On a gauntlet thrown by David Gries.

by

Edsger W.Dijkstra

It is requested to design a program that will generate the N! permutations of the values from 0 through N-1 in such an order that the transition from one permutation to the next is always performed by exactly one swap of two neighbours.

In a permutation each pair of values such that the larger value precedes the smaller one, presents a so-called "inversion". (In particular: the one and only permutation with zero inversions is the one in which the values are placed in monotonically increasing order.) The notion of inversions can be expected to be relevant because the swapping of two neighbours changes the total number of inversions by (plus or minus) 1 , and it is, therefore, suggested to characterize each permutation by its inversions. This can be done by introducing N inversion counters $inv[i]$ for $0 \leq i < N$, where $inv[i]$ equals the number of inversions between the value i and the values smaller than i . (From this definition $0 \leq inv[i] \leq i$ follows; the total number of inversions of a permutation is the sum of the corresponding $inv[i]$-values.) That each permutation defines the $inv[i]$-values uniquely is obvious; that the $inv[i]$-values define the permutation uniquely is easily seen by considering the algorithm constructing the permutation from the $inv[i]$-values --processing these values in the order of increasing i -- : this algorithm leaves us no choice.

There is, therefore, a one-to-one correspondence between the N! possible inv-values and the N! permutations, and the question becomes, which modifications of the inv-value correspond to a swap of neighbours: each swap of two neighbours changes exactly one $inv[i]$-value by 1 , viz. with i = the larger of the two values swapped. The value of $inv[i]$ is to be increased if the swap increases the number of inversions; otherwise it is to be decreased.

A feasible sequence of inv-values to be generated is now reasonably obvious: it is the generalization of the Gray-code. For N = 4 it would begin

inv[0]	inv[1]	inv[2]	inv[3]
0	0	0	0
0	0	0	1
0	0	0	2
0	0	0	3
0	0	1	3
0	0	1	2
0	0	1	1
0	0	1	0
0	0	2	0
0	0	2	1
0	0	2	2
0	0	2	3
0	1	2	3
0	1	2	2

etc.

The rule is as follows: a number is changeable when it may be increased or decreased by 1 . It may be increased if the sum of the numbers to its left is even and it has not reached its maximum value; it may be decreased if the sum of the numbers to its left is odd and it has not reached its minimum value zero. At each step, always the right-most changeable number is changed. It is not difficult to see that in the permutation, the value i is, indeed, swapped with a smaller value.

After having established the value i , such that $inv[i]$ has to be changed, and, also, whether the value i has to be swapped with its predecessor in the permutation (corresponding to an increase of $inv[i]$) or with its successor in the permutation (corresponding to a decrease of $inv[i]$) , we have to establish the place c in the permutation, where the value i is located because for all $j > i$, $inv[j]$ has an extreme value, c is given by

$c = i - inv[i] +$ (the number of values j such that $j > i$ <u>and</u> $inv[j] = j$)

In the following program we have given $inv[0]$ --which should be constantly 0-- the funny value -2; this is the usual, mean, little coding trick, in order to let the search for the right-most changeable $inv[i]$-value terminate ar-

tificially when there is no more such an inv[i]-value. The value "totinv" records the total number of inversions in the array a , that is used to record the permutation; the variable "linv" records the sum of the (non-funny) inv[j]-values to the left of inv[i] (i.e. with $j < i$).

```
begin integer array a, inv [0:N-1]; boolean ready; integer totinv, i, c, linv;
i:= 0; do i < N → a[i]:= i; inv[i]:= 0; i:= i + 1 od; inv[0]:= - 2;
ready:= false; totinv:= 0;
do non ready →  printarray(a);
      i:= N - 1; c:= 0; linv:= totinv - inv[i];
      do inv[i] = i and even(linv) →
            c:= c + 1; i:= i - 1; linv:= linv - inv[i]
       [] inv[i] = 0 and odd(linv) →
            i:= i - 1; linv:= linv - inv[i]
      od;
      c:= c + i - inv[i];
      if even(linv) and i > 0 →
            inv[i]:= inv[i] + 1; totinv:= totinv + 1; swap(a, c-1, c)
       [] odd(linv) and i > 0 →
            inv[i]:= inv[i] - 1; totinv:= totinv - 1; swap(a, c, c+1)
       [] i = 0 → ready:= true
      fi
od
end
```

Post Scriptum.

The problem solved above has been posed to me by David Gries, who told me that he had found it a non-trivial task to present a solution to it in a convincing manner. The argument shown led quickly to the above solution, which is submitted for publication upon his request. (End of Post Scriptum.)

Burroughs
Plataanstraat 5
NUENEN - 4565
The Netherlands

prof.dr.Edsger W.Dijkstra
Burroughs Research Fellow