Tabulation Techniques for Recursive Programs

R. S. BIRD

Department of Computer Science, University of Reading, Reading, Berkshire RG6 2AX, England

Many recursively defined functions specify redundant computations in that subsidiary function values may be evaluated more than once. Such recursions can be implemented more efficiently by a process of tabulation in which function values are computed at most once and then stored for later use. This tutorial paper examines a number of general strategies for introducing tabulation into recursive programs. The advantages of tabulation as a technique of recursion elimination are explored through various programming examples, and a simple theorem quantifying the potential increase in efficiency is presented.

Keywords and Phrases: recursive functions, redundancy, tabulation, recursion elimination, program transformation, pebble games, dynamic programming

CR Categories: 4.0, 4.2, 5.20, 5.24, 5.25, 5.30

INTRODUCTION

One source of inefficiency in the evaluation of recursively defined functions or procedures is that the same calculation may be repeated many times over. The most often quoted example of this phenomenon is the recursive definition of the Fibonacci function:

$$fib(x) \text{ is if } x \leq 2 \text{ then } 1 \text{ else } fib(x-1) + fib(x-2) \text{ fi} \quad (1)$$

For positive integer values of x *fib* computes the xth term in the sequence of Fibonacci numbers. It is well known that $fib(x) = O(\phi^x)$ where ϕ is the golden ratio $(1 + \sqrt{5})/2 = 1.618\ldots$. The trouble with using the above definition to compute *fib* is that it is very inefficient; the evaluation of $fib(x)$ requires two separate evaluations of $fib(x-2)$, three evaluations of $fib(x-3)$, and so on. This duplication of effort means that the time to compute $fib(x)$ using the definition is exponential, in fact proportional to ϕ^x.

On the other hand, the work done in computing *fib* can be greatly reduced by a process in which function values are computed once only and then stored in some conveniently represented table for future use. Subsequent requests for the value are answered by looking up the appropriate table entry. This strategy is known by a number of names but we refer to it simply as one of *tabulation.* Various concrete representations of the table are possible, depending on the nature of the problem in hand; no specific structure is implied by the choice of name. Assuming the object of the exercise is to obtain the value $f(x_0)$ for a given recursively defined function f and argument x_0, all that is required in essence of the table structure is that it should be capable of

(1) storing, at each stage, those previously computed values of f which are needed to continue the evaluation of $f(x_0)$, and
(2) determining for a given argument x whether or not the value $f(x)$ has pre-

CONTENTS

viously been computed, and if so what it is.

Requirement 1 is a bit vague, but at least its import should be clear: function values need not be retained beyond their useful life. Of course it is only possible to take advantage of this fact if one can determine in advance the useful life span of function values. The Fibonacci definition provides a simple illustration of this point. One way to introduce tabulation is to use a linear array F as in the following procedure.

```
fib(x) is begin array F[1:x]; int k;
            F[1] := F[2] := 1;
            for k := 3 upto x
            do F[k] := F[k − 1]                    (2)
               + F[k − 2] od;
            return F[x]
          end
```

However an array is not really essential here because at each stage of the computation only the previous two values of f are required to continue the evaluation. Thus the procedure

```
fib(x) is begin int u, v, t := 1; int k;
            for k := 3 upto x
            do t := v; v := u + v;                 (3)
               u := t od;
            return v
          end
```

will do the job just as well and uses only three integer variables (of which one, t, is used only as a temporary variable to implement what is essentially a simultaneous assignment to u and v). In either case the recursive definition is transformed into a nonrecursive algorithm with a linear running time.

Tabulation as a means of improving the efficiency of recursive programs is a familiar device to programmers. One early reference is a research memorandum by McCarthy [McCa], who applies the idea to a recursive definition of the number of partitions of a given integer. Tabulation forms a major component of a basic tool of algorithm design known as dynamic programming, a technique popularized by Bellman [Bell57]. Two recent texts which discuss dynamic programming applications are Aho, Hopcroft, and Ullman [Aho75] and Horowitz and Sahni [Horo79]. Many combinatorial and language-theoretic problems are susceptible to treatment by tabulation techniques (see, e.g., Rein77, Well71, and Aho72). Tabulation is also closely related to the idea of "memo functions" advocated by Michie [Mich67]. Nevertheless, the extent of the role played by tabulation as a technique of program transformation is not so well appreciated as it might be. For instance, the elegant linear pattern matching algorithm of Knuth, Morris, and Pratt and the related linear time simulation of a class of exponential algorithms due to Cook (see Aho75) can both be systematically derived by using tabulation in exactly the same way as in the case of the Fibonacci function (the pattern matching algorithm is derived in Bird77b, and a tabulation-based proof of Cook's theorem appears in Wata80). Furthermore, characterization of some of the forms of redundancy which can be exhibited by recursively defined functions has only recently been undertaken, in Cohe79, for example, and the mathematics of the problem has some interesting aspects.

For these reasons it is worthwhile taking a closer look at the problem of tabulation. In the next section we introduce the fundamental concept of dependency upon which the whole basis of tabulation rests. In Sections 2–4 we discuss a number of strategies for introducing tabulation into a recursive program. Section 4 also contains a sketch of the proof of a general theorem on the power of tabulation as an optimizing transformation. Loosely speaking, this theorem states that any recursive function

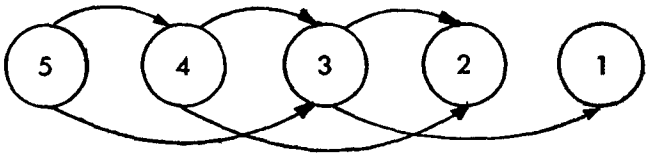


FIGURE 1. *G(fib, 5)*.

which can be evaluated in a time linearly related to the number of subsidary function calls can always be transformed into an equivalent program with an $O(n)$ running time, where n is the number of distinct arguments for which the function value is required. For instance, it follows from the theorem that any number function $f(x)$ taking arguments in the range $0 \leq x \leq n$, and which possesses a bounded number of recursive calls in its defining body, can be evaluated in $O(n)$ steps provided the other operations and conditions appearing in the definition can be carried out in constant time.

The reader is assumed to be familiar with the idea of recursion and to be able to understand how recursive programs work. There are many sources for this material; WIRT76 is especially good. Most of the mathematics in this paper is straightforward, and the reader is encouraged to try to "hand simulate" the execution of the various programs to see how they operate and to better understand the mathematical formulations.

