pile time. Second, we assume no problems of finite precision. Both of these assumptions could lead to difficulties in an actual implementation. Whenever the effective position of a computation is moved, as it is in these strength reduction methods, proper attention should be paid to such "safety" considerations, lest an unexpected error interrupt occur. It does no good to produce a faster version of the program if the faster version is incorrect. The interested reader can find an expanded treatment of safety in [11].

A final assumption made in using such a strength reduction technique is that it is profitable. The profitability assumption is based on the observations that code within a loop is executed frequently and multiplications are significantly more expensive than additions on most machines. These assumptions are not valid in all cases, however. For example, if we were to replace a multiplication on a little-used branch in the loop by additions on more frequent paths, we might significantly deoptimize the compiled code. This situation can be avoided by a more careful analysis of profitability such as the one described in [12].

Acknowledgments. The authors wish to thank Frances Allen of IBM and Jack Schwartz of New York University for their many helpful suggestions. The referee provided several valuable suggestions for improvement of the final manuscript. We are particularly grateful for his comments on the question of safety, some of which we adapted in the summary and conclusions.

Received October 1974, revised August 1976

References

1. Allen, F.E. Program Optimization. *Annual Review of Automatic Programming, Vol. 5,* Pergamon Press, New York, 1969.
2. Cocke, J., and Schwartz, J. *Programming Languages and Their Compilers.* Courant Institute of Mathematical Sciences, New York U., New York, 1970.
3. Kennedy, K. A global flow analysis algorithm. *Int. J. Comptr. Math., Sect. A, 3* (Dec. 1971), 5-15.
4. Hecht, M.S., and Ullman, J.D. Flow graph reducibility. *SIAM J. Comptng. 1*, 2 (June 1972), 188-202.
5. Kildall, G.A. A unified approach to global program optimization. Conf. Rec. ACM Conf. on Principles of Programming Languages, Boston, Mass., Oct. 1973, pp. 194-206.
6. Ullman, J. D. Fast algorithms for the elimination of common subexpressions. *Acta Informatica 2* (1973), 191-213.
7. Earnest, C., Balke, K.G., and Anderson, J. Analysis of graphs by ordering of nodes. *JACM 19*, 1 (Jan 1972), 23-42.
8. Allen, F.E. Control flow analysis, SIGPLAN Notices (ACM) *5*, 7 (July 1970), 1-19.
9. Kennedy, K. Variable subsumption with constant folding. SETL
York, Feb. 1974.
10. Kennedy, K. Use-definition chains with applications. Tech. Rep. 476-093-9, Dept. Math. Sci., Rice U., Houston, Tex., April 1975.
11. Kennedy, K. Safety of code motion. *Int. J. Comptr. Math., Sect. A*, 3 (1972), 117-130.
12. Cocke, J., and Kennedy, K. Profitability computations on program flow graphs. Tech. Rep. 476-093-3, Dept. Math. Sci., Rice U., Houston, Tex., May 1974.

Programming Techniques

R.L. Rivest,* S.L. Graham
Editors

Improving Programs by the Introduction of Recursion

R.S. Bird
University of Reading

A new technique of program transformation, called "recursion introduction," is described and applied to two algorithms which solve pattern matching problems. By using recursion introduction, algorithms which manipulate a stack are first translated into recursive algorithms in which no stack operations occur. These algorithms are then subjected to a second transformation, a method of recursion elimination called "tabulation," to produce programs with a very efficient running time. In particular, it is shown how the fast linear pattern matching algorithm of Knuth, Morris, and Pratt can be derived in a few steps from a simple nonlinear stack algorithm.

Key Words and Phrases: program transformation, optimization of programs, recursion elimination, pattern matching algorithms, stacks, computational induction

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

Copyright © 1977, Association for Computing Machinery, Inc. General permission to republish, but not for profit, all or part of this material is granted provided that ACM's copyright notice is given and that reference is made to the publication, to its date of issue, and to the fact that reprinting privileges were granted by permission of the Association for Computing Machinery.

* This paper was submitted prior to the time that Rivest became editor of the department, and editorial consideration was completed under the former editor, G. K. Manacher.

Author's address: Department of Computer Science, University of Reading, Whiteknights Park, Reading, Berkshire, England.

1. Introduction

Formula manipulation is a basic activity in many branches of mathematics, not only because of a natural desire to express relationships in their simplest possible terms, but also as a means to develop new insights about the problem in hand. In computer science we manipulate programs rather than formulas or equations, but the activity is much the same. Usually the chief object of program transformation is to produce not necessarily a simpler program, but at least a program which is more efficient in an appropriate sense, say one that uses less space or has a faster running time.

