

# Fibonacci Numbers and Exponents: A Surprising Result

Collin Park

April 16, 2015

## Abstract

I used to interview a lot of candidates for software-engineering positions. For a quarter-century or so, I asked them to write a program (a *function* or *subroutine* actually) to calculate the  $n$ th “Fibonacci number.”<sup>1</sup>

It’s easy to write a short routine that’s surprisingly slow (exponential in  $n$  actually); it’s almost as easy to write one that’s linear in  $n$ . And many candidates were surprised that, given  $f_0$  and  $f_1$ , it doesn’t take much room to simply store all values of  $f_n$  that fit into a 32-bit `int`.

But one day, after asking the same programming question for 20 years, I saw an algorithm that could compute  $f_n$  in  $O(\log n)$  time—on that day, it was my turn to be surprised.

## How I presented the problem

Basically, I’d ask candidates to write an “integer”<sup>2</sup> function of an “integer” parameter, that would compute this mathematical function:

$$f(n) = \begin{cases} -1 & \text{if } n < 0, \\ 0 & \text{if } n = 0, \\ 1 & \text{if } n = 1, \\ f(n-1) + f(n-2) & \text{if } n \geq 2 \end{cases}$$

I would then ask them to describe how they’d approach the problem, and write something on paper or a whiteboard, using any programming language I could read.

<sup>1</sup>The  $n$ th Fibonacci number is the sum of the  $(n-1)$ st and  $(n-2)$ nd Fibonacci numbers; if we denote the  $n$ th by  $f_n$ , then  $f_n = f_{n-1} + f_{n-2}$ . The starting point is arbitrary, but I’ve typically seen  $f_0 = 0, f_1 = 1$ .

<sup>2</sup>I put quotes around “integer” here because most programming languages use 32- or 64-bit quantities, rather than actual integers. That is, we compute with a subset of the integers, typically  $(-2^{31}, 2^{31} - 1)$ .

## A very slow solution

Candidates usually spoke intelligently about handling the various cases. When they started writing, they typically wrote a recursive routine embodying this:

```
if n < 0 then
  | return -1;
else if n = 0 then
  | return 0;
else if n = 1 then
  | return 1;
else
  | return f(n-1) + f(n-2);
```

Algorithm 1:  $n$ th Fibonacci number, recursively

In actual code it would look like Example 1. (Note: This code was actually run on Python 2.7.6; the example also includes a small verification program.)

```
def f(n):
    "nth Fibonacci number recursively"
    if n < 0:
        return -1
    elif n < 2:
        return n
    else:
        return f(n-1) + f(n-2)

for k in range(40):
    print k, f(k)
```

Example 1: Recursive realization (Python 2) of  $f(n)$

The recursive routine is slow because every time  $n$  increases by 2, the execution time roughly doubles. Here’s what I mean:

- To compute  $f(k+2)$ , it computes  $f(k+1)$  and  $f(k)$ ; to compute  $f(k+1)$ , it computes  $f(k)$  and  $f(k-1)$ .
- Thus, to compute  $f(k+2)$ , it computes  $f(k)$  twice, besides computing  $f(k-1)$ ; it therefore takes at least 2 times as long as computing  $f(k)$ .
- Therefore, computing  $f(k+2m)$  takes at least  $2^m$  times as long as computing  $f(k)$ . This might not seem like a big deal, but consider that computing  $f(k+40)$  takes over a million times as long as computing  $f(k)$ .

You can see this, even on a modern<sup>3</sup> processor, by running the Python program in Example 1; the first 25 entries appear almost instantly, but things get visibly slow after that.

## Iterating in $n$

Programmers are unpleasantly surprised to find they’ve written such a slow routine, and I would typically guide them toward a solution replicating the computations they would do by hand. Basically one would jot down 0 and 1, then another 1 (since  $0 + 1 = 1$ ), then 2 (*i.e.*,  $1 + 1$ ), then 3 ( $1 + 2$ ), then 5 ( $2 + 3$ ), *etc.* In other words, you’d track the two previous values, and add them to get the next, something like Algorithm 2.

Now allow me to explain something about the loop. The first time through,  $k = 2$ ; we run it with  $k = 3$ ,  $k = 4$ , *etc.*, up to and including  $k = n$ .

Each time we enter the loop,  $back1 = f(k-1)$  and  $back2 = f(k-2)$ ; this is true the first time through (where  $k = 2$ ), and each time through we do the computation to keep it true. This is called the *loop invariant*.

