In fast pivot selection, each median of a set of elements can be computed independently of other sets. Break-glass pivot selection using median-of-medians can be similarly parallelized. Processing sub-arrays may proceed in parallel. Sorting networks are readily parallelized.

The principle compute-bound task in quicksort is partitioning, and the primary partitioning mechanism could be partially parallelized; provided that the pivot element is either swapped initially or maintained as a copy, the left and right scanning could take place in parallel, with some synchronization mechanism for swapping and continuation when there is a greater-than element on the left and a less-than element on the right.

The implementation described in this paper has not taken advantage of these opportunities.

22 Conclusions

A polymorphic function has been designed which provides in-place sorting and order statistic selection, including selection of multiple order statistics. Using several techniques from a prior sorting implementation with a few new ones, sorting performance has been enhanced while adding selection capability. Improved quality and quantity of sampling, an extended method of pivot element selection which benefits large problem size, ternary median-of-3, more versatile swapping, sorting networks for small array sorting, and a recovery mechanism to prevent quadratic worst-case behavior have been combined with efficient block swapping and efficient partitioning.

Swapping and sampling improvements, an increased number of samples for large arrays with remedian of samples for pivot selection, and incorporation of Bentley & McIlroy's efficient partitioning method with Kiwiel's improvements, can be applied to implementations of Musser's introsort, which itself provides a

[1] Valois' variant reportedly runs 4 to 6 times as slow as heapsort; poorer performance than Musser's version.

[2] Both lower- and upper-medians for arrays with an even number of elements.

[3] As with sorting, the optional avoidance of recomparisons during partitioning after a repivot improves performance: worst-case $5.4096N$ comparisons (at 1162 elements) and $4.35N$ at 16 Mi elements.

different mechanism to avoid quadratic worst-case behavior. Improvements to swapping and sampling can also be applied to multi-pivot sorting methods, but multi-pivot partitioning inherently requires more comparisons and swaps than Bentley & McIlroy's efficient single-pivot partitioning scheme, and performs poorly for input sequences with non-distinct values and for non-trivial data types. *Quickselect* as described outperforms dual-pivot sorting for all data types and array sizes; significantly so for non-trivial data types. Modified introsort can run faster than *quickselect* for trivial data types at moderate array sizes and non-adverse sequences, but not for non-trivial types nor for very large arrays, unless the final insertion sort is replaced by smaller dedicated sorts.

Finally, the break-glass mechanism for avoiding quadratic behavior applies equally well for sorting and for selection, and provides better performance for adverse inputs than introsort's recursion-depth limit, at the expense of a tiny increase – typically a small fraction of 1% – in cost for sorting non-adverse input sequences[1].

The contributions of this work are:

- Explanation of the effect of poor sampling on pivot selection, especially the all-too-common case of first, middle, and last elements for median-of-3, and the means for improvement of sampling. The improvement is applicable to many sorting and selection implementations.
- Practical implementation of near-optimal sampling quantity for pivot selection, documented in theoretical papers but rarely implemented. This improvement is also applicable to many sorting and selection implementations.
- A mechanism for early detection and correction of poor partitioning balance due to adverse inputs, while avoiding significant performance penalties and maintaining compatibility with both sorting and order statistic selection.
- Application of sorting networks rather than insertion sort for sorting small sub-arrays, which reduces the comparison and swapping costs on average.
- Combination of sorting and multiple order statistic selection in a single implementation.

23 Related work

Chen & Dumitrescu [28] have reported on a median approximation method which is related to median-of-medians with sets of 3 and can be considered as finding the median of a truncated version of remedian with base 3. Alexandrescu [29] has used this in a (single order statistic) selection function.

24 Acknowledgments

The published papers by Lent & Mahmoud [4] and Bentley & McIlroy [9] have been indispensible in development of the function described in this paper. McIlroy's antiqsort [13] provided a useful tool for rigorous testing of sorting functions. Papers by Blum, Floyd, Pratt, Rivest, & Tarjan [2] and by Rousseeuw & Bassett [17] were essential for developing the median-of-medians and remedian of samples pivot selection implementations. Analysis of optimal sampling by McGeoch & Tygar [18] and by Martínez & Roura [19] provided a basis for sample size for pivot selection as a function of array size. Musser's introsort paper [10] provided ideas for alternatives to an internal stack for maintaining access to element ranks. John Gamble's web site [24] provided a useful way to generate sorting network implementations.

References

[1] Mike Paterson. Progress in selection. In *Proceedings of the 5th Scandinavian Workshop on Algorithm Theory*, SWAT '96, pages 368–379, London, UK, UK, 1996. Springer-Verlag.

[2] Manuel Blum, Robert W. Floyd, Vaughan R. Pratt, Ronald L. Rivest, and Robert E Tarjan. Two papers on the selection problem: Time bounds for selection [by Manuel Blum, Robert W. Floyd, Vaughan Pratt, Ronald L. Rivest, and Robert E. Tarjan] and expected time bounds for selection [by Robert W. Floyd and Ronald L. Rivest]. Technical Report CS-TR-73-349, Stanford, CA, USA, 1973.