As Knuth [8] observes, a "calculus" of program transformations is gradually emerging, a set of operations which can be applied to programs without rethinking the specific problem each time. Among such operations one may mention the following: doubling up of loops, Boolean variable elimination, and a number of techniques for recursion elimination. The first two are described in Knuth [8] and are embodied in the linear speed-up transformation of Bird [2]; the last is described in Bird [3] and Darlington and Burstall [7], among others. Recursion elimination is particularly important because so many algorithms can be expressed naturally and succinctly in recursive form, even though this form may not be the most suitable one for execution on a computer. Instead of using the facilities for recursion provided by the programming language in which the algorithm is to be represented (always assuming, of course, that the language does possess such a facility) and relying on the compiler to take care of the details, the programmer can often obtain a significant increase in efficiency by designing his or her own implementation of recursion, specially tailored to the given problem.

The particular technique of program transformation to be described in this paper represents a sort of inverse activity to recursion elimination, namely the conscious introduction of recursion where none existed before. For this reason, we call it *recursion introduction*. Although recursion makes its appearance only as an intermediate step and is eliminated again almost immediately, the insight gained leads to much faster versions of the programs to which the technique can be applied. The characteristic property of the programs which can make use of recursion introduction is that they manipulate a stack. The relationship between recursion and stacks is well known; indeed it is precisely this relationship that is exploited in many methods of recursion elimination. What is not so often appreciated is that the relationship is a complementary one; the manipulation of a stack can be eliminated in favor of a recursive mode of operation (see Brown, Gries, and Szymanski [5] or Chandra [6] for a precise formulation and proof of this result). It is this idea which forms the basis of the technique.

Recursion introduction would be a pointless activity were it not for the fact that the recursion can be eliminated again without recourse to a stack. To do this a second transformation called *tabulation* is invoked. Briefly, tabulation is the process of reducing the amount of work done in calculating the values of a recursively defined function f by storing the values in a table. In this way, each value of f is computed just once from the recursive definition; subsequent requests for the value are obtained simply by looking up the table. Tabulation is thus a variant of the well-known technique of dynamic programming, discussed in Aho, Hopcroft, and Ullman [1]. Once the general strategy of tabulation has been adopted, it can often be implemented very efficiently by taking into account the properties of the function under consideration. We shall see the technique at work in Sections 3 and 4.

Although programs which manipulate a stack are not, perhaps, that common (excepting the case where the stack has been introduced specifically to eliminate recursion), there is at least one important area in which such algorithms are conceived. This is the area of pattern matching. The simplest example of a pattern matching problem is the problem of determining whether a given string of m symbols, called a *pattern*, occurs as a substring of another given string of n symbols, called a *text*. As we shall see in Section 3, a comparatively simple stack algorithm can be given to solve this problem, one which works within $O(mn)$ steps. Recently, a linear time (i.e. $O(m + n)$) solution has been given by Knuth, Morris, and Pratt [9]. The discovery of the fast algorithm has an interesting history which is detailed in [9]. Briefly, the naive stack algorithm conforms to the conditions of a theorem due to Cook (discussed in Aho, Hopcroft, and Ullman [1]), who showed by a fairly complicated simulation that certain stack algorithms, no matter how long they took to execute, could be simulated in linear time. By pondering the details of this simulation, Knuth was able to extract the essentials of the fast pattern matching algorithm. This algorithm was later generalized so that its running time was not dependent on the size of the input alphabet. Cook's theorem is important because it shows that for a variety of problems linear solutions do exist. However, as Aho, Hopcroft, and Ullman [1] remark, the constant factor arising from the use of Cook's simulation directly is quite large; so further search for better linear solutions is necessary. The immediate significance of recursion introduction, as the following examples show, is that once, possibly inefficient, stack algorithms are given, they can be mechanically transformed into fast linear solutions for the problem.

In Section 3 we demonstrate how the fast linear pattern matching algorithm can be derived in a few steps from a simple nonlinear stack algorithm. A second example of the use of recursion introduction is considered in Section 4, where a nonlinear stack algorithm for determining whether given patterns in a set are all different is transformed into a linear algorithm. The next section outlines the recursion introduction method.

