Theoretical Computer Science 1 (1975) 185–190. © North-Holland Publishing Company

A FAST STABLE SORTING ALGORITHM WITH ABSOLUTELY MINIMUM STORAGE*

F. P. PREPARATA**

Istituto di Scienze dell'Informazione, Università di Pisa, Pisa, Italy

Communicated by M. Paterson
Received March 1974
Revised November 1974

Abstract. An algorithm is described which sorts n numbers in place with the property of stability, i.e., preserving the original order of equal elements. The algorithm requires absolutely minimum storage $0(\log_2 n)$ bits for program variables and a computation time at most $0(n(\log_2 n)^2)$.

1. An algorithm which sorts n numbers in order is said to be *stable* if the initial ordering of equal numbers is preserved in the final arrangement. The feature of stability is clearly desirable in several applications and may be achieved, for example, by using one of the known sorting algorithms after tagging the elements to be sorted with their initial locations; such approach, however, requires additional storage for the tags of the order of $n \log_2 n$ bits. Other schemes of stable sorting may be devised, which require an additional storage of the same order as above.

D. E. Knuth ([1] p. 388) defines a sorting algorithm to require *minimum storage* if it uses at most $0((\log_2 n)^2)$ bits of additional storage. To justify this definition [1, 2] at most $\log_2 n$ program variables are postulated, each requiring $\log_2 n$ bits.

Knuth has also raised the question of the existence of a stable minimum storage (SMS) sorting algorithm which requires less than $0(n^2)$ units of time in its worst case, and/or in the average.

This problem has received some attention in the very recent past and a number of results have been obtained. R. L. Rivest [2] presented an SMS sorting algorithm which requires $0(n(\log_2 n)^2)$ units of time in the average, its worst case still being $0(n^2)$. Almost at the same time, R. B. K. Dewar [3] showed that SMS sorting can be achieved in time $0(n^{3/2})$ in the worst case.

In this paper we present the following result: stable sorting of n numbers can be achieved in time $0(n(\log_2 n)^2)$ with $0(\log_2 n)$ bits of storage for program variables.

* This work was supported in part by Grant N. 72.00254.42 of the National Research Council of Italy and in part by I.R.I.A., Le Chesnay, France.

** Present address: Coordinated Science Laboratory, University of Illinois at Urbana-Champaign, Urbana, Illinois 61801. U.S.A.

Notice that the latter requirement is the absolute minimum for any algorithm designed to handle n items.

After the original submission of this paper, additional interesting results by E. C. Horvath appeared in the literature [4]. Horvath obtained independently a result equivalent to ours, and also showed that running time can be asymptotically reduced to 0 $(n.\log_2 n.G(n))$. The algorithm described in this paper, however, is considerably simpler that Horvath's algorithms, and is based on a substantially different approach. For the benefit of readers familiar with Horvath's work, the two methods can be briefly contrasted as follows. Horvath's stable sorting is based on forming "contiguents" of increasing length, where contiguents are defined as segments of the ultimately sorted sequence. Our algorithm, instead, forms contiguents of the smallest length only at the last step of recursion.

2. In the interest of clarity, we shall describe at first a recursive version of the SMS algorithm. Subsequently we shall present its iterative rendition, which performs exactly the same merging steps and achieves the minimum storage performance. The procedure will be presented in the orderly style of Knuth [1]. Data structures to be used are unidimensional arrays of records. By $A[1:r]$ we shall denote a sequence $A[1], A[2], ..., A[r]$.

Algorithm S

This algorithm sorts a sequence $V[1:n]$ by iteration. At each iteration the stable merging Algorithm SM is applied to pairs of consecutive sorted subsequences of $V[1:n]$, whose length doubles at each iteration step.

S1. For $i = 0, 1, ..., \lceil n/4 \rceil - 1$* sort by straight insertion $V[4i+1:\min(4(i+1), n)]$.
S2. Set $M \leftarrow 1$ and go to step S5.
S3. Set $p \leftarrow \lfloor \lceil n/2^M \rceil /2 \rfloor$.
S4. For $j = 0, 1, ..., p-1$ apply Algorithm SM to $V[2^{M+1}j+1:2^M(2j+1)]$, $V[2^M(2j+1)+1:\min(n, 2^{M+1}(j+1))]$.
S5. Set $M \leftarrow M+1$. If $2^M \geqslant n$, halt; else go to step S3.

Remark 1. Moves of data are performed by steps S1 and S4. It is well-known that straight insertion is stable. We will show below that Algorithm SM is also stable (see Remark 2).