At the bottom of the loop, we have  $runningTotal = f(k)$ , and in particular when we exit the loop (*i.e.*, when  $k = n$ ), we have  $runningTotal = f(n)$ , which is what we wanted.

The principle of finding the *loop invariant* is a powerful one in writing loops correctly. If each pass through the loop preserves the invariant, and if the loop is guaranteed to terminate, then the invariant will still be true upon exit. I don’t always remember to figure out the invariant, but when I do, the code is more likely to be correct; if I add a comment, it’s also easier to maintain.

<sup>3</sup>e.g., 2.3 GHz Intel Core i7

```

if  $n < 0$  then
  | return  $-1$ ;
else if  $n = 0$  then
  | return  $0$ ;
else if  $n = 1$  then
  | return  $1$ ;
else
  |  $runningTotal \leftarrow 1$ ;
  |  $back2 \leftarrow 0$ ;
  |  $back1 \leftarrow 1$ ;
  | for  $k \leftarrow 2$  to  $n$  do
    | /* Compute  $runningTotal \leftarrow f(k)$ . */
    | /* On entry  $back1 = f(k-1)$  */
    | /* and  $back2 = f(k-2)$  */
    |  $runningTotal \leftarrow back2 + back1$ ;
    |  $back2 \leftarrow back1$ ;
    |  $back1 \leftarrow runningTotal$ ;
  | return  $runningTotal$ ;

```

**Algorithm 2:** Iterative realization of  $f(n)$

## Precomputing?

Now, what if you need  $f(n)$  in approximately constant time? Could you just compute all reasonable values of  $f(n)$  ahead of time and just look them up in a table?

Most programmers react with at least mild surprise when I suggest this. They might ask if it’s worth all that space. “How much space would it take?” I’d ask. If that puzzled them I’d say, “Let’s suppose  $f(n)$  grew about as fast as  $2^n$ , how much space would we need?”

About as many entries as there are bits in the computer language’s `int` type. If  $f(n)$  doubles every time you bump  $n$  by 1, then around  $f(32)$  you’d be beyond the range of a 32-bit `int`; you’d thus need only about 32 entries in a table of precomputed values.

Then I’d ask, “What if it grew about as fast as  $2^{n/2}$ ?”

Usually they could see that it would be about twice the number of bits; that is, if  $f(n)$  doubles every time you bump  $n$  by 2, then we’d need no more than about 64 entries.

And the Fibonacci numbers do grow at least that fast. Since  $f(n) = f(n-2) + f(n-1)$ , and  $f(n-1) = f(n-2) + f(n-3)$ , we can see that  $f(n) = 2f(n-2) + f(n-3)$ ; that is  $f(n) \geq 2f(n-2)$ .

## The new solution

One day at work, maybe 10 years ago, I mentioned this programming problem. One of our senior people<sup>4</sup> sent me an algorithm I'd never seen before; it resembled Example 2. I had no idea how it could possibly work.

```
def f(n):
    if n < 0:
        return -1
    elif n < 2:
        return n
    else:
        a = 0
        b = c = 1
        x = 0
        y = 1
        while n:
            if n & 1:
                t = a*x + b*y
                y = b*x + c*y
                x = t
            t = a*b + b*c
            a = a*a + b*b
            c = b*b + c*c
            b = t
            n = n >> 1
        return x

for k in range(40):
    print k, f(k)
```

**Example 2:**  $O(\log n)$  realization of  $f(n)$

I ran his code and indeed it did work. But I still had no idea why.

I emailed him (it was a Saturday I think) and his reply overestimated my intelligence. Or sophistication. Probably both. Fortunately a web search helped me find an explanation I could actually understand.

The code relies on two big ideas. The first one is the observation that

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

which can be applied recursively:

$$\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} f_n \\ f_{n+1} \end{bmatrix} = \begin{bmatrix} f_{n+2} \\ f_{n+3} \end{bmatrix}$$

