# A Fast Algorithm for Computing Order-$K$ Fibonacci Numbers

**M. C. Er**

Department of Computing Science, The University of Wollongong, PO Box 1144, Wollongong, NSW 2500, Australia

**A fast algorithm for computing order-$k$ Fibonacci numbers in $O(k^2 \lg n/2k)$ units of time is presented. Furthermore the time complexity of the algorithm is $O((k-1)n)$ below threshold when $n$ is small and $k$ is large. Finally, the space complexity of this optimal algorithm is better than that of most other reported algorithms for doing the same task.**

## 1. INTRODUCTION

Gries and Levin,[1] Pettorossi,[2] Urbanek[3] and Wilson and Shortt[4] repeatedly formulate almost identical algorithms for computing order-$k$ Fibonacci numbers in $O(\lg n)$ units of time, using matrix or non-matrix methods. Wilson and Shortt's algorithm is the only one among the many mentioned above using a non-matrix method which necessitates the establishment of complex identity equations. This algorithm has a time complexity of $O(k^2 \lg n)$, using $O(k^2)$ multiplicative and $O(k^3)$ additive operations, respectively, per loop. Pettorossi's algorithm has a time complexity of $O(1.5k^3 \lg n)$ using $O(1.5k^3)$ multiplicative and $O(1.5k^3)$ additive operations, respectively, per loop because the second matrix multiplication has a probability of 0.5 of being activated. Urbanek's algorithm has a time complexity of $O(k^3 \lg n)$, with multiplicative and additive operations at $O(k^3 + 0.5k^2)$ per loop because the second matrix multiplication involves a $1 \times k$ matrix and a $k \times k$ matrix, which has a probability of 0.5 of being carried out. Gries and Levin give no detailed coding but their arguments together with Refs 5 and 6 for exponentiation algorithms establish that the time complexity of their algorithm is $O(1.5k^2 \lg n)$. It uses $O(1.5k^2)$ multiplicative and $O(3k^2)$ additive operations per loop, again because of the 0.5 probability of executing the second matrix multiplication. The connection between Wilson and Shortt's formulation and the matrix approach is discussed in Ref. 2.

The non-matrix formulation, and hence the resulting algorithm, are unduly complex, leaving very much to be desired. In contrast, the matrix approach offers a conceptually simple algorithm, but it suffers from the penalty of a second matrix multiplication. Moreover, when $n \to 0$ and $k \to \infty$, the matrix approach turns out to be even slower than the conventional linear approach. We show that all such deficiencies can be repaired, and present an improved algorithm for computing order-$k$ Fibonacci numbers, which is faster than any known algorithms for doing the same task.

## 2. THEORY

Traditionally the order-$k$ Fibonacci series is defined by[1-4]

$$\left.\begin{aligned} &u_{n,k} = \sum_{i=1}^{k} u_{n-i,k}, \quad \text{for } n > k > 1 \\ &u_{k-j,k} = 0, \quad \text{for } j = 1, 2, \ldots, k-1 \\ &\text{and} \\ &u_{k,k} = 1 \end{aligned}\right\} \quad (0)$$

where $u_{n,k}$ is the $n$th term of the order-$k$ Fibonacci sequence. We deviate from the tradition by defining $k$ sequences of the order-$k$ Fibonacci series as follows:

$$\left.\begin{aligned} &f_{n,j} = \sum_{1=1}^{k} f_{n-i,j}, \quad \text{for } j = 1, 2, \ldots, k, \text{ and } n > 0 \\ &f_{n,j} = \begin{cases} 1, j = 1 - \text{n} \\ 0, \text{otherwise} \end{cases} \\ &\text{for } n = 1-k, 2-k, \ldots, 0, \text{ and} \\ &f_{1,j} = 1, \quad \text{for } j = 1, 2, \ldots, k \end{aligned}\right\} \quad (1)$$

where $f_{n,j}$ is the $n$th term of the $j$th sequence of the order-$k$ Fibonacci series. Here, it is understood that $k > 1$; when $k = 2$, the order-$k$ Fibonacci series is the standard Fibonacci series.

