THE MATRICES OF FIBONACCI NUMBERS

M. C. Er
University of Wollongong, N.S.W. 2500, Australia
(Submitted June 1982)

1. INTRODUCTION

In a recent paper, Kalman [3] derives many interesting properties of generalized Fibonacci numbers. In this paper, we take a different approach and derive some other interesting properties of matrices of generalized Fibonacci numbers. As an application of such properties, we construct an efficient algorithm for computing matrices of generalized Fibonacci numbers.

The topic of generalized Fibonacci sequences discussed here is related to the theory of polyphase sorting in an interesting way; in fact, it is used in optimizing the polyphase sort (see [1] and [7]). The theory of polyphase sorting, in return, helps shape the construction of a fast algorithm for computing the order-k Fibonacci numbers in $O(k^2 \log n)$ steps (see [2] and [5]).

2. DEFINITIONS

Define k sequences of generalized order-k Fibonacci numbers, for some $k \geqslant 2$, as follows:

$$F_t^n = \sum_{i=1}^{k} F_t^{n-i}, \text{ for } 1 \leqslant t \leqslant k, \tag{1}$$

where F_t^n is the n^{th} Fibonacci number of the t^{th} sequence. We may arrange these k sequences in k columns extending to infinity in both directions. Define the *window* at level n, W_n, to be the $k \times k$ matrix of generalized Fibonacci numbers such that

$$W_n = (a_{ij}^n), \text{ for } 1 \leqslant i, j \leqslant k, \tag{2}$$

where $a_{ij}^n = F_j^{n-k+i}$.

A set of initial values of these k sequences, defined by (1), may be given by

$$F_t^n = \begin{cases} 1, & t = n + k \\ 0, & \text{otherwise} \end{cases}, \text{ for } 1 - k \leqslant n \leqslant 0. \tag{3}$$

In other words, W_0 is the $k \times k$ identity matrix.

By an application of (1)-(3), we deduce that

$$F_t^1 = 1, \text{ for } 1 \leqslant t \leqslant k. \tag{4}$$

In consequence, we have

$$W_1 = \begin{bmatrix} 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ 0 & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & & \vdots \\ 0 & 0 & 0 & \cdots & 1 \\ 1 & 1 & 1 & \cdots & 1 \end{bmatrix} \tag{5}$$

To derive the n^{th} Fibonacci number such that $n \leq -k$, we simply invert (1):

$$F_t^n = F_t^{n+k} - \sum_{i=1}^{k-1} F_t^{n+i}, \text{ for } 1 \leq t \leq k. \tag{6}$$

In this way, the k columns can be extended to infinity in both directions, starting from the identity matrix, W_0.

3. SOME PROPERTIES

By the definition of generalized order-k Fibonacci numbers, we have

$$W_n = W_1 W_{n-1}. \tag{7}$$

In other words, W_1 may be viewed as a row operator as it shifts a window vertically by one level. From (7), we derive

$$W_n = W_1^n W_0. \tag{8}$$

Since $W_0 = I$, we have just derived

$$W_n = W_1^n. \tag{9}$$

Abbreviating W_1 as W, we may write W_n for W^n.

As a consequence of (9), we have

$$W_n = W_1 W_{n-1} = W_{n-1} W_1. \tag{10}$$

The above equation shows that matrix multiplication of windows is commutative. Indeed, $\{W_n \mid n \in Z\}$ is an infinite cyclic group and satisfies the usual laws of exponents.

From $W_n = W_{n-1} W_1$, we obtain the following two equations relating elements of two adjacent rows;

$$F_t^n = F_k^{n-1} + F_{t-1}^{n-1}, \text{ for } 2 \leq t \leq k, \tag{11}$$

and

$$F_1^n = F_k^{n-1}. \tag{12}$$

Interestingly, these two equations are foundational to the basic theory of polyphase sorting [1]. The n^{th} row of Fibonacci numbers is precisely the so-called *ideal distribution* in the sorting context.

