Code and Pseudo Code

Ilan Vardi, Macalester College

This article illustrates how a computer language like *Mathematica*, which incorporates symbolic computation and mathematical notation, can be used to write high-level descriptions of mathematical algorithms. Several examples are given, including a simplification of a little-known algorithm of R.W. Gosper to implement continued-fraction arithmetic.

It is quite common that mathematical results rely on explicit descriptions of algorithms. These are often written in what is called "pseudo code," usually based on a computer language like Pascal or C [Gonnet and Baeza-Yates 1991; Sedgewick 1988; Sedgewick 1990]. The question addressed here is how to check that pseudo code is correct. In other words, how do you get a "pseudo compiler" to check your pseudo code? I propose that a way to do this is to use a high-level language such as *Mathematica* to write a real implementation. The high-level and built-in mathematical capabilities of such a language allow you to write pseudo code that is essentially a transcription of the computer code, and so hopefully free of "pseudo bugs."

A Simple Example

Consider the process of adding and multiplying positive integers using their decimal expansions. The problem is to describe algorithms that return the strings of digits corresponding to the sum and product of two integers, given the two strings of digits that represent the integers.

The input will be sequences of digits $[d_n, \dots, d_0]$. Concatenation $x * y$ of sequences will be used, with the unusual convention that $[0] * x$ returns x. This condition corresponds to the usual convention that omits leading zeroes in decimal expansions.

Here are two algorithms described in pseudo code.

Addition algorithm. Given as input two sequences of digits $a = [a_m, \dots, a_0]$ and $b = [b_n, \dots, b_0]$, return a sequence $a \oplus b$:

```
intialize $k \Leftarrow 0$
for $i = 0, 1, \dots, \max(m, n)$ do
    $k \Leftarrow a_i + b_i + \lfloor k/10 \rfloor$. If $i > m$, replace $a_i$ with 0 and
      if $i > n$, replace $b_i$ with 0
    $c_i \Leftarrow k \bmod 10$
return $[\lfloor k/10 \rfloor] * [c_{\max(m,n)}, \dots, c_0]$
```

Ilan Vardi received his Ph.D. in 1982 in analytic number theory. He has also worked in other areas of mathematics, including combinatorics, probability, and analysis. Much of his work on Mathematica has appeared in the book *Computational Recreations in* Mathematica. He is finishing a second book on mathematics and Mathematica.

Multiplication algorithm. Given as input two sequences of digits $a = [a_m, \dots, a_0]$ and $b = [b_n, \dots, b_0]$, return a sequence $a \otimes b$:

```
initialize $c \Leftarrow [\,]$
for $i = 0, 1, \dots, m$ and $j = 0, 1, \dots, n$ do
    $c \Leftarrow c \oplus ([\lfloor a_i\, b_j/10 \rfloor] * [a_i b_j\ mod\ 10, \overbrace{0, 0, \dots, 0}^{i+j}])$
return $c$
```

The *Mathematica* implementation starts by defining the $*$ operator:

```
x_~join~y_ := If[x == {0}, y, x~Join~y]
```

The infix notation `x~join~y`, which means the same as `join[x, y]`, has been used in order to make the program look more like the mathematical description of the algorithm.

The next step is to write functions that take a list $[d_k, \dots, d_0]$ and return the last element d_0, and the list without its last element, $[d_k, \dots, d_1]$.

```
last[x_] := {Last[x]}
drop[x_] := Drop[x, -1]
```

The addition and multiplication programs are then

```
a_~plus~b_ :=
Block[{c = {}, i, k = 0},
    Do[ k = If[i > Length[a], 0, a[[-i]]] +
            If[i > Length[b], 0, b[[-i]]] + Floor[k/10];
        PrependTo[c, Mod[k, 10]],
        {i, 1, Max[Length[a], Length[b]]}];
    Return[{Floor[k/10]}~join~c]
]

a_~times~b_ :=
Block[{c = {}, i, j, k},
    Do[ k = a[[-i]] b[[-j]];
        c = c~plus~({Floor[k/10]}~join~
                Prepend[Table[0, {i+j-2}], Mod[k, 10]]),
       {i, 1, Length[a]}, {j, 1, Length[b]}];
    Return[c]
]
```

