Engineering Radix Sort

Peter M. McIlroy

Keith Bostic

Computer Science Research Group
University of California at Berkeley

M. Douglas McIlroy

AT&T Bell Laboratories

ABSTRACT

Radix sorting methods have excellent asymptotic performance on string data, for which comparison is not a unit-time operation. Attractive for use in large byte-addressable memories, these methods have nevertheless long been eclipsed by more easily programmed algorithms. Three ways to sort strings by bytes left to right—a stable list sort, a stable two-array sort, and an in-place ''American flag'' sort—are illustrated with practical C programs. For heavy-duty sorting, all three perform comparably, usually running at least twice as fast as a good quicksort. We recommend American flag sort for general use.

1. Introduction

For sorting strings you can't beat radix sort—or so the theory says. The idea is simple. Deal the strings into piles by their first letters. One pile gets all the empty strings. The next gets all the strings that begin with *A*-; another gets *B*- strings, and so on. Split these piles recursively on second and further letters until the strings end. When there are no more piles to split, pick up all the piles in order. The strings are sorted.

In theory radix sort is perfectly efficient. It looks at just enough letters in each string to distinguish it from all the rest. There is no way to inspect fewer letters and still be sure that the strings are properly sorted. But this theory doesn't tell the whole story: it's hard to keep track of the piles.

Our main concern is bookkeeping, which can make or break radix sorting as a practical method. The paper may be read as a thorough answer to excercises posed in Knuth chapters 5.2 and 5.2.5, where the general plan is laid out.[1] Knuth also describes the other classical sorting methods that we refer to: radix exchange, quicksort, insertion sort, Shell sort, and little-endian radix sort.

1.1. Radix exchange

For a binary alphabet, radix sorting specializes to the simple method of *radix exchange.*[2] Split the strings into three piles: the empty strings, those that begin with 0, and those that begin with 1. For classical radix exchange assume further that the strings are all the same length. Then there is no pile for empty strings and splitting can be done as in quicksort, with a bit test instead of quicksort's comparison to decide which pile a string belongs in.

Program 1.1 sorts the part of array A that runs from $A[lo]$ to $A[hi-1]$. All the strings in this range have the same b-bit prefix, say x-. The function `split` moves strings with prefix $x0$- to the beginning of the array, from $A[0]$ through $A[mid-1]$, and strings with prefix $x1$- to the end, from $A[mid]$ through $A[hi-1]$.

Program 1.1

```
RadixExchange(A, lo, hi, b) =
        if hi – lo ≤ 1
                then return
        if b ≥ length(A[lo])
                then return
        mid := Split(A, lo, hi, b)
        RadixExchange(A, lo, mid, b+1)
        RadixExchange(A, mid, hi, b+1)
```

To sort an *n*-element array, call

```
RadixExchange(A, 0, n, 0)
```

When strings can have different lengths, a full three-way split is needed, as in Program 1.2.[3] The pile of finished strings, with value x, say, begins at $A[lo]$; the $x0$- pile begins at $A[i0]$; the $x1$- pile begins at $A[i1]$.

Program 1.2.

```
RadixExchange(A, lo, hi, b) =
        if hi - lo ≤ 1
                then return
        (i0, i1) := Split3(A, lo, hi, b)
        RadixExchange(A, i0, i1, b+1)
        RadixExchange(A, i1, hi, b+1)
```

Three-way splitting is the famous problem of the Dutch national flag: separate three mixed colors into bands like the red, white and blue of the flag.[4] For us, the three colors are ∅ (no bit), 0 and 1. A recipe for splitting is given in Figure 1.1 and Program 1.3. The index $i0$ points to the beginning of the 0- pile, $i1$ points just beyond the end of the 0- pile, and $i2$ points to the beginning of the 1- pile. The notation $A[i].b$ denotes the bth bit, counted from 0, in string $A[i]$. When `Split3` finishes, $i1$ points to the beginning of the 1- pile as desired. The test for ∅ is figurative; it stands for a test for end of string.

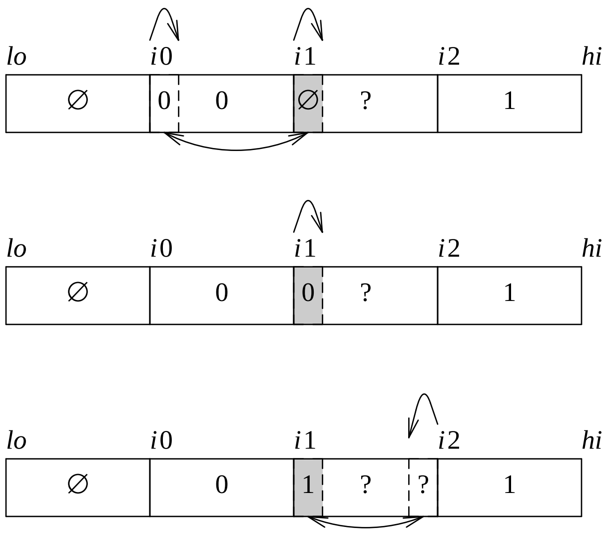

Figure 1.1. How `split3` works. The four parts of the array hold strings known to have ended (∅), strings known to have 0 in the selected position, unknown strings, and strings known to have 1 there. Repeatedly look at the selected position of the first unknown string—the shaded box. Update according to the matching diagram.