1. THE DEPENDENCY GRAPH

The Fibonacci definition has a straightforward tabulation because the dependency relationship between function values is particularly simple: $fib(x)$ depends at most on $fib(y)$ for $y < x$ so table entries can be computed in ascending order. The notion of dependency is an important one and can be made precise as follows. With each recursive definition of a function f and given argument x for which $f(x)$ is defined, one can associate a certain directed acyclic graph G called the *dependency graph* of f and x. The nodes of G consist of a source node labeled x together with further nodes, one for each value y such that $f(y)$ is needed for the calculation of $f(x)$. The presence of a directed edge from y to z in G signifies that $f(y)$ depends directly on the value $f(z)$, that is, $f(z)$ is needed at the first level of recursion. A directed path from y to z means that $f(z)$ is required at some stage during computation of $f(y)$. Clearly, if G possessed a cyclic path, some function value would depend on itself, corresponding to an infinite computation sequence in the recursive definition. But we are supposing $f(x)$ to be defined, so this situation cannot arise and G must be acyclic.

It is important to observe that dependency graphs are associated with given recursive definitions of functions, not with the functions themselves. Different definitions of the same function may have quite unrelated graphs. Moreover, there need not be any uniform relationship between dependency graphs for varying arguments even when the definition is kept fixed. Figure 1 gives the dependency graph for the Fibonacci function with argument 5.

As further examples, consider the definitions

$$f(x, y) \text{ is if } y = 0 \text{ then } x \text{ else } f(x, y-1) + f(x+1, y-1) \text{ fi} \quad (4)$$

$$s(x) \text{ is if } x = 0 \text{ then } 1 \text{ else } s(s(x-1)-1) + 1 \text{ fi} \quad (5)$$

The graphs $G(f, 1, 3)$ and $G(s, 3)$ appear in Figures 2 and 3.

Figure 3 takes a little thought because (5) is a tricky second-order recursion in which recursive calls are nested, that is, used as arguments for further calls. However the picture becomes clear once it is realized that (5) is just a rather elaborate (and time-consuming) way of calculating $s(x) = x + 1$. This can be proved by straightforward induction on x. Of course the construction of dependency graphs associated with second- or higher order recursions re-

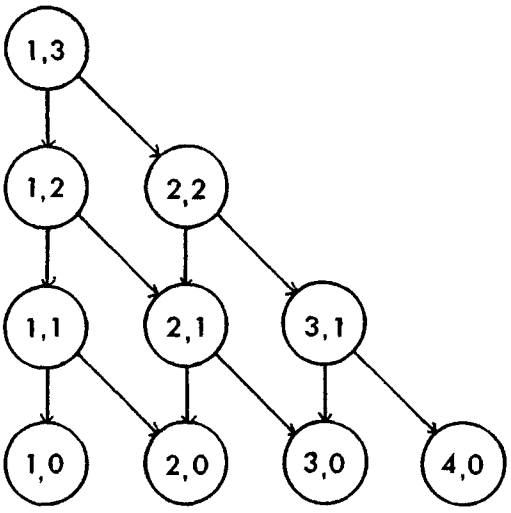


FIGURE 2. $G(f, 1, 3)$.

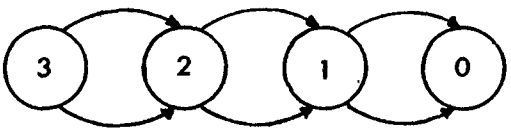


FIGURE 3. $G(s, 3)$.

quires prior knowledge of those function values which appear as arguments to other function calls, so such graphs are of limited use in actually deriving tabulation schemes.

So far we have been assuming the case of a single recursive definition. When f is defined as one of a set of mutually recursive functions, the dependency graph has to carry an additional label on each node to indicate the function to which the associated argument applies. Alternatively, and what in the end is the same thing, one can always define a set of mutually recursive functions in terms of a single recursive function over an argument domain augmented with a finite component to identify the original functions.

Each dependency graph induces a partial ordering of its vertices and the problem of efficient tabulation can be viewed in part as the problem of embedding this partial order in a linear order, that is, arranging the nodes into a linear sequence $v_1, v_2, \ldots, v_n$ such that $j < k$ whenever there is a path from v_k to v_j in the graph. Once this is done the tabulation of f can be put in the form

for $j := 1$ **upto** n
do compute and store $f(v_j)$,
    using previously computed values
    as necessary **od**;

The other aspect of the problem concerns the efficient management of space, for it is clearly wasteful to keep computed function values beyond the period of their usefulness. One instructive way of thinking about the problem of space is in terms of a particular *pebble game* played on the dependency graph. This pebble game was first described by Paterson and Hewitt [PATE70] and further developed by Walker [WALK71]. In the game there is a potentially infinite supply of labeled pebbles; the supply is called the *pool*. Each pebble can be thought of as a distinct unit of space such as a register, variable, or table entry. At each move of the game, a pebble is placed on a node of the graph, and the game ends when the source node—that is, the node corresponding to the desired function value—is pebbled. Legal moves have to obey the following conditions:

(C1) A pebble can always be placed on a terminal node, that is, a node with no descendants.

(C2) A pebble can only be placed on a nonterminal node x when there are already pebbles on every immediate descendant of x.

In each case the pebble placed on the node may be a new pebble from the pool or one already on the graph. In particular, one can choose to move a pebble from an immediate descendant of x in order to cover x; this is a fairly common type of move. The object of the game is to pebble the source using the smallest number of pebbles, and this clearly corresponds to the problem of evaluating the desired function value using the least number of units of space. For example, consider the Fibonacci graph of Figure 1. We can pebble G using just two pebbles as follows (the notation $u \leftarrow k$ means "cover node k with pebble labeled u"):

$$u \leftarrow 1; \quad v \leftarrow 2; \quad u \leftarrow 3; \quad v \leftarrow 4; \quad u \leftarrow 5$$

The strategy can be extended to deal with

arbitrary arguments and corresponds to the tabulation of the Fibonacci function via the algorithm

```
fib(x) is begin int u, v := 1; int i;
          for i := 1 upto ⌊x/2⌋ − 1
            do u := u + v; v := u + v od;
          if x even
            then return v
            else u := u + v; return u fi
        end
```

However in some cases pebbling a graph with the smallest number of pebbles can only be achieved at the expense of pebbling a node more than once. Consider, for example, the graph G of Figure 4. G cannot be pebbled with fewer than three pebbles (why?), and three are in fact sufficient, but only by a strategy such as

$$u \leftarrow g;\quad v \leftarrow h;\quad w \leftarrow e;\quad u \leftarrow c;\quad v \leftarrow d$$
$$w \leftarrow b;\quad u \leftarrow g;\quad v \leftarrow h;\quad v \leftarrow f;\quad u \leftarrow a$$

in which the terminal nodes g and h are pebbled twice. This corresponds to a recomputation of certain function values, so the price paid for minimizing space usage is an increase in time. Time-space trade-off relations are what make pebble games so interesting, and they have been studied by a number of people (for recent references see PAUL78 and PIPP78). Of course we can always change the nature of the game with an extra condition:

(C3) No vertex may be pebbled more than once.

