Software Exploratorium

HISTORY OF A HEAPSORT
BY JON BENTLEY

For a long time, all was quiet on the sorting front. When I needed to sort an array, I routinely used the `qsort` library function. The February column told some bad news: my favorite `qsort`, written right here at Bell Labs, goes into an (almost) infinite loop on some reasonable inputs. The first part of this column gives even worse news: a popular Berkeley `qsort` blows it just as badly. Writing a good sort function, it seems, is harder than I thought.

But there is light at the end of the tunnel. The old, trusted Heapsort guarantees good worst-case performance, at the cost (I thought) of taking twice as much time on the average. But an ancient and little-known trick of Bob Floyd's allows us to halve the number of comparisons made by Heapsort, which makes it pretty competitive with Quicksort in average running time. This column starts with the bad news, and works its way to a new `qsort` that might be useful in some applications.

More Trouble With Qsort

The February column tells how Allan Wilks and Rick Becker discovered trouble in the `qsort` distributed with the Seventh Edition UNIX System. On a random array of n elements, the program makes $O(n \lg n)$ comparisons. On ''organ-pipe'' inputs like ''01233210'', though, the program makes about n^2 comparisons. For sorting half a million elements, a sort that should take a few minutes instead takes a few weeks.

I looked at several other `qsorts`, and found that they were all derived from code written at Berkeley in 1983. I had high hopes that some variant of this function would meet our needs. Those hopes were eventually dashed by this trivial program, which sprinkles an array of n elements with random zeroes and ones, and then calls `qsort`:

```
int intcmp(int *i, int *j) { return *i - *j; }
main(int argc, char *argv[])
{   int n = atoi(argv[1]), i, x[10000];
    for (i = 0; i < n; i++) x[i] = rand() % 2;
    qsort(x, n, sizeof(int), intcmp);
}
```

When I ran the program, I found that doubling the input size quadruples the run time, which is the signature of a quadratic algorithm:

```
$ time a.out 1000
0.9u 0.0s 0r     a.out 1000
$ time a.out 2000
3.7u 0.0s 4r     a.out 2000
$ time a.out 4000
14.2u 0.2s 19r   a.out 4000
$
```

‘‘Fool me once, shame on you; fool me twice, shame on me.’’ Quicksort is a great algorithm, on the average, but sometimes you want guarantees. It was time to look at sorting algorithms with good worst-case running time.

Exercise 1. Test your system `qsort` on nasty inputs to see if it takes quadratic time.

The Heapsort We Love

Heapsort seems like the right tool for the job. It is easy to code, it doesn't require extra storage, and it never takes more than $O(n \lg n)$ comparisons. This sorting method is described in most algorithms texts, so I'll just sketch it here.

The key to Heapsort is to interpret the array a as a heap data structure, which is an implicit binary tree. The root of the tree is `a[1]`, and its two children are `a[2]` and `a[3]`. In general, the node `a[i]` has the two children `a[2*i]` and `a[2*i+1]`, and its parent is `a[i/2]`. A complete array is a heap if the value in each node is not less than the values in its two children. The subarray `a[l..u]` has the heap property if every node in that range has a value greater than the value of its children. We write this as `heap(l,u)` and define it formally as $\forall\ i\colon 2l \le i \le u\colon a[i/2] \ge a[i]$.

The key function in Heapsort is `siftdown`. If we have the property `heap(l+1,u)`, the call `siftdown(l,u)` will sift the element in `a[l]` down† the tree to establish `heap(l,u)`. This pseudocode implementation of `siftdown(l, u)` is taken from the Further Reading; it uses the ‘‘`:=:`’’ operator to swap elements.

† Throughout this column, we'll sift an element ‘‘down’’ the tree, from the root at the top towards the leaves at the bottom. Beware that some authors refer to this as sifting an element ‘‘up’’ from a lower-numbered position in the array to a higher-numbered position.

```
i := l
loop
    /* Invariant: Heap(l, u) except perhaps
        between i and its (0, 1 or 2) children */
    c := 2*i
    if c > u then break
    if c+1 <= u and a[c+1] > a[c] then c++
    /* c is the greater child of i */
    if a[i] >= a[c] then break
    a[c] :=: a[i]
    i := c
```

The integer `i` is the node we're currently visiting, and `c` is the greater child of `i`. We always swap `i` with its greater child, and terminate the loop when either `c` goes out of range or the value of node `i` is greater than the value of its child `c`.

