P-8

SORTING NUMBERS IN LINEAR EXPECTED TIME AND CONSTANT EXTRA SPACE[†]

DONALD B. JOHNSON and TEOFILO F. GONZALEZ
Computer Science Department
The Pennsylvania State University
University Park, Pennsylvania 16802

ABSTRACT

An algorithm is shown which sorts n numbers in $O(n)$ time on the average where the numbers to be sorted are selected randomly. The algorithm uses only constant extra space, independent of n, and therefore is an advance over known algorithms which sort in linear expected time.

INTRODUCTION

Among the most widely used internal sorting algorithms are quicksort [2,3,4] and heapsort [8,4]. The expected running time of both of these algorithms is $\theta(n\log n)$ (time asymptotically proportional to $n\log n$) when sorting n numbers each permutation of which is equally likely. Quicksort consumes $O(\log n)$ extra space whereas heapsort requires only constant extra space. Both space bounds apply in the worst case.

In 1965, MacLaren [5] presented an algorithm which sorts numbers in $\theta(n)$ time in the expectation when the input numbers are independently and uniformly distributed. We note that this randomness assumption implies each permutation equally likely, but the converse does not hold. One reason, perhaps, that this result has received little attention is that MacLaren's algorithm requires for some fixed integer $p \geq 1$ at least $n + n^{1/p}$ extra locations which in general must be as large as those required for the numbers to be sorted. Variants proposed by others also require asymptotically significant extra space. Knuth's improvement [4] needs at least $\sqrt{n}$ extra locations. Dobosiewicz's recursive version of MacLaren's algorithm [1] requires at least $2n$ extra locations.

Our algorithm sorts n numbers in $\theta(n)$ expected time under the randomness assumption, and it requires only constant extra space independent of n. In addition to these asymptotic results we also discuss evidence below which indicates our algorithm could be implemented to perform slightly faster than quicksort on files of size practical for internal sorting. It has been asserted (see [6], for instance) that quicksort is the best extant sorting algorithm for practical purposes, so an estimate of comparative running times on files of practical size is of some interest.

In order to determine the extra space required by our algorithm it is necessary to assume that the registers, or memory locations, which hold a single input number, or key, have a fixed size which in general is necessary to represent the key. Under this assumption it is assumed impossible to encode additional information in a location containing a key without loss of information in the key. To make this assumption concrete, let each location containing a key be capable of storing 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 assumptions for they are in fact independent of the question of number representation provided arithmetic operations on numbers of magnitude adequate to represent keys are taken to be of unit cost.

[†] This work was partially supported under NSF grant MCS77-21092.

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 comprised 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, it is clear that 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. This stage runs in time asymptotically proportional to

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

where r_i is the number of keys for which the subkeys comprised of the m most significant bits have value i and the cost to fully sort a sequence of y numbers is $\mathrm{cost}(y)$. To reflect the asymptotic cost of insertion sort, we choose

$$\mathrm{cost}(y) = \begin{cases} 0 & y=0\ , \\ 1 & y=1\ , \\ y(y-1) & y=2,\dots,n\ . \end{cases}$$

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 $k=0,\dots,n$. Thus, for $i=0,\dots,2^m-1$,

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

and

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

Substituting,

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

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

$$E(T(n,\lceil \log_2 n \rceil)) \le \frac{4n}{3\sqrt{e}} + n - 1 = O(n), \qquad n>1 .$$

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 $\sqrt{n}$ locations needed for bucket headers. The algorithm reported by Dobosiewicz is MacLaren's algorithm with $p=1$, but it applies stage 1 of MacLaren's algorithm 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 $\lfloor n/2 \rfloor$ 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.

SORTING IN LINEAR EXPECTED TIME AND CONSTANT EXTRA SPACE

Insertion sort requires only constant extra space. Thus to present a linear expected time sorting algorithm which runs in constant extra space it suffices to show how to implement a one-pass bucket sort on large keys, the magnitude of which is of the order of the number of keys, which runs in linear time and no more than constant extra space.

Given an array X of keys, the array being indexed from L to R , the algorithm shown below permutes these keys so that they are sorted in nondecreasing order on the subkeys $k(i)$, which are comprised of $m=\lceil \log_2(R-L+1) \rceil$ bits of $X(i)$ for $i=L,\dots,R$. We assume the following definitions for the variables describing the several fields of key $X(i)$ which the algorithm references. If the key $X(i)$ has h bits numbered from 0 through $h-1$, then for $i=L,\dots,R$,

$$\begin{aligned} code(i) &= X(i)[0,1] \\ k(i) &= X(i)[2,m+1] \\ z(i) &= X(i)[m+2,h] \\ y(i) &= X(i)[2,h] \end{aligned}$$

where $X(i)[a,b]$ for $a\leq b$ denotes the field composed of bits a through b of $X(i)$. See Figure 1. It is assumed that h satisfies $h\geq m+2$. Alternatively we can express $X(i)[a,b]$ as

