A Fast Radix Sort

I. J. DAVIS

Department of Computing and Physics, Wilfrid Laurier University, Waterloo, Ontario, Canada N2L 3C5

Almost all computers regularly sort data. Many different sort algorithms have therefore been proposed, and the properties of these algorithms studied in great detail. It is known that no sort algorithm based on key comparisons can sort N keys in less than $O(N \log N)$ operations, and that many perform $O(N^2)$ operations in the worst case. The radix sort has the attractive feature that it can sort N keys in $O(N)$ operations, and it is therefore natural to consider methods of implementing such a sort efficiently.

In this paper one efficient implementation of a radix sort is presented, and the performance of this algorithm compared with that of Quicksort. Empirical results are presented which suggest that this implementation of a radix sort is significantly faster than Quicksort, and that it therefore has wide applicability.

Received July 1989, revised August 1992

1. INTRODUCTION

In 1984 Wilfrid Laurier University developed an in-house database system, called MSQ, which was written in PL6, and which ran under Honeywell's CP6 operating system.[5,6] This product was recently translated into 'C' so that it could run in a UNIX† environment. Since the performance of any database system is critical, extensive performance studies were then undertaken. Investigating the speed with which MSQ sorted data was an important component of this study, since users sort large volumes of data frequently: when creating reports, when constructing auxiliary indices, and when verifying the correctness of data and/auxiliary indices. In addition, MSQ internally sorts data when performing garbage collection.

Historically MSQ used Quicksort[10] to sort data. However, this algorithm was not particularly efficient when sorting large numbers of keys, and alternative sort algorithms were therefore investigated, in the hope that performance could be improved. This paper presents an alternative algorithm for sorting fixed-length keys of arbitrary size, which typically performs very much better than Quicksort. Empirical results justifying this claim are also presented.

Numerous algorithms have been proposed for sorting data efficiently, and the properties and operational complexity of these algorithms are now well known.[11] One algorithm which has been widely implemented is Quicksort, and many papers have studied or suggested enhancements to this algorithm.[3,4,14,15] Quicksort has developed a reputation for being one of the fastest sorting algorithms available, even though it is known that its worst-case performance is very poor.[9] When sorting N keys, Quicksort has an expected operational complexity of $O(N \log N)$ and a worst case performance of $O(N^2)$ (but see Ref. 13).

Unfortunately, any algorithm which has an operational complexity of at least $O(N \log N)$ remains slow, when presented with large numbers of keys. Recently, there has therefore been renewed interest in historical methods of sorting data which have $O(N)$ operational complexity.[2,7,8,16] The radix sort represents one such algorithm.

Unlike sorts based on direct key comparisons, a radix sort orders keys by iteratively partitioning keys based on successive bytes within keys. At each iteration the partitions are then potentially reordered. Such sorts were well understood long before the advent of computers, and are still used in some manual card systems. Typically, in a manual card system cards are partitioned and reordered by using increasingly significant bytes within their keys.

A radix sort may also be implemented by recursively examining bytes having decreasing significance within the sort keys. This implementation of a radix sort is superficially similar to that used when implementing Quicksort. The keys to be sorted are partitioned into p ordered partitions, using the most significant byte of each key. Each partition is then recursively sorted using successive bytes within keys. This implementation of a radix sort is the one used in this paper, since it does not require that partitioning is stable, and has the advantage that it only examines significant bytes within each sort key. However, it may be less efficient than sorting on increasingly significant bytes within keys, since it typically performs many more partitioning operations.

2. RADIX SORT IMPLEMENTATION

The effectiveness of any radix sort is critically dependent on the method used to partition keys. The radix sort implemented, presented in Fig. 1, partitions key pointers (addressing fixed-length keys) using three efficient operations. The size of each partition is first determined by using a static table of 256 integer counters. These counters are initially assigned zero values, and are subsequently zero on every entry to the recursive radix sort. Having determined the size of each partition, this information is used to construct a linked list whose entries address consecutive non-null partitions. For efficiency, since this linked list contains at most 256 entries, it is implemented using two arrays, one containing 256 byte offsets which link the list and one containing 256 pointers which address partitions. Using this linked list, key pointers are partitioned in place by performing a second scan of the relevant byte within each key to be sorted.

* *Address for correspondence:* Dr I. J. Davis, 41 High Street, Waterloo, Ontario, Canada N2L 3X7.

† UNIX is a trademark of AT&T; UNIX is a trademark of Bell Laboratories.

