STANFORD ARTIFICAL INTELLIGENCE PROJECT
MEMO AIM-167

STAN-CS-72-282

EFFICIENT COMPILATION OF LINEAR RECURSIVE PROGRAMS

BY

ASHOK K. CHANDRA

SUPPORTED BY
ADVANCED RESEARCH PROJECTS AGENCY
ARPA ORDER NO. 457

APRIL 1972

COMPUTER SCIENCE DEPARTMENT
School of Humanities and Sciences
STANFORD UNIVERSITY

STANFORD ARTIFICIAL INTELLIGENCE PROJECT
MEMO AIM-167

JUNE 1972

COMPUTER SCIENCE DEPARTMENT
REPORT CS-282

EFFICIENT COMPILATION OF LINEAR RECURSIVE PROGRAMS

by

Ashok K. Chandra

ABSTRACT: We consider the class of linear recursive programs. A linear recursive program is a set of procedures where each procedure can make at most one recursive call. The conventional stack implementation of recursion requires time and space both proportional to n, the depth of recursion. It is shown that in order to implement linear recursion so as to execute in time n one doesn't need space proportional to n: n^{ϵ} for arbitrarily small ϵ will do. It is also known that with constant space one can implement linear recursion in time n^2. We show that one can do much better: $n^{1+\epsilon}$ for arbitrarily small ϵ. We also describe an algorithm that lies between these two: it takes time $n.\log(n)$ and space $\log(n)$.

It is shown that several problems are closely related to the linear recursion problem, for example, the problem of reversing an input tape given a finite automaton with several one-way heads. By casting all these problems into a canonical form, efficient solutions are obtained simultaneously for all.

This research was supported by the Advanced Research Projects Agency, Department of Defense (SD-183), U.S.A.

TABLE OF CONTENTS

1. Introduction

We consider the following two problems.

(1) The linear recursion problem

A linear recursive program is a set of ALGOL-like procedures each of which contains at most one procedure call. All parameters are passed by value. There is one specified procedure that is evaluated with certain given inputs. The problem is to compile a given linear recursive program into an efficient program without recursion. The reader may wonder at this point what is wrong with the conventional stack implementation of recursion since that represents about as fast as one can hope to go. The problem is that it takes a great deal of space. In this context efficiency refers to space efficiency. On the other hand, most compiler writers are aware of an implementation that requires space for just one or two values but takes a great deal of time - proportional to the square of the recursion depth. On comparison with this algorithm efficiency refers to time efficiency.

For a treatment of some problems related to the linear recursion problem see Chandra [1972].

(2) The schema problem

A schema is a program in which the base functions and predicates are left uninterpreted. A schema along with a given interpretation characterizes a computation. The following recursive definition specifies a schema:

Compute $F(a)$ where

$F(y) \leftarrow \underline{\text{if}}\ p(y)\ \underline{\text{then}}\ h(y)\ \underline{\text{else}}\ g(y,F(f(y)))$.

This schema has been considered in some form or the other by several authors e.g., Paterson and Hewitt [1970], Hewitt [1970], Garland and Luckham [1971], Strong and Walker [1972]. The base functions f,g and h, the individual constant a, and the predicate p are not interpreted. In this schema there is some implicit storage allocation, i.e., the value of y is stored while F(f(y)) is computed, and the two are then used to obtain g(y,F(f(y))). The problem is to translate (or compile) this recursive schema into an efficient flowchart schema that does its storage allocation explicitly.

We solve the linear recursion problem by first converting it to the schema problem and then solving that. In some sense the schema problem looks like a simplified version of the linear recursion problem where there is just one procedure which consists of a single <u>if</u>-<u>then</u>-<u>else</u> statement. However, the use of a schema as an intermediate step in the solution has some other bonuses stemming from the fact that the base functions and predicates of the schema are uninterpreted. Hence, by specifying appropriate interpretations one immediately obtains solutions for several different problems that can be modelled by the schema (as shown in Appendix I).

Section 2 defines the schema problem and the linear recursion problem in somewhat greater detail and shows how the latter can be reduced to the former. The reader may safely omit this section, though it may be desirable to read subsections 2.1 and the first part of 2.2, which define the schema problem and the linear recursion problem. Section 3 presents the main results of this paper. Efficient solutions for the schema problem are given, and space-time tradeoffs are considered. Section 4 demonstrates some practical aspects of these results. In Appendix I (Section 5) we mention two other problems that can be reduced to the schema problem. Most of the detailed algorithms are not inserted in the mainstream of the paper to allow for ease in reading. These are given in Appendix II (Section 6).

2. Translation of the Linear Recursion Problem to the Schema Problem

2.1 A Definition of the Schema Problem

A flowchart schema has a finite number of variables. We use the symbols $y_1, y_2, y_3, \ldots$ to represent variables. The schema we consider has a zero-ary function a (i.e., an individual constant), a unary predicate p, unary functions f and h and the binary function g. Statements in the schema are of the following types:

Start statement:	START
Halt statement:	HALT(y)
Assignment statement:	$y \leftarrow a$
	$y_i \leftarrow f(y_j)$
	$y_i \leftarrow g(y_j, y_k)$
	$y_i \leftarrow h(y_j)$
Predicate test:	<u>if</u> p(y) <u>then</u> <u>goto</u> L_1 <u>else</u> <u>goto</u> L_2

where L_1 and L_2 are arbitrary labels. Any statement may be labelled, and unconditional goto statements are allowed. While the schema is called a "flowchart schema" for the reason that it can be represented as a flowchart, we use the more compact and convenient ALGOL-like notation. We also allow the use of block structures and "while-statements" with the understanding that these features may be translated by using goto statements to get a "legal" schema.

A flowchart schema with arrays has the following additional features:

$c_1, c_2, c_3, \ldots$	counters that can have nonnegative integer values
$A_1, A_2, A_3, \ldots$	one-dimensional, semi-infinite arrays that may be subscripted with counters.

For convenience we will frequently use identifiers other than those given above for counters, variables and arrays. For this reason the artifice of declarations can be used to clarify the meaning of identifiers, where necessary. The type <u>data</u> is used to identify variables, <u>counter</u> for counters, and <u>array</u> for arrays. The following operations on counters and arrays are allowed:

Counter operations:	$c \leftarrow c + 1$
	$c \leftarrow c - 1$
	$c_i \leftarrow c_j$
	<u>if</u> $c = 0$ <u>then</u> <u>goto</u> L_1 <u>else</u> <u>goto</u> L_2
Array operations:	$y \leftarrow A[c]$
	$A[c] \leftarrow y$

The execution of each statement is assumed to take a unit amount of time. The space required is the number of variables, counters and the number of array locations from zero to the maximum ever referenced.

We may consider the use of additional statements that may be assumed to take unit amounts of time each. An example is the test of equality between two variables. This case is briefly considered by Chandra and Manna [1972]. Other examples include the comparison of the values of two counters, halving and doubling the values of counters, multiplying two counters etc. While these operations are certainly useful, their direct application in the algorithms to be described results only in a factor in time efficiency (though equality tests can also detect looping).

The given recursive schema is

Compute $F(a)$ where

$F(y) \leftarrow$ <u>if</u> $p(y)$ <u>then</u> $h(y)$ <u>else</u> $g(y,F(f(y)))$.

The problem is to translate this into an efficient equivalent flowchart schema (with and without arrays).

For any interpretation, if there is an integer n such that:

$$p(f^n(a)) \quad \text{is true,}$$

and $\forall k < n \quad p(f^k(a)) \quad$ is false,

then the term to be computed is

$$g(a,g(f(a),g(f(f(a)), \ldots g(f^{n-1}(a),h(f^n(a))) \ldots))).$$

If, for any interpretation, no such n exists then the schema diverges for that interpretation. The time and space bounds will be considered in terms of n.

2.2 <u>Reduction of the linear recursion problem to the schema problem</u>

A linear recursive program is a set of ALGOL-like procedures having the following features:

(a) There are no global variables. This imposes no constraints because we allow procedures to return a vector of arguments.

(b) Each procedure is loop-free. Loops can be eliminated by using recursive calls instead (McCarthy [1962]). However, standard techniques exist for implementing iteration; and we do not wish to complicate matters by considering this case as well.

(c) Each procedure can call at most one other procedure. This is the crucial reason why the program is called "linear".

