SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 23(11), 1249–1265 (NOVEMBER 1993)

Engineering a Sort Function

JON L. BENTLEY
M. DOUGLAS McILROY
AT&T Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974, U.S.A.

SUMMARY

We recount the history of a new `qsort` function for a C library. Our function is clearer, faster and more robust than existing sorts. It chooses partitioning elements by a new sampling scheme; it partitions by a novel solution to Dijkstra's Dutch National Flag problem; and it swaps efficiently. Its behavior was assessed with timing and debugging testbeds, and with a program to certify performance. The design techniques apply in domains beyond sorting.

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

INTRODUCTION

C libraries have long included a `qsort` function to sort an array, usually implemented by Hoare's Quicksort.[1] Because existing `qsort`s are flawed, we built a new one. This paper summarizes its evolution.

Compared to existing library sorts, our new `qsort` is faster—typically about twice as fast—clearer, and more robust under nonrandom inputs. It uses some standard Quicksort tricks, abandons others, and introduces some new tricks of its own. Our approach to building a `qsort` is relevant to engineering other algorithms.

The `qsort` on our home system, based on Scowen's 'Quickersort',[2] had served faithfully since Lee McMahon wrote it almost two decades ago. Shipped with the landmark Seventh Edition Unix System,[3] it became a model for other `qsort`s. Yet in the summer of 1991 our colleagues Allan Wilks and Rick Becker found that a `qsort` run that should have taken a few minutes was chewing up hours of CPU time. Had they not interrupted it, it would have gone on for weeks.[4] They found that it took n^2 comparisons to sort an 'organ-pipe' array of $2n$ integers: $123..nn..321$.

Shopping around for a better `qsort`, we found that a `qsort` written at Berkeley in 1983 would consume quadratic time on arrays that contain a few elements repeated many times—in particular arrays of random zeros and ones.[5] In fact, among a dozen different Unix libraries we found no `qsort` that could not easily be driven to quadratic behavior; all were derived from the Seventh Edition or from the 1983 Berkeley function. The Seventh

0038-0644/93/111249–17$13.50
Received 21 August 1992
Revised 10 May 1993

Edition `qsort` and several others had yet another problem. They used static storage and thus would fail if called recursively from within the comparison function or if used in a multithreaded computation.

Unable to find a good enough `qsort`, we set out to build a better one. The algorithm should avoid extreme slowdowns on reasonable inputs, and should be fast on ‘random’ inputs. It should also be efficient in data space and code space. The sort need not be stable; its specification does not promise to preserve the order of equal elements.

THE QSORT INTERFACE

Despite its suggestive name, the `qsort` function need not be implemented by Quicksort. We’ll first implement the function with an insertion sort, which, though slow, will prove useful in an industrial-strength sort later. Our starting point, `iisort`, sorts `n` integers in the array `a` but lacks the `qsort` interface. (In the naming scheme used throughout this paper, the first `i` in `iisort` stands for ‘integer’, the second for ‘insertion’.) The algorithm varies index `i` from the second through the last element of the array, and varies `j` to sift the `i`-th element down to its proper place in the preceding subarray.

```
void iisort(int *a, int n)
{
    int i, j;

    for (i = 1; i < n; i++)
        for (j = i; j > 0 && a[j-1] > a[j]; j--)
            iswap(j, j-1, a);
}
```

The function `iswap(i, j, a)` exchanges the integers `a[i]` and `a[j]`. Insertion sort uses about $n^2/4$ comparisons on a randomly permuted array, and it never uses more than $n^2/2$ comparisons.

This `iisort` function sorts only integers. For the moment, we take the general `qsort` interface to be†

```
void qsort(char *a, int n, int es, int (*cmp)());
```

The first parameter points to the array to be sorted. The next two parameters tell the number of elements and the element size in bytes. The last parameter is a comparison function that takes two pointer arguments. The function returns an integer that is less than, equal to, or greater than zero when the first argument points to a value less than, equal to, or greater than the second. Here is a typical comparison function and a sample call to sort an array of non-negative integers.

† We have used simple parameter declarations for readability. The official prototype in the ANSI standard header file, `<stdlib.h>`, is[6]

```
void qsort(void *, size_t, size_t, int (*)(const void *, const void *));
```

This declaration can be used no more honestly than ours. The first argument compels casting in the source of `qsort`; the last compels casting in programs that call it. In practical terms, though, our declaration precludes portable compatibility with library `qsort`s. Hence we will change the `int` parameters to `size_t` in our production model, Program 7.

```
void swap(char *i, char *j, int n)
{
    do {
        char c = *i;
        *i++ = *j;
        *j++ = c;
    } while (--n > 0);
}
```

Program 1. A simple swap function

```
int intcomp(int *i, int *j) { return *i - *j; }
 ...
qsort((char *) a, n, sizeof(int), intcomp);
```

To sort an array of `len`-byte strings with terminal null characters, use the standard string-comparison routine, `strcmp`:

```
qsort(a, n, len, strcmp);
```

To sort an array of pointers to strings, use `strcmp` with another level of indirection.

```
int pstrcmp(char **i, char **j) { return strcmp(*i, *j); }
 ...
qsort(a, n, sizeof(char *), pstrcmp);
```

By straightforward rewriting we convert `iisort` to handle a `qsort`-like interface.

```
void isort(char *a, int n, int es, int (*cmp)())
{
    char *pi, *pj;

    for (pi = a + es; pi < a + n*es; pi += es)
        for (pj = pi; pj > a && cmp(pj-es, pj) > 0; pj -= es)
            swap(pj, pj-es, es);
}
```

The function `swap(i,j,n)`, defined in Program 1, interchanges `n`-byte fields pointed to by `i` and `j`. We will say more about swapping later.