Downloaded from http://comjnl.oxfordjournals.org/ by guest on December 18, 2014

```
radix_sort(sortPP,endPP,offset)          /* All characters are unsigned!!    */
char    **sortPP;                        /* Array of pointers to keys        */
char    **endPP;                         /* Address beyond end of array      */
int     offset;                          /* Byte offset in key to sort on    */
{
    extern  int         keysize;         /* Size of each key examined        */
    extern  unsigned    slot_size[256]; /* Zero on each entry to routine    */

    static  int         slot_head;       /* Head of linked list              */
    char                slot_list[256]; /* Offset of next node in list      */
    char                **slotPP [256]; /* Pointer to each partition        */

    register    char    **PP;            /* Working register                 */
    register    int     head;            /* Working register                 */
    register    int     off;             /* Current offset in key            */

    off = offset;
    while (1) {
        {   register    char    **sPP;   /* Register copy of sortPP          */
            register    int     keys;    /* Number of keys to sort           */

            PP   = endPP;
            sPP  = sortPP;               /* Place parameters in registers    */
            keys = PP - sPP;             /* Number of keys to sort           */

            if (keys < 16) return;       /* Don't sort < 16 keys             */

            do {                         /* Count byte value distribution -- */
                head = (*--PP)[off];     /* Value of significant byte        */
                slot_size[head]++;       /* Count frequence of value         */
            } while (PP > sPP);          /* -- for all keys                  */

            if (slot_size[head] == keys) {
                slot_size[head] = 0;     /* All keys have same byte value    */
                if (++off >= keysize) return;
                continue;                /* Sort on next byte in each key    */
        }   }
        {   register    int     size;    /* Size of a partition              */
            register    int     i;       /* Counter being examined           */

            head = 255;                  /* End of list marker               */
            i    = -1;
            do {                         /* For each occurring byte value -- */
                while (!(size = slot_size[++i]));
                slot_list[i] = head;     /* Add partition for this byte to   */
                head         = i;        /* linked list                      */
                slotPP[i]    = PP;       /* Start of this partition          */
                PP          += size;     /* Start of next partition          */
            } while (PP < endPP);        /* -- until all partitions defined  */
            slot_head = head;            /* Save header of linked list       */
        }
        {   register    int     c;
            register    char    *keyP, *key1P;

            head = slot_list[head];      /* End partition done automatically */
            do {                         /* For all other partitions ==      */
                PP = slotPP[head];       /* Last unsorted partition          */
                while (slot_size[head]){/* While this partition not formed  */
                    keyP    = *PP;       /* First unsorted key in partition  */
                    do {                 /* Rotate keys --                   */
                        c = keyP[off];   /* Partition keyP belongs in        */
                        slot_size[c]--; /* One less slot in partition       */
                        PP    = slotPP[c]++;
                        key1P = *PP;     /* Move key to the partition        */
                        *PP   = keyP;    /* that it belongs in and make      */
                        keyP  = key1P;   /* keyP the one this replaces       */
                    } while (c != head);/* -- until cycle finished          */
                    PP++;                /* Next unsorted key because        */
                }                        /* replacement has cycled           */
                head = slot_list[head]; /* Find key from earlier partition  */
            } while (head != 255);       /* == while not at end of list      */
            head = slot_head;
            slot_size[head] = 0;         /* Not necessarily zero             */
        }
```

Downloaded from http://comjnl.oxfordjournals.org/ by guest on December 18, 2014

```
        if (++off >= keysize) return;   /* No subsequent bytes in key       */

        head = slot_list[head];         /* Partition before last            */
        do {                            /* Sort each partition --           */
            PP      = slotPP[head];     /* Start of next partition          */
            radix_sort(PP,endPP,off);   /* Sort last unsorted partition     */
            endPP   = PP;               /* End of previous partition        */
            head    = slot_list[head];  /* Move back through partitions     */
        } while (head != 255);          /* -- until only the first not done */
}   }                                   /* Now sort first partition         */
```

Figure 1. Radix sort implemented.

```
insertion_sort()
{
    extern      char    *sort_tblPP[];      /* Array of pointers to keys     */
    extern      char    **end_tblPP;        /* Beyond end of array           */
                                            /* Contains pointer to highvalue */
    register    char    **PP, *P, *P1;

    for (PP = end_tblPP-1; PP > sort_tblPP;) {
        P1 = *PP--; P  = *PP;
        if (memcmp(P,P1,keysize) > 0) {     /* If P is out of sequence       */
            register    char    **PP1;      /* insert P at the proper point  */
                                            /* in subsequent key pointers    */
            PP1 = PP;
            do {
                *PP1++ = P1; P1 = PP1[1];
            } while (memcmp(P,P1,keysize) > 0);
            *PP1 = P;
}   }   }
```

Figure 2. Insertion sort implemented.