Thus we study only those pebbling strategies guaranteed to be as speedy as possible.

As far as the problem of tabulation is concerned, pebble games are only useful when there is some uniformity among the dependency graphs associated with different arguments. It is no good having a brilliant strategy for one particular graph if it is only applicable for one input. Furthermore, the given recursive definition has to yield sufficient information about the form of the dependency graph in order for a coherent pebbling strategy to be devised. Some simple examples of the sort of recursive definitions which are susceptible to this kind of treatment are examined in the next section.

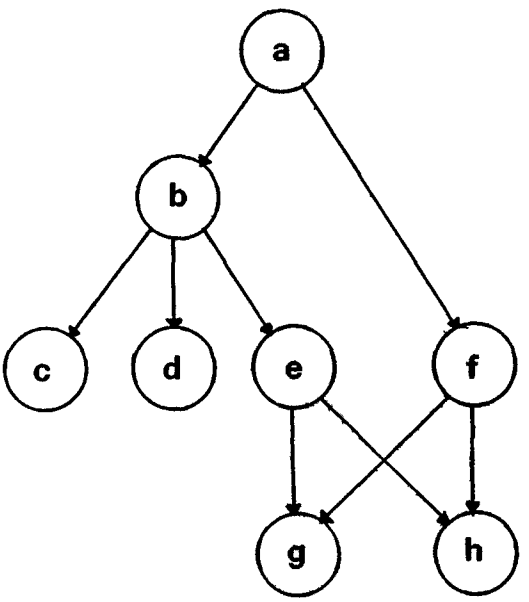


FIGURE 4. Pebble game graph.

2. TABULATION BASED ON DESCENT CONDITIONS

Some degree of uniformity among the dependency graphs associated with a given function will always be present when certain functional relationships between the arguments of recursive calls are satisfied. Consider, for instance, a scheme

$$f(x) \text{ is if } p(x) \text{ then } c(x) \text{ else } \theta(x, f(a(x)), f(b(x))) \text{ fi} \qquad (6)$$

of the same general form as the Fibonacci definition, in which f is recursively defined in terms of given functions p, c, θ, a, and b. The functions a and b are called *descent functions*. Now it may be the case that functions a and b commute, so $a(b(x)) = b(a)(x))$, that is, $ab = ba$. If they do, some uniformity is imposed on the dependency graphs and some measure of redundancy is involved in the definition of f. A number of the recursion removal schemes of Burstall and Darlington [BURS75] are based on simple relations of this kind. More recently, N. Cohen (who calls such relationships *descent conditions*) has developed uniform tabulation strategies for a simple hierarchy of descent conditions, of which commutativity is the weakest. Cohen's hierarchy

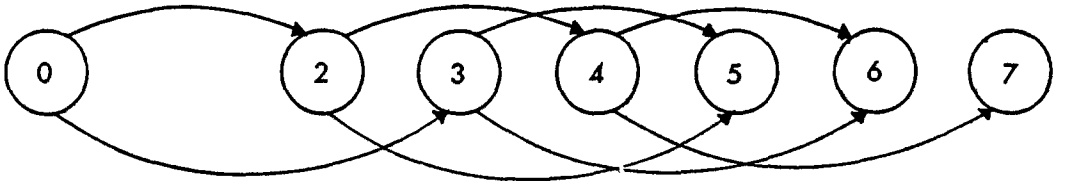


FIGURE 5. $r = 2, s = 3, n = 5$.

does not include all the cases that can arise, but it does represent the first systematic approach to the problem of efficient tabulation through analysis of descent conditions. We can give the flavor of the sort of things which can be done with descent conditions by looking at some simple examples. Descent conditions are useful in that they can provide a general framework for discussing a variety of actual descent functions based on their properties. For concreteness, however, most of our examples emphasize specific instances of recursive programs.

Example 1

Suppose in (6) there is a function e and two positive integers r and s such that $a = e^r$ and $b = e^s$ (e^k denotes the k-fold composition of e with itself). The Fibonacci definition satisfies this condition since we can take e to be the predecessor function, $r = 1$ and $s = 2$. Suppose also that $p(x)$ implies $p(ex)$ for all x. The purpose of this condition is to ensure an easily identified boundary between terminal and nonterminal nodes in the dependency graph; such conditions are called *frontier* conditions for obvious reasons. Given these assumptions, the dependency graph of (6) for varying inputs x takes on a simple form which, apart from r and s, depends only on the value of $n(x)$, $n(x)$ being the least value of n which makes $p(e^n x) = true$. Figure 5 pictures the case $r = 2, s = 3, n(x) = 5$.

In Figure 5, a node labeled j denotes the argument $e^j(x)$. Notice how the frontier condition ensures a clean separation between terminal and nonterminal nodes. This graph, and indeed the graph for any x for which $n(x)$ is finite, can be pebbled using three pebbles (for general r and s, $\max(r, s)$ pebbles will be required). This fact forms the basis of the following tabulation of $f(x)$ in the case $r = 2, s = 3$.

The first job is to determine $n = n(x)$. The values $c(e^k x)$ for $n \le k \le n + 2$ are set up in three variables: u, v, and w. The frontier condition guarantees that they are just the values of $f(e^k x)$ for $n \le k \le n + 2$. Next the assignments

$$w := \theta(e^{k-1}x, v, w)$$
$$v := \theta(e^{k-2}x, u, v)$$
$$u := \theta(e^{k-3}x, w, u)$$

are performed, in order, for $k = n, n - 3, \ldots$, continuing for as long as $k \ge 3$. The assignments correspond to the obvious way of pebbling the graph until the left boundary is reached. Finally, depending on the value of n (mod 3), a further assignment is performed to bring the final answer into some specified variable, say u. Naturally, the values $e^k(x)$ for $0 \le k \le n + 2$ have to be made available, at least if θ does depend on its first argument. These need not be stored if $e^k(x)$ has an explicit formula or if e is invertible, so that each can be obtained from its successor by applying e^{-1}. If neither of these pleasant properties holds, then they have to be precomputed ready for use. Actually, these values are not all required at the same time, and one can use various constructions due to Chandra [CHAN73] that are guaranteed to produce the desired sequence using only an amount of space proportional to $\log n$, or even a constant. Unfortunately, if Chandra's constructions are used, there is a penalty to be paid by way of an increase in running time.

