# SPACE-TIME TRADEOFFS FOR LINEAR RECURSION*

Sowmitri Swamy
Coordinated Science Laboratory and
Department of Electrical Engineering
University of Illinois
Urbana, Illinois 61801, USA

John E. Savage
Program in Computer Science and
Division of Engineering
Brown University
Providence, Rhode Island 02912, USA

## Abstract

A linear recursive procedure is one in which a procedural call can activate at most one other procedural call. When linear recursion cannot be replaced by iteration, it is usually implemented with a stack of size proportional to the depth of recursion. In this paper we analyze implementations of linear recursion which permit large reductions in storage space at the expense of a small increase in computation time. For example, if the depth of recursion is n, storage space can be reduced to $\sqrt{n}$ at the cost of a constant factor increase in running time. The problem is treated by abstracting linear recursion into the pebbling of a simple graph and for this abstraction we exhibit the optimal space-time tradeoffs.

## 1. Introduction

Many high-level languages permit the use of recursion and hence allow linear recursion in which a procedural call can activate at most one other procedural call. Linear recursion is usually implemented with a stack when a compiler either cannot or does not replace it with iteration. The size of such a stack will grow linearly with the depth of recursion and may in fact occupy much more storage space than the procedure itself. In this paper we investigate a general method to reduce the storage space required to implement linear recursion at the expense of computation time. Our approach to this tradeoff issue is one introduced by Paterson and Hewitt [1] and further analyzed by Chandra [2].

A schema for linear recursion is given below in which F is the procedure variable, p is a unary predicate, h and f are unary functions, g is a binary function and each is uninterpreted.

F(y): = <u>if</u> p(y) <u>then</u> h(y) <u>else</u> g(y, F(f(y)) ) <u>fi</u>

The semantics of such an expression are well understood; F calls itself until the predicate is TRUE and the call sequence is then reversed and F is computed from inside out. We present a simple (linear recursion) graph model of this sequence of steps and show that execution of linear recursion with limited storage space can be abstracted as a "pebble game" on this graph. The pebble game models register (or memory) allocation, and the placement of a pebble on a node indicates that the value of the function at that node has been computed and placed in a register (or memory location).

We find the optimal space-time tradeoff for pebblings of the linear recursion graph and state these results in an easily understood form. For example, if the depth of recursion is n and p pebbles are used, the number of moves required, $T_p(n)$, has the following behavior for large n:

$$T_p(n) = \begin{cases} p\, n^{1+\frac{1}{p}}(p/1+p) & p \ll \log_2 n \\ K_1\, n \log_2 n & p = K_2 \log_2 n \\ n\,\dfrac{\log_2 n}{\log_2 p} & p \gg \log_2 n \end{cases}$$

Here $K_1$ is a function of $K_2$. Thus, if p is on the order of $n^{1/k}$, $T_p(n)$ remains linear in n which implies that a large decrease in space can be achieved at the expense of a small increase in running time. We also exhibit the class of algorithms which achieve these optimal space-time exchanges and show that at least one of these algorithms is easily implemented.

In the next section, we develop the graph model for linear recursion and in the following section, Section 3, we derive the optimal algorithms and the statement of space-time exchange. In Section 4 we present a program implementation of "partial stack algorithms" for linear recursion. Section 5 is concerned with two implementation issues, one of which is the determination of the amount of storage space that needs to be allocated to achieve a user or compiler space-time exchange.

Paterson and Hewitt [1] introduced the pebble game and demonstrated for linear recursion that for fixed temporary storage, time grows as $n^{1+1/k}$. Chandra [2] demonstrated that the time must grow at least this rapidly under these conditions even if counters are also available. He also gave some nonoptimal algorithms for linear recursion which use small, medium, and large amounts of space. Our contribution is to simplify the analysis, determine the full class of optimal algorithms, derive in simple terms the space-time tradeoffs and show that they can be achieved with some simple algorithms. We also observe that the problem studied by

*This work was supported in part by the National Science Foundation under Grant MCS 76-20023.

Paterson, Hewitt, Chandra and us is really that of replacing stacks by partial stacks in which missing stack elements are recomputed. For this reason, the techniques of this paper are also applicable to some forms of recursion which are not strictly linear.

The pebble game has been investigated by several authors. Hopcroft, Paul, and Valiant [3] have studied the pebble game on a graph on N nodes and show that they can be pebbled using space $O(N/\log N)$, where the constant of proportionality depends on the maximum in-degree of nodes. Paul, Tarjan and Celoni [4] have shown the existence of arbitrarily large graphs for which every pebbling strategy uses space $\Omega(N/\log N)$, and Pippenger [5] has exhibited a graph on N nodes for which the time T and space S satisfy

$$\frac{2T}{N} = \log_2 \frac{N}{2S} + O(1)$$

so that more than linear time (in N) is necessary to pebble the graph with space $O(N/\log N)$. Paul and Tarjan [6] have shown the existence of graphs on N nodes such that reduction in S by a constant factor causes T to expand from $O(N)$ to $2^{O(\sqrt{N})}$. Savage and Swamy [7] have derived tight upper and lower bounds to the space-time exchanges for the FFT algorithm on n inputs which show that the space-time product is about $O(n^2)$.

