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. Although quicksort can sort data in-place, other sorting implementations may require substantial additional memory. During implementation of a polymorphic in-place selection function based on quicksort-like divide-and-conquer techniques, some remaining performance issues in a well-known and widely-used implementation of quicksort were analyzed and addressed. The resulting in-place polymorphic selection function can be used for sorting as well as selection, and has performance for sorting comparable to or better than other quicksort implementations; in particular, it cannot be driven to quadratic sorting or selection behavior. Analysis techniques and algorithm design provided insights which yielded improvements applicable to other well-known sorting implementations and which may be applicable to other software projects.

Sorting and selection without tears

Bruce Lilly
bruce.lilly@gmail.com

March 7, 2017

1 Introduction

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

Insight: Simplify maintenance by appropriately combining functionality.

As the basic operations for multiple quickselect are the same as for quicksort, several well-known implementations of quicksort were examined for applicability to this implementation of multiple quickselect. One such quicksort implementation is described by Bentley & McIlroy. [9] Testing and analysis revealed a few shortcomings of that implementation. The tests and analyses, the shortcomings they uncovered, and improvements to address those shortcomings are described in later sections of this paper. Several of the improvements are applicable to 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 (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. Sorting processes both regions; selection processes only regions which might contain the desired order statistics.

2 Description of tests and test framework

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

- inverse organ-pipe (... 2 1 0 1 2 ...)
- median-of-3-killer [10]
- dual-pivot-killer
- all permutations of N distinct values (for very small N)

- all combinations of N zeros and ones (for small N)
- all-equal-elements
- many-equal elements (several variants; equal elements on the left, in the middle, on the right, shuffled)
- random sequences (full-range integers, integers limited to the range [0,N), integers restricted to the range $[0,\sqrt{N})$, randomly shuffled distinct integers)
- McIlroy's antiqsort adversary [11]
- prepared sequences contained in 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. [11] 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 (allegedly) sorted array. If a pair of adjacent elements was found to be out-of-order, the sorting function had failed to sort correctly. 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 Crashing, quadratic sorting, and selection

Bentley & McIlroy's qsort can be driven to quadratic performance via McIlroy's adversary; McIlroy [11] has published results. That qsort implementation as originally written uses two recursive calls, with no attempt to order the calls based on region size, as stated in Bentley & McIlroy's paper. [9] It is quite easy to get that implementation to overflow its program stack. At least one modification [12] of Bentley & McIlroy's qsort eliminates the tail recursion, resulting in the ability to handle larger worst-case inputs before overrunning the program stack.

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

Instead of recursive calls which lose rank information, an internal stack of regions to be processed can be maintained, and ranks can be determined from the invariant base pointer initially supplied. If the stack of regions is maintained in sorted order, with small regions processed first, only a small stack is required. An implementation for sorting *size_t nmemb* elements on a machine with 64-bit *size_t* types requires at most 64 region entries. An internal stack which facilitates selection (while maintaining a qsort-compatible interface) has the beneficial side-effect of preventing crashes caused by program stack overflow.

An alternative to an internal stack is 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. 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. 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. As with an internal stack, processing small sub-arrays first limits program stack size requirements.

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

Bentley & McIlroy's [9] qsort implementation has quite good performance for most inputs, however the partitioning is sometimes surprisingly lopsided. The pivot selection methods used provide only a very limited guarantee of problem size reduction. When given an input sequence congruent to that generated by McIlroy's adversary, the result is quadratic performance as the problem size is reduced only by $O(1)$ at each stage.

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 [11] stated:

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

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

- break-glass is used when a partition is exceptionally poor; introsort waits until recur depth grows, by which time the expected average-case run time has already been exceeded by some factor
- break-glass switches pivot selection mechanisms, but continues to use cache-friendly quicksort or quickselect; 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.

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.

4 Pivot selection with guaranteed rank limits

Blum et al. [2] describe a median-of-medians method which can limit the rank to the range 30% to 70% asymptotically using medians of sets of 5 elements and a recursive call to find the median of medians. Median-of-5 requires a minimum of 6 comparisons for distinct-valued elements; code to achieve that minimum is somewhat complex, and involves element swapping, which is typically more costly (for data-size-agnostic swapping) than comparisons. Median-of-5 using at most 7 comparisons can be coded (and maintained) easily. In median-of-medians, medians are obtained for sets of elements, then the median of those medians is found. If selection (i.e. finding the median) has some cost kM for finding the median of M medians, and the cost of each of the M medians is 7 comparisons, then the overall cost of finding the median-of-medians for sets of 5 elements using 7 comparisons per set is $1.4 + 0.2kN$ [1].

Sets of 3 elements can be used; the rank guarantee is 33.33% to 66.67% asymptotically, which is a tighter bound than for sets of 5. The cost of obtaining the median of a set of 3 elements is at most 3 comparisons giving an overall cost for median of medians using sets of three elements of at most $1 + 0.333kN$. If the cost k of selection is less than 3, the tighter bound on pivot rank provided by median-of-medians using sets of 3 elements can obtained at lower cost than the looser guarantee from median-of-medians using sets of 5 elements. A simplification of median-of-medians ignores "leftover" elements if the array size is not an exact multiple of the set size, with a slight increase in the range of pivot rank.

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

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

5 Repivoting decision

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

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

Or in words, sorting proceeds by pivot selection and partitioning, followed by recursion on the two regions resulting from the partition. The logartihm of a fraction is the difference between the logarithm of the numerator and the logarithm of the denominator:

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

Rearranging terms:

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

For large N, $N \approx n$ and

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

If the two regions resulting from partitioning have equal size, $d = 2$ and $c = a + b$. For an input sequence and pivot selection method which always results in the same split d, there is a constant factor for complexity given by equation (4) (let $a + b = 1$). For example, a split with approximately a third of the elements in one region and two thirds in the other, $d = 3$, yields a complexity of $\approx 1.089 N \log_2 N$. A much more lopsided partition, with 15⁄16 of the elements in one region produces a complexity of $\approx 2.965 N \log_2 N$.[1]

Fast 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 $a \approx 1$. Median-of-medians pivot selection using sets of 3 elements costs at most $1 + 0.333kN$ 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, $b = 1 + 0.333k$ and $c = a + b = 2 + 0.333k$. The worst asymptotic split for median-of-medians with sets of three elements is 2:1, so the worst case would be $1.089 \times (2 + 0.33k) = 2.178 + 0.363k$. If the number of comparisons per element for median selection, $k = 2$, then repivoting is advantageous at $d > 14$. Less pessimistic assumptions, e.g. $\frac{8}{3}N$ average cost per median-of-3, possible savings when repartitioning (described in a later section), etc. lead to a lower limit for repivoting, around $d > 7$.

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 that the final step of the derivation specifies large N. However, it provides an idea of where to consider repivoting.

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 (or indeed, empty) – at minimum at least the pivot will have been

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

eliminated from the problem size. So it is only the size of the larger of the non-pivot regions to the whole which is important, and repivoting is used only if the size of that region represents an inadequate reduction in the problem size from the size of the sub-array from which is was partitioned. As mentioned earlier, an occasional lopsided partition is produced when processing inputs which are not particularly adverse. To prevent excessive repivoting, it may be desirable to ignore some small number of lopsided partitions. More details are provided in a later section on the topic of performance tuning.

6 Element swapping in Bentley & McIlroy's qsort

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. After the function returns, the only evidence that anything at all happened is the lapse of time and the heat generated by the hardware. Performance can be improved by replacing the swap function with one which first checks for identical pointers and returns early. This results in about a factor of 3 reduction in run time for sorting an array with all equal elements, and about a 25% reduction in run time for sawtooth inputs. Bentley & McIlroy [9] noted that their implementation

> depends more heavily on macros than we might like

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

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

7 Partitions

Bentley & McIlroy considered various partitioning schemes, arriving at one which starts with swapping the selected pivot element to the first position, 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 [13] [14] were investigated in the present work. However, the split-end scheme described by Bentley & McIlroy works best. Its performance advantage is a result of two factors:

- The use of two pointers to process the unknown region from both ends, then placing two out-of-place elements in their correct regions with a single swap.

- The split-end equals regions permit putting an element comparing equal to the pivot (when scanning from the upper end of the unknowns) into the upper equals region with a single swap, rather than the two swaps (or equivalent 3-way exchange) that would be required with an equals region only on the left. Partitioning schemes that do not separate elements comparing equal to a pivot fare even worse; they entail additional comparisons which result in additional swaps.

Those two factors reduce the number of element swaps, and the use of a single pivot reduces the number of comparisons and swaps compared to multi-pivot schemes[1].

Insight: Explore options; test, measure, and compare

[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 (with additional comparisons and swaps) to process elements comparing equal to one or more of the pivots, or requires more complex swaps.