A SIMPLE QSORT

Quicksort is a divide-and-conquer algorithm: partition the array, placing small elements on the left and large elements on the right, and then recursively sort the two subarrays. Sedgewick studied Quicksort in his Ph.D. thesis[7] and later papers;[8-10] it is widely described in texts[11, 12] and bibliographies.[13]

Program 2 is a trivial Quicksort, which uses a partitioning scheme due to Lomuto.[14] This code partitions around the first element in the array, which is finally placed in `a[j]`. To sort the left subarray `a[0..j-1]`, we call `iqsort0(a, j)`. To sort the right subarray `a[j+1..n-1]`, we use C pointer arithmetic and call `iqsort0(a+j+1, n-j-1)`.

Hoare proves that on n randomly permuted distinct elements, Quicksort makes $C_n \approx 2n \ln n \approx 1{\cdot}386 n \lg n$ comparisons.[1] Unfortunately, Program 2 is sometimes much slower. On arrays that are already sorted, it makes roughly $n^2/2$ comparisons. To avoid this problem Hoare suggests partitioning around a random element; we adopt the technique in Program 3. Program 2 also takes quadratic time on arrays of identical elements.

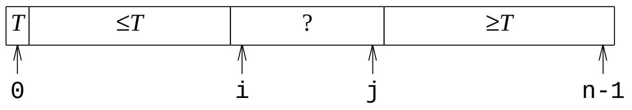
```
void iqsort0(int *a, int n)
{
    int i, j;

    if (n <= 1) return;
    for (i = 1, j = 0; i < n; i++)
        if (a[i] < a[0])
            swap(++j, i, a);
    swap(0, j, a);
    iqsort0(a, j);
    iqsort0(a+j+1, n-j-1);
}
```

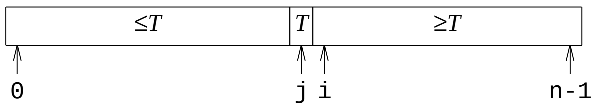
Program 2. A toy Quicksort, unfit for general use

A more efficient (and more familiar) partitioning method uses two indexes `i` and `j`. Index `i` scans up from the bottom of the array until it reaches a large element (greater than or equal to the partition value), and `j` scans down until it reaches a small element. The two array elements are then swapped, and the scans continue until the pointers cross.

This algorithm is easy to describe, and also easy to get wrong—Knuth tells horror stories about inefficient partitioning algorithms.[15] We avoid problems by using an invariant due to Sedgewick:[7]



Partition around the element `a[0]`, which we abbreviate as *T*. Increment `i` over elements less than *T*; on reaching an element greater than or equal to *T*, start moving `j` down. When both scans stop, swap the two elements and continue the process. (It is important that both inner loops stop on an equal element. The Berkeley `qsort` takes quadratic time on random zeros and ones precisely because it scans over equal elements; try an example to understand why.) At the end of the partitioning loop `i`=`j`+1. Swap `a[0]` and `a[j]` to yield:



Now call `qsort` recursively on the subarrays `a[0..j-1]` and `a[j+1..n-1]`.

Program 3 combines these ideas into a clean and efficient Quicksort specialized for integers—a fine starting point for building more elaborate functions. One such elaboration is Program 4, which supports the `qsort` interface. Only a third the size of the Seventh Edition `qsort`, it is still about twenty percent faster on average; and it avoids the bad quadratic case. It is the seed for our final algorithm.

A COST MODEL

Before we speed up Program 4, we need an idea of the costs of the critical operations. Bentley, Kernighan and Van Wyk describe a program that generates cost estimates for common C operations on a particular hardware/software system.[16] We modified that program to mea-

```
void iqsort1(int *a, int n)
{
    int i, j;

    if (n <= 1) return;
    i = rand() % n;
    swap(0, i, a);
    i = 0;
    j = n;
    for (;;) {
        do i++; while (i < n && a[i] < a[0]);
        do j--; while (a[j] > a[0]);
        if (j < i) break;
        swap(i, j, a);
    }
    swap(0, j, a);
    iqsort1(a, j);
    iqsort1(a+j+1, n-j-1);
}
```

Program 3. A simple Quicksort for integers

```
void qsort1(char *a, int n, int es, int (*cmp)())
{
    int  j;
    char *pi, *pj, *pn;

    if (n <= 1) return;
    pi = a + (rand() % n) * es;
    swap(a, pi, es);
    pi = a;
    pj = pn = a + n * es;
    for (;;) {
        do pi += es; while (pi < pn && cmp(pi, a) < 0);
        do pj -= es; while (cmp(pj, a) > 0);
        if (pj < pi) break;
        swap(pi, pj, es);
    }
    swap(a, pj, es);
    j = (pj - a) / es;
    qsort1(a, j, es, cmp);
    qsort1(a + (j+1)*es, n-j-1, es, cmp);
}
```

Program 4. A simple qsort

sure the cost of about forty common sorting operations. Table I shows the cost of a dozen representative operations using the `lcc` compiler[17] for ANSI C on a VAX 8550.

On this system, bookkeeping operations cost a few tenths of a microsecond, comparisons start at a few microseconds and go up from there, and swapping four-byte fields weighs in at a whopping dozen microseconds. (The `strcmp` time is for comparing two 10-byte strings that differ only in the last byte.)