These results are derived for pebblings of specific graphs. However, Grigoryev [8] has derived lower bounds to the product ST for $n \times n$ matrix multiplication modulo-2 and for multiplication of n-degree polynomials which are $\Omega(n^3)$ and $\Omega(n^2)$, respectively, and which apply to any straight-line algorithms for these problems. Tompa [9] has similar results for superconcentrators, which include bilinear algorithms for convolution, and matrix multiplication for special matrices, for grates [10] and for the discrete Fourier Transform.

## 2. A Graph Model for Linear Recursion

Given a linear recursive procedure F and an input a, as defined above, F calls itself n times where n, the depth of recursion, is the smallest integer such that $p(f^{(n)}(a))$ is True. (Here $f^{(0)}(y) = y$ and for $n \geq 1$, $f^{(n)}(y) = f(f^{(n-1)}(y))$. It follows that

$$F(f^{(n)}(a)) = h(f^{(n)}(a))$$

and in general for $0 \leq r < n$

$$F(f^{(r)}(a)) = g(f^{(r)}(a)\ ,\ F(f^{(r+1)}(a)))$$

In a stack implementation, the depth of recursion is determined by successively calling F until the predicate p is TRUE and pushing $f^{(r)}(a)$ onto the stack for $0 \leq r \leq n-1$.

When f is an invertible function, we can compute $f^{(r)}(a)$ from $f^{(r+1)}(a)$ by

$$f^{(r)}(a) = f^{-1}(f^{(r+1)}(a))$$

so that linear recursion can be replaced by iteration, as indicated schematically below.

```
y : = a
while p(y) ≠ TRUE do y : = f(y) od
z : = h(y)
while y ≠ a do y : = f^{-1}(y); z : = g(y,z) od
F(a) : = z
```

Thus, if f is invertible, as in the following example, then linear recursion can be realized in a fixed amount of space.

Fac(n): = if n = 0 then 1 else n × Fac(n-1)fi

However, f may not be invertible, difficult to invert or it may not be clear to a compiler (or compiler writer) that it is invertible. We note that every partial recursive function can be realized by ALGOL-like programs which use linear recursion and elementary functions.

We consider methods for exchanging space for time that do not depend upon the specific interpretations given to p, h, g, and f. Thus, these methods will compute functions defined by linear recursive procedures by simulating, perhaps with repetition, the computations carried out in a stack implementation. Figure 1 shows a very simple directed acyclic graph $L_n$ (called a chain) which is the basis for describing the simulation of linear recursion. Node r, $1 \leq r \leq n$, corresponds to $f^{(r)}(a)$ by application of the function f, as indicated by the directed edge from node r to node r+1. In linear recursion the object is to compute the items represented by the nodes in reverse order, namely in the order $f^{(n)}(a)$, $f^{(n-1)}(a)$, $f^{(n-2)}(a)$, ..., $f^{(2)}(a)$, $f^{(1)}(a)$. In a stack implementation these items are stored in increasing order as they are computed, so they can be retrieved directly. However, if too much space is used by a stack, a partial stack can be retained in which intermediate stack results are saved. This requires the recomputation of results that have been discarded. We call such algorithms as "partial stack algorithms."

The space-time tradeoff problem is now abstracted as a pebble game on the chain $L_n$ of Figure 1. It is assumed that the depth of recursion n is known. (This is easily determined in fixed space, as indicated above for the case of an invertible function f.) In the pebble game, pebbles are placed and removed on the nodes of the graph according to certain rules and when a node is pebbled this indicates that the function associated with that node has been computed and the result has been placed in a register (or memory location). Any input node can be pebbled at any time and a non-input node can be pebbled only when all nodes which have edges directed into that node have been pebbled. Pebbles can be removed at any time. We count only the moves made to place pebbles on nodes.

## 3. An Optimal Pebbling Strategy

The problem of computing a function defined by a linear recursive procedure with depth of recursion n has been reduced to the pebbling of the nodes of the graph $L_n$ of Figure 1 in the order n, n-1, n-2, ..., 2, 1 while honoring the dependencies indicated by the directed edges. If p pebbles are allowed, we let $T_p(n)$ denote the number of times pebbles are placed on nodes to complete this task. (Pebble removals are not counted.) We note that if node r has a pebble on it when it is to be visited, the visitation is not counted in $T_p(n)$. The number of steps to visit as well as pebble is no more than $T_p(n) + n-p$.