Algorithm SM (Stable-merge)

This algorithm accepts two sorted sequences $V[1:u]$ and $V[u+1:v]$, and recursively merges them with stability. The result is achieved by reducing the problem to the merging of two pairs of sorted sequences, of which the first pair has total length 2^r (where r is the unique integer for which $v/2 \leqslant 2^r < v$). For ease of reference,

* As usual, "$\lceil\ \rceil$" denotes "smallest integer not smaller than" and "$\lfloor\ \rfloor$" denotes "greatest integer not greater than".

by "reduction" we mean the task of replacing the original pair of sequences with the two shorter pairs.

SM1. Set $r \leftarrow \lfloor \log_2 v \rfloor - 1$
SM2. If $r = 1$, merge by straight insertion $V[1:u]$ and $V[u+1:v]$ and halt; else set $i \leftarrow 0, j \leftarrow 0$.
SM3. While $i+j < 2^r$, if $\{(V[i+1] \leqslant V[u+j+1]) \wedge (i < u)\}$ or $(j = v-u)$ set $i \leftarrow i+1$; else set $j \leftarrow j+1$.
SM4. Apply Algorithm SH to $\{V[i+1:u+j], j\}$
(Comment: the two sequences $V[i+1:u]$ and $V[u+1:u+j]$ are exchanged).
SM5. Apply Algorithm SM to $V[1:i]$, $V[i+1:2^r]$
SM6. Apply algorithm SM to $V[2^r+1:u+j]$, $V[u+j+1:v]$.

Remark 2. The only movement of data is the exchange performed by step SM4. This exchange is stable since, by the condition of step SM3, either $V[i+1] > V[u+j]$ or $V[i+1:u]$ is empty. In the latter case step SM4 is dummy. Notice also that at each point in the recursive execution of steps SM5 and SM6, all pairs of sorted subsequences have total length equal to a power of 2, except, possibly, the rightmost pair.

Algorithm SH (shift)

This algorithm accepts a sequence $V[a:b]$ and an integer $r < q = b-a+1$ and cyclically shifts $V[a:b]$ in place r positions to the right.

SH1. Set $l \leftarrow 0, h \leftarrow a$.
SH2. Set $V \leftarrow V[h]$ (store $V[h]$) and set $v \leftarrow h$.
SH3. Set $w \leftarrow v-r$.
SH4. If $w < a$, set $w \leftarrow w+q$.
SH5. If $w = h$, move $V[v] \leftarrow V$, set $l \leftarrow l+1$ and go to step SH7.
SH6. Move $V[v] \leftarrow V[w]$, set $l \leftarrow l+1$, $v \leftarrow w$ and go to step SH3.
SH7. If $l = q$, halt; else set $h \leftarrow h+1$ and go to step SH2.

3. We now estimate the time complexity of the algorithm described in the preceding section. Except for the initialization and control of counters (such as M in Algorithm S), the computational work consists of comparisons of elements and of data moves. These two types of operations occur in steps S1 and S4 of Algorithm S. With reference to step S1, it is well-known that in straight insertion sorting the worst-case number of data moves is proportional to the number of comparisons. Step S4 is a call of Algorithm SM; in the latter algorithm, comparisons of elements are performed in step SM3 to determine the two subsequences to be exchanged by Algorithm SH, which is called in step SM4. At most 2^r comparisons are performed in SM3 and at most twice as many data moves are performed in SM4. We conclude that the upper-bounds to the numbers of comparisons and of data moves are of

the same order, so that we shall use the number of comparisons as the criterion of complexity.

We now recall that Algorithm SM replaces the problem of merging the sequence pair $(V[1:u], V[u+1:v])$ with that of merging the two sequence pairs $(V[1:i], V[i+1:2^r])$ and $(V[2^r+1:u+j], V[u+j+1:v])$. Since the parameter r is a function of v but not of u, the computational work required by this reduction is a function only of the total length of the original pair, and consists of two parts:

(i) the determination of the two subsequences to be exchanged (Step SM3). As noted above, this requires at most 2^r comparisons.

(ii) the recursive application of Algorithm SM (first level of recursion) to $V[1:2^r]$ and $V[2^r+1:v]$. (Steps SM5 and SM6).

Therefore, denoting with $\psi(v)$ the worst-case computational work required for merging a sequence pair of total length v, we have the recurrence relation

$$\psi(v) = \psi(2^r)+\psi(v-2^r)+2^r, \qquad \text{with } r = \lceil \log_2 v \rceil - 1. \tag{1}$$

Notice that $\psi(v)$ is monotonic for $v \leqslant 4$. Assuming now inductively that $\psi(v)$ is monotonic for $v \leqslant 2^r$, equation (1) proves the monotonicity of $\psi(v)$. Thus, since we are interested in the rate of growth of $\psi(v)$, we may assume for ease of calculation that v be a power of 2. In this case, (1) becomes

$$\psi(v) = 2\psi\left(\frac{v}{2}\right) + \frac{v}{2} \tag{2}$$

which yields, $\psi(v) = 0(v \log_2 v)$.

