Introspective Sorting and Selection Algorithms

David R. Musser*
Computer Science Department
Rensselaer Polytechnic Institute, Troy, NY 12180
musser@cs.rpi.edu

Abstract

Quicksort is the preferred in-place sorting algorithm in many contexts, since its average computing time on uniformly distributed inputs is $\Theta(N \log N)$ and it is in fact faster than most other sorting algorithms on most inputs. Its drawback is that its worst-case time bound is $\Theta(N^2)$. Previous attempts to protect against the worst case by improving the way quicksort chooses pivot elements for partitioning have increased the average computing time too much—one might as well use heapsort, which has a $\Theta(N \log N)$ worst-case time bound but is on the average 2 to 5 times slower than quicksort. A similar dilemma exists with selection algorithms (for finding the i-th largest element) based on partitioning. This paper describes a simple solution to this dilemma: limit the depth of partitioning, and for subproblems that exceed the limit switch to another algorithm with a better worst-case bound. Using heapsort as the "stopper" yields a sorting algorithm that is just as fast as quicksort in the average case but also has an $\Theta(N \log N)$ worst case time bound. For selection, a hybrid of Hoare's FIND algorithm, which is linear on average but quadratic in the worst case, and the Blum-Floyd-Pratt-Rivest-Tarjan algorithm is as fast as Hoare's algorithm in practice, yet has a linear worst-case time bound. Also discussed are issues of implementing the new algorithms as generic algorithms and accurately measuring their performance in the framework of the C++ Standard Template Library.

KEY WORDS Quicksort Heapsort Sorting algorithms Introspective algorithms Hybrid algorithms Generic algorithms STL

Introduction

Among sorting algorithms with $O(N \log N)$ average computing time, median-of-3 quicksort [1, 2] is considered to be a good choice in most contexts. It sorts in place, except for $\Theta(\log N)$ stack space, and is usually faster than other in-place algorithms such as heapsort [3], mainly because it does substantially fewer data assignments and other operations. These characteristics make median-of-3 quicksort a good candidate for a standard library sorting routine, and it is in fact used as such in the C++ Standard Template Library (STL) [4, 5, 6] as well as in older C libraries—it is the algorithm most commonly used for *qsort*, for example. However, its worst-case time is $\Theta(N^2)$, and although the worst-case behavior appears to be highly unlikely, the very existence of input sequences that can cause such bad performance may be a concern in some situations. The best alternative in such situations has been to use heapsort, which has a $\Theta(N \log N)$ worst-case as well as average-case time bound, but doing so results in computing times 2 to 5 times longer than quicksort's time on most inputs. Thus, in order to protect

*This work was partially supported by a grant from IBM Corporation.

completely against deterioration to quadratic time it would seem necessary to pay a substantial time penalty for the great majority of input sequences.

A similar dilemma appears with selection algorithms for finding the i-th smallest element of a sequence. Hoare's algorithm [7] based on partitioning has a linear bound in the average case but is quadratic in the worst case. The Blum-Floyd-Pratt-Rivest-Tarjan linear-time worst-case algorithm [8] is much slower on the average than Hoare's algorithm. In this paper, we concentrate on the sorting problem and return to the selection problem only briefly in a later section.

The next section presents a class of arbitrarily long input sequences that do in fact cause median-of-3 quicksort to take quadratic time (call these sequences "median-of-3 killers"). It is pointed out that such input sequences might not be as rare in practice as a probabilistic argument based on uniform distribution of permutations would suggest.

The third section then describes *introsort* (for "*introspective sort*"), a new, hybrid sorting algorithm that behaves almost exactly like median-of-3 quicksort for most inputs (and is just as fast) but which is capable of detecting when partitioning is tending toward quadratic behavior. By switching to heapsort in those situations, introsort achieves the same $O(N \log N)$ time bound as heapsort but is almost always faster than just using heapsort in the first place. On a median-of-3 killer sequence of length 100,000, for example, introsort switches to heapsort after 32 partitions and has a total running time less than 1/200-th of that of median-of-3 quicksort.[1] In this case it would be somewhat faster to use heapsort in the first place, but for almost all randomly-chosen integer sequences introsort is faster than heapsort by a factor of between 2 and 5.

