Dr.Dobb's
THE WORLD OF SOFTWARE DEVELOPMENT

Sorting Strings with Three-Way Radix Quicksort

Jon and Robert describe a new algorithm for sorting strings that combines the best of quicksort and radix sort.

November 01, 1998
URL:http://www.drdobbs.com/database/sorting-strings-with-three-way-radix-qui/184410724

Jon is a member of technical staff at Bell Labs. Robert is the William O. Baker Professor of Computer Science at Princeton University. They can be reached at jlb@ research.bell-labs.com and rs@cs.princeton .edu, respectively.

Quicksort is a champion all-around sorting algorithm. Radix sort, however, is often faster for sorting strings because it decomposes a string into characters. This month, we'll examine a "three-way radix quicksort" algorithm that applies the general approach of quicksort character-by-character. A simple implementation of the algorithm is competitive with the most-efficient string-sorting programs we know.

We first described the algorithm in a 1997 paper entitled "Fast Algorithms for Sorting and Searching Strings" (*Proceedings of the 8th Annual ACM-SIAM Symposium on Discrete Algorithms,* January 1997). This "SODA" paper gives the theory behind the algorithm and extensive references. It also describes the algorithm's relationship to a data structure for storing strings. For more information, see "Ternary Search Trees," by Jon Bentley and Robert Sedgewick (*DDJ*, April 1998).

But why do you need yet another sorting function? Isn't the standard library *qsort* good enough? On most systems, it is indeed an efficient and powerful general-purpose tool. But the power of the general *compare* function comes at a price in performance. Experiments we've conducted show that our specialized string sort can be faster than *qsort by* a factor of four. If your strings contain 16-bit Unicode characters rather than standard eight-bit characters, the savings might be even more dramatic.

Three-Way Quicksort

Quicksort was first described by C.A.R. Hoare in 1962 (see "Quicksort," *Computer Journal 5*, 1, 1962). It is a textbook divide-and-conquer algorithm. To sort an array, you choose a partitioning element, permute the elements, placing smaller elements on one side and larger elements on the other, and then recursively sort the two subarrays. But what happens to elements equal to the partitioning value? Hoare's method uses two-way partitioning: As Figure 1(a) illustrates, it places lesser elements on the left and greater elements on the right, but equal elements may appear on either side. A three-way quicksort partitions the elements to leave equal items in the middle, as in Figure 1(b).

Once we have achieved this state, we can recur on the left and right subarrays and ignore the middle elements (which are already in place). Figure 2 is a sketch of this in pseudocode, with a *qs* function that sorts the sequence *s* of length *n*. An efficient implementation of three-way partitioning (see "Engineering a Sort Function," by J.L. Bentley and M.D. McIlroy, *Software: Practice and Experience 23*, 1, 1993) uses the counterintuitive loop invariant in Figure 3(a).

You may think of the index *lt* as pointing above the elements that are less than the partitioning value *v*, and of *le* as indexing the equal elements that happen to fall on the lesser side. Similarly, *gt* indexes the elements that are greater than *v*, and *ge* indexes the equal elements on the greater side. The main partitioning loop has two inner loops. The first inner loop moves *lt* up: It scans over lesser elements, swaps equal elements to *le*, and halts on a greater element:

```
for ( ; lt <= gt && a[lt] <= v; lt++)

if (a[lt] == v)

swap(a, le++, lt);
```

The *swap* function exchanges two elements of its first argument; we'll see its implementation shortly. The second inner loop moves *gt* down correspondingly. The main loop then swaps the elements to which *lt* and *gt* point, increments *lt* and decrements *gt*:

```
swap(a, lt++, gt--);
```

The loop terminates when *lt* and *gt* cross.

After the loop ends, the array is in the state shown in Figure 3(b). At this time, the equal elements on the edges are swapped to the middle of the array.

There are *lt-le* lesser elements, and *le* equal elements on that side. We will use a vector swap function to move the smaller of those two sequences:

```
r = min(le, lt-le);

vecswap(a, 0, lt-r, r);
```

Similar code swaps the equal elements on the high end of the array. Once the equal elements are in their proper places, we can recur:

```
iqs(a, lt-le);

iqs(a + n-(ge-gt), ge-gt);
```

The first call sorts the first *lt-le* (lesser) elements at the bottom of array *a*. The second call sorts the *ge-gt* (greater) elements at the top of the array.

