FASTER FIBONACCI, KILL! KILL!

STEPHANIE YOUNG

ADVISOR:PAUL CULL
OREGON STATE UNIVERSITY

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 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. It is 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 quite competitive.

1. INTRODUCTION

This research was inspired by the work of Adam Murakami [1], a participant in the 2004 REU program at Oregon State University. Murakami was investigating a method for computing Fibonacci numbers based on the Chinese Remainder Theorem (CRT), an idea that was first explored by Babb. Babb was one of the very first participants in this program on Oregon State's campus, and he left behind very little documentation of his work and results.

Last year, Murakami reviewed the remnants of Babb's notes and attempted to expand them– with the addition of proofs– into a complete, cohesive report. Unfortunately, the time trials at the end of the report offered results that were not consistent with his earlier calculations of the bit operations used by each of his algorithms. Furthermore, in his conclusion Murakami expressed surprise at the factor of 3 time increase upon doubling the input size of the algorithms. Assuming that his system was using an $O(n^2)$ method, one would have expected the run-time of his algorithms to increase by a multiple of 4 upon doubling the input size. This paper offers corrected calculations of the bit operations required for each algorithm, as well as supplies an explanation for the factor of 3 time increase.

We begin by defining the Fibonacci sequence and supplying a closed formula for computing the n^{th} Fibonacci number. Then we review Murakami's timing results and supply an explanation for the factor of 3 time increase. A brief description of the multiplication algorithms used by Maple is discussed. Next, we review the iterative method, Two-Term and Three-Term CRT methods, and the product of Lucas numbers method for computing Fibonacci numbers. A Maple program will be provided for each one, as well as a calculation of the bit operations used. Finally, we will review and analyze my own timing results for all four algorithms.

Date: 12 August, 2005.

This work was done during the Summer 2005 REU program in Mathematics at Oregon State University.

1.1. **The Fibonacci Sequence.**

Definition 1.1. *The* Fibonacci Sequence *is defined as the sequence of numbers with* $f_0 = 0$, $f_1 = 1$, *and* $f_n = f_{n-1} + f_{n-2}$ *for* $n \geq 2$.

Example 1.2. *Some of the first elements of the Fibonacci sequence are: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233...*

The characteristic polynomial associated with the Fibonacci sequence is f(x)=x^2-x-1. We call the roots of this equation λ_1 and λ_2, where

$$\lambda_1 = \frac{1}{2}(1+\sqrt{5}) \text{ and } \lambda_2 = \frac{1}{2}(1-\sqrt{5}).$$

Binet's formula [CH] for computing the n^{th} Fibonacci number is

$$f_n = \frac{1}{\sqrt{5}}(\lambda_1^n - \lambda_2^n).$$

Although this is a "closed" formula, it is still somewhat difficult to use because of the $\sqrt{5}$ term, and since λ_1 and λ_2 are irrational.

1.2. **Algorithms.** This paper will review and compare four algorithms for computing Fibonacci numbers: the iterative method, the two methods by Murakami, and the product of Lucas numbers method, which is currently the fastest known algorithm for computing Fibonacci numbers. In order to measure the efficiency of each algorithm, we will calculate the number of bit operations used. Since f_n is asymptotic to $\frac{\lambda_1^n}{\sqrt{5}}$, our analysis should use something like γn to represent the number of bits in f_n, where $\gamma = \log_2 \lambda_1 \approx 0.69424$. That is, since the value of f_n can be approximated by $\frac{\lambda_1^n}{\sqrt{5}}$ for large values of n, taking $\log_2$ of that value will give us the number of bits. However, we will denote this term n for the sake of simplicity. We will focus our analysis on calls for multiplication, since that is the operation which most significantly contributes to the number of bit operations used by each algorithm.

Most of the algorithms we are dealing with can be conveniently thought of as two phase operations. There is a *top* phase which calculates the final result, and there is a *recursive* phase which (recursively) calculates the information needed to compute the final result.

Murakami claimed that his two-term and three-term CRT methods used the same number of bit operations as the product of Lucas numbers method, which suggested that all three algorithms ought to be equally efficient on a computer. However, this did not hold when he actually timed each of the procedures. This report will review all four algorithms and attempt to determine the correct number of bit operations used by each.

1.3. **Multiplication.** In the section of his report entitled *Timing*, Murakami included a table of results of the time (in seconds) that it took Maple to compute Fibonacci numbers using three different programs. The first program featured the iterative method, and the second and third had algorithms based on the Two-Term and Three-Term CRT methods. This table can be viewed in Figure 1.

