Sorting

WILLIAM A. MARTIN*

Massachusetts Institute of Technology, Cambridge, Massachusetts 02139

The bibliography appearing at the end of this article lists 37 sorting algorithms and 100 books and papers on sorting published in the last 20 years. The basic ideas presented here have been abstracted from this body of work, and the best algorithms known are given as examples. As the algorithms are explained, references to related algorithms and mathematical or experimental analyses are given. Suggestions are then made for choosing the algorithm best suited to a given situation.

Key words and phrases: sorting

CR category: 5.31

INTRODUCTION

Sorting is used to put items in order. The sorting algorithms themselves are not difficult to understand, but a comparison of the relative merits of the many algorithms does require some effort. In fact, the question of when an ordering is required is not a simple one: for example, a file that is best maintained in sorted order when stored in magnetic tape might be kept more efficiently on disk with a scatter storage technique. Whether or not a file should be sorted depends on how it is to be used, the extent to which the storage medium can be randomly accessed, and the statistics of the particular item of information on which the file might be sorted. Once the various sorting algorithms have been analyzed, one can see how these factors come into play.

In data management applications it is customary to define a *file* as a collection of records, and a *record* as consisting of one or more *information groups*. Each information group may contain several *items of information*. Records within a file are often sorted, and sometimes information groups and individual items of information are sorted as well. Records are sorted by identifying a particular item of information in the record as the *key;* the records are then sorted into the ascending or descending order of their keys. Most sorting schemes involve moving the elements to be sorted from one place to another. The elements are generally moved several times before the final sorted order is achieved. Thus, when sorting records it may be better either to sort their keys first and then move the records into the final sorted order, or to use the sorted keys as an index to the records. Questions of this type will be considered in the latter sections of this paper, after presentation of the various sorting algorithms.

No one sorting technique is best for every situation. The fastest methods are more difficult to program and are not considered worth the effort for a few short lists of numbers. Even if programming effort is not a consideration, the choice of method would depend on: the length of the list to be sorted; the relation between the length of the list and the number of cells in the main memory of the machine used for sorting; the number and size of any disk or tape units used in the sort; the extent to which the elements are

* The work herein was supported in part by Project MAC, an MIT research project sponsored by the Advanced Research Projects Agency, Department of Defense, under Office of Naval Research Contract N00014-70-A-0362-0001.

CONTENTS

Copyright © 1971, Association for Computing Machinery, Inc. General permission to republish, but not for profit, all or part of this material is granted, provided that reference is made to this publication, to its date of issue, and to the fact that reprinting privileges were granted by permission of the Association for Computing Machinery.

already in sorted order; and the distribution of the values of the elements.

In addition to describing the techniques themselves, the following sections identify the principles used in their construction. A technique is generally described along with the presentation of the principles it best illustrates. Properties of the sorting techniques described in this survey are summarized in tabular form in the last section, "Comparison of Sorting Methods." The reader with a particular problem in mind may consult that section in order to decide which techniques he should study in detail.

MINIMAL STORAGE SORTING

If the list of items to be sorted is already in the main memory of the machine, it is often convenient to use a sorting subroutine that does not require many additional memory cells to those holding the items. A sorting routine can avoid using extra space if it sorts the items by *interchanging* them two at a time. Such a sort is known as an *interchange* sort.

The Bubble Sort

One of the simplest interchange sorts is the *bubble* sort [A18, A19, A27, A30]. The bubble sort is also a *comparative* sort because it determines which interchanges to make by comparing the items two at a time. An example of a bubble sort of a list of numbers is shown in Figure 1. The first two numbers in the list are compared. If the second is smaller than the first, as in this case, the numbers are interchanged; if the first is smaller, no change is made. Then the second and third numbers are compared. The third, 05, is smaller than the second, 19, so they are interchanged. Next, 05 is "bubbled" as far up as it will go. Comparing 05 with 13, we see that it is smaller, so another interchange is made. As 05 has now reached the top of the list, we go back and compare 19 with 27; since 19 is smaller, no interchange is made and we go to the next pair, 27 and 01. The sort is completed by proceeding in this manner.

The number of comparisons and exchanges

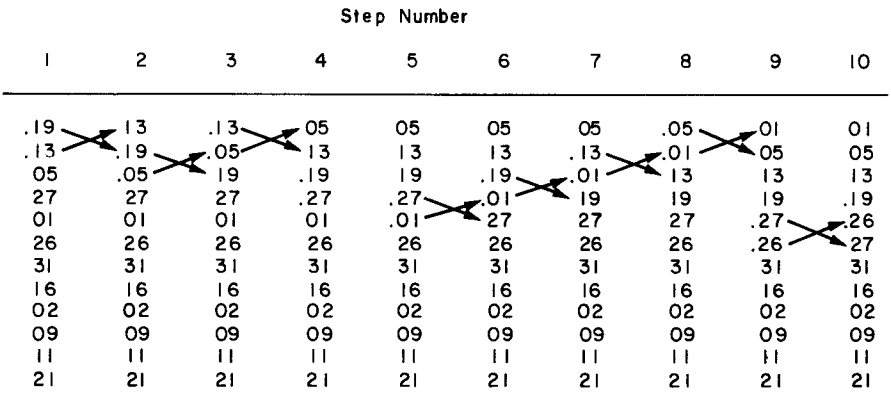

FIG. 1. The initial steps of a bubble sort. The dots indicate the elements being compared; the arrows indicate those being exchanged.

required to sort n items by a bubble sort depends on how badly out of order the items are to begin with. There will be, at most, $n - 1$ more comparisons than exchanges. If the items are in order, no exchanges are made. The maximum number of exchanges is required when the items are in reverse order, in which case $n(n - 1)/2$ exchanges must be made. An average of all possible permutations of the input items can be used as a measure of the number of exchanges required for an input list in random order. This average is easily found to be $n(n - 1)/4$. Since the sorts yet to be described have a behavior with n closer to $n \log n$ than to n^2, the bubble sort cannot compete in terms of time for long lists of items. However, the bubble sort is easy to remember and to program, and little time is required to complete a single step. One author found this sort to be best for lists of 11 numbers or fewer [A31].

The process of bubbling an item up has been explained as a series of exchanges. A slightly faster program can be written if the item to be bubbled is picked up and the resulting vacant position propagated toward the top of the list by moving successive items down one position. It is not necessary to store the bubbled item back in the list until the final step of the bubbling operation.

In a popular variation of the bubble sort, several passes are made through the list. The first pass puts the largest item in position n. On pass two, the next largest item is placed in position $n - 1$. Specifically, on pass j the item in place i is compared with the item in place $i + 1$ for $1 \leqslant i \leqslant n - j$. At each comparison an exchange is made if necessary. The sort is complete when a pass is made without any exchanges. The variation requires the same number of exchanges as the regular bubble sort, but usually more comparisons.

The Shell Sort

A well known comparative sort for longer lists of numbers is the Shell short [28, 49, 82, 90, A2, A5, A10]. The Shell sort is also an interchange sort in which several passes are made through the list of numbers. After each pass, the numbers are more nearly sorted. The last pass of a Shell sort is a bubble sort. A Shell sort is shown in Figure 2.

The ith pass of a Shell sort is done as for a bubble sort except that, instead of comparing adjacent numbers, we compare numbers a specified distance, d_i, apart; for example, in pass 1 of Figure 2, where $d_1 = 6$, we compare 19 with 31. Given a list of n numbers, we initially take d_i to be $2^k - 1$, where k has been chosen so that $2^k < n \leqslant 2^{k+1}$. On successive passes we take $d_{i+1} = (d_i - 1)/2$. This formula is not known to be optimum, but it has been found to be better than the choice shown in Figure 2 [82]. In this regard, it is instructive to note that when $d_1 = 6$, for example, we are, in effect, breaking the original list into six separate lists and sorting each of them with a

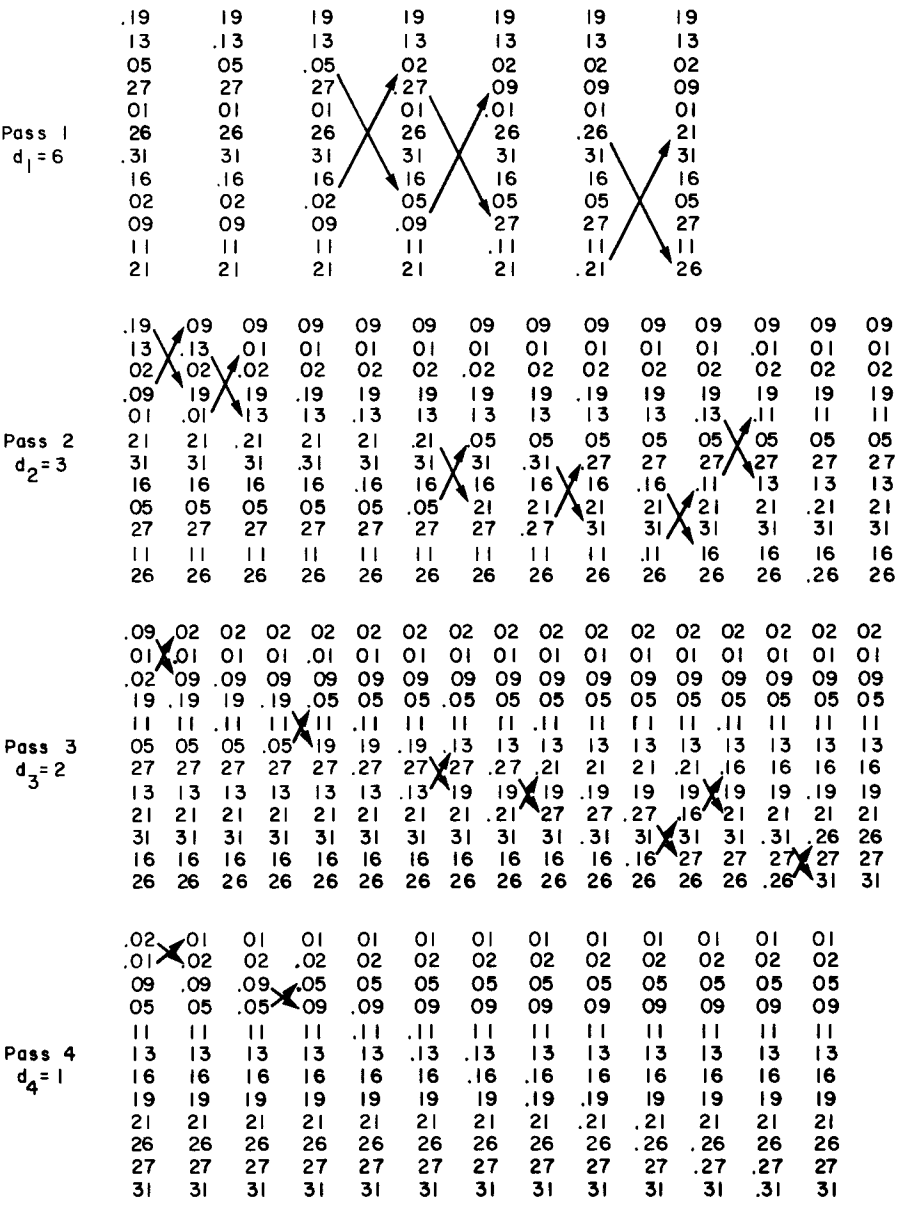

FIG. 2. A Shell sort. Dots indicate the elements compared at each step; arrows indicate the elements exchanged. The procedure for choosing the d_i is given in the text.

bubble sort. When we take $d_2 = 3$ we combine pairs of these six lists to form three separate lists. A little thought shows that this is because d_2 divides d_1. On the other hand, if we choose d_{i+1} so that it has no common factors with d_i, then each list at step $i + 1$ will contain some numbers from *each* of the lists at step i. It has been found that this makes the Shell sort faster. No exact derivations of the average time required by the Shell sort have been published. Approximate calculations and experiments indicate that it rises slightly faster with n than $n \log n$.

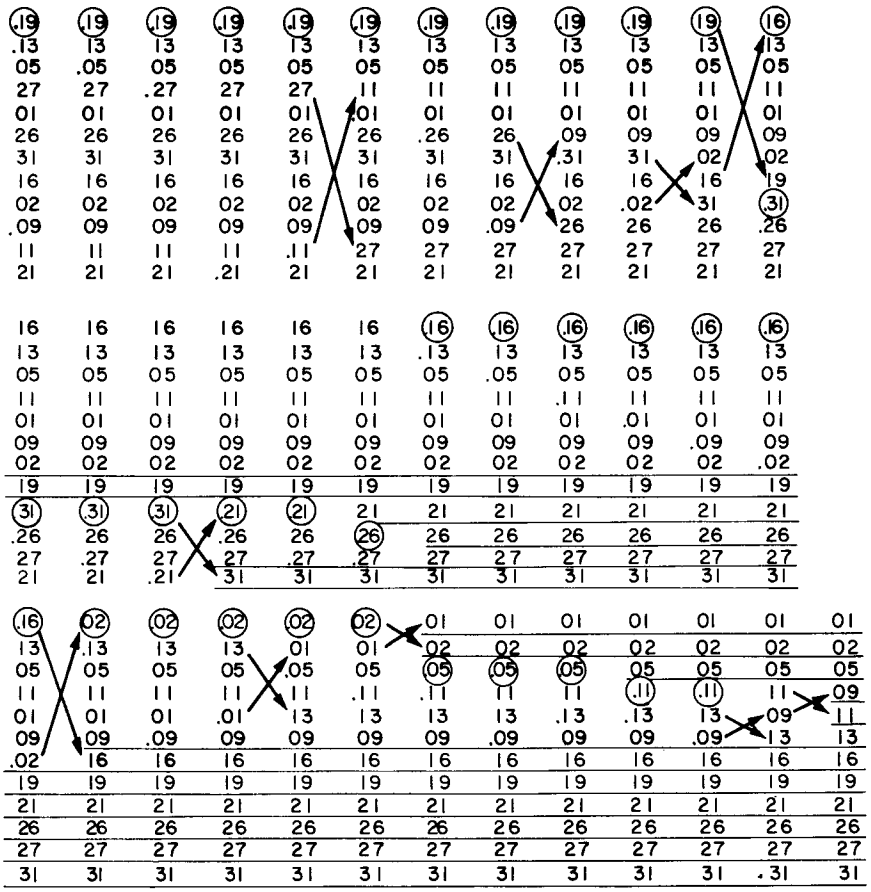

FIG. 3. Quicksort. At each iteration, Quicksort places all items less than or equal to the circled item above all items greater than the circled item. This is accomplished by scanning down from the top to find an item greater than the circled item, and then scanning up from the bottom to find an item less than or equal to the circled item. If such items are found, they are exchanged. When the two scans meet, the circled item is placed in the center.