<sup>4</sup>Blake Lewis

and so on; indeed

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

The function definition from Example 2 is expressed more meaningfully in Algorithm 3.

```
/* What Example 2 means */
1 if n < 0 then
2   | return -1;
3 else if n = 0 then
4   | return 0;
5 else if n = 1 then
6   | return 1;
7 else
8   |  $\begin{bmatrix} a & b \\ b & c \end{bmatrix} \leftarrow \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}$ ; /* a = 0; b = c = 1 */
9   |  $\begin{bmatrix} x \\ y \end{bmatrix} \leftarrow \begin{bmatrix} 0 \\ 1 \end{bmatrix}$ ; /* x = 0; y = 1 */
10  |  $\begin{bmatrix} x \\ y \end{bmatrix} \leftarrow \begin{bmatrix} a & b \\ b & c \end{bmatrix}^n \begin{bmatrix} x \\ y \end{bmatrix}$ ; /* while n:... */
11  | return x;
```

**Algorithm 3:** Interpretation of Example 2

Note that line 10 expresses the entire “while n:” loop from Example 2; this is the second big idea.

This idea is that you can raise something to the  $n$ th power using  $O(\log n)$  multiplications, by successively shifting  $n$  right 1 bit and squaring a temporary variable.

Let’s consider calculating  $x^n$  in general. I mean, suppose you wanted to calculate  $3^8$ ; how would you do it?

You might just do 7 multiplications, which would be easy enough:

$$\begin{aligned} 3 \cdot 3 &= 9 \\ 9 \cdot 3 &= 27 \\ 27 \cdot 3 &= 81 \\ 81 \cdot 3 &= 243 \\ 243 \cdot 3 &= 729 \\ 729 \cdot 3 &= 2187 \\ 2187 \cdot 3 &= 6561 \end{aligned}$$

But suppose we wanted  $\pi^8$ ; far more efficient to calculate  $\pi^{2^{2^2}}$ . That is, to do 3 multiplications rather than 7 as shown in Algorithm 4.

We can generalize this for values of  $n$  that aren’t powers of 2, something like Algorithm 5.

```

set  $p \leftarrow \pi \cdot \pi$ ;
set  $p \leftarrow p \cdot p$ ;
return  $p \cdot p$ ;

```

**Algorithm 4:**  $\pi^8$  in just 3 multiplications

```

/* Set  $X \leftarrow A^n X$  */
1 while  $n > 0$  do
2   if  $n$  is odd then
3      $X \leftarrow A \cdot X$ ;
4    $n \leftarrow \lfloor n/2 \rfloor$ ;
   /* if  $n = 0$  then break; */
5    $A \leftarrow A^2$ ;

```

**Algorithm 5:**  $O(\log n)$  realization of  $A^n X$

The loop invariant for Algorithm 5 is that our desired result for  $X$  is always  $A^n \cdot X$ . It's true on entry to the loop, and it's true after line 5.

Each pass through the loop, we modify  $n$  and  $A$ , and maybe  $X$ ; the loop must terminate since  $n$  has only a finite number of bits. When  $n = 0$  we'll end the loop.

Let's run through it with  $X = 2$ ,  $A = 3$  and  $n = 5$ ; we'll calculate  $2 \cdot 3^5 = 486$ .

- The first time we enter the loop, we have  $X = 2$ ,  $A = 3$ ,  $n = 5$ .

Trivially, our desired value is  $A^n X$ . Here  $n$  is odd, so we set  $X \leftarrow X \cdot A$ ; now  $X = 6$ . We then set  $n \leftarrow 2$  and  $A \leftarrow 3^2 = 9$ .

Note that our desired value is still  $A^n X$ , because the calculations we've done compensate for each other:  $n$  is smaller but since  $A$  has been squared, we have  $9^2$ . Also,  $X$  was multiplied by  $A$ 's previous value; this makes up for the fact that when we divided  $n$  by 2 in line 4 we discarded its low-order bit.

- We enter again with  $X = 6$ ,  $A = 9$ ,  $n = 2$ .

Since  $n$  is even we skip line 3, so we leave  $X = 6$ . We set  $n \leftarrow 1$  and  $A \leftarrow 9^2 = 81$ .

Our desired value is still  $A^n X$ .

- We enter the third time with  $X = 6$ ,  $A = 81$ ,  $n = 1$ .

Since  $n$  is odd we set  $X \leftarrow X \cdot A = 486$  (line 3). When we shift  $n$  right one bit (line 4, it becomes zero.

At this point,  $X$  has reached our desired value. With  $n = 0$  we again have trivially that our desired value is  $A^n X$ .

