SIAM J. COMPUT.
Vol. 6, No. 2, June 1977

FAST PATTERN MATCHING IN STRINGS*

DONALD E. KNUTH†, JAMES H. MORRIS, JR.‡ AND VAUGHAN R. PRATT¶

Abstract. An algorithm is presented which finds all occurrences of one given string within another, in running time proportional to the sum of the lengths of the strings. The constant of proportionality is low enough to make this algorithm of practical use, and the procedure can also be extended to deal with some more general pattern-matching problems. A theoretical application of the algorithm shows that the set of concatenations of even palindromes, i.e., the language $\{\alpha\alpha^R\}^*$, can be recognized in linear time. Other algorithms which run even faster on the average are also considered.

Key words. pattern, string, text-editing, pattern-matching, trie memory, searching, period of a string, palindrome, optimum algorithm, Fibonacci string, regular expression

Text-editing programs are often required to search through a string of characters looking for instances of a given "pattern" string; we wish to find all positions, or perhaps only the leftmost position, in which the pattern occurs as a contiguous substring of the text. For example, *c a t e n a r y* contains the pattern *t e n*, but we do not regard *c a n a r y* as a substring.

The obvious way to search for a matching pattern is to try searching at every starting position of the text, abandoning the search as soon as an incorrect character is found. But this approach can be very inefficient, for example when we are looking for an occurrence of *a a a a a a a b* in *a a a a a a a a a a a a a a b*. When the pattern is a^nb and the text is $a^{2n}b$, we will find ourselves making $(n+1)^2$ comparisons of characters. Furthermore, the traditional approach involves "backing up" the input text as we go through it, and this can add annoying complications when we consider the buffering operations that are frequently involved.

In this paper we describe a pattern-matching algorithm which finds all occurrences of a pattern of length m within a text of length n in $O(m+n)$ units of time, without "backing up" the input text. The algorithm needs only $O(m)$ locations of internal memory if the text is read from an external file, and only $O(\log m)$ units of time elapse between consecutive single-character inputs. All of the constants of proportionality implied by these "O" formulas are independent of the alphabet size.

* Received by the editors August 29, 1974, and in revised form April 7, 1976.

† Computer Science Department, Stanford University, Stanford, California 94305. The work of this author was supported in part by the National Science Foundation under Grant GJ 36473X and by the Office of Naval Research under Contract NR 044-402.

‡ Xerox Palo Alto Research Center, Palo Alto, California 94304. The work of this author was supported in part by the National Science Foundation under Grant GP 7635 at the University of California, Berkeley.

¶ Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts 02139. The work of this author was supported in part by the National Science Foundation under Grant GP-6945 at University of California, Berkeley, and under Grant GJ-992 at Stanford University.

We shall first consider the algorithm in a conceptually simple but somewhat inefficient form. Sections 3 and 4 of this paper discuss some ways to improve the efficiency and to adapt the algorithm to other problems. Section 5 develops the underlying theory, and § 6 uses the algorithm to disprove the conjecture that a certain context-free language cannot be recognized in linear time. Section 7 discusses the origin of the algorithm and its relation to other recent work. Finally, § 8 discusses still more recent work on pattern matching.

1. Informal development. The idea behind this approach to pattern matching is perhaps easiest to grasp if we imagine placing the pattern over the text and sliding it to the right in a certain way. Consider for example a search for the pattern *a b c a b c a c a b* in the text *b a b c b a b c a b c a a b c a b c a b c a c a b c*; initially we place the pattern at the extreme left and prepare to scan the leftmost character of the input text:

```
a b c a b c a c a b
b a b c b a b c a b c a a b c a b c a b c a c a b c
↑
```

The arrow here indicates the current text character; since it points to *b*, which doesn't match that *a*, we shift the pattern one space right and move to the next input character:

```
  a b c a b c a c a b
b a b c b a b c a b c a a b c a b c a b c a c a b c
  ↑
```

Now we have a match, so the pattern stays put while the next several characters are scanned. Soon we come to another mismatch:

```
  a b c a b c a c a b
b a b c b a b c a b c a a b c a b c a b c a c a b c
        ↑
```

At this point we have matched the first three pattern characters but not the fourth, so we know that the last four characters of the input have been *a b c x* where $x \neq a$; we don't have to remember the previously scanned characters, since *our position in the pattern yields enough information to recreate them.* In this case, no matter what x is (as long as it's not a), we deduce that the pattern can immediately be shifted four more places to the right; one, two, or three shifts couldn't possibly lead to a match.

Soon we get to another partial match, this time with a failure on the eighth pattern character:

```
          a b c a b c a c a b
b a b c b a b c a b c a a b c a b c a b c a c a b c
                        ↑
```

Now we know that the last eight characters were *a b c a b c a x*, where $x \neq c$. The pattern should therefore be shifted three places to the right:

```
                a b c a b c a c a b
b a b c b a b c a b c a a b c a b c a b c a c a b c
                        ↑
```

We try to match the new pattern character, but this fails too, so we shift the pattern four (not three or five) more places. That produces a match, and we continue scanning until reaching *another* mismatch on the eighth pattern character:

```
                        a b c a b c a c a b
b a b c b a b c a b c a a b c a b c a b c a c a b c
                                      ↑
```

Again we shift the pattern three places to the right; this time a match is produced, and we eventually discover the full pattern:

```
                              a b c a b c a c a b
b a b c b a b c a b c a a b c a b c a b c a c a b c
                                                ↑
```

The play-by-play description for this example indicates that the pattern-matching process will run efficiently if we have an auxiliary table that tells us exactly how far to slide the pattern, when we detect a mismatch at its jth character *pattern*$[j]$. Let *next*$[j]$ be the character position in the pattern which should be checked next after such a mismatch, so that we are sliding the pattern $j - next[j]$ places relative to the text. The following table lists the appropriate values:

$j=$	1	2	3	4	5	6	7	8	9	10
pattern$[j]=$	*a*	*b*	*c*	*a*	*b*	*c*	*a*	*c*	*a*	*b*
next$[j]=$	0	1	1	0	1	1	0	5	0	1

(Note that *next*$[j]=0$ means that we are to slide the pattern all the way *past* the current text character.) We shall discuss how to precompute this table later; fortunately, the calculations are quite simple, and we will see that they require only $O(m)$ steps.

At each step of the scanning process, we move either the text pointer or the pattern, and each of these can move at most n times; so at most $2n$ steps need to be performed, after the *next* table has been set up. Of course the pattern itself doesn't really move; we can do the necessary operations simply by maintaining the pointer variable j.

2. Programming the algorithm. The pattern-match process has the general form

```
place pattern at left;
while pattern not fully matched
    and text not exhausted do
    begin
        while pattern character differs from
              current text character
              do shift pattern appropriately;
         advance to next character of text;
     end;
```

For convenience, let us assume that the input text is present in an array *text*$[1:n]$, and that the pattern appears in *pattern*$[1:m]$. We shall also assume that $m>0$, i.e., that the pattern is nonempty. Let k and j be integer variables such that *text*$[k]$ denotes the current text character and *pattern*$[j]$ denotes the corresponding pattern character; thus, the pattern is essentially aligned with positions $p+1$ through $p+m$ of the text, where $k=p+j$. Then the above program takes the following simple form:

```
j := k := 1;
while j ≦ m and k ≦ n do
    begin
        while j > 0 and text[k] ≠ pattern[j]
              do j := next[j];
        k := k+1; j := j+1;
    end;
```

If $j>m$ at the conclusion of the program, the leftmost match has been found in positions $k-m$ through $k-1$; but if $j \leqq m$, the text has been exhausted. (The **and** operation here is the "conditional and" which does not evaluate the relation *text*$[k] \neq$ *pattern*$[j]$ unless $j>0$.) The program has a curious feature, namely that the inner loop operation "$j :=$ *next*$[j]$" is performed no more often than the outer loop operation "$k := k+1$"; in fact, the inner loop is usually performed somewhat *less* often, since the pattern generally moves right less frequently than the text pointer does.

To prove rigorously that the above program is correct, we may use the following invariant relation: "Let $p=k-j$ (i.e., the position in the text just preceding the first character of the pattern, in our assumed alignment). Then we have *text*$[p+i]=$*pattern*$[i]$ for $1 \leqq i<j$ (i.e., we have matched the first $j-1$ characters of the pattern, if $j>0$); but for $0 \leqq t<p$ we have *text*$[t+i] \neq$ *pattern*$[i]$ for some i, where $1 \leqq i \leqq m$ (i.e., there is no possible match of the entire pattern to the left of p)."

The program will of course be correct only if we can compute the *next* table so that the above relation remains invariant when we perform the operation $j :=$ *next*$[j]$. Let us look at that computation now. When the program sets

$j := next[j]$, we know that $j > 0$, and that the last j characters of the input up to and including $text[k]$ were

$$pattern[1] \ldots pattern[j-1]\, x$$

where $x \neq pattern[j]$. What we want is to find the least amount of shift for which these characters can possibly match the shifted pattern; in other words, we want $next[j]$ to be the largest i less than j such that the last i characters of the input were

$$pattern[1] \ldots pattern[i-1]\, x$$

and $pattern[i] \neq pattern[j]$. (If no such i exists, we let $next[j] = 0$.) With this definition of $next[j]$ it is easy to verify that $text[t+1] \ldots text[k] \neq pattern[1] \ldots pattern[k-1]$ for $k - j \leqq t < k - next[j]$; hence the stated relation is indeed invariant, and our program is correct.

Now we must face up to the problem we have been postponing, the task of calculating $next[j]$ in the first place. This problem would be easier if we didn't require $pattern[i] \neq pattern[j]$ in the definition of $next[j]$, so we shall consider the easier problem first. Let $f[j]$ be the largest i less than j such that $pattern[1] \ldots pattern[i-1] = pattern[j-i+1] \ldots pattern[j-1]$; since this condition holds vacuously for $i = 1$, we always have $f[j] \geqq 1$ when $j > 1$. By convention we let $f[1] = 0$. The pattern used in the example of § 1 has the following f table:

$j=$	1	2	3	4	5	6	7	8	9	10
$pattern[j]=$	a	b	c	a	b	c	a	c	a	b
$f[j]=$	0	1	1	1	2	3	4	5	1	2.