2. Recursion Introduction

The basic strategy governing the conversion of stack algorithms into recursive programs is to regard each symbol on the stack as recording an obligation to execute an appropriate recursive procedure. More precisely, if at some stage during the operation of a stack algorithm P the stack contains symbols $s_1, s_2, \ldots, s_n$, then the recursive program Q will be designed so that at the corresponding stage the calls $R(s_1)$, $R(s_2)$, . . . , $R(s_n)$ of a recursive procedure R are waiting for future evaluation. The definition of R is determined by the instructions of P which process the stack symbols. Rather than deal with the general implementation of this strategy for an arbitrary stack algorithm, we consider only the case where P is an algorithm of the form

```
S := empty;
A;
while S not empty do
  begin z ⇐ S;
    B
  end;
C
```

in which the processing of a stack S is determined by a single loop. Portions A and B of P may contain stack assignments (written in the form $S \Leftarrow x$) as well as nonstack instructions, while C may contain only the latter type. The important point is that P may not contain any further instructions of the form $x \Leftarrow S$. Not only is this form of algorithm sufficient for the purposes of the present paper, it also enables recursion to be introduced in a straightforward fashion. The recursive program Q corresponding to P takes the form

```
begin procedure R(x);
    begin z := x; B* end;
  A*; C
end,
```

where A^* and B^* are translations of the portions A and B, respectively, and may involve calls to the procedure R. To see how A^* is defined, suppose that for a given vector v of variables the overall effect of A is to carry out the operations

$$v := f(v);\ S \Leftarrow x_1;\ S \Leftarrow x_2;\ \ldots;\ S \Leftarrow x_n$$

so that the string $x_n\, x_{n-1} \ldots x_1$ is added to the top of the stack. The principle determining the construction of A^* is that A^* should, in effect, carry out the operations

$$v := f(v);\ R(x_n);\ R(x_{n-1});\ \ldots;\ R(x_1)$$

in accordance with the strategy outlined above. Exactly the same principle operates in translating B into B^*. Since C does not contain stack instructions, we have $C^* = C$. Notice that the state vector v is treated as global to the procedure R. Once the principle of the translation is understood, there is no need to formulate exact rules for the construction of A^* and B^*, as particular cases may best be dealt with in different ways. For instance, in the case that B contains no assignment to the variable z, and so only the initial value of z is relevant to B, the instruction $z := x$ can be removed from the body of R and the definition

```
procedure R(z);
  begin B* end
```

used instead.

The examples which follow in Sections 3 and 4 illustrate further aspects of the method and also contain details as to how the recursion is eliminated once again without recourse to a stack.

3. First Example of Recursion Introduction

In this section we apply recursion introduction to the basic pattern matching problem and hence derive the fast algorithm of Knuth, Morris, and Pratt. The problem is this: Given a pattern of symbols $p[1]$, $p[2]$, . . . , $p[m]$, to determine whether or not it matches some substring of a text $t[1]$, $t[2]$, . . . , $t[n]$. We can design a stack algorithm to solve this problem in the following way. First, the text is stored on a stack S, with $t[1]$ on top, and a pointer is initialized to the first symbol of the pattern. At this stage the first pass over the pattern begins (in general, the effect of the kth pass will be to determine whether or not the pattern occurs as an initial substring of $t[k]$, $t[k + 1]$, . . . , $t[n]$): As long as the stack is not empty, the top symbol is removed and compared to the current symbol of the pattern, provided such a symbol exists. If they match, then the pointer is advanced and the process repeated. If they do not match, then the stack is restored to its initial configuration at the beginning of the pass by loading the pattern symbols to the left of the pointer. The first symbol on the stack is then removed, ready for the next pass. If the pattern is exhausted during a pass, then the algorithm stops, indicating a successful match; otherwise, when all the passes have been completed, the algorithm stops, indicating no match.

This description is formalized in the following program:

Algorithm 1

```
begin stack S; S := empty;
  j := n + 1; repeat j := j − 1; S ⇐ t[j] until j = 1;
  while j ≤ m and S not empty do
    begin x ⇐ S;
      if p[j] = x then j := j + 1 else
      if j ≠ 1 then
        begin S ⇐ x; j := j − 1;
          while j ≠ 1 do
            begin S ⇐ p[j]; j := j − 1 end
        end
    end;
  if j > m then MATCH else NO MATCH
end
```

The first step in the process of simplifying this algorithm is to eliminate the stack S by introducing recursion in the manner described in Section 2. Prior to the stack processing loop, the stack S is initialized to contain the symbols $t[1], t[2], \ldots t[n]$ in order, and j is set to 1; so the recursive algorithm takes the form

Algorithm 2

```
j := 1; for k := 1 to n do R(t[k]);
if j > m then MATCH else NO MATCH.
```

The definition of R is determined by the body of the central loop and is given by

```
procedure R(x);
   begin if j > m then goto M;
      if p[j] = x then j := j + 1 else
      if j ≠ 1 then
         begin T; R(x) end
   M: end
```

The first point to note is the presence of the test $j > m$. This reflects the fact that the central loop of Algorithm 1 could equally well have been stated in the slightly different form

```
while S not empty do
   begin x ⇐ S; if j > m then goto M;
   . . . . . . . . . .
   M : end
```

the difference being only that this version always empties the stack before terminating.