Quicksort

Quicksort is a comparative, interchange sort that requires an average of about $2\,n \log n$ comparisons [48, 52, A17]. One would thus expect it to be faster than the Shell sort, and one author has found this to be the case [A4]. It requires only a small amount ($\log n$) of additional storage for bookkeeping. An example of Quicksort is shown in Figure 3.

In addition to being a comparative, interchange sort, Quicksort is also an example of a *distributive* sort. In each step of a distributive sort, a set of elements is partitioned into two or more subsets. Quicksort starts out by taking the first number in the list, 19, as an estimate of the median of the numbers. (Obviously, this can be a poor estimate and the sort can be improved by the estimating procedures given below.) Next, as shown in the first row of Figure 3, the sort places all numbers greater than 19 below it and all numbers less than or equal to 19 above it. This is accomplished by the following exchange procedure. Starting at the top of the list, the procedure scans down until a number greater than 19 is found. In Figure 3 the first such number is 27. Then, starting at the bottom of the list, the procedure scans up until a number less than or

equal to 19 is found. In Figure 3 the first such number is 11. Then 27 and 11 are interchanged. Again scanning down from the top, 26 is found to be the next number greater than 19, and from the bottom, 09 is found to be the next number less than or equal to 19, so 09 and 26 are exchanged. Similarly, 02 and 31 are exchanged. At this point, the scan coming down from the top meets the scan coming up from the bottom, and the algorithm knows that all numbers less than or equal to 19 lie above all numbers greater than 19. The final step of the partition is to exchange the bounding number, 19, with the number currently at the boundary, 16, so that 19 lies between the two subsets. The same procedure is now applied to each of the two subsets. Finally, only lists of length one remain, and the numbers are sorted.

On computers that do not have an exchange instruction, a modification such as the one used in the bubble sort may speed up the exchange procedure. The bounding number is picked up, and the resulting slot is filled with the first number to be moved from the bottom. The slot vacated by that number is then filled with a number from the top, etc.

When a list contains only two numbers it is not necessary to estimate a median. Clearly, the two numbers need only be compared and interchanged if necessary. In fact, it has been found that a bubble sort is faster than a version of Quicksort for lists of fewer than 12 numbers [A31]. Therefore, one may gain speed by switching to a bubble sort whenever Quicksort has sorted a large list down to small sublists.

The first pass of Quicksort divides a list

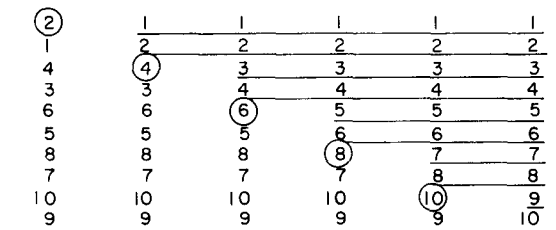

FIG. 4. When Quicksort produces an uneven division of elements, sorting the longest list first requires more memory.

of numbers into the median estimate and two sublists, each of which must be sorted in turn. Since a serial machine can only sort one of these at a time, it must remember where the second one is while it is sorting the first. A list can be remembered by placing the locations of its first and last elements on a pushdown stack. The following example makes it clear that the longer of the two lists should be remembered and the shorter one sorted immediately.

If the top list were remembered each time, the sort would go as follows: sort (2 1 4 3 6 5 8 7 10 9), remember (1); sort (4 3 6 5 8 7 10 9), remember (3); sort (6 5 8 7 10 9), remember (5); sort (8 7 10 9), remember (7); and sort (10 9). At this point four lists are being remembered, and one would have to go back and sort (7), then (5), then (3), and then (1). On the other hand, if the longest list were remembered, then the maximum number of lists to be remembered at any one time would be one instead of four. Thus, storage is saved by remembering the longest list (see Figure 4).

It is clear that an uneven division of the list at each step is undesirable. As shown in the previous example, the first element of a list does not necessarily give a good estimate of its median. When all permutations of n numbers are considered, Quicksort requires an average of about $2n \log n$ comparisons. A perfect choice of the median would bring this down to $n \log n/\log 2$, an improvement of 34%. Of course, the value of a good estimate of the median is not determined by how well it works on all possible inputs, but, rather, by how well it works on those inputs actually encountered. Note, for example, that Quicksort gives the worst possible estimate for input lists that have already been sorted. This particular failing of Quicksort can be remedied by using the median of the first, middle, and last elements of the list as the median estimate. This method has been found to produce a faster sort than Quicksort for other lists as well [A15, A23, A31].

The Radix Exchange Sort

Another variation of Quicksort is known as the Radix Exchange sort. It uses knowledge

of an upper bound on the magnitude of the numbers in order to estimate the median [51]. Let k be the smallest integer such that every number in the list is less than 2^k. The median of the entire list is taken as 2^{k-1}, that is, the center of the range 0 to 2^k. Each half of the range is then, in turn, divided in half until the sort is complete. It is clear that this method gives a good estimate if and only if the numbers are uniformly distributed. If the input list contained every number from zero to $2^k - 1$, the Radix Exchange sort would give a perfect estimate of the median.

Samplesort

Samplesort estimates the median by sorting a sample of the n objects of size j [29]. The median of the j objects is taken as the median of the n objects. The second partition of the n objects is made using the upper and lower quartile points of the sample. This process is continued until the n objects have been divided into $j + 1$ classes. Assuming that the sample and the objects in each class are sorted with Quicksort, the optimum sample size is shown below:

Range of n		*j*
from	*to*	
1000	2008	255
2009	4521	511
4522	10058	1023
10059	22154	2047
22155	48392	4095
48393	50000	8191

Note that j is of the form $2^k - 1$. For these values of n, Samplesort gives about 17% fewer comparisons than Quicksort. Since the best sample size is a significant fraction of the objects, one may wish to sort the $n - j$ objects remaining after the sample is taken and then merge the sorted sample objects with the result.

Using information theoretic arguments, van Emden has shown that the average number of comparisons for Quicksort is asymptotically equal to

$$\frac{n \log n}{-2 \int_0^1 g(x) x \log (x) dx}$$

where $g(x)$ measures the goodness of the median estimating procedure, $g(x)$ is the probability density function of the ratio x = (rank of median estimate)/(number of items in the list) [34, 95], the rank of an element being its index in the sorted order. Using this formula, van Emden shows that when the values of the n items are drawn from a uniform distribution, a sampling procedure such as that used by Samplesort works very well for large n. The average number of comparisons approaches the optimum, $n \log n/\log 2$, and the time spent in sampling becomes an arbitrarily small fraction of the total.

A Lower Bound

Morris has shown that *any* comparative sort will require an *average* number of comparisons asymptotically equal to $n \log n/\log 2$ [77]. Let the two results of a comparison be denoted by 0 and 1. Note that sorting the elements is equivalent to discovering their original permutation. Each original permutation must lead to a distinct series of zeros and ones for the comparisons made, otherwise the sort could not distinguish among the different permutations and make the appropriate rearrangements for each. For the same reason, no permutation can lead to a series of zeros and ones that is a prefix of a series produced by a second permutation. Thus, any comparative sort of n items must produce a unique series of zeros and ones, not the prefix of another series, for each of the $n!$ permutations of the items. There are 2^k unique strings of zeros and ones of length k. If $2^k \leqslant n! < 2^{k+1}$, then $n! - 2^k$ of the strings of length k must be extended to length $k + 1$. Thus $2n! - 2^{k+1}$ strings of length $k + 1$ and $2^{k+1} - n!$ strings of length k are required. The average series length, $\bar{s}$, is then

$$\bar{s} = \frac{(k+1)(2n! - 2^{k+1}) + k(2^{k+1} - n!)}{n!}$$

$$= k + 2 - \frac{2^{k+1}}{n!} \quad (2^k \leq n! < 2^{k+1}).$$

For large n, $\bar{s}$ approaches k, which in turn approaches $\log_2 n!$. $\text{Log}_2 n!$ approaches $n \log n/\log 2$ by Stirling's formula.

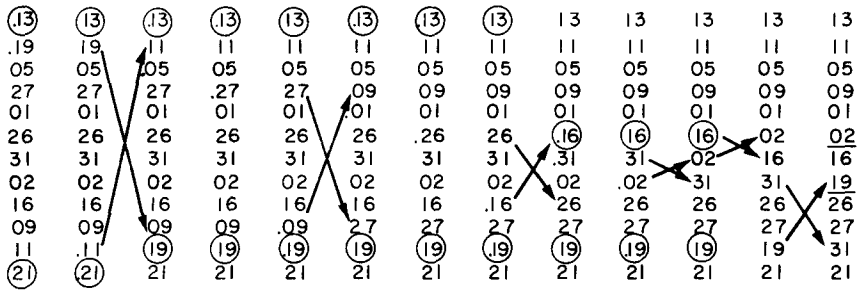

FIG. 5. Van Emden's exchange procedure for Quicksort. The circled elements are the numerical limits of the interval containing the median estimate; the dots and arrows mark the elements compared and exchanged at each step.

To obtain the absolute minimum average number of comparisons, a sort would have to take full advantage of the results of comparisons already made [8] and also ensure that the outcome of each new comparison is completely unpredictable. The best known method was constructed by Ford and Johnson [25], and was analyzed and improved for certain values of n by Hwang and Lin [55]. Finding the minimum number of comparisons is a challenging problem, but it is not of great practical importance because Quicksort is not only close to the minimum, but requires very little work per comparison.

Van Emden found that the time required by Quicksort could be reduced by about 15% merely by postponing the choice of a precise median estimate as long as possible [95, A32]. Instead of a median estimate, an interval that must contain the estimate is maintained during the exchange procedure. Initially, the larger and smaller, respectively, of the first and last elements of the list are used as the upper and lower limits of the interval. If the last element of the list is smaller than the first, these elements are interchanged. The algorithm then repeats the steps of an exchange procedure until the list has been partitioned.

An example of the exchange procedure is shown in Figure 5. The algorithm scans down from the top of the list until an element larger than the lower limit of the interval is found. Next, it scans up from the bottom until an element smaller than the upper limit of the interval is found. If neither of these elements falls within the interval they are exchanged as in Quicksort, and the next step in the exchange procedure is performed. Otherwise, they are exchanged if necessary so that the smaller lies above the larger; then the interval is decreased in size so that neither of these elements lies within it.

DISTRIBUTIVE SORTS

The Quicksort, Radix Exchange, and Samplesort algorithms are known as distributive sorts because they distribute the items into subsets, such that all the items in one subset are greater than all those in the other. The items are divided two ways at each iteration, and about $\log_2 n$ iterations are required. If the items were divided evenly in k ways at each iteration, only about $\log_k n$ iterations would be required. In order to distribute the items in k ways at each iteration, k buckets are required. Note that the total number of items to be stored in any bucket can vary from 1 to n, while the total number of items in all buckets is n.

There are four different ways to set up the buckets. First, suppose we have available as many sets of k buckets, each of length n, as are required. New buckets can then be taken as they are needed. If this is not the case, a more complicated method must be used. This second method involves choosing a value of k large enough to ensure that, at most, one item will fall in a bucket. All buckets can then be of length one, and only one iteration would be needed. For example, any subset of the integers from 1 to k can be

sorted by taking an array of k consecutive cells and assigning the integer with value i to the ith cell. After the items have been assigned, a linear sweep through the array can be used to compact them into consecutive cells.

The third method is an elaboration of the second: k is chosen sufficiently larger than n so that the probability that more than one item will fall in a bucket is small [21, 22, 46, 56, 76]. The buckets are then taken to be k consecutive cells, as in the second method. Whenever an item falls into a bucket that is already full, it is placed in an adjacent bucket, as shown in Figure 6. The item is moved up or down from the home bucket until its place in the sorted order is found, and other items are shifted as necessary so that it can be inserted. An example of a sort of this type is shown in Figure 7. This is known as an *address calculation* sort because the address of the home bucket is calculated from the value of the item.

