SWAPPING SECTIONS*

David Gries[1]
Harlan Mills[2]

TR 81-452
January 1981

[1]Department of Computer Science, Cornell University, Ithaca NY 14853

[2]IBM Corporation, 18100 Frederick Pike, Gaithersburg MD 20760

*This work was partially supported under NSF grant MCS76-22360

Swapping Sections

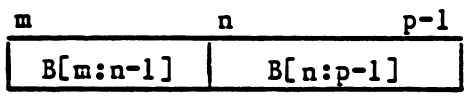
David Gries, Cornell University[+]
Harlan Mills, IBM

January 1981

1. Introduction

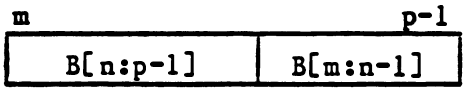
Given are fixed integer variables m, n and p satisfying $m < n < p$. Given is (part of) an array, b[m:p-1], considered as two sections:

(1.1)



where B denotes the initial value of array b —i.e. b[m] = B[m], b[m+1] = B[m+1], ..., b[p-1] = B[p-1]. The problem is to swap (interchange) the two sections, using only a constant amount of extra space, thus establishing the predicate

(1.2) R:



We present three different algorithms for this problem. They all execute in linear (in p-m) time and, indeed, their execution times are close enough that any of the three could be used. But the algorithms are so different and interesting that we thought it worthwhile to describe them. We also generalize the solution to swap non-adjacent sections.

2. Reversal swap

It is easy enough to write an algorithm that reverses a section b[i:j] (i.e. changes $b[i:j] = (B_i, B_{i+1}, \cdots, B_j)$ into $b[i:j] = (B_j, B_{j-1}, \cdots, B_i)$. The algorithm is given in (2.2); the invariant of the loop is

[+]This work partially supported under NSF grant MCS76-22360.

(2.1) $i \le f \le h+1 \le j+1 \ \wedge$ b

i … f	f … h	h … j
already reversed	not reversed	already reversed

(2.2) Reverse(b, i, j):
{Reverse the values in b[i:j], where $i \le j+1$}
var f, h: integer ;
{$i \le j \ \wedge$ b[i:j] = $(B_i, \cdots, B_j)$}
f, h:= i, j;
do f < h → b[f], b[h], f, h:= b[h], b[f], f+1, h-1 **od**
{b[i:j] = $(B_j, B_{j-1}, \cdots, B_i)$}

Note that execution of the loop halts when the unreversed section contains less than two elements; the reverse of a 1-element section is that section.

But then execution of algorithm (2.3) swaps b[m:n-1] and b[n:p-1]! This algorithm was shown to us by Alan Demers; it is used in the text editor on the TERAK on which this report is begin written and edited.

(2.3) Reverse(b, m, n-1); Reverse(b, n, p-1); Reverse(b, m, p-1).

3. Successive swap

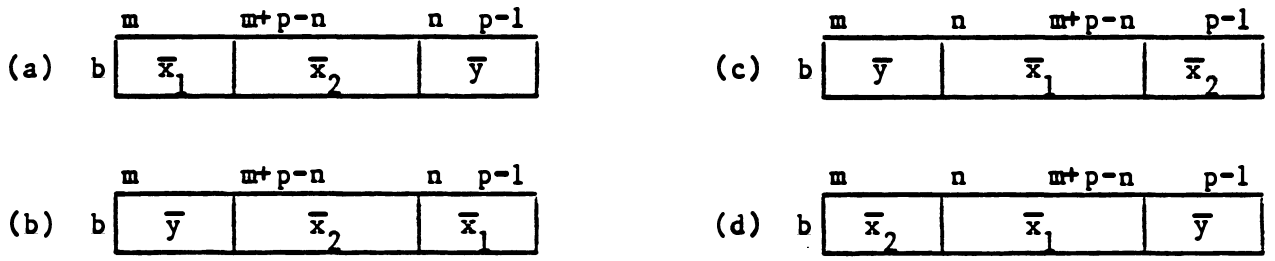
One can swap two, non-overlapping, array sections b[i:i+k-1] and b[j:j+k-1] of the same length k by first swapping b[i] and b[j], then b[i+1] and b[j+1], and so forth. Using again identifier B to denote the initial value of array b, the invariant of the loop of the algorithm is:

(3.1) P: $0 \le t \le k \ \wedge$
b[i:i+t-1] = B[j:j+t-1] $\wedge$ b[j:j+t-1] = B[i:i+t-1] $\wedge$
b[i+t:i+k-1] = B[i+t:i+k-1] $\wedge$ b[j+t:j+k-1] = B[j+t:j+k-1]

Informally, the invariant indicates that the two subsections b[i:i+t-1] and b[j:j+t-1] have been swapped and the rest of the sections still have to be swapped. The algorithm is then

(3.2) Swapequals(b, i, j, k):
{Swap non-overlapping sections b[i:i+k-1] and b[j:j+k-1]}
var t: integer;
t:= 0; {P}
do $t \neq k$ → b[i+t], b[j+t], t:= b[j+t], b[i+t], t+1 **od**

Now consider the original problem of swapping b[m:n-1] and b[n:p-1], and suppose for the moment that the second section, b[n:p-1], is smaller. Consider the first section as consisting of a sequence $\bar{x}_1$ of length p-n followed by a sequence $\bar{x}_2$, and the second section as consisting of a sequence $\bar{y}$. Then execution of Swapequals(b, m, n, p-n) causes the initial state described by (a) in the diagram below to be transformed into the state described by (b). Thus, execution of Swapequals(b, m, n, p-n) causes p-n elements to be put into place, and to establish the desired result it remains only to swap the sections b[m+p-n:n-1] and b[n:p-1]. Similarly, if section b[n:p-1] is larger than b[m:n-1], Swapequals can be used to transform (c) in the diagram into (d).



This gives us the idea for the following algorithm. Note in the transformations that n remains the beginning of the rightmost section to be swapped. Use variables i and j to denote the lengths of the sections still to be swapped, use the informal loop invariant

(3.3) P: $0 < i \leq n-m \wedge 0 < j \leq p-n \wedge$ b

m	n-i	n	n+j p-1
already swapped	swap with b[n:n+j-1]	swap with b[n-i:n-1]	already swapped

and write the algorithm as follows:

```
(3.4)  var i, j: integer ;
       i, j:= n-m, p-n;
       do i ≠ j → if i > j → Swapequals(b, n-i, n, j); i:= i-j
                  □ i < j → Swapequals(b, n-i, n+j-i, i); j:= j-i
                  fi
       od;
       {P ∧ i = j}
       Swapequals(b, n-i, n, i)
```

4. The Dolphin algorithm

Consider again the original problem. Suppose the value b[m] is placed in a variable x. Then the value $b[a_1]$ (say) whose destination is b[m] can be moved into b[m]. This frees $b[a_1]$, so that the value $b[a_2]$ (say) whose destination is $b[a_1]$ can be moved to $b[a_1]$. This frees $b[a_2]$, etc. We hope that a sequence of such moves will end up with a free element $b[a_k]$ in which x belongs, and that after executing $b[a_k]$:= x all elements will have been moved to their final destinations.

The first step in deriving the algorithm is to analyze the sequence of indices a_i, which defines precisely the destination of each array value. Let r and s be the sizes of the two partitions: r = n-m and s = p-n. Then we rewrite the result assertion R as

(4.1) R: $(\forall i: 0 \le i < s: b[m+i] = B[m+i+r]) \wedge$
$(\forall i: s \le i < r+s: b[m+i] = B[m+i-s])$

But this we can simplify to

(4.2) R: $(\forall i: 0 \le i < r+s: b[m+i] = B[m + i+r \bmod r+s])$.

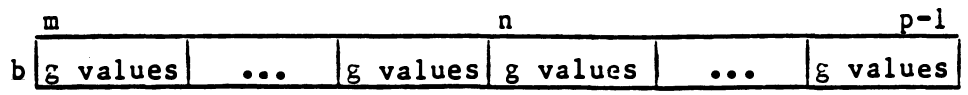
Here is a well-known fact. The sequence of r+s values

