Best Sorting Algorithm for Nearly Sorted Lists

Computer Science Department

Do Jin Kim

26 July 1978

I. INTRODUCTION

Many different sorting algorithms have been developed [1], and no one sorting algorithm is best for every case; in other words, each algorithm has its own advantages and disadvantages depending upon the environment in which it is applied as well as the characteristics of the list to be sorted.

When programmer effort is not a consideration, Martin [2] pointed out that the choice of method depends on:

(a) the length of the list to be sorted;

(b) the relation between the length of the list and the number of cells in the main memory of the machine used for sorting;

(c) the number and the size of any disk/tape used in sorting;

(d) the extent to which the elements are already in sorted order; and

(e) the distribution of the values of the elements.

Usually factor (d) above is neglected as the efficiency of a sorting algorithm is usually based on its performance on uniformly distributed random lists. However, the sortedness of the original list definitely affects the efficienty of some sorting algorithms.

The purpose of this paper is to explore (d) above in an attempt to discover the "best" sorting algorithm for "nearly" sorted lists, which are encountered quite often in real practice.

"Best" sorting algorithm can be interpreted in different ways depending upon what is considered to be most important. Chapter II discusses several possible interpretations, and gives the definition of "best" used in this paper.

As for "nearly" sorted lists, there is no commonly accepted measure for the orderedness of an input list and a simple, easily defined measure for "nearly" sorted list is elusive. Furthermore, it should be noted that the performance of a sorting algorithm is continuous on the ordering of the input list. In other words, sorting algorithms sort the unordered lists almost the same way (no. of steps) regardless of the small changes on input lists. This may explain the absence of the measures for the sortedness and the difficulty in defining the sortedness.

Much of the effort in this paper has been devoted to defining and proposing measures for the sortedness. In Chapter III, two measures are carefully developed and precisely defined.

We also define the "nearly" sorted region by those two measures.

In Chapter IV, several sorting algorithms are compared on "nearly" sorted lists. The sorting algorithms compared in this paper are Straight Insertion Sort, Tree Sort, Quicker Sort, Straight Merge Sort, Shell Sort, and Heap Sort. Descriptions for each algorithm are given in Appendix A.

We chose these sorting algorithms because they either take into account the sortedness of the input list or are known to be fairly fast on random input lists.

The names of sorting algorithms are not consistent between some books or papers. This paper will use the names given in Knuth [1]. For example, Straight Insertion Sort in Knuth is called the Bubble Sort in some papers [2, 7].

Straight Insertion Sort and Quicker Sort performed best on nearly sorted lists. The Straight Insertion Sort performed best on very nearly sorted or small lists and the Quicker Sort performed best on the remaining cases of nearly sorted lists.

In addition, a New Sorting Algorithm which is a combination of Quicker Sort and Merge Sort, is developed especially for nearly sorted list and is shown to perform as well as the Straight Insertion Sort and Quicker Sort. As a consequence, New Sorting Algorithm is the best sorting algorithm on the nearly sorted list.

Chapter V summarizes the paper and suggests several possible extensions of the work.

II. BEST SORTING ALGORITHM

The commonly used measures for the "best" sorting algorithm are Storage Requirements and Average Number of Comparisons [8]. Computer execution time is also used in some papers [3].

In this paper, we will not consider the storage requirement, so the "best" sorting algorithm is the "fastest" algorithm.

Computer execution time seems to be the simplest and most reasonable measure for comparing the sorting algorithms. However, computer execution time depends upon the programming technique, programming languages used and the computer system environment.

Furthermore, it is practically difficult to obtain very accurate computer execution times, and execution times cannot be compared unless the size of input is very large.

For example, CYBER system gives CPU time accurate to two decimal digits and most sorting algorithm can sort the list with 50 elements in 0.01 seconds.

The number of comparisons needed to sort an unordered list is called the "sort effort" [8] and generally provides a good measure. However, when a list is sorted, exchanges between elements in the list take longer than the comparisons on typical computer systems, and some sorting algorithms can sort the unordered list with extensive exchanges but with a small number of comparisons and vice versa.

Therefore, the number of exchanges should be considered along with the number of comparisons, when computing the sorting effort. In this paper, each sorting algorithm is compared by the number of comparisons, moves and exchanges between the elements in the list when it is sorted.

Any effort spent in bookkeeping, such as keeping track of the pointers or doing simple arithmetics involved in the sorting process will not be considered here.

Although this measure is not perfect, it seems to be the best compromise between simplicity and accuracy.

Furthermore, there exists no one absolutely best measure in comparing the sorting algorithms, but the results can be useful despite what appears to be overly simplified computational models [4].

We also need to assign a weight to exchange, move and comparison. Due to potential complexity, it is not possible to obtain an exact ratio, and there can be many possible ratios depending upon the computer system.

Assuming a typical computer system, we adpoted the ratio of (2:2:1). In other words, two comparisons, two moves and one exchange are equivalent to one another. Then we can get a weighted value as follows:

Weighted Value

= (no. of comparisons) x 2 + (no. of moves) x 2 + (no. of exchanges)

Thus, a "best" sorting algorithm is the one whose weighted value is smallest.

III. NEARLY SORTED LIST

A. Intuitive Definition and Examples.

Intuitively, the number of elements which are not in order and how far the unsorted elements are out of order are good measures for the unsortedness of a list.

A list which has a small number of unsorted elements and whose unsorted elements are not too far from their proper positions would be called "nearly" sorted.

In this section, we will explore the meaning of the terms used in our intuitive definition of nearly sorted list.

First, the number of sorted (or unsorted) elements should be counted by the relative order in the list rather than by their absolute positions.

In the list (2, 3, 4, 5, 6, 1, 7), for instance, only one element (7) is in its proper sorted position, however, we will consider only one element (1) is out of relative order and the other six elements (2, 3, 4, 5, 6, 7) are in order. From now on, "order" denotes the "relative order" unless otherwise specified.

Which elements are not in order is ambiguous in general, since there can be several relatively ordered, overlapping sublists as can be seen in the list (6, 2, 1, 3, 5, 4, 7). Some of their sublists are (6, 7), (2, 3, 5, 7), and (2, 3, 4, 7).

Furthermore, counting the number of unsorted elements is not unique either.

For example, in the list (4, 5, 1, 2, 3, 6, 7), the number of unsorted elements is two (4, 5) or three (1, 2, 3) depending upon which elements are considered as not being in order.

Therefore, we assume that the number of unsorted elements for a given list means the minimum number of elements which should be removed from the list to make the remainder of it be completely sorted. Then the number of unsorted elements in the list (4, 5, 1, 2, 3, 6, 7) is two and the unsorted elements are 4 and 5.

Then the number of unsorted elements in the list is always unique, although the selection of the unsorted elements may not be unique. For example, possible selections of unsorted elements for the list (6, 2, 1, 3, 5, 4, 7) above could be (6, 2, 5), (6, 2, 4), (6, 1, 5) and (6, 1, 4), however, the (minimum) number of unsorted elements is always 3.

We have clarified the meaning of the number of unsorted elements, however, the distribution of the unsorted elements cannot be defined without employing the mathematical tools: permutation, inversion, and inversion table, which are introduced in the next few sections.

Furthermore, counting the number of unsorted elements by hand is impractical and no algorithm can be easily implemented on computer to count that number. So an algorithm for counting the number of unsorted elements is also developed later in this chapter.

B. Permutations and Inversion.

A sequence of n numbers is called a permutation if it consists of only integers 1 through n, i.e., a permutation p = {1, 2, 3, ... n}. If the elements in the permutation are in ascending order, i.e., p = (1, 2, 3, ... n), then it is called an identity permutation.

It is easy to see that any list can be related to a permutation by associating the smallest number with 1, second smallest with 2, etc., if considering ascending order and the reverse if descending order.

For example, the list (4.1, 2.7, 6, 9.85, 0, -2.2) corresponds to the permutation (4, 3, 5, 6, 2, 1).

Here we assume that no elements in the list are repeated and all lists are to be sorted in ascending order.

For a given permutation, P:

$$P = \{a_i \mid a_i = 1, 2, 3, \ldots n\}$$

If $i<j$ and $a_i>a_j$, then the pair (a_i, a_j) is called an inversion.

For example, P = (4, 3, 1, 2, 7, 6, 5) has eight inversions: (4, 3), (4, 1), (4, 2), (3, 1), (3, 2), (7, 6), (7, 5), (6, 5).

If there is no inversion in P, then P is the identity permutation. So sorting can be thought of as a process of removing the inversions from the corresponding permutation derived from a given list.

C. Inversion Table.

The Inversion Table $b_1, b_2, \ldots b_n$ of the permutation $a_1, a_2, \ldots a_n$ is obtained by letting b_j be the number of elements to the left of j that are greater than j(1); b_j is the number of a_i's where

$$(a_i, 1 \leq i < j) \wedge (j < a_i) \text{ for all } 1 \leq j \leq n.$$

For example, if

$$P = (4, 3, 1, 2, 7, 6, 5)$$

then IV = (2, 2, 1, 0, 2, 1, 0).