Consider the case in which one pebble is used to pebble $L_n$. This pebble must be placed successively on nodes $1,2,\ldots,n$ to pebble node n. Once this is done, the problem is reduced to pebbling the nodes of $L_{n-1}$ in reverse order. Thus,

$$T_1(n) = n + T_1(n-1)$$

and since $T_1(1) = 1$, this reduces to

$$T_1(n) = n(n+1)/2.$$

The case in which two pebbles are used is more interesting and indicates how the general case should be approached. As two pebbles advance into $L_n$ there will be several points in time at which one pebble is on a node i and another on node j where $j > i+1$, that is, there is a gap between them. If the pebble on node i is removed before it could be used to make another advance into the graph, then the pebbling is non optimal because over some portion of the graph the algorithm will have made use of only one of the two pebbles. This same argument applies to two adjacent pebbles on $L_n$ when there are more than two pebbles, namely, in an optimal algorithm the pebble on the node with lower index is not removed <u>after a gap develops between them</u> until it is used to make a subsequent advance.

Consider a time-optimal pebbling of $L_n$ with p pebbles, $p \leq n$. There will be a point in time at which p pebbles are on $L_n$ and such that the pebble on the node of lowest index, say r, will be held in place while the remaining p-1 pebbles are used to pebble the subgraph of nodes $r+1,r+2,\ldots,n$ in reverse order. Since the pebble on node r is used for this advance, the problem is equivalent to pebbling $L_{n-r}$ with p-1 pebbles. After nodes $n, n-1, \ldots, r+1$ and r are pebbled, the p pebbles are used to pebble nodes $1,2,\ldots,r-1$ in reverse order. Since r is chosen optimally, we have

$$T_p(n) = \min_{1 \leq r \leq n-p+1} (r+T_p(r-1)+T_{p-1}(n-r)) \quad (1)$$

since r moves are necessary to bring a pebble to node r. We call this node a splitting node because it divides the pebbling problem into two subproblems.

To solve this recurrence we introduce a binomial number system. Given a positive integer $p \geq 2$ (to be interpreted later as the number of pebbles), for each positive integer N there are unique non-negative integers m and $\ell$ such that

$$N = S_{p-1,m} + \ell,\ 0 \leq \ell \leq S_{p-2,m+1} - 1$$

where

$$S_{q,m} = \binom{m+q}{q+1}$$

The uniqueness of these integers follows from the monotonicity of $S_{p-1,m}$ with m and the following identity

$$S_{q,m+1} = S_{q,m} + S_{q-1,m+1} \quad (2)$$

The number system can be extended to the case p=1, which is important below, if we set $S_{-1,m} = 1$. Then when p=1 we have $\ell$=0 and m=N. Also $S_{p-1,2} = p+1$ so if $p \geq N$ we have m=1 and $\ell$=N-1.

<u>Theorem 1</u>: For all $p \geq 1$, the minimum number of placements of pebbles required to pebble the chain $L_n$ of $n \geq 1$ nodes with at most p pebbles, $T_p(n)$, satisfies

$$T_p(n) = \frac{p}{p+1} (m-1)S_{p-1,m} + m(\ell+1) \quad (3)$$

where $1 \leq m$ and $0 \leq \ell \leq S_{p-2,m+1} - 1$ are the unique integers such that

$$n = S_{p-1,m} + \ell \quad (4)$$

<u>Proof</u>: The proof is by induction on n and p.

<u>Basis</u>: a) The case of p=1, namely $T_1(n) = n(n+1)/2 = m(m+1)/2$ has been established above which agrees with (3).

b) For $p \geq n$, $L_n$ can be completely pebbled in n moves so $T_p(n) = n$. Also, m=1 and $\ell$=n-1 in this case, which agrees with (3).

The basis states expressions for $T_p(n)$ on those boundaries which are shown in Figure 2.

<u>Inductive Hypothesis</u>: If $T_p(n)$ is given by (3) for all $1 \leq n \leq p$ when $p \leq P-1$ and for $1 \leq n \leq N-1$ when p=P, then $T_p(N)$ is also given by (3).

Figure 2 also shows the order in which the induction sequence is carried out. We now state the conditions under which the minimum of equation (1) is achieved.

Let

$$G(r) = r + T_p(r-1) + T_{p-1}(n-r) \quad (5)$$

Then,

$$T_p(n) = \min_{1 \leq r \leq n-p+1} G(r) \quad (6)$$

Consider the forward difference

$$\nabla G(r) = G(r+1) - G(r)$$
$$= 1 + \nabla T_p(r-1) - \nabla T_{p-1}(n-r-1) \quad (7)$$

Then, the minimum in (6) is achieved at a value of r such that $\nabla G(r) \geq 0$. Therefore, we further evaluate $\nabla G(r)$.

Since $1 \leq r \leq n-p+1$ and $p \geq 2$ we invoke the inductive hypothesis and use (3) to evaluate the forward differences in (7). To do this we let (u,h) where $u \geq 0$, $0 \leq h \leq S_{p-2,u+1}-1$ and (v,i) where $v \geq 0$, $0 \leq i \leq S_{p-3,v+1}-1$, be the unique pairs of integers such that

$$r = S_{p-1,u} + h$$
$$n-r = S_{p-2,v} + i \quad (8)$$

Clearly, from (4) we have

$$S_{p-1,m} + \ell = S_{p-1,u} + S_{p-2,v} + h+i \quad (9)$$

which will be used later.

