

FAST FIBONACCI!

PAUL CULL[†], ADAM MURAKAMI[†], AND STEPHANIE YOUNG[†]

ABSTRACT. Fibonacci numbers have been discussed for nearly a thousand years. For reasonable values of n , the n^{th} Fibonacci number can be efficiently computed by the standard iterative algorithm, based on the Fibonacci recurrence relation. This method uses simple addition. For large values of n , algorithms that use multiplication are faster. We explore a variety of algorithms some based on the matrix representation, some based on Lucas numbers, and some based on the Chinese Remainder Theorem. We predict which methods should be fastest.

We used Maple to explore the run-times of several multiplication-based methods. We found that predicting run-times was made difficult by the fact that Maple uses a variety of multiplication algorithms, based on the size of the operands. We found that the product of Lucas numbers method is still the fastest known algorithm, although for large numbers, the Two-Term method based on the Chinese Remainder Theorem seems to be competitive.

1. INTRODUCTION

Fibonacci's rabbit problem (see [27] [10]) was a challenge with a point. The point was that the then-new-to-Europe Arabic numerals were superior to the Roman numerals for computation. Computing the 13th or even the 30th Fibonacci number is straight forward, all one has to do is repeatedly add the last two numbers to get the next number. While this repeated addition is easy using Arabic numerals, it is much more difficult using Roman numerals. Try computing the sequence I, I, II, III, V, VIII, XIII, XXI, XXXIV, etc.

In the millenium since Fibonacci, there have been many advances in mathematics, but have we found a better way to compute Fibonacci numbers? In this paper, we want to say **YES** there are better methods, but these methods may only become faster for very large Fibonacci numbers. To discuss this claim, we will need to consider models of computation and distinguish between the *arithmetic operation* model and the *bit operation* model. Further, we will need to distinguish between two related but different problems: one, computing the first n

[†]Research supported by NSF grant DMS-0139678.

elements of the Fibonacci sequence, and two, computing the n^{th} term of the Fibonacci sequence.

In the **arithmetic operation** model we assume that all operations take about the same time, and so we can get a reasonable time estimate by simply counting arithmetic operations. This model makes sense if the data items stay bounded, that is, as the size of the problem grows, the size of the operands does **not** grow. In particular, if each operand fits within a computer word, we may expect that the operations on these items will all take about the same time. In contrast in the **bit operation** model, the time for an operation depends on the size of the operands. Since numbers are normally represented in some base, and a number n has about $\log n$ digits, this is sometimes called the log-cost model. [1] [8]

As is traditional when analyzing algorithms, we will be looking at asymptotic complexity. Specifically, we will say that a running time is $O(g(n))$ if the time is bounded above by a constant times $g(n)$ as the size n grows, and that a running time is $\Theta(g(n))$ if the time is bounded both above and below by constant multiples of $g(n)$ as the size n grows. [6]

We will consider a variety of methods. While some of these are specific to the Fibonacci sequence, most of the methods apply to other sequences which are solutions to linear difference equations.

The representation does affect the complexity of computation. In particular, computing Fibonacci numbers in the Zeckendorf representation is very easy.[18] The Zeckendorf representation generalizes base notation by using Fibonacci numbers in place of powers of the base. Specifically, for any natural x ,

$$x = \sum_{i=2}^k b_i f_i \quad \text{where} \quad f_{k-1} < x \leq f_k,$$

where, f_i is the i^{th} Fibonacci number and each b_i is in $\{0, 1\}$. Since in Zeckendorf, f_k is represented as $[10 \cdots 0]$ where there are $k - 2$ 0's, computing f_k from k is trivial. In the following we will assume that we are representing numbers in binary, i.e. base 2.

2. COMPUTING FIBONACCI NUMBERS

Here we'll see a variety of methods ranging from the classical addition method to some recent methods based on the Chinese Remainder Theorem.

2.1. Classical Methods. The most classical method for computing Fibonacci numbers says:

start with $f_0 = 0$ and $f_1 = 1$,

and then iteratively use the formula $f_m = f_{m-1} + f_{m-2}$

$n - 2$ times to compute f_n . This is a very nice algorithm and it can be highly recommended for computing f_n when n is a reasonably small number. Not only does this algorithm compute f_n , but it also also computes the sequence $f_0, f_1, f_2, \dots, f_n$. At this stage it may not be clear whether f_n can be computed without computing all of the smaller Fibonacci numbers. In some ways this method seems to be the best possible since it computes f_n using only $n - 2$ additions, but as we'll see there are faster methods when we use other operations and consider the size of the Fibonacci numbers.