Ideally, quicksort partitions sequences of size N into sequences of size approximately $N/2$, those sequences in sequences of size $N/4$, and so on, implicitly producing a tree of subproblems whose depth[2] is approximately $\log_2 N$. The quadratic-time problem arises on sequences that cause many unequal partitions, resulting in the subproblem tree depth growing linearly rather than logarithmically. Introsort avoids the problem by putting a logarithmic bound on the depth and switching to heapsort whenever the bound is reached.

Another way of obtaining an $O(N \log N)$ worst-case time bound with an algorithm based on quicksort is to install a linear-time median-finding algorithm for choosing partition pivots [8]. However the resulting algorithm is useless in practice since the large overhead of the median-finding algorithm makes the overall sorting algorithm much slower than heapsort. This algorithm could however be used as the "stopper" in an introspective algorithm, in place of heapsort. There might be some advantage in this algorithm in terms of commonality of code with the selection algorithms described in a later section.

Instead of finding the median exactly, other less expensive alternatives have been suggested, such as adaptively selecting from larger arrays a larger sample of elements from which to estimate the median. A new version of the C library function *qsort* based on this technique and other improvements [9] outperforms median-of-3 versions in most cases, but still takes $\Theta(N^2)$ time in the worst-case.

Yet another possibility is to use a randomized version of quicksort, in which pivot elements are chosen at random. The worst-case time bound is still $\Theta(N^2)$, but the probability of there

[1] 0.83 seconds on a 70 MHz microSPARC II versus 172 seconds for the generic `sort` algorithm from the Hewlett-Packard (HP) implementation of the C++ Standard Template library, with both algorithms compiled with the Apogee C++ compiler, version 3.0. Performance comparisons that are more architecture- and compiler-independent are given in a later section.

[2] For a purely recursive version of quicksort the subproblem depth would be the same as the recursion depth, but the most efficient versions recurse, directly or using a stack, on only one branch of the subproblem tree and iterate down the other.

being sufficiently many bad partitions to achieve this bound is extremely low. However, choosing a single element of the sequence at random does not produce good enough partitions on the average to compete with median-of-3 choices, and choosing three elements at random requires too much time for random number generation. Introsort beats these randomized variations in practice.

Median-of-3 Killer Sequences

As is well-known, the simplest methods of choosing pivot elements for partitioning, such as always choosing the first element, cause quicksort to deteriorate to quadratic time on already (or nearly) sorted sequences. Since there are many situations in which one applies a sorting algorithm to an already sorted or almost sorted sequence, a more robust way of choosing pivots is needed. While the ideal choice would be the median value, the amount of computation required is too large. Instead, one of the most frequently used methods is to choose the first, middle, and last elements and then choose the median of these three values [2]. This method produces good partitions in most cases, including the sorted or almost sorted cases that cause the simpler pivoting methods to blow up.[3] Nevertheless, there are sequences that can cause median-of-3 quicksort to make many bad partitions and take quadratic time.

For example, let k be any even positive integer, and consider the following permutation of $1, 2, \ldots, 2k$ [10]:

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & \ldots & k-2 & k-1 & k & k+1 & k+2 & k+3 & \ldots & 2k-1 & 2k \\ 1 & k+1 & 3 & k+3 & 5 & \ldots & 2k-3 & k-1 & 2k-1 & 2 & 4 & 6 & \ldots & 2k-2 & 2k \end{pmatrix}$$

Call this permutation K_{2k}; for reasons to be seen, it will also be called a "median-of-3 killer" permutation (or sequence). Assume the three elements chosen by a median-of-3 quicksort algorithm on an array $a[1..2k]$ are $a[1], a[k+1]$, and $a[2k]$. Thus for K_{2k}, the three values chosen are $1, 2$, and $2k$, and the median value chosen as the pivot for partitioning is 2. In the partition, the only elements exchanged are $k+1$ and 2, resulting in