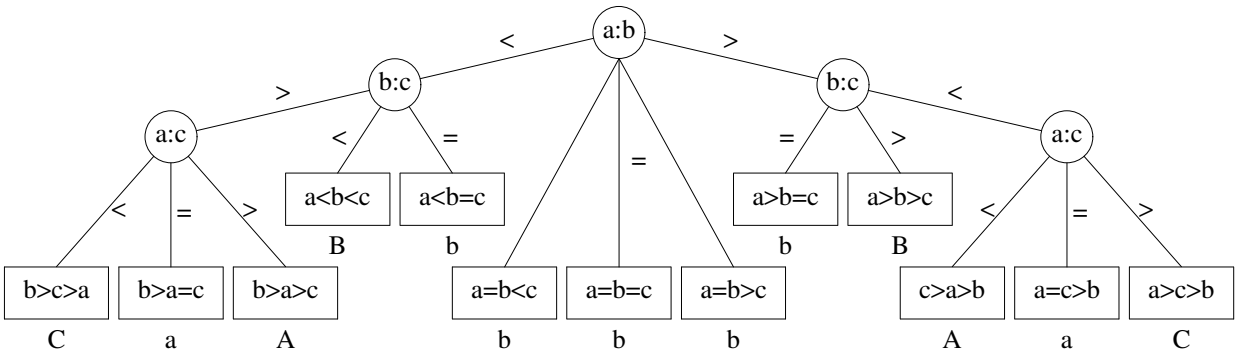

Figure 1: Ternary median-of-3 decision tree

8 Ternary median of three

Median-of-3 is used extensively in pivot selection; by itself and as a component of more complex pivot selection methods. The implementation in Bentley & McIlroy's qsort fails to take advantage of elements which compare equal. If any two of the three elements in a median-of-3 set compare equal, either one can be taken as the median; the value of the third element is irrelevant. Modification of the median-of-3 function to return early on an equal comparison which determines the result reduces the number of comparisons required for pivot selection when some elements compare equal. For example, an array of size 41 of all-equal elements requires 44 rather than 52 comparisons to sort if equal comparison results are used.

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

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

9 Pivot element selection

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

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

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

As noted earlier in this paper, the limited rank guarantee provided by these methods permits quadratic behavior with adverse input sequences. While the break-glass mechanism presented earlier is sufficient to prevent quadratic behavior, it is desirable to have a pivot selection method which provides a better guarantee of pivot rank than the pseudo-median of nine elements. The use of only nine elements for the pseudo-median also 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, [15] 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 [15] give the range of 1-based rank as

> the smallest possible rank is exactly $\lfloor b/2 \rfloor^k$, 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 [16] and Martínez & Roura [17] have determined that sample size 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 the square of the number of samples, particularly at small sizes.

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. [18] This does, of course, have a higher cost than an implementation of Tukey's ninther that uses storage (e.g. for pointer variables) outside of the array being processed. On the other hand, the data movement which may take place during in-place remedian reduces data disorder, which is a beneficial side-effect.

Judicious use of bias in the ternary median-of-3 can reduce the amount of data movement required to implement such an in-place remedian of samples.

One drawback of all of the pivot selection methods using element comparisons is that the same comparisons may be repeated during partitioning. This redundancy is only addressed in one special instance, described in a later section, in part because the cost of doing so generally would outweigh the benefit, but also because it would tend to introduce disorder into already-sorted inputs.

10 Sampling for pivot selection

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

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

They go on to state:

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

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

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

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

Although performance was measured in the present study at each value of array size N from 7 through 4096 for the data for Figure 2, points which lie on or very near a straight line between a pair of enclosing

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

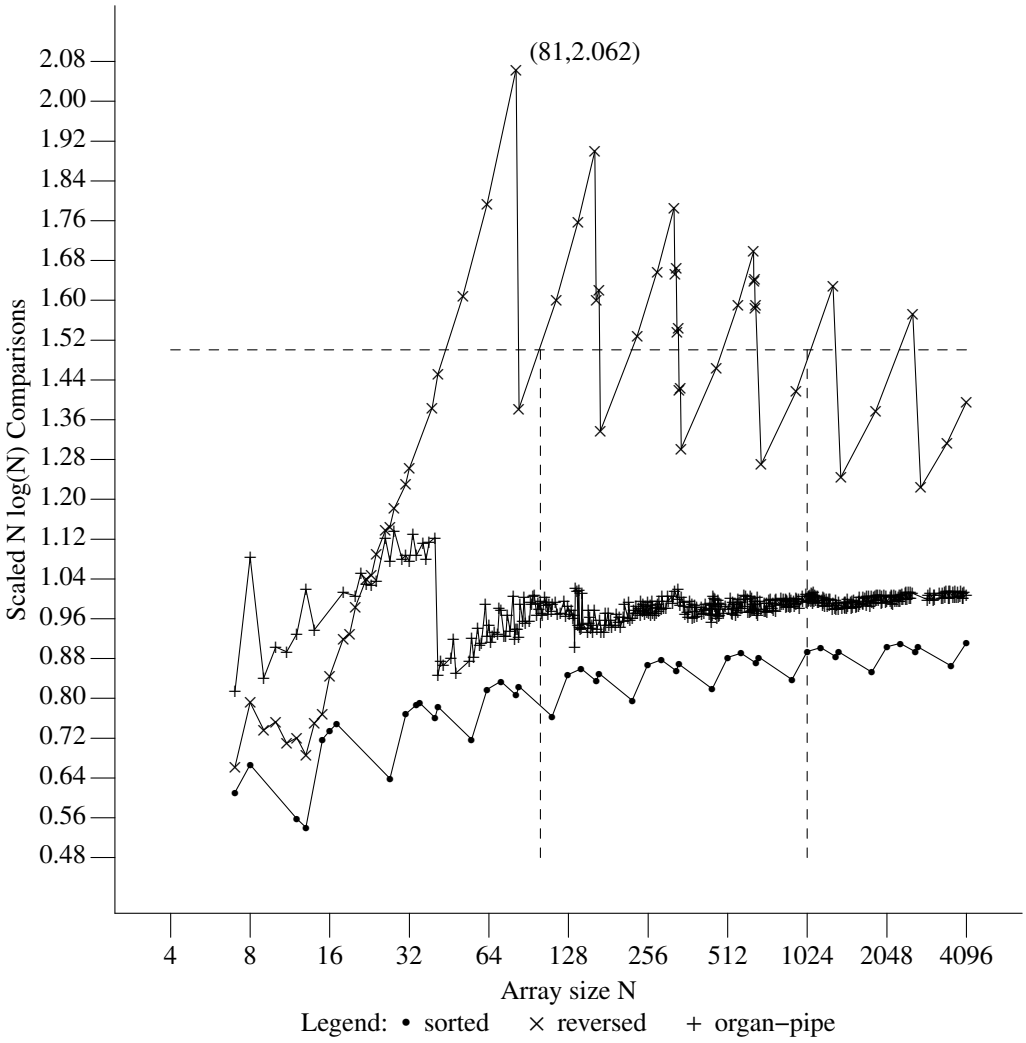

Figure 2: Comparisons for Bentley & McIlroy qsort

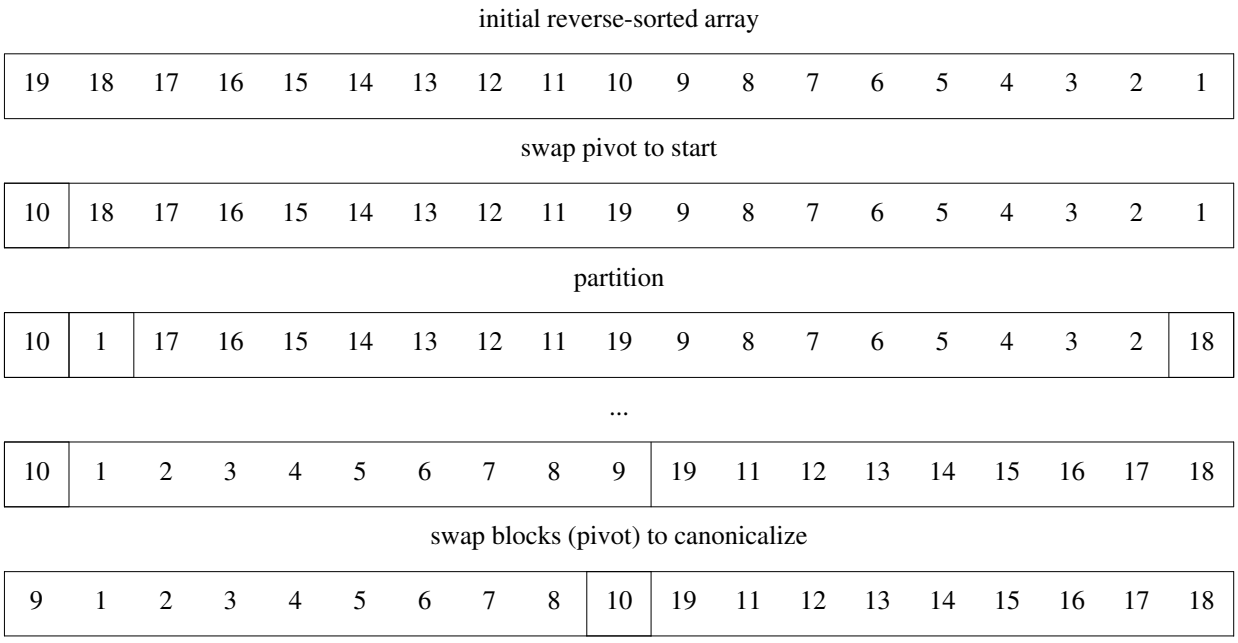

Figure 3: Partitioning of reverse-sorted input in Bentley & McIlroy qsort

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 peaks of high numbers of comparisons for reversed input shown in Figure 2 occur at values of array size N satisfying $N = 40 \times 2^k + 1, \quad k = 1, 2...$ where 40 is the cutoff point between median-of-3 and Tukey's ninther for pivot selection. Bentley & McIlroy [9] noted

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

and it is this disorder coupled with the use of the first and last array elements in computing median-of-3 pivot elements which is responsible for the peaks in the comparison counts evident in Figure 2. The peaks above the dashed horizontal line at $1.5N \log_2 N$ evident in Figure 2 would not have been noted at the array sizes tested by Bentley & McIlroy, which were 100 and 1023–1025 (dashed vertical lines in Figure 2).