Note that *Mathematica* has a somewhat idiosyncratic syntax, as evident, for example, in the fact that arrays can be accessed from the right using negative indices, or in expressions like

```
Quotient[#, 10]~join~Mod[#, 10]& [last[a] last[b]]
```

used below. On the other hand, the control structure of the program is identical to the one in the high-level description, and once you get used to the peculiarities of the language, the high-level description can be directly transcribed from the program.

An even better example is to use a recursive algorithm:

Addition algorithm. Given as input two sequences of digits $a = [a_m, \ldots, a_0]$ and $b = [b_n, \ldots, b_0]$, return a sequence $a \oplus b$:

```
if $a = [\ ]$ or $b = [\ ]$ then $a * b$
else $(([a_m, \ldots, a_1] \oplus [\lfloor (a_0 + b_0)/10 \rfloor]) \oplus [b_n, \ldots, b_1]) *$
        $[(a_0 + b_0) \mod 10]$
```

Multiplication algorithm. Given as input two sequences of digits $a = [a_m, \ldots, a_0]$ and $b = [b_n, \ldots, b_0]$, return a sequence $a \otimes b$:

```
if $n = 0$ then
    if $m = 0$ then $[\lfloor a_0\, b_0/10 \rfloor] * [a_0\, b_0 \mod 10]$
    else $b \otimes a$
else $((a \otimes [b_n, \ldots, b_1] * [0]) \oplus (a \otimes [b_0])$
```

The return command has been omitted from the description as it is assumed that the algorithm returns the last evaluation. This is true of the *Mathematica* language, so the `Return` statements will be omitted from the programs as well. The *Mathematica* implementation is

```
a_~plus~b_ :=
If[Length[a] Length[b] == 0,
    a~join~b,
    Block[{sum = Quotient[#, 10]~join~
                    Mod[#, 10]& [last[a] + last[b]]},
        ((drop[a]~plus~drop[sum])~plus~drop[b])~join~last[sum]
   ]    ]
```

where `Quotient[a, b]` means $\lfloor a/b \rfloor$.

```
a_~times~b_ :=
If[Length[b]==1,
    If[Length[a] == 1,
      Quotient[#, 10]~join~Mod[#, 10]& [last[a] last[b]],
      b~times~a],
    ((a~times~drop[b])~join~{0})~plus~(a~times~last[b])
]
```

In this case, one can argue that a better choice of computer language would be Lisp or Scheme since the programs relied on list operations and recursions, the basic paradigms of these languages.

Using a Lower-Level Language

I recently asked an undergraduate class to solve the problem of the previous section and then to implement their algorithm. It turned out that all of them managed to write correct programs in C or Pascal, but none of them wrote correct high-level descriptions.

The students had a harder time writing the high-level description because they wrote low-level programs. The following C implementation does in fact follow the first two algorithms of the previous section closely, but this is hidden by code dealing with input, output, and memory allocation. Of course, this program will run at least 1000 times faster than the *Mathematica* program, but speed is not the point of this exercise. The data structure is an array `a`, where `a[0]` denotes the length of `a`. The main control structure is

```
#include  < stdio.h>

int *plus(int *num_a, int *num_b);
int *times(int *tnum_a, int *tnum_b);

main()
{int x[1001], y[1001], *z, s;          /* up to 1000 digits */
 for(x[0] = 1; (x[x[0]]=getchar()-'0') >= 0; x[0]++);
                                             /* first number */
 for(y[0] = 1; (y[y[0]]=getchar()-'0') >= 0; y[0]++);
                                            /* second number */
 z = ((x[x[0]] + '0') == '+')?plus(x, y):times(x, y);
                                          /* add or multiply */
 for(s = 1; s  <  z[0]; s++) putchar(z[s]+'0');
                                             /* print answer */
}
```