The fourth method uses list processing techniques [30, 70, A34, A36]. Each of the k buckets contains a pointer to a linked list which contains all of the items that were distributed into that bucket. Exactly n linked list elements are required no matter how the items are distributed into the various buckets. Only k cells, $A(1), \cdots, A(k)$, are needed to contain the pointers to the beginnings of each of the k linked lists.

The *radix* sort, used on card-sorting machines, employs the same set of k buckets at each iteration [29, 46, 66]. An example of a radix sort is shown in Figure 8. First, the numbers are distributed into ten buckets according to the values of their least significant digits. They are then removed from the buckets and recombined into a single list so that the order of the least significant digits is preserved, as shown. The buckets are then re-used for a distribution on the tens digit. The second recombination produces a fully sorted list.

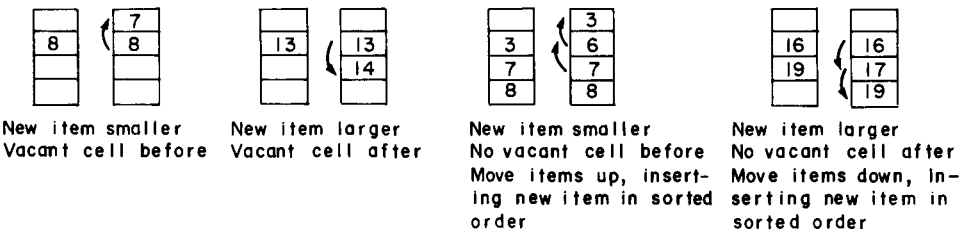

FIG. 6. Resolution of identical address assignments in an address calculation sort.

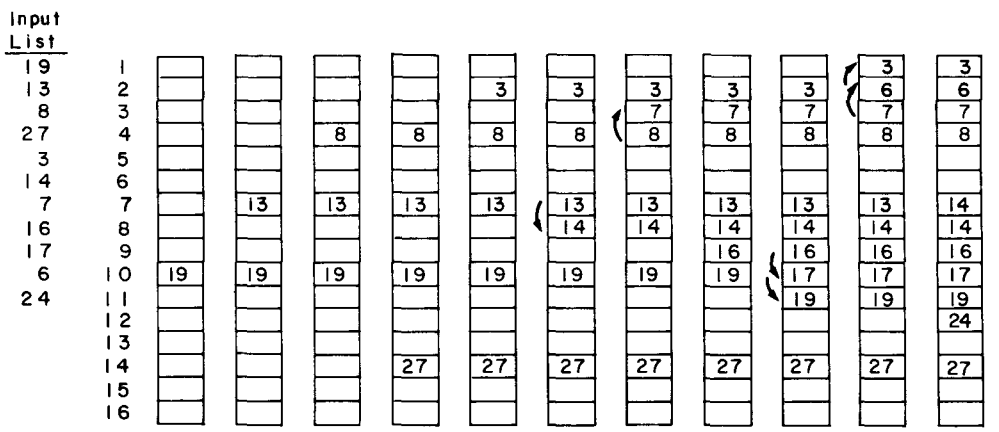

FIG. 7. An address calculation sort of 11 numbers. Addresses are calculated by dividing the item value by 2 and rounding up. Identical assignments are resolved as shown in Figure 6.

Original Table	First Distribution	Merge	Second Distribution	Final Merge
19		01		01
13	0)	31	0) 01,02,05,09	02
05	1) 01,31,11,21	11	1) 11,13,16,19	05
27	2) 02	21	2) 21,26,27	09
01	3) 13	02	3) 31	11
26	4)	13	4)	13
31	5) 05	05	5)	16
16	6) 26,16	26	6)	19
02	7) 27	16	7)	21
09	8)	27	8)	26
11	9) 19,09	19	9)	27
21		09		31

FIG. 8. Demonstration of radix sorting.

ADDRESS CALCULATION

Most distributive sorts are improved by an even distribution of the elements into buckets at each iteration. The degree of improvement is determined by the extent to which the resources required to sort the items in each bucket grow more than linearly with the number of items. (Note that the radix sort, described above, is not improved by an even distribution.)

For example, consider an address calculation sort such as the one shown in Figure 7. The time required by this sort is divided among three operations: calculating a home address for each item, inserting the item in sorted order when the home address is full, and compacting the output area after all items have been entered. If the number of buckets is chosen proportional to n, then the time required for the first and last operations is proportional to n and does not depend on the distribution of items into buckets. If at most one item falls into each bucket, the insertion time is zero. On the other hand, if all items fall into the same bucket, the sort becomes an *insertion* sort with insertion time proportional to n^2 [22]. Thus, the total sort time can range from very good to very bad depending on the distribution of items into buckets. If one knows the probability distribution from which the values of the input items are drawn, it may be possible to find an address calculating function that will produce at least a random distribution of addresses—that is, one where the fact that a cell is already full neither increases nor decreases the probability that another item will fall into it. The insertion time for a random distribution is proportional to

$$\frac{n\left(1 - \frac{n}{2k}\right)}{1 - \frac{n}{k}} \quad [76]$$

In this case, by keeping the *loading factor* or *fullness ratio*, n/k, constant, the total sort time can be made proportional to n. Thus, using a knowledge of the number of items and their values, one can break the "n log n barrier."

An interesting question at this point is whether one can obtain enough information about the items to realize a linear growth of sort time with n by making a computation on the items that is also linear in n. The type of information required is the cumulative distribution of item values [56]. If one knows some facts about a distribution from which the items were drawn, then the parameters of the cumulative distribution can be estimated. For example, the algorithm Mathsort [A11, A26] sets up an array in which a distribution of the values of the items is built by making one pass through them. The range of item values is broken into k segments, and the number of items with value in each segment is tabulated. Some a priori knowledge about the distribution is required to set up such an array. Without any knowledge of the distribution, it appears necessary to sort the items to find the cumulative distribution.

In practice, distributive sorts that result

in worse than a random distribution are used. In one such method, the values of the items are expressed as numbers to the base k [4]. The first distribution is made on the most significant digit; all those with the same most significant digit are distributed on the next digit; etc. If the values of the items are words, successive distributions can be made on successive letters [93]. This usually produces a worse-than-random distribution for the first few iterations, but then a better-than-random distribution as the ends of the words are reached. There is often a tradeoff between the uniformity of the distribution into buckets and the time required to calculate it. For example, a distribution of words on their first letters can be improved by referencing a table showing the relative frequencies of first letters.

ITEM-AT-A-TIME SORTS

Typically, a list of items is stored in sorted order so that any given item in the list can be more easily located. New items may be added to the sorted list from time to time. If the original list is sorted by adding items one at a time to those already sorted, then the same sorting procedure can be used both to add new items and to look up old ones. To look up an old item, one need only follow the sorting procedure through to the position in which that item should be. Distributive sorts can be done one item at a time by saving all the buckets used at each iteration and any information, such as median estimates, that was used to make the distributions.

For example, consider the distributive sort shown in Figure 3. Using pointers, this can be represented in memory as the tree shown in Figure 9 [48]. Each node in the tree contains the item used to estimate the median, an up-pointer, and a down-pointer. During sorting, as each item reaches a node it is placed above or below the node depending on whether its value is less than or greater than that of the item at that node. For example, to add the item with value 17 to the tree in Figure 9, we must first compare 17 with 19. As 17 is less than 19, the up-pointer is followed to 16; 17 is greater than 16, so 16 is given a new down-pointer to 17. Since items arrive one at a time, the first item, 17, is taken as the median estimate for all items greater than 16 and less than 19. Since the first item to reach a node is used to estimate the median of those that follow, the form of the tree structure depends on the order in which the items arrive. The most desirable tree will have the most frequently looked up

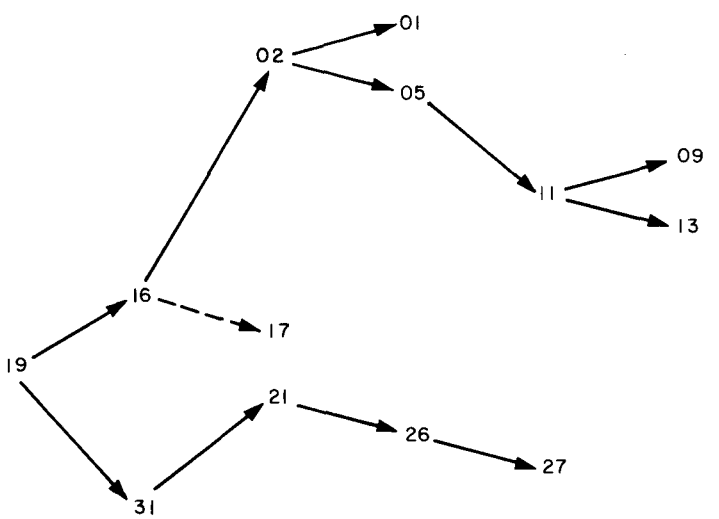

FIG. 9. Tree structure form of a binary distributive sort.

items near the root; thus, it may pay to convert a bad tree into a good one [26, 61, 73].

There are two popular methods for representing the nodes when the distribution is into k buckets. The first, known as *trie memory*, takes each node as an array of length k [30]. The second takes each node as a linked list structure with an element for each bucket that actually contains one or more items [93].

MERGING

Two or more sorted lists may be combined into one longer one by *merging* them, as shown in Figure 10. (They are merged here in ascending order, but they could also be merged in descending order.) Each step of the merge consists of adding to the output list the smallest number that remains on any of the input lists. A list of n numbers can be sorted by starting with n lists of length one and merging these into longer and longer lists. This series of merges forms a tree with the final list at the root.

It is interesting to contrast this procedure with that used in a distributive sort. A distributive sort is an example of a *top-down* algorithm; i.e., one that partitions a set of items into smaller and smaller subsets. A merge sort is a *bottom-up* algorithm, one in which individual items are assembled into larger and larger subsets. Just as the best results are obtained in a distributive sort by dividing the items into subsets of equal size, the best results in merging are obtained by combining the lists so that all lists at the same depth in the merge tree are of equal size [12]. In distributive sorting, an even partition depends upon our ability to estimate how many items will fall into a given range of values. On the other hand, the lengths of strings to be combined during merging can be planned in advance, independently of the item values. The best plan gives an average number of comparisons close to the lower bound, $n \log n/\log 2$, mentioned earlier. To see that this is so, note that at least one item goes into the output string for each comparison made in a merge of two input strings. Thus, using binary merges, the total number of comparisons in a merge sort of n items is fewer than the number of items in all strings of the resulting binary merge tree. A balanced tree will have a depth of $\log n/\log 2$. Since the total number of items in all strings at a given level is $\leqslant n$, we have a total number of comparisons $\leqslant n \log n/\log 2$.

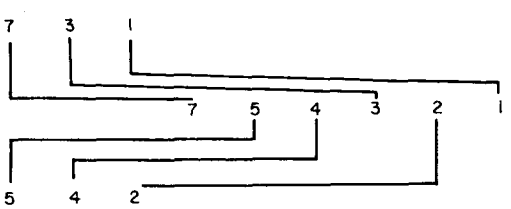

FIG. 10. Merging two sorted lists to form one longer sorted list.

Merge sorts have not replaced distributive sorts because they cannot be done in place; they cannot be done an item at a time; they require at least as many data moves as comparisons; and because knowledge of the distribution of item values cannot be used to reduce the average number of steps below $n \log n/\log 2$. Readers familiar with list processing will note that if the items to be sorted are the members of a linked list, then a merge sort can be carried out by manipulating the pointers of this list [100, A35, A37].

Batcher has observed that many of the comparisons required to merge two lists of items can be made in parallel [3]. Currently, special hardware is required to implement such parallel computation efficiently. The basic step in Batcher's parallel merge is shown in Figure 11. Two lists of length one can be merged by a simple comparison. To merge longer sorted lists, the odd elements of one list are merged with the odd elements of the second list, and, at the same time, the even elements of each list are merged. The resulting two merged lists are then combined as shown in Figure 11. The squares in the figure are two-item sorters; a two-item sorter takes just two items as input, and interchanges them if necessary to put them in sorted order. It is not obvious that this process merges the items as desired, but it can be proved by a straightforward consideration of the locations in which an item in

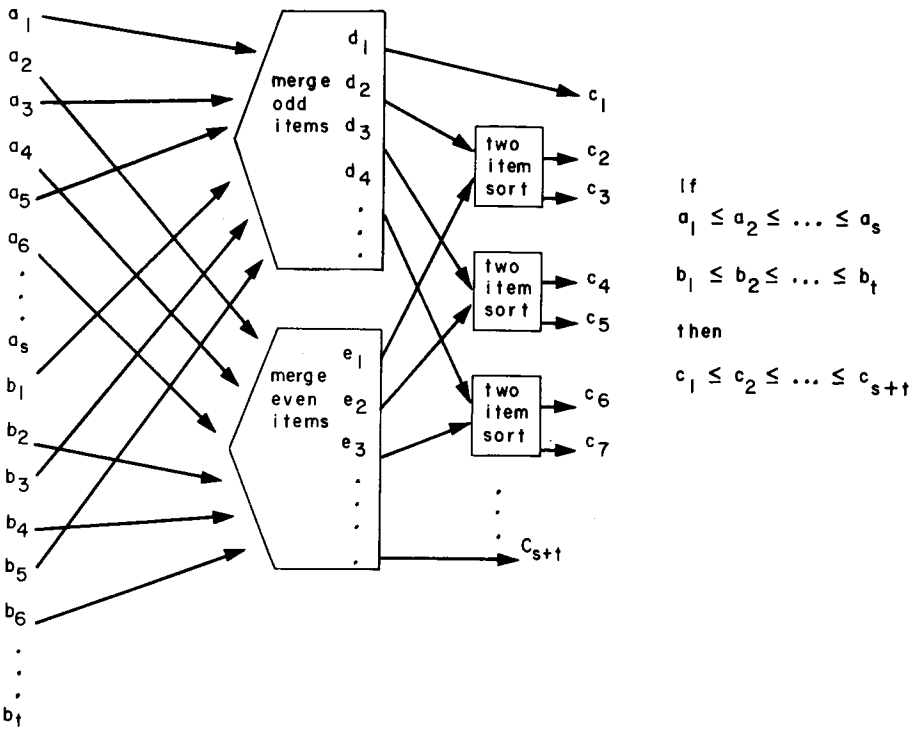

FIG. 11. The basic step in a parallel merge.

one of the input lists can possibly end up [68], or by an algebraic approach [67].