It is informative to work through an example to see exactly how and why this disorder occurs and how it interacts with sampling the first and last sub-array positions for median-of-3 pivot selection. Such an example is shown in Figure 3. At the end of partitioning, **both** regions resulting from the partition have an extreme-valued element at the first position. 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. All partitioning methods which initially swap the pivot element to one end of the array share this characteristic, and if pivot selection uses median-of-3 with samples at the array endpoints, the same poor performance with reverse sorted input sequences can be expected. There are two interacting factors responsible for the poor performance:

1. The initial swap of the pivot element to one end of the array, which moves an extreme-valued element initially there to near the middle of the array, where it ends up at one side of the oppsosite 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.

2. Use of samples at the first, middle, and last elements for median-of-3 pivot selection.

Arrays of *double* types were affected in Bentley & McIlroy's qsort because of an optimization its authors made for *long* integer types (and types of the same size); macro PVINIT places the pivot in a separate location rather than swapping to the first position, but only for types of the same size as *long*.. On 32-bit architectures such as were available in 1992, doubles were of a different size than long integers; therefore the PVINIT macro was ineffective for doubles, which were found by Bentley & McIlroy to exhibit the worst performance for reverse-sorted inputs. On 64-bit architectures commonly available in 2016, sorting doubles

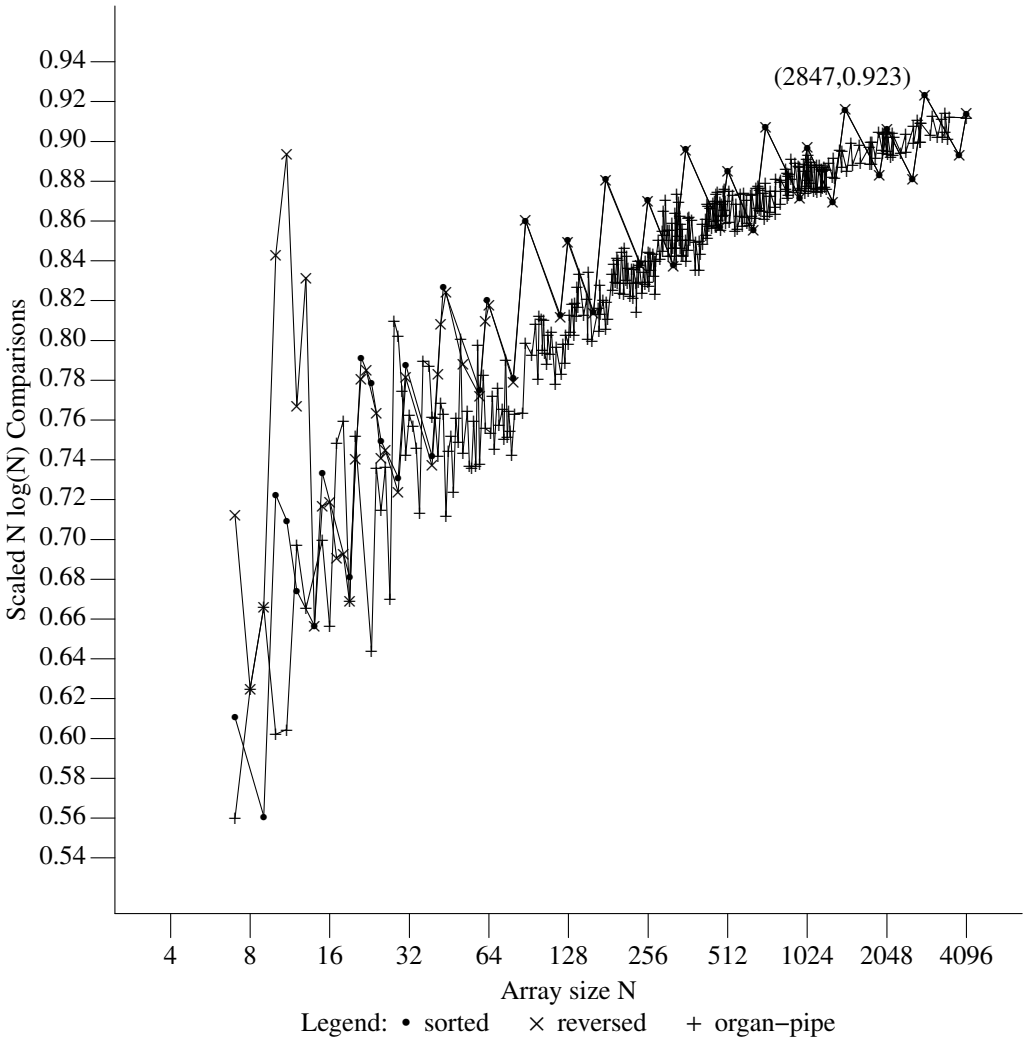

Figure 4: Comparisons for modified Bentley & McIlroy qsort

does not exhibit the same behavior noted by Bentley & McIlroy; the behavior instead shows up for other data types, such as the integers used to generate Figure 2.

A size-independent modification to Bentley & McIlroy's partitioning routine can achieve the same prevention of introduction of disorder. Initially, do **not** swap the pivot to the first position. When an element comparing equal to the pivot (perhaps the pivot itself) is found, swap it to the corresponding left or right equal-elements block **and** update the pivot pointer to point to that element. Like Bentley & McIlroy's PVINIT, this has the effect of deferring the swap of the pivot element, which causes less disorder. Type-size-independent deferral of pivot element swapping allows removal of the PVINIT macro (and its susceptibility to data type size changes). Also like Bentley & McIlroy's [9] PVINIT

> When the trick helps, the speedup can be impressive, sometimes even an order of magnitude. On average, though, it degrades performance slightly because the partition scan must visit n instead of $n-1$ elements. We justify the small loss in average speed — under 2 percent in our final program — on the same psychological grounds that impelled us to fat partitioning in the first place: users complain when easy inputs don't sort quickly.

A side-effect of updating the pivot as described is that the pivot becomes the closest suitable element to the unknown status elements which are compared to it, slightly improving locality of access compared to the pivot at the extreme end of the array.

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

Use of the array endpoints with median-of-3 is detrimental when finding a pivot for partitioning organ-pipe inputs as well as when disorder is introduced as shown for reverse-sorted inputs. Interestingly, Bentley &

McIlroy began work on their implementation as the result of a reported problem in an earlier implementation of qsort — when presented with an organ-pipe input sequence:

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

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

Input sequences congruent to an organ-pipe sequence arise in many instances. For example, consider an array where values represent the number of elements in histogram bins for some approximately normally distributed data. The values are low at the ends and high in the middle, just like an organ-pipe sequence.

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

Quality (as distinct from quantity) of sampling for pivot selection in quicksort is rarely discussed; many published papers 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 the ¼ and ¾ positions rather than the array ends.

By deferring the movement of the pivot element coupled with modified sampling, performance of Bentley & McIlroy's qsort operating on reverse-sorted 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 4. The anomalous high comparison count region ($> 1N \log_2 N$) for organ-pipe input below array size 41 seen in Figure 2 has been eliminated, as has the poor performance with reverse-sorted input sequences. Performance for both reverse-sorted and organ-pipe input sequences is markedly improved; nearly the same as for already-sorted input.

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

> The curse it is cast
> The slow one now
> Will later be fast [19]

Insight: Be wary of clever macros.

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

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

The third sampling method used by Bentley & McIlroy, use of the middle array element as the sole sample, also yields poor results with organ-pipe input sequences, for the same reason that use of the array endpoints does; the endpoints and the middle are where the extreme values of an organ-pipe sequence are located. Using an extreme value as a pivot virtually guarantees poor quicksort performance. 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.

11 Improved sampling

The three sampling methods used by Bentley & McIlroy have one thing in common: the number of samples used is a power of 3, and that is also characteristic of the larger sample sizes used for remedian of samples. The smallest number, $3^0 = 1$, presents a dilemma: the middle element is ideal for sorted and reversed input, but is a disaster for organ-pipe and similar input sequences. Conversely, the best choices for organ-pipe sequences, namely ¼ or ¾ position, are sub-optimal (but not disastrous) for sorted input.

