

# Lookup Tables, Recurrences and Complexity

Richard J. Fateman  
University of California, Berkeley

## Abstract

The use of lookup tables can reduce the complexity of calculation of functions defined typically by mathematical recurrence relations. Although this technique has been adopted by several algebraic manipulation systems, it has not been examined critically in the literature. While the use of tabulation or “memoization” seems to be a particularly simple and worthwhile technique in some areas, there are some negative consequences. Furthermore, the expansion of this technique to other areas (other than recurrences) has not been subjected to analysis.

This paper examines some of the assumptions. A more detailed technical report [9] is under preparation.

This paper appeared in ACM-SIGSAM 1989 proceedings.

## 1 Fibonacci Numbers: A Famous Example

Consider the Fibonacci numbers, defined by a two-term recurrence

$$f_n := f_{n-1} + f_{n-2}, \quad f_0 = 0, \quad f_1 = 1.$$

This is expressed in Macsyma [4] by

```
(f[n]:=f[n-1]+f[n-2], f[0]:0, f[1]:1).
```

If this recurrence were applied simply as stated, for each higher-order member of the sequence there would be substantial re-calculation of lower-order members. In fact, the cost would be exponential in  $n$ . By “remembering” the values of each  $f[n]$  when first computed, Macsyma makes the algorithm essentially linear in time. Other systems, most notably SMP [5], Maple [3], and Mathematica [10], make a point of emphasizing similar features. This technique is eloquently described

in an introductory Computer Science text by H. Abelson and G. Sussman (exercise 3.27, page 218 of [1]) Considered in some more detail, the algorithm is not truly linear. The cost of the “unit” addition grows substantially. For example,  $f[100000]$  has 20899 decimal digits. Nevertheless, we should perhaps be pleased that such a simple programming trick, one that takes no effort on the programmer’s part, reduces complexity by an exponential ratio.

Could you actually compute  $f[100000]$  this way? If you consider that  $f$  is monotonic, and  $f[50000]$  already has 10450 decimal digits, you’d need to be able to store about  $10^9$  digits. That’s a considerable amount of storage. You’d also have to support a rather large stack, since the usual implementation of recursion will require several words of storage per level of call. Wolfram ([10] page 203), quite reasonably, notes “You should usually define functions to remember values only if the total number of different values that will be produced is comparatively small.” In other words, if this is the right way to compute Fibonacci numbers, we’d better not need any large ones.

Here’s another program in Macsyma, slightly more complicated, with the same objective.

```
% f[n]:= if evenp(n) then f[n/2] * (f[n/2]+2*f[n/2-1])
%
else f[n-1] + f[(n-1)/2]^2 + f[(n-1)/2]^2 )$

%
(f[0]:0, f[1]:1,
  f[n]:= if evenp(n) then f[n/2] * (f[n/2]+2*f[n/2-1])
        else f[(n+1)/2]^2 + f[(n-1)/2]^2 )$
```

This program also computes the Fibonacci numbers. (See [6], ex. 26, p. 464, answer p. 637.) but it takes time and storage proportional to  $\log n$ . For  $f[100000]$ , it needs to store about 50 elements. Now we can compute much higher values in a modest amount of time and space.

But it turns out that with a bit more programming effort, we can do away with the need for logarithmic storage. We know that only 2 terms are needed if we compute the Fibonacci numbers iteratively; using this

logarithmic method, carefully programmed, only two terms are needed at any one time as well.<sup>1</sup>

*Conclusion:* There is hardly much worth praising in the earlier version of the Fibonacci program unless you are truly dealing with a small number of terms or are obsessed with slavishly following a brief and naive mathematical statement<sup>2</sup>. If we can't come up with a better reason to use remembering than the Fibonacci sequence, we are in trouble.

## 2 Chebyshev Polynomials: Another Example

A referee report for an early version of reference [9] says that

... "remembering" can theoretically reduce the running time from exponential to linear. Such examples include two-term recurrences, E.g. the Chebyshev polynomials defined by

$$T_n(x) = 2xT_{n-1}(x) - T_{n-2}(x), \quad T_0(x) = 1, \quad T_1(x) = x.$$

If coded directly from this recursive definition, computing  $T_n(x)$  would take time exponential in  $n$ . By remembering previous values, the running time is polynomial in  $n$ .