The outlier is the `swap` function from Program 1. As the appendix reports, it is even more expensive on other systems. There are several reasons: four bytes are swapped in a loop, the loop maintains two pointers and a count, and the function calling sequence takes a couple of microseconds. As a benchmark, swapping two integers in inline code takes just under a microsecond. In the appendix, the `swap` code is tuned to exploit this common spe-

Table I

	CPU Microseconds
Int Operations	
`i1 = i2 + i3`	0·20
`i1 = i2 - i3`	0·20
Pointer Operations	
`p1 -= es`	0·17
`p1 += es`	0·16
Control Structures	
`if (p1 == p2) i1++`	0·32
`while (p1 < p2) i1++`	0·26
Comparison Functions	
`i1 = intcomp(&i2, &i3)`	2·37
`i1 = floatcomp(&f2, &f3)`	3·67
`i1 = dblcomp(&d2, &d3)`	3·90
`i1 = strcmp(s2, s3)`	8·74
Swap Functions	
`swap(p1, p2, 4)`	11·50
`t = *i1, *i1 = *i2, *i2 = t`	0·84

cial case, using inline swaps for integer-sized objects and a function call otherwise. This reduces the cost of swapping two four-byte fields to around a microsecond.

The tuned `swap` costs about three times as much as bookkeeping operations, and comparisons cost about three times as much as swaps. These ratios differ dramatically from the costs of sorts specialized to integers, which figure heavily in textbook analyses. In the integer models, of which Knuth's MIX programs are classic examples,[12] comparison costs about as much as bookkeeping, while swapping is a few times more expensive. The two cost models can be summarized as:

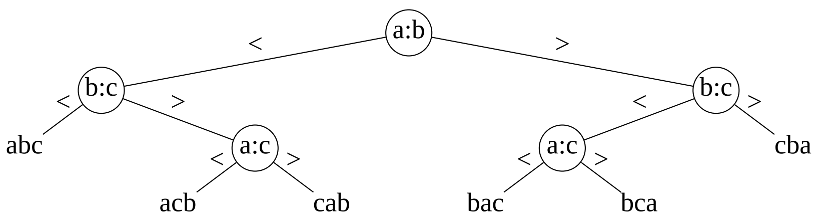
MIX:	overhead	$\approx$	comparisons	$<$	swaps
`qsort`:	overhead	$<$	swaps	$<$	comparisons

The second model reflects the generality of the `qsort` interface, in which comparison is a function, not a machine primitive. The inappropriateness of the MIX model was pointed out by Linderman in discussing the Unix `sort` utility: 'Because comparison ... is interpretive, it is generally more time-consuming than the standard paradigm of comparing two integers. When a colleague and I modified `sort` to improve reliability and efficiency, we found that techniques that improved performance for other sorting applications sometimes degraded the performance of `sort`.'[18]

CHOOSING A PARTITIONING ELEMENT

Partitioning about a random element takes $C_n \approx 1{\cdot}386 n \lg n$ comparisons. We now whittle away at the constant in the formula. If we were lucky enough to choose the median of every subarray as the partitioning element, we could reduce the number of comparisons to about $n \lg n$. In default of the true median, which is expensive to compute, we seek a cheap approximation.

Hoare proposed using the median of a small sample of array elements. Singleton suggested the median of the first, middle and last elements of the array.[19] Partitioning around the median of three random elements reduces the expected number of comparisons to $C_{n,3} \approx$



```
static char *med3(char *a, char *b, char *c, int (*cmp)())
{    return cmp(a, b) < 0 ?
          (cmp(b, c) < 0 ? b : cmp(a, c) < 0 ? c : a)
        : (cmp(b, c) > 0 ? b : cmp(a, c) > 0 ? c : a);
}
```

Program 5. Decision tree and program for median of three

$12/7 n \ln n \approx 1{\cdot}188 n \lg n$. But while Quicksort originally took $C_n/6$ swaps, a median-of-three partition increases the swaps to $C_{n,3}/5$, for an increase of about three percent.[12]

Program 5 finds the median of three elements, using the `qsort` comparison function. It takes two or three comparisons (8/3 on average) and no swaps to evaluate the decision tree shown.

With more effort finding a central element, we can push the number of comparisons closer to $n \lg n$. We adopted Tukey's 'ninther', the median of the medians of three samples, each of three elements. Weide analyzes the quality of such pseudo-medians.[20] The ninther yields a better-balanced recursion tree at the cost of at most twelve extra comparisons. While this is cheap for large arrays, it is expensive for small arrays. 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. In the following code the size breaks were determined empirically.

```
pm = a + (n/2)*es;    /* Small arrays, middle element */
if (n > 7) {
    pl = a;
    pn = a + (n-1)*es;
    if (n > 40) {    /* Big arrays, pseudomedian of 9 */
        s = (n/8)*es;
        pl = med3(pl, pl+s, pl+2*s, cmp);
        pm = med3(pm-s, pm, pm+s, cmp);
        pn = med3(pn-2*s, pn-s, pn, cmp);
    }
    pm = med3(pl, pm, pn, cmp); /* Mid-size, med of 3 */
}
```

This scheme performs well on many kinds of nonrandom inputs, such as increasing and decreasing sequences. We could get fancier and randomize the sample, but a library sort has no business side-effecting the random number generator.

We experimented to find the average number of comparisons used by this algorithm. We set n to each power of two from 128 to 65,536, generated 11 sets of n random 30-bit integers, and counted the number of comparisons used by Program 4. A weighted least-squares regression fit the data to the function $1{\cdot}362 n \lg n - 1{\cdot}41 n$, which is close to the theoretical $1{\cdot}386 n \lg n + O(n)$. Similar experiments on Program 7, which uses the adaptive scheme, showed that it makes $1{\cdot}094 n \lg n - 0{\cdot}74 n$ comparisons. This is a substantial improvement over choosing a random element, and close to the lower bound of $n \lg n - 1{\cdot}44 n$.