For small array sizes where single-element sampling is used it is fairly easy to compare alternative single-sample sampling strategies. It is clear that the middle element is disastrous for organ-pipe inputs: paradoxically, that was the input sequence that prompted Bentley & McIlroy [9] to redesign qsort, but the middle element was chosen for their redesign.

Insight: Maintain perspective (what is the problem to be solved?).

Because the middle element is one of the worst possible choices for organ-pipe sequences, some alternative should be used when a single sample selects the pivot element. The element at the ¼ position is optimal for organ-pipe inputs; it is reasonable to consider the effect of using that element when sorting already-sorted inputs. Another possibility to consider is some compromise between what is best for organ-pipe inputs (¼)

and what is best for sorted inputs (½), such as the element at ⅓ position. Yet another possibility is to forego single-element sampling and use the median of the three elements at ¼, ½, and ¾ positions, which will work equally well for organ-pipe and sorted input sequences, but entails the additional comparisons for the median-of-3. Through array size 9 for already-sorted inputs, use of the ¼ position element as the pivot is no worse than using the ⅓ position or the median of three described above. Both of those alternatives are always worse for organ-pipe inputs; ¼ is always optimal for that sequence. Therefore, among the alternatives under consideration, the ¼ sample is reasonable for arrays through size 9. At and above an array of 10 elements, the median-of-3 outperforms the other alternatives for already-sorted inputs. The ⅓ position alternative can therefore be eliminated from consideration, and the relevant tuning decision becomes a matter of choosing at what array size to switch from use of the ¼ position element to the median-of-3, with the goal of providing reasonable overall performance.

Switching to median-of-3 at array size 10 would add about a 23% increase over optimum for organ-pipe inputs, decreasing as a proportion for larger arrays. Switching at a larger array size would impose a smaller worst-case increase for organ pipe inputs. Deferring the switch to larger array sizes penalizes already-sorted inputs; using the ¼ position costs 3% to 7% more comparisons for sorting than does use of the median-of-3 for arrays of size 10 through 13. At array size 14, the penalty for continuing to use a single sample at the ¼ position jumps to almost 15% for already-sorted inputs, then to more than 16% at 15 elements before dropping back to below 11% for 16 through 18 elements. Experimentally, switching to median-of-3 at array size 11 results in better performance when sorting already-sorted input with quickselect than with Bentley & McIlroy's qsort, however performance for random input sequences suffers somewhat because of the added comparisons.

Insight: Analyze algorithms (in addition to experimentally measuring performance).

The choice of cutoff represents a tradeoff between the cost of the comparisons to find the median of three elements, and the benefit of an improved pivot rank. While a cutoff of 11 would provide excellent performance for already-sorted inputs, it has poor performance for random inputs for small and large array sizes. A cutoff value of 13 provides reasonable performance for random input sequences, and penalizes organ-pipe and sorted input sequences by less than 15%.

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 an empirically-determined cutoff where median-of-3 was used at arrays of size 8 or more, whereas the optimum cutoff in the present work lies at 13. Using median-of-3 for 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 for small arrays leads to improved performance for all array sizes.

A reasonable choice for selecting three elements is to use elements at (or as near as practicable to) ¼, ½, and ¾ positions. That will work ideally for both sorted and organ-pipe input sequences, and avoids the troublesome array endpoints.

For more than 3 samples, it is still desirable to include samples at the ¼, ½, and ¾ positions. However, these samples should be members of separate columns. (Consider 9 samples from an 81-element organ-pipe sequence with ¼, ½, and ¾ samples in the same column to see why.)

Median-of-3 used by remedian of samples uses a number of samples which is a power of 3 and requires 3 rows of samples for column-wise medians-of-3. An array of N elements sampled by $S = 3^k$ samples has $R = \frac{N}{3}$ elements per row and $T = \frac{S}{3} = 3^{(k-1)}$ samples per row. Avoiding endpoints in each row avoids array endpoints and avoids clustering samples near row boundaries. Sample spacing within each row is then $Q = \frac{R}{(T-1)}$ elements. Offsetting the ¼ and ¾ element columns from the column containing the ½ element implies row spacing of $\frac{N}{4} + Q$ elements. This sampling strategy gives 3 evenly spaced rows each containing evenly spaced samples with the ¼, ½, and ¾ sampled elements in separate columns.

An adaptive sampling method was adopted for the present work:

- The number of samples is always a power of 3, and is approximately the square root of the number of array elements.
- The element at the ¼ position is used for small arrays for which only a single sample is required.
- Three elements chosen close to ¼, ½, and ¾ positions are sampled for arrays requiring 3 samples. These elements provide optimal partitioning for already-sorted and organ-pipe input sequences,
- Larger arrays use a selection of a power of three samples with row and column spacing chosen to include samples near ¼, ½, and ¾ positions, but to avoid placing them in the same columns.

Such a sampling method was used to produce Figure 4, using no more than 9 samples (same as Bentley & McIlroy's qsort), which was used for array sizes 87 and larger.

12 Costs and benefits

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

- Sampling improvements require a small amount of code, certainly a bit more than simply selecting a middle element, but the performance improvements are substantial. Improved quality of sampling accounts for much of the improved sorting performance of the present work for non-adversarial input sequences.

- Improved median-of-3, taking equality comparisons into account adds no cost, but provides substantial performance improvement for some common input sequences.

- Replacement of SWAPINIT and related macros with an inline swap function which avoids excess work simplifies maintenance and provides performance improvement. There is a noticeable increase in object file size, but that is because of the additional code which supports efficient swapping in units of size in addition to *char* and *long*.

- Remedian of samples for fast pivot selection is slightly more complicated than Tukey's ninther, but provides performance improvement for large arrays. The code increase is minimal and subsumes computation of ninther; it uses the same basic median-of-3 used by other pivot selection methods.

- An internal stack would add some code, however would also facilitate selection, which might otherwise have to implemented and maintained separately. In addition to being one of the project goals, selection is required for median-of-medians pivot selection, which is used to avoid quadratic sorting behavior. An alternative chosen for the implementation described in this paper is a qsort-compatible wrapper function with an internal implementation that uses a range of indices to bound the sub-array being processed. Small subarrays are processed recursively; large ones iteratively, avoiding the overhead of maintaining an internal stack.

- The "break-glass" mechanism consists of examination of the sizes of the regions resulting from partitioning, determination whether or not to repartition a large region, and implementation of a guaranteed-rank pivot selection mechanism (median-of-medians). All of these require some additional code, parts of which reuse code already used, but without some mechanism, quadratic sorting behavior is unavoidable. The break-glass mechanism does not interfere with selection. Performance against adverse input sequences is compared to introsort in a later section of this paper.

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

- 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 [16] . Saving about 1/3 of the comparisons required for repartitioning requires some additional code to skip over already-partitioned elements[1]. This further requires some transfer of information about the extents of the partial partitioning via added function call parameters. The additional overhead provides some benefit to a rarely-used part of the whole. Because it provides a small but measurable benefit, it is available as a compile-time option in the implementation described in this paper. An alternate means of avoiding recomparison was proposed by Martínez & Roura [17] : 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.

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

13 Program and related text analysis

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

Insight: Match code to specification where possible.

14 Assembling the product

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

- An internal function using indices to bracket the sub-array being processed, with an invariant array base pointer so that element ranks may be determined. A wrapper function provides the standard *qsort* interface. When performing (multiple) selection, regions not containing desired order statistic ranks are ignored. Smallest regions are processed first to prevent overrunning the program stack when sorting.
- Partitioning based on the efficient split-end method used by Bentley & McIlroy, with swapping of the pivot element deferred to reduce introduction of disorder.
- A break-glass mechanism to obtain an improved pivot in the event of extremely lopsided partitions, improving performance and preventing quadratic worst-case performance. The mechanism defends against adverse inputs without sacrificing speed.
- A pivot selection algorithm for use with the break-glass mechanism using median-of-medians with sets of 3 elements to provide a guaranteed relatively narrow range of pivot rank, computed at reasonable cost. Optionally, skipping over the partially partitioned elements resulting from median selection, avoiding recomparisons of those elements.
- A fast pivot selection method using remedian with base 3 over a sample of array elements; sample size is a power of 3, increasing slowly as array size increases. The use of more samples for larger arrays provides an improved pivot rank guarantee and statistical efficiency of the pseudo-median.
- Improved sampling of elements for single sample pivot selection, median-of-3 pivot selection, and the other pivot selection methods, avoiding array endpoints. Improved sampling results in improved performance for commonly-occurring input sequences, such as organ-pipe sequences and less need for use of median-of-3 pivot selection for small arrays, resulting in overall performance improvement due to reduced comparison count.
- Improved median-of-3 taking advantage of comparison results indicating equal-valued elements.
- An inline function is used for swapping, eliminating macros and special-case code; it avoids self-swapping and supports swapping by a variety of available sizes.
- Insertion sort for sorting very small arrays, as in Bentley & McIlroy's qsort implementation.