Program 1.3.

```
Split3(A, lo, hi, b) =
        (i0, i1, i2) := (lo, lo, hi)
        while i2 > i1 do
                case A[i1].b of
                ∅:      (A[i0], A[i1], i0, i1) := (A[i1], A[i0], i0+1, i1+1)
                0:      i1 := i1 + 1
                1:      (A[i1], A[i2-1], i2) := (A[i2-1], A[i1], i2-1)
        return (i0, i1)
```

1.2. Quicksort to the fore

After enjoying a brief popularity, radix exchange was upstaged by quicksort.[5] Not only was radix exchange usually slower than quicksort, it was not good for programming in Fortran or Algol, which hid the bits that it depends on. Quicksort became known as simple and nearly unbeatable; radix sorting disappeared from textbooks.

Nevertheless, radix exchange cannot be bettered in respect to amount of data looked at. Quicksort doesn't even come close. Quicksort on average performs $\Theta(\log n)$ comparisons per string and inspects $\Omega(\log n)$ bits per comparison. By this measure the expected running time for quicksort is $\Omega(n \log^2 n)$, for radix exchange only $\Omega(n \log n)$. Worse, quicksort can ''go quadratic,'' and take time $\Omega(n^2 \log n)$ on unfortunate inputs. This is not just an abstract possibility. Production quicksort routines have gone quadratic on perfectly reasonable inputs.[6]

The theoretical advantages of radix exchange are usually swamped by the cost of bit picking. Quicksort is nimbler in several ways.

- Quicksort can use machine instructions to compare whole words or bytes instead of bits.
- Quicksort splits adaptively. Because it picks splitting values from the data, quicksort can be expected to get roughly 50-50 splits even on skewed data. Such splits are necessary to realize minimal expected times in either quicksort or radix exchange.
- Quicksort can sort anything, not just strings. Change the comparison routine and it is ready to handle different data. Because radix sort intrinsically assumes string data on a finite alphabet, it requires one to make the data fit the routine, not vice versa. For example, to sort dates with quicksort, one might provide code to parse ordinary notation (e.g. February 11, 1732) as each key is looked at, while with radix sort one would preconvert all dates to a canonical form (e.g. 17320211).

In other ways, quicksort and radix exchange are quite alike. They both sort in place, using little extra space. Both need a recursion stack, which we expect to grow to size $\Theta(\log n)$. In either method, if the strings are long or have different lengths, it is well to address strings through uniform descriptors and to sort by rearranging small descriptors instead of big strings.

The wisdom that blesses quicksort dates from the era of small memories. With bigger machines, the difference between $n \log n$ and $n \log^2 n$ becomes more significant. And with bigger machines we can afford more space. Thus the wisdom deserves to be reexamined.

2. List-based sort

For most modern machines, the 8-bit byte is a natural radix, which should overcome the bit-picking slowness of radix exchange. A byte radix makes for 256- or 257-way splitting, depending on how the ends of strings are determined. This raises the problem of managing space for so many piles of unknown size at each level of recursion. An array of linked lists is an obvious data structure. Dealing to the piles is easy; just index into the array. Picking up the sorted piles and getting them hooked together into a single list is a bit tricky, but takes little code. Program 2.1 does the job. It is written in C rather than pseudocode, because the troubles with radix sort are in implementation, not in conception. The input variables are

a	a linked list of null-terminated strings.
b	the offset of the byte to split on; the strings agree in all earlier bytes.
sequel	a sorted linked list of strings that compare greater than the strings in list *a*.

Three in-line functions are coded as macros:

`ended(`*a*`,` *b*`)` tells whether byte position *b* is just beyond the null byte at the end of string *a*.
`append(`*s*`,` *a*`)` appends list *s* to the last element of non-empty list *a*.
`deal(`*a*`,` *p*`)` removes the first string from list *a* and deals it to pile *p*.

Program 2.1 has four parts. First, if the list is empty, then the result of sorting it together with *sequel* is *sequel*. Next, at ‘‘pile finished,’’ if the last byte seen in the strings in list *a* was null (0), they cannot be sorted further. Put them in front of the *sequel* and return the combined list. At ‘‘split,’’ all strings have a *b*th byte. Clear all the piles and then deal the strings out according to byte *b* of each string. Finally, at ‘‘recur on each pile,’’ sort the piles from last to first. At each stage append to the sorted current pile the sorted list accumulated from all following piles.