In the table, the programs are labelled, "itfib," "fib2," and "fib3," respectively. Murakami ran out of time before implementing a working product of Lucas numbers program. For each timing, the

inputs increase exponentially by powers of 2. So, for example, $f_{2^{19}}$ took just under 11 minutes to compute using the iterative method, but it was computed in about 1 second and 1.5 seconds using the CRT-based methods.

Exponent	itfib	fib2	fib3
8	0.002085	0.002435	0.00179
9	0.005105	0.00303	0.0022
10	0.01223	0.003745	0.002585
11	0.0402	0.00451	0.003337
12	0.106615	0.005445	0.00405
13	0.261125	0.007345	0.005945
14	0.778185	0.011085	0.01085
15	2.624305	0.020575	0.02379
16	9.53832	0.04708	0.061565
17	36.428915	0.124955	0.17064
18	141.515	0.347255	0.50022
19	647.397	1.01283	1.47754
20	3060.648	3.0075	4.4224

FIGURE 1. Murakami's Timing Results

For the last third of these results, one can see that the time begins to increase by approximately a factor of 3, upon doubling the input sizes. This result surprised Murakami, who was under the impression that Maple used an $O(n^2)$ multiplication algorithm. If this were the case, the time increase upon doubling input sizes should have been much closer to a factor of 4. It is worth noting that we do see the factor of 4 increase for the iterative method, since this method relies on addition rather than multiplication.

I contacted Maplesoft, the company that distributes Maple software, and was informed that Maple actually has numerous algorithms available for multiplication [M]. These algorithms are based on the GNU Multiple Precision Arithmetic Library (GMP) [TG]. The algorithm that is used for any particular call is based on the size of the operand. If numbers are small enough to fit into a single computer word, then multiplication is carried out by a single machine (computer) instruction. So for small-enough numbers, we expect the time to multiply to be constant, independent of the numbers themselves. When numbers are larger, they can be broken into smaller numbers and the products of the large numbers can be easily calculated from various products of these smaller components.

What follow are brief descriptions of three multiplication algorithms used by Maple. They are ordered beginning with the algorithm that would be used for small input sizes and ending with the algorithm that would be used for largest input sizes.

For each algorithm, let our goal be to multiply two polynomials in base 2,

$$A = a_0 + a_1 2 + \cdots + a_{n-1} 2^{n-1}$$

and

$$B = b_0 + b_1 + \cdots + b_{n-1} 2^{n-1}.$$

1.3.1. *Classic Multiplication.* This algorithm organizes our numbers A and B such that $A = A_0 + A_1 2^{\frac{n}{2}}$ and $B = B_0 + B_1 2^{\frac{n}{2}}$, where

$$\begin{aligned} A_0 &= a_0 + a_1 2 + \cdots + a_{\frac{n}{2}-1} 2^{\frac{n}{2}-1}, \\ A_1 &= a_{\frac{n}{2}} + a_{\frac{n}{2}+1} 2 + \cdots + a_{n-1} 2^{\frac{n}{2}-1}, \\ B_0 &= b_0 + b_1 2 + \cdots + b_{\frac{n}{2}-1} 2^{\frac{n}{2}-1}, \text{ and} \\ B_1 &= b_{\frac{n}{2}} + b_{\frac{n}{2}+1} 2 + \cdots + b_{n-1} 2^{\frac{n}{2}-1}. \end{aligned}$$

Multiplication is accomplished using the FOIL method:

$$\begin{aligned} A * B &= [A_0 + A_1 2^{\frac{n}{2}}] * [B_0 + B_1 2^{\frac{n}{2}}] \\ &= A_0 * B_0 + [A_0 * B_1 + B_0 * A_1] 2^{\frac{n}{2}} + A_1 * B_1 2^n \end{aligned}$$

It is important to note that multiplication by powers of 2 are really just shifts in base 2, so this operation requires little work. Basecase multiplication, as described here, is an $O(n^2)$ algorithm.

1.3.2. *Karatsuba (3-Half Size) Multiplication.* This method organizes our polynomials A and B into parts A_0, A_1, and B_0, B_1, as in the Basecase Multiplication. The algorithm continues in the following fashion:

$$\begin{aligned} \text{Let } m_1 &= [A_0 + A_1] * [B_0 + B_1] = A_0 * B_0 + A_0 * B_1 + B_0 * A_1 + A_1 * B_1, \\ m_2 &= A_0 * B_0, \text{ and} \\ m_3 &= A_1 * B_1. \\ \text{Then } A * B &= m_2 + [m_1 - m_2 - m_3] 2^{\frac{n}{2}} + m_3 2^n. \end{aligned}$$

Karatsuba is asymptotically an $O(n^{\log_2 3})$ (or $\approx O(n^{1.585})$) method, where $\log 3$ comes from the three multiplications and the base-2 accounts for their being half the size of the input.

