A TIME-SPACE TRADEOFF FOR
IN-PLACE ARRAY PERMUTATION

by
Robert Melville

TR 79-385

Department of Computer Science
Cornell University
Ithaca, New York 14853

A time-space tradeoff for in-place array permutation

Let $a[1:n]$ be a one dimensional array of elements of some type T . Let f be a permutation of the integers between 1 and n . We consider the problem of permuting a according to f , i.e., letting A_k denote the initial contents of $a[k]$, we want to set $a[1] = A_{f(1)}$, set $a[2] = A_{f(2)}$, ..., set $a[n] = A_{f(n)}$. This can be described neatly as the concurrent assignment:

$$a[1], a[2], \dots, a[n] := a[f(1)], a[f(2)], \dots, a[f(n)]$$

Of course, the problem is easy if a second array $b[1:n]$ is available. However, the use of a second array may be inconvenient:

- (1) for large n or complicated T this may require too much extra storage;
- (2) in a programming language that does not allow allocation of dynamically dimensioned arrays, it may be difficult to write a subroutine able to permute arrays of different lengths.

We say that an array permutation algorithm operates "in-place" if the only operation modifying the array is $\text{swap}(i, j)$, which interchanges the contents of locations i and j . By an index variable we mean an integer variable restricted to the range $[0:n+1]$. Note that such a variable requires $O(\log n)$ bits of storage.

An in place permutation algorithm is not hard to develop if we are allowed a boolean array $\text{mark}[1:n+1]$. Define:

$$\begin{aligned} \text{cycle}(i) &=_{\text{def}} \text{cycle}_f(i) =_{\text{def}} \\ &\{i, f(i), f^2(i), \dots, f^{s-1}(i)\} \quad \text{where } f^s(i) = i. \end{aligned}$$

A basic fact about permutations is that such a unique s exists for any i in the domain and two cycles are either the same or disjoint.

Algorithm 1 below permutes the array by tracing the cycle structure of f . The locations in $\text{cycle}(i)$ are permuted by the operations:

$$\text{swap}(i, f(i)); \text{swap}(f(i), f^2(i)); \dots; \text{swap}(f^{s-1}(i), i)$$

where $f^s(i) = i$. The mark array is used to indicate the locations that have been permuted. An informal invariant for the outer loop would be:

(I1) ($\underline{A} i \ 1 \leq i \leq n: \text{mark}[i] \Rightarrow \text{cycle}(i) \text{ has been permuted, i.e.}$
 $\quad \underline{A} j \ j \in \text{cycle}(i): a[j] = A_{f(j)}$
 $\quad \neg \text{mark}[i] \Rightarrow \text{cycle}(i) \text{ unchanged, i.e.}$
 $\quad \underline{A} j \ j \in \text{cycle}(i): a[j] = A_j$)

Algorithm 1

```
(A j 1 ≤ j ≤ n+1: mark[j] := false);
k := 1;
do mark[k] → k := k+1
[] ¬ mark[k] and k ≠ n+1 →
  {permute and mark cycle(k)}
  j , fj := k , f(k);
  do fj ≠ k →
    swap( j , fj );
    mark[fj] := true;
    j , fj := fj , f( fj )
  od;
  mark[k] := true
od
```

od

Algorithm 1 runs in time $O(n)$ and requires extra space $n + O(\log n) = O(n)$ bits.

Another algorithm requires no extra storage except for index variables, i.e. $O(\log n)$ bits of extra storage. Algorithm 2 given below is discussed more fully in [1]. We give only an informal invariant:

Algorithm 1 below permutes the array by tracing the cycle structure of f . The locations in $\text{cycle}(i)$ are permuted by the operations:

$$\text{swap}(i, f(i)); \text{swap}(f(i), f^2(i)); \dots; \text{swap}(f^{s-1}(i), i)$$

where $f^s(i) = i$. The mark array is used to indicate the locations that have been permuted. An informal invariant for the outer loop would be:

(I1) ($\underline{A} i \ 1 \leq i \leq n; \text{mark}[i] \Rightarrow \text{cycle}(i) \text{ has been permuted, i.e.}$
 $\quad \underline{A} j \ j \in \text{cycle}(i); a[j] = A_{f(j)}$
 $\neg \text{mark}[i] \Rightarrow \text{cycle}(i) \text{ unchanged, i.e.}$
 $\quad \underline{A} j \ j \in \text{cycle}(i); a[j] = A_j$)

Algorithm 1

($\underline{A} j \ 1 \leq j \leq n+1; \text{mark}[j] := \text{false}$);
 $k := 1$;

do $\text{mark}[k] \rightarrow k := k+1$

$\square \neg \text{mark}[k] \text{ and } k \neq n+1 \rightarrow$

{permute and mark $\text{cycle}(k)$ }

$j, fj := k, f(k)$;

do $fj \neq k \rightarrow$

$\text{swap}(j, fj)$;

$\text{mark}[fj] := \text{true}$;

$j, fj := fj, f(fj)$

od;

$\text{mark}[k] := \text{true}$

od

Algorithm 1 runs in time $O(n)$ and requires extra space $n + O(\log n) = O(n)$ bits.

