

Middle and Ripple, fast simple $O(\lg n)$ algorithms for Lucas Numbers

L. F. Johnson

University of Waterloo

Abstract

A fast and simple $O(\log n)$ iteration algorithm for individual Lucas numbers is given. This is faster than using Fibonacci based methods because of the structure of Lucas numbers. Using a $\sqrt{5}$ conversion factor gives a faster Fibonacci algorithm because the speed up proposed in [5] also directly applies.

A fast simple recursive algorithm for individual Lucas numbers is given that is $O(\log n)$.

Keywords: Lucas numbers, algorithms, iteration, recursion, Fibonacci, recurrence relations, software engineering

1. Introduction

When initial conditions for the Fibonacci recurrence, $F_n = F_{n-1} + F_{n-2}$, are changed to 1,3 the result is the Lucas numbers (1891), $L_n = L_{n-1} + L_{n-2}$, with $L_1 = 1$, $L_2 = 3$. It is convenient to define $L_0 = 2$, then $L_2 = L_1 + L_0$.

There are a number of iterative solutions for this recursive definition, using addition, that are $O(n)$ [6], the lower bound for enumerating the sequence; however, individual numbers can be found in sublinear time. DeMoivre published a closed formula in 1730 that requires n multiplications [1]. F_n was first shown to be found in $O(\log n)$ time in 1978 [3] followed by improved algorithms [4,5]. None of these dealt directly with Lucas numbers perhaps because $L_n = F_{n-1} + F_{n+1}$, requiring two calls.

For simplicity, we need not discuss the size of L_n , which grows as $O(F_n)$, because this is the same for all algorithms (when the bit model executions

Email address: lfjsde@gmail.com (L. F. Johnson)

costs are excluded [4,5]). Execution costs are based on the representation operations, such as number of additions and multiplications.

The basic idea for obtaining an $O(\log n)$ algorithm is the use of doubling formulas. Simple doubling only gives values of n that are powers of 2. Some manipulation is required to solve for all n . Using the well known identities [1], $F_{n+m} = F_m * F_{n+1} + F_{m-1} * F_n$ and $F_{n+1} * F_{n-1} = F_n^2 + (-1)^n$, algorithms can be developed for F_n . A clever approach is to combine the related Fibonacci and Lucas sequences as for example, $F_{2n} = F_n * L_n$ and $L_{2n} = L_n^2 - 2$, where $2n$ is even [3,4,5]. A single formula can only be used to find L_n when n is a power of 2 and some manipulation is required to obtain a general solution for all n .

Noting that the general doubling formula for Lucas is actually $L_{2k} = L_k^2 - (-1)^k * 2$ and substituting $k+1$ for k then gives $L_{2k+2} = L_{k+1}^2 + (-1)^k * 2$. This single fundamental equation form is then the basis for a compact solution compared to [5].

2. Iteration

To calculate L_n for any n requires three adjacent terms in the sequence. Using a method based on a recursive based calculation the two terms to double are selected by markOdd() in the following algorithm.

2.1. Middle

For simplicity, the sequence variables are renamed as follows: $LL = L_{2k}$, $LM = L_{2k+1}$, $LH = L_{2k+2}$ The following formulas are used to double until L_n is found.

$$\begin{aligned} LL &= LL * LL - p * 2 \\ LH &= LM * LM + p * 2 \\ LM &= LH - LL \end{aligned}$$

Depending on the step, two different updates are required to prepare for the next step these are selected using an odd/even tag. This can be seen in the algorithm.

Two square multiplications, two sign calculations, and three additions in the iterative loop make this very fast compared to [5]. No pre conditions nor post conditions are used as was the case in [5]. All general methods need some kind of odd(i) selected calculation, when n is not a power of 2, and

this has been reduced here to the insertion of a shift forward of two sequence terms and one assignment.