Program 2.1. Simple list-based sort.

```
typedef struct list {
        struct list     *next;
        unsigned char   *data;
} list;

list *rsort(list *a, int b, list *sequel)
{
#define ended(a, b)     b>0 && a->data[b-1]==0
#define append(s, a)    tmp=a; while(tmp->next) tmp=tmp->next; tmp->next=s
#define deal(a, p)      tmp = a->next, a->next = p, p = a, a = tmp

        list            *pile[256], *tmp;
        int             i;

        if(a == 0)
                return sequel;

        if(ended(a, b)) {                       /* pile finished */
                append(sequel, a);
                return a;
        }

        for(i = 256; --i >= 0; )                /* split */
                pile[i] = 0;
        while(a)
                deal(a, pile[a->data[b]]);

        for(i = 256; --i >= 0; )                /* recur on each pile */
                sequel = rsort(pile[i], b+1, sequel);

        return sequel;
}
```

Program 2.1 works—slowly. Empty piles are the root of the trouble. Except possibly at the first level or two of recursion, most piles will be empty. The cost of clearing and recursively ‘‘sorting’’ as many as 255 empty piles for each byte of data is overwhelming. Some easy improvements will speed things up.

I2.1 Always deal into the same array and clear only the occupied piles between deals, meanwhile stacking the occupied piles out of the way.

I2.2 Manage the stack directly. Since the number of occupied piles is unpredictable, and probably small except at the first level or two of recursion, much space can be saved. The piles may be stacked in first-to-last order so they will pop off in last-to-first order just as in Program 2.1.

I2.3 Don’t try to split singleton piles.

I2.4 Optimize judiciously: eliminate redundant computation; replace subscripts by pointers.

I2.5 Avoid looking at empty piles.

Program 2.2 implements improvements I2.1-I2.3. Here the state-carrying parameters *b* and *sequel* become hidden, as they should be.

Program 2.2. Improved list-based sort. See Program 2.1 for some macros.

```
list* rsort(list* a)
{
#define singleton(a)    a->next == 0
#define push(a, b)      sp->sa = a, (sp++)->sb = b
#define pop(a, b)       a = (--sp)->sa, b = sp->sb
#define stackempty()    (sp <= stack)
        struct { list *sa; int sb; } stack[SIZE], *sp = stack;
        static list     *pile[256];
        list            *tmp, *sequel = 0;
        int             i, b;
        if(a)
                push(a, 0);
        while(!stackempty()) {
                pop(a, b);
                if(singleton(a) || ended(a, b)) { /* pile finished */
                        append(sequel, a);
                        sequel = a;
                        continue;
                }
                while(a)                          /* split */
                        deal(a, pile[a->data[b]]);
                for(i = 0; i<256; i++)            /* stack the pieces */
                        if(pile[i]) {
                                push(pile[i], b+1);
                                pile[i] = 0;
                        }
        }
        return sequel;
}
```

On typical alphabetic data Program 2.2 runs about 15 times as fast as Program 2.1—not a bad return from such simple optimizations, but not yet good enough. Most of the time for Program 2.2 is still wasted scanning empty piles. Thus we turn to improvement I2.5, avoiding looking at empty piles, which can be done in many ways.

For textual keys, such as names or decimal numbers, the piles are likely to be bunched. Single-case letters span only 26 of the 256 piles, digits only 10. To exploit bunching, we use a simple *pile-span* heuristic: keep track of the range of occupied piles. The finished 0- pile is an expected outlier and is kept track of separately.

Taken together, improvements I2.1-I2.5 speed up Program 2.1 by a factor of 100 on typical inputs. The result, Program A in the appendix, is a creditable routine. It usually sorts arrays of 10,000 to 100,000 keys twice as fast as do competitive quicksorts.

None of our programs so far sorts stably. Because piles are built by pushing records on the front of lists, the order of equal-keyed records is reversed at each deal. To stabilize the sort we can reverse each backwards pile as we append to it. Alternatively we can maintain the piles in forward order by keeping track of head and tail of each pile. Program A does the latter. Sorting times differ negligibly among forward/reverse and stable/unstable versions.

3. Two-array sort

Suppose the strings come in an array as for radix exchange. In basic radix exchange, the two piles live in known positions against the bottom and top of the array. For larger radixes, the positions of the piles can be calculated in an extra pass that tallies how many strings belong in each pile. Knowing the sizes of the piles, we don't need linked lists.

Program 3.1 gets the strings home by moving them as a block to the auxiliary array *ta*, and then moving each element back to its proper place. The upper ends of the places are precomputed in array *pile* as shown in Figure 4.1. (This ‘‘backward’’ choice is for harmony with the programs in section 4.) Elements are moved stably; equal elements retain the order they had in the input. As in Program 2.2, the stack is managed explicitly; the stack has a third field to hold the length of each subarray.

Program 3.1. Simple two-array sort.

```
typedef unsigned char *string;

void rsort(string *a, int n)
{
#define push(a, n, b)   sp->sa = a, sp->sn = n, (sp++)->sb = b
#define pop(a, n, b)    a = (--sp)->sa, n = sp->sn, b = sp->sb
#define stackempty()    (sp <= stack)
#define splittable(c)   c > 0 && count[c] > 1

        struct { string *sa; int sn, sb; } stack[SIZE], *sp = stack;
        string          *pile[256], *ak, *ta;
        static int      count[256];
        int             i, b;

        ta = malloc(n*sizeof(string));

        push(a, n, 0);
        while (!stackempty()) {
                pop(a, n, b);
                for(i = n; --i >= 0; )                  /* tally */
                        count[a[i][b]]++;
                for(ak = a, i = 0; i < 256; i++) {      /* find places */
                        if(splittable(i))
                                push(ak, count[i], b+1);
                        pile[i] = ak += count[i];
                        count[i] = 0;
                }
                for(i = n; --i >= 0; )                  /* move to temp */
                        ta[i] = a[i];
                for(i = n; --i >= 0; )                  /* move home */
                        *--pile[ta[i][b]] = ta[i];
        }
        free(ta);
}
```

Program 3.1 is amenable to most of the improvements listed in section 2; they appear in Program B. In addition, the piles are independent and need not be handled in order. Nor is it necessary to record the places of empty piles. These observations are embodied in the ‘‘find places’’ step of Program B.