15 Performance tuning

Cutoff values for increasing the number of samples used for pivot selection using remedian of samples were determined by measuring the number of comparisons used to sort various sizes of arrays of randomly shuffled distinct integers with different cutoff values. Each cutoff value was then selected as the value which required the fewest comparisons for sorting the shuffled integers. In most cases, and for sizes beyond what is practical to measure, the cutoff values are at the square of the number of samples, as suggested by McGeoch & Tygar [16] and Martínez & Roura [17] . However, at small sizes the optimum deviates from the square, e.g. 3 samples are used for arrays of size 13 (rather than 9) or larger.

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. For example, an array with 18 elements with a pivot selected by median-of-3 must have at least one element in the smaller partitioned region, and therefore at most 16 elements in the larger region, limiting the ratio to at most 16:1. Given any desired threshold for the ratio to re-pivot, there is a minimum array size at or above the median-of-3 pivot selection cutoff for which that ratio is feasible. Testing the ratio can therefore be avoided below that size. The repivoting threshold is not a clear case of finding an optimum; there isn't one. Repivoting is a tradeoff between expending some effort to save possible (but not certain) greater future effort and expending unnecessary effort when an occasional lopsided partition is encountered. Repivoting at a moderately low ratio ensures good performance with adverse input sequences, but also raises the amortized cost of processing randomized input sequences, which sometimes result in a lopsided partition. Conversely, repivoting only when a partition is extremely lopsided protects against reducing performance when processing random input sequences, but permits adverse input sequences to result in poor performance. The tradeoff can be managed with three compile-time parameters, a small integer called LOPSIDED_LIMIT, a ratio called REPIVOT_FACTOR, and a cutoff array size called REPIVOT_CUTOFF. Below array size REPIVOT_CUTOFF, partitions are not checked for lopsided regions. At and above that cutoff, repivoting takes place only if the ratio of the size of the large region to the number of other elements minus 1[1] is at least as large as REPIVOT_FACTOR, and then only if the number of lopsided regions (including the present one) seen since the last repivot (or the start of sorting or selection) is at least as great as LOPSIDED_LIMIT. At one extreme, with LOPSIDED_LIMIT set to 1, REPIVOT_FACTOR set to 6, and REPIVOT_CUTOFF set to 7, 128 Ki element arrays of randomly shuffled sequences of distinct integers sort using about $1.0478N \log_2 N$ comparisons with about 15.3% of the partitions resulting in repivoting, and the performance against sequences generated by McIlroy's adversary is worst at about $1.329N \log_2 N$ at an array size of 119 elements, decreasing to below $1.2N \log_2 N$ at large sizes. At the opposite extreme, if repivoting is effectively disabled, the randomly shuffled sequence complexity is lowered to about $1.01794N \log_2 N$ comparisons and adverse inputs can still lead to quadratic behavior, although the improved pivot rank guarantees provided by remedian of samples pivot selection limits the effect compared to Bentley & McIlroy's qsort, and processing small regions first ensures that quickselect cannot overrun its program stack. A reasonable compromise is LOPSIDED_LIMIT set to 3, a REPIVOT_FACTOR of 6, and REPIVOT_CUTOFF of 12, which provides randomly shuffled array performance of $1.0183N \log_2 N$ (i.e. 0.035% higher than with repivoting disabled) with about 0.17% of partitions resulting in repivoting and worst adversarial performance of $1.601N \log_2 N$ comparisons at an array of size 47 elements, decreasing to around $1.2N \log_2 N$ at large sizes. That limits worst-case performance to less than 60% worse than the expected performance, at a cost of $\sim 0.035\%$ for randomly shuffled sequences, which is not a great sacrifice in speed.

16 Performance results

Comparisons were made with gcc version 6.2.0 using -Ofast optimization.

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 adverse inputs at modest array sizes. Quickselect maintains linearithmic complexity even against McIlroy's antiqsort adversary. [11] 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 even at large array sizes, as shown in Figure 6. Low comparison count is most important when the comparison function is slow, such as when comparing large data (e.g. long character strings) or when multiple keys need to be compared to resolve partial matches. Figure 5 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 ternary median-of-3, type-independent deferred pivot swap, modified sampling, cutoff values (12 for median-of-3 and 87 for ninther) previously discussed, avoids self-swapping, swaps in any appropriate basic type size, and processes the smaller partitioned region first, recursively, then processes the large region iteratively (without repeating SWAPINIT).

In tests, dual-pivot quicksort [14] performed poorly in terms of number of comparisons, number of swaps, and run time compared to the modified version of Bentley & McIlroy's qsort and compared to the implementation of quicksort described in this paper. 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