Second, note that the definition of R involves a call to a further as yet unspecified procedure T; this procedure will be designed to simulate the stack operations in the code

```
j := j - 1; while j ≠ 1 do begin S ⇐ p[j]; j := j - 1 end
```

Finally, observe the terminal call $R(x)$ in the definition of R; this corresponds to the assignment $S \Leftarrow x$ in the central loop. Actually, this call can immediately be eliminated in favor of a direct jump to the beginning of R. The principle involved here is the simplest case of recursion elimination, namely, when the last action of a recursive procedure is to call itself, then that call can be replaced, after reassigning the parameter if necessary, by a direct jump to the first instruction of the procedure. We shall not pause to give a justification of this principle as an excellent discussion can be found in Knuth [7]. Eliminating the call gives the following new definition for R:

```
procedure R(x);
   begin L: if j > m then goto M;
            if p[j] = x then j := j + 1 else
            if j ≠ 1 then begin T; goto L end
   M: end
```

We can even go one stage further and incorporate the body of R directly into Algorithm 2. After simplifying the looping structure in an obvious way, we obtain the following version of the algorithm:

Algorithm 3

```
j := k := 1;
while k ≤ n and j ≤ m do
   begin if p[j] = t[k] then begin j := j + 1; k := k + 1 end
      else if j = 1 then k := k + 1 else T
   end;
if j > m then MATCH else NO MATCH
```

We are thus left with the definition of T to consider. One way of defining T is as follows:

```
procedure T;
   begin integer k; j := j - 1;
      if j ≠ 1 then begin k := j; T; R(p[k]) end
   end
```

To justify this definition, observe that the code which T has to simulate places the string $p[2]\,p[3] \ldots p[j-1]$ on the stack and also sets j to 1. This translates in the recursive version to procedure calls $R(p[2]), \ldots, R(p[j-1])$ which are invoked after j has been set to 1. One way of achieving this is given above. In a similar manner as before, we can now substitute the definition of $R(p[k])$ directly into the body of procedure T. This eliminates all references to procedure R; so we are left with just the single recursive procedure T.

As preparation for the next step, notice that the effect of T is simply to change the value of j. In other words, we can regard T as an assignment of the form $j := f(j)$ for an appropriate function f. We can even say what f is since its definition can be extracted from that of T. We have

```
function f(x);
   begin integer y;
      if x = 2 then return 1 else
         begin y := f(x - 1);
            L: if y > m then return y else
               if p[y] = p[x - 1] then return y + 1 else
               if y ≠ 1 then begin y := f(y); goto L end else
               return y
   end
```

Although the definition of f is given for a general x, we are only interested in the range $2 \le x \le m$, simply because the assignment $j := f(j)$ (i.e. procedure T) is only invoked for $2 \le j \le m$. We are guaranteed that f is well defined in this range since f has been derived from a terminating algorithm.

At this stage we apply the idea of *tabulation* to reduce the amount of work done in calculating the values of f. The basic strategy is to compute the value $f(x)$ from the recursive definition just once for each x and store the result in a table F so that $F[x] = f(x)$. Subsequent calculations of $f(x)$ are obtained simply by using $F[x]$ instead. Initially each entry in F is set to zero, and the recursive call $y := f(y)$ in the definition of f is replaced by the code

```
if F[y] ≠ 0 then y := F[y] else
              begin z := f(y); F[y] := z; y := z end
```

Since $f(x) > 0$ for $x \ge 2$, these instructions determine whether or not the value of $f(y)$ has previously been calculated, and, if not, then the value is calculated and stored in the table for future use. Similar instructions have to be defined for the assignment $y := f(x-1)$.