For 1 in P, there are two elements, 4, 3 in P which are bigger than 1 and to the left of 1, so the first element in IV is 2 ($b_1 = 2$), for 2 in P, there are two elements ($b_2 = 2$), 4, 3, which are bigger than 2 and to the left of 2, and for 3 in P, only one elent ($b_3 = 1$), 4, and for 4 in P, there is no element ($b_4 = 0$) in P which is bigger than 4 and to the left of 4, and so on.

Note that the last element of Inversion Table (b_n) is always 0, since no element in P can be bigger than the number of elements (n).

There are some very important and interesting properties in Inversion Table.

First of all, there is a one-to-one correspondence between permutations and Inversion Tables. This fact, discovered by Hall [5] and Knuth [1] gives an algorithm for constructing a permutation from an inversion table, and we will give an example to illustrate the algorithm.

For

IV = (2, 2, 1, 0, 2, 1, 0),

starting from the rightmost and going backwards, place the corresponding element into its proper position to satisfy that the number of bigger elements to the left is identical to the value in Inversion Table (Fig. 1).

Figure 1

Corresponding elements in permutation	IV (backwards)	Permutation	Comment
7	0	7	7 is biggest ... 0
6	1	7 6	7>6 ... 1
5	2	7 6 5	7>6 6>5 ... 2
4	0	4 7 6 5	 0
3	1	4 3 7 6 5	4>3 ... 1
2	2	4 3 2 7 6 5	4>2 3>2 ... 2
1	2	4 3 1 2 7 6 5	4>1 3>1 ... 2

We can easily see that the final permutation obtained from IV is identical to the original P.

Another interesting property of the Inversion Table is that each

element indicates the number of interchanges between two adjacent elements needed to place the corresponding element in the original permutation in its proper position.

Using the Inversion Table (2, 2, 1, 0, 2, 1, 0), the first element 2 in IV says that 1 in P can be in its proper position (i.e. leftmost) after two interchanges, and Fig. 2 shows the relationship between the number of interchanges and the values of elements in Inversion Table as interchanges (indicated by arrows) are done.

Figure 2

$p = (4, 3, 1, 2, 7, 6, 5)$	$IV = (2, 2, 1, 0, 2, 1, 0)$
$p_1 = (1, 4, 3, 2, 7, 6, 5)$	$IV_1 = (0, 2, 1, 0, 2, 1, 0)$
$p_2 = (1, 2, 4, 3, 7, 6, 5)$	$IV_2 = (0, 0, 1, 0, 2, 1, 0)$
$p_3 = (1, 2, 3, 4, 7, 6, 5)$	$IV_3 = (0, 0, 0, 0, 2, 1, 0)$
$p_4 = (1, 2, 3, 4, 5, 7, 6)$	$IV_4 = (0, 0, 0, 0, 0, 1, 0)$
$p_5 = (1, 2, 3, 4, 5, 6, 7)$	$IV_5 = (0, 0\ \ 0, 0, 0, 0, 0)$

Also, the sum of each element in Inversion Table is equal to the total number of inversions in the given permutation, and also is equal to the total number of interchanges of adjacent elements to form the identity permutation.

For example, the sum is 8 (2 + 2 + 1 + 0 + 2 + 1 + 0 = 8) for the inversion table of the list (4, 3, 1, 2, 7, 6, 5), and that is the number of inversions as shown in the beginning of this section and it becomes identity permutation.

D. "HOW-MANY".

From the previous section on Inversion Tables, it follows that a non-zero element in IV indicates that its corresponding element in P is

not in order, and if all elements in P, whose corresponding elements in IV are non-zero, are removed from P, then obviously it will yield a completely sorted subset of the permutation.

e.g. P = (2, 3, 4, 5, 1, 6, 7)

IV = (4, 0, 0, 0, 0, 0, 0)

There is only one non-zero element in IV, and if we eliminate the corresponding 1 in P, then P is in order.

This method seems quite easy and natural, however, this does not guarantee the minimum number as shown by the following example:

e.g. P = (7, 1, 2, 3, 4, 5, 6)

IV = (1, 1, 1, 1, 1, 1, 0)

There are six non-zero elements in IV, but the minimum number of elements to be removed from P is only one, 7, instead of six elements.

Recall that there are two cases when an element is out of order, i.e., one is when a small number is placed to the right from its proper position as shown in the first example of this section, and the other is when a big number is placed to the left from its proper position as in the second example.

Considering the two cases, we come up with an algorithm which uses the Inversion Table to count the minimum number of unsorted elements in the list, i.e., the minimum number to be removed from the permutation to make it completely sorted.

Algorithm 1: Count the number of unsorted elements.

(1) First step:

Divide Inverison Table into subgroups of consecutive entries, whose first element is non-zero, but each subgroup should end with at least one zero. The number of zeros that should

follow at the last portion of the subgroup is given by:

$$\text{Min } \{\text{no. of non-zero elements in subgroup}, \quad \text{value of last non-zero element in subgroup}\}$$

and zeros not included in subgroups can be completely neglected since the corresponding elements in P_i are in order and not affected by the unsorted elements in subgroups.

Recall again that the former part of the formula represent the case as shown in the first example in this section, i.e., small numbers should be removed from the list, while the latter represents the second case.

e.g. IV = (1 1 1 3 0 0 0 3 2 1 0 2 2 2 2 0 0 5 0 0 1 0 0) would be divided into 5 subgroups: (1 1 1 3 0 0 0), (3 2 1 0), (2 2 2 2 0 0), (5 0) Ø, (1 0) Ø, and 20th and 23rd elements (zeros) are omitted as indicated in the algorithm. The number of trailing zeros in each subgroup was calculated by applying the formula, i.e:

For the 1st subgroup min {4, 3} = 3

2nd subgroup min {3, 1} = 1

3rd subgroup min {4, 2} = 2

4th subgroup min {1, 5} = 1

5th subgroup min {1, 1} = 1

(2) Second Step:

For each subgroup, if the number of non-zero elements is less than the largest element in the i^{th} subgroup,

then: the minimum number of elements to be removed from i^{th} subgroup is equal to the number of non-zero elements and we are done with i^{th} group.

otherwise: subtract 1 from each non-zero element in i^{th} subgroup and go back to First Step unless every element in

i^{th} subgroup is zero, in which case we are done with i^{th} subgroup. Whenever this process is executed, it also increases the number of elements to be removed by one ($n_i = n_i + 1$).

This algorithm ends in either process of Second Step and the total number of the unsorted elements in the list is $n = \sum_{n=1}^{k} n_i$, where k is the total number of subgroups.

e.g.,

Applying the above algorithm on IV:

IV = (1 1 1 3 0 0 0 3 2 1 0 2 2 2 2 0 0 5 0 0 1 0 0),

we will show how to find the minimum number of elements to be eliminated.

As mentioned earlier, we come up with five subgroups. And for each subgroup:

For (1 1 1 3 0 0 0):

a) no. of non-zero element (4)>largest element (3).

b) $n_1 = 0 + 1 = 1$.

c) subtract 1 from each non-zero element, then we get (0, 0, 0, 2, 0, 0, 0).

d) it forms only one subgroup (2, 0).

e) no. of non-zero element (1)<largest element (2).

$\therefore n_1 = 1 + 1 = 2$.

For (3, 2, 1, 0):

a) no. of non-zero element (3) = largest element (3).

b) $n_2 = 0 + 1 = 1$.

c) subtract 1 from each non-zero element, then we get (2, 1, 0, 0).

d) neglect last zero, and we get (2, 1, 0).

e) no. of non-zero element (2) = largest element (2).

f) $n_2 = 1 + 1 = 2$.

g) subtract 1 from each non-zero element, then we get (1, 0, 0).

h) neglect last zero, and we get (1, 0).

i) no. of non-zero element (1) = largest element (1).

j) $n_2 = 2 + 1 = 3$.

k) subtract 1 from the non-zero element, then all elements are zero, and we are done

$\therefore n_2 = 3$.

For (2, 2, 2, 2, 0, 0,):

a) no. of non-zero element (4)>largest element (2).

b) $n_3 = 0 + 1 = 1$.

c) subtract 1 from each non-zero element, then we get (1, 1, 1, 1, 0, 0).

d) it forms only one subgroup (1, 1, 1, 1, 0).

e) no. of non-zero element (4)>largest element (1).

f) $n_3 = 1 + 1 = 2$.

g) subtract 1 from each non-zero element, then we get (0, 0, 0, 0, 0), therefore done.

$\therefore n_3 = 2$.

For (5,0):

a) no. of non-zero element (1)<largest element (5)

$\therefore n_4 = 1$.

For (1,0):

a) no. of non-zero element (1) = largest element (1)

$\therefore n_5 = 1$.

$$\text{Total no. } n = n_1 + n_2 + n_3 + n_4 + n_5$$

$$= 2 + 3 + 2 + 1 + 1 = 9$$

We can check this value directly from the corresponding permutation, i.e.,

P = (5), 1, 2, 3, 6, 7, (4), 11, (10), (9), (8), (16), (17), 12, 13, 14, 15, 19, 20, 22, (21), 23, (18),

and elements with circles should be removed to make P completely ordered.

Suppose Algorithm 1 gives the number of unsorted elements, k. Since we will represent the "HOW-MANY" by the ratio of the unsorted elements to the total number of elements in permutation:

$$\text{"HOW-MANY"} = \frac{k}{n}$$

where n is the total number of elements.