Binet's well known closed form formula for the Fibonacci numbers is

$$f_n = (\lambda_0^n - \lambda_1^n) / \sqrt{5}$$

where $\lambda_0 = (1 + \sqrt{5})/2$ and $\lambda_1 = (1 - \sqrt{5})/2$. [31] [20] Since $|\lambda_1|$ is small, $f_n \approx \lambda_0^n / \sqrt{5}$ and the number of bits in f_n is about γn where $\gamma = \log_2 \lambda_0$. Here, we can see that the number of bits is growing and hence, the bit operation model may be more appropriate.

While Binet gives us an approximation, in fact, $f_n = \text{Round}[\lambda_0^n / \sqrt{5}]$. (This rounding result holds more generally, Capocelli and Cull have shown that such rounding is possible for Generalized Fibonacci numbers and for a class of nonnegative difference equations. [4] [5]) The Binet formula also gives us a recursive method

$$f_{2n} = \lceil f_n \cdot \sqrt{5} \rceil.$$

In the arithmetic model this is an $O(\log n)$ method, but since f_n is growing and more correct bits of $\sqrt{5}$ are needed for a correct result, this method will be more complex in the bit operation model.

2.2. Matrix Methods. A sequence x_n which satisfies a difference equation

$$x_n = c_1 x_{n-1} + c_2 x_{n-2} + \dots + c_k x_{n-k}$$

has a corresponding companion matrix

$$M = \begin{pmatrix} c_1 & c_2 & \dots & c_{k-1} & c_k \\ 1 & 0 & \dots & 0 & 0 \\ 0 & \ddots & & 0 & 0 \\ \vdots & \ddots & & \vdots & \vdots \\ 0 & 0 & \dots & 1 & 0 \end{pmatrix}.$$

The n^{th} element of the sequence can be found by computing M^n or M^{n-k} and multiplying this calculated matrix times a vector with the appropriate initial conditions. At first glance this might seem silly since it is more complicated to take a matrix product than to compute one iteration of a difference equation. But matrix multiplication is associative, so

$$M^{2n} = (M^n)^2,$$

and M^n can be computed using about $\log n$ matrix squarings. As pointed out by Gries and Levin [17], since a $k \times k$ matrix can be squared using $O(k^3)$ arithmetic operations, $O(k^3 \log n)$ arithmetic operations suffice to find the n^{th} element of the sequence, and since k is constant with respect to n , $O(\log n)$ arithmetic operations suffice.

For Fibonacci numbers the situation is even better:

$$M = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \quad \text{and} \quad M^n = \begin{pmatrix} f_{n+1} & f_{n+1} \\ f_n & f_{n-1} \end{pmatrix}.$$

From the matrix product $M^{i+j} = M^i + M^j$ we have:

$$\begin{aligned} f_{i+j+1} &= f_{i+1}f_{j+1} + f_1f_j \\ f_{i+j} &= f_{i+1}f_j + f_if_{j-1} = f_if_{j+1} + f_{i-1}f_j \\ f_{i+j-1} &= f_if_j + f_{i-1}f_{j-1} \end{aligned}$$

which are sometimes called the Vorob'ev formulas. [31]

Also associated with a difference equation (or a matrix) is its characteristic polynomial $ch(x)$. Let $ch(x) = x^k - \sum_{i=1}^k c_i x^{k-i}$ and $p(x) = \sum_{i=1}^k c_i x^{k-i}$, then $[p(x)]^2 \bmod ch(x)$ will give the coefficients of M^{2k} in its expansion in terms of $M^{k-1}, M^{k-2}, \dots, I$. Obviously this procedure may be iterated and any power of the matrix may be computed. Specifically for Fibonacci numbers, the characteristic polynomial is $x^2 - x - 1$ and $p(x) = x + 1$ and iterating this procedure for e steps will produce $f_{2^e} M + f_{2^e-1} I$, or in polynomial form $f_{2^e} x + f_{2^e-1}$.

In the arithmetic model this method is better in the sense that it replaces the $O(k^3)$ matrix multiplication with $O(k^2)$ polynomial multiplication and reduction. As Petorossi [24] pointed out, this gives an $O(k^2 \log n)$ arithmetic method for finding the n^{th} element of a sequence defined by a k^{th} order difference equation.