If $pattern[j] = pattern[f[j]]$ then $f[j+1] = f[j]+1$; but if not, we can use essentially the same pattern-matching algorithm as above to compute $f[j+1]$, with $text = pattern$! (Note the similarity of the $f[j]$ problem to the invariant condition of the matching algorithm. Our program calculates the largest j less than or equal to k such that $pattern[1] \ldots pattern[j-1] = text[k-j+1] \ldots text[k-1]$, so we can transfer the previous technology to the present problem.) The following program will compute $f[j+1]$, assuming that $f[j]$ and $next[1], \ldots, next[j-1]$ have already been calculated:

```
t := f[j];
while t > 0 and pattern[j] ≠ pattern[t]
    do t := next[t];
f[j+1] := t+1;
```

The correctness of this program is demonstrated as before; we can imagine two copies of the pattern, one sliding to the right with respect to the other. For example, suppose we have established that $f[8] = 5$ in the above case; let us consider the computation of $f[9]$. The appropriate picture is

```
        a b c a b c a c a b
  a b c a b c a c a b
                ↑
```

Since $pattern[8] \neq b$, we shift the upper copy right, knowing that the most recently scanned characters of the lower copy were $a\,b\,c\,a\,x$ for $x \neq b$. The *next* table tells us to shift right four places, obtaining

$$\begin{array}{ccccccccccccc} & & & & a & b & c & a & b & c & a & c & a\ b \\ a & b & c & a & b & c & a & c & a\ b & & & \\ & & & & & & & \uparrow & & & & & \end{array}$$

and again there is no match. The next shift makes $t = 0$, so $f[9] = 1$.

Once we understand how to compute f, it is only a short step to the computation of $next[j]$. A comparison of the definitions shows that, for $j > 1$,

$$next[j] = \begin{cases} f[j], & \text{if } pattern[j] \neq pattern[f[j]]; \\ next[f[j]], & \text{if } pattern\ [j] = pattern[f[j]]. \end{cases}$$

Therefore we can compute the *next* table as follows, without ever storing the values of $f[j]$ in memory.

```
j := 1; t := 0; next[1] := 0;
while j < m do
    begin comment t = f[j];
        while t > 0 and pattern[j] ≠ pattern[t]
            do t := next[t];
        t := t+1; j := j+1;
        if pattern[j] = pattern[t]
        then next[j] := next[t]
        else next[j] := t;
    end.
```

This program takes $O(m)$ units of time, for the same reason as the matching program takes $O(n)$: the operation $t := next[t]$ in the innermost loop always shifts the upper copy of the pattern to the right, so it is performed a total of m times at most. (A slightly different way to prove that the running time is bounded by a constant times m is to observe that the variable t starts at 0 and it is increased, $m-1$ times, by 1; furthermore its value remains nonnegative. Therefore the operation $t := next[t]$, which always decreases t, can be performed at most $m-1$ times.)

To summarize what we have said so far: Strings of text can be scanned efficiently by making use of two ideas. We can precompute "shifts", specifying how to move the given pattern when a mismatch occurs at its jth character; and this precomputation of "shifts" can be performed efficiently by using the same principle, shifting the pattern against itself.

3. Gaining efficiency. We have presented the pattern-matching algorithm in a form that is rather easily proved correct; but as so often happens, this form is not very efficient. In fact, the algorithm as presented above would probably not be competitive with the naive algorithm on realistic data, even though the naive algorithm has a worst-case time of order m times n instead of m plus n, because the chance of this worst case is rather slim. On the other hand, a well-implemented form of the new algorithm should go noticeably faster because there is no backing up after a partial match.

It is not difficult to see the source of inefficiency in the new algorithm as presented above: When the alphabet of characters is large, we will rarely have a partial match, and the program will waste a lot of time discovering rather awkwardly that $text[k] \neq pattern[1]$ for $k = 1, 2, 3, \ldots$. When $j = 1$ and $text[k] \neq pattern[1]$, the algorithm sets $j := next[1] = 0$, then discovers that $j = 0$, then increases k by 1, then sets j to 1 again, then tests whether or not 1 is $\leqq m$, and later it tests whether or not 1 is greater than 0. Clearly we would be much better off making $j = 1$ into a special case.

The algorithm also spends unnecessary time testing whether $j > m$ or $k > n$. A fully-matched pattern can be accounted for by setting $pattern[m+1] =$ "@" for some impossible character @ that will never be matched, and by letting $next[m+1] = -1$; then a test for $j < 0$ can be inserted into a less-frequently executed part of the code. Similarly we can for example set $text[n+1] =$ "$\perp$" (another impossible character) and $text[n+2] = pattern[1]$, so that the test for $k > n$ needn't be made very often. (See [17] for a discussion of such more or less mechanical transformations on programs.)

The following form of the algorithm incorporates these refinements.

```
          $a := pattern[1]$;
          $pattern[m+1] :=$ '@' ; $next[m+1] := -1$;
          $text[n+1] :=$ '$\perp$' ; $text[n+2] := a$;
          $j := k := 1$;
      get started: **comment** $j = 1$;
          **while** $text[k] \neq a$ **do** $k := k+1$;
          **if** $k > n$ **then go to** input exhausted;
      char matched: $j := j+1$; $k := k+1$;
      loop: **comment** $j > 0$;
          **if** $text[k] = pattern[j]$ **then go to** char matched;
          $j := next[j]$;
          **if** $j = 1$ **then go to** get started;
          **if** $j = 0$ **then**
              **begin**
                  $j := 1$; $k := k+1$;
                  **go to** get started;
              **end**;
          **if** $j > 0$ **then go to** loop;
      **comment** $text[k-m]$ through $text[k-1]$ matched;
```

This program will usually run faster than the naive algorithm; the worst case occurs when trying to find the pattern $a\ b$ in a long string of a's. Similar ideas can be used to speed up the program which prepares the *next* table.

In a text-editor the patterns are usually short, so that it is most efficient to translate the pattern directly into machine-language code which implicitly contains the *next* table (cf. [3, Hack 179] and [24]). For example, the pattern in § 1 could be compiled into the machine-language equivalent of

```
L0:  k := k+1;
L1:  if text[k] ≠ a then go to L0;
     k := k+1;
     if k > n then go to input exhausted;
L2:  if text[k] ≠ b then go to L1;
     k := k+1;
L3:  if text[k] ≠ c then go to L1;
     k := k+1;
L4:  if text[k] ≠ a then go to L0;
     k := k+1;
L5:  if text[k] ≠ b then go to L1;
     k := k+1;
L6:  if text[k] ≠ c then go to L1;
     k := k+1;
L7:  if text[k] ≠ a then go to L0;
     k := k+1;
L8:  if text[k] ≠ c then go to L5;
     k := k+1;
L9:  if text[k] ≠ a then go to L0;
     k := k+1;
L10: if text[k] ≠ b then go to L1;
     k := k+1;
```

This will be slightly faster, since it essentially makes a special case for *all* values of j.

It is a curious fact that people often think the new algorithm will be slower than the naive one, even though it does less work. Since the new algorithm is conceptually hard to understand at first, by comparison with other algorithms of the same length, we feel somehow that a computer will have conceptual difficulties too—we expect the machine to run more slowly when it gets to such subtle instructions!

4. Extensions. So far our programs have only been concerned with finding the leftmost match. However, it is easy to see how to modify the routine so that all matches are found in turn: We can calculate the *next* table for the extended pattern of length $m+1$ using $\mathit{pattern}[m+1] =$ "@", and then we set $\mathit{resume} := \mathit{next}[m+1]$ before setting $\mathit{next}[m+1]$ to -1. After finding a match and doing whatever action is desired to process that match, the sequence

$$j := \mathit{resume}; \textbf{go to } \text{loop};$$

will restart things properly. (We assume that *text* has not changed in the meantime. Note that *resume* cannot be zero.)

Another approach would be to leave $\mathit{next}[m+1]$ untouched, never changing it to -1, and to define integer arrays $\mathit{head}[1:m]$ and $\mathit{link}[1:n]$ initially zero, and to insert the code

$$\mathit{link}[k] := \mathit{head}[j]; \mathit{head}[j] := k;$$

at label "char matched". The test "**if** $j>0$ **then**" is also removed from the program. This forms linked lists for $1 \leq j \leqq m$ of all places where the first j characters of the pattern (but no more than j) are matched in the input.

Still another straightforward modification will find the longest initial match of the pattern, i.e., the maximum j such that *pattern*[1] . . . *pattern*[j] occurs in *text*.

In practice, the text characters are often packed into words, with say b characters per word, and the machine architecture often makes it inconvenient to access individual characters. When efficiency for large n is important on such machines, one alternative is to carry out b independent searches, one for each possible alignment of the pattern's first character in the word. These searches can treat *entire words* as "supercharacters", with appropriate masking, instead of working with individual characters and unpacking them. Since the algorithm we have described does not depend on the size of the alphabet, it is well suited to this and similar alternatives.

Sometimes we want to match two or more patterns in sequence, finding an occurrence of the first followed by the second, etc.; this is easily handled by consecutive searches, and the total running time will be of order n plus the sum of the individual pattern lengths.

We might also want to match two or more patterns in parallel, stopping as soon as any one of them is fully matched. A search of this kind could be done with multiple *next* and *pattern* tables, with one j pointer for each; but this would make the running time kn plus the sum of the pattern lengths, when there are k patterns. Hopcroft and Karp have observed (unpublished) that our pattern-matching algorithm can be extended so that the running time for simultaneous searches is proportional simply to n, plus the alphabet size times the sum of the pattern lengths. The patterns are combined into a "trie" whose nodes represent all of the initial substrings of one or more patterns, and whose branches specify the appropriate successor node as a function of the next character in the input text. For example, if there are four patterns $\{a\,b\,c\,a\,b,\ a\,b\,a\,b\,c,\ b\,c\,a\,c,\ b\,b\,c\}$, the trie is shown in Fig. 1.

node	substring	**if** *a*	**if** *b*	**if** *c*
0		1	7	0
1	*a*	1	2	0
2	*a b*	5	10	3
3	*a b c*	4	7	0
4	*a b c a*	1	*a b c a b*	*b c a c*
5	*a b a*	1	6	0
6	*a b a b*	5	10	*a b a b c*
7	*b*	1	10	8
8	*b c*	9	7	0
9	*b c a*	1	2	*b c a c*
10	*b b*	1	10	*b b c*

FIG. 1

Such a trie can be constructed efficiently by generalizing the idea we used to calculate *next*[j]; details and further refinements have been discussed by Aho and Corasick [2], who discovered the algorithm independently. (Note that this algorithm depends on the alphabet size; such dependence is inherent, if we wish to keep the coefficient of n independent of k, since for example the k patterns might each consist of a single unique character.) It is interesting to compare this approach to what happens when the LR(0) parsing algorithm is applied to the regular grammar $S \to a\,S \mid b\,S \mid c\,S \mid a\,b\,c\,a\,b \mid a\,b\,a\,b\,c \mid b\,c\,a\,c \mid b\,b\,c$.

