Engineering a Sort Function

Jon Bentley
Doug McIlroy
AT&T Bell Laboratories

Outline

The Trouble With Qsort
A Simple Successor
Tuning The Code

The Qsort Function

Nonnegative Integers

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

Fixed-Length Strings

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

Pointers to Strings

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

The average running time is $O(n \log n)$

A Bug: A run that should take a few minutes took a few weeks

A Bug Report from Wilks and Becker

We found that Qsort is unbearably slow on ‘‘organ-pipe’’ inputs like ‘‘01233210’’:

```
main(int argc, char **argv)
{   int n = atoi(argv[1]), i, x[100000];
    for (i = 0; i < n; i++)
        x[i] = i;
    for ( ; i < 2*n ; i++)
        x[i] = 2*n-i-1;
    qsort(x, 2*n, sizeof(int), intcmp);
}
```

Here are the timings on a VAX-8550:

```
$ time a.out 250
0.7u 0.0s 2r     a.out 250
$ time a.out 500
2.8u 0.2s 4r     a.out 500
$ time a.out 1000
11.4u 0.6s 47r   a.out 1000
$
```

This is clearly quadratic behavior — each time we double the input size, the run time goes up by a factor of four.

A Closer Look

A Profiled Run

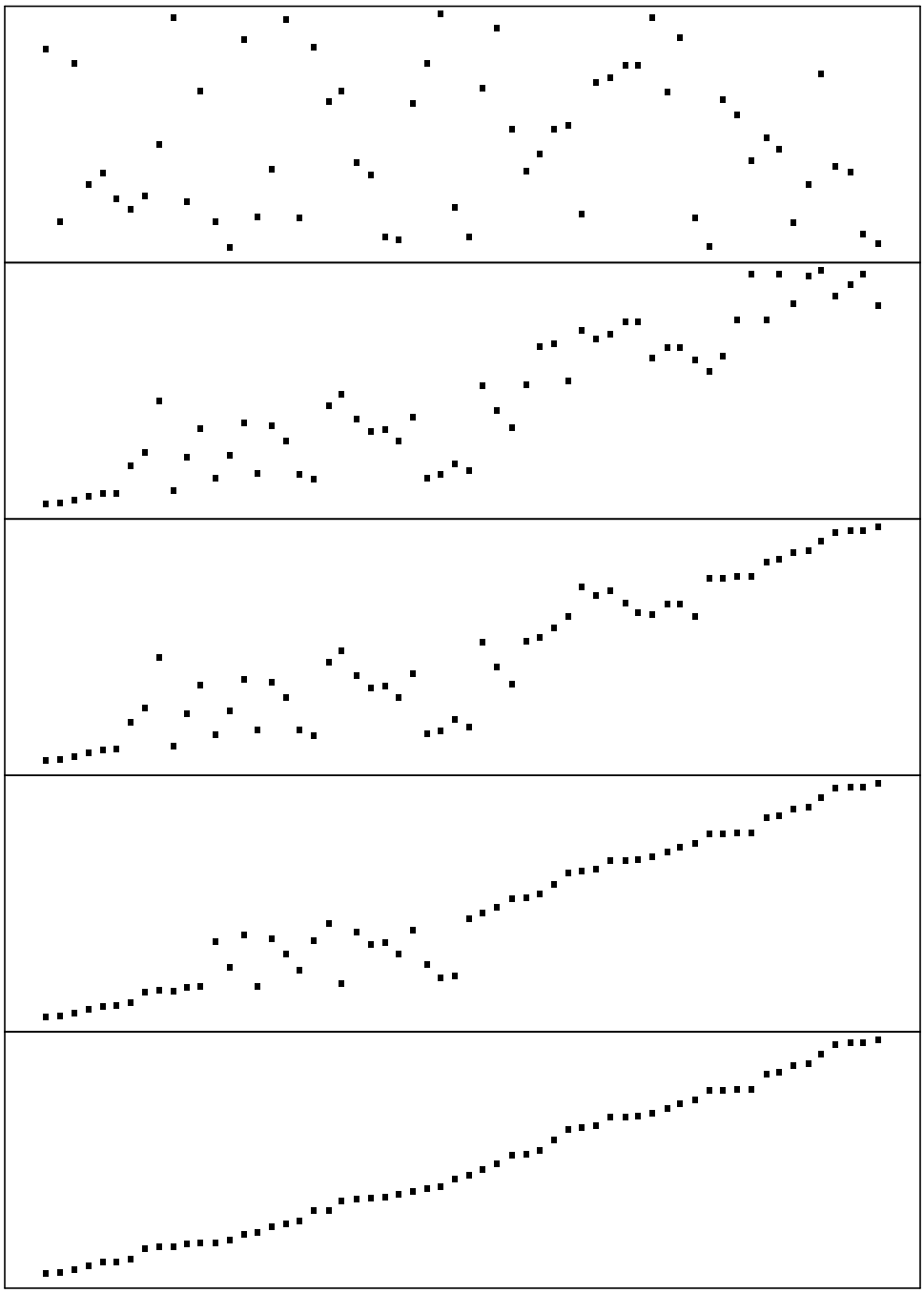
```
$ cc -p qsorttest.c
$ time a.out 100
0.1u 0.0s 0r      a.out 100
$ prof
  %   Time      Calls   Name
87.5   0.117        0   _qsort
12.5   0.017    10000   _intcmp
 0.0   0.000        1   _main
$
```