Like many efficient algorithms, the behaviour of our radix sort is poor when sorting small numbers of keys. For this reason no attempt is made to partition sets of less than 16 keys. Instead, an insertion sort is performed on the key pointers, after these keys have been processed by the radix sort.[15] This insertion sort is presented in Fig. 2.

Because the insertion sort is executed after the radix sort, it only makes local changes to the ordering of key pointers, and therefore has $O(N)$ operational complexity. If the radix sort is not performed first, the insertion sort would have $O(N^2)$ operational complexity.

The decision to sort sets of less than 16 keys using an insertion sort was based on theoretical and empirical studies of both Quicksort and the radix sort. In Ref. 15, Quicksort behaved best when only sorting partitions of at least 9 keys, but it was acknowledged that this optimum value would vary. It was also observed that 'The precise choice is not highly critical and any value between 5 and 20 would do about as well'. It is clearly desirable that small numbers of distinct keys should not be sorted using our radix sort, since any such attempt involves examining up to 256 integer counters at least once. An insertion sort can be expected to sort 15 keys using $(n(n+3)/4)-H_n \approx 65$ comparisons, while Quicksort (which has some additional minor inefficiencies) can be expected to perform only $2(n+1)H_n-4n \approx 46$ comparisons.[9] In the worst case both the insertion sort and Quicksort perform approximately $n(n-1)/2 = 105$ comparisons.

3. ANALYSIS OF RADIX SORT

Suppose that the radix sort is presented with N fixed-length keys, each containing m bytes, and further assume that these bytes are assigned random values from an alphabet of p symbols. Then the expected number of significant bytes per key is approximately $\log_p N$, whenever keys are distinct and $\log_p N \leqslant m$.[11] However, since we do not attempt to sort sets of less than 16 keys, this reduces to approximately $\log_p(N/16)$. In the worst case this increases to m. The radix sort examines each significant byte in each key at most twice; once when counting the frequency of byte values, and once when partitioning on these byte values. Thus we can expect to perform fewer than $c_1 N \log_p(N/16)$ operations when counting bytes and exchanging key pointers, and in the worst case will perform fewer than $c_1 mN$ operations, for some small constant c_1.

Consider now the number of times that our algorithm creates and subsequently uses a linked list to partition keys. When $p = 1$ this never occurs. So assume that $p < 1$. The expected number of internal nodes in a trie containing random keys drawn from an alphabet of $p > 1$ symbols is $N/\ln p$, and this corresponds to the expected number of attempts to partition keys. However, since we do not partition sets of less than 16 keys, this reduces to approximately $N/(16 * \ln p)$, when all such keys are distinct.[11]

A very much larger number of partitions may be constructed. If keys are always partitioned into r subsets and there are $r^k \leqslant r^m$ keys, then a total of $\sum_{i=0}^{k-1} r^i$ partitioning steps are performed, or $\sum_{i=0}^{k-1} r^i / r_k = \sum_{i=-k}^{-1} r^i$ partitioning steps per key. As k increases to m the number of partitioning steps per key therefore also increases, regardless of the value of r. Let $s > r$. Then, comparing terms, $\sum_{i=-k}^{-1} r^i > \sum_{i=-k}^{-1} s^i$. Thus the number of partitioning steps per key is larger when partitioning r^k keys r ways, than when partitioning s^k keys s ways. Since $r^k < s^k$, this implies that the maximum

Downloaded from http://comjnl.oxfordjournals.org/ by guest on December 18, 2014

number of partitioning steps per key is realised when r is a minimum and $k = m$. Thus in the worst case, $\sum_{i=-m}^{-1} 2^i = 1-2^{-m} \approx 1$ partitioning step will occur per key. However, since sets of less than 16 keys are not partitioned this reduces to 1 partitioning step per 16 keys.

