Sorting and selection without tears

This is a pre-peer-reviewed version of an article which may be submitted for publication by the author.

Bruce Lilly
bruce.lilly@gmail.com

April 19, 2018

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. During implementation of a high-performance polymorphic in-place selection function based on quicksort-like divide-and-conquer techniques, some remaining performance issues in well-known and widely-used implementations 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 sorting implementations and which may be applicable to other software projects. Improved quality of sampling for pivot element choice, especially compared to median-of-3 using the first, middle, and last elements, is one example of such an improvement.

1 Introduction

Although the selection problem is well known to have linear bounds [1] [2] [3] [4], 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 (all 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. A final argument provides for tweaking operation, e.g. to more aggressively reduce the number of comparisons used. In the implementation, if the array of order statistics is absent (i.e. has a *NULL* base pointer) or has zero size, a full sort is performed. Therefore, the implementation (hereafter referred to as *quickselect*, in italics to differentiate it from the generic quickselect algorithm) 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 *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. Although the original objective was an implementation of order statistic selection as a library function, research into the mechanisms responsible for the shortcomings found in quicksort implementations have led to a library function which may also be be used for sorting, with performance (if partial order stability is not required) rivaling many other sorting implementations.

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 moving 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. Multiple quickselect 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.

Goals for *quickselect* were:

- Multiple order statistic selection, with sorting also handled as a simplified case.
- Sorting performance on par with or better than other sorting implementations for common structured input sequences and for random inputs.
- Linear selection and linearithmic sorting performance even with adverse inputs.
- Polymorphic comparisons with a *qsort*-compatible interface.
- Data type independence for swapping as well as comparisons via an internal polymorphic swapping function interface.
- In-place operation (other than at most logarithmic stack space for recursion).

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 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, 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:

- median-of-3-killer [10]
- 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 (uniformly distributed full-range integers, or limited to the range [0,N), or restricted to the range $[0,\sqrt{N})$, randomly shuffled distinct integers, approximately normally distributed random integers [12], random integers with a reciprocal distribution [12], $\pm 6\sigma$ histogram data for approximately normal distribution [12])
- A modified version of McIlroy's antiqsort adversary [13]. The modifications support multiple data types, provide for greater adversity of input by limiting the number of elements "frozen" during pivot selection, and reduce problem size from both ends of the array to provide greater adversity for median selection.
- 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 selection and sorting

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.

In many applications, multivariate data is sorted or order statistics are found using one or more keys. A subset of these applications require that data elements comparing equal with respect to a key remain in the same partial order after sorting. This characteristic of a sorting or selection function is called "stability". If a hierarchy of comparison conditions is known in advance, a comparison function which resolves partial equality by comparing additional element fields can be constructed; ideally no elements will compare equal unless truly identical in all fields, in which case stability is guaranteed even if the sorting or selection function makes no inherent guarantee of stability. Use of such a hierarchical comparison function to order data in one sorting (or selection) operation is likely to be considerably more efficient than multiple sorting operations using different key comparisons with a stable sorting (or selection) function. However, in some cases the comparison conditions are not known in advance, for example when additional conditions are imposed based on the results of previous sorting or selection operations. It is possible for quicksort and quickselect to provide a stability guarantee if 3 conditions are met:

1. Partitioning, the fundamental reordering operation used, preserves partial ordering.

2. Dedicated sorting or selection operations, typically used for small sub-arrays, provide a stability guarantee.

3. No other operations (e.g. pivot selection) alter partial orderings.

Sorting and selection stability is available as an option to the caller of *quickselect*, although there is a considerable performance penalty as will be explained in subsequent sections of this paper.

Order statistic selection using a variant of quickselect is subject to the same potential problems noted above for quicksort. 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 unfavorably lopsided 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.

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 emergencies, 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. Rather than process the large region by again selecting a pivot which may result in another small reduction in problem size, it 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. In median-of-medians, medians are obtained for sets of elements, then the median of those medians is found. Median-of-5 requires a minimum of 6 comparisons for distinct-valued elements. If finding the median (of medians) has some cost kM for 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 $(^{(6+k)}/_{5})N$[1].

Sets of 3 elements can be used; the rank guarantee is 1/3 to 2/3 asymptotically, which is a tighter bound than for sets of 5. The cost of obtaining the median of a set of 3 distinct-valued elements is on average 8/3 comparisons giving an overall cost for median of medians using sets of three elements of $(^{(8/3+k)}/_{3})N$. If the cost k of selection is less than $^{8}/_{3}$[2], the tighter bound on pivot rank provided by median-of-medians using sets of 3 elements can be 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. Problem size reduction is $O(N)$, but may be less than ideal. The total complexity of sorting an array of size N is

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

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:

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

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

[2] If repeated comparisons are avoided during repartitioning (described in a later section), the cost of selection may rise to 56/15.

Rearranging terms:

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

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 $c = a + b \approx 1$. 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). For example, a split with approximately 1/3 of the elements in one region and 2/3 in the other, $d = 3$, yields a complexity of $\approx 1.089N \log_2 N$. A much more lopsided partition, with 15/16 of the elements in one region produces a complexity of $\approx 2.965N \log_2 N$.[1]

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 $1.089 \times (1.889 + 0.333k) = 2.057 + 0.363k$. If the number of comparisons per element for median selection, $k = 2.667$, then repivoting is advantageous for $d > 13$. For $k \approx 2$, repivoting is advantageous for $d > 9$.

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 sub-arrays; note also that the final step of the derivation specifies large N. 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 problem size. 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. 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. That expectation is confirmed by performance testing; results are presented in a later section of this paper.

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[2].

Bentley & McIlroy [9] noted that their implementation

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

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

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 *char* and *long* (4 times larger than *char* on 32-bit architectures) only; in the present implementation four sizes representing 1, 2, 4, and 8 times the size of *char* are supported. The wrapper function which provides the *qsort* interface determines the appropriate size initially based on array alignment and element size, and a pointer to the appropriate swapping function is passed to the internal quicksort/quickselect implementation. Use of a function call for swapping loses the inline swap optimization of Bentley & McIlroy's implementation for elements with size and alignment equivalent to *long* types, but provides substantial performance improvements for other element sizes, and avoids unexpected changes in performance as basic type sizes evolve.

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.

Bentley & McIlroy gave a cost model as:

MIX: overhead $\approx$ comparisons $<$ swaps
qsort: overhead $<$ swaps $<$ comparisons

which strictly speaking only applies to sorting elements with size and alignment equivalent to *long* types. For other types[1], and in *quickselect* for all types, a more appropriate cost model for basic types is:

overhead $<$ swaps $\approx$ comparisons

and for complex types significantly larger than the basic type size which is used for data movement:

overhead $<$ comparisons $<$ swaps

with the caveat that the relative costs of comparisons and swaps is somewhat under the control of the library function caller. Comparisons might be arbitrarily complex (e.g. in order to resolve partial equality). Although swapping functions are provided by the library to provide efficient swapping of types with size and alignment corresponding to the basic types used, direct swapping of large elements involves multiple data swaps per element. The caller can mitigate that cost by indirection; sorting pointers to elements (swapping a pointer requires a single data swap per pointer) and providing a comparison function which accesses the element by indirection. With care the caller can almost always ensure that swaps are no more costly than comparisons.

Swapping elements can disrupt partial ordering, precluding stable sorting and selection. Rearranging elements by rotation of sub-arrays preserves partial ordering.

7 Element rotation

There are a number of algorithms for array element rotation [16]. The method used in *quickselect* uses pairwise swaps as illustrated in Figure 1. The rotations shown rearrange group "bcd" to "cdb" while preserving order within each group, effectively interchanging block "b" with the block comprised of groups "c" and "d". The top example in Figure 1 shows a rotation using 4 swaps. If blocks to be interchanged are of unequal size, rotation can be achieved by first rotating by a distance equal to the smaller block size, which places that many elements into final position, then the remainder can be placed into position by another application of the same algorithm.

8 Partitions

Bentley & McIlroy considered various partitioning schemes, arriving at one which starts with swapping the selected pivot element to the first position[2], 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] [17] were investigated in the present work. The split-end scheme described by Bentley & McIlroy [9] as modified by Kiwiel's algorithm L [15] is fastest. Its performance advantage is a result of four factors:

1. Aggressive reduction of problem size; 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. Partitioning schemes that do not separate elements comparing equal to a pivot or which use multiple pivots to separate additional regions fare poorly; they entail additional comparisons which result in additional swaps.

[1] Bentley & McIlroy show a function based swap which swaps long-sized data as 4 chars, at about 4 times the cost of a comparison; a single data type swap via function call would therefore cost about the same as a comparison (also a function call).

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

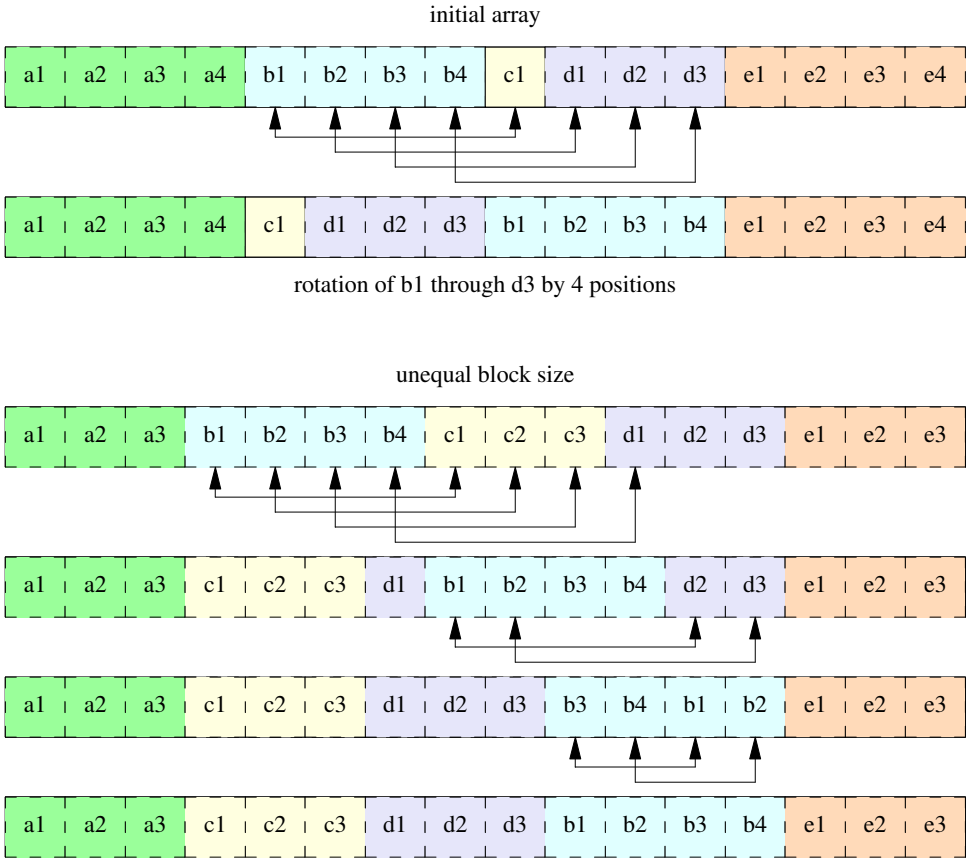

Figure 1: Rotation of array elements by pairwise swapping

2. Efficient use of swaps through 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.

3. Additional swapping efficiency because 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 one side. That efficiency is partially offset by the need to swap the equal elements back to their place between less-than and greater-than regions.

4. 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 swaps compared to multi-pivot schemes[1]. Partitioning in multi-pivot variants of quicksort always uses at least as many comparisons as single-pivot quicksort using reasonable pivot-selection methods.

Quicksort and quickselect with an efficient partitioning algorithm use fewer swaps than comparisons; combined with the relative costs of those operations, the dominant performance measure is the number of comparisons used.

The partitioning methods considered above do not preserve partial ordering of elements, and therefore cannot be used if sorting stability is a requirement. Disruption of partial ordering occurs in the above scheme when the pivot element is initially swapped, when out-of-place elements are swapped, and when canonical partitioning order is restored by block swaps.

An in-place means of partitioning while preserving partial ordering can be constructed from a procedure for merging adjacent partitions. Figure 2 shows two adjacent partitions[2]. The partitions may be merged by 1 or 2 rotations which interchange regions. In simple cases, rotation of the 4 middle regions by 2 regions produces a merged partition, with the two regions containing elements comparing equal to the pivot out of order, which can be restored by a second

[1]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 over the array (with additional comparisons and swaps) during partitioning to process elements comparing equal to one or more of the pivots, or requires more complex multi-element swaps. Moreover, use of Napierian logarithms complicates comparisons; the oft-reported result of $2Nln(N)$ for single-pivot quicksort corresponds to $1.386N \log_2 N$, which applies to random selection of a pivot element from an array of distinct elements, and $1.9Nln(N)$ and $1.8Nln(N)$ (variously reported for dual-pivot implementations) correspond to $1.3167N \log_2 N$ and $1.2474N \log_2 N$ respectively, whereas e.g. Bentley & McIlroy reported $1.188N \log_2 N$ for quicksort using a simple median-of-3 (single-)pivot selection and $1.094N \log_2 N$ for their implementation.

[2]Some regions may be empty.