Specializing the Vorob'ev formulas to $i = j = n$ gives a recursive algorithm for the Fibonacci numbers:

$$\begin{aligned} f_{2n} &= f_{n+1}f_n + f_n f_{n-1} = f_n (f_{n+1} + f_{n-1}) = f_n (f_n + 2f_{n-1}) \\ f_{2n-1} &= (f_n)^2 + (f_{n-1})^2. \end{aligned}$$

The same algorithm can be obtained from the characteristic polynomial, since

$$\begin{aligned} (f_{2^e}x + f_{2^e-1})^2 &= (f_{2^e})^2x^2 + 2f_{2^e}f_{2^e-1}x + (f_{2^e-1})^2 \\ &\equiv [(f_{2^e})^2 + 2f_{2^e}f_{2^e-1}]x + [(f_{2^e})^2 + (f_{2^e-1})^2] \end{aligned}$$

and so

$$\begin{aligned} f_{2^{e+1}} &= f_{2^e}[(f_{2^e}) + 2f_{2^e-1}] \\ f_{2^{e+1}-1} &= (f_{2^e})^2 + (f_{2^e-1})^2. \end{aligned}$$

A slightly faster algorithm can be constructed from these formulas. Use the formulas recursively to compute $f_{n/2}$ and $f_{n/2-1}$ and then use

$$f_n = (f_{n/2} + 2f_{n/2-1})f_{n/2}$$

to compute f_n . This will save some time and effort because f_{n-1} does not have to be computed.

2.3. Lucas Based Methods. The Lucas numbers are defined by

$$l_0 = 2, l_1 = 1, \quad l_n = l_{n-1} + l_{n-2}, \quad \text{for } n \geq 2.$$

Some of the first elements of the Lucas numbers sequence are: 2, 1, 3, 4, 7, 11, 18, 29, 47, 76, 123, 199, 322, 521... These numbers satisfy the formula

$$l_n = \lambda_1^n + \lambda_2^n,$$

and it's easy to show that

$$l_{2n} = l_n^2 - 2(-1)^n.$$

So if n is a power of 2, l_n can be computed by essentially $\log n$ squarings and f_n can be computed with $\log n$ more squarings because from

$$\frac{f_{2n}}{f_n} = \frac{\lambda_1^{2n} - \lambda_2^{2n}}{\lambda_1^n - \lambda_2^n} = \lambda_1^n + \lambda_2^n = l_n,$$

it follows that

$$f_{2^n} = \prod_{k=0}^{n-1} l_{2^k}.$$

These observations give us a very quick algorithm to compute f_n when n is a power of 2.

When n is not a power of 2, say $n = 2^i + k$, we can use the relation

$$f_{2^i+k} = f_{2^{i-1}+k}l_{2^{i-1}} - f_k$$

to compute f_n if we knew f_k and f_{k+2} and the sequence of Lucas numbers $l_{2^{i-1}}$. Of course, we can then compute f_k and f_{k+1} by the same technique. (We can find f_{k+2} by an addition.) Notice, that we do not

have to recompute the Lucas numbers, we can save them and reuse them.

2.4. CRT (Chinese Remainder Theorem) Based Methods. The well known Chinese Remainder Theorem (CRT) would seem to be a bad basis for fast Fibonacci methods because it requires the calculation of inverses and the use of several multiplications. [33] [22] [14] But the Fibonacci numbers are so special that some of these calculations reduced to multiplications by ± 1 . Here we will present a sequence of theorems which lead to two algorithms for computing Fibonacci numbers. One method is based on 3 consecutive Fibonacci numbers and the other method is based on 2 consecutive Fibonacci numbers.

Theorem 2.1. (*Chinese Remainder Theorem*) *Let m_1, m_2, \dots, m_n be integers greater than 0, relatively prime in pairs, and let r_1, r_2, \dots, r_n be any integers. Consider the system of congruences*

$$f_x \equiv r_1 \pmod{m_1}$$

$$f_x \equiv r_2 \pmod{m_2}$$

...