Now, given a linear recursive program we will first "erase" the definitions of the base functions and predicates to get a linear recursive schema. The reason for this is that we are not interested in the detailed implementation of the basic operations like addition, multiplication, cons (in LISP), test for zero, etc.

A linear recursive schema is a set of non-looping procedures. The first statement of each procedure is the start statement. Subsequent statements may be assignment, test, or return statements. As mentioned earlier, we will use ALGOL-like notation to represent the flow of control. The variables used in the procedure body may be formal parameters (called by value) or local variables. No global variables are allowed. The "type" of a variable may be <u>data</u> or <u>boolean</u> . Data variables may take values from some domain D1 (to be specified with the interpretation), whereas boolean variables take values from the domain B = {true,false}. Further, procedures may return a vector of values. This is important because side effects are not allowed. The same effect could be achieved (very inefficientl by calling several procedures in sequence each of which returns a single value but even this mechanism is not available to us since linear recursive schemas can have at most one procedure call in every path from the start statement to a return statement. It is assumed that the given linear recursive schema S_1 has no illegal reference, i.e., no variable is referenced unless it has been assigned a value. This assumption is not strictly required for we can augment the domains D_1 and B by adding one "undefined element" to each. This represents only minor modifications of the discussion below.

The given linear recursive schema S_1 is to be reduced to the standard form

S: Compute F(a) where

$$F(y) \leftarrow \underline{\text{if}}\ p(y)\ \underline{\text{then}}\ h(y)\ \underline{\text{else}}\ g(y,F(f(y))).$$

The reduction to the standard form is effected in two steps:

1. If the given recursive schema S_1 has d distinct procedure definitions, combine them into one procedure by adding $\log_2(d)$ boolean

variables which are tested at the start of the new procedure. Also, the vector of arguments and the vector of values returned are padded with arbitrary constants so that they both have the same type, say $D_1^s \times B^t$; i.e., the vector may be represented as $\langle y_1, y_2, \ldots y_s, z_1, z_2, \ldots z_t \rangle$ where all the y's are of type data and all the z's are of type boolean. The single procedure obtained in this manner will be called F_2, and the schema (the procedure F_2 along with the initial values) will be called S_2.

2. Now, given the schema S_2 and the interpretation I_1 for its functions and predicates (over the domain D_1) it is our objective to produce an interpretation I (over some domain D) for the constant a, the functions f,g,h and the predicate p such that the schema in standard form effectively computes the same value as S_2. It is a requirement that the time taken to compute the new base functions a,f,g,h and the predicate p be dependent only on the schema S_2 and not on the number of recursive calls required for its computation under I_1 (with the usual assumption that the base functions and predicates in I_1 take unit time to compute). If this requirement is satisfied then the assumption (in the schema problem) that each basic statement takes a unit (or bounded) amount of time, is justified.

The domain D is $D_1^S \times B^t$, i.e., each element in the new domain is a vector of data and boolean values.

The zero-ary function a is defined to be that element of D which is the vector corresponding to the arguments initially supplied to the procedure F_2 in the schema S_2.

The predicate p is defined as follows: for any vector $v \in D$, $v = \langle y_1, \ldots, y_s, z_1, \ldots, z_t \rangle$, if $F_2(y_1, \ldots, y_s, z_1, \ldots, z_t)$ executes a recursive call then p(v) is false, else p(v) is true.

The function h is defined as follows: for any vector $v \in D$, $v = \langle y_1, \ldots, z_t \rangle$, if $p(v)$ is false then $h(v)$ is arbitrary, but if $p(v)$ is true (i.e., there is no recursive call) then $h(v)$ is the vector returned on execution of $F_2(y_1, \ldots, z_t)$.

The function f is defined as follows: for any vector $v \in D$, $v = \langle y_1, \ldots, z_t \rangle$, if $p(v)$ is true then $f(v)$ is arbitrary, else it is the vector argument of the recursive call to F_2.

The function g is defined as follows: for any $v_1, v_2 \in D$, $v_1 = \langle y_1, \ldots, z_t \rangle$, if $p(v_1)$ is true then $g(v_1, v_2)$ is arbitrary, else it is the value that would be returned by $F_2(y_1, \ldots, z_t)$ if v_2 were substituted for the value of the one recursive call executed.

For somewhat greater detail the reader is urged to examine the example presented below.

The interesting thing about the translation presented above is that it is reversible. Given I and the schema S, or a schema equivalent to it, one can substitute the values of the constant a, the functions f,g,h and the predicate p to obtain a schema equivalent to the original schema S_1. In this manner, once we have an efficient, non-recursive flowchart program equivalent to S, on substitution we obtain an efficient program equivalent to S_1.

It should be noted, however, that the word "efficient" as used above denotes only the order of space-time dependence on n - the number of recursive calls. Blind substitution for the constant a, the functions f,g,h and the predicate p would result in some redundant computations, for example. in a computation of $f(v)$ immediately following $p(v)$ (see the example below). A practical compiler based on this method would find it relatively simple, however, to avoid this duplicated effort.

2.3 An Example

The given linear recursive schema S_1 has the following base functions:

a_1	a zero-ary function (an individual constant),
f_1	a binary function,
f_2	a ternary function, and
p_1, p_2	unary predicates.

The schema S_1 is:

```
            S_1:       Compute y_1 where <y_1,y_2> ← F_1b(a_1);
F_1a(y_1,y_2) ← START  data y_3; boolean z_1;
                y_1 ← f_1(y_1,y_2);
                if p_1(y_1)
                then begin <y_3,z_1> ← F_1b(y_2);
                           if z_1
                           then RETURN(y_3)
                           else RETURN(f_2(y_1,y_2,y_3));
                     end
                else if p_2(y_2)
                     then RETURN(y_2)
                     else begin y_3 ← F_1a(y_1,y_2);
                                RETURN(y_3);
                          end  .

F_1b(y_1) ←     START  data y_2, boolean z_1;
                if p_2(y_1)
                then begin <y_2,z_1> ← F_1b(y_1);
                           RETURN(y_2,true);
                     end
                else begin y_2 ← F_1a(y_1,y_1);
                           RETURN(y_2,false);
                     end  .
```

Step 1: a boolean variable z_2 is added. z_2 = true signifies a call to F_{1a}, z_2 = false signifies a call to F_{1b}. The boolean variable z_1 (below) plays the role of z_1 in the above definitions of F_{1a} and F_{1b}. A "redundant" data variable y_4 is added to match the padded data element in the vector returned. Both the argument vector and the return vector have type $D_1 \times D_1 \times B$. The resulting schema S_2 is:

S_2: Compute y_1 where $<y_1,y_2,z_1> \leftarrow F_2(a_1,a_1,\text{false})$:

```
F_2(y_1,y_2,z_2) ←    START data y_3,y_4; boolean z_1;
                      if z_2
(1) - - -             then begin y_1 ← f_1(y_1,y_2);
                                 if p_1(y_1)
(2) - - -                        then begin <y_3,y_4,z_1> ← F_2(y_2,a_1,false);
                                            if z_1
(3) - - -                                   then RETURN(y_3,a_1,true)
                                            else RETURN(f_2(y_1,y_2,y_3),a_1,true);
                                      end
                                 else if p_2(y_2)
                                      then RETURN(y_2,a_1,true)
(4) - - -                             else begin <y_3,y_4,z_1> ← F_2(y_1,y_2,true);
                                                 RETURN(y_3,a_1,true);
                                           end;
(5) - - -                  end
(6) - - -             else begin if p_2(y_1)
                                 then begin <y_2,y_4,z_1> ← F_2(y_1,a_1,false);
(7) - - -                                   RETURN(y_2,a_1,true);
                                      end
```

```
                    else begin <$y_2$,$y_4$,$z_1$> ← $F_2$($y_1$,$y_1$,true);
                               RETURN($y_2$,$a_1$,false);
                         end
(8) - - -      end.
```

Lines (1) - (5) effectively define F_{1a}, and lines (6) - (8) define F_{1b}. Line (2) invokes a call to F_{1b}. Since F_{1b} takes only one argument, the value a_1 is padded. Line (4) is a call to F_{1a}. F_{1a} really returns just one value. So return statements such as in line (3) are padded with two elements: a_1 and true. F_{1b} returns 2 values, so only one value needs to be padded when F_{1b} returns: e.g. a_1 in line (7) is the padded element.

