# SORTING NUMBERS IN LINEAR EXPECTED TIME AND OPTIMAL EXTRA SPACE *†

Teofilo F. GONZALEZ

*Mathematical Sciences Program, The University of Texas at Dallas, Richardson, TA 75080, U.S.A.*

Donald B. JOHNSON

*Department of Computer Science, The Pennsylvania State University, University Park, PA 16802, U.S.A.*

Received 30 October 1981; revised version received 9 July 1982

An algorithm is shown that sorts n numbers, each representable in h bits, in $\Theta(n)$ time on the average where the numbers to be sorted are selected randomly from some interval. The algorithm only uses $\Theta(\log n)$ bits of extra space, which is asymptotically optimal.

*Keyword:* Sorting in linear expected time

## 1. Introduction

Among the most widely used internal sorting algorithms are Quicksort [4,6,7] and Heapsort [7,12]. The expected running time of both these algorithms is $\Theta(n \log n)$ when sorting n numbers, each representable in h bits and each permutation of which is equally likely. Quicksort uses $\Theta(\log^2 n)$ bits of extra space; Heapsort uses $\Theta(\log n)$.

In 1965 MacLaren [8] presented an algorithm which sorts numbers in $\Theta(n)$ expected time when the input numbers are independently and identically distributed uniformly. We note that this *randomness assumption* implies each permutation to be equally likely, but the converse does not hold. MacLaren's algorithm requires for some fixed integer $p \geq 1$ at least $n + n^{1/p}$ extra locations, each large enough to hold the number n. Variants proposed by others also require asymptotically significant extra space. Knuth's improvement [7] needs at least $n^{1/2}$ extra locations. Dobosiewicz's recursive version of MacLaren's algorithm [1,2] requires at least 2n extra locations, as do other related algorithms [9,13].

Our algorithm sorts n numbers in $\Theta(n)$ expected time under the randomness assumption and requires only $\Theta(\log n)$ bits of extra space. This amount of space is equivalent to a constant number of registers that can hold a pointer to an input number and therefore is optimal to within a constant factor. Our complexity model charges unit cost for arithmetic operations. We assume that execution of arithmetic operations and swapping the contents of two memory locations do not use extra space. Swapping locations a and b may be implemented, for example, by a sequence of three exclusive OR operations applied to the locations, assigning the results to a, to b and to a again.

In order to determine the extra space required by our algorithm it is necessary to assume that the memory locations holding the input numbers, or *keys*, have a fixed size which in the worst case is necessary to represent the keys. Thus, it is impossible to encode additional information in a location containing a key without risking loss of information in the key. To make this assumption concrete, let each location containing a key be capable of

* This research was supported in part by the National Science Foundation under Grants MCS 77-21092 and MCS 80-02684.

† A preliminary version of this paper was presented at the Allerton Conference, September, 1978 [6].

0020-0190/82/0000-0000/$02.75 

containing a binary number of at most h bits and let the keys be h-bit nonnegative numbers. Our results could easily be restated under other specific assumptions, for they are in fact independent of the question of number representation.

To understand our algorithm it is helpful to understand MacLaren's algorithm. It has two stages. In the first stage, the n input keys are bucket sorted according to a subkey composed of the m most significant bits of each key. If extra space is allowed for bucket headers and pointers with which to chain elements in buckets, this stage can be implemented to run in $O(pn + p2^{m/p})$ time for any input, where $p \geq 1$ bucket-sorting passes are employed. The second stage is an insertion sort on the whole keys. Given the initial bucket sort on m-bit subkeys, insertion sort runs in time asymptotically proportional to

$$T(n, m) = \sum_{i=0}^{2^m-1} \text{cost}(r_i)$$

where $r_i$ is the number of keys for which the subkeys composed of the m most significant bits have value i, and cost(y) is the cost to fully sort a sequence of y numbers. Thus, for $i = 0, \ldots, 2^m - 1$,

$$\begin{aligned} \mathbf{E}(T(n,m)) &= \mathbf{E}\left( \sum_{i=0}^{2^m-1} \text{cost}(r_i) \right) \\ &= \sum_{i=0}^{2^m-1} \mathbf{E}(\text{cost}(r_i)) \\ &= 2^m \mathbf{E}(\text{cost}(r_i)), \end{aligned}$$

where $\mathbf{E}(x)$ denotes the expected value of x.