A second method of merging, also found by Batcher, is illustrated in Figure 12. This method is harder to understand and requires more two-item sorters. However, if the method is realized in hardware it is more flexible (one network can take inputs of different length) and modular. In Figure 12, the outputs $c_1 \cdots c_{2n}$ will be sorted if the inputs $a_1 \cdots a_{2n}$ form a *bitonic* sequence. To define bitonic, consider that the inputs $a_1 \cdots a_{2n}$ form a circle such that a_1 follows a_{2n} ; then find the smallest item a_s . The sequence $a_1 \cdots a_{2n}$ is bitonic if, starting at a_s and going once around the circle of items, the items first increase monotonically to a maximum and then decrease monotonically back to a_s .

The minimum number of two-item sorters required for a network using a *fixed* sequence of comparisons to sort a list of length n is not yet known for arbitrary n. From Morris' result we know that at least $n \log_2 n$ comparisons, and thus $n \log_2 n$ sorters, are required. Batcher's first network requires $(n/4)\ (\log_2 n)^2$. It is the best known general method; however, better networks have been constructed for particular values of n [10, 23, 24, 62].

REPLACEMENT SELECTION

When m sorted lists are merged simultaneously, the smallest (or largest) of the m items must be chosen at each step. This is known as a *selection* sort [22, 46], and it involves sequencing through the items, keeping a running track of the smallest found. By using additional storage, the number of items to be compared at each step can be reduced. For example, as shown in Figure 13, a set of m lists can be divided into $\sqrt{m}$ subsets, each containing $\sqrt{m}$ lists. The smallest item in each of the $\sqrt{m}$ subsets is found and stored in memory; the smallest item of all is then the smallest of these $\sqrt{m}$ items and the winner of the first round. This

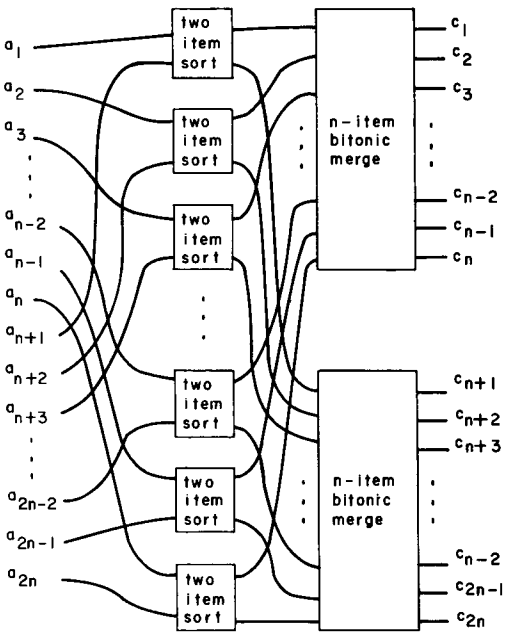

FIG. 12. A $2n$ item bitonic merge.

method of finding the smallest item requires $m - 1$ comparisons at the first step, as does the straightforward selection sort. However, far fewer comparisons are required in the subsequent steps, for in rounds after the first the smallest item in each of the $\sqrt{m}$ subsets remains unchanged, except in the subset that contained the winner of the previous round. The above sort is known as a *quadratic replacement* sort. Clearly, one can have cubic, quartic, etc., sorts. Higher-order sorts require fewer comparisons and more storage. When binary comparisons are made at each node the sort is known as a *tournament replacement* sort; it requires the fewest comparisons [22].

Note that the same number of comparisons are required when nine lists are merged by the quadratic replacement sort as when three lists are merged at a time and the resulting three lists then merged. (The important difference is in the temporary storage requirements.) Because the tournament

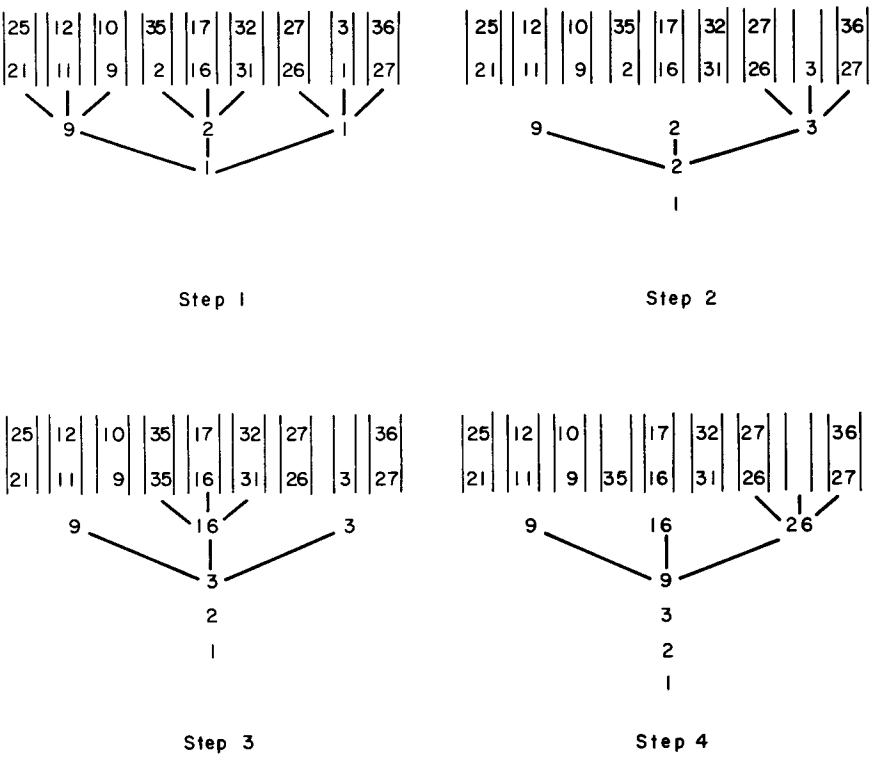

FIG. 13. The first steps of a quadratic replacement sort of 9 lists.

sort requires the fewest comparisons, a binary merge tree does also.

SORTING WITH TAPES

Magnetic tapes are used when the items to be sorted cannot fit into core memory at one time. Since tapes must be read and written sequentially, merge sorts are often used. The problem is how to best distribute the various merged strings onto the different tapes.

The Polyphase Sort

With six tape units or fewer, the best known method is the *polyphase* sort [22, 36, 37, 42, 59, 86, 87]. Let us consider a polyphase sort of nine numbers using four tapes, as shown in Figure 14. Step 1 shows the nine unsorted numbers on tape 1; the other three tapes are empty. In step 2, the numbers are read from tape 1 and distributed onto tapes 2, 3, and 4; all tapes have then been rewound. The number of items placed on each tape is determined by a formula explained below. At this point, tape 2 is considered to contain two sorted strings of numbers of length 1, and tape 4 is considered to contain four sorted strings of numbers of length 1. We then merge the first string from tapes 2, 3, and 4 to form one sorted string (05 19 26) of length 3, which is placed on tape 1. Next, the second string from tapes 2, 3, and 4 is merged to form the string (13 27 31) of length 3, which is also placed on tape 1. Tape 2 is now empty, so we begin merging onto it. We first rewind tapes 1 and 2 so that the numbers can be read in the order in which they were written out. Then, in step 4, we merge a string (05 19 26) of length 3 from tape 1, a string (01) of length 1 from tape 3, and a string (16) of length 1 from tape 4 to form the string (01 05 16 19 26) of length 5, which we place on tape 2. Each tape, except 3, now contains one sorted string. We rewind tapes 2 and 3 and merge these three strings to form the answer on tape 3, as shown in step 5.

The initial distribution of numbers in step 2 was chosen so that we would end up in step 4 with one sorted string on three of the four tapes. To study this situation let us form a table showing the number of sorted strings on each tape at each step (see Figure 15). To find the correct distribution in step 2, we must start with the desired distribution in step 4 and work backwards. Thus, to get from step 4 to step 3, we must add the number of strings on tape 2 in step 4 to each of the numbers for the other tapes in step 4, which results in the number of strings for each tape in step 3. One can write equations for this process to estimate the efficiency of the sort [22, 59].

Figure 15 also shows us that only certain numbers of initial strings allow the polyphase sort to "come out even." For example, with four tapes the exact numbers are 1, 3, 5, 9, 17, 31, 57, ... (For k tapes, each number in this series is the sum of the previous $k - 1$ numbers.) During the distribution phase one should proceed as if there were enough null strings available to bring the initial number of strings up to some exact number [71]. The optimum number of null strings to use and the choice of tapes to which they should be distributed has been studied by Shell [91]. This problem is further clarified in Figure 16.

Step 1 of this figure shows the initial distribution of 57 strings to four tapes. Each string is represented by a square; the number written in each square represents the number of times that those items in the string initially occupying that position on the tape must be read during the remainder of the sort. The derivation of these numbers is straightforward if one works back from step 7. Suppose that some of the 57 strings in step 1 are null strings. Presumably, tape reading time is minimized if the null strings are assigned to the string positions that are read most often, since reading and merging null strings requires only internal bookkeeping. Assuming that the total number of non-null strings to be distributed is known, Shell gives an algorithm for the optimum distribution of null and non-null strings. Since, as can be seen in Figure 16, the string positions read most frequently occur at various positions along the tapes, it is not surprising that this algorithm is complex. It is interesting to note that the criterion of minimizing the non-

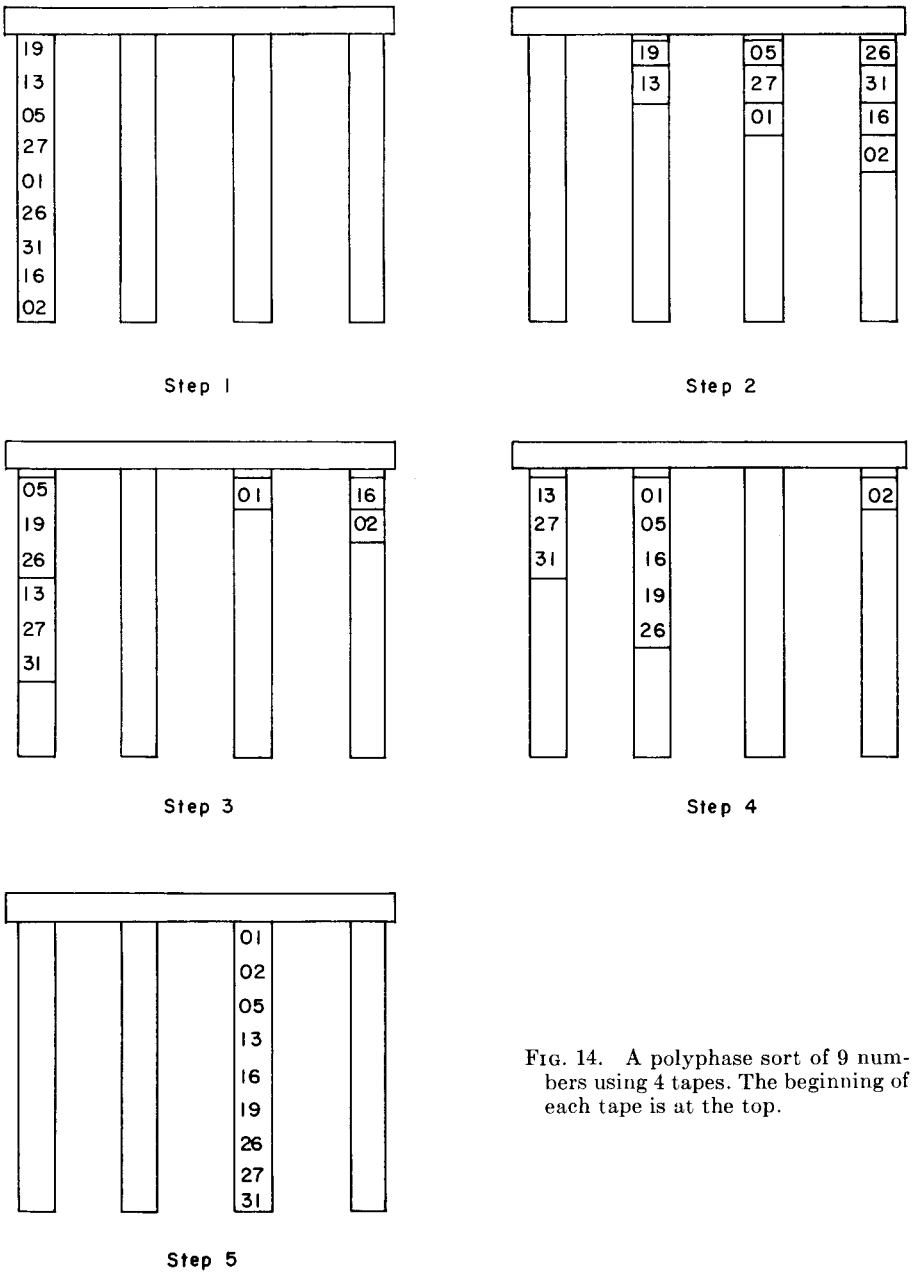

FIG. 14. A polyphase sort of 9 numbers using 4 tapes. The beginning of each tape is at the top.

null strings read sometimes leads to using more null strings than just enough to reach the next exact number. Since each additional exact number reached requires one more merge step, with the accompanying tape rewinds, the optimum tradeoff depends on the equipment.