5. Theoretical considerations. If the input file is being read in "real time", we might object to long delays between consecutive inputs. In this section we shall prove that the number of times $j := next[j]$ is performed, before k is advanced, is bounded by a function of the approximate form $\log_\phi m$, where $\phi = (1+\sqrt{5})/2 \approx 1.618\ldots$ is the golden ratio, and that this bound is best possible. We shall use lower case Latin letters to represent characters, and lower case Greek letters $\alpha, \beta, \ldots$ to represent strings, with ε the empty string and $|\alpha|$ the length of α. Thus $|a| = 1$ for all characters a; $|\alpha\beta| = |\alpha| + |\beta|$; and $|\varepsilon| = 0$. We also write $\alpha[k]$ for the kth character of α, when $1 \leqq k \leqq |\alpha|$.

As a warmup for our theoretical discussion, let us consider the *Fibonacci strings* [14, exercise 1.2.8–36], which turn out to be especially pathological patterns for the above algorithm. The definition of Fibonacci strings is

(1) $$\phi_1 = b, \quad \phi_2 = a; \quad \phi_n = \phi_{n-1}\phi_{n-2} \quad \text{for } n \geqq 3.$$

For example, $\phi_3 = a\,b$, $\phi_4 = a\,b\,a$, $\phi_5 = a\,b\,a\,a\,b$. It follows that the length $|\phi_n|$ is the nth Fibonacci number F_n, and that ϕ_n consists of the first F_n characters of an infinite string ϕ_∞ when $n \geqq 2$.

Consider the pattern ϕ_8, which has the functions $f[j]$ and $next[j]$ shown in Table 1.

TABLE 1

j =	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
$pattern[j]$ =	a	b	a	a	b	a	b	a	a	b	a	a	b	a	b	a	a	b	a	b	a
$f[j]$ =	0	1	1	2	2	3	4	3	4	5	6	7	5	6	7	8	9	10	11	12	8
$next[j]$ =	0	1	0	2	1	0	4	0	2	1	0	7	1	0	4	0	2	1	0	12	0

If we extend this pattern to ϕ_∞, we obtain infinite sequences $f[j]$ and $next[j]$ having the same general character. It is possible to prove by induction that

(2) $$f[j] = j - F_{k-1} \quad \text{for } F_k \leqq j < F_{k+1},$$

because of the following remarkable near-commutative property of Fibonacci strings:

(3) $$\phi_{n-2}\phi_{n-1} = c(\phi_{n-1}\phi_{n-2}), \quad \text{for } n \geqq 3,$$

where $c(\alpha)$ denotes changing the two rightmost characters of α. For example, $\phi_6 = a\,b\,a\,a\,b \cdot a\,b\,a$ and $c(\phi_6) = a\,b\,a \cdot a\,b\,a\,a\,b$. Equation (3) is obvious when $n = 3$; and for $n > 3$ we have $c(\phi_{n-2}\phi_{n-1}) = \phi_{n-2}c(\phi_{n-1}) = \phi_{n-2}\phi_{n-3}\phi_{n-2} = \phi_{n-1}\phi_{n-2}$ by induction; hence $c(\phi_{n-2}\phi_{n-1}) = c(c(\phi_{n-1}\phi_{n-2})) = \phi_{n-1}\phi_{n-2}$.

Equation (3) implies that

(4) $$next[F_k - 1] = F_{k-1} - 1 \quad \text{for } k \geqq 3.$$

Therefore if we have a mismatch when $j = F_8 - 1 = 20$, our algorithm might set $j := next[j]$ for the successive values 20, 12, 7, 4, 2, 1, 0 of j. Since F_k is $(\phi^k/\sqrt{5})$ rounded to the nearest integer, it is possible to have up to $\sim \log_\phi m$ consecutive iterations of the $j := next[j]$ loop.

We shall now show that Fibonacci strings actually are the worst case, i.e., that $\log_\phi m$ is also an upper bound. First let us consider the concept of *periodicity* in strings. We say that p is a *period* of α if

$$(5) \qquad \alpha[i] = \alpha[i+p] \quad \text{for } 1 \leqq i \leqq |\alpha| - p.$$

It is easy to see that p is a period of α if and only if

$$(6) \qquad \alpha = (\alpha_1\alpha_2)^k\alpha_1$$

for some $k \geqq 0$, where $|\alpha_1\alpha_2| = p$ and $\alpha_2 \neq \varepsilon$. Equivalently, p is a period of α if and only if

$$(7) \qquad \alpha\theta_1 = \theta_2\alpha$$

for some θ_1 and θ_2 with $|\theta_1| = |\theta_2| = p$. Condition (6) implies (7) with $\theta_1 = \alpha_2\alpha_1$ and $\theta_2 = \alpha_1\alpha_2$. Condition (7) implies (6), for we define $k = \lfloor |\alpha|/p \rfloor$ and observe that if $k > 0$, then $\alpha = \theta_2\beta$ implies $\beta\theta_1 = \theta_2\beta$ and $\lfloor |\beta|/p \rfloor = k - 1$; hence, reasoning inductively, $\alpha = \theta_2^k\alpha_1$ for some α_1 with $|\alpha_1| < p$, and $\alpha_1\theta_1 = \theta_2\alpha_1$. Writing $\theta_2 = \alpha_1\alpha_2$ yields (6).

The relevance of periodicity to our algorithm is clear once we consider what it means to shift a pattern. If $pattern[1] \ldots pattern[j-1] = \alpha$ ends with $pattern[1] \ldots pattern[i-1] = \beta$, we have

$$(8) \qquad \alpha = \beta\theta_1 = \theta_2\beta$$

where $|\theta_1| = |\theta_2| = j - i$, so the amount of shift $j - i$ is a period of α.

The construction of $i = next[j]$ in our algorithm implies further that $pattern[i]$, which is the first character of θ_1, is unequal to $pattern[j]$. Let us assume that β itself is subsequently shifted leaving a residue γ, so that

$$(9) \qquad \beta = \gamma\psi_1 = \psi_2\gamma$$

where the first character of ψ_1 differs from that of θ_1. We shall now prove that

$$(10) \qquad |\alpha| > |\beta| + |\gamma|.$$

If $|\beta| + |\gamma| \geqq |\alpha|$, there is an overlap of $d = |\beta| + |\gamma| - |\alpha|$ characters between the occurrences of β and γ in $\beta\theta_1 = \alpha = \theta_2\psi_2\gamma$; hence the first character of θ_1 is $\gamma[d+1]$. Similarly there is an overlap of d characters between the occurrences of β and γ in $\theta_2\beta = \alpha = \gamma\psi_1\theta_1$; hence the first character of ψ_1 is $\beta[d+1]$. Since these characters are distinct, we obtain $\gamma[d+1] \neq \beta[d+1]$, contradicting (9). This establishes (10), and leads directly to the announced result:

THEOREM. *The number of consecutive times that $j := next[j]$ is performed, while one text character is being scanned, is at most $1 + \log_\phi m$.*

Proof. Let L_r be the length of the shortest string α as in the above discussion such that a sequence of r consecutive shifts is possible. Then $L_1 = 0$, $L_2 = 1$, and we have $|\beta| \geqq L_{r-1}$, $|\gamma| \geqq L_{r-2}$ in (10); hence $L_2 \geqq F_{r+1} - 1$ by induction on r. Now if r shifts occur we have $m \geqq F_{r+1} \geqq \phi^{r-1}$. □

The algorithm of § 2 would run correctly in linear time even if $f[j]$ were used instead of $next[j]$, but the analogue of the above theorem would then be false. For example, the pattern a^n leads to $f[j]=j-1$ for $1 \leqq j \leqq m$. Therefore if we matched a^m to the text $a^{m-1}b\alpha$, using $f[j]$ instead of $next[j]$, the mismatch $text[m] \neq pattern[m]$ would be followed by m occurrences of $j := f[j]$ and $m-1$ redundant comparisons of $text[m]$ with $pattern[j]$, before k is advanced to $m+1$.

The subject of periods in strings has several interesting algebraic properties, but a reader who is not mathematically inclined may skip to § 6 since the following material is primarily an elaboration of some additional structure related to the above theorem.

LEMMA 1. *If p and q are periods of α, and $p+q \leqq |\alpha| + \gcd(p, q)$, then $\gcd(p, q)$ is a period of α.*

Proof. Let $d = \gcd(p, q)$, and assume without loss of generality that $d < p < q = p + r$. We have $\alpha[i] = \alpha[i+p]$ for $1 \leqq i \leqq |\alpha| - p$ and $\alpha[i] = \alpha[i+q]$ for $1 \leqq i \leqq |\alpha| - q$; hence $\alpha[i+r] = \alpha[i+q] = \alpha[i]$ for $1 + r \leqq i + r \leqq |\alpha| - p$, i.e.,

$$\alpha[i] = \alpha[i+r] \quad \text{for } 1 \leqq i \leqq |\alpha| - q.$$

Furthermore $\alpha = \beta\theta_1 = \theta_2\beta$ where $|\theta_1| = p$, and it follows that p and r are periods of β, where $p + r \leqq |\beta| + d = |\beta| + \gcd(p, r)$. By induction, d is a period of β. Since $|\beta| = |\alpha| - p \geqq q - d \geqq q - r = p = |\theta_1|$, the strings θ_1 and θ_2 (which have the respective forms $\beta_2\beta_1$ and $\beta_1\beta_2$ by (6) and (7)) are substrings of β; so they also have d as a period. The string $\alpha = (\beta_1\beta_2)^{k+1}\beta_1$ must now have d as a period, since any characters d positions apart are contained within $\beta_1\beta_2$ or $\beta_1\beta_1$. □

The result of Lemma 1 but with the stronger hypothesis $p + q \leqq |\alpha|$ was proved by Lyndon and Schützenberger in connection with a problem about free groups [19, Lem. 4]. The weaker hypothesis in Lemma 1 turns out to give the best possible bound: If $\gcd(p, q) < p < q$ we can find a string of length $p + q - \gcd(p, q) - 1$ for which $\gcd(p, q)$ is *not* a period. In order to see why this is so, consider first the example in Fig. 2 showing the most general strings of lengths 15 through 25 having both 11 and 15 as periods. (The strings are "most general" in the sense that any two character positions that can be different *are* different.)

```
a b c d e f g h i j k a b c d
a b c d a f g h i j k a b c.d a
a b c d a b g h i j k a b c d a b
a b c d a b c h i j k a b c d a b c
a b c d a b c d i j k a b c d a b c d
a b c d a b c d a j k a b c d a b c d a
a b c d a b c d a b k a b c d a b c d a b
a b c d a b c d a b c a b c d a b c d a b c
a b c a a b c a a b c a b c a a b c a a b c a
a a c a a a c a a a c a a c a a a c a a a c a a
a a a a a a a a a a a a a a a a a a a a a a a a a
```

FIG. 2

Note that the number of degrees of freedom, i.e., the number of distinct symbols, decreases by 1 at each step. It is not difficult to prove that the number cannot decrease by *more* than 1 as we go from $|\alpha| = n - 1$ to $|\alpha| = n$, since the only new relations are $\alpha[n]=\alpha[n-q]=\alpha[n-p]$; we decrease the number of distinct symbols by one if and only if positions $n-q$ and $n-p$ contain distinct symbols in the most general string of length $n-1$. The lemma tells us that we are left with at most $\gcd(p,q)$ symbols when the length reaches $p+q-\gcd(p,q)$; on the other hand we always have exactly p symbols when the length is q. Therefore each of the $p-\gcd(p,q)$ steps *must* decrease the number of symbols by 1, and the most general string of length $p+q-\gcd(p,q)-1$ must have exactly $\gcd(p,q)+1$ distinct symbols. In other words, the lemma gives the best possible bound.