1.3.3. *Fast Fourier Transform (FFT).* Last, we have the Fast Fourier Transform (FFT) method. The key to this method is to think of the operands as being polynomials rather than numbers. A polynomial with n coefficients can be represented by its coefficients or by the values of the polynomial at n distinct points. The value of a product polynomial $PQ(x_i)$ at x_i is just the product of $P(x_i)$ and $Q(x_i)$. So the products can be computed easily in value-space. Such an algorithm becomes useful if "evaluation" and "interpolation" can be done quickly. These transformations can be done quickly using the Fast Fourier Transform, which is based on evaluation at the k^{th} roots of unity, where k is a power of 2.

The transform follows the pattern that is illustrated in Figure 2 [C]. It is asymptotically an $O(n \log n \log \log n)$ method.

In light of these algorithms, we are able to review Murakami's timing results and conclude that the switch to the factor of 3 time increase corresponds to the multiplication algorithm switching to the Karatsuba method. That is, upon doubling the input size we have

$$\frac{T(2n)}{T(n)} = (2)^{\log_2 3} = 3.$$

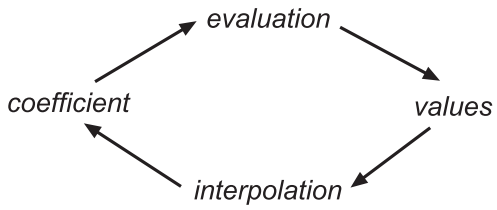

FIGURE 2. Model of Fast Fourier Transform

It is important to bear in mind each of these multiplication methods when computing the bit operations used by a particular algorithm, as the total bit operations used will vary depending on the input size and corresponding multiplication algorithm. Now, let us consider our algorithms for computing Fibonacci numbers.

2. ITERATIVE METHOD

The iterative method is by far the most elementary of the algorithms for the Fibonacci sequence that we will explore. It simply uses the difference equation to compute the desired terms. That is, it remains faithful to the elementary idea of determining the n^{th} term of the sequence by adding the two previous terms. This procedure is repeated until the desired term is achieved.

2.1. **Algorithm.** The algorithm that uses the iterative method to produce the n^{th} Fibonacci number is programmed in Maple as shown below. In order for the algorithm to be compatible with the needs of Murakami's two-term and three-term CRT methods, this particular algorithm outputs the n^{th} and $(n+1)^{st}$ Fibonacci numbers.

```
itfib:=proc(n) local a, b, i;
    a:=0;
    b:=1;
    if n < 1 then
        return (0,1)
    elif n = 1 then
        return (1,1)
    else
        for i from 2 to n do
            if a < b then a:=a+b else b:=a+b end if;
        end do;
        if a > b then return (a,a+b) else return (b,a+b) end if;
    end if;
end proc;
```

2.2. **Bit Operations.** According to a report by Cull and Holloway [CH], the number of bit operations used by this method to compute f_n is

$$\frac{\gamma n(n-1)}{2}.$$

So this predicts that for large-enough n, $\frac{T(2n)}{T(n)} \approx 4$. This prediction is verified by the timings in Figures 1 and 3.

3. Two-Term CRT Method

The algorithms explored by Murakami are based on the Chinese Remainder Theorem. While this report does not intend to reproduce all of his work, a general understanding of the Chinese Remainder Theorem would be beneficial in comprehending the basis of his algorithms.

Theorem 3.1. *Let m_1, m_2, ... , m_n be integers greater than 0, pairwise relatively prime, and let r_1, r_2, ..., r_n be any integers. Assuming $0 \leq f \leq M$, consider the following system of congruences:*

$$\begin{aligned} f &\equiv r_1 (\text{mod } m_1) \\ f &\equiv r_2 (\text{mod } m_2) \\ &\vdots \\ f &\equiv r_n (\text{mod } m_n). \end{aligned}$$

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

$$M_i y_i \equiv 1 (\text{mod } m_i) \text{ for } i = 1, 2, ..., n.$$

Then a unique solution to the original system of congruences is

$$f = \sum_{i=1}^{n} M_i y_i r_i (\text{mod } M).$$

Both the two-term and the three-term CRT methods for computing Fibonacci numbers rely on the fact that the Fibonacci sequence yields a periodic orbit modulo f_n, where f_n is a Fibonacci number ≥ 2. This allows us to easily compute terms that are required by the CRT. Furthermore, the simple Fibonacci difference equation causes many terms of the orbit to turn out being congruent to 1 or -1, which greatly reduces the overall number of operations in the algorithm.