This general strategy can be implemented more efficiently by taking advantage of the properties of f. It is not too difficult to show that $f(x) \le x$ for $x \ge 2$, so the value of $f(x+1)$ depends only on the values $f(2), \ldots, f(x)$. This means we can compute F sequentially, as in the following algorithm:

```
begin array $F[2:m]$;
          $F[2] := 1$;
          for $x := 3$ to $m$ do
            begin $y := F[x-1]$;
              $L$: if $p[y] = p[x-1]$ then $F[x] := y+1$ else
                if $y \neq 1$ then
                  begin $y := F[y]$; goto $L$ end
                else $F[x] := 1$
            end
end
```

Notice that the array element $F[1]$ does not exist. We can, however, simplify the logic of the algorithm by introducing a value $F[1] = 0$ and constructing the array as follows:

```
begin array $F[1:m]$;
  $F[1] := 0$; $F[2] := 1$;
  for $x := 3$ to $m$ do
    begin $y := F[x-1]$;
      while $y > 0$ and $p[y] \neq p[x-1]$ do $y := F[y]$;
      $F[x] := y+1$
    end
end
```

Having calculated the array F, we can replace the procedure call T in Algorithm 3 by the assignment $j := F[j]$. This gives us the Knuth-Morris-Pratt algorithm. Although we shall not prove it here, the computation of F and the pattern matching procedure itself both have a running time linear in $m + n$, the constant factor being independent of the size of the input alphabet. (See [1] or [9].)

Notice how the function f has been revealed to us simply by a sequence of mechanical transformations from the stack algorithm. At no stage was it necessary to relate f to the particular symbols in the pattern or text, even though a natural interpretation of f does exist (again, see [1] or [9]). Such a situation seems to be in the best mathematical tradition: A purely formal manipulation of expressions has brought forth new and unexpected relationships about the problem under consideration.

3. Second Example of Recursion Introduction

Our second example of the use of recursion introduction deals with the problem of recognizing a sequence of distinct patterns. Given is a string $cx_1cx_2 \ldots cx_nc$ of symbols, where each x_j is a (possibly empty) string of symbols over some alphabet Σ and c is a symbol not in Σ. The problem is to determine whether or not there exist j and k, with $j \neq k$, such that $x_j = x_k$.

An algorithm which uses a stack S and a text pointer j can be developed for this problem based on the following idea. At stage k $(1 \leq k \leq n)$ in the computation, S is empty and j points to the symbol c immediately following the pattern x_k. S is now initialized by back-spacing j and copying the input onto S until a symbol c is reached. Thus S will now contain x_kc. The pointer j is advanced to the first symbol of x_{k+1}, and the following process now takes place. As long as the top stack symbol matches the current input symbol, the stack is popped and the pointer is advanced. If there is a mismatch, the stack is restored by back-spacing the pointer to the previous c while copying each input symbol back onto S. The pointer is then advanced again to the first symbol of the next pattern, regarding pattern x_1 as following x_n, and the process is repeated. Clearly, the stack must be emptied by this procedure because eventually we shall successfully match x_kc against itself if no previous match were found. Thus the stack will be emptied either because we match x_k against an x_m for $k < m \leq n$, in which case we can immediately terminate the algorithm and reject the input, or because we match x_k against itself. We know there can be no m with $1 \leq m < k$ such that $x_m = x_k$; otherwise this fact would have been detected at stage m. In the case that x_k is matched only against itself, the pointer is advanced to the next c, and stage $k+1$ is entered if $k+1 < n$. If there are no more stages to do, the algorithm terminates and the input is accepted.

We can formalize the algorithm with the help of an auxiliary function σ, designed so that $\sigma(j)$ gives the position of the first c to the right of position j. The definition is

```
function $\sigma(x)$;
  begin repeat $x := x+1$ until $a[x] = c$;
    if $x = N$ then return 0 else return $x$
  end
```

Here we are supposing that the complete input text is stored in an array $a[0], \ldots, a[N]$. The central loop on which the algorithm is based can now be described as follows:

Central Loop

```
repeat $S \Leftarrow a[j]$; $j := j-1$ until $a[j] = c$;
$j := \sigma(j)+1$;
while $S$ not empty do
  begin $b \Leftarrow S$;
    if $a[j] = b$ then $j := j+1$ else
      begin $S \Leftarrow b$; $j := j-1$;
        while $a[j] \neq c$ do
          begin $S \Leftarrow a[j]$; $j := j-1$ end;
        $j := \sigma(j)+1$
      end
  end;
$j := j-1$
```

For the reasons given above, this program must always terminate. If the final value of j is equal to its initial value, then we must go on to the next stage of the computation, as no proper match was found; otherwise we terminate the algorithm and reject the input. The complete algorithm can therefore put in the form