When p and q are relatively prime, the strings of length $p+q-2$ on two symbols, having both p and q as periods, satisfy a number of remarkable properties, generalizing what we have observed earlier about Fibonacci strings. Since the properties of these pathological patterns may prove useful in other investigations, we shall summarize them in the following lemma.

LEMMA 2. *Let the strings $\sigma(m,n)$ of length n be defined for all relatively prime pairs of integers $n \geqq m \geqq 0$ as follows:*

$$\sigma(0,1)=a, \quad \sigma(1,1)=b, \quad \sigma(1,2)=ab;$$

$$(11) \qquad \left.\begin{aligned} \sigma(m,m+n)&=\sigma(n \bmod m, m)\sigma(m,n)\\ \sigma(n,m+n)&=\sigma(m,n)\sigma(n \bmod m, m)\end{aligned}\right\} \quad \textit{if } 0<m<n.$$

These strings satisfy the following properties:

(i) $\sigma(m,qm+r)\sigma(m-r,m)=\sigma(r,m)\sigma(m,qm+r)$, *for* $m>2$;

(ii) $\sigma(m,n)$ *has period* m, *for* $m>1$;

(iii) $c(\sigma(m,n))=\sigma(n-m,n)$, *for* $n>2$.

(The function $c(\alpha)$ was defined in connection with (3) above.)

Proof. We have, for $0<m<n$ and $q \geqq 2$,

$$\sigma(m+n,q(m+n)+m)=\sigma(m,m+n)\,\sigma(m+n,(q-1)(m+n)+m),$$

$$\sigma(m+n,q(m+n)+n)=\sigma(n,m+n)\,\sigma(m+n,(q-1)(m+n)+n),$$

$$\sigma(m+n,2m+n)=\sigma(m,m+n)\,\sigma(n \bmod m,m),$$

$$\sigma(m+n,m+2n)=\sigma(n,m+n)\,\sigma(m,n);$$

hence, if $\theta_1=\sigma(n \bmod m,m)$ and $\theta_2=\sigma(m,n)$ and $q \geqq 1$,

$$(12) \quad \sigma(m+n,q(m+n)+m)=(\theta_1\theta_2)^q\theta_1, \qquad \sigma(m+n,q(m+n)+n)=(\theta_2\theta_1)^q\theta_2.$$

It follows that

$$\sigma(m+n,q(m+n)+m)\sigma(n,m+n)=\sigma(m,m+n)\,\sigma(m+n,q(m+n)+m),$$

$$\sigma(m+n,q(m+n)+n)\sigma(m,m+n)=\sigma(n,m+n)\,\sigma(m+n,q(m+n)+n),$$

which combine to prove (i). Property (ii) also follows immediately from (12), except for the case $m=2$, $n=2q+1$, $\sigma(2,2q+1)=(ab)^qa$, which may be verified directly. Finally, it suffices to verify property (iii) for $0<m<\frac{1}{2}n$, since $c(c(\alpha))=\alpha$; we must show that

$$c(\sigma(m,m+n))=\sigma(m,n)\,\sigma(n \bmod m,m) \quad \text{for } 0<m<n.$$

When $m \leqq 2$ this property is easily checked, and when $m > 2$ it is equivalent by induction to

$$\sigma(m, m+n) = \sigma(m, n)\,\sigma(m-(n \bmod m), m) \quad \text{for } 0 < m < n, \quad m > 2.$$

Set $n \bmod m = r$, $\lfloor n/m \rfloor = q$, and apply property (i). □

By properties (ii) and (iii) of this lemma, $\sigma(p, p+q)$ minus its last two characters is the string of length $p+q-2$ having periods p and q. Note that Fibonacci strings are just a very special case, since $\phi_n = \sigma(F_{n-1}, F_n)$. Another property of the σ strings appears in [15]. A completely different proof of Lemma 1 and its optimality, and a completely different definition of $\sigma(m, n)$, were given by Fine and Wilf in 1965 [7]. These strings have a long history going back at least to the astronomer Johann Bernoulli in 1772; see [25, § 2.13] and [21].

If α is any string, let $P(\alpha)$ be its shortest period. Lemma 1 implies that all periods q which are not multiples of $P(\alpha)$ must be greater than $|\alpha| - P(\alpha) + \gcd(q, P(\alpha))$. This is a rather strong condition in terms of the pattern matching algorithm, because of the following result.

LEMMA 3. *Let $\alpha = pattern[1] \ldots pattern[j-1]$ and let $a = pattern[j]$. In the pattern matching algorithm, $f[j] = j - P(\alpha)$, and $next[j] = j - q$, where q is the smallest period of α which is not a period of αa. (If no such period exists, $next[j] = 0$.) If $P(\alpha)$ divides $P(\alpha a)$ and $P(\alpha a) < j$, then $P(\alpha) = P(\alpha a)$. If $P(\alpha)$ does not divide $P(\alpha a)$ or if $P(\alpha a) = j$, then $q = P(\alpha)$.*

Proof. The characterizations of $f[j]$ and $next[j]$ follow immediately from the definitions. Since every period of αa is a period of α, the only nonobvious statement is that $P(\alpha) = P(\alpha a)$ whenever $P(\alpha)$ divides $P(\alpha a)$ and $P(\alpha a) \neq j$. Let $P(\alpha) = p$ and $P(\alpha a) = mp$; then the (mp)th character from the right of α is a, as is the $(m-1)p$th, . . . , as is the pth; hence p is a period of αa. □

Lemma 3 shows that the $j := next[j]$ loop will almost always terminate quickly. If $P(\alpha) = P(\alpha a)$, then q must not be a multiple of $P(\alpha)$; hence by Lemma 1, $P(\alpha) + q \geqq j + 1$. On the other hand $q > P(\alpha)$; hence $q > \frac{1}{2}j$ and $next[j] < \frac{1}{2}j$. In the other case $q = P(\alpha)$, we had better not have q too small, since q will be a period in the residual pattern after shifting, and $next[next[j]]$ will be $< q$. To keep the loop running it is necessary for new small periods to keep popping up, relatively prime to the previous periods.

6. Palindromes.

6. Palindromes. One of the most outstanding unsolved questions in the theory of computational complexity is the problem of how long it takes to determine whether or not a given string of length n belongs to a given context-free language. For many years the best upper bound for this problem was $O(n^3)$ in a general context-free language as $n \to \infty$; L. G. Valiant has recently lowered this to $O(n^{\log_2 7})$. On the other hand, the problem isn't known to require more than order n units of time for any particular language. This big gap between $O(n)$ and $O(n^{2.81})$ deserves to be closed, and hardly anyone believes that the final answer will be $O(n)$.

Let Σ be a finite alphabet, let Σ^* denote the strings over Σ, and let

$$P = \{\alpha\alpha^R \mid \alpha \in \Sigma^*\}.$$

Here α^R denotes the reversal of α, i.e., $(a_1 a_2 \ldots a_n)^R = a_n \ldots a_2 a_1$. Each string π in P is a *palindrome* of even length, and conversely every even palindrome over

Σ is in P. At one time it was popularly believed that the language P^* of "even palindromes starred", namely the set of *palstars* $\pi_1 \ldots \pi_n$ where each π_i is in P, would be impossible to recognize in $O(n)$ steps on a random-access computer.

It isn't especially easy to spot members of this language. For example, $a\,a\,b\,b\,a\,b\,b\,a$ is a palstar, but its decomposition into even palindromes might not be immediately apparent; and the reader might need several minutes to decide whether or not

$$b\;a\;a\;b\;b\;a\;b\;b\;a\;a\;b\;a\;b\;b\;a\;a\;b\;b\;a\;b\;b\;a\;b\;a\;a$$
$$b\;b\;a\;b\;b\;a\;b\;b\;a\;b\;b\;a\;a\;b\;a\;b\;a\;b\;b\;a\;b\;b\;a\;a\;b$$

is in P^*. We shall prove, however, that palstars can be recognized in $O(n)$ units of time, by using their algebraic properties.

Let us say that a nonempty palstar is *prime* if it cannot be written as the product of two nonempty palstars. A prime palstar must be an even palindrome $\alpha\alpha^R$ but the converse does not hold. By repeated decomposition, it is easy to see that every palstar β is expressible as a product $\beta_1 \ldots \beta_t$ of prime palstars, for some $t \geqq 0$; what is less obvious is that such a decomposition into prime factors is unique. This "fundamental theorem of palstars" is an immediate consequence of the following basic property.