The two-term CRT method computes the desired Fibonacci number by using two systems of congruences with relatively prime moduli and applying the CRT.

3.1. **Algorithm.** Murakami programmed his two-term method for computing Fibonacci numbers in a recursive fashion. The algorithm calls on itself until reaching a lower limit L (supplied by the user), at which point it calls on the iterative method for the desired values. This algorithm is programmed in Maple as shown below. Once again, the algorithm outputs both the n^{th} and $(n+1)^{st}$ terms.

```
fib2:=proc(n) local nb, i, k, l, m, s, t, a, b;
    if n < L then
        itfib(n)
    else
        nb:=floor(1/2*n)+1;
        (f(0),f(1)):=fib2(nb-2);
        for i from 2 to 4 do
            f(i):=f(i-1)+f(i-2)
```

```
        end do;
        for i from 0 to 1 do
            s:=floor(1/2*n+1/2*i)+1;
            k:=‘mod‘(n+i,2)+1;
            l:=ceil(1/2*n+1/2*i);
            m:=‘mod‘(n+i+1,2);
            t:=f(s-nb+2)*(2^{m}*f(s-nb+2-m)+(-1)^{k}*f(s-nb))+(-1)^{l};
            a:=b;
            b:=t;
        end do;
    return (a,b);
    end if;
end proc;
```

3.2. **Bit Operations.** This algorithm features one recursive call, with a half-size input. It appears as though there is only one multiply and some shifts and minor additions/subtractions, but it actually loops through the procedure twice. So there are two multiplies in total. Thus, we represent the time for the recursion by

$$T_2(n) = T_2(\frac{n}{2}) + 2M(\frac{n}{2}),$$

where $M(n)$ is the time it takes to multiply two n-bit numbers. Let k represent the multiplication variant. Then assuming $M(n) = n^k$ and $T_2(n) = \alpha n^k$ gives us

$$\begin{aligned} \alpha n^k &= \alpha(\frac{n}{2})^k + 2(\frac{n}{2})^k \\ 2^k\alpha &= \alpha + 2 \\ \alpha &= \frac{2}{2^k - 1}. \end{aligned}$$

At the top level we have a single multiplication of two half-size values. So we say

$$T_1(n) = M(\frac{n}{2}).$$

Combining the top and recursive levels, we have total time for the algorithm represented by the equation

$$\begin{aligned} T(n) &= T_1(n) + T_2(\frac{n}{2}) \\ &= M(\frac{n}{2}) + \frac{2}{2^k - 1}(\frac{n}{2})^k. \end{aligned}$$

3.2.1. *Bit Operations with Classic Multiplication.* For classic multiplication, we have $k = 2$. Inserting this into the formula above, we calculate the total number of bit operations with this multiplication to be

$$(\frac{n}{2})^2 + \frac{2}{2^2 - 1}(\frac{n}{2})^2 = (\frac{n}{2})^2 + \frac{2}{3}(\frac{n}{2})^2 = \frac{5}{12}n^2.$$

3.2.2. *Bit Operations with Karatsuba Multiplication.* For Karatsuba multiplication, we have $2^k = 3$. This substitution results in a total count of bit operations to be

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

3.2.3. *Bit Operations with FFT Multiplication.* To calculate the bit operations used during FFT multiplication, we still have our time equation for recursion being

$$T_2(n) = T_2(\frac{n}{2}) + 2M(\frac{n}{2}).$$

Then assuming $M(n) \approx n\log n$ and $T_2(n) = \beta n\log n$ gives us

$$\beta n\log n = \beta\frac{n}{2}\log\frac{n}{2} + 2\frac{n}{2}\log\frac{n}{2}$$
$$2\beta\log n = \beta\log\frac{n}{2} + 2\log\frac{n}{2}$$
$$\text{So } \beta \approx 2 \text{ and } T_2 \approx 2n\log n.$$

At the top level, we still have a single multiplication of two half-size values. So we say

$$T_1(n) = M(\frac{n}{2}) = \frac{n}{2}\log\frac{n}{2}.$$

In total, we have

$$T(n) = T_1(n) + T_2(\frac{n}{2})$$
$$\approx \frac{n}{2}\log\frac{n}{2} + 2\frac{n}{2}\log\frac{n}{2} \approx \frac{3}{2}n\log\frac{n}{2}.$$

4. Three-Term CRT Method

The three-term CRT method differs from the two-term method by using three (instead of two) systems of congruences with relatively prime moduli and applying the CRT.