The forward difference $\nabla T_p(r-1)$ is easily seen to be u when $h \geq 1$ (and $r-1 \geq S_{p-1,u}$) and can also be shown equal to u when h = 0 by straightforward manipulation of binomial coefficients. Similarly, $\nabla T_{p-1}(n-r-1)$ is equal to v. Summarizing we have

$$\nabla T_p(r-1) = u, \quad 0 \le h \le S_{p-2,u+1}-1$$

$$\nabla T_{p-1}(n-r-1) = v, \quad 0 \le i \le S_{p-3,v+1}-1$$

We will also encounter the forward differences $\nabla T_p(r-2)$ and $\nabla T_{p-1}(n-r)$ and we have

$$\nabla T_p(r-2) = \begin{cases} u & h > 0 \\ u-1 & h = 0 \end{cases}$$

$$\nabla T_{p-1}(n-r) = \begin{cases} v+1 & i = S_{p-3,v+1}-1 \\ v & i < S_{p-3,V+1}-1 \end{cases}$$

as a direct consequence of the above analysis.

From these observations and (7) we have that

$$\nabla G(r) = 1 + u - v \tag{10}$$

and since

$$\nabla G(r-1) = 1 + T_p(r-2) - T_{p-1}(n-r)$$

we have

$$\nabla G(r-1) = \begin{cases} u-v & h > 0, \quad i = S_{p-3,v+1}-1 & \text{(11a)} \\ 1+u-v & h > 0, \quad i < S_{p-3,v+1}-1 & \text{(11b)} \\ u-v-1 & h = 0, \quad i = S_{p-3,v+1}-1 & \text{(11c)} \\ u-v & h = 0, \quad i < S_{p-3,v+1}-1 & \text{(11d)} \end{cases}$$

As indicated above, the set of integers $\{r\}$ which minimize (6), the optimal splitting nodes, satisfy $\nabla G(r) \ge 0$ although not all such integers minimize this expression. We consider two classes of integers which minimize (6) and satisfy $\nabla G(r) \ge 0$, namely

$$A = \{r, r+1 | \nabla G(r) = 0\}$$

$$B = \{r | \nabla G(r) \ge 1, \ \nabla G(r-1) \le -1\}$$

The integers for which $\nabla G(r) \ge 1$ and $\nabla G(r-1) \ge 1$ do not minimize (6) while those for which $\nabla G(r) \ge 1$ and $\nabla G(r-1) = 0$ fall into A.

Consider $r \in A$ such that $\nabla G(r) = 0$; then from (10)

$$\nabla G(r) = 1 + u - v = 0$$

or v = u + 1. From (9) and the identity (2) we have

$$S_{p-1,m} + \ell = S_{p-1,u+1} + (h+i)$$

and since

$$0 \le h+i \le S_{p-2,u+1}-1 + S_{p-3,v+1}-1 = S_{p-2,u+2}-2$$

we conclude that

$$u = m-1, \ v = m, \ 0 \le h+i = \ell \le S_{p-2,m+1}-2 \tag{12}$$

Consider next the case of $r \in B$; here we have $u-v \ge 0$ and $\nabla G(r-1) \le -1$ and the only case for which both conditions hold is (11c). This requires

$$u = v, \ h = 0, \ i = S_{p-3,v+1}-1$$

and from (2) and (9) we have

$$S_{p-1,m} + \ell = S_{p-1,u} + S_{p-2,u} + S_{p-3,u+1}-1$$

$$= S_{p-1,u} + S_{p-2,u+1}-1$$

from which it follows that

$$u = m, \ v = m, \ \ell = S_{p-2,m+1}-1, \ h = 0, \ i = S_{p-3,m+1}-1 \tag{13}$$

Thus, if $\ell = S_{p-2,m+1}-1$ there is exactly one value for r that minimizes G(r), namely, $r = S_{p-1,m}$, while if $0 \le \ell \le S_{p-2,m+1}-2$ then the minimizing value of r satisfies $S_{p-1,m-1} \le r \le S_{p-1,m}-1$.

It remains to use the minimizing values for r in (5) to show that the minimum is indeed $T_p(n)$. This task is left to the reader. □

We extract some additional information from this theorem that will facilitate the construction of a partial stack algorithm for linear recursion.

Corollary. If $r_0$ is a splitting node of $L_n$ then it satisfies the following conditions:

Case a) For $S_{p-1,m} \le n \le S_{p-1,m+1}-2$

$$S_{p-1,m-1} \le r_0 \le S_{p-1,m}-1,$$

$$S_{p-2,m} \le n-r_0 \le S_{p-2,m+1}-1$$

Case b) If $n = S_{p-1,m+1}-1$

then $r_0 = S_{p-1,m}$, and $n-r_0 = S_{p-2,m+1}-1$

We now identify a single, simply computed integer $r_1$ which is the label of a splitting node of $L_n$.