As we observed in the introduction, radix sorting is most advantageous for large arrays. When the piles get small, we may profitably divert to a simple low-overhead comparison-based sorting method such as insertion sort or Shell sort. Diversion thresholds between 10 and 50 work well; the exact value is not critical. Program B in the appendix is such a hybrid two-array sort. It is competitive with list-based sort; which of the two methods wins depends on what computer, compiler, and test data one measures.

For library purposes, an array interface is more natural than a list interface. But two-array sort dilutes that advantage by using $\Theta(n)$ working space and dynamic storage allocation. Our next variant overcomes this drawback.

4. American flag sort

Instead of copying data to an auxiliary array and back, we can permute the data in place. The central problem, a nice exercise in practical algorithmics, is to rearrange into ascending order an array of n integer values in the range 0 to $m-1$. Here m is a value of moderate size, fixed in our case at 256, and n is arbitrary. Special cases are the partition step of quicksort ($m = 2$) and the Dutch national flag problem

($m = 3$). By analogy with the latter, we call the general problem the *American flag problem.* (The many stripes are understood to be labeled distinctly, as if with the names of the several states in the original American union.)

American flag sort differs from the two-array sort mainly in its final phase. The effect of the ‘‘move to temp’’ and ‘‘move home’’ phases of Program 3.1 is attained by the ‘‘permute home’’ phase shown in Program 4.1 and Figures 4.1-4.4. This phase fills piles from the top, making room by cyclically displacing elements from pile to pile.*

Let $a[k]$ be the first element of the first pile not yet known to be completely in place. Displace this element out of line to *r* (Figure 4.2). Let *c* be the number of the pile the displaced element belongs to. Find in the *c*- pile the next unfilled location, just below $pile[c]$ (Figure 4.3). This location is the home of the displaced element. Swap the displaced element home after updating $pile[c]$ to account for it.

Repeat the operation of Figure 4.3 on the newly displaced element, following a cycle of the permutation until finally the home of the displaced element is where the cycle started, at $a[k]$. Move the displaced element to $a[k]$. Its pile, the current *c*- pile, is now filled (Figure 4.4). Skip to the beginning of the next pile by incrementing *k*. (Values in the *count* array must be retained from the ‘‘find places’’ phase.) Clear the count of the just-completed pile, and begin another permutation cycle. It is easy to check that the code works right when $a + k = pile[c]$ initially, that is, when the pile is already in place.

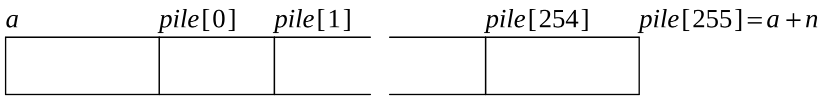

Figure 4.1. Array *a* before permuting home.

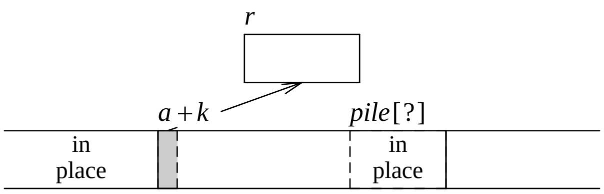

Figure 4.2. After first displacement. Arrow shows completed action.

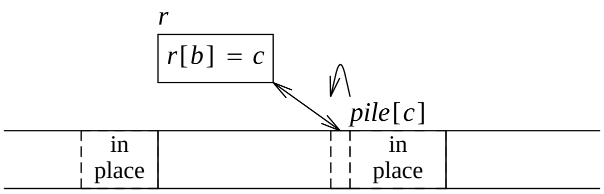

Figure 4.3. During displacement cycle. The *b*th byte of the string pointed to by *r* is *c*. Arrows show actions to do, except no swap happens in last iteration.

* A similar algorithm in Knuth chapter 5.2, exercise 13, does without the array *count*, but involves more case analysis and visits $O(n)$ elements more than once. The speed and simplicity of Program 4.1 justify the cost of the extra array.

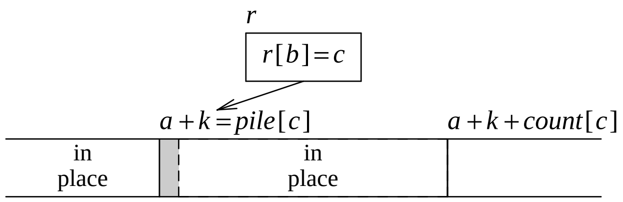

Figure 4.4. Last move.