$$f_x \equiv r_n \pmod{m_n}.$$

Let $M = m_1 m_2 \cdots m_n$ and $M_i = \frac{M}{m_i}$ for $i = 1, 2, \dots, n$. Let y_i be a solution to

$$M_i y_i \equiv 1 \pmod{m_i} \quad \text{for } i = 1, 2, \dots, n.$$

Then a solution to the original system of congruences is

$$f_x = \sum_{i=1}^n M_i y_i r_i \pmod{M}.$$

Theorem 2.2. *If n is even, the Fibonacci sequence mod f_n has cycle length $2n$ and the sequence is*

$$0, 1, 1, f_3, \dots, f_{n-1},$$

$$0, f_{n-1}, -f_{n-2}, f_{n-3}, -f_{n-4}, \dots, -f_4, f_3, -1, 1.$$

If n is odd, then the cycle length is $4n$ and the sequence is

$$0, 1, 1, f_3, \dots, f_{n-1},$$

$$0, f_{n-1}, -f_{n-2}, f_{n-3}, -f_{n-4}, \dots, f_4, -f_3, 1, -1$$

$$0, -1, -1, -f_3, \dots, -f_{n-1},$$

$$0, -f_{n-1}, f_{n-2}, -f_{n-3}, f_{n-4}, \dots, -f_4, f_3, -1, 1.$$

Theorem 2.3. *Given an integer n , let the three pairwise prime values used in the Chinese Remainder Theorem m_1 , m_2 , and m_3 be f_n , f_{n+1} , and f_{n+2} respectively. Then if $M_i = \frac{m_1 m_2 m_3}{m_i}$ the solutions to the equation $M_i y_i \equiv 1 \pmod{m_i}$ for $i = 1, 2$, and 3 are*

$$\begin{aligned} y_1 &\equiv (-1)^n \pmod{f_n} \\ y_2 &\equiv (-1)^{n+1} \pmod{f_{n+1}} \\ y_3 &\equiv (-1)^{n+1} \pmod{f_{n+2}}. \end{aligned}$$

Theorem 2.4. *If $n = \lfloor \frac{x}{3} \rfloor + 1$ and $a = x \pmod{3}$ the remainders of f_x modulo f_n , f_{n+1} , and f_{n+2} are:*

$$\begin{aligned} r_1 &= (-1)^n f_{n-3+a} \\ r_2 &= (-1)^{n+1} f_{n-5+a} \\ r_3 &= (-1)^n f_{n-7+a}. \end{aligned}$$

The Three Term CRT Method

The x^{th} Fibonacci number, f_x , can be determined using the following set of equations:

$$\begin{aligned} n &= \lfloor \frac{x}{3} \rfloor + 1 \\ k &= x \pmod{3} \\ l &= \lceil \frac{k}{2} \rceil \\ m &= k \pmod{2} \\ f_x &= 5f_n f_{n-1} f_{n-2+k} + (-1)^{n+1} (-2)^{|l-1|} f_{n-1+3m-l}. \end{aligned}$$

Theorem 2.5. *If f_n and f_{n+1} are the moduli in the Chinese Remainder Theorem then the necessary inverses are:*

$$(y_1, y_2) = \begin{cases} (f_{n-1}, f_{n-1}) & \text{if } n \text{ is even.} \\ (f_{n-2}, f_n) & \text{if } n \text{ is odd.} \end{cases}$$

Theorem 2.6. *If $n = \lfloor \frac{x}{2} \rfloor + 1$ and $a = x \pmod{4}$ then the remainders r_1 , r_2 of $f_x \pmod{f_n}$ and $\pmod{f_{n+1}}$ are:*

$$\langle r_1, r_2 \rangle = \begin{cases} \langle 1, -3 \rangle & \text{if } a = 0. \\ \langle -1, 2 \rangle & \text{if } a = 1. \\ \langle -1, 3 \rangle & \text{if } a = 2. \\ \langle 1, -2 \rangle & \text{if } a = 3. \end{cases}$$

The Two Term CRT Method

The x^{th} Fibonacci number, f_x , can be determined using the following set of equations:

$$\begin{aligned} n &= \lfloor \frac{x}{2} \rfloor + 1 \\ m &= x + 1 \pmod{2} \\ k &= (x \pmod{2}) + 1 \\ l &= \lceil \frac{x}{2} \rceil \\ f_x &= f_n(2^m f_{n-m} + (-1)^k f_{n-2}) + (-1)^l. \end{aligned}$$

3. FASTER MULTIPLICATION

The standard method for multiplication of n bit numbers (or n^{th} degree polynomials) is an $\Theta(n^2)$ method. But faster methods are possible. Consider:

$$\begin{aligned} x &= a_0 + a_1 2^r \\ y &= b_0 + b_1 2^r, \end{aligned}$$

$$\text{then } x * y = a_0 * b_0 + (a_0 * b_1 + a_1 * b_0) 2^r + a_1 * b_1 2^{2r}.$$