Observe that the frontier condition is not really an essential feature of the tabulation. Once the right boundary is found (by a more complicated search, to be sure) we can work backward pebbling terminal and nonterminal nodes alike; the only change necessary is to make each assignment conditional on p for the relevant argument. Notice also that the algorithm may some-

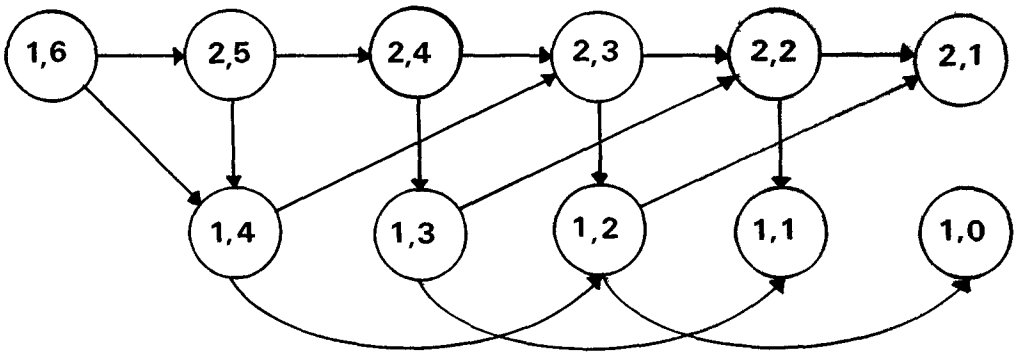


FIGURE 6. $a^2b = ab^3$, $bab = b^3$.

times make a redundant evaluation, namely, of $f(ex)$, the missing node 1 in Figure 5.

Example 2

As a second illustration of the kind of thing that can be done with descent conditions, consider the following mutually recursive definitions of two functions f and g:

$$\begin{aligned} f(x) &\text{ is if } x \leq 1 \text{ then } 1 \\ &\quad \text{else } 2*f(x-2) - 3*g(x-1) \text{ fi} \\ g(x) &\text{ is if } x \leq 1 \text{ then } 1 \\ &\quad \text{else } f(x-1) + g(x-1) \text{ fi.} \end{aligned} \tag{7}$$

Suppose it is desired to evaluate $f(n)$ for some positive integer n. As indicated in the previous section, it is possible to compress these definitions into a single recursive definition of a function h designed so that $h(1, x) = f(x)$ and $h(2, x) = g(x)$. The definition is

$$\begin{aligned} h(s, x) &\text{ is if } x \leq 1 \text{ then } 1 \\ &\quad \text{else } \theta(s, h(a(s, x)), h(b(s, x))) \\ &\quad \text{fi,} \end{aligned} \tag{8}$$

where $a(s, x) = (1, x - 3 + s)$, $b(s, x) = (2, x - 1)$, and

$$\begin{aligned} \theta(s, x, y) = &\text{ if } s = 1 \text{ then } 2*x - 3*y \\ &\text{else } x + y \text{ fi.} \end{aligned}$$

Here a and b satisfy the descent conditions $a^2b = ab^3$ and $bab = b^3$. The dependency graph G of h for the argument (1, 6) appears in Figure 6. Here, arcs that enter nodes with first subscript 1 are identified with application of the function a, while those entering nodes with first subscript 2 are applications of b. Notice that application of b followed by two applications of a (i.e., a^2b) is equivalent to three applications of b followed by one of a (i.e., ab^3). This is the first descent condition, and the other is similar. G can also be obtained directly from the definitions in (7), without explicitly identifying the descent conditions.

It should be fairly easy to see how G can be pebbled using just three pebbles: one pebble u, which is used to pebble the nodes along the top line from right to left, and two pebbles v and w, which move alternately along the nodes in the bottom line, also from right to left. When this strategy is implemented in terms of the associated assignment instructions, we arrive at a method for evaluating $f(n)$ in $O(n)$ steps, using just three integer variables.

The example is also interesting from another point of view, since the form of the definitions (7) is such that a much faster way of computing $f(n)$ can be given. It comes from representing the recursive case of (7) as a matrix equation

$$\begin{pmatrix} f(x) \\ f(x-1) \\ g(x) \end{pmatrix} = \begin{pmatrix} 0 & 2 & -3 \\ 1 & 0 & 0 \\ 1 & 0 & 1 \end{pmatrix} \begin{pmatrix} f(x-1) \\ f(x-2) \\ g(x-1) \end{pmatrix}$$

in which a vector $\mathrm{v}(x) = (f(x), f(x-1), g(x))$ of selected function values is related to $\mathrm{v}(x-1)$ by the linear transformation implicit in the definitions. It follows that $\mathrm{v}(x) = A^{x-1}\,\mathrm{v}(1)$, where A denotes the above 3×3 matrix. Now A^{x-1} can be eval-

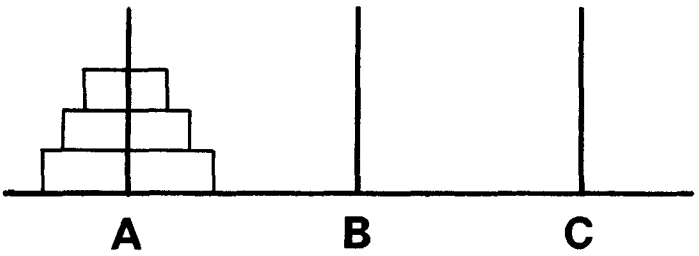


FIGURE 7. Towers of Hanoi puzzle.

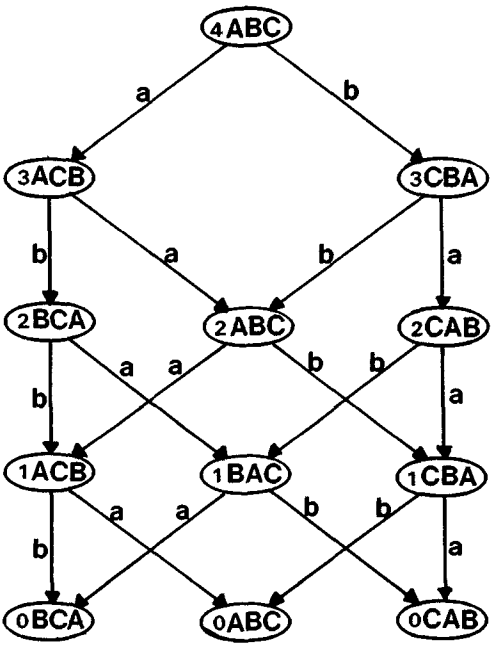


FIGURE 8. $a^2 = b^2$, $aba = bab$.

uated in $O(\log x)$ steps in any one of a number of ways for computing powers quickly (see, e.g., KNUT69). In fact if we write $p(x) = A^x$, then p satisfies the recursive definition

$p(x)$ **is if** $x = 0$ **then** 1
elsf x *odd* **then** $A * p(x - 1)$
else $p(x/2) * p(x/2)$ **fi**

which is also a definition susceptible to treatment by descent conditions, since the condition satisfied in this case is the simplest one of all, namely, $a = b$! Naturally the method, which leads to an $O(\log n)$ algorithm for computing $f(n)$, only works because of the way f and g are defined in terms of addition and multiplication, though it is capable of generalization to other operations (see STEI77).