[1] For d in equations (1) through (4), that ratio is $d - 1$

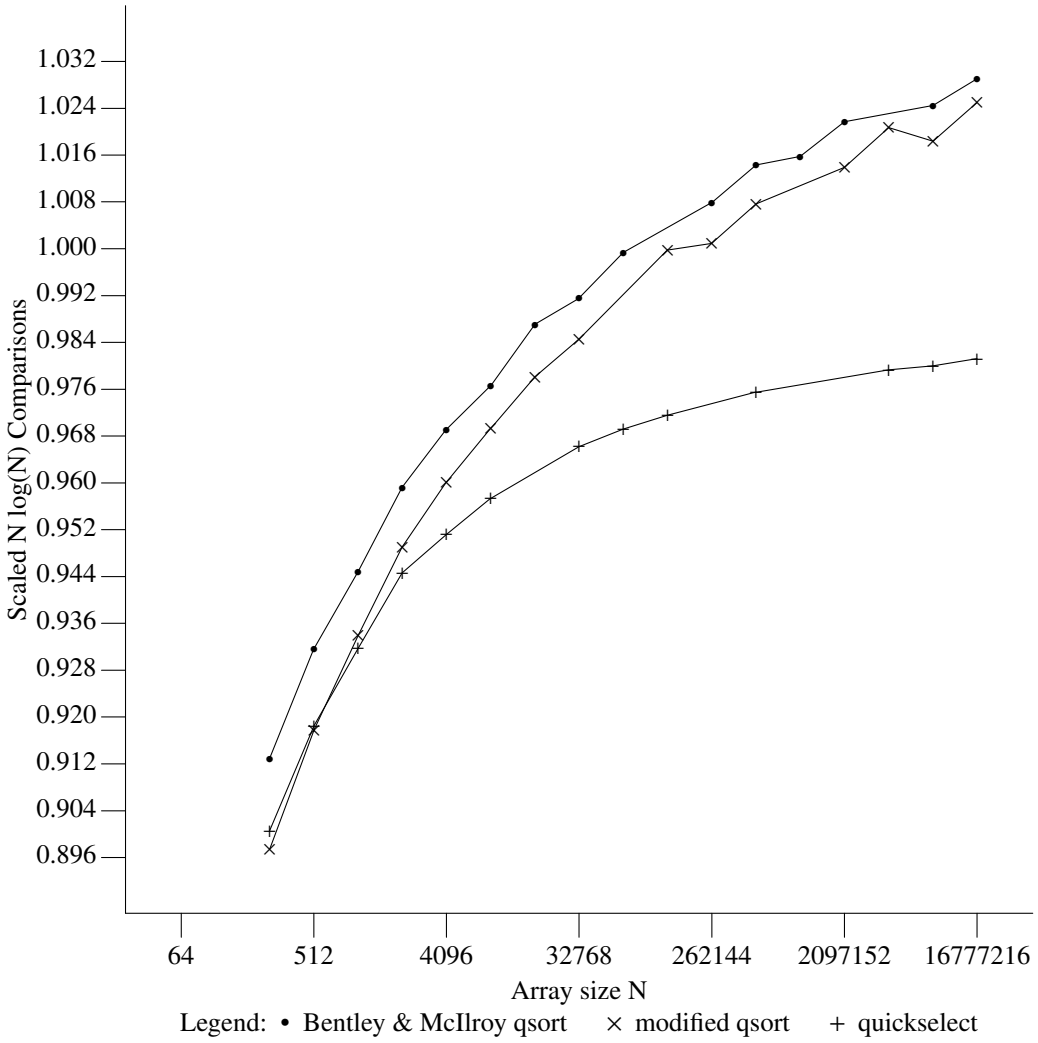

Figure 5: Comparisons for Bentley & McIlroy qsort and quickselect, random input

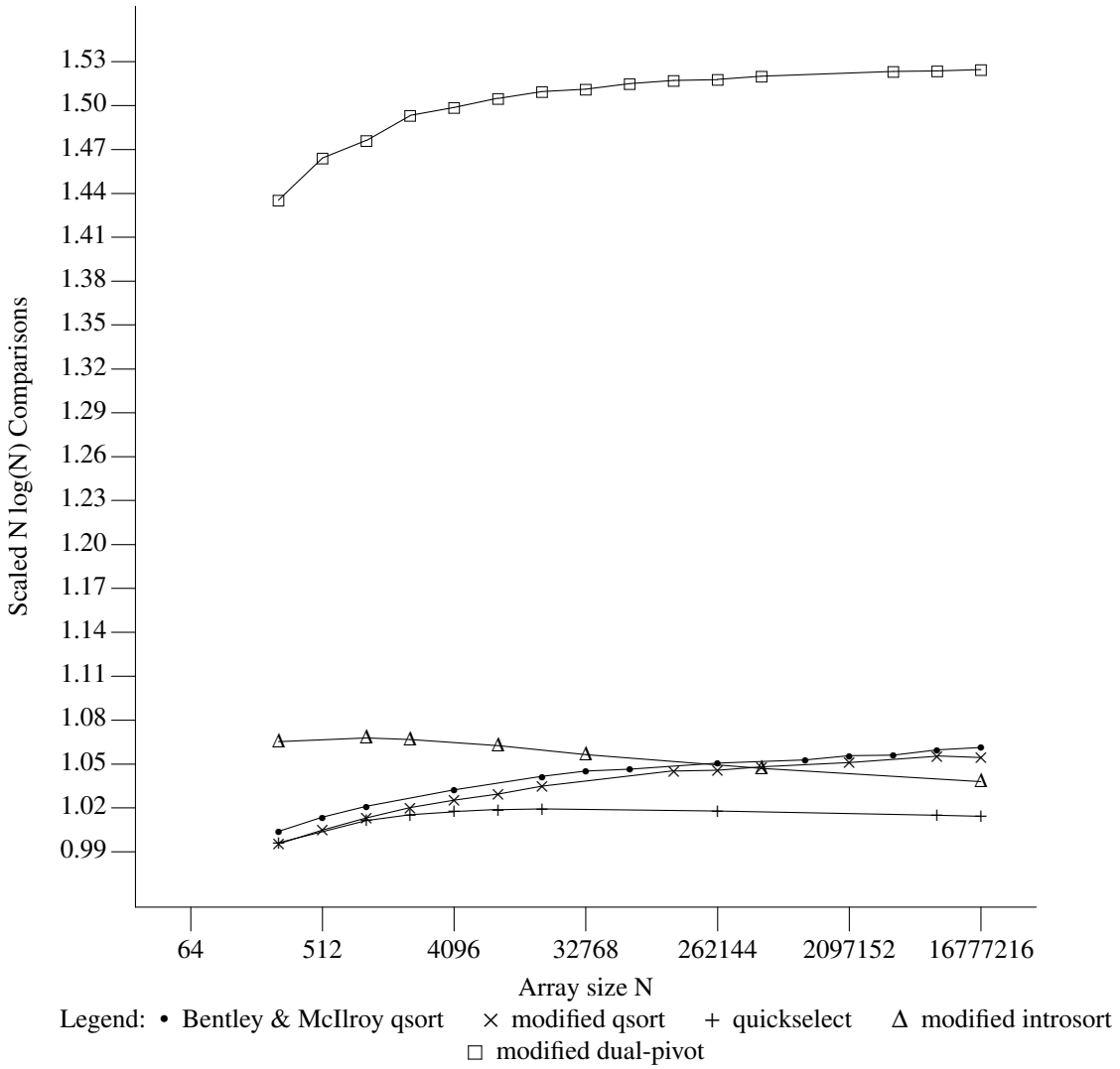

Figure 6: Comparisons for sorting implementations, shuffled input

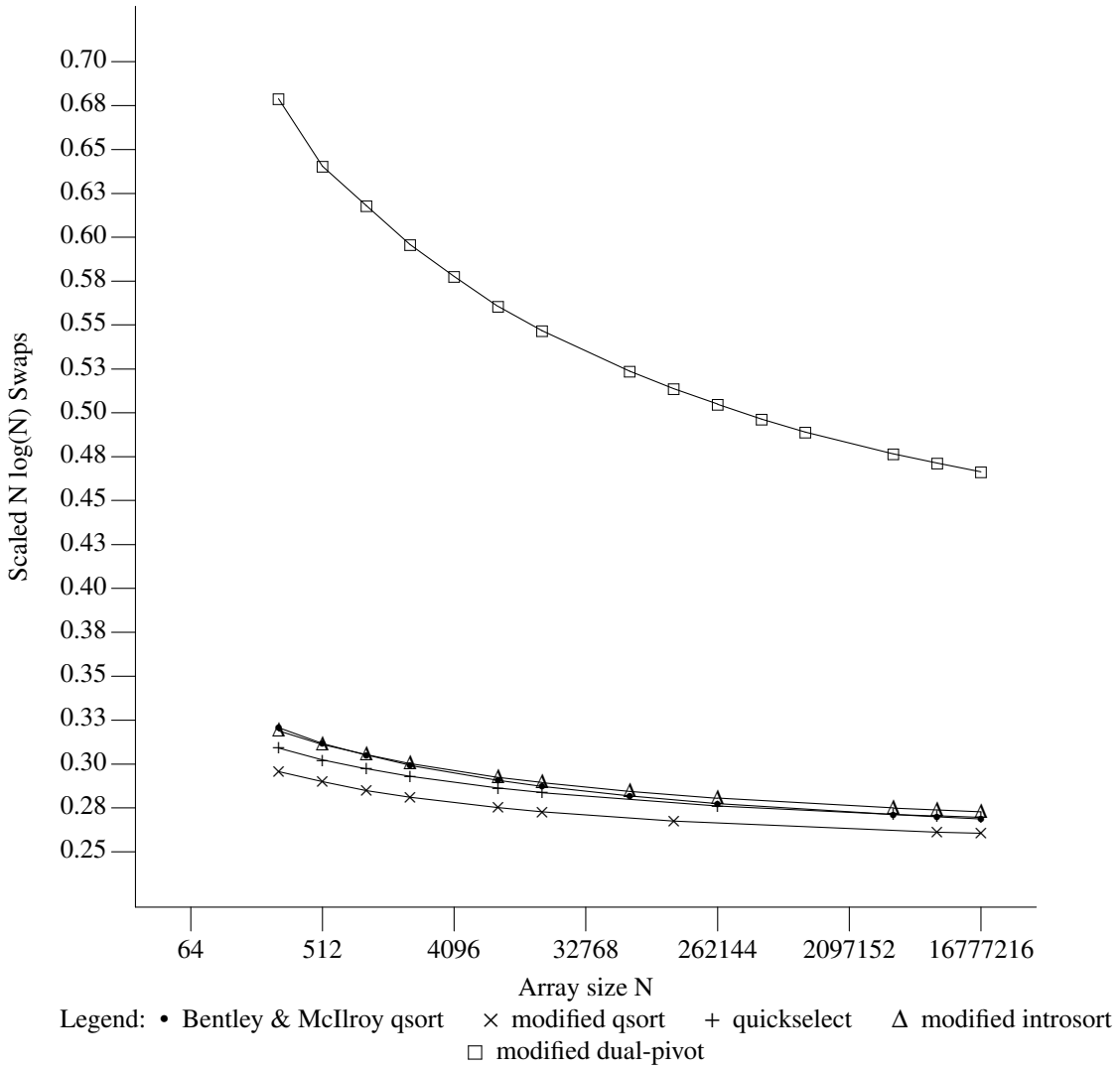

Figure 7: Swaps for sorting implementations, shuffled input

of modest amounts of data, the dual-pivot quicksort exhibited much longer run times than the presently described implementation. It is conjectured [20] 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 are insufficient to achieve competititve performance even for trivial data types. Kushagra et al. [20] measured 7-8% performance improvement of some dual-pivot sort implementation over some unspecified 1-pivot quicksort using some version of median-of-3 pivot selection with distinct-valued input. The modifications to Bentley & McIlroy's qsort described above result in run time about 5% lower than the unmodified qsort as measured in the present study, which is itself about 5% faster than Yaroslavskiy's dual-pivot sort with a similar polymorphic qsort-like interface, all measured sorting 100000 randomly shuffled distinct plain integers over 1000 trials. Simple input sequences (mod-3 sawtooth, random zeros and ones) that should sort quickly take significantly less time for the modified Bentley & McIlroy qsort than the original which in turn take significantly less time than Yaroslavskiy's dual-pivot sort: differences in each case amounted to more than 20% run time. Analysis by Wild et al. [21] indicates a disadvantage of dual-pivot schemes for selection. Multi-pivot partitioning schemes have several shortcomings for sorting and selection:

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

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

An implementation of introsort was also compared to the implementation described in this paper (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 with deferred pivot swapping and with introsort modified to process only less-than and greater-than regions, improved swapping, ternary median-of-3), performance was made even better than quickselect for simple data types at moderate array sizes, but not for data types involving non-trivial comparison or swapping or for large arrays. Improvement in sampling quality and quantity yielded additional improvement to introsort for larger arrays, but because the number of comparisons and swaps is greater for introsort than for quickselect, introsort performance for data types with non-trivial comparisons or swaps is lower than for quickselect. The cause of the higher number of comparisons and swaps is the final insertion sort over the entire array; if that is changed to use insertion sort only at the end of the introsort loop on subarrays, the number of comparisons and swaps for introsort with all of the above modifications becomes slightly lower than for quickselect (because of occasional repivoting in the latter). A final insertion sort over the entire array includes all elements which were selected as pivots during partitioning, and all elements which compared equal to them, also any elements which ended up as the sole element of a partitioned region; those elements are not involved in any of the many smaller insertion sorts when those are performed at the end of the loop. Introsort avoids quadratic behavior, but shows worse performance than quickselect for adverse input sequences. 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.