This can be considered as a divide-and-conquer algorithm which uses 4 half size multiplications and some additions and shifts to compute a full size multiply. [9] Like the standard multiplication algorithm, this is a $\Theta(n^2)$ method. It can be made faster by using a trick to replace one of the half size multiplications. The trick is that

$$(a_0 + a_1) * (b_0 + b_1) - a_0 * b_0 - a_1 * b_1 = (a_0 * b_1 + a_1 * b_0).$$

So computing only the 3 half size multiplies

$$(a_0 + a_1) * (b_0 + b_1) \quad \text{and} \quad a_0 * b_0 \quad \text{and} \quad a_1 * b_1$$

will allow the full size product to be computed using some additional additions, subtractions, and shifts. This 3 half size method, sometimes called the Karatsuba–Ofman algorithm, is an $\Theta(n^{\log_2 3})$ method. and it will be faster than the standard method when the number of digits is large enough. [19]

But why stop here? There are faster methods. The key idea is to consider numbers as polynomials. The powers of the base serve as place holders just as the powers of x serve as place holders in a polynomial. A value of the product of two polynomials is the product of the corresponding values of the two polynomials. So, if one could choose some good points to evaluate at, one could evaluate the two polynomials, multiply corresponding values, and interpolate (invert the evaluation

process), to find the product polynomial. Toom and Cook [19] showed that there is a sequence of methods so that this sequence of methods has run time $O(n^{1+\epsilon})$ with $\epsilon > 0$ but arbitrarily small by choosing enough evaluation points. Schonhage and Strassen [25] showed that by using the Fast Fourier Transform in an appropriate ring there is an $\Theta(n \log(n) \log \log(n))$ method to multiply two n bit integers.

4. TIMING PREDICTIONS

One of the major goals of analysis of algorithms is to be able to predict the run time of programs. [9] This is usually done by calculating a theoretical run time in Big Theta or Big Oh form, and then running actual programs to find the appropriate time unit for the programs under the specific execution circumstances. These circumstances would include such factors as the speed of the processor, the programming language, and the operating system.

For our Fibonacci algorithms, we will attempt to compute the number of bit operations used. We expect the run times to be proportional to these counts but, as above, we cannot predict actual run times. On the other hand, these operation counts will allow us to predict that one algorithm will be faster than another when run under exactly the same conditions. In particular, several of the algorithms are based on multiplication, and to Big Theta order the run times of these algorithms will be the same, but if one algorithm uses two multiplications and the other uses only one multiplication, we will confidently predict that the algorithm using only one multiply will be faster.

The run time for the classic Addition algorithm is easy to predict. It is $\Theta(n^2)$ or more specifically $\gamma n(n-1)/2$ bit operations. This follows because $n-1$ additions are used to calculate f_i , and j bit operations are used to add j bit numbers, and f_i has γi bits. ($\sum_{i=1}^{n-1} \gamma i = \gamma n(n-1)/2$.)

The other algorithms we've considered all use multiplication of numbers with n/K bits, so if standard multiplication is used all of these methods have $\Theta(n^2)$ run times. So these faster methods might not be faster than the Addition algorithm. But, looking at bit counts, we will argue that even using standard multiplication, these methods are faster than the Addition algorithm.

Several of the faster methods work in two phases. There is a recursive phase in which various numbers are computed, and a final phase in which these numbers are combined to produce f_n . For the Lucas method and the Two Term CRT method, the recursion solves one problem of half size, and does two half size multiplications. This gives the

recurrence

$$T_R(n) = T_R\left(\frac{n}{2}\right) + 2M\left(\frac{n}{2}\right)$$

where T_R is the recursive time and $M(n)$ is the time to multiply n bit numbers. Assuming that $M(n) = n^k$, this recurrence gives

$$T_R(n) = \frac{2}{2^k - 1} n^k.$$

If the top level uses the results of the recursion up to $n/2$ and then one multiplication of size $n/2$, we have

$$\begin{aligned} T_{total}(n) &= T_R\left(\frac{n}{2}\right) + M\left(\frac{n}{2}\right) \\ &= \frac{2}{2^k - 1} \left(\frac{n}{2}\right)^k + \left(\frac{n}{2}\right)^k = \frac{2^k + 1}{2^k(2^k - 1)} n^k. \end{aligned}$$

Since the actual number of bits in f_n is γn rather than n , the predicted bit operations for these methods is

$$\frac{2^k + 1}{2^k(2^k - 1)} (\gamma n)^k$$

and when $k = 2$ (standard multiplication) this is

$$\frac{5}{12} \gamma^2 n^2,$$

and it's easy to see that this is fewer operations than used by the addition method.