Example 1 illustrates this algorithm with three-way quicksort code to sort an array of integers. The *swap* statement in the fifth line randomly chooses a partition value. The *swap* function exchanges two integers:

```
void swap(int a[], int i, int j)

{ int t = a[i];

a[i] = a[j];

a[j] = t;

}
```

To move the equal elements into the middle, we will employ a vector swap function that exchanges the sequences of *n* elements that begin at positions *i* and *j*:

```
void vecswap(int a[], int i, int j, int n)

{ while (n-- > 0)

swap(a, i++, j++);

}
```

Radix Sort

Suppose that you want to sort a scrambled dictionary of 10,000 English words, each one of which is represented on a 3×5-inch index card. Your first step might be to put the cards into 26 smaller piles, one for the words that begin with the letter "a," one for "b," and so forth. You then sort each pile, perhaps using a similar approach but looking at the second letter of each word. After all 26 piles are sorted, you concatenate them to form the final sorted list.

This is the idea underlying radix sort. To sort a set of strings, we partition the set into "piles" based on their first characters. For an eight-bit character set, we usually implement the piles with an array of length 256. Radix sort then recurs on each pile, and finally concatenates the sorted piles to form the output. Turning this sketch into an efficient program requires careful choice of data structures (see "Engineering Radix Sort," by P.M. McIlroy, K. Bostic, and M.D. McIlroy, *Computing Systems 6*, 1, 1993). Radix sort can be blazingly fast because it inspects each character of each word at most once.

An Algorithm for Sorting Strings

A standard way to sort an array of pointers to strings in C is to call the *qsort* library function (historically implemented by quicksort); one of the parameters to the *qsort* function is the *strcmp* function to compare strings. This method does not exploit any structural properties of string keys. When it is used for huge files and long strings, it accesses the characters at the beginning of the strings much more frequently than necessary. We'll now turn to an algorithm that cuts down the number of characters accessed to nearly the absolute minimum used by radix sort.

Building on an insight of P. Shackleton, Hoare sketched a quicksort modification for sorting multiword keys (strings may be viewed as keys composed of many characters):

> When it is known that a segment comprises all the items, and only those items, which have key values identical to a given value over the first *n* words, in partitioning this segment, comparison is made of the *(n*+1)th word of the keys.

We will use this idea to sort a set of C strings. Like regular quicksort, the algorithm partitions its input into sets less than, equal to, and greater than a given value. Like radix sort, when the current input contains equal initial characters, the algorithm moves on to the next character.

Three-way partitioning is the key to our implementation of Hoare's multiword quicksort. This recursive pseudocode in Figure 4 sorts the sequence *s* of length *n* that is known to be identical in characters *0..depth*-1; it is originally called as *ssort(s, n, 0)*.

A Simple String Sort Implementation

We will now implement the three-way quicksort algorithm as a C function to sort strings. The primary sort function:

```
void ssortmain(char *a[], int n)

{ ssort(a, n, 0); }
```

is passed the array *a* of *n* pointers to character strings; its job is to permute the pointers so that the strings occur in lexicographic order. The *ssort* function in Example 2 is a straightforward extension of Example 1. It is passed both *a* and *n*, and the additional integer *depth* to tell which characters are to be compared. The algorithm terminates either when the vector contains at most one string or when the current depth "runs off the end" of a string by encountering a terminating null character. The *ssort* function uses several supporting functions; Listing One (at the end of this article) contains the complete code. As in Example 1, *swap* exchanges a pair of vector elements, and *vecswap* exchanges a sequence of elements. The *ch* macro (for "character") accesses character *depth* of string *a[i]*:

```
#define ch(i) a[i][depth]
```

This simple code is but one implementation of three-way radix quicksort. *Algorithms in C*, Third Edition, by Robert Sedgewick (Addison-Wesley, 1998) implements the same loop invariant with substantially different code.

Standard Quicksort Speedups

Although quicksort is efficient for large arrays, simpler algorithms sometimes have less overhead for small arrays. Example 2 uses the termination test:

```
if (n <= 1)

return;
```

We replaced it with the standard speedup of terminating the recursion by switching to an insertion sort for small arrays (the constant 10 was determined experimentally):

```
if (n <= 10) {

inssort(a, n, depth);

return;

}
```

Because we know that all strings are equal through *depth* characters, the insertion sort can start comparisons there. (For this reason, we could not employ the further speedup of using a single insertion sort after the original quicksort.) The nine-line *inssort* function is in Listing One.