Let the operational cost of constructing and then using the linked list, detailing the location of partitions, be at most c_2. Then c_2 will be a moderate constant, since the cost is dominated by the need to scan at most 256 integer values, and to recursively invoke the radix sort while stepping through a linked list containing at most 256 nodes. The expected operational complexity of the algorithm is therefore

$$N*(c_1*\min(m, \log_p(N/16)) + c_2/(16 \ln p)).$$

In the worst case this increases to $N*(c_1*m+c_2/16)$. Thus, as expected, the algorithm is $O(m*N)$, for any fixed p.

4. EMPIRICAL RESULTS

Although the operational characteristics of our radix sort are linear, and thus asymptotically better than any sort based on key comparisons, it remained unclear how effective our radix sort was likely to be in practice. Other sorts are very much better known,[12] and it seemed likely that the overheads associated with using this radix sort were prohibitive, when applied to any reasonable number of keys.

Since the proposed radix sort was superficially similar to Quicksort, and Quicksort is considered to be one of the better sorting algorithms, these two algorithms were implemented using 'C'. Both algorithms attempted to use efficient code, and were tested extensively, profiled, and compiled using optimisation. Care was taken to ensure that Quicksort performed reasonably well when presented with partitions containing ordered or duplicated keys. Effort was made to preserve commonality between these two algorithms, whenever possible. The Quicksort algorithm is presented in Fig. 3.

A third method of sorting, which used the standard AT&T Quicksort subroutine, *Qsort*,[1] was also studied, since it provided independent verification that our implementation of Quicksort was behaving reasonably. It is not known how this subroutine is implemented, but it may be related to its namesake.[17]

These three algorithms were executed repeatedly using different numbers of fixed-length keys, different sizes for these keys, and different distributions of data within these keys. Specifically, sorts were performed on 2^r keys, where r ranged from 4 to 16, having key sizes of 2^{2s} bytes, where s ranged from 0 to 3. During any one sort, each byte in each key contained a random value drawn from an alphabet of 2^t uniformly distributed values, where t was assigned the value 0, 1, 4, 5, 6, or 8.

When the alphabet contained fewer than the 256 possible byte values, it contained consecutive ASCII values beginning at 64 (i.e. '@'). Thus separate tests explored the behaviour of the various sort algorithms when examining duplicate keys, keys containing only two possible values in each byte, keys containing pseudo-numeric text, upper-case text, regular text, and arbitrary values.

The program designed to test all three algorithms was executed on an AT&T 3B4000, and ran under AT&T's UNIX System V (Release 3) operating system. During any one experiment all three algorithms operated on identical data. The average user execution time, associated with using each sorting method to completely sort all keys, was determined by repeating sorts many times on keys containing different randomly assigned characters. Sorts performed on more than 1000 keys were repeated 10 times, while smaller sorts (which used very little execution time) were repeated 100 times. The execution time was measured in units of $\frac{1}{100}$ of a second, and should reflect the time spent executing user code. It

```
quicksort(sortPP,endPP)
char    **sortPP;               /* Array of pointers to keys        */
char    **endPP;                /* Beyond end of array              */
{
    register    int     keys;
    register    char    *keyP, **highPP, **lowPP;

    while ((keys = endPP - sortPP) >= 16) {
        lowPP   = sortPP;
        highPP  = lowPP + (keys >> 1);
        keyP    = *highPP;      /* optimal partitioning key         */
        *highPP = *lowPP;       /* for sorted sequences             */
        highPP  = endPP;

        while (1) {             /* optimal partition for equal keys */
            do {
                if (lowPP == --highPP) goto done;
            } while (memcmp(keyP,*highPP,keysize) < 0);
            *lowPP = *highPP;
            do {
                if (++lowPP == highPP) goto done;
            } while (memcmp(*lowPP,keyP,keysize) < 0);
            *highPP = *lowPP;
        }
done:   *lowPP = keyP;
        quicksort(sortPP,lowPP);
        sortPP = ++lowPP;       /* Now sort first partition         */
}   }
```

Figure 3. Quicksort implemented.

Downloaded from http://comjnl.oxfordjournals.org/ by guest on December 18, 2014

is not known how accurate this execution time really is. Finally, this average execution time was divided by the number of keys being sorted, so that the average time to sort each key could be determined. Other possible measures of performance, such as system service time, elapsed time, memory usage, etc. are not reported.

5. RESULTS

The subroutine Qsort is inherently less efficient than our version of Quicksort, since Qsort performs key comparisons by indirectly invoking a user-specified subroutine. As expected, Qsort did not perform very well and typically took considerably longer than our implementation of Quicksort when sorting large files. However, Qsort performed better than our implementation of Quicksort when keys contained only a limited number of values, and better than our radix sort when sorting duplicate keys. This suggests that Qsort explicitly avoids partitioning identical sets of keys.

