

14 THE PROBLEM OF THE DUTCH NATIONAL FLAG

There is a row of buckets numbered from 1 through N . It is given that

$P1$: each bucket contains one pebble

$P2$: each pebble is either red, white, or blue.

A mini-computer is placed in front of this row of buckets and has to be programmed in such a way that it will rearrange (if necessary) the pebbles in the order of the Dutch national flag, i.e. in order from low to high bucket number first the red, then the white, and finally the blue pebbles. In order to be able to do so, the mini-computer has been equipped with one output command that enables it to interfere with pebble positions, viz.

“buck:swap(i, j)” for $1 \leq i \leq N$ and $1 \leq j \leq N$:

for $i = j$: the pebbles are left as they are

for $i \neq j$: two computer-controlled hands pick up the pebbles from buckets $nr. i$ and j respectively and then drop them in each other's bucket respectively. (This operation leaves relations $P1$ and $P2$ invariantly true.)

and one input command that can inspect the colour of a pebble, viz.

“buck(i)”

for $1 \leq i \leq N$:

when the computer program prescribes the evaluation of this function of type “colour”, a movable “eye” is directed upon bucket $nr. i$, and delivers to the mini-computer as the value of the function the colour (i.e. red, white, or blue) of the pebble currently lying in the bucket, the contents of which is inspected by the “eye”.

The constant N is a global constant from the context in which our program is to be embedded as an inner block. Our program, however, has to meet three special requirements:

1. It must be able to cope with all possible forms of "degeneration" as presented by missing colours: the buckets may have been filled with pebbles of two colours only, of one colour only, or of no colour at all (if $N = 0$).
2. The mini-computer has a very small store compared with the values of N it should be able to cope with, and therefore we are not allowed to introduce arrays of any sort, only a fixed number of variables of the types "integer" and/or variables of type "colour". (With variables of type integer we mean here variables that cannot take on much more than N different values.)
3. The program may direct the "eye" at most once upon each pebble (it is assumed that the input operation is so time-consuming that looking twice at the same pebble would lead to an unacceptable loss of time).

Furthermore, regarding programs of the same degree of complication, the one that needs (on the average) the fewer swaps is to be preferred.

Although our pebbles are of only three different colours, the fact that our eye can only inspect pebbles one at a time, together with requirement (3), implies that halfway through the rearrangement process, we have to distinguish between pebbles of four different categories, viz. "established red" (ER), "established white" (EW), "established blue" (EB), and "as yet uninspected" (X). Requirement (2) excludes that pebbles of these different categories lie arbitrarily mixed: inside the mini-computer we then cannot store "who is what". Our only way out is to divide the row of buckets into a fixed number of (possibly empty) zones of consecutively numbered buckets, each zone being reserved for pebbles of a specific category. Because four different zones is the minimum, the introduction of just four zones seems the first thing to try. But in what order? I found that many programmers tend to decide without much thinking upon the order "ER", "EW", "EB", "X", but this is a rash decision. As soon as anyone is of the opinion that it is attractive to place the zone "ER" at the low end, considerations of symmetry should suggest that the zone "EB" at the high end is equally attractive. Still sticking to our earlier decision of only four different zones, we come to the conclusion that the zones "EW" and "X" should be in the middle in some order (convince yourself that it is now immaterial in which order!), for instance:

"ER", "X", "EW", "EB"

Once we have chosen the above "general situation", our problem is essentially solved, for here we have a general situation of which both the

initial state (all buckets in zone “X”) and the final state (zone “X” empty) are special cases! We can establish it, and then a repetitive construct has to decrease the size of zone “X” while maintaining this general situation. In our mini-computer we need three integer variables for keeping track of the place of the zone boundaries, e.g. “ r ”, “ w ”, and “ b ” with the meanings

$l \leq k < r$:	the k th bucket is in zone “ER” (number of buckets $r - l \geq 0$)
$r \leq k \leq w$:	the k th bucket is in zone “X” (number of buckets $w - r + l \geq 0$)
$w < k \leq b$:	the k th bucket is in zone “EW” (number of buckets $b - w \geq 0$)
$b < k \leq N$:	the k th bucket is in zone “EB” (number of buckets $N - b \geq 0$)

Establishing the relation P to be kept invariant means initializing these three variables in accordance with “all buckets in zone “X””, and the overall structure of our program could be:

```

begin glovar buck; glocon N; privar r, w, b;
  r vir int, w vir int, b vir int := l, N, N; {P has been established}
  do  $w \geq r \rightarrow$  “decrease number of buckets in zone “X”
    under invariance of  $P$ ”
  od
end

```

Immediately we are faced with the question: by which amount shall the guarded statement decrease the number of buckets in zone “X”? There are three arguments—and as the reader will notice, they are of a fairly general nature—in favour of trying first whether we can come away with “decrease the number of buckets in zone “X” by l ”. The arguments are:

1. Decreasing by l is sufficient.
2. As we have chosen our guard “ $w \geq r$ ”, we can guarantee the presence of at most one bucket in zone “X”; for two, we would have needed the guard “ $w > r$ ”.
3. The one pebble inspected will face us with three different cases, inspecting two confronts us already with nine different cases; this multiplicative building up of cases to be considered should be interpreted, in principle, as a heavy price to pay for whatever we can gain by it.