This definition has distinct advantages over the traditional definition, and will be used throughout this paper.

Let $M$ be a $k \times k$ matrix, such that

$$\left.\begin{aligned} &M^n = (a_{ij}) \\ &\text{where} \\ &a_{ij} = f_{n+1-i,j} \end{aligned}\right\} \quad (2)$$

By induction, it can be shown that

$$M^n = M^{n-1} M^1 = M^1 M^{n-1} \quad (3)$$

So, in general, we have

$$M^{r+c} = M^r M^c \quad (4)$$

where $r$ and $c$ are positive integers. Let $R^n$ be the first row of $M^n$. Thus, from Eqn (3), we have

$$R^{r+1} = R^r M^1 \quad (5)$$

or in general

$$R^{r+c} = R^r M^c \quad (6)$$

From Eqn (5), we establish

$$\left.\begin{aligned} &f_{n,j} = f_{n-1,j+1} + f_{n-1,1}, \quad \text{for } j = 1, 2, \ldots, k-1 \\ &\text{and} \\ &f_{n,k} = f_{n-1,1} \\ &\text{both for } n \geq 0. \end{aligned}\right\} \quad (7)$$

Equations (6) and (7) form the theoretical basis for optimizing an algorithm for computing order-$k$ Fibonacci numbers using the matrix method.

## 3. ALGORITHM

The basic idea in computing an order-$k$ Fibonacci number using the matrix method is to first of all compute

CCC-0010-4620/83/0026-0224$02.00

$M^n$ and then extract $f_{n,1}$. Therefore, starting from $M^1$, an efficient algorithm should compute $M^n$ in the shortest time. This is equivalent to saying that the exponent of $M$ should be increased to $n$ as quickly as possible. One well-known technique is to double the exponent each time by means of squaring the matrix. However, as Eqn (6) implies, $R^{2i}$ can be computed when $M^i$ is known:

$$R^{2i} = R^i M^i \tag{8}$$

If $R^{2i}$ is assigned to the bottom row of the new matrix, we can compute $M^{2i+k-1}$ by using Eqn (7) to generate its rows from the $(k-1)$th to the first. Thus the exponent $n$ can be reached in a much shorter time without paying any penalty at all. In general, any positive integer $n$ can be rewritten using the following formula which corresponds to the increments of the exponents:

$$n = (\ldots(((y)*2 + k - 1 + x)*2 + k - 1 + x)*2 \ldots)*2 + k - 1 + x \tag{9}$$

where $x$ is either 0 or 1, and $y > 1$. When $x$ is 1, Eqn (7) is applied while assigning $R^{2i}$ to the last row of the new matrix; otherwise, $R^{2i}$ is copied straightaway to the last row.

Assuming that $k$ is a constant and the following type definitions,

```
size   = 1 . . . k;
vector = array [size] of integer;
matrix = array [size] of vector;
```

and the global array

```
M: matrix;
```

the detailed algorithm may be described as follows.

```
function Fibonacci (n: integer): integer;
  { This function computes order-k Fibonacci numbers
    by calculating M^n and then extracting f_{n,1}, where
    n > 0. }
var i, j: size;
  x: integer;
  T: vector;
begin
  if n < 3*k then
    begin
    InitM;
    if n - k + 1 ≤ 0 then i := k + 1 - n else i := 1;
    for j := 1 to k do M[k][j] := M[i][j];
    for x := 2 to n - k + 1 do IncRow (M[k], M[k])
    end
  else begin
    Fibonacci := Fibonacci ((n - k + 1) div 2);
    for i := 1 to k do
      begin
      T[i] := 0;
      for j := 1 to k do
        T[i] := T[i] + M[1][j]*M[j][i]
      end;
    if ((n - k + 1) mod 2) = 1 then IncRow (T, M[k])
    else for j := 1 to k do M[k][j] := T[j]
    end;
  FillMatrix;
  Fibonacci := M[1][1]
end { Fibonacci };
```

The procedure *InitM* initializes the global array $M$ to $M^1$ when activated, and may be described as follows.

```
procedure InitM;
  { This procedure assigns M^1 to the global array M. }
var j: size;
begin
  for j := 1 to k do M[k][j] := 0;
  M[k][k - 1] := 1;
  FillMatrix
end { InitM };
```