Finally, we notice that in its last pass Algorithm S performs the merging (step S4) of two sorted sequences $V[1:2^M]$ and $V[2^M+1:n]$, with $M = \lceil \log_2 n \rceil - 1$. Thus, the worst-case computational work $\varphi(n)$ required for stably sorting n elements is given by the following recurrence relation:

$$\varphi(n) = \varphi(2^M)+\varphi(n-2^M)+\psi(n)$$

Here again, monotonicity of $\varphi(n)$ is immediately proved. Thus, referring to the case in which n is a power of 2, we obtain

$$\varphi(n) = 2\varphi\left(\frac{n}{2}\right)+\psi(n)$$

which yields, by standard methods, that $\varphi(n)$ is $0(n(\log_2 n)^2)$.

It is obvious that the efficiency of the stable sorting algorithm could be improved (by a multiplicative constant) when n is not a power of 2, by applying the merging algorithm to sorted sequences of nearly equal length.

4. Finally we consider the storage requirement for program variables. Algorithms S and SH use a number of variables independent of n. Instead, Algorithm SM uses a number of variables equal to the depth of recursion (i.e. $\log_2 n$ pointers or, equiv-

alently, $(\log_2 n)^2$ bits). These pointers indicate the abscissae of separation between pairs of sorted subsequences to be merged.

This storage requirement, however, can be reduced by noticing that the pointers mentioned above need not be stored explicitly, for they can be easily determined in the execution of the algorithm without altering the order of the time complexity. Moreover, straight insertion merging is done exclusively on pairs of sorted subsequences whose total length is 4 (except, possibly, once at the end of the execution).

With these observations in mind, it is simple to provide an iterative version of Algorithm SM, which operates as follows. At first a sequence of reduction passes is performed. Each pass halves the common total length of the sorted sequence pairs. When this length attains the value 4, each segment of length 4 is sorted by (stable) straight insertion and the merging process is completed. The only additional problem encountered is that, whenever a reduction must be performed, we know the total length of the two sorted subsequences to be processed, but not their separating abscissa u. Therefore a preliminary search for u must be performed, thereby adding at most a term linear in v to the right hand side of equation (2) and leaving unaltered the order of the time complexity. Obviously, it is also possible to devise a faithful iterative simulation of Algorithm SM, namely, one which performs the same *sequence* of merging steps: such algorithm is negligibly more efficient than that given below (since not all abscissae u's need be computed), but is considerably more complicated to describe.

Algorithm ISM (iterative stable merge)

This algorithm stably merges two sorted sequences $V[1:u]$ and $V[u+1:v]$.

ISM1. Set $r \leftarrow \lceil \log_2 v \rceil - 1$.

ISM2. Set $s \leftarrow 0$.

ISM3. (Comment: steps ISM3-5 determine the abscissa u separating the two sorted sequences in $V[s+1:\min(v, s+2^{r+1})]$.) Set $k \leftarrow s+1$.

ISM4. If $V[k] \leqslant V[k+1]$ set $k \leftarrow k+1$; else set $u \leftarrow k$ and go to step ISM6.

ISM5. If $k = \min(v, s+2^{r+1})$, set $r \leftarrow r-1$ and go to step ISM9; else go to ISM4.

ISM6. (Comment: steps ISM6-7 perform a reduction). While $i+j < 2^r$, if $\{(V[s+1+i] \leqslant V[u+1+j]) \wedge (i+s < u)\}$ or $(j+u = \min(v, s+2^{r+1}))$ set $i \leftarrow i+1$; else set $j \leftarrow j+1$.

ISM7. Apply Algorithm SH to $\{V[s+1+i:u+j], j\}$, and set $s \leftarrow s+2^{r+1}$.

ISM8. If $s > v$ set $r \leftarrow r-1$; else go to step ISM3.

ISM9. If $r > 1$ go to step ISM2.

ISM10. For $i = 0, 1, \ldots, \lceil v/4 \rceil - 1$, sort by straight insertion $V[4i+1:\min(4(i+1), v)]$ and halt.

It is evident that Algorithm ISM uses a constant number of program variables. We have therefore proved the following result:

Theorem. *Stable sorting of n numbers can be performed in time at most $0(n(\log_2 n)^2)$ with $0(\log_2 n)$ bits of additional storage for program variables.*

Acknowledgment

The author gratefully acknowledges the suggestions of an anonymous referee, which were of crucial importance in improving the time performance of the described algorithm.

References

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

[2] R. L. Rivest, A fast stable minimum-storage sorting algorithm, Report 43, IRIA, Le Chesnay, France (1973).

[3] R. B. K. Dewar, A stable minimum storage sorting algorithm, Information Processing Letters 2 (1974) 162–164.

[4] E. C. Horvath, Some efficient stable sorting algorithms, Proc. of Sixth ACM Symposium on Theory of Computing, Seattle, Wash. (1974) 194–215.