In Macsyma, one can directly implement this definition:

```
(T[n]:=2*x*T[n-1]-T[n-2], T[0]:1, T[1]:x)$
```

Certainly it is concise, but by a modest change we can make a major improvement. We reduce the work by requiring only the computation of  $\log n$  previous values in the sequence, rather than  $n$  previous values. Recall that

$$T_{2n}(x) = T_2(T_n(x)) \quad T_{2n+1} = 2T_{n+1}T_n - x.$$

This can be expressed in Macsyma as

```
(T[0]:1,T[1]:rat(x),
 T[n]:= if evenp(n) then 2*T[n/2]^2-1
        else 2*T[(n-1)/2]*T[(n+1)/2])$
```

<sup>1</sup>The built-in function `fib` in Macsyma stores only 2 non-trivial terms. The version of Fibonacci in Maple of October, 1984 uses the same recurrence.

<sup>2</sup>Lest one be misled into thinking there are no more ways of computing this function, here is another, shorter, scheme which computes floating point approximations to Fibonacci numbers:

$$F_n := ((1 + \sqrt{5})/2)^n / \sqrt{5}$$

This has to be done with sufficient floating-point precision so that when  $F_n$  is rounded to an integer, it is correct. Carrying 20,000 decimal places may be expensive, but might easily be less expensive than the integer computation. On the other hand, you may be willing to settle for an approximation.

For  $n = 64$  this provides a speed-up of over a factor of 8 over the linear tabulation technique. It uses far less storage as well.

The question naturally arises as to whether there is a *generally applicable* technique, for this speedup, or were we lucky in choosing Fibonacci and Chebyshev sequences. We answer this question in section 4 below.

Even if we did not have this clever transformation based on identities we might not in general expect, it is not hard to write a Macsyma program to compute the values of a linear recurrence which does not encounter exponential overhead, and which does not require permanent allocation of storage for all values in the sequence up to the desired one. This is presented in the next section.

## 3 Two-term Recurrences in Linear Time, Generally

As a matter of general interest, computing the values of two-term recurrences with constant or non-constant coefficients in "linear" time (assuming unit-time evaluation of certain component calculations) can be done by automatically compiling them, via the following template. For the general form

$$f(n) := h(n, f(n-1), f(n-2)), \quad f(0) = f_0, \quad f(1) = f_1,$$

the Macsyma program is<sup>3</sup>

```
(f(n):=if n=0 then f0 else fprime(n,f0,f1),
 fprime(m,fk,fkp1):=
   if m=1 then fkp1 else
   fprime(m-1, fkp1,h(n-m,fkp1,fk))
```

The main program, `f` merely checks one boundary condition and calls an auxiliary function, `fprime`. The function `fprime` is called with three formal arguments, where `fk` is a mnemonic for  $f(k)$ , `fkp1` means  $f(k+1)$  and, initially, an implicit  $k = 0$  with  $f(0) = f_0$ ,  $f(1) = f_1$  initializes the computation. The program proceeds by decrementing  $m$  from  $n$  down to 1 (implicitly incrementing  $k$ ) until  $n = k + 1$ .

It may seem that this recursive function will use a substantial amount of stack space, but many compilers, especially those for Lisp, recognize a tail-recursive function such as `fprime` as equivalent to a loop, and compile it as such. We could choose to explicitly unravel it ourselves to a iteration, at the cost of some clutter.

<sup>3</sup>See also Abelson[1], page 32.

Of course there are cases in which all or nearly all the intermediate terms are required for other purposes, and it would be just as appropriate to compute them along the way. We have no argument there. But to compute and store them *to change the complexity of finding a single value* is unnecessary.