The next question to be settled is: which one of the uninspected pebbles will be looked at? This question is not necessarily irrelevant, because in the meantime in the ordering “ER”, “X”, “EW”, “EB” an asymmetric situation

has been created. No experienced programmer will suggest an arbitrary one, the ones at the low and the high end respectively are the most likely candidates. With equal probabilities for the three colours, an inspection of the pebble in the r th bucket will give rise to $(0 + 1 + 2)/3 = 1$ swap; inspection of the pebble in the w th bucket, however, will give rise only to $(1 + 0 + 1)/3 = 2/3$ swap, and this settles the choice. Thus we arrive at the following program:

```

begin glovar buck; glocon N; privar r, w, b;
  r vir int, w vir int, b vir int := 1, N, N;
  do  $w \geq r \rightarrow$ 
    begin glovar buck, r, w, b; pricon col;
      col vir colour := buck(w);
      if col = red  $\rightarrow$  buck:swap(r, w); r := r + 1
         $\parallel$  col = white  $\rightarrow$  w := w - 1
         $\parallel$  col = blue  $\rightarrow$  buck:swap(w, b); w, b := w - 1, b - 1
      fi
    end
  od
end

```

Note. The program is robust in the sense that it will lead to abortion when fed with erroneous data such as one of the pebbles being green. (*End of note.*)

In the case that all pebbles are red and no swaps are necessary, our program will prescribe N swaps and as conscious programmers we should investigate how complicated a possibly more refined solution becomes: perhaps we have acted too cowardly in rejecting it. (As a general strategy I would recommend not to try the more refined solution before having constructed the more straightforward one; that strategy gives us, besides a working program, an inexpensive indication of what the considered refinement as such has to compete with.) I have always thought the above solution perfectly satisfactory, and up till now I have never considered a more complicated one. So, here we go!

Inspecting just one can be extended to "inspecting one or two" or "inspecting as many as we can conveniently place". In view of the case "all pebbles red" something along the latter line seems indicated. Before inspecting the uninspected pebble at the high end we could try to move the boundary indicated by r to the high end as much as we possibly can without swapping, because it seems a pity to replace a red pebble in a perfectly OK-position by another red pebble. The outer repetition could then begin with

```

do  $w \geq r \rightarrow$ 
  begin glovar buck, r, w, b; privar colr;
    colr vir colour := buck(r);
  do colr = red and  $r < w \rightarrow r := r + 1$ ; colr := buck(r) od;

```

The inner repetition stops, either because all pebbles have been inspected ($r = w$) or because we have hit a nonred pebble. The case $r = w$, where *colr* may have one of three different values, reduces to the alternative clause of the earlier program but for the fact that the “*buck*:*swap*(*r*, *w*)” — *r* and *w* being equal to each other— can be omitted. The case $r < w$ implies *colr* $\neq$ *red*; now we *must* be willing to inspect another pebble, for otherwise our solution reduces to the one which always inspects the pebble at the low end of the zone “X” and of that solution we know that on the average it generates more swaps than our first effort. Because $r < w$, there is indeed another uninspected pebble and the one in the *w*th bucket is the obvious candidate.

Again, with *colr* = *white*, it seems a pity to swap the pebble in the *r*th bucket with a white one in the *w*th bucket and in case of $r < w$ it seems indicated to enter a new inner block

```

begin glovar buck, r, w, b; glocon colr; privar colw;
  colw vir colour := buck(w);
  do colw = white and  $w > r + 1 \rightarrow w := w - 1$ ; colw := buck(w) od;

```

We have now for *colr* the two possibilities *white* or *blue*, for *colw* the three possibilities *red*, *white*, or *blue*, and the set of uninspected pebbles may be empty or not. For a moment I feared that I might have to distinguish between about 12 cases! But after looking at it for a long time (and after one false start), I discovered that the way to proceed now is first to place the pebble now in the *w*th bucket and to see to it that in all three cases the pebble originally in the *r*th bucket is left in the (new) *w*th bucket. Then the three alternatives can merge and a single text deals uniformly with the second pebble, the colour of which is still given by *colr*.

```

if colw = red  $\rightarrow$  buck:swap(r, w); r := r + 1
  || colw = white  $\rightarrow w := w - 1$ 
  || colw = blue  $\rightarrow$  buck:swap(w, b); w, b := w - 1, b - 1;
    buck:swap(r, w)
fi;
if colr = white  $\rightarrow w := w - 1$ 
  || colr = blue  $\rightarrow$  buck:swap(w, b); w, b := w - 1, b - 1
fi

```

Note 1. It is nice that we could achieve that the concatenation of two alternative constructs caters to the $2 * 3$ colour-combinations! (*End of note 1.*)

Note 2. Convince yourself that the case $colw = white$ has been correctly dealt with, because in this case $w = r + 1$ initially. (*End of note 2.*)

Remark. For pedagogical reasons I slightly regret that the final treatment of “two inspected pebbles in wrong buckets” did not turn out to be worse; perhaps I should have resisted the temptation to do even this messy job as decently as I could. (*End of remark.*)

I leave the final composition of our second program to the reader, if he still so desires (I hope he does not!); I think the point has been made. I have carried the case-analysis up to this stage in order to drive home the message that such a case-analysis, also known under the name “combinatorial explosion”, should nearly always be avoided like the plague. It lengthens the program texts, and that may easily impair the efficiency! It always impairs the program’s reliability, for it imposes upon the poor programmer’s shoulders a burden under which he usually succumbs (not in the last place because the work becomes so boring). The reader may interpret the above exercise as a hint to software managers not to grade their programmers by the number of lines of code produced per month; I would rather let them pay the punched cards they use out of their own pocket.

I have solved this problem often with students, either arriving at zones in the order “ER”, “X”, “EW”, “EB” or in the order “ER”, “EW”, “X”, “EB”. When asked which pebble to inspect, their first suggestion had always been “the leftmost one”. I had the idea that this preference could be traced to our habit of reading from left to right. Later I encountered students that suggested first the rightmost one: one was an Israeli computing scientist, the other one was of Syrian origin. It is somewhat frightening to discover the devious ways in which our habits influence our thinking!

And this concludes my treatment of the problem of the Dutch national flag, a problem that I owe to W.H.J. Feijen.