Developing Two of Arsac's Funny Algorithms*

David Gries

TR 85-711
November 1985

Department of Computer Science
Cornell University
Ithaca, NY 14853

Abstract: In Some Funny Program (Ecole Normale Supérieure, Paris, June 1985) J. Arsac discusses several algorithms, but not from the standpoint of their development. Here, we develop the algorithms given their specification using the methods espoused in The Science of Programming.

* This research supported by NSF grant DCR-8320274.

Arsac's Longest Common Prefix Problem

David Gries
Computer Science, Cornell University

6 November 1985

Two days ago, Doug McIlroy told me of the following programming problem, which he attributed to J. Arsac. Given two natural numbers, find the natural number whose decimal representation is the longest common prefix of the decimal representations of the given numbers. (The decimal representation of zero is the empty sequence; the decimal representation of a positive integer is the sequence consisting of its decimal digits, from most- to least-significant, with no leading zeroes.)

I record my algorithmic development here not because of the immensity of the problem or the difficulty in finding a solution but because it is a fine illustration of an important technique: begin the development with a search for a suitable formal specification. A suitable specification will be free of unecessary detail. It may actually make a solution almost obvious. Like it or not, different forms of specification may suggest different solutions.

I began work on the problem by pondering —just thinking about it, becoming familiar with it. I then took my Mont Blanc pencil (this note is being written with my Mont Blanc fountain pen) and tried to specify the problem in terms of the conventional definition of decimal representation: for $a \geq 0$, it is the sequence $a(0..k-1)$ that satisfies

(0) $\quad 0 \leq a_i < 10 \quad$ for $\quad 0 \leq i < k$,

$\quad\quad a = \Sigma(i: 0 \leq i < k: a_i * 10^i)$

$\quad\quad k \neq 0 \Rightarrow a_{k-1} \neq 0$

Trying to deal with form (0) was frustrating; three integers, the two givens and the result, had to be expressed in that form, and the subscripts, superscripts, and summations just got in the way. I resolved to eliminate them and did so as follows. (Below, characters from the beginning of the alphabet are integer variables; those from the end, sequence variables. For z a sequence variable, $z.0$ denotes its first element (or the empty sequence, if z is empty). The symbol ^ denotes catenation of sequences. Finally, for integer a, a_{10} denotes its decimal representation.)

The specification of the input numbers A and B is

$$\begin{aligned} Q:\ & A_{10} = X \mathbin{\hat{}} Z \ \wedge \\ & B_{10} = X \mathbin{\hat{}} Y \ \wedge \\ & (Z = Y = [\,] \ \vee \ Z.0 \neq Y.0) \end{aligned} \qquad \text{(for some sequences } X, Y, \text{ and } Z)$$

Thus, X is the desired common prefix, and we can formulate the postcondition as

$$R:\ a_{10} = X$$

where integer variable a contains the desired result.

Now an idea presents itself. Start with $a = A$ and iteratively delete elements from the end of a_{10} —by executing $a := a \div 10$— until $a_{10} = X$. Can we do this? Let's see.

Because of the symmetric nature of A and B (and Z and Y) in Q, we find an invariant P by replacing A, B, Z, and Y by fresh variables and then, with some reflection, strengthening with two conjuncts:

$$\begin{aligned} P:\ & a_{10} = X \mathbin{\hat{}} z \ \wedge \ z \ \underline{\text{prefixof}}\ Z \ \wedge & \text{(for some } z) \\ & b_{10} = X \mathbin{\hat{}} y \ \wedge \ y \ \underline{\text{prefixof}}\ Y & \text{(for some } y) \end{aligned}$$

Using # to denote "length of", we write the bound function:

$$t:\ \#x + \#y\ .$$

Reducing the bound function is done by deleting the last element of a_{10} or b_{10}. Deleting the last element of a_{10} is done by executing $a := a \div 10$. But this maintains P only if $z \neq []$. Some reflection yields $P \wedge a > b \Rightarrow z \neq []$. So we can use $a > b$ as the guard of $a := a \div 10$. This leads directly to the program

```
a,b := A,B;
do  a > b → a := a ÷ 10
 []  b > a → b := b ÷ 10
od
{P ∧ Q ∧ a = b; Hence a = b = X}
```

Remark The invariant was found by replacing expressions in the precondition, not the postcondition, because all the structure of the problem appears there. □