## 4 Linear Recurrences in Log Time

As long as we are considering programming by automatic template, we can go a bit further. We can provide a template that computes the results for certain types of recurrences in *logarithmic* time. The restriction is that the function  $h$  above must be independent of  $n$ . This works for Fibonacci numbers as well as Chebyshev polynomials, but does not work for, for example, Legendre polynomials, which are defined by

$$p_n := (2n - 1)/n xp_{n-1} + ((n-1)/n)p_{n-2}, \quad p_0 = 1, \quad p_1 = x.$$

D. E. Knuth [6] p. 637 cites an observation of J. C. P. Miller and D. J. S. Brown [7] (see also, [2]) that for the general linear recurrence  $x_n = a_1 x_{n-1} + \dots + a_d x_{n-d}$ , we can compute  $x_n$  in  $O(d^3 \log n)$  by computing the  $n$ th power of an appropriate  $d \times d$  matrix<sup>4</sup>. This requires that the coefficients  $\{a_i\}$  do not depend on  $n$ . This certainly works for the Fibonacci example, where we observe that

$$\begin{pmatrix} f_k \\ f_{k-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} f_{k-1} \\ f_{k-2} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{k-1} \begin{pmatrix} f_1 \\ f_0 \end{pmatrix}. \quad P = \begin{pmatrix} \lambda_1 & \lambda_2 \\ 1 & 1 \end{pmatrix}, \quad P^{-1} = \frac{1}{\lambda_2 - \lambda_1} \begin{pmatrix} -1 & \lambda_2 \\ 1 & -\lambda_1 \end{pmatrix}$$

It works as well for the Chebyshev polynomials where the matrix equation looks like this:

$$\begin{pmatrix} T_k \\ T_{k-1} \end{pmatrix} = \begin{pmatrix} 2x & -1 \\ 1 & 0 \end{pmatrix}^{k-1} \begin{pmatrix} T_1 \\ T_0 \end{pmatrix}.$$

Let us call the square matrix  $A$  above. It is a so-called “companion” matrix where the first row consists simply of the coefficients in the recurrence relation. The second row in a 2-term system is always  $(1, 0)$ . (This can be simply checked by setting  $k = 1$  and  $k = 2$ .) Extension to an  $n$ -term system is straightforward,

The general matrix form can be deduced by looking at the  $3 \times 3$  companion matrix. A system generated from the recurrence

$$y_n := c_1 y_{n-1} + c_2 y_{n-2} + c_3 y_{n-3}$$

<sup>4</sup>We have modified the arrangement of the matrix for ease of exposition.

is

$$\begin{pmatrix} c_1 & c_2 & c_3 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}.$$

Naturally it is possible to compute matrix powers by a sequence of repeated squarings and multiplications. Since the computation of the  $k$ th term is dependent upon computing the  $k$  power of the matrix, we have reduced the problem to an operation taking  $O(\log k)$  matrix operations for serial computation. It does not take much imagination to figure schemes whereby this can be parallelized.

This analysis assumes the cost of squaring a matrix is constant, which is not strictly true if the elements grow in size as they do for the Fibonacci recurrence. It is far from true if the elements grow in degree as they do in the Chebyshev sequence.

Another feasible method suggested by W. Kahan is based on decomposing the matrix  $A$  into a product,  $PEP^{-1}$  where  $E$  is a diagonal matrix with the distinct eigenvalues of  $A$  as the diagonal elements, and  $P$  is a Vandermonde matrix of eigenvectors.  $A^n$  can be computed as  $PE^nP^{-1}$ .

Furthermore, powers of  $E$  are easily computed element-wise, since

$$\begin{pmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{pmatrix}^n = \begin{pmatrix} \lambda_1^n & 0 \\ 0 & \lambda_2^n \end{pmatrix}.$$

and powers of numeric eigenvalues can be computed as  $\exp(n \log \lambda)$  in time essentially independent of  $n$ .

For a  $2 \times 2$  system,

For the general  $2 \times 2$  system  $y_k = ay_{k-1} + by_{k-2}$  with companion matrix

$$A = \begin{pmatrix} a & b \\ 1 & 0 \end{pmatrix}$$

the eigenvectors are simply the roots of  $\lambda^2 - a\lambda - b$ .