A TIMING TESTBED

We built a simple testbed to measure the times of `qsort`s on random data. It takes five inputs: the name of a sort routine, the number and kind of data to sort, a modulus by which numbers should be reduced, and a count of the number of experiments to perform. Here is an interactive session:

```
q7 10000 i 50000 3
q7 10000 i 50000 3      73      77      68      9·11
q1 10000 i 50000 3
q1 10000 i 50000 3      55      56      58      7·07
```

The user typed the first line, requesting that the Seventh Edition `qsort` (`q7`) be run on an array of `10000` `i`ntegers taken modulo `50000` in `3` experiments. The program writes the second line, which echoes the five input fields, followed by the three run times in software clock ticks (sixtieths of a second on a VAX). The final field says that the sort required an average of $9{\cdot}11 n \lg n$ microseconds. The third line requests similar experiments on Program 4 (`q1`), and the fourth line reports an average run time of $7{\cdot}07 n \lg n$ microseconds, about twenty percent faster than the Seventh Edition `qsort`.

Integers are cheap to compare and to swap; sorting them highlights overhead costs. By sorting the same sets of values in other representations, the testbed reveals performance across a spectrum of relative costs for overhead, swaps and comparisons. Single-precision floating-point numbers are the same size as integers to swap and slightly more expensive to compare; doubles are more expensive in both categories. Twenty-byte records whose first four bytes are integer keys yield fast comparisons and slow swaps. Twenty-character strings of five spaces followed by the decimal representation of integers are slow to compare and to swap. Finally, pointers to the latter are slow to compare and fast to swap.

THE OPPORTUNITY OF EQUAL ELEMENTS

Our original performance goal was good running time across most inputs. We deliberately decided not to tune the code to be fast on particular classes of inputs, such as 'almost sorted', even though others have found this approach fruitful.[21, 22] We were soon moved from our extreme position.

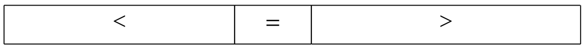
The version of `qsort` that we first offered locally was essentially Program 4 with adaptive partitioning. A friendly user found that the new `qsort` was typically thirty or forty percent faster than the existing function. But he also reported that on arrays with many identical elements our `qsort` slowed down slightly, while the Seventh Edition `qsort` ran like the wind. We explained our philosophical position, to which he rightly replied that it was unthinkable to replace a library routine with a program that was inferior on a common class of inputs: many users sort precisely to bring together equal elements.

Our testbed shows that Program 4 takes eight times as long to sort an array of 100,000 random integers mod 2 (zeros and ones) as does the Seventh Edition `qsort`:

```
q1 100000 i 2 3        768     756     756     7·63
q7 100000 i 2 3         96      93      91     0·94
```

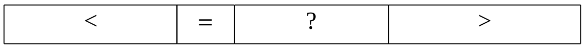
On an array of equal elements, Program 4 exchanges every possible pair of elements in a perfectly balanced recursion tree. Seventh Edition `qsort`, though, uses a 'fat partition' that finishes after a single scan through the elements. We therefore sought further to sort many identical elements efficiently without slowing down on distinct elements. (Sedgewick analyzes the performance of standard Quicksorts on equal elements.[9])

While our previous partitioning code divided the input into two subsequences, a fat partition divides the input into three:

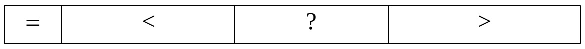


After partitioning, recur on the two subarrays at the ends, and ignore the equal elements in the middle.

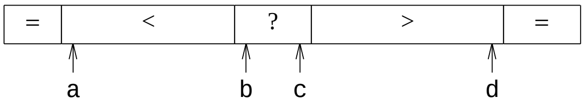
Tripartite partitioning is equivalent to Dijkstra's 'Dutch National Flag' problem.[23] Many programs (including Dijkstra's and the Seventh Edition `qsort`) use an invariant like



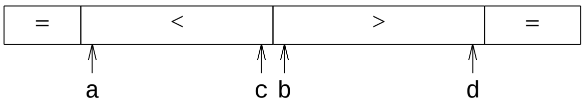
This gives code that is complex and slow. Equal elements at the left of the array take $O(n)$ extra swaps as they sift up to the middle. To get a more efficient fat partition we turn to another invariant,



After partitioning, bring the equals to the middle by swapping outer ends of the two left portions. Though this invariant is workable, a symmetric version leads to clearer code:



As index `b` moves up, scan over lesser elements, swap equal elements to the element pointed to by `a`, and stop at a greater element. Move index `c` down similarly, then swap the elements pointed to by `b` and `c`, adjust those pointers, and continue. (Wegner proposes exactly this invariant but maintains it with more complex three-way exchanges.[24]) On reaching the state



swap equals from the ends back to the middle. Program 6 shows an integer Quicksort that employs such 'split-end' partitioning. For brevity, it uses a random partitioning element.