Other Runs

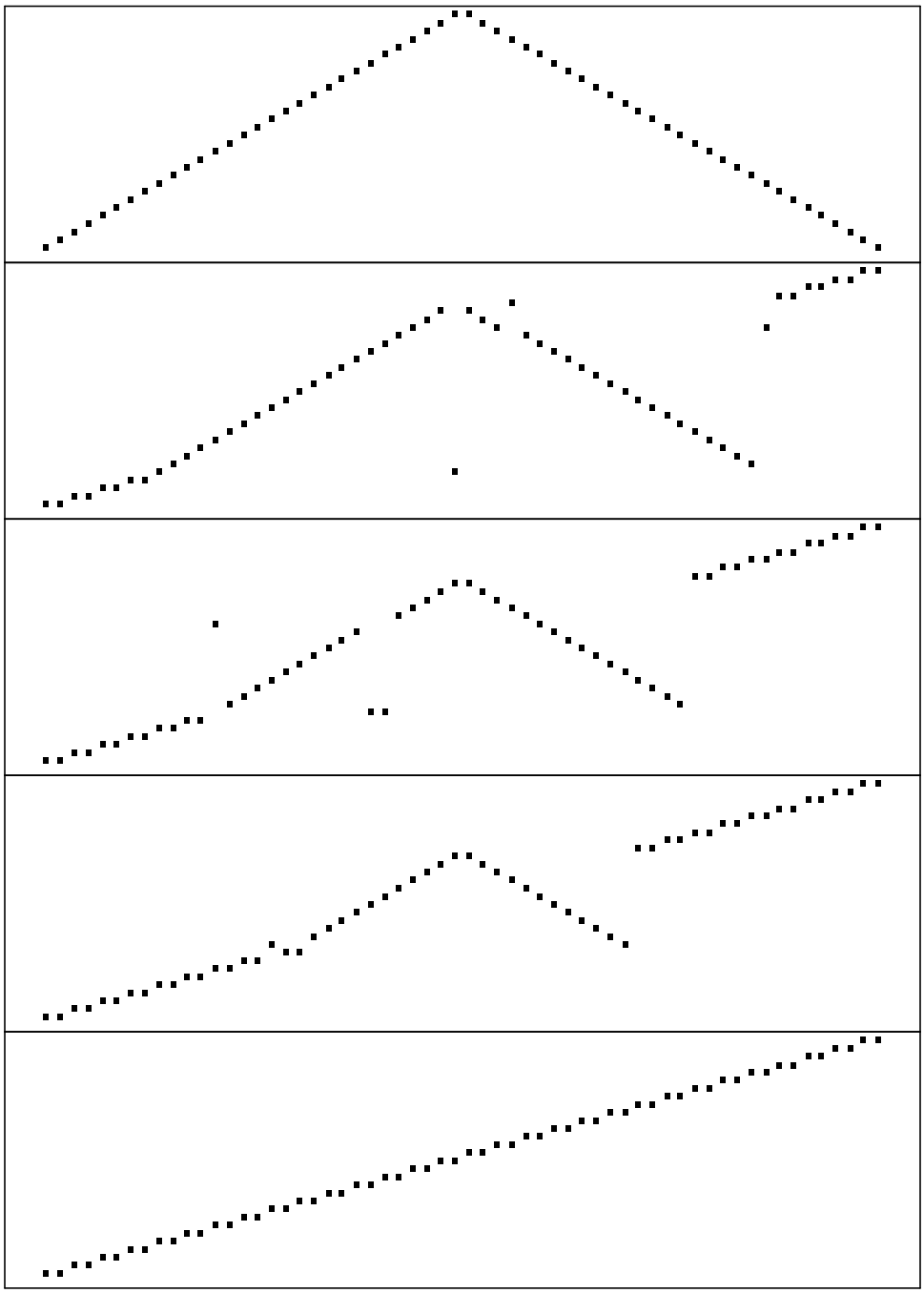
Same results for 200, 300, 400, 500

An Even Closer Look

An animation of `qsort` on random input

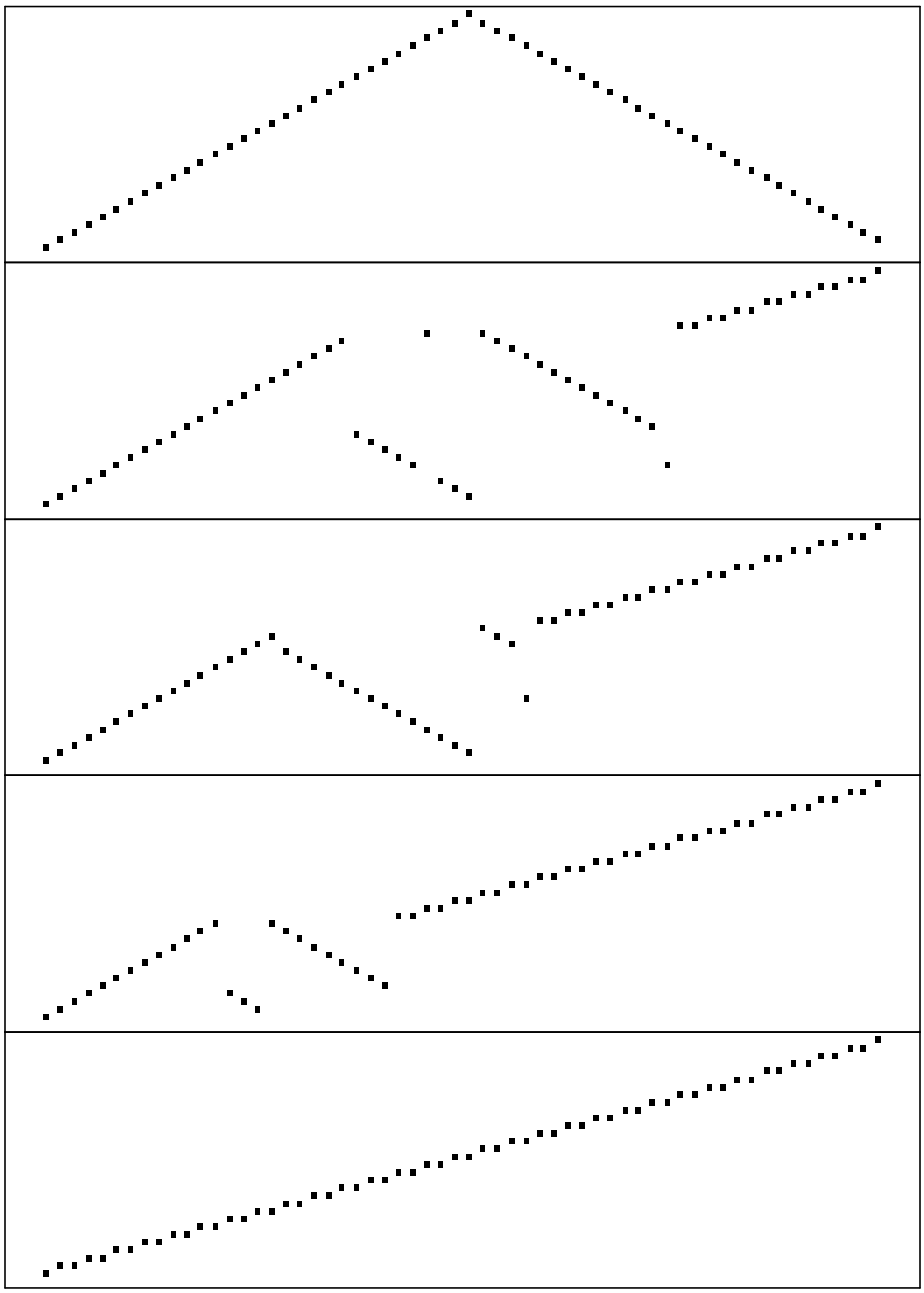


A Quadratic Run



Nudge the Input, Shove the Output

Delete `-1` in `x[i] = 2*n - i - 1;`



Making An Animation

The Animation File

```
x2: text 2 7 dot
```

Animation Code

```
animishow(int i)
{   printf("x%d: text %d %d dot\n",
            i, i, x[i]);
}

animcshow(char *c)
{   animishow((int)(c-animarr)/qses); }
  ...
for (i = 0; i < 2*n; i++)
    animishow(i);
  ...
static qsexc(char *i, char *j)
{   ...
    animcshow(i);
    animcshow(j);
}
```

Goals for a New Qsort

Correctness

- Permutes input into nondecreasing order
- Correct if `cmp` function calls `qsort`

Speed

- Faster on most inputs, preferably near optimal
- $O(n \lg n)$ time on most inputs
- Almost never quadratic worst-case behavior

Memory

- Little stack space, preferably no malloc
- Small object code

Insertion Sort — A Quadratic Algorithm

For Integers

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

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

The Qsort Interface

```
void isort(char *a, unsigned 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);
}
void swap(char *i, char *j, int n)
{
    char c;

    do {
        c = *i;
        *i++ = *j;
        *j++ = c;
    } while (--n > 0);
}
```

Quicksort

A Toy Quicksort, Unfit for General Use

```
void iqsort0(int *a, unsigned 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);
}
```