If one goes through the motions to see the final result of the calculation symbolically in terms of the eigenvalues, we find that the value for  $y_k$  simplifies to:

$$y_k = \frac{(\lambda_2^k - \lambda_1^k) y_1 + (\lambda_1^k \lambda_2 - \lambda_1 \lambda_2^k) y_0}{\lambda_2 - \lambda_1}$$

Not surprisingly, in any particular case, some simplification can be applied, and given the form of the expression, a better numerical scheme can be arranged. A version of this program is also given in the Appendix.

In the case of Chebyshev polynomials, after various simplifications, the program is rather short. It can be stated as the expansion of a polynomial in  $x$  and the algebraic function  $z = \sqrt{x^2 - 1}$ .

```
(algebraic:true, tellrat(z^2-(x^2-1)),
  T(n):= rat(1/2*((x+z)^n+(x-z)^n))
```

This is not a very fast way of computing Chebyshev polynomials; for the 64th polynomial it is only about twice as fast as the “linear” recurrence; it is four times slower than the more direct method below.

Returning to our earlier method of computing powers of the matrix directly, it is possible to compute the elements through a rather more subtle technique which is based on symbolic expansion of the matrix multiplication to obtain specific entries. Although this reduces the computation slightly, the complexity is of the same order. We give two versions of this in the Appendix. The first version does some remembering, in the sense that it saves certain special values it has computed along the way. For example, computing  $y_{2^n}$ , it will store  $n$  constants that will make it possible to compute, for example,  $y_{2^{n-1}}$  rapidly, as well.

The second version does no remembering, and uses only stack space. It does some communication via global variables, which should perhaps be cleaned up for “production” work.

In some simple tests using Macsyma, computing the 32nd Chebyshev polynomial was 2 times faster, and the 64th was 8 times faster using a “logarithmic” method. The difference for Fibonacci is more dramatic, but in each case the time is probably dominated by implementation details of the programming language. Thus we do not see a full  $n/\log n$  speedup. Either program could be rewritten in a language more amenable to efficient compilation, for an additional substantial speedup. A more efficient underlying language implementation would also reduce the initial set-up and provide a higher boost to the logarithmic method.

The above study reinforces the earlier conclusion that “remembering” cannot be justified merely on the basis that it converts exponential-cost  $n$ -term linear recurrence algorithms to linear-cost (or at least polynomial) algorithms. It does so only in the face of insistently naive programming (coded directly from a mathematical statement) using a system that does not itself convert it to a good computational pattern. There are general and simple techniques that use no extra storage; there are less-general and only slightly more complex techniques that may be much faster yet again, and also use no extra storage.

## 5 Remembering: What Else?

Where else can remembering be used? Well, certainly it helps in circumstances in which the filling-in of a tableau in different dimensions, on demand, is a challenge to program.

But, is remembering good for the everyday bread-and-butter of algebraic manipulation? That is, how does it work in situations in which the user is NOT demanding that the system simply recompute the same results again? After all, what is the probability that someone will attempt to factor the very same polynomial twice?

The major situation in which one stands to gain is one in which a single large problem is given to a system which proceeds to break it down into related subproblems, some of which are identical. By identifying such problems, the system will benefit by avoiding recomputation. A trivial example is the removal of simple common subexpressions during evaluation. If we need to evaluate  $(1 + \sqrt{x})/\sqrt{x}$  at the point  $x = 5$  we should not have to compute  $\sqrt{5}$  twice.

Yet the same optimization would not work if the computation of  $\sqrt{x}$  had a side effect: compare, for example, the use of a function which returns a random number, instead of the square-root.

Unfortunately, remembering the results of evaluation may be difficult in algebraic manipulation systems not only for such obvious “impure” functions as a random number generator. It is, in particular, difficult in Macsyma because so many “flags” affect results. For example, computation of  $\sqrt{5}$  is not only affected by the `numer` flag (which results in conversion to floating-point), but also may be affected, in the context of `bigfloat` arithmetic, by the global floating-point precision.