LEMMA 1. *A prime palstar cannot begin with another prime palstar.*

Proof. Let $\alpha\alpha^R$ be a prime palstar such that $\alpha\alpha^R = \beta\beta^R\gamma$ for some nonempty even palindrome $\beta\beta^R$ and some $\gamma \neq \varepsilon$; furthermore, let $\beta\beta^R$ have minimum length among all such counterexamples. If $|\beta\beta^R| > |\alpha|$ then $\alpha\alpha^R = \beta\beta^R\gamma = \alpha\delta\gamma$ for some $\delta \neq \varepsilon$; hence $\alpha^R = \delta\gamma$, and $\beta\beta^R = (\beta\beta^R)^R = (\alpha\delta)^R = \delta^R\alpha^R = \delta^R\delta\gamma$, contradicting the minimality of $|\beta\beta^R|$. Therefore $|\beta\beta^R| \leqq |\alpha|$; hence $\alpha = \beta\beta^R\delta$ for some δ, and $\beta\beta^R\gamma = \alpha\alpha^R = \beta\beta^R\delta\delta^R\beta\beta^R$. But this implies that γ is the palstar $\delta\delta^R\beta\beta^R$, contradicting the primality of $\alpha\alpha^R$. $\square$

COROLLARY (Left cancellation property.) *If $\alpha\beta$ and α are palstars, so is β.*

Proof. Let $\alpha = \alpha_1 \ldots \alpha_r$ and $\alpha\beta = \beta_1 \ldots \beta_s$ be prime factorizations of α and $\alpha\beta$. If $\alpha_1 \ldots \alpha_r = \beta_1 \ldots \beta_r$, then $\beta = \beta_{r+1} \cdots \beta_s$ is a palstar. Otherwise let j be minimal with $\alpha_j \neq \beta_j$; then α_j begins with β_j or vice versa, contradicting Lemma 1. $\square$

LEMMA 2. *If α is a string of length n, we can determine the length of the longest even palindrome $\beta \in P$ such that $\alpha = \beta\gamma$, in $O(n)$ steps.*

Proof. Apply the pattern-matching algorithm with *pattern* $= \alpha$ and *text* $= \alpha^R$. When $k = n+1$ the algorithm will stop with j maximal such that *pattern*$[1] \ldots$ *pattern* $[j-1] =$ *text*$[n+2-j] \ldots$ *text*$[n]$. Now perform the following iteration:

```
while $j \geqq 3$ and $j$ even do $j := f(j)$.
```

By the theory developed in § 3, this iteration terminates with $j \geqq 3$ if and only if α begins with a nonempty even palindrome, and $j-1$ will be the length of the largest such palindrome. (Note that $f[j]$ must be used here instead of *next*$[j]$; e.g. consider the case $\alpha = a\;a\;b\;a\;a\;b$. But the pattern matching process takes $O(n)$ time even when $f[j]$ is used.) $\square$

THEOREM. *Let L be any language such that L^* has the left cancellation property and such that, given any string α of length n, we can find a nonempty $\beta \in L$*

such that α begins with β or we can prove that no such β exists, in $O(n)$ steps. Then we can determine in $O(n)$ time whether or not a given string is in L^*.

Proof. Let α be any string, and suppose that the time required to test for nonempty prefixes in L is $\leqq Kn$ for all large n. We begin by testing α's initial subsequences of lengths $1, 2, 4, \dots, 2^k, \dots$, and finally α itself, until finding a prefix in L or until establishing that α has no such prefix. In the latter case, α is not in L^*, and we have consumed at most $(K+K_1)+(2K+K_1)+(4K+K_1)+\cdots+(|\alpha|K+K_1)<2Kn+K_1 \log_2 n$ units of time for some constant K_1. But if we find a nonempty prefix $\beta \in L$ where $\alpha=\beta\gamma$, we have used at most $4|\beta|K+K(\log_2|\beta|)$ units of time so far. By the left cancellation property, $\alpha \in L^*$ if and only if $\gamma \in L^*$, and since $|\gamma|=n-|\beta|$ we can prove by induction that at most $(4K+K_1)n$ units of time are needed to decide membership in L^*, when $n>0$. □

COROLLARY. *P^* can be recognized in $O(n)$ time.*

Note that the related language

$$P_1^*=\{\pi \in \Sigma^* \mid \pi=\pi^R \text{ and } |\pi| \geqq 2\}^*$$

cannot be handled by the above techniques, since it contains both *a a a b b b* and *a a a b b b b a*; the fundamental theorem of palstars fails with a vengeance. It is an open problem whether or not P_1^* can be recognized in $O(n)$ time, although we suspect that it can be done.[1] Once the reader has disposed of this problem, he or she is urged to tackle another language which has recently been introduced by S. A. Greibach [11], since the latter language is known to be as hard as possible; no context-free language can be harder to recognize except by a constant factor.

7. Historical remarks.

The pattern-matching algorithm of this paper was discovered in a rather interesting way. One of the authors (J. H. Morris) was implementing a text-editor for the CDC 6400 computer during the summer of 1969, and since the necessary buffering was rather complicated he sought a method that would avoid backing up the text file. Using concepts of finite automata theory as a model, he devised an algorithm equivalent to the method presented above, although his original form of presentation made it unclear that the running time was $O(m+n)$. Indeed, it turned out that Morris's routine was too complicated for other implementors of the system to understand, and he discovered several months later that gratuitous "fixes" had turned his routine into a shambles.

In a totally independent development, another author (D. E. Knuth) learned early in 1970 of S. A. Cook's surprising theorem about two-way deterministic pushdown automata [5]. According to Cook's theorem, any language recognizable by a two-way deterministic pushdown automaton, in *any* amount of time, can be recognized on a random access machine in $O(n)$ units of time. Since D. Chester had recently shown that the set of strings beginning with an even palindrome could be recognized by such an automaton, and since Knuth couldn't imagine how to recognize such a language in less than about n^2 steps on a conventional computer, Knuth laboriously went through all the steps of Cook's construction as applied to Chester's automaton. His plan was to "distill off" what

[1] (Note added April, 1976.) Zvi Galil and Joel Seiferas have recently resolved this conjecture affirmatively.

was happening, in order to discover why the algorithm worked so efficiently. After pondering the mass of details for several hours, he finally succeeded in abstracting the mechanism which seemed to be underlying the construction, in the special case of palindromes, and he generalized it slightly to a program capable of finding the longest prefix of one given string that occurs in another.

This was the first time in Knuth's experience that automata theory had taught him how to solve a real programming problem better than he could solve it before. He showed his results to the third author (V. R. Pratt), and Pratt modified Knuth's data structure so that the running time was independent of the alphabet size. When Pratt described the resulting algorithm to Morris, the latter recognized it as his own, and was pleasantly surprised to learn of the $O(m+n)$ time bound, which he and Pratt described in a memorandum [22]. Knuth was chagrined to learn that Morris had already discovered the algorithm, *without* knowing Cook's theorem; but the theory of finite-state machines had been of use to Morris too, in his initial conceptualization of the algorithm, so it was still legitimate to conclude that automata theory had actually been helpful in this practical problem.

The idea of scanning a string without backing up while looking for a pattern, in the case of a two-letter alphabet, is implicit in the early work of Gilbert [10] dealing with comma-free codes. It also is essentially a special case of Knuth's LR(0) parsing algorithm [16] when applied to the grammar

$$\begin{aligned} &S \to aS, && \text{for each } a \text{ in the alphabet,} \\ &S \to \alpha, \end{aligned}$$

where α is the pattern. Diethelm and Roizen [6] independently discovered the idea in 1971. Gilbert and Knuth did not discuss the preprocessing to build the *next* table, since they were mainly concerned with other problems, and the preprocessing algorithm given by Diethelm and Roizen was of order m^2. In the case of a binary (two-letter) alphabet, Diethelm and Roizen observed that the algorithm of § 3 can be improved further: we can go immediately to "char matched" after $j := next[j]$ in this case if $next[j] > 0$.

A conjecture by R. L. Rivest led Pratt to discover the $\log_\phi m$ upper bound on pattern movements between successive input characters, and Knuth showed that this was best possible by observing that Fibonacci strings have the curious properties proved in § 5. Zvi Galil has observed that a real-time algorithm can be obtained by letting the text pointer move ahead in an appropriate manner while the j pointer is moving down [9].

In his lectures at Berkeley, S. A. Cook had proved that P^* was recognizable in $O(n \log n)$ steps on a random-access machine, and Pratt improved this to $O(n)$ using a preliminary form of the ideas in § 6. The slightly more refined theory in the present version of § 6 is joint work of Knuth and Pratt. Manacher [20] found another way to recognize palindromes in linear time, and Galil [9] showed how to improve this to real time. See also Slisenko [23].

It seemed at first that there might be a way to find the *longest common substring* of two given strings, in time $O(m+n)$; but the algorithm of this paper does not readily support any such extension, and Knuth conjectured in 1970 that such efficiency would be impossible to achieve. An algorithm due to Karp, Miller, and Rosenberg [13] solved the problem in $O((m+n)\log(m+n))$ steps, and this tended to support the conjecture (at least in the mind of its originator). However, Peter Weiner has recently developed a technique for solving the longest common substring problem in $O(m+n)$ units of time with a fixed alphabet, using tree structures in a remarkable new way [26]. Furthermore, Weiner's algorithm has the following interesting consequence, pointed out by E. McCreight: a text file can be processed (in linear time) so that it is possible to determine exactly how much of a pattern is necessary to identify a position in the text uniquely; as the pattern is being typed in, the system can be interrupt as soon as it "knows" what the rest of the pattern must be! Unfortunately the time and space requirements for Weiner's algorithm grow with increasing alphabet size.

If we consider the problem of scanning finite-state languages in general, it is known [1 § 9.2] that the language defined by any regular expression of length m is recognizable in $O(mn)$ units of time. When the regular expression has the form

$$\Sigma^*(\alpha_{1,1}+\cdots+\alpha_{1,s(1)})\Sigma^*(\alpha_{2,1}+\cdots+\alpha_{2,s(2)})\Sigma^* \ldots \Sigma^*(\alpha_{r,1}+\cdots+\alpha_{r,s(r)})\Sigma^*$$

the algorithm we have discussed shows that only $O(m+n)$ units of time are needed (considering Σ^* as a character of length 1 in the expression). Recent work by M. J. Fischer and M. S. Paterson [8] shows that regular expressions of the form

$$\Sigma^*\alpha_1\Sigma\alpha_2\Sigma \ldots \Sigma\alpha_r\Sigma^*,$$

i.e., patterns with "don't care" symbols, can be identified in $O(n \log m \log\log m \log q)$ units of time, where q is the alphabet size and $m = |\alpha_1\alpha_2 \ldots \alpha_r|+r$. The constant of proportionality in their algorithm is extremely large, but the existence of their construction indicates that efficient new algorithms for general pattern matching problems probably remain to be discovered.

A completely different approach to pattern matching, based on hashing, has been proposed by Malcolm C. Harrison [12]. In certain applications, especially with very large text files and short patterns, Harrison's method may be significantly faster than the character-comparing method of the present paper, on the average, although the redundancy of English makes the performance of his method unclear.

8. Postscript: Faster pattern matching in strings.[2]

In the spring of 1974, Robert S. Boyer and J. Strother Moore and (independently) R. W. Gosper noticed that there is an even faster way to match pattern strings, by skipping more rapidly over portions of the text that cannot possibly lead to a match. Their idea was to look first at *text*[m], instead of *text*[1]. If we find that the character *text*[m] does not appear in the pattern at all, we can immediately shift the pattern right m places. Thus, when the alphabet size q is large, we need to inspect only about n/m characters of the text, on the average! Furthermore if *text*[m] does occur in the pattern, we can shift the pattern by the minimum amount consistent with a match.

[2] This postscript was added by D. E. Knuth in March, 1976, because of developments which occurred after preprints of this paper were distributed.