Program 4.1. In-place permutation to substitute in Program 3.1.

```
#define swap(p, q, r) r = p, p = q, q = r
        string          r, t;
        int             k, c;

        for(k = 0; k < n; ) {
                r = a[k];                       /* Figure 4.2 */
                for(;;) {
                        c = r[b];               /* Figure 4.3 */
                        if(--pile[c] <= a+k)
                                break;
                        swap(*pile[c], r, t);
                }
                a[k] = r;                       /* Figure 4.4 */
                k += count[c];
                count[c] = 0;
        }
```

When all piles but one are in place, the last pile must necessarily be in place, too. Program 4.2, otherwise a condensed Program 4.1, exploits this fact. Program 4.2 and Program B form the the basis of Program C in the appendix.

Program 4.2. Improved in-place permutation.

```
        cmax = /* index of last occupied pile */;
        n -= count[cmax];
        count[cmax] = 0;
        for(ak = a; ak < a+n; ak += count[c], count[c] = 0) {
                r = *ak;
                while(--pile[c = r[b]] > ak)
                        swap(*pile[c], r, t);
                *ak = r;
        }
```

4.1. Stack growth

In the programs so far, the stack potentially grows linearly with running time. We can bound the growth logarithmically by arranging to split the largest pile at each level last—a trick well known from quicksort. This biggest-pile-last strategy is easy to install in array-based sorts, but inconvenient for list-based, where stack order matters and pile sizes are not automatically available.

Even with a logarithmic bound, the stack can be sizable. In the worst case, a split produces no 0-piles, 253 little (size 2) piles, and two big piles of equal size for the rest of the data. One of the big piles is immediately popped from the stack and split similarly. For $n = 1,000,000$, the worst-case stack has 2800 entries (33,000 bytes on a 32-bit machine). By contrast, a stack of about $254 \log_{256} n$ entries (only 630 when $n = 1,000,000$) suffices for uniform data. Still smaller stacks work for realistic data. Instead of a worst-case stack, we may allocate a short stack, say enough for two levels of full 256-way splitting, and call the routine recursively in the rare case of overflow.

For completeness, both stacking tactics are shown in program C, though they will almost surely never be needed in practice. Stack control adds one-third to the executable code, but only about one percent to the running time.

4.2. Tricks for tallying

The pile-span heuristic for coping with empty piles presupposes a not unfavorably distributed alphabet. Other ways to avoid looking at empty piles may be found in the literature. One is to keep a list of occupied piles. Each time a string goes into an empty pile, record that pile in the list. After the deal, sort the list of occupied piles by pile number. If the list is too long, ignore it and scan all the piles.[7] A version of Program C with this strategy instead of pile-span ran slightly faster on adverse data, but slower on reasonable data.

Alternatively, an occupancy tree may be superimposed on the array of piles. Then the amount of work to locate occupied piles will diminish with diminishing occupancy. The best of several tree-tallying schemes that we have tried is quite insensitive to the distribution of strings and beats pile-span decisively on adverse data, but normally runs about one-third to one-half slower then pile-span.

Noting that only the identity and not the order of the piles matters in splitting, Paige and Tarjan propose to scan piles in one combined pass after all splits are done.[8] Their method favors large radixes; it runs faster with radix 64K than with radix 256. Unfortunately, overhead—from $4n$ to $8n$ extra words of memory—swamps the theoretical advantage.

Little-endian (last-letter first) sorting mitigates the problem of scanning empty piles. In little-endian sorts the number of splits is equal to the number of letters in the longest key, whereas in big-endian sorts like ours the number of splits typically exceeds the number of keys. Aho, Hopcroft, and Ullman show how to eliminate pile scanning at each deal of a little-endian sort by using a $\Theta(\text{total size})$ presort of all letters from all keys to predict what piles will occur.[9] A little-endian radix sort, however, must visit all letters of all keys instead of just the letters of distinguishing prefixes.

In practice, exotic tricks for tallying are rendered moot by diverting to an alternate algorithm for small n. For it is only when n is small that the time to scan piles is noticeable in comparison to the time to deal and permute. Nevertheless, we still like the extremely cheap pile-span heuristic as a supplemental strategy, for it can improve running times as much as 10% beyond diversion alone.

5. Performance

The merits of the several programs must be judged on preponderant, not decisive, evidence. In theory, all have the same worst-case asymptotic running time, $O(S)$, where S is the size of the data measured in bytes. None is clearly dominant in practical terms, either. The relative behaviors vary with data, hardware, and compiler.

In assessing performance, we shall consider only large data sets, where radix sorting is most attractive. Just how attractive is indicated by comparison with quicksort. The tested quicksort program, which compares strings in line, chooses a random splitting element, and diverts to a simple sort for small arrays, was specialized from a model by Bentley and McIlroy. The routine is not best possible, but probably within 1/3 of the ultimate speed for C code. (We recoiled from adapting their fastest model, which would require 23 in-line string comparisons.)