Time Analysis

Average $C_n \approx 2n \ln n \approx 1.386n \lg n$ comparisons

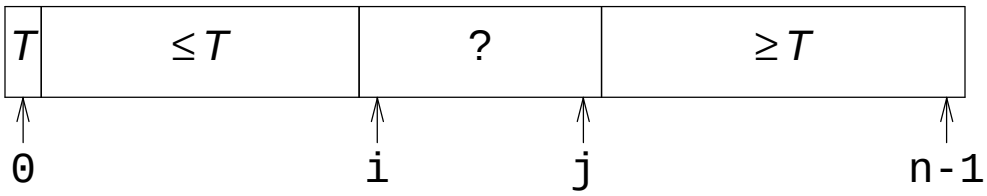
Quadratic for sorted inputs

Quadratic for identical elements

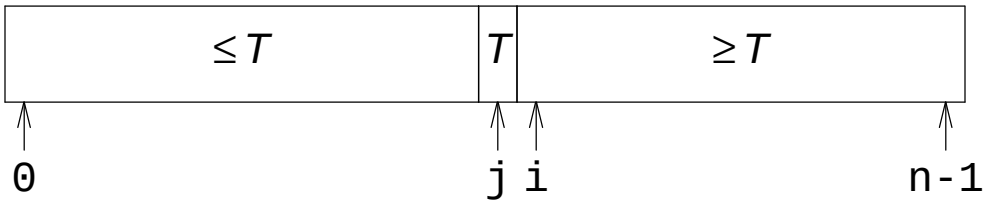
A Useful Qsort

Partitioning Invariants

During the loop



At termination



Integer Quicksort Code

```
void iqsort1(int *a, unsigned 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);
}
```

Simple Quicksort

```
void qsort1(char *a, unsigned 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);
}
```

The Cost of Everything

A program to exercise `qsort1`:

```
#include <stdio.h>
#include "qsort1.c"

int cmp(int *i, int *j) { return *i - *j; }

void main(int argc, char *argv[])
{   int x[100000], i, n = atoi(argv[1]);
    for (i = 0; i < n; i++) x[i] = rand();
    qsort1((char *) x, n, sizeof(int), cmp);
}
```

Profile with n = 20,000:

```
     %   Time      Calls   Name
   31.1   1.067   348683   _cmp
   23.8   0.817    26015   _qsort1
   21.8   0.750    83334   _swap
   12.1   0.417        0   mcount
    5.8   0.200        0   _rand
    2.9   0.100        0   nospace
    1.5   0.050        1   _main
    1.0   0.033        0   urem
```

A Cost Model

	Microsecs
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

Summary: bookkeeping low, comparisons high, swaps huge. After tuning swaps, they become medium.

Comparison to MIX: Bookkeeping and comparisons low, swaps medium.

A Tuned Swap

Set `swaptype` = 0 if one long, 1 if many longs, 2 if chars

```
#define SWAPINIT(a, es) swaptype =                                  \
    (a - (char*) 0) % sizeof(long) || es % sizeof(long) ? 2 : \
    es == sizeof(long) ? 0 : 1;

#define swapcode(TYPE, parmi, parmj, n) {  \
    long i = (n) / sizeof(TYPE);           \
    register TYPE *pi = (TYPE *) (parmi);  \
    register TYPE *pj = (TYPE *) (parmj);  \
    do {                                   \
        register TYPE t = *pi;             \
        *pi++ = *pj;                       \
        *pj++ = t;                         \
    } while (--i > 0);                     \
}

void swapfunc(char *a, char *b, int n, int swaptype)
{   if (swaptype <= 1) swapcode(long, a, b, n)
    else swapcode(char, a, b, n)
}

#define swap(a, b)                         \
    if (swaptype == 0) {                   \
        long t = * (long *) (a);           \
        * (long *) (a) = * (long *) (b);   \
        * (long *) (b) = t;                \
    } else                                 \
        swapfunc(a, b, es, swaptype)
```

Choosing a Partitioning Element

A Bad Idea: First Element

- Quadratic on sorted inputs

Slightly Better: Middle Element

- Quadratic on pipe organ inputs

Worth Considering: Random Element

- $C_n \approx 1.386n \lg n$ expected comparisons for any fixed input
- Implementation issues: system generator?

Median of Three

Idea: Choose median of first, middle and last elements

Reduces comparisons from $1.386n \lg n$ to $1.188n \lg n$

Computing the Median of Three

Unrolled Insertion Sort

```
if (a > b) a :=: b;
if (b > c) b :=: c;
if (a > b) a :=: b;
```

Decision Tree

```
a < b ?
      (b < c ? b : (a < c ? c : a ) )
    : (b > c ? b : (a < c ? a : c ) )
```

Singleton's Scheme

```
pm = med3(a, a + (n/2)*es, a + (n-1)*es);
swap(a, pm);
```

Better Partitions

Pseudomedian of Nine

Take median of three medians of three.