For the Three Term CRT method we again have a recursion and a top level. In the recursion there is a call to the recursive procedure with size $n/3$, and there are also two multiplications of size $n/3$ and two multiplications of size $2n/3$. This gives the recurrence

$$T_R(n) = T_R\left(\frac{n}{3}\right) + 2M\left(\frac{n}{3}\right) + 2M\left(\frac{2n}{3}\right).$$

Assuming $M(n) = n^k$,

$$T_R(n) = \frac{2^{k+1} + 2}{3^k - 1} n^k.$$

At the top level three third-size values are multiplied which results in one size $n/3$ multiply and one size $2n/3$ multiply, and so:

$$\begin{aligned} T_{total}(n) &= T_R\left(\frac{n}{3}\right) + M\left(\frac{n}{3}\right) + M\left(\frac{2n}{3}\right) \\ &= \frac{(2^k + 1)(3^k + 1)}{3^k(3^k - 1)} n^k. \end{aligned}$$

Since the actual number of bits in f_n is γn rather than n , the predicted bit operations for this method is

$$\frac{(2^k + 1)(3^k + 1)}{3^k(3^k - 1)} (\gamma n)^k$$

and when $k = 2$ (standard multiplication) this is

$$\frac{25}{36} \gamma^2 n^2,$$

and it's easy to see that this is fewer operations than used by the addition method, but more operations than used by the Lucas or Two Term CRT methods.

5. EXPERIMENTS

Our theoretical predictions suggest that the Lucas based method will be best at least when n is a power of 2. Some years ago, we compared methods. The following table gives some of the timing results for methods we implemented in Ibuki Common Lisp and ran on a Sequent Balance 21000.

| Exponent | Addition | Binet | Lucas |
|----------|----------|-------|--------|
| 8 | 0.22 | 14.83 | 0.02 |
| 9 | 0.62 | 11.10 | 0.02 |
| 10 | 5.17 | 77.8 | 0.05 |
| 11 | 16.73 | 287.7 | 0.12 |
| 12 | 71.30 | — | 0.37 |
| 13 | 279.5 | — | 1.23 |
| 14 | 1133.6 | — | 4.40 |
| 15 | — | — | 17.80 |
| 16 | — | — | 74.00 |
| 17 | — | — | 280.2 |
| 18 | — | — | 1121.9 |

FIGURE 1. Timing Results using LISP. (Times in seconds.)

From our predictions all of these methods should be $\Theta(n^2)$. Since the n for each row is double the n in the previous row, we expect the run times to be 4 times the run times in the previous row. The Addition method shows this pattern for $n > 2^{11}$ and the Lucas method shows this pattern for $n > 2^{14}$. The Binet method is more erratic, but seems to have a ratio slightly bigger than 4.

Figure 2 shows the run times for four algorithms which were run using MAPLE. MAPLE was chosen because it automatically handles large integers. The data for the Addition method shows the approximate ratio of 4 which was expected, but it does seem to show increasing ratios which become about 4.2 for the largest n values that we tried. A real surprise was that the other methods did not show the expected ratio of 4. In fact the ratios seemed to be going to 2. We contacted MAPLE and found that they had implemented the GNU Multiprecision Arithmetic Library. [21] We checked the documentation for this library and found that it claimed to use Toom-Cook multiplication. [16] Since this is an $O(n^{1+\epsilon})$ sequence of methods, the ratios should approach $O(2^{1+\epsilon})$ which does seem to be consistent with the run time data.