Remark There may be a tendency to terminate the loop as soon as one of a and b becomes 0 —using guards $a > b > 0$ and $b > a > 0$. If neither a nor b is 0 initially, this change saves exactly one iteration at the cost of guards that are twice as expensive. This is not recommended. □

Remark After writing this, I received a copy of J. Arsac's paper "Some funny programs", which discusses this and other algorithms

Arsac's Modulo-a-Power-of-two Problem

David Gries
Computer Science, Cornell University

14 November 1985

Consider writing an algorithm for the following problem. Given are three positive integers b, c, and d satisfying

(0) $\quad Q: \quad b>0 \wedge \text{odd}.b$
$\qquad\qquad c>0$
$\qquad\qquad d$ is a power of 2

Find integers x and q satisfying

(1) $\quad R: \quad b*x + c = q*d.$

One solution is found easily. An answer is known to exist satisfying $0 \le x < d$, $0 \le q \le max(b,c)$. Consider the function $f(x,q) = b*x+c-q*d$. It is increasing in x and decreasing in q. Therefore, the Saddleback Search technique can be used to find a 0 of f in the space given above in time proportional to $d + max(b,c)$. See [2] for Saddleback Search.

However, the restriction on d may allow a faster algorithm. The presence of powers of 2 often yield logarithmic solutions. Let's attempt to develop another algorithm.

A loop (or recursion) is needed, so let's look for a loop invariant P. P is often derived from R by replacing a constant by a fresh variable. What should we replace — b, c, or d (or a combination of them)? The fact that d is the power of 2 focuses attention on d, and we try

$\quad P: \quad b*x + c = q*s.$

This assertion can be established by $x, q, s := 0, c, 1$. Here, $s = 1$ is a power of 2, and s can be raised to d by $\log.d$

multiplications. Could this be a skeleton of a logarithmic algorithm? Let's see. First rewrite P to include information about s:

$$P:\ b*x + c = q*s \ \wedge\ s \text{ is a power of } 2 \ \wedge\ 1 \le s \le d$$

And let's give a bound function:

$$t:\ \log.d - \log.s$$

We write the algorithm

```
x, q, s := 0, c, 1;
do s ≠ d → Establish $P^{s}_{2*s}$ ; s:= 2*s od
```

Now, how do we establish P^{s}_{2*s} given $P \wedge s \ne d$:

$$P^{s}_{2*s}:\ b*x + c = q*2*s \ \wedge\ 2*s \text{ is a power of } 2 \ \wedge\ 1 \le 2*s \le d$$

The only difficulty is in establishing $b*x+c = q*2*s$. If q is even, this can be done by executing $q := q \div 2$. But what if q is odd? Perhaps we can _make_ it even. Note that b is odd, so adding b to odd q makes q even. Let's investigate:

$$\begin{aligned} & b*x + c = q*s \\ \equiv\ & b*x + b*s + c = q*s + b*s \\ \equiv\ & b*(x+s) + c = (q+b)*s \end{aligned}$$

Hence, if q is odd, then execution of $x, q := x+s, q+b$ makes q even and maintains P. So we have the algorithm

```
x, q, s := 0, c, 1;
do s ≠ d → if odd.q → x, q:= x+s, q+b
            [] even.q → skip
           fi;
           {P ∧ even.Q}
           q:= q÷2;
           s:= 2*s
od
```

Note that $x < a$ and $q < b+c$ are loop invariants, so that upon termination $x < d$ and $q < b+c$.

This algorithm was developed by J. Arsac — I don't know by what method. Arsac discussed the algorithm in [1] and at a summer 1985 meeting of WG2.3 of IFIP from a completely different viewpoint: Given the algorithm (only), how could one determine what it did? After skimming his analysis in [1], I tried to develop the algorithm from its specification (0) and (1), and the above development is the result.

The development wasn't as straightforward as it sounds! I spent several hours on false paths that came from thinking about some of the formulas that Arsac used in [1]. It was only when I began to consciously apply the principles I espoused in [2] that a development began emerging. I hope this experience serves me well in solving future problems.

<u>References</u>

[1] Arsac, J. Some funny programs. Ecole Normale Supérieure, Paris, 3 June 1985.

[2] Gries, D. <u>The Science of Programming</u>, Springer Verlag, N.Y., 1981.