Split-end partitioning is usually efficient. When all elements are distinct, it swaps only pairs of elements that are out of order. When equals predominate, it wastes time swapping them to the end and back to the middle. But we may amortize that effort against recursive calls not made on those equal elements. On arrays of random zeros and ones, our final

```
void iqsort2(int *x, int n)
{
    int a, b, c, d, l, h, s, v;

    if (n <= 1) return;
    v = x[rand() % n];
    a = b = 0;
    c = d = n-1;
    for (;;) {
        while (b <= c && x[b] <= v) {
            if (x[b] == v) iswap(a++, b, x);
            b++;
        }
        while (c >= b && x[c] >= v) {
            if (x[c] == v) iswap(d--, c, x);
            c--;
        }
        if (b > c) break;
        iswap(b++, c--, x);
    }
    s = min(a, b-a);
    for(l = 0, h = b-s; s; s--) iswap(l++, h++, x);
    s = min(d-c, n-1-d);
    for(l = b, h = n-s; s; s--) iswap(l++, h++, x);
    iqsort2(x, b-a);
    iqsort2(x + n-(d-c), d-c);
}
```

Program 6. An integer qsort with split-end partitioning

Quicksort with split-end partitioning (Program 7) is about twice as fast as the Seventh Edition `qsort`.

Fat partitioning allows us to alleviate another drawback of Programs 3 and 4. The disorder induced by swapping the partition element to the beginning is costly when the input is ordered in reverse or near-reverse. (Try an example to see why.) Instead of so swapping, Program 6 copies the partition value to an auxiliary variable, `v`. 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.

OTHER IMPROVEMENTS

We now have the ingredients for an efficient `qsort`: combine split-end partitioning with an adaptively sampled partitioning element. Program 7 incorporates several additional features for speed and portability:

1. We use an efficient `swap` macro and another macro, `SWAPINIT`, to set up for it; both are given in the appendix.
2. The macro `PVINIT`, also given in the appendix, arranges to keep the partition value in a temporary variable as in Program 6 when it is convenient to do so, otherwise in `a[0]` as in Program 4.

3. Although Quicksort is efficient for large arrays, its overhead can be severe for small arrays. Hence we use the old trick of sorting small subarrays by insertion sort.
4. We guard recursive calls to Quicksort on `n` elements with the test `if (n > 1)`.
5. Where Program 6 calls the swap function in a loop to bring equal elements to the middle of the array, we call a new `vecswap` function. On data with highly repetitive keys, `vecswap` saves twenty percent in overall sorting time.
6. Program 7 contains three integer constants (7, 7, 40) for choosing among insertion sort and various partition methods. Although the values were tuned for our home machine, the settings appear to be robust. (The range that consists of the single integer 7 could be eliminated, but has been left adjustable because on some machines larger ranges are a few percent better.)
7. The VAX object code of the venerable Seventh Edition `qsort` occupies about 500 bytes, while the Berkeley `qsort` requires a kilobyte. An early version of our sort grew to an excessive three kilobytes. With the help of a ‘space profiler’ that displays the number of bytes devoted to each source line, we beat the code down to a kilobyte. The largest space reductions came from moving code from macros to functions.
8. While the `size_t` parameters conform to the prototype in the standard header file `<stdlib.h>`, the other parameters do not. Conformance would require more casts. These type mismatches are usually benign, however, so Program 7 can be used verbatim as a library routine on most systems.

We have not adopted many customary improvements. By managing a private stack we could cut stack space from nearly 20 variables to 2 per level. By sorting the larger side of the partition last and eliminating tail recursion, we could reduce worst-case stack growth from linear in n to logarithmic. Neither trick is worth the effort. Since the expected stack size is logarithmic in n, the stack is likely to be negligible compared to the data—only about 2,000 bytes when $n = 1,000,000$. In the tests described in the next section, the stack depth reached three times the minimum possible depth, but no more. Moreover, if worst-case performance is important, Quicksort is the wrong algorithm. (A quick memory fault might even be preferred to wasting weeks of cycles on a worst-case run.)

There are other well known roads to optimization that we have not traveled.

1. Sedgewick does one big final insertion sort instead of little insertion sorts at the bottom the recursion.[7] This replaces several bookkeeping operations by a single comparison between array elements. It wins in the MIX cost model but loses in ours.
2. Sentinels at the ends of the array gain speed in the MIX cost model but lose in ours, and disobey the specification of `qsort` anyway.
3. Various improvements to insertion sort, including binary search, loop unrolling, and handling n=2 as a special case, were not helpful. The simplest code was the fastest.
4. More elaborate sampling schemes to choose a partitioning element were less effective than pseudomedians.
5. We declined to do special-case (`#ifdef` and `register`) tuning for particular machines and compilers,[8] though we did see cases where that would win.

CERTIFYING PERFORMANCE

We do well to heed Wirth’s advice: ‘It becomes evident that sorting on the basis of Quicksort is somewhat like a gamble in which one should be aware of how much one may afford to lose if bad luck were to strike.'[25] But we would like assurance that our `qsort` does not degrade on likely inputs. We therefore emulated Knuth's approach to testing TeX: 'I get into the meanest, nastiest frame of mind that I can manage, and I write the nastiest code I can think of; then I turn around and embed that in even nastier constructions that are almost obscene.'[26] For the comparatively simple `qsort`, we are concerned with performance as

```
void qsort(char *a, size_t n, size_t es, int (*cmp)())
{
    char *pa, *pb, *pc, *pd, *pl, *pm, *pn, *pv;
    int r, swaptype;
    WORD t, v;
    size_t s;

    SWAPINIT(a, es);
    if (n < 7) {     /* Insertion sort on smallest arrays */
        for (pm = a + es; pm < a + n*es; pm += es)
            for (pl = pm; pl > a && cmp(pl-es, pl) > 0; pl -= es)
                swap(pl, pl-es);
        return;
    }
    pm = a + (n/2)*es;    /* Small arrays, middle element */
    if (n > 7) {
        pl = a;
        pn = a + (n-1)*es;
        if (n > 40) {    /* Big arrays, pseudomedian of 9 */
            s = (n/8)*es;
            pl = med3(pl, pl+s, pl+2*s, cmp);
            pm = med3(pm-s, pm, pm+s, cmp);
            pn = med3(pn-2*s, pn-s, pn, cmp);
        }
        pm = med3(pl, pm, pn, cmp); /* Mid-size, med of 3 */
    }
    PVINIT(pv, pm);       /* pv points to partition value */
    pa = pb = a;
    pc = pd = a + (n-1)*es;
    for (;;) {
        while (pb <= pc && (r = cmp(pb, pv)) <= 0) {
            if (r == 0) { swap(pa, pb); pa += es; }
            pb += es;
        }
        while (pc >= pb && (r = cmp(pc, pv)) >= 0) {
            if (r == 0) { swap(pc, pd); pd -= es; }
            pc -= es;
        }
        if (pb > pc) break;
        swap(pb, pc);
        pb += es;
        pc -= es;
    }
    pn = a + n*es;
    s = min(pa-a,  pb-pa   ); vecswap(a,  pb-s, s);
    s = min(pd-pc, pn-pd-es); vecswap(pb, pn-s, s);
    if ((s = pb-pa) > es) qsort(a,    s/es, es, cmp);
    if ((s = pd-pc) > es) qsort(pn-s, s/es, es, cmp);
}
```