4.1. **Algorithm.** This algorithm varies only slightly from the two-term CRT algorithm. Because of the additional congruence, slightly different calculations were required to solve for the values used by the CRT. Once again, the algorithm outputs the n^{th} and $(n+1)^{st}$ Fibonacci terms. The algorithm is programmed in Maple as follows:

```
fib3:=proc(n) local nb, a, b, i, k, l, m, s, t;
    if n < L then
        itfib(n)
    else
        nb:=floor(1/3*n)+1;
        (f(0),f(1)):=fib3(nb-2);
        for i from 2 to 4 do
            f(i):=f(i-1)+f(i-2)
        end do;
        for i from 0 to 1 do
            s:=floor(1/3*n+1/3*i)+1;
            k:=‘mod‘(n+i,3);
```

```
            l:=ceil(1/2*k);
            m:='mod'(k,2)
            t:=5*f(s-nb+2)*f(s-nb+1)*f(s-nb+k)+(-1)^{n+1}*(-2)^{abs(l-1)}*
               f(s-nb+1+3*m-l);
            a:=b;
            b:=t;
        end do;
    return (a,b);
    end if;
end proc;
```

4.2. **Bit Operations.** As with the two-term case, Murakami claimed that the three-term algorithm required $\frac{5}{12}\gamma^2 n^2$ bit operations. The later timings showed that, not only was the three-term method slower than the product of Lucas numbers method, but it was also slightly slower than the two-term method, given a large-enough input. What follows is a recalculation of the bit operations used for this method, determined in the same fashion as above.

Once again, this algorithm features a recursive call with two loops. Each loop multiples three third-size inputs. Let us assume that, to multiply these three terms, we first multiply just two of them and get a value of about size $\frac{2n}{3}$, and then multiply that product by the last $(\frac{n}{3})$-size term. Then we say that the total time for recursion is

$$T_3(n) = T_3(\frac{n}{3}) + 2M(\frac{n}{3}) + 2M(\frac{2n}{3}).$$

$$\text{Assuming } M(n) = n^k \text{ and } T_3(n) = \alpha n^k, \text{ we find}$$

$$\begin{aligned}
\alpha n^k &= \alpha(\frac{n}{3})^k + 2(\frac{n}{3})^k + 2(\frac{2n}{3})^k \\
3^k\alpha &= \alpha + 2 + 2^{k+1} \\
\alpha &= \frac{2^{k+1}+2}{3^k-1}.
\end{aligned}$$

At the top level we have a single multiplication of three third-size values. So we say

$$T_1(n) = M(\frac{n}{3}) + M(\frac{2n}{3}) = (\frac{n}{3})^k + (\frac{2n}{3})^k.$$

Combining the top and recursive levels, we have

$$\begin{aligned}
T(n) &= T_1(n) + T_2(\frac{n}{3}) \\
&= (\frac{n}{3})^k + (\frac{2n}{3})^k + \frac{2^{k+1}+2}{3^k-1}(\frac{n}{3})^k.
\end{aligned}$$

4.2.1. *Bit Operations with Classic Multiplication.* For classic multiplication, we have that $k = 2$. Inserting this into the formula above, we calculate the total number of bit operations with this multiplication to be

$$(\frac{n}{3})^2 + (\frac{2n}{3})^2 + \frac{2^{2+1}+2}{3^2-1}(\frac{n}{3})^2 = \frac{n^2}{9} + \frac{4n^2}{9} + \frac{10}{8}\frac{n^2}{9} = \frac{25}{36}n^2.$$

4.2.2. *Bit Operations with Karatsuba Multiplication.* For Karatsuba multiplication, we have $2^k = 3$. This substitution results in a total count of bit operations to be

$$\begin{aligned}(\frac{n}{3})^k + (\frac{2n}{3})^k + \frac{2^{k+1}+2}{3^k-1}(\frac{n}{3})^k &= [1+2^k+\frac{2*3+2}{3^k-1}](\frac{n}{3})^k \\ &= \frac{8+4(3^k-1}{3^k-1}(\frac{n}{3})^k \\ &= \frac{4(3^k+1)}{3^k(3^k-1)}n^k \\ &\approx 0.986n^{\log_2 3}.\end{aligned}$$

4.2.3. *Bit Operations with FFT Multiplication.* To calculate the bit operations used during FFT multiplication, we use the same time equation for recursion as above:

$$T_3(n) = T_3(\frac{n}{3}) + 2M(\frac{n}{3}) + 2M(\frac{2n}{3}).$$

Then assuming $M(n) \approx n\log n$ and $T_2(n) = \beta n\log n$ gives us

$$\begin{aligned}\beta n\log n &= \beta\frac{n}{3}\log\frac{n}{3} + 2\frac{n}{3}\log\frac{n}{3} + 2\frac{2n}{3}log\frac{2n}{3} \\ 3\beta\log n &= \beta\log\frac{n}{3} + 2\log\frac{n}{3} + 4\log\frac{2n}{3} \\ \text{So } 3\beta &\approx \beta+2+4 \\ \text{and } \beta &\approx \frac{6}{2} = 3.\end{aligned}$$

