McLaren's Masterpiece[1]

by

D. Gries and J.F. Prins
Computer Science Department
Cornell University
January 1986

TR 86-729

Introduction

Consider writing an algorithm for the following problem. Given is an array $b(0..n-1)$, where $n \geq 0$. Given also is a sequence H that is a permutation of $0..n-1$. Array b is to be rearranged into the order specified by H, i.e. the following multiple assignment is to be executed:

(0) $b.0, b.1, ..., b.(n-1) := b.(H.0), b.(H.1), ..., b.(H.(n-1))$

In executing the algorithm, extra space of only $O(1)$ is to be used, but H may be destroyed.

Remark on notation. We view arrays and sequences as functions and use the period "." as an infix function-application operator. Also, juxtaposition is used for catenation of sequences and sequence elements. Finally, for sequences X and Y, by $X \subseteq Y$ we mean "every element of X is also an element of Y". □

H is actually represented using a simple variable p and a second array $c(0..n-1)$: H and c, p are coupled by the representation invariant

(1) $H.0 = p$, and
for each element r of H, sequence X, and nonempty sequence Y satisfying $H = X\ r\ Y$, $c.r$ is the *follower* of r, i.e. first element of Y.

For example, the sequence $H = (3, 2, 1, 4, 0)$ is represented in p and c as $p = 3$, $c.3 = 2$, $c.2 = 1$, $c.1 = 4$, $c.4 = 0$, and $c.0$ any value; i.e. p and c give a linked-list representation of sequence H. *Any* sequence of distinct elements from $0..n-1$ can be represented in p and c using representation invariant (1).

[1] This research was supported by the NSF under grant DCR-8320274.

An in-situ linear algorithm for this problem was invented by Donald McLaren in the 1960s; it is presented in guarded-command notation in (2) below, where its structure appears startlingly simple. It appears as an exercise in Knuth [3], but Knuth's description of McLaren's Masterpiece is difficult to understand. McLaren's Masterpiece was proved correct in [1], a paper on the multiple assignment statement; however, the algorithm was still not satisfactorily described.

(2) $k := 0;$
do $k \neq n \rightarrow$
  do $p < k \rightarrow p := c.p$ **od**;
  $b.k, b.p := b.p, b.k;$
  $p, c.p, c.k := c.p, c.k, p;$
  $k := k+1$
od

This note is an attempt to give a better description of algorithm (2). The new twist is the use of "thought" variable H and an initial description of the algorithm in terms of b and H, together with a simple "coordinate translation" from H into variables p, c. (The term "thought variable" was introduced by W.H.J. Feijen and A.J.M. van Gasteren, and the same technique was used in [0]).

The use of H reduces the amount of formal manipulation that must be performed, particularly with array subscripts, and makes the algorithm easier to understand. For the same reason, it aids in algorithmic development, especially when one restricts attention to manipulations of H that are efficiently implementable using p, c.

For example, suppose $H = r\ Y\ k\ Z$, where k and r are elements and Y and Z sequences of elements. The assignment $H := Y\ k\ r\ Z$ is easily implementable in constant time in terms of p, c as follows. We have, in the initial state,

$p = r$,
$c.r =$ first element of $Y\ k$,
$c.k =$ first element of Z (immaterial if Z is empty) .

After execution of $H := Y\ k\ r\ Z$ we have

$p =$ first element of $Y\ k$,
$c.r =$ first element of Z (immaterial if Z is empty),
$c.k = r$.

In addition, all the followers of elements of Y and Z remain unchanged. Since initially $p = r$, the assignment is implemented by

(3) $p, c.p, c.k := c.p, c.k, p$.

However, in the same situation, the assignment $H := k\ Y\ r\ Z$ takes time proportional to the length of Y because the follower of the last element of Y must be changed, and one must find this last element by searching through Y.

Throughout, we assume that H consists of distinct elements only.