Because his optimum distribution algorithm is so complex and requires a knowledge of the number of strings to be distributed, Shell also investigated several simpler ones. The best of the simple ones, the horizontal distribution algorithm, is 2 to 3% below optimum. It fills the tapes with non-null strings so that each tape has the proper number of strings whenever an exact

	Tape 1	Tape 2	Tape 3	Tape 4
Step 1	9	0	0	0
Step 2	0	2	3	4
Step 3	2	0	1	2
Step 4	1	1	0	1
Step 5	0	0	1	0

FIG. 15. The number of strings on each tape at each step of the polyphase sort in Figure 14.

number is reached. Suppose, for example, that three tapes have already been filled with 31 non-null strings, as shown in step 2 of Figure 16. To reach the next exact number, 57, eleven strings must be added to the tape now containing the most strings (13), nine strings to the tape now containing the second largest number of strings (11), and six strings to the tape now containing the fewest strings (7). The horizontal algorithm first adds strings to the tape requiring the most additional strings (11), until it and the next tape both require the same number, (9). It then adds strings alternately to both of these tapes until they require the same number as the third tape, (6). Strings are then added alternatively to all three tapes until the next exact number is reached. When the non-null strings are used up, the horizontal procedure is continued to the next exact number using null strings.

As presented here, the polyphase sort requires each tape to be rewound before it is read; however, when using tape drives that can be read backwards the rewind can be eliminated [37]. Since a string written out in ascending order will appear in descending order when read backwards, care must be taken to ensure that all strings to be merged at each step are in the same order. This is achieved if all strings are initially written out so that the strings on each tape are alternately in ascending and descending order.

As mentioned earlier, a merge sort forms a tree of strings. The root of the tree is the final string and each node in the tree represents a merge. The tree representing the merge formed by the polyphase sort in Figure 14 is shown in Figure 17.

The Oscillating Sort

The number of times each item is passed in and out of core during a tape sort is often taken as a measure of the sort efficiency. This

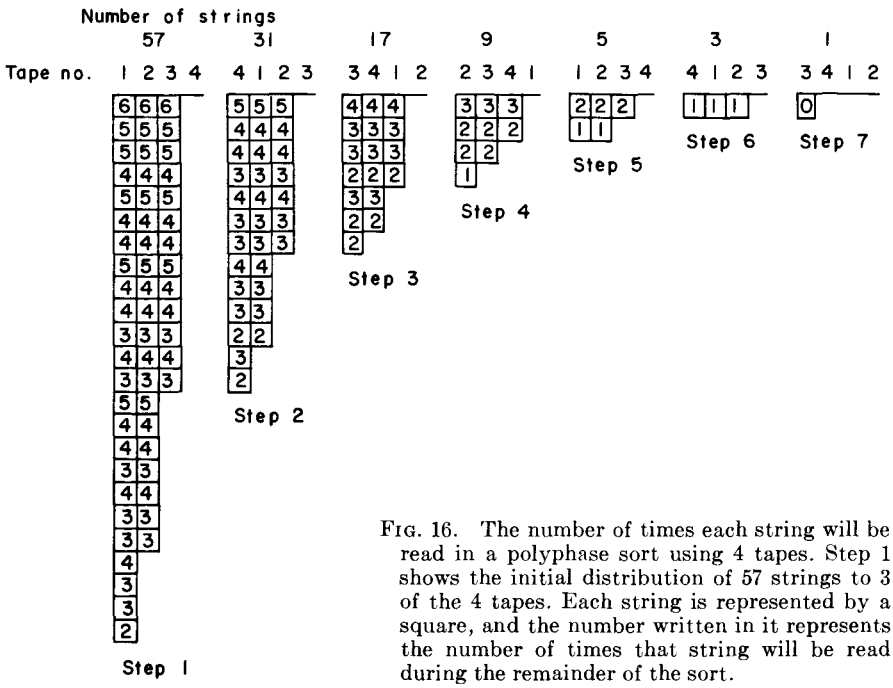

FIG. 16. The number of times each string will be read in a polyphase sort using 4 tapes. Step 1 shows the initial distribution of 57 strings to 3 of the 4 tapes. Each string is represented by a square, and the number written in it represents the number of times that string will be read during the remainder of the sort.

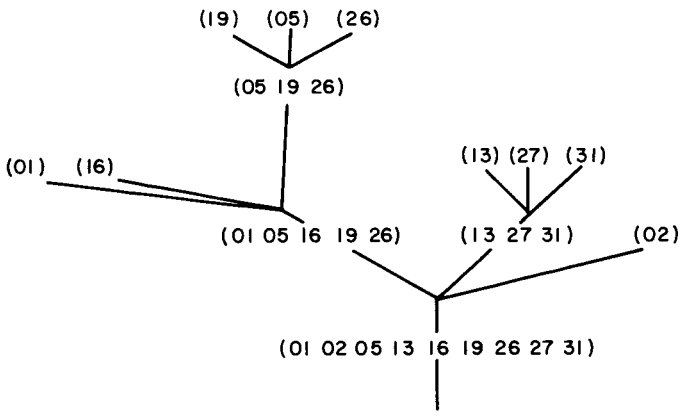

FIG. 17. The tree of string merges formed by the polyphase sort in Figure 14.

number is given roughly by the average depth of the merge tree of the sort. The depth of the tree can be reduced both by maximizing the order of the merge at each node and by ensuring that each merge involves strings of equal length. The polyphase sort maximizes the order of the merge by using all but one of the tapes for inputs to the merge at each step. However, as shown in Figure 17, it does not merge strings of equal length. On the other hand, the *oscillating* sort always merges strings of equal length but uses one fewer of the tapes for inputs than does the polyphase sort. It also requires tapes that can be read in both directions and quickly reversed. The minimum possible depth of the merge tree is roughly $\log n/\log k$, where k is the number of strings merged at each node and n is the total number of strings to be merged. Thus, as the number of tape units increases, there is a decrease in the possible value of using each additional unit for increasing the order of the merge at each node. As it turns out because of the uneven distribution, additional tapes have even less value in the polyphase merge [59].

Thus, if one has more than six tape units that can be reversed quickly and read in both directions, it is better to get the even distribution by using an oscillating sort [22, 92]. An example of an oscillating sort is shown in Figure 18. In step 1 of Figure 18 the nine unsorted numbers are shown on tape 1, and the other four tapes are empty. In step 2, three numbers are read from tape 1 and placed on tapes 3, 4, and 5. They are considered to be three sorted lists of length 1. Now, these three lists are merged in descending order to form one list (19 13 05) of length 3, which is placed on tape 2. In steps 4 and 5, this process is repeated to place a sorted list of length 3 on tape 3; it is repeated again in steps 6 and 7. Finally, the three lists of length 3 are merged in ascending order to form the answer list of length 9. If more than nine numbers were being sorted, we would go on to form three lists of length 9, then three lists of length 27, and so forth.

If one has tapes that can only be read in the forward direction, the oscillating sort cannot be used. Goetz [44] describes a version of this sort that does not require backward read; but it does require the ability to write on the front of a tape without destroying information further down that is to be read later, and it also requires that all initial strings be the same length.

Under special circumstances, the cascade merge [7] might also be useful for more than six tapes. However, with eight or more such tapes, an improvement over the polyphase sort can be obtained by using the simplest merge sort. For this sort, the tapes are divided into two sets as equal in number as possible. The initial strings are distributed evenly onto one set. All tapes are then rewound, and the strings are merged back onto the first set of tapes. This process continues until the merge is complete.

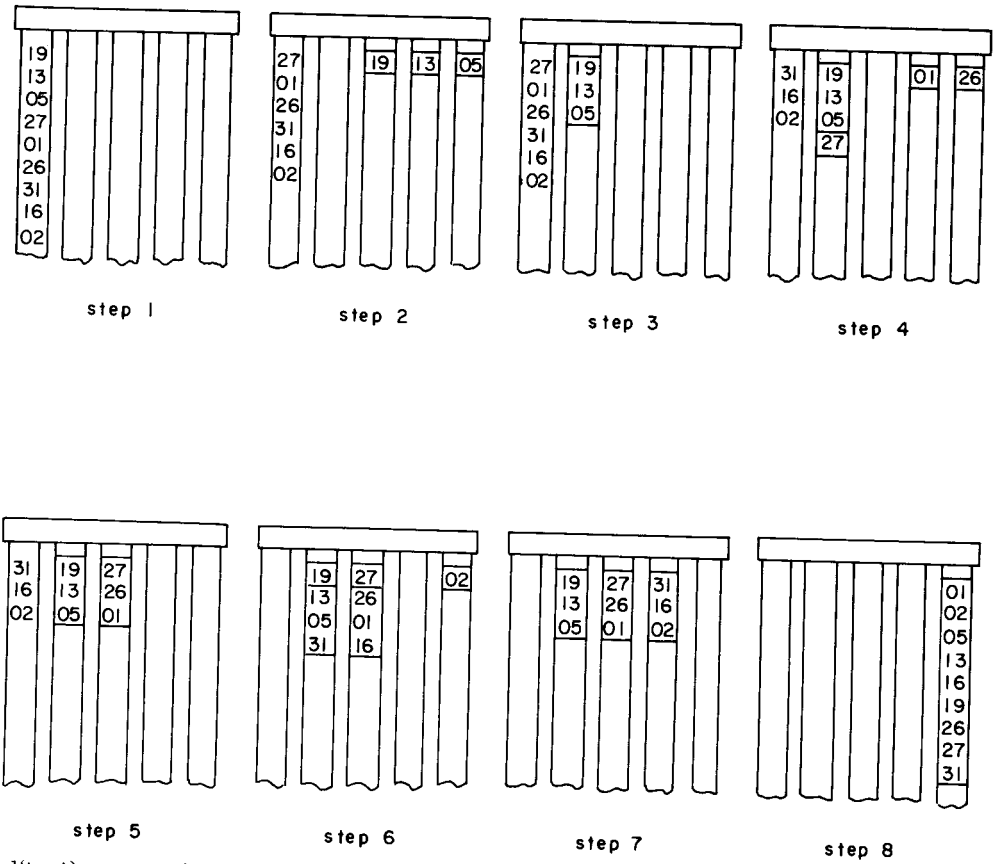

FIG. 18. Demonstration of an oscillating sort of 9 numbers using 5 tapes. The beginning of each tape is at the top.

Replacement Sorts

In the polyphase, oscillating, and simple merge sorts we have started off by merging lists of length 1. In practice, however, the first step in these sorts usually involves reading as many numbers as possible into the memory of the machine, sorting them, and placing them on one of the tapes; then reading in another batch, sorting those, and writing them out. Thus, in step 2 of Figure 14 or Figure 18 we would distribute these sorted lists onto the tapes as we did, but the lists would have length greater than 1.

It is desirable to make these initial lists as long as possible in order to reduce the number of merge steps required to complete the sort. If memory can hold L numbers, we could form sorted lists of length L by sorting in place. However, by using a type of replacement sort, we can form lists with an *average* length of $2L$ [35, 40, 60]. (If the strings are written in alternately ascending and descending order, the average length is only $1.5L$.) Such a replacement sort is shown in Figure 19. We assume that the memory can hold three numbers at a time ($L = 3$). We begin by reading in three numbers in step 2. The smallest of these, 09, is then output as the first number in the first sorted list. It is replaced in memory by the next input number, 02. Note that since 02 is smaller than the number it replaced, 09, it cannot go into the sorted list with 09, but must go into the next list. We therefore choose the smaller of 21 and 11 for the next output number; 11 is chosen, output, and replaced with 16 in step 4. Proceeding in this way, we reach step 8, at point all three numbers are too small, so the next sorted list is started. The sort winds up

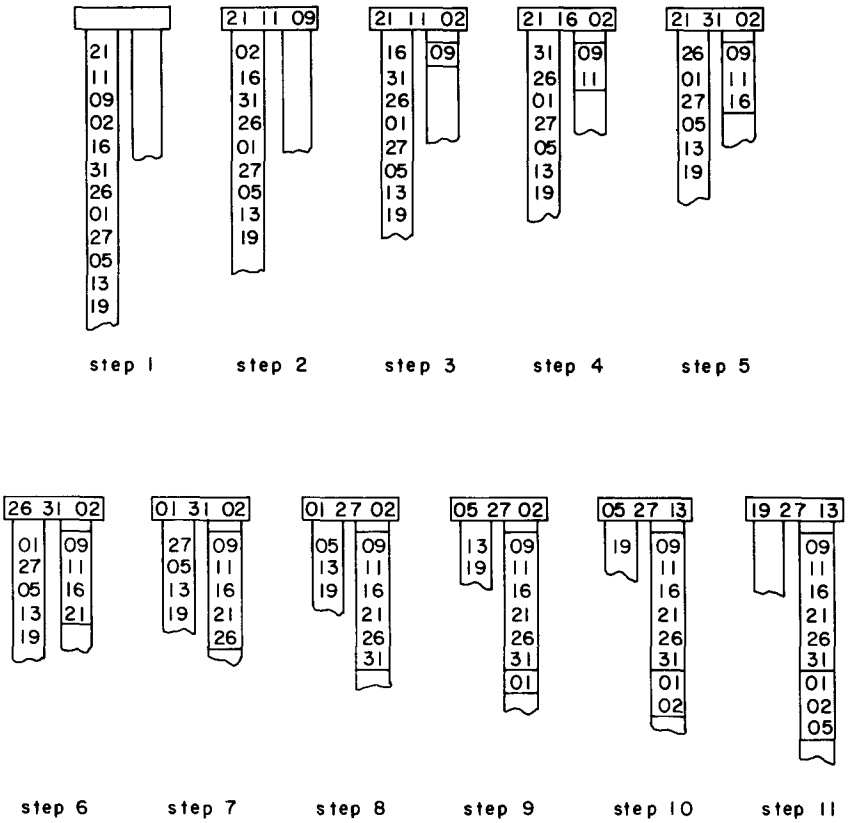

FIG. 19. Forming two sorted lists of 6 items each from an unsorted list of 12 items using a replacement sort. The beginning of each tape is at the top.

with two sorted lists of six numbers each, one following the other on the output tape.