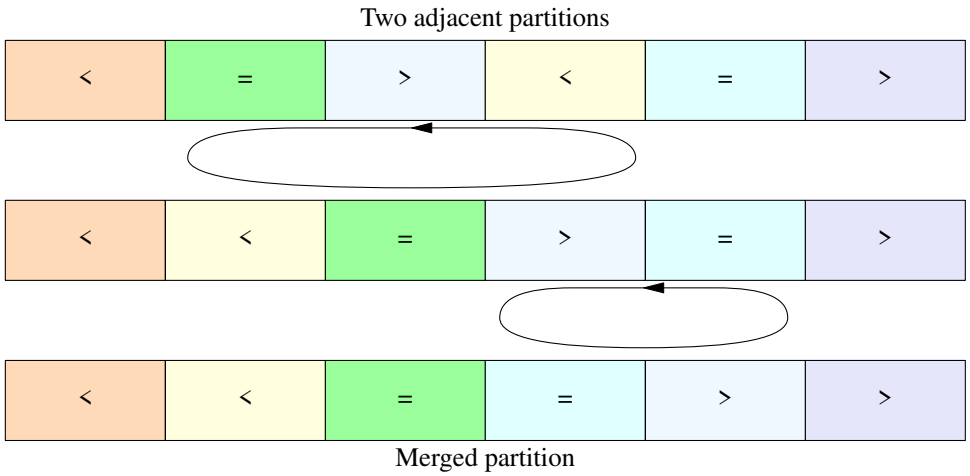

Figure 2: Merging two adjacent partitions, preserving partial order

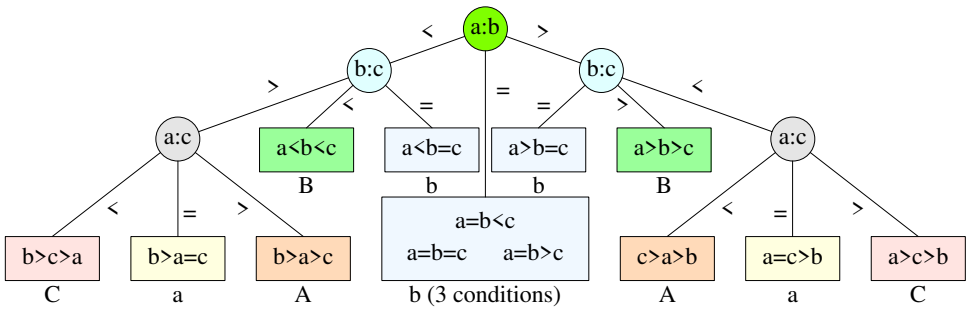

Figure 3: Ternary median-of-3 decision tree

rotation. The example shown in Figure 2 using two rotations may be more efficient depending on the relative sizes of the regions, and a third alternative uses essentially a mirror image of the rotations shown in the figure. Partitioning of a sub-array employing the merging of partitions begins with the pivot element, which is a partition having only an equals region. The elements to the left of the pivot, if any, are partitioned (using recursion), as are the elements to the right of the pivot. Two merges complete the partitioning of the sub-array. If the pivot element is not within the bounds of the sub-array, the sub-array is instead split into two pieces which are partitioned, then merged. The total number of comparisons used for a sub-array of size N is $N-1$, the same as for a conventional partition. The number of swaps is greater, $O(N \log_2 N)$, which results in $O(N(\log_2 N)^2)$ sorting complexity and $O(N \log_2 N)$ selection complexity, but with no additional memory requirement.

Insight: Explore options; test, measure, and compare.

9 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 (as in many other implementations) 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[1]. Modification of the median-of-3 function 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 9 all-equal elements requires 9 rather than 11 comparisons[2] to sort if equal comparison results are used.

In Figure 3, 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

[1] Bentley & McIlroy's implementation performs as poorly as possible for all-equal elements, making three comparisons when one would suffice.

[2] Twelve comparisons in Bentley & McIlroy's implementation for data with size and alignment of *long*.

bias as shown in Figure 3, 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 comparison 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 3 to the binary decision tree shown as Program 5 in Bentley & McIlroy [9].

10 Pivot element selection

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

1. A single sampled element at the middle, used only for a sub-array with 7 elements.
2. The 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 second and third methods use binary comparison 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 [18], 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 [18] 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 [19] and Martínez & Roura [20] 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 [21]. 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, and it precludes sorting stability.

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 and to permit stable sorting and selection.

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 benefit of doing so generally would not outweigh the cost.

The implementation of median-of-medians for pivot selection with a tight range of pivot rank reorders data elements and therefore precludes stable sorting and selection. The remedian, computed over as many data elements as practical rather than a small sample, is used instead of median-of-medians for pivot selection when repivoting is necessary and when stability is required.

11 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

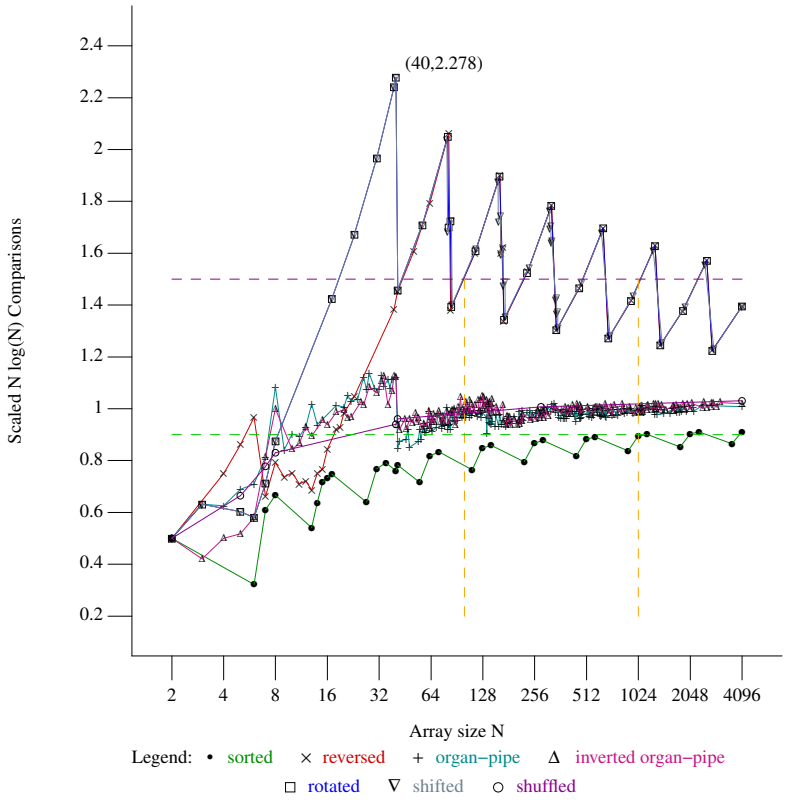

Figure 4: Comparisons for Bentley & McIlroy qsort

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 using Bentley & McIlroy's qsort for short integer data are shown in Figure 4[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 4 includes a dashed horizontal line at $1.5N \log_2 N$ comparisons mentioned in Bentley & McIlroy's paper [9] (the other dashed lines will be explained shortly).

Although performance was measured in the present study at each value of array size N from 2 through 4096 for the data for Figure 4, 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 most graphs 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 4 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.

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

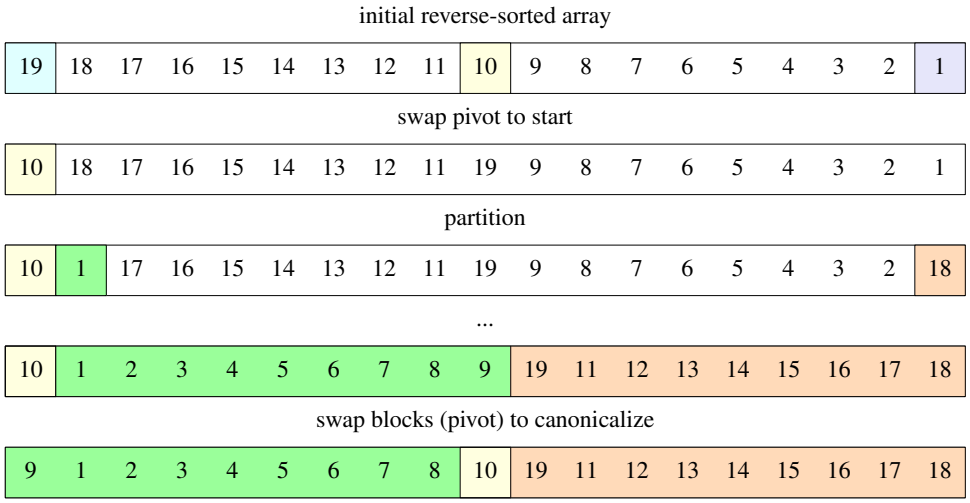

Figure 5: Partitioning of reverse-sorted input in Bentley & McIlroy qsort

and it is this disorder (reordering) 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 4. The peaks above the dashed horizontal line at $1.5N \log_2 N$ evident in Figure 4 would not have been noted at the array sizes tested by Bentley & McIlroy, which were 100 and 1023–1025 (dashed vertical lines in Figure 4).

It is informative to work through an example to see exactly how and why this reordering 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 5. At the end of partitioning, **both** regions resulting from the partition have an extreme-valued element at the first position. Both regions are rotated sequences, which Valois [11] found to result in poor performance, and Figure 4 confirms that rotated sequences exhibit at least as poor performance as 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 convert reversed sequences to rotated sequences in the partitioned regions, and if pivot selection uses median-of-3 with samples at the array endpoints, the same poor performance with rotated sequences (whether input or transformed from reversed input) 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 in the partitioned regions.

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 reordering caused 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 and alignment); macro PVINIT places the pivot in a separate location rather than swapping to the first position, but only for types of the same size and alignment 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 2 decades later, sorting doubles does not exhibit the same behavior noted by Bentley & McIlroy; the behavior instead shows up for other data types, such as the short integers used to generate Figure 4.

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.

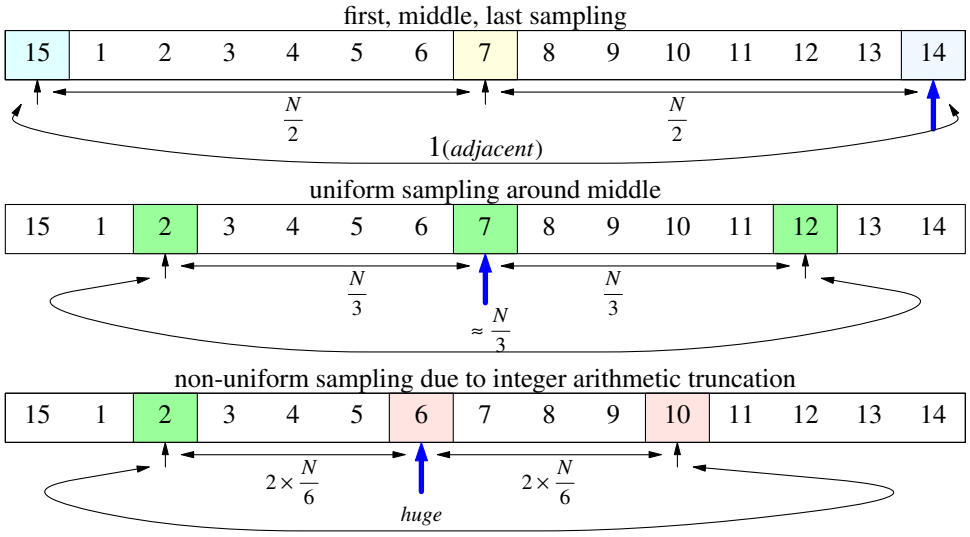

Figure 6: Median-of-3 sampling at ends and middle, uniformly spaced about middle, and non-uniform due to truncation

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 array elements are reordered as shown for reverse-sorted inputs.

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.

Uniform sampling is achieved for 3 samples by sampling at ⅙, middle, and ⅚ positions; this is not optimal for unrotated organ-pipe inputs (first, middle, last sampling is the **worst** possible sampling for such a sequence!) but performance is not much worse than optimal, and remains reasonable as that sequence is rotated.

Consider the partitioned regions at the bottom of Figure 5, which are increasing sequences rotated one position to the right, a larger version of which is shown in Figure 6. Sampling at the endpoints and middle, as is often done in quicksort implementations, produces a poor pivot element (shown with a longer and thicker arrow below the element) with a rotated or organ-pipe input, resulting in a very lopsided partition, and poor performance as can be seen in Figure 4. Uniform sampling around the middle works as well for already-sorted and reverse-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 [22] 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 6. The present work computes ⅓ spacing and applies that from the middle element, which produces more nearly uniform sampling at all sub-array sizes.

Although plan9 qsort sampling is not quite uniform due to integer arithmetic truncation, it is more uniform than sampling the first, middle, and last elements. The improvement can be seen in Figure 7, which does not exhibit the very high and erratic response to rotated input sequences. The peaks with organ-pipe input at 9 and 15 elements are due to use of the middle element only for the pivot at size 9 (the middle element is one of the worst possible choices for a pivot for a bitonic sequence), and due to the integer arithmetic truncation at size 15 (see Figure 6). Plan9 qsort does not use insertion sort or any other dedicated sorting method for small sub-arrays, which partly accounts for the somewhat higher comparison count, especially at small sizes. Also a factor at larger sizes is the fact that plan9 qsort never uses more than 3 samples for pivot selection. The dashed horizontal line at 0.90 $N \log_2 N$ appearing in the performance plots for qsort implementations with non-random input sequences may be used to compare implementations.

Bentley & McIlroy's qsort is not alone in using the first, middle, and last elements for pivot selection. GNU glibc qsort [23] also uses those elements. Like plan9 qsort, glibc qsort never uses more than 3 elements for pivot selection. As glibc qsort uses first, middle, and last elements for pivot selection at all sizes, its performance with bitonic input sequences is particularly poor, as shown in Figure 8. Performance with bitonic and rotated input sequences continues to deteriorate as array size increases.

With modified sampling, performance of Bentley & McIlroy's qsort operating on reverse-sorted, rotated, 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 9. 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 reordering caused by swapping the pivot to one end of the array. In Figure 9, insertion sort is used for arrays of 6 or fewer elements, as in Bentley & McIlroy's qsort, leading to a relatively large number of comparisons for reverse-sorted input

[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.