If a permutation is an identity, "HOW-MANY" becomes 0, and for a completely out-of-ordered permutation, "HOW-MANY" will approach to 1, and if half of the elements are unsorted, its value will be around 0.5. Thus this ratio seems to coincide with our intuition.

E. "HOW FAR".

"HOW-FAR" is the ratio of total number of inversions to the average number of inversions for a given number of unsorted elements, and will be derived as follows:

Suppose there are n elements in a permutation, and only one element is out of order. Then the element can be placed in the i^{th} position with the probability $\frac{1}{n}$, and the probability that j^{th} position is in its proper place is also $\frac{1}{n}$.

The expected number of transfers that an unsorted element should move to be in its proper place is

$$\sum_{i=1}^{n} \sum_{j=1}^{n} |i-j| \frac{1}{n^2}$$

$$= \frac{2}{n^2} \sum_{i=1}^{n} i(n-i) = \frac{(n+1)(n-1)}{3n}$$

If there are k unsorted elements in the list, the expected number of transfers becomes

$$\sum_{i=1}^{k} \frac{(n-i)(n-i+2)}{3(n-i+1)}$$

and we can approximate the value, since $\sum_{i=1}^{k} (n-i+1)$ is much bigger than $\sum_{i=1}^{k} \frac{1}{(n-i+1)}$, i.e.,

$$\sum_{i=1}^{k} \frac{(n-i)(n-i+2)}{3(n-i+1)}$$

$$= \frac{1}{3} \left(\sum_{i=1}^{k} (n-i+1) - \sum_{i=1}^{k} \frac{1}{(n-i+1)} \right) \tilde{=} \frac{1}{3} \sum_{i=1}^{k} (n-i+1)$$

$$= \frac{1}{3} \left(nk + k - \frac{k(k+1)}{2} \right)$$

Therefore, "HOW-FAR" would be defined as $\frac{SUM_{IV}}{\frac{1}{3}\left(nk + k - \frac{k\,(k+1)}{2}\right)}$,

however, we divide it by 3 to make its range between 0 and 1, i.e.,

$$\text{"HOW-FAR"} = \frac{SUM_{IV}}{\left(nk + k - \frac{k\,(k+1)}{2}\right)}$$

where SUM_{IV} is the sum of corresponding inversion table elements.

When k = 0, i.e., identity permutation, "HOW-FAR" becomes $\frac{0}{0}$ and undefined, but we will assume it to be 0 to make "HOW-FAR" be continuous for all possible k values.

We will derive the upper limit of "HOW-FAR" as follows:

For given k number of unsorted elements, maximum value of SUM_{IV} is $(n-1) + (n-2) + \cdots + (n-k) = \frac{(2n-k-1)k}{2}$.

Thus, upper bound of "HOW-FAR"

$$= \frac{\frac{k\,(2n - k - 1)}{2}}{nk + k - \frac{k\,(k + 1)}{2}} = \frac{\left(2 - \frac{k}{n} - \frac{1}{n}\right)}{2 - \frac{k}{n} + \frac{1}{n}}$$

$$\tilde{=} 1 \text{ (assumed } \frac{1}{n} \cong 0).$$

Notice that both "HOW-MANY" and "HOW-FAR" are defined between 0 and 1.

Then we define the "nearly" sorted region on the graph in terms of "HOW-MANY" and "HOW-FAR" as in Fig. 3.

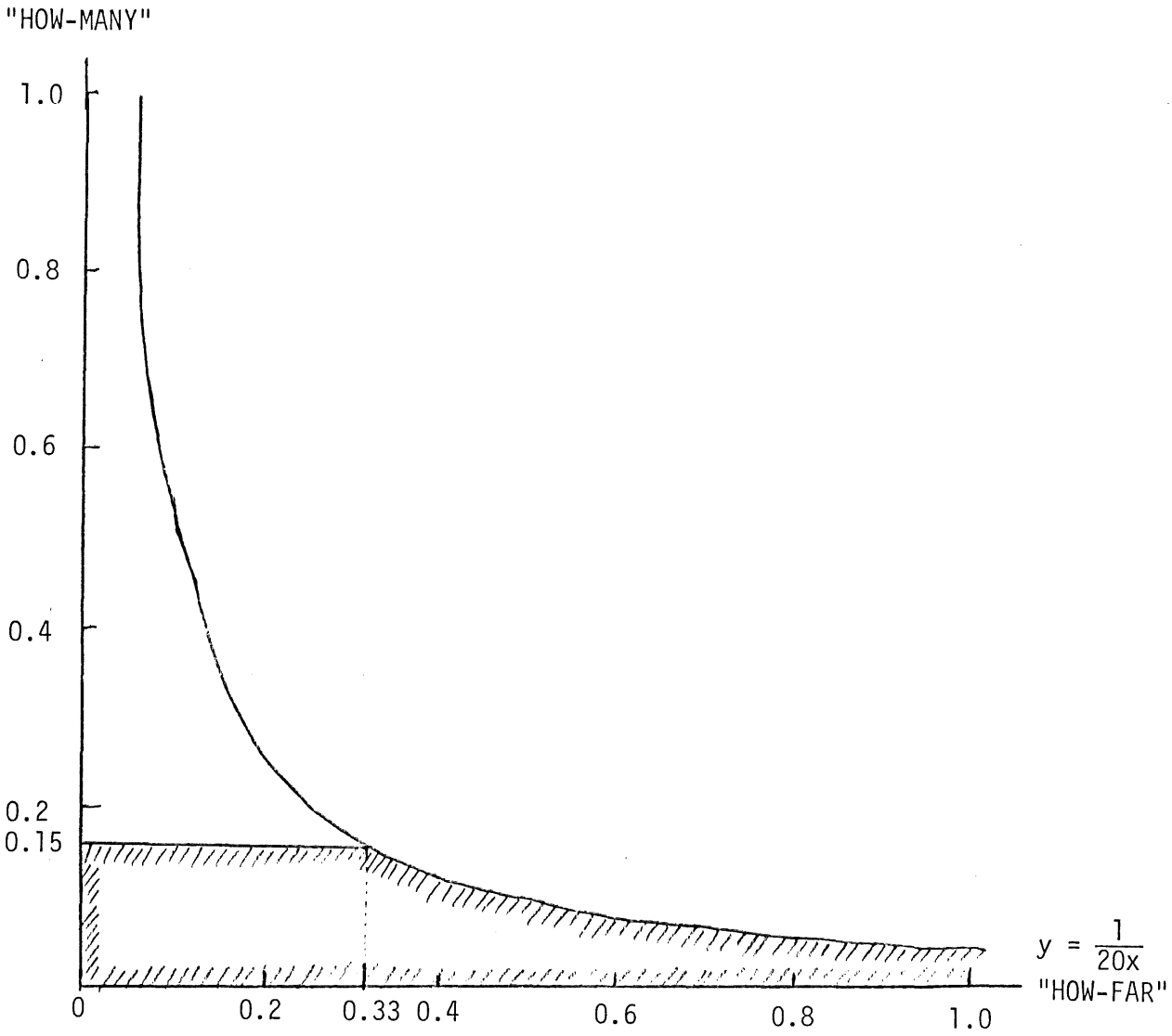
If both "HOW-FAR" and "HOW-MANY" are very small, undoubtedly the list is nearly sorted.

We should also add that if the "HOW-MANY" is very small, then the list is also nearly sorted regardless of the value of "HOW-FAR", i.e., if "HOW-MANY" of a list is around 0.05, it will be called nearly sorted regardless of the "HOW-FAR."

For other extreme cases, a list is also called nearly sorted with "HOW-MANY" around 0.15, if "HOW-FAR" is less than or equal to 0.33.

Figure 3

Nearly Sorted Region



IV. TEST RESULT

A. Tested Sorting Algorithms.

Testing was performed on CYBER system. The sorting algorithms tested were:

(1) Straight Insertion Sort

(2) Quicker Sort

(3) Shell Sort

(4) Straight Merge Sort

(5) Tree Sort

(6) Heap Sort

Two well-known sorting algorithms, Bubble Sort and Quick Sort, were not included for comparison. Because Bubble Sort as defined in Knuth [1] requires too many unnecessary comparisons between elements. Some modified Bubble Sorts have been developed, however, none of these requirements leads to an algorithm better than Straight Insertion Sort [1].

Quick Sort is replaced by Quicker Sort, since the middle element is usually an excellent choice, especially if the input list is nearly sorted and it can split the list into two halves of equal size [3].

Furthermore, it has been empirically determined that Quicker Sort is the fastest sorting technique on most machines [8].

B. Test Set-up and Results.

"HOW-FAR" as defined in Chapter III affects the performance of some soritng algorithms, however, the effect of "HOW-FAR" was not tested in this paper due to the substantial execution time required.

As commented in Introduction, the length of the input list also affects the efficiency of sorting algorithms. Hence, nearly sorted list with various "HOW-MANY" values and of various lengths were tested.