[1] Note that the "loose" repivoting table provides worst-case adverse-input performance better than introsort's best-case adverse-input performance while actually **decreasing** average non-adverse input sorting cost.

[3] 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.

[4] Janice Lent and Hosam M. Mahmoud. Average-case analysis of multiple Quickselect: An algorithm for finding order statistics. *Statistics & Probability Letters*, 28(4):299–310, August 1996.

[5] C source code/find the median and mean. URL `https://wwwb-front3.us.archive.org/web/20150507020959/http://en.wikiversity.org/wiki/C_Source_Code/Find_the_median_and_mean`. [7 May 2015].

[6] C++ program to compute the median of numbers. URL `https://wwwb-front3.us.archive.org/web/20150514144742/http://www.sanfoundry.com/cpp-program-compute-median-numbers`. [14 May 2015].

[7] C program to find the median of n numbers. URL `https://wwwb-front3.us.archive.org/web/20160204001614/http://www.programmingwala.com/2013/07/median-n-numbers-c-program.html`. [4 February 2016].

[8] C programming code for mean, median, mode. URL `https://wwwb-front3.us.archive.org/web/20150626164209/https://www.easycalculation.com/code-c-program-mean-median-mode.html`. [26 June 2015].

[9] Jon L. Bentley and M. Douglas McIlroy. Engineering a sort function. *Software–Practice and Experience*, 23(11):1249–1265, November 1993.

[10] David R. Musser. Introspective sorting and selection algorithms. *Software–Practice and Experience*, 27(8):983–993, August 1997.

[11] John D. Valois. Introspective sorting and selection revisited. *Software: Practice and Experience*, 30(6):617–638, 2000.

[12] Richard Wesley Hamming. *Numerical Methods for Scientists and Engineers, Second Edition*. Dover Publications, Inc., New York, 1986.

[13] M. D. McIlroy. A killer adversary for quicksort. *Software–Practice and Experience*, 29(4):341–344, April 1999.

[14] URL `http://cvsweb.netbsd.org/bsdweb.cgi/src/lib/libc/stdlib/qsort.c?rev=1.22`. [9 March 2016].

[15] Krzysztof C. Kiwiel. Partitioning schemes for quicksort and quickselect. *CoRR*, cs.DS/0312054, 2003.

[16] Vladimir Yaroslavskiy. Dual-pivot quicksort. URL `http://codeblab.com/wp-content/uploads/2009/09/DualPivotQuicksort.pdf`. [4 February 2017].

[17] Peter J. Rousseeuw and Gilbert W. Bassett Jr. The remedian: A robust averaging method for large data sets. *Journal of the American Statistical Association*, 85(409):97–104, 1990.

[18] C. C. McGeoch and J. D. Tygar. Optimal sampling strategies for quicksort. *Random Structures & Algorithms*, 7(4):287–300, 1995.

[19] Conrado Martínez and Salvador Roura. Optimal sampling strategies in quicksort and quickselect. *SIAM Journal on Computing*, 31(3):683–705, March 2002.

[20] Sebastiano Battiato, Domenico Cantone, Dario Catalano, Gianluca Cincotti, and Micha Hofri. An efficient algorithm for the approximate median selection problem. In *Proceedings of the 4th Italian Conference on Algorithms and Complexity*, CIAC '00, pages 226–238, London, UK, UK, 2000. Springer-Verlag.

[21] URL `http://plan9.bell-labs.com/sources/plan9/sys/src/libc/port/qsort.c`. [9 April 2017].

[22] The times they are a-changin'; — the official bob dylan site. URL `http://bobdylan.com/songs/times-they-are-changin/`. [11 June 2016].

[23] K. E. Batcher. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, AFIPS '68 (Spring), pages 307–314, New York, NY, USA, 1968. ACM.

[24] John M. Gamble. Sorting networks. URL `http://pages.ripco.net/~jgamble/nw.html`. [4 December 2016].

[25] Michael Codish, Luís Cruz-Filipe, Michael Frank, and Peter Schneider-Kamp. Twenty-five comparators is optimal when sorting nine inputs (and twenty-nine for ten). *CoRR*, abs/1405.5754, 2014.

[26] Shrinu Kushagra, Alejandro López-Ortiz, Aurick Qiao, and J. Ian Munro. Multi-pivot quicksort: Theory and experiments. In *ALENEX*, 2014.

[27] Sebastian Wild, Markus E. Nebel, and Hosam Mahmoud. Analysis of quickselect under Yaroslavskiy's dual-pivoting algorithm. *Algorithmica*, 74(1):485–506, January 2016.

[28] Ke Chen and Adrian Dumitrescu. *Select with Groups of 3 or 4*, pages 189–199. Springer International Publishing, Cham, 2015.

[29] Andrei Alexandrescu. Fast deterministic selection. *CoRR*, abs/1606.00484, 2016.