SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 0000; **00**:1–24
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe

Sorting and selection without tears

Bruce Lilly*

Retired, Upton MA 01568

SUMMARY

Sorting and selection are fundamental algorithms, described and analyzed in detail in the literature. A 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; even the best 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 an 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 selection function can be used for sorting as well as selection, and has performance for sorting comparable to or better than the well-known quicksort implementation; in particular, it cannot be driven to quadratic sorting behavior. Analysis techniques and algorithm design provided insights which may be applicable to other software projects.

Received ...

KEY WORDS: Quicksort; Sorting algorithms; Selection; Testing; Performance tuning; Algorithm design and implementation

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 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 qsort-like interface, 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

*Correspondence to: E-mail: bruce.lilly@gmail.com

zero size, a full sort is performed. Therefore, the implementation is suitable for both selection and sorting.

Insight: Simplify maintenance by appropriately combining functionality.

The algorithm described by Lent & Mahmoud[4] requires that the array of order statistic ranks be sorted; that requirement is relaxed in the implementation described in this paper.

As the basic operations for multiple quickselect are the same as for quicksort, a well-known and widely-used implementation of quicksort was chosen as the starting point for this implementation of multiple quickselect. That 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.

Both quicksort and quickselect use the same basic operations: sample the input array to select a pivot element, 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. It is similar to that described by Bentley & McIlroy[9] with several additional test sequences:

- inverse organ-pipe (... 2 1 0 1 2 ...)
- median-of-3-killer[10]
- 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 restricted to the range [0,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. The constructed sequence can be saved to a file for reuse.

Data types generated were similar to those described by Bentley & McIlroy[9] . Note that 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 counts, including a breakdown of less-than, equal-to and greater-than results, and partition size analysis.

DOI: 10.1002/spe

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. The 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 function useful for multiple order statistic selection as well as for 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.

The stack which is required to support selection (while maintaining a qsort-compatible interface) has the beneficial side-effect of preventing crashes caused by program stack overflow.

Quadratic behavior in quicksort 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 $O(N)$, whereas Tukey's ninther has only $O(1)$ complexity. And weak guarantees of pivot rank are not sufficient to prevent quadratic behavior. 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, 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 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

Prepared using speauth.cls
DOI: 10.1002/spe

pivot choice results in only an $O(1)$ reduction in problem size. Note that the break-glass mechanism differs from Musser's introsort[10] in two important ways:

- break-glass is used when any partition is exceptionally poor; introsort waits until recursion depth grows
- 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 to find the median of the medians. When the break-glass mechanism is used, the result is up to three regions to be processed; the small region produced by the unfavorable partition, and the two more balanced regions resulting from repartitioning the larger part of the original partition using a pivot which has a guaranteed range of rank.

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 is $1.4 + 0.2kN$ †.

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 approximately $1 + 0.333kN$. 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. Paterson[1] in a review article summarized results indicating an upper bound on complexity of median finding which is indeed less than 3. A simplification of median-of-medians ignores "leftover" elements if the array size is not an exact multiple of the set size (3), with a slight increase in the range of pivot rank.

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 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

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

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

Prepared using speauth.cls
DOI: 10.1002/spe

Or in words, sorting proceeds by pivot selection and partitioning, followed by recursion on the two regions resulting from the partition.

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

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

For large N, $N \approx n$ and

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

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 $\approx 1.089N\log_2 N$. A much more lopsided partition, with $^{15}/_{16}$ of the elements in one region produces a complexity of $\approx 2.965N\log_2 N$.†

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 $a = 1$. Median-of-medians pivot selection using sets of 3 elements costs $1 + 0.333kN$ 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 $d > 14$.

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.

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;

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

Prepared using speauth.cls
DOI: 10.1002/spe

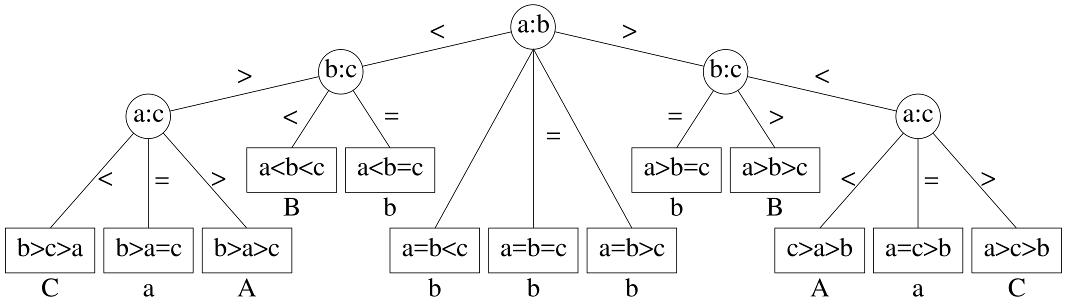

Figure 1. Ternary median-of-3 decision tree

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*.

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, settling on 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 were investigated in the present work. However, the split-end scheme described by Bentley & McIlroy seemed to work 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 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.

Insight: Explore options; test, measure, and compare

8. TERNARY MEDIAN OF THREE

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

DOI: 10.1002/spe

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. Note that as 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 1⁄3 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.

Note that 3 of the 13 conditions are handled with a single comparison, that 4 result from exactly two comparisons, and that 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,[14] 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[14] 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. Sample size may be increased based on array size by increasing the power of 3 each time the array size increases by some factor B, where $B \geq 3$. The sample size is then $3^{\lfloor \log_B N \rfloor}$ for an array of N elements. For $B < 7$, the cumulative number of comparisons used for pivot selection when sorting an array of N elements (assuming ideal partitions) is greater than for ninther. When $7 \leq B \leq 9$, the cumulative number of comparisons is greater than for ninther at some array sizes, but is generally

DOI: 10.1002/spe

smaller than for ninther. The cumulative number of comparisons for pivot selection is never greater than for ninther when $B \geq 10$.

The reduction in the cumulative number of comparisons while increasing the number of samples with array size so as to provide an improved pivot rank guarantee is achieved by reducing the number of samples for small arrays. For a sampling base B, equivalence in number of samples to ninther occurs for array size $N \geq B^2$. For example, for $B = 8$, 9 samples (the same as used by ninther) are used for $N = 64$ to $N = 511$ (vs. above a threshold of 40 for Bentley & McIlroy's qsort). This has the effect of providing a lesser pivot rank guarantee for small arrays and a better guarantee for large arrays. The statistical efficiency is also improved with the larger number of samples used for large arrays.