The addition program first finds the longest input string and allocates memory accordingly:

```
int *plus(int *p_a, int *p_b)
{int *a, *b, *c, size_a, size_b, k;
 size_a = (a = (p_a[0] > p_b[0])?p_b:p_a)[0];
 size_b = (b = (p_a[0] > p_b[0])?p_a:p_b)[0];
 (c = (int *) calloc(size_b+1, sizeof(int)))[0] = size_b + 1;
 k = 0;
 while(size_b > 1)
   c[size_b] = (k=(--size_a?a[size_a]:0)+b[--size_b]+ k/10)%10;
 c[1] = (k/10)?1:--(c++[0]);                    /* last digit */
 return(c);
}
```

The multiplication program uses the fact that the memory allocated by `calloc` is initialized to zero. Right shifts simply consist of incrementing the array length.

```
int *times(int *t_a, int *t_b)
{int *c, *temp, j, size_a, size_b, k;
 (c = (int *) calloc(1, sizeof(int)))[0] = 2;      /* c = 0 */
 temp = (int *) calloc(2001, sizeof(int));
 for(size_a = t_a[0]-1; size_a; size_a--){
     temp[0] = t_a[0] - size_a + 1;
     for(size_b = t_b[0]-1; size_b; size_b--){
         temp[1] = (k = t_a[size_a] * t_b[size_b])/10;
         temp[2] = k%10;
         temp[0]++;                              /* left shift */
         c = plus(c, temp);
     }
   }
 if(!c[1]) c[1] = --(c++[0]);         /* remove leading zero */
 return(c);
}
```

A Nontrivial Example

Writing an algorithm to do addition and multiplication does not require the programming language to have any mathematical features and, as noted above, using Lisp or Scheme would be a good way to go. I will now give a more complicated example where such things as built-in matrix multiplication will be required, so that a computer algebra system like Macsyma, Maple, or *Mathematica* is advantageous.

Recall that a continued fraction is a generalization of compound fractions like $14/11 = 1\ 3/11$.

$$\frac{14}{11} = 1\frac{3}{11} = 1 + \frac{1}{11/3} = 1 + \frac{1}{3\frac{2}{3}} = 1 + \cfrac{1}{3 + \cfrac{1}{3/2}}$$

$$= 1 + \cfrac{1}{3 + \cfrac{1}{1\frac{1}{2}}} = 1 + \cfrac{1}{3 + \cfrac{1}{1 + \cfrac{1}{2}}}.$$

In general, every rational number p/q can be written in the form

$$\frac{p}{q} = a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \ddots + \cfrac{1}{a_r}}}$$

where the a_i, $i > 0$, are positive integers. For ease of notation this expression is usually written as

$$\frac{p}{q} = [a_0, a_1, \ldots, a_r].$$

Consider the problem of adding and multiplying the sequence of digits of two continued fraction expansions. Even multiplying a continued fraction by 2 is a time-consuming task based on "Hurwitz rules" [Knuth 1981, exercise 4.5.3.14]. The general problem was declared to be hopeless by Khinchin, the renowned expert in the field, who wrote [Khinchin 1964]:

There is, however, another and yet more significant practical demand that the aparatus of continued fractions does not satisfy at all. Knowing the representations of several numbers, we would like to be able, with relative ease, to find the representations of the simpler functions of these numbers (especially, their sum and product). In brief, for an apparatus to be suitable from a practical standpoint, it must admit sufficiently simple rules for arithmetical operations; otherwise, it cannot serve as a tool for calculation. We know how convenient systematic fractions are in this respect. On the other hand, for continued fractions there are no practically applicable rules for arithmetical operations; even the problem of finding the continued fraction for a sum from the continued fraction representing the addends is exceedingly complicated, and unworkable in computational practice.