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & \ldots & k-2 & k-1 & k & k+1 & k+2 & k+3 & \ldots & 2k-1 & 2k \\ 1 & 2 & 3 & k+3 & 5 & \ldots & 2k-3 & k-1 & 2k-1 & k+1 & 4 & 6 & \ldots & 2k-2 & 2k \end{pmatrix}$$

where the partition point is after $a[2] = 2$. Now the array $a[3..k]$ now holds a sequence with the structure of K_{2k-2}, as can be seen by subtracting 2 from every element. As this step can be repeated $k/2$ times, we have the following

Theorem 1 *For any even integer N divisible by 4, the permutation K_N causes median-of-3 partitioning to produce a sequence of $N/4$ partitions into subsequences of length* 2 *and* $N-2$, 2 *and* $N-4$, ..., 2 *and* $N/2$.

Corollary 1 *On any sequence permuted from an ordered sequence by K_N, where N is divisible by 4, median-of-3 quicksort takes $\Omega(N^2)$ time.*

Proof: By the theorem, the subproblem tree produced by partitioning K_N has at least $N/4$ levels. Since the total time for all partitions at each level is $\Theta(N)$, the total time for all levels is $\Omega(N^2)$. $\Box$

One might argue that the K_N sequences are not likely to occur in real applications, or at least not nearly so likely as the almost sorted sequences that cause simpler versions of quicksort

[3]This assumes that care is also taken with other aspects of the algorithm, such as programming the partition algorithm so that it swaps equal elements in order to ensure that good partitions result on sequences of all equal elements or sequences whose elements are chosen from a relatively small set.

to blow up. While this is probably true, perhaps there is still some reason to expect these or equally damaging sequences to occur more often in practice than a probabilistic argument based on uniform distribution of permutations would suggest. Indeed, some authors have proposed substituting for the uniform distribution a "universal distribution" that assigns much higher probability to sequences that can be produced by short programs, as can the K_N sequences, than to random sequences (those that require programs of length proportional to the sequence length) [11]. Under the universal distribution, both quicksort's worst-case *and average* computing times are $\Theta(N^2)$. The average and worst-case times for heapsort—and introsort—remain $\Theta(N \log N)$ under the universal distribution.

The Algorithm

The details of the introsort algorithm are perhaps easiest to understand in terms of its differences from the following version of median-of-3 quicksort—the one used in the HP implementation of the C++ Standard Template Library [12] for the *sort* function. This version uses a constant called size_threshold to stop generating subproblems for small sequences, leaving the problem instead for a later pass using insertion sort. Leaving small subproblems to insertion sort is one of the usual optimizations of quicksort; the merits of leaving them until a final pass rather than calling insertion sort immediately are discussed later. As in [13], block structure is indicated in the pseudo-code using indentation rather than bracketing.

```
Algorithm QUICKSORT(A, f, b)
                    Inputs: A, a random access data structure containing the sequence
                       of data to be sorted, in positions A[f], ..., A[b − 1];
                      f, the first position of the sequence
                      b, the first position beyond the end of the sequence
                    Output: A is permuted so that A[f] ≤ A[f+1] ≤ ... ≤ A[b − 1]
   QUICKSORT_LOOP(A, f, b)
   INSERTION_SORT(A, f, b)

Algorithm QUICKSORT_LOOP(A, f, b)
                    Inputs: A, f, b as in QUICKSORT
                    Output: A is permuted so that A[i] ≤ A[j]
                      for all i, j: f ≤ i < j < b and size_threshold < j − i
   while b − f > size_threshold
      do   p := PARTITION(A, f, b, MEDIAN_OF_3(A[f], A[f+(b−f)/2], A[b−1]))
           if (p − f ≥ b − p)
              then QUICKSORT_LOOP(A, p, b)
                   b := p
              else QUICKSORT_LOOP(A, f, p)
                   f := p
```

The test $\mathrm{p} - \mathrm{f} \geq \mathrm{b} - \mathrm{p}$ is to ensure that the recursive call is on a subsequence of length no more than half of the input sequence, so that the stack depth is $O(\log N)$ rather than $O(N)$. The details of INSERTION_SORT, MEDIAN_OF_3 and PARTITION are not important for this discussion; these routines are also used without change in INTROSORT.

