# Recursive and Iterative Functions for Generating Fibonacci Numbers

J. S. Rohl*

TR 89-1017
June 1989

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

*On sabbatical leave from The University of Western Australia.

# Recursive and iterative functions for generating Fibonacci numbers

J.S.Rohl[1]
Department of Computer Science
Cornell University
Ithaca, New York, 14853-7501

**Abstract**
As reference to the popular computing press will confirm, there is a great deal of misunderstanding about the efficient calculation of Fibonacci numbers. As the "obvious" iterative version is linear and the "obvious" recursive version is exponential, many assume that recursion is inherently less efficient than iteration. Even in the technical press, the more efficient logarithmic versions are given in an abstract way, which makes their use rather inconvenient. This report gives complete functions, both iterative and recursive, for the linear and logarithmic algorithms

[1]On sabbatical leave from The University of Western Australia

## 1. Introduction

The Fibonacci number sequence is well-known in biology, mathematics and computer science. Its first two numbers are 0 and 1 and each subsequent number is the sum of its two predecessors. Thus the sequence starts with

0 1 1 2 3 5 8 13 21 34 55

From the formal definition of the sequence corresponding to the above definition:

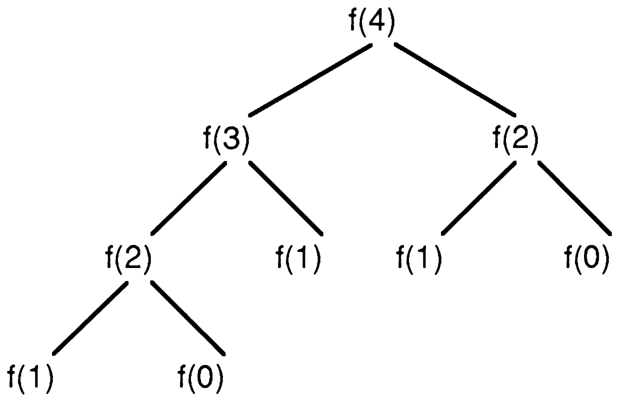
$$f_n = \begin{cases} 0, & n=0 \\ 1, & n=1 \\ f_{n-1} + f_{n-2}, & n\geq 2 \end{cases}$$

an obvious recursive function can be produced. The one below is in the Dijkstra-Gries notation.

```
function f(n : nat) → nat;
{pre : 0≤n}
{yields f_n}
  if n=0 ∨ n=1  → yield n
  [] n≥2        → yield f(n-1) + f(n-2)
  fi
```

Note that the function calculates all Fibonacci numbers within the range of type *natural*, including $f_0$. We will be careful to ensure that all the functions we give have this property.

Regrettably, this function is slow, since each recursive call is evaluated independently of the others. The tree of procedure calls for the evaluation of f(4) is given below.




Thus f(1) is evaluated 3 times, f(0) and f(2) twice and so on. Indeed the number of procedure calls required to evaluate f(n) is $2f_{n+1}$-1. It is this fact that has lead many to believe that recursion is inherently inefficient, especially when it is contrasted with the classical iterative version, given below. Note that to avoid unnecessary overflow, we calculate $f_{i-1}$ (in a) and $f_i$ (in b) rather than $f_i$ and $f_{i+1}$; and to ensure that f(0) is calculated we assume the sequence extended backwards to $f_{-1} = 1$, initializing a and b to 1 and 0 respectively.

```
function f(n : nat) → nat;
{pre : 0≤n}
{yields f_n}
   var i, a, b : nat;
   begin
      i, a, b := 0, 1, 0;
      {inv : 0≤i≤n ∧ a = f_{i-1} ∧ b = f_i}
      {bound : n-i}
      do i < n → i, a, b := i+1, b, a+b od;
      yield b
   end;
```

This is obviously a linear procedure (f(n) is evaluated in a time proportional to n), in contrast to the recursive, which is exponential (f(n) is evaluated in a time proportional to $f_{n+1}$). The case against recursion seems proved! BUT these two functions do not describe the same algorithm: whereas the recursive version clearly calculates one Fibonacci number from two others, the iterative version calculates a pair of (consecutive) numbers from another (overlapping) pair. Within the loop new values for a and b are calculated from the old values.