At the top level, we still have a single multiplication of three third-size values. So we say

$$T_1(n) = M(\frac{n}{3}) + M(\frac{2n}{3}) \approx \frac{n}{3}\log\frac{n}{3} + \frac{2n}{3}\log\frac{2n}{3}.$$

In total, we have

$$\begin{aligned}T(n) &= T_1(n) + T_3(\frac{n}{3}) \\ &\approx \frac{n}{3}\log\frac{n}{3} + \frac{2n}{3}\log\frac{2n}{3} + 3\frac{n}{3}\log\frac{n}{3} \\ &\approx [\frac{1}{3}+\frac{2}{3}+1]n\log\frac{n}{3} = 2n\log\frac{n}{3}.\end{aligned}$$

This means that for large values of n, the time ratio between the Two-Term and Three-Term CRT methods comes out to approximately

$$\frac{fib3(n)}{fib2(n)} \approx \frac{2\log\frac{n}{3}}{\frac{3}{2}\log\frac{n}{2}} \approx \frac{4}{3}.$$

5. Product of Lucas Numbers Method

This method is based on the Lucas numbers, which are defined by

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

Example 5.1. *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...*

It is worth noting that the sequence of Lucas numbers satisfies the Fibonacci difference equation. Lucas numbers can be computed quickly using the following formula [CH]:

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

with λ_1 and λ_2 the same as those associated with the Fibonacci sequence. Since

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

it follows that

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

With a few minor alterations, this concept can be expanded to find the value of any Fibonacci term. However, since Murakami worked with inputs that were powers of two, I wrote my Maple program to use this formula.

5.1. **Algorithm.** Last summer, Murakami ran out of time before he could implement a working product of Lucas numbers algorithm. The algorithm I constructed uses the product formula listed above. Unlike the earlier algorithms that I have described, this one calls for an input that is just the exponent of the desired power of two term. That is, to compute the $(2^k)^{th}$ Fibonacci number, one would only input the k-value, rather than the entire 2^k term.

```
lucas:=proc(k)
local i, l, f:
f:=1:
l:=3:
    for i from 2 to k-1 do
        f:=f*l:
        l:=l*l-2:
    end do:
f:=f*l;
return(f);
end proc;
```

5.2. **Bit Operations.** The basic idea with this method is that Fibonacci numbers can be computed as products of several Lucas numbers. In the recursive phase, both a Fibonacci number and a Lucas number are calculated. If we look at the algorithm, we see that we compute the Fibonacci number by multiplying the previous Fibonacci number by the previous Lucas number. The new

Lucas number is computed by squaring the previous Lucas number and subtracting two. Letting $M(n)$ represent the time it takes to multiply, our time equation for recursion is

$$T_2 = T(\frac{n}{2}) + 2M(\frac{n}{2}).$$

Assuming $M(n) = n^k$ and $T(n) = \alpha n^2$, we find

$$\begin{aligned} \alpha n^k &= \alpha(\frac{n}{2})^k + 2(\frac{n}{2})^k \\ 2^k\alpha &= \alpha + 2 \\ \alpha &= \frac{2}{2^k - 1}. \end{aligned}$$

At the top level, we simply multiply two half-size numbers. So the corresponding equation is

$$T_1(n) = M(\frac{n}{2}) = (\frac{n}{2})^k.$$

Combining the top and recursive levels, we find the equation for total time:

$$\begin{aligned} T(n) &= T_1(n) + T_2(\frac{n}{2}) \\ &= (\frac{n}{2})^k + \frac{2}{2^k - 1}(\frac{n}{2})^k. \end{aligned}$$

5.2.1. *Bit Operations with Classic Multiplication.* For classic multiplication, we have $k = 2$. Substituting this into the above formula, we get the total number of bit operations to be

$$\begin{aligned} T(n) &= (\frac{n}{2})^2 + \frac{2}{2^2 - 1}(\frac{n}{2})^2 \\ &= [\frac{1}{4} + \frac{1}{6}]n^2 \\ &= \frac{5}{12}n^2. \end{aligned}$$

5.2.2. *Bit Operations with Karatsuba Multiplication.* For this multiplication, we have $2^k = 3$. Thus, we calculate the total number of bit operations to be

$$\begin{aligned} T(n) &= (\frac{n}{2})^k + \frac{2}{2^k - 1}(\frac{n}{2})^k \\ &= \frac{n^k}{3} + \frac{2}{2}\frac{n^k}{3} \\ &= \frac{2}{3}n^k. \end{aligned}$$