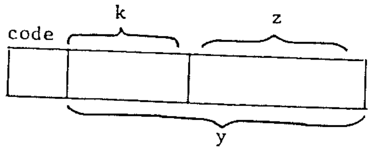

Figure 1. Fields of key X .

$$\left\lfloor \frac{\underline{\text{rem}}(X(i),2^{h-a})}{2^{h-b-1}} \right\rfloor$$

We assume code(i)=0 for i=L,...,R . We will also use a bitwise notation, ie. code(i)='00' . In addition we assume X(i)[2,2] , the leftmost bit in k(i); satisfies X(i)[2,2]=0 , for i=L,...,R , so it is possible to count up to R-L in any k(i) while restricting $|I| \leq R-L+1$ for I = {k(i)| i=L,...,R} , the set of distinct subkeys k . We obtain from arbitrary input X an input satisfying the restrictions on the first three bits of each key by sorting X on X(i)[0,2] for i=L,...,R and recording the positions of the eight subsequences in the resulting X which are uniform on these fields. Thus these fields may be zeroed, to be restored at the end of the computation.

The algorithm which runs under these assumptions is as follows. The phases in which more than one loop traverses X can be condensed to a single loop in each case. We have decomposed these phases to aid in understanding the correctness proof.

```
//Phase 1: For each distinct key, move one instance to corres-
           ponding address, mark and count. //

for i←L to R do
   t ← k(i)+L
   while k(t)≠t-L do
      X(i) ↔ X(t)
      t ← k(i)+L
   endwhile
endfor

for i←L to R do
   if k(i)+L=i then [ code(i) ← '01'
                      k(i) ← 0 ]
endfor

for i←L to R do
   if code(i)='00' then [ k(k(i)+L) ← k(k(i)+L)+1 ]
endfor

//Phase 2: Set initial positions for the distinct keys and mark
           their initial positions. //

t ← R+1-L
for i←R to L by -1 do
   if code(i)='01' then [ t ← t-k(i)-1
                          k(i) ← t ]
endfor

for i←L to R do
   if code(i)='01' or code(i)='11' then [ code(k(i)+L) ←
                                          code(k(i)+L)or'10' ]
endfor

//Phase 3: Set final positions for all keys. //

for i←L to R do
   if code(i)='10' or code(i)='00' then [ k(k(i)+L) ←
                                              k(k(i)+L)+1
                                          k(i) ← k(k(i)+L)-1 ]
endfor
```

```
     //Phase 4:  Rearrange.  //
27   for  $i \leftarrow L$ to R  do
28      while  $k(i) \neq i-L$  do
29         $y(i) \leftrightarrow y(k(i)+L)$
30      endwhile
31   endfor

     //Phase 5:  Substitute subkey values and replace codes.  //
32   $i \leftarrow j \leftarrow L$
33   loop
34      while  code(i)='00' or code(i)='10'  do
35         $i \leftarrow i+1$
36      endwhile
37      repeat
38         $k(j) \leftarrow i-L$
39         $j \leftarrow j+1$
40         if  $j>R$  then  [ exit loop ]
41      until  code(j)='10' or code(j)='11'
42      $i \leftarrow i+1$
43   forever
44   for  $i \leftarrow L$ to R  do
45      code(i) $\leftarrow$ '00'
46   endfor
```

Before proving correctness of our algorithm shown above we state the following definitions. Let $\hat{X}$ be any sorted order of X. Then, for $j \varepsilon I$, ,

$$s(j) = \min_{L \leq i \leq R} \{i-L \mid \hat{k}(i)=j\} \ ,$$

$$f(j) = \max_{L \leq i \leq R} \{i-L \mid \hat{k}(i)=j\} \ , \text{ and}$$

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

Furthermore, let X^0 be a copy of X just before phase 1 is executed.

LEMMA. Let X^0 and X be as defined above. There exist π_1 and π_2, permutations of $\{0,\ldots,R-L\}$, for which the following conditions hold after execution of phases 1 through 5:

(i) $X(\pi_2(\pi_1(j))+L) = X^0(j+L)$, $j=0,\ldots,R-L$, and

(ii) $k(j+L) \leq k(j+L+1)$, $j=0,\ldots,R-L-1$.

Furthermore, the algorithm terminates following phase 5.