Adaptive Partitioning

Middle element of tiny arrays, median of three for mid-sized, and pseudo-median of nine for large.

```
pm = a + (n/2) * es;
if (n > 7) {
    pl = a;
    pn = a + (n-1) * es;
    if (n > 40) { /* On big arrays, pseudomedian of 9 */
        d = (n/8) * es;
        pl = med3(pl, pl+d, pl+2*d, cmp);
        pm = med3(pm-d, pm, pm+d, cmp);
        pn = med3(pn-2*d, pn-d, pn, cmp);
    }
    pm = med3(pl, pm, pn, cmp); /* On mid arrays, med of 3 *
}
swap(a, pm); /* On tiny arrays, partition around middle */
```

A Timing Testbed

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
```

Implementation — 140 lines of C

`main` — 60. Read input, generate data, time sorts, print output. Check each sort.

Comparison functions — 30.

Random objects — 20.

Header info — 40.

The Opportunity of Equal Elements

A Philosophical Choice

Fast on most inputs; don’t seek special cases

An Early User

New qsort slower than old when many identical elements

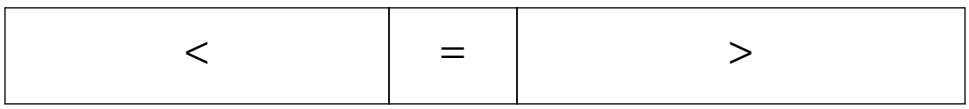
A philosophical excuse

A true response: we can’t replace a utility with an inferior version

An Illustration

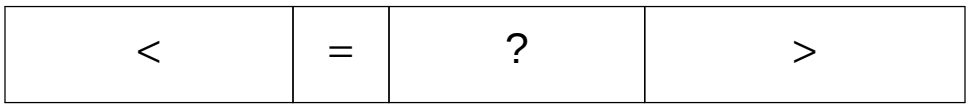
```
q1 100000 i 2 3  768  756  756  7.63
q7 100000 i 2 3  96   93    91  0.94
```

A Common Solution: Fat Partition

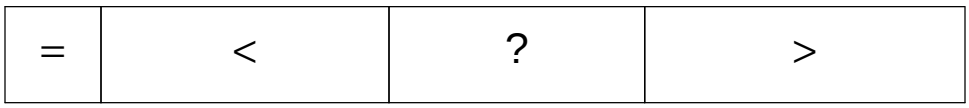


A Sequence of Partitioning Invariants

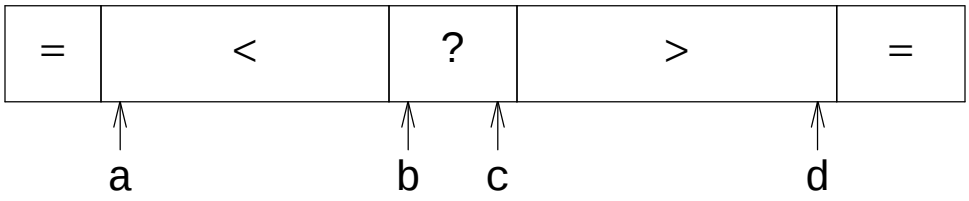
The Classic



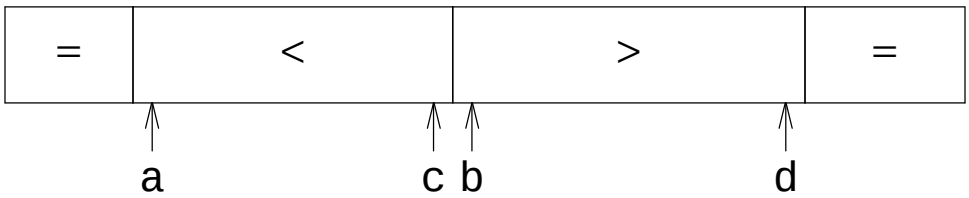
Aha! 1: Swap Two Leftmost



Aha! 2: Symmetry



The Final State



The Current Code

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

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

Time Analysis

- Anecdotes: a few distinct elements
- Theorem: always close to entropy

Other Optimizations

The Current State

- Adaptive sample partitioning element
- New partitioning algorithm

Other Optimizations

- Fast swap
- Insertion sort small inputs
- Guard recursive calls with `if`
- Tune parameters
- Squeeze space with a code space profiler

The Final Qsort