5.2.3. *Bit Operations with FFT Multiplication.* To calculate the bit operations used with FFT multiplication, we still have our time equation for recursion being

$$T_2(n) = T_2(\frac{n}{2}) + 2M(\frac{n}{2}).$$

Then assuming $M(n) \approx n\log n$ and $T_2(n) = \beta n\log n$ gives us

$$\beta n\log n = \beta\frac{n}{2}\log\frac{n}{2} + 2\frac{n}{2}\log\frac{n}{2}$$
$$2\beta\log n = \beta\log\frac{n}{2} + 2\log\frac{n}{2}$$
$$\text{So } \beta \approx 2 \text{ and } T_2 \approx 2n\log n.$$

At the top level, we still have a single multiplication of two half-size values. So we say

$$T_1(n) = M(\frac{n}{2}) = \frac{n}{2}\log\frac{n}{2}.$$

In total, we have

$$T(n) = T_1(n) + T_2(\frac{n}{2})$$
$$\approx \frac{n}{2}\log\frac{n}{2} + 2\frac{n}{2}\log\frac{n}{2} \approx \frac{3}{2}n\log\frac{n}{2}.$$

6. TIMING

Figure 3 offers a table of the time (in seconds) that it took Maple to run each of the algorithms described in this paper, according to inputs with sizes that increase exponentially by powers of two. For very small input sizes, I created a looping function that performed the desired procedure as many as 1,000 times. That allowed me to divide the total resulting time by 1,000, which resulted with a fairly accurate timing for a single iteration of the procedure. Upon reaching inputs of size 2^{24}, the iterative method took an unreasonably long amount of time to compute. Therefore, the last four rows in that column are blank.

6.1. **Analysis.**

One might note that my timing results vary considerably from Murakami's timings of the same algorithms. This is hardly surprising, since I ran my experiments on a different computer as well as with an updated version of Maple. What did surprise me was the fact that the constant factor of three did not appear as the ratio between consecutive timings, as it had in Murakami's results. This led me to believe that my version of Maple either didn't call on the Karatsuba multiplication algorithm for any of my computations, or it did, but only for select input sizes lying somewhere between a pair of consecutive powers of two.

In order to test the validity of the latter possibility, I considered the equation

$$T(n) = cn^k,$$

where $T(n)$ is the time an algorithm takes to compute given an input, k is the multiplication variant, and c is the leading coefficient. Then

$$\log T(n) = \log c + k\log n.$$

For large values of n, this approximates to:

$$\log T(n) = k\log n.$$

This equation tells me that, on a plot of $\log(T(n))$ versus $\log(n)$, the value of the slope over a given interval would be approximately equal to the multiplication variant for computations over

Exponent	itfib	fib2	fib3	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	.026170	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 3. Timing Results

the interval. I created such plots for the Two-Term and Three-Term CRT algorithms, as well as for the product of Lucas numbers algorithm. The results for all three varied only slightly, so for the purpose of saving space I will include and analyze only the resulting graph of the Two-Term CRT algorithm. Figure 4 shows that graph.

One may notice that on the intervals $[8,13]$, $[18,19]$, and $[19,26]$ the curve seems to assume nearly-linear slopes. This suggests that the multiplication variant was constant over the intervals for input sizes $[2^8,2^{13}]$, $[2^{18},2^{19}]$, and $[2^{19},2^{26}]$. The next step was to fit lines to these intervals in order to determine the slope (and corresponding k-value) of each. I was able to produce and graph three line fits using Maple, which are displayed in Figure 5.

6.1.1. *Line Fit Analysis.* The straight line fit over the first interval has a slope-value of 0.207. This can be understood by the fact that, for small input values, Maple can do single-word multiplications in constant (almost zero) time. We can conjecture that additional timings for inputs smaller than 2^8 would yield an even smaller slope.

The slope over the second interval has a value of 1.387, which is rather close to the value of $\log_2 3$. Since this interval was characterized by only two end-point values $n_1 = 2^{18}$ and $n_2 = 2^{19}$, it seemed very likely that the desired 1.585-slope (which would identify Karatsuba multiplication) could be approached if I computed more timings along this interval. I found that line-fitting to a plot over the interval of inputs [340000, 380000] yielded a slope-value of 1.588. With a little more tinkering, the desired 1.585-slope might very well be accomplished.