The key modification to quicksort to avoid quadratic worst-case behavior is to make the algorithm "aware" of how many subproblem levels it is generating. It can then avoid the performance problem by switching to heapsort when the number of levels reaches a limit. According to Theorem 2 below, the depth limit must be $O(\log N)$. Although any choice for the constant factor in this bound will ensure an overall time bound of $O(N \log N)$, it must not be so small that the algorithm calls heapsort too frequently (causing it to be little better than using heapsort in the first place). In the following pseudo-code the depth limit is $2\lfloor \log_2 N \rfloor$, since this value empirically produces good results.

```
Algorithm INTROSORT(A, f, b)
                    Inputs: A, a random access data structure containing the sequence
                        of data to be sorted, in positions A[f], ..., A[b − 1];
                      f, the first position of the sequence
                      b, the first position beyond the end of the sequence
                    Output: A is permuted so that A[f] ≤ A[f+1] ≤ ... ≤ A[b − 1]
    INTROSORT_LOOP(A, f, b, 2 * FLOOR_LG(b − f))
    INSERTION_SORT(A, f, b)

Algorithm INTROSORT_LOOP(A, f, b, depth_limit)
                    Inputs: A, f, b as in INTROSORT;
                      depth_limit, a nonnegative integer
                    Output: A is permuted so that A[i] ≤ A[j]
                      for all i, j: f ≤ i < j < b and size_threshold < j − i
    while b − f > size_threshold
        do if depth_limit = 0
              then HEAPSORT(A, f, b)
                   return
           depth_limit := depth_limit − 1
           p := PARTITION(A, f, b, MEDIAN_OF_3(A[f], A[f+(b−f)/2], A[b−1]))
           INTROSORT_LOOP(A, p, b, depth_limit)
           b := p
```

In INTROSORT_LOOP it is possible to omit the test for recursing on the smaller half of the partition, since the depth limit puts an $O(\log N)$ bound on the stack depth anyway. This omission nicely offsets the added test for exceeding the depth-limit, keeping the algorithm's overhead essentially the same as QUICKSORT's. We can skip the details of the HEAPSORT algorithm, since they are unimportant for the time bound claimed for INTROSORT, as long as HEAPSORT, or any other algorithm used as the "stopper," has an $O(N \log N)$ worst-case time bound.

Theorem 2 *Assuming an $O(\log N)$ subproblem tree depth limit and an $O(N \log N)$ worst-case time bound for* HEAPSORT*, the worst-case computing time for* INTROSORT *is $\Theta(N \log N)$.*

Proof: The time for partitioning is bounded by N times the number of levels of partitioning, which is bounded by the subproblem tree depth limit. Hence the total time for partitioning is $O(N \log N)$. Suppose INTROSORT calls HEAPSORT j times on sequences of length $n_1, \ldots, n_j$. Let c be a constant such that $cN \log_2 N$ bounds the time for HEAPSORT on a sequence of length N; then the time for all the calls of HEAPSORT is bounded by

$$\sum_{i=1}^{j} c n_i \log_2 n_i \leq c \log_2 N \sum_{i=1}^{j} n_i \leq c N \log_2 N.$$

or $O(N \log N)$ also. Therefore the total time for INTROSORT is $O(N \log N)$.

The lower bound is also $\Omega(N \log N)$, since a sequence such as an already sorted sequence produces equal length partitions in all cases, resulting in $\log_2 N$ levels each taking $\Omega(N)$ time. □

