A fast algorithm for computing large Fibonacci numbers

Daisuke Takahashi

Department of Information and Computer Sciences, Saitama University, 255 Shimo-Okubo, Urawa-shi, Saitama 338-8570, Japan

Received 13 March 2000; received in revised form 19 June 2000

Communicated by K. Iwama

Abstract

We present a fast algorithm for computing large Fibonacci numbers. It is known that the product of Lucas numbers algorithm uses the fewest bit operations to compute the Fibonacci number $F_n$. We show that the number of bit operations in the conventional product of Lucas numbers algorithm can be reduced by replacing multiplication with the square operation.

Keywords: Program derivation; Fibonacci numbers

1. Introduction

Many algorithms for computing Fibonacci numbers have been well studied [11,12,5,8,4,3,7,9,2]. It is known that the product of Lucas numbers algorithm uses the fewest bit operations to compute $F_n$. We show that the number of bit operations in the conventional product of Lucas numbers algorithm can be reduced by replacing multiplication with the square operation.

1.1. Fibonacci and Lucas numbers

We define the Fibonacci numbers as

$$F_0 = 0, \quad F_1 = 1,$$
$$F_{n+2} = F_{n+1} + F_n, \quad n \geq 0.$$  (1)

The Lucas numbers are defined as

$$L_0 = 2, \quad L_1 = 1,$$
$$L_{n+2} = L_{n+1} + L_n, \quad n \geq 0.$$  (2)

We also have the formulas

$$F_n = \frac{\alpha^n - \beta^n}{\alpha - \beta},$$  (3)
$$L_n = \alpha^n + \beta^n,$$  (4)

where $\alpha = (1 + \sqrt{5})/2$ and $\beta = (1 - \sqrt{5})/2$.

We will use $\gamma n$ to represent the number of bits in $F_n$ where $\gamma = \log_2 \alpha \approx 0.69424$ since $F_n$ is asymptotic to $\alpha^n/5$ [2].

E-mail address: daisuke@ics.saitama-u.ac.jp (D. Takahashi).
1.2. Multiplication and square operations

It is known that multiplication of \( n \)-bit numbers can be performed in \( O(n \log n \log \log n) \) bit operations by using the Schönhage–Strassen algorithm \([10]\) which is based on the fast Fourier transform (FFT) \([1]\). In multiplication of many thousand bits or more, FFT-based multiplication is the fastest.

In this paper, we use FFT-based multiplication for computing large Fibonacci numbers. We denote the number of bit operations used to multiply two \( n \)-bit numbers as \( M(n) \), and the number of bit operations used to square an \( n \)-bit number as \( S(n) \). Also, we assume \( M(n) = O(n \log n \log \log n) \) and \( S(n) = O(n \log n \log \log n) \). In general, we have \( S(n) \leq M(n) \).

From
\[
ab = \frac{(a + b)^2 - (a - b)^2}{4}
\]
we have \( M(n) \leq 2S(n) \).

Then, we have
\[
\frac{1}{2}M(n) \leq S(n) \leq M(n).
\]

In FFT-based multiplication and square operations, computation of FFTs consumes the most computation time. Although three FFTs are needed to multiply two \( n \)-bit numbers, only two FFTs are needed to square an \( n \)-bit number. Therefore we assume \( S(n) \approx \frac{3}{2}M(n) \) in FFT-based multiplication and square operations.

Since the number of bit operations used to add/subtract two \( n \)-bit numbers is \( O(n) \) and the number of bit operations used to multiply/divide an \( n \)-bit number by an \( O(1) \)-bit number is also \( O(n) \), no consideration of these is made in this paper.

2. Conventional product of Lucas numbers algorithm

Cull and Holloway \([2]\) presented the product of Lucas numbers algorithm.

The identity
\[
F_{2k} = F_k L_k
\]
follows directly from (3) and (4).

From (4),
\[
\text{fib}(n)
\]
\[
f \leftarrow 1
\]
\[
l \leftarrow 3
\]
for \( i = 2 \) to \( \log_2 n - 1 \)
\[
f \leftarrow f \times l
\]
\[
l \leftarrow l \times l - 2
\]
\[
f \leftarrow f \times l
\]
return \( f \)

Fig. 1. Conventional product of Lucas numbers algorithm \([2]\).

\[
L_{2k} = \alpha^{2k} + \beta^{2k}
\]
\[
= (\alpha^k + \beta^k)^2 - 2(\alpha \beta)^k
\]
\[
= L_k^2 - 2 \cdot (-1)^k.
\]

We can compute \( F_{2^i} \) \([2]\) by
\[
F_{2^i} = \prod_{j=0}^{i-1} L_{2^j}.
\]

The conventional product of Lucas numbers algorithm to compute \( F_{n} \) \((n = 2^i, i \geq 2)\) is shown in Fig. 1. One multiplication and one square operation are needed in the conventional product of Lucas numbers algorithm in each iteration.