## 2. Recursive functions with linear complexity

Of course, it is possible to write a recursive version of this algorithm - in fact, it is possible to write two. Let us use the convention that pair i consists of $f_{i-1}$ and $f_i$ so that pair 0 consists of $f_{-1}$ and $f_0$. The more directly related of the two recursive functions is one, that, given pair i, $f_{i-1}$ and $f_i$, calls itself recursively to calculate to pair i+1 with the information that it consists of $f_i$ and $(f_{i-1}+f_i)$, the process terminating when i = n. It is usual to nest it within a function that has the traditional heading:

```
function f(n : nat) → nat;
{pre : 0≤n}
{yields f_n}

   function f1(i, a, b : nat) → nat;
   {pre : 0≤i≤n ∧ a=f_{i-1} ∧ b=f_i}
   {yields f_n}
      if i = n → yield b
      [] i < n → yield f1(i+1, b, a+b)
      fi;

   yield f1(0,1,0)
```

To those who have seen recursive procedures only in the style of the one this paper opened with - those using a divide-and-conquer strategy - this procedure looks strange. It seems to be a rendering into recursive form of a function that is inherently iterative. Nevertheless, this class of recursive procedure is an important one. Such procedures are called *tail-recursive* or *pre-order*, so called because the recursive call is the last (tail) action that takes place, and equivalently because the action comes before (pre) the recursive call. Parameters a and b are called *accumulating parameters*, because their values in successive invocations accumulate the value or values of the function. Such procedures can be trivially converted to iterative form - and all single loop procedures can be converted to them. We will, in what follows, generally give only the recursive version. Note that both versions of this function start at pair 0 and work inexorably upwards to pair n. (Hence the need for the variable i.)

Note that a pair of Fibonacci, as described above, can be formally defined.

$$\langle f_{n-1}, f_n\rangle = \begin{cases} \langle 0,\ 1\rangle, & n=0 \\ \langle f_n, f_{n-1} + f_n\rangle, & n\geq 1 \end{cases}$$

Thus we can write a "real" recursive procedure in the divide-and-conquer style, too. We have extended the notation of Dijkstra and Gries in a fairly obvious way to handle functions that yield a tuple of results. Below, the inner function calls itself recursively for the values of $f_{n-2}$ and $f_{n-1}$, which it uses to calculate $f_{n-1}$ and $f_n$.

```
function f(n : nat) → nat;
{pre : 0≤n}
{yields fn}
   var a, b : nat;

   function f2(n : nat) → nat × nat;
   {pre : 0≤n}
   {yields < fn-1, fn>}
      var a, b : nat;
      if n = 0 → yield <1, 0>
      [] n > 0 → yield <a, a+b> where <b, a> := f2(n-1)
      fi;

   yield b where <a, b> := f2(n)
```

This is, of course, a post-order procedure, since a+b is calculated after the recursive call. Note its operation in contrast with that of the pre-order function: in attempting to find pair n, it sets out to find the pair n-1 and so on, until the pair 0 is sought. The Fibonacci numbers themselves are calculated from $f_0$ upwards, of course, as they are in the pre-order versions; but the decision about which pair of Fibonacci numbers needs to be calculated is made in opposite ways. Since all Fibonacci numbers in the range 0 to n are calculated in each case, the different order of decision is of no consequence.

In general pre-order procedures cannot be converted to iterative form without some form of a stack. In this particular example, the procedure has an important property: the operation applied to its value parameter (decrementing) is invertible (incrementing) and so the stack can be avoided. In fact, by the use of this property, the iterative function above is produced.

