PB-237 360

FAST PATTERN MATCHING IN STRINGS

Donald E. Knuth, et al

Stanford University

Prepared for:

National Science Foundation

August 1974

BIBLIOGRAPHIC DATA SHEET	1. Report No. STAN-CS-74-440	2.	PB 237 360

4. Title and Subtitle

FAST PATTERN MATCHING IN STRINGS

5. Report Date: August 1974

6.

7. Author(s): Donald E. Knuth, James H. Morris, Jr. and Vaughan R. Pratt

8. Performing Organization Rept. No. STAN-CS-74-440

9. Performing Organization Name and Address
Stanford University
Computer Science Department
Stanford, California 94305

10. Project/Task/Work Unit No.

11. Contract/Grant No. GJ 36473X

12. Sponsoring Organization Name and Address
National Science Foundation
1800 G Street, N.W.
Washington, D. C. 20550

13. Type of Report & Period Covered: technical, Aug. 1974

14.

15. Supplementary Notes

16. Abstracts

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.

17. Key Words and Document Analysis. 17a. Descriptors

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

17b. Identifiers/Open-Ended Terms

Reproduced by
NATIONAL TECHNICAL INFORMATION SERVICE
U. S. Department of Commerce
Springfield VA 22151

17c. COSATI Field/Group

18. Availability Statement

Approved for public release; distribution unlimited.

19. Security Class (This Report) UNCLASSIFIED

20. Security Class (This Page) UNCLASSIFIED

21. No. of Pages 35

22. Price $2.75 $2.25

Fast Pattern Matching in Strings

by Donald E. Knuth (Stanford University),*/
James H. Morris, Jr. (Xerox Palo Alto Research Center),**/
and Vaughan R. Pratt (Massachusetts Inst. of Technology).***/

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.

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

CR Categories: 4.40, 5.25, 5.23, 3.89

*/ The research reported here was done at Stanford University, supported in part by National Science Foundation grant GJ 36473X and by the Office of Naval Research contract NR 044-402. Reproduction in whole or in part is permitted for any purpose of the United States Government.

**/ The research reported here was done at the University of California, Berkeley, supported in part by National Science Foundation grant number GP 7635.

***/ The research reported here was done at the University of California, Berkeley, supported in part by National Science Foundation grant GP-6945; and at Stanford University, supported in part by National Science Foundation grant GJ-992.

Fast Pattern Matching in Strings

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, catenary contains the pattern ten, but we do not regard canary as one of its substrings.

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 we find characters that don't match. But this approach can be very inefficient, for example when we are looking for an occurrence of aaaaaaab in aaaaaaaaaaaaaaab. When the pattern is $a^n b$ 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, and 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.

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 Section 6 uses the algorithm to disprove the conjecture that a certain context-free language cannot be recognized in linear time. Finally, Section 7 discusses the origin of the algorithm and its relation to other recent work.

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 the 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, from the fact that we have matched the first three pattern characters but not the fourth, we know that the last four characters of the input have been $\underline{a}\,\underline{b}\,\underline{c}\,\underline{x}$ 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 $\underline{a}$), we deduce that the pattern can immediately be shifted four more places to the right; one, two, or three shifts can'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 $\underline{a}\,\underline{b}\,\underline{c}\,\underline{a}\,\underline{b}\,\underline{c}\,\underline{a}\,\underline{x}$, 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 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 j-th character pattern[j] . Let next[j] be the character position to check 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

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] . Let j and k 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 := 0;
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 ≤ m , the text has been exhausted.

The above program is easily proved correct using the following invariant relation: "Let $p = k-j$ (the position in the text just preceding the first character of the pattern, in our assumed alignment). Then we have $\underline{text}[p+i] = \underline{pattern}[i]$ for $1 \le i < j$; but for $1 \le t < p$ we have $\underline{text}[t+i] \neq \underline{pattern}[i]$ for some i , where $1 \le i \le m$."