Table 1 documents the time to sort a single key, averaged over all experiments. Table 2 presents the average time to sort one of 2^{16} keys using each sorting method, alphabet, and key size studied. In Table 3 the time to sort 2^{16} keys using Quicksort and Qsort is then expressed as a ratio of the time to sort the same keys using the radix sort. For conciseness, no further details are provided for Qsort.

We now present detailed graphs (Figs 4–11), documenting the behaviour of the radix sort, and Quicksort. These graphs use a logarithmic scale to plot the total number of keys sorted against the time to sort each key. Constant results therefore suggest that the total sort time is $O(N)$, while linearly increasing results suggest that the total sort time is $O(N \log N)$. Since the time taken to sort 64-byte keys is considerably more than the time taken to sort 1-byte keys, not all graphs use the same vertical scale.

The graphs and tables presented below show very clearly that our radix sort is significantly faster than

Table 1. Average number of μs/key over all experiments

Alphabet size	1-Byte key			4-Byte key			16-Byte key			64-Byte key		
	Radix	Quick	Qsort	Radix	Quick	Qsort	Radix	Quick	Qsort	Radix	Quick	Qsort
1	18.9	109.9	42.0	25.6	153.5	56.7	90.3	376.2	81.1	341.2	1220.6	207.0
2	26.4	131.5	58.3	60.1	185.0	142.4	183.9	325.0	479.7	179.7	322.3	502.8
16	17.3	125.9	130.6	59.7	208.5	365.7	68.3	200.8	379.3	67.5	205.7	376.2
32	18.2	130.4	159.3	60.0	193.3	362.0	62.0	195.7	381.7	63.3	193.5	388.7
64	22.0	144.2	191.8	55.4	191.4	360.7	57.8	186.0	372.5	62.2	196.5	365.9
256	26.1	154.8	241.1	48.1	175.5	352.6	56.8	181.0	377.4	57.8	186.9	359.7

Table 2. Average number of μs/key when sorting 65536 keys

Alphabet size	1-Byte key			4-Byte key			16-Byte key			64-Byte key		
	Radix	Quick	Qsort	Radix	Quick	Qsort	Radix	Quick	Qsort	Radix	Quick	Qsort
1	15.8	200.3	41.8	32.9	295.3	49.9	96.9	685.0	79.8	354.2	2248.8	200.0
2	22.2	210.3	58.9	56.5	309.9	168.5	276.4	642.5	890.4	307.7	671.0	946.4
16	21.4	227.3	156.6	80.9	374.7	649.1	95.4	370.3	668.9	101.5	382.8	678.4
32	21.6	224.7	180.6	68.7	340.9	635.2	70.6	353.2	657.5	74.8	367.3	667.1
64	22.4	230.7	221.0	83.2	339.1	635.1	85.4	337.8	639.8	88.3	346.3	639.1
256	23.3	237.1	291.9	58.1	319.7	631.6	60.7	325.2	639.0	64.3	341.1	643.3

Table 3. Time to sort 65536 keys as multiple of radix sort time

Alphabet size	Number of bytes in each key							
	Quicksort				Qsort			
	1	4	16	64	1	4	16	64
1	12.68	8.98	7.07	6.36	2.65	1.52	0.82	0.56
2	9.47	5.48	2.32	2.18	2.65	2.98	3.22	3.08
16	10.62	4.63	3.88	3.77	7.32	8.02	7.01	6.68
32	10.40	4.96	5.00	4.91	8.36	9.25	9.31	8.92
64	10.30	4.08	3.96	3.92	9.87	7.63	7.49	7.24
256	10.18	5.50	5.36	5.30	12.53	10.87	10.53	10.00

Downloaded from http://comjnl.oxfordjournals.org/ by guest on December 18, 2014

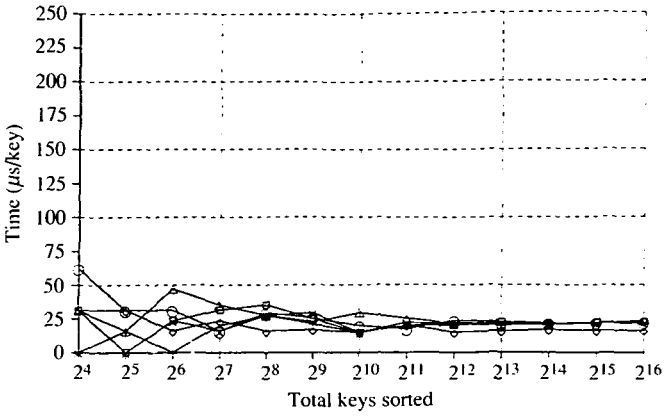

Figure 4. Time to radix sort a 1-byte key. Explanation of markings for Figs 4–11: ◇---◇, all keys same; ○---○, 2 symbols; +---+, 16 symbols; ×---×, 32 symbols; □---□, 64 symbols; △---△, 265 symbols (all).