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. 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, remedian of samples is inherently adaptive, using more samples as the array size increases, without separate code or size cutoff parameters. And the data movement which may take place during in-place remedian reduces data entropy, 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. No attempt is made to eliminate this redundancy, in part because the cost of doing so would outweigh the benefit, but also because it would tend to introduce disorder into already-sorted inputs.

10. SAMPLING IN BENTLEY & MCILROY'S QSORT

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[†]. 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]

[†]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

DOI: 10.1002/spe

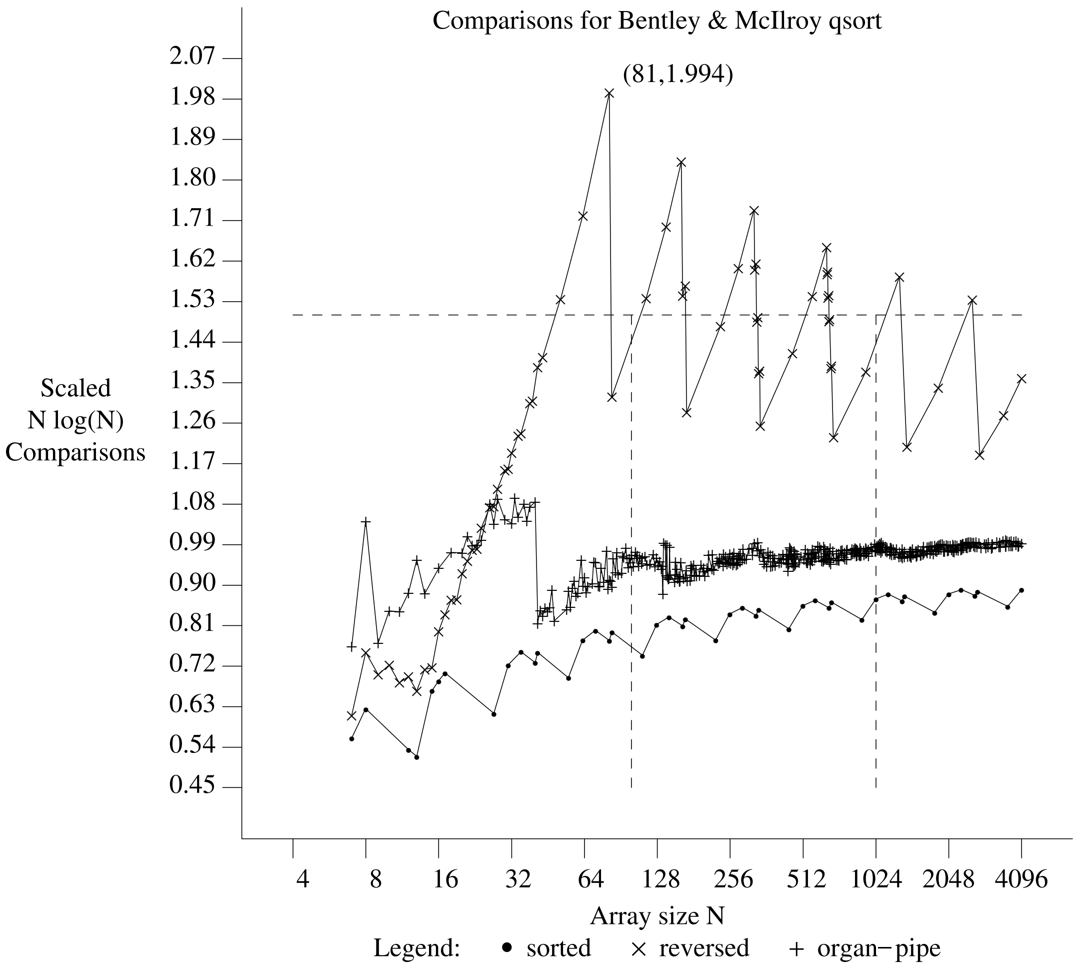

Figure 2. Comparisons for Bentley & McIlroy qsort

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 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 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 array element in computing median-of-3 pivot elements which is responsible for the peaks in the comparison counts evident in Figure 2. Note that 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 sub-array position for using median-of-3 pivot selection. Such an example is shown in Figure 3. Note that 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

Prepared using speauth.cls
DOI: 10.1002/spe

initial reverse-sorted array

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

swap pivot to start

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

partition

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

...

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

swap blocks (pivot) to canonicalize

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

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

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. There are actually two interacting factors responsible for the poor performance:

1. The inital swap of the pivot element to the first position, which moves an extreme-valued element initially there to near the middle of the array, where it ends up at the left-hand side of the greater-than 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 the less-than 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. On 32-bit architectures such as were available in 1992, doubles were of a different size than long integers; therefore the PVINIT macro was ineffective for doubles, which were found by Bentley & McIlroy to exhibit the worst performance for reverse-sorted inputs. On 64-bit architectures commonly available in 2016, sorting doubles does not exhibit the same behavior noted by Bentley & McIlroy; the behavior instead shows up for other data types, such as the integers used to generate Figure 2.

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 deferment 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.