Several interesting variations on this strategy are possible. For example, if *text*$[m]$ does occur in the pattern, we might continue the search by looking at *text*$[m-1]$, *text*$[m-2]$, etc.; in a random file we will usually find a small value of r such that the substring *text*$[m-r] \ldots$ *text*$[m]$ does not appear in the pattern, so we can shift the pattern $m-r$ places. If $r = \lfloor 2 \log_q m \rfloor$, there are more than m^2 possible values of *text*$[m-r] \ldots$ *text*$[m]$, but only $m-r$ substrings of length $r+1$ in the pattern, hence the probability is $O(1/m)$ that *text*$[m-r] \ldots$ *text*$[m]$ occurs in the pattern; If it doesn't, we can shift the pattern right $m-r$ places; but if it does, we can determine all matches in positions $<m-r$ in $O(m)$ steps, shifting the pattern $m-r$ places by the method of this paper. Hence the expected number of characters examined among the first $m - \lfloor 2 \log_q m \rfloor$ is $O(\log_q m)$; this proves the existence of a linear worst-case algorithm which inspects $O(n(\log_q m)/m)$ characters in a random text. This upper bound on the average running time applies to all patterns, and there are some patterns (e.g., a^m or $(a\,b)^{m/2}$) for which the expected number of characters examined by the algorithm is $O(n/m)$.

Boyer and Moore have refined the skipping-by-m idea in another way. Their original algorithm may be expressed as follows using our conventions:

```
$k := m$;
while $k \leqq n$ do
    begin
        $j := m$;
        while $j > 0$ and $text[k] = pattern[j]$ do
            begin
                $j := j - 1$; $k := k - 1$;
            end;
        if $j = 0$ then
            begin
                match found at $(k)$;
                $k := k + m + 1$
            end else
            $k := k + \max(d[text[k]], dd[j])$;
    end;
```

This program calls *match found at* (k) for all $0 \leqq k \leqq n-m$ such that *pattern*$[1] \ldots$ *pattern*$[m] =$ *text*$[k+1] \ldots$ *text*$[k+m]$. There are two precomputed tables, namely

$$d[a] = \min\{s \mid s = m \text{ or } (0 \leqq s < m \text{ and } pattern[m-s] = a)\}$$

for each of the q possible characters a, and

$$dd[j] = \min\{s + m - j \mid s \geqq 1 \text{ and } ((s \geqq 1 \text{ or } pattern[i-s] = pattern[i]) \text{ for } j < i \leqq m)\},$$

for $1 \leqq j \leqq m$.

The d table can clearly be set up in $O(q+m)$ steps, and the dd table can be precomputed in $O(m)$ steps using a technique analogous to the method in § 2 above, as we shall see. The Boyer–Moore paper [4] contains further exposition of the algorithm, including suggestions for highly efficient implementation, and gives both theoretical and empirical analyses. In the remainder of this section we shall show how the above methods can be used to resolve some of the problems left open in [4].

First let us improve the original Boyer–Moore algorithm slightly by replacing $dd[j]$ by

$$\begin{aligned} dd'[j] = \min\{s+m-j \mid\, & s \geqq 1 \text{ and } (s \geqq j \text{ or } pattern[j-s] \neq pattern[j]) \\ & \text{and } ((s \geqq i \text{ or } pattern[i-s] = pattern[i]) \text{ for } j < i \leqq m)\}. \end{aligned}$$

(This is analogous to using $next[j]$ instead of $f[j]$; Boyer and Moore [4] credit the improvement in this case to Ben Kuipers, but they do not discuss how to determine dd' efficiently.) The following program shows how the dd' table can be precomputed in $O(m)$ steps; for purposes of comparison, the program also shows how to compute dd, which actually turns out to require slightly *more* operations than dd':

```
for $k$ := 1 step 1 until $m$ do $dd[k]$ := $dd'[k]$ := $2\times m-k$;
$j$ := $m$; $t$ := $m+1$;
while $j>0$ do
    begin
        $f[j]$ := $t$;
        while $t\leqq m$ and $pattern[j]\neq pattern[t]$ do
            begin
                $dd'[t]$ := min ($dd'[t]$, $m-j$);
                $t$ := $f[t]$;
            end;
        $t$ := $t-1$; $j$ := $j-1$;
        $dd[t]$ := min ($dd[t]$, $m-j$);
    end;
for $k$ := 1 step 1 until $t$ do
    begin
        $dd[k]$ := min ($dd[k]$, $m+t-k$);
        $dd'[k]$ := min ($dd'[k]$, $m+t-k$);
    end;
```

In practice one would, of course, compute only dd', suppressing all references to dd. The example in Table 2 illustrates most of the subtleties of this algorithm.

TABLE 2

$j=$	1	2	3	4	5	6	7	8	9	10	11
$pattern[j]=$	b	a	d	b	a	c	b	a	c	b	a
$f[j]=$	10	11	6	7	8	9	10	11	11	11	12
$dd[j]=$	19	18	17	16	15	8	7	6	5	4	1
$dd'[j]=$	19	18	17	16	15	8	13	12	8	12	1

To prove correctness, one may show first that $f[j]$ is analogous to the $f[j]$ *in* § 2, but with right and left of the pattern reversed; namely $f[m]=m+1$, and for $j<m$ we have

$$f[j]=\min\{i \mid j<i\leqq m \text{ and } pattern[i+1]\dots pattern[m]=pattern[j+1]\dots pattern[m+j-i]\}.$$

Furthermore the final value of t corresponds to $f[0]$ in this definition; $m-t$ is the maximum overlap of the pattern on itself. The correctness of $dd[j]$ and $dd'[j]$ for all j now follows without much difficulty, by showing that the minimum value of s in the definition of $dd[j_0]$ or $dd'[j_0]$ is discovered by the algorithm when $(t,j)=(j_0,j_0-s)$.

The Boyer–Moore algorithm and its variants can have curiously anomalous behavior in unusual circumstances. For example, the method discovers more quickly that the pattern $a\,a\,a\,a\,a\,a\,a\,c\,b$ does not appear in the text $(a\,b)^n$ if it suppresses the d heuristic entirely, i.e., if $d[t]$ is set to $-\infty$ for all t. Likewise, dd actually turns out to be better than dd' when matching $a^{15}\,b\,c\,b\,a\,b\,a\,b$ in $(b\,a\,a\,b\,a\,b)^n$, for large n.

Boyer and Moore showed that their algorithm has quadratic behavior in the worst case; the running time can be essentially proportional to pattern length times text length, for example when the pattern $c\,a(b\,a)^m$ occurs together with the text $(x^{2m}\,a\,a(b\,a)^m)^n$. They observed that this particular example was handled in linear time when Kuiper's improvement (dd' for dd) was made; but they left open the question of the true worst case behavior of the improved algorithm.

There are trivial cases in which the Boyer–Moore algorithm has quadratic behavior, when matching all occurrences of the pattern, for example when matching the pattern a^m in the text a^n. But we are probably willing to accept such behavior when there are so many matches; the crucial issue is how long the algorithm takes in the worst case to scan over a text that does *not* contain the pattern at all. By extending the techniques of § 5, it is possible to show that the modified Boyer–Moore algorithm is *linear* in such a situation:

THEOREM. *If the above algorithm is used with dd' replacing dd, and if the text does not contain any occurrences of the pattern, the total number of characters matched is at most* $6n$.

Proof. An execution of the algorithm consists of a series of *stages*, in which m_k characters are matched and then the pattern is shifted s_k places, for $k=1,2,\dots$. We want to show that $\sum m_k\leqq 6n$; the proof is based on breaking this cost into three parts, two of which are trivially $O(n)$ and the third of which is less obviously so.

Let $m'_k=m_k-2s_k$ if $m_k>2s_k$; otherwise let $m'_k=0$. When $m'_k>0$, we will say that the leftmost m'_k text characters matched during the kth stage have been "tapped". It suffices to prove that the algorithm taps characters at most $4n$ times, since $\sum m_k\leqq\sum m'_k+2\sum s_k$ and $\sum s_k\leqq n$. Unfortunately it is possible for some characters of the text to be tapped roughly $\log m$ times, so we need to argue carefully that $\sum m'_k\leqq 4n$.

Suppose the rightmost m''_k of the m'_k text characters tapped during the kth stage are matched again during some later stage, but the leftmost $m'_k-m''_k$ are being matched for the last time. Clearly $\sum (m'_k - m''_k) \leqq n$, so it remains to show that $\sum m''_k \leqq 3n$.

Let p_k be the amount by which the pattern would shift after the kth stage if the $d[a]$ heuristic were not present ($d[a] = -\infty$); then $p_k \leqq s_k$, and p_k is a period of the string matched at stage k.

Consider a value of k such that $m''_k > 0$, and suppose that the text characters matched during the kth stage form the string $\alpha = \alpha_1\alpha_2$ where $|\alpha| = m_k$ and $|\alpha_2| = m''_k + 2s_k$; hence the text characters in α_1 are matched for the last time. Since the pattern does not occur in the text, it must end with $x\alpha$ and the text scanned so far must end with $z\alpha$, where $x \neq z$. At this point the algorithm will shift the pattern right s_k positions and will enter stage $k+1$. We distinguish two cases: (i) The pattern length m exceeds $m_k + p_k$. Then the pattern can be written $\theta\beta\alpha$, where $|\beta| = p_k$; the last character of β is x and the last character of θ is $y \neq x$, by definition of dd'. Otherwise (ii) $m \leqq m_k + p_k$; the pattern then has the form $\beta\alpha$, where $|\beta| \leqq p_k \leqq s_k$. By definition of m''_k and the assumption that the pattern does not occur in the text, we have $|\beta\alpha| > s_k + |\alpha_2|$, i.e., $|\beta| > s_k - |\alpha_1|$. In both cases (i) and (ii), p_k is a period of $\beta\alpha$.

Now consider the first subsequent stage k' during which the *leftmost* of the m''_k text characters tapped during stage k is matched again; we shall write $k \to k'$ when the stages are in this relation. Suppose the mismatch occurs this time when text character z' fails to match pattern character x'. If z' occurs in the text within α_1, regarding α as fixed in its stage k position, then x' cannot be within $\beta\alpha$ where $\beta\alpha$ now occurs in the stage k' position of the pattern, since p_k is a period of $\beta\alpha$ and the character p_k positions to the right of x' is a z' (it matches a z' in the text). Thus x' now appears within θ. On the other hand, if z' occurs to the left of α, we must have $|\alpha_1| = 0$, since the characters of α_1 are never matched again. In either event, case (ii) above proves to be impossible. Hence case (i) always occurs when $m''_k > 0$, and x' always appears within θ.