We begin by giving two other descriptions of the problem. Using $\circ$ for function composition, we can write assignment (0) as $b := b \circ H$. And we specify the algorithm in terms of a precondition Q and postcondition R, with B denoting the ***final*** value of b, as follows:

Q: $0 \leq n \ \wedge \ perm.(H, 0..n-1) \ \wedge \ B = b \circ H$
R: $B = b$

where $perm.(s, t)$ means "sequence s is a permutation of sequence t". It is this specification that we use in describing the algorithm.

Notation and a simple lemma

The algorithm will swap two elements $b.r$ and $b.k$ of array b. (for some values r and k). This will require a corresponding swap of values of sequence H. In order to understand these swaps, we introduce some notation and a simple lemma. Let $(b; k{:}e)$ denote a function that is the same as b except that at argument k its value is e. Also, $(b; k.e; j{:}f) = ((b; k.e); j.f)$. Thus, $(b; r{:}b.k; k{:}b.r)$ denotes b with elements $b.r$ and $b.k$ swapped. See [1], [2] for an introduction to this notation and the concept of arrays as functions.

(4) **Lemma.** Suppose $r \neq k$, $(X\ r\ Y\ k\ Z) \subseteq 0..n-1$, and $r, k \notin X, Y, Z$. Let $\bar{b} = (b; r{:}b.k; k{:}b.r)$. Then

$$b \circ (X\ r\ Y\ k\ Z) = \bar{b} \circ (X\ k\ Y\ r\ Z)$$

Proof.
$b \circ (X\ r\ Y\ k\ Z)$
$= (b \circ X)\ b.r\ (b \circ Y)\ b.k\ (b \circ Z)$
$=$ {since $r, k \notin X$, $b \circ X = \bar{b} \circ X$; similarly for Y, Z}
$(\bar{b} \circ X)\ \bar{b}.k\ (\bar{b} \circ Y)\ \bar{b}.r\ (\bar{b} \circ Z)$
$= \bar{b} \circ (X\ k\ Y\ r\ Z)$ □

The algorithm in terms of b and H

It is clear that either iteration or recursion is needed in the algorithm, and we decide on iteration. Since initially the value $b.(H.0)$ belongs in $b.0$, the first iteration will probably execute $b.0 := b.(H.0)$, and we surmise that each iteration k of the loop will store a final value in $b.k$. A first approximation to the loop invariant is found in the standard manner by finding a suitable generalization of R and Q (with the help of a fresh variable k):

$P0'$: $0 \leq k \leq n$
$P1'(k, H)$: $perm(H, k..n-1)$
$P2'(k, H)$: $B = b \circ ((0..k-1)\ H)$

$P2'$ indicates that the first k values are in their final position and that H shows how the rest of the values in b are to be permuted —in exactly the same way that the initial value of H shows how the initial values of b are to be permuted.

A first approximation to the algorithm is then written:

$k := 0$;
do $k \neq n \rightarrow$ Change b and H to establish $P1'.(k+1, H) \ \wedge \ P2'.(k+1, H)$;
$k := k+1$
od

We investigate how to change b and H within the loop body. If $k = H.0$, then $P1'.(k+1, H) \ \wedge \ P2'.(k+1, H)$ can be established simply by deleting k from H, so let us look at the harder case, $k \neq H.0$.

Suppose $k \neq H.0$. since k is in H, we can write $H = r\ Y\ k\ Z$, for some element r and sequences Y and Z. This means that we have $B.k = b.(H.k)$, and it seems reasonable to consider swapping $b.k$ and $b.r$ to establish $B.k = b.k$. In this situation, lemma (4) forces consideration of the assignment $H := k\ Y\ r\ Z$. Immediately, we recognizes the inefficiency of this statement (in terms of p, c) and look for alternatives. The one that comes to mind is $H := Y\ k\ r\ Z$, since, by our earlier discussion, it can be implemented in terms of p, c in constant time. However, then we won't be able to delete k from H. Can we modify the invariant so that the occurrence of k in H can be tolerated? This can be done, for example, by changing $P2'$ to

$$B = b \circ (0..k-1\ (H \mid k..n-1))$$

where $X \mid Y$ denotes the subsequence of X found by deleting from X all elements not in Y (read "sequence X restricted to Y"). Thus, we allow values less than k in H but just disregard them. The full invariant is changed to