Proof: It can be verified that all phases terminate. Therefore let X^1 be a copy of the program variable X following termination of the loop ending at line 7. Since the only operations on X are exchanges of elements,

$$X^1(\pi_1(j)+L) = X^0(j+L) \ , \qquad j=0,\ldots,R-L \ ,$$

where π_1 is a permutation of $\{0,\ldots,R-L\}$. It can be seen that X^1 satisfies $k^1(j+L)=j$ for $j \varepsilon I$. Thus, to complete the proof that the algorithm correctly sorts the input it remains to be shown that there exists, upon termination of phase 5, a permutation π_2 of $\{0,\ldots,R-L\}$ for which

(i) $X(\pi_2(j)+L) = X^1(j+L)$, $j=0,\ldots,R-L$, and

(ii) $k(j+L) \leq k(j+L+1)$, $j=0,\ldots,R-L-1$.

The remainder of phase 1 sets code(j+L)='01' and $k(j+L)=0$ for $j\varepsilon I$ in lines 8 through 11 and then counts distinct keys in $k(j+L)$ for $j\varepsilon I$ yielding these conditions when phase 1 terminates:

$$z(j+L) = z^1(j+L) , \quad j=0,\ldots,R-L ,$$

$$k(j+L) = \begin{cases} c(j)-1 , & j\varepsilon I , \\ k^1(j+L) , & \text{otherwise, and} \end{cases}$$

$$\text{code}(j+L) = \begin{cases} \text{'01'} , & j\varepsilon I , \\ \text{'00'} , & \text{otherwise .} \end{cases}$$

It can be verified that the information retained in X is sufficient to restore X^1 and therefore to complete the sort.

In phase 2, the loop comprised of lines 15 through 19 sets $k(j+L)$ to $s(j)$ for $j\varepsilon I$. The loop at lines 20 through 22 sets code(s(j)+L) to code(s(j)+L) <u>or</u> '10' for $j\varepsilon I$. Consequently X^2, the result of phase 2, has the properties

$$z^2(j+L) = z^1(j+L) , \quad j=0,\ldots,R-L ,$$

$$k^2(j+L) = \begin{cases} s(j) , & j\varepsilon I , \\ k^1(j+L) , & \text{otherwise, and} \end{cases}$$

$$\text{code}^2(j+L) = \begin{cases} \text{'q1'} , & \text{iff } j\varepsilon I , \\ \text{'1q'} , & \text{iff } j=s(\ell) \text{ for some } \ell\varepsilon I , \end{cases}$$

where $q\varepsilon\{0,1\}$. Again it can be verified that X contains information sufficient to restore X^1 .

The assignments in phase 3 are executed for those values of i for which $i-L\not\varepsilon I$. The effect is to set $k(j+L)$ to $\pi_2(j)$, for $j=0,\ldots,R-L$, for some π_2 which will sort X^1 . Thus each field $k(j+L)$ will encode the position of $X^1(j+L)$ in the final sort. We support this assertion by considering one fixed value $t\varepsilon I$. Since there are $f(t)-s(t)$ distinct values of i for which $i-L\not\varepsilon I$ and $k^1(i)=t$, there will occur a sequence of assignments induced by these values, which we denote $\{i_1,\ldots,i_{f(t)-s(t)}\}$, as follows.

$$\begin{aligned} k(t+L) &\leftarrow k(t+L)+1 = s(t)+1 \\ k(i_1) &\leftarrow k(t+L)-1 = s(t) \\ k(t+L) &\leftarrow k(t+L)+1 = s(t)+2 \\ k(i_2) &\leftarrow k(t+L)-1 = s(t)+1 \\ &\vdots \\ k(t+L) &\leftarrow k(t+L)+1 = f(t) \\ k(i_{f(t)-s(t)}) &\leftarrow k(t+L)-1 = f(t)-1 \end{aligned}$$

where $\{X^1(i_1),\ldots,X^1(i_{f(t)-s(t)}),X^1(t+L)\}$ is precisely the set of keys which must be in positions $s(t)+L$ through $f(t)+L$ in any sorted order $\hat{X}$ of X^1. Furthermore this sequence of assignments references locations disjoint from the locations referenced for any $t' \neq t$, $t' \varepsilon I$. Thus it can be seen that the sequence $k^3(L),\ldots,k^3(R)$ records a suitable permutation π_2 where X^3 is a copy of X after phase 3 is completed.

With this understood, the operation of phase 4 is clear. It simply applies π_2 to y^3 (X^3 exclusive of the code fields). The code fields still record the entire structure of $\hat{k}(j)$ for $j=L,\ldots,R$ for any sorted order $\hat{X}$ of X^1 (or X^0 for that matter). For X after phase 4, then, these facts may be stated precisely as

$$z(\pi_2(j)+L) = z^1(j+L)\ , \qquad j=0,\ldots,R-L\ ,$$

$$\text{code}(j+L) = \begin{cases} \text{'q1'}\ , & \text{iff } j \varepsilon I\ , \\ \text{'1q'}\ , & \text{iff } j=s(\ell) \text{ for some } \ell \varepsilon I\ , \end{cases}$$

where $q\varepsilon\{0,1\}$. The values in k are now immaterial since the code fields for the entire array encode what each k value should be.