Figure 6 is a graph of comparison counts for sorting randomly shuffled distinct input values vs. array size, and Figure 7 shows the scaled number of swaps. The modified version of Bentley & McIlroy's qsort uses fewer comparisons and fewer swaps than the unmodified version. The number of comparisons for

Sequence	Bentley & McIlroy qsort	quickselect
sorted	0.91789	0.93850
reversed	1.18971	0.93850
shuffled	1.04752	1.01881
organ	1.02951	0.94847
sawtooth	0.09032	0.09041
binary	0.09032	0.09046
equal	0.06021	0.06028
m3k	1.06122	0.99953
random	1.04838	1.01885
limited	1.00190	0.97055

Table 1: Scaled $N \log_2 N$ Comparisons for Bentley & McIlroy qsort and quickselect input sequences

quickselect is lower still, especially at large array sizes, at the expense of slightly more swaps (during in-place median-of-3 and remedian of samples pivot selection). At large array sizes, it trades a reduction of $\approx 0.04N \log_2 N$ comparisons for an increase (relative to the modified qsort) of $\approx 0.01N \log_2 N$ swaps; a net performance increase provided that swaps are no more than 4 times as costly as comparisons. For very large element sizes where swaps might be expected to be expensive, one can use an auxiliary array of pointers to elements, accessing element data for comparisons via the pointers and (inexpensively) swapping the pointers rather than the large elements. Dual-pivot and introsort implementations incorporating improvements noted above are also shown in Figures 6 and 7; both use many more comparisons and swaps than quickselect. The reduction in scaled comparisons at large array sizes for the modified introsort and for quickselect result from the increased number of samples used for pivot selection for large arrays. That modification was not incorporated into the modified Bentley & McIlroy qsort, but was applied to the dual-pivot implementation, where it results in fewer comparisons than if not applied, but fails to prevent the scaled comparisons from continuing to rise, at least up to the largest array sizes tested.

Figures 6 and 7 show several effects:

- The differences between the unmodified and modified Bentley & McIlroy qsort curves result from improved sampling for very small arrays, which in turn improves performance for large arrays which are partitioned into many small arrays and permits higher cutoff values for median-of-3 and ninther pivot selection methods.
- The differences between the modified qsort and quickselect curves results from increasing sample size as the array size increases, using in-place remedian of samples for pivot selection. The result is considerably fewer comparisons and slightly more swaps for quickselect.
- The gap between introsort and quickselect results from the costly final insertion sort used by introsort.
- The very large gap between dual-pivot and the other curves 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 twice as many comparisons and swaps as the efficient Bentley & McIlroy partitioning method. Cache effects lessen the impact of the gap, but cannot overcome the disadvantages.

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

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

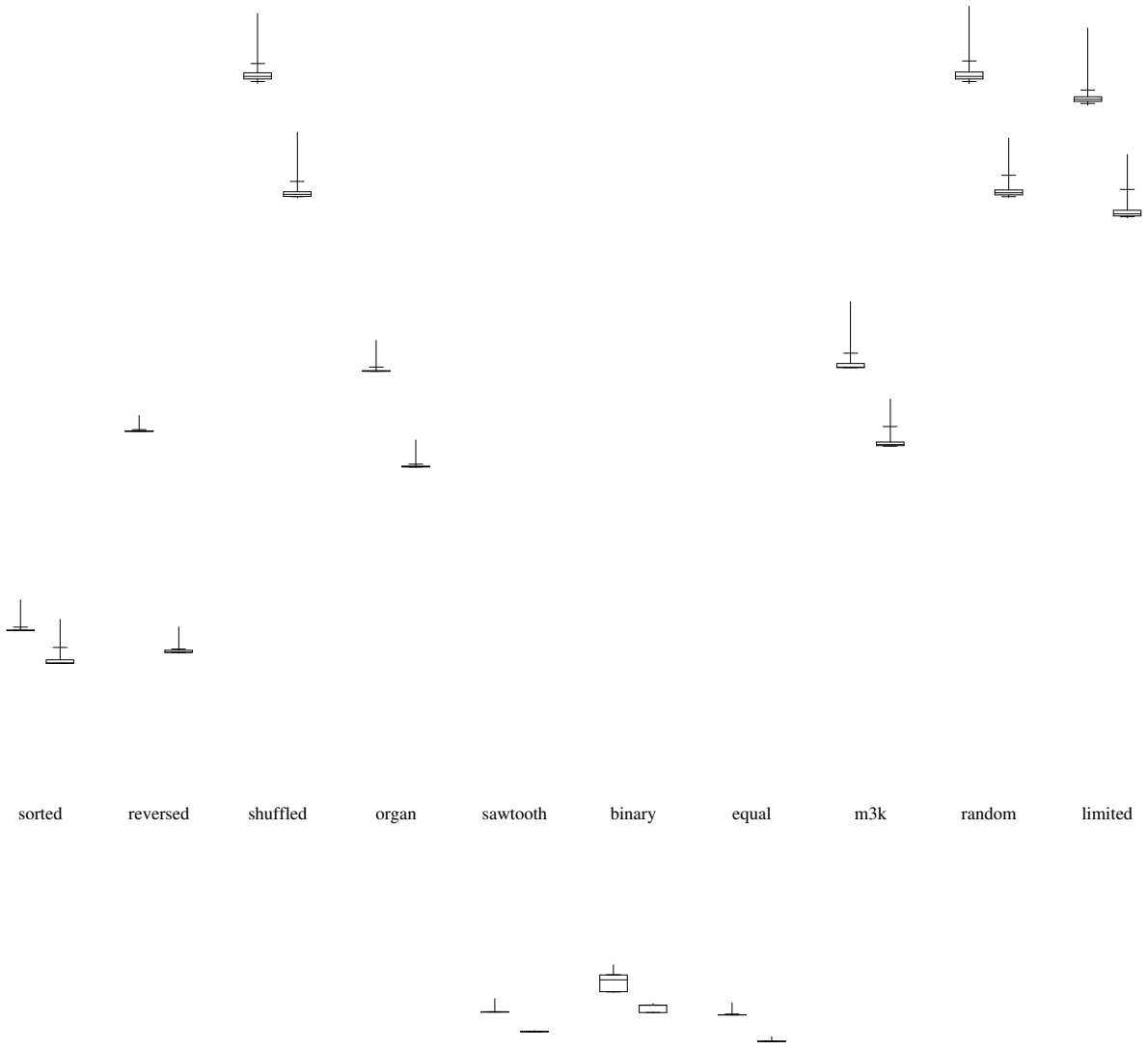

Figure 8: Run time for Bentley & McIlroy qsort and quickselect sorting input sequences