Implementation and Optimization

An important detail not indicated by the pseudo-code is the parameterization of the algorithms by the type of data and the type of function used for comparisons. Such parameterization is a fairly common use of the C++ template feature, but the Standard Template Library goes further by also parameterizing on the sequence representation. Specifically, instead of passing an array base address and using integer indexing, the boundaries of the sequence are passed via iterators, which are a generalization of ordinary C/C++ pointers. The iterator type is a template parameter of the algorithms, as in [4, 5]. The implementation of introsort is parameterized in the same way; for example, the C++ source code for the main function is

```
template <class RandomAccessIterator, class T, class Distance>
void __introsort_loop(RandomAccessIterator first,
                      RandomAccessIterator last, T*,
                      Distance depth_limit) {
    while (last - first > __stl_threshold) {
      if (depth_limit == 0) {
        partial_sort(first, last, last);
        return;
      }
      --depth_limit;
      RandomAccessIterator cut = __unguarded_partition
        (first, last, T(__median(*first, *(first + (last - first)/2),
                                 *(last - 1))));
     __introsort_loop(cut, last, value_type(first), depth_limit);
     last = cut;
    }
}
```

STL defines five categories of iterators, the most powerful being random-access iterators, for which $i+n$, for iterator i and integer n, is a constant time operation. Introsort, like quicksort and heapsort, requires random-access iterators, which means that it cannot be used with linked lists. However it is not restricted just to arrays; in STL, random-access iterators are also provided by vectors and deques, which are like arrays in most respects but dynamically expandable at one or at both ends, respectively.

For heapsort, the implementation calls the STL generic `partial_sort` algorithm. Similarly, the call to insertion sort is coded as a call to an internal insertion sort routine already provided in the HP STL code.

Performance Measurements

To compare the computing time performance of algorithms that have the same asymptotic time bounds, as is the case here, we must look for more discriminating ways of describing them. For this purpose it is traditional to use formulas that express the actual time an algorithm uses or the number of operations it does, in the worst or average case, when applied to any input in a certain class. Actual times are the most accurate description but may only apply to a single hardware architecture and compiler configuration. Operation counts are more portable—except for dependence, in some cases, on compiler optimizations—but they may bear little relation to actual times when only certain operations are counted, such as only counting the comparison operations done by a sorting algorithm. For example in STL heapsort, present as a special case of its `partial_sort` generic algorithm, does fewer comparisons than the STL median-of-3 quicksort version (`sort`), but its actual computing time is usually significantly higher because it does more assignments and other operations. And since these algorithms are generic, they access most of the operations they do through type parameters, and thus the relative cost of different kinds of operations can vary when they are specialized with different types.

With algorithms that are generic in the sequence representation, as in STL, the number of *iterator operations* and their cost relative to comparisons and assignments are also important. As discussed in more detail at the end of this section, in the deque representation in the HP STL implementation most iterator operations cost several times as much as vector or array iterator operations, and this fact is significant in assessing the relative performance of different variants of introsort.

A fourth kind of operations that appears in these algorithms is *distance operations*, which are arithmetic operations on integer results of iterator subtractions, such as the division by 2 in the expression `(last - first)/2`. The cost of a distance operation is mostly independent of the data or iterator types, but it is still useful to know how many such operations are performed. Heapsort, for example, does many more distance operations than quicksort or introsort, which, along with its higher number of iterator operations, accounts for its poorer performance in most cases.

Thus, the performance tables below do not show actual times but instead give separate counts for comparison, assignment, iterator, and distance operations. Table 1 shows the operation counts for introsort, quicksort, and heapsort on randomly-ordered sequences ranging in size from 1,000 to 1,024,000 elements. Each value is the median of measurements on seven random sequences. The table shows that on such sequences the operation counts for introsort are almost identical to those of quicksort. Heapsort comes in with an approximately 15% smaller comparison count but does about 1.5 times as many assignments, 2.5 times as many iterator operations and more than 250 times as many distance operations. The last column gives the total number of operations, which is a guide to how actual computing times compare when each kind of operation takes about the same time, as for example with arrays containing integer elements. The total for heapsort is more than 4 times the total for introsort or quicksort.

Table 2 shows the operation counts for the three algorithms on the median-of-3 killer sequences. For quicksort the operation total grows quadratically, as predicted by the analysis. Note also that on these sequences introsort is somewhat slower than heapsort, because it performs $2\lfloor\log_2 N\rfloor$ partitions before switching to heapsort on a sequence of length $N - 4\lfloor\log_2 N\rfloor$. Thus the time for the heapsort call within introsort is almost as great as the heapsort time on the original sequence, but introsort adds to this the time for the partitions.