In Macsyma there are several hundred flags, and some can be set to arbitrary values. It would be necessary to impose some check on all these flag settings for consistency in order to re-use a value for the evaluator. It is also necessary to associate with each expression some unique index by which its value can be retrieved. (This was not a problem with Chebyshev or Fibonacci numbers which have natural indices). We have explored this technique [9], using hash-coding of expressions, and after exploring this in some depth, we found the “production” version of the Macsyma evaluator was just not amenable to efficient hash-coding and remembering. Even if the hash-coding cost itself were to be zero, it would still not be a major benefit.

We found that the Macsyma simplifier had much the same difficulty, but plunged ahead with the rather gross approach that we would simply erase all associations based on our lookup tables whenever a flag affecting the simplifier was changed.

Our results were disappointing, especially when compared with a system that uses hash-coding as a uniform technique for unique storage of expressions. In particular, benchmarks run on the Maple system provided by Michael Monagan [8] suggests that by using “option remember” on `expand` and `diff`, in the computation

of the F and G series ([9] for example) one can speed Maple up by a factor of 2 or more. Yet the equivalent in Macsyma actually slowed the computation.

The explanation is fairly simple: Maple already pays the cost for hash-coding (computing indices) for expressions, whether it remembers the results or not. Turning on “option remember” in the case of a highly repetitive computation tends to make it run faster. In the case of Macsyma, it appears that incorporating both hash-coding and remembering is not effective: one major benefit of hash-coding – which is used to map identical expressions to the same location to avoid re-simplification – is achieved by tagging expressions in Macsyma as “already simplified”. Thus the cost of hash-coding is not amortized over as many operations as possible. (see also [9])

## 6 Conclusion

Remembering and using lookup tables is useful, but naive use of them instead of slightly cleverer programs can create problems with space for large problems. For simple recurrences, there are simple linear-style alternatives with less storage, and for linear constant recurrences, we can reduce the cost by a factor of about  $n/\log n$ .

Remembering may work as a substitute for certain kinds of more costly “canonicalization” in algebra systems, especially when a system such as Maple, is oriented to using this approach from the beginning of system design. In work with Carl Ponder [9] we argue in more detail that it doesn’t seem to work well to retrofit an algebra system (in particular, Macsyma) with a remembering facility as a general purpose speedup.

## 7 Acknowledgments

Special thanks to Michael B. Monagan for his contributions of Maple timings with and without hashing; discussions with C. Ponder, W. Kahan (who suggested using eigenvalues for computing matrix powers), and T. Einwohner have influenced parts of this paper.

This work was supported in part by the following grants: National Science Foundation under grant number CCR-8812843; Army Research Office, grant DAAG29-85-K-0070, through the Center for Pure and Applied Mathematics, University of California at Berkeley; the Defense Advanced Research Projects Agency (DoD) ARPA order #4871, monitored by Space & Naval Warfare Systems Command under contract N00039-84-C-0089, through the Electronics Research Laboratory, University of California at Berkeley; the IBM Corporation; a matching grant from the State of California MICRO program.

## 8 Appendix

Here is a Macsyma program that will compute any 2-term linear recurrence “fast” with less waste.

```
/* to compute  $y[n]:=k1*y[n-1]+k2*y[n-2]$ ,  $y[0]=y0$ ,  $y[1]=y1$  rapidly, we set up a scheme to compute the equivalent  $y[n] = a[n]*y[1]+b[n]*y[0]$  . This sets up and saves a logarithmic number of intermediate results in the arrays a and b. This may speed up computation of "other" terms in the
```

```
setup2term(y0,y1,k1,k2):=
( a[1]:1, b[1]:0,
  a[2]:k1, b[2]:k2,
  b[m]:=block([n:quotient(m,2)],
              if evenp(m) then
                b[n]^2+a[n]^2*b[2]
              else
                a[n]*b[2]*(2*b[n]+a[2]*a[n])),
  a[m]:=block([n:quotient(m,2)],
              if evenp(m) then
                a[n]*(2*b[n]+a[n]*a[2])
              else
                (a[n]*a[2]+b[n])^2+a[n]^2*b[2]),
  y[0]:y0,
  y[1]:y1,
  y[n]:=b[n]*y[0]+a[n]*y[1])$
```

```
/* This sets up the function/array y such that  $y[n]=T[n,x]$ , the nth Chebyshev polynomial */
setup2term(1,x,rat(2*x),-1)$
```

```
/* This sets up the Fibonacci numbers */
setup2term(0,1,1,1)$
```