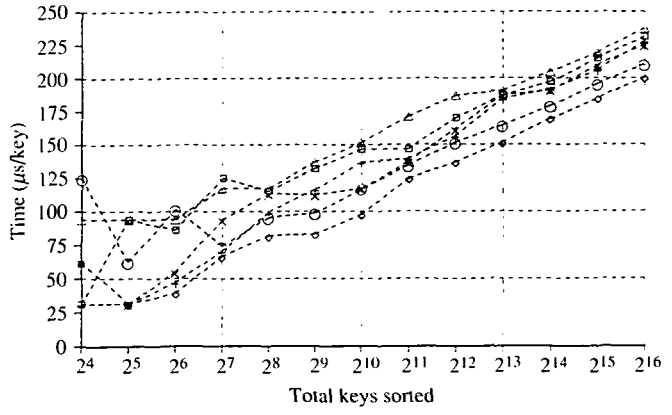

Figure 5. Time to Quicksort a 1-byte key.

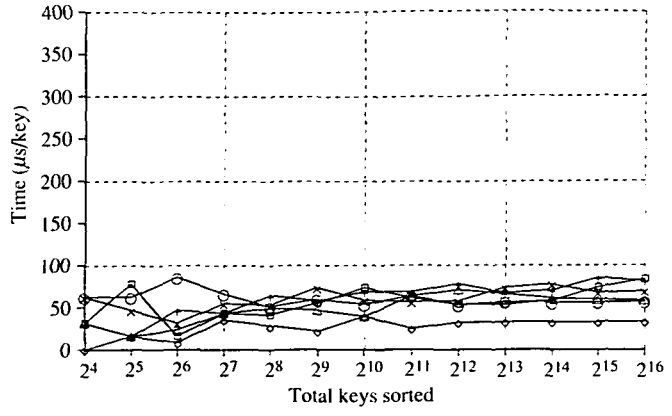

Figure 6. Time to radix sort a 4-byte key.

Quicksort. Quicksort was faster in only five isolated cases, three of which involved sorting 16 keys containing 16 bytes, one of which involved sorting 32 keys containing 64 bytes, and one of which involved sorting 64 keys containing 4 bytes. Since 72 sorts were conducted on 64 or less keys, and the overall execution time of very small sorts remains small anyway, this is not of great concern. In most cases the time to sort a key using our radix sort was essentially constant, compared to a time proportional to log N when using Quicksort. This becomes particularly significant when sorting large files.

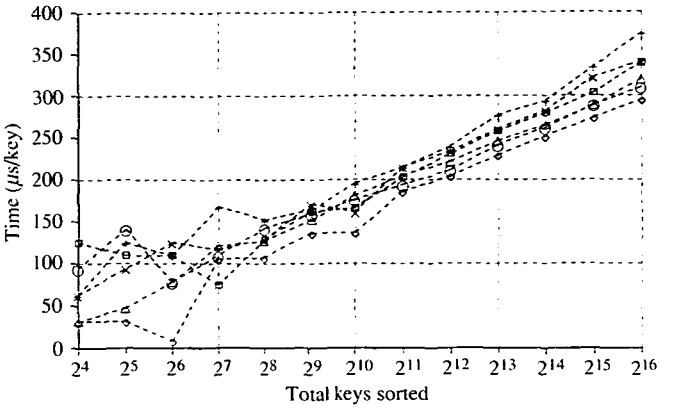

Figure 7. Time to Quicksort a 4-byte key.