The problem was solved in a little-known paper of Marshall Hall [1947], but an actual description of the addition and multiplication algorithm was not given until the work of R.W. Gosper [Beeler, Gosper, and Schroeppel 1972; Gosper 1976; Knuth 1981, exercise 4.5.3.15; Levy 1984, 78]. Gosper's solution is based on three ideas. First, a computation like $3x$, where x is given as a continued fraction, will rapidly lead to more complicated forms. For example, computing $3 \cdot 14/11$ gives

$$3\,[1,3,1,2] = 3\left(1 + \frac{1}{[3,1,2]}\right) = \frac{3\,[3,1,2]+1}{[3,1,2]}$$

and so forth. Gosper's idea was not to try to simplify these forms, but to analyze the worst that can happen. It's not hard to see that you always get a linear fractional form

$$\frac{ax+b}{cx+d}, \tag{1}$$

where a, b, c, d are integers, and x is given as a continued fraction.

Gosper's second idea was to consider the x as a *formal* symbol that could input continued fraction digits into (1). In other words, think of x not as representing a rational number, but as a symbolic quantity that transforms as

$$x \mapsto q + \frac{1}{x}, \tag{2}$$

where q represents the next continued fraction digit. It remains to see how (1) transforms under (2). A simple computation shows that

$$\frac{ax+b}{cx+d} \mapsto \frac{aqx+bx+a}{cqx+dx+c}.$$

So if the form (1) is represented by a matrix

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

then the transformation law that corresponds to inputting a digit is

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \mapsto \begin{pmatrix} aq+b & a \\ cq+d & c \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} q & 1 \\ 1 & 0 \end{pmatrix}.$$

The third idea is to realize that you can output continued fraction coefficients without having complete knowledge of x. Gosper's original example was

$$\frac{70x+29}{12x+5}.$$

If $x > 0$, then

$$\frac{29}{5} \le \frac{70x+29}{12x+5} < \frac{70}{12}.$$

This means that

$$\frac{70x+29}{12x+5} = 5 + \frac{10x+4}{12x+5}.$$

Similarly,

$$\frac{12}{10} \le \frac{12x+5}{10x+4} < \frac{5}{4}$$

so

$$\frac{12x+5}{10x+4} = 1 + \frac{2x+1}{10x+4},$$

and

$$\frac{10x+4}{2x+1} = 4 + \frac{2x}{2x+1}.$$

This means that for *any* $x > 0$ you get a continued fraction expansion

$$\frac{70x+29}{12x+5} = 5 + \cfrac{1}{1 + \cfrac{1}{4 + \cfrac{1}{(2x+1)/x}}}$$

In other words, you are able to output three continued fraction coefficients of the form $(10x+25)/(2x+1)$ without knowing too much about x. (Note that you cannot output a fourth coefficient since $(2x+1)/x \mapsto 2 + 1/x$ does not give the correct coefficient unless $x > 1$.)

In the case when x represents a sequence of continued fraction coefficients, the situation is even better because the coefficients are greater than or equal to one. Now, since one is assuming that there is a continuing sequence of continued fraction coefficients, the further assumption that $x > 1$ can be made, so being able to output a continued fraction corresponds to checking that for some integer n

$$n \le \frac{a}{c} \le \frac{a+b}{c+d} < n+1,$$

in other words, that

$$\left\lfloor \frac{a}{c} \right\rfloor = \left\lfloor \frac{a+b}{c+d} \right\rfloor.$$

When this happens, the output will be the common value $q = \lfloor a/c \rfloor$. The point is that this always happens, given a sufficient number of digits of x.