Program 7. The final qsort; see Appendix for macro and type definitions

```
for n in { 100, 1023, 1024, 1025 }
   for (m = 1; m < 2*n; m *= 2)
      for dist in { sawtooth, rand, stagger, plateau, shuffle }
         for (i = j = 0, k = 1; i < n; i++)
            switch (dist)
               case sawtooth: x[i] = i % m
               case rand:     x[i] = rand() % m
               case stagger:  x[i] = (i*m + i) % n
               case plateau:  x[i] = min(i, m)
               case shuffle:  x[i] = rand()%m? (j+=2): (k+=2)
         for type in { int, double }
            test copy(x)            /* work on a copy of x */
            test reverse(x, 0, n)   /* on a reversed copy  */
            test reverse(x, 0, n/2) /* front half reversed */
            test reverse(x, n/2, n) /* back half reversed  */
            test sort(x)            /* an ordered copy     */
            test dither(x)          /* add i%5 to x[i]     */
```

Figure 1. A qsort certification program in pseudocode

much as with correctness; our certifier produces what Lyness and Kaganove call a 'performance profile.'[27] It generates adverse inputs, including all the hard cases we found in the literature, sorts them, and complains when `qsort` uses too many comparisons. It tests both `int`s and `double`s to assess the different treatments of the partition element. The details are sketched in Figure 1.

Each `test` in Figure 1 calls `qsort` with a `cmp` function that counts the number of comparisons and reports cases in which that number exceeds $An \lg n$, where A is typically 1·2. If the number of comparisons becomes huge (A typically 10), the sort is aborted (with a C `longjmp`). Although the program lists only four different values of n, `qsort` recursively calls many lesser values of n, which should flush out any bugs lurking near the algorithmic breakpoints at $n = 7$ and $n = 40$. Function `test` checks answers against a trusted sort; we found several bugs this way.

The performance test reported the organ-pipe bug in Seventh Edition `qsort` and discovered the quadratic time for random zeros and ones and several other problems with the Berkeley code. (For instance, a mod-4 'sawtooth' with its front half reversed ran a factor of 8 slower than expected.)

By contrast, Program 7 proved to be quite robust,† as did Program 4, and (as expected) a Heapsort and a merge sort. The number of comparisons used by Program 7 exceeded the warning threshold, $1{\cdot}2n \lg n$, in fewer than one percent of the test cases with `long`-size keys and fewer than two percent overall. The number never exceeded $1{\cdot}5n \lg n$. The most consistently adverse tests were reversed shuffles of `double`s.

COMPARING SORTS

Compared to its two main competitors, the Seventh Edition and Berkeley `qsort`s, our program is pleasantly short, despite its elaborate partition selection method. The central function comprises only 48 lines of code versus 80 for Seventh Edition and 117 for Berkeley.

† Of course, quadratic behavior is still possible. One can generate fiendish inputs by bugging Quicksort: Consider key values to be unknown initially. In the code for selecting a partition element, assign values in increasing order as unknown keys are encountered. In the partitioning code, make unknown keys compare high.

Table II

Type	VAX 8550			MIPS R3000		
	7th Edition	Berkeley	New	7th Edition	Berkeley	New
integer	1·25	0·80	0·60	0·75	0·41	0·11
float	1·39	0·89	0·70	0·73	0·43	0·14
double	1·78	1·23	0·91	1·33	0·78	0·19
record	3·24	2·01	0·92	3·10	1·75	0·26
pointer	2·48	2·10	1·73	1·09	0·73	0·41
string	3·89	2·82	1·67	3·41	1·83	0·37

(Overall counts of noncommentary source, pretty-printed by the Unix `cb` utility, are 89, 102 and 153 lines respectively.) Our program exhibits none of the quadratic behavior on simple inputs that afflicts its predecessors.

To assess timing, we ran the programs on computers of disparate architectures: a VAX 8550 and a 40MHz MIPS R3000 (with 64-kilobyte data and instruction caches and a secondary cache of one megabyte), compiling with `lcc` on both. Table II reports times in seconds to sort 10,000 integers chosen randomly from the range 0 to 999,999 on each of the testbed's six input types. Each time is the average of ten runs. On both machines, our `qsort` is strictly faster than the Berkeley function, which is in turn faster than the Seventh Edition function. But how much faster? The running-time improvements vary with both machine and data type, from twenty percent to a factor of twelve. (The biggest factors are due largely to better swapping, as described in the appendix). In any event, the new code represents enough of an improvement to have been adopted in our own lab and by Berkeley.