Example 2 selects a random element as the partition value and swaps it to the beginning with the statement:

```
swap(a, 0, rand() % n);
```

Another well-known speedup finds a partitioning value near the center of the set by choosing the median of three elements:

```
pm = med3(0, n/2, n-1);

swap(a, 0, pm);
```

The *med3* function returns the index that has the median value of the three indices; its 10-line implementation is in Listing One. We use an extension that selects the median of three medians-of-three for large arrays:

```
pl = 0;

pm = n/2;

pn = n-1;

if (n > 50) {

d = n/8;

pl = med3(pl, pl+d, pl+2*d);

pm = med3(pm-d, pm, pm+d);
```

```
pn = med3(pn-2*d, pn-d, pn);

}

pm = med3(pl, pm, pn);

swap(a, 0, pm);
```

Another standard quicksort optimization saves space by replacing recursion with an explicit stack and sorting the smaller subfile first. Our string quicksort has excellent stack utilization on the average, and good utilization in the worst case, so we chose to avoid the extra code. We experimented with standard programming speedups and found that most were not needed with modern hardware and optimizing compilers. For instance, without optimization, rewriting the swap function as a macro gave a substantial speedup; with optimization enabled, it made no difference. Common subexpression elimination performed by compilers made it unnecessary to store the result of the character comparisons in the innermost loops. Similarly, we did not need to convert array indices to pointers. On less advanced systems, though, it might be profitable to incorporate such speedups in the source code.

A Speedup for Equal Characters

What happens when the input array contains identical keys at the specified depth? This case is not as far-fetched as it might seem: A few scans over 26-character alphabets quickly pares the input down to single (leading) characters. Example 2 deals with this case rather gracelessly: It performs a *swap* to move each input pointer back to its current position.

Fortunately, we are able to handle this common case efficiently while introducing little overhead. Immediately after the partitioning value *v* is chosen, a new loop moves the *le* index up as far as possible. If it moves all the way to the end of the array, we recur for the next character and *return* from the function. In the very worst case (of the first element being unequal), the new code costs just three extra comparisons. After the loop and test, we set *lt* to *le*, and proceed as before:

```
v = ch(0);

for (le = 1; le < n && ch(le) == v; le++)

;

if (le == n) {

if (v != 0)

ssort2(a, n, depth+1);

return;

}

lt = le;
```

We tested this speedup on the extreme input of 100,000 equal keys, each of which consisted of 20 "0" characters. The new code was 20 percent faster on a MIPS R10000 and 40 percent faster on a Pentium Pro.

Experiments

We have tested many sort algorithms on many string inputs on many computers. Three-way radix quicksort performs well under a broad range of conditions. We sketch one small but representative experiment in this section; additional data may be found in our SODA paper and in Sedgewick's 1998 *Algorithms in C*.

We ran our simple experiments on two machines: a 250-MHz MIPS R10000 and a 200-MHz Pentium Pro. We compiled the identical source code under the highest optimization level available on each machine. We used two input files: one of 100,000 identical keys (each of 20 zeros) and a dictionary word list of 234,936 words (in 2,486,813 characters). We timed four different algorithms: the system *qsort* function calling *strcmp*, Example 2, the tuned version of three-way radix quicksort, and the fastest radix sort we know (from the paper by McIlroy, Bostic, and McIlroy). Table 1 gives the number of seconds required by the various functions.

The one eternal truth contained in this table is that timing algorithms is difficult. The tuned sort is always faster than the simple Example 2; we have never seen it run more slowly. The simple sort is usually faster than the system sort (except when one system sort exploited the special case of equal keys). Our tuned sort appears to be quite competitive with the fast radix sort. The primary challenge in implementing a radix sort is the case when the number of distinct keys is much less than the number of bins, either because the keys are all equal or because there are not many of them. Three-way radix quicksort may be thought of as a radix sort that gracefully adapts to handle this case, at the cost of slightly more work when the bins are all full. We encourage you to experiment with the algorithms yourself. The code that we used for this experiment is available online; see "Resource Center," page 3. If your results differ from ours for an important class of inputs, we would be interested in knowing about it.

Conclusion

Traditional library sorts are usually effective general-purpose tools, but we can do better for some particular applications. Three-way partitioning (Example 1) works well when the number of distinct key values is small, and three-way radix quicksort (Example 2) can be very effective for string keys.

Three-way radix quicksort is well suited for 16-bit Unicode applications (for example, those written in Java). The algorithm exploits the fact that only a small fraction of possible character values are typically in ASCII strings. This fraction is even smaller for Unicode strings.

Many sorts involve keys with multiple fields. A sort must arrange items in order according to the first key, with all equal values in order by their second keys, and so forth. Dates, for instance, are usually sorted first by year, then by month, then by day of the month. Our SODA paper generalizes Example 2 to a multikey quicksort; that algorithm might provide a useful basis for a general-purpose multikey sort program.