It remains to see what happens to the form after this has been output. This is

$$\frac{1}{\frac{ax+b}{cx+d} - q} = \frac{cx+d}{ax - cqx + b - dq},$$

and the corresponding matrix transformation is

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \mapsto \begin{pmatrix} c & d \\ a-cq & b-dq \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & -q \end{pmatrix} \begin{pmatrix} a & b \\ c & d \end{pmatrix}.$$

All these steps can be combined into an algorithm, but first an algorithm to compute the continued fraction of a rational number must be written. The following algorithm essentially follows the 14/11 example. (Ordinary concatenation $[0] * [x_0, \dots, x_r] = [0, x_0, \dots, x_r]$ will be used from now on.)

Algorithm to compute the continued fraction expansion of a rational number. Given a rational number q, return a sequence $f(q)$:

```
    if $q = 1/0$ then [ ] else $[\lfloor q \rfloor] * f(1/(q - \lfloor q \rfloor))$
```

Algorithm to compute the continued fraction expansion of $(ax+b)/(cx+d)$. Given a matrix

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

and the continued fraction expansion $x = [x_0, \dots, x_r]$, return the sequence

$$f\left(\begin{pmatrix} a\, b \\ c\, d \end{pmatrix}, x\right):$$

```
    if $r = 0$ then $f((a\, x_0 + b)/(c\, x_0 + d))$.
    else
        if $a, b, c, d > 0$ and $\lfloor a/c \rfloor = \lfloor (a+b)/(c+d) \rfloor$
        then $[\lfloor a/c \rfloor] * f\left(\begin{pmatrix} 0 & 1 \\ 1 & -\lfloor a/c \rfloor \end{pmatrix} \begin{pmatrix} a & b \\ c & d \end{pmatrix}, x\right)$ (Output a digit.)
    else $f\left(\begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} x_0 & 1 \\ 1 & 0 \end{pmatrix}, [x_1, \dots, x_r]\right)$    (Input a digit.)
```

In *Mathematica*, these algorithms are

```
ContinuedFraction[q_] :=
If[q == 1/0, {},
   {Floor[q]}~Join~ContinuedFraction[1/(q-Floor[q])]

ContinuedFraction[matrix_, x_] :=
If[Length[x] == 1,
  ContinuedFraction[Divide @ @ (matrix . {First[x], 1})],
  Block[{q = Positive[Min[matrix]] &&
           Quotient @ @ (matrix . {{1, 1}, {0, 1}})},
    If[q && Min[q] == Max[q],
      {#}~Join~ContinuedFraction[{{0, 1}, {1, -#}}.matrix, x]&
           [First[q]],
      ContinuedFraction[matrix.{{First[x],1}, {1,0}}, Rest[x]]
    ] ] ]
```

Since division by zero has been allowed, the line

```
Off[Power::infy]
```

is useful in order to avoid annoying warning messages.

Note the similarity in the control structure of the program and the pseudo code. It should be clear that the pseudo code is essentially a transcription of the computer program.

Finally, the algorithms for addition and multiplication can be described. Just as before, computing $x + y$ or xy leads to more complicated expressions. The worst expression you get is a "bilinear fractional form"

$$\frac{axy + bx + cy + d}{exy + fx + gy + h}\,. \tag{3}$$

Treating x and y as formal variables, this can be represented by an ordered pair of matrices (or tensor)

$$\left(\begin{pmatrix} a & b \\ c & d \end{pmatrix}, \begin{pmatrix} e & f \\ g & h \end{pmatrix}\right).$$

Letting $y \mapsto q + 1/y$ in (3) gives a transformation for the tensor

$$\left(\begin{pmatrix} a & b \\ c & d \end{pmatrix}, \begin{pmatrix} e & f \\ g & h \end{pmatrix}\right) \mapsto \left(\begin{pmatrix} a & b \\ c & d \end{pmatrix}, \begin{pmatrix} e & f \\ g & h \end{pmatrix}\right) \begin{pmatrix} q & 1 \\ 1 & 0 \end{pmatrix}$$

where ordinary matrix multiplication is used on each component. Similarly, letting $x \mapsto q + 1/x$ in (3) corresponds to

$$\left(\begin{pmatrix} a & b \\ c & d \end{pmatrix}, \begin{pmatrix} e & f \\ g & h \end{pmatrix}\right) \mapsto \begin{pmatrix} q & 1 \\ 1 & 0 \end{pmatrix} \left(\begin{pmatrix} a & b \\ c & d \end{pmatrix}, \begin{pmatrix} e & f \\ g & h \end{pmatrix}\right)$$