Figure 5.1 shows the variation with size for 15 tests of each of four routines on one computer for two kinds of random key: (1) strings of 8 random decimal digits and (2) strings of random bytes, exponentially distributed in length with mean 9. The range of this experiment is too narrow to reveal quicksort's $n \log^2 n$ departure from linearity, or to fully smooth quantizing effects. (Across this range the expected length of comparisons varies by only one digit for the decimal data and half a byte for the byte data.) Nevertheless the figure clearly shows that the radix sorts run markedly faster than quicksort. This observation is robust. No comparable generalization can be drawn about the relative performance of the three radix sorts. As the figure shows, the rank order depends on the kind of data. Other experiments show similar variation with hardware.

The sensitivity of the pile-span heuristic to key distribution shows up in Figure 5.1 as a large variation of slope for list-based sort. In the other two routines, diversion almost completely damps the variation. The sensitivity is even greater on sparse random data. In the extreme case of random keys strings containing just two byte values, the list-based Program A took about 5 times as long to sort keys with distant byte values as with adjacent values. The hybrid Programs B and C varied by a factor of 1.2 or less. Quicksort was unaffected.

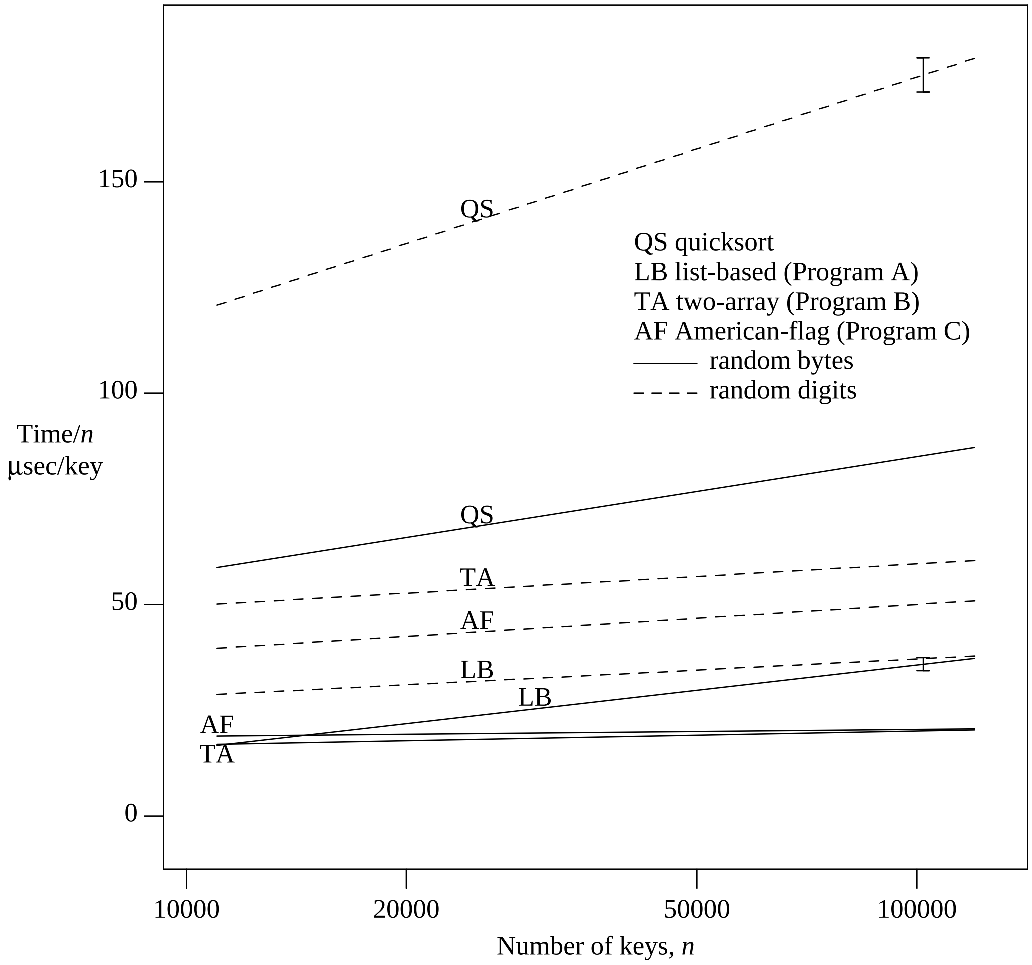

Figure 5.1. Least-squares fits to sorting time per key versus $\log n$ for a DEC VAX 8550. Representative $\pm 1\sigma$ error bars are shown; other curves fit comparably.

Realistic sorting problems are usually far from random. We sorted the 73,000 words of the Merriam-Webster Collegiate Dictionary, 7th edition, using the machines and compilers listed in Table 5.1. The word list consists mainly of two interleaved alphabetical lists, of capitalized and uncapitalized words. We sorted, into strict ASCII order, three input configurations: (1) as is, (2) two copies of the list concatenated, and (3) ordered by reversed spelling, which mixes the data well. The running times of programs A, B, and C were usually within a factor of 1.2 of each other, with no clear winner. American-flag sort won consistently on the MIPS, two-array sort on the Cray, and list-based on the Vax, a result roughly consonant with the degree of pipelining on the several machines. The list-based program was the most erratic. It lost consistently on the MIPS, and decisively—by a factor of 1.6—on some Cray and MIPS runs. As in Figure 5.1, quicksort fell far behind the radix sorts, usually by a factor of two or more.