Phase 5 completes the k fields by extracting this information. Index i is advanced until $i=\min_{\ell \varepsilon I}\{\ell+L\}$, indicating that $i-L$ is the first distinct subkey k in X^0. Index j is then incremented from $s(i-L)$ to $f(i-L)$ in order to store $i-L$ as needed. Repetition for each $i-L \varepsilon I$ yields, for X after line 43,

$$z(\pi_2(j)+L) = z^1(j+L)\ , \qquad j=0,\ldots,R-L\ ,$$

$$k(\pi_2(j)+L) = k^1(j+L)\ , \qquad j=0,\ldots,R-L\ , \text{ and}$$

$$k(j+L) \leq k(j+L+1)\ , \qquad j=0,\ldots,R-L-1\ .$$

When phase 5 resets all code fields to '00' we then have

$$X(\pi_2(j)+L) = X^1(j+L)\ , \text{ or}$$

$$X(\pi_2(\pi_1(j)+L)) = X^0(j+L)\ , \qquad j=0,\ldots,R-L\ , \text{ and}$$

$$k(j+L) \leq k(j+L+1)\ , \qquad j=0,\ldots,R-L-1\ ,$$

as required. □

THEOREM. It is possible to sort n numbers in $O(n)$ expected time and constant extra space provided the numbers are identically and uniformly distributed and at least $h=\lceil\log_2 n\rceil+c$ bits are required to represent the numbers, for any fixed constant c.

Proof: For $c \geq 2$ a complete algorithm is:

(1) Bucket sort X on three most significant bits.

(2) Sort each subsequence of X produced in stage (1) using the algorithm described above.

(3) Insertion sort X.

It is well known how to implement stage (1) to run in $O(n)$ time and to use only constant extra space. It has been established in the lemma given above that stage (2) can be performed using only constant extra space. The procedure exhibited for stage (2) can easily be seen to run in $O(n)$ time when it is noticed that there can be no more than n iterations in total for each of the while loops at lines 3 and 28. Given the analysis in the introduction, it is clear that stage (3) runs in $O(n)$ expected time. Of course, it is well known how to implement insertion sort to use only constant extra space.

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. Of course, the theorem holds when h is a constant in any event.

CONCLUSION

We do not consider whether this algorithm is practical in the sense that it should be used in data processing. However, the recent analysis of quicksort by Sedgewick [6,7] invites an attempt to estimate at what value of n does $L(n)=Q(n)$ where $L(n)$ is the running time of our linear time algorithm and $Q(n)$ is the running time of quicksort. Certainly if this n were very large, our tacit assumption that our algorithm can reasonably be called an internal sort would be brought into question.

Sedgewick employs a computational model which essentially counts memory references on an idealized computer in some implementation at the memory reference level. Although our algorithm is susceptible to a similar analysis, we have taken a different approach. Using the second paper cited above we have produced an implementation of quicksort on the IBM 370 which appears to admit only minor improvements. Then the task has been to implement our algorithm as well as we can and call the n at which the running times are equal, an estimate for an upper bound on the crossover. For simplicity we choose the number of machine instructions executed as our time measure.

We have an implementation of a more complicated algorithm, a predecessor of the one presented in the previous section, for which our crossover estimate is $n=200{,}000$. Preliminary results on this new version indicate that a value of n between 20,000 and 50,000 can easily be demonstrated above which $L(n)<Q(n)$.

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

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

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

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

which for $m=\lceil \log_2 n\rceil$ evaluates asymptotically to less than or equal to

$$\frac{1}{1-\frac{1}{e}} = \frac{e}{e-1} \simeq 1.58 \ .$$

It is easy to see that our algorithm is not stable (see [4]). However, neither are quicksort, heapsort, Knuth's modification of MacLaren's algorithm, nor Dobosiewicz's algorithm. It is an open question whether there exists a stable sorting algorithm which runs in $O(n)$ expected time and constant extra space.

REFERENCES

1. W. Dobosiewicz, Sorting by distributive partitioning, Inf. Proc. Letters 7, 1(Jan. 1978) 1-6.

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

3. C. A. R. Hoare, Quicksort, Computer J. 5, 4(April 1962) 10-15.

4. D. E. Knuth, The Art of Computer Programming, Vol. 3: Sorting and Searching, Addison-Wesley, 1973.

5. M. D. MacLaren, Internal sorting by radix plus sifting, J. ACM 13, 3 (July 1966) 404-411.

6. R. Sedgewick, The analysis of quicksort programs, Acta Informatica 7 (1977) 327-355.

7. R. Sedgewick, Implementing quicksort programs, Tech. Report No. CS-38, Brown University (March 1977), (to appear Comm. ACM).

8. J. W. J. Williams, Algorithm 232: Heapsort, Comm. ACM 7, 6(1964) 347-348.