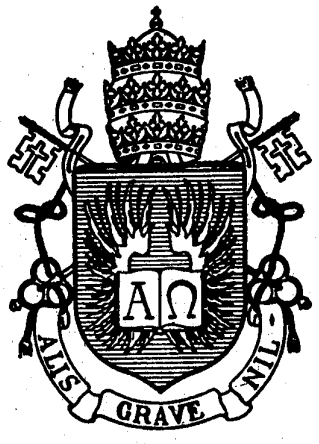

PUC

Séries: Monografias em Ciência da Computação

Nº 12/78

A PUZZLE

by

Jacques J. Arsac

Departamento de Informática

Pontifícia Universidade Católica do Rio de Janeiro
Rua Marquês de São Vicente, 225 – CEP-22453
Rio de Janeiro – Brasil

Series: Monografias em Ciência da Computação
Nº: 12/78

Series Editor: Michael F. Challis June, 1978

A PUZZLE

by

Jacques J. Arsac**

* This work has been partially sponsored by FINEP and was carried out while the author was visiting the Department, in September 1977.

**Université de Paris.

For copies contact:

Rosane T.L.Castilho
Head, Setor de Documentação e Informação
Depto. de Informática - PUC/RJ
Rua Marques de São Vicente, 209 - Gávea
22453 - Rio de Janeiro - RJ - Brasil

Contents

ABSTRACT:

The use of some programming techniques is illustrated by the development of a program to solve a puzzle. The method used consists of getting an initial solution (a recursive one in the present case) and applying some transformations to the program to improve the solution according to a given goal. Program transformations seem to be a major way to reduce the role of invention in programming.

KEY WORDS:

Program proving, recursion, program transformations, program correctness

RESUMO:

A utilização de algumas técnicas de programação é ilustrada através do desenvolvimento de um programa para resolver um quebra-cabeça. O método usado consiste na obtenção de uma solução inicial, recursiva neste caso, e na aplicação de algumas transformações ao programa para melhorar a solução de acordo com um objetivo dado. Achamos que transformações de programas é uma técnica importante para reduzir a parte inventiva da programação.

PALAVRA CHAVE:

Prova de programas, recursividade, transformações de programas, verificação de programas.

A PUZZLE

Jacques J. Arsac

1 - The Game

The game is made of a set of identical pieces, numbered from 1 to n. Each piece may be put in two different positions, say up and down. There are locks which prevent pieces from being moved. Pieces are made in such a way that:

Setting piece p down locks piece p+1
Setting piece p up unlocks piece p+1, but locks all the pieces following p+1

Piece 1 is never locked.

Problem ON. All the pieces being in the state down, find a sequence of moves which sets them up

Problem OFF - All the pieces being in the state up, find a sequence of moves which sets them down

The state of piece p will be represented by s(p). move(p) is the action changing the state of piece p (assuming that it is not locked)

2 - Recursive Solution

The piece p may be moved if it is not locked; this will be so if and only if the piece (p-1) is up (otherwise, it locks the piece p) and all the pieces before p-1 are down (if one of them is up, it locks a set of pieces including p to n)

Therefore, the only piece (other than piece 1) which can be moved, is the one following the first piece which is up. Let us assume that we have been able to solve
ON (p) OFF(p) $\forall p[1 \leq p \leq n-1]$

In order to perform ON(n), we have to move n, and thus reach a state where:

s(1: n-2) = down and s(n-1) = up

We can do so by first setting all pieces from 1 to n-1 up:

ON(n-1)

then putting all pieces from 1 to n-2 down:

OFF(n-2)

So we have:

ON(n) = ON(n-1); OFF(n-2); move(n); ON(n-2)
ON(1) = move(1) ON(0) = void

(ON(0) is just a void action).
Similarly:

OFF(n) = OFF(n-2); move(n); ON(n-2); OFF(n-1)
OFF(1) = move(1) OFF(0) = void

---

3 - Complexity

Let f(p) be the number of moves needed to perform ON(p), and g(p) for OFF(p). From the recursive definition, we have the following recursive relations between f and g:

f(n) = f(n-1) + g(n-2) + 1 + f(n-2)
f(1) = 1 f(0) = 0

$$g(n) = g(n-2) + 1 + f(n-2) + g(n-1)$$
$$g(1) = 1 \qquad g(0) = 0$$

From this, we easily derive that $f(p) = g(p)\ \forall p$, then that

$$f(n) = f(n-1) + 2f(n-2) + 1$$

giving $f(n) + f(n-1) + 1 = 2(f(n-1) + f(n-2) + 1)$

$$f(n) + f(n-1) = c*2^n - 1$$

Considering initial values f(1), f(2):

$$f(n) + f(n-1) = 2^n - 1$$

and finally $f(n) = 2^{n+1} \div 3$ (integer quotient)

