Chapter 1: Single Keyword Matching

Introduction

Single keyword matching means locating all occurrences of a given pattern in the input text string. It occurs naturally as part of data processing, text editing, text retrieval, and so on. Many text editors and programming languages have facilities for matching strings. The simplest technique is called the brute-force (BF), or naive, algorithm. This approach scans the text from left to right and checks the characters of the pattern character by character against the substring of the text string beneath it. Let m and n be the lengths of the pattern and the text, respectively. In the BF approach, the longest (worst-case) time required for determining that the pattern does not occur in the text is $O(mn)$.

Three major pattern matching algorithms for the improvement of efficiency over the BF technique exist. One of them is the KMP algorithm, developed by Knuth, Morris, and Pratt. The KMP algorithm scans the text from left to right, using knowledge of the previous characters compared to determine the next position of the pattern to use. The algorithm first reads the pattern and in $O(m)$ time constructs a table, called the *next function*, that determines the number of characters to slide the pattern to the right in case of a mismatch during the pattern matching process. The expected theoretical behavior of the KMP algorithm is $O(n+m)$, and the next function takes $O(m)$ space.

The next algorithm, the BM algorithm, was proposed by Boyer and Moore. The BM approach is the fastest pattern matching algorithm for a single keyword in both theory and practice. The BM algorithm compares characters in the pattern from right to left. If a mismatch occurs, the algorithm computes a shift, that is, the amount by which the pattern is moved to the right before a new matching is attempted. It also preprocesses the pattern in order to produce the shift tables. The expected theoretical behavior of the BM algorithm is equal to that of the KMP algorithm, but many experimental results show that the BM algorithm is faster than the KMP algorithm.

The last approach is the KR algorithm, presented by Karp and Rabin. The KR algorithm uses extra memory to advantage by treating each possible m-character section (where m is the pattern length) of the text string as a keyword in a standard hash table, computing the hash function of it, and checking whether it equals the hash function of the pattern. Although the KR algorithm is linear in the number of references to the text string per characters passed, the substantially higher running time of this algorithm makes it unfeasible for pattern matching in strings.

In the rest of the chapter, many improvements, including parallel approaches, and variants of the basic single keyword matching algorithms introduced above are discussed along with the corresponding references.

In order to introduce these typical single keyword matching techniques, I have selected the three papers Knuth, Morris, and Pratt (1977), Boyer and Moore (1977), and Davies and Bowsher (1986). The first two papers are the original papers of the KMP and BM algorithms, respectively. The third paper includes comprehensive descriptions and useful empirical evaluation of the BF, KMP, BM, and KR algorithms. Good surveys of single keyword matching are in [Baeza-Yates, 89a], [Baeza-Yates, 92], and [Pirkldauer, 92].

Brute-force (BF) algorithm

This approach scans the text from left to right and checks the characters of the pattern character by character against the substring of the text string beneath it. When a mismatch occurs, the pattern is shifted to the right one character. Consider the following example.

Pattern:	text
Text:	In this example the algorithm searches in the text ...

In this example the algorithm searches in the text for the first character of the pattern (indicated by underline). It continues for every character of the pattern, abandoning the search as soon as a mismatch occurs; this happens if an initial substring of the pattern occurs in the text and is known as a *false start*. It is not difficult to see that the worst-case execution time occurs if, for every possible starting position of the pattern in the text, all but the last character of the pattern matches the corresponding character in the text. For pattern $a^{m-1}b$ and for text a^n with $n>>m$, O(mn) comparisons are needed to determine that the pattern does not occur in the text.

Knuth-Morris-Pratt (KMP) algorithm

The KMP algorithm scans the text from left to right, using knowledge of the previous characters compared, to determine the next position of the pattern to use. The algorithm first reads the pattern and in O(m) time constructs a table, called the next function, that determines how many characters to slide the pattern to the right in case of a mismatch during the pattern matching process. Consider the following example.

Position:	1	2	3	4	5
Pattern:	a	b	a	a	a
next:	0	1	0	2	2

By using this next function, the text scanning is as follows:

	i	1	2	3	4	5					
Pattern:		a	b	a	a	a					
Text:		a	b	a	b	a	a	b	a	a	a
	j	1	2	3	4	5	7	8	9	10	11

Let i and j be the current positions for the pattern and the text, respectively. In the position j=4, which is a b in the text, matching becomes unsuccessful in the same position, i=4, which is an a in the pattern. By adjusting i=next[4]=2 to j=4, the pattern is shifted 2 characters to the right as follows:

			i	1	2	3	4	5			
Pattern:				a	b	a	a	a			
Text:		a	b	a	b	a	a	b	a	a	a
	j	1	2	3	4	5	7	8	9	10	11

After aa is matched, a mismatch is detected in the comparison of a in the pattern with b in the text. Then, the pattern is shifted 3 characters to the right by adjusting i=next[5]=2 to j=8, and then the algorithm finds a successful match as follows:

						i	1	2	3	4	5	
Pattern:							a	b	a	a	a	
Text:		a	b	a	b	a	a	b	a	a	a	
	j	1	2	3	4	5	7	8	9	10	11	

Summarizing, the expected theoretical behavior of the KMP algorithm is O(n+m), and takes O(m) space for the next function. Note that the running time of the KMP algorithm is independent of the size of the alphabet.

