Fibonnacci revisited

by Alexandru Lazăr

std@cwazy.co.uk

1 Introduction

I came across (by a silly mistake, but that's another story) an interesting way of computing Fibonacci series, which, unfortunately, is not taught as much as it should be.

The algorithm which is usually taught relies on the initial formulation given by Fibonacci (more correctly, Leonardo of Pisa):

$F(n) = 1, \text{if } n = 1, \text{or}$

$= F(n-1) + F(n-2), \text{ if } n > 1$

This formula gives us a rather trivial algorithm, which can either be put in its recursive form (like above) or easily translated into an iterative form. It's the algorithm you probably know from your CS classes and it's easy to implement as well.

The problem of this algorithm is, however, that in order to compute a term in the series, you need the values of all the terms before it. It's ok if we have to compute the first, say, 100 terms. However, when I tried to compute the first 100,000 terms, my 2.6 Ghz Pentium IV (hyper-threading enabled) worked for about one minute and 35 seconds. And what I was really trying to do was see how long would it take to compute the first 1,000,000 terms.

What we will look into this time is a way to compute the nth term in the Fibonacci series, requiring less steps than the usual algorithm.

2 Taking things in particular

Let's play a bit with Fibonacci numbers. More precisely, what would happen if, instead of having

$F(n) = F(n-1) + F(n-2)$

we would write:

$F(\text{n}) = \text{F(n-1)} + \text{F(n-2)} + \text{F(n-3)}$?

Well, the second formula actually gives us 3rd order Fibonacci numbers. You mean there are more Fibonacci numbers? Well, yes:

$F^{(k)}(n) = F^{(k)}(n-1) + F^{(k)}(n-2) + \ldots F^{(k)}(n-k)$, where:

$F^{(k)}(k-1) = 1 \text{ and } F^{(k)}(n) = 0 \text{ for any } 0 \leq n \leqslant k-2$ by definition.

It's easy to see that for k=2 we get the "usual" Fibonacci numbers which we all know.

Using this and replacing n with 2x and respectively 2x-1, we can end up with the rather well-known (by mathematicians) formulae:

$$F(2x-1) = (F(x-1))^2 + (F(x))^2$$

$$F(2x) = (2F(x-1) + F(x)) \cdot F(x) \qquad (1)$$

, with F(1) = F(2) = 1 by definition

The main difficulty here is that in this case, it's not directly appliable in a computer language. It's easy for us to realize that 2x and 2x-1 are actually "abstractions" of even or odd numbers, but for a compiler it's not that clear. It may also seem less efficient at a first look – many operations, many terms to calculate etc. But if you take a closer look, you will see that the number of steps decreases significantly. If we wanted to calculate F(1000), the "usual" approach would need to calculate all the terms before it. If we use these formulae, we only need to calculate 22 of them.

3 Implementation

Trying to implement this in C is not that hard either, if we make the right changes. This is simply a problem of changing the argument so that the compiler will be able to calculate things correctly.

Let n and k be natural numbers where n=2k. At first, we write (1) for the number k. Replacing 2k with n, we get the following:

$$F(n-1)=(F(\frac{n}{2}-1))^2+(F(\frac{n}{2}))^2$$
$$F(n)=(2F(\frac{n}{2}-1)+F(\frac{n}{2}))\cdot F(\frac{n}{2}) \qquad (2)$$

also keeping $F(1) = F(2) = 1$ by definition.

This is easier to follow. However, if we will implement this recursively, for the even number n-1 we will run into a rounding error problem. More precisely, taking the value of n=2k, we took away from the compiler the possibility of knowing exactly what value k is. For a clearer example, let n = 7, which is 2 x 4 -1, with k=4. In this case, we would have:

$F(\frac{n}{2}-1)=F(3.5-1)$, which would lead us to believe that k=3.5

Which is not exactly what we expected. However, since we're only dealing with natural numbers, the compiler will ignore anything after the decimal point. This is even worse because it would be like saying k=3, which would lead us to 2 x 3 - 1 = 5 instead of 7. You may be tempted to say that this is never going to happen because n = 2k and it's always even. But bear in mind that we're talking about implementing this recursively. If k is even, k -1 (which will be the argument for the following recursive step, i.e. n) is odd.

It's not hard to solve this problem. Since we will certainly not find an odd number which divides exactly by 2, we can be sure that n/2 will be exactly one unit smaller than k. Therefore, when writing things for the compiler, we will rewrite the first formula as:

$F(n-1)=(F(\frac{n}{2}))^2+(F(\frac{n}{2}+1))^2$

Note that this is not true mathematically (i.e. for a human). It's just a workaround.

Writing this as a recursive function in C should be fairly easy now:

```
long long int fibbo(long long int n) {
if (n==1) { return 1; }
if (n==2) { return 1; }
if ((n % 2) == 0) {
//Even
return ( (2 * fibbo(n/2 -1) + fibbo(n/2) ) * fibbo(n/2));
}
else {
//Odd. Fun part here
return ( (fibbo(n/2) * fibbo(n/2)) + (fibbo (n/2 + 1) * fibbo(n/2 + 1)));
}
}
```

Note that the usage of long long int is not mandatory. I used this just in case I would end up with very big numbers.

4 Conclusion

This algorithm has the advantage requiring a small number of values to be calculated. However, the recursive implementation is slow, but faster than the traditional recursive implementation. I found it to be at least 30% faster than the traditional recursive implementation on my computer. For the record, it is a 2.6 GHZ Pentium 4 with 1024 MB of RAM, running Gentoo Linux with GCC 3.4.