Step 2: the standard schema is:

S: Compute F(a) where

```
    F(y) ← if p(y) then h(y) else g(y,F(f(y))).
```

The required interpretation I for S follows. The definitions of p,h and g below can be simplified. We choose not to do so for the sake of clarity.

$D = D_1 \times D_1 \times B$

$a = <a_1,a_1,\text{false}>$

```
p(<$y_1$,$y_2$,$z_2$>) = if $z_2$
                    then if $p_1$($f_1$($y_1$,$y_2$))
                         then false
                         else if $p_2$($y_2$)
                              then true
                              else false
                    else if $p_2$($y_1$)
                         then false
                         else false
```

```
$h(<y_1,y_2,z_2>)$ = if $z_2$
          then if $p_1(f_1(y_1,y_2))$
               then arbitrary
               else if $p_2(y_2)$
                    then $<y_2,a_1,true>$
                    else arbitrary
          else if $p_2(y_1)$
                    then arbitrary
                    else arbitrary
$f(<y_1,y_2,z_2>)$ = if $z_2$
          then if $p_1(f_1(y_1,y_2))$
               then $<y_2,a_1,false>$
               else if $p_2(y_2)$
                    then arbitrary
                    else $<f_1(y_1,y_2),y_2,true>$
          else if $p_2(y_1)$
               then $<y_1,a_1,false>$
               else $<y_1,y_1,true>$
$g(<y_1,y_2,z_2>, <w_1,w_2,x_1>)$ =
          if $z_2$
          then if $p_1(f_1(y_1,y_2))$
               then if $x_1$
                    then $<w_1,a_1,true>$
                    else $<f_2(f_1(y_1,y_2),y_2,w_1),a_1,true)$
               else if $p_2(y_2)$
                    then arbitrary
                    else $<w_1,a_1,true>$
else if $p_2(y_1)$
    then $<w_1, a_1$, true$>$
    else $<w_1, a_1$, false$>$.
```

3. Solutions for the Schema Problem

3.1 Introduction

In the previous sections it was shown that the linear recursion problem can be reduced to the schema problem. Appendix I presents two other problems that can be converted to the schema problem. Algorithms for this problem are now described, and the other problems can be solved by substituting the appropriate interpretations for the functions and the predicate of the schema.

The schema is:

Computer $F(a)$ where

$F(y) \leftarrow$ *if* $p(y)$ *then* $h(y)$ *else* $g(y,F(f(y)))$.

Let n be the depth of recursion i.e., n is the smallest integer for which $p(f^n(a))$ is true. Then the computation may be represented by the following:

Define	$t(n)$ to be $h(f^n(a))$.
$\forall i<n$ define	$t(i)$ to be $g(f^i(a),t(i+1))$.

Then the desired output is $t(0)$ - assuming, of course, that n exists, for if n does not exist then the schema loops forever. Essentially this rule is used in all algorithms below. They differ only in the way they compute $f^i(a)$.

The standard implementation uses a stack:

```
START
counter c; array A;
c ← 0;
x ← a;
```

```
(1) - - -  while ¬ p(x) do
             begin
               A[c] ← x;
               x ← f(x);
               c ← c + 1;
(2) - - -     end;
           y ← h(x);
(3) - - -  while c ≠ 0 do
             begin
               c ← c - 1;
               y ← g(A[c],y);
(4) - - -    end;
          HALT(y).
```

The array A acts as push-down stack. The first loop (lines (1) - (2)) implements recursive calls, and the second loop (lines (3) - (4)) pops the stack to compute the final value.

This implementation takes time and space both proportional to n - the number of recursive calls.

It is also well known that the recursion can be implemented using only a constant amount of memory and time proportional to n^2 (see for example Garland and Luckham [1971]). One program that avoids even the use of counters is presented below. In this program the variables w_1 and w_2 effectively play the role of counters. Counters are implemented by letting the term $f^i(a)$ $(i \leq n)$ represent the value $n - i$. A counter, e.g., w_1, can be set to its maximum value n by an assignment statement $w_1 \leftarrow a$, the counter can be decremented by $w_1 \leftarrow f(w_1)$, and it can be tested for zero by $p(w_1)$.

```
START
$x \leftarrow a$;
while $\neg\, p(x)$ do $x \leftarrow f(x)$;
$y \leftarrow h(x)$;
$w_1 \leftarrow a$;
while $\neg\, p(w_1)$ do
  begin
    $w_1 \leftarrow f(w_1)$;
    $x \leftarrow a$;
    $w_2 \leftarrow w_1$;
    while $\neg\, p(w_2)$ do
      begin
        $x \leftarrow f(x)$;
        $w_2 \leftarrow f(w_2)$;
      end;
     $y \leftarrow g(x,y)$;
end;
HALT(y).
```

It is shown in this paper that with a constant amount of memory the time can be brought down to $n^{1+\epsilon}$ for any arbitrarily small positive ϵ. This answers in the negative, a conjecture due to Hewitt [1970] that n^2 is the best one can hope to do. Further, to solve the problem in time proportional to n one does not need space proportional to n as in the stack implementa on. just n^{ϵ}. It is also satisfying that there exists an algorithm whose space-time tradeoff falls "midway" between these two algorithms; it takes space proportional to log(n) and time proportional to n.log(n).

Time	Space	
n	n	stack implementation
n	n^{ϵ}	linear-time algorithm (sec. 3.2)
$n \log n$	$\log n$	log(n) algorithm (sec. 3.3)
$n^{1+\epsilon}$	1	constant-space algorithm (sec. 3.4)
n^2	1	conventional constant space program

3.2 Linear-time algorithm

The main problem encountered in flowcharting the given recursive schema is that of inverting the function f i.e., given a value $f^{i+1}(a)$, to find $f^{i}(a)$. This cannot be done directly because the function f is not invertible. The stack implementation solves this problem by storing all the values

$$a,\ f(a),\ f^{2}(a),\ \ldots,\ f^{n}(a),$$

and picking them off in reverse order. The constant-space algorithm above, on the other hand, doesn't really save any values as such, but rather computes $a, f(a), f^{2}(a), \ldots, f^{i-1}(a), f^{i}(a)$. A control mechanism is used to keep the count.

Between these two extremes of saving all values, and saving none, there exist schemes for saving an intermediate number of carefully spaced values. We first informally describe an algorithm that is linear in n but requires space proportional to $n^{1/2}$ (order-2 algorithm). We then present a generalization of this algorithm which takes space $n^{1/k}$ (order-k algorithm). Details are given in sec. 6.1 of Appendix II.

Order-2 algorithm

We use the notation v(i) to stand for the term $f^i(a)$. Thus v(0) stands for the constant a itself.

Let n be a perfect square and let m denote $\sqrt{n}$. In the initialization phase the following values are calculated and saved:

v(0), v(m), v(2m),...,v((m-2)*m),
v((m-1)*m), v((m-1)*m+1),...,v((m-1)*m+m-1).

Now the first m values (right to left) can be picked off and used in the computation of the final output. Each step takes a constant amount of time. The following values are left:

v(0), v(m), v(2m),...,v((m-2)*m).

A redistribution can now be performed to compute and save the values

v((m-2)*m+1), v((m-2)*m+2),..., v((m-2)*m+m-1).

Another set of m values can now be picked off before a second distribution is required; and so on.

The following are the contributions to the computation time:

1. The first initialization phase - takes time proportional to n.

2. Picking off values - n steps, taking constant time each - total proportional to n.

3. Redistribution - $\sqrt{n}$ steps, each taking time proportional to $\sqrt{n}$ - total proportional to n.

Thus the overall algorithm is linear, and takes space $2 * \sqrt{n}$. We call the above an order-2 linear-time algorithm. It can be generalized to an order-k linear-time algorithm as follows.