Under the randomness assumption the probability that $r_i = k$ is

$$\binom{n}{k} \left( \frac{1}{2^m} \right)^k \left( 1 - \frac{1}{2^m} \right)^{n-k}$$

for $k = 0, \ldots, n$. If we choose

$$\text{cost}(y) = \begin{cases} 0, & y = 0, \\ 1, & y = 1, \\ y(y-1), & y = 2, \ldots, n, \end{cases}$$

to represent the asymptotic cost of insertion sort, then, for any i,

$$\begin{aligned} &\mathbf{E}(\text{cost}(r_i)) = \\ &= \sum_{k=0}^{n} \text{cost}(k) \binom{n}{k} \left( \frac{1}{2^m} \right)^k \left( 1 - \frac{1}{2^m} \right)^{n-k} \\ &= \frac{n}{2^m} \left( 1 - \frac{1}{2^m} \right)^{n-1} \\ &\quad + \sum_{k=2}^{n} k(k-1) \binom{n}{k} \left( \frac{1}{2^m} \right)^k \left( 1 - \frac{1}{2^m} \right)^{n-k} \\ &= \frac{n}{2^m} \left( 1 - \frac{1}{2^m} \right)^{n-1} \\ &\quad + \frac{n(n-1)}{2^{2m}} \sum_{k=2}^{n} \binom{n-2}{k-2} \left( \frac{1}{2^m} \right)^{k-2} \left( 1 - \frac{1}{2^m} \right)^{n-k} \\ &= \frac{n}{2^m} \left( 1 - \frac{1}{2^m} \right)^{n-1} + \frac{n(n-1)}{2^{2m}}. \end{aligned}$$

Substituting, we find the expected cost of the insertion sort to be

$$\mathbf{E}(T(n, m)) = n \left( 1 - \frac{1}{2^m} \right)^{n-1} + \frac{n(n-1)}{2^m}.$$

To obtain linear running time for this stage, MacLaren sets $m = \log_2 n$ for n a power of two. If we choose $m = \lfloor \log_2 n \rfloor$, then

$$\begin{aligned} &\mathbf{E}(T(n, \lfloor \log_2 n \rfloor)) < \\ &< \frac{4n}{3e^{1/2}} + 2(n-1) = O(n), \quad n > 1. \end{aligned}$$

MacLaren proposed $p = 2$ in stage 1 but it is clear that any constant number of passes $p \geq 1$ will yield an algorithm with O(n) running time overall, provided $m = \Theta(\log n)$. Knuth's modification, alluded to above, is MacLaren's algorithm with $p = 2$, in which the bucket sort is replaced by an address computation sort, thereby eliminating the pointers used to chain keys within buckets. Extra space is still consumed by the $n^{1/2}$ locations needed for bucket headers. The algorithm reported by Dobosiewicz can be viewed as MacLaren's algorithm with $p = 1$, but in which stage 1 of MacLaren's algorithm is applied recursively in place of the stage 2 insertion sort. In addition, a linear-time median-finding algorithm is used to balance the number of elements between the first $\lfloor \frac{1}{2}n \rfloor$ buckets

and the remaining buckets. This latter feature improves the worst case running time to O(n log n) while worsening the expected running time by a constant factor. Van der Nat [13] achieves the same improvement in worst case running time simply by dividing the given file into two subfiles of equal size. Each of the subfiles is bucket sorted as in MacLaren's algorithm. After recursive application of the entire algorithm to each of the buckets, the results on the two subfiles are combined by merging. Meijer and Akl's algorithm [9] is easily understood by viewing it as MacLaren's algorithm in which the insertion sort stage is replaced by a heapsort of each bucket independently. An analysis similar to that given above yields an O(n log n) worst case bound directly. Ehrlich [3] has also treated the subject of sorting numbers.

