Radix Exchange—An Internal Sorting Method for Digital Computers*

PAUL HILDEBRANDT AND HAROLD ISBITZ

System Development Corporation, Santa Monica, California

Abstract This note describes a new technique—Radix Exchange—for sorting data internal to the high speed memory of an electronic binary digital computer The technique is faster than Inserting by the ratio $(\log_2 n)/n$ for sorting $n = 2^c$ items with values distributed evenly in the range 0 to $2^c - 1$ Its speed compares favorably with internal merging and it has the significant advantage of requiring essentially no working area in addition to that storage for the data being sorted and the instructions comprising the Radix Exchange routine itself The nomenclature used is, in most cases, that of E H Friend [1]

Introduction

Sorting on digital computers having a high speed random access memory (e.g., magnetic cores) and other "auxiliary" memory (tapes, drums, etc.) may conveniently be classified according to whether or not the data being sorted is internal to the high speed memory during all of the sorting process. Clearly, an over-all sorting problem may involve the transfer of blocks of data into memory, sorting these blocks, and merging. That part of the process concerned with sorting each individual block would, in this case, be considered an internal sort; whereas, alternatively, the transfer of data (from tape or drum) *item by item*, merging (for example) each item into its proper position in memory, would not be considered internal sorting.

When each of the items to be sorted is long in relation to the "key" field on which the data is to be ordered, it is sometimes convenient to sort words containing just the key and an address locating the remainder of the item. After the sort, the original data is put in the order determined thereby. The routine herein described is concerned only with the sorting of fixed-length items containing the key, whether they are the original or derived items.

Radix Exchange

The Radix Exchange method has been written for a fixed word length, internally binary machine. The restriction to binary is what makes this method feasible In general, a radix R sort involves the repeated distribution of the items into R "pockets" or areas according to a value of each digit in the key, expressed as a number to the base R using a positional representation. Unless the distribution of each digit is well known in advance, each of the R areas must be sufficiently

* Received August, 1958 The research reported in this paper was supported by the United States Air Force under contract AF33(600)-37684 with the System Development Corporation Reproduction in whole or in part for purposes of the United States Government is permitted

large to accommodate all the items, or there must be included program logic to take care of an overflow. When $R = 2$ (as here) we can and do use only *one* area of length equal to that of the F items to be sorted. Each end of the area is treated as the beginning of a separate area, thus giving the required $R = 2$ areas but a total length of F rather than $2F$.

Starting with the first item in a block to be sorted on a given pass on a particular bit b_i, we consider successive items to occupy the "zeros" pocket. Successive items working backward from the last occupy the "ones" pocket. The dividing line between the pockets is determined by sorting the items in the block as follows:

Items are scanned sequentially starting with the first in the zeros pocket, looking only at b_i. When a "one" is encountered, the ones pocket is scanned, starting with the last item in the block, until a zero is encountered. The two items are then *exchanged* and scanning continues (with the next item from the zeros end) until all items in the block have been examined. At the conclusion of a sort pass on the ith bit, b_i, all items with $b_i = 0$ precede all those with $b_i = 1$ in the block just sorted.

The method is analogous to block sorting on punched card equipment, but in the Radix Exchange the blocking is continued right down to the least significant bit. Let B denote the number of bits in each key. The complete block of items is first sorted on the most significant bit (b_B), exchanging as described above, yielding two subblocks. The first of these, the block with $b_B = 0$, is blocked (sorted) on b_{B-1} yielding two subblocks, etc. Thus, the process of sorting a block of items on a key of B bits consists of one sort pass on bit b_B followed by sorting two subblocks on $B - 1$ bits each. To describe the process recursively, let n represent a sort pass on bit b_n (b_1 is the least significant bit in the key). Let $S(m)$ be the sequence of positive integers whose terms represent the successive passes made in sorting a subblock on m bits. Then we have the following:

$$S(1) = 1$$

$$S(2) = 2, 1, 1 = 2, S(1), S(1),$$

that is, a sort on bit 2 followed by two 1-bit block sorts.