We considered `qsort` implementations that were not based on Quicksort. P. McIlroy's merge sort has guaranteed $O(n \log n)$ worst-case performance and is almost optimally adaptive to data with residual order (it runs the highly nonrandom certification suite of Figure 1 almost twice as fast as Program 7), but requires $O(n)$ additional memory.[21] An ancient but little known modification of Heapsort due to Floyd uses $n \lg n + O(n)$ comparisons, but requires almost that many swaps, four times as many as Quicksort.[5] While these algorithms beat our `qsort` in some contexts and lose in others, on balance our function remains the general `qsort` implementation of choice.

To illustrate the cost of the general `qsort` interface, and the benefits of the optimizations in Program 7, we specialized it in two ways. The first specialization sorts only pointer-sized objects, thus eliminating the parameter `es` and reducing the cost of swaps. A further specialization sorts only integers, thus eliminating parameter `cmp` and the overhead of calling it. Table III compares these specializations and the basic Program 3 in sorting 100,000 random integers. For the conventional VAX, the only useful specialization is eliminating the overhead of calling `cmp`. For the highly pipelined MIPS, the more useful specialization is simplifying the swap code, which eliminates a conditional branch. Relative to the simple Program 3, the improvements in Program 7 (`med3` and `vecswap`) pay off modestly. This is not surprising, as the improvements were intended to retain efficiency in the face of nonrandom inputs and inputs where comparison is expensive. Random integers make no such demands.

Stylistically, Program 7 has more variables and depends more heavily on macros than we might like. Observing this, Ken Thompson extended Program 4 in a simpler direction that gives highly consistent performance. Thompson's program uses median-of-three partitioning, an inner procedure with function parameters for both swapping and comparison, and iteration instead of tail recursion. It compiles into half the space of Program 7 and runs only a few percent slower on average. On low-entropy inputs, however, Program 7 often beats it

Table III

Sort	CPU Seconds	
	VAX 8550	MIPS R3000
General `qsort`, Program 7	7·24	2·02
Specialized to `es==sizeof(char*)`	7·28	1·74
Specialized to `int`s	3·49	1·63
Basic integer Quicksort, Program 3	4·40	1·77

dramatically. (Comparison counts summed over the certification suite of Figure 1 differed by a factor of 1·5.) Herein lies justification for the complications in Program 7.

CONCLUSIONS

We have added a few new tricks to Quicksort's bag: an adaptive sampling scheme for choosing a partition element, a new partitioning algorithm, fast swapping code, and a more appropriate cost model. We mixed these with a few old tricks, ignored many others, and arrived at the champion Program 7. Many of these techniques apply immediately to other sorting and selection algorithms. The history of the algorithm illustrates lessons that apply well beyond sorting:

Simplicity. The key to performance is elegance, not battalions of special cases. The terrible temptation to tweak should be resisted unless the payoff is really noticeable.

Profiling Tools. A function-time profiler tells us where the CPU cycles go, and a line-count profiler tells us why. A cost model gives us ballpark estimates for key operations. We sometimes used an algorithm animation system to make movies of sort functions.[4]

Testbeds for Timing and Debugging. A tiny driver gives us one glimpse of the program; a more complex testbed exercises it in more interesting ways. The testbeds check correctness, count comparisons and measure CPU times.

Testing and Certification. The correctness and performance of key routines should be validated by certification programs.

ACKNOWLEDGEMENTS

We are grateful for helpful input from Daniel Carrai, Ken Clarkson, Steve Fortune, Chris Fraser, Eric Grosse, Dave Hanson, Andrew Hume, David Johnson, Brian Kernighan, John Linderman, Peter McIlroy, Bob Sedgewick, Ken Thompson, Howard Trickey and Peter Weinberger.

APPENDIX: TUNING THE SWAP FUNCTION

Among all tuning issues, swapping is the most sensitive to differences in hardware and compilers. Table IV shows how widely swapping can vary. The first entry gives times for swapping 4-byte words in line; the other entries give times for subroutines that swap byte strings word-wise and byte-wise. The VAX world is predictable: an inline swap is fastest, and after function call overhead is subtracted, swappint by `char`s take about four times as long as swapping by `long`s. On the MIPS machine, however, the time ratio is about 15. One factor of four is the size ratio of `long` to `char`, and a second factor of almost four is

Table IV

Swap	CPU Microseconds	
	VAX 8550	MIPS R3000
Inline long	0·8	0·6
4 bytes, long	4·9	1·5
4 bytes, char	7·9	14·5
40 bytes, long	16·2	6·5
40 bytes, char	66·9	106·8

due to cache interference from writing a byte into the word to be read next. Since the Seventh Edition and Berkeley `qsort`s always swap `char`-wise, they are quite slow on this machine.