| Exponent | Addition | fb2 | fb3 | lucas |
|----------|-------------|----------|----------|----------|
| 8 | 0.00044 | 0.00074 | 0.00048 | 0.00002 |
| 9 | 0.00088 | 0.00083 | 0.00066 | 0.00005 |
| 10 | 0.00259 | 0.00098 | 0.00070 | 0.00006 |
| 11 | 0.00642 | 0.00114 | 0.00077 | 0.00009 |
| 12 | 0.01506 | 0.00130 | 0.00105 | 0.00011 |
| 13 | 0.03477 | 0.00150 | 0.00102 | 0.00017 |
| 14 | 0.09558 | 0.00184 | 0.00142 | 0.00027 |
| 15 | 0.29811 | 0.00258 | 0.00236 | 0.00062 |
| 16 | 0.98001 | 0.00436 | 0.00423 | 0.00158 |
| 17 | 3.46880 | 0.00862 | 0.00989 | 0.00411 |
| 18 | 12.52145 | 0.02056 | 0.02566 | 0.01148 |
| 19 | 60.91000 | 0.05376 | 0.06412 | 0.03170 |
| 20 | 201.13000 | 0.11920 | 0.17980 | 0.07590 |
| 21 | 768.27000 | 0.26170 | 0.33060 | 0.17940 |
| 22 | 3203.81000 | 0.55720 | 0.81360 | 0.38300 |
| 23 | 13532.33000 | 1.26600 | 1.74500 | 0.88090 |
| 24 | — | 2.72800 | 3.90500 | 1.85900 |
| 25 | — | 6.32800 | 9.24500 | 4.37000 |
| 26 | — | 13.46200 | 20.12200 | 9.31700 |
| 27 | — | 27.86700 | 44.26400 | 20.49400 |

FIGURE 2. Timing Results using MAPLE. (Times in seconds.)

A visual check on these timings is shown in Figure 3. We plotted the log of run times for the 2-Term-CRT algorithm versus $\log n$. For comparison, we included the line $y = \log_2 3 \log n$. If the multiplication used was the Karatsuba–Ofman method, we would expect the time

curve to become parallel with the line. The plot shows that while the curve may be parallel to the line over part of its range, the trend is for the curve to have lower slope than the line indicating that a faster multiplication method was used. The data seem to be consistent with the assumption that MAPLE is using the Toom-Cook multiplication method.

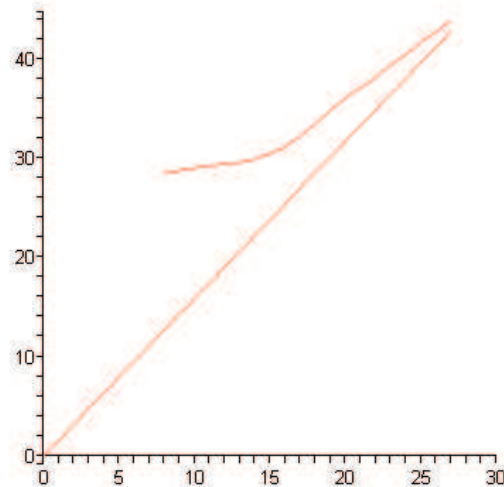


FIGURE 3. Plot of $\log T(n)$ versus $\log n$, along with line $y = \log_2 3 \log n$

6. CONCLUSION

There are a multitude of methods for computing Fibonacci numbers. The traditional addition based method will be good for small numbers (or if one wants all the Fibonacci numbers up to some bound), but for finding single large Fibonacci numbers, multiplication based methods are faster. These multiplication based methods become even faster when fast multiplication methods for large integers are used. Standard mathematical packages like MAPLE, MATLAB, and MATHEMATICA have implemented the fast multiplication methods using the GNU library. So, it is reasonably easy to get large Fibonacci numbers quickly using these packages. The only difficulty we found was running out of space on the computers we used.

Our results indicate that the Lucas method and the Two Term CRT method were fastest.