Algorithm 1

```
$j := \sigma(0)$; $x := j$;
```

```
while $j \neq 0$ and $x = j$ do
  begin central loop;
    if $x = j$ then begin $j := \sigma(j)$; $x := j$ end
  end;
if $x = j$ then NOMATCH else MATCH
```

Once again, the transformation used in the previous example is applied to reduce Algorithm 1 to a linear-time algorithm. We replace the use of S by calls to a recursive procedure R. If at some point during the execution of algorithm 1 the stack contains $a[j_1], \ldots, a[j_m]$, then, at the corresponding point during the recursive algorithm, procedure calls $R(j_1), \ldots, R(j_m)$ are waiting for evaluation. The definition of R is extracted from the definition of *central loop* and is as follows:

```
procedure $R(k)$;
  begin if $a[j] = a[k]$ then $j := j + 1$ else
          begin $T$; $R(k)$ end
  end
```

Here T is a procedure which simulates the code

```
$j := j - 1$;
while $a[j] \neq c$ do begin $S \Leftarrow a[j]$; $j := j - 1$ end;
$j := \sigma(j) + 1$.
```

Its definition is thus

```
procedure $T$;
  begin integer $k$; $j := j - 1$;
    if $a[j] = c$ then $j := \sigma(j) + 1$ else
      begin $k := j$; $T$; $R(k)$ end
  end
```

T can also be used to define the appropriate simulation of *central loop*, which now becomes the operations

$$k := j;\ T;\ R(k);\ j := j - 1.$$

The full algorithm therefore takes the following form:

Algorithm 2

```
$j := \sigma(0)$; $k := j$;
while $j \neq 0$ and $k = j$ do
  begin $T$; $R(k)$; $j := j - 1$;
    if $k = j$ then begin $j := \sigma(j)$; $k := j$ end
  end;
if $k = j$ then NOMATCH else MATCH
```

The explicit recursion in procedure R can be eliminated as in Section 3 and yields the new definition

```
procedure $R(k)$;
  begin while $a[j] \neq a[k]$ do $T$;
    $j := j + 1$
  end
```

This leads to the following new definition of T:

```
procedure $T$;
  begin integer $k$; $j := j - 1$;
    if $a[j] = c$ then $j := \sigma(j) + 1$ else
      begin $k := j$;
        repeat $T$ until $a[j] = a[k]$;
        $j := j + 1$
      end
  end
```

As in Section 3, we can regard T as equivalent to an assignment $j := f(j)$, where we have

```
function $f(x)$;
  begin integer $y$;
    if $a[x - 1] = c$ then return $\sigma(x - 1) + 1$ else
      begin $y := x - 1$;
        repeat $y := f(y)$ until $a[y] = a[x - 1]$;
        return $y + 1$
      end
  end
```

At this stage the full algorithm takes the form

Algorithm 3

```
$j := \sigma(0)$; $k := j$;
while $j \neq 0$ and $k = j$ do
  begin repeat $j := f(j)$ until $a[j] = a[k]$;
    if $k = j$ then begin $j := \sigma(j)$; $k := j$ end
  end;
if $k = j$ then NOMATCH else MATCH
```

Looking closely at Algorithm 3, we can discover the way to further simplification. Suppose we let $j := g(k)$ abbreviate the instructions

$$j := k;\ \textbf{repeat}\ j := f(j)\ \textbf{until}\ a[j] = a[k].$$

Using g we can put Algorithm 3 in the form

Algorithm 4

```
$j := \sigma(0)$;
while $j \neq 0$ and $j = g(j)$ do $j := \sigma(j)$;
if $j = 0$ then NOMATCH else MATCH.
```

This very simple statement of the algorithm suggests that g is going to be a more useful function than f. We can define f in terms of g as follows:

```
function $f(x)$;
  if $a[x - 1] = c$ then return $\sigma(x - 1) + 1$
                  else return $g(x - 1) + 1$
```

This means that we can give a definition of g solely in terms of g, namely,

```
function $g(x)$;
  begin integer $y$; $y := x$;
    repeat if $a[y - 1] = c$ then $y := \sigma(y - 1) + 1$
                             else $y := g(y - 1) + 1$
    until $a[y] = a[x]$;
    return $y$
  end
```

This recursive definition of g can be simplified by a change of variable. If we replace y by $y + 1$, then the following definition is obtained:

```
function $g(x)$;
  begin integer $y$; $y := x - 1$;
    repeat if $a[y] = c$ then $y := \sigma(y)$
                        else $y := g(y)$
    until $a[y + 1] = a[x]$;
    return $y + 1$
  end
```