$$\begin{aligned} S(3) &= 3, 2, 1, 1, 2, 1, 1 \\ &= 3, S(2), S(2) \end{aligned}$$

and, in general,

$$S(B) = B, S(B - 1), S(B - 1).$$

In a similar fashion, we can describe the subblocks of items sorted on each pass. We note that a subblock (as used here) can be identified uniquely by specifying the value of each of the key bits of higher order than those in the subblock. For example, given a block of data with a 5-bit key, there would be two 4-bit subblocks denoted by 0 and 1, corresponding to the values of the most significant bit (b_5). Similarly, there would be four 3-bit subblocks: 00, 01, 10 and 11, etc.

Then, let (n, r) denote a sort pass on bit b_n over the subblock "r." (r is a nonnegative binary integer equal to the higher order key bits as above. For $n = B$, the most significant bit, we want to sort the entire block of data and we denote this by r = "blank".) Let $S(n, r)$ be the sequence of ordered pairs whose terms represent the successive sort passes made in sorting subblock "r" on n bits. Then we have the following:

$$S(0, r) = \text{no pass (included just for completeness)}$$

$$S(1, r) = (1, r)$$

$$S(2, r) = (2, r), (1, r0), (1, r1)$$

$$S(3, r) = (3, r), (2, r0), (1, r00), (1, r01), (2, r1), (1, r10), (1, r11)$$

$$= (3, r), S(2, r0), S(2, r1)$$

and in general,

$$S(B - j, r) = (B - j, r), S(B - j - 1, r0), S(B - j - 1, r1),$$

$$j = 0, 1, \cdots, B - 1, \quad r < 2^j \text{ (including leading zeros)},$$

$$j = 0 \Leftrightarrow r = \text{"blank"},$$

and finally,

$$S(B, r) = (B, r), S(B - 1, r0), S(B - 1, r1).$$

For example, sorting a block of items on a 3-bit key we have the following:

$$S(3, \ ) = (3, \ ), S(2, 0), S(2, 1)$$

$$= (3, \ ), (2, 0), (1, 00), (1, 01), (2, 1), (1, 10), (1, 11).$$

Details of the Method

With $F = 2^B$ distinct items to be sorted there would be F subblocks. The technique used, however, requires that we store at any one time only B subblock boundary addresses. The details of the method resolve into two parts: sorting passes over subblocks on one bit at a time as detailed above, and the computation of $S(B, \ )$ which determines the successive subblocks to be sorted and the bit upon which to sort on each pass.

The calculation of $S(B, \ )$ is accomplished in the following manner: (For purposes of discussion consider items of one word each, bearing in mind that the logic of the routine permits items of any fixed length, the complete item being exchanged as a unit.) Consider an address table of $B + 1$ words and one additional word, the "lower limit register," (LLR). Initially, the address of the first key is placed in the lower limit register, the address of the last key in word B, a completion indicator (e.g., a negative number) in word $B + 1$, and zero elsewhere. Each sort pass is made over the subblock bounded by the address in LLR and the address in word i of the address table, where i is the first nonzero word

found in searching the address table as described below. The sort pass is made on bit b_i, and, for $i > 1$, the dividing line so found (address of last item with $b_i = 0$) is placed in word $i - 1$ of the address table. For $i = 1$, the dividing address is discarded, the contents of LLR, $C(\text{LLR})$, are replaced by $C(1) + 1$ (or by the contents of word 1 plus the number of words in each item), and starting with word 2 the address table is scanned for a nonzero address. The first such address found, say in word j, is put in word $j - 1$, word j is cleared to zero and sorting resumed over the area bounded by addresses in LLR and word $j - 1$, on bit $j - 1$, then bit $j - 2$, etc. The process terminates when, in scanning the address table, we encounter the completion indicator.