To complete the argument, we shall show that $\sum_{k \to k'} m''_k$, for all fixed k', is at most $3s_{k'}$. Let $p' = p_{k'}$ and let α' denote the pattern matched at stage k'. Let $k_1 < \cdots < k_r$ be the values of k such that $k \to k'$. If $|\alpha'| + p' \leqq m$, let $\beta'\alpha'$ be the rightmost $p' + |\alpha'|$ characters of the pattern. Otherwise let α'' be the leftmost $|\alpha'| + p' - m$ characters of α'; and let $\beta'\alpha'$ be α'' followed by the pattern. Note that in both cases α' is an initial substring of $\beta'\alpha'$ and $|\beta'| = p'$. In both cases, the actions of the algorithm during stages $k_1 + 1$ through k' are completely known if we are given the pattern and β', and if we know z' and the place within β' where stage $k_1 + 1$ starts matching. This follows from the fact that β' by itself determines the text, so that if we match the pattern against the string $z'\beta'\beta'\beta' \ldots$ (starting at the specified place for stage $k_1 + 1$) until the algorithm first tries to match z' we will know the length of α'. (If $|\alpha'| < p'$ then β' begins with α' and this statement holds trivially; otherwise, α' begins with β' and has period p'; hence $\beta'\beta'\beta' \ldots$ begins with α'.) Note that the algorithm cannot begin two different stages at exactly the same position within β', for then it would loop indefinitely, contradicting the fact that it does terminate. This property will be out key tool for proving the desired result.

Let the text strings matched during stages $k_1, \ldots, k_r$ be $\alpha_1, \ldots, \alpha_r$, and let their periods determined as in case (i) be $p_1, \ldots, p_r$ respectively; we have $p_j < \frac{1}{2}|\alpha_j|$

for $1 \leqq j \leqq r$. Suppose that during stage k_j the mismatch of $x_j \neq z_j$ implies that the pattern ends with $y_j\beta_j\alpha_j$, where $|\beta_j| = p_j$. We shall prove that $|\alpha_1| + \cdots + |\alpha_r| \leqq 3p'$. First let us prove that $|\alpha_j| < p'$ for all j: We have observed that x' always occurs within θ_j; hence $y_j\beta_j\alpha_j$ occurs as a rightmost substring of $x'\alpha'$. If $|\alpha_j| \geqq p'$ then $p_j + p' \leqq |\beta_j\alpha_j|$; hence the character p_j positions to the right of y_j in $x'\alpha'$ is x_j, as is the character $p_j + p'$ positions to the right of y_j. But the character p' positions to the right of y_j in $x'\alpha'$ is a y_j, since p' is a period of $x'\alpha'$; hence the character $p' + p_j$ positions to the right of y_j is also y_j, contradicting $x_j \neq y_j$.

Since $|\alpha_j| < p'$, each string α_j for $j \geqq 2$ appears somewhere within β', when β' is regarded as a cyclic string, joined end-for-end. (It follows from the definition of $k \rightarrow k'$ that $z_j\alpha_j$ is a substring of α' for $j \geqq 2$.) We shall prove that the rightmost halves of these strings, namely the rightmost $\lceil \frac{1}{2}|\alpha_j| \rceil$ characters as they appear in β', are disjoint. This implies that $\frac{1}{2}|\alpha_2| + \cdots + \frac{1}{2}|\alpha_r| \leqq p'$, and the proof will be complete (since $|\alpha_1| \leqq p'$).

Suppose therefore that the right half of the appearance of α_i overlaps the right half of the appearance of α_j within β', for some $i \neq j \geqq 2$, where the rightmost character of α_i is within α_j. This means that the algorithm at stage k_i begins to match characters starting within α_j at least p_j characters to the right of z_j where $z_j\alpha_j$ appears in β', when the text α' is treated modulo p'. (Recall that $p_j < \frac{1}{2}|\alpha_j|$.) The pattern ends with $x_j\alpha_j$, and p_j is a period of $x_j\alpha_j$. The algorithm must work correctly when the text equals the pattern, so there must come a stage, before shifting the pattern to the right of the appearance of α_j, where the algorithm scans left until hitting z_j. At this point, call it stage k'', there must be a mismatch of $z_j \neq x_j$, since p_j or more characters have been matched. (The character p_j positions to the right of z_j is x_j, by periodicity.) Hence $k'' < k'$; and it follows that $k'' = k_i$. (If $k'' > k_i$ we have $z_i\alpha_i$ entirely contained within α'', but then $k_i \rightarrow k'$ implies that $k'' = k'$.) Now $k'' = k_i$ implies that $z_j = z_i$ and $x_j = x_i$. We shall obtain a contradiction by showing that the algorithm "synchronizes" its stage $k_i + 1$ behavior with its stage $k_j + 1$ behavior, modulo p', causing an infinite loop as remarked above. The main point is that the dd' table will specify shifting the pattern p_j steps, so that y_j is brought into the position corresponding to z_j, in stage k_i as well as in stage k_j. (Any lesser shift brings an x_j into position p_j spaces to the right of z_j; hence it puts $y_i = x_j$ into the position corresponding to z_j, by periodicity, contradicting $x_i \neq y_i$.) The amount of shift depends on the maximum of the d and dd' entries, and the d entry will be chosen (in either k_i or k_j) if and only if z_j is not a character of β_j; but in this case, the d entry will also specify the same shift both for stage k_i and stage k_j. □

The constant 6 in the above theorem is probably much too large, and the above proof seems to be much too long; the reader is invited to improve the theorem in either or both respects. An interesting example of the rather complex behavior possible with this algorithm occurs when the pattern is $b\psi_r$ and the text is $\psi_r a \psi_r$ for large r, where

$$\psi_0 = a, \qquad \psi_{n+1} = \psi_n\psi_n b\psi_n.$$

COROLLARY. *The worst case running time of the Boyer–Moore algorithm with dd' replacing dd is $O(n + rm)$ character comparisons, if the pattern occurs r times in the text.*

Proof. Let $T(n, r)$ be the worst case running time as a function of n and r, when m is fixed. The theorem implies that $T(n, 0) \leqq 7n$, counting the mismatched characters as well as the matched ones. Furthermore, if $r > 0$ and if the first appearance of the pattern ends at position n_0 we have $T(n, r) \leqq 7(n_0 - 1) + m + T(n - n_0 + m - 1, r - 1)$. It follows that $T(n, r) \leqq 7n + 8rm - 14r$. □

When the Boyer–Moore algorithm implicitly shifts the pattern to the right, it forgets all it "knows" about characters already matched; this is why the linearity theorem is not trivial. A more complex algorithm can be envisaged, with a finite number of states corresponding to which text characters are known to match the pattern in its current position; when in state q we fetch the character $x := text[k - t[q]]$, then we set $k := k + s[q, x]$ and go to state $q'[q, x]$. For example, consider the pattern $a\ b\ a\ c\ b\ a\ b\ a$, and the specification of t, s, and q' in Table 3; exactly 41 distinguishable states can arise. An asterisk (*) in that table shows where the pattern has been fully matched.

The number of states in this generalization of the Boyer–Moore algorithm can be rather large, as the example shows, but the patterns which occur most often in practice probably do not imply many states. The number of states is always less than 2^m, and perhaps a much smaller upper bound is possible; it is unclear which patterns of a given length lead to the most states, and it does not seem obvious that this maximum number of states is exponential in m.

If the characters of the pattern are distinct, say $a_1 a_2 \ldots a_m$, this generalization of the Boyer–Moore algorithm leads to exactly $\frac{1}{2}(m^2 + m)$ states. (Namely, all states of the form $\bullet \ldots \bullet\, a_k \bullet \ldots \bullet\, a_{j+1} \ldots a_m$ for $0 \leqq k < j \leqq m$, with a_k suppressed if $k = 0$.) By merging several of these states we obtain the following simple algorithm, which uses a table $c[x]$ where

$$c[x] = \begin{cases} m - j, & \text{if } x = a_j; \\ -1, & \text{if } x \notin \{a_1, \ldots, a_m\}. \end{cases}$$

The algorithm works only when all pattern characters are distinct, but it improves slightly on the Boyer–Moore technique in this important special case.

```
j := k := m;
while k ≦ n do
    begin i := c[text[k]];
        if i < 0 then j := m
        else if i = 0 then
        begin for i := 1 step 1 until m − 1 do
            if text[k − i] ≠ pattern[m − i] then go to nomatch;
            match found at (k − m);
        nomatch: j := m;
        end else if i + j ≥ m then j := i else j := m;
        k := k + j;

    end;
```

Let us close this section by making a preliminary investigation into the question of "fastest" pattern matching in strings, i.e., *optimum* algorithms. What algorithm minimizes the number of text characters examined, over all conceivable algorithms for the problem we have been considering? In order to make this question nontrivial, we shall ask for the minimum *average* number of characters examined when finding *all* occurrences of the pattern in the text, where the average is taken uniformly with respect to strings of length n over a given alphabet. (The minimum worst case number of characters examined is of no interest, since it is between $n-m$ and n for all patterns[3]; therefore we ask for the minimum average number. It might be argued that the minimum average number, taken over random strings, is of little interest, since people rarely search in random strings; they usually search for patterns that actually appear. However, the random-string model is a reasonable approximation when we consider those stretches of text that do not contain the pattern, and the algorithm obviously must examine every character in those places where the pattern does occur.)

The case of patterns of length 2 can be solved exactly; it is somewhat surprising to find that the analysis is not completely trivial even in this case. Consider first the pattern $a\,b$ where $a \neq b$. Let q be the alphabet size, $q \geqq 2$. Let $f(n)$ denote the minimum average number of characters examined by an algorithm which finds all occurrences of the pattern in a random text of length n; and let $g(n)$ denote the minimum average number of characters examined in a random text of length $n+1$ which is known to begin with a, not counting the examination of the known first character. These functions can be computed by the following recurrence relations:

$$f(0)=f(1)=g(0)=0, \qquad g(1)=1.$$

$$f(n)=1+\min_{1\leqq k\leqq n}\Big(\frac{1}{q}(f(k-1)+g(n-k))+\frac{1}{q}(g(k-1)+f(n-k)) + \Big(1-\frac{2}{q}\Big)(f(k-1)+f(n-k))\Big),$$

$$g(n)=1+\frac{1}{q}g(n-1)+\Big(1-\frac{1}{q}\Big)f(n-1), \qquad n\geqq 2.$$

The recurrence for f follows by considering which character is examined first; the recurrence for g follows from the fact that the second character must be examined in any case, so it can be examined first without loss of efficiency. It can be shown that the minimum is always assumed for $k=2$; hence we obtain the closed form solution

$$f(n)=\frac{n(q^2+q-1)}{q(2q-1)}-\frac{(q-1)(q^2+2q-1)}{q(2q-1)^2}+\frac{(1-q)^n}{q^{n-3}(q-1)(2q-1)^2},$$

$$g(n)=\frac{n(q^2+q-1)}{q(2q-1)}+\frac{(q-1)(q^2-3q+1)}{q(2q-1)^2}-\frac{(1-q)^n}{q^{n-2}(2q-1)^2}, \qquad n\geqq 1.$$

(To prove that these functions satisfy the stated recurrences reduces to showing that the minimum of

$$\Big(\frac{1-q}{q}\Big)^{k-1}+\Big(\frac{1-q}{q}\Big)^{n-k}$$

for $1\leqq k\leqq n$ occurs for $k=2$, whenever $n\geqq 2$ and $q\geqq 2$.)

