IRIA
laboria

Institut de Recherche
d'Informatique
et d'Automatique

Domaine de Voluceau
Rocquencourt
78150 - Le Chesnay
Tél. 954.90.20

laboratoire de recherche
en informatique
et automatique

A fast stable minimun-storage sorting algorithm

R.L Rivest

Rapport de Recherche n° 43
décembre 1973

A FAST STABLE MINIMUM-STORAGE SORTING ALGORITHM

Ronald L. Rivest

Abstract :

An algorithm is presented which sorts n numbers into order in average time $O(n(\log_2(n))^2)$ using at most $O((\log_2(n))^2$ bits of additional storage. Furthermore, the algorithm is stable (that is, it preserves the initial order of equal elements).

Résumé :

Nous présentons un algorithme qui trie n nombres dans un temps moyen $O(n(\log_2(n))^2$ en utilisant au plus $O(\log_2(n))^2$ bits de mémoire supplémentaires. De plus, cet algorithme est stable (c'est-à-dire qu'il conserve les éléments égaux dans leur ordre initial).

In this paper we answer in the affirmative a question raised by Knuth :

"Is there a stable minimum-storage sorting algorithm which requires less than $\mathcal{O}(n^2)$ units of time in its worst case, and/or on the average ?" [3, exercise 5.5.3, p.388].

A sorting algorithm rearranges a given set of n numbers into non-decreasing order. Thus if the input numbers are stored in memory locations M[1],..., M[n], when sorting is complete we have rearranged them so that

$$M[1] \leq M[2] \leq \ldots \leq M[n]. \qquad (1)$$

If there are equal elements in the input, more than one rearrangement will achieve the result (1).

A sorting algorithm is said to be stable if the initial relative ordering of equal elements is preserved in the final rearrangement. Thus if each element were "tagged" with its initial location, we would have at the end

$$(i < j) \Rightarrow (M[i] < M[j]) \vee ((tag[i] < tag[j]) \wedge (M[i] = M[j])). \qquad (2)$$

Equal elements are thus not to be considered "equal" but ordered according to their initial ordering in the input.

Stable sorting algorithms are very useful when some order already existing in the input file is to be preserved. For example, suppose we have a file of transactions, where each transaction lists a customer's name and a date, and suppose further that this file is already ordered by date. If we now sort this file by customer name in a stable manner, the output file will still be ordered by date within the group of transactions for each customer. This obviates considering the (name, date) pairs as single keys obeying a lexicographic ordering rule.

Stable sorting algorithms are easy to construct. For example, one can add the "tagging" scheme mentioned above to any standard sorting algorithm. Problems arise, however, when one wishes to increase the storage efficiency of the algorithm without drastically increasing the running time. Knuth defines a "minimum-storage" sorting algorithm as one which uses only $\mathcal{O}((\log_2(n))^2)$ additional bits of storage for program variables, in addition to the space used by the array M for the set of numbers being sorted. This is a reasonable definition since

it represents the amount of storage used by a recursive algorithm whose depth of recursion is bounded by $\log_2(n)$. Thus there is not enough available storage to allocate $\log_2(n)$ bits for a "tag" for each of the n numbers. How fast can a minimum storage stable sorting algorithm sort ?

The best previous answer to this question is $O(n^2)$. The classic "bubble sort" [3, p. 106] demonstrates this easily enough, since it uses only three variables, so that its storage utilization is $O(\log_2(n))$. Furthermore, it only exchanges values in adjacent memory locations, and then never if those values are equal, so it is stable. Other $O(n^2)$ minimum-storage stable sorting algorithms also exist [3, p. 388], but none were known which required less than $O(n^2)$ units of time, on the average or in the worst case.

In this paper we present a stable minimum-storage sorting algorithm whose average running time for any input multiset of numbers is $O(n(\log_2(n))^2)$. The worst case running time is still $O(n^2)$, so we have really only answered part of Knuth's question ; the best possible worst-case running time remains an open problem.

The algorithm has an overall structure similar to Quicksort [2]:

Algorithm A.

Records M[b], ..., M[c] are sorted stably in place, where $b \le c$, using $O((\log_2(n))^2)$ bits of additional storage.