4 - Derivation of an iterative algorithm

Several methods have been proposed for the transformation of recursive procedures into iterative ones. Most of them are based on catalogues giving recursive schemes with the associated iterative schemes. Burstall and Darlington (BD1) consider recursive functions, while Irlik (IR1) considers parameterless procedures acting on global variables. Those catalogues are too restricted to allow handling of a recursive schema as complicated as the one in procedures ON and OFF, where we have two mutually recursive procedures.

We have considered the use of program equations (AR1) for transformations of recursion into iteration. This technique may be used here, but needs some preliminary transformations.

As for Irlik's technique, procedures must be made parameterless, acting on global variables. A stack may be needed. In the present example, it is not necessary: we have assertions allowing redefinition of the parameter n after each call. Let us change n into p - Assertions are written between symbols{}

```
ON(p)
{s(1:p) = 0 }
 ON(p-1) {s(1:p-1) = 1; s(p) = 0};
 OFF(p-2) {s(1:p-2) = 0; s(p-1) = 1; s(p) = 0};
 move(p)  {s(1:p-2) = 0; s(p-1:p) = 1};
 ON (p-2)
 {s(1:p) = 1 }
```

Let us define the first piece up:

fu : s(fu) = 1 and fu = 1 or s(1: fu-1) = 0

and similarly the first piece down:

fd: s(fd) = 0 and fd = 1 or s(1: fd-1) = 1

For a game of n pieces, fu = n+1 if s(1:n) = 0
fd = n+1 if s(1:n) = 1

With these definitons, the parameterless procedures ON and OFF acting on the global variable p are defined by:

ON ≡

```
IF p < 2 THEN IF p = 1 THEN move(1) ELSE FI
         ELSE p := p-1; ON; p := fd; p := p-2; OFF;
              P := fu + 1; move(p); p := p-2; ON FI
```

OFF ≡

```
IF  p < 2 THEN  IF p = 1 THEN move(1) ELSE FI:
          ELSE p := p-2; OFF; p = fu + 1; move(p); p := p-2
               ON; p := fd; p := p-1; OFF FI
```

Notice i that in both procedures, OFF is followed by the same sequence:

```
p := fu + 1; move(p); p := p-2; ON
```

while ON is followed by

```
p:= fd; p := p-2; OFF
```

in ON, and

```
p := fd; p := p-1; OFF
in OFF
```

Those cases must be distinguis hed, which is possible by proving the very simple theorem:
The final value of p after ON is mod (p_i,2) where p_i is the value of p when entering ON.

This is simply derived from the recursive definition: ON(p) ends with ON(p-2), and so ends as ON(mod(p,2)). (mod(p,2) is 0 if p is even, 1 otherwise).

Complete derivation of an iterative program is a relatively long process. It has been made using an interactive system for program manipulation, that we have implemented at the Pontifícia Universidade Católica do Rio de Janeiro. The result is simple enough, and so we had to find a direct way to state it.

5 - The iterative solution

Let us consider the ON problem. At the beginning , s(1:n) = 0, and only the first piece may be moved. The first operation is move(1).

On the second operation, we can move either piece 1, or piece 2. But moving piece 1 undoes what has been done on the first step, and so is of no use. The only possible second move is

move (fu+1) = move(2)

On the third operation, we cannot move fu+1 = 2, because it would give a cycle with step 2. So we move the piece(1).

Iterating the process, we see that the solution is made up of an alternative sequence of moves of piece 1 and of another piece.