DDJ

Listing One

```
/* Support functions */



#ifndef min
#define min(a, b) ((a)<=(b) ? (a) : (b))
#endif



void swap(char *a[], int i, int j)
{     char *t = a[i];
      a[i] = a[j];
      a[j] = t;
}
void vecswap(char *a[], int i, int j, int n)
{     while (n-- > 0)
         swap(a, i++, j++);
}



/* Simple version */



#define ch(i) a[i][depth]



void ssort(char *a[], int n, int depth)
{     int le, lt, gt, ge, r, v;
      if (n <= 1)
         return;
      swap(a, 0, rand() % n);
      v = ch(0);
      le = lt = 1;
      gt = ge = n-1;
      for (;;) {
          for ( ; lt <= gt && ch(lt) <= v; lt++)
             if (ch(lt) == v)
               swap(a, le++, lt);
          for ( ; lt <= gt && ch(gt) >= v; gt--)
             if (ch(gt) == v)
               swap(a, gt, ge--);
          if (lt > gt)
             break;
          swap(a, lt++, gt--);
     }
     r = min(le, lt-le);
     vecswap(a, 0, lt-r, r);
     r = min(ge-gt, n-ge-1);
     vecswap(a, lt, n-r, r);
     ssort(a, lt-le, depth);
     if (v != 0)
```

```
        ssort(a + lt-le, le + n-ge-1, depth+1);
    ssort(a + n-(ge-gt), ge-gt, depth);
}



void ssortmain(char *a[], int n)
{    ssort(a, n, 0); }



/* Faster version */



int med3func(char *a[], int ia, int ib, int ic, int depth)
{   int va, vb, vc;
    if ((va=ch(ia)) == (vb=ch(ib)))
        return ia;
    if ((vc=ch(ic)) == va || vc == vb)
        return ic;
    return va < vb ?
          (vb < vc ? ib : (va < vc ? ic : ia ) )
        : (vb > vc ? ib : (va < vc ? ia : ic ) );
}
#define med3(ia, ib, ic) med3func(a, ia, ib, ic, depth)



void inssort(char *a[], int n, int depth)
{   int i, j;
    for (i = 1; i < n; i++)
      for (j = i; j > 0; j--) {
        if (strcmp(a[j-1]+depth, a[j]+depth) <= 0)
            break;
        swap(a, j, j-1);
      }
}



void ssort2(char *a[], int n, int depth)
{    int le, lt, gt, ge, r, v;
     int pl, pm, pn, d;
     if (n <= 10) {
        inssort(a, n, depth);
        return;
     }
     pl = 0;
     pm = n/2;
     pn = n-1;
     if (n > 50) {
        d = n/8;
        pl = med3(pl, pl+d, pl+2*d);
        pm = med3(pm-d, pm, pm+d);
        pn = med3(pn-2*d, pn-d, pn);
     }
     pm = med3(pl, pm, pn);
     swap(a, 0, pm);
     v = ch(0);
     for (le = 1; le < n && ch(le) == v; le++)
       ;
     if (le == n) {
         if (v != 0)
            ssort2(a, n, depth+1);
         return;
     }
     lt = le;
     gt = ge = n-1;
     for (;;) {
```

```
        for ( ; lt <= gt && ch(lt) <= v; lt++)
            if (ch(lt) == v)
                swap(a, le++, lt);
        for ( ; lt <= gt && ch(gt) >= v; gt--)
            if (ch(gt) == v)
                swap(a, gt, ge--);
        if (lt > gt)
            break;
        swap(a, lt++, gt--);
    }
        r = min(le, lt-le);
        vecswap(a, 0, lt-r, r);
        r = min(ge-gt, n-ge-1);
        vecswap(a, lt, n-r, r);
        ssort2(a, lt-le, depth);
        if (v != 0)
            ssort2(a + lt-le, le + n-ge-1, depth+1);
        ssort2(a + n-(ge-gt), ge-gt, depth);
}



void ssort2main(char *a[], int n)
{ ssort2(a, n, 0); }
```

Back to Article

DDJ

Sorting Strings with Three-Way Radix Quicksort

By Jon Bentley and Robert Sedgewick

Dr. Dobb's Journal November 1998

```
void iqs(int a[], int n)
{   int le, lt, gt, ge, r, v;
    if (n <= 1)
         return;
    swap(a, 0, rand() % n);
    v = a[0];
    le = lt = 1;
    gt = ge = n-1;
    for (;;) {
        for ( ; lt <= gt && a[lt] <= v; lt++)
            if (a[lt] == v)
                swap(a, le++, lt);
        for ( ; lt <= gt && a[gt] >= v; gt--)
            if (a[gt] == v)
                swap(a, gt, ge--);
        if (lt > gt)
                break;
        swap(a, lt++, gt--);
    }
    r = min(le, lt-le);
    vecswap(a, 0, lt-r, r);
    r = min(ge-gt, n-ge-1);
    vecswap(a, lt, n-r, r);
    iqs(a, lt-le);
    iqs(a + n-(ge-gt), ge-gt);
}
```

Example 1: Three-way quicksort for integer arrays.