Another algorithm requires no extra storage except for index variables, i.e. $O(\log n)$ bits of extra storage. Algorithm 2 given below is discussed more fully in [1]. We give only an informal invariant:

(I3.2) I3.1 and $k \leq kt$ and $\neg \text{mark}[kt]$ and

$(\forall i \ k \leq i < kt: \text{mark}[i] \Rightarrow \text{cycle}(i) \text{ permuted})$.

What makes the coding a bit tricky is that the implication in I3.2 cannot be reversed. If a location j is unmarked we cannot conclude that location j is unpermuted. The algorithm traces the sequence $k, f(k), f^2(k), \dots$ until encountering a marked location or a location with index $< k$. The invariant for the innermost loop is:

(I3.3) $(\forall i \ k \leq i \leq kt: \text{mark}[k] \Rightarrow \text{cycle}(i) \text{ permuted } \underline{\text{or}} \ i \in \text{cycle}(k))$.

Cycles are disjoint, so if $\text{cycle}(k)$ is currently unpermuted the sequence $k, f(k), f^2(k), \dots$ must return to k without encountering a marked location or a location $< k$. If the sequence returns to k the required swaps are performed to permute the new cycle.

Algorithm 3

$k := 1; \{I3.1\}$

do $k \neq n+1 \rightarrow$

$\{I3.1 \text{ and } k \leq n\}$

$kt := \min(k+t, n+1);$

allocate array $\text{mark}[k:kt];$

$(\underline{\text{A}}p \ k \leq p \leq kt: \text{mark}[p] := \text{false});$

$\{I3.2\}$

do $\text{mark}[k] \rightarrow k := k+1 \ \{I3.2\}$

$\square \neg \text{mark}[k] \text{ and } k \neq kt \rightarrow$

$\{I3.2 \text{ and } \neg \text{mark}[k] \text{ and } k < kt\}$

$j := k; \{I3.3\}$

do $kt \leq j \rightarrow j := f(j)$

$\square \ k \leq j < kt \text{ and } \neg \text{mark}[j] \rightarrow$

$j, \text{mark}[j] := f(j), \text{true}$

od;

$\{I3.3 \text{ and } (k \neq j \Rightarrow \text{cycle}(k) \text{ has been permuted})$
 $\text{and } (k = j \Rightarrow \text{cycle}(k) \text{ unpermuted})\}$

if $k \neq j \rightarrow \text{skip}$

$\square \ k = j \rightarrow \text{do } f(j) \neq k \rightarrow \text{swap}(j, f(j));$

$j := f(j) \text{ od}$

fi

$\{I3.2 \text{ and } \text{mark}[k] \text{ and } k < kt\}$

od $\{k = kt\}$

od

To show the claimed running time we must show that execution of the middle loop costs at most $O(n)$. All iterations of the first guarded command

" $\text{mark}[k] \rightarrow k := k+1$ "

costs at most $O(t(n)) \leq O(n)$.

We evaluate the cost of the iterations of the second guarded command by "charging" parts of the computation to cycle members. Suppose a portion of a cycle is traced first by the second guarded command starting at some point k_0 . Since each point of the cycle has a unique predecessor, retracing the same portion from another starting point k_1 would mean passing through k_0 . This is impossible since location k_0 would be marked. Since a portion of a cycle is never traced twice by different iterations of the second guarded command, all iterations of this command can cost no more than the sum of all cycle lengths. This is $O(n)$. Finally, the outermost loop iterates $\lfloor n/t \rfloor$ times, so the total running time is $\lfloor n/t \rfloor O(n) = O(n^2/t(n))$.

Finally, we offer an alternative to algorithm 2 obtained from algorithm 3 by substituting $t = 1$:

Algorithm 2'

```

k := 1;
do k ≠ n+1 →
    j := f( k );
    do j > k → j := f( j ) od;
    if j < k → skip {cycle(k) already permuted}
    || j = k → do f( j ) ≠ k → swap( j , f( j ) );
                j := f( j ) od
    fi;
    k := k + 1

```

od

with invariant

(I2') ($\underline{A}p \ 1 \leq p < k$: cycle(p) is permuted
and
 any permuted cycle has a member $\leq k$).

This algorithm seems easier to understand than algorithm 2 above.

Acknowledgement

Alan Demers helped with some early discussion of this problem. The nicely coded form of algorithm 3 incorporates numerous suggestions from David Gries and Gary Levin.

References

1. Duijvestijn, A. J. , "Correstness Proof of an In-Place Permutation", BIT 12 (1972), Pages 318-324.