### 3. Some properties of Fibonacci numbers

It has long been known that the problem of calculating Fibonacci numbers can be expressed in terms of matrices. From the defining relationships given in the first paragraph we have:

$$\begin{bmatrix} f_i \\ f_{i+1} \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} f_{i-1} \\ f_i \end{bmatrix}$$

It is easy to see that:

$$\begin{bmatrix} f_i & f_{i+1} \\ f_{i+1} & f_{i+2} \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} f_{i-1} & f_i \\ f_i & f_{i+1} \end{bmatrix}$$

so that:

$$\begin{bmatrix} f_{i+n-1} & f_{i+n} \\ f_{i+n} & f_{i+n+1} \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^n \begin{bmatrix} f_{i-1} & f_i \\ f_i & f_{i+1} \end{bmatrix}$$

When i=0 this reduces to:

$$\begin{aligned} \begin{bmatrix} f_{n-1} & f_n \\ f_n & f_{n+1} \end{bmatrix} &= \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^n \begin{bmatrix} f_{-1} & f_0 \\ f_0 & f_1 \end{bmatrix} \\ &= \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^n \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \\ &= \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^n \end{aligned}$$

That is the problem of finding $f_n$ (and $f_{n-1}$ and $f_{n+1}$) can be reduced to the evaluation of the $n^{th}$ power of a special matrix. We will use this form directly in Section 5.

When n is even this last formula can be re-expressed :

$$\begin{bmatrix} f_{n-1} & f_n \\ f_n & f_{n+1} \end{bmatrix} = \left(\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^{n/2}\right)^2$$

$$= \begin{bmatrix} f_{n/2-1} & f_{n/2} \\ f_{n/2} & f_{n/2+1} \end{bmatrix}^2$$

Note that, although the equation is valid if n=0, it is in that case vacuous, so that we use the equation only when n≥2.

If we evaluate the right-hand-side, we arrive at the following:

$$f_n = 2f_{n/2-1}f_{n/2} + f^2_{n/2}$$

Similarly, when n is odd :

$$\begin{bmatrix} f_{n-1} & f_n \\ f_n & f_{n+1} \end{bmatrix} = \left(\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^{n/2}\right)^2 \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} f_{n/2-1} & f_{n/2} \\ f_{n/2} & f_{n/2+1} \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}$$

from which we have:

$$f_n = f^2_{n/2} + f^2_{n/2+1}$$

Thus we can define f as follows:

$$f_n = \begin{cases} 0, & n=0 \\ 1, & n=1 \\ 2f_{n/2-1}f_{n/2} + f^2_{n/2}, & n\geq 2 \text{ and even} \\ f^2_{n/2} + f^2_{n/2+1}, & n>3 \text{ and odd} \end{cases}$$

Note that this equation is quite independent of the notion of matrices. It can be derived without recourse to them, and proved by induction.

**4. Logarithmic procedures based on the last relationship**

Since this expresses the value of a Fibonacci number in terms of other Fibonacci numbers whose subscripts are approximately half of that of the original, we would expect to be able to reduce the complexity to logarithmic. But we must be careful. A direct expression of those formulae gives:

```
function f(n : nat) → nat;
{pre : 0≤n}
{yields fn}
  if n=0 ∨ n=1       →   yield n
  [] n≥2 ∧ even(n)   →   yield 2*f(n÷2-1)*f(n÷2) + f(n÷2)²
  [] n≥2 ∧ odd(n)    →   yield f(n÷2)² + f(n÷2+1)²
  fi
```

Because each call of f produces another 2 calls this function is only linear! We need to be more subtle, and write the function so that it calculates pairs of values as we did in the previous section. There are many such formulations. One is:

$$\langle f_{n-1}, f_n\rangle = \begin{cases} \langle 1, 0\rangle, & n=0 \\ \langle 0, 1\rangle, & n=1 \\ \langle f^2_{n/2-1} + f^2_{n/2}, 2f_{n/2-1}f_{n/2} + f^2_{n/2}\rangle, & n\geq 2 \text{ and even} \\ \langle 2f_{n/2-1}f_{n/2} + f^2_{n/2}, f^2_{n/2} + f^2_{n/2+1}\rangle, & n\geq 3 \text{ and odd} \end{cases}$$