Order-k algorithm

Let $n > 0$ be a power of k, and let m denote $n^{1/k}$.

```
(1) The initialization phase: compute and save the values
    $v(0)$, $v(n-(m-1).m^{k-1})$, $v(n-(m-2).m^{k-1})$, ..., $v(n-2m^{k-1})$,

    $v(n-m.m^{k-2})$, $v(n-(m-1).m^{k-2})$, ..., $v(n-2m^{k-2})$,

    $v(n-m)$, $v(n-(m-1))$, ..., $v(n-2)$, $v(n-1)$,
    Set counters $c_1 \leftarrow c_2 \leftarrow \dots \leftarrow c_{k-1} \leftarrow m-1$.
    Set $y \leftarrow h(v(n))$.

(2) The main computation phase: "pick off" the last m values, i.e.,
    for m steps do $y \leftarrow g(x,y)$, where x takes on the saved values
    from right to left.

(3) Redistribution phase:
    (3.1) Level 1 redistribution:
          If $c_1 = 0$ then goto step 3.2.
          Redistribute m-1 new values by steps of 1 using the latest
          saved value.
          $c_1 \leftarrow c_1 - 1$.
          Goto step 2.
    (3.2) Level 2 redistribution:
          If $c_2 = 0$ then goto step 3.3.
          Redistribute m-1 new values by steps of m using the
          latest saved value.
          $c_2 \leftarrow c_2 - 1$.
          $c_1 \leftarrow m$.
          Goto step 3.1.
```

```
(3.3)  Level 3 redistribution:
       If $c_3 = 0$ then goto step 3.4.
       Redistribute m-1 new values by steps of $m^2$ using the latest
       saved value.
       $c_3 \leftarrow c_3 - 1$
       $c_2 \leftarrow m$.
       Goto step 3.2.
            .
            .
            .
(3.k-1)  Level k-1 redistribution
       If $c_{k-1} = 0$ then goto step 4.
       Redistribute m-1 new values by steps of $m^{k-2}$ using the latest
       saved value.
       $c_{k-1} \leftarrow c_{k-1} - 1$.
       $c_{k-2} \leftarrow m$.
       Goto step 3.k-2.
(4)    HALT(y)
```

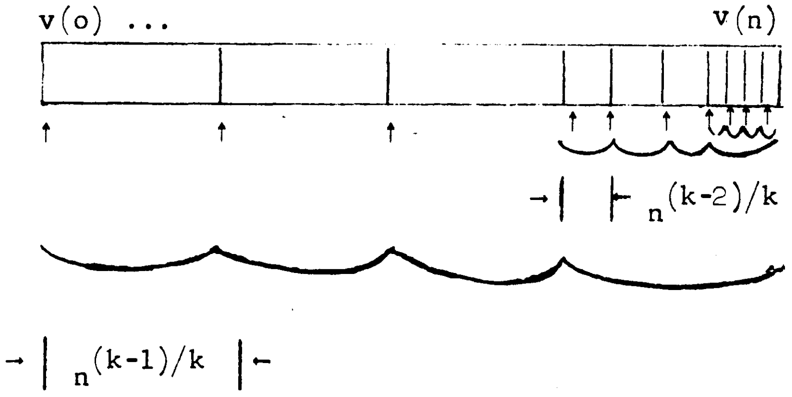
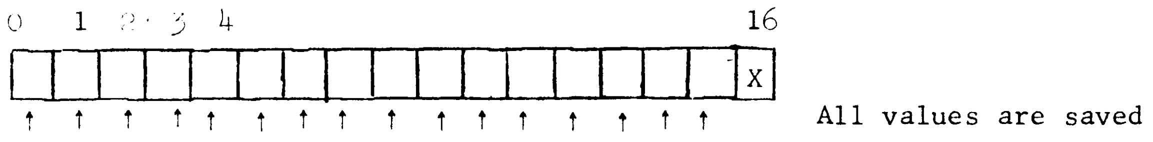


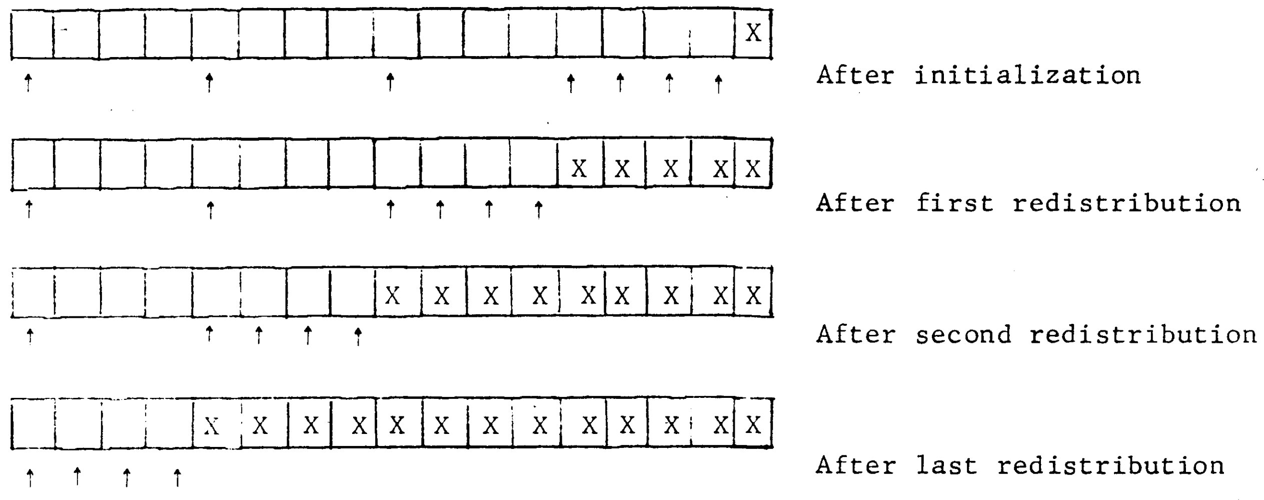
Figure 1

Figure 1 shows the values saved on the initialization phase. Figure 2 shows the computations in somewhat greater detail for the order-1, the order-2, and the order-4 algorithms with n = 16. In Figure 2, adjacent squares represent values $a, f(a), \ldots, f^{16}(a)$. Squares marked X represent values that have already been used in building up the final output. Up-arrows ↑ denote values that are saved at that stage of the algorithm. Note that the order-1 algorithm is precisely the conventional stack implementation of recursion.



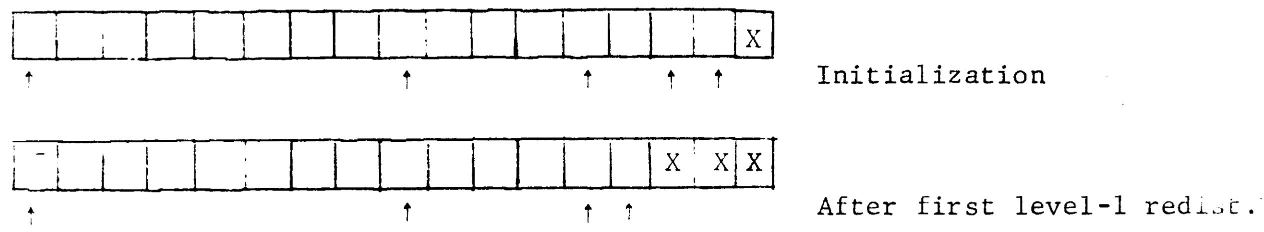
Order-1 linear-time algorithm

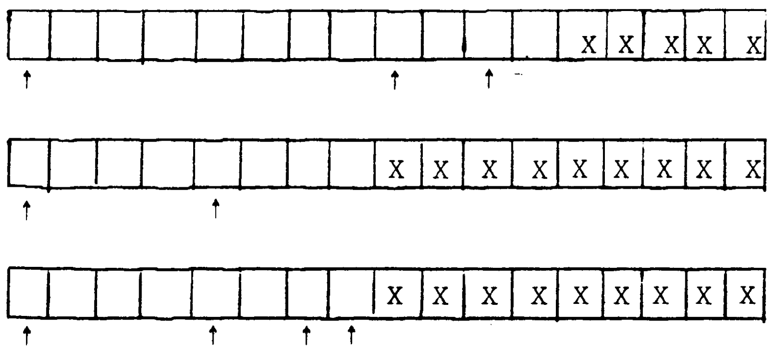
Figure 2a


X
After initialization
X X X X X
After first redistribution
X X X X X X X X X
After second redistribution
X X X X X X X X X X X X X
After last redistribution

Order-2 linear-time algorithm

Figure 2b




X
After first level-2 redist.
After first level-3 redist.
After subsequent level-2 and level-1 redistributions.