## 2. Sorting in linear expected time and optimal extra space

Insertion sort requires no more than O(log n) bits of extra space. Therefore, to present our result it is sufficient to consider the problem of sorting n keys on m-bit subkeys, for $m = \lfloor \log_2 n \rfloor$.

For any input **X**, which we may take to be indexed from zero, let I be the set of distinct subkeys in **X**, and let $\hat{\mathbf{X}}$ be **X** in some sorted order on the subkeys. Looking only at the subkeys, $\hat{\mathbf{X}}$ can be described in terms of the start and finish indices of each run of numbers with equal subkeys. Thus, for $j \in I$, we may define

$$s(j) = \min\{i \mid (\text{subkey at } i \text{ in } \hat{\mathbf{X}}) = j\}$$

and

$$f(j) = \max\{i \mid (\text{subkey at } i \text{ in } \hat{\mathbf{X}}) = j\}.$$

It is also convenient to define the number of subkeys equal to j as

$$c(j) = f(j) - s(j) + 1.$$

Our algorithm uses a logarithmically succinct representation for the order of the m-bit subkeys of $\hat{\mathbf{X}}$. This representation is constructed in two n-bit arrays, **ska** (for 'subkey-equals-address') and **sor** (for 'start-of-run' on subkey) where a 1 in **ska**(j) indicates j is a subkey and a 1 in **sor**(j) says that a run of identical subkeys in $\hat{\mathbf{X}}$ begins at j. Concisely stated, **ska** and **sor** represent $\hat{\mathbf{X}}$ when, for $j = 0, \ldots, n-1$,

$$\mathbf{ska}(j) = \begin{cases} 1, & j \in I, \\ 0, & \text{otherwise}, \end{cases} \tag{1}$$

$$\mathbf{sor}(j) = \begin{cases} 1, & s(i) = j \text{ for some } i \in I, \\ 0, & \text{otherwise}. \end{cases} \tag{2}$$

A proof of the sufficiency of this representation is given later.

The first step of our algorithm obtains space for the bit arrays **ska** and **sor** (plus an additional bit per word, to be discussed shortly) by bucket sorting the keys on their three most significant bits. This can be done in linear time and within the space bound required. It leaves **X** arranged so that eight pointers into **X** suffice to identify subsequences that are equal on the first three bits. The eight pointers therefore suffice to record the values of the three most significant bits over the entire array **X**. The algorithm then zeros the first three bits in each key and sorts each of the subsequences delineated by the pointers. When all subsequences are sorted, the algorithm uses the pointers to restore the leading three bits in each key.

Of the bits freed in **X** by the initial bucket sort, let the first be **ska** and the second be **sor**. The third is used to augment the next $m = \lfloor \log_2 n \rfloor$ bits, which comprise the subkey, so that subkeys will be taken to have $m+1$ bits but to range in value from 0 to $2^m - 1 \leq n$. These $m+1$ bits can contain pointers as large as n. In our discussion we denote the $(m+1)$-bit subkey at position j by **k**(j) and the remaining less significant bits at position j by **z**(j).

We have now reduced the problem to sorting $(h-2)$-bit keys ranging in value from 0 to $2^{h-3} - 1$ on $(m+1)$-bit subkeys **k**(j), having bit arrays **ska** and **sor** available as extra space, initialized to zero. The basic idea of the algorithm to solve this problem is to record the sorted order on the subkeys in **ska** and **sor** in the manner just described. Since **ska** encodes I, the subkey fields can be replaced with pointers to the location each key is to occupy in $\hat{\mathbf{X}}$, allowing each key (with pointer in place of subkey) to be moved to its final location. The information

in **ska** and **sor** can then be used to restore the subkeys. We now present the algorithm for this problem in some detail. It is composed of five steps. In our discussion we use superscripts on the input array and its associated fields to indicate the contents of the array or field after a given step. Taking the input of (h − 2)-bit keys as **X**, we thus use $\mathbf{X}^i$ to denote the array after step i.