A1. If $b \ge c$, the algorithm terminates. Otherwise set $m \leftarrow M[c]$.

A2. Using algorithm B, partition the elements M[b],..., M[c] about the value m, so that at the end we have, for some k, $b \le k \le c$, and all i, $b \le i \le c$,

$$(M[i] \le m) \equiv (i \le k). \tag{3}$$

That is, all values less than or equal to m are stored in M[b], ..., M[k] and all values greater than m are stored in M[k+1], ..., M[c]. Furthermore, the record originally in M[c] (with value m) has been moved to M[k], since algorithm B is stable.

A3. Use algorithm B again on the subfile M[b], ..., M[k], partitioning it about the value m, but modifying the definition of algorithm B

slightly so that all values equal to m are grouped together. Thus, after this step, we have for some values k' and k, $b \le k' < k \le c$:

$$\begin{aligned} i \le k' &\Rightarrow M[i] < m \\ k' < i \le k &\Rightarrow M[i] = m \\ k < i &\Rightarrow M[i] > m. \end{aligned} \tag{4}$$

A4. Use algorithm A recursively to sort the smaller of the two subfiles : M[b], ..., M[k'] and M[k+1], ..., M[c]. Then adjust b and c to point to the ends of the larger subfile, and go to step A1. □

Initially this algorithm is invoked with b = 1 and c = n.

It is clear that this algorithm uses $\mathcal{O}((\log_2(n))^2)$ bits of additional storage, since no more than $\log_2(n)$ levels of recursion can be invoked.

The standard partitioning algorithm [1] is very efficient (its running time is $\mathcal{O}(n)$), but not stable. We next show how a stable partitioning can be performed in $\mathcal{O}(n \log_2(n))$ units of time using minimal storage.(This is algorithm B referred to above).

The partitioning algorithm has to deal with only two types of numbers : those that are less than or equal to m, and those that are greater. For simplicity let us call the former "zeros" and the latter "ones". Thus we wish to move all the zeros to the left end of M, and all the ones to the right end, in a stable manner. (Here we adopt the usual convention that M[i] is to the "left" of M[i+1].)

Let us first consider the simple case where we have a run of s ones followed by a run of s zeros. That is, the input looks like :

$$1^s\ 0^s \tag{5}$$

In this case we can clearly perform a stable partition with only s transpositions, the i-th of which exchanges the i-th one with the i-th zero.

Encouraged by this much success, let us continue. How about s ones followed by t zeros, for some $t > s$? That is, we have :

$$1^s\ 0^t \tag{6}$$

We can clearly exchange the first s ones with the first s zeros in a stable fashion, as above. What remains is a similar problem involving s ones and only $t-s$ zeros, since the leftmost s zeros are now in their proper places :

$$0^s\ 1^s\ 0^{t-s} \tag{7}$$

This process can clearly be iterated, moving the block of s ones to the right s places each time, until only u zeros remain to their right where $u < s$. At this point we can reverse the roles of left and right, ones and zeros ; and move the block and forth in a manner which follows the Euclidean algorithm for computing gcd (s,t), until every element is in its proper place. The total number of transpositions involved is at most $s+t-1$, since every transposition puts at least one digit into its final location.

The following algorithm describes the above procedure exactly. (We will use this routine later.)

Algorithm C.

Records $M[j], \ldots, M[j+s+t-1]$ are stably partitioned about the value m, where records $M[j];\ldots, M[j+s-1]$ are known to be greater than m, and records $M[j+s], \ldots, M[j+s+t-1]$ are known to be less than or equal to m.

C1. Set $d \leftarrow \min(s,t)$. If $d = 0$, halt.

C2. For $i = 0,1,\ldots, d-1$, exchange the values of $M[j+s+i]$ and $M[j+s+i-d]$.

C3. If $s < t$ set $j \leftarrow j+d$ and $t \leftarrow t-d$; otherwise set $s \leftarrow s-d$. Go to step C1.

Now we consider the general partitioning problem. There will occur naturally in the input runs of ones followed by runs of zeros. That is, we consider the general case :

$$1^{s_1}\ 0^{t_1}\ 1^{s_2}\ 0^{t_2}\ \ldots\ 0^{t_w} \tag{8}$$

for some integer w. This formulation is completely general if we allow s_1 and t_w to assume the value zero.