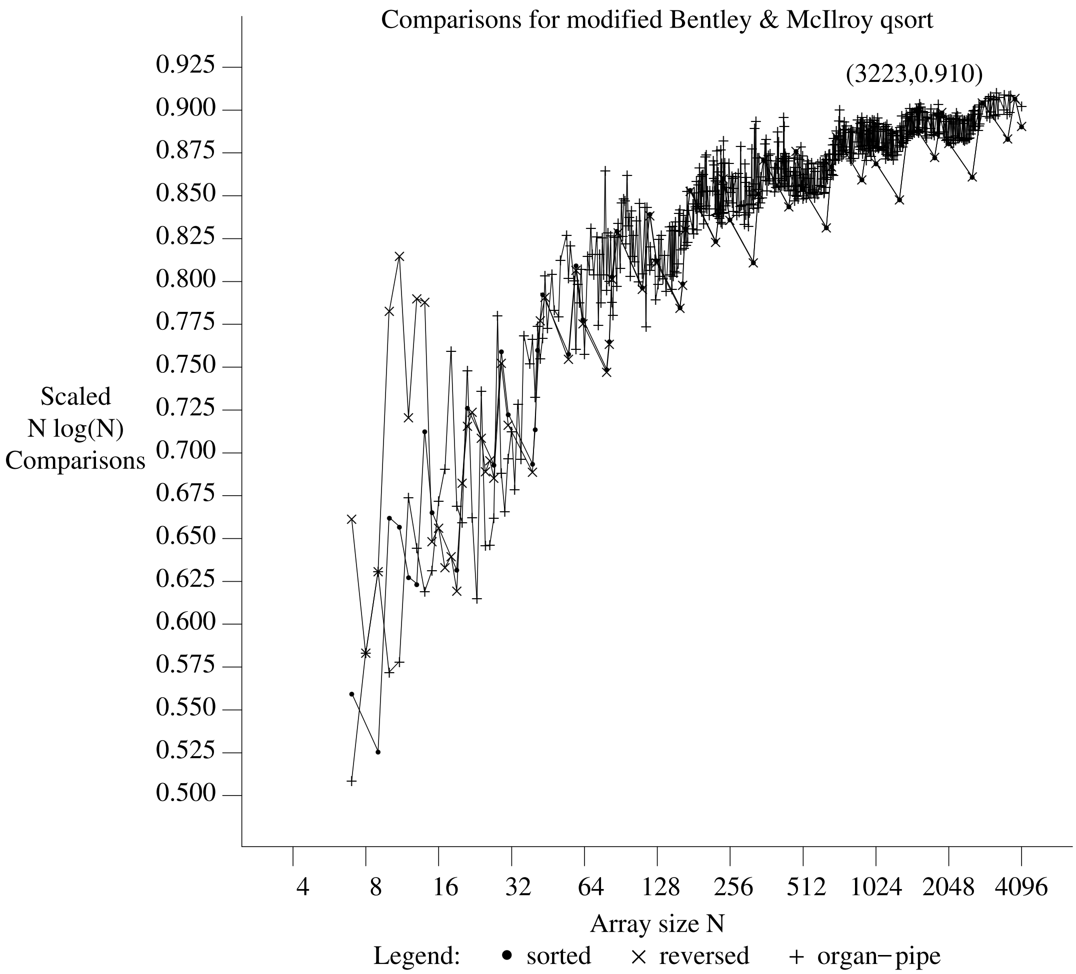

Figure 4. Comparisons for modified Bentley & McIlroy qsort

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 method of sampling of the input array which is 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). 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.

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 input sequences, as shown in Figure 4. Note that 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

DOI: 10.1002/spe

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 [13]

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, 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 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, 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 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

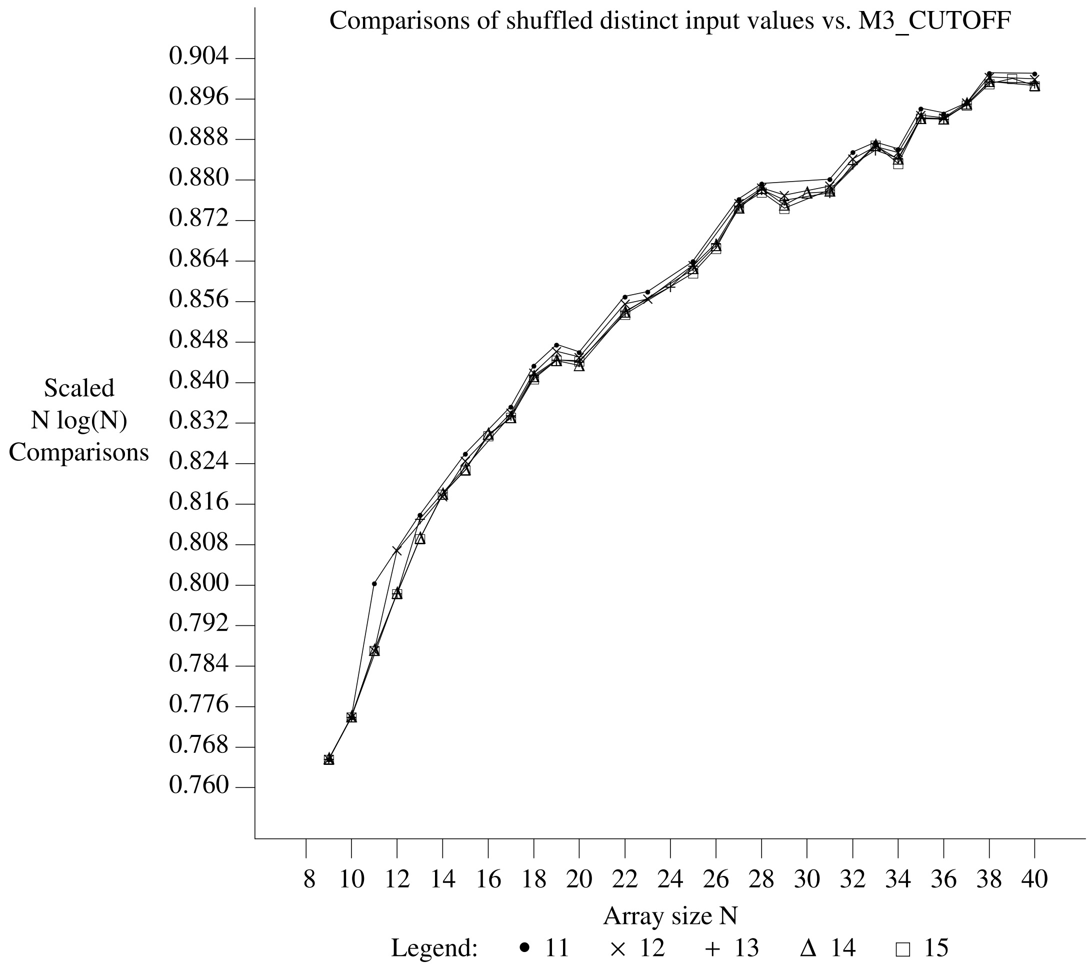

Figure 5. Scaled comparisons for shuffled input vs. M3_CUTOFF

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 then with Bentley & McIlroy's qsort, however performance for random input sequences suffers somewhat because of the added comparisons.