(4.3) 0, r mod r+s, 2r mod r+s, ..., (r+s-1)r mod r+s

contains exactly the set {0, 1, ..., r+s-1} if and only if r and r+s, and hence r and s, are relatively prime. Furthermore, the next value in the sequence, (r+s)r mod r+s, is 0. Hence, if and only if r and s are relatively prime will the following sequence of assignments establish the desired result:

```
      x:= b[m];
      b[m]:= b[m + r mod r+s];
(4.4) b[m + r mod r+s]:= b[m + 2r mod r+s];
     ...;
      b[m + (r+s-2)r mod r+s]:= b[m + (r+s-1)r mod r+s];
      b[m + (r+s-1)r mod r+s]:= x .
```

What if r and s are not relatively prime? Let g be the greatest common divisor of r and s: g = gcd(n-m, p-n). Let $\bar{r} = r/g$ and $\bar{s} = s/g$. Note that $\bar{r}$ and $\bar{s}$ are relatively prime. Consider b[m:p-1] to be divided into blocks of g values each:



We see that sequence (4.3) can be written for each relative place j, $0 \le j < g$, in these blocks as

(4.5) $j+0,\ j+(\overline{r} \bmod \overline{r}+\overline{s}),\ j+(2\overline{r} \bmod \overline{r}+\overline{s}),\ \ldots,\ j+((\overline{r}+\overline{s}-1)\overline{r} \bmod \overline{r}+\overline{s})$.

That is, execution of (4.4) with $\overline{r},\overline{s}$ in place of r, s moves the first value of each block to its final destination. Similarly, execution of the same sequence, but with m replaced by m+1, will move the second element of each block to its final destination. This idea leads us to algorithm (4.6). Note that, in the algorithm, the sequence of subscript values is determined using the form given in (4.1), which is equivalent to but simpler to calculate with (in this context) than (4.2).

In the algorithm, r, s and g are as defined above. The invariant of the outer loop of the algorithm is:

P: $r = n-m \ \wedge\ s = p-n \ \wedge\ g = \gcd(r, s) \ \wedge\ 0 \le j \le g \ \wedge$
the first j elements of each block contain their final values $\wedge$
the last g-j elements of each block contain their initial values.

The purpose of the inner loop is to move the $j+1^{st}$ element of each block to its final destination, using the technique described above. We don't give the full, formal invariant, but simply state partial information:

P1: $k = m+j+(i+r \bmod r+s)$ for some i $\wedge$
$t = m+j+(k+r \bmod r+s) \ \wedge$
$x = b[m+j] \ \wedge$
the next step is to move b[t] to its final destination b[k]

and that the sequence of values that k takes on is $m+j,\ m+j+r \bmod r+s,\ m+j+2r \bmod r+s,\ \ldots,\ m+j+(r+s-1)r \bmod r+s$.

The idea for this algorithm was shown to the authors by Ted Nelson in Tokyo at the IFIP 80 Congress. Nelson had had the idea for some time, but was unable to bring the idea to fruition. So the authors had an enjoyable time on the plane from Tokyo to Hong Kong solving it. Nelson called it the Dolphin algorithm, because, to him, the movement of the array values looked like dolphins leaping out of the water and disappearing again at random places. Later, a search of the literature discovered a similar algorithm by William

(4.6) Dolphin(b, m, n, p):

```
var r, s, g, j, k, t: integer ;
r, s:= n-m, p-n;  g:= gcd(r, s);
j:= 0; {P}
do j < g → Move the j+1st value of each block to its destination:
            k:= m+j; t:= k+r; x:= b[k];
            do t ≠ m+j →
                b[k], k:= b[t], t;
                if k < m+s → t:= k+r
                 □ k ≥ m+s → t:= k-s
                fi
            od;
            b[k]:= x;
          j:= j+1 od
```

Fletcher (Algorithm 284, Interchange of two blocks of data, CACM 9 (May 1966), 326). Fletcher's is different in that it requires swaps instead of assignments, swapping blocks of g values at a time.

5. Partial analysis of execution times

Reversal swap requires 1 swap (i.e. execution of b[f], b[h]:= b[h], b[f]) per array element. In general, successive swap requires fewer swaps. To see this, first delete all calls on Swapequals —the result, (5.1), is a well-known algorithm to compute the greatest commmon divisor (gcd) of the section sizes. (It is nice to see this little gem cropping up in a useful algorithm!)