Example 3

In the Towers of Hanoi problem [PART76], it is required to move a pile of n disks of increasing size from a peg A to a peg B, using a third peg C as intermediate storage, under the restriction that no disk may ever be placed on top of a smaller one. The situation is pictured in Figure 7.

One can formulate a recursive solution to the problem, as a sequence of moves to be performed, in the following way:

$$\begin{aligned}&\text{seq}(n, A, B, C) \textbf{ is if } n \neq 0\\ &\qquad \textbf{then } \text{seq}(n-1, A, C, B)\\ &\qquad :: \text{move } (A, B)\\ &\qquad :: \text{seq}(n-1, C, B, A) \textbf{ fi}\end{aligned} \tag{9}$$

Here the operation :: signifies concatenation, and move(A, B) denotes the act of moving one disk from A to B. Definition (9) corresponds to an interpretation of the scheme (6) in which $a(n, A, B, C) = (n - 1, A, C, B)$ and $b(n, A, B, C) = (n - 1, C, B, A)$. As such, the functions a and b satisfy the descent conditions $a^2 = b^2$ and $aba = bab$. The dependency graph for the argument $(4, A, B, C)$ is depicted in Figure 8.

In Figure 8 the edges have also been labeled with the appropriate descent functions. The pebbling of this dependency graph can be carried out using just four pebbles. There are a number of ways to do it; one version is as follows. After setting up u, v_0, and w along the baseline, we can pebble the middle node in the line of nodes immediately above with a new pebble v_1,

and then move u and w up to bring the pebbles u, v_1, and w to the next level. Subsequent levels are pebbled in the same fashion but alternating between v_0 and v_1. When the top level has been constructed, the "roof" is put on by a further pebbling operation. Notice that the final value of v_1 (or v_0) contains a computed value not required for the final answer.

Example 4

Finally, let us consider a different type of descent condition. In addition to its usual definition, the Fibonacci function may also be defined by the equations

$$fib(2n) = fib^2(n) + fib^2(n+1) \quad \text{if} \quad n > 1$$

$$fib(2n+1) = 2fib(n)fib(n+1) + fib^2(n+1) \quad \text{if} \quad n \geq 1$$

(see, e.g., VORO66 for a proof of these formulas). This means that *fib* can be constructed according to the definition

fib(*x*) **is if** $x \leq 2$ **then** 1
else $\theta(x, fib(a(x)), fib(b(x)))$ **fi**,

where

$$a(x) = [x/2], \qquad b(x) = [x/2] + 1$$

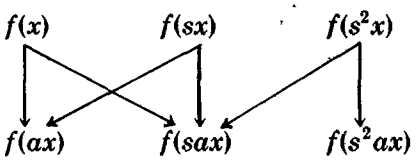
and

$$\theta(x, y, z) = y^2 + z^2 \quad \text{if } x \text{ is even},$$
$$= 2yz + z^2 \quad \text{if } x \text{ is odd}.$$

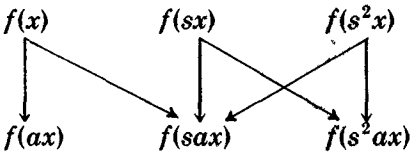
The descent conditions in this case are not quite as simple as in previous examples, for a and b cannot be related on the basis of the composition operator alone. However we do have $b = sa$, where s is the successor function, and it is also true that $as^2 = sa$ and

$$as(x) = a(x) \quad \text{if } x \text{ is even},$$
$$= sa(x) \quad \text{if } x \text{ is odd}.$$

These conditions imply that the values $f(x)$, $f(sx)$, and $f(s^2x)$ can be obtained from the values $f(ax)$, $f(sax)$, and $f(s^2ax)$ (abbreviating *fib* by f). If $x > 2$ is even, then we will have



as part of the dependency graph. On the other hand, if $x > 2$ is odd, the picture will be



Thus the dependency graph of f looks like a ladder with three values on each rung. If three variables u, v, and w hold these values, they can be moved up one rung by the code

```
if x even
  then t := 2uv + v²;
       u := u² + v²;
       w := v² + w²
  else t := v² + w²;
       u := 2uv + v²;
       w := 2vw + w²
fi;
v := t.
```

On the bottom rung we have, depending on the initial value of u, either the values of $f(1)$, $f(2)$, $f(3)$ or the values of $f(2)$, $f(3)$, $f(4)$; in each case these values can be computed directly. The tabulation can be cast as a function that computes the values $f(x)$, $f(x+1)$, $f(x+2)$ into the (global) variables u, v, and w, and returns u:

```
fib(x) is begin int u, v, w;
          proc F(x) is
          if x = 1
          then u := 1, v := 1; w := 2
          elsf x = 2
          then u := 1; v := 2; w := 3
          else F(⌊x/2⌋);
                  move up one rung
          fi;
          F(x);
          return u
          end
```

This procedure yields a tabulation of *fib*(x) in $O(\log x)$ arithmetic steps; for an alter-

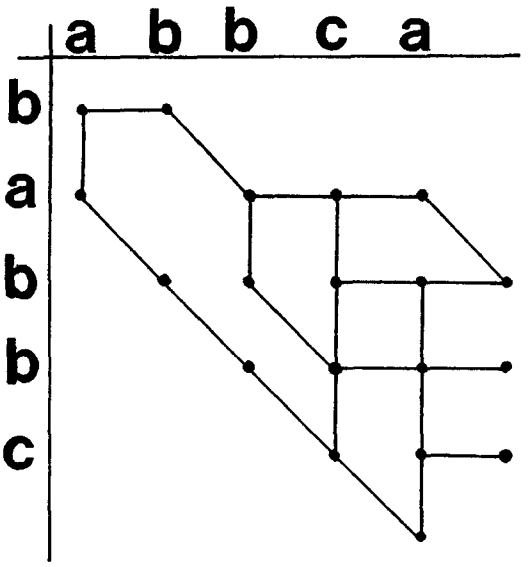


FIGURE 9. Dependency graph of LCS function.

native version see SHOR78. (An even simpler $O(\log x)$ method for $fib(x)$ can be based on the matrix equation idea of Example 2.)

Before leaving this example, it is instructive to see the general form of the descent conditions used in the tabulation. Basically we require that $b = sa$ for some function s, not necessarily the successor function, together with the additional property that there exists a positive integer r and a function σ such that for $0 \leq j \leq r$

$$as^j(x) = s^{\sigma(j,x)}a(x)$$

where $0 \leq \sigma(j, x) < r$. In the case of the Fibonacci definition we have $r = 2$, $\sigma(0, x) = 0$, $\sigma(1, x) = 0$ if x is even, 1 if x is odd, and $\sigma(2, x) = 1$. The general condition ensures that the values of $f(x)$, $f(sx)$, ..., $f(s^r x)$ can be obtained in some manner from the values of $f(ax)$, $f(sax)$, ..., $f(s^r ax)$, so the tabulation can be carried out using $(r + 1)$ variables.