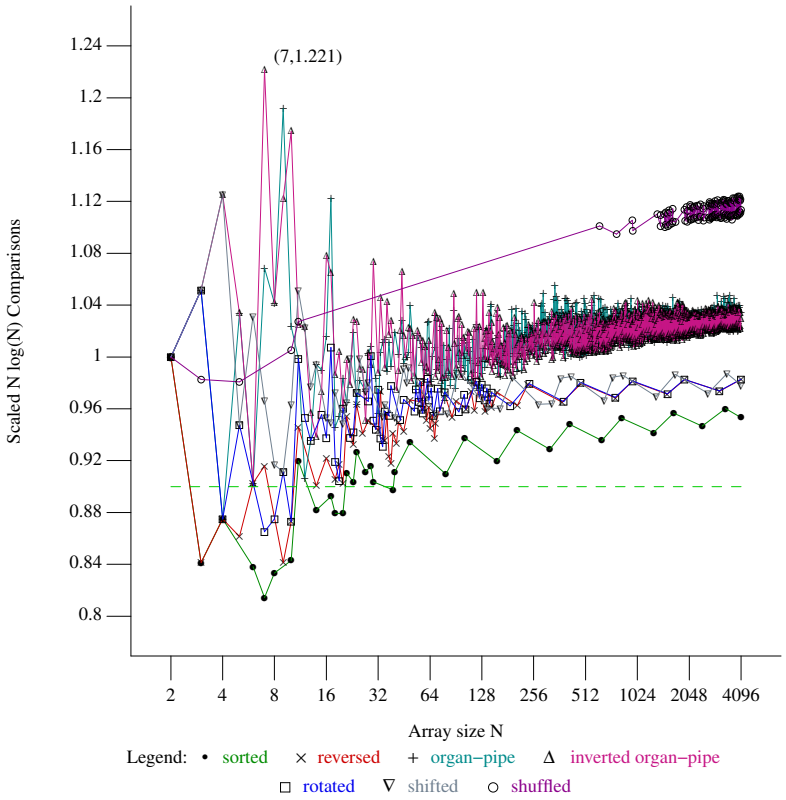

Figure 7: Comparisons for plan9 qsort

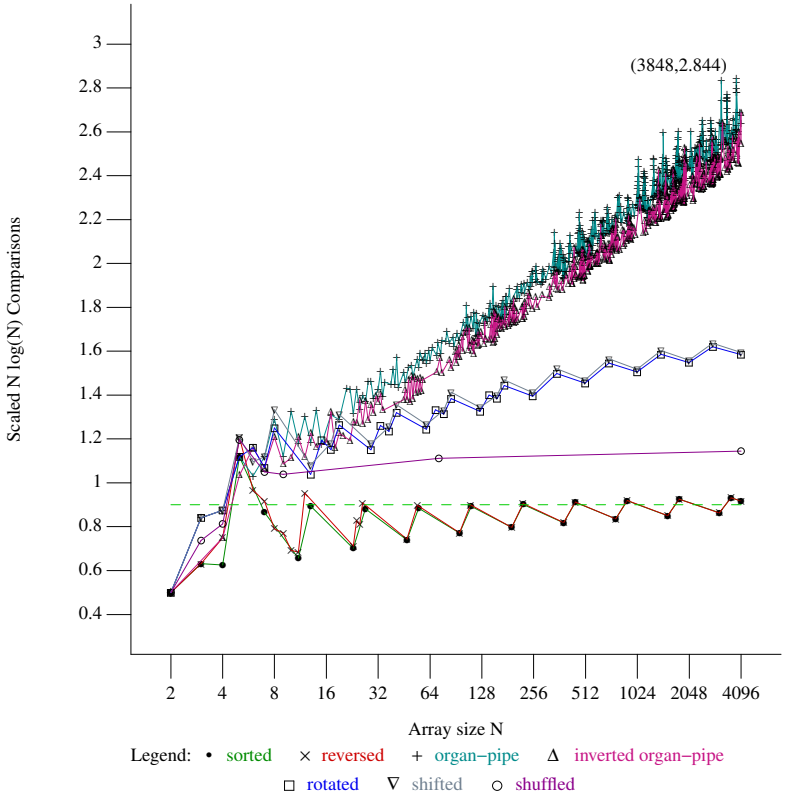

Figure 8: Comparisons for GNU glibc qsort

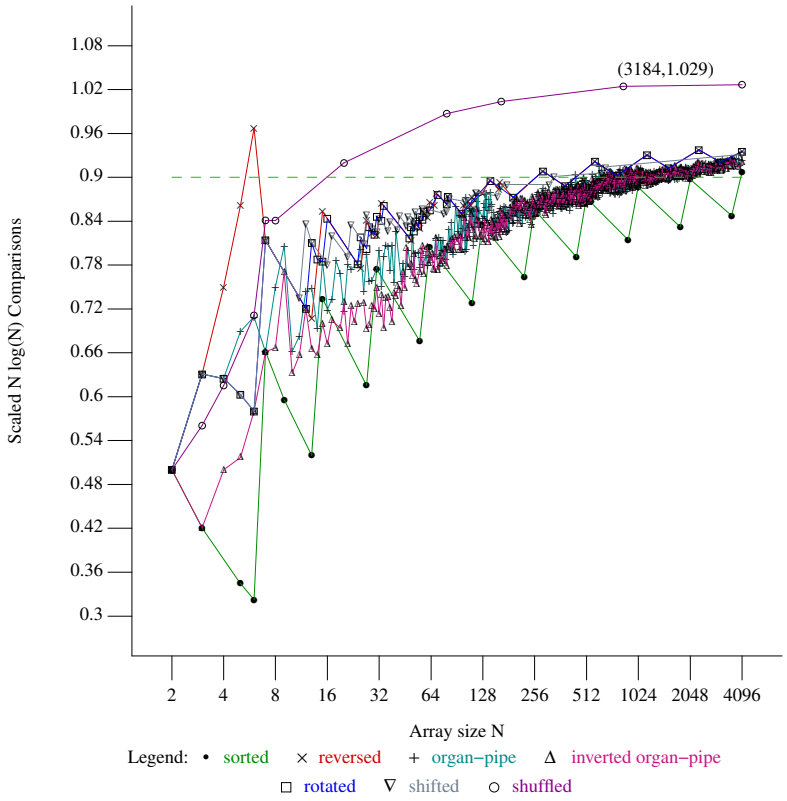

Figure 9: Comparisons for Bentley & McIlroy qsort modified for improved sampling quality

sequences. Otherwise, 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 4 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. The modifications applied to Bentley & McIlroy's qsort are also used in *quickselect*, and can be applied to most quicksort implementations.

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

Insight: Be wary of macros with side-effects.

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. Plan9 qsort implementation shows the same effect at size 9 as shown in Figure 7. 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.

12 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 rotations 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 in *quickselect* that case never arises, as discussed in a later section of this paper.

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

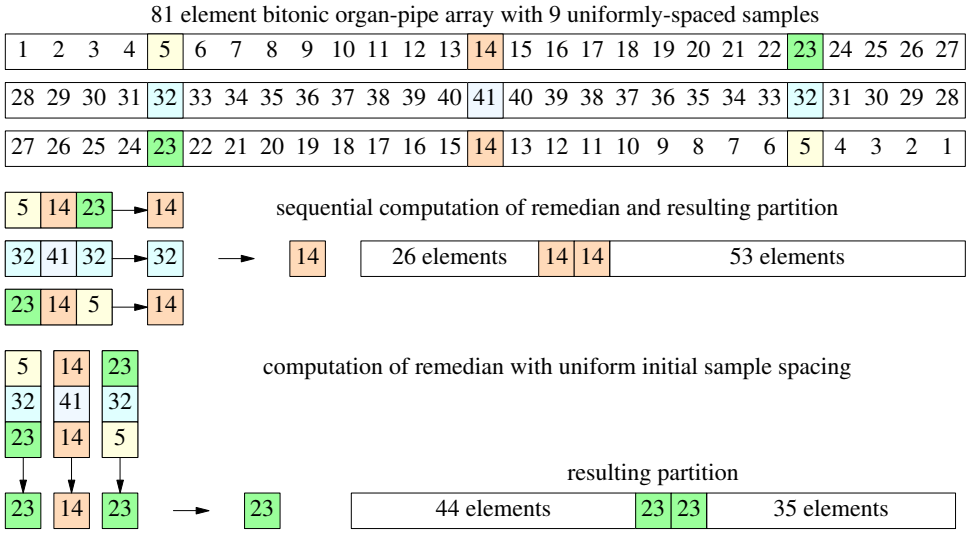

Figure 10: Two methods of computing the remedian

For more than 3 samples, it is still desirable to include samples at uniform spacing centered in the array[1].

The method for computing the remedian presented by Rousseeuw & Bassett [18] computes the remedian sequentially using samples taken in order, as does the method presented by Battiato et al. [21] 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 10 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 somewhat lopsided partition. Taking elements at uniform spacing (shown in rows in the diagram) selects a pivot which results in a more balanced partition. Using uniformly spaced samples at the lowest level as shown at the bottom of Figure 10 produces a partition which is about as lopsided as that method will produce; a rotation by 2 elements in either direction will yield the best possible partitioning split. This example further illustrates the importance of uniformly spaced sampling for non-random input sequences.

An adaptive sampling method was adopted for the present work:

- The number of samples is always a power of 3, and is roughly proportional to the square root of the number of array elements when sorting. Sorting of small sub-arrays is performed by dedicated sorting, obviating sampling and pivot selection. The maximum size for which dedicated sorting is used depends on specified options, such as partial order stability, and on the ratio of element size to the basic type size suitable for swapping. Order statistic selection uses 3 samples down to an array size of 3 elements.
- Dedicated selection functions (described later) are used for selection of 1 or 2 order statistics at the array ends, ensuring that no sampling for pivot selection is required for sub-arrays of fewer than 3 elements.
- Three elements with uniform spacing and centered in the array are used as samples for arrays through 79 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 9.

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 greater than those for which dedicated sorting is used, and 9 samples are used for arrays starting at 80 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 [18] [21]. 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

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

[1] Although Bentley & McIlroy [9] 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.

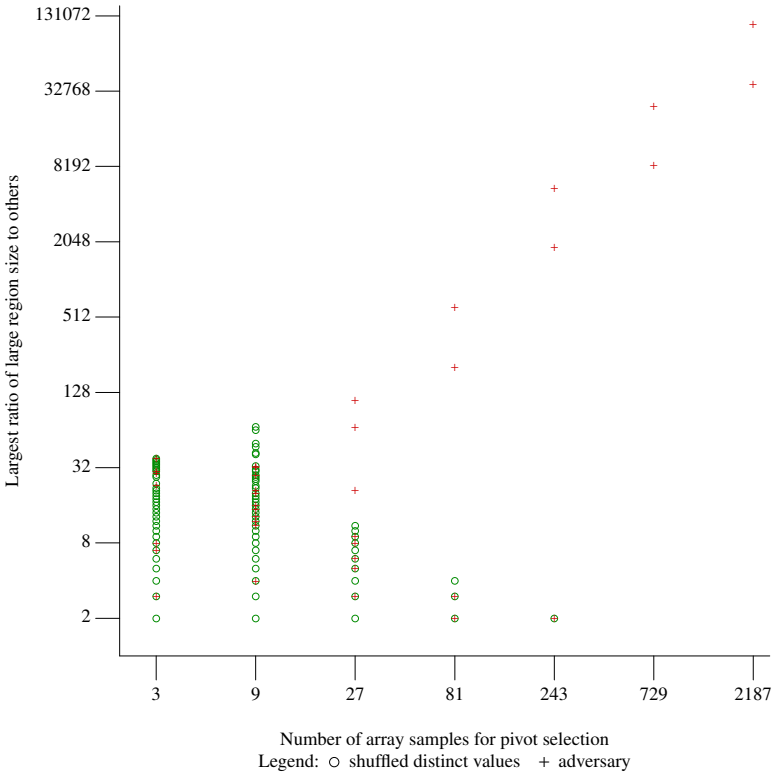

Figure 11: Partition lopsidedness vs. pivot sample quantity

9 through 14348907 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 2187 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 sorting. Arrays of the same sizes were sorted with adverse inputs generated by McIlroy's adversary [13]. and the corresponding ratios were recorded. Figure 11 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 79 elements (3 samples), the maximum possible ratio occurs when median-of-3 pivot selection results in a single element in the small region and 77 in the large region, giving a ratio of 38.5:1. 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 largest ratio observed 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 $\propto \sqrt{N}$). Adaptive detection of adverse inputs at large sub-array sizes is therefore possible, and is discussed in more detail in a later section.

13 Small sub-arrays

Most partitions resulting from pivot selection using remedian of samples with sample size $\propto \sqrt{N}$ are well balanced. Figure 12 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 4000 elements and 1000 arrays for each sequence type were generated and sorted. Although it is possible to have very large ratios, none greater than 0:1 were observed for randomly shuffled values in this experiment. There is a clear difference between the characteristics for the two sequence types; random inputs produce a large proportion of reasonably balanced partitions, with a rapid decline in number as ratios increase, whereas high ratios are produced nearly as often as small ones for adverse inputs[1].

Quicksort's overhead of sampling, pivot selection, partitioning, and recursion can be relatively expensive for small arrays. Large arrays are divided into smaller sub-arrays by the divide-and-conquer algorithm; these are further sub-divided into an exponentially increasing number of even smaller sub-arrays. Because of the extremely large number

[1]The number of partitions 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.

