Chapter 40 : Another Log(N) example

Given constants A, B, N : int where $0 \le N$, construct an efficient algorithm to achieve this postcondition.

Post : $r = \langle + i : 0 \le i \le N : A^{(N-i)} * B^i \rangle$

We strengthen Post

Post' : $r = \langle + i : 0 \le i \le n : A^{(n-i)} * B^i \rangle \;\wedge\; n = N$

Modeling.

* (0) $C.n = \langle + i : 0 \le i \le n : A^{(n-i)} * B^i \rangle$

We now look for some theorems about C

$C.0$
= {(0)}
$\langle + i : 0 \le i \le 0 : A^{(n-i)} * B^i \rangle$
= {1-point}
$A^0 * B^0$
= {Algebra}
1

$C(n+1)$
= {(0)}
$\langle + i : 0 \le i \le n+1 : A^{(n+1 -i)} * B^i \rangle$
= {Split off i = n+1 term}
$\langle + i : 0 \le i \le n : A^{(n+1 -i)} * B^i \rangle + A^{(n+1 - (n+1))} * B^{(n+1)}$
= {Algebra}
$\langle + i : 0 \le i \le n : A^{(n+1 -i)} * B^i \rangle + B^{(n+1)}$
= {Algebra}
$\langle + i : 0 \le i \le n : A*A^{(n -i)} * B^i \rangle + B^{(n+1)}$
= {*/+}
$A * \langle + i : 0 \le i \le n : A^{(n-i)} * B^i \rangle + B^{(n+1)}$
= {(0)}
$A * C.n + B^{(n+1)}$

So we have

- (1) $C.0 = 1$
- (2) $C.(n+1) = A*C.n + B^{(n+1)}$

Invariants.

$P0 : r = C.n \land s = B^{(n+1)}$
$P1 : 0 \le n \le N$

Establish Invariants.

$n, r, s := 0, 1, B$

Achieving postcondition.

$P0 \land P1 \land n=N \Rightarrow r = C.N$

Guard.

$n \neq N$

Variant.

$N - n$

Loop body.

$(n, r, s := n+1, U, U').P0$
$=$ {text substitution}
$U = C.(n+1) \land U' = B^{(n+2)}$
$=$ {(2) , algebra }
$U = A*C.n + B^{(n+1)} \land U' = B*B^{(n+1)}$
$=$ {P0}
$U = A*r + s \land U' = B*s$

Algorithm.

```
n, r, s := 0, 1, B ;
Do $n \neq N$ —>

    n, r, s := n+1, A*r + s, B*s

Od
{$r = C.n \land n = N$}
```

The algorithm has complexity O(N).

We note that the assignment within the loop body assigns to the variables a linear combination of themselves, so we will try to develop a more efficient solution.

Switching to a matrix representation our algorithm becomes

$$n, \begin{pmatrix} r \\ s \end{pmatrix} := 0, \begin{pmatrix} 1 \\ B \end{pmatrix};$$

Do n $\neq$ N —>

$$n, \begin{pmatrix} r \\ s \end{pmatrix} := n+1, \begin{pmatrix} A & 1 \\ 0 & B \end{pmatrix} * \begin{pmatrix} r \\ s \end{pmatrix};$$

Od

$$\{ \begin{pmatrix} r \\ s \end{pmatrix} = \begin{pmatrix} A & 1 \\ 0 & B \end{pmatrix}^N * \begin{pmatrix} 1 \\ B \end{pmatrix} \}$$

where

$$\begin{pmatrix} C.n \\ B^{(N+1)} \end{pmatrix} = \begin{pmatrix} A & 1 \\ 0 & B \end{pmatrix}^N * \begin{pmatrix} 1 \\ B \end{pmatrix}$$

We are going to construct a new algorithm based on a tail invariant so as to exploit the nice properties of multiplication of square matrices.

Invariants.

$$\text{P0} : \begin{pmatrix} A & 1 \\ 0 & B \end{pmatrix}^N * \begin{pmatrix} 1 \\ B \end{pmatrix} = D^n * \begin{pmatrix} r \\ s \end{pmatrix}$$

P1 : $0 \leq \text{n} \leq \text{N}$

Establish invariants.

$$n, \begin{pmatrix} r \\ s \end{pmatrix}, D := N, \begin{pmatrix} 1 \\ B \end{pmatrix}, \begin{pmatrix} A & 1 \\ 0 & B \end{pmatrix}$$

Achieving postcondition.

P0 $\wedge$ P1 $\wedge$ n=0 => r = C.N $\wedge$ s = $B^{(N+1)}$

Algorithm.

```
$n, \begin{pmatrix} r \\ s \end{pmatrix}, D := N, \begin{pmatrix} 1 \\ B \end{pmatrix}, \begin{pmatrix} A & 1 \\ 0 & B \end{pmatrix};$

Do n ≠ 0 —>

    If even.n —> n, D := n div 2, D*D
    [] odd.n  —> $n, \begin{pmatrix} r \\ s \end{pmatrix} := n-1, D * \begin{pmatrix} r \\ s \end{pmatrix}$
    fi

Od

{r = C.N}
```

The algorithm has complexity O(log(N)).

Final refinement.

Our language however does not provide matrices, so it is necessary to try to remove them. We will represent the matrix using 4 variables.

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix}$$

and we note that

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} * \begin{pmatrix} a & b \\ c & d \end{pmatrix} = \begin{pmatrix} a*a+b*c & (a+d)*b \\ (a+d)*c & b*c+d*d \end{pmatrix}$$

We can now eliminate the matrix operations in our algorithm.

D := $\begin{pmatrix} A & 1 \\ 0 & B \end{pmatrix}$	becomes a, b, c, d := A,1, 0, B
D := D*D	becomes a, b, c, d := a*a+b*c , (a+d)*b, (a+d)*c, b*c+d*d
$\begin{pmatrix} r \\ s \end{pmatrix} := D * \begin{pmatrix} r \\ s \end{pmatrix}$	becomes r, s := a*r + b*s, c*r + d*s

Our final algorithm is

```
n, r, s, a, b, c, d  := N, 1, B, A, 1, 0, B ;

Do n ≠ 0 —>

    If even.n —> n, a, b, c, d  := n div 2, a*a+b*c , (a+d)*b, (a+d)*c, b*c+d*d
    [] odd.n   —> n, r, s := n-1, a*r + b*s, c*r + d*s
    fi

Od

{r = C.N}
```

The algorithm has complexity O(log(N)) and doesn't involve any matrix operations.

Conclusions.

After correctness, efficiency is one of the most desirable qualities a program can have. Often we try to make efficiency gains by small "tweaks" to our program code. However, the real substantial gains come when we can move from one complexity class down to a more efficient one. For example to move from $O(N^3)$ to $O(N^2)$.

Here we have shown a technique which allows us to move from an algorithm with complexity O(N) down to one with complexity O(Log(N)). The key insight was to realise that if the assignment within the loop body assigns to a set of variables a linear combination of those same variables, then it is possible to represent this assignment using matrix operations. Then using properties of multiplication of square matrices we can reduce the complexity.

Relatively few people are aware of this technique and it receives few mentions in the literature. We believe it is a valuable technique which Computing Scientists should be aware of.