In the tape sorts shown earlier, initial strings were distributed to several tapes. We can take advantage of this to produce strings with an average length even longer than $2L$. To do this, we proceed as before until no more items can be placed on the first string. We then start the second string on a different output tape. While the second string is being formed, additional items that can go onto the first string may be read into core.

In the replacement sort in Figure 19 it was necessary to scan through all of the numbers in memory at each step in order to determine which number should be output. Implementation of this scan in firmware is discussed by Barsamian [2]. As mentioned earlier, some of this scanning can be avoided by using a quadratic or tournament sort, but at the cost of extra storage. A tournament replacement sort would require extra storage for a number of keys (or possibly pointers to keys) equal to the number of records held in core at one time. Since each record usually contains data items in addition to its key, the fractional increase in storage for a tournament replacement sort may be small. Nevertheless, it is interesting to see how a tournament replacement sort can be done without extra storage [A33]. Figure 20 shows such a sort.

In step 1, $L = 7$ items have been read into memory. The sort works for any number of items, although $L = 2^k - 1$ items makes it a perfect binary tree. The purpose of steps 1–5 is to rearrange the items by means of exchanges so that the item in cell i is smaller than the items in cells $2i$ and $2i + 1$, for $1 \leqslant i \leqslant L/2$. To do this, we start in step 1 with the item in cell 3, which must be smaller

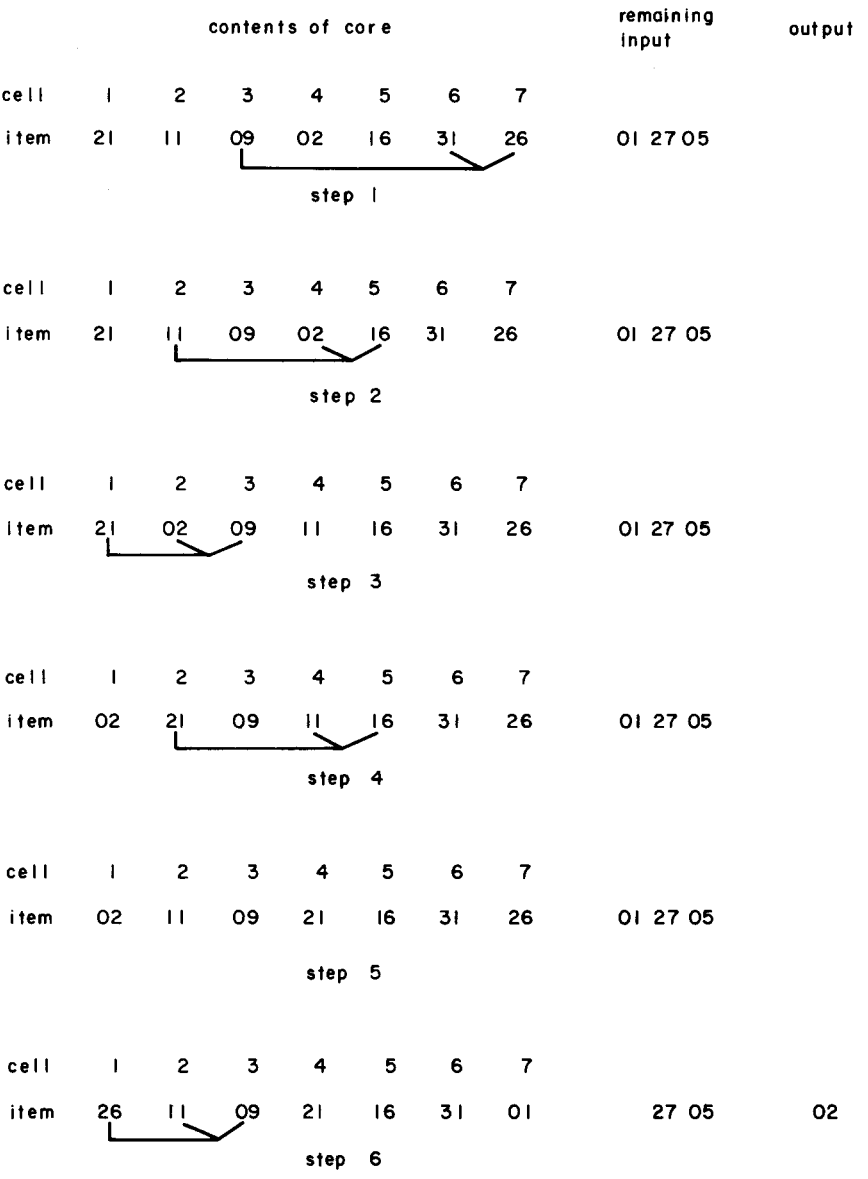

FIG. 20. A tournament replacement sort in place.

than the items in cells 6 and 7. Comparing 31, the item in cell 6, with 26, the item in cell 7, we find that 26 is the smaller, so we then compare 09, the item in cell 3, with 26. Since 09 is smaller, no exchanges are required. In step 2 we repeat this process for the items in cells 2, 4, and 5. This time it is necessary to make an exchange. In step 3 the process is again repeated for the items in cells 1, 2, and 3. Since, 21, the item in cell 1, is larger than 02, the item in cell 2, an exchange is required. Thus, in step 4 we are no longer certain that the item in cell 2 is smaller than those in cells 4 and 5; we must go back and make this comparison. It turns out that 11 and 21 must be exchanged. Note, however, that it is not necessary to recheck the items in cells 1 and 2 after this exchange. We are now guaranteed that the item in cell i is smaller than the items in cells $2i$ and

$2i + 1$. Therefore, in step 5 the item in cell 1 is the smallest of the L items, and it can be output. We replace it with 01 the next input item, but since 01 is smaller than 02 it cannot go out in the current string. It can be ruled out by putting the cell 7 item in cell 1, putting 01 in cell 7, and then considering only cells 1 through 6 during the rest of the current string. This is shown in step 6. There, we proceed as we did in step 3 to get the smallest item into cell 1. When the number in cell 1 is replaced with an input number that is larger, we go to step 3 directly.

With a slight change, the above sort becomes the minimal storage comparative sort known as Treesort 3 [A1, A14]. Instead of writing out the item in cell 1, it is exchanged with the item in cell 7. Cell 7 then contains the smallest item and is removed from consideration as above. The smallest item in cells 1 to 6 is then brought into cell 1 and exchanged with the item in cell 6. The items are sorted in seven such steps.

SORTING WITH DISKS

Very little has been written about disk sorts [18, 41, 45, 54]. Perhaps this is because disks became widespread after research on sorting methods lost the popularity it had had in the early sixties; or perhaps it is because current operating systems make it difficult to evaluate the importance of minimizing seek time, latency, or volume passed. Then, too, sorting is not as important when random-access devices are used; if records stored on a disk need not be accessed sequentially in sorted order, it may not be necessary to place them in sorted order. It may be sufficient to sort the keys to the records in core and then place these sorted keys on the disk as an index to the records.

Seek time is the aspect of disk sorting most often discussed since it is the most straightforward. To minimize seek time, more than one disk module (or modules with more than one head) should be used. When several modules are available, good results can be obtained using the same sorts described for magnetic tapes. For example, with two modules one can do a merge sort using one module for input and the other for output. In this case, seek time is reduced by interleaving the strings to be merged on alternate cylinders, rather than keeping all of each string on adjacent cylinders [22]. On the other hand, when a merge is done with only one module, seek time is minimized by keeping the strings on adjacent cylinders. If the seek time for each string access is assumed to be a known constant, the optimum order of the merge is easy to compute [9]. When using one module, any sorting procedure requires at least enough seek time to carry the items directly from their initial positions to their final positions. For an initial list in random order this carrying time is minimized by placing the final list in the area occupied by the input. Remember that in the bubble sort each item was always moved in the direction of its final destination; applying this bubble sort principle to disk can minimize carrying time. Starting at the top of the list to be sorted, we require that when the reading head is moved down, the value of the items in core memory are greater than those on the disk cylinder currently under the reading head. Similarly, the items with the lowest values are left in core when the reading head is moved up.

Item-at-a-time sorts are often used on disk in real-time data management applications. For these sorts, seek time is minimized if the cylinder to contain the item is reached on the first distribution.

SORTING RECORDS

Records are sorted on their keys. When sorting in core, it is common to sort the keys keeping only the address of its record with each key. Alternatively, one can sort a list of pointers to the keys [11]. Once the keys, or pointers to keys, are sorted, the sorted keys can be used as an index to the records, or the records can be copied into sorted order. Whether the keys should be sorted separately depends upon the length of the records and upon the sorting method. In a bubble sort the items are moved to many intermediate positions; Quicksort moves each item about $\log_2 n$ times; while item-at-

TABLE I. COMPARISON OF SAMPLE CORE SORTING ALGORITHMS
(There are n items. They are distributed k ways in address calculation.)

Parameters for comparison	*Bubble sort*	*Quicksort*	*Address calculation sort*	*Binary merge sort*	*Parallel merge sort*
Average number of item comparisons or distributions	$\sim n^2/4$ comparisons	$\sim 2n \log n$ comparisons	n distributions; $(n^2/2k)(1 - n/k)$ comparisons	$\approx \log (n/2)/\log 2$ comparisons	$\approx n (\log n/\log 2)^2$ comparisons
Maximum number of item comparisons or or distributions	$\sim n^2/2$ comparisons (items in reverse order)	$\sim n^2/8$ comparisons (median estimate always chooses smallest item)	n distributions; $\sim n^2/4$ comparisons (all items hit the same cell)	$\approx n \log (n/2)/\log 2$ comparisons	$\sim n (\log n/\log 2)^2$ comparisons
Minimum number of item comparisons or distributions	$\sim n$ comparisons (items already sorted)	$\sim n \log n/\log 2$ comparisons (median estimate always correct)	n distributions; no comparisons (no items hit the same cell)	$\approx (n/2) \log (n/2)/\log 2$ comparisons	$\sim n (\log n/\log 2)^2$ comparisons
Average number of data moves	At most $n - 1$ more comparisons than moves	$\approx n \log n$	Equal to distributions plus comparisons	$\approx n (\log n)/\log 2$	—
Average sort time	$\approx n^2$	$\approx n \log n$	$\approx n$ (assuming $k \approx n$ and good knowledge of distribution)	$\approx n \log n$	$\approx (\log n)^2$
Additional storage required	None	$\approx \log n$	k; a typical value would be $1.25n$	n	None
Special hardware required	None	None	None	None	$\sim n (\log_2 n)^2$ two-item sorters
Does sort depend on the distribution of element values?	No	Knowledge of distribution can improve median estimate.	Yes. Note best and worst cases above.	No	No
Are most memory accesses sequential?	n random accesses each followed by an average of $n/4$ sequential accesses	No	Distributions, no; comparisons, yes	$\approx 2n$ random; $\sim n \log n$ sequential	

a-time sorts sometimes place the item directly in its final location. In the case of disk sorts, the random-access time is so long that when the records are to be copied into sorted order only the very long ones should have their keys sorted separately [22]. When fetching records, corresponding to the sorted keys, on disk, it is advantageous to fetch several at once. They can then be fetched in an order that minimizes seek time. It is rarely advantageous to sort keys separately when using tapes.

Sometimes a *major* key is used for a primary sort, and then a *minor* key is used to sort all those items having the same major key. This can be accomplished with some sorting schemes, such as the bubble sort, by first sorting the entire list on the minor key, then sorting the entire list on the major key, taking care in both cases not to interchange records having the same key. This cannot be done with the Shell sort or Quicksort; these methods will throw the minor key out of sort. However, one can use any sorting method by concatenating the major and minor keys into one long key.