6. Discussion

Our programs synopsize experiments that we have made jointly and severally over the past few years. Bostic wrote a two-array radix sort similar to Program B for the Berkeley BSD library, based in part on a routine by Dan Bernstein. P. McIlroy adapted that routine for use in the BSD version of the Posix standard[11] sort utility.[12] P. McIlroy also conceived American flag sort as a replacement for the two-array

Table 5.1. Machines and compilers tested.

Machine	Compiler	Author
DEC Vax 8850		
	lcc	Fraser and Hanson[10]
	gcc	Gnu
	gcc –O	same, with optimization
MIPS 6280		
	cc	MIPS
	cc –O4	same, with optimization
	lcc	Fraser and Hanson
Sun Sparcstation		
	lcc	Fraser and Hanson
	gcc –O	Gnu
Cray XMP		
	scc	Cray

library routine. Independently, D. McIlroy wrote a Posix utility around a list-based radix sort, and installed it on research systems at AT&T. Both the Berkeley and the AT&T utilities typically run twice as fast overall as the venerable quicksort-based programs[13] that they replace.

Although radix sorts have unbeatable asymptotic performance, they present problems for practical implementation: (1) managing scattered piles of unpredictable size and (2) handling complex keys. We have shown that the piles can be handled comfortably. Our utilities cope with complex keys by preconverting them into strings. Although it costs memory roughly proportional to the volume of keys, this strategy is simple and effective for sorting records after the fashion of the proposed Posix standard.

List-based radix sort is faster than pure array-based radix sorts. The speed disparity is overcome by hybrid routines that divert from radix sorting to simple comparison-based sorting for small arrays. The natural array-argument interface makes them attractive for library purposes. Both list-based and two-array sorts entail $\Theta(n)$ space overhead. That overhead shrinks to $\Theta(\log n)$ in American flag sort, which, like quicksort, trades off stability for space efficiency.

We recommend American flag sort as an all-round algorithm for sorting strings.

We are grateful to Peter McCauley for critical examination of our programs. And we have profited, even to the wording of our title, from the advice and exemplary style of Jon Bentley.

References

[1] Knuth, D. E., *The Art of Computer Programming*, 3, Sorting and Searching, Addison Wesley (1973).

[2] Hildebrandt, P. and Isbitz, H., ‘‘Radix exchange—an internal sorting method for digital computers,’’ *JACM* **6**, pp. 156-163 (1959).

[3] Bentley, J. L., *Programming Pearls*, Addison-Wesley (1986). Solution 10.6.

[4] Dijkstra, E. W., *A Discipline of Programming*, Prentice-Hall (1976).

[5] Hoare, C. A. R., ‘‘Quicksort,’’ *Computer Journal* **5**, pp. 10-15 (1962).

[6] Bentley, J. L., ‘‘The trouble with qsort,’’ *Unix Review* **10**(2), pp. 85-93 (Feb. 1992).

[7] McCauley, P. B., *Sorting method and apparatus*, U. S. Patent 4,809,158 (1989).

[8] Paige, R. and Tarjan, R. E., ‘‘Three partition refinement algorithms,’’ *SIAM J. Comput.* **16**, pp. 973-989 (1987).

[9] Aho, A. V., Hopcroft, J. E., and Ullman, J. D., *The Design and Analysis of Computer Algorithms*, Addison-Wesley (1974).

[10] Fraser, C. W. and Hanson, D. R., ‘‘A retargetable compiler for ANSI C,’’ *ACM SIGPLAN Notices*

26(10), pp. 29-43 (October 1991).

[11] IEEE, *Draft Standard for Information Technology – Operating System Interface (POSIX) Part 2: Shell and Utilities*, Vol. P1003.2/D11 (1991).

[12] McIlroy, P. M., ‘‘Intermediate files and external radix sort,’’ (). submitted for publication.

[13] Linderman, J. P., ‘‘Theory and practice in the construction of a working sort routine,’’ *AT&T Bell Laboratories Technical Journal* **63**, pp. 1827-1844 (1984).