In the inner function of the following a holds $f_{n\div 2-1}$ and b holds $f_{n\div 2}$.

```
function f(n : nat) → nat;
{pre : 0≤n}
{yields f_n}
var a, b : nat;

   function f2(n : nat) → nat × nat;
   {pre : 0<i≤n}
   {yields < f_{n-1}, f_n>}
      var a, b : nat;
      if n = 1  → yield <0, 1>
      [] n ≠ 1  → if even(n) →   yield <a^2+b^2, 2*a*b + b^2>
                  [] odd(n)  →   yield <2*a*b - a^2, a^2 + b^2>
                  fi where <a, b> = f2((n+1)÷2)
      fi

   if n = 0 → yield 0
   [] n ≠ 0 → yield b where <a, b> := f2(n)
   fi
```

Note that the n=0 case is treated separately.

This is a post-order function whose value parameter is not invertible. (The **div** operation loses information which cannot be recovered by multiplying.) Thus we can convert this procedure only by using some form of stack. What has to be stacked is a sequence of values of i, which are subsequently accessed in reverse order. When accessed, i is tested only for oddness, so that we need stack only its least significant digit. Successive values of i are obtained by incrementing and dividing by 2, so that we can replace the stack by an integer whose digits are the reverse of those obtained from i in this way, preceded by a "1" to indicate where the significant digits end. To describe the invariant, we define sequences $n_i$, $r_i$, $0 \le i \le \#n$, of the numbers generated by the increment and divide process. For example, if n=13, then $n_i$ = 13 7 4 2 1 and $r_i$ = 1 3 7 14 28.

```
function f(n : nat) : nat;
{pre : 0≤n}
{yields f_n}
  var a, b, r : nat;
  if n = 0 →   yield 0
  [] n ≠ 0 →   r := 1;
               {inv : (∃i, 0≤i≤#N, n=n_i ∧ r=r_i}
               {bound : n-1}
               do even(n)          → r, n := 2*r, n÷2
               [] n ≠ 1 ∧ odd(n)   → r, n := 2*r+1, (n+1)÷2
               od;
               a, b := 0, 1;
               {inv : (∃i, 0≤i≤#N, r=r_i) ∧ <a, b> = <f_{n_i-1}, f_{n_i}>}
               {bound : r-1}
               do even(r)          → a, b, r := a^2 + b^2, 2*a*b + b^2, r÷2
               []r ≠ 1 ∧ odd(r)    → a, b, r := 2*a*b - a^2, a^2 +b^2, r÷2
               od
               yield b
  fi
```

The two functions of this section are the fastest logarithmic procedures - though the recursive one requires logarithmic store for the stack. Note that, if we had not been concerned with overflow, we could have defined pair i as $<f_i, f_{i+1}>$ instead of $<f_{i-1}, f_i>$. In this case the control would have been a little simpler.

## 5. Logarithmic procedures based on matrix multiplication

In section 3 we derived the relationship:

$$\begin{bmatrix} f_{n-1} & f_n \\ f_n & f_{n+1} \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^n$$

Therefore we can calculate Fibonacci numbers by powering, which we know can be done in logarithmic time. There are two ways of doing this. The first is essentially iterative. If the exponent, n, is expressed in binary as $b_k, b_{k-1}, \ldots b_0$, then $x^n$ is the product of $x^{2^k}$ for all those values of k for which $b_k$=1. The terms, $x^{2^k}$, and the product are accumulated within the loop.

```
function Power (x : sometype; n : nat) → sometype;
   {pre : 0≤n}
   {yields x^n}
var y : sometype;
begin
   y := 1;
   {inv : 0≤n ∧ y*x^n = X^N}
   {bound : ⌊log(n+1)⌋}
   do n ≠ 0 →
              {inv : 0≤n ∧ y*x^n = X^N}
              {bound : ⌊log(n+1)⌋}
              do even(n) →   x, n := x*x, n÷2 od;
              y, n := y*x, n-1
   od;
   yield y
end
```