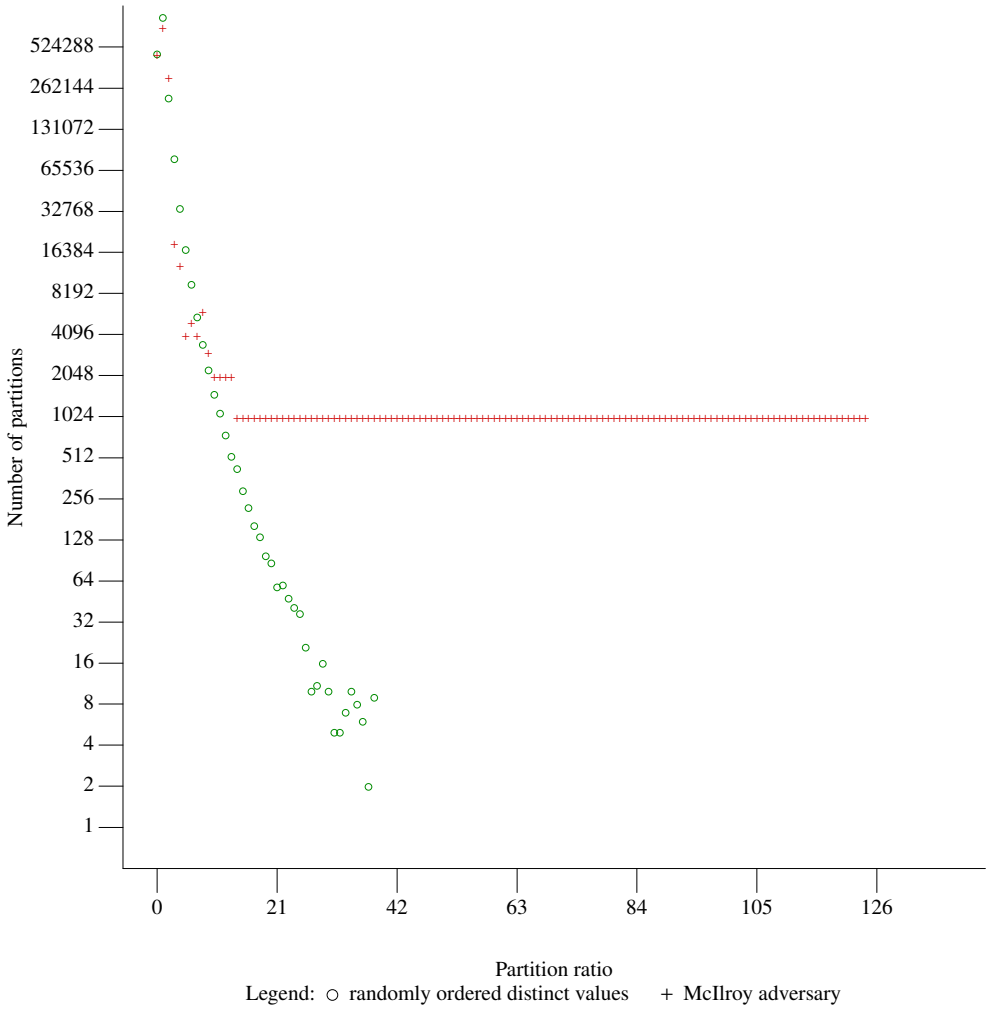

Figure 12: Partitions vs. lopsidedness

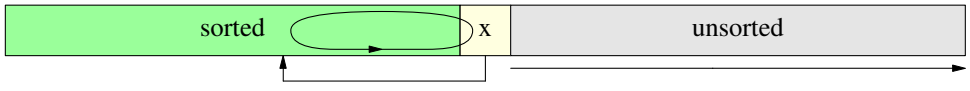

Figure 13: Insertion sort

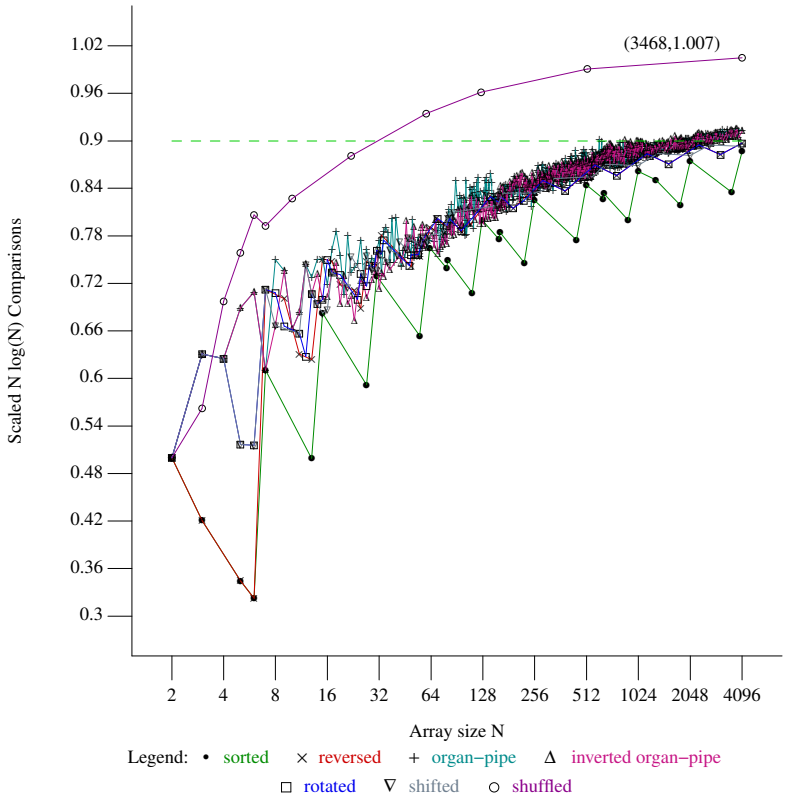

Figure 14: Comparisons for Bentley & McIlroy qsort modified with binary-search insertion sort

of very small sub-arrays produced, overall performance can be substantially improved by improvements to the performance of sorting very small sub-arrays. Conversely, poor performance for small sub-arrays adversely affects overall performance, as can be observed in the "echoes" of poor small-array performance due to poor sampling as seen in Figure 4.

Insight: Performance of small (base) cases affects overall performance for recursive algorithms including divide-and-conquer.

It is common for quicksort implementations to use insertion sort 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 $N(N-1)/2$ using linear search for insertion with reversed input.

Insertion sort implementation can be improved over the implementation used in Bentley & McIlroy's qsort. That implementation examines a pair of adjacent elements, and if they are out of order they are swapped and another comparison is made with the next element. That alternates execution between the comparison function and the swapping function, combining a linear search for the insertion point with a waltzing of the element into position. Searching for the insertion position using successive comparisons (without swapping) followed by swapping (with no further comparisons) to insert the element may make better use of the hardware instruction cache. Binary search uses fewer comparisons than a linear search for a sorted region of 4 or more elements, decreasing the worst-case comparison complexity from quadratic to linearithmic (worst-case swap complexity remains quadratic). Avoiding a test for sorted region size, using binary search in all cases, leads to simpler and faster code. Insertion of the element into position may be performed by a rotation of elements from the insertion position through the element to be inserted by one position as shown in Figure 13[1]. Rotation makes more efficient use of temporary storage and uses about half as many array element accesses (reads and writes) as one-by-one swapping. Carefully-implemented insertion sort is stable whether linear or binary search is used and whether waltzing swaps or rotations are used for insertion.

One characteristic of the insertion sort used in Bentely & McIlroy's qsort is hidden in its generally high comparisons count; its performance is asymmetric for similar inputs such as inverted vs. non-inverted organ-pipe sequences. The asymmetry can be seen at the left side of Figure 9, and can be removed by an initial pass over the elements to find the longest in-order run, followed by insertion of the remaining elements into that run of already-sorted elements. A side-effect of that initial pass is the ability to detect reverse-sorted (zero length in-order run) and already-sorted (all in-order) inputs. Reverse-sorted inputs can be rearranged into sorted order by pairwise swaps; already-sorted input obviously requires no additional action. Performance results are as shown in Figure 14; the number of comparisons required for reverse-sorted input is now the same as for already-sorted input and asymmetry has been eliminated. The improved performance for what would have been unpropitious inputs results in slightly improved performance for random inputs.

[1] A mirror-image version of Figure 13 is also possible.

Average comparisons: all distinct value permutations

size	quicksort	insertion	merge	network
2	1	1	1	1
3	2.667	2.667	2.667	2.667
4	5	5.417	5	5
5	9	8.692	7.5	9
6	11.863	12.272	10.576	12
7	15.397	16.595	14.034	16
8	19.101	20.817	17.28	19
9	22.784	25.097	20.326	25
10	27.077	29.957	24.589	29
11	31.343	34.97	28.83	35
12	35.698	40.022	33.063	39

Best- and worst-case comparisons

array	quicksort		insertion		merge		network
size	best	worst	best	worst	best	worst	
4	5	5	3	7	4	6	5
5	8	10	4	11	5	9	9
6	10	13	5	16	6	12	12
7	12	19	6	21	8	17	16
8	16	23	7	26	10	21	19
9	20	30	8	31	11	25	25
10	23	35	9	37	12	29	29
11	26	43	10	43	13	34	35
12	30	49	11	49	14	39	39

Table 1: Small-array sorting comparisons performance

It may be 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. Fourth, if sorting stability is required, the candidate algorithm should guarantee stability. A decrease in comparison or swap complexity compared to insertion sort is desirable.

Optimal sorting networks [24] [25] [26] [27] 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. Optimum sorting networks do not provide stable sorting, however sorting networks with additional comparisons (essentially equivalent to worst-case unrolled insertion sort, but amenable to some parallel operations) are possible.

The use of sorting networks as an alternative to insertion sort for small sub-arrays was evaluated for sorting networks through 12 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 (with binary search, and with rotation for insertion), optimum sorting networks (N.B. not stable above size 2), and a recursive in-place merge sort implementation (described below). The average number of comparisons at each sub-array size are shown in the upper part of Table 1. Each method exhibits the same number of comparisons for a sub-array of size 2; the sorting network has the lowest overhead. For 3 through 12 element sub-arrays, all alternatives use on average fewer comparisons than divide-and-conquer with median-of-3 pivot selection except for optimum sorting networks at sub-array sizes of 5 through 7 or more than 9 elements. For 3 elements, a software implementation of an optimization of the sorting network is possible; first perform a compare-exchange of the outer elements. Then if the middle element is smaller than (and is therefore exchanged with) the first element, it cannot be larger than the third element, saving one comparison in some cases. However, that does not guarantee sorting stability. More generally, 3 elements can be sorted using a decision tree similar to Figure 3, with different optimizations possible by variation of the order of comparisons. Such an optimization has been included in Table 1.

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 4. Insertion sort using binary search and an initial scan to counter asymmetry reduces the variation by reducing the worst-case number of comparisons. Sorting networks always use the same number of comparisons regardless of the input data. Stable sorting networks always use as many comparisons as worst-case linear-search insertion sort, which exceeds the average number of comparisons used by median-of-3 quicksort at a sub-array size of 6 or more elements. Best-case comparisons and worst-case comparisons for median-of-3 quicksort, (symmetric binary search) insertion sort, in-place merge sort, and optimum (non-stable)

Average swap-equivalents: all distinct value permutations

size	quicksort	insertion	merge	network
2	0.5	0.5	0.5	0.5
3	1.167	2	1.5	1.167
4	2.583	4.917	3	2.583
5	4.067	8.496	4.558	3.117
6	4.944	12.579	6.745	4.794
7	6.315	17.701	9.077	6.196
8	7.611	23.292	12.081	8.295
9	9.005	29.662	14.58	12.811
10	10.366	36.805	18.103	11.12
11	11.849	44.826	21.411	15.633
12	13.236	53.476	25.85	22.414

Best- and worst-case swap-equivalents

array	quicksort		insertion	merge	network
size	best	worst	worst	worst	worst
2	0	1	1	1	1
3	0	2	3.5	3	2
4	0	5	8.5	6	5
5	2	6	15	10.5	6
6	2	8	23	14.5	10
7	2	10	32.5	18	13
8	2	12	43.5	24.5	18
9	2	16	56	30.5	23
10	4	17	70	39.5	25
11	4	19	85.5	45.5	28
12	4	21	102.5	54	39

Table 2: Small-array sorting swap-equivalents performance

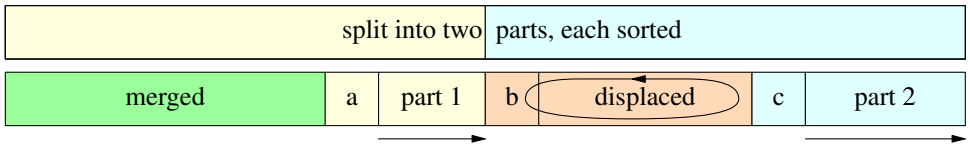

Figure 15: In-place merge sort

sorting networks are shown in the second part of Table 1 (best-case is zero comparisons except for quicksort).

Another alternative to insertion sort is an in-place variant of merge sort as shown in Figure 15. Given two adjacent sorted sub-arrays, merging begins by comparing the smallest elements of the two sub-arrays; if the smallest element of the second sub-array is smaller than the smallest element of the first sub-array, those elements are swapped, expanding the initially empty region of merged elements and decreasing the size of the second sub-array. The element displaced from the first sub-array now constitutes a region of such displaced elements. Once such a region exists, it is no longer necessary to make comparisons with elements from the first sub-array; because all elements in the displaced elements region were initially to the left of the leftmost element of the remaining unmerged elements in the first sub-array, continuing the merge will exchange the leftmost unmerged element with either the smallest displaced element or the smallest unmerged element from the second sub-array, whichever is smaller[1]. The elements in the displaced elements region are maintained in sorted order; automatically if the element from the second sub-array is merged, and by rotation of the displaced elements if the smallest of that set is merged[2]. When all elements from the first sub-array have been merged, the process begins anew, merging the (in-order) displaced elements and the remainder of the second sub-array. The merge is complete when either there is no displaced elements region at the end of merging the first part or all of the elements from the second sub-array have been merged.

The simple in-place merge sort described performs at least as well on average (considering the number of comparisons) as insertion sort; performance is the same at 2 and at 3 elements, with an advantage to merge sort at larger sizes. The worst-case performance of the simple in-place merge sort is considerably better than for insertion sort for comparisons. Because the split regions are recursively sorted, the in-place merge sort has higher overhead than insertion sort; although merge sort uses fewer comparisons (on average, as well as in the worst case) than insertion sort, its run time for simple comparisons can be longer than insertion sort's run time. This disadvantage can be overcome by unrolling the recursion; for example, a sub-array of 4 elements is split into 2 parts of 2 elements and each

[1]Preference given to displaced elements in the event of an equal comparison, if sorting stability is required.

[2]The rotation would be prohibitively costly for large arrays, however the size of the displaced elements region is on average no larger than the region for which rotation would be used to implement insertion sort.

part is sorted with a simple compare-exchange, avoiding recursion.

At each split into two sub-arrays to be sorted, merge sort can take advantage of parallelization, with serialization required for the subsequent merges. The ability to take advantage of parallel operations appears to be comparable to that for optimized sorting networks.