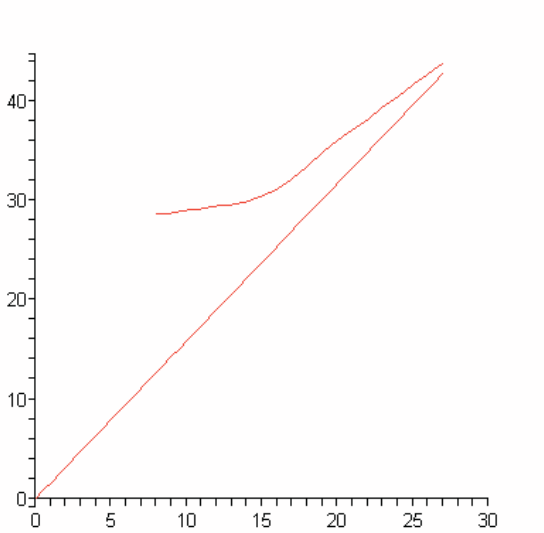

FIGURE 4. Plot of $\log T(n)$ versus $\log n$, along with line $y = log_2 3$

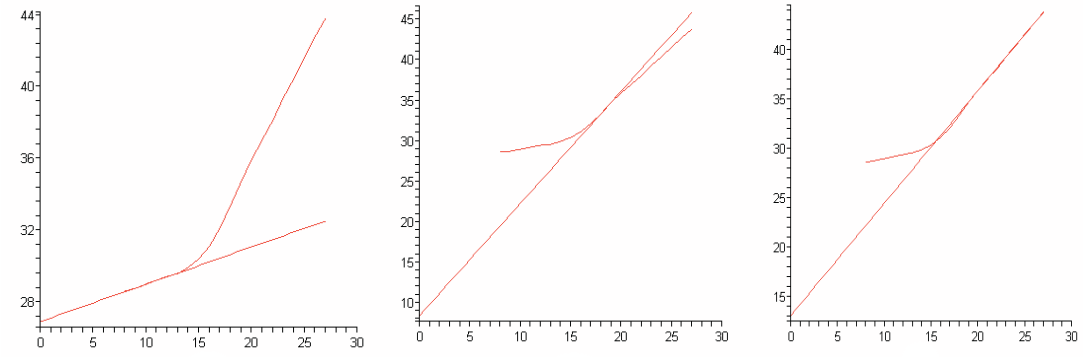

FIGURE 5. Lines y = 26.927 + 0.207x, y = 8.436 + 1.387x, and y = 13.118 + 1.140x, respectively

The straight line fit over the third interval offered a slope of 1.140. This suggests that, beginning with input values of approximately 2^{19}, Maple shifts to an even faster multiplication algorithm.

7. CONCLUSION

According to my calculations, the Two-Term CRT algorithm uses the same number of bit operations as the product of Lucas numbers algorithm. Although my timings show that the Lucas algorithm was faster for input values up to 2^{27}, if multiplication really is the dominant operation then we can expect the algorithms to be nearly equally efficient for very large n-values. As for my timings, we can account for the time difference between the two algorithms by the fact that the

product of Lucas numbers algorithm is very clean and concise. That is, it does not feature nearly the amount of intermediary computations that are necessary in the CRT-based algorithm.

The iterative method behaves just as we expected, with a factor of 4 time increase once the Fibonacci numbers became too large to fit within a single computer word. Additionally, the Three-Term CRT method was slower than the Two-Term and product of Lucas numbers methods, just as our calculations of the bit operations suggested.

Although I am pleased with the results of this research, it would have been nice to have been able to produce time results for still-larger inputs. Unfortunately, my Maple program refused to operate for inputs larger than 2^{27}, and I could not alter the settings to allow computations beyond this level. Further timings might verify some of my conclusions that are, at this point, still conjectures.

Last, it was incredibly educating to discover Maple's various multiplication algorithms, and to see concretely the impact they had on the outcomes of my timings.

REFERENCES

[AM] Murakami, Adam. "Computing Fibonacci Numbers Fast using the Chinese Remainder Theorem." *Proceedings of the Research Education for Undergraduates Program in Mathematics*, 2003: 155-83. NSF and Oregon State University.

[M] Maplesoft, Technical Support. "Question regarding Maple." Email to Stephanie Young. 1 July 2005.

[TG] Granlund, Torbjörn. "GNU MP." *The GNU Multiple Precision Arithmetic Library*, ed. 4.1.4, 2004: 86-92.

[C] Cull, Paul. "A Brief Introduction to Algorithms." 2004: 36. Department of Computer Science, Oregon State University.

[CH] Cull, Paul and Holloway, James L. "Computing Fibonacci Numbers Quickly." Information Processing Letters 32, 1989.

SANTA CLARA UNIVERSITY
E-mail address: stephinsalamanca@gmail.com