Calculating Fibonacci Numbers

Nitin Verma
mathsanew.com

May 30, 2020

The *Fibonacci Sequence* is defined as:

$$\begin{aligned} F_1 &= 1 \\ F_2 &= 1 \\ F_n &= F_{n-2} + F_{n-1} \quad \forall n > 2 \end{aligned} \tag{1}$$

So its first few terms are:

n	1	2	3	4	5	6	7	8	9	10	11	12	13	14
F_n	1	1	2	3	5	8	13	21	34	55	89	144	233	377

Given an integer $n > 2$, we need to write a program to calculate F_n.

Fibonacci numbers grow very fast with n and cannot be represented in 64-bits for n larger than 93:

$$\begin{aligned} F_{94} &= 19740274219868223167 \\ 2^{64} - 1 &= 18446744073709551615 \end{aligned}$$

So such a program needs to use additional software mechanism to represent these large numbers and perform operations on them. In this article, we do not discuss about handling large numbers and only write programs with integer type "long" in C. These programs can be adapted to use data-types and operations for large numbers, while keeping the algorithm same.

Sequential Algorithm

We will simply use the recurrence definition (1) of Fibonacci Numbers and sequentially calculate $F_3, F_4, F_5, \ldots, F_n$. Only the two latest calculated fibonacci numbers need to be remembered at any point, which will be added to obtain the next number of the sequence. Below is a portion of the C program for this. x and y remember the two latest fibonacci numbers.

```
int i;
unsigned long x, y, z;

x = 1;
y = 1;
i = 2;

while (i < n)
{
  z = x + y;
  x = y;
  y = z;
  i = i + 1;
}
```

We define the *Predicate* P as:

$$P : (x = F_{i-1} \ \wedge \ y = F_i)$$

P is true just before we enter the loop, because $F_1 = F_2 = 1$. It remains true after each iteration of the loop — P is the loop-invariant. So, when the loop terminates and i has become equal to n, $(i = n \ \wedge \ P)$ holds true. Further,

$$\begin{aligned} & (i = n \ \wedge \ P) \\ \Leftrightarrow \ & (i = n \ \wedge \ (x = F_{i-1} \ \wedge \ y = F_i)) \\ \Rightarrow \ & (y = F_n) \end{aligned}$$

Thus, we would receive the value of F_n in y.

This algorithm requires $n-2$ iterations, doing one addition of large numbers in each iteration. Thus it performs linear (in n) number of additions on large numbers. Though, such addition itself need not have constant time-complexity.

Improved Algorithm

In an earlier article titled "Some Proofs About Fibonacci Numbers", we arrived at below in Relation 4. For all $m > 1$:

$$F_{2m} = 2F_mF_{m+1} - {F_m}^2 \tag{2}$$
$$F_{2m+1} = {F_m}^2 + {F_{m+1}}^2 \tag{3}$$

Thus for n being even ($2m$) or odd ($2m+1$), we are able to express F_n in terms of two smaller fibonacci numbers which occur almost halfway earlier in the fibonacci sequence. If we know F_m and F_{m+1}, we only need four multiplications, one addition and one subtraction to calculate F_{2m} and F_{2m+1}. As mentioned earlier for Sequential Algorithm, we will need to handle large numbers and these operations over them.

Say $n = 2m$. What are all the smaller fibonacci numbers we need to calculate F_n? In what order should they be calculated? F_n can be calculated using (2) if we know F_m and F_{m+1}. Now, to calculate F_m and F_{m+1}, again we could use (2) or (3), and will need to know fibonacci numbers located almost halfway of m in the sequence. Thus an algorithm can be written to calculate F_n using recursion, starting at index n and keep doing its half at each recursion. We should also avoid repeated calculation of the same fibonacci number. As with any recursive algorithm, this too will require nested function calls and corresponding space on call-stack.

In this article, we discuss a non-recursive variant of the algorithm. Since $n > 2$, say the binary representation of n is: $(1b_1b_2b_3 \ldots b_k)$, where each b_i is 0 or 1. We have ignored all the starting 0 bits for simplicity. It is pleasant to observe that if we start with an integer $x = 1$, we can make it equal to n only by doubling and incrementing by 1, as in below C program. Variable $b[i]$ represents b_i.

```
int i, x;

x = 1;

for (i = 1; i <= k; i++)
{
  x = x * 2;

  if (b[i] == 1)
    x = x + 1;
}
```

We saw earlier that both F_{2m} and F_{2m+1} can be calculated just by knowing F_m and F_{m+1}. So, in above loop, whenever we update x to $x * 2$, we can calculate F_{2x} and F_{2x+1} if we know F_x and F_{x+1}.

We now add two variables Fx and $Fxp1$ (read "xp1" as "x plus 1"), and a temporary storage variable tmp to the above program. Our aim is to maintain the following predicate whenever we update x:

$$Q : (Fx = F_x \ \wedge \ Fxp1 = F_{x+1})$$

```
int i, x;
unsigned long Fx, Fxp1, tmp;

x    = 1;
Fx   = 1;
Fxp1 = 1;

for (i = 1; i <= k; i++)
{
  /* Block-1 start */

  x    = x * 2;
  tmp  = 2*Fx*Fxp1 - Fx*Fx;
  Fxp1 = Fx*Fx + Fxp1*Fxp1;
  Fx   = tmp;

  /* Block-1 end */

  if (b[i] == 1)
  {
    /* Block-2 start */

    x    = x + 1;
    tmp  = Fx + Fxp1;
    Fx   = Fxp1;
    Fxp1 = tmp;

    /* Block-2 end */
  }
}
```

Q is true just before we enter the loop, because $F_1 = F_2 = 1$. In Block-1 of the code, we calculate F_{2x} and F_{2x+1} from F_x and F_{x+1}, using equations (2) and (3). Variables Fx and $Fxp1$ are updated to reflect the new value of x. Thus Block-1 maintains Q.

In Block-2, we calculate F_{x+2} from F_x and F_{x+1} using recurrence definition (1), and variables Fx and $Fxp1$ are updated to reflect the new value of x. Thus Block-2 maintains Q.

Hence, each iteration of the loop, with $b[i]$ as 1 or 0, will maintain Q — Q is the loop-invariant. So, when the loop terminates and x has become equal to n, $(x = n \ \wedge \ Q)$ must hold true. Further,

$$\begin{aligned} & (x = n \ \wedge \ Q) \\ \Leftrightarrow \ & (x = n \ \wedge \ (Fx = F_x \ \wedge \ Fxp1 = F_{x+1})) \\ \Rightarrow \ & (Fx = F_n) \end{aligned}$$

Thus, we would receive the value of F_n in Fx.

Note that we can avoid calculating $Fx * Fx$ twice in each iteration by storing it in a variable. Also, instead of using bit-array $b[i]$, we can directly check a specific bit of n by maintaining a bit-mask.

The pair of consecutive fibonacci numbers Fx and $Fxp1$ is updated by each iteration to fibonacci numbers of almost double indices. So we may call the algorithm *Doubling Fibonacci Pair*.

This algorithm requires $\lfloor \log_2(n) \rfloor$ iterations, doing four multiplications, two additions and one subtraction of large numbers in each iteration. Thus it performs logarithmic (in n) number of operations on large numbers. These operations themselves need not have a constant time-complexity, and so the overall complexity will vary depending upon the implementation of large numbers.