## Computing $A^2$ where $A$ is a $2 \times 2$ matrix

Actually we're going to talk about the case where  $A$  is a particular kind of  $2 \times 2$  matrix viz., a matrix expressible as

$$\begin{bmatrix} a & b \\ b & c \end{bmatrix} \quad (2)$$

Recalling that matrix multiplication is row-by-column, we can see that

$$\begin{bmatrix} a & b \\ b & c \end{bmatrix}^2 = \begin{bmatrix} a^2 + b^2 & ab + bc \\ ab + bc & b^2 + c^2 \end{bmatrix} \quad (3)$$

Now, let's use (3) to show how Algorithm 5 applies to a  $2 \times 2$  matrix  $A$  and a  $2 \times 1$  vector  $X$ . The result is Algorithm 6.

```

/* Set  $\begin{bmatrix} x \\ y \end{bmatrix} \leftarrow \begin{bmatrix} a & b \\ b & c \end{bmatrix}^n \begin{bmatrix} x \\ y \end{bmatrix}$  */
1 while  $n > 0$  do
2   if  $n$  is odd then
3      $\begin{bmatrix} x \\ y \end{bmatrix} \leftarrow \begin{bmatrix} ax + by \\ bx + cy \end{bmatrix}$ 
4    $n \leftarrow \lfloor n/2 \rfloor$ ;
   /* if  $n = 0$  then break; */
5    $\begin{bmatrix} a & b \\ b & c \end{bmatrix} \leftarrow \begin{bmatrix} a^2 + b^2 & ab + bc \\ ab + bc & b^2 + c^2 \end{bmatrix}$ ;

```

**Algorithm 6:**  $A^n X$  for matrix  $A$ , vector  $X$

That's all very interesting (or not), but what can we do with it? Remember Algorithm 3, whose line 10 took an entire **while n:** loop from Example 2? Algorithm 6 is that loop, essentially.

So how would Algorithm 3 look if we replaced line 10 by Algorithm 6? Something like Algorithm 7.

Now some programming languages, like C, don't let you assign

$$\begin{bmatrix} x \\ y \end{bmatrix} \leftarrow \begin{bmatrix} ax + by \\ bx + cy \end{bmatrix} \quad (4)$$

in a single statement.

That, plus the fact that many programming languages don't support mathematical notation, makes the correspondence between Example 2 and Algorithm 7 not as clear and obvious as one might like.

```

/* Example 2 more fully explained */
1 if  $n < 0$  then
2   | return -1;
3 else if  $n = 0$  then
4   | return 0;
5 else if  $n = 1$  then
6   | return 1;
7 else
8   |  $\begin{bmatrix} a & b \\ b & c \end{bmatrix} \leftarrow \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix};$  /*  $a = 0; b = c = 1$  */
9   |  $\begin{bmatrix} x \\ y \end{bmatrix} \leftarrow \begin{bmatrix} 0 \\ 1 \end{bmatrix};$  /*  $x = 0; y = 1$  */
10  | while  $n > 0$  do
11    | if  $n$  is odd then
12      |  $\begin{bmatrix} x \\ y \end{bmatrix} \leftarrow \begin{bmatrix} ax + by \\ bx + cy \end{bmatrix};$ 
13      |  $n \leftarrow \lfloor n/2 \rfloor;$ 
14      | /* if  $n = 0$  then break; */
15      |  $\begin{bmatrix} a & b \\ b & c \end{bmatrix} \leftarrow \begin{bmatrix} a^2 + b^2 & ab + bc \\ ab + bc & b^2 + c^2 \end{bmatrix};$ 
16  | return  $x$ ;

```

**Algorithm 7:** Algorithm 3 with more details

A few notes on Algorithm 7:

- (4), *i.e.*, line 12, takes three lines in Example 2:

```

t = a*x + b*y
y = b*x + c*y
x=t

```

- $n \leftarrow \lfloor n/2 \rfloor$  is implemented in Example 2 as:

```

n = n >> 1

```

That is, we shift  $n$  right one bit to do the divide-and-truncate operation.

- Line 14 is implemented in Example 2 by

```

t = a*b + b*c
a = a*a + b*b
c = b*b + c*c
b=t

```

## Conclusion

If we compare Algorithm 2 *vs.* Algorithm 7, I would much prefer the former. It's straightforward and obvious, and thus much easier to understand and maintain than Algorithm 7, and it requires no multiplications.

One might imagine a circumstance wherein Algorithm 7 would be preferred, but I'm not sure what it would be.

That said, Algorithm 7 was really interesting to learn about, at least for me—over twenty years, a new solution!