Insight: Analyze algorithms (in addition to measuring heuristic 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. To evaluate the tradeoff, a cutoff parameter, called M3_CUTOFF, was established. For arrays with fewer than M3_CUTOFF elements, the ¼ position element is used as a single sample pivot. At size M3_CUTOFF and above, but at sizes below which more than three samples are used, the median of elements at the ¼, ½, and ¾ position are used, which will produce the ideal partitioning for both organ-pipe and sorted input sequences. The effect of the choice of value for M3_CUTOFF was tested on shuffled sequences of distinct valued array elements at array sizes from 9 through 40 and at much larger sizes, over many sequences, plotting the median comparison count scaled to $N \log_2 N$ vs. array size at several values of M3_CUTOFF. Figures 5 and 6 show an example. 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 15 is provides reasonable performance for random input sequences, and penalizes organ-pipe and sorted input sequences by less than 15%.

DOI: 10.1002/spe

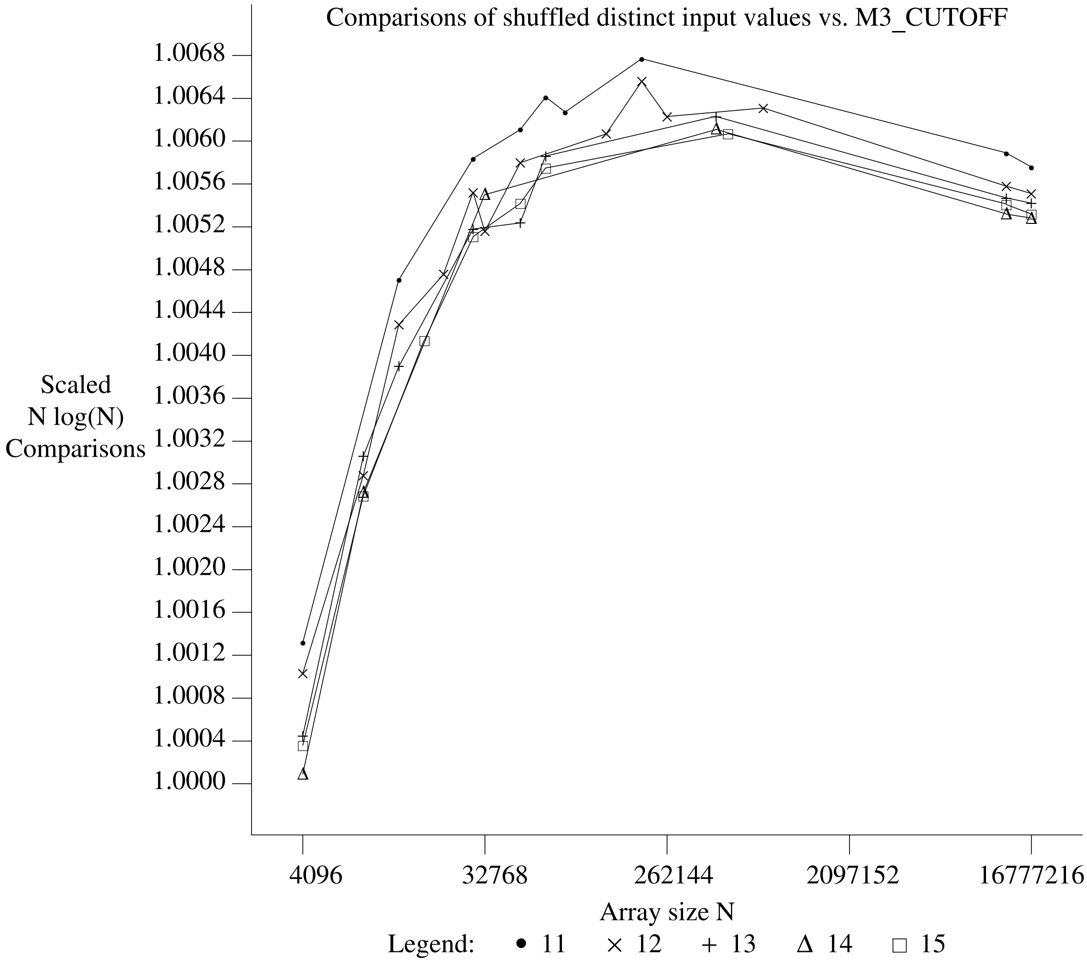

Figure 6. Scaled comparisons for shuffled input vs. M3_CUTOFF

Note that the improvement in performance that results from the improved choice 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 15. 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.

The obvious 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.

Including samples at ¼ and ¾ positions among a number of samples which is a power of three implies that the denominator used should be a multiple of 12, the product of four and three. That gives at least two plausible choices for 9 reasonably well-spaced samples which include $^3/_{12}$ (i.e. ¼), $^6/_{12}$, and $^9/_{12}$. The numerators of the fractions of those sets are 2, 3, 4, 5, 6, 7, 8, 9, 10 for one choice, and 1, 2, 3, 5, 6, 7, 9, 10, 11 for the other. Both choices are comprised of three rows of three columns of elements, with uniform spacing between elements in each column and between each row. In the first choice, the sequence wraps around the end of the first row, whereas the second choice has a larger gap at row boundaries. When the first set is used for an organ-pipe sequence, a problem arises: when the samples are grouped into three rows of three columns, the ¼, ½, and ¾

Prepared using speauth.cls
DOI: 10.1002/spe

position elements are all in the same column. For organ-pipe inputs, the ¼ and ¾ position elements are medians, and the column containing them will have one of them as the column median. But the other two columns also each contain two elements with like values, both values greater than the values of the ¼ and ¾ elements (i.e. the median of the middle column, containing the values from the ¼ and ¾ array positions, is smaller than the medians of the other columns). Therefore one of those other values will become the value selected for the pivot by Tukey's ninther (or equivalently, by remedian). Having all three "good" elements in one column is a disaster! The alternate choice does not exhibit the same problem; the ¼, ½, and ¾ position elements are in different columns (as well as different rows), and the ninther produced for either sorted or organ-pipe inputs will be the median element.

An adaptive sampling method was adopted for the present work:

- The element at the ¼ position is used for arrays of size below M3_CUTOFF (15). For arrays of size 2 or 3, that degenerates to the first element; that in turn facilitates partitioning, as it obviates swapping to the first position.
- Three elements chosen close to ¼, ½, and ¾ positions are sampled for arrays starting at size M3_CUTOFF. These elements provide optimal partitioning for already-sorted and organ-pipe input sequences, and the choice of cutoff provides optimum average performance for random 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).