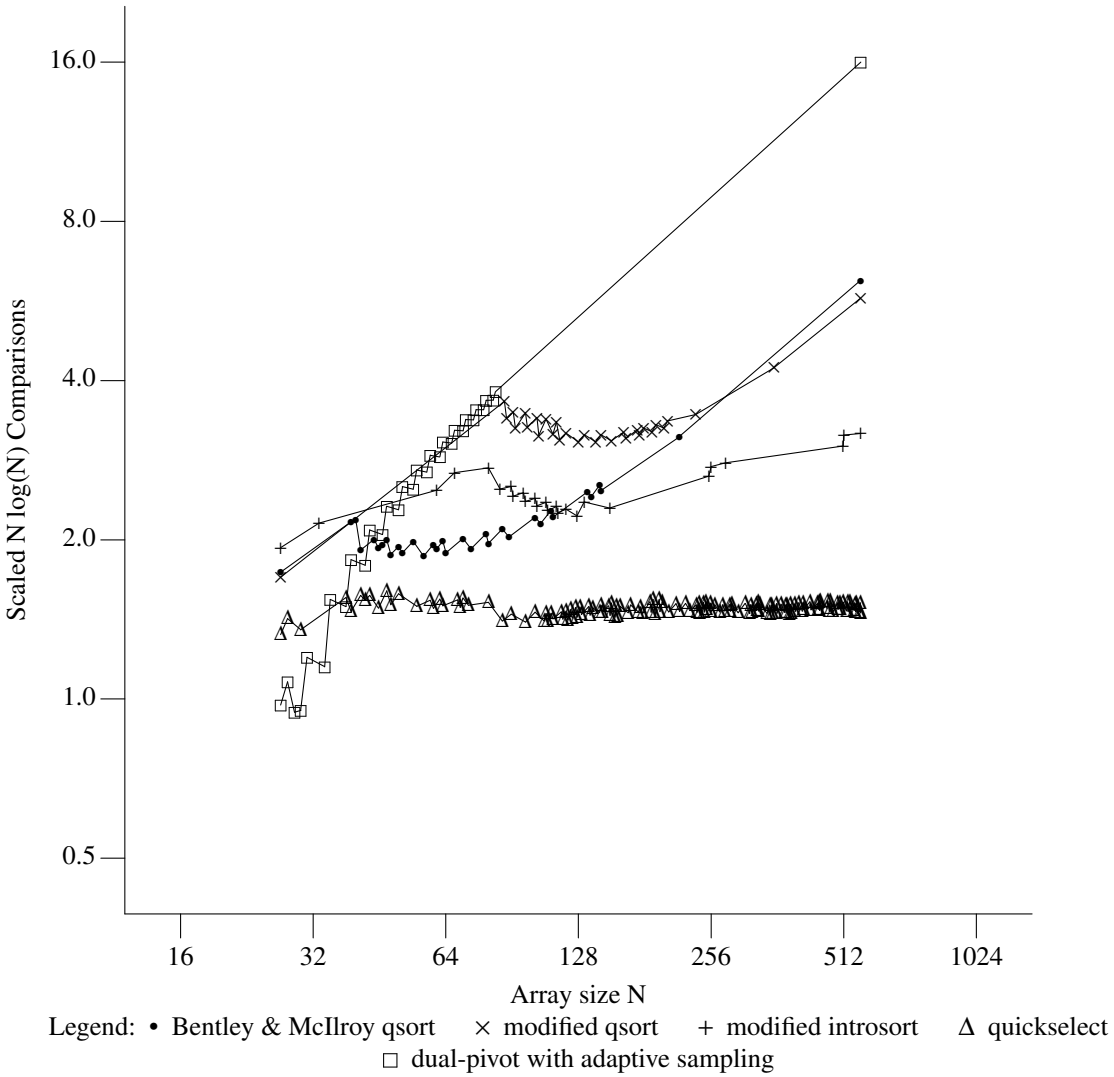

Figure 9: Adverse input comparisons for selected sorting implementations

- median-of-3 killer sequence
- 64-bit random non-negative integers
- limited-range random integers in [0,100000)

Comparison counts were averaged over the runs and scaled to $N \log_2 N$ and are tabulated in Table 1. With four exceptions, quickselect shows lower comparison counts than Bentley & McIlroy's qsort.

Running time statistics are plotted as pairs of box-and-whisker plots in Figure 8. For each input sequence, Bentley & McIlroy's qsort is the left of the pair and quickselect is on the right. The central box of each plot extends from the first to third quartile and has a line at the median value. Whiskers extend to the 2 and 98 percentile values, with horizontal ticks at the 9 and 91 percentile values. In several cases, there is very little spread in the running times, so the box-and-whisker plots as a horizontal bar. The vertical axis is linear in run time; a baseline at zero time appears at the bottom. Run time can be compared for different input sequences as well as between sorting implementations.

Quickselect is sometimes slightly slower on already-sorted inputs. This was an intentional redesign; Bentley & McIlroy's qsort was optimized for already-sorted inputs (using the middle element for a pivot derived from a single sample, first, middle and last element samples for median-of-3 pivot selection) at the expense of other input sequences. Quickselect processes already-sorted inputs slightly slower in return for improved general performance and greatly improved performance for reverse-sorted and organ-pipe input sequences.

Quickselect is much better at most sizes and data types for decreasing input sequences (sampling unaffected by pivot movement; see Figures 2 & 4). Comparison count is much lower for quickselect.

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

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

Quickselect is significantly better (∼30% less run-time) for sawtooth inputs (improvements in median-of-3, swapping). However, comparison count is marginally higher for quickselect.

Quickselect is significantly better (∼25% less run-time) 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 better than Bentley & McIlroy's qsort (< 50% run-time) for all-equal inputs (improvements in median-of-3, swapping). However, comparison count is marginally higher for quickselect when sorting all-equal inputs. The higher comparison count is a result of the larger number of array elements sampled for pivot selection by remedian of samples. It is more than compensated for by the improvements in median-of-3 and swapping. Performance of quickselect is also superior for input sequences with many (but not all) equal elements.

Quickselect is usually comparable on median-of-3-killer inputs. The comparison count is usually lower for quickselect.

No discernible effects because of pivot movement (see Figures 2 & 4).

Quickselect tends to run faster for large data types (improvements in swapping; reduced comparison count). Quickselect does not intentionally alter basic performance with data type size.

Performance improves relative to Bentley & McIlroy qsort with larger array sizes because of slower (or no) growth in scaled comparisons (improved pivot rank through use of remedian-of-samples), as shown in Figure 6.

Quickselect is much better (non-quadratic) vs. McIlroy's antiqsort adversary and other adverse input sequences (because of the break-glass mechanism and guaranteed-rank-range pivot selection). See Figure 9, which shows performance of original and modified Bentley & McIlroy qsort, quickselect, a dual-pivot sort using adaptive sampling for pivot selection, and a modified introsort vs. McIlroy's adversary. Bentley & McIlroy's qsort quickly becomes quadratic. The higher cutoff values for median-of-3 and ninther pivot selection in the modified version result in higher comparison counts for highly adverse input sequences at small array sizes, but lower comparison counts for moderate and large arrays. Introsort's recursion depth limit and use of heapsort when that limit is reached results in large-array comparison count which is heapsort plus $O(N)$ times the recursion depth limit factor. The implementation of heapsort used with the tested implementation of introsort uses about $1.87N \log_2 N$ comparisons for an array of 16 Mi elements; introsort with a recursion depth limit of $2 \log_2 N$ uses about $3.87N \log_2 N$ comparisons to sort that size array. The number of swaps is dominated by heapsort, and approaches $1N \log_2 N$ for large arrays. Some details

are dependent on the specific implementation of heapsort (or other "stopper" algorithm), but for a given recursion depth limit $k \log_2 N$, adverse input will result in at least $kN \log_2 N$ comparisons. Introsort typically uses $k = 2$ for the recursion depth limit, resulting in a minimum asymptotic complexity of $2N \log_2 N$ (typically considerably higher) for adverse input sequences. The reduced number of comparisons and swaps used by quickselect results in significantly lower adverse input run time (locality of access, poor in heapsort, is also a factor). For quickselect configured with a REPIVOT_FACTOR of 9 or lower, equation (4) gives an expected asymptotic complexity of $\approx 1.987N \log_2 N$ or lower ($\approx 1.84N \log_2 N$ at a REPIVOT_FACTOR of 8). Typical configuration therefore provides a maximum asymptotic complexity for quickselect lower than the minimum asymptotic complexity for introsort, for adverse inputs. Adaptive sampling for pivot selection with a dual-pivot sort clearly shows quadratic performance; adaptive sampling is not sufficient to prevent quadratic behavior, although performance would be much worse with a fixed sample size such as is typically used in dual-pivot implementations.

And of course quickselect can be used for selection (in linear time for a single order statistic), while other sorting implementations considered cannot. As noted in the introduction, a full sort is an extraordinarily expensive way to obtain an order statistic. On random inputs, quickselect finds medians using about $2.08N$ comparisons for large arrays. Comparison counts for median-of-3-killer median selection are also slightly higher than $2N$. Simple cases (reversed, all-equal, sawtooth) take slightly more than $1N$ comparisons to find the median. The most difficult structured input sequence is organ-pipe, taking about $2.31N$ comparisons to find the median.

17 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/`.

18 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.

The principle compute-bound task in quicksort is partitioning, and the primary partitioning mechanism could be partially parallelized; the left and right scanning could take place in parallel, with some synchronization mechanism for continuation when there is a greater-than element on the left and a less-than element on the right.

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

19 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 sampling, an extended method of pivot element selection which benefits large problem size, ternary median-of-3, more versatile swapping, and a recovery mechanism to prevent quadratic worst-case behavior have been combined with efficient element and 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, 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. Introsort's mechanism for avoiding quadratic behavior could also be applied to multi-pivot sorting, although the recursion depth limit would be calculated from a different logarithm base. 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, but not for non-trivial types nor for very large arrays, unless the final insertion sort is replaced by smaller insertion sorts. Finally, the break-glass mechanism for avoiding quadratic behavior applies equally well for

sorting and for selection, and provides better performance for adverse inputs than introsort's recursion-depth limit, at the expense of a tiny increase – typically a fraction of 1% – in cost for sorting non-adverse input sequences.

20 Acknowledgments

The published papers by Lent & Mahmoud [4] and Bentley & McIlroy [9] have been indispensible in development of the function described in this paper. McIlroy's antiqsort [11] provided a useful tool for rigorous testing of sorting functions. Papers by Blum, Floyd, Pratt, Rivest, & Tarjan [2] and by Rousseeuw & Bassett [15] were essential for developing the median-of-medians and remedian of samples pivot selection implementations. Analysis of optimal sampling by McGeoch & Tygar [16] and by Martínez & Roura [17] 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://wwwb-front3.us.archive.org/web/20160204001614/http://www.programmingwala.com/2013/07/median-n-numbers-c-program.html`. [4 February 2016].

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

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

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

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

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

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

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

[15] 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.

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

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

[18] 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.

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

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

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