We can clearly use algorithm C to reverse each set of s_i ones followed by t_i zeros, and iterate to completion. This idea unfortunately fails ; take $s_1 = t_1 = \ldots = t_w = 1$, for example. In this case we must use $n^2/4$ transpositions altogether, since each zero gets transposed with each one. The problem is that algorithm C is efficient only for long runs ; but in this example the run lengths remain fixed at 1, (except for the runs at the ends of M , which are already in their final places).

The way to overcome this difficulty is to only exchange <u>every other</u> block of ones with its subsequent block of zeros. That is, we exchange 1^{s_i} and 0^{t_i} only for <u>odd</u> values of i. Thus after the first pass we get the following situation :

$$0^{t_1}\,1^{s_1+s_2}\,0^{t_2+t_3}\,1^{s_3+s_4}\ldots\ . \qquad (9)$$

Now every exchange of adjacent runs has caused those runs to "coalesce" with the neighboring runs of the same type which were left fixed. It is clear that the length of the <u>shortest</u> run must double at each step, (if we don't consider the lengths of the runs at the ends of M). Thus at most $\log_2(n)$ passes are necessary to completely partition M about m, and the total partitioning time is $O(n \log_2(n))$.

For the sake of completeness we list the above algorithm in detail :

<u>Algorithm B</u>.

Records $M[\ell], \ldots, M[r]$ are stably partitioned about the value $m = M[r]$.

<u>B1</u>. (Skip over elements already in place.)
While $\ell < r$ and $M[\ell] \le m$, set $\ell \leftarrow \ell+1$.
While $\ell < r$ and $M[r] > m$, set $r \leftarrow r-1$.
If $\ell \ge r$, the algorithm terminates.

<u>B2</u>. Set $j \leftarrow \ell$

<u>B3</u>. (Get lengths of next two runs.)
Set $s \leftarrow 0$, $t \leftarrow 0$.
While $M[j+s] > m$, set $s \leftarrow s+1$.
While $j+s+t < r$ and $M[j+s+t] \le m$, set $t \leftarrow t+1$.

B4. Use algorithm C to exchange the s "ones" at M[j],...,M[j+s-1] with the subsequent t "zeros" in M[j+s], ..., M[j+s+t-1], then set j ← j+s+t, so that now either j > r or M[j] is the beginning of the next string of "ones".

B5. If j > r, go to step B1.

B6. (Skip next two runs.)
While M[j] > m, set j ← j+1.
While j < r and M[j] ≤ m, set j ← j+1.
Go to step B1. □

To modify this algorithm as required for step A3 of algorithm A, we need only to consider each element equal to m as a "one" instead of a zero. Thus each test of the form "M[...] ≤ m" becomes "M[...] < m" and each test of the form "M[...] > m" becomes "M[...] ≥ m".

An exact analysis of the preceding algorithms appears to be very difficult. For example, the running time of algorithm C, in terms of the number of transpositions used, is

$$s + t - \gcd(s,t), \text{ for } s > 0,\ t > 0. \tag{10}$$

Carrying this gcd(s,t) term into an analysis of algorithm B would require knowing the distribution of $\gcd(s_i,t_i)$ at each step, which in turn requires knowing the relative probability that equal elements occur.

Let B(n) denote the number of transpositions required by algorithm B to partition n numbers in the worst case. From our previous discussion we know that

$$B(n) \le n \log_2 (n). \tag{11}$$

To analyze the average running time of algorithm A, we need first to define what we mean by "average". Most analyses of sorting algorithms merely assume that the n input numbers are distinct. Here, however, we are considering a stable sorting algorithm, which is designed for situations in which equal elements are known to occur, and are perhaps even likely. We wish to show that our algorithm works well in all these cases.