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 sampling accounts for much of the improved sorting performance of the present work for non-adversarial input sequences at moderate array size.
- 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 adds some code, however it is essential for selection, which would 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.
- 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 additional code, but without this mechanism, quadratic sorting behavior is unavoidable.

DOI: 10.1002/spe

- 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, ninther, remedian pivot selection).

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 selection/sorting function incorporates components based on analysis, algorithm design, and testing described earlier in this paper.

- An iterative implementation of multiple quickselect with an internal stack and *qsort*-like interface. When performing (multiple) selection, regions not containing desired order statistic ranks are ignored. The internal stack, with regions sorted so as to process the smallest region first, also prevents overrunning the stack when sorting, and avoids the overhead of recursive function calls.

Insight: Prepare for worst-case behavior; it will happen.

- Partitioning based on the split-end method used by Bentley & McIlroy, with swapping of the pivot element deferred to prevent introduction of disorder.
- A break-glass mechanism to obtain an improved pivot in the event of an extremely lopsided partition, 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.
- 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. Cumulative comparison count when sorting large arrays may be made smaller than for continued use of Tukey's ninther by appropriate tuning. 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.
- Insertion sort for sorting very small arrays, as in Bentley & McIlroy's qsort implementation.

DOI: 10.1002/spe

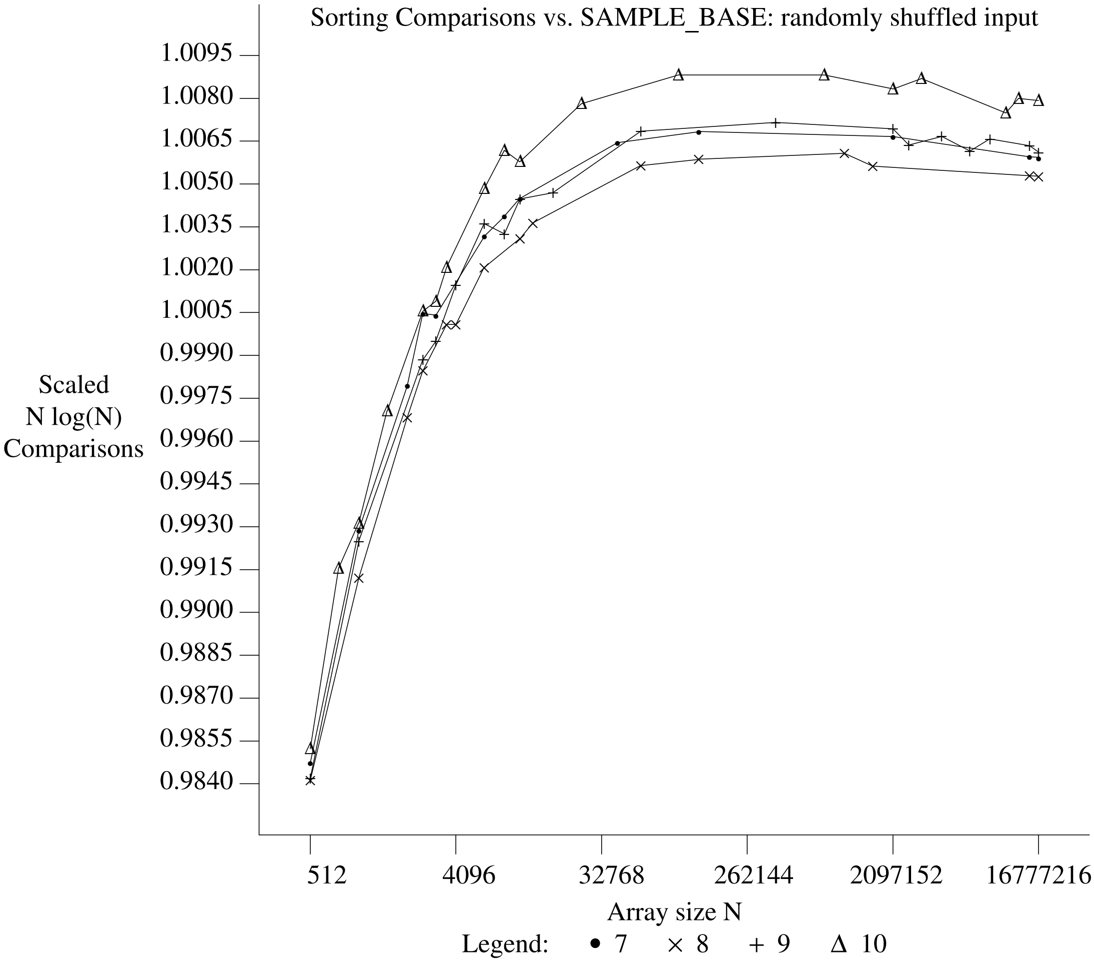

Figure 7. Sorting comparison count vs. randomly shuffled input

15. PERFORMANCE TUNING

Choice of configuration for median-of-3 cutoff has already been discussed. The base used for sampling for remedian-of-samples pivot selection affects the cumulative number of comparisons and the range of guaranteed rank. This parameter, called SAMPLE_BASE, is a tunable parameter. SAMPLE_BASE was varied over the range 7 through 10 and quickselect performance was tested against randomly shuffled distinct inputs and McIlroy's adversary. See Figure 7. A small SAMPLE_BASE yields more samples and a tighter rank range; the larger number of samples increases the comparison count cost. If SAMPLE_BASE is large, there are fewer samples and the possibility of very lopsided partitions. A SAMPLE_BASE value of 8 appears to be optimum, yielding a lower cumulative comparison count and tighter rank range than Tukey's ninther for arrays with more than 512 elements.

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 repivot, 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. Unlike SAMPLE_BASE, the repivoting threshold is not a clear case of finding an optimum; there isn't one. Repivoting is a tradeoff between expending

DOI: 10.1002/spe

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 two parameters, 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 size of the small region is at least as large as REPIVOT_FACTOR. At one extreme, with REPIVOT_FACTOR set to 14 and REPIVOT_CUTOFF set to 19, large arrays of random sequences sort using about $1.0073N\log_2 N$ comparisons, and the worst-case performance against sequences generated by McIlroy's adversary is $1.66495N\log_2 N$ at an array of size 91. At the opposite extreme, if repivoting is effectively disabled, the random sequence complexity is lowered to about $1.0037N\log_2 N$ 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 quickselect cannot overrun its internal stack. A reasonable compromise is a REPIVOT_FACTOR of 16 and REPIVOT_CUTOFF of 27, which provides random array performance of $1.0059N\log_2 N$ and worst-case adversarial performance of $1.66495N\log_2 N$ comparisons at an array of size 91. That limits worst-case performance to less than a factor of two worse than expected performance, at a cost of $\sim 0.22\%$, 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 9. 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 8 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.