The program will of course be correct only if we can compute the $\underline{next}$ table so that the above relation remains invariant when we perform the operation $j := \underline{next}[j]$. Let us look at that computation now. When the program sets $j := \underline{next}[j]$, we know that $j > 0$, and that the last j characters of the input were

$$\underline{pattern}[1] \ldots \underline{pattern}[j-1]\ x$$

where $x \neq \underline{pattern}[j]$. 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 $\underline{next}[j]$ to be the largest i less than j such that the last i characters of the input were

$$\underline{pattern}[1] \ldots \underline{pattern}[i-1]\ x$$

and $\underline{pattern}[i] \neq \underline{pattern}[j]$. (If no such i exists, we let $\underline{next}[j] = 0$.) With this definition of $\underline{next}[j]$ it is easy to verify that $\underline{text}[t+1] \ldots \underline{text}[k] \neq \underline{pattern}[1] \ldots \underline{pattern}[k-t]$ for $k-j \le t < k - \underline{next}[j]$; 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 $\underline{next}[j]$ in the first place. This problem would be easier if we didn't require $\underline{pattern}[i] \neq \underline{pattern}[j]$ in the definition of $\underline{next}[j]$, so we shall consider the easier problem first. Let $f(j)$

be the largest i less than j such that pattern[1] ... pattern[i-1] = pattern[j-i+1] ... pattern[j-1] ; since this condition holds vacuously for $i = 1$, we always have $f(j) \geq 1$ when $j > 1$. By convention we let $f(1) = 0$. The pattern used in the example of Section 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] ... pattern[j-1] = text[k-j+1] ... text[k-1] , so we can transfer the previous technology to the present problem.) The following program will compute $f(j+1)$, assuming that next[1] ... next[j-1] and $f(j)$ 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] ≠ 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 ≠ b . The next table tells us to shift right four places, obtaining

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

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 ,

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

Therefore we can compute the next table as follows.

```
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 is increased $m-1$ times, by 1; furthermore its value remains nonnegative. Therefore the operation $t := \underline{next}[t]$, 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. (1) A table of "shifts", specifying how to move the given pattern when a mismatch occurs at its j-th character, can be precomputed. (2) This computation 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 $\underline{\text{text}}[k] \neq \underline{\text{pattern}}[1]$ for $k = 1,2,3,\ldots$. When $j = 1$ and $\underline{\text{text}}[k] \neq \underline{\text{pattern}}[1]$, the algorithm sets $j := \underline{\text{next}}[1] = 0$, then discovers that $j = 0$, then increases k by 1, then sets j to 1 again, then tests whether or not 1 is $\leq m$, 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 $\underline{\text{pattern}}[m+1] =$ '@' for some impossible character @ that will never be matched, and by letting $\underline{\text{next}}[m+1] = -1$; then a test for $j < 0$ can be inserted into a less-frequently executed part of the code. Similarly we can set $\underline{\text{text}}[n+1] =$ '⊥' (another impossible character) and $\underline{\text{text}}[n+2] = \underline{\text{pattern}}[1]$, so that the test for $k > n$ needn't be made very often.

The following form of the algorithm incorporates these refinements.

```
        k := 0; a := pattern[1];
        pattern[m+1] := '@'; next[m+1] := -1;
        text[n+1] := '⊥'; text[n+2] := a;
    advance: comment  j = 0 in previous program;
        repeat k := k+1 until text[k] = a;
        if k > n then go to input exhausted;
            j := 1;
    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 = 0 then go to advance;
        if j = 1 then begin
            if text[k] ≠ a then go to advance
            else go to char matched end;
        if j > 0 then go to loop;
    comment  text[k-m] through text[k-1] matched;
```

Except that we are now assuming a non-null pattern ($m > 0$), this program preserves the robustness of the original. It 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. [2, Hack 179].) For example, the pattern in Section 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;
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!

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 pattern[m+1] = '@' , and then we set resume := next[m+1] before setting next[m+1] to -1 . After finding a match and doing whatever action is desired to process that match, the sequence

```
j := resume; go to 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 next[m+1] untouched, not changing it to -1 , and to define integer arrays head[1:m] and link[1:n] initially zero, and to insert the code

```
link[k] := head[j]; head[j] := k;
```

at label 'char matched' . This forms linked lists for $1 \le j \le m$ of all places where the first j characters of the pattern 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 {abcab, ababc, bcacc, bbc}, the trie is

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

Such a trie can be constructed efficiently by generalizing the idea we used to calculate next[j] ; details are left to the reader. (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.)

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 $|\epsilon| = 0$. We also write $\alpha[k]$ for the k-th character of α.