There are also some sequences on which introsort fares worse than quicksort, though never so dramatically as the differences in the other direction. Such sequences T_N can be constructed by taking K_N and randomly shuffling the elements in positions $4\lfloor\log_2 N\rfloor$ through $N/2$ and $N/2 + 2\lfloor\log_2 N\rfloor$ through N (call such sequences "two-faced"). On T_N, introsort switches to heapsort after $2\lfloor\log_2 N\rfloor$ partitions, but since the heapsort call is on a random sequence it takes longer than continuing to partition would.

Table 1: Performance of Introsort, Quicksort, and Heapsort on Random Sequences (Sizes and Operations Counts in Multiples of 1,000)

Size	Algorithm	Comparisons	Assignments	Iterator Ops	Distance Ops	Total Ops
1	Introsort	11.9	9.4	52.9	1.2	75.4
	Quicksort	11.9	9.4	53.3	1.2	75.7
	Heapsort	10.3	15.5	136.1	159.1	320.9
4	Introsort	57.2	43.6	246.9	4.7	352.5
	Quicksort	57.2	43.6	248.6	4.6	354.0
	Heapsort	49.3	70.2	640.5	748.8	1508.8
16	Introsort	265.7	203.4	1130.6	18.5	1618.2
	Quicksort	265.7	203.4	1137.2	18.5	1624.8
	Heapsort	229.2	318.9	2945.1	3442.5	6935.7
64	Introsort	1235.1	934.6	5125.7	73.6	7369.0
	Quicksort	1235.1	934.6	5152.3	73.5	7395.5
	Heapsort	1044.7	1435.6	13316.9	15562.6	31359.7
256	Introsort	5644.4	4093.4	22965.6	293.5	32996.8
	Quicksort	5644.4	4093.4	23072.2	293.4	33103.4
	Heapsort	4691.0	6254.4	59411.1	69419.7	139776.3
1024	Introsort	24945.6	17805.8	100946.7	1177.1	144875.1
	Quicksort	24945.6	17805.8	101374.4	1176.4	145302.2
	Heapsort	20812.4	27065.6	262222.8	306349.8	616450.6

Experiments were also performed with an introsort version in which the insertion sort call is placed at the end of INTROSORT_LOOP, where it does many separate calls on small subarrays, rather than at the end of INTROSORT, where it does one final pass over the entire array. The one-final-pass position is one of the quicksort optimizations suggested by Sedgewick [14]. But with modern memory caches the savings in overhead may be cancelled or outweighed for large arrays, since the complete pass at the end can double the number of cache misses [15].[4] However, in these experiments the many-insertion-sort-call version almost never ran faster than the one-final-pass version, and in some cases ran considerably slower. One important case is sorting a deque, whose implementation in HP STL is in terms of a two-level structure consisting of a block of pointers to fixed-sized blocks. This representation supports random access, but iterator operations are slower than array or vector iterator operations by a constant factor, typically three to five. Measurements show that the many-call version performs about 8% more iterator operations than the one-call version, causing it to take 25-40% more time when sorting a deque. Thus the one-call version is definitely to be preferred when sorting a deque, and the only question is whether the many-call version should be used for arrays or vectors (giving up some genericity). For now, it seems better to retain only the one-call version, since the cases in which the many-call version might be faster are rare, but this decision should probably be revisited as relative cache-miss penalties continue to increase.

Finally, it may be of some interest that the operation counts were obtained without modifying the algorithm source code at all, by specializing their type parameters with classes whose

[4] In INTROSORT the many-call version also has the advantage that insertion sorting is never done on a region already sorted by HEAPSORT, although this is relatively unimportant since HEAPSORT is very rarely called, and insertion-sorting an already sorted array is very fast anyway.