Note that the procedure *FillMatrix* is called in turn to fill in rows $k-1$ to 1 of the array $M$, and the details are described in the following algorithm.

```
procedure FillMatrix,;
var i: size
begin for i := k - 1 downto 1 do IncRow (M[i + 1], M[i])
end { FillMatrix };
```

The procedure *IncRow* is simply an application of Eqn (7) and may be stated as follows.

```
procedure IncRow (var A, B: vector);
  { This procedure computes the next level of row entries
    and assigns to the array B from a given row storing
    in the array A using Eqn (7). The arrays A and B need
    not be distinct. }
var A1: integer;
  j: size;
begin
  A1 := A[1];
  for j := 1 to k - 1 do B[j] := A[j + 1] + A1;
  B[k] := A1
end { IncRow };
```

All of the cited algorithms for computing order-$k$ Fibonacci numbers using the matrix method suggest that $M^n$ be computed starting from $M^1$. This is undesirable because these algorithms spend unnecessary time doing matrix multiplications for gaining initially limited increments of exponents. Our solution uses Eqn (7) to compute $M^i$, $1 \le i < 3k$ initially, which is much faster than the approach involving matrix multiplications as it involves no multiplication at all. Since the recursion of *Fibonacci* is discontinued when $n$ is reduced to less than $3k$, the smallest starting value of the exponent of $M$ is $k$. Since each starting value between $k$ and $(3k-1)$ is equally likely, the average starting value is $(4k-1)/2$. That is, $y = (4k-1)/2$.

## 4. ANALYSIS

From Eqn (9), we have

$$n = y*2^m + z \tag{10}$$

where

$$z = (k - 1 + x)\sum_{i=0}^{m-1} 2^i$$

Rearranging it, we derive

$$m = \lg\left(\frac{n-z}{y}\right) \tag{11}$$

Since $y$ and $z$ are positive integers greater than 1, $m < \lg n$. As $m$ indicates the number of looping operations

(or the number of recursive calls), the algorithm thus results in fewer cycles than those best algorithms claimed by others in computing order-$k$ Fibonacci numbers.

Furthermore, we observe that, during each cycle, the algorithm performs $k^2$ multiplicative operations involving a $1 \times k$ matrix and a $k \times k$ matrix, and $((k-1)^2 + k^2)$ additive operations for filling $(k-1)$ rows of a matrix and for doing partial sums during matrix multiplications.

Putting them together, the time complexity of the algorithm is $O(k^2 \lg ((n-z)/y))$, or simplifying it, bounded by $O(k^2 \lg n/2k)$. In summary, the algorithm is faster than any reported algorithms for computing order-$k$ Fibonacci numbers.

## 5. REFINEMENT

As remarked in Section 1, when $n$ is small and $k$ is big, most of the cited algorithms are slower than the naïve approach using Eqn (0), which requires only $O(kn)$ additive operations. Clearly there must be a threshold beyond which the matrix approach will run faster than the naïve approach. Let $t$ be such a threshold. Further assume $r$ to be the ratio of the cost of multiplication to the cost of addition. Equating the running times of algorithms using matrix and naïve approaches at threshold, we have,

$$(rk^2 + (k-1)^2 + k^2) \lg \frac{t}{2k} = kt$$

After differentiation and simplification, we derive

$$t = 1.44\,(r+2)k \qquad (12)$$

Notice that the algorithm uses Eqn (7) to compute $M^n$ when $n$ is small. An application of Eqn (7) requires only $(k-1)$ additive operations. Therefore, the computation of the $n$th order-$k$ Fibonacci number takes $O((k-1)n)$ units of time, which is marginally faster than the naïve approach. If the threshold guard of *Fibonacci* is replaced by $n < 1.44\,(r+2)k$, we achieve an optimal algorithm for computing order-$k$ Fibonacci numbers. On the one hand, it is marginally faster than the naïve approach and significantly faster than other matrix approaches, when $n$ is small. On the other hand, it is faster than all other reported algorithms for doing the same task, when $n$ is large.