In-place merge sort as described above is stable provided that the recursive sorting operations applied to the split sections (which need not be merge sort) are stable.

Average data movement for the various sorting methods is shown in Table 2. Although the average number of comparisons for alternatives to divide-and-conquer are lower for some methods, most alternative methods involve a greater amount of data movement for most sizes of arrays, effectively trading a reduction in comparisons for an increase in data movement. Exceptions to that generalization are optimal sorting networks at sizes 5 through 7, which involve less average data movement. Coupled with the low overhead of sorting networks, those networks can be faster than divide-and conquer sorting at those sizes despite the higher number of comparisons; the advantage may extend to arrays of 8 elements (lower comparisons and overhead, but more data movement).

The effect of data movement on running time depends on the size of data elements because movement of a single data element which is larger than the basic type which is suitable for alignment and size of elements requires multiple moves in increments of the basic type. For large data elements, a reduction in data movement such as is characteristic of optimal sorting networks for 5 elements can lead to substantial run-time improvement compared to the other sorting methods. When array elements have the same size as basic data types, the amount of swapping and rotation is less important than comparisons to overall performance[1]. For structured data, including character strings, the importance of swapping and rotations increases because multiple data moves in increments of the basic type size used for data movement are required to move elements.

Another method of reducing the cost of data movement is to sort indirectly, that is to rearrange pointers to the data such that accessing the data via successive pointers accesses the data in the desired (sorted) order even though the data itself might not be moved. That involves random access of data (for random inputs) which may exhibit cache effects due to poor locality of access. If the sorted data will be accessed multiple times, for example if sorting is a prelude to multiple searches using binary search, it may be advantageous to rearrange the data per se in sorted order.

The performance improvement for indirect sorting of large data elements can be substantial; sorting involves $O(N \log N)$ moves, each of which requires r basic type moves for a ratio r of element size to basic type size. With indirect sorting, the pointers are moved, eliminating the factor of r (but adding $O(NlogN)$ pointer dereferences; the savings here depends on the relative costs of pointer dereference and data movement – pointer dereference is usually considerably less expensive than data movement). Rearranging the array elements with *rearrange_array* uses $O(N)$ data moves with the ratio r instead of $O(N \log N)$ such moves. For big-O constant k, direct sorting would use $krN \log N$ moves, whereas indirect sorting uses $kN \log N + rN$ moves, the savings being $N((r-1)k \log N - r)$ moves. For $k \approx 1$ and large r ($r - 1 \approx r$), the savings is approximately $r/2N \log N$ moves. A fixed limit on the number of pointers used for indirect merge sort establishes an upper limit on the sub-array size for which it can be used. Because the pointers are associated with the initial sub-array, merges (including ones recursively required) must be serialized to ensure sharing of the pointer array without interference.

Whereas Bentley & McIlroy's qsort uses a single dedicated sort (insertion sort with linear search) and a fixed cutoff, the final dedicated sort used for the modified qsort and for *quickselect* uses the best method for the combination of the number of array elements, the size of data elements compared to the size used for data movement, and the specified options (e.g. stable sorting, or minimization of the number of comparisons), with performance shown in Figure 16. For 2 elements, a simple comparison-exchange is always used. Three elements are sorted with a decision tree if partial order stability is not required, or by an unrolled merge sort if such stability is specified. At larger (but still small) sizes, the ratio of element size to basic type size is a consideration as well, as mentioned earlier with respect to optimal sorting networks. Large data elements for which either partial order stability or the minimization of the number of comparisons is desired might be most efficiently sorted via an indirect mergesort, with the array size limited by the available processor data cache size to avoid cache misses during pointer and data movement. Figure16 was generated with data elements of a basic type size; optimal sorting networks are used for 4 through 8 elements, and divide-and conquer is used for larger sizes. Reduced comparisons for optimal sorting networks compared to insertion sort contributes to the further reduction in the number of comparisons used for random inputs compared to Figure 14.

The 2 merge sort implementations (in-place and indirect) provide the minimum number of comparisons, which provides a performance advantage when comparisons are expensive. However, the higher overhead associated with the recursive calls results in an overall run-time increase unless the comparisons are truly expensive in terms of run-time. For example, arithmetic operations on data elements during comparisons are generally not sufficiently expensive to justify use of the merge sorts to minimize comparisons; operations such as database lookups in external storage may well be sufficiently expensive. In cases where minimization of comparisons is justified, the preformance improvement can be impressive, as shown in Figure 17. The number of comparisons for non-random input sequences does not continue to decline indefinitely; once data cache size limits are reached, divide-and-conquer is used, and the number of comparisons increases.

[1] Asymptotic non-stable sorting and selection performance of random inputs uses about 4 times as many comparisons as swaps, $\approx N \log_2 N$ comparisons and $\approx 0.25N \log_2 N$ swaps for sorting and $\approx 2N$ comparisons and $0.5N$ swaps for single order statistic selection.

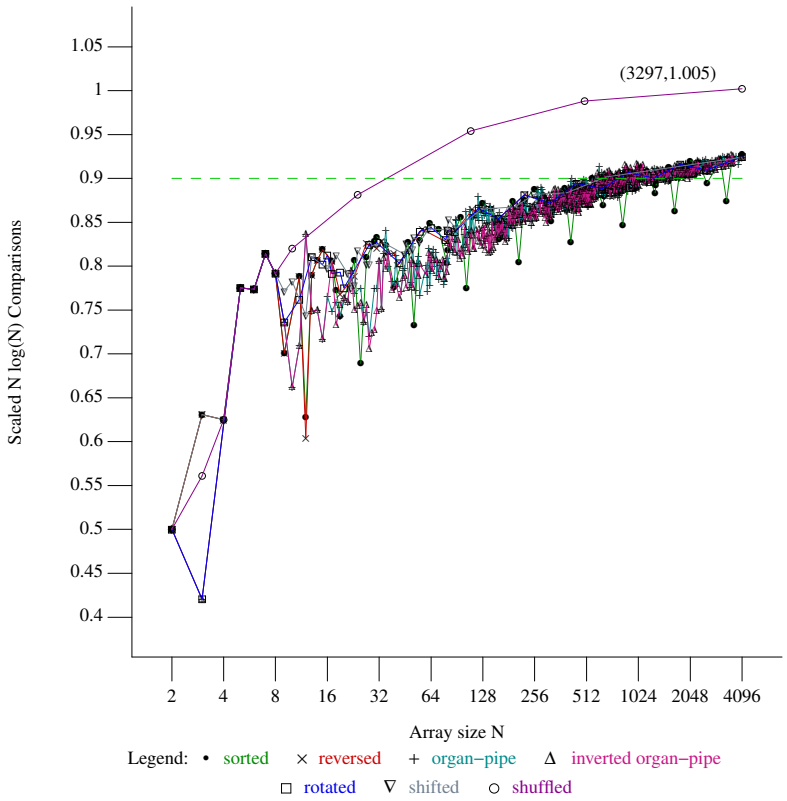

Figure 16: Comparisons for Bentley & McIlroy qsort modified with dedicated sort using the fastest method for simple comparisons

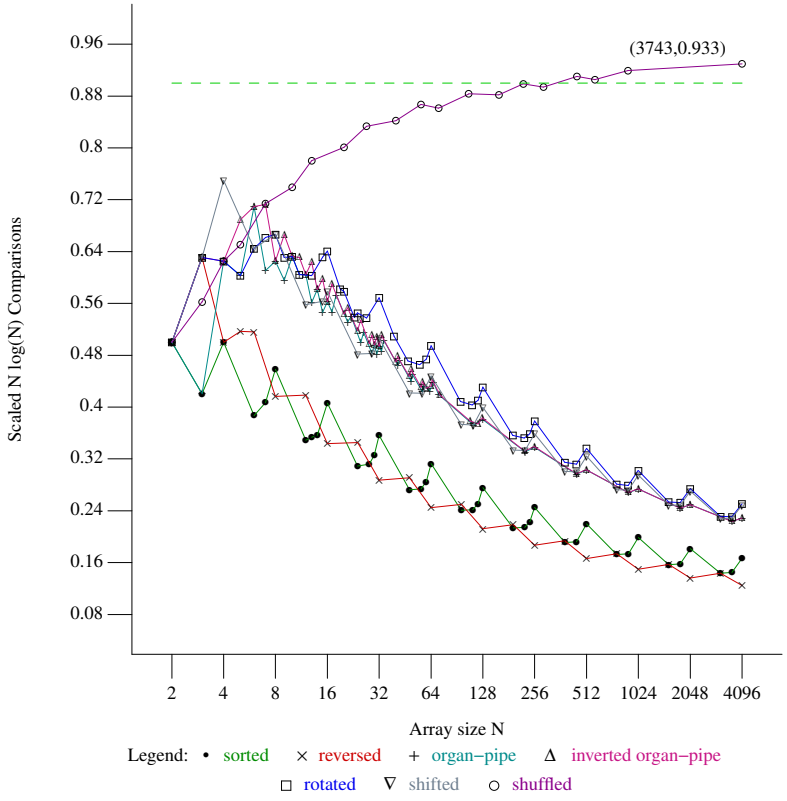

Figure 17: Comparisons for Bentley & McIlroy qsort modified with dedicated sort optimized for expensive comparisons

14 Order statistic 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 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 two categories of order statistic rank distribution could be used to manage reasonably efficient sampling for pivot selection, and two categories for determination of whether to sort or continue order statistic selection:

- If there is a desired order statistic rank near the middle of the sub-array, additional cost (compared to the case for sorting) in selecting a pivot element close to the median is justified; if the desired order statistic is selected as the pivot element, the problem size is directly reduced. If the pivot rank is adjacent to the desired order statistic, partitioning will leave the desired order statistic at one end of a partitioned region, and low-cost methods of selection can be used (as described in the next section of this paper).
- Conversely, if there are no desired order statistic ranks near the middle of the sub-array, there is no point in taking extra measures to obtain a pivot which partitions the sub-array into regions of nearly equal size; subsequent partitions will discard those areas near the middle of the original sub-array (which have no desired order statistic ranks).
- Desired order statistic ranks which are grouped together result in eliminating large regions at early partitioning stages, after which the remaining region(s) contain a high proportion of elements with desired ranks, possibly suitable for sorting. At large array sizes, multiple selection may be more efficient than sorting even if $\approx {}^{15}/_{16}$ of the array's ranks (grouped) are desired order statistics.
- Widely distributed order statistic ranks result in performance similar to sorting; most regions resulting from partitioning require further processing. At large array sizes, sorting is generally more efficient if the number of widely distributed order statistics is ${}^{1}/_{8}$ or more of the array size.

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 4 through 306 elements (arrays of 3 or fewer elements are always more efficiently processed by sorting, except when selecting the median of medians when repartitioning after an initial lopsided partition). Larger arrays use the ${}^{1}/_{8}$ and ${}^{15}/_{16}$ ratios for widely distributed and for grouped ranks mentioned above. When selection rather than sorting is used, separate tables of sampling breakpoints are used for pivot selection sample quantity based on the distribution of order statistic ranks as described above. In total, 3 such sampling tables are used; the table for sorting, and 2 for selection.

[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.

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. 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 11 and 12).

15 Few order statistics

Selection of a single order statistic by divide-and-conquer leads to a complexity of $\approx 2N$ comparisons. Selection of some specific order statistics can be accomplished with fewer comparisons.

Selection of minimum or maximum can be performed efficiently by a single linear pass over the array elements followed by swapping (or rotation, if stability is required) to the required position; $N-1$ comparisons and at most 1 swap or rotation. Selection of the two smallest or two largest elements can be performed by repeating selection of the minimum or maximum of the remaining elements once the overall minimum or maximum has been put in its position, for a total of $2N-2$ comparisons and at most 2 swaps or rotations. Even if the minimum is not required, use of two passes to find the second-smallest element uses fewer than the expected number of comparisons; likewise for the second-largest element. Selection of both minimum and maximum could similarly be performed in successive passes, with the same complexity.

Another method of finding both the minimum and maximum of an array of N elements is to compare the first and last elements, tentatively taking the smaller-valued element as the minimum and the larger as the maximum. The remaining $N-2$ elements are then compared to these in a single pass. If an element is found to be smaller than the current tentative minimum, it becomes the new tentative minimum; likewise for the maximum. The total number of comparisons is at most $1 + 2 \cdot (N-2) = 2N-3$, which is a slight improvement over the two-pass method. Divide-and-conquer can be used to provide a further reduction in the number of comparisons required to find both the minimum and maximum. If the array is split into two equal parts, finding the minima and maxima of the two parts requires at most a total of $2N-6$ comparisons, and an additional 2 comparisons suffices to determine the overall minimum (which must be the smaller of the two minima) and maximum, for a total of at most $2N-4$ comparisons, a further savings of one comparison **per split** compared to the single-pass method. Recursively splitting until the trivial cases of 1 or 2 elements are obtained gives an asymptotic cost of $\approx 1.5N$ comparisons and at most 2 swaps or rotations to find the 2 order statistics.

Given a sub-array of 2 or 3 elements for which 1 or 2 order statistics are desired, one of the above methods is applicable; therefore the sub-array is not processed by partitioning, and there is never a need to select a pivot element from such a small sub-array.

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*, as specified in ISO draft C language standards [28]. Unfortunately, the Open Group standards [29] specify different names, *nel* for the element count and *width* for the element size.

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. Several of the components work together synergistically. Some entail non-trivial cost in terms of object code or data space, which need to be juxtaposed against the benefits provided. The relevant components are:

- 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. Smallest regions are processed first to prevent overrunning the program stack when sorting. 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. When performing (multiple) selection, regions not containing desired order statistic ranks are ignored. In addition to

being one of the project goals, selection is used for median-of-medians pivot selection, which in turn is used to avoid poor adverse-input sorting behavior. The internal function also has provision for selecting performance optimized for speed with simple comparisons or for minimizing the number of comparisons (useful when comparisons are costly), and for specifying a requirement for partial order stability (which entails lower performance than non-stable sorting and/or selection) when the *quickselect* interface is used.