(5) $P0$: $\quad 0 \leq k \leq n$
$P1(k, H)$: $\quad k..n-1 \subseteq H \subseteq 0..n-1$
$P2(k, H)$: $\quad B = b \circ (0..k-1\ (H \mid k..n-1))$

We modify the algorithm to take into account the change in the invariant. The one important change concerns the first element of H; at each iteration, it may now be less than k. Hence, the first step of the loop body should be to delete such initial elements of H that are less than k, using, say a loop

do $H.0 < k \rightarrow H := H.(1..)$ **od** .

Note that this loop is executed only when H contains at least one element in $k..n-1$, so it terminates, and with $H.0 \geq k$. Also, it does not falsify invariant (5). This modification leads to the following algorithm:

(6) $k := 0$;
$\{$*invariant*: $P0 \wedge P1.(k, H) \wedge P2.(k, H)\}$
do $k \neq n \rightarrow$
    do $H.0 < k \rightarrow H := H.(1..)$ **od**;
    $\{P0 \wedge P1.(k, H) \wedge P2.(k, H) \wedge k \leq H.0 < n\}$
    Change b and H to establish $P1.(k+1, H) \wedge P2.(k+1, H)$;
    $k := k+1$
od

It is easy to verify that the proof outline is correct, and the only remaining step is, again, to refine the statement "Change b and H ...". Our earlier discussion leads directly to the refinement

(7) "Change b and H to establish $P1.(k+1, H) \wedge P2.(k+1, H)$":

Let r, X satisfy $H = r\ X$;
if $r = k \rightarrow H := X$
$[]\ r \neq k \rightarrow$ Let Y, Z satisfy $X = Y\ k\ Z$;
$b.k,\ b.r := b.r, b.k$;
$H := Y\ k\ r\ Z$
fi

We now verify that this implementation is correct. Statement (7) is executed when H has at least one element. Write $H = r\ X$ for some element r and sequence X and consider two cases: $k = r$ and $k \neq r$.

Case r = k. This means that $b.k$ already contains its final value, and deleting the value k from the beginning of H establishes the result. This the refinement does, since in this case $H := X$ is executed. More formally, we have:

$$
\begin{array}{ll}
 & P0 \wedge P1.(k, r\ X) \wedge P2.(k, r\ X) \wedge r = k < n \\
\Rightarrow & k..n-1 \subseteq (k\ X) \subseteq 0..n-1 \wedge B = b \circ ((0..k-1)\ (k\ X \mid\mid k..n-1)) \\
= & k+1..n-1 \subseteq X \subseteq 0..n-1 \wedge B = b \circ ((0..k)\ (X \mid\mid k+1..n-1)) \\
= & wp(\text{“}H := X\text{”}, P1.(k+1, H) \wedge P2.(k+1, H))
\end{array}
$$

Case r ≠ k. We show that execution of

(8) $b.k,\ b.r := b.r, b.k;\ H := Y\ k\ r\ Z$

establishes $P1.(k+1, H)$ and $P2.(k+1, H)$. First, consider establishing $P1.(k+1, H)$. Under the condition $0 \leq k < r < n$, we have

$$
\begin{array}{ll}
 & P1.(k,\ r\ Y\ k\ Z) \\
= & k..n-1 \subseteq r\ Y\ k\ Z \subseteq 0..n-1 \\
= & k..n-1 \subseteq Y\ k\ r\ Z \subseteq 0..n-1 \\
= & k+1..n-1 \subseteq Y\ k\ r\ Z \subseteq 0..n-1 \\
\Rightarrow & wp(\text{“}b.k, b.r := b.r, b.k;\ H := Y\ k\ r\ Z\text{”}, P1.(k+1, H))
\end{array}
$$