Table 2: Performance of Introsort, Quicksort, and Heapsort on Median-of-3 Killer Sequences (Sizes and Operations Counts in Multiples of 1,000)

Size	Algorithm	Comparisons	Assignments	Iterator Ops	Distance Ops	Total Ops
1	Introsort	28.8	17.1	175.6	153.6	375.1
	Quicksort	199.1	6.4	421.3	3.6	630.4
	Heapsort	10.4	15.6	136.9	159.4	322.3
4	Introsort	140.8	78.2	845.8	743.2	1808.1
	Quicksort	3063.1	28.8	6226.2	14.4	9332.4
	Heapsort	49.7	70.9	644.5	750.2	1515.3
16	Introsort	662.0	353.1	3918.2	3446.0	8379.3
	Quicksort	48334.2	132.5	97103.0	57.3	145627.1
	Heapsort	231.1	321.8	2966.6	3454.1	6973.6
64	Introsort	3035.4	1574.7	17748.3	15602.5	37961.0
	Quicksort	769724.0	609.4	1541328.4	229.4	2311891.2
	Heapsort	1052.6	1447.3	13403.2	15610.8	31513.8

operations have counters built into them. This is an obvious technique for counting element type comparison and assignment operations, but iterator operations and even distance operations can be counted in the same way since STL generic algorithms also access them through type parameters. The most useful way of defining a counting class is as an *adaptor* [4], which is a generic component that takes another component and outfits it with a new interface, or provides the same interface but with different or additional "behind-the-scenes" functionality. A counting adaptor is an example of adding both to the interface (for initializing and reporting counts) and internal functionality (incrementing the counts).

Introspective Selection Algorithms

Hoare's *find* algorithm [7] for selecting the i-smallest element of a sequence is similar to quicksort, but only one of the two subproblems generated by partitioning must be pursued. With even splits, the computing time is $O(N+N/2+N/4+\ldots)$, or $O(N)$. But the same median-of-3 killer sequences described in Section 2 cause the selection algorithm to do $N/4$ partitions when finding the median, and the computing time thus becomes $\Omega(N^2)$.

If, as in introsort, we limit the depth to a logarithmic bound, the average time remains linear but the worst case is $\Theta(N \log N)$. We could get an overall linear worst-case bound by putting a *constant* bound on partitioning depth, but that would mean that for sufficiently large sequences we would switch to the stopper algorithm on most inputs even though Hoare's algorithm would run faster. A better approach is to require that the sequence size is cut at least in half by any k consecutive partitions, for some positive integer k; if this test fails, then switch to the linear time algorithm. This limits the partitioning depth to $k\lceil \log_2 N \rceil$ and the total time to $O(N)$. Another approach that achieves this time bound is limiting the sum of the sizes of all partitions generated to some constant times N.

The details of such "introselect" algorithms and the results of experiments comparing them with other selection algorithms will be given in a separate paper.

Implications for Generic Software Libraries

Let us conclude by noting the possible advantages and disadvantages of introsort and introselect relative to other sorting and selection algorithms included in a generic software library such as STL.

STL is not a set of specific software components but a set of requirements which components must satisfy. By making time complexity part of the requirements for components, STL insures that compliant components not only have the specified interfaces and semantics but also meet certain computing time bounds. The complexity requirements were chosen based on the characteristics of the best algorithms and data structures known at the time the requirements were proposed. Whenever possible, the requirements are expressed as worst-case time bounds. Requiring $O(N \log N)$ worst-case time for the generic `sort` algorithm would have meant heapsort could be used but would have precluded quicksort. And of course, only requiring $O(N^2)$ worst-case time would have permitted insertion sort (or even bubble sort!) to be used. So in this case the requirements only stipulate $O(N \log N)$ *average* time, in order to allow quicksort to be used for this component. The specification then appends the caveat, "if the worst-case behavior is important `stable_sort` or `partial_sort` should be used." In view of introsort's combination of parity with quicksort on most sequences while meeting an $O(N \log N)$ worst-case bound, the STL requirement could be now best be expressed as an $O(N \log N)$ worst-case bound, with no need for the caveat.