- 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, taking equality comparisons into account adds no cost, but provides substantial performance improvement for some common input sequences.

- Use of dedicated sorting for small sub-arrays avoids choosing between a single sample for pivot selection and use of median-of-3 pivot selection for very small arrays when sorting. Dedicated sorting using sorting networks and in-place mergesort or binary-search insertion sort for small sub-arrays reduces the average number of comparisons used for sorting. Because the savings for small sub-arrays accrues for each partitioning of larger arrays, the benefit extends to all array sizes. Use of improved dedicated sorting for small sub-arrays significantly contributes to improved performance. There is potential for further performance improvement because sorting networks and in-place mergesort can be parallelized, which is not true of insertion sort. The implementations of insertion sort and in-place mergesort provide partial order stability; if stability is required, non-optimum sorting networks can be used although they use more comparisons and have less opportunity for parallelization than the non-stable optimum networks. The number of comparisons used by optimal sorting networks is lower than insertion sort's worst-case; stable sorting networks use as many comparisons as worst-case insertion sort. While sorting networks permit parallel compare-exchange operations, the overhead involved in synchronization in a software implementation on a general purpose computer exceeds any benefit which might be obtained by parallel execution of the relatively simple compare-exchange operations. Because of the limited benefits at large cost in object code size, use of many sizes of sorting networks is probably unwarranted for general library code, though it might be useful in some embedded applications. On the other hand, small size sorting networks have lower overhead than insertion sort or mergesort, and use of sorting networks for sub-arrays of 4 and fewer elements obviates consideration of linear search for insertion position in insertion sort.

- 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, and results in improved performance for other commonly-occurring input sequences, such as bitonic sequences and reverse-sorted sequences. 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 structured input sequences.

- Tables for sampling breakpoints and for repivoting region ratio thresholds avoid expensive computation and accommodate variations in characteristics which are not readily computable. Use of tables of sampling breakpoints is a space vs. time tradeoff, increasing object file size in order to avoid nested if-then tests or repeated computation of breakpoints.

- A fast pivot selection method using remedian with base 3 over a sample of array elements; sample size is a power of 3, with an increasing number of samples used as sub-array size increases, leading to a progressive improvement in the balance of the resulting partitions and limiting worst-case imbalance. That improvement leads directly to near-optimum recursion depth and indirectly facilitates early detection and correction of adverse inputs. 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.

- Partitioning based on the efficient split-end method used by Bentley & McIlroy [9], as modified by Kiwiel's algorithm L [15] to avoid self-swapping, and with swapping of the pivot element to the farthest array end to reduce disorder with only a modest increase in code size. When partial order stability is required, that partitioning method cannot be used; a separate recursive divide-and-conquer method using element rotations and partition merges is used. The number of comparisons used for partitioning is unchanged, but the rotations to preserve partial order are more expensive than the stability-destroying swaps used by the split-end partitioning method. Rotations could be used with a variant of split-end partitioning (deferring pivot movement), but that would lead to quadratic data movement complexity. The divide-and-conquer stable partitioning method used in *quickselect* retains linear complexity for partitioning comparisons, but increases data movement complexity by a factor of $\log_2 N$.

Sampling table excerpts

	maximum nmemb		
Samples	sorting	middle	ends
3	79	24	108
9	551	114	SIZE_MAX
27	3567	772	
81	23958	3867	
243	179087	19329	
729	1611788	102563	

Table 3: Sampling table excerpts

- 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*.
- A break-glass mechanism to obtain an improved pivot in the event of extremely lopsided partitions, improving performance and preventing polynomial worst-case performance[1]. 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. The break-glass 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. 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).
- Because the median-of-medians pivot selection method rearranges elements and destroys partial order, an alternative, viz. remedian of elements[2] (rather than of a sample of elements), is used when partial order stability is required.
- Selection of medians for median-of-medians is facilitated by moving the medians-of-3 to a contiguous region at the start of the array, using bias in ternary median-of-3 to minimize data movement. *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 [19]. Saving about ⅓ of the comparisons required for repartitioning requires some additional code to rearrange and skip over already-partitioned elements. This further requires some transfer of information about the extents of the partial partitioning via added function call parameters. The need for information about the extent of the equal-to-pivot region for avoiding recomparison precludes the use of dedicated sorting for small sub-arrays when repartitioning with median-of-medians; divide-and-conquer is used down to sub-arrays of 3 elements. Because it provides a small but measurable benefit[3], use of the partial partitioning information to avoid recomparisons is used in the implementation described in this paper. An alternate means of avoiding recomparison was proposed by Martínez & Roura [20]: 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. Because medians of sets are rearranged prior to selection of the median-of-medians, this method precludes partial order stability. Pivot rank improvement while maintaining partial order stability is provided by selecting a pivot using remedian with base 3 over many elements (using a power of 3 and ignoring leftovers); the rank guarantee is theoretically not as good as median-of-medians, but in practice it is adequate to provide acceptable worst-case performance.

18 Performance tuning

Performance in sorting and selection is affected by table values used for sample quantity and repivoting decisions. Martínez & Roura [20] 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.

[1] $O(N^{1.5})$ for $\sqrt{N}$ samples is better than ($O(N^2)$, but is much worse than $O(N \log_2(N))$ (for sorting) and $O(N)$ (selection).

[2] The remedian uses a subset of elements, the number of which is the largest power of 3 no larger than the total number of elements in the sub-array.

[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.

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. Breakpoints were determined by sorting with different breakpoint values, choosing breakpoints where a clear reduction in sorting cost was observed. Progression of breakpoint values for very large array sizes was extrapolated from breakpoints for moderately large arrays (up to several million elements). Multiple selection uses different breakpoints based on the distribution of desired order statistics, as previously presented in this paper. While the optimum number of samples for sorting is roughly proportional to $\sqrt{N}$, the optimum number of samples for order statistic selection for rank distributions with a desired rank near the middle of the sub-array appears to be nearer to being proportional to $N^{0.678}$. With no desired order statistics near the middle of the sub-array, there appears to be no advantage to use of more than 9 samples for pivot selection. Note that the distribution of remaining desired order statistics within sub-arrays varies as multiple selection progresses, splitting the original array into smaller sub-arrays.

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 guarantees provided by median-of-3 and remedian pivot selection methods limit the need for repivoting. 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 $1.0074N \log_2 N$ comparisons and adverse inputs can still lead to poor performance, although the improved pivot rank guarantee provided by remedian of samples pivot selection with the increased number of samples limits 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 11 and 12, 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 "relaxed" set of parameters, arrays of up to 79 elements use 3 samples: a region with a ratio of 40:1 or larger (which cannot happen with the sampling breakpoints used) is immediately repivoted. On the second occurrence of a ratio of 39:1 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 2187 or more samples a ratio of 3:1 or more triggers an immediate repivot [1] – Figure 11 shows that that is exceedingly rare for random input sequences. The performance summary for the "relaxed" parameter set indicates no effect on the number of comparisons used for a 131072 element array of randomly shuffled distinct values. Against McIlroy's adversary, the absolute worst-case scaled comparison factor is $1.3867N \log_2 N$ comparisons at an array size of 343 elements, and the scaled comparison complexity factor at an array size of 16 Mi elements drops to $1.0987N \log_2 N$ comparisons, about 9% higher than for random input. 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 adverse input generated by McIlroy's adversary scaled to less than $1.5N \log_2 N$ comparisons[2] with negligible effect on random input sequences.

The "complex" table entry refers to sorting optimized for expensive comparisons, which is otherwise the same as the "relaxed" entry. A side-effect of the use of merge sort variants for modest array sizes in order to minimize the number of comparisons is that input sequences which would be adverse for divide-and-conquer sorting have no effect on merge sort (which does not use pivot selection or partitioning), with the result that such input sequences do not degrade performance.

Because selection performance is more sensitive to lopsided partitions, a very aggressive repivoting table (Table 5) is used for sub-arrays for which selection rather than sorting is used.

19 Performance results

Comparisons reported below were made with gcc version 7.3.1 using −Ofast optimization, running on a machine with an Intel® Core™ i7-5500U CPU @ 2.40GHz and with 6.0 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 efficiency at large array sizes, as shown in Figure 18. 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

[1] Recall that median-of-medians with sets of 3 elements asymptotically produces a ratio under 2:1. At 9 or more elements, the ratio is always less than 3:1.

[2] A lower maximum – in this case including against McIlroy's adversary – than that claimed for Bentley & McIlroy's qsort implementation (but actually performing as in Figures 4 & 23).

Aggressive repivot tuning parameters

Samples	factor1	factor2
3	40	9
9	11	10
27	57	6
81	16	15
243	7	6
729	4	3

Relaxed repivot tuning parameters

Samples	factor1	factor2
3	40	39
9	57	10
27	57	6
81	16	15
243	7	6
729	4	3

Transparent repivot tuning parameters

Samples	factor1	factor2
3	40	39
9	57	56
27	57	56
81	16	15
243	7	6
729	4	3

Complex repivot tuning parameters

Samples	factor1	factor2
3	40	39
9	57	10
27	57	6
81	16	15
243	7	6
729	4	3

Parameters	random shuffle @128Ki		adversary worst	@	@
	scaled	penalty	scaled	size	16Mi
aggressive	1.0079	0.049%	1.3865	383	1.0990
relaxed	1.0074	0.000%	1.3867	343	1.0987
transparent	1.0074	0.000%	3.2044	1854	1.0987
complex	0.9560	0.000%	1.1773	8877	1.0623

Table 4: Sorting repivot parameter tables and performance characteristics

Aggressive repivot tuning parameters

Samples	factor1	factor2
3	30	3
9	4	3
27	6	6
81	16	15
243	7	6
729	4	3

Table 5: Selection repivot parameter table

Processor	RAM	Operating System	Compiler
AMD Athlon LE-1620	4 GB	Linux 3.12.67	clang 3.3
Intel Celeron SL54Q	512 MB	NetBSD 7.1.2	gcc 7.2.0
Intel Core 2 E6550	2 GB	OpenIndiana	Sun cc 0x5100
Intel Core 2 Q6600	4 GB	NetBSD 7.1.2	gcc 7.2.0
Intel i7-5500U	6 GB	Linux 4.16.0	gcc 7.3.1
Intel Pentium III	512 MB	NetBSD 7.1.2	gcc 7.2.0
Intel Xeon 3520	24 GB	NetBSD 7.1	gcc 7.2.0

Table 6: Test hardware and software environment

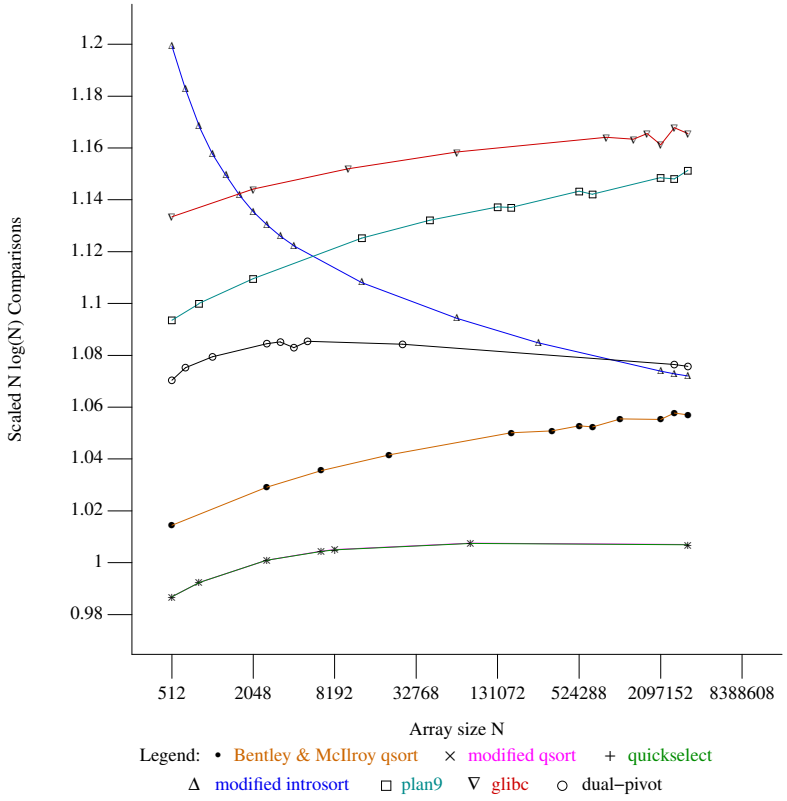

Figure 18: Comparisons for sorting implementations, shuffled input

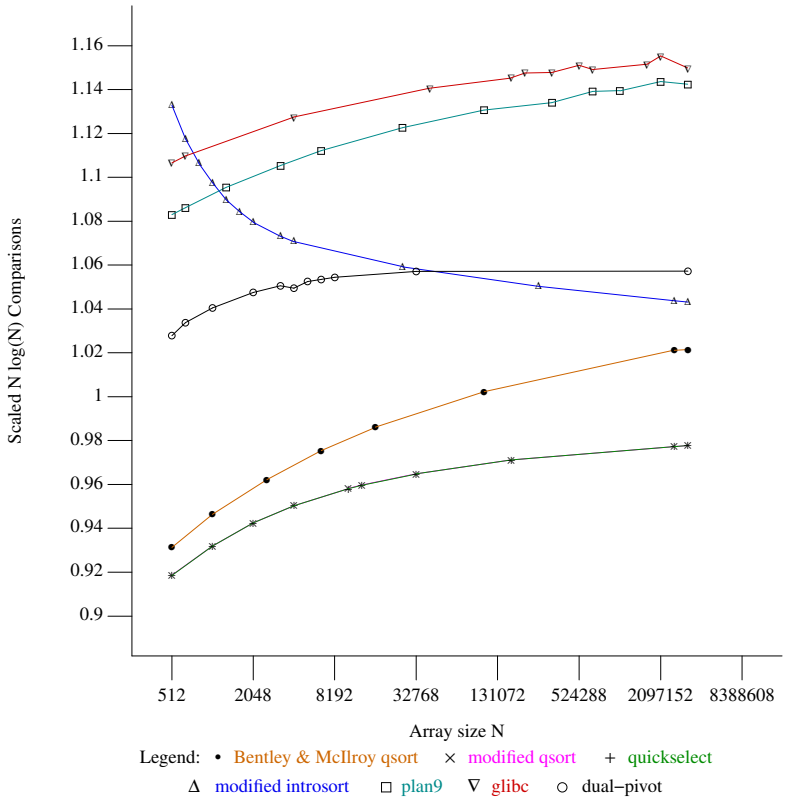

Figure 19: Comparisons for sorting implementations with limited-range random input

matches of multivariate data. The comparison complexity data for *quickselect* with randomly shuffled distinct inputs shown in Figure 18 has a least squares fit to $0.984525N \log_2 N + 0.81036N - 7.10974\ {}^{N}/_{\log_2 N}$. Figure 19 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 (uniform) sampling, Kiwiel's Algorithm L modifications to prevent self-swapping during partitioning, dedicated sorting methods for small sub-arrays, uses remedian of samples with more than 9 samples at large array sizes, 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* by lack of provision for order statistic selection and by lack of protection against quadratic behavior. The performance of the modified qsort and *quickselect* are nearly indistinguishable for sorting random inputs (look very closely at the lowest two nearly overlapping curves in Figures 18 and 19).