Appendix

Program A. Stable list-based sort.

```
typedef struct list {
        struct list     *next;
        unsigned char   *data;
} list;

list    *rsort(list *a)
{
#define push(a, t, b)   sp->sa = a, sp->st = t, (sp++)->sb = b
#define pop(a, t, b)    a = (--sp)->sa, t = sp->st, b = sp->sb
#define stackempty()    (sp <= stack)
#define singleton(a)    (a->next == 0)
#define ended(a, b)     b>0 && a->data[b-1]==0

        struct { list *sa, *st; int sb; } stack[SIZE], *sp = stack;
        static list     *pile[256], *tail[256];
        list            *atail, *sequel = 0;
        int             b, c, cmin, nc = 0;

        if(a)
                push(a, a, 0);

        while(!stackempty()) {
                pop(a, atail, b);
                if(singleton(a) || ended(a, b)) { /* pile finished */
                        atail->next = sequel;
                        sequel = a;
                        continue;
                }

                cmin = 255;                       /* split */
                for( ; a; a = a->next) {
                        c = a->data[b];
                        if(pile[c] == 0) {
                                tail[c] = pile[c] = a;
                                if(c == 0) continue;
                                if(c < cmin) cmin = c;
                                nc++;
                        } else
                                tail[c] = tail[c]->next = a;
                }

                if(pile[0]) {                     /* stack the pieces */
                        push(pile[0], tail[0], b+1);
                        tail[0]->next = pile[0] = 0;
                }
                for(c = cmin; nc > 0; c++)
                        if(pile[c]) {
                                push(pile[c], tail[c], b+1);
                                tail[c]->next = pile[c] = 0;
                                nc--;
                        }
        }

        return sequel;
}
```

Program B. Hybrid two-array sort.

```
typedef unsigned char *string;

void simplesort(string*, int, int);

void rsort(string *a, int n)
{
#define push(a, n, i)   sp–>sa = a, sp–>sn = n, (sp++)–>si = i
#define pop(a, n, i)    a = (––sp)–>sa, n = sp–>sn, i = sp–>si
#define stackempty()    (sp <= stack)

        struct { string *sa; int sn, si; } stack[SIZE], *sp = stack;
        string          *pile[256], *ai, *ak, *ta;
        static int      count[256];
        int             b, c, cmin, *cp, nc = 0;

        ta = malloc(n*sizeof(string));

        push(a, n, 0);
        while(!stackempty()) {
                pop(a, n, b);
                if(n < THRESHOLD) {                 /* divert */
                        simplesort(a, n, b);
                        continue;
                }

                cmin = 255;                         /* tally */
                for(ak = a+n; ––ak >= a; ) {
                        c = (*ak)[b];
                        if(++count[c] == 1 && c > 0) {
                                if(c < cmin) cmin = c;
                                nc++;
                        }
                }

                pile[0] = ak = a + count[0];        /* find places */
                count[0] = 0;
                for(cp = count+cmin; nc > 0; cp++, nc––) {
                        while(*cp == 0) cp++;
                        if(*cp > 1)
                                push(ak, *cp, b+1);
                        pile[cp–count] = ak += *cp;
                        *cp = 0;
                }

                for(ak = ta+n, ai = a+n; ak > ta; ) /* move to temp */
                        *––ak = *––ai;

                for(ak = ta+n; ak–– > ta; )         /* move home */
                        *––pile[(*ak)[b]] = *ak;
        }
        free(ta);
}
```

Program C. Hybrid American flag sort; optional stack control in fine print.

```
enum { SIZE = 510, THRESHOLD = 16 };
typedef unsigned char *string;
typedef struct { string *sa; int sn, si; } stack_t;
void simplesort(string*, int, int);
static void rsorta(string *a, int n ,int b)
{
#define push(a, n, i)   sp->sa = a, sp->sn = n, (sp++)->si = i
#define pop(a, n, i)    a = (--sp)->sa, n = sp->sn, i = sp->si
#define stackempty()    (sp <= stack)
#define swap(p, q, r)   r = p, p = q, q = r
        stack_t         stack[SIZE], *sp = stack, stmp, *oldsp, *bigsp;
        string          *pile[256], *ak, *an, r, t;
        static int      count[256], cmin, nc;
        int             *cp, c, cmax/*, b = 0*/;
        push(a, n, b);
        while(!stackempty()) {
                pop(a, n, b);
                if(n < THRESHOLD) {             /* divert */
                        simplesort(a, n, b);
                        continue;
                }
                an = a + n;
                if(nc == 0) {                   /* untallied? */
                        cmin = 255;             /* tally */
                        for(ak = a; ak < an; ) {
                                c = (*ak++)[b];
                                if(++count[c] == 1 && c > 0) {
                                        if(c < cmin) cmin = c;
                                        nc++;
                                }
                        }
                        if(sp+nc > stack+SIZE) {        /* stack overflow */
                                rsorta(a, n, b);
                                continue;
                        }
                }
                oldsp = bigsp = sp, c = 2;              /* logarithmic stack */
                pile[0] = ak = a+count[cmax=0]; /* find places */
                for(cp = count+cmin; nc > 0; cp++, nc--) {
                        while(*cp == 0) cp++;
                        if(*cp > 1) {
                                if(*cp > c) c = *cp, bigsp = sp;
                                push(ak, *cp, b+1);
                        }
                        pile[cmax = cp-count] = ak += *cp;
                }
                swap(*oldsp, *bigsp, stmp);
                an -= count[cmax];              /* permute home */
                count[cmax] = 0;
                for(ak = a; ak < an; ak += count[c], count[c] = 0) {
                        r = *ak;
                        while(--pile[c = r[b]] > ak)
                                swap(*pile[c], r, t);
                        *ak = r;
                }               /* here nc = count[...] = 0 */
        }
}
void rsort(string *a, int n) { rsorta(a, n, 0); }
```