Figure 9 is a graph of comparison counts for sorting randomly shuffled distinct input values vs. array size.

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
- median-of-3 killer sequence

DOI: 10.1002/spe

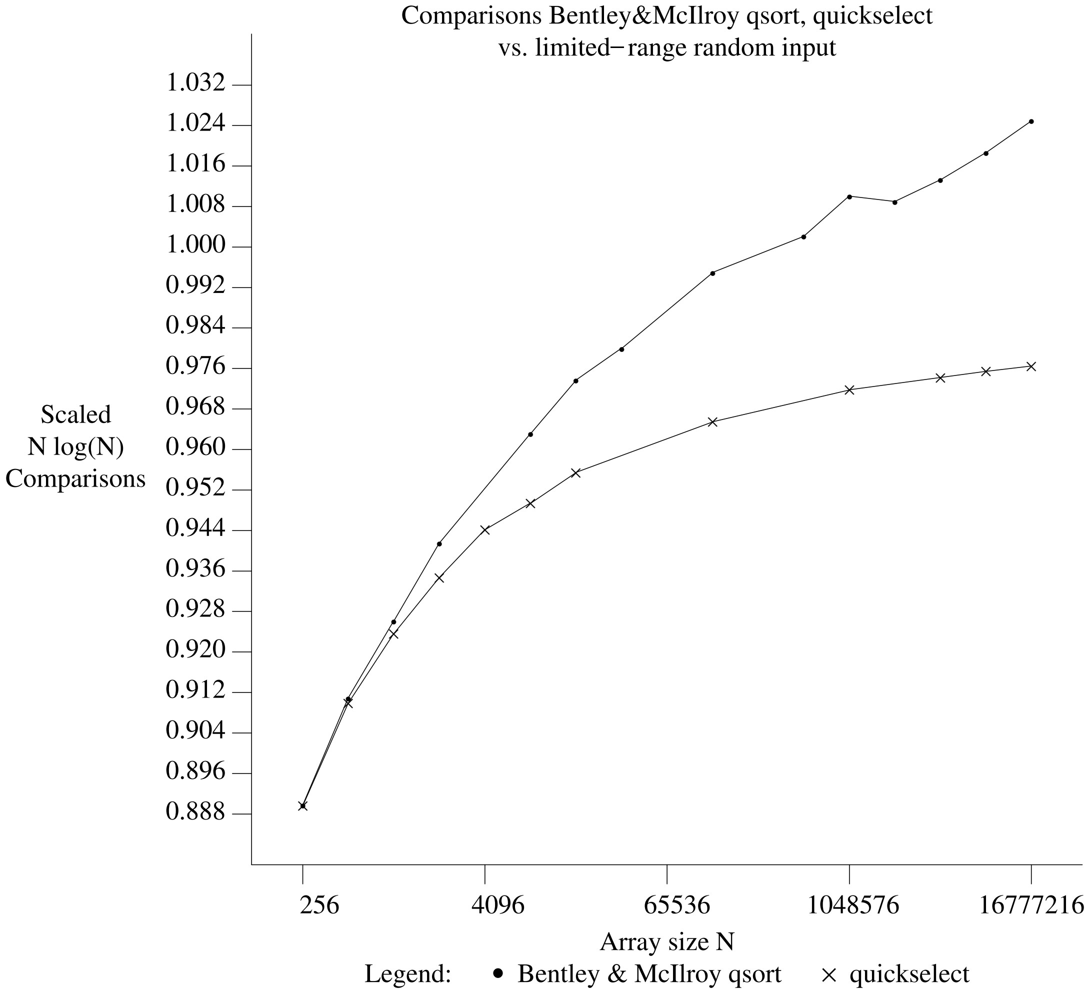

Figure 8. Comparisons for Bentley & McIlroy qsort and quickselect, random input

- 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 Figure 10. 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 11. 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 input sequences.