As a warmup for our theoretical discussion, let us consider the Fibonacci strings [9, exercise 1.2.8-36], which turn out to be especially pathological patterns for the above algorithm. The definition of Fibonacci strings is

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

For example, $\phi_3 = ab$, $\phi_4 = aba$, $\phi_5 = abaab$. It follows that the length $|\phi_n|$ is the n-th Fibonacci number F_n, and that ϕ_n consists of the first F_n characters of an infinite string ϕ_∞.

Consider the pattern ϕ_8, which has the following functions f(j) and next[j]:

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

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

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

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

where $c(\alpha)$ denotes changing the two rightmost characters of α. For example, $\phi_6 = \mathtt{abaab \cdot aba}$ and $c(\phi_6) = \mathtt{abaa \cdot abaab}$. 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

$$\underline{\text{next}}[F_k - 1] = F_{k-1} - 1 \quad , \quad \text{for } k \geq 3 \ . \tag{4}$$

Therefore if we have a mismatch when $j = F_8 - 1 = 20$, our algorithm might set $j := \underline{\text{next}}[j]$ for 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 $j := \underline{\text{next}}[j]$ loop.

We will 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 <u>periodicity</u> in strings. We say that p is a <u>period</u> of α if

$$\alpha[i] = \alpha[i+p] \quad \text{for } 1 \leq i \leq |\alpha| - p \ . \tag{5}$$

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

$$\alpha = (\alpha_1\alpha_2)^k\alpha_1 \tag{6}$$

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

$$\alpha\theta_1 = \theta_2\alpha \tag{7}$$

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 $\underline{\text{pattern}}[1] \ldots \underline{\text{pattern}}[j-1] = \alpha$ ends with $\underline{\text{pattern}}[1] \ldots \underline{\text{pattern}}[i-1] = \beta$, we have

$$\alpha = \beta\theta_1 = \theta_2\beta \tag{8}$$

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

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

$$\beta = \gamma\psi_1 = \psi_2\gamma \tag{9}$$

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

$$|\alpha| > |\beta| + |\gamma| \quad . \tag{10}$$

For if $|\beta| + |\gamma| \geq |\alpha|$, 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 $j := \text{next}[j]$ is performed, while one text character is being scanned, is less than $\log_\phi m + K$ for some constant K .

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 $|\beta| \geq L_{r-1}$, $|\gamma| \geq L_{r-2}$ in (10), hence $L_r \geq F_{r+1}-1$ by induction on r . □

The subject of periods in strings has several interesting algebraic properties, but a reader who is not mathematically inclined may skip to Section 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 $p+q \leq |\alpha| + \gcd(p,q)$, 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 $1 \leq i \leq |\alpha|-p$ and $\alpha[i] = \alpha[i+q]$ for $1 \leq i \leq |\alpha|-q$; hence $\alpha[i+r] = \alpha[i+q] = \alpha[i]$ for $1+r \leq i+r \leq |\alpha|-p$, i.e.,

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

Furthermore $\alpha = \beta\theta_1 = \theta_2\beta$ where $|\theta_1| = p$, and it follows that p and r are periods of β , where $p+r \leq |\beta| + d = |\beta| + \gcd(p,r)$. By

induction, d is a period of β. Since $|\beta| = |\alpha|-p \geq q-d \geq q-r = p = |\theta_1|$, 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 $\beta_2\beta_1$. □