In tests, dual-pivot quicksort [17] performed poorly in terms of number of comparisons, number of swaps, and run time compared to *quickselect* and other sorting implementations tested. 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 [30] 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 Figures 18 and 19) are insufficient to achieve competitive performance even for trivial data types. Whereas *quickselect* approaches the theoretical optimum asymptotic sorting performance for quicksort of $1N \log_2 N$ comparisons and $0.25N \log_2 N$ swaps [20], dual-pivot quicksort with sampling and swapping improvements asymptotically uses about ${}^{5}/_{3}N \log_3 N \approx 1.052N \log_2 N$ comparisons (greater than 5% more) and several times as many swaps. The modifications to Bentley & McIlroy's qsort described above result in run time about 5% faster than Yaroslavskiy's dual-pivot sort [17] with a similar polymorphic qsort-like interface, measured sorting 100000 randomly shuffled distinct plain integers over 10000 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 Yaroslavskiy's dual-pivot sort: differences in each case amounted to more than 20% run time. Analysis by Wild et al. [31] 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.
- The number of swaps required for partitioning become more numerous as well as increasingly complex as the number of pivots increases.
- 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 *quickselect* (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 is greater for introsort than for *quickselect*, introsort performance for data types with non-trivial comparisons is lower than for *quickselect*. The cause of the higher number of comparisons is the final insertion sort over the entire array[1]; if that is changed to use insertion sort (or some other dedicated sort) only at the end of the introsort loop on sub-arrays, the number of comparisons for introsort with all of the above modifications becomes slightly lower than for *quickselect* (because of overhead and 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 Figure 18 in order to illustrate the magnitude of the effect. With smaller sorts at the end of the introsort loop, the number of comparisons would be essentially the same as for *quickselect*. Introsort avoids quadratic behavior, but shows worse performance than *quickselect* for adverse input sequences (see Figure 23 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 18 is a graph of comparison counts for sorting randomly shuffled distinct input values vs. array size. The modified version of Bentley & McIlroy's qsort uses fewer comparisons but somewhat more swaps than the unmodified version[2]. 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 than *quickselect*. The leveling off in scaled comparisons at large array sizes for the modified introsort, the modified qsort, the dual-pivot implementation, and for *quickselect* results from the increased number of samples used for pivot selection for large arrays.

Figure 18 shows several effects:

- The differences between the unmodified and modified Bentley & McIlroy qsort curves result from improved dedicated sort 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 from increasing sample size as the array size increases, using remedian of samples for pivot selection when many samples are taken.
- Plan9 qsort has relatively high comparison counts due to lack of dedicated sorting for very small sub-arrays, pivot selection sampling limited to 3 elements, and sampling nonuniformity due to integer arithmetic truncation.
- The gap between introsort and *quickselect* results from the costly final insertion sort used by introsort.
- An overall insertion sort is also used by glibc qsort, which also shows high comparison counts. Non-uniform (first, middle, last) sampling limited to 3 samples also contributes to the high comparison count.

[1] Because this insertion sort operates over the entire array, only linear search is used to locate the insertion point; binary search would be more costly on average.

[2] Note that Bentley & McIlroy's qsort will use many more swaps for data types with size and alignment different from *long*.

Sequence	Bentley & McIlroy qsort		quickselect	
	short	long	basic types	complex
sorted	0.91156	0.91156	0.92826	0.70579
reversed	1.39526	0.91160	0.92480	0.71845
shuffled	1.03233	1.03226	1.00286	0.92993
bitonic	1.00812	1.01689	0.92448	0.72943
rotated	1.39528	1.35970	0.92480	0.72839
shifted	1.39563	1.39563	0.92409	0.74514
binary	0.12548	0.12548	0.12660	0.59563
constant	0.08358	0.08358	0.08413	0.16638
reciprocal	1.01758	1.03257	0.99083	0.93245
normal	1.00400	1.03243	0.97945	0.92995

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

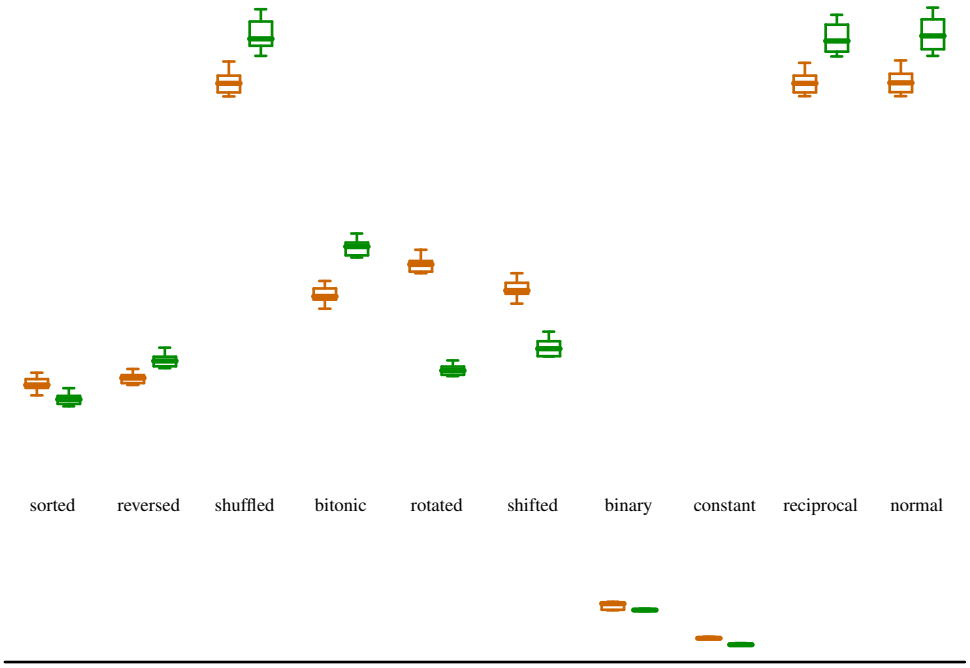

Figure 20: Run time for Bentley & McIlroy qsort and *quickselect* sorting long input sequences

- Dual-pivot performance is worse than unmodified Bentley & McIlroy qsort for real-world inputs; this 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 67% more comparisons and many more swaps than the efficient Bentley & McIlroy partitioning method. Cache effects lessen the run-time impact of the gap, but cannot overcome the disadvantages compared to *quickselect*.

As performance of dual-pivot, glibc, and plan9 sorting are not as good as Bentley&McIlroy's qsort implementation, remaining performance comparisons will focus on the latter and introsort vs. *quickselect*. Bentley & McIlroy's qsort and *quickselect* were tested sorting 8192 element arrays of plain and long integers and with string and structured data, with various initial sequences. String data were variable-length character strings in a subset of CCSDS 301.0-B-4 [32] ASCII time code format in the range 1970-01-01T00:00:00 through 7815-07-17T19:45:37.09551614 (on 64-bit architectures, 2004-01-10T13:37:03.5 on 32-bit architectures) corresponding to nonnegative long integers (in half-second steps on 32-bit architectures, 20 ns steps on 64-bit architectures). Structured data consisted of corresponding calendar segmented date and time fields. Each sequence was generated and sorted in 10000 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 0-8191
- organ-pipe sequence .., 4094, 4095, 4095, 4094, ..

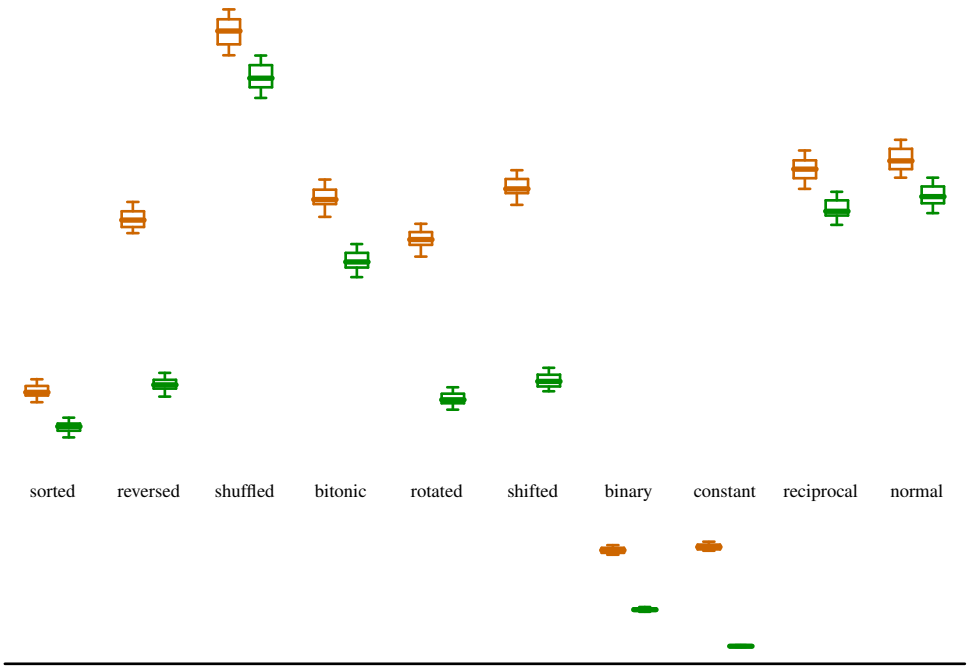

Figure 21: Run time for Bentley & McIlroy qsort and *quickselect* sorting structured data

- rotated sequence $1, 2, .., 8191, 0$
- random binary-valued elements (i.e. ones and zeros)
- all-equal element values
- normally distributed random values
- random values with a reciprocal distribution

Comparison counts were averaged over the runs and scaled to $N \log_2 N$ and are tabulated in Table 7. With two exceptions, *quickselect* shows lower comparison counts than Bentley & McIlroy's qsort. Two columns are listed for Bentley & McIlroy's qsort because of the special treatment it applies to data with element size and alignment the same as *long* integers; the principal effect of which can be seen for reversed and rotated input sequences.

Running time statistics are plotted as pairs of box-and-whisker plots in Figures 20 through 22. For each pair, 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 10 and 90 percentile values, which are marked with a hhorizontal tick. 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. Figure 22 displays runtime for the two sorting implementations across data types for a normally distributed random input sequence. Bentley & McIlroy's qsort is optimized for long integers; its performance is lower for data with types of different size.

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

Quickselect is faster at most sizes and data types for decreasing input sequences and always for rotated sequences (sampling unaffected by pivot movement; see Figures 4 & 9). Comparison count is lower for *quickselect*.

Quickselect is usually faster, sometimes slightly slower on random inputs 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 often 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

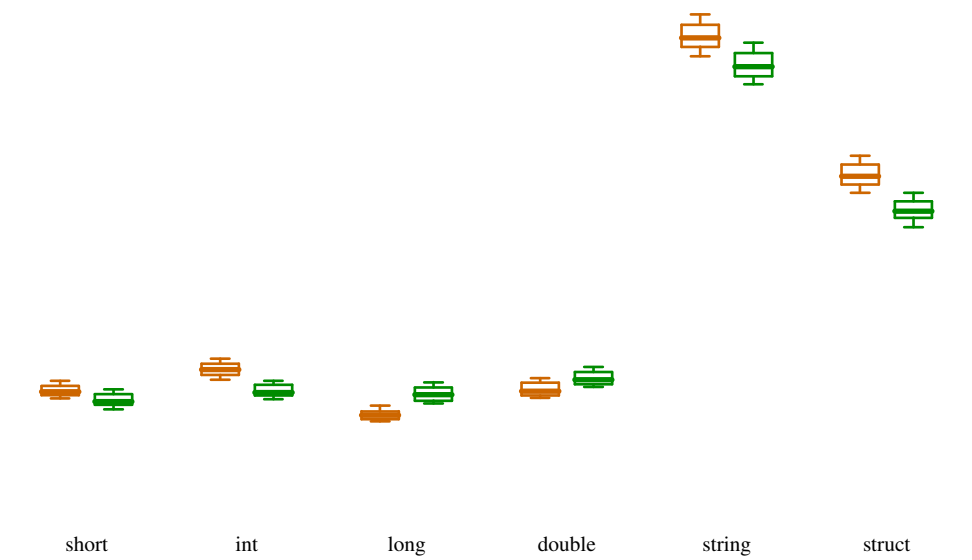

Figure 22: Run time for Bentley & McIlroy qsort and *quickselect* sorting normally distributed data

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 in *quickselect* (see Figures 4 & 9).

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 optimizes performance for data types with alignment and size equal to long integers; compare Figures 20 vs. 21 and see Figure 22.

Performance improves relative to Bentley & McIlroy qsort with larger array sizes because of slower growth in scaled comparisons (improved pivot rank through use of remedian-of-samples), as shown in Figures 18 and 19. The data for Table 7 and Figures 20 through 21 were collected using arrays of 8192 elements. This size represents a moderate difference between performance of *quickselect* and Bentley & McIlroy's qsort. Performance favors *quickselect* at larger array sizes (and at smaller sizes for distinct inputs) (see Figures 18 and 19).