DOI: 10.1002/spe

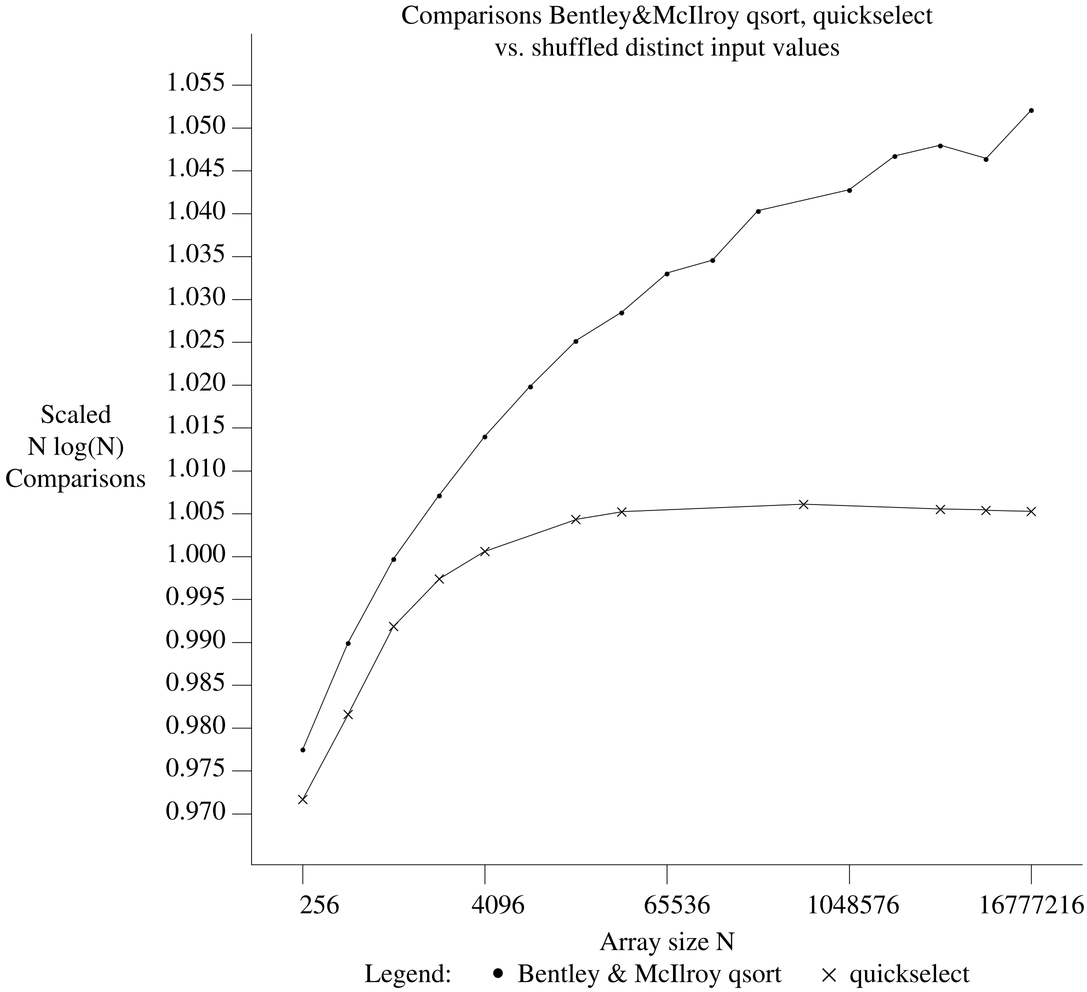

Figure 9. Comparisons for Bentley & McIlroy qsort and quickselect, shuffled input

Sequence	Bentley & McIlroy qsort	quickselect
sorted	0.90803	0.92442
reversed	1.16519	0.92442
shuffled	1.03488	1.00578
organ	1.01805	0.94880
sawtooth	0.09032	0.09046
binary	0.09032	0.09050
equal	0.06021	0.06028
m3k	0.94683	0.94158
random	1.03548	1.00578
limited	0.99056	0.96432

Figure 10. Scaled $N \log_2 N$ Comparisons for Bentley & McIlroy qsort and quickselect 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.

DOI: 10.1002/spe

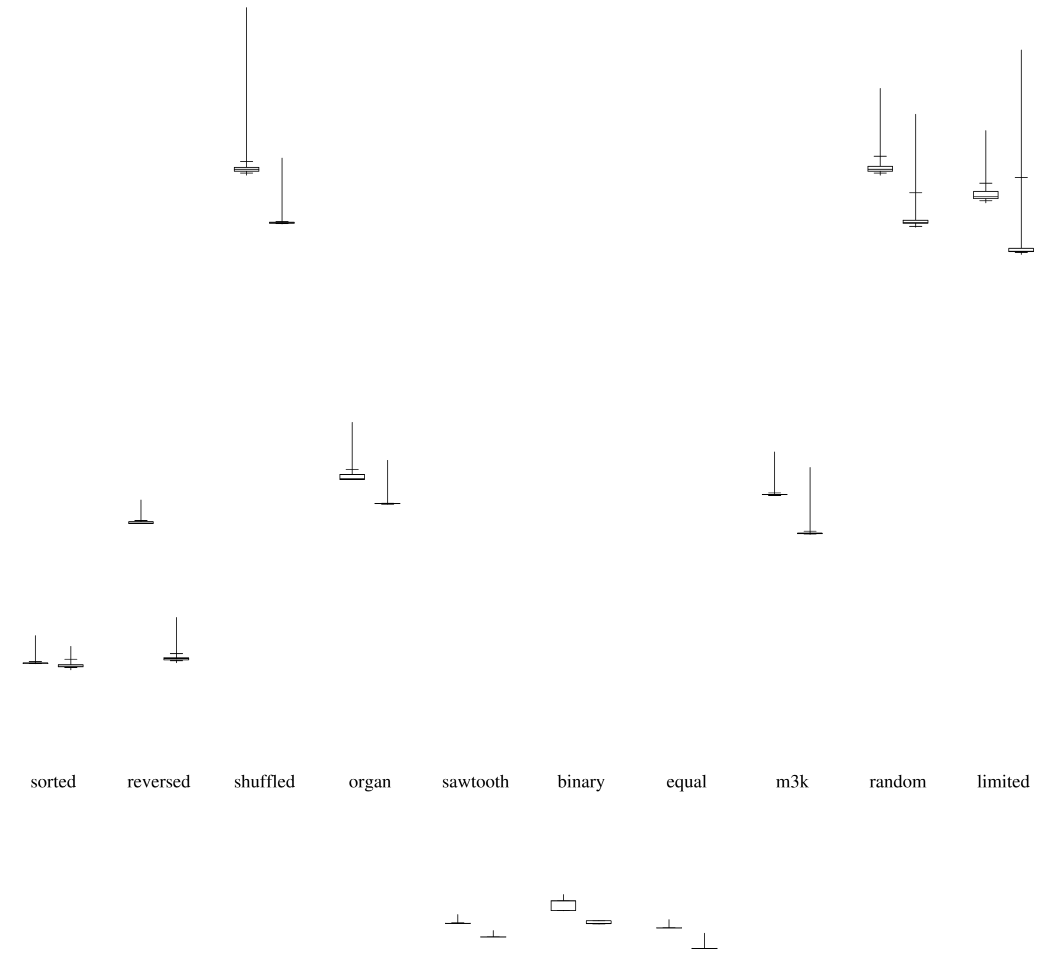

Figure 11. Timing for Bentley & McIlroy qsort and quickselect input sequences

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 faster 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.