**Algorithm to sort on subkeys.**

*Step* 1. Encode one instance of each subkey as an address by permuting the input $\mathbf{X}^0$ so that $\mathbf{k}(j) = j$ for each distinct subkey value $j \in I$. Set $\mathbf{ska}(j) = 1$ for each such j.

*Step* 2. Complete recording of some sorted order $\hat{\mathbf{X}}$ as follows: Set $\mathbf{k}(j) = c(j)$ for each distinct subkey value j. Then, for each such j, accumulate these counts so that $\mathbf{k}(j) = s(j)$, and set $\mathbf{sor}(s(j)) = 1$.

*Step* 3. Record the sorting permutation implied by **ska** and **sor** in the subkeys $\mathbf{k}(i) = \pi(i)$, for $i = 0, \ldots, n-1$, for a permutation $\pi$ which takes $\mathbf{X}^1$ to $\hat{\mathbf{X}}$ (that is, $\hat{\mathbf{X}}(i) = \mathbf{X}^1(\pi(i))$ for all i).

*Step* 4. Apply $\pi$ to $\mathbf{X}^3$ by moving **X**(i) to position **k**(i) for all i.

*Step* 5. Restore subkeys using **ska** and **sor** bits.

At this level of presentation, the following observations are sufficient to establish correctness. Step 1 merely permutes $\mathbf{X}^0$. Step 2 modifies the contents of $\mathbf{X}^1$ only in **k**(j) and at values of j where $\mathbf{ska}(j) = 1$ indicates that the contents of **k**(j) are encoded as j. Therefore, sufficient information is retained to restore some permutation of the input. Thus, Steps 3 and 4 can be performed correctly. Under our assumption of the sufficiency of the information in **ska** and **sor**, Step 5 completes the sort on subkeys.

We now give for each step an implementation which runs in linear time and within $\Theta(\log n)$ bits of extra space. We continue our inessential assumption that **X** is indexed from zero.

//**Step 1.** Encode one instance of each subkey as an address by setting $\mathbf{k}(j) = j$ and $\mathbf{ska}(j) = 1$ for each $j \in I$.//

```
for i:=0 to n−1 do
  t:=k(i)
  while k(t)≠t do
    X(i):=X(t)
    ska(t):=1
    t:=k(i)
  endwhile
  ska(t):=1
endfor
```

The only data movements in Step 1 are swaps, so Step 1 maintains a permutation of **X**. A key part of the invariant of the main loop is: each key **k**(j), $0 \leq j < i$, satisfies $\mathbf{k}(j) = \mathbf{k}(\mathbf{k}(j))$. It is true initially (for $i = 0$); it is never falsified because no **X**(j) for which $\mathbf{k}(j) = j$ is ever moved; and each iteration establishes $\mathbf{k}(i) = \mathbf{k}(\mathbf{k}(i))$. Therefore, upon termination each key **k**(j) in the array satisfies $\mathbf{k}(j) = \mathbf{k}(\mathbf{k}(j))$.

Since **X**(j) is never moved for any j once $\mathbf{k}(j) = j$, it follows from Step 1 that $\mathbf{ska}(j) = 1$ if and only if $\mathbf{k}(j) = j$ upon termination. Step 1 runs in linear time because each iteration of the **while** loop increases the number of indices j for which $\mathbf{k}(j) = j$. Finally, no more than $O(\log n)$ bits of extra space are used.