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). Figure 23 shows performance of original and modified Bentley & McIlroy qsort, *quickselect*, glibc qsort, and a modified introsort vs. McIlroy's adversary. Implementations, such as the one in glibc, which use only a small number of samples for pivot selection quickly exhibit quadratic performance. Bentley & McIlroy's qsort starts to do so but recovers somewhat when the number of samples increases to 9 (at 41 elements), but soon returns to quadratic behavior. The higher cutoff values for median-of-3 and ninther pivot selection coupled with improved dedicated sorting in the modified version result in higher comparison counts for highly adverse input sequences at small array sizes, but lower comparison counts for very small arrays. Increased sample size as array size increases limits worst-case complexity of the modified qsort to $O(N^{1.5})$. Whereas performance of the modified qsort was nearly indistinguishable from performance of *quickselect* for input patterns and for random input sequences, the two perform quite differently with adverse input. Introsort's recursion depth limit and use of heapsort when that limit is reached results in large-array comparison complexity which is heapsort's plus 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 with adverse input. 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 (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 because of the added complexity of the "stopper" algorithm) 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, e.g. with the "relaxed" repivoting parameters shown in Table 4. It should be noted that McIlroy's

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

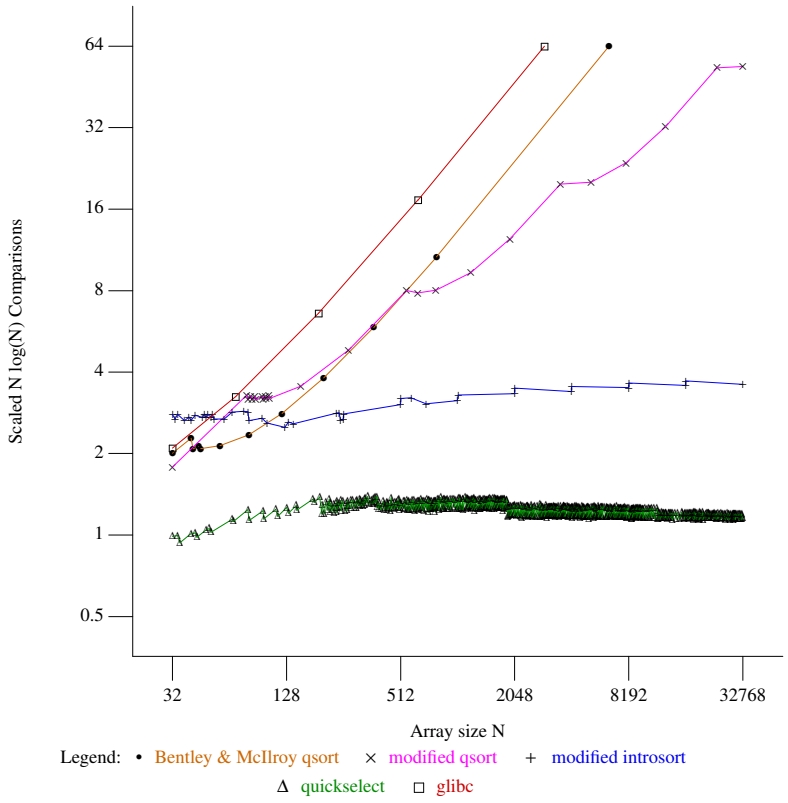

Figure 23: Adverse input comparisons for several sorting implementations

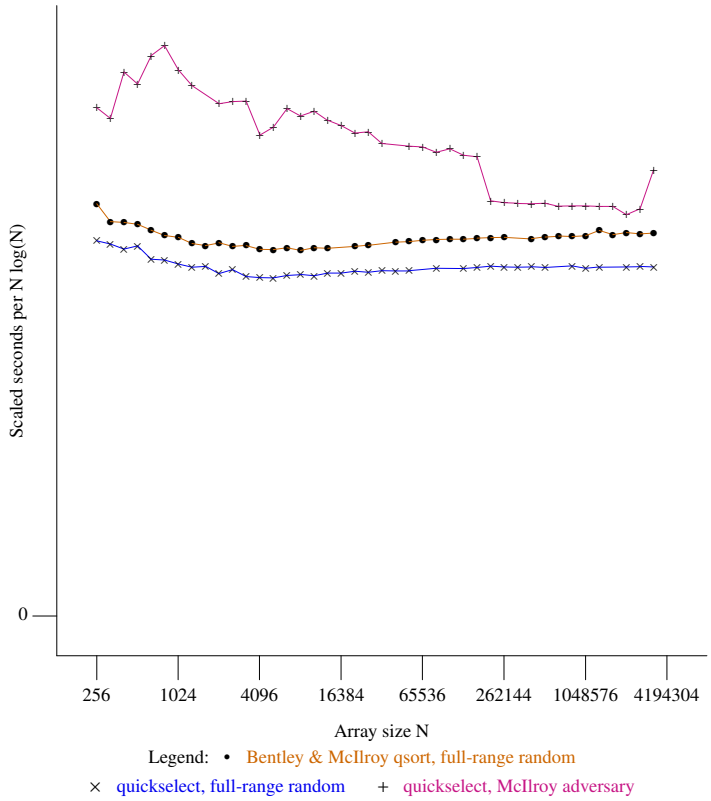

Figure 24: Scaled run time for Bentley & McIlroy qsort and *quickselect* with full-range random structured data, and *quickselect* vs. McIlroy's adversary

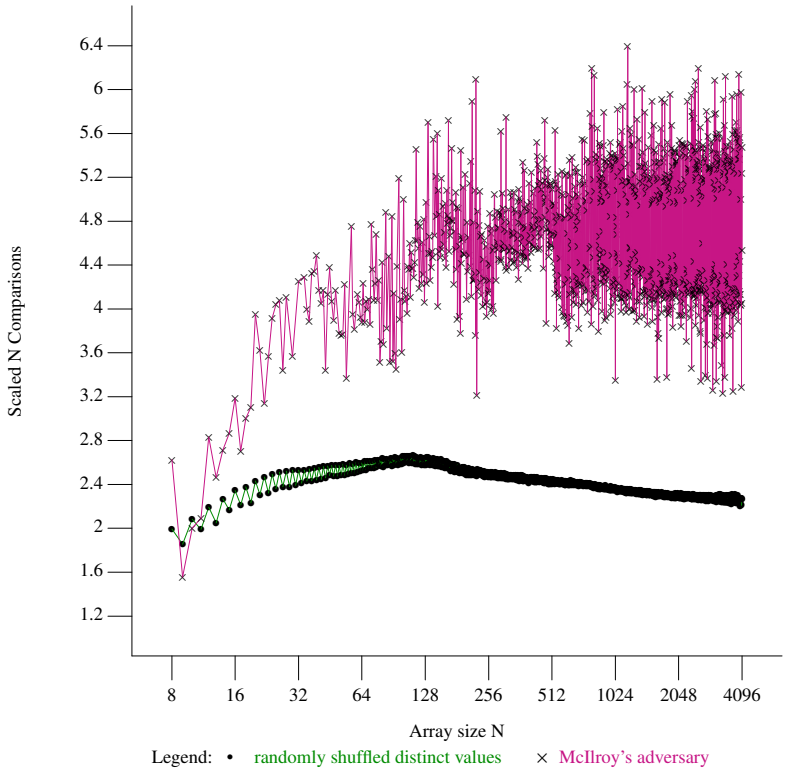

Figure 25: Scaled comparisons for *quickselect* median(s) selection

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 23, where the scaled number of comparisons drops after each increase in the number of samples used for pivot selection. McIlroy [13] has also noted that the adversary performs poorly (i.e. does not produce the most adverse result) against dedicated sorting methods such as insertion sort.

Early detection of adverse input and use of median-of-medians to correct lopsided partitions is quite effective. Figure 24 shows run time sorting performance (wall clock time) of *quickselect* with full-range random input and with adverse input generated by McIlroy's adversary as well as Bentley & McIlroy's qsort with full-range random input. The slight initial downward slope of the run-time scaled to $N \log_2 N$ vs. log array size for sorting of random inputs is due to amortization of overhead. The slight increase in scaled run-time at larger array sizes is due to cache effects.

Of course *quickselect* can be used for efficient selection (in linear time for a small number of order statistics), 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[1] 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. Against the modified McIlroy's adversary, there is an absolute worst-case of $6.3925N$ for an array of 1172 elements. Figure 25 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 $1.42539N + 12.7787\ {}^{N}/_{\log_2 N} - 34.3374\ {}^{N}/_{(\log_2 N)^2}$. Selection of a modest number of contiguous order statistic ranks has asymptotic comparison complexity $\approx 2N$. Selection of a modest number of order statistic ranks split into two roughly equal size groups has asymptotic comparison complexity $\approx 3N$. Complexity of selection of a small proportion of P widely distributed order statistic ranks in an array of N elements was found to be $\approx (2 + \log_2 P)N$ comparisons. That precludes an efficient deterministic multi-pivot sorting implementation. Single-pivot sorting can achieve $\approx 1.0N \log_2 N$ or equivalently $\approx 1.443N \ln N$ complexity for random input sequences and only slightly higher for adverse inputs; *quickselect* constitutes an existence proof. That is half the cost of fully deterministic single-pivot sorting (using the exact median, obtained with $2N$ comparisons, for partitioning (selection partitions the array around the selected order statistic, in this case the median)). The savings is due to the existence of efficient pseudomedians such as the remedian, which obtain a pivot with high probability of being close to the median at small cost. Literature comparing multi-pivot schemes to quicksort generally presume pseudorandom pivot selection for single-pivot quicksort, which is rarely used in practice due to poor performance, and more importantly high software maintenance cost (replication of a bug would

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

be well nigh impossible without detailed information regarding the implementation and state of pseudorandom number generation). That is why the literature frequently claims $2N \ln N \approx 1.386N \log_2 N$ for single-pivot quicksort cost when in practice no viable single-pivot quicksort implementation is even close to such a high asymptotic cost for random input (see Figures 18 and 19). Due to a lack of suitably efficient means of estimating tertiles, etc., random pivot selection is usually used for multi-pivot schemes, with the implicit maintenance problems that that entails. Reported complexity of $1.9N \ln N \approx 1.317N \log_2 N$ and $1.8N \ln N \approx 1.248N \log_2 N$ for multi-pivot schemes are unimpressive (refer again to Figures 18 and 19; note that $1.248N \log_2 N$ is off the top of the graphs) compared to performance of practical, maintainable single-pivot implementations (approaching $1.443N \ln N \approx 1.0N \log_2 N$ complexity). Even a highly-optimized dual-pivot implementation (incorporating improved dedicated sort and optimized sample size for pivot selection) cannot compete with the similarly optimized single-pivot implementations (modified qsort and *quickselect*; refer once again to Figures 18 and 19). A dual-pivot implementation, even with zero-cost pivot selection which magically provides pivots which are the exact tertiles, necessarily uses more than 5% more comparisons ($5/3N \log_3 N \approx 1.052N \log_2 N$) and many more swaps than single-pivot quicksort.

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, and sorting of pieces split in the first step of in-place mergesort may proceed in parallel.

The implementation described in this paper has not taken advantage of these opportunities, primarily due to the excessive overhead involved in parallel computing in a general computing environment.

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 and in-place mergesort 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 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 in cost – typically a small fraction of 1% – for sorting non-adverse input sequences[1].

The contributions of this work are:

- Explanation of the effect of poor sampling quality 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 quality. 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.

[1] Note that the "relaxed" repivoting table provides worst-case adverse-input performance better than introsort's best-case adverse-input performance.

- Improvements to insertion sort (separation of search and insertion, binary search for insertion position, insertion by rotation, elimination of asymmetry) for small array sorting. Application of sorting networks and in-place merge sort in addition to insertion sort for sorting small sub-arrays, which reduces the comparison and swapping costs on average and for many common input sequences.
- Combination of sorting and multiple order statistic selection in a single implementation.
- In-place stable partition by divide-and-conquer combined with merging of adjacent partitions by rotation of elements.

23 Related work

Chen & Dumitrescu [33] 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 [34] 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 indispensable 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 [18] were essential for developing the median-of-medians and remedian of samples pivot selection implementations. Analysis of optimal sampling by McGeoch & Tygar [19] and by Martínez & Roura [20] 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.

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.

[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://web.archive.org/web/20170817135549/http://programmingwala.blogspot.in/2013_07_22_archive.html`. [17 August 2017].

[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] Jon Louis Bentley. *Progranning Pearls, 2nd Edition*. Addison Wesley, New York, NY, USA, 2000. Errata at `https://wwwb-front3.us.archive.org/web/20010606181020/http://www.programmingpearls.com:80/errata.html`.

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

[18] 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.

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

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

[21] 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.

[22] URL `https://wwwb-front3.us.archive.org/web/20160601175829/plan9.bell-labs.com/sources/plan9/sys/src/libc/port/qsort.c`. [9 April 2017].

[23] URL `http://sourceware.org/git/?p=glibc.git;a=blob_plain;f=stdlib/qsort.c;hb=refs/heads/release/2.25/master`. [11 August 2017].

[24] 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.

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

[26] 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.

[27] Daniel Bundala and Jakub Zavodny. Optimal sorting networks. *CoRR*, abs/1310.6271, 2013.

[28] ISO/IEC JTC1/SC22/WG14 - C. ISO/IEC9899:201x. URL `http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf`, April, 2011. [23 September 2017].

[29] The Open Group. Single UNIX®specification, version 4, 2016 edition. URL `https://www2.opengroup.org/ogsys/publications/viewDocument.html?publicationid=12310%26documentid=11130`, 2016. [23 September 2017].

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

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

[32] The Consultative Committee for Space Data Systems. Time code formats. blue book. issue 4. ccsds 301.0-b-4. URL `https://public.ccsds.org/Pubs/301x0b4e1.pdf/`, November 2010. [27 April 2013].

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

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