Lemma 1. The integer $r_1$ defined by

$$r_1 = \max(S_{p-1,m-1}, \ n-S_{p-2,m+1}+1) \tag{14}$$

satisfies the conditions of the above corollary.

Proof. The conditions of the corollary can be stated as bounds on $r_0$, when $S_{p-1,m} \le n \le S_{p-1,m+1}-2$, as shown below.

$$S_{p-1,m-1} \le r_0 \le S_{p-1,m}-1,$$

$$n-S_{p-2,m+1} + 1 \le r_0 \le n-S_{p-2,m}$$

It is easy to demonstrate that $r_1$, the larger of the two lower bounds satisfies both upper bounds. When $n = S_{p-1,m+1}-1$, $n-S_{p-2,m+1}+1 = S_{p-1,m}$ so that $r_1 = S_{p-1,m}$, which is the optimizing value of $r_0$ in this case. □

We now examine a number of implementation issues.

## 4. Partial Stack Algorithms

The space-time exchange that will be obtained from a partial stack algorithm will be determined by the number of temporary stack locations p available and the depth of recursion n. We postpone until another section a discussion of this exchange and present here a program for the realization of an optimal partial stack algorithm. Our program will use the rule given in Lemma 1 for the selection of a splitting node.

The recurrence of equation (1) defines a partial stack algorithm when $2 \leq p \leq n-1$. It consists of pebbling up to a splitting node $r_1$ with a pebble being left on this node followed by a pebbling of nodes $r_1+1, r_1+2, \ldots, n$ in reverse order with p-1 pebbles and then a pebbling of nodes $1,2,\ldots,r_1-1$ with p pebbles. If $p = 1$, the single pebble strategy is used to pebble nodes $1,2,\ldots,n$ in reverse order. If $p \geq n$, a standard stack algorithm is optimal.

The partial stack algorithm that uses $r_1$ as a splitting node where

$$r_1 = \max(S_{p-1,m-1},\ n-S_{p-2,m+1}+1) \qquad (14)$$

is given below in pseudo-ALGOL. The determination of p, the size of the partial stack and the calculation of integers m and $\ell$ are discussed in the next section. Observe that once the depth of recursion n is known $f^{(n)}(a)$ and $h(f^{(n)}(a))$ are available so the problem reduces to computing $f^{(1)}(a), f^{(2)}(a), \ldots, f^{(n-1)}(a)$ in reverse order or to the pebbling of $L_{n-1}$. Comments are shown in brackets.

```
[Determine the depth of recursion, n, and com-
 pute h(f^(n)(a)).]
   y: = a; n: = 0
   while NOT(p(y)) do y: = f(y); n: = n+1 od
   F: = h(y); d: = n-1
[Determine p; compute m, and $S_{p-1,m}$ such that
 $S_{p-1,m} \leq d \leq S_{p-1,m+1} - 1$.]
   s:=$S_{p-1,m}$
[Observe that $S_{p-1,m+1} = (m+p)s/m$.]
[Use the single pebble strategy if p = 1, a full
 stack if p ≥ d and the partial stack algorithm
 otherwise.]
     if p = 1 then call SPL(d, a, F)
              else if p≥d then call STK(d, a, F)
              else call PSTK(d,a,m,p,s,F) fi fi
[The following procedure computes F with one
 pebble.]
   Procedure SPL(d,a,F)
   begin
      i: = d; z: = a
      while i ≠ 0 do j: = i; while j ≠ 0 do
                        z: = f(z); j: = j-1 od;
                        i: = i-1; F: = G(z,F) od
      F: = G(a,F)
   end
[The following procedure computes F with a
 complete stack.]
     Procedure STK(d,a,F)
     begin
        i: = d-1; z: = a
        while i ≠ 0 do z: = f(z); PUSH(z);
                                     i: = i-1 od
        z: = f(z); i: = d-1; F: = G(z,F)
        while i ≠ 0 do z: = POP; F: = G(z,F);
                                    i: = i-1 od
        F: = G(a,F)
     end
     Procedure PSTK(d,a,m,s,p,F)
[$s \leq d \leq (m+p)s/m-1$]
     begin
        z: = a
[If p=1, the single pebble algorithm is optimal.]
     if p = 1 then call SPL(d,a,F); return fi
[If m=1, then $S_{p-1,1}=1$ and $S_{p-1,2}-1=p$ so
 $1 \leq d \leq p$ and the full stack algorithm is used.]
   if m=1 then call STK(d,a,F); return fi
[The splitting node $r=\max(S_{p-1,m-1}, d-S_{p-2,m+1}+1)$
 $\geq S_{p-1,m-1} \geq 1$ when $m \geq 2$. If $r=S_{p-1,m-1}$ then
 $S_{p-1,m-2} \leq r-1 \leq S_{p-1,m-1}-1$ and if $r > S_{p-1,m-1}$
 then $S_{p-1,m-1} \leq r-1 \leq S_{p-1,m}-1$. In either case
 $S_{p-2,m} \leq d-r \leq S_{p-2,m+1}-1$. We use the identi-
 ties $S_{p-1,m-1} = s(m-1)/(m+p-1)$, $S_{p-2,m+1}$ =
 $sp/m$, $S_{p-2,m} = sp/(m+p-1)$ and $S_{p-1,m-2}$ =
 $((m-2)/(m+p-2))S_{p-1,m-1}$]
   sℓ:=s(m-1)/(m+p-1); su:=sp/(m+p-1); mℓ:=m-1;
   pu:=p-1; u:=d-s(p/m)+1;
   if sℓ ≥ u then r:=sℓ; mℓ:=mℓ-1; sℓ:=sℓ(mℓ)/(mℓ+p)
           else r:=u   fi
    i:=r;dℓ :=r-1; du:=d-r;
[dℓ ≥ 0, du ≥ 1]
[Pebble up to the splitting node.]
   while i ≠ 0 do z:=f(z); i:=i-1 od
[Nodes r+1, r+2,...,n are pebbled followed
 by nodes 1,2,..., r+1 unless r = 1.]
   call PSTK(du, z, m, su, pu, F);
   if dℓ = 0 then f:=G(a,F) else call
   PSTK(dℓ, a, mℓ, sℓ, p, F) fi end
```