Input test data was created as follows: an identity permutation of desired size (50, 200, 500, 1000 and 2000) was rearranged by random number generator to obtain the desired "HOW-MANY" values, which were 0.0 (in order), 0.02 (2% out of order), 0.05, 0.10 and 0.15. As mentioned, "HOW-FAR" was not considered, and actual "HOW-FAR" values for tested lists were around 0.3.

Computer programs were written in FORTRAN for each sorting algorithm. Any programming language will give the same results, since we only count the number of comparisons, moves and exchanges.

After running the program for input generated, we obtained weighted values from the number of comparisons, moves and exchanges as defined in Chapter II. Entries for Table 1 are the weighted values divided by the length of lists. With these values, it is easier to compare the lists of different sizes.

Performance of Straight Insertion Sort, Shell Sort and Quicker Sort are summarized in Fig. 4, Fig. 5 and Fig. 6, respectively. Some more variations of "HOW-MANY" (0.01, 0.03, 0.08, 0.12) and the length (100) were added to draw those graphs.

Performances of all sorting algorithms are depicted in Fig. 7.

As the results of a sequence of tests indicate, the Straight Merge Sort, Tree Sort and Heap Sort do not take into account the sortedness of the input list. In other words, they spent almost the same effort in sorting the unsorted lists, regardless of the unorderedness of the input lists.

Straight Merge Sort is better than Tree Sort and Heap Sort, however, it is far from being selected as the best sorting algorithm on nearly sorted list.

Table 1

Sorting Algorithm	HOW-MANY	No. of Elements 50	200	500	1000	2000
Straight Insertion	0.0	2.94	2.99	2.99	3.0	3.0
	0.02	3.56	5.12	9.13	16.47	26.77
	0.05	5.34	12.72	18.36	32.71	91.65
	0.10	6.78	13.90	31.24	70.06	124.77
	0.15	5.34	19.53	49.37	130.30	173.60
Quicker	0.0	6.32	7.57	10.43	11.20	12.34
	0.02	6.72	8.42	10.25	12.05	14.08
	0.05	6.41	8.87	11.11	12.58	13.77
	0.10	6.70	9.32	11.21	13.84	14.01
	0.15	7.26	10.01	12.51	13.51	14.0
Shell	0.0	5.06	6.01	7.01	8.0	9.0
	0.02	5.30	9.50	14.20	15.46	18.57
	0.05	6.52	12.78	15.31	18.65	22.72
	0.10	8.38	13.69	16.82	21.97	24.38
	0.15	8.72	15.62	16.63	22.72	25.92
Straight Merge	0.0	10.50	14.0	16.95	17.70	
	0.02	10.50	14.05	17.20	18.06	20.92
	0.05	10.52	14.44	17.28	18.14	21.02
	0.10	10.64	14.47	17.31	18.24	21.22
	0.15	10.58	14.56	17.43	18.32	21.37
Tree	0.0	15.38	19.43	21.95	24.95	25.95
	0.02	15.38	19.43	21.95	24.95	25.95
	0.05	15.38	19.43	21.95	24.95	25.95
	0.10	15.38	19.43	21.95	24.95	25.95
	0.15	15.38	19.43	21.95	24.95	25.95
Heap	0.0	17.04	23.32	26.22	29.25	32.23
	0.02	16.64	23.27	27.17	30.22	33.20
	0.05	17.02	23.14	24.77	30.16	33.26
	0.10	16.76	22.23	26.90	30.57	33.24
	0.15	16.82	22.34	26.54	29.94	32.66