Here is a memory-less version of the same concept. It requires that we have available two functions from Common Lisp, integer-length and logand. These allow us to pick up the bits in the binary expansion of the sequence index, and use appropriate formulas for computing  $2n$  or  $2n+1$  steps. A similar, but uncomfortably long program can be set up for 3 (or more) -term linear recurrences.

```
(h(m,a2,b2,y0,y1):=
block([i:integer_length(m)],
if m=0 then y0 else h1(2^(i-2),1,0)),

h1(s,a,b):=
if s<1 then a*y1+b*y0 else
if logand(m,s)=0 then
  h1(s/2,a*(2*b+a*a2),b^2+a^2*b2)
else
```

```
h1(s/2,(a*a2+b)^2+(a^2*b2),
    a*b2*(2*b+a*a2)) )$
```

```
/*Examples: define Fibonacci */
f(n):=h(n,1,1,0,1)$
```

```
/*define Chebyshev */
t(n):=h(n,rat(2*x),-1,1,x)$
```

Here is a program that computes any 2-term constant coefficient recurrence by the eigenvalue-expansion method. This is not as fast as the above technique, and can suffer numerical problems.

```
s(n,a,b,y0,y1):= block([d:sqrt(4*b+a^2), r1, r2, r1n, r2n],
  r1: (a-d) /2, r2: (a+d)/2,
  /* r1, r2 are eigenvalues */
  r1n: r1^n, r2n:r2^n,
  ((r2n-r1n)*y1+ (r1n*r2 -r1*r2n)*y0)
  /(r2-r1))$
```

```
/*define Fibonacci */
f(n):=s(n,1.0,1.0,0,1)$
```

```
/*define Chebyshev */
t(n):=s(n,2*x,-1,1,rat(x))$
```

## References

- [1] Harold Abelson, Sussman, G.J., and Sussman, J. *Structure and Interpretation of Computer Programs*, MIT Press/ McGraw-Hill Book Co., New York, N.Y. 1985.
- [2] M. Beeler, R.W. Gosper, R. Schroepfel. "HAK-MEM," MIT Artificial Intelligence Lab. Memo 239, 1972. items 13, 14.
- [3] Bruce Char *et al.* *On the Design and Performance of the Maple System*. Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, CS-84-13, 1989.
- [4] Richard Fateman. Macsyma's General Simplifier: Philosophy and Operation. In Lewis, V.E. (ed), *Proceedings of the 1979 Macsyma Users Conference*. Washington D.C., 1979. 563-582.
- [5] Jeffrey Greif. The SMP Pattern Matcher. In B.F. Caviness (ed), *Proc. Eurocal '85*, vol. 2, Lecture Notes in Computer Science 204, Springer-Verlag, 1985, 303-314
- [6] D. E. Knuth. *The Art of Computer Program, volume 2: Seminumerical Algorithms*, second edition, Addison-Wesley, Reading Ma. 1984.
- [7] J. C. P. Miller and Brown, D. J. S. An algorithm for evaluation of remote terms in a linear recurrence sequence, *Comp. J.* **9** (1966), 188-190.
- [8] Michael B. Monagan, private communication.
- [9] Carl Ponder. Augmenting expensive functions in Macsyma with lookup tables, Chapter 9 in *Evaluation of "Performance Enhancements" in Algebraic Manipulation Systems*, Ph.D. diss. Univ. Calif. at Berkeley, Dept. of EECS, also UCB/CSD 88/453 p. 85-100.
- [10] Stephen Wolfram. *Mathematica - A System for Doing Mathematics by Computer* Addison-Wesley Publ. Co., Redwood City, CA., 1988.