The number of times that this algorithm computes a function $f^{(r)}(a)$ for some $1 \leq r \leq n$ is $n + T_p(n-1)$ because the first loop that computes n computes each of these functions once and the remaining portion of the program pebbles the graph corresponding to the functions $f^{(1)}(a)$, $f^{(2)}(a), \ldots,$ $f^{(n-1)}(a)$, since $f^{(n)}(a)$ and $h(f^{(n)}(a))$ are computed by the first loop. For each computation of $f^{(r)}(a)$ for some $1 \leq r \leq n-1$, there is a fixed upper bound on the number of additional functions that are computed, assignments that are made and tests that are performed.

In the next section we examine the dependence of $T_p(n)$ on p and n. We also examine ways to compute m, $\ell$ and $S_{p-1,m}$ from n. This information can be used to compute p based upon user or compiler defined criteria.

## 5. Implementation Issues

The number of moves required to pebble $L_n$ with p pebbles, $T_p(n)$, is given in Theorem 1 and is seen to be a linear function of n for values of n between $S_{p-1,m'}$ for $m' = 1,2,3,\ldots$ . At $n = S_{p-1,m}$ we have

$$T_p(n) = \frac{p}{p+1}(m-1)n \qquad (15)$$

When $p = 2$, $n = (m+1)m/2$, $m \cong \sqrt{2n}$ and

$$T_p(n) \cong (2\sqrt{2}/3)n^{3/2}$$

for large n. Also, when $m = 3$, $n = (p+2)(p+1)/2$, $p \cong \sqrt{2n}$ and

$$T_p(n) \cong 2n$$

for large n. If $m = 4$ the coefficient of n is increased to 3 and p is reduced to about $3\sqrt{3n}$. These results demonstrate the strong reduction in space that can be achieved at the expense of a small increase in running time.

From (15) it is clear that the size of m largely determines the gap between n and $T_p(n)$ since $p/(p+1)$ lies between 2/3 and 1 when $p \geq 2$. Thus, if m is to be bounded from above this will determine p if n is known. On the other hand, we may wish to bound p at the expense of m or perhaps attempt to achieve a balance between p and m. Thus, we explore the relationships between m, p and n when $n = S_{p-1,m}$.

The function

$$n = \binom{m+p-1}{p} \qquad (16)$$

is symmetric in p and m-1 and monotone increasing in both. Thus, it is easy to show from the following inequalities [10, p. 530] that the smaller of p and (m-1) is no larger than $2 \log_2 n$ when $n \geq 4$. (Here, $N = m+p-1$)

$$\frac{1}{\sqrt{2N}} \leq \sqrt{\frac{N}{8k(N-k)}} \leq \binom{N}{k} 2^{-NH(\frac{k}{N})} \leq \sqrt{\frac{N}{2\pi k(N-k)}} \leq 1 \qquad (17)$$

Here $1 \leq k \leq N-1$, $N \geq 2$ and H(x) is the entropy function.

$$H(x) = - x \log_2 x - (1-x)\log_2(1-x) \qquad (18)$$

Since n is one term in the binomial expansion of $(1+1)^{m+p-1}$, we have

$$n \leq 2^{m+p-1} \qquad (19)$$

or that the sum $m+p-1$ is at least $\log_2 n$. Furthermore, from (17) if m and p are comparable in size and n is large, it follows that they are both comparable to $\log_2 n$. Thus, we consider three cases when n is large

$m \ll p \quad \Rightarrow \quad m \ll \log_2 n$ and

$p \ll m \quad \Rightarrow \quad p \ll \log_2 n$

p comparable to m $\Rightarrow \quad p/(m+p-1) = \lambda, \; 0 < \lambda < 1.$

and examine $T_p(n)$.