Sorting Strings with Three-Way Radix Quicksort

By Jon Bentley and Robert Sedgewick

Dr. Dobb's Journal November 1998

```
void ssort(char *a[], int n, int depth)
{    int le, lt, gt, ge, r, v;
     if (n <= 1)
         return;
     swap(a, 0, rand() % n);
     v = ch(0);
     le = lt = 1;
     gt = ge = n-1;
     for (;;) {
         for ( ; lt <= gt && ch(lt) <= v; lt++)
             if (ch(lt) == v)
                swap(a, le++, lt);
         for ( ; lt <= gt && ch(gt) >= v; gt--)
             if (ch(gt) == v)
                swap(a, gt, ge--);
         if (lt > gt)
             break;
         swap(a, lt++, gt--);
     }
     r = min(le, lt-le);
     vecswap(a, 0, lt-r, r);
     r = min(ge-gt, n-ge-1);
     vecswap(a, lt, n-r, r);
     ssort(a, lt-le, depth);
     if (v != 0)
          ssort(a + lt-le, le + n-ge-1, depth+1);
     ssort(a + n-(ge-gt), ge-gt, depth);
}
```

Example 2: C program to sort strings.

Sorting Strings with

By Jon Bentley and Robert Sedgewick

Dr. Dobb's Journal November 1998

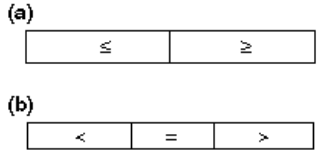

Figure 1: (a) Hoare's quicksort method uses two-way partitioning; (b) ours uses three-way partitioning.

Sorting Strings with Three-Way Radix Quicksort

By Jon Bentley and Robert Sedgewick

Dr. Dobb's Journal November 1998

```
qs(s,n)
  if n ≡ 1 return;
  choose a partitioning value v;
  partition s around value v to form
    sequences s<, s=, s> of sizes n<, n=, n>;
  qs(s<, n<);
  qs(s>,n>);
```

Figure 2: qs function in pseudocode.

Sorting Strings with Three-Way Radix Quicksort

By Jon Bentley and Robert Sedgewick

Dr. Dobb's Journal November 1998

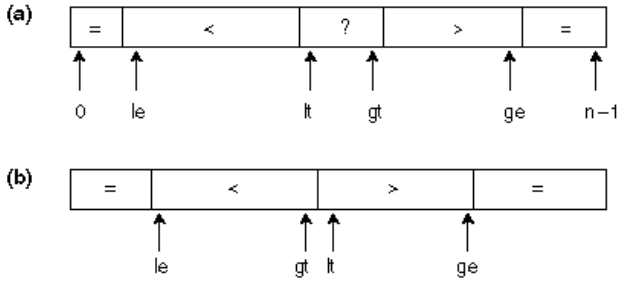

Figure 3: (a) Bentley-McIlroy three-way partitioning uses a counterintuitive loop invariant; (b) state of array after loop terminates.

Sorting Strings with Three-Way Radix Quicksort

By Jon Bentley and Robert Sedgewick

Dr. Dobb's Journal November 1998

```
ssort(s,n,depth)
  if n ≤ 1 return;
  choose a partitioning value v;
  partition s around value v on character depth to form
    sequences s<, s=, s> of sizes n<, n=, n>;
  ssort(s<,n<,depth);
  if the partitioning value v is not the null character
    ssort(s=,n=,depth+1);
  ssort(s>,n>,depth);
```

Figure 4: Recursive pseudocode that sorts the sequence s of length n.

Sorting Strings with Three-Way Radix Quicksort

By Jon Bentley and Robert Sedgewick

Dr. Dobb's Journal November 1998

	MIPS R10000		Pentium Pro	
	Dictionary	Equal	Dictionary	Equal
System	1.14	0.78	2.20	0.28
Simple	0.49	0.19	1.12	0.85
Tuned	0.29	0.16	0.77	0.55
Radix	0.31	0.57	0.47	1.39

Table 1: Number of seconds required by the various functions.