Our model of the input runs as follows. We assume that there are $p_1 n$ occurences of the value 1, $p_2 n$ occurrences of the value 2, and so on, up to p_t occurrences of the value t, for some t, $1 \le t \le n$. This we may do without loss of generality. The vector $\overline{p} = (p_1,\ldots,p_t)$ thus satisifes the equality

$$\sum_{1\le i\le t} p_i = 1. \tag{12}$$

Let $A(n,\overline{p})$ denote the average number of transpositions used over the set of all arrangements of the $p_1 n$ 1's, $p_2 n$ 2's, etc, and let $A(n)$ denote the maximum value of $A(n,\overline{p})$, for any $\overline{p}$. We will show by induction on n that

$$(\exists c>0)A(n) \le c\ n\ (\log_2(n))^2. \tag{13}$$

The basis for the induction is easily established by choosing c large enough. For convenience we define

$$q_i = \sum_{1\le j\le i} p_i \text{ , for } 1 \le i \le t. \tag{14}$$

The following inequality for $A(n)$ is then easily derived :

$$A(n) \le 2n \log_2(n) + \sum_{1\le i\le t} p_i\ [A(nq_{i-1}) + A(n(1-q_i))]. \tag{15}$$

The first term on the right is an upper bound for the cost of the two partitioning steps A2 and A3. The occurrence $m = i$ (which occurs with probability p_i), causes the remaining two subproblems to be of size nq_{i-1} and $n(1-q_i)$.

Substituting (13) into (15) and simplifying, we obtain

$$A(n) \le f(\overline{p})\ n(\log_2(n))^2 + (g(\overline{p}) + 2)\ n \log_2(n) + h(\overline{p})\ n, \tag{16}$$

where

$$f(\overline{p}) = c\left[1 - \sum_{1\le i\le t} p_i^{\,2}\right], \tag{17}$$

$$g(\overline{p}) = c \sum_{1\le i\le t} p_i[\ q_{i-1} \log_2(q_{i-1}) + (1-q_i) \log_2(1-q_i)], \tag{18}$$

and

$$h(\overline{p}) = c \sum_{1\le i\le t} p_i\,(1-p_i)[(\log_2(1-p_i))^2 - 2 \log_2(1-p_i) + 1]. \tag{19}$$

We first note that

$$(\exists c')\ (\forall \overline{p})\ h(\overline{p}) \leq c' \tag{20}$$

so we need not worry about the term $h(\overline{p})n$ in (16) becoming large for certain $\overline{p}$. We next note that

$$(\forall \overline{p})\ g(\overline{p}) \leq o, \text{ and} \tag{21}$$

$$(\forall \overline{p})\ f(\overline{p}) \leq c, \tag{22}$$

so the only positive term to consider is the term $2\ n\ \log_2(n)$.

Let ϵ be a small positive real number, $0 < \epsilon < 1/2$. If $\max_i(p_i) \geq \epsilon$ then we have

$$2\ n\ \log_2(n) \leq (c-f(\overline{p}))\ n\ (\log_2(n))^2 \tag{23}$$

if $c = c(\epsilon)$ is chosen sufficiently large. On the other hand, if $\max_i(p_i) < \epsilon$, it is not difficult to show that

$$g(\overline{p}) + 2 < 0, \tag{24}$$

again assuming that $c = c(\epsilon)$ is chosen sufficiently large.

In any case, we arrive at the result

$$A(n) = \mathcal{O}(n(\log_2(n))^2). \tag{25}$$

which was to be shown.

Conclusions

The above algorithm demonstrates that stable minimum-storage sorting can be done quickly on the average. It remains an open problem whether this result can be improved to $\mathcal{O}(n\ \log_2(n))$ or whether $\mathcal{O}(n(\log_2(n))^2)$ is indeed optimal. In addition, it remains to be shown whether stable minimum-storage sorting can be done in the worst-case in time less than $\mathcal{O}(n^2)$.*

*I have just been informed by Professor Donald E. Knuth that Professor R.B.K. Dewar (Illinois Inst. Tech.) has a minimal-storage stable sorting algorithm that runs in worst-case time $\mathcal{O}(n^{3/2})$.

References

[1] Hoare, C.A.R. "Algorithm 63 (Partition)", CACM 4 (July 1961), 321.

[2] Hoare, C.A.R. "Algorithm 64 (Quicksort)", CACM 4 (July 1961), 321.

[3] Knuth, Donald E. The Art of Computer Programming, Vol 3 : Sorting and Searching. Addison-Wesley Publishing Company. (Menlo Park, California, 1973).