Order-4 linear-time algorithm

Figure 2c

Figure 2

The discussion above is a simplified version ignoring the important case when n is not the k-th power of any integer. The total algorithm is described in Appendix II. The main differences from the simplified algorithm are:

(1) the initialization phase is somewhat more complex,

(2) level-k redistributions too are called for.

Nevertheless, the space-time considerations below remain valid.

Space requirements

The maximum number of saved values in the simplified algorithm is $(k-1).n^{1/k}+1$. For the general algorithm the number of saved values is at most

$$(k-1).\ \lfloor n^{1/k} \rfloor + \frac{n}{\lfloor n^{1/k} \rfloor^{k}}$$

A small amount of extra storage is required for counters (proportional to k) and some variables for manipulating values (constant number).

Thus, asymptotically, the data-space requirement is proportional to $k.n^{1/k}$.

Time requirements

The main components contributing to the running time are the following:

1. The initialization phase: proportional to n.
2. The computation phase: n steps, each taking constant time - total proportional to n.
3. Level 1 redistribution: $n^{1-1/k}$ steps, each taking time $n^{1/k}$ - total proportional to n.
4. Level 2 redistribution: $n^{1-2/k}$ steps, each taking time $n^{2/k}$ - total proportional to n.

.

.

.

k+1. Level k-1 redistribution: $n^{1/k}$ steps, each taking time $n^{1-1/k}$ total proportional to n.

Thus the total running time is proportional to k.n.

In summary

There exist linear-time algorithms that take space significantly less than the conventional stack algorithm - merely $n^{1/k}$ where k is the order of the algorithm. It is interesting to note that the constant of proportionality increases linearly with k. The running time too increases linearly with k. It is for this reason (i.e. the constants of proportionality do not increase too rapidly with k) that high-order algorithms can be appealing in practice.

3.3 The log(n) algorithm

Before we go on to describe a class of constant-space algorithms we mention an algorithm lying between the linear-time and the constant-space algorithms. It can be approximately described in terms of the class of linear-time algorithms as follows: if n is the depth of recursion and $p = \log_2(n)$ then execute the order-p linear-time algorithm.

The idea behind the method is the following. We convert the given linear recursive program into a "nonlinear" recursive program which is then implemented by the usual stack method. The result is significant savings in space at the expense of extra computation. The algorithm is given below. As before we use the notation $v(i)$ to represent $f^i(a)$.

```
Main program:
(1)  Compute $n$, $v(n)$.
(2)  Set $y \leftarrow h(v(n))$.
(3)  $y \leftarrow G(v(0), n, y)$.
(4)  HALT(y).
Recursive procedure $G(x,i,y)$:
(local counter j),
(1)  If $i = 0$ then RETURN(y).
(2)  If $i = 1$ then RETURN(g(x,y)).
(3)  Set $j \leftarrow i/2$   (integer division).
(4)  $y \leftarrow G(f^j(x), i-j, y)$.
(5)  $y \leftarrow G(x, j, y)$.
(6)  RETURN(y).
```

The procedure G works by dividing the given "interval" into two parts, and calling itself recursively on the second half, and then on the first half of the interval. The algorithm takes space proportional to log(n) and time proportional to n.log(n).

3.4 Constant-space algorithm

We would like to implement constant-space algorithms using only a finite control, i.e. we do not wish to use arrays or counters. It has been shown (sec. 3.1) however, that bounded counters in the range 0-n can be implemented without the use of an explicit counter. We will thus allow ourselves the liberty of using bounded counters. We have to be a little careful because incrementing a counter is no longer a unit operation, but makes time proportional to n. Decrementing a counter and testing for zero, however, remain unit operations.

As before, we will first informally describe the order-2 algorithm and then generalize to the higher order case, leaving the details for sec. 6.2 of Appendix II.

Order-2 algorithm

Let n be a perfect square and let m denote $\sqrt{n}$. In the initialization phase n,m are evaluated and two values are saved - v(0) and v(n-m). The latter is now used as the base for computing v(n-1), v(n-2),..., v(n-m) in that order. The advantage obtained by using v(n-m) as the base as against v(0) is that the average computation time for each term is only $\sqrt{n}$ instead of n. Now after the m values have been evaluated, it is time to reset the base to v(n-2m). This, of course, takes time n, but then these resets have to be done quite infrequently - $\sqrt{n}$ times. The main contributors to the computation time are (1) the initialization which takes time $n^{3/2}$, (2) the actual computation: n steps taking $n^{1/2}$ average - total $n^{3/2}$, and (3) resets of the base value - $n^{1/2}$ steps averaging n - total $n^{3/2}$. Thus the total computation time is just $n^{3/2}$ when 2 values are saved.

Order-k algorithm

Generalizing this to an order-k constant-space algorithm let $n > 0$ be a power of k, and let m denote $n^{1/k}$.

```
(1) The initialization phase: compute and save the values
    $x_{k-1} \leftarrow v(0)$,
    $x_{k-2} \leftarrow v(n-m^{k-1})$,
    . . .
    $x_0 \leftarrow v(n-m)$.
    Set counters $c_1 \leftarrow c_2 \leftarrow \dots \leftarrow c_{k-1} \leftarrow m-1$,
    Set $y \leftarrow h(v(n))$.
(2) The main computation phase: "pick off" the m values to the
    right of $x_0$ (including $x_0$ itself) using $x_0$ as the base,
    and apply to y.
(3) The reset phase:
    (3.1) Level 1 reset:
          If $c_1$ = then goto step 3.2.
          Reset $x_0$ to a position m steps to the "left" using $x_1$ as
          the base.
          $c_1 \leftarrow c_1-1$.
          Goto step 2.
    (3.2) Level 2 reset:
          If $c_2 = 0$ then goto step 3.3.
          Reset $x_1$ to a position $m^2$ steps to the left using $x_2$
          as the base.
          $c_2 \leftarrow c_2 - 1$.
          $c_1 \leftarrow m$
          Goto step 3.1.
                .
                .
                .
```

(3.k-1) Level k-1 reset:

If $c_{k-2} = 0$ then goto step 4.

Reset x_{k-2} to a position m^{k-1} steps to the left using

x_{k-1} as the base.

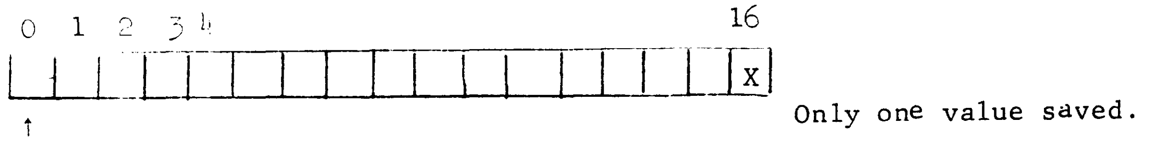
$c_{k-1} \leftarrow c_{k-1} - 1$.

$c_{k-2} \leftarrow m$.

Goto step 3.k-2.

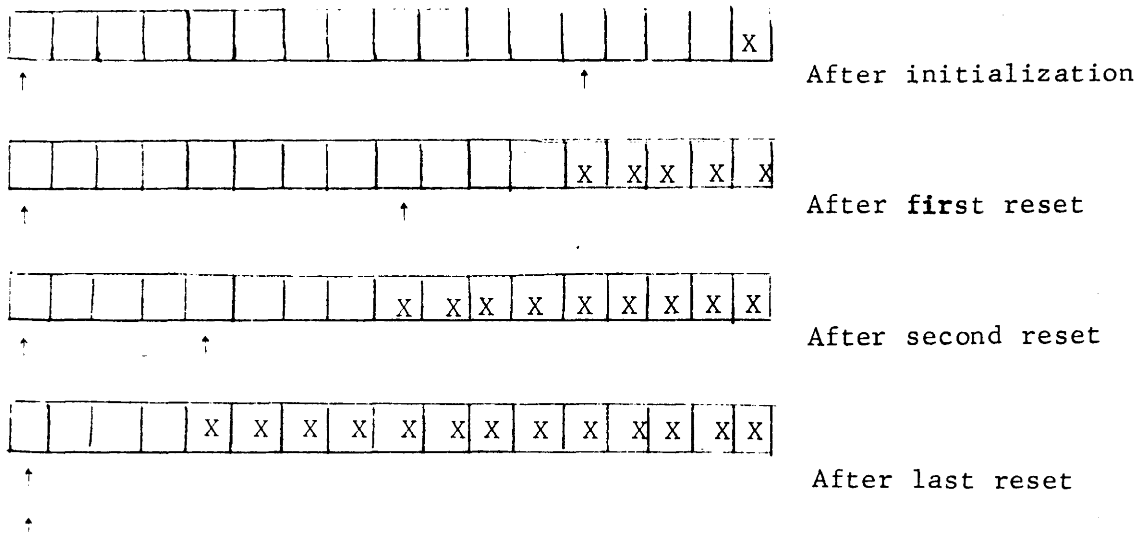
(4) HALT(y).