Tabulation schemes as exemplified in this section are not always possible of course. For one reason or another the descent functions may just not satisfy any useful conditions; the shape of the recursion may be too complicated, or the descent functions may change with different inputs. Also, such delicate and finely tuned tabulations only apply in highly specific instances and, though it is a very interesting problem to characterize their range of application, the whole approach may well be of limited practical importance. What is really needed is a cruder method which can be employed in more general situations. We take up these points in the following two sections.

3. OVERTABULATION

Although a given class of dependency graphs $G(f, x)$, x varying, may not have the sort of uniform properties described in the previous section, it may be possible to embed the class in a second one which does. Each graph in the second class contains a member of the former as a subgraph. To put the point another way, the order of evaluation of subsidiary function values, upon which the final value depends, may be difficult to determine; but it may be possible to embed the order in an extended sequence which can be found more easily. In the extended sequence, function values are calculated which are not necessarily crucial to the final answer. We call schemes based on such extended sequences *overtabulations*. A simple illustration of the idea is provided by the following recursive definition which arises in connection with the problem of finding the longest common subsequence of two strings $x_1\, x_2 \cdots x_n$ and $y_1\, y_2 \cdots y_m$ (see HIRS75)

$$\begin{aligned} f(j,k)\ &\textbf{is if } j > n \textbf{ or } k > m \textbf{ then } 0 \\ &\quad \textbf{elsf } x_j = y_k \\ &\quad \textbf{then } 1 + f(j+1, k+1) \\ &\quad \textbf{else } max\{f(j, k+1), f(j+1, k)\} \\ &\quad \textbf{fi} \end{aligned} \tag{10}$$

The value of $f(j, k)$ gives the length of the longest common subsequence of $x_j x_{j+1} \cdots x_n$ and $y_k y_{k+1} \cdots y_m$, and the object of the exercise, in part, is to calculate $f(1, 1)$. Now the dependency graph of $f(1, 1)$ can be quite complex, depending on the particular constitution of the two strings (which are, of course, also parameters of the definition). For example, Figure 9 illustrates the case $x = babbc$ and $y = abbca$.

However, no matter what the given strings are, the dependency graph will always be a subgraph of the one depicted in Figure 10. Thus f can always be tabulated

with the help of a two-dimensional array $F[1{:}n + 1, 1{:}m + 1]$ initially cleared to 0, as in the following algorithm.

```
for j := n downto 1
do for k := m downto 1
   do F[j, k] :=
   if x_j = y_k then 1 + F[j + 1, k + 1]
   else max {F[j, k + 1], F[j + 1, k]}
   fi od od                                   (11)
```

Perhaps slightly less obvious is the fact that only a one-dimensional array $F[1{:}m + 1]$ is needed, as in

```
for j := n downto 1
do for k := m downto 1
   do F[k] :=
   if x_j = y_k then 1 + F[k + 1]
   else max {F[k], F[k + 1]}
   fi od od                                   (12)
```

These tabulations, which have a running time proportional to mn, are therefore succinctly expressed and more efficient than using the recursive definition (which may take up to 2^{mn} steps in the event that the two strings have no common symbol), but on the other hand they may calculate up to $(m + 1)(n + 1) - min(m, n)$ unnecessary values of f (in the event that one string turns out to be an initial substring of the other).

As a second illustration of the value of overtabulation, consider the function

```
f(j, x) is if j = 0 then 0
        elsf x < w_j then f(j − 1, x)
        else max {f(j − 1, x),
             f(j − 1, x − w_j) + p_j)} fi          (13)
```

which arises in connection with the *knapsack* problem (see, e.g., HORO79). This problem can be described as follows: We are given n items with positive integer weights $w_1, w_2, \ldots, w_n$ and a knapsack which can be filled with a selection of the items up to a total weight W. Packing item k yields a profit p_k, also specified as part of the problem. The task is to determine the maximum profit obtainable by a suitably wise choice of items, and definition (13) is the result of solving the problem by a fairly typical dynamic programming scheme: $f(j, x)$ is the maximum profit obtainable from a selection among the first j items when the remaining capacity of the knapsack is x. The answer wanted is therefore $f(n, W)$.

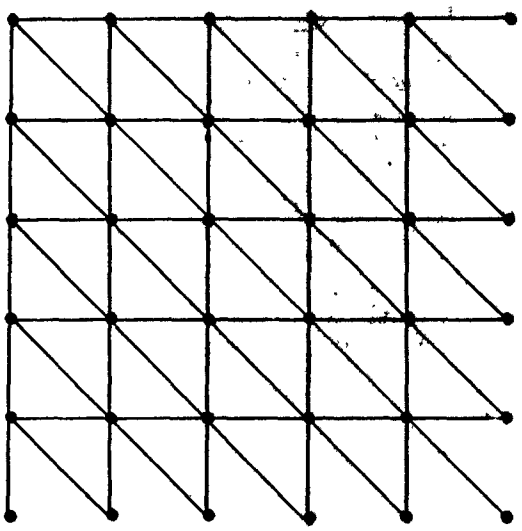


FIGURE 10 Supergraph of LCS function.

The dependency graph of $f(n, W)$ depends on the values of $w_1, w_2, \ldots, w_n$ and can be quite intricate; nevertheless, each $f(j, x)$ depends at most on the values $f(k, y)$ for $0 < k \leq j$ and $0 \leq y \leq x$, and this leads to an acceptable overtabulation of f. We can use a one-dimensional array $F[1{:}W]$ in much the same way as the subsequence problem:

```
for x := 0 upto W do F[x] := od;
for j := 1 upto n
do for x := w_j upto W
   do F[x] := max{F[x], F[x − w_j] + p_j}
   od od                                      (14)
```

Once again the tabulation is succinct but straightforward, more efficient than the recursive program, but still potentially wasteful of space and time.

4. EXACT TABULATION

Even if the dependency graph has no useful uniform properties and cannot be embedded in one which has, there still remains a further avenue of attack. When all else fails one can always use the recursive definition itself to direct an efficient tabulation scheme. After all, the given definition does implicitly contain all the necessary information about dependent function values and the order in which they are required. A

recursive top-down approach will therefore succeed even when there is no obvious way of implementing one of the essentially bottom-up methods of previous sections.

In order to illustrate the idea, consider again the definition

$$f(x) \text{ is if } p(x) \text{ then } c(x) \text{ else } \theta(x, f(ax), f(bx)) \text{ fi} \quad (15)$$

Function f can be tabulated by constructing a recursive procedure R which acts on a globally defined table T of computed function values. For convenience we also suppose that maintained along with T is a list S of those arguments whose corresponding values appear in T. Initially both S and T are empty. A call $R(x)$ of the procedure R is designed to carry out the following operations:

(1) If $x \in S$, then $R(x)$ is to do nothing since, by assumption, value $f(x)$ already appears in T.
(2) If $x \notin S$, then $R(x)$ is to compute $f(x)$, store it in the table T, and add x to S.