To adapt this to the generation of Fibonacci numbers we have only to consider the multiplication of a pair of $2 \times 2$ matrices. In fact we need keep only the first column containing $f_{n-1}$ and $f_n$, since the second column contains a repeated value, $f_n$, and a value, $f_{n+1}$, which is the sum of the two saved values. We adopt the convention that a and b will hold the matrix elements corresponding to y and c and d those corresponding to x.

```
function  f(n : nat) → nat;
   {pre : 0≤n}
   {yields f_n}
var a, b, c, d : nat;
begin
   a, b, c, d := 1, 0, 0, 1;
   {inv: 0≤n ∧ (∃p, 0≤p≤N, <c, d> = <f_p-1, f_p> ∧ <a, b> = <f_N-pn-1, f_N-pn>)}
   {bound : ⌊log(n+1)⌋}
   do n ≠ 0 →
              {inv : 0≤n ∧ (∃p, 0≤p≤N, <c, d> = <f_p-1, f_p>
                   ∧ <a, b> = <f_N-pn-1, f_N-pn>}
              {bound : ⌊log(n+1)⌋}
              do  even(n) →  c, d, n := c^2+d^2, 2*c*d+d^2, n÷2 od;
              a, b, n := a*c+b*d, a*d+b*(c+d), n-1
   od;
```

```
    yield b
  end;
```

This is the procedure usually given in the literature. Of course, the function can be expressed recursively using accumulating parameters:

**function** f(n : nat) → nat;
  {**pre** : n≥0}
  {**yields** $f_n$}

  **function** f1 (**value** a, b, c, d, n : nat) → nat;
    {**pre** : 0≤n ∧ (∃p, 0≤p≤N, <c, d> = <$f_{p-1}$, $f_p$> ∧ <a, b> = <$f_{N-pn-1}$, $f_{N-pn}$>)}
    {**yields** $f_n$}
    **if** n=0 → **yield** b
    [] 0≤n ∧ even(n) → **yield** f1(a, b, $c^2 + d^2$, $2*c*d + d^2$, n÷2)
    [] odd(n) → **yield** f1(a*c + b*d, a*d + b*(c+d), c, d, n-1)
    **fi**;

  **yield** f1(1, 0, 0, 1, n)

Note that this is less efficient than the iterative version because of the extra assignments required by the 4 parameters - a frequent occurrence with accumulating parameters - and because of the extra tests that are avoided in the iterative version because of the inner loop.

The second method of matrix multiplication is essentially recursive in that it uses the divide-and-conquer technique, with the definition:

$$x^n = \begin{cases} 0, & n=0 \\ x^{n\div 2} \,.\, x^{n\div 2}, & n \text{ even and } >0 \\ x^{n\div 2} \,.\, x^{n\div 2} \,.\, x, & n \text{ odd} \end{cases}$$

It is given below.

```
function Power (x : sometype; n : nat) → sometype;
  {pre : 0≤n}
  {yields x^n}
  if n = 0               →yield 1 {of sometype}
  [] n ≠ 0 ∧ odd(n)      →yield Power(x, n÷2)^2 * x
  [] n ≠ 0 ∧ even(n)     →yield Power(x, n÷2)^2
  fi
```

When n is even we calculate:

$$\begin{bmatrix} f_{n-1} & f_n \\ f_n & f_{n+1} \end{bmatrix} = \begin{bmatrix} f_{n/2-1} & f_{n/2} \\ f_{n/2} & f_{n/2+1} \end{bmatrix}^2$$

and, when n is odd we calculate:

$$\begin{bmatrix} f_{n-1} & f_n \\ f_n & f_{n+1} \end{bmatrix} = \begin{bmatrix} f_{n/2-1} & f_{n/2} \\ f_{n/2} & f_{n/2+1} \end{bmatrix}^2 \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}$$

These are exactly the formulae used in Section 4, and lead to the same functions.

## 5. Conclusions

We have shown that for all methods of calculating Fibonacci numbers, linear and logarithmic, we can write both iterative and recursive procedures. In some cases the iterative one is the more natural (where the recursive one uses accumulating parameters); in others the recursive one is (where the iterative one has to represent a stack). Both have the same complexity. If there are any differences in speed between a pair of procedures, it is due to the way a compiler handles procedure calls, and loops. At worst this is a small constant.