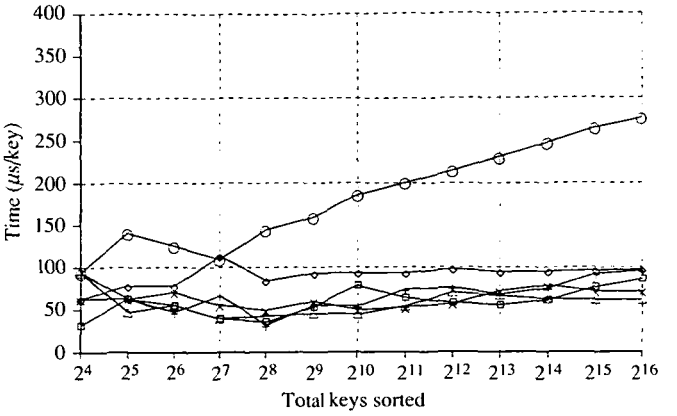

Figure 8. Time to radix sort a 16-byte key.

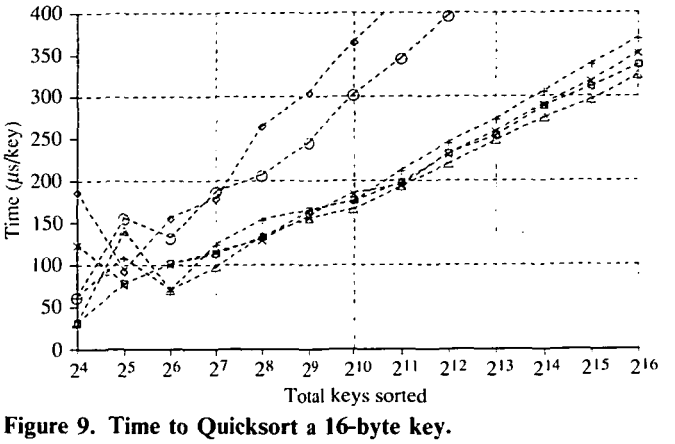

Figure 9. Time to Quicksort a 16-byte key.

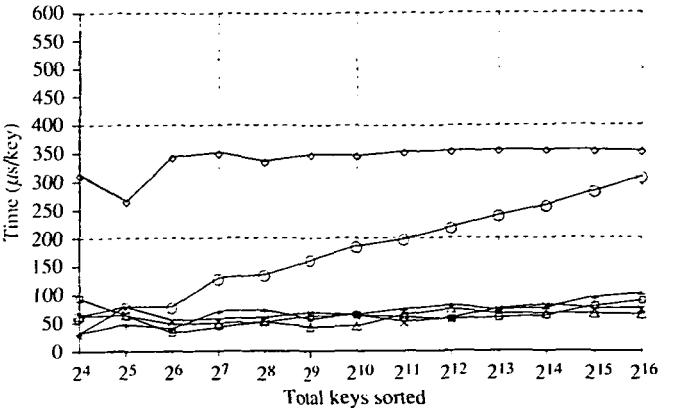

Figure 10. Time to radix sort a 64-byte key.

Downloaded from http://comjnl.oxfordjournals.org/ by guest on December 18, 2014

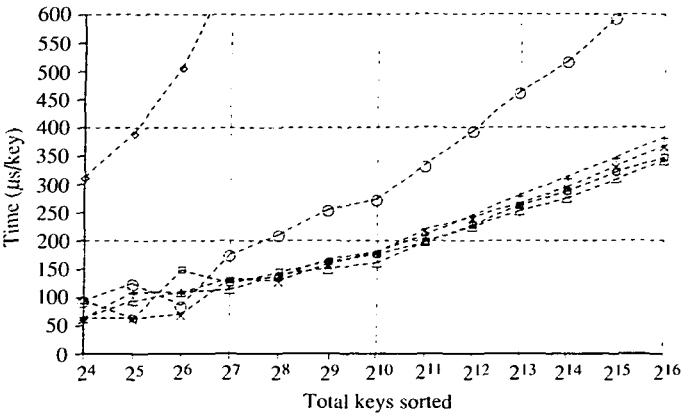

Figure 11. Time to Quicksort a 64-byte key.

6. CONCLUSIONS

This paper has presented an efficient implementation of the historical radix sort algorithm, and has shown that this implementation of a radix sort is superior to the more widely used Quicksort. In almost all experiments the radix sort was considerably faster than Quicksort, regardless of the number of keys being sorted, the size of these keys, and the contents of these keys. Out of a total of 312 different tests, Quicksort performed nominally better in only 5 tests, none of which involved sorting more than 64 keys.