Just as before, one uses an Euclidean algorithm to output coefficients, but this time there won't be a guarantee that intermediate values will be greater than one. A tricky part of the algorithm is to decide whether to choose x or y to input the next digit. A direct solution is to alternate between them, which requires a componentwise transpose

$$\left(\begin{pmatrix} a & b \\ c & d \end{pmatrix}, \begin{pmatrix} e & f \\ g & h \end{pmatrix}\right)^T = \left(\begin{pmatrix} a & c \\ b & d \end{pmatrix}, \begin{pmatrix} e & g \\ f & h \end{pmatrix}\right),$$

corresponding to switching x and y in (3).

Algorithm to compute the continued fraction of $(axy + bx +$ $cy + d)/(exy + fx + gy + h)$. Given as input a tensor

$$\left(\begin{pmatrix} a & b \\ c & d \end{pmatrix}, \begin{pmatrix} e & f \\ g & h \end{pmatrix}\right)$$

and continued fraction expansions $x = [x_0, \ldots, x_r]$, $y = [y_0, \ldots, y_s]$, return a sequence

$$f\left(\left(\begin{pmatrix} a\,b \\ c\,d \end{pmatrix}, \begin{pmatrix} e\,f \\ g\,h \end{pmatrix}\right), x, y\right):$$

```
if $s = 0$ then $f\left(\left(\left(\begin{smallmatrix} a & b \\ c & d \end{smallmatrix}\right), \left(\begin{smallmatrix} e & f \\ g & h \end{smallmatrix}\right)\right) \left(\begin{smallmatrix} y_0 \\ 1 \end{smallmatrix}\right), x\right)$.
else
if $a, b, c, d, e, f, g, h > 0$ and
        $\lfloor a/e \rfloor = \lfloor b/f \rfloor = \lfloor c/g \rfloor = \lfloor d/h \rfloor$ then
    $[\lfloor a/e \rfloor] * f\left(\left(\begin{smallmatrix} 0 & 1 \\ 1 & -\lfloor a/e \rfloor \end{smallmatrix}\right) \left(\left(\begin{smallmatrix} a & b \\ c & d \end{smallmatrix}\right), \left(\begin{smallmatrix} e & f \\ g & h \end{smallmatrix}\right)\right), x, y\right)$
    (Output a digit.)
else  $f\left(\left\{\left(\left(\begin{smallmatrix} a & b \\ c & d \end{smallmatrix}\right), \left(\begin{smallmatrix} e & f \\ g & h \end{smallmatrix}\right)\right) \left(\begin{smallmatrix} y_0 & 1 \\ 1 & 0 \end{smallmatrix}\right)\right\}^T, [y_1, \ldots, y_s], x\right)$
    (Input a digit and switch $x$, $y$.)
```

This algorithm allows one to add and multiply continued fractions according to the rules

$$x \oplus y = f\left(\left(\left(\begin{smallmatrix} 0 & 1 \\ 1 & 0 \end{smallmatrix}\right), \left(\begin{smallmatrix} 0 & 0 \\ 0 & 1 \end{smallmatrix}\right)\right), x, y\right),$$
$$x \otimes y = f\left(\left(\left(\begin{smallmatrix} 1 & 0 \\ 0 & 0 \end{smallmatrix}\right), \left(\begin{smallmatrix} 0 & 0 \\ 0 & 1 \end{smallmatrix}\right)\right), x, y\right).$$