Figure 3 demonstrates the order-1, order-2 and order-4 algorithms for n = 16. Note that the order-1 algorithm is precisely the conventional n^2 constant space algorithm.



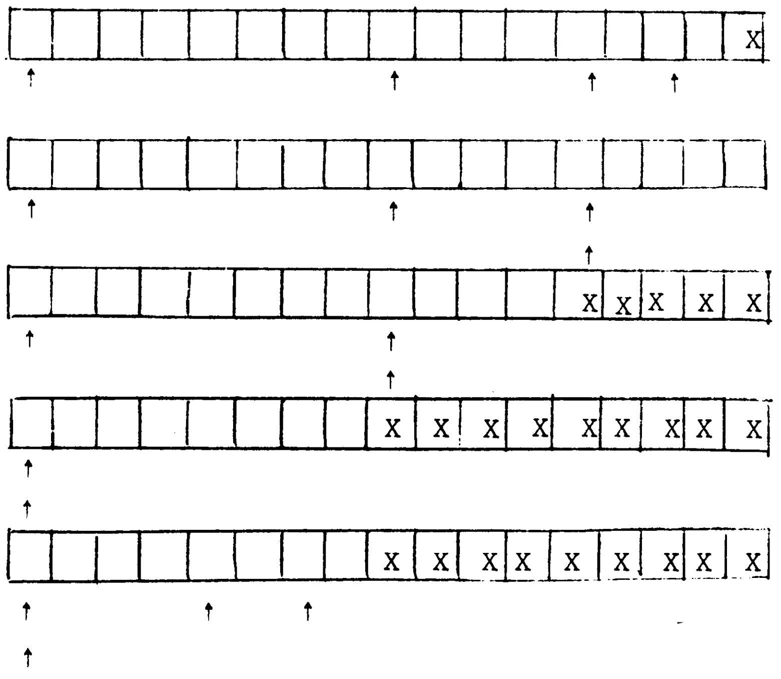
Order-1 constant-space algorithm

Fig. 3a



Order-2 constant-space algorithm

Fig. 3b


X
Initialization
After first level-1 reset
X X X X X
After first level-2 reset
X X X X X X X X X
After first level-3 reset
X X X X X X X X X
After subsequent level-2 and level-1 resets

Order-4 constant-space algorithm

Fig. 3c

Figure 3

The above description deals with the simplified case where n is a k-th power. The general case is given in Section 6.2 of Appendix II, and differs from the above only in technicalities.

Space requirements

The algorithm saves k values. Strictly, v(0) does not have to be saved as it is simply the constant a. In addition, there are a fixed number of bounded counters and additional variables for manipulation of values. Thus the data-space is constant (with respect to n), and proportional to k. The size of the program (the number of states of the automaton for the automaton problem - see Section 5.1) grows linearly with k.

Time requirements

The running time can be divided into

1. The initialization phase: proportional to $n^{1+1/k}$.

2. The computation phase: n steps averaging $n^{1/k}$ each - total $n^{1+1/k}$.

3. Level 1 resets: $n^{1-1/k}$ averaging $n^{2/k}$ each - total $n^{1+1/k}$.

4. Level 2 resets: $n^{1-2/k}$ averaging $n^{3/k}$ each - total $n^{1+1/k}$.

.
.
.

k+1. Level k-1 resets: $n^{1/k}$ averaging time n-total $n^{1+1/k}$.

Thus the total running time grows as $n^{1+1/k}$ and the constant of proportionality is linear with k.

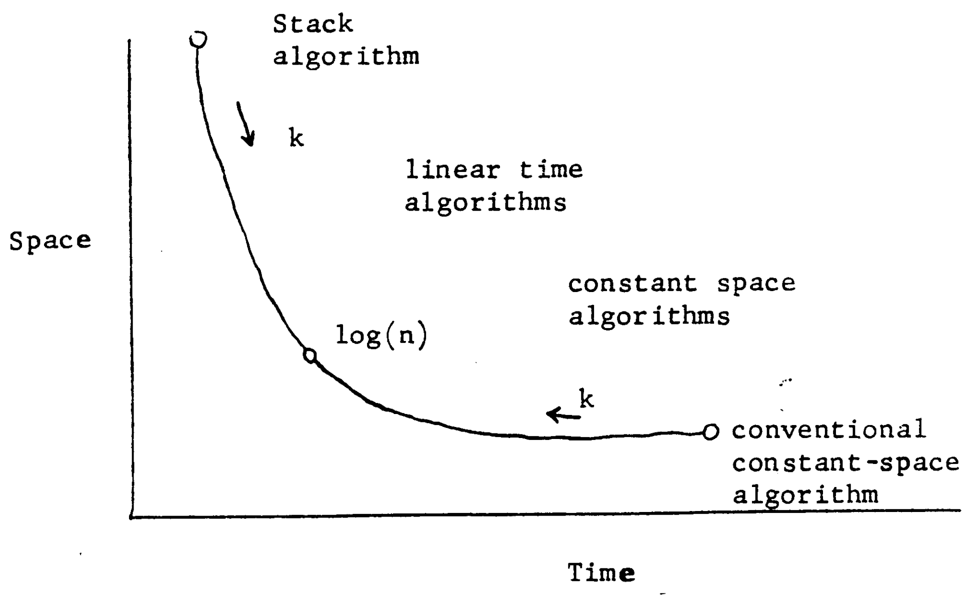
In summary

Given a fixed amount of space one can do significantly better than n^2; in fact the running time can be made $n^{1+1/k}$ for arbitrarily large k. Storage space grows linearly with k, as does the complexity (size) of the program (or finite state automaton).

Constant space algorithms can be quite attractive because all values used in the computation could be stored in the registers of a computer, and in any case the addressing is easier than the log(n) and the linear-time algorithms.

It is fascinating to note that if we let p represent $\log_2(n)$ then the effect of the order-p linear-time, the order-p constant-space, and the log(n) algorithms is (approximately) the same with regard to the running time and storage requirements.

The relationships between the algorithms described above is shown in Figure 4.



Algorithm	time/n	space
Linear-time order-k	k	$k.n^{1/k}$
Log(n)	log(n)	log(n)
Constant-space order-k	$k.n^{1/k}$	k

Figure 4

4. Conclusions

Solving a problem by first converting it into a problem-schema is an interesting concept that merits a great deal more study. The advantage gained is that the solution of the schema can be used to solve several seemingly unrelated problems. An associated advantage is that conversion to a schema usually helps to formalize the problem too. An example of this is the delineation of the kind of statements allowed in a schema. Because of this, however, some care has to be exercised when optimal solutions are required because in this case conversion to a schema requires more stringent conditions: for each construct in the schema there should exist a corresponding base problem construct, and vice versa. Also, simple changes in the ground rules of the base problem can significantly alter the corresponding schema problem.

It was not our objective in this paper to give optimal solutions, just to give good solutions and observe the space-time tradeoffs one can expect. It may have been obvious to the reader that the constant-space algorithms, for example, are not optimal. The number of base operations required (other than the control mechanism) for the order-k constant-space algorithm is

$$\frac{k}{2} \cdot n^{1+1/k}$$

whereas

$$\frac{k!^{1/k}}{1+1/k} \cdot n^{1+1/k}$$

is feasible with the same space, representing a 6% improvement for the order-2 algorithm, 9% for order-3 and 18% for order-10; and even in the limit our simple algorithm does not become arbitrarily bad compared with the other. The price paid for the improvement is the somewhat greater complexity of the control structure, and an increased number of counter operations (which were neglected).