More interestingly, the column and row recursions of windows can be interpreted as follows. Multiplying by W on the left of any window has the effect of rolling the window down, exposing a window at the next level. More generally, multiplying two windows at levels r and c, respectively, may be viewed as rolling the window at level c down by r levels, with the resulting window placed at level $(r + c)$, i.e.,

$$W^{r+c} = W^r W^c, \tag{13}$$

where r and c are any integers. In contrast, multiplying any window by W on the right has the effect of bringing the row recursion into play. If

$$R_n = [F_1^n F_2^n \ldots F_k^n]$$

is the n^{th} row of Fibonacci numbers (therefore, the last row of W_n), then the $(n+1)^{st}$ row,

$$R_{n+1} = R_n W, \tag{14}$$

may be obtained by:

a) shifting each element of R_n one position to the right (filling the vacant position with a zero and truncating the element, F_k^n, moved out of place); and

b) adding the truncated element, F_k^n, to each entry.

These two steps may be illustrated as follows:

$$[F_1^n\ F_2^n \ldots F_k^n] \rightarrow [0\ F_1^n \ldots F_{k-1}^n] \quad (F_k^n \text{ drops out})$$

$$\rightarrow [F_k^n\ F_k^n + F_1^n \ldots F_k^n + F_{k-1}^n].$$

We see, from the above discussions, that matrix W contains the mechanisms for computing (1), (11), and (12). Surprisingly, to compute generalized Fibonacci numbers, (1) need not be used directly; instead, (11), (12), and (13) are used.

4. APPLICATIONS

As an application of the interesting properties of windows, discussed previously, we describe the construction of a fast algorithm for computing generalized Fibonacci numbers. Paradoxically, when n is large, it is faster to compute the n^{th} Fibonacci number by using the matrix method discussed in the previous two sections than by using (1) alone (see [2] and [5]). As shown in equation (13), it is possible to increase the exponent of W through matrix multiplication, by treating each window as a single entity. In hand calculation or in computer implementation, it is desired to keep $r = c$ so that (1) only one $k \times k$ matrix needs to be maintained during the computation and (2) the destination level can be reached in the shortest time.

Note that any positive integer n can be expressed in terms of the binary representation:

$$n = \sum x_i 2^i, \tag{15}$$

where $x_i = 0$ or 1. Therefore, we may write

$$W^n = \prod_{x_i = 1} W^{2^i}. \tag{16}$$

If an algorithm starts with the window at level 1 and doubles the window level each time, then W^n can be reached in $O(\log n)$ steps. However, this approach requires two matrix multiplications: one for matrix squaring, another for accumulating the result by applying (16) (see Urbanek's implementation [5]). We now give an algorithm for computing the generalized order-k Fibonacci numbers, which is better than the algorithm given by Urbanek because it requires only one matrix multiplication per cycle.

Note that (15) can be rewritten as follows:

$$n = (\ldots((1 * 2 + x_{j-1}) * 2 + x_{j-2}) * 2 \ldots) * 2 + x_0, \tag{17}$$

where j is the smallest positive integer such that $n < 2^{j+1}$, and $x_i = 0$ or 1.

Consequently, we have

$$W^n = W^{(\ldots((1*2+x_{j-1})*2+x_{j-2})*2\ldots)*2+x_0}$$
$$= (\ldots((W^2 * W^{x_{j-1}})^2 * W^{x_{j-2}})^2 * W^{x_0}. \tag{18}$$

For instance,

$$W^{25} = W^{(((1*2+1)*2+0)*2+0)*2+1}$$
$$= (((W^2 * W)^2)^2)^2 W.$$

Equation (18) shows that, by working from the central parenthetical quantity outwards, W^n can be computed through successive steps requiring either matrix squaring or matrix squaring followed by multiplying by W. Fortunately, multiplying by W can be accomplished by applying (11) and (12) without the need of matrix multiplication.

An efficient algorithm for implementing the ideas described above is best based on the following recursive expressions:

$$W^n = \begin{cases} (W^{n/2})^2, & n \text{ is even} \\ (W^{\lfloor n/2 \rfloor})^2 W, & n \text{ is odd,} \end{cases}$$