//**Step 2.** Complete recording of the sorted order $\hat{\mathbf{X}}$ by setting $\mathbf{k}(j) = s(j)$ and $\mathbf{sor}(s(j)) = 1$ for each $j \in I$.//

```
for i:=0 to n−1 do
  if ska (i)=1 then k(i):=1
endfor
for i:=0 to n−1 do
  if ska (i)≠1 then k(k(i)):=k(k(i))+1
endfor
t:=n
for i:=n−1 by −1 to 0 do
  if ska (i)=1 then
    t:=t−k(i)
    k(i):=t
    sor(t):=1
  endif
endfor
```

The first two loops of Step 2 set $\mathbf{k}(j) = c(j)$ for $j \in I$. Then the third loop computes the accumulated counts so that $\mathbf{k}(j) = s(j)$ and $\mathbf{sor}(s(j)) = 1$ for $j \in I$. It is evident that the requirements for linear time and $O(\log n)$ extra space are met.

//**Step 3.** Record the sorting permutation implied by **ska** and **sor** by setting **k**(i) = $\pi$(i) for a suitable permutation $\pi$.//

```
for i:=0 to n-1 do
  if ska (i)=0 then
    k(i):=k(k(i))
    k(k(i)):=k(k(i))+1
  endif
endfor
```

Suppose **k**(j) = j for some j. In Step 3, for each $\ell$ statisfying $\ell \neq$ j and **k**($\ell$) = j, a unique index in {s(j),...,f(j) − 1} is assigned to **k**($\ell$) and **k**(j) is increased by 1. This leaves **k**(j) = f(j). Time and space bounds are as required.

//**Step 4.** Apply $\pi$ to $\mathbf{X}^3$ by moving **X**(i) to position **k**(i) for all i.//

```
for i:=0 to n-1 do
  t:=k(i)
  while t≠i do
    z(i):=z(t)
    k(i):=k(t)
    t:=k(i)
  endwhile
endfor
```

An argument similar to that given for Step 1 establishes the correctness of Step 4 and its linear running time. The space bound is obviously met.

//**Step 5.** Restore subkeys using **ska** and **sor** bits.//

```
i:=-1
j:=0
while j<n do
  if sor(j)=1 then
    repeat i:=i+1 until ska(i)=1
  k(j):=i
  j:=j+1
endwhile
```

We note that **ska** and **sor** satisfy properties (1) and (2). The invariant of the outer loop is

$-1 \leqslant i < n \wedge 0 \leqslant j \leqslant n \wedge$
$\wedge$ elements **k**(0:j − 1) have been assigned their final value
$\wedge$ **k**(j − 1) has the value i (unless j = 0 and i = −1).

Because of properties (1) and (2), the value to assign to **k**(j) is either i (if **sor**(j) = 0) or the next greatest integer i′ satisfying **ska**(i′) = 1 (if **sor**(j) = 1). The execution time is clearly linear and the space bound is met.

We thus have the following theorem.

**Theorem.** *It is possible to sort* n *number in* O(n) *expected time and* O(log n) *bits of extra space provided the numbers are independently and identically distributed uniformly and at least* $h = \lfloor \log_2 n \rfloor + c$ *bits are used to represent the input numbers, for any fixed integer constant* c.

**Proof.** For $c \geqslant 2$ a complete algorithm is the following:

*Step* (1) Bucket sort **X** on the three most significant bits.
*Step* (2) Sort each of the subsequences induced by Step (1) using the algorithm described above.
*Step* (3) Insertion-sort the result of Step (2).

Whenever $c < 2$ at most $2^{2-c}$ auxiliary counters are required.

The above theorem can be shown to hold for any h whenever n is divisible by the number of distinct keys. The construction involves grouping equal keys in adjacent locations to expand counting capacity. We omit the details.

In the implementation of the five steps of the algorithm to sort on the subkeys, there are seven loops that pass over the entire array **X**.