SEARCHING

The principal reason for sorting is to facilitate search by placing an ordering on the space to be searched. The search time for one item in an unordered, random-access file

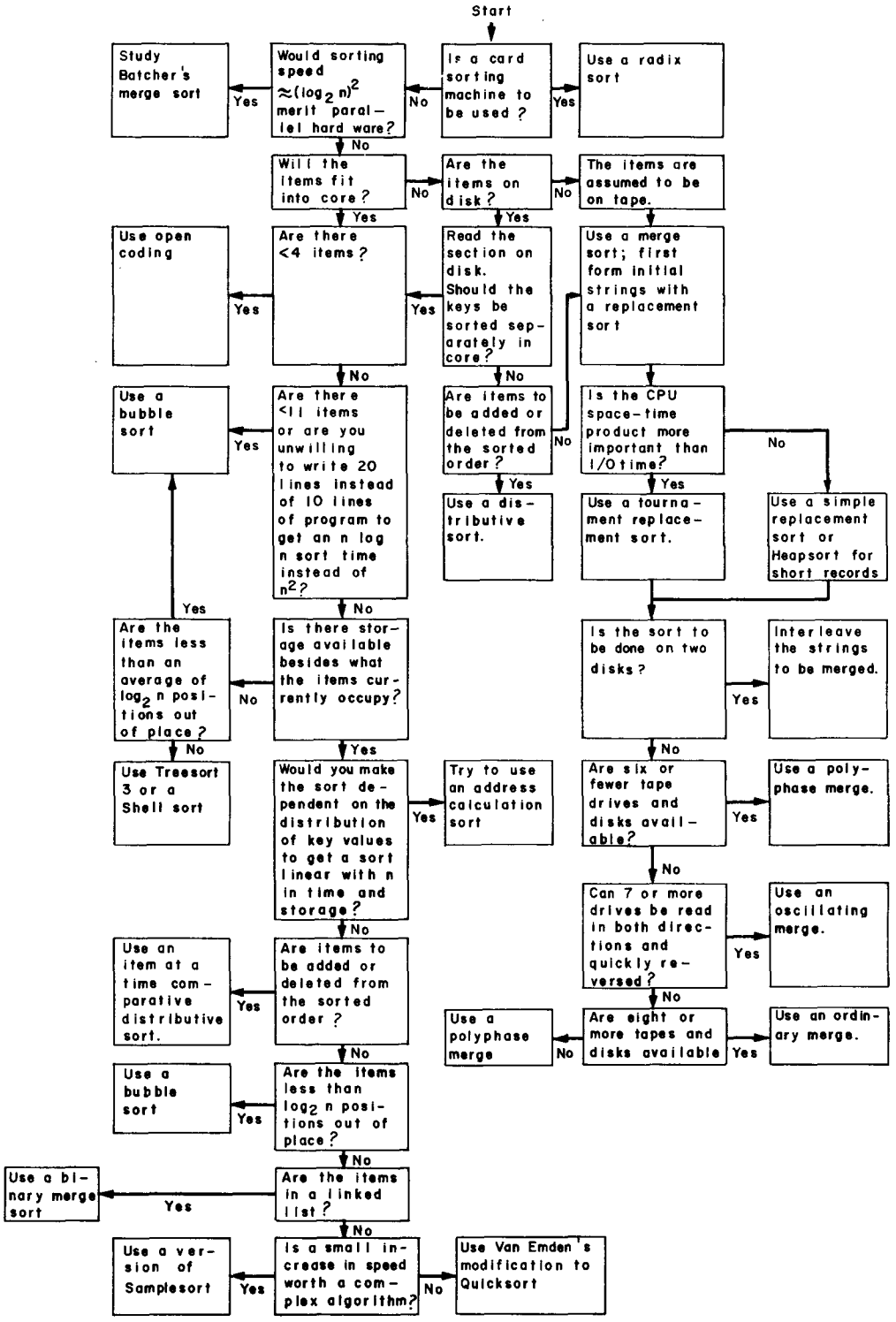

FIG. 21. Suggestions on the use of the various sorting techniques.

is proportional to n, whereas in an ordered file it is proportional to $\log n$. This is a significant reduction, but if an address calculation sort that produces at least a random distribution can be found, then the search time does not depend directly on n at all. It then depends only on the fullness ratio, n/k. To obtain this good result, one needs a knowledge of the distribution of the values of the keys. Such a search time can also be obtained independent of the key distribution by using scatter storage techniques [76]. However, since these techniques require random-access memory and do not allow range searches, neither method is always best.

COMPARISON OF SORTING METHODS

Table I presents some facts about sample sorting algorithms. (The symbol $\approx$ is used to indicate behavior for large values of n.) A study of this table provides a rough idea of what to expect from the various algorithms. However, one should be careful not to draw final conclusions from this table, but should go deeper into more detail. For example, although the bubble sort takes time proportional to n^2, the proportionality constant is usually smaller than for some other sorts. Also, the definition of "random access" is not clear: an access may be nonsequential, yet refer to a recently accessed location in a predictable way. Using the techniques described in the paper one can construct special-purpose algorithms tuned to the number, order, and distribution of the keys, the supplies of sequential and random-access storage, and the intended use of the sorted items.

Some ideas about which methods might be relevant to a given situation are shown in Figure 21.

ACKNOWLEDGMENTS

The author thanks the referee for his excellent criticism.

BIBLIOGRAPHY

An attempt has been made to make the bibliography complete. Published algorithms are listed separately at the end. Many of the papers in the bibliography have not been referenced in this survey because they present methods that have been better described elsewhere, or that are dominated in most applications by some other technique. I recommend Knuth's book [62] for a thorough, up-to-date treatment of the entire topic of sorting and searching.

1. APPLEBAUM, F. H. "Variable word sorting in the RCA 501 System." In *Proc. ACM 14th Natl. Conf.*, 1959, paper #44.
2. BARSAMIAN, H. "Firmware sort processor with LSI components." In *Proc. AFIPS 1970 SJCC*, Vol. 36, AFIPS Press, Montvale, N.J., 183–190.
3. BATCHER, K. E. "Sorting networks and their applications." In *Proc. AFIPS 1968 SJCC*, Vol. 32, AFIPS Press, Montvale, N.J., 307–314.
4. BAYES, A. "A generalized partial pass block sort." *Comm. ACM* **11**, 7 (July 1968), 491–493.
5. BELL, D. A. "The principles of sorting." *Computer J.* **1**, 1 (1958), 71–77.
6. BENDER, B. K.; AND A. J. GOLDMAN. "Analytic comparison of suggested configurations for automatic mail sorting equipment." *J. Research NBS* **63B**, 4 (Oct.–Dec. 1959), 83–104.
7. BETZ, B. K.; AND W. C. CARTER. "New merge sorting techniques." In *Proc. ACM 14th Natl. Conf.*, 1959, paper #14.
8. BEUS, H. L. "The use of information in sorting." *J. ACM* **17**, 3 (July 1970), 482–495.
9. BLACK, N. A. "Optimum merging from mass storage." *Comm. ACM* **13**, 12 (Dec. 1970), 745–749.
10. BOSE, R. C.; AND R. J. NELSON. "A sorting problem." *J. ACM* **9**, 2 (April 1962), 282–296.
11. BRAWN, B. S.; F. G. GUSTAVSON; AND E. S. MANKIN. "Sorting in a paging environment." *Comm. ACM* **13**, 8 (Aug. 1970), 483–494.
12. BURGE, W. H. "Sorting, trees, and measures of order." *Information & Control* 1 (1958), 181–197.
13. CARTER, W. C. "Mathematical analysis of merge-sorting techniques." In *Proc. IFIP Cong. 1962*, North-Holland Publ. Co., Amsterdam, 1963, 62–66.
14. COOKE, W. S. "A tape file merge pattern generator." *Comm. ACM* **6**, 5 (May 1963), 227–230.
15. DAVIES, D. W. "Sorting of data on an electronic computer." *Proc. Inst. Electronic Engineers* (London) **103**, Part B supplement (1956), 87–93.
16. DE BEAUCLAIR, W. "Das sortieren von Magnetband-Daten in einfachen Buchungsanlagen." *Elektr. Rechen.* 2 (April 1961), 75–82. (German)
17. DEFIORE, C. E. "Fast sorting." *Datamation* **16**, 8 (Aug. 1, 1970), 47–51.
18. FALKIN, J.; AND S. SAVASTANO, JR. "Sorting with large volume, random access, drum storage." *Comm. ACM* **6**, 5 (May 1963) 240–244.

19. FEERST, S.; AND F. SHERWOOD. "The effect of simultaneity on sorting operations." In *Proc. ACM 14th Natl. Conf.*, 1959, paper #42.
20. FLORES, I. "Computer time for address calculation sorting." *J. ACM* **7**, 4 (Oct. 1960), 389–409.
21. FLORES, I. "Analysis of internal computer sorting." *J. ACM* **8**, 1 (Jan. 1961), 41–80.
22. FLORES, I. *Computer sorting.* Prentice Hall, Inc., Englewood Cliffs, N.J., 1969.
23. FLOYD, R. W.; AND D. E. KNUTH. "Improved constructions for the Bose-Nelson sorting problem." *Notices Amer. Math. Society* **14** (1967), 283.
24. FLOYD, R. W.; AND D. E. KNUTH. "The Bose-Nelson sorting problem." Stanford Computer Science Memo., STAN-CS-70-177, Nov. 1970.
25. FORD, L. R.; AND S. M. JOHNSON. "A tournament problem." *Amer. Math. Monthly* **66** (May 1959), 387–389.
26. FOSTER, C. C. "Information storage and retrieval using AVL trees." In *Proc. ACM 20th Natl. Conf.*, 1965, 192–205.
27. FOSTER, C. C. "Sorting almost ordered arrays." *Computer J.* **11**, 2 (Aug. 1968), 134–137.
28. FRANK, R. M.; AND R. B. LAZARUS. "A high speed sorting procedure." *Comm. ACM* **3**, 1 (Jan. 1960), 20–22.
29. FRAZER, W. D.; AND A. C. MCKELLAR. "SAMPLESORT: a sampling approach to minimal storage tree sorting." *J. ACM* **17**, 3 (July 1970), 496–507.
30. FREDKIN, E. "Trie memory." *Comm. ACM* **3**, 9 (Sept. 1960), 490–499.
31. FRENCH, N. C. "Computer planned collates." *Comm. ACM* **6**, 5 (May 1963), 225–227.
32. FRIEND, E. H. "Sorting on electronic computer systems." *J. ACM* **3**, 3 (July 1956), 134–168.
33. GALE, D.; AND R. KARP. "A phenomenon in the theory of sorting." In *IEEE* Conf. Record of the *11th* Annual Symposium on Switching and Automata Theory, 1970, 51–59.
34. GALLAGER, R. C. *Information theory and reliable communication.* John Wiley & Sons, New York, 1968.
35. GASSNER, BETTY JANE. "Sorting by replacement selecting." *Comm. ACM* **10**, 2 (Feb. 1967), 89–93.
36. GILSTAD, R. L. "Polyphase merge sorting—an advanced technique." In *Proc. EJCC*, Vol. 18, Dec. 1960, Spartan Books, New York, 143–148.
37. GILSTAD, R. L. "Read-backward polyphase sorting." *Comm. ACM* **6**, 5 (May 1963), 220–223.
38. GLICKSMAN, S. "Concerning the merging of equal length tape files." *J. ACM* **12**, 2 (April 1965), 254–258.
39. GLORE, J. B. "Sorting non-redundant files—techniques used in the FACT compiler." *Comm. ACM* **6**, 5 (May 1963), 231-240.
40. GOETZ, M. A. "Internal and tape sorting using the replacement selection technique." *Comm. ACM* **6**, 5 (May 1963), 201–206.
41. GOETZ, M. A. "Organization and structure of data on disk file memory systems for efficient sorting and other data processing programs." *Comm. ACM* **6**, 5 (May 1963), 245–248.
42. GOETZ, M. A.; AND G. S. TOTH. "A comparison between the polyphase and oscillating sort techniques." *Comm. ACM* **6**, 5 (May 1963), 223–225.
43. GOETZ, M. A. "Design and characteristics of a variable-length record sort using new fixed length record sorting techniques." *Comm. ACM* **6**, 5 (May 1963), 264–267.
44. GOETZ, M. A. "Some improvements in the technology of string merging and internal sorting." In *Proc. AFIPS 1964 SJCC*, Vol. 25, Spartan Books, New York, 599–607.
45. GOETZ, M. A. "A disk file sorting problem." In *Proc. 3rd Annual Princeton Conf. on Info. Science and Systems*, Dept. Electrical Engineering, Princeton Univ., 1969, 281–285.
46. GOTLIEB, C. C. "Sorting on computers." *Comm. ACM* **6**, 5 (May 1963), 194–201.
47. HALL, M. H. "A method of comparing the time requirements of sorting methods." *Comm. ACM* **6**, 5 (May 1963), 206–213.
48. HIBBARD, THOMAS N. "Some combinatorial properties of certain trees with application to searching and sorting." *J. ACM* **9**, 1 (Jan. 1962), 13–28.
49. HIBBARD, THOMAS N. "An empirical study of minimal storage sorting." *Comm. ACM* **6**, 5 (May 1963), 206–213.
50. HIBBARD, THOMAS N. "A simple sorting algorithm." *J. ACM* **10**, 2 (April 1963,) 142–150.
51. HILDEBRANDT, P.; AND H. ISBITZ. "Radix exchange—an internal sorting method for digital computers." *J. ACM* **6**, 2 (April 1959), 156–163.
52. HOARE, C. A. R. "Quicksort." *Computer J.* **5**, 1 (1962), 10–15.
53. HOSKEN, J. C. "Evaluation of sorting methods." In *Proc. EJCC*, Vol. 8, Spartan Books, New York, Nov. 1955, 39–55.
54. HUBBARD, G. U. "Some characteristics of sorting in computing systems using random access storage devices." *Comm. ACM* **6**, 5 (May 1963), 248–255.
55. HWANG, F. K.; AND S. LIN. "An analysis of Ford and Johnson's sorting algorithm." In *Proc. 3rd Annual Princeton Conf. on Info. Science and Systems*, Dept. Electrical Engineering, Princeton Univ., 1969, 292–296.
56. ISAAC, E. J.; AND R. C. SINGLETON. "Sorting by address calculation." *J. ACM* **3**, 3 (July 1956), 169–174.
57. JONES, R. W. "Sorting by merging." *Computer J.* **2**, 2 (July 1959), 95–96.
58. KISLITSYN, S. S. "On finding the kth element of an ordered population by means of pair-wise matching." *Sibirsky Matem. Zh.* **5**, 3 (1964), 557–564.
59. KNUTH, D. E.; AND M. A. GOETZ. Letter to the Editor. *Comm. ACM* **6**, 10 (Oct. 1963), 585–597.