## 6. FURTHER IMPROVEMENT

It may be suggested that the algorithm discussed above is not flexible enough. For instance, having computed the 520th order-$k$ Fibonacci number, if the next required Fibonacci number is 521st, the algorithm knows no better than to start again right from the beginning. Now we show that the algorithm can be modified fairly easily to cater for such an application.

To remember the old exponent of $M$, a global variable of type integer, *oldn*, is introduced. The modification is to make the algorithm starting from *oldn* instead of from 1 when appropriate, and the modified fragment may be described as follows.

```
function Fibonacci2 (n: integer): integer;
  { This function computes order-k Fibonacci numbers
    starting from M^oldn if n ≥ oldn or from M^1 otherwise. }
var i, j: size;
  x: integer;
  T: vector;
begin
  if n − oldn < trunc (1.44*(r + 2)*k) then
    begin
    if (n < oldn) or (oldn = 0) then
      begin
      InitM;
      oldn := 1
      end;
    if n − k + 1 < oldn then i := k − (n − oldn) else i := 1;
    for j := 1 to k do M[k][j] := M[i][j];
    for x := oldn + 1 to n − k + 1 do IncRow (M[k],
                                              M[k])
    end
  else { as per the function Fibonacci }
  FillMatrix;
  Fibonacci2 := M[1][1]
end { Fibonacci2 };
```

To activate the procedure *Fibonacci2*, a proper context must be set up; and a possible sequence of instructions is suggested in the following procedure.

```
function Fib (i: integer; var firsttime: boolean): integer;
begin
  if firsttime then oldn := 0;
  Fib := Fibonacci2 (i);
  firsttime := false;
  oldn := i
end { Fib };
```

The function *Fib* should be called with *firsttime* set to **true** initially, and it need not be reset in the subsequent calls.

## 7. REMARKS

The various analyses discussed above show that an optimal algorithm for computing order-$k$ Fibonacci numbers is indeed possible. The algorithm uses only a $k \times k$ matrix and a $1 \times k$ matrix. The space complexity is better than the algorithms reported by Gries and Levin[1] and Pettorossi,[2] and equal to that reported by Urbanek.[3]

Furthermore, the recursion used in the algorithm can be trivially converted to iteration without introducing stack. For clarity's sake, this was not carried out.

The order-$k$ Fibonacci numbers arise in many interesting computer applications. For instance, they are used in polyphase sort for achieving optimal merging involving several magnetic tapes.[7] They are also used in solving linear homogeneous difference equations with constant coefficients.[1,3]

It may be suggested that the recurrence relation Eqn (1) be solved to yield a general closed-form equation which is then used for computing the order-$k$ Fibonacci numbers. An experiment was carried out but it did not produce satisfactory results. The main obstacle was the rounding error involving floating point arithmetic. For instance, the closed-form equation of order-2 Fibonacci number is

$$f_{n,2} = \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right)$$

When $n$ is sufficiently large ($\geq 26$), the cumulative

rounding errors start to affect the accuracy of Fibonacci numbers with a 32-bit machine. It seems that some numerical analysis techniques are needed to overcome this problem, which are beyond the scope of this paper.

**Acknowledgements**

The author would like to thank the referee for his constructive comments. This research was supported by RGC under grant 05–143–105.

## REFERENCES

1. D. Gries and G. Levin, Computing Fibonacci numbers (and similarly defined functions) in log time. *Information Processing Letters* **11**, 68–69 (1980).
2. 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. *Information Processing Letters* **11**, 172–179 (1980).
3. 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).
4. T. C. Wilson and J. Shortt, An $O(\log n)$ algorithm for computing general order-$k$ Fibonacci numbers. *Information Processing Letters* **10**, 68–75 (1980).
5. R. Conway and D. Gries, *An Introduction to Programming*, Winthrop, Cambridge, 3rd edn (1979).
6. E. W. Dijkstra, *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs, NJ (1976).
7. M. C. Er and B. G. T. Lowden, The theory and practice of constructing an optimal polyphase sort. *The Computer Journal* **25**, 93–101 (1982).

Received September 1982