The program is simplified by the fact that ordinary *Mathematica* matrix multiplication for generalized matrices behaves in the same way as the componentwise matrix multiplication defined above.

```
ContinuedFraction[tensor_, x_, y_] :=
If[Length[y] == 1,
  ContinuedFraction[tensor.{First[y], 1}, x],
  Block[{q = Positive[Min[tensor]] && Quotient @ @ tensor},
    If[q && Min[q] == Max[q],
      {#}~Join~ContinuedFraction[{{0, 1},{1, -#}}.tensor, x, y]&
           [q[[1, 1]]],
      ContinuedFraction[
        Transpose / @ (tensor.{{First[y], 1},{1,0}}), Rest[y], x]
    ]
  ]
]

ContinuedFractionPlus[x_, y_] :=
ContinuedFraction[{{{0,1}, {1,0}}, {{0,0}, {0,1}}}, x, y]

ContinuedFractionTimes[x_, y_] :=
ContinuedFraction[{{{1,0}, {0,0}}, {{0,0}, {0,1}}}, x, y]
```

Concluding Remarks

Subtraction and division can be also be computed in this way by using the formulas

$$x \ominus y = f\left(\left(\left(\begin{smallmatrix}0 & 1\\ -1 & 0\end{smallmatrix}\right), \left(\begin{smallmatrix}0 & 0\\ 0 & 1\end{smallmatrix}\right)\right), x, y\right) \ ,$$

$$x \oslash y = f\left(\left(\left(\begin{smallmatrix}0 & 1\\ 0 & 0\end{smallmatrix}\right), \left(\begin{smallmatrix}0 & 0\\ 1 & 0\end{smallmatrix}\right)\right), x, y\right) \ .$$

The real advantage of Gosper's method is that it allows you to add and multiply numbers with known continued fraction expansions, for example, $e = [2, 1, 2, 1, 1, 4, 1, 1, 6, \ldots]$ and $\phi = (\sqrt{5}+1)/2 = [1, 1, 1, \ldots]$. The algorithms above are easily modified to compute quantities like $e + \phi$, where the digit input would be the functions

$$e(n) = \begin{cases} 2 & \text{if } n = 0 \\ 2(n+1)/3 & \text{if } n \equiv 2 \pmod 3 \\ 1 & \text{otherwise} \end{cases}$$

and $\phi(n) = 1$.

Finally, there are problems with continued fraction arithmetic. For example, the computation $[1, 2, 2, 2, \ldots] \otimes [1, 2, 2, 2, \ldots]$, corresponding to $\sqrt{2} \times \sqrt{2}$, will either return completely incorrect continued fraction coefficients, or return $[2, a]$, where a is a large integer, depending on whether one uses an odd or even length input. This is slightly different from the decimal case.

Acknowledgments

I would like to thank Henry Cejtin and Igor Rivin for helpful comments.

References

Beeler, M., R.W. Gosper, and R. Schroeppel. 1972. *HAKMEM*. A.I. Lab Memo # 239. M.I.T.

Gonnet, G.H., and R. Baeza-Yates. 1991. *Handbook of Algorithms and Data Structures*. 2nd ed. Addison-Wesley.

Gosper, R.W. 1976. *Continued Fraction Arithmetic*. Preprint.

Hall, M., Jr. 1947. On the sum and product of continued fractions. *Annals of Math*. 48:966–993.

Khinchin, A.Ya. 1964. *Continued Fractions*. University of Chicago Press.

Knuth, D.E. 1981. *Seminumerical Algorithms*. Vol. 2 of *The Art of Computer Programming*. 2nd ed. Addison-Wesley.

Levy, S. 1984. *Hackers: Heroes of the Computer Revolution*. Doubleday.

Sedgewick, R. 1988. *Algorithms*. 2nd ed. Addison-Wesley.

Sedgewick, R. 1990. *Algorithms in C*. Addison-Wesley.

Ilan Vardi
Mathematics Dept., Macalester College
St. Paul, MN 55105