Heapsort has two simple loops. The first `for` loop uses `siftdown` to build the heap from the bottom up. The second `for` loop repeatedly swaps the current largest element to the end of the array and uses `siftdown` to regain the heap:

```
for j := n/2 downto 1 do
    siftdown(j, n)
for j := n downto 2 do
    a[1] :=: a[j]
    siftdown(1, j-1)
```

Exercise 2. Prove that the first loop makes $O(n)$ comparisons and that the second loop makes at most $2n \lg n$ comparisons.

Heapsort in C

We'll start with an integer version of Heapsort. The function

```
void ihsort(int *a, int n);
```

sorts the array `a` of `n` integers. Restricting the sort to integers makes it easier to understand and faster on this common case; we'll soon see the complete `qsort` interface. Heapsort assumes that the array is indexed `a[1..n]`, while C's zero-based arrays are indexed `a[0..n-1]`. We'll hack around this discrepancy by decrementing `a` at the start of the program, which usually works correctly.

Exercise 3. Explain the problems that may arise in decrementing a pointer to the start of an array and how to solve them.

With this background, it is easy to write the main function:

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

    a--;
    for (j = n/2; j >= 1; j--)
        siftdown(j, n, a);
    for (j = n; j >= 2; j--) {
        swap(1, j, a);
        siftdown(1, j-1, a);
    }
}
```

It is straightforward to translate the pseudocode for `siftdown` into a C `for` loop:

```
void siftdown(int l, int u, int *a)
{   int i, c;

    for (i = l; (c = 2*i) <= u; i = c) {
        if (c+1 <= u && a[c+1] > a[c]) c++;
        if (a[i] > a[c]) break;
        swap(i, c, a);
    }
}
```

We also need a swap function:

```
void swap(int i, int j, int *a)
{   int t = a[i]; a[i] = a[j]; a[j] = t; }
```

That's all there is to an integer version of Heapsort: 20 lines of C gives us a sort that never uses more than about $2n \lg n$ comparisons. That's a guarantee — we won't lose sleep worrying about nasty inputs.

The Way To Up Is Down

The only bad news about Heapsort is that it uses $2n \lg n$ comparisons on most inputs, while good Quicksorts use about half as many, on the average. Typical Quicksorts are therefore usually twice (or more) as fast as Heapsorts. But in Exercise 5.2.3-18 of *Sorting and Searching*, Knuth describes a technique that Bob Floyd used to reduce the number of comparisons made by Heapsort to just $n \lg n$, on the average.

When `siftdown` moves an element down the tree, it makes two comparisons at each level, and on the average goes down almost all of the $\lg n$ levels of the tree. Thus each of the n `siftdowns` has an average cost of about $2 \lg n$ comparisons. Floyd observed that it is more efficient to push the element all the way down to the bottom of the tree making just one comparison at each level, and then let it sift back up as far as it needs to. Knuth's experiments show that about 84.8 percent of the time the element will stay at the bottom level, 13.5 percent of the time it will move up one level, and 1.6 percent of the time it will move up just two levels. Floyd's technique is a huge win.

The pseudocode to sift up the node at position `i` uses `p` to represent the parent of `i`, which is in location `i/2`. It repeatedly swaps `i` with its parent until either the parent goes out of range or it has a greater value:

```
loop
    p := i/2
    if p < l then break
    if a[p] > a[i] then break
    a[p] :=: a[i]
    i := p
```

The new `siftdown` removes one test in the first loop and adds a second loop, so the body of `siftdown` becomes:

```
for (i = l; (c = 2*i) <= u; i = c) {
    if (c+1 <= u && a[c+1] > a[c]) c++;
    swap(i, c, a);
}
for ( ; (p = i/2) >= l; i = p) {
    if (a[p] >= a[i]) break;
    swap(p, i, a);
}
```

Exercise 4. Experiment to find how many comparisons this algorithm uses; fit the results to the function $an \lg n + bn$.

The Qsort Interface

So far we've sorted an array of integers; we'll now build a Heapsort that implements the standard `qsort` interface, which we glimpsed earlier. We have to add two additional parameters: `es` (for element size) tells the `sizeof` each element, and the `cmp` function compares two elements.

The basic algorithms remain the same; we just have a little more bookkeeping to deal with arbitrary elements rather than integers. We'll start with the `swap` function, which must now exchange two arbitrary elements of size `es`. This function does the work:

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

We'll hide some messy details inside a macro:

```
#define swap(i, j) swapfunc(a+(i)*es, a+(j)*es, es)
```

With `swap` as a warmup, the `hsort` function itself is easy. We add the two new parameters, declare some new integers, and decrement the array name by `es` rather than by one:

```
void hsort(char *a, int n, int es, int (*cmp)())
{   int j, i, c, p, u;

    a -= es;
    for (j = n/2; j >= 1; j--)
        siftdown(j, n);
    for (j = n; j >= 2; j--) {
        swap(1, j);
        siftdown(1, j-1);
    }
}
```

The remaining part is the `siftdown` function. We'll start off with the (slightly premature) optimization of making it a macro, rather than a function; this makes the program a little faster, and also avoids having to pass `a`, `es` and `cmp` as parameters. The `siftdown` macro is a straightforward modification of the function, with the comparisons now made through the `cmp` function:

```
#define siftdown(l, u) {                                    \
    for (i = l; (c = 2*i) <= u; i = c) {                    \
        if (c+1 <= u && cmp(a+(c+1)*es, a+c*es) > 0)        \
            c++;                                            \
        swap(i, c);                                         \
    }                                                       \
    for ( ; (p = i/2) >= l; i = p) {                        \
        if (cmp(a + p*es, a + i*es) >= 0) break;            \
        swap(p, i);                                         \
    }                                                       \
}
```

Moving Swaps

The complete `hsort` requires about thirty lines of C to implement the `qsort` specification. It is guaranteed to run in $O(n \lg n)$ time on all inputs, but it is a little slower than Quicksort (about fifty percent slower than the Seventh Edition `qsort`, and about twice the run time of the Berkeley `qsort`).

The time has come to tune this code, so the first step is to use "`cc -p`" to profile the driver we saw earlier (after deleting "`% 2`"). This run sorts $n = 10{,}000$ random integers, so $n \lg n$ is about 133,000:

```
  %   Time      Calls   Name
43.7   1.150   132805   _swapfunc
31.0   0.817        1   _hsort
14.6   0.383   136718   _intcmp
 5.1   0.133        0   mcount
 1.9   0.050        0   _rand
 1.9   0.050        0   nospace
 1.9   0.050        1   _main
```

The time hog here is clearly the `swap`. Our Heapsort uses roughly $n \lg n$ comparisons and about the same number of swaps. Each swap costs about 9

microseconds, while a call to the integer comparison function costs about 3 microseconds.

When we look at how `siftdown` uses `swap`, we see that it repeatedly swaps the same element down the heap. We would do better to store that element in a safe `buf`, move up each element in the path to the root, and then copy the element from `buf` to the bottom of the tree.

Exercise 5. A prototype `hsort` can allocate `buf` to be of some fixed (but large) size. How should a production version handle the case that `es` is even larger?

We can now replace the `swaps` with calls to a new `copy` function, so the first loop in `siftdown` is replaced by this code:

```
copy(buf, a + (l)*es);                              \
for (i = l; (c = 2*i) <= u; i = c) {                \
    if (c+1 <= u && cmp(a+(c+1)*es, a+c*es) > 0) \
        c++;                                        \
    copy(a + i*es, a + c *es);                      \
}                                                   \
copy(a + i*es, buf);                                \
```

A simple and reasonably efficient implementation of `copy` uses a standard C library function:

```
#define copy(i, j) memcpy((i), (j), es)
```

We can squeeze even more speed out of the function by exploiting the common case in which we are sorting an array of pointers or `longs`. Rather than moving a pointer by moving each of the four bytes in a loop, we'll move it in a single instruction. We'll set the `int copytype` to be zero if the array is aligned as `longs` and `es` is exactly `sizeof(long)`:

```
copytype = (a - (char*) 0) % sizeof(long)
           ||  es != sizeof(long);
```

The faster `copy` is contained in Program 1.

Exercise 6. Explain why setting `copytype` might not be portable. Explore other solutions to the problem of reducing the time Heapsort spends in swaps.

Choosing A Sort

The two speedups in the last section combine to give a speedup of about forty percent. There is, however, more fat to be trimmed.

Exercise 7. Tune the current version of Heapsort to further reduce its run time.

A relatively efficient `hsort` is presented as Program 1; it is about twice as fast as our original code. On arrays of random integers, it is about the same speed as the Berkeley `qsort`.

Program 1 isn't the perfect sorting algorithm: it uses a buffer of `es` extra bytes, it isn't stable, and Quicksort is

```
/* hsort -- Heapsort implementation of qsort interface  */

void swapfunc(char *i, char *j, int n)
{   do {
        char c = *i;
        *i++ = *j;
        *j++ = c;
    } while (--n > 0);
}
#define swap(i, j) swapfunc(a + (i)*es, a + (j)*es, es)

#define copy(i, j) { if (copytype == 0)                 \
                        *(long *)(i) = *(long *)(j);    \
                     else                               \
                        memcpy((i), (j), es); }