The algorithm can be expressed as a recursion in a direct way, which is done in the next section. Using the relation, $F_n = \lceil (L_n/\sqrt{5} - 0.5) \rceil$, our algorithm also gives a very fast and simple calculation of a Fibonacci number.

```

Algorithm : Middle(n) given n > 1, return LL = L_n
N ← ⌊lg n
array markOdd[N] ← 0
i ← n; j ← N
while(j > 0){if(odd(i)) markOdd(j) = 1; i = i/2; j = j - 1}
j ← 1
while(j ≤ N - 1)
    p ← 1
    if(markOdd(j)){LL ← LM; LM ← LH; p ← -1 }
    LL ← LL * LL - p * 2
    LH ← LM * LM + p * 2
    LM ← LH - LL
    j ← j + 1;
endwhile
if(markOdd(j)) LL ← LM
return LL

```

3. Recursion

The recursion $L_{2n} = L_n^2 - 2$ when n is even can be used to find L_n in $O(\log n)$ time when n is a power of 2. This can be executed recursively by using $L_k = L_{k/2}^2 - 2$. The general form of this doubling formula for Lucas is $L_{2k} = L_k^2 - (-1)^k * 2$ [1] and substituting $k+1$ for k then gives $L_{2k+2} = L_{k+1}^2 + (-1)^k * 2$. The result on two adjacent position terms is two even position semi-adjacent terms. To obtain L_n when n is odd, calculate $L_{n+1} - L_{n-1}$. Thus to calculate L_n for any n requires three adjacent terms in the sequence.

3.1. Ripple

For simplicity, the sequence variables are renamed as follows: $LL = L_k, LH = L_{k+2}$. The following formulas are used to double until L_n is found.

$$\begin{aligned} LL &= LL * LL - p * 2 \\ LH &= LM * LM + p * 2 \end{aligned}$$

Depending on the odd and even values of n and $n/2$, different updates are required. The correct equation to use is selected by the variable p , as can be seen in the algorithm.

```

Algorithm : Ripple(n) given n > 1,
return Ln
if(n = 2) return 3
if(n = 3) return 4
if(n = 4) return 7
if (even(n/2)) then p ← 1 else p ← -1
LL ← Ripple(⌊n/2⌋)
if (even(n)) then return LL * LL - p * 2
else LH ← Ripple(⌈n/2⌉)
return LH * LH - LL * LL + p * 4

```

If a compiler is able to remember multiple identical calls, the algorithm can be simplified to remove LH and HL.

```

if (even(n)) then return L(n/2)2 - p * 2
else return L(⌈n/2⌉)2 - L(⌊n/2⌋)2 + p * 4

```

Each call uses a selection of p , one or two square multiplications, and one signed addition. Making this very fast compared to [5], when the cost of recursion is ignored. No pre conditions nor post conditions are used as was the case in [5], except to define the recursion basis.

4. Conclusions

For linear Fibonacci algorithms, a change in initial conditions gives a Lucas algorithm. Any $O(\log n)$ Fibonacci algorithm can be used to find the Lucas number indirectly by the relation $L_n = F_{n-1} + F_{n+1}$. While this requires two external calls, these algorithms can usually be modified internally to produce L_n in a single call (which may only hide the two external calls). Conversely, Fibonacci numbers can be derived from Lucas numbers using a $\sqrt{5}$ division factor. By the nature of the simple foundation relation of Lucas squares, our algorithm is simpler and faster than other Fibonacci algorithms.

Since our algorithm only uses two square multiplication per iteration, the speed argument for squares in [5], claimed to be the fastest for very large n , also applies to our method. This algorithm is much simpler and faster by a constant factor than that in [5] (which did not internally implement the fast FFT-based Schoenhage-Strassen process for fast multiplication). Thus, the two algorithms can be directly compared.

In Ripple the the average number of multiplications is less than 2. The algorithm only uses squares, and the argument in [5], claimed to be the fastest for very large n , also applies to this method.

While Lucas numbers can be derived from finding Fibonacci numbers, it is always more efficient to calculate the Lucas numbers directly.

References

- [1] D.E. Knuth, The Art of Computer Programming: Fundamental Algorithms, Addison-Wesley, Reading, MA, 1973.
- [2] T. Koshy, Fibonacci and Lucas Numbers with Applications, Wiley, New York, 2001.
- [3] J. Shortt, An iterative program to calculate Fibonacci numbers in $O(\log n)$ arithmetic operations, Inform. Process. Lett. 7 (1978) 299-303.
- [4] P. Cull, J.L. Holloway, Computing Fibonacci numbers quickly, Inform. Process. Lett. 32 (1989) 143-149.
- [5] D. Takakaski, A fast algorithm for computing large Fibonacci numbers, Inform. Process. Lett. 75 (2000) 243-246.
- [6] L. F. Johnson, Tumble, a fast simple iteration algorithm for Fibonacci, Inform. Process. Lett. 89 (2004) 187-189.
- [7] G. Rawlins, Compared to What?, Computer Science Press, NY, 1992.