We can even go one stage further. Notice that $a[x] = a[g(x)]$; in particular, $a[x] = c$ if and only if $a[g(x)] = c$. Now we certainly have $a[\sigma(x)] = c$ by the definition of σ, so we can remove the test $a[y] = c$ from the body of the loop and obtain the following more efficient version:

```
function $g(x)$;
```

```
begin integer y; y := x − 1;
   if a[y] = c then repeat y := σ(y) until a[y + 1] = a[x]
              else repeat y := g(y) until a[y + 1] = a[x];
   return y + 1
end
```

As in the previous section, the next problem to be faced is how to compute the values of $g(x)$ for $1 \leq x \leq N$ by a faster process than simply using the definition of g. Once again, the answer is to employ tabulation, whereby the values of g are computed once only and stored in a table for subsequent use. Tabulation can be implemented directly, but it is instructive to see how a more efficient version goes. What we need is the interesting result that g in fact defines a *permutation* of $1, 2, \ldots, N$, i.e.

(a) $1 \leq g(x) \leq N$, for $1 \leq x \leq N$, and

(b) $g(x) = g(y)$ implies $x = y$, for $1 \leq x, y \leq N$.

Of course this property of g must be proved solely from the given definition. We outline only how to prove (b) since (a) is reasonably straightforward. The proof of (b) is instructive because it illustrates the use of a general induction technique which deserves to be more widely appreciated. This is the method of *computational induction*, which is described fully in Morris [11], Manna [10], and Bird [4].

Briefly stated, the idea is to consider an infinite sequence of partial functions $g_0\ g_1, \ldots$ which represent, in a suitable sense, "approximations" to the "limit" function g. We can then prove any property of g by proving it for each approximation function and passing to the limit. The sequence of approximations is defined by taking g_0 to be the everywhere undefined function, and defining g_{k+1} by exactly the same definition as g, except that recursive calls to g are replaced by calls to g_k. In the present case, this simply means that we replace $y := g(y)$ by $y := g_k(y)$ in the definition of g. The result we wish to prove is the following:

(c) for all $k \geq 0$ and all x and y, if $g_k(x)$ and $g_k(y)$ are defined and $g_k(x) = g_k(y)$, then $x = y$.

The proof is by induction on k. The case $k = 0$ is vacuously true since neither $g_0(x)$ nor $g_0(y)$ can possibly take defined values. For reasons of space we shall not give the details of the straightforward induction step except to say that the proof divides into three cases depending on whether the symbols $a[x - 1]$ and $a[y - 1]$ are both equal to c, both different from c, or otherwise.

Now that we know g is a permutation of $1, 2, \ldots, N$, the tabulation of g can be carried out efficiently in an order determined by the cycle structure of g. More precisely, we can compute the table G which holds the values of g by the following scheme:

```
x := 1;
while x ≤ N do
begin 1. Compute the entries G[x], G[G[x]], . . . etc., stopping
            when we reach an n such that x = G^n[x];
      2. Increment x to the first uncalculated value of G[x]
end
```

This scheme can be refined into the following program.

```
 1.  array G[1: N + 1]; clear G;
 2.  x := 1;
 3.  while x ≤ N do
 4.     begin y := z := x − 1;
 5.        repeat if a[y] = c then repeat y := σ(y)
                                until a[y + 1] = a[x]
 6.                         else repeat y := G[y]
                                until a[y + 1] = a[x];
 7.           G[x] := y + 1;
 8.           x := G[x]
 9.        until y = z;
10.        repeat x := x + 1 until G[x] = 0
11.     end
```

In this program the operation **clear** G sets every entry of G to zero. The element $G[N + 1] = 0$ serves to terminate the search for new uncalculated values as soon as $x = N + 1$.

In order to time the program, let n be the total number of patterns in the input string, k be the number of cycles of the permutation g, and s the size of the alphabet of symbols over which the input string is defined. We estimate the total contributions of each line of the program as follows:

Line 1. $(N + 1)$ units to clear the array.

Line 2. 1 unit.

Lines 3, 4. $(3k + 1)$ units, as there are k cycles and so k executions of the loop.

Lines 5 to 9. Less than $(6N + 2ns)$ units. Note first that the instruction $G[x] := y + 1$ is executed exactly N times; this means the body of the outer loop is executed N times and so contributes $4N$ units. The loop in line 6 contributes at most $2N$ units since the instruction $y := G[y]$ cannot be executed more than N times. The contribution of $2\ ns$ units from line 5 is more difficult to see, but a little thought shows that the instruction $y := \sigma(y)$ (which we can count as 1 unit as σ can itself be easily tabulated) is executed exactly n times for each distinct symbol in the first position of a pattern, and there may be as many as s such symbols.