Figure 4

Straight Insertion Sort

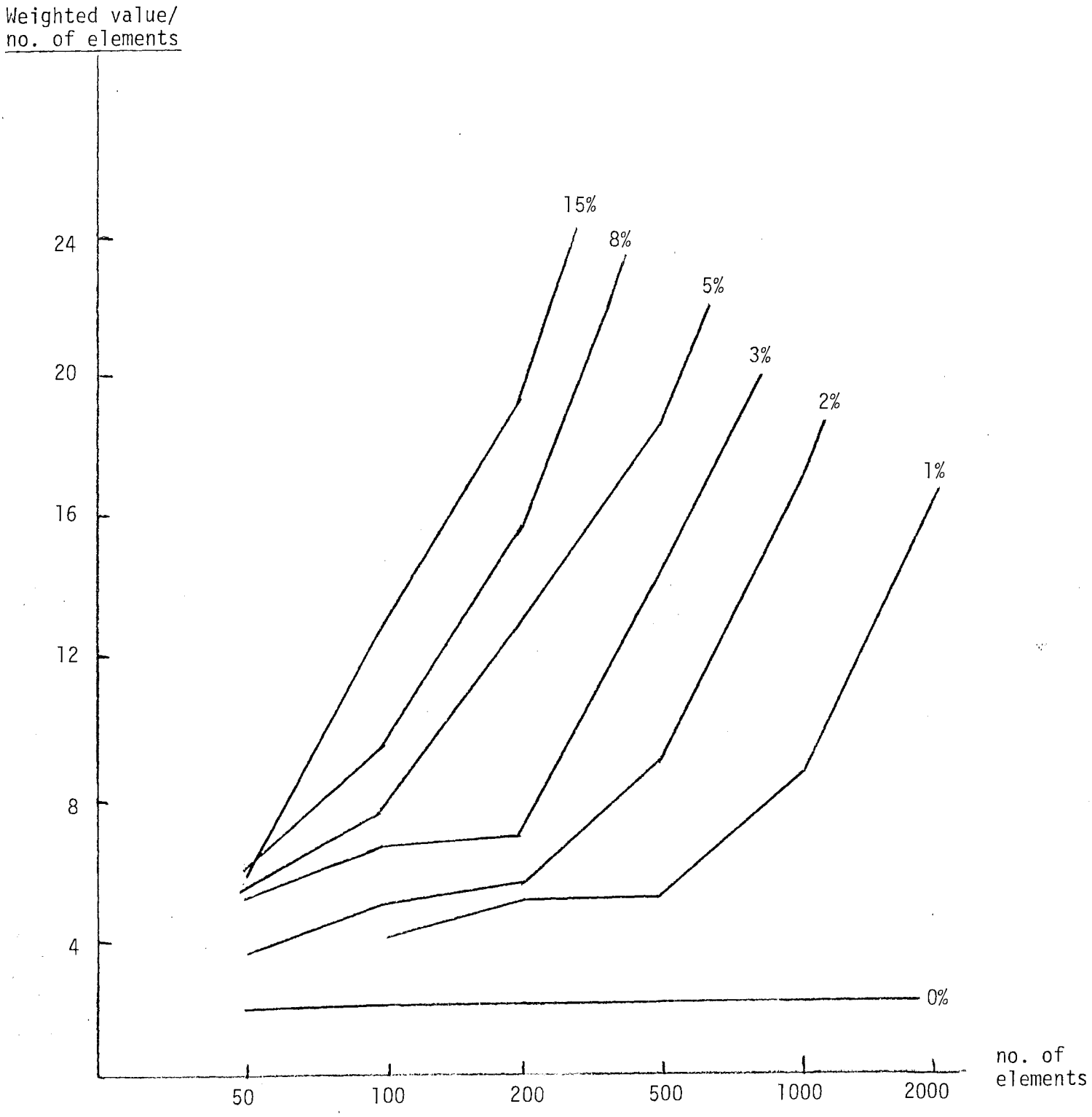


Figure 5

Shell Sort

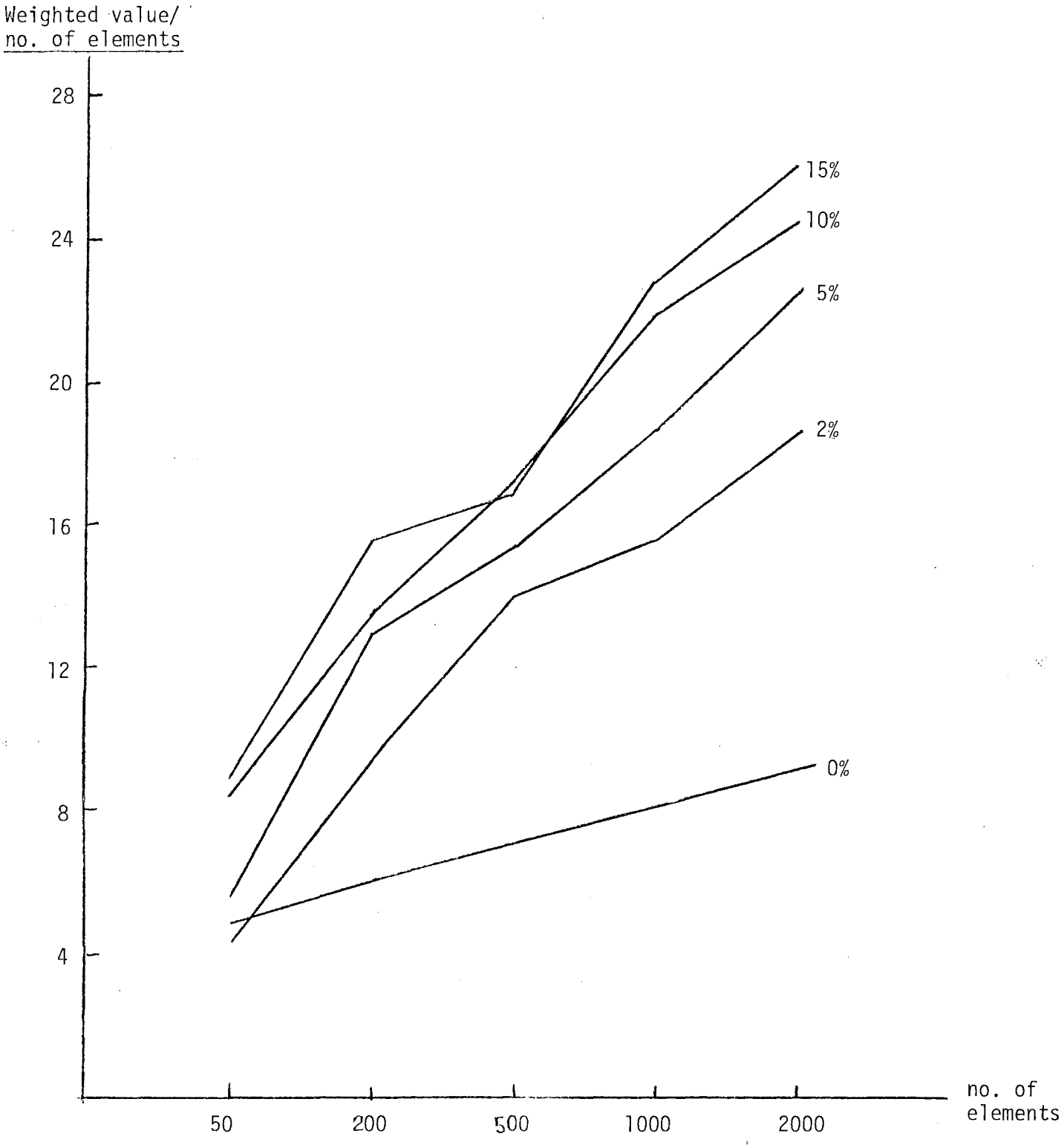


Figure 6

Quicker Sort

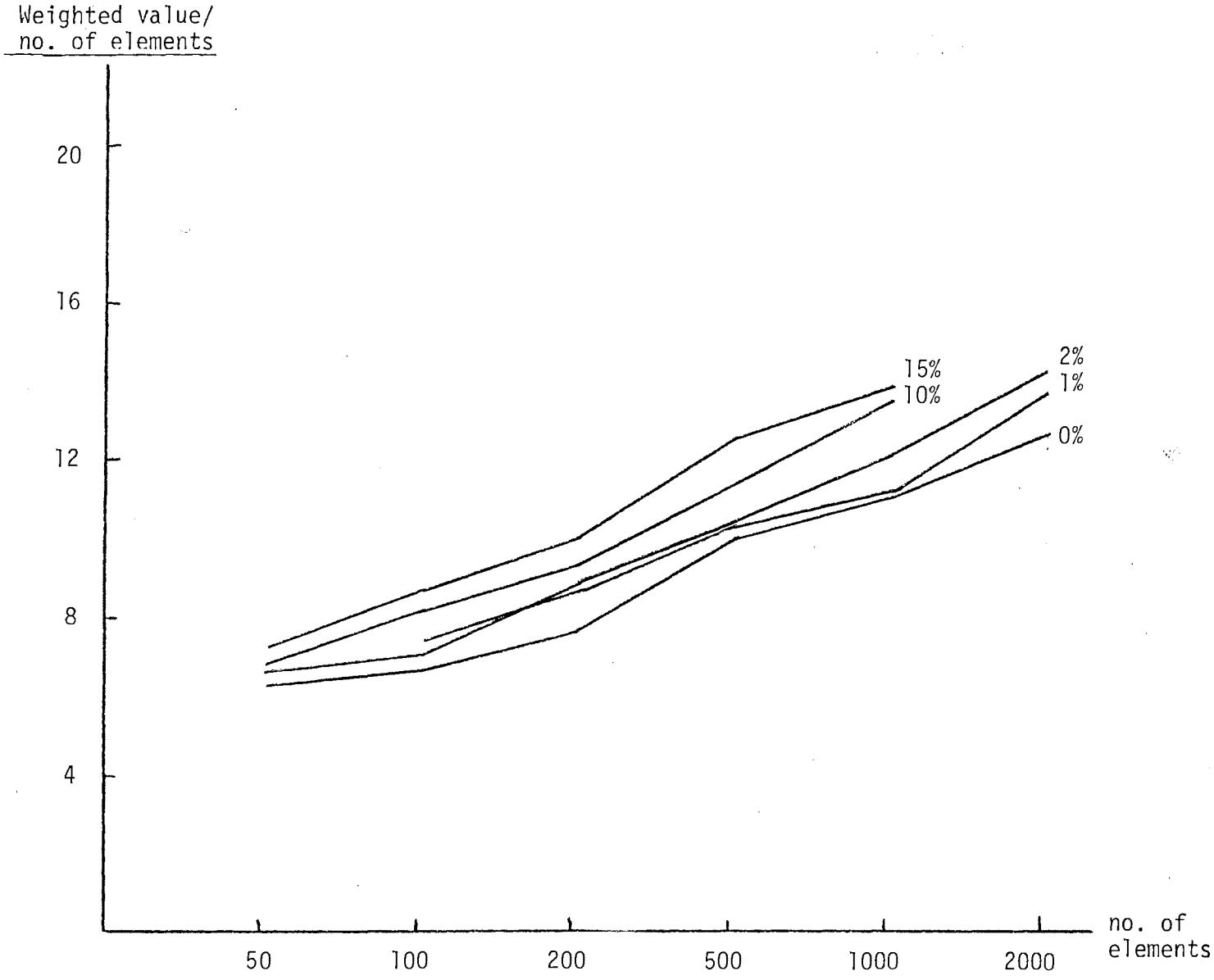
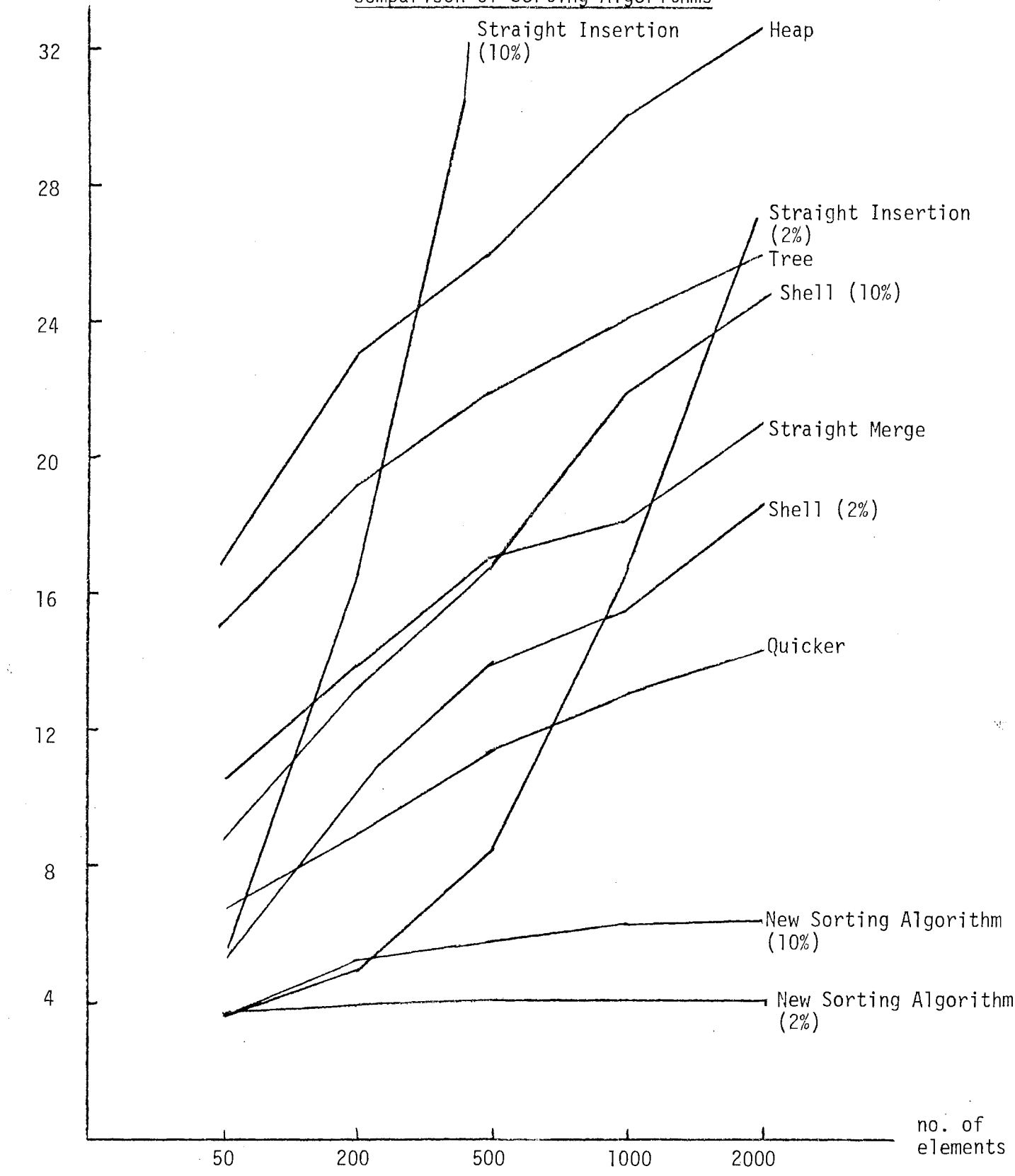


Figure 7

Comparison of Sorting Algorithms



Meanwhile, Straight Insertion Sort and Shell Sort are greatly affected by the sortedness of input and are very good for very nearly sorted and small size lists. But the efficiency of both of these sorting algorithms decrease rapidly as the size of input increases. Thus, both sorting algorithms are not only recommended for small sized nearly sorted lists whose number of elements are less than or equal to 200, or very nearly sorted lists (2% or less out of order). Recall that Straight Insertion Sort does not require any overhead and is simple to program. So it would be better to use Straight Insertion Sort.