If, when we are to begin a sort on b_j, the subblock to be sorted is either empty or contains just one item (indicated by $C(\text{word } j) - C(\text{LLR}) \leqq 0$), then we can omit this and all subsequent sorts of this area on bits b_i, $i \leqq j$. Hence, we immediately place $C(\text{word } j) + 1$ in LLR, zero in word j, and the address table is scanned for a nonzero address starting with word $j + 1$.

Example

The following example illustrates the method. The block of items and the address table are shown after each sort pass. The location of data in core memory is completely arbitrary. Initially, we have:

Data Block		*Address Table* (Located anywhere else in core)		
Core Address	Key Bits 321			
1000:	010	LLR:	1000	
1001:	111	1:	0	
1002:	100	2:	0	
1003:	101	3:	1005	
1004:	000	4:	−1	(Completion indicator)
1005:	001			

After (3,) over the entire block (words 1000–1005) we have:

1000:	010	LLR:	1000	
1001:	001	1:	0	
1002:	000	2:	1002	
1003:	101	3:	1005	
1004:	100	4:	−1	
1005:	111			

After (2, 0) (words 1000–1002) we have:

1000:	000	LLR:	1000	
1001:	001	1:	1001	
1002:	010	2:	1002	
1003:	101	3:	1005	
1004:	100	4:	−1	
1005:	111			

After (1, 00) (words 1000–1001) we replace $C(\text{LLR})$ by $C(1) + 1$ to yield:

1000:	000	LLR:	1002
1001:	001	1:	1001
1002:	010	2:	1002
1003:	101	3:	1005
1004:	100	4:	−1
1005:	111		

The address table is searched, starting with word 2, and we find $C(2) \neq 0$. Hence, we replace $C(1)$ by $C(2)$, put zero in word 2 yielding:

LLR:	1002
1:	1002
2:	0
3:	1005
4:	−1

At this point we are ready to sort on bit 1. However, $C(1) - C(\text{LLR}) \leqq 0$. Hence, we immediately replace $C(\text{LLR})$ by $C(1) + 1$ and search for a nonzero address starting in word 2. Such a word is word 3 and we replace $C(2)$ by $C(3)$, and put zero in word 3 to yield:

LLR:	1003
1:	1002
2:	1005
3:	0
4:	−1

After (2, 1) (words 1003–1005) we have:

1000:	000	LLR:	1003
1001:	001	1:	1004
1002:	010	2:	1005
1003:	101	3:	0
1004:	100	4:	−1
1005:	111		

After (1, 10) (words 1003–1004) and $C(\text{LLR})$ replaced by $C(1) + 1$ we have:

1000:	000	LLR:	1005
1001:	001	1:	1004
1002:	010	2:	1005
1003:	100	3:	0
1004:	101	4:	−1
1005:	111		

The same situation occurs as after (1, 00) and we replace $C(\text{LLR})$ by $C(j) + 1$, $j = 2$, and search for a nonzero address starting in word 3.

LLR:	1006
1:	1004
2:	1005
3:	0
4:	−1

The process then terminates when we find −1 in word 4.

Evaluation of the Method

The sort routine can be analyzed into three distinct areas: (1) a sorting pass on any bit, (2) the bookkeeping after completing a sort on b_i, $i \geq 1$, and (3) the bookkeeping after completing a sort on bit 1 (the least significant bit). The case most readily analyzed is that of sorting F distinct items, of W words each, whose keys are the F integers zero through $2^B - 1$. The three areas of the sort program then operate on the average (1) $FB/4$ times, (2) $2^{B-1} - 1 = F/2 - 1$ times, and (3), operating in two parts, 2^{B-1} and 2^B times, respectively. An actual count of the instruction cycles in each area yields the final result:

$$64\,\frac{FB}{4} + 36\left(\frac{F}{2} - 1\right) + 71\,\frac{F}{2} + 8F$$

or

$$16FB + 61F - 36 \text{ instruction cycles}$$

This particular result of sorting the first $2^B - 1$ integers has not been verified experimentally. Tests have been made on random numbers with, in general, $F < 2^B$, and F varying at constant B. The resulting time to sort was essentially linear in F. Similarly, sorting F ($= 1000$) random numbers (32 bits each) on 1, 2, $\cdots$, 32 bits, yield times linear in B up to about 20 bits. The extent of the linear range would be expected to increase with increasing F, at constant B.