Line 10. $2N$ units. To see this, observe that the final value of x on exit from the **repeat** loop is just the same as it was on entry. Thus the instruction $x := x + 1$ in line 10 executed exactly N times.

Since there may be as many as N patterns, i.e. $n \leq N$, the running time $T(N)$ of the tabulation of G satisfies

$$T(N) \leq 9N + 3 + 2ns \leq 2sN + 9N + 3.$$

Notice that, although $T(N) = O(N)$, the constant of proportionality is dependent on the size of the input alphabet. It is straightforward to verify that the running time of Algorithm 4 is $O(N)$, as is the necessary tabulation of σ. Algorithm 4 amounts to the process of checking whether or not $G[j] = j$ for each j such that $a[j] = c$. In fact, table G gives added information about the input since the patterns x_j and x_k are identical just in the case that $G[\sigma^j(0)] = \sigma^k(0)$. We have therefore arrived at the desired linear algorithm for the original problem.

This example has been treated at some length because of the many fascinating issues involved, and it is worthwhile to summarize the main steps. First, we employed the idea of introducing recursion to eliminate the stack operations. Then we manipulated and simplified the resulting recursive definitions to reveal, in turn, the recursive functions f and g. Next we looked more closely at g, saw that it was a permutation, and used this fact in an efficient tabulation of g. Finally, we analyzed the resulting algorithm to establish its $O(N)$ running time.

4. Conclusions

We are gradually learning about program manipulation, but a lot remains to be discovered. It is surprising how often programs can be optimized, constants of proportionality can be reduced, even asymptotic growth rates can sometimes be lowered. As Knuth observes, it is important that these optimizations are carried out in the source language of the algorithm, and therefore subject to the programmer's imagination and control. The manipulations described in the present paper mirror very closely the style of derivation of mathematical formulas, and we did not start out, as no mathematician ever does, with the preconception that such derivations should be described with a view to immediate mechanization; such a view would severely limit the many ways in which an algorithm can be simplified and polished. As the length of the derivations testify, we still lack a convenient shorthand with which to describe programs, but this will come with a deeper understanding about the right sequencing mechanisms.

The particular techniques of program optimization described in this paper can be viewed as an application of Cook's theorem on two-way deterministic pushdown automata. Indeed, it appears that Cook's theorem amounts to a statement, couched in automata-theoretic language, on the power of tabulation as an optimization method. If this is the case, then it seems that the essential ideas are revealed more clearly in the ordinary language of computer science. Automata theory may be a useful vehicle for formalizing our knowledge of programs, but this is a debatable point. What is not in doubt is that theory and practice have to be brought closer together for computer science to advance.

Received January 1976; revised October 1976

References

1. Aho, A.V., Hopcroft, J.E., Ullman, J.D. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, Mass., 1974.
2. Bird, R.S. Speeding up programs. *Computer J. 17,* 4 (1975), 337–339.
3. Bird, R.S. Notes on recursion elimination. To appear in *Comm. ACM.*
4. Bird, R.S. *Programs and Machines—An Introduction to the Theory of Computation.* John Wiley, London, 1976.
5. Brown, S., Gries, D., and Szymanski, T. Program schemes with pushdown stoves. *SIAM J. Comptng.* 1 (1972), 242–268.
6. Chandra, A.K. On the properties and application of program schemes. Ph.D. Th., Rep. No. CS-336 Comptr. Sci. Dept., Stanford U., Stanford, Calif., 1973.
7. Darlington, J., and Burstall, R.M. A system which automatically improves programs. Proc. 3rd Int. Conf. on Artif. Intell., Stanford U., Stanford, Calif., 1973, pp. 479–485.
8. Knuth, D.E. Structured programming with go to statements. *Computing Surveys 6,* 4 (Dec. 1974), 261–302.
9. Knuth, D.E., Morris, J.H., Jr., and Pratt, V.R. Fast pattern matching in strings. Res. Rep. CS-74-440, Comptr. Sci. Dept., Stanford, U., Stanford, Calif., 1974.
10. Manna, Z. *Mathematical Theory of Computation.* McGraw-Hill, New York, 1974.
11. Morris, J.H. Jr. Another recursion induction principle. *Comm. ACM 14,* 5 (May 1971), 351–354.