REFERENCES

- [1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, MA, 1974.
- [2] R. M. Capocelli. A Generalization of Fibonacci Trees. In G. E. Bergum, A. N. Philppou, and A. F. Horadam, editors, *Applications of Fibonacci Numbers, Volume 3*, pages 37–56. Kluwer, Dordrecht, 1990.
- [3] R. M. Capocelli, editor. *Sequences*. Springer-Verlag, New York City, NY, 1990.
- [4] R. M. Capocelli and P. Cull. Generalized Fibonacci Numbers are Rounded Powers. In G. E. Bergum, A. N. Philppou, and A. F. Horadam, editors, *Applications of Fibonacci Numbers, Volume 3*, pages 57–62. Kluwer, Dordrecht, 1990.
- [5] R. M. Capocelli and P. Cull. Rounding the solutions of Fibonacci-like difference equations. *Fibonacci Quarterly*, 41:133–141, 2003.
- [6] T. Cormen, A. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms(second edition)*. McGraw-Hill, Boston, MA, 2001.
- [7] P. Cull and J. Holloway . Computing Fibonacci Numbers Quickly. *Information Processing Letters*, 32:143–149, 1989.
- [8] P. Cull. Analysis of algorithms. In L. M. Ricciardi, editor, *Lectures in Applied Mathematics and Informatics*, pages 1–61. Manchester University Press, 1990.
- [9] P. Cull. *A Brief Introduction to Algorithms*. Department of Computer Science, Oregon State University, 2004.
- [10] P. Cull, M. Flahive, and R. Robson. *Difference Equations: from Rabbits to Chaos*. Springer, New York, 2005.
- [11] P. Cull and E.F. Ecklund Jr. Towers of Hanoi and Analysis of Algorithms. *American Mathematical Monthly*, 92(6):407–420, June-July 1985.
- [12] M. C. Er. A fast algoritm for computing order- k Fibonacci numbers in log time. *Computer Journal*, 26:224–227, 1983.
- [13] M. C. Er. Computing sums of order- k Fibonacci numbers in log time. *Information Processing Letters*, 17:1–5, 1983.
- [14] C. Vanden Eynden. *Elementary Number Theory (second edition)*. McGraw-Hill, New York City, NY, second edition, 2001.
- [15] M. Feinberg. Fibonacci-Tribonacci. *The Fibonacci Quarterly*, 1:71–74, 1963.
- [16] Torbjörn Granlund. Gnu mp. *The GNU Multiple Precision Arithmetic Library*. ed. 4.1.4:86–92, 2004.
- [17] D. Gries and G. Levin . Computing Fibonacci numbers (and similarly defined functions) in log time. *Information Processing Letters*, 11:68–69, 1980.
- [18] V. E. Hoggatt Jr. and M. Bicknell. Generalized Fibonacci Polynomials and Zeckendorf’s Representations. *The Fibonacci Quarterly*, 11:399–419, 1973.
- [19] D. E. Knuth. *The Art of Computer Programming*. Addison-Wesley, New York City, NY, third edition, 1997.
- [20] T. Koshy. *Fibonacci and Lucas Numbers*. Wiley-Interscience, New York City, NY, 2001.
- [21] Technical Support Maplesoft. Question regarding maple. Email to Stephanie Young., July 2005.
- [22] Mathworld. Chinese Remainder Theorem. *Mathworld [website]* <http://mathworld.wolfram.com>.
- [23] E. P. Miles. Generalized Fibonacci Numbers and Associated Matrices. *American Mathematical Monthly*, 67:745–757, 1960.

- [24] A. Petorossi. Derivation of an $O(k^2 \log n)$ for computing order- k Fibonacci numbers from the $O(k^3 \log n)$ matrix multiplication method. *Information Processing Letters*, 11:172–179, 1980.
- [25] A. Schonhage and V. Strassen. Schnelle Multiplikation grosser Zahlen. *Computing*, 7:281–292, 1971.
- [26] J. Shortt. An iterative program to calculate Fibonacci numbers in $O(\log n)$ arithmetic operations. *Information Processing Letters*, 7:299–303, 1977.
- [27] L. Sigler. *Fibonacci's Liber Abaci*. Springer-Verlag, New York City, NY, 2002.
- [28] W. R. Spickerman. Binet's Formula for the Tribonacci Sequence. *The Fibonacci Quarterly*, 20:118–120, 1982.
- [29] Daisuke Takahashi. A fast algorithm for computing large fibonacci numbers. *Information Processing Letters*, 75:243–246, 2000.
- [30] F. J. Urbanek. An $O(\log n)$ algorithm for computing the n^{th} element of the solution of a difference equation. *Information Processing Letters*, 11:66–67, 1980.
- [31] N. N. Vorobev. *Fibonacci Numbers*. Blaisdell, New York City, NY, 1961.
- [32] D. D. Wall. Fibonacci series modulo m . *American Mathematical Monthly*, 67:525–532, 1960.
- [33] Wikipedia. Chinese Remainder Theorem. *Wikipedia [website]*
<http://www.wikipedia.org/wiki/>.

COMPUTER SCIENCE, OREGON STATE UNIVERSITY, CORVALLIS, OR 97331
E-mail address: `pc@cs.orst.edu`

COMPUTER SCIENCE, OREGON STATE UNIVERSITY, CORVALLIS, OR 97331

SANTA CLARA UNIVERSITY, SANTA CLARA, CALIFORNIA 95053
E-mail address: `stephinsalamanca@gmail.com`