The definition of such an R is quite straightforward since it is derived directly from (15). In the following version the notation $x \rightarrow S$ is used to indicate the operation add x to S, and $y \rightarrow T(x)$ and $y \leftarrow T(x)$ are used for storing and retrieving the value associated with x, that is, $f(x)$, from the table T:

```
proc R(x) is
if x ∉ S
then if p(x)then c(x) → T(x)
     else R(ax); R(bx);
          y ← T(ax); z ← T(bx);
          θ(x, y, z) → T(x)
     fi;
     x → S
fi                                    (16)
```

Observe that R is recursively defined, so in addition to the space needed to maintain S and T, there is also the hidden space requirement of the stack which implements the recursion. Using well-established techniques of recursion elimination (see BIRD77a), one can transform R into an iterative stack algorithm which brings this extra space requirement to the surface, but we do not do so here.

Observe also that scheme (16) guarantees that only those function values actually needed for calculating $f(x)$ are computed and stored in the table. We call such schemes *exact tabulations*. On the other hand, there is no apparent way of recognizing the existence of function values which have outlived their usefulness and removing them from T. This aspect, together with the recursive nature of R, implies that a definite penalty in terms of storage is incurred when using exact tabulation.

The notation employed in (16) entails no special commitment as to the concrete representations of S and T, or even that they should necessarily be maintained as logically distinct structures. Different cases merit different treatments. For instance, the knapsack function $f(j, x)$ of Section 3 has the property that it is a step function for each fixed j, and hence can be represented by a sequential list of its corners (see HORO79 for a scheme based on this approach). Nevertheless one obvious representation of T, certainly applicable in the case of number-theoretic recursive definitions, is in terms of an array F of appropriate dimensions. For this to work it is convenient (but not essential; see below) to assume the existence of some special value ω, outside the range of values of f, to act as the undefined value. The table is initialized by setting $F[x] := \omega$ for all x in some appropriate range V of values. To do this it has to be assumed that V can be determined from the nature of the descent functions and the initial argument x_0. Ideally V should be just the set of arguments in the dependency graph associated with x_0; certainly it must be no smaller. Retrieval and storage are handled by the obvious assignments to array elements, and the test for membership in S is just the condition $F[x] \neq \omega$. Thus all table operations can be carried out in constant time.

It is not necessary to assume that function arguments are integers or tuples of integers for this simple representation to work. Provided there is some coding function h which maps function arguments into the set $\{1, 2, \ldots, |V|\}$ in a one-one manner, an array can still be used. Table operations code into the appropriate assignments, for example, $y \leftarrow T(x)$ codes as $y := F[(h(x)]$. If h has constant complexity, then the time spent on table operations,

apart from initialization, will increase the overall running time of the tabulation only by a proportional constant. For the sake of a formal statement to be made later about the cost of exact tabulation, we say that recursive functions suceptible to this form of representation by arrays can be tabulated with *constant overhead.*

There is a built-in disadvantage with the above treatment however. To illustrate the problem, imagine we have carried out an exact tabulation of the longest common subsequence function introduced in Section 3, using a two-dimensional array $F[1{:}n + 1, 1{:}m + 1]$. In general, exact tabulation will perform better than overtabulation; in fact if the two strings are identical, exact tabulation will take only $O(n)$ steps compared with $\theta(n^2)$ steps for overtabulation. However the array F still has to be initialized and if we execute the assignment $F[j, k] := 0$ for $1 \leq j, k \leq n + 1$, then we are still using $\theta(n^2)$ steps just to clear F. Of course this may not matter too much in practice as on many computers special instructions exist to clear areas of store quickly; nevertheless, initialization remains a theoretical difficulty. One solution is to maintain S as a separate data structure, say by a balanced tree scheme (see AHO75). If the exact tabulation computes n function values, then S will never be larger than n, and so the membership and insertion operations can both be carried out in $O(\log n)$ steps. The overall running time goes up by a factor of $\log n$, but such a solution may still be more efficient than one based on overtabulation, even though more machinery has to be invoked.

Remarkably, there is an alternative solution (see, for instance, AHO75), one which avoids array initialization altogether; however it uses more space. Suppose in addition to array F we maintain two further arrays, G and H. Array G has the same size and shape as F, while H is linear and acts like an ever-expanding list. An integer variable j, initially 0, is used as a pointer to H. Whenever a value $f(x)$ is to be stored in $F[x]$, the following assignments take place:

$$j := j + 1; \qquad H[j] := x;$$
$$F[x] := f(x); \qquad G[x] := j.$$

Using this setup we can now tell of a given entry $F[x]$ whether or not it contains a computed value or some totally random value left over from a previous program which utilized the same storage area. In fact $F[x]$ contains $f(x)$ just in the case that

$$1 \leq G[x] \leq j \quad \text{and} \quad H[G[x]] = x,$$

where j denotes the current value of the array pointer. If $G[x] = g$ lies in the required range, then we know $H[g]$ has been set by the program; the fact that $H[g] = x$ tells us that $F[x]$ has indeed been used to hold $f(x)$. If $1 \leq g \leq j$ but $H[g] \neq x$, then g must be some spurious value left in $G[x]$ from a previous calculation, and so $F[x]$ cannot have been initialized; the same conclusion must follow if g is outside the required range. Thus the test for $F[x]$ being well defined or not can be carried out in constant time, so there is only a constant increase in the overall running time of the tabulation.

Once we avoid the problem of initialization, it is no longer strictly necessary to predetermine the range V of function arguments used in the tabulation. Of course, unless some suitable range is known, we have to allow for an array of potentially infinite length, and this is clearly not possible in practice. The obvious alternatives are to implement T as a hash table or by a nonlinear structure such as a balanced tree. Such representations also serve, as we have already mentioned, in the case of functions whose arguments are not integers, but unfortunately one can then no longer always guarantee that the three fundamental operations of membership, storage, and retrieval can be carried out in constant time. When the array representation is possible, one can formulate a useful general rule for the timing of an exact tabulation scheme. We state it in terms of a single recursive definition, but it can be easily generalized to the case of a finite system of mutually recursive definitions.

Theorem

Suppose f is a recursively defined total function which can be tabulated with constant overhead. Suppose further that the time needed to compute $f(x)$ using the def-

inition is linear in the number of recursive calls. If the dependency graph of f and x possesses n nodes and e edges, then f(x) can always be tabulated in O(n + e) steps.