We now prove that execution of (8) under the condition $0 \leq k < r < n$ establishes $P2.(k+1, H)$. Here, we will rely on lemma (4).

$$
\begin{array}{ll}
 & P0 \wedge P2.(k, r\ Y\ k\ Z) \wedge k < r < n \\
\Rightarrow & B = b \circ ((0..k-1)\ ((r\ Y\ k\ Z) \mid\mid k..n-1)) \\
= & \{\text{by lemma (4)}\} \\
 & B = \overline{b} \circ ((0..k-1)\ (k\ Y\ r\ Z) \mid\mid k..n-1) \\
\Rightarrow & B = \overline{b} \circ ((0..k)\ ((Y\ r\ Z) \mid\mid k+1..n-1)) \\
\Rightarrow & B = \overline{b} \circ ((0..k)\ ((Y\ k\ r\ Z) \mid\mid k+1..n-1)) \\
= & wp(\text{“}b.k, b.r := b.r, b.k;\ H := Y\ k\ r\ Z\text{”}, P1.(k+1, H))
\end{array}
$$

Hence, the implementation, and algorithm (6), is correct.

The coordinate transformation from H to p,c

Our final step is to translate the operations on H into operations on p and c, the variables used to implement H, thus transforming algorithm (6) with refinement (7) into algorithm (2). The transformation relies on representation invariant (1).

(a) $H.0$ is replaced by p .
(b) $H := H.(1..)$ and $H := X$ are replaced by $p := c.p$.
(c) A reference r to the first value of H is replaced by p .
(d) The statement $H := Y\ k\ r\ Z$ is translated into the multiple assignment (3). Justification for this occurs in the paragraph surrounding (3).

This yields an algorithm similar to (2), but with a conditional statement in the loop body. Look at algorithm (6). In the case $r \neq k$, the refinement of "Change b and H ..." is translated into the following by the coordinate transformation:

(9) $\quad b.k,\ b.p := b.p, b.k;\quad p, c.p, c.k := c.p, c.k, p$

We now argue that (9) can also be used in the case $r = k$, which means that (9), by itself, can be used as implementation (7) in terms of variables p and c. This yields algorithm (2).

Here is the argument. The statement $H := X$ that is guarded by $r = k$ is transformed into $p := c.p$. Since $r = k = p$, a swap of $b.r$ and $b.k$ has no effect, so we can write this as

$$b.k,\ b.p := b.p, b.k;\quad p := c.p$$

Finally, since $r = k = p \notin X = H.(1..)$, changing $c.p$ (and thus $c.k$) does not change followers of elements of X, so $c.p$ can be changed without altering the relation between the final values of H and c, p.

Linearity of the algorithm

Note that each operation on H is performed in constant time in terms of p, c. Each execution of the inner loop reduces the length of sequence H by one element and no operation on H increases it, so the body of the inner loop can be executed at most n times. Hence, the algorithm is takes time proportional to n, the length of b.

Acknowledgements

Thanks go to David Rossiter for constructive comments on various drafts of this note.

References

[0] Feijen, W.H.J., A.J.M. van Gasteren, and D. Gries. In-situ inversion of a cyclic permutation. Tech. Rpt., Computer Science Department, Cornell University, 1985. (submitted for publication in IPL).

[1] Gries, D. The multiple assignment statement. *IEEE Trans. Software Eng 4* (March 1978), 87-93.

[2] Gries, D. *The Science of Programming.* Springer Verlag, New York, 1981.

[3] Knuth, D.E. *The Art of Computer Programming, vol. 3.* Addison-Wesley, Reading, MA, 1973.