```
void qsort(char *a, unsigned n, int es, int (*cmp)())
{
    char *pa, *pb, *pc, *pd, *pl, *pm, *pn;
    int  d, r, swaptype;
    SWAPINIT(a, es);
    if (n < 7) { /* Insertion sort on small 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;
    if (n > 7) {
        pl = a;
        pn = a + (n-1) * es;
        if (n > 40) { /* On big arrays, pseudomedian of 9 */
            d = (n/8) * es;
            pl = med3(pl, pl+d, pl+2*d, cmp);
            pm = med3(pm-d, pm, pm+d, cmp);
            pn = med3(pn-2*d, pn-d, pn, cmp);
        }
        pm = med3(pl, pm, pn, cmp); /* On mid arrays, med of 3 */
    }
    swap(a, pm); /* On tiny arrays, partition around middle */
    pa = pb = a + es;
    pc = pd = a + (n-1)*es;
    for (;;) {
        while (pb <= pc && (r = cmp(pb, a)) <= 0) {
            if (r == 0) { swap(pa, pb); pa += es; }
            pb += es;
        }
        while (pb <= pc && (r = cmp(pc, a)) >= 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;
    r = min(pa-a,  pb-pa);    vecswap(a,  pb-r, r);
    r = min(pd-pc, pn-pd-es); vecswap(pb, pn-r, r);
    if ((r = pb-pa) > es) qsort(a, r/es, es, cmp);
    if ((r = pd-pc) > es) qsort(pn-r, r/es, es, cmp);
}
```

Optimization Roads Not Traveled

Saving Space

- Handmade stack, smaller subfile first

Sentinels

One Final Insertion Sort

Binary Insertion Sort

Loop Unrolling

Tuning Innermost Loops

Certifying Performance

The Problem

Seventh Edition Qsort: Quadratic on pipe organ inputs

Berkeley Qsort: Quadratic on random zeroes and ones

```
qu  250 i 2 3    4    4    3   30.69
qu  500 i 2 3   16   15   15   57.01
qu 1000 i 2 3   60   58   58   98.11
qu 2000 i 2 3  214  214  211  161.87
qu 4000 i 2 3  859  861  877  301.44
```

A New Qsort

The Solution — A Certification Program

Base inputs: `x[i] = i % m, rand() % m;`, m = 1, 2, 4, ... n

Transformations: reverse all, reverse half, dither, ...

Results

Battle of the Qsorts

Type	VAX 8550			MIPS R3000		
	7th Ed	UCB	New	7th Ed	UCB	New
integer	1.25	.80	.57	.97	.53	.17
float	1.39	.89	.68	.96	.56	.21
double	1.78	1.23	.88	1.67	1.01	.27
record	3.24	2.01	.88	3.89	2.20	.28
pointer	2.48	2.10	1.70	1.36	.93	.52
string	3.89	2.82	1.62	4.17	2.31	.45

Choosing A Sort

Competing Algorithms

- Old Qsorts — Seventh Edition, UCB
- Msort — Malloc 3*n* extra words, but stable
- Hsort — One element of extra storage, fast, many moves
- Sorts that exploit order
- String sorts

A Simple Runoff

```
q7 10000 i 0 3  74  71  73  9.11
m  10000 i 0 3  58  55  56  7.07
q1 10000 i 0 3  55  58  55  7.02
qu 10000 i 0 3  47  47  48  5.94
h  10000 i 0 3  47  47  47  5.90
q  10000 i 0 3  33  36  37  4.43
```

Principles

Simplicity

Profiling Tools

Testbeds for Timing and Debugging

Testing and Certification