We have

$$n = \frac{(m+p-1)(m+p-2)\cdots(m)}{p!} \geq \left(\frac{m}{p}\right)^p$$

which is also a good approximation when $p \ll m$. This implies

$$m \leq pn^{1/p}$$

and by the symmetry of n in (m-1) and p we have

$$p+1 \leq (m-1)n^{1/(m-1)}$$

which is a good approximation when $m \ll p$. Reworking this equation we have

$$(m-1) \leq \frac{\log_2 n}{\log_2 p - \log_2 (m-1)} \cong \frac{\log_2 n}{\log_2 p}$$

and the approximation holds when $m \ll p$. Since $m \ll p$ if $p \gg \log_2 n$ (note that $m+p-1 \geq \log_2 n$), we have

$$T_p(n) \cong \begin{cases} \frac{p^2 n^{1+1/p}}{p+1} & p \ll \log_2 n \\ \frac{n \log_2 n}{\log_2 p} & p \gg \log_2 n \end{cases}$$

In the remaining case, when p is proportional to $\log_2 n$, we use (17) to approximate n. If

$$\lambda = \frac{p}{m+p-1} \quad \text{or} \quad m-1 = \frac{(1-\lambda)}{\lambda} p$$

for $0 < \lambda < 1$ and if n is large, then taking logarithms we have

$$\log_2 n \cong (m+p-1)\, H(\lambda) = p\, \frac{H(\lambda)}{\lambda}$$

which implies that p is proportional to $\log_2 n$. Then,

$$T_p(n) \cong \frac{1-\lambda}{H(\lambda)}\, n \log_2 n$$

when

$$p \cong \frac{\lambda}{H(\lambda)} \log_2 n$$

The expressions $\lambda/H(\lambda)$ and $(1-\lambda)/H(\lambda)$ are shown in Figure 3.

Summarizing, we find that $T_p(n)$ grows as $pn^{1+1/p}$ for p small, $n\log n/\log p$ for $p \gg \log_2 n$ and as $n\log n$ if $\lambda$ is neither near zero ($p \ll m$) nor near 1 ($m \ll p$), that is, for p proportional to $\log_2 n$. The three different rates of growth of $T_p(n)$ with n can be selected by choosing p as a function of n which grows more slowly than $\log_2 n$, such as $\sqrt{\log_2 n}$ or a constant, more rapidly than $\log_2 n$, such as $(\log_2 n)^2$ or $n^{1/5}$, or which grows in proportion to $\log_2 n$, respectively. The actual value of p may also be determined by an upper limit on temporary storage space.

Once p is chosen, the next step is to determine m and $S_{p-1,m}$ such that

$$S_{p-1,m} \leq n < S_{p-1,m+1} \qquad (20)$$

By a previous argument, the smaller of p and m-1 is no larger than $2 \log_2 n$ for $n \geq 4$, hence $S_{p-1,m}$ can be computed in at most $8 \log_2 n$ multiplications or divisions from one of the following two expressions:

$$S_{p-1,m} = \frac{(m+p-1)(m+p-2)\cdots(m)}{p!} =$$

$$= \frac{(m+p-1)(m+p-2)\cdots(p+1)}{m!}$$

In fact, many fewer multiplications may suffice if either p or m are very small. To compute m, start m at n (note that $S_{o,m} = m$) and use binary search by halving m until (20) is satisfied. This will take $O(\log_2 n)$ steps so the entire process can be done in $O(\log^2 n)$ steps.

## 6. Conclusions

Linear recursion has been modeled as a pebble game on a directed graph which is a simple chain of n nodes, n the depth of recursion. We have exhibited the class of optimal "partial-stack" algorithms for this pebbling problem and have derived simple explicit expressions for the time-space tradeoff for linear recursion. We have also given a simple program for implementing an optimal partial stack algorithm and we have studied the asymptotic behavior of the time-space tradeoff function for the purpose of providing rules for selection of the amount of space that should be used to achieve various degrees of performance.

While linear recursion occurs frequently in practice, other forms of recursion do also. For example, if H is a procedure defined by linear recursion and F is the procedure defined below then F is not linear.

F(y): = if p(y) then h(y) else G(H(y), F(f(y)))

However, the partial stack algorithm described above can be used to realize H and F if individual stacks are used for each. Also the same could be said of other forms of recursion. It is not known, however, whether this approach will yield optimal or near optimal programs.