and

$$W^1 = W.$$

The details of the algorithm are presented below using a programming notation commonly used in Computer Science (see [6]). Note that (n div 2) = $\lfloor n/2 \rfloor$, and that A[1,] is row 1 of A.

```
function Fibonacci (n, k : integer) : integer;
    {Given n ε Z and k ≥ 2, this function returns F_k^n as a result.}
var  A : k x k matrix;
procedure  Window (n : integer);
    {Compute W^n.}
    var  R : 1 x k matrix;
    begin
        if n = 1 then A := W^1
        else begin
            Window (n div 2);
            R := A[1,] * A; {R = W^m[1,] * W^m, m = n div 2}
            if odd(n) then
                A[1,] := R * W^1  {A[1,] = W^2m[1,] * W}
            else A[1,] := R;      {A[1,] = W^2m[1,]}
            Compute rows 2 to k of A from previous rows
            end
    end {Window};
```

```
begin
    if n = 0 then return (1);
    if n < 0 then
        begin
        Window(-n);
        Inverse {Inverse matrix A}
        end
    else Window (n);
    return (A[k,k])
end  {Fibonacci};
```

The procedure "Window" is called recursively for achieving the effect of starting the computation from the innermost pair of brackets of (18). It halves the value of n per recursion, truncating the remainder for odd n, and terminates the recursion when n is reduced to 1. In the last recursive activation, matrix A is initialized to W^1. Thus, the number of activations of Window is $O(\log n)$. In contrast, a direct application of (1) takes $O(n)$ steps.

Note that every row of a window satisfies (14). Therefore, in squaring a window, it is unnecessary to compute the value of every element of the resulting window by matrix multiplication (where a total of k^3 multiplicative operations would be required). Instead, we compute the first row of the resulting window as $A[1,] * A$ (see the procedure Window), and then compute the remaining rows by using (11) and (12). In this case, k^2 multiplicative operations are needed for squaring a window. Note further that, if the level of a window is odd, a fine adjustment of the window by multiplying it by W is required. Again this operation can be carried out economically by using (11) and (12). If such an adjustment is required, it is more economical to carry it out immediately after $A[1,] * A$ is computed than otherwise; hence, the test for odd(n), and $R * W^1$ in the procedure Window. Thus, the total number of multiplicative operations per procedure call of Window is k^2.

Since the cost of computation of additions is negligible in comparing with that of multiplications, it is ignored in the calculation of cost. Thus, the overall running time of this algorithm is $O(k^2 \log n)$. In contrast, Urbanek's implementation [5] requires two matrix multiplications. Since the probability of carrying out the second matrix multiplication is 0.5, the overall running time for his algorithm is $O(1.5k^2 \log n)$, taking into account that matrix multiplications could be done in $O(k^2)$ steps. Our algorithm thus runs 33% faster than Urbanek's algorithm. Moreover, our algorithm supports the computation of $-n^{th}$ Fibonacci numbers, as seen in the procedure Fibonacci, which is not addressed in [2] and [5]. Alternatively, it is computationally faster by making procedure Window take the initial window as a second parameter. If $n < 0$, W^{-1} is passed as a second parameter to Window; whereas, if $n > 0$, W^1 is passed as a parameter.

For an interesting application of the generalized order-k Fibonacci numbers to the polyphase sorting, the reader is referred to [1].

5. REMARKS

The material presented here could easily be adapted to computing solutions of linear difference equations with constant coefficients [4]. This is left as an exercise for the interested reader.

ACKNOWLEDGMENTS

The author is indebted to the referee for his valuable comments, suggestions, additional references and, especially, for improving the presentation of this paper. This research was supported by the RGC under Grant 05-143-105.

REFERENCES

1. M. C. Er & B. G. T. Lowden. "The Theory and Practice of Constructing an Optimal Polyphase Sort." *Computer Journal* 25 (1982):93-101.
2. D. Gries & G. Levin. "Computing Fibonacci Numbers (and Similarly Defined Functions) in Log Time," *Information Processing Letters* 11 (1980):68-69.
3. D. Kalman. "Generalized Fibonacci Numbers by Matrix Methods." *The Fibonacci Quarterly* 20 (1982):73-76.
4. K. A. Miller. *Linear Difference Equations*. New York: Benjamin, 1968.
5. 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 (1980):66-67.
6. N. Wirth. *Algorithms + Data Structures = Programs*. Englewood Cliffs, N.J.: Prentice-Hall Inc., 1976.
7. D. A. Zave. "Optimal Polyphase Sorting." *SIAM J. Computing* 6 (1977):1-39.

◆◇◆◇◆