Overall performance of Quicker Sort is very good. The efficiency of Quicker Sort is a little affected by the sortedness of the input list, however, it is best when the size of input list is big.

Best sorting algorithm regions on nearly sorted lists is shown in Fig. 8 in terms of the size of input lists and "HOW-MANY."

In addition to the six sorting algorithms listed in the previous section, extra tests were performed on modified versions of sorting algorithms: combination of Quicker Sort and Straight Insertion Sort, and the Revised Heap Sort, without any noticeable improvement.

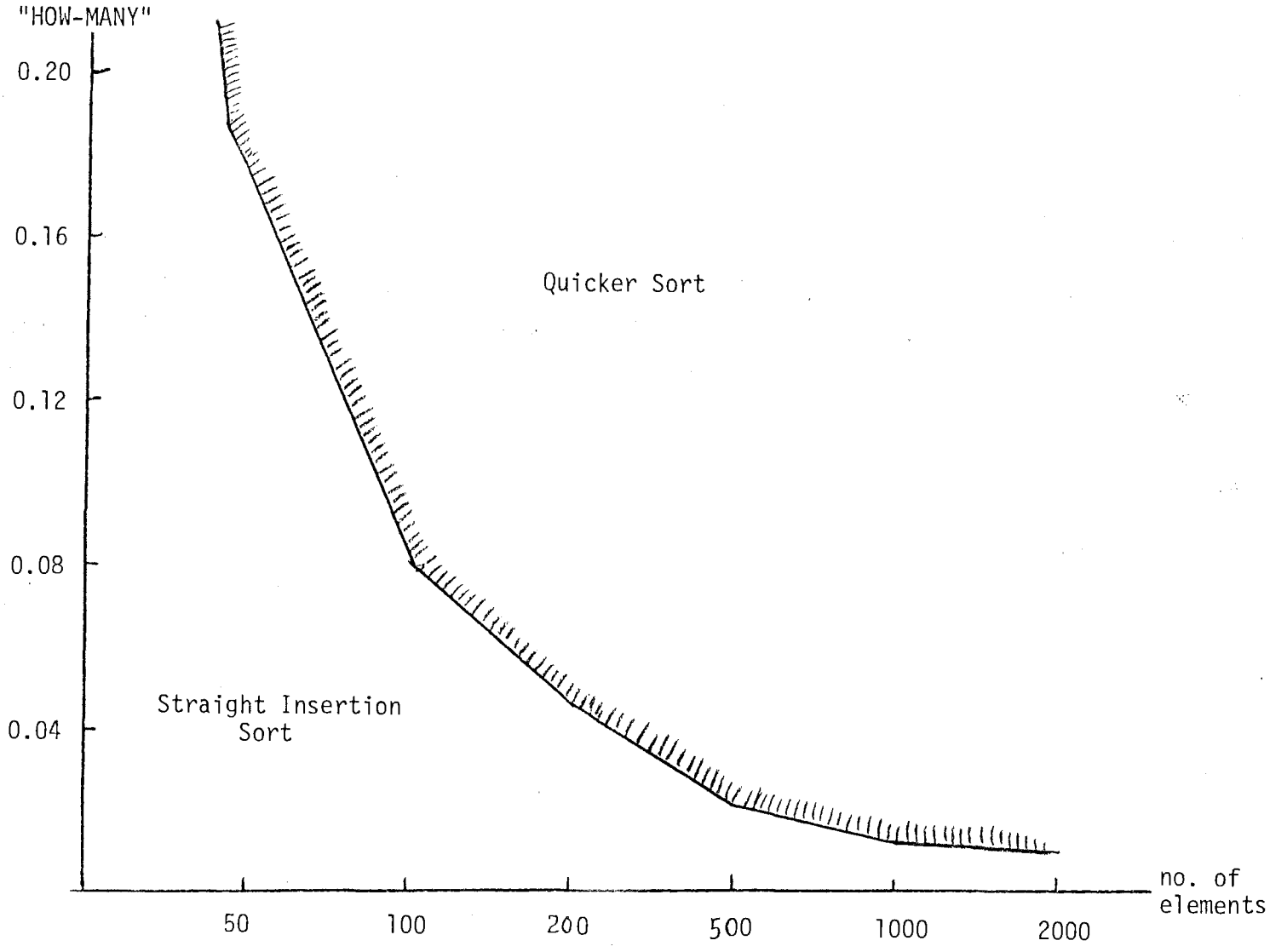
It is empirically proven by Singleton [7] that Straight Insertion is the best sorting algorithm when the number of elements in the list to be sorted is less than or equal to 11.

And as mentioned, Quicker Sort is very fast. But it can be improved by combining it with Straight Insertion Sort. In other words, apply Quicker Sort to sort the list, and whenever the number of elements in the subgroup is less than or equal to 11, apply Straight Insertion to sort that subgroup elements.

This method was tried on nearly sorted lists, but it only improved Quicker Sort slightly.

Figure 8

Best Sorting Algorithm
(Straight Insertion Sort vs. Quicker Sort)



As for Revised Heap Sort, we reversed the definition of Heap. In other words, a sequence of keys $K_1, K_2, \ldots K_n$ forms a "reversed heap" if $K_{\lfloor j/2 \rfloor} \leq K_j$ for $1 \leq \lfloor j/2 \rfloor < j \leq n$. Then the smallest key appears on the top of the reversed heap. When the input list is nearly sorted, this method requires much fewer exchanges than the original Heap Sort does. However, the Reversed Heap Sort has almost the same number of comparisons as Heap Sort, and hard to be considered as best sorting algorithm on nearly sorted lists.

So even the Reversed Heap Sort cannot compete against Straight Insertion Sort or Quicker Sort on nearly sorted list.

C. New Sorting Algorithm.

A new sorting algorithm was developed especially for nearly sorted list.

It is a combination of Quicker Sort and Merge Sort. Basically it scans the source list and selects the already sorted elements, and stores them in an array. The remaining unsorted elements and the elements which are not known whether they are in order or not at scanning time are stored in another array.

We sort only this array by applying Quicker Sort, then we have two arrays whose elements are all in order in each array.

Hence we can merge them together to have a completely sorted list.

Complete New Sorting Algorithm is given in Appendix B, written in hypothetical structured programming language. And test results are shown in Table 2. Performance of New Sorting Algorithm for selected "HOW-MANY" (0.02, 0.10) was depicted in Fig. 7 for comparison purposes. As we can see in Fig. 7, New Sorting Algorithm is very fast and stable as the size of input lists grows, while most of the other sorting

algorithms are seriously affected by the size of input lists and performances are drastically dropped.

New Sorting Algorithm is, at worst, almost as fast as Quicker Sort and fully takes advantage of the sortedness of the input list. From Fig. 7, it looks substantially better than Quicker Sort for 200 or more elements, and New Sorting Algorithm is recommended on nearly sorted lists, especially if the size of input is large (more than 1000).

Table 2. New Sorting Algorithm.

New Sorting Algorithm	HOW-MANY	No. of Elements					
		50	100	200	500	1000	2000
	0.0	3.0	3.0	3.0	3.0	3.0	3.0
	0.01		3.78	3.73	3.96	4.02	4.13
	0.02	3.78	3.83	3.97	4.21	4.31	4.34
	0.03	3.94	3.97	4.21	4.38	4.47	
	0.05	3.90	4.52	4.56	4.73	4.92	5.02
	0.08	3.92	4.82	5.08	5.45	5.51	
	0.10	3.90	4.95	5.31	5.72	6.26	6.22
	0.12	3.98	5.48	5.61	6.40	6.37	
	0.15	3.90	5.85	6.07	6.97	7.06	7.48

V. SUMMARY

In this paper, two measures, "HOW-MANY" and "HOW-FAR," were defined to represent the orderedness of a list and nearly sorted region suggested by those two measures.

This paper compared the performance of several sorting algorithms on nearly sorted lists. Straight Insertion Sort performed best on very nearly sorted lists or small lists and Quicker Sort performed best on the remaining cases of nearly sorted lists, when New Sorting Algorithm was not considered.