Table IV leads us to prefer in-line swaps and swapping by word-sized chunks. Thus, for the important special case of word-sized objects (typically pointers or integers) we swap in line. For more complicated swaps, we call a function, `swapfunc`, which in turn distinguishes objects that occupy an exact number of properly aligned words. We have chosen `long` as a generally appropriate word size. The kind of swapping is given by a variable, `swaptype`, with value 0 for single-word swaps, 1 for general swapping by words, and 2 for swapping by bytes. The variable is set by a macro, `SWAPINIT`:†

```
typedef long WORD;
#define W sizeof(WORD)    /* must be a power of 2 */
#define SWAPINIT(a, es) swaptype =                 \
    (a-(char*)0 | es) % W ? 2 : es > W ? 1 : 0
```

The `swap` function becomes a macro that chooses between a function call and an in-line exchange.

```
#define exch(a, b, t) (t = a, a = b, b = t)
#define swap(a, b)                                 \
    swaptype != 0 ? swapfunc(a, b, es, swaptype) : \
    (void)exch(*(WORD*)(a), *(WORD*)(b), t)
```

Another macro swaps sequences of records.

```
#define vecswap(a, b, n) if (n > 0) swapfunc(a, b, n, swaptype)
```

The function called by these macros is straightforward.

```
#include <stddef.h>
static void swapfunc(char *a, char *b, size_t n, int swaptype)
{
    if (swaptype <= 1) {
        WORD t;
        for( ; n > 0; a += W, b += W, n -= W)
            exch(*(WORD*)a, *(WORD*)b, t);
    } else {
        char t;
        for( ; n > 0; a += 1, b += 1, n -= 1)
            exch(*a, *b, t);
    }
}
```

† The strange formula to check data size and alignment works even on Cray computers, where plausible code such as `((long)a | es) % sizeof(long)` fails because the least significant part of a byte address occupies the most significant part of a `long`.

As explained in ‘The Opportunity of Equal Elements’, we prefer to store the partition value out of line instead of swapping it to position `a[0]`. This improvement is inconvenient in C unless the element size is fixed, so we adopt it only for the important special case of word-size objects. Variable `pv` is made to point to `a[0]` or to an out-of-line variable `v`, whichever is used. This macro does the setup:

```
#define PVINIT(pv, pm)                          \
    if (swaptype != 0) pv = a, swap(pv, pm); \
    else pv = (char*)&v, v = *(WORD*)pm
```

REFERENCES

1. C. A. R. Hoare, ‘Quicksort’, *Computer Journal*, **5**, 10-15 (1962).
2. R. S. Scowen, ‘Algorithm 271: quickersort’, *Communications of the ACM*, **8**, 669-670 (1965).
3. B. W. Kernighan and M. D. McIlroy (eds), *UNIX Programmer’s Manual, 7th Edition*, Bell Telephone Laboratories, Murray Hill, NJ (1979). Republished by Holt, Rinehart and Winston, 1983.
4. J. L. Bentley, ‘Software exploratorium: the trouble with qsort’, *UNIX Review*, **10**, (2), 85-93 (1992).
5. J. L. Bentley, ‘Software exploratorium: history of a heapsort’, *UNIX Review*, **10**, (8), 71-77 (1992).
6. American National Standards Institute, *American National Standard for Information Systems — Programming Language — C*, ANSI X3.159-1989, New York (1979).
7. R. Sedgewick, ‘Quicksort’, *PhD Thesis,* Stanford University (1975).
8. R. Sedgewick, ‘Implementing quicksort programs’, *Communications of the ACM*, **21**, 847-857 (1978).
9. R. Sedgewick, ‘Quicksort with equal keys’, *SIAM J. Comp*, **6**, 240-267 (1977).
10. R. Sedgewick, ‘The analysis of quicksort programs’, *Acta Informatica*, **7**, 327-355 (1977).
11. R. Sedgewick, *Algorithms in C*, Addison-Wesley, Reading, MA (1990).
12. D. E. Knuth, *The Art of Computer Programming, volume 3: Sorting and Searching*, Addison-Wesley, Reading, MA (1975).
13. R. L. Rivest and D. E. Knuth, ‘Bibliography 26: computer sorting’, *Computing Reviews*, **13**, 283-289 (1972).
14. J. L. Bentley, *Programming Pearls*, Addison-Wesley, Reading, MA (1986).
15. D. E. Knuth, ‘Structured programming with goto statements’, *Computing Surveys*, **6**, 261-301 (1974).
16. J. L. Bentley, B. W. Kernighan, and C. J. Van Wyk, ‘An elementary C cost model’, *UNIX Review*, **9**, (2), 38-48 (1991).
17. C. W. Fraser and D. R. Hanson, ‘A retargetable compiler for ANSI C’, *ACM SIGPLAN Notices*, **26**, (10), 29-43 (1991).
18. J. P. Linderman, ‘Theory and practice in the construction of a working sort routine’, *Bell System Technical Journal*, **63**, 1827-1843 (1984).
19. R. C. Singleton, ‘Algorithm 347: an efficient algorithm for sorting with minimal storage’, *Communications of the ACM*, **12**, 185-187 (1969).
20. B. W. Weide, ‘Space-efficient on-line selection algorithms’, *Computer Science and Statistics: Eleventh Annual Symposium on the Interface*, (1978), pp. 308-312.
21. P. McIlroy, ‘Optimistic sorting and information theoretic complexity’, *Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, Austin, (1993), pp. 467-474.
22. O. Petersson and A. Moffat, ‘A framework for adaptive sorting’, *Proc. Third Scandinavian Workshop on Algorithms and Theory*, O. Nurmi and E. Ukkonen (eds), Springer-Verlag Lecture Notes in Comp. Sci. #621, (1992), pp. 422-433.
23. E. W. Dijkstra, *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs, NJ (1976).
24. L. M. Wegner, ‘Quicksort for equal keys’, *IEEE Transactions on Computers*, **C-34**, 362-367 (1985).
25. N. Wirth, *Algorithms + Data Structures = Programs*, Prentice-Hall, Englewood Cliffs, NJ (1976).
26. D. E. Knuth, ‘The errors of Tex’, *Software—Practice and Experience*, **19**, 607-685 (1989).
27. J. N. Lyness and J. J. Kaganove, ‘Comments on the nature of automatic quadrature routines’, *ACM Transactions on Mathematical Software*, **2**, 65-81 (1976).