Is there now any need to have heapsort publicly available in a generic library? Yes, if as in STL it is present in a more general form, called `partial_sort`, capable of placing the smallest k elements of a sequence of length N in the first k positions, leaving the remaining $N - k$ elements in an undefined order. Its time complexity requirement is $O(N \log k)$, which is not as good as could be done using a linear selection algorithm to place the k smallest elements at the beginning of the array and sorting them with heapsort, for a total time of $O(N + k \log k)$. However, it does permit an algorithm that makes a heap out of the first k elements, then maintains in it the k smallest elements of the i elements seen so far as i goes to N, and finally sorts the heap. This algorithm, which is used in HP STL, is faster on the average than the select-and-sort algorithm.

Introsort, like quicksort, is not stable—does not preserve the order of equivalent elements—so there is still a need to have a separate requirement for a stable sorting routine. In STL the requirements for `stable_sort` are written to be satisfiable by a merge sort algorithm.

A possible disadvantage of introsort is the extra code space required, roughly double the requirement for quicksort or heapsort alone. In a compile-time generic library such as STL, this problem is exacerbated when many different instances of a generic component are required in a single application program. The problem fades away however if there is a separate need for `partial_sort`, since the code for introsort can just call it and need not include it in-line.

Finally, combining introspection with partitioning methods other than median-of-3 should be explored. For example, it can also be applied to partitioning using the simpler choice of the first element as the pivot; the resulting algorithm still has a $\Theta(N \log N)$ time bound. However, because the simpler choice of pivot does not partition as evenly as median-of-3 partitioning does, the average time is higher. Another possibility is to apply depth limiting to the algorithm described in [9], which adaptively selects from larger arrays a larger sample of elements from which to estimate the median. Experimentation with this algorithm and other variations of will be the subject of a future paper.

Acknowledgments C. Stewart suggested the idea for the class of sequences K_N. J. Valois and two anonymous referees made many useful comments on an earlier draft of this paper.

References

[1] C. A. R. Hoare. Quicksort. *Computer Journal*, 5(1):10-15, 1962.

[2] R. C. Singleton. *Communications of the ACM*, 12:195–187, 1969.

[3] J. W. J. Williams. Algorithm 132 (heapsort). *Communications of the ACM*, 7:347–348, 1964.

[4] D. R. Musser and A. A. Stepanov. Algorithm-oriented generic libraries. *Software Practice and Experience*, 24(7), July 1994.

[5] A. A. Stepanov and M. Lee. *The Standard Template Library*, Technical Report HPL-94-34, Hewlett-Packard Laboratories, May 31, 1994, revised October 31, 1995; incorporated into Accredited Standards Committee X3 (American National Standards Institute), Information Processing Systems, *Working Paper for Draft Proposed International Standard for Information Processing Systems—Programming Language C++*. Doc. No. X3J16/96-0185, WG21/N0785, April 1995.

[6] D. R. Musser and A. Saini. *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library*. Addison-Wesley, Reading, MA, 1996.

[7] C. A. R. Hoare. Algorithm 63 (partition) and algorithm 65 (find). Communications of the ACM, 4(7):321-322, 1961.

[8] M. Blum, R. W. Floyd, V. Pratt, R. L. Rivest, and R. E. Tarjan. Time bounds for selection. *Journal of Computer and System Sciences*, 7(4):448–461, 1973.

[9] J. L. Bentley and M. D. McIlroy. Engineering a sort function. *Software Practice and Experience*, 23(11), November 1993.

[10] C. V. Stewart, Private communication.

[11] M. Li and P. M. B. Vitányi. Average case complexity under the universal distribution equals worst-case complexity. *Information Processing Letters*, 42:145–149, 1992.

[12] A. A. Stepanov, M. Lee, and D. R. Musser. Hewlett-Packard Laboratories reference implementation of the Standard Template Library, source files available from ftp://ftp.cs.rpi.edu/pub/stl.

[13] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 1990.

[14] R. Sedgewick. Implementing quicksort programs, *Communications of the ACM*, 21(10):847-857, 1978.

[15] A. LaMarca and R. E. Ladner. The influence of caches on the performance of sorting. *Proceedings of Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, January 1997.