New Sorting Algorithm which is developed in this paper for nearly sorted list performed as well as or better than any other sorting algorithm. In terms of computer execution time, New Sorting Algorithm took only one-half to one-third as much time as Quicker Sort on nearly sorted lists with 1000 or more elements.

Some topics were left for further investigation. "HOW-FAR," which has a great effect on the efficiency of Straight Insertion Sort, and possible affects the efficiency of other sorting algorithms, was not considered in the test results in this paper.

Repeated elements in the list were not allowed, and some modifications on permutation may be necessary to take care of such cases.

Finally, a better sorting algorithm than the one developed in this paper may be possible on nearly sorted list and other measures of nearly sorted list may be defined and investigated.

REFERENCES

(1) KNUTH, D.E. "Sorting and Searching." The Art of Computer Programing, Vol. 3, Addison-Wesley, 1975.

(2) MARTIN, W.A. "Sorting." Computing Survey, Vol. 3, No. 4, Dec. 1971.

(3) SCOWEN, R.S. "Algorithm 271: Quicker Sort." CACM 8, 11 (Nov. 1965).

(4) WEIDE, B. "A Survey of Analysis Techniques for Discrete Algorithms." Computing Survey, Vol. 9, Dec. 1977.

(5) HALL, M. Proc. Symp. Applied Math. 6, American Math. Society, 1956.

(6) NIVAT, P. "Sorting of Permutation." PERMUTATIONS, Actes du Colloque, Paris, Juillet. 1972.

(7) SINGLETON, R.C. "Algorithm 347: An Efficient Algorithm for Sorting with Minimal Storage." CACM 12, 3 (March 1969).

(8) LEWIS, T.G. "Applying Data Structure." Houghton-Mifflin, 1976.

(9) BOOTHROYD, J. "Algorithm 201: Shell Sort." CACM 8, 6. Aug. 1963.

(10) KAUPE, A.F. "Algorithm 144: Tree Sort 2." CACM 5, 12. Dec. 1962.

APPENDIX A

1. Straight Insertion Sort [Knuth, "Sorting and Searching" (1975), pp. 81].

Algorithm S (*Straight insertion sort*). Records $R_1, \ldots, R_N$ are rearranged in place; after sorting is complete, their keys will be in order, $K_1 \leq \cdots \leq K_N$.

S1. [Loop on j.] Perform steps S2 through S5 for $j = 2, 3, \ldots, N$; then terminate the algorithm.

S2. [Set up i, K, R.] Set $i \leftarrow j - 1$, $K \leftarrow K_j$, $R \leftarrow R_j$. (In the following steps we will attempt to insert R into the correct position, by comparing K with K_i for decreasing values of i.)

S3. [Compare K, K_i.] If $K \geq K_i$, go to step S5. (We have found the desired position for record R.)

S4. [Move R_i, decrease i.] Set $R_{i+1} \leftarrow R_i$, then $i \leftarrow i - 1$. If $i > 0$, go back to step S3. (If $i = 0$, K is the smallest key found so far, so record R belongs in position 1.)

S5. [R into R_{i+1}.] Set $R_{i+1} \leftarrow R$. ∎

2. Quicker Sort [Collected algorithms from CACM, 271-P1-0].

ALGORITHM 271
QUICKERSORT [M1]

R. S. Scowen* (Recd. 22 Mar. 1965 and 30 June 1965)
National Physical Laboratory, Teddington, England

* The work described below was started while the author was at English Electric Co. Ltd, completed as part of the research programme of the National Physical Laboratory and is published by permission of the Director of the Laboratory.

```
procedure quickersort(a, j);
  value j; integer j; array a;
begin integer i, k, q, m, p; real t, x; integer array ut,
  lt[1:ln(abs(j)+2)/ln(2)+0.01];
comment The procedure sorts the elements of the array a[1:j]
  into ascending order. It uses a method similar to that of QUICK-
  SORT by C. A. R. Hoare [1], i.e., by continually splitting the
  array into parts such that all elements of one part are less than
  all elements of the other, with a third part in the middle con-
  sisting of a single element. I am grateful to the referee for point-
  ing out that QUICKERSORT also bears a marked resemblance
  to sorting algorithms proposed by T. N. Hibbard [2, 3]. In par-
  ticular, the elimination of explicit recursion by choosing the
  shortest sub-sequence for the secondary sort was introduced by
  Hibbard in [2].
    An element with value t is chosen arbitrarily (in QUICKER-
  SORT the middle element is chosen, in QUICKSORT a random
  element is chosen). i and j give the lower and upper limits of
  the segment being split. After the split has taken place a value
  q will have been found such that a[q] = t and a[I] ≤ t ≤ a[J]
  for all I, J such that i ≤ I < q < J ≤ j. The program then
  performs operations on the two segments a[i:q-1] and a[q+1:j]
  as follows. The smaller segment is split and the position of the
  larger segment is stored in the lt and ut arrays (lt and ut are
  mnemonics for lower temporary and upper temporary). If the
  segment to be split has two or fewer elements it is sorted and
  another segment obtained from the lt and ut arrays. When no
  more segments remain, the array is completely sorted.
    REFERENCES:
1. HOARE, C. A. R. Algorithms 63 and 64. Comm. ACM 4 (July
     1961), 321.
2. HIBBARD, THOMAS N. Some combinatorial properties of
     certain trees with applications to searching and sorting.
     J. ACM 9 (Jan. 1962), 13.
3. ——. An empirical study of minimal storage sorting. Comm.
     ACM 6 (May 1963), 206-213;
  i := m := 1;
V: if j-i > 1 then
  begin comment This segment has more than two elements,
    so split it;
    p := (j+i) ÷ 2;
    comment p is the position of an arbitrary element in the
      segment a[i:j]. The best possible value of p would be one
      which splits the segment into two halves of equal size, thus
      if the array (segment) is roughly sorted, the middle ele-
      ment is an excellent choice. If the array is completely
      random the middle element is as good as any other.
        If however the array a[1:j] is such that the parts a[1:j÷2]
      and a[j÷2+1:j] are both sorted the middle element could
      be very bad. Accordingly in some circumstances
      p := (i+j) ÷ 2 should be replaced by p := (i+3×j) ÷ 4
      or p := RANDOM(i, j) as in QUICKSORT;
    t := a[p];
    a[p] := a[i];
    q := j;
    for k := i + 1 step 1 until q do
    begin comment Search for an element a[k] > t startin
      from the beginning of the segment;
      if a[k] > t then
      begin comment Such an a[k] has been found;
        for q := q step -1 until k do
        begin comment Now search for a[q] < t starting fro
          the end of the segment;
          if a[q] < t then
          begin comment a[q] has been found, so exchan
            a[q] and a[k];
            x := a[k];
            a[k] := a[q];
            a[q] := x;
            q := q-1;
            comment Search for another pair to exchange;
            go to L
          end
        end for q;
        q := k - 1;
        comment q was undefined according to Para. 4.6.5
          the Revised ALGOL 60 Report [Comm. ACM 6 (J
          1963), 1-17];
        go to M
      end;
L:  end for k;
    comment We reach the label M when the search going
      wards meets the search coming down;
M:  a[i] := a[q];
    a[q] := t;
    comment The segment has been split into the three pa
      (the middle part has only one element), now store
      position of the largest segment in the lt and ut arrays a
      reset i and j to give the position of the next largest segme
    if 2 × q > i + j then
    begin
      lt[m] := i;
      ut[m] := q-1;
      i := q+1
    end
    else
    begin
      lt[m] := q+1;
      ut[m] := j;
      j := q-1
    end;
    comment Update m and split this new smaller segme
    m := m+1;
    go to N
  end
  else if i ≥ j then
  begin comment This segment has less than two elements
    go to P
  end
  else
  begin comment This is the case when the segment has j
      two elements, so sort a[i] and a[j] where j = i + 1;
    if a[i] > a[j] then
```

```
    begin
      x := a[i];
      a[i] := a[j];
      a[j] := x
    end;
    comment If the lt and ut arrays contain more segments
      to be sorted then repeat the process by splitting the smallest
      of these. If no more segments remain the array has been
      completely sorted;
P:  m := m-1;
    if m > 0 then
    begin
      i := lt[m];
      j := ut[m];
      go to N
    end;
  end
end quickersort
```

3. Shell Sort [Collected algorithms from CACM, 201-P1-0].

ALGORITHM 201
SHELLSORT
J. BOOTHROYD
English Electric-Leo Computers, Kidsgrove, Staffs, England

```
procedure Shellsort (a, n); value n; real array a; integer n;
comment a[1] through a[n] of a[1:n] are rearranged in ascending
  order. The method is that of D. A. Shell, (A high-speed sorting
  procedure, Comm. ACM 2 (1959), 30-32) with subsequences
  chosen as suggested by T. N. Hibberd (An empirical study of
  minimal storage sorting, SDC Report SP-982). Subsequences
  depend on m1 the first operative value of m. Here m1 = 2^k - 1
  for 2^k ≤ n < 2^(k+1). To implement Shell's original choice of m1 =
  [n/2] change the first statement to m := n;
begin integer i, j, k, m; real w;
  for i := 1 step i until n do m := 2 × i - 1;
  for m := m ÷ 2 while m ≠ 0 do
    begin k := n - m;
      for j := 1 step 1 until k do
        begin for i := j step -m until 1 do
          begin if a[i+m] ≥ a[i] then go to 1;
            w := a[i]; a[i] := a[i+m]; a[i+m] := w;
          end i;
        1: end j
    end m
end Shellsort;
```

4. Straight Merge Sort [Knuth, "Sorting and Searching" (1975), pp. 164].

Algorithm S (*Straight two-way merge sort*). Records $R_1, \ldots, R_N$ are sorted using two memory areas as in Algorithm N.