For $F \ll 2^B$, where 2^B is the range of the B bit key, a better approximation to the time is probably achieved by substituting $\log_2 F$ for B in the preceding expressions.

Of the six internal sorting schemes mentioned by Friend [1], the only two in which the working storage area length[1] is $\ll O(F)$ (as is that of the Radix Exchange, it being only the address table length $B = O(\log_2 F)$) are "Inserting" and "Exchanging." Of these two, Friend states, "Exchanging is usually inferior"; hence we discuss it no further.

Inserting takes a length of time on the average equal to $O(F^2/2)$ to sort the first F integers. On this basis alone, the Radix Exchange is faster in the ratio $O(B/F)$ or $O(\log_2 F/F)$. This result is strictly applicable only to the average of times to sort $F = 2^B$ integers, with no duplicated or missing values. It is difficult to describe analytically the behavior of either method for different distributions of data. Qualitatively, we can state that Inserting is much more sensitive to the original distribution than is Radix Exchange. To get some measure of this sensitivity, the following experiment was conducted:

a. $F < 2^B$ random numbers were generated;

b. These F numbers were sorted by both methods (Radix and Inserting), and the times noted. (The computer on which these experiments were conducted has a built-in "clock" whose value at any time can be determined by the program.)

[1] Space needed in addition to that occupied by the sort program instructions and the F items.

c. The same F numbers but in reverse order were sorted by both methods (essentially to test the "randomness" of the random number generator).

d. The same F numbers were sorted and the sorted table sorted again by both methods;

e. The same F numbers were sorted, the order reversed, and this reverse order table sorted by both methods.

The results are tabulated below. While the numerical values depend on the particular computer, their ratios do not (except in the case of computers having special logic favoring a particular method). Certain observations are in order. The near equality of times to sort the random and reverse random tables suggests that the data were really random from the sorting standpoint. The time taken by Inserting in the worst case (reverse sorted) was over 700 times that for the best case. The Radix Exchange is much less sensitive to order (ratios 65/63 versus 2974/4). It is, however, in the case of $F \ll 2^B$, sensitive to where in the range the data actually lie, independent of their relative order. This sensitivity has not yet been measured.

TIME TO SORT

(32nds of a second)

	Radix	*Inserting*
(Random)	82	1457
(Reverse Random)	81	1524
(Sorted)	63	4
(Reverse Sorted)	65	2974

For random data, the Radix Exchange method was better than Inserting by a factor of 17.

Finally, we make the comment that the next most comparable technique, Internal Merging, takes time $O(F \cdot \log_2 F)$ and, in addition, requires a working area kF ($k \geqq 1$). Hence, Radix Exchange may prove more satisfactory.

Conclusions

While the Radix Exchange method was written for a particular computer, we do not feel that the computer logic and hence the experimental results above are biased against Inserting. Accordingly, we conclude that in situations where space is at a premium the Radix Exchange routine may be useful. Further, even where there is ample room in memory, it may prove to be faster than possible alternatives.

A large area we feel is worthy of further investigation is that of choosing *a priori* from among several sort routines to sort a particular set of data. To this end, we need to know more about the sensitivity of the routine to the distribution of the data. (It is easy to make quantitative statements only for very uniform or regular distributions.) The authors would welcome any further discussion in this area.

The Radix Exchange routine developed out of an extended conversation with Messrs. Hawley Rising of Massachusetts Institute of Technology Lincoln Laboratory and Jules Schwartz of System Development Corporation, both of whom contributed significantly.

REFERENCE

1. E H Friend, Sorting on electronic computer systems, *J Assoc Comp. Mach. 3* (1956), 134–168