Using the notations for regular expressions, the game is:

(move(1) move(fu+1))*

As we noticed earlier, ON(p) ends as ON(mod(p,2)) and so with move(1) if n is odd, giving for ON(p) an expected solution

ON(p) = (move(1) (move(fu+1))*(even(p) | odd(p) move(1)))
(last move is move(fu+1) for p even, move(1) for p odd)

There is no other possibility. Thus the problem is not that of finding an iterative solution, but of proving that this sequence produces the wanted result. It can be done by induction. We assume that ON(p) is solved by this sequence for p ≤ n-1. Similarly, using the fact that OFF(p) starts with OFF(p-2), and so with OFF(mod(p,2)), we assume

OFF(p) = (even(p)) odd(p) move(1) (move(fu+1)move(1))*

for p ≤ n-1.

We just have to put these results into ON(n) and OFF(n). We do that for ON(n):

ON(n) = ON(n-1); OFF(n-2); move(n); ON(n-2)
= (move(1) move(fu+1))*(even(n-1) | odd(n-1)move(1))
(even(n-2) | odd(n-2) move(1)) (move(fu+1) move(1))*move(n)
(move(1) move(fu+1))* (even(n-2) | odd(n-2) move(1)).

For even(n), the sequence:
(even(n-1) | odd(n-1) move(1)) (even(n-2) | odd(n-2) move(1))
is: (move(1))() = move(1)
For odd(n), it is:
() (move(1)) = move(1)
even (n-2) = even(n) and so
ON(n) = (move(1) move(fu+1))*move(1) (move(fu+1) move(1))*
move(n) (move(1) move(fu+1)*)(even(n) | odd(n) move(1))
This is very easily rearranged into:
ON(n) = (move(1) move(fu+1))*(move(1) move(fu+1))*
move(1) move(n) (move(1) move(fu+1))*(even(n) |

odd(n) move(1))

= (move(1) move(fu+1))*(even(n) | odd(n) move(1))

This proves the result.

Thus we have the following iterative programs, valid for $n \geq 1$:

```
ON(n)  ≡ DO  move(1);  IF fd > n THEN EXIT FI ;
           move(fu+1); IF fd> n THEN EXIT FI ; OD

OFF(n) ≡ IF  odd(n)  THEN  move(1) FI;
         DO  move(fu+1); move(1);
             IF fu > n THEN EXIT FI OD
```

---

6 - Concluding remarks

This is an interesting programming example. It illustrates some points which, in our opinion, are very general and important.

6.1 - Stating the problem - The main difficulty here was to give a precise definition of the problem. The reader has not been faced with the need to abstract the rules of the game from its physical implementation. As a matter of fact, nothing can be done while:

- data types have not been correctly described in an abstract way (even if not very formal, as we did here)
- wanted results and initial states are not well defined.

6.2 - Finding a Solution:

The next difficulty is to exhibit one solution, regardless of any possible quality other than validity. Comparing the iterative solution and our starting recursive procedures, these ones appear as very inefficient, both for execution time and memory requirement (if implemented as such in a language whose compiler uses a stack for recursion). Morever , the recursive forms may be considered as obvious and easily readable, but they do not give any indication on a possible strategy.

6.3 - Computing a program:

Using program transforms to replace a recursive system of procedures by iterative procedures is really "making computation" on the program. It needs techniques, not imagination. It is a very strong indication that there is in computations on programs a major way to limit the role of invention in programming, leading to real programming techniques. Computer-assisted programming removes the tedious part of the job, and also reduces the need for imagination to direct the computation. Computation costs nothing, and can be tried in a lot of directions.

6.4 - A posterior derivation

The result computation is a strategy: we did not discover the iterative solution because we are very clever; we just computed it. But where a result is so simple, a simple way to reach it must be found. Knowing the result, it is not at all as difficult as inventing it for the first time.

It is our absolute conjecture that program transformation and computer-assisted programming will play a rapidly increasing role. But it implies drastic change in teaching programming. Here is certainly the more crucial issue.

6.5 - An exercise

The reader is kindly requested to consider, and solve, the following problem.

An integer n>1 being given, generate a permutation of the sequence of integers from 0 to 2^n-1 in such an order that two consective integers in the sequence differ by only one bit (in binary notation).

Good luck!

REFERENCES:

AR1 ARSAC Jacques J.
Construction de Programmes Structurés - Paris - DUNOD 1977

BD1 BURSTALL R.M. DARLINGTON. A System which automatically improves programs-Acta Informatica Vol. 6 1976 p. 41-60

IR1 IRLIK J. Translating some recursive procedures into iterative schemes-2nd International Symposium on programming DUNOD PARIS 1977