#define siftdown(l, u) {                                \
    pu = (u) * es;                                      \
    copy(buf, a + (l)*es);                              \
    for (pi = (l)*es; (pc = 2*pi) <= pu; pi = pc) {     \
        if (pc < pu && cmp(a+pc+es, a+pc) > 0)          \
            pc += es;                                   \
        copy(a + pi, a + pc);                           \
    }                                                   \
    copy(a + pi, buf);                                  \
    for (i = pi / es; (p = i/2) >= l; i = p) {          \
        if (cmp(a + p*es, a + i*es) >= 0) break;        \
        swap(p, i);                                     \
    }                                                   \
}

#define MYBUFSIZE 1024
void hsort(char *a, int n, int es, int (*cmp)())
{   int  j, i, c, p, u, copytype, pi, pu, pc;
    char *buf, mybuf[MYBUFSIZE], *malloc();

    buf = mybuf;
    if (es > MYBUFSIZE && !(buf = malloc(es))) {
        qsort(a, n, es, cmp);
        return;
    }
    copytype = (a - (char*) 0) % sizeof(long)
              || es!= sizeof(long);
    a -= es;
    for (j = n/2; j >= 1; j--)
        siftdown(j, n);
    for (j = n; j >= 2; j--) {
        swap(1, j);
        siftdown(1, j-1);
    }
    if (es > MYBUFSIZE)
        free(buf);
}
```

Program 1. A complete `hsort` function.

much faster if there are many duplicated elements. Heapsort makes about $n \lg n$ moves, while Quicksort makes only about a quarter that many swaps. But Program 1 does have a great worst-case bound, and Floyd's trick makes it pretty competitive with Quicksort on the average. Programmers should consider using it, and system administrators might even consider whether this

`hsort` should replace the current `qsort`.

Exercise 8. Implement Program 1 on your system and compare its run time to the system `qsort`.

Principles

Simplicity. Gordon Bell observes that ‘‘The cheapest, fastest and most reliable components of a computer system are those that aren’t there.’’ The same is true of programs.

History. The best way to describe some complex programs is to start with a simple version and then apply a sequence of straightforward transformations.

The True Path to Performance. Adding bells and whistles sometimes makes programs faster, but often makes them slower. Substantial speedups, like halving the number of comparisons, usually come from a new way of looking at the problem. Floyd saw that we can save work by going all the way to the bottom of the tree then retracing our steps back up.

Further Reading

This Heapsort algorithm was originally described by Bob Floyd as ‘‘Algorithm 245: Treesort 3’’ in *Communications of the ACM 7*, 12, December 1964, p. 701. His wonderful paper presents a fundamental algorithm in less than a third of a page, complete with incredibly elegant code. A quarter of a century later, this gem is still a delight to read.

For a longer introduction to Heapsort, try Column 12 of my *Programming Pearls*, published by Addison-Wesley in 1986. It is the source of the pseudocode in this column. It uses techniques of program verification to derive Heapsort, and also describes how the heap data structure can be used to implement priority queues.

Solutions to Selected Exercises

3. Decrementing a pointer to the start of an array presents no problem on many machines. On a segmented architecture, though, a decrement could generate an illegal address. ANSI Standard C therefore does not allow such operations. We could patch our code by subtracting one from a in each reference; this is hard now, and will become harder later. An alternative solution is given in Solution 6.

4. A simple experiment set n to the powers of 2 from 128 to 65536, then sorted eleven sets of n random integers at each value. A weighted least-squares regression showed that the average number of comparisons was $1.0016n \lg n + .31n$ — that was close enough to one for me.

5. We will still keep our own buffer, so we can quickly sort elements of reasonable size. When `es` exceeds our buffer size, we attempt to `malloc` a larger buffer; if that

fails, we punt to a standard `qsort` (or any other appropriate sort). At the end of the sort, we free the buffer. Program 1 implements this solution.

6. The test that sets `memtype` fails if a `long *` points to the last byte rather than the first byte in a word. We can sometimes reduce the time spent in swapping by allocating a vector of $n+1$ pointers (we'll avoid the problems of Exercise 3 by ignoring the first element of the array). We initialize the i^{th} pointer to the i^{th} element in the array to be sorted, apply our sorting algorithm to the pointer vector (where swaps are fast, but each comparison requires an additional level of indirection), and at the end of the algorithm we permute the input vector.

7. Program 1 avoids multiplication by `es` by keeping the variables `pi`, `pu` and `pc` to be a `es` times the variables `i`, `u` and prog(c), respectively. On some machines and inputs, the pointer array in Solution 6 represents a substantial speedup.

Jon Bentley is a Member of Technical Staff in the Computing Science Research Center at AT&T Bell Laboratories.