(5.1)
```
{n-m > 0 ∧ p-n > 0}
i, j:= n-m, p-n;
do i ≠ j → if i > j → i:= i-j □ i < j → j:= j-i fi
od
{i = j = gcd(n-m, p-n)}
```

Each swap in successive swap, except the last gcd(n-m, p-n) swaps, places one value in its final position, and each of the last gcd(n-m, p-n) swaps places two values in position. Thus, the total number of swaps is p-m - gcd(n-m, p-n).

The Dolphin algorithm performs no swaps; it performs p-m + gcd(n-m, p-n) assignments involving array elements. To compare the work performed by the

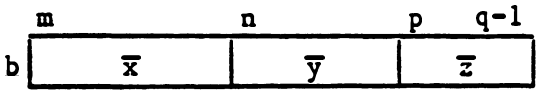
three algorithms, let us assume that a swap will be carried out in its usual manner by three assignments. The number of array assignments for each algorithm is then: 3(p-m) for reversal swap, 3(p-m) - 3gcd(n-m, p-n) for successive swap, and p-m + gcd(n-m, p-n) for Dolphin. Clearly, reversal swap is the worst with respect to array assignments. Successive swap does as badly as reversal swap when the section sizes are relatively prime, for example when one section has 1 value in it, but in general it performs fewer array assignments than reversal swap. Dolphin is best, doing as badly as successive swap only when the section sizes are equal.

But array assignment is not the whole story; there is overhead in testing, adding and subtracting, and computing subscript values. Here, the analysis is harder, first because the algorithms can be modified slightly to save some additions and second because the time needed for some subscript calculations and tests may depend heavily on the computer and implementation of the algorithm.

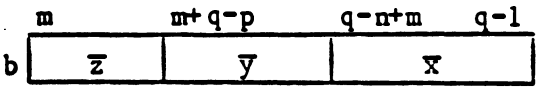
We leave a complete analysis to the reader, and just say the following. Reversal swap requires a fixed number of tests (p-m) and additions (2(p-m)). The number of tests and additions for successive-swap depends on how <u>small</u> gcd(r, s) is —the smaller the gcd, the more tests and additions it uses. The Dolphin algorithm is the opposite case: the larger the gcd, the more tests and additions are required.

6. <u>Swapping non-adjacent sections</u>

Consider the problem of swapping non-adjacent sections. Given are integers m, n, p and q satisfying $m < n \leq p < q$ and an array section b[m:q-1], arranged in three sections:



(where $\bar{x}$, $\bar{y}$ and $\bar{z}$ are vectors of values). Sections b[m:n-1] and b[p:q-1] should be swapped so that the following is established:



This problem can be solved by swapping the first two sections and then

swapping them as a unit with the third section. For example, Dolphin can be used in two different ways to perform the non-adjacent swap:

(6.1) Dolphin(b, m, n, p); Dolphin(b, m, p, q) and

(6.2) Dolphin(b, n, p, q); Dolphin(b, m, n, q)

Algorithms (6.1) and (6.2) require the following number of array assignments, respectively:

(6.3) p-m + gcd(n-m, p-n) + q-m + gcd(p-m, q-p) and

(6.4) q-n + gcd(p-n, q-p) + q-m + gcd(n-m, q-n).

An algorithm can easily be written that executes either (6.1) or (6.2), depending on which of (6.3) and (6.4) is the smaller; we leave this to the reader.

Although reversal swap is the least efficient for swapping adjacent sections, it does have a more efficient generalization to non-adjacent sections. The non-adjacent sections can be swapped by reversing all three sections and then reversing them as a unit:

```
(6.5) Reverse(b, m, n-1);
      Reverse(b, n, p-1);
      Reverse(b, p, q-1);
      Reverse(b, m, q-1)
```

This requires q-m array element swaps, or 3*(q-m) array element assignments. From (6.3) and (6.4), we see that the Dolphin approach ((6.1) or (6.2)) requires the following number of array element assignments:

```
q-m + min(p-m + gcd(n-m, p-n) + gcd(p-m, q-p)),
          q-n + gcd(p-n, q-p) + gcd(n-m, q-n)).
```

In the worst case, when all the sections are of equal length, this reduces to (7/3)*(q-m) array element assignments, which is slightly better than 3*(q-m) for (6.5). But the Dolphin approach does have more overhead.