There are cases where our radix sort should not be used. The partitioning is not stable, which may be significant in some applications. When performing external sorting, or in other applications where the cost of exchanging keys (or pointers to keys) is high, our algorithm can be expected to behave poorly. In applications where very little memory is available, the small amount of memory used by our radix sort might be viewed as excessive. In very critical applications, or those known to sort only a limited number of keys, other sort algorithms may be preferred. Finally, when sort keys cannot be divided into smaller keys having decreasing significance, no radix sort can be used.

Our radix sort can be used to sort text, binary values, and floating point values, in ascending or descending sequences, by merely employing encoding schemes which result in keys having the desired collating sequence. Our radix sort can also sort variable-length keys, if minor modifications are made to it. It would therefore be appropriate to consider using this sort in most practical applications.

Acknowledgements

I would very much like to thank Dr H. Bezner for motivating the study presented above, and for subsequently encouraging me to write this paper. I would also like to thank Dr D. J. Taylor for his continued interest in my research activities, and for his assistance in the production of this paper.

This research was funded by Wilfrid Laurier University. The production of this paper was funded, in part, by the Natural Sciences and Engineering Research Council of Canada, under grant A3078.

REFERENCES

1. Programmer's reference manual. In *UNIX System V Release 3*. AT&T (1986).
2. D. C. S. Allison and M. T. Noga, Usort: an efficient hybrid of distributive partitioning sorting. *BIT* **22**, 135–139 (1982).
3. J.-L. Baer and Y.-B. Lin, Improving quicksort performance with a codeword data structure. *IEEE Transactions on Software Engineering* **15** (5), 622–631 (1989).
4. C. M. Davidson, Quicksort revisited. *IEEE Transactions on Software Engineering* **14** (10), 1480–1481 (1988).
5. I. J. Davis, The MSQ database system. *HLSUA Forum XLIII Proceedings*, pp. 764–768 (5–8 October 1986).
6. I. J. Davis, The development of MSQ at Wilfrid Laurier. *Honeywell Bulletin* **5** (8), 6–20 (1987).
7. W. Dobosiewicz, Sorting by distributive partitioning. *Information Processing Letters* **7** (1), 1–6 (1978).
8. W. Dobosiewicz, The practical significance of D. P. sort revisited. *Information Processing Letters* **8** (4), 170–172 (1979).
9. G. H. Gonnet, *Handbook of Algorithms and Data Structures*. Addison-Wesley, London (1984).
10. C. A. R. Hoare, Quicksort. *Computer Journal* **5** (1), 10–15 (1962).
11. D. E. Knuth, Sorting and searching. In *The Art of Computer Programming*, p. 499. Addison-Wesley, Reading, Mass. (1973).
12. R. Loeser, Some performance tests on quicksort and descendants. *Comm. ACM* **17** (3), 143–152 (1974).
13. J. Rohrich, A hybrid of quicksort with $O(n \log n)$ complexity. *Information Processing Letters* **14** (3), 119–123 (1982).
14. R. S. Scowen, Algorithm 271, Quickersort. *Comm. ACM* **9** (5), 354 (1966).
15. R. Sedgewick, The analysis of quicksort programs. *Acta Informatica* **7**, 327–355 (1977).
16. F. Suraweera and J. M. Al-Anzy, Analysis of a modified address calculation sorting algorithm. *The Computer Journal* **31** (6), 561–563 (1988).
17. M. H. VanEmden, Algorithm 402, qsort. *Comm. ACM* **13** (11), 693–694 (1970).

Announcement

18–20 APRIL 1993.

RIDE-IMS '93 Third International Workshop on Research Issues in Data Engineering: Interoperability in Multidatabase Systems, Vienna, Austria. Sponsored by the IEEE Computer Society.

RIDE-IMS '93 is the third of a series of annual workshops on Research Issues in Data Engineering (RIDE). RIDE workshops are held in conjunction with the IEEE CS International Conferences on Data Engineering. Following the successful RIDE-IMS '91 held in Kyoto, Japan, the next RIDE workshop will also focus on interoperability of heterogeneous and autonomous database and knowledge systems.

The proceedings, consisting of the accepted papers, will be published by IEEE Computer Society and will be widely available.

For further information contact:
Elisa Bertino, Dipartimento di Matematica, Università di Genova. Tel.: +39-10-353-8034. Email: bertino@icnucevm.cnuce.cnr.it.
or
Susan Urban, Computer Science Department, Arizona State University. Tel.: +1-602-965-2784. Email: urban@asuvax.eas.asu.edu.

Downloaded from http://comjnl.oxfordjournals.org/ by guest on December 18, 2014