S1. [Initialize.] Set $s \leftarrow 0$, $p \leftarrow 1$. (For the significance of variables s, i, j, k, l, d, see Algorithm N. Here p represents the size of ascending runs to be merged on the current pass; q and r keep track of the number of unmerged items in a run.)

S2. [Prepare for pass.] If $s = 0$, set $i \leftarrow 1$, $j \leftarrow N$, $k \leftarrow N$, $l \leftarrow 2N + 1$; if $s = 1$, set $i \leftarrow N + 1$, $j \leftarrow 2N$, $k \leftarrow 0$, $l \leftarrow N + 1$. Then set $d \leftarrow 1$, $q \leftarrow p$, $r \leftarrow p$.

S3. [Compare $K_i{:}K_j$.] If $K_i > K_j$, go to step S8.

S4. [Transmit R_i.] Set $k \leftarrow k + d$, $R_k \leftarrow R_i$.

S5. [End of run?] Set $i \leftarrow i + 1$, $q \leftarrow q - 1$. If $q > 0$, go back to step S3.

S6. [Transmit R_j.] Set $k \leftarrow k + d$. Then if $k = l$, go to step S13; otherwise set $R_k \leftarrow R_j$.

S7. [End of run?] Set $j \leftarrow j - 1$, $r \leftarrow r - 1$. If $r > 0$, go back to step S6; otherwise go to S12.

S8. [Transmit R_j.] Set $k \leftarrow k + d$, $R_k \leftarrow R_j$.

S9. [End of run?] Set $j \leftarrow j - 1$, $r \leftarrow r - 1$. If $r > 0$, go back to step S3.

S10. [Transmit R_i.] Set $k \leftarrow k + d$. Then if $k = l$, go to step S13; otherwise set $R_k \leftarrow R_i$.

S11. [End of run?] Set $i \leftarrow i + 1$, $q \leftarrow q - 1$. If $q > 0$, go back to step S10.

S12. [Switch sides.] Set $q \leftarrow p$, $r \leftarrow p$, $d \leftarrow -d$, and interchange $k \leftrightarrow l$. If $j - i < p$, return to step S10; otherwise return to S3.

S13. [Switch areas.] Set $p \leftarrow p + p$. If $p < N$, set $s \leftarrow 1 - s$ and return to S2. Otherwise sorting is complete; if $s = 0$, set

$$(R_1, \ldots, R_N) \leftarrow (R_{N+1}, \ldots, R_{2N}).$$

(The latter copying operation will be done if and only if $\lceil \lg N \rceil$ is odd, regardless of the distribution of the input. Therefore it is possible to predict the location of the sorted output in advance, and copying will usually be unnecessary.) ∎

5. Tree Sort [Collected algorithms from CACM, pp. 144-P1-0].

ALGORITHM 144
TREESORT 2
ARTHUR F. KAUPE, JR.
Westinghouse Electric Corp., Pittsburgh, Penn.

```
procedure TREESORT 2 (UNSORTED, n, SORTED, k, ordered);
  value n, k;
integer n, k; array UNSORTED, SORTED; Boolean procedure ordered;
comment TREESORT 2 is a generalized version of TREESORT
  1. The Boolean procedure ordered is to have two real arguments. The array SORTED will have the property that ordered
  (SORTED[i], SORTED[j]) is true when j > i if ordered is a
  linear order relation;
begin integer i, j; array m1 [1:2×n−1]; integer array m2
  [1:2×n−1];
procedure minimum; if ordered (m1[2×i], m1[2×i+1]) then
  begin m1[i] := m1[2×i]; m2[i] := m2[2×i] end else
  begin m1[i] := m1[2×i+1]; m2[i] := m2[2×i+1] end minimum;
for i := n step 1 until 2 × n − 1 do begin m1[i] := UNSORTED
  [i−n+1]; m2[i] := i end
for i := n − 1 step −1 until 1 do minimum;
for j := 1 step 1 until k do
  begin SORTED[j] := m1[1]; i := m2[1]; m1[i] := infinity;
    for i := i ÷ 2 while i > 0 do minimum end
end TREESORT 2
```

6. Heap Sort [Knuth, "Sorting and Searching" (1975), pp. 146-147].

Algorithm H (*Heapsort*). Records $R_1, \ldots, R_N$ are rearranged in place; after sorting is complete, their keys will be in order, $K_1 \leq \cdots \leq K_N$. First we rearrange the file so that it forms a heap, then we repeatedly remove the top of the heap and transfer it to its proper final position. Assume that $N \geq 2$.

H1. [Initialize.] Set $l \leftarrow \lfloor N/2 \rfloor + 1$, $r \leftarrow N$.

H2. [Decrease l or r.] If $l > 1$, set $l \leftarrow l - 1$, $R \leftarrow R_l$, $K \leftarrow K_l$. (If $l > 1$, we are in the process of transforming the input file into a heap; on the other hand if $l = 1$, the keys $K_1 K_2 \ldots K_r$ presently constitute a heap.) Otherwise set $R \leftarrow R_r$, $K \leftarrow K_r$, $R_r \leftarrow R_1$, and $r \leftarrow r - 1$; if this makes $r = 1$, set $R_1 \leftarrow R$ and terminate the algorithm.

H3. [Prepare for "sift-up."] Set $j \leftarrow l$. (At this point we have

$$K_{\lfloor k/2 \rfloor} \geq K_k \quad \text{for} \quad l < \lfloor k/2 \rfloor < k \leq r; \tag{6}$$

and record R_k is in its final position for $r < k \leq N$. Steps H3–H8 are called the "sift-up" algorithm; their effect is equivalent to setting $R_l \leftarrow R$ and then rearranging $R_l, \ldots, R_r$ so that condition (6) holds also for $\lfloor k/2 \rfloor = l$.)

H4. [Advance downward.] Set $i \leftarrow j$ and $j \leftarrow 2j$. (In the following steps we have $i = \lfloor j/2 \rfloor$.) If $j < r$, go right on to step H5; if $j = r$, go to step H6; and if $j > r$, go to H8.

H5. [Find "larger" son.] If $K_j < K_{j+1}$, then set $j \leftarrow j + 1$.

H6. [Larger than K?] If $K \geq K_j$, then go to step H8.

H7. [Move it up.] Set $R_i \leftarrow R_j$, and go back to step H4.

H8. [Store R.] Set $R_i \leftarrow R$. (This terminates the "sift-up" algorithm initiated in step H3.) Return to step H2. ∎

APPENDIX B

New Sorting Algorithm

```
PROCEDURE NEW (A, N); VALUE N;
ARRAY A[N+1], B[N], C[N/3];
     COMMENT - A[N] contains source input when this procedure is called,
     and returns sorted list through this array.
          B[N] contains sorted elements from source input,
          C[N/3] contains unsorted adjacent element pairs;
     INTEGER IND1, IND2, JND1, JND2, I; N;
     COMMENT -
          IND1 is index for B-array at first stage,
          IND2 is index for C-array at first stage,
          JND1 is index for B-array when merged,
          JND2 is index for C-array when merged;
     IND1←IND2←0;
     A[N+1]←infinity;
     For I = 1 STEP 1 UNTIL N
          If A[I] ≤ A[I + 1]
               THEN IND1←IND1+1;
                    B[IND1]←A[I];
               ELSE IND2←IND2+2;
                    C[IND2-1]←A[I+1];
                    C[IND2]←A[I];
                    I = I+1;
               WHILE AND (IND1>0, B[IND1]>A[I+1], I≤N) DO
```

```
                    IND2←IND2+2;
                    C[IND2]←B[IND1];
                    C[IND2-1]←A[I+1];
                    IND1←IND1-1;
                    I←I+1;
               ENDWHILE;
          ENDIF;
     ENDFOR;
```

COMMENT: At this point, all elements in B-array are in order and those in the C-array is not. So only sort C-array by Quicker Sort and then merge B- and C-array into A-array.

```
CALL QUICKER (C, IND2);
JND1←JND2←1;
FOR I = 1 STEP 1 UNTIL N
     IF B[JND1]≥C[JND2]
          THEN
               A[I]←C[JND2];
               JND2←JND2+1;
               IF JND2>IND2
                    THEN
                         FOR I = I+1 STEP 1 UNTIL N
                         JND1←JND1+1;
                         A[I]←B[JND1];
                         ENDFOR;
                    ENDIF;
               ELSE
                    A[I]←B[JND1];
                    JND1←JND1+1 ;
```

```
                IF JND1>IND1
                THEN
                    FOR I = I+1 STEP 1 UNTIL N
                        JND2←JND2+1;
                        A[I]←C[JND2];
                    ENDFOR;
                ENDIF;
            ENDIF;
        ENDFOR;
END PROCEDURE;
```