The result of Lemma 1 but with the stronger hypothesis $p+q \leq |\alpha|$ was proved by Lyndon and Schützenberger in connection with a problem about free groups [11, Lemma 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 <u>not</u> a period. In order to see why this is so, consider first the following example 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 car be different <u>are</u> 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
```

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 $n \ge m \ge 0$ as follows:

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

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

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 Equation (5) above.]

Proof: We have, for $0 < m < n$ and $q \geq 2$,

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

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

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

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

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

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

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 $m = 2$, $n = 2q+1$, $\sigma(2,2q+1) = (ab)^q a$, 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

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

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

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

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 [10]. 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 [4].

If α is any string, let $P(\alpha)$ be its shortest period. Lemma 1 implies that all periods q which are not multiplies 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 α = pattern[1] ... 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 p-th, 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, $P(\alpha)+q \geq j+1$. On the other hand $q > P(\alpha)$, hence $q > \frac{1}{2}j$ and $\underline{next}[j] < \frac{1}{2}j$. 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 $\underline{next}[\underline{next}[j]]$ 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.

It appears to be extremely difficult to analyze the 'average' behavior of this algorithm instead of the worst case behavior. However, average behavior on random strings is surely unrealistic because there would only rarely be a match in a random string.

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

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

Here α^R denotes the reversal of α, i.e., $(a_1 a_2 \cdots a_n)^R = a_n \cdots a_2 a_1$. 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 all palstars $\pi_1 \cdots \pi_n$ 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, aabbabba 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

baabbabbaababbaabbabbabaabbabbabbabbaababbabbbaab

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 $\beta_1 \cdots \beta_t$ of prime palstars, for some $t \geq 0$; what is less obvious is that such a decomposition with 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 $\gamma \neq \epsilon$; 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 $\delta \neq \epsilon$; 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 $|\beta\beta^R| \leq |\alpha|$, 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 $\alpha\alpha^R$. □

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

Proof: Let $\alpha = \alpha_1 \cdots \alpha_r$ and $\alpha\beta = \beta_1 \cdots \beta_s$ be prime factorizations of α and $\alpha\beta$. If $\alpha_1 \cdots \alpha_r = \beta_1 \cdots \beta_r$ 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. □

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 = α and text = α^R . When $k = n+1$ the algorithm will stop with j maximal such that pattern[1] ... pattern[j-1] = text[n+2-j] ... text[n] . Now perform the following iteration:

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

By the theory developed in Section 3, this iteration terminates with $j \geq 3$ 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 $\alpha = \underline{a}\,\underline{a}\,\underline{b}\,\underline{a}\,\underline{a}\,\underline{b}$. But the pattern matching process takes $O(n)$ time even when $f(j)$ is used.)

□

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 $\leq Kn$ for all large n. We begin by testing α's initial subsequences of lengths $1,2,4,\ldots,2^k,\ldots$, 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

$$(K+K_1) + (2K+K_1) + (4K+K_1) + \ldots + (|\alpha|K+K_1) < 2Kn + K_1 \log_2 n$$

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

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

cannot be handled by the above techniques, since it contains both a a a b b b and a a a b b b 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. 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 [6], 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 [3]. 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 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, 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 [12]. 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.

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. Cook had proved that P^* was recognizable in $O(n \log n)$ steps on a random-access machine, and Pratt improved this to $O(n)$.

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 1973 that such efficiency would be impossible to achieve. An algorithm due to Karp, Miller, and Rosenberg [8] 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, by using tree structures in a remarkable new way [13]. 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 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] 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

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

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 [5] 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 $O(n \log m \log\log m \log t)$ units of time, where t 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 [7]. 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.

References

[1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman, The Design and Analysis of Computer Algorithms (Reading, Mass.: Addison-Wesley, 1974), Section 9.2.

[2] M. Beeler, R. W. Gosper, R. Schroeppel, "HAKMEM," M.I.T. Artificial Intelligence Laboratory Memo No. 239 (February 29, 1972), 95 pp.

[3] S. A. Cook, "Linear time simulation of deterministic two-way pushdown automata," Proc. IFIP Congress (1971), 75-80.

[4] N. J. Fine and H. S. Wilf, "Uniqueness theorems for periodic functions," Proc. Amer. Math. Soc. 16 (1965), 109-114.

[5] Michael J. Fischer and Michael S. Paterson, "String matching and other products," memorandum, M.I.T. Project MAC (January, 1974); 21 pp.

[6] Sheila A. Greibach, "The hardest context-free language," SIAM J. Computing 2 (1973), 304-310.

[7] Malcolm C. Harrison, "Implementation of the substring test by hashing," Comm. ACM 14 (1971), 777-779.

[8] 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 4 (May, 1972), 125-136.

[9] Donald E. Knuth, Fundamental Algorithms, The Art of Computer Programming 1 (Reading, Mass.: Addison-Wesley, 1968, 2nd edition 1973), 634 pp.

[10] D. E. Knuth, "Sequences with precisely k+1 k-blocks," Solution to problem E2307, Amer. Math. Monthly 79 (1972), 773-774.

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

[12] J. H. Morris, Jr., and Vaughan R. Pratt, "A linear pattern-matching algorithm," Technical report 40, University of California, Berkeley, California (June, 1970); 6 pp.

[13] Peter Weiner, "Linear pattern matching algorithms," IEEE Symposium on Switching and Automata Theory 14 (1973), 1-11.