It is reasonable to ask whether the algorithms described can have any real practical significance. Exact machine times for the algorithms are difficult to evaluate owing to machine dependent questions like register allocation, indirect addressing machinery, cache allocation, parallelism and swapping (in a time shared system). We can approximate times, however, by using reasonable assumptions. In the program we assume that each of the base routines (the functions f,g and h, and the predicate p) takes 40 microseconds to evaluate and that operations on counters entail negligible cost. Storing a value (all data variables) is assumed to take 4 machine words, and 64K words of core are available to the user. The following table gives the running times for the various algorithms for recursion depths of 16K, 64K, 256K, and 1M.

	Linear - time				Const-space		
n	k=1	k=2	k=6	Log(n)	k=6	k=2	k=1
16K	1.97 sec	3.27 sec	5.71 sec	6.55 sec	10.4 sec	44.2 sec	89.5 min
64K	impos.	13.1 sec	22.8 sec	28.8 sec	47.5 sec	5.75 min	59.7 hr
256K	impos.	52.4 sec	86.5 sec	126 sec	3.67 min	45.4 min	20 days
1M	impos.	3.49 min	6.51 min	9.09 min	21.3 min	15 hr	1 year

It is clearly indicated that for large recursion depths the stack implementation is not attractive; and for very large recursion depths even the order-2 linear-time algorithm would approach the memory capacities if implemented on present day minicomputers (8K words required for 1M recursion depth). And finally it should be pointed out that even for relatively small recursion depths higher-order linear-time (and even some constant-space) algorithms may be preferred as background jobs in a time sharing system because they need not be swapped out as their core requirements are quite nominal.

In the preceding discussion the model of computation assumes that the size of the data structure remains bounded as computation proceeds. Often, it is more reasonable to assume that the size of the data increases with the depth of recursion, as does the time for a unit operation on the data. The algorithms presented in this paper retain their significance under these conditions, and if anything, become more useful vis a vis the stack algorithm because space restrictions become more severe. For example, if both time for a unit operation and the size of the data structure increase linearly with the depth of recursion, the stack implementation would take space and time n^2 whereas the so called "linear-time" algorithm would take space $k.n^{1+1/k}$ and time $k.n^2$, and the "constant-space" algorithm would take space $k.n$ and time $k.n^{2+1/k}$.

5. Appendix I

5.1 The automaton problem

What is the time required for a finite automaton, with an arbitrary number of reading heads, to output the symbols on its input tape from right to left? The heads can only read from left to right, but the automaton has the capability of taking some reading head and setting it to the same position on the input tape as some other head (note that an automaton without this capability cannot even perform the given task for arbitrarily long input tapes).

To reduce the automaton problem to the schema problem the following correspondence between the finite state automaton and the schema may be set up:

Finite state automaton	Schema
Head i	variable y_i
Move read i to the right	$y_i \leftarrow f(y_i)$
Set head i to the same position as head j	$y_i \leftarrow y_j$
Test if head i is on the last character of the tape	$p(y_i)$
The output file	a special variable y
Output the first character from head i	$y \leftarrow h(y_i)$
Add to the output file from head i	$y \leftarrow g(y_i,y)$

On comparison with the recursive schema

Compute $F(a)$ where
$F(x) \leftarrow$ <u>if</u> $p(x)$ <u>then</u> $h(x)$ <u>else</u> $g(x,F(f(x)))$

we see that if x represents a square on the input tape then $F(x)$ represents the value of the output file with all characters on the right of x (and including it) written in reverse order. This is obtained by first writing all characters on the right of x, i.e. $F(f(x))$, and then appending x to it, i.e. $g(x,F(f(x)))$.

Thus the automaton problem is reduced to the schema problem (without arrays), but with the constraint that the functions h and g can be used only in conjunction with the special variable y as in the statements described, and that the statement $y \leftarrow h(y_i)$ cannot be executed more than once.

The reduction is one-way i.e. a solution of the schema problem (with the constraints) gives a solution of the automaton problem. Of course, the automaton may do fancy things e.g., it may check if its entire tape contains just one character, repeated over and over again, and in this special case it could produce its output in time $2n$. However, the flowchart schema cannot do this as equality tests are not allowed.

It may be argued that the variable y requires not just a unit amount of space but space proportional to n. However, since the finite state automaton is not expected to remember the contents of its output file we may consider that y takes zero space. Hence the assumption that all variables take unit space gives a value for memory requirement one greater than the number of heads required by the automaton.

5.2 The list problem

Given a one way list, to output the elements of the list in reverse order. We are not allowed to change the pointers of the list itself as in the case where the list structure is common to several concurrent processes; and we ask what are the time-memory tradeoffs.

$$\begin{tikzcd}0 \arrow[r] & 0 \arrow[r] & 0 \arrow[r] & 0 & \cdots \arrow[r] & 0\end{tikzcd}$$

This problem is a generalization of the automaton problem because our random-access computer has several features not available to the finite automaton; the number of pointers into the list structure can vary with the size of the given list, two pointers can be tested to see if they happen to point to the same node, etc. In the special case where we restrict our computer to have the capability of a finite automaton we obtain the automaton problem.

The reduction of the list problem to the schema problem is analogous to the reduction of the automaton problem, except that in this case counters and arrays are allowed. Arrays can be used to hold pointers into the list. Pointers are analogous to the heads of the automaton. However, as the arrays are semi-infinite, the number of pointers can increase with the size of the list structure. As in Section 5.1, there is a special variable y representing the output file and the only operations allowed on y are $y \leftarrow h(y_i)$, and $y \leftarrow g(y_i, y)$.

6. Appendix II

6.1 The Linear-Time Algorithm

```
          START
          counter  $\ell, m, n, c_0, \dots, c_k, d_0, \dots, d_k$;
          data $x, y$;
          array  $A_0, \dots, A_k$;
    STEP1:  if $p(a)$ then HALT($h(a)$);
          $n \leftarrow 0$; $x \leftarrow a$;
(1) ---   while $\neg p(a)$ do begin $x \leftarrow f(x)$; $n \leftarrow n + 1$ end;
          $y \leftarrow h(x)$;
(2) ---   $m \leftarrow n^{1/k}$;
(3) ---   $d_0 \leftarrow 1$; $d_1 \leftarrow m$; $d_2 \leftarrow m^2$; ...; $d_k \leftarrow m^k$;
          $\ell \leftarrow n$;  $x \leftarrow a$;
          $c_0 \leftarrow c_1 \leftarrow \dots \leftarrow c_k \leftarrow 0$;
          $A_k[0] \leftarrow x$ ;
          while $\ell > d_k$  do
               begin $\ell \leftarrow \ell - d_k$; $x \leftarrow f^{d_k}(x)$; $c_k \leftarrow c_k + 1$; $A_k[c_k] \leftarrow x$ end;
                 .
                 .
                 .
          $A_0[0] \leftarrow x$;
          while $\ell > d_0$ do
               begin $\ell \leftarrow \ell - d_0$; $x \leftarrow f^{d_0}(x)$; $c_0 \leftarrow c_0 + 1$; $A[c_0] \leftarrow x$ end;
     STEP2:  $y \leftarrow g(A[c_0], y)$;
          if $c_0 = 0$ then goto STEP3;
          $c_0 \leftarrow c_0 - 1$;
          goto STEP2;
```

```
STEP3:

STEP3.1:  if $c_1 = 0$ then goto STEP3.2;
        $c_1 \leftarrow c_1 - 1$;
        $A_0[0] \leftarrow x \leftarrow A_1[c_1]$;
        for $c_0 \leftarrow 1$ step 1 until m-1 do $A_0[c_0] \leftarrow x \leftarrow f^{d_0}(x)$;
        goto STEP2;
                .
                .
                .
STEP3.k:  if $c_k = 0$ then goto STEP4;
        $c_k \leftarrow c_k - 1$;
        $A_{k-1}[0] \leftarrow x \leftarrow A_k[c_k]$;
        for $c_{k-1} \leftarrow 1$ step 1 until m-1 do $A_{k-1}[c_{k-1}] \leftarrow x \leftarrow f^{d_{k-1}}(x)$;
STEP4:  HALT(y).
```

The program follows the algorithm of Section 3.2 very closely. In the initialization phase (step 1) line (1) computes the value of n. Line (2) assigns to the counter m the largest value such that $m^k \leq n$. Note that this can be done just with the operations of +1, -1 and test for zero in linear time. Line (3) computes the relevent powers of m (these can be computed simultaneously while m is being computed). The counters $c_0, \ldots, c_k$ denote the number of values saved at each level. There is some overlap in values saved which could be avoided. As shown the initialization phase involves two passes over the range of data values a through $f^n(a)$. This can be done in a single pass for k=1 since all increments are constant (one) independent of the value of n.