The number of bit operations used by the conventional product of Lucas numbers algorithm is as follows:
\[
T(n) = \sum_{i=2}^{\log_2 n - 1} \left(M(\gamma \cdot 2^{i-1}) + S(\gamma \cdot 2^{i-1})\right)
\]
\[
+ M(\gamma \cdot n/2) \leq \frac{5}{2}M(\gamma n).
\]

3. Presented product of Lucas numbers algorithm

In this section, we show that the number of bit operations in the conventional product of Lucas numbers algorithm can be reduced by replacing multiplication with the square operation.

The identities
\[
F_{n+m} = \frac{1}{2}(F_n L_m + F_m L_n),
\]
\[
L_{n+m} = \frac{1}{2}(L_n L_m + 5F_n F_m)
\]
follow from (3) and (4).

Then, the identities
\[
F_{k+1} = \frac{1}{2}(F_k + L_k),
\]
\[
L_{k+1} = \frac{1}{2}(F_k + L_k)
\]

return \( f \)
That is, we can compute $F$ follows from (6) and (13). Also, $L$ follows from (5), (11) and (13). Furthermore, the identity $F_{2k} = F_k L_k$ follows from (9) and (10). From (3) and (4),

$$L_k^2 = (\alpha^k + \beta^k)^2 = 5\left(\frac{\alpha^k - \beta^k}{\alpha - \beta}\right)^2 + 4(\alpha\beta)^k = 5F_k^2 + 4 \cdot (-1)^k. \quad (13)$$

Then, the identity $F_{2k} = F_k \cdot L_k$ follows from (5), (11) and (13).

Since $F_k^2$ is a number that should be computed in (14) and (15), the multiplication of $F_{2k} = F_k L_k$ will be substantially obtained from the calculation of $F_{k+1}^2$. That is, we can compute $F_{k+1}^2$ instead of $F_k L_k$.

The presented product of Lucas numbers algorithm to compute $F_n$ ($n = 2^i$, $i \geq 2$) is shown in Fig. 2. In the algorithm shown in Fig. 2, although a pair of Fibonacci and Lucas numbers is calculated in the $for$ loop, we need only $F_n$ after the $for$ loop. Therefore we use (5) instead of (14) after the $for$ loop.

```python
fib(n)
    f ← 1
    l ← 3
    for i = 2 to log_2 n - 1
        temp ← f * f
        f ← (f + l)/2
        f ← 2 * (f * f) - 3 * temp - 2
        l ← 5 * temp + 2
        f ← f * l
    return f
```

Fig. 2. Presented product of Lucas numbers algorithm.

$$L_{k+1} = \frac{1}{2}(5F_k + L_k) = F_{k+1} + 2F_k \quad (12)$$

The number of bit operations used by the presented product of Lucas numbers algorithm is as follows:

$$T(n) = \sum_{i=2}^{\log_2 n-1} 2S(\gamma \cdot 2^{i-1}) + M(\gamma \cdot n/2) \leq \frac{7}{2}M(\gamma n). \quad (16)$$

Table 1 compares the number of operations in each iteration of the product of Lucas numbers algorithms. We note that the presented algorithm requires only two square operations in each iteration.

We conclude that our algorithm is better than the conventional product of Lucas numbers algorithm if $S(n) < M(n)$.

To compute $F_n$ for arbitrary $n$, we can use the binary method for exponentiation [6]. We need four numbers: $F_{k+1}, L_{k+1}, F_{2k}$, and $L_{2k}$ where $k \geq 0$. These numbers can be obtained from (11), (12), (14), and (15).

As with the algorithm in Fig. 2, although a pair of Fibonacci and Lucas numbers is calculated in the $for$ loop, we need only $F_n$ finally. If $n$ is even, we can use (5) to compute $F_{2k}$ where $k = n/2$. On the other hand, if $n$ is odd, we need a number of $F_{2k+1}$ where $k = \lfloor n/2 \rfloor$. We have derived a formula for computing $F_{2k+1}$ in terms of $F_{k+1}$ and $L_k$ with one multiplication:

$$F_{2k+1} = \frac{1}{2}(F_{2k} + L_{2k}) = \frac{1}{2}(F_k L_k + L_k^2 - 2 \cdot (-1)^k) = F_{k+1} L_k - (-1)^k. \quad (17)$$

The product of Lucas numbers algorithm to compute $F_n$ for arbitrary $n$ is presented in Fig. 3. Some additions, subtractions, multiplications by 2 and divisions by 2 are added to this algorithm compared with the presented algorithm shown in Fig. 2.
fib(n)
if n = 0 return 0
else if n = 1 return 1
else if n = 2 return 1
else
  f ← 1
  l ← 1
  sign ← −1
  mask ← 2[log2n]−1
  for i = 1 to [log2n] − 1
    temp ← f * f
    f ← (f + l)/2
    f ← 2 * (f * f) − 3 * temp − 2 * sign
    l ← 5 * temp + 2 * sign
    sign ← 1
    if (n & mask) ≠ 0
      temp ← f
      f ← (f + l)/2
      l ← f + 2 * temp
      sign ← −1
      mask ← mask/2
    if (n & mask) = 0
      f ← f + l
else
  f ← (f + l)/2
  f ← f + l − sign
return f

Fig. 3. Presented product of Lucas numbers algorithm to compute \( F_n \) for arbitrary \( n \).

References