DOI: 10.1002/spe

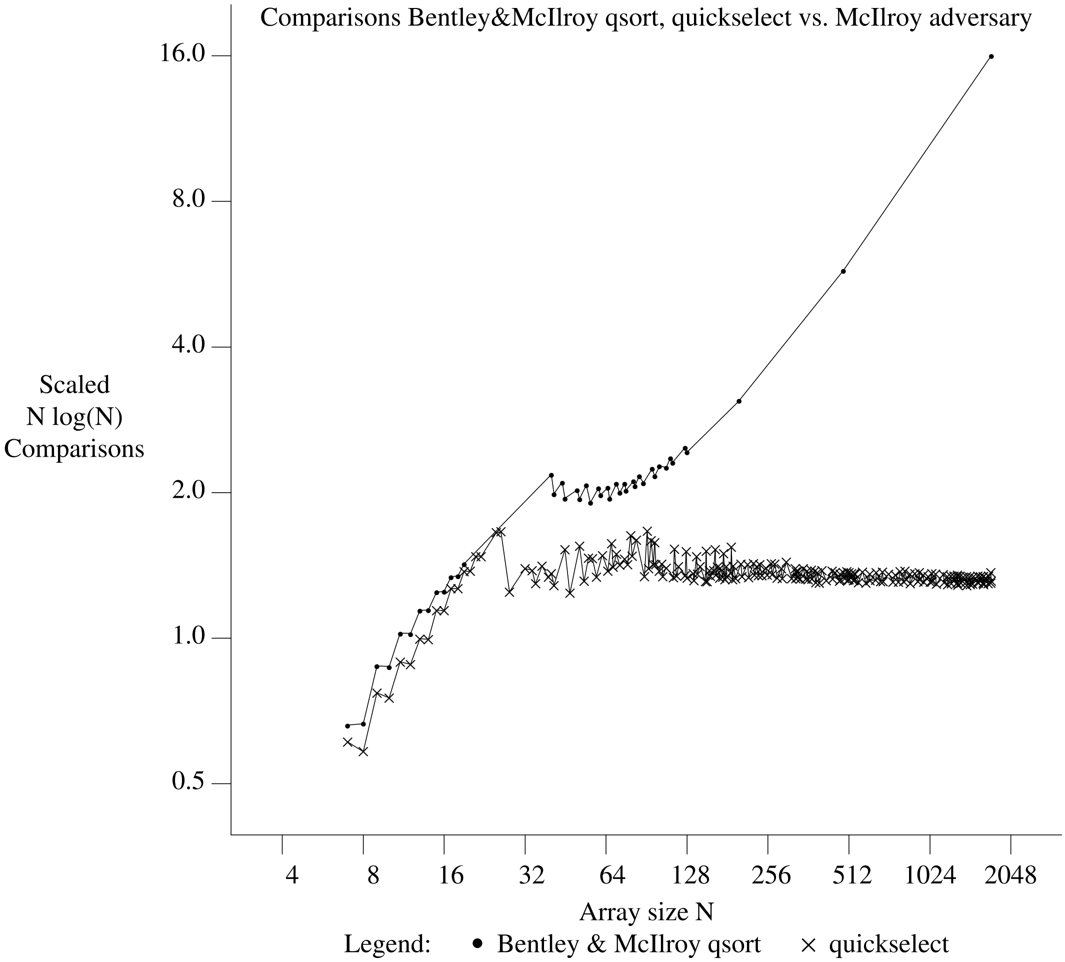

Figure 12. Comparisons for Bentley & McIlroy qsort and quickselect vs. McIlroy adversary

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 9.

Quickselect is much better (non-quadratic) vs. McIlroy's antiqsort adversary (because of the break-glass mechanism and guaranteed-rank-range pivot selection). See Figure 12. The absolute worst-case comparison complexity against adverse inputs is $1.665N \log_2 N$ comparisons at an array of size 91 elements. Bentley & McIlroy's qsort quickly becomes quadratic.

And of course quickselect can be used for selection (in linear time for a single order statistic), while Bentley & McIlroy qsort 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 (sorted, 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. McIlroy's adversary can push the median-finding comparison count up to around $4.8N$ at large N, with an absolute maximum of $7.50633N$ at array size 79.

DOI: 10.1002/spe

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 provided that suitable locking and synchronization mechanisms are applied to the list of sub-array regions to be processed.

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 function has been designed which provides in-place sorting and order statistic selection, including 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, and a recovery mechanism to prevent quadratic worst-case behavior have been combined with efficient element and block swapping and efficient partitioning.

20. ACKNOWLEDGEMNTS

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 the implementation. Papers by Blum, Floyd, Pratt, Rivest, & Tarjan[2] and by Rousseeuw & Bassett[14] were essential for developing the median-of-medians and remedian of samples pivot selection implementations.

REFERENCES

1. Paterson, M. "Progress in Selection", DOI: 10.1007/3-540-61422-2_146
2. Blum, M. et al, "Expected Time Bounds for Selection", STAN-CS-73-349, Leland Stanford Junior University, Stanford, CA
3. Knuth, D., The Art of Computer Programming, Volume 3: Sorting and Searching, Third Edition. Addison-Wesley, 1997. ISBN 0-201-89685-0
4. Lent, J. & Mahmoud, H., "Average-case analysis of multiple Quickselect: An algorithm for finding order statistics", Statistics & Probability Letters 28 (1996) 299-310
5. `https://web.archive.org/web/20150507020959/http://en.wikiversity.org/wiki/C_Source_Code/Find_the_median_and_mean`
6. `https://web.archive.org/web/20150514144742/http://www.sanfoundry.com/cpp-program-compute-median-numbers`
7. `https://web.archive.org/web/20160204001614/http://www.programmingwala.com/2013/07/median-n-numbers-c-program.html`
8. `https://web.archive.org/web/20150626164209/https://www.easycalculation.com/code-c-program-mean-median-mode.html`
9. Bentley, J.L. & McIlroy, M.D. "Engineering a Sort Function" SOFTWARE-PRACTICE AND EXPERIENCE, VOL. 23(11), 1249-1265 (NOVEMBER 1993)
10. Musser, D.R., "Introspective Sorting and Selection Algorithms", Software-Practice and Experience, (8): 983-993 (1997)

11. McIlroy, M.D. ”A Killer Adversary for Quicksort”, SOFTWARE–PRACTICE AND EXPERIENCE, VOL. 29(0), 1–4 (0 1999)
12. http://cvsweb.netbsd.org/bsdweb.cgi/src/lib/libc/stdlib/qsort.c?rev=1.22
13. The Times They Are A-Changin'; — The Official Bob Dylan Site http://bobdylan.com/songs/times-they-are-changin/
14. Rousseeuw, P.J. & Bassett G.W., Jr., ”The Remedian: A Robust Averaging Method for Large Data Sets”, Journal of the American Statistical Association, March 1990, Vol. 85, No. 409

DOI: 10.1002/spe