60. KNUTH, D. E. "Length of strings for a merge sort." *Comm. ACM* **6**, 11 (Nov. 1963), 685–687.

61. KNUTH, E. D. "Optimum binary search trees." Stanford Computer Science Memo. CS 149, 1970.

62. KNUTH, D. E. *The art of computer programming*, Vol. 3. Addison Wesley Publ. Co., Reading, Mass. (To be published).

63. KRONMAL, R.; AND M. TARTER. "Cumulative polygon address calculation sorting." In *Proc. ACM 20th Natl. Conf.*, 1965, 376–385.

64. LAUTZ, W. "Sortierverfahren für technische Dual-Computer: Part 1." *Elektronische Datenverarbeitung*, 2 (April 1963), 69–81. (German)

65. LAUTZ, W. Sortierverfahren für technische Dual-Computer: Part 2." *Elektronische. Datenverarbeitung*, 3 (May 1963), 133–141 (German)

66. LEHMER, D. H. "Sorting cards with respect to a modulus." *J. ACM* **4**, 1 (Jan. 1957), 41–46.

67. LEVY, S. Y.; AND M. C. PAULL. "An algebra with application to sorting algorithms." In *Proc. 3rd Annual Princeton Conf. on Info. Sciences and Systems*, Dept. Electrical Engineering, Princeton Univ., 1969, 286–291.

68. LIU, D. "Construction of sorting plans." Presented at *Internatl. Symposium on the Theory of Machines and Computations*, Haifa, Israel, Aug. 1971.

69. LYNCH, W. C. "More combinatorial properties of certain trees." *Computer J.* **7**, 4 (Jan. 1965), 299–302.

70. MACLAREN, M. D. "Internal sorting by radix plus shifting." *J. ACM* **13**, 3 (July 1966), 404–411.

71. MALCOLM, W. D., JR. "String distribution for the polyphase sort." *Comm. ACM* **6**, 5 (May 1963), 217–220.

72. MANKER, H. H. "Multiphase sorting." *Comm. ACM* **6**, 5 (May 1963), 214–217.

73. MARTIN, W. A.; AND D. N. NESS. "Optimizing binary trees grown with a sorting algorithm." *Comm. ACM* (to appear).

74. MENDOSA, A. G. A dispersion pass algorithm for the polyphase merge." *Comm. ACM* **5**, 10 (Oct. 1962), 502–504.

75. MOELLER, D. "MR-1401 a generalized sort program for the card-RAMAC 1401." In *IBM Systems Engineering Conf.*, New York, 1961.

76. MORRIS, R. "Scatter storage techniques." *Comm. ACM* **11**, 1 (Jan. 1968), 38–44.

77. MORRIS, R. "Some theorems on sorting." *SIAM J. Appl. Math.* **17**, 1 (Jan. 1969), 1–6.

78. NAGLER, H. "Amphisbaenic sorting." *J. ACM* **6**, 4 (Oct. 1959), 459–468.

79. NAGLER, H. "An estimation of the relative efficiency of two internal sorting methods." *Comm. ACM* **3**, 11 (Nov. 1960), 618–620.

80. NICHOLS, J. H.; AND A. TIEDRICH. "A multivariant generalized sort program employing auxiliary drum storage." In *Proc. ACM 17th Natl. Conf.*, 1962.

81. O'CONNOR, D. G.; AND R. J. NELSON. "Sorting system with n-line sorting switch." US Patent 3029413, issued April 10, 1962.

82. PAPERNOV, A. A.; AND G. V. STASEVICH. "A method of information sorting in computer memories." *Problems of Information Transmission* **1**, 3 (1967), 63–75.

83. PATERSON, J. B. "The COBOL sort verb." *Comm. ACM* **6**, 5 (May 1963), 255–258.

84. PICARD, C. "Several recent ideas on the sorting problem." *Rev. Franc. Trait. Inf.* **9**, 1 (1966), 41–46. (French)

85. RADKE, C. E. "Merge-sort analysis by matrix techniques." *IBM Systems J.* **5**, 4 (1966), 226–247.

86. REYNOLDS, S. W. "A generalized polyphase merge algorithm." *Comm. ACM* **4**, 8 (Aug. 1961), 347–349.

87. REYNOLDS, S. W. "Addendum to 'A Generalized Polyphase Merge Algorithm.' " *Comm ACM* **4**, 11 (Nov. 1961), 495.

88. SCHICK, T. "Disk file sorting." *Comm. ACM* **6**, 6 (June 1963), 330–331, 339.

89. SEEBER, R. R. "Associative self-sorting memory." In *Proc. EJCC*, Vol. 18, 1960, Spartan Books, N.Y. 179–187.

90. SHELL, D. L. "A high speed sorting procedure." *Comm. ACM* **2**, 7 (July 1959), 30–32.

91. SHELL, D. L. "Optimizing the polyphase sort." *Comm. ACM* **14**, 11 (Nov. 1971), 713–719.

92. SOBEL, S. "Oscillating sort—a new sort merging technique." *J. ACM* **9**, 3 (July 1962), 372–374.

93. SUSSENGUTH, E. H., JR. "Use of tree structures for processing files." *Comm. ACM* **6**, 5 (May 1963), 272–279.

94. TARTER, M. E.; AND R. A. KRONMAL. "Non-uniform key distribution and address calculation sorting." In *Proc. ACM 21st Natl. Conf.*, 1966, MDI Publns., Wayne, Pa., 331–337.

95. VAN EMDEN, M. H. "Increasing the efficiency of quicksort." *Comm. ACM* **13**, 9 (Sept. 1970), 563–567.

96. WAKS, DAVID J. "Conversion, reconversion and comparison techniques in variable-length sorting." *Comm. ACM* **6**, 5 (May 1963), 267–271.

97. WIERZHOWSKI, J. "Sorting by means of random access store." *Algorytmy* **2**, 4 (1965), 59–68.

98. WINDLY, P. F. "The influence of storage access time on merging processes in a computer." *Computer J.* **2**, 2 (July 1959), 49–53.

99. WINDLY, P. F. "Trees, forests, and rearranging." *Computer J.* **3**, 2 (July 1960), 84–88.

100. WOODRUM, L. J. "Internal sorting with minimal comparing." *IBM Systems J.* **8**, 3 (1969), 189–203.

Published Algorithms

A1. ABRAMS, P. S. "Certification of algorithm 245: Treesort 3." *Comm. ACM* **8**, 7 (July 1965), 445.

A2. BATTY, M. A. "Certification of algorithm 201: Shellsort." *Comm. ACM* **7**, 6 (June 1964), 349.

A3. BLAIR, C. R. "Certification of algorithm

207: Stringsort." *Comm. ACM* **7,** 10 (Oct. 1964), 585.

A4. BLAIR, C. R. "Certification of Algorithm 271: Quickersort." *Comm. ACM* **9,** 5 (May 1966), 354.

A5. BOOTHROYD, J. "Algorithm 201: Shellsort." *Comm. ACM* **8,** 6 (Aug. 1963), 445.

A6. BOOTHROYD, J. "Algorithm 207: Stringsort." *Comm. ACM* **6,** 10 (Oct. 1963), 615.

A7. BOOTHROYD, J. "Algorithm 25. Sort a section of the elements of an array by determining the rank of each element." *Computer J.* **10,** 3 (Nov. 1967), 308–309.

A8. BOOTHROYD, J. "Algorithm 26. Order the subscripts of an array section according to the magnitudes of the elements." *Computer J.* **10,** 3 (Nov. 1967), 309–310.

A9. BOOTHROYD, J. "Algorithm 27. Rearrange the elements of an array section according to a permutation of the subscripts." *Computer J.* **10,** 3 (Nov. 1967), 310.

A10. CHANDLER, J. P.; AND W. C. HARRISON. "Remark on algorithm 201: Shellsort." *Comm. ACM* **13,** 6 (June 1970), 373–374.

A11. FEURZEIG, W. "Algorithm 23: Mathsort." *Comm. ACM* **3,** 11 (Nov. 1960), 601.

A12. FLORES, I. "Algorithm 76: sorting procedures." *Comm. ACM* **5,** 1 (Jan. 1962), 48–50.

A13. FLOYD, R. W. "Algorithm 113: Treesort." *Comm. ACM* **5,** 8 (Aug. 1962), 434.

A14. FLOYD, R. W. "Algorithm 245: Treesort 3." *Comm. ACM* **7,** 12 (Dec. 1964), 701.

A15. GRIFFIN, R.; AND K. A. REDISH. "Remark on algorithm 347: an efficient algorithm for sorting with minimal storage." *Comm. ACM* **13,** 1 (Jan. 1970), 54.

A16. HILLMORE, J. S. "Certification of algorithms 63, 64, 65: Partition, Quicksort, Find." *Comm. ACM* **5,** 8 (Aug. 1962), 439.

A17. HOARE, C. A. R. "Algorithms 63: Partition, and 64: Quicksort." *Comm. ACM* **4,** 7 (July 1961), 321.

A18. JUELICH, O. C. "Remark on algorithm 175: Shuttlesort." *Comm. ACM* **6,** 12 (Dec. 1963), 739.

A19. JUELICH, O. C. "Remark on algorithm 175: Shuttlesort." *Comm. ACM* **7,** 5 (May 1964), 296.

A20. KAUPE, A. F. "Algorithm 143: Treesort 1." *Comm. ACM* **5,** 12 (Dec. 1962), 604.

A21. KAUPE, A. F. "Algorithm 144: Treesort 2." *Comm. ACM* **5,** 12 (Dec. 1962), 604.

A22. LONDON, R. L. "Certification of algorithm 245: Treesort 3: proof of algorithms—a new kind of certification." *Comm. ACM* **13,** 6 (June 1970), 371–373.

A23. PETO, R. "Remark on algorithm 347: an efficient algorithm for sorting with minimal storage." *Comm. ACM* **13,** 10 (Oct. 1970), 624.

A24. RANDELL, B. "Remark on algorithm 76: sorting procedures." *Comm. ACM* **5,** 6 (June 1962), 348.

A25. RANDELL, B.; AND L. J. RUSSELL. "Certification of algorithms 63, 64, and 65: Partition, Quicksort, and Find." *Comm. ACM* **6,** 8 (Aug. 1963), 446.

A26. RANSHAW, R. W. "Certification of algorithm 23: Mathsort." *Comm. ACM* **4,** 5 (May 1961), 238.

A27. SCHUBERT, G. R. "Certification of algorithm 175: Shuttlesort." *Comm. ACM* **6,** 10 (Oct. 1963), 619.

A28. SCOWAN, R. S. "Algorithm 271: Quickersort." *Comm. ACM* **8,** 11 (Nov. 1965), 669–670.

A29. SCOWEN, R. S. "Notes on algorithms 25, 26." *Computer J.* **12,** 4 (Nov. 1969), 408–409.

A30. SHAW, C. J.; AND T. N. TRIMBLE. "Algorithm 175: Shuttlesort." *Comm. ACM* **6,** 6 (June 1963), 312–313.

A31. SINGLETON, R. C. "Algorithm 347: an efficient algorithm for sorting with minimal storage." *Comm. ACM* **12,** 3 (March 1969), 185–187.

A32. VAN EMDEN, M. H. "Algorithm 402: increasing the efficiency of Quicksort." *Comm. ACM* **13,** 11 (Nov. 1970), 693.

A33. WILLIAMS, J. W. J. "Algorithm 232: Heapsort." *Comm. ACM* **7,** 6 (June 1964), 347–348.

A34. WOODALL, A. D. "Algorithm 43: a listed radix sort." *Computer J.* **12,** 4 (Nov. 1969), 406.

A35. WOODALL, A. D. "Algorithm 45: an internal sorting procedure using a two-way merge." *Computer J.* **13,** 1 (Feb. 1970), 110–111.

A36. WOODALL, A. D. "Note on algorithms 25, 26." *Computer J.* **13,** 3 (Aug. 1970), 326.

A37. WOODALL, A. D. "Algorithm 55: an internal merge sort giving ranks of items." *Computer J.* **13,** 4 (Nov. 1970), 424–425.