## References

1. Paterson, M. S. and C. E. Hewitt, "Comparative Schematology," Proj. MAC Conf. on Concurrent Systems and Parallel Computation, Woods Hole, MA, pp. 119-127, June 2-5, 1970.
2. Chandra, A. K., "Efficient Compilation of Linear Recursive Programs," IBM Research Rept. RC4517, 10 pp. August 29, 1973 and the Proceedings of the Fourteenth Annual Symposium on Switching and Automata Theory, pp. 16-25, October 15-17, 1973.
3. Hopcroft, J. E., W. J. Paul, and L. G. Valiant, "On Time Versus Space," JACM, Vol. 24, No. 2, pp. 332-337, 1977.
4. Paul, W. J., R. E. Tarjan, and J. R. Celoni, "Space Bounds for a Game on Graphs," Eighth Ann. Symp. on Theory of Computing, Hershey, PA, pp. 149-160, May 3-5, 1976.
5. Pippenger, N., "A Time-Space Tradeoff," IBM preprint, May 1977, to appear in JACM.
6. Paul, W. J. and R. E. Tarjan, "Time-Space Tradeoffs in a Pebble Game," preprint, May 1977 and Fourth Colloquium on Automata, Languages and Programming, Turku, Finland, 1977.
7. Savage, J. E. and S. Swamy, "Space-Time Tradeoffs on the FFT Algorithm," to appear in the Sept. 1978 issue of the IEEE Transactions on Information Theory.
8. Grigoryev, D. Yu, "An Application of Separability and Independence Notions for Proving Lower Bounds of Circuit Complexity," Notes of Scientific Seminars, Steklov Math. Inst., Leningrad Branch, Vol. 60, pp. 38-48, 1976.
9. Tompa, M., "Time-Space Tradeoffs for Computing Functions, Using Connectivity Properties of Their Circuits," Proceedings of the Tenth Annual ACM Symposium on Theory of Computing, May 1-3, 1978.
10. Gallager, R. G., Information Theory and Reliable Communicstions, John Wiley and Sons, New York, 1968.
11. A. J. Guttman, "Programming Recursively Defined Functions in FORTRAN" Intl. Journal of Computer and Info. Sciences, Vol. 5, No. 2, 1976.

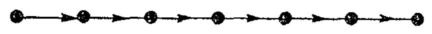

FIG. 1 THE CHAIN $L_n$ FOR $n = 7$

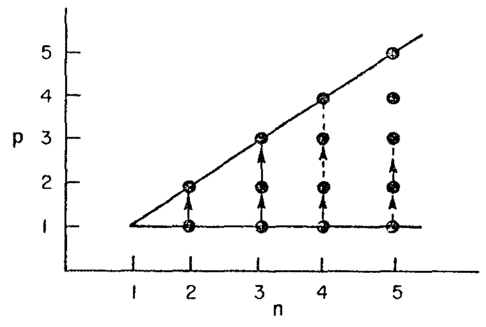

FIG. 2 BOUNDARIES AND INDUCTION SEQUENCE FOR THEOREM 1

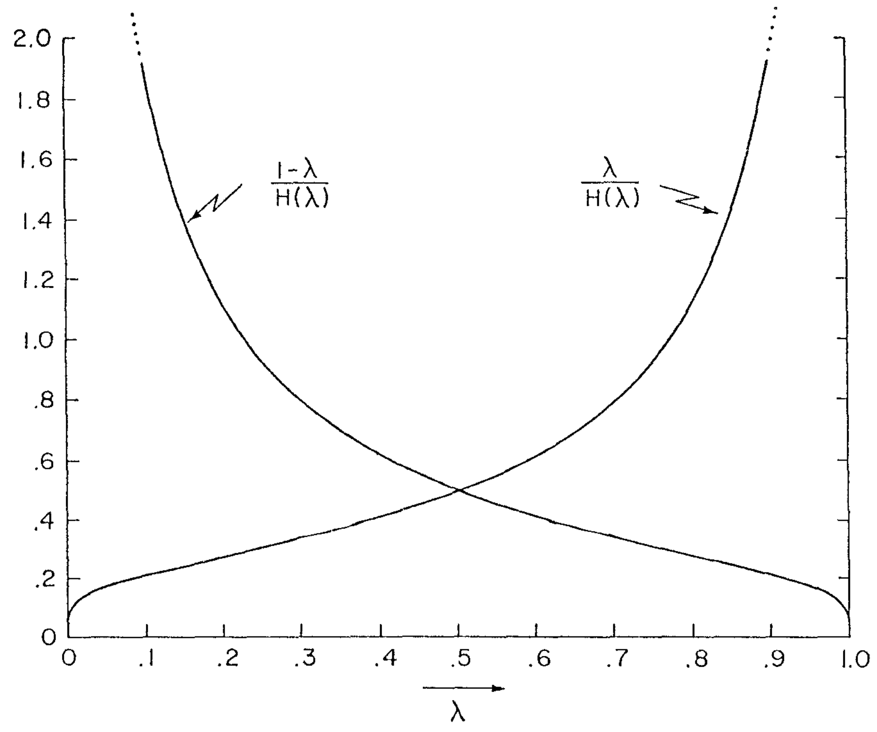

FIG. 3 THE FUNCTIONS $\lambda/H(\lambda)$ AND $(1-\lambda)/H(\lambda)$