ELSEVIER

Information Processing Letters 75 (2000) 243–246

Information Processing Letters

www.elsevier.com/locate/ipl

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 [2].

In this paper, we present a fast algorithm for computing large Fibonacci numbers. This algorithm is based on the product of Lucas numbers algorithm [9, 2]. The conventional product of Lucas numbers algorithm uses multiplication and square operation. In general, it is known that the number of bit operations used to square an n-bit number is less than that used to multiply two n-bit numbers. Thus, 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

$$F_0 = 0, \qquad F_1 = 1,$$
$$F_{n+2} = F_{n+1} + F_n, \quad n \geqslant 0. \tag{1}$$

The Lucas numbers are defined as

$$L_0 = 2, \qquad L_1 = 1,$$
$$L_{n+2} = L_{n+1} + L_n, \quad n \geqslant 0. \tag{2}$$

We also have the formulas

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

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

We will use γn 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).

0020-0190/00/$ – see front matter
PII: S0020-0190(00)00112-5

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 $S(n) \leqslant M(n)$.

From

$$ab = \frac{(a+b)^2 - (a-b)^2}{4}$$

we have $M(n) \leqslant 2S(n)$.

Then, we have

$$\tfrac{1}{2}M(n) \leqslant S(n) \leqslant M(n).$$

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 $S(n) \approx \frac{2}{3}M(n)$ 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

$$F_{2k} = F_k L_k \tag{5}$$

follows directly from (3) and (4).

From (4),

```
fib(n)
  f ← 1
  l ← 3
  for i = 2 to log2 n − 1
    f ← f * l
    l ← l * l − 2
  f ← f * l
  return f
```

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

$$\begin{aligned} L_{2k} &= \alpha^{2k} + \beta^{2k} \\ &= (\alpha^k + \beta^k)^2 - 2(\alpha\beta)^k \\ &= L_k^2 - 2 \cdot (-1)^k. \end{aligned} \tag{6}$$

We can compute F_{2^i} [2] by

$$F_{2^i} = \prod_{j=0}^{i-1} L_{2^j}. \tag{7}$$

The conventional product of Lucas numbers algorithm to compute F_n $(n = 2^i,\ i \geqslant 2)$ 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:

$$\begin{aligned} T(n) = &\sum_{i=2}^{\log_2 n - 1} \left(M(\gamma \cdot 2^{i-1}) + S(\gamma \cdot 2^{i-1}) \right) \\ &+ M(\gamma \cdot n/2) \leqslant \tfrac{4}{3} M(\gamma n). \end{aligned} \tag{8}$$

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

$$F_{n+m} = \tfrac{1}{2}(F_n L_m + F_m L_n), \tag{9}$$

$$L_{n+m} = \tfrac{1}{2}(L_n L_m + 5F_n F_m) \tag{10}$$

follow from (3) and (4).

Then, the identities

$$F_{k+1} = \tfrac{1}{2}(F_k + L_k), \tag{11}$$

```
fib(n)
  f ← 1
  l ← 3
  for i = 2 to log2 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.

$$\begin{aligned} L_{k+1} &= \tfrac{1}{2}(5F_k + L_k) \\ &= F_{k+1} + 2F_k \end{aligned} \tag{12}$$

follow from (9) and (10).

From (3) and (4),

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

Then, the identity

$$\begin{aligned} F_{2k} &= F_k L_k \\ &= \frac{(F_k + L_k)^2 - (F_k^2 + L_k^2)}{2} \\ &= \frac{(F_k + L_k)^2 - (F_k^2 + 5F_k^2 + 4\cdot(-1)^k)}{2} \\ &= 2F_{k+1}^2 - 3F_k^2 - 2\cdot(-1)^k \end{aligned} \tag{14}$$

follows from (5), (11) and (13).

Furthermore, the identity

$$L_{2k} = 5F_k^2 + 2\cdot(-1)^k \tag{15}$$

follows from (6) 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 $(n = 2^i,\ i \geqslant 2)$ 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.

Table 1
Comparison between operation counts in each iteration of the product of Lucas number algorithms

	Conventional	Presented
Multiplication	1	0
Square	1	2

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

$$\begin{aligned} T(n) &= \sum_{i=2}^{\log_2 n - 1} 2S(\gamma\cdot 2^{i-1}) + M(\gamma\cdot n/2) \\ &\leqslant \tfrac{7}{6}M(\gamma n). \end{aligned} \tag{16}$$

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 $k \geqslant 0$. 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:

$$\begin{aligned} F_{2k+1} &= \tfrac{1}{2}(F_{2k} + L_{2k}) \\ &= \tfrac{1}{2}\left(F_k L_k + L_k^2 - 2\cdot(-1)^k\right) \\ &= F_{k+1}L_k - (-1)^k. \end{aligned} \tag{17}$$

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^(⌊log2 n⌋−1)
    for i = 1 to ⌊log2 n⌋ − 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

[1] J.W. Cooley, J.W. Tukey, An algorithm for the machine calculation of complex Fourier series, Math. Comput. 19 (1965) 297–301.

[2] P. Cull, J.L. Holloway, Computing Fibonacci numbers quickly, Inform. Process. Lett. 32 (1989) 143–149.

[3] M.C. Er, Computing sums of order-k Fibonacci numbers in log time, Inform. Process. Lett. 17 (1983) 1–5.

[4] M.C. Er, A fast algorithm for computing order-k Fibonacci numbers, Comput. J. 26 (1983) 224–227.

[5] D. Gries, G. Levin, Computing Fibonacci numbers (and similarly defined functions) in log time, Inform. Process. Lett. 11 (1980) 68–69.

[6] D.E. Knuth, The Art of Computer Programming, Vol. 2: Seminumerical Algorithms, Addison-Wesley, Reading, MA, 1997.

[7] A.J. Martin, M. Rem, A presentation of the Fibonacci algorithm, Inform. Process. Lett. 19 (1984) 67–68.

[8] A. Pettorossi, Derivation of an $O(k^2 \log n)$ algorithm for computing order-k Fibonacci numbers from the $O(k^3 \log n)$ matrix multiplication method, Inform. Process. Lett. 11 (1980) 172–179.

[9] M. Protasi, M. Talamo, On the number of arithmetical operations for finding Fibonacci numbers, Theoret. Comput. Sci. 64 (1989) 119–124.

[10] A. Schönhage, V. Strassen, Schnelle Multiplikation grosser Zahlen, Computing (Arch. Elektron. Rechnen) 7 (1971) 281–292.

[11] J. Shortt, An iterative program to calculate Fibonacci numbers in $O(\log n)$ arithmetic operations, Inform. Process. Lett. 7 (1978) 299–303.

[12] F.J. Urbanek, An $O(\log n)$ algorithm for computing the nth element of the solution of a difference equation, Inform. Process. Lett. 11 (1980) 66–67.