The first three can be combined, leaving five loops in a reasonable implementation. While it is possible to reduce the number of bit arrays from two to one, as has been demonstrated by Tornaž Pisanski, this innovation appears to require an algorithm with more passes over the array.

## 3. Conclusion

Our algorithm sorts numbers in linear expected time and using only a constant number of registers of extra space. One barrier to practical use is that the program itself will be more lengthy than the programs for commonly used methods. But, this

question aside, is an implementation possible that is competitive in running time to other methods on files of reasonable size?

To shed light on this question we have made a comparison with Quicksort. Using the best version of Quicksort from the study of this method by Sedgewick [10,11], we implemented both Quicksort and our algorithm in IBM 370 assembler language. For $n \geq 500$ keys, our implementation takes no more than twice as much time as Quicksort on the average, where we measure time as the number of machine instructions executed. We would expect substantially better results on machines with instructions more suited to operate on arbitrary fields of bit within computer words. It is also likely that our assembler language program can be made faster.

It is not likely that our algorithm can be speeded up significantly by replacing insertion sort in the final pass of the algorithm with another sorting method. This judgment follows from observing that $E(r_i > 0)$, the expected length of subsequences that the insertion-sort pass sorts, is small. This quantity is given by

$$E(r_i > 0) = \sum_{k=1}^{n} k \frac{\text{Prob}(r_k = k)}{\text{Prob}(r_i \geq 0)}$$

$$= \sum_{k=1}^{n} k \frac{\binom{n}{k}\left(\frac{1}{2^m}\right)^k\left(1-\frac{1}{2^m}\right)^{n-k}}{1-\binom{n}{0}\left(1-\frac{1}{2^m}\right)^n}$$

$$= \frac{n}{2^m\left(1-\left(1-\frac{1}{2^m}\right)^n\right)},$$

which for $m = \lfloor \log_2 n \rfloor$ is less than 2.313.

Our algorithm is not stable (see [7]). However, neither are Quicksort, Heapsort, Knuth's modification of MacLaren's algorithm and Dobosiewicz's algorithm. It is an open question whether there exists a stable sorting algorithm that runs in $O(n)$ expected time and $O(\log n)$ bits of extra space.

## Acknowledgment

We are grateful to David Gries for helpful comments.

## References

[1] W. Dobosiewicz, Sorting by distributive partitioning, Inform. Process. Lett. 7 (1)(1978) 1–6.

[2] W. Dobosiewicz, The practical significance of d.p. sort revisited, Inform. Process. Lett. 8(4) (1979) 170–172.

[3] G. Ehrlich, Searching and sorting real numbers, J. Algorithms 2 (1982) 1–12.

[4] C.A.R. Hoare, Partition (Algorithm 63); Quicksort (Algorithm 64); and Find (Algorithm 65), Comm. ACM 4(7) (1961) 321–322.

[5] D.B. Johnson and T. Gonzalez, Sorting numbers in linear expected time and constant extra space, Proc. 16th Ann. Allerton Conf. on Comm., Control and Computing (1978) pp. 64–72.

[6] C.A.R. Hoare, Quicksort, Comput. J. 5(4) (1982) 10–15.

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

[8] M.D. MacLaren, Internal sorting by radix plus sifting, J. ACM 13(3) (1966) 404–411.

[9] H. Meijer and S.G. Akl, The design and analysis of a new hybrid sorting algorithm, Inform. Process. Lett. 10 (4,5) (1980) 213–218.

[10] R. Sedgewick, The analysis of quicksort programs, Acta Inform. 7 (1977) 327–355.

[11] R. Sedgewick, Implementing quicksort programs, Comm. ACM 21 (10) (1978) 847–857.

[12] J.W.J. Williams, Algorithm 232: Heapsort, Comm. ACM 7 (6) (1964) 347–348.

[13] M. Van der Nat, A fast sorting algorithm, a hybrid of distributive and merge sorting, Inform. Process. Lett. 10 (3) (1980) 163–167.