The theorem is justified by the fact that, since each table operation takes constant time, the time taken to compute each table entry $f(y)$ is linear in the number of directly dependent function values, that is, proportional to the number of edges emanating from node y in the dependency graph. Since each of the n table entries is computed just once and there are e edges altogether, the conclusion follows.

Of course, if the number of recursive calls in the definition of f is bounded, then the number of edges in the dependency graph is $O(n)$ and $f(x)$ can be computed in $O(n)$ steps. One way to ensure that the running time of f is linear in a bounded number of recursive calls is to restrict the defining body of f to be built up from finite conditional expressions involving base functions and predicates, each of which can be computed in constant time. One example of such a class of functions is McCarthy's number-theoretic formalism described in McCa63; the Fibonacci, knapsack, and common subsequence functions all come into this class.

The theorem relates the cost of tabulation to the size of the dependency graph, but this is not very useful if we have no idea of what this size is. Certainly if f only takes arguments from a finite domain of size n, then no dependency graph can have more than n nodes. Hence the following:

Corollary

Given the assumptions of the theorem about f and in addition that (a) the number of recursive calls is bounded and (b) f takes arguments only in a finite domain of size n, then f(x) can be tabulated in O(n) steps.

As we have seen in the case of the Fibonacci definition, there exist functions which require an exponential number of steps if they are to be evaluated directly from their definitions, and thus tabulation can reduce the time considerably.

SUMMARY

Essentially three different types of tabulation strategies have been described in the previous sections:

A. optimal time and space tabulations based on solutions to the pebble game played on the uniform class of dependency graphs of the recursive function for varying arguments;
B. overtabulations in which the pebble game is played not on the dependency graphs themselves but on a class of supergraphs which are more tractable for pebbling purposes;
C. exact tabulations where in effect each dependency graph is pebbled by a possibly nonoptimal depth-first strategy which never moves pebbles once they are placed on the graph's vertices.

A-type tabulations can be subtle and effective, but they do require the dependency graphs to have a certain degree of uniformity. B-type tabulations can overcome lack of uniformity by embedding the dependency graphs in larger but more uniform graphs; but once again this strategy is only possible if a suitable class of supergraphs is known and available. Finally, C-type tabulations are simple, powerful, and efficient as far as time is concerned, but they may use up quite a lot of space.

ACKNOWLEDGMENTS

The author would like to thank Bruce Weide and several anonymous referees for suggestions for improving the presentation of the paper.

REFERENCES

Aho72 Aho, A. V., and Ullman, J. D. *The theory of parsing, translation and compiling. vol. 1, Parsing*, Prentice-Hall, Englewood Cliffs, N.J., 1972.

Aho75 Aho, A. V., Hopcroft, J. E., and Ullman, J. D. *The design and analysis of computer algorithms*, Addison-Wesley, Reading, Mass., 1975.

Bell57 Bellman, R. E. *Dynamic programming*, Princeton University Press, Princeton, N.J., 1957.

Bird77a Bird, R. S. "Notes on recursion elimination," *Commun. ACM* **20**, 6 (June 1977), 434–439.

Bird77b Bird, R. S. "Improving programs by the

introduction of recursion," *Commun. ACM* **20**, 11 (Nov. 1977), 856–863.

BURS75 BURSTALL, R. M., AND DARLINGTON, J. "Some transformations for developing recursive programs," in *Proc. 1975 Internat. Conf. Reliable Software*, Los Angeles, 1975, pp. 465–472.

CHAN73 CHANDRA, A. K. "Efficient compilation of linear recursive programs," in *Conference Record, IEEE 14th Ann. Symp. Switching and Automata Theory*, 1973, pp. 16–25.

COHE79 COHEN, N. H. "Characterization and elimination of redundancy in recursive programs," in *6th Ann. Symp. Principles of Programming Languages*, 1979, pp. 143–157.

HIRS75 HIRSCHBERG, D. S. "A linear space algorithm for computing maximal common subsequences," *Commun. ACM* **18**, 6 (June 1975), 341–343.

HORO79 HOROWITZ, E., AND SAHNI, S. *Fundamentals of computer algorithms*, Fearon-Pitman, Belmont, Calif., 1979.

KNUT69 KNUTH, D. E. *The art of computer programming*, vol. 2, Addison-Wesley, Reading, Mass., 1969.

MCCA MCCARTHY, J. "On efficient ways of evaluating certain recursive functions," Project MAC A.I. Memo 32 (undated), M.I.T., Cambridge, Mass.

MCCA63 MCCARTHY, J. "A basis for a mathematical theory of computation," in *Computer programming and formal systems*, E. Braffort and D. Hirschberg (Eds.), North-Holland, Amsterdam, 1963, pp. 33–70.

MICH67 MICHIE, D. "Memo functions: A language feature with rote learning properties," DMIP Memo. MIP-R-29, Edinburgh, 1967.

PART76 PARTSCH, H., AND PEPPER, P. "A family of rules for recursion removal related to the Towers of Hanoi problem," *Inf. Process. Lett.* **5**, 6 (Dec. 1976), 174–177.

PATE70 PATERSON, M. S., AND HEWITT, C. E. "Comparative schematology," in *Record of Project MAC Conf. on Concurrent Systems and Parallel Computation*, 1970, pp. 119–127.

PAUL78 PAUL, W. J., AND TARJAN, R.-E. "Time-space trade-offs in a pebble game," *Acta Inf.* **10** (1978), 111–115.

PIPP78 PIPPENGER, N. "A time-space trade-off," *J. ACM* **25**, 4 (1978), 509–515.

REIN77 REINGOLD, E. M., NIEVERGELT, J., AND DEO, N. *Combinatorial algorithms: Theory and practice*, Prentice-Hall, Englewood Cliffs, N.J., 1977.

SHOR78 SHORTT, J. "An iterative program to calculate Fibonacci numbers in $O(\log n)$ arithmetic operations," *Inf. Process. Lett.* **7**, 6 (Oct. 1978), 299–303.

STEI77 STEINBRUGGEN, R. "Equivalent recursive definitions of certain number theoretical functions," TUM-INFO-7714, Technische Univ. München, West Germany, 1977.

VORO66 VOROBOYOV, N. N. *The Fibonacci numbers*, D. C. Heath, Boston, Mass., 1966.

WALK71 WALKER, S. A. "Some graph games related to the efficient calculation of expressions," IBM Res. Rep. RC-3628, 1971.

WATA80 WATANABE, O. "Another application of recursion introduction," *Inf. Process. Lett.* **10**, 3 (1980), 116–119.

WELL71 WELLS, M. B. *Elements of combinatorial computing*, Pergamon Press, Elmsford, N.Y., 1971.

WIRT76 WIRTH, N. *Algorithms + data structures = programs*, Prentice-Hall, Englewood Cliffs, N.J., 1976.

RECEIVED JANUARY 1980; FINAL REVISION ACCEPTED AUGUST 1980.