6.2 The Constant-space Algorithm

```
START
counter $\ell, m, n, c_0, \ldots, c_k, d_0, \ldots, d_k$;
data $y, x_0, \ldots, x_k$;
STEP1: if $p(a)$ then HALT($h(a)$);
$n \leftarrow 0$;  $x \leftarrow a$;
while $\neg p(a)$ do begin $x \leftarrow f(x)$; $n \leftarrow n + 1$ end;
$y \leftarrow h(x)$;
$m \leftarrow n^{1/k}$;
$d_0 \leftarrow 1$;  $d_1 \leftarrow m$;  $d_2 \leftarrow m^2$; ... ; $d_k \leftarrow m^k$;
$\ell \leftarrow n$;  $x \leftarrow a$;
$c_0 \leftarrow c_1 \leftarrow \ldots \leftarrow c_k \leftarrow 0$;
$x_k \leftarrow x$;
while $\ell > d_k$ do
     begin $\ell \leftarrow \ell - d_k$;  $x \leftarrow f^{d_k}(x)$; $c_k \leftarrow c_k + 1$ end;
          .
          .
          .
$x_0 \leftarrow x$;
while $\ell > d_0$ do
     begin $\ell \leftarrow \ell - d_1$; $x \leftarrow f^{d_0}(x)$; $c_0 \leftarrow c_0 + 1$ end;
STEP2:  $y \leftarrow g(f^{c_0}(x_0), y)$;
if $c_0 = 0$ then goto STEP3;
$c_0 \leftarrow c_0 - 1$;
goto STEP2;
STEP3 :
STEP3.1: if $c_1 = 0$ then goto STEP3.2;
$c_1 \leftarrow c_1 - 1$;
```

```
$x_0 \leftarrow f^{c_1 * d_1}(x_1)$;
$c_0 \leftarrow m - 1$;
goto STEP2;
        .
        .
        .
STEP3.k:  if $c_k = 0$ then goto STEP4;
$c_k \leftarrow c_k - 1$;
$x_{k-1} \leftarrow f^{c_k * d_k}(x_k)$;
$c_{k-1} \leftarrow m - 1$;
goto STEP3.k-1;
STEP4:  HALT(y).
```

The initialization phase is shown here for the case where explicit counters are allowed. It closely parallels the initialization phase in the linear-time program (Section 6.1). The rest of the program can be implemented using only the counter operations -1 and test for zero which means it can be directly implemented without any explicit counter (see Section 3.1).

The initialization phase can be implemented without any explicit counters as follows. Variables $m, c_0 \ldots, c_k, d_0, \ldots, d_k$ are used to represent the corresponding counters in the rest of the program. In addition, variables $m', x', c'_0, \ldots, c'_k$ are used as temporaries. We make use of the following nonrecursive procedures for convenience in defining the operation of the program.

```
procedure invert (c');
    begin local c,d;
    $c \leftarrow a$;  $d \leftarrow c'$;
    while $\neg p(d)$ do begin $c \leftarrow f(c)$; $d \leftarrow f(d)$; end;
    return(c);
    end;
```

```
procedure multiply ($c_1$,$c_2$);
    begin local $d_1$,$d_2$, val;
    val ← a;  $d_1$ ← $c_1$;
    while ¬p($d_1$) do
        begin
        $d_2$ ← $c_2$;
        while ¬p($d_2$) do
            begin
            val ← f(val);
            $d_2$ ← f($d_2$);
            end;
        $d_1$ ← f($d_1$);
        end;
    return(invert(val));
    end;
procedure right (x,c);
    begin local d,val;
    val ← x;  d ← c;
    while ¬p(d) ∧ ¬p(val) do
        begin
        val ← f(val);
        d ← f(d);
        end;
    return(val);
    end;
```

The initialization part can now be written as:

```
STEP1:  if p(0) then HALT(h(a));
    if p(f(a)) then HALT(g(a,h(f(a))));
```

```
          $m' \leftarrow a$;
          $y \leftarrow h(\text{invert}(m'))$;
TRY:  $m' \leftarrow f(m')$;  $m \leftarrow \text{invert}(m')$;
          $\ell \leftarrow a$;
          $c_1 \leftarrow m$;
$\text{TRY}_1$:  if $p(c_1)$ then goto TRY:
          $c_1 \leftarrow f(c_1)$;
          $c_2 \leftarrow m$;
$\text{TRY}_2$:  if $p(c_2)$ then goto $\text{TRY}_1$;
          $c_2 \leftarrow f(c_2)$;
          $c_3 \leftarrow m$;
             .
             .
             .
$\text{TRY}_k$:  if $p(c_k)$ then goto $\text{TRY}_{k-1}$;
          $c_k \leftarrow f(c_k)$;
          if $p(\ell)$ then goto FOUND;
          $\ell \leftarrow f(\ell)$;
          goto $\text{TRY}_k$;
FOUND:  $m \leftarrow f(m)$;  comment:  m  has now been found;
          $d_0 \leftarrow \text{invert}(f(a))$;
          $d_1 \leftarrow m$;
          $d_2 \leftarrow \text{multiply}(d_1,d_1)$;
             .
             .
             .
          $d_k \leftarrow \text{multiply}(d_1,d_{k-1})$;   comment:  $d_0,\ldots,d_k$ have been determined;
          $x \leftarrow a$;
          $c'_0 \leftarrow c'_1 \leftarrow \ldots \leftarrow c'_k \leftarrow a$;
```

```
$x_k \leftarrow x$;
while $\neg p(right(x,d_k))$ do
    begin $x \leftarrow right(x,d_k)$; $c'_k \leftarrow f(c'_k)$ end;
  .
  .
  .
$x_0 \leftarrow x$;
while $\neg p(right(x,d_0))$ do
    begin $x \leftarrow right(x,d_0)$;  $c'_0 \leftarrow f(c'_0)$end;
$c_0 \leftarrow invert(c'_0)$; ... ; $c_k \leftarrow invert(c'_k)$;
                                        comment:  this completes the initialization;
```

This completes the description of the constant-space algorithm.

Acknowledgement

I am indebted to Zohar Manna for his helpful advice and many suggestions, and to Dave Plaisted for his careful reading of an earlier version of this paper and detailed comments.

REFERENCES

[1] Chandra, A.K., "On the Properties and Applications of Program Schemas", (1972), Ph.D. Thesis, Computer Science Department, Stanford University, (to appear).

[2] Chandra, A.K., Manna, Z., "Program Schemas With Equality", Proceedings of the Fourth Annual ACM Symposium on Theory of Computing, Denver, Colorado, May 1972, pp 52-64.

[3] Cocke, J., and Schwartz, J.T., "Programming Languages and Their Compilers", preliminary notes, Courant Inst. of Math. Sciences, New York University (1970).

[4] Constable, R.L., and Gries, D., "On Classes of Program Schemata", SIAM Journal on Computing, Vol. 1, No. 1, March 1972, pp. 66-118.

[5] Garland, S.J., and Luckham, D.C., "Program Schemes, Recursion Schemes, and Formal Languages", UCLA Report No. ENG-7154, June 1971.

[6] Gries, D., "Compiler Construction for Digital Computers", John Wiley and Sons, Inc., New York, 1971.

[7] Hewitt, C., "More Comparative Schematology", Artificial Intelligence Memo No. 207, Project MAC, M.I.T., August 1970.

[8] Hopgood, F.R.A., "Compiling Techniques", American Elsevier, Inc., New York, 1969.

[9] McCarthy, J., "Towards a Mathematical Science of Computation", Proc. ICIP, 1962.

[10] Paterson, M.S., and Hewitt, C.E., "Comparative Schematology", Artificial Intelligence Memo, No. 201, Project MAC, M.I.T., November 1970.

[11] Strong, H.R., and Walker, S.A., "Properties Preserved Under Recursion Removal", pp 97-103 in the Proceedings of the ACM Conference on Proving Assertions about Programs, Las Cruces, January 1972.