[3] This is clear when we must find *all* occurrences of the pattern; R. L. Rivest has recently proved it also for algorithms which stop after finding *one* occurrence. (*Information Processing Letters*, to appear.)

TABLE 3

			$s[q,x], q'[q,x]$			
state q	known characters	$t[q]$	$x=a$	$x=b$	$x=c$	other x
0	• • • • • • • •	0	0, 1	1, 8	4, 9	8, 0
1	• • • • • • • a	1	7, 10	0, 2	7, 10	7, 10
2	• • • • • • b a	2	0, 3	7, 10	2, 11	7, 10
3	• • • • • a b a	3	5, 12	0, 4	5, 12	5, 12
4	• • • • b a b a	4	5, 12	5, 12	0, 5	5, 12
5	• • • c b a b a	5	0, 6	5, 12	5, 12	5, 12
6	• • a c b a b a	6	5, 12	0, 7	5, 12	5, 12
7	• b a c b a b a	7	*5, 12	5, 12	5, 12	5, 12
8	• • • • • • b •	0	0, 2	8, 0	8, 0	8, 0
9	• • • c • • • •	0	0, 13	6, 14	4, 9	8, 0
10	a • • • • • • •	0	0, 15	1, 8	4, 9	8, 0
11	• • • c b a • •	0	0, 16	6, 14	8, 0	8, 0
12	a b a • • • • •	0	0, 17	3, 18	4, 9	8, 0
13	• • • c • • • a	1	7, 10	0, 19	7, 10	7, 10
14	• b • • • • • •	0	0, 20	3, 18	4, 9	8, 0
15	a • • • • • • a	1	7, 10	0, 21	7, 10	7, 10
16	• • • c b a • a	1	7, 10	0, 5	7, 10	7, 10
17	a b a • • • • a	1	7, 10	0, 22	7, 10	7, 10
18	• • • • b • • •	0	0, 23	3, 24	8, 0	8, 0
19	• • • c • • b a	2	0, 25	7, 10	7, 10	7, 10
20	• b • • • • • a	1	7, 10	0, 26	7, 10	7, 10
21	a • • • • • b a	2	0, 27	7, 10	2, 11	7, 10
22	a b a • • • b a	2	0, 28	7, 10	2, 29	7, 10
23	• • • • b • • a	1	7, 10	0, 30	7, 10	7, 10
24	• b • • b • • •	0	0, 31	3, 24	8, 0	8, 0
25	• • • c • a b a	3	5, 12	0, 5	5, 12	5, 12
26	• b • • • • b a	2	0, 32	7, 10	2, 11	7, 10
27	a • • • • a b a	3	5, 12	0, 33	5, 12	5, 12
28	a b a • • a b a	3	5, 12	0, 34	5, 12	5, 12
29	a • • c b a • •	0	0, 35	6, 14	8, 0	8, 0
30	• • • • b • b a	2	0, 4	7, 10	7, 10	7, 10
31	• b • • b • • a	1	7, 10	0, 36	7, 10	7, 10
32	• b • • • a b a	3	5, 12	0, 37	5, 12	5, 12
33	a • • • b a b a	4	5, 12	5, 12	0, 38	5, 12
34	a b a • b a b a	4	5, 12	5, 12	*5, 12	5, 12
35	a • • c b a • a	1	7, 10	0, 38	7, 10	7, 10
36	• b • • b • b a	2	0, 37	7, 10	7, 10	7, 10
37	• b • • b a b a	4	5, 12	5, 12	0, 39	5, 12
38	a • • c b a b a	5	0, 40	5, 12	5, 12	5, 12
39	• b • c b a b a	5	0, 7	5, 12	5, 12	5, 12
40	a • a c b a b a	6	5, 12	*5, 12	5, 12	5, 12

If the pattern is $a\ a$, the recurrence for f changes to

$$f(n)=1+\min_{1\leqq k\leqq n}\left(\frac{1}{q}(g(k-1)+g(n-k))+\left(1-\frac{1}{q}\right)(f(k-1)+f(n-k))\right),\qquad n\geqq 2;$$

but this is actually no change!

Hence the following is an optimum algorithm for all patterns of length 2, in the sense of minimum average text characters inspected to find all matches in a random string:

```
$k := 2$;
while $k \leqq n$ do
    begin $c :=$ text[$k$];
        if $c =$ pattern[2] and text[$k-1$] = pattern[1]
        then match found at $(k-2)$;
        while $c =$ pattern[1] do
            begin $k := k+1$; $c :=$ text[$k$];
                if $c =$ pattern[2] then match found at $(k-2)$;
            end;
        $k := k+2$;
    end;
```

For patterns of length 3 the recurrence relations become more complex; they depend on more than simply the length of the strings and knowledge about characters at the boundaries. The determination of an optimum strategy in this case remains an open problem. The algorithm sketched at the beginning of this section shows that an average of $O(n(\log m)/m)$ bit inspections suffices over a binary alphabet. Clearly $\lfloor n/m \rfloor$ is a lower bound, since the algorithm must inspect at least one bit in any block of n consecutive bits. The pattern a^m can be handled with $O(n/m)$ bit inspections on the average; but it seems reasonable to conjecture that patterns of length m exist for arbitrarily large m, such that an average of at least $cn(\log m)/m$ bits must be inspected for all large n. Here c denotes a positive constant, independent of m and n.

Acknowledgment. Robert S. Boyer and J. Strother Moore suggested many important improvements to early drafts of this postscript, especially in connection with errors in the author's first attempts at proving the linearity theorem.

REFERENCES

[1] Alfred V. Aho, John E. Hopcroft and Jeffrey D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Mass., 1974.

[2] Alfred V. Aho and Margaret J. Corasick, *Efficient string matching: An aid to bibliographic search*, Comm. ACM, 18 (1975), pp. 333–340.

[3] M. Beeler, R. W. Gosper and R. Schroeppel, *HAKMEM*, Memo No. 239, M.I.T. Artificial Intelligence Laboratory, Cambridge, Mass., 1972.

[4] Robert S. Boyer and J. Strother Moore, *A fast string searching algorithm*, manuscript dated December 29, 1975; Stanford Research Institute, Menlo Park, Calif., and Xerox Palo Alto Research Center, Palo Alto, Calif.

[5] S. A. Cook, *Linear time simulation of deterministic two-way pushdown automata*, Information Processing 71, North-Holland, Amsterdam, 1972, pp. 75–80.

[6] Pascal Diethelm and Peter Roizen, *An efficient linear search for a pattern in a string*, unpublished manuscript dated April, 1972; World Health Organization, Geneva, Switzerland.

[7] N. J. Fine and H. S. Wilf, *Uniqueness theorems for periodic functions*, Proc. Amer. Math. Soc., 16 (1965), pp. 109–114.

[8] Michael J. Fischer and Michael S. Paterson, *String matching and other products*, SIAM-AMS Proc., vol. 7, American Mathematical Society, Providence, R.I., 1974, pp. 113–125.

[9] Zvi Galil, *On converting on-line algorithms into real-time and on real-time algorithms for string-matching and palindrome recognition*, SIGACT News, 7 (1975), No. 4, pp. 26–30.

[10] E. N. Gilbert, *Synchronization of binary messages*, IRE Trans. Information Theory, IT-6 (1960), pp. 470–477.

[11] Sheila A. Greibach, *The hardest context-free language*, this Journal, 2 (1973), pp. 304–310.

[12] Malcolm C. Harrison, *Implementation of the substring test by hashing*, Comm. ACM, 14 (1971), pp. 777–779.

[13] Richard M. Karp, Raymond E. Miller and Arnold L. Rosenberg, *Rapid identification of repeated patterns in strings, trees, and arrays*, ACM Symposium on Theory of Computing, vol. 4, Association for Computing Machinery, New York, 1972, pp. 125–136.

[14] Donald E. Knuth, *Fundamental Algorithms, The Art of Computer Programming*, Vol. 1, Addison-Wesley, Reading, Mass., 1968; 2nd edition 1973.

[15] ———, *Sequences with precisely $k+1$ k-blocks*, Solution to problem E2307, Amer. Math. Monthly, 79 (1972), pp. 773–774.

[16] ———, *On the translation of languages from left to right*, Information and Control, 8 (1965), pp. 607–639.

[17] ———, *Structured programming with* **go to** *statements*, Computing Surveys, 6 (1974), pp. 261–301.

[18] Donald E. Knuth, James H. Morris, Jr. and Vaughan R. Pratt, *Fast pattern matching in strings*, Tech. Rep. CS440, Computer Science Department, Stanford Univ., Stanford, Calif., 1974.

[19] R. C. Lyndon and M. P. Schützenberger, *The equation $a^M = b^N c^P$ in a free group*, Michigan Math. J., 9 (1962), pp. 289–298.

[20] Glenn Manacher, *A new linear-time on-line algorithm for finding the smallest initial palindrome of a string*, J. Assoc. Comput. Mach., 22 (1975), pp. 346–351.

[21] A. Markoff, *Sur une question de Jean Bernoulli*, Math. Ann., 19 (1882), pp. 27–36.

[22] J. H. Morris, Jr. and Vaughan R. Pratt, *A linear pattern-matching algorithm*, Tech. Rep. 40, Univ. of California, Berkeley, 1970.

[23] A. O. Slisenko, *Recognition of palindromes by multihead Turing machines*, Dokl. Steklov Math. Inst., Akad. Nauk SSSR, 129 (1973), pp. 30–202. (In Russian.)

[24] Ken Thompson, *Regular expression search algorithm*, Comm. ACM, 11 (1968), pp. 419–422.

[25] B. A. Venkov, *Elementary Number Theory*, Wolters-Noordhoff, Groningen, the Netherlands, 1970.

[26] Peter Weiner, *Linear pattern matching algorithms*, IEEE Symposium on Switching and Automata Theory, vol. 14, IEEE, New York, 1973, pp. 1–11.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.