Variants that compute the next function are presented by [Bailey et al., 80], [Barth, 81], and [Takaoka, 86]. Barth ([Barth, 84] and [Barth, 85]) has used Markov-chain theory to derive analytical results on the expected number of character comparisons made by the BF and KMP algorithms on random strings.

Boyer and Moore (BM) algorithm

The BM approach is the fastest pattern matching algorithm for a single keyword in both theory and practice. In the KMP algorithm the pattern is scanned from left to right, but the BM algorithm compares characters in the pattern from right to left. If mismatch occurs, then the algorithm computes a shift; that is, it computes the amount by which the pattern is moved to the right before a new matching attempt is undertaken. The shift can be computed using two heuristics. The *match* heuristic is based on the idea that when the pattern is moved to the right, it has to match over all the characters previously matched and bring a *different* character over the character of the text that caused the mismatch. The *occurrence* heuristic uses the fact that we must bring over the character of the text that caused the mismatch the first character of the pattern that matches it. Consider the following example of [Boyer et al., 77].

```
Pattern:  AT-THAT
  Text:   WHICH-FINALLY-HALTS.--AT-THAT-POINT
```

At the start, comparing the seventh character, *F*, of the text with the last character, *T*, fails. Since *F* is known not to appear anywhere in the pattern, the text pointer can be automatically incremented by 7.

```
Pattern:          AT-THAT
  Text:   WHICH-FINALLY-HALTS.--AT-THAT-POINT
```

The next comparison is of the hyphen in the text with the rightmost character in the pattern, *T*. They mismatch, and the pattern includes a hyphen, so the pattern can be moved down 4 positions to align the two hyphens.

```
Pattern:        AT-THAT
  Text:   WHICH-FINALLY-HALTS.--AT-THAT-POINT
```

After *T* is matched, comparing *A* in the pattern with *L* in the text fails. The text pointer can be moved to the right by 7 positions, since the character causing the mismatch, *L*, does not occur anywhere in the pattern.

```
Pattern:                    A T - T H A T
Text:   WHI CH- F I NAL L Y - HAL T S . - - A T - THAT - P OI NT
```

After *AT* is matched, a mismatch is detected in the comparison of *H* in the pattern with the hyphen in the text. The text pointer can be moved to the right by 7 places, so as to align the discovered substring *AT* with the beginning of the pattern.

```
Pattern:                              A T - THAT
Text:   WHI CH- F I NALL Y - HAL T S . - - A T - THAT - P OI NT
```

Karp and Rabin (KR) algorithm

An algorithm developed in [Karp et al., 87] is an improvement of the brute-force approach to pattern matching. This algorithm is a probabilistic algorithm that adapts hashing techniques to string searching. It uses extra memory to advantage by treating each possible *m*-character section of the text string (where *m* is the pattern length) as a key in a standard hash table, computing the hash function of it, and checking whether it equals the hash function of the pattern. Similar approaches using signature files will be discussed in chapter 5.

Here the hash function is defined as follows:

$$h(k) = k \bmod q, \text{ where } q \text{ is a large prime number.}$$

A large value of q makes it unlikely that a collision will occur. We translate the *m*-character into numbers by packing them together in a computer word, which we then treat as the integer k in the function above. This corresponds to writing the characters as numbers in a radix d number system, where d is the number of possible characters. The number k corresponding to the *m*-character section text[i]...text[i+m-1] is

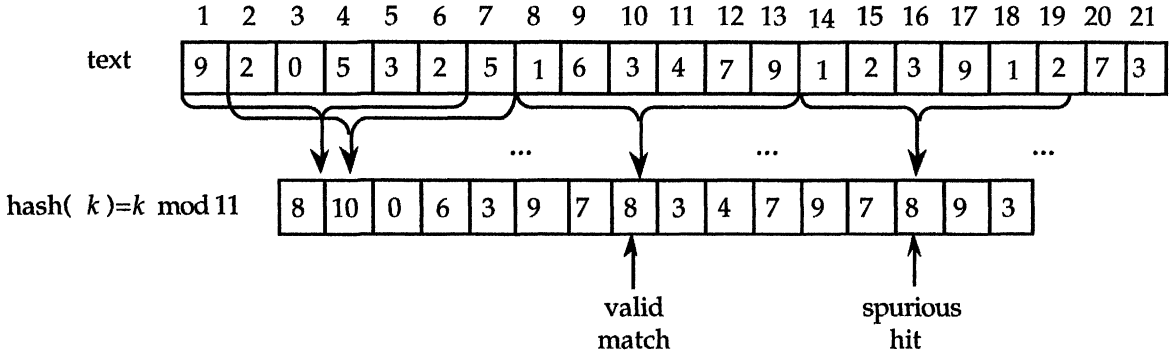
$$k = \text{text}[i] \times d^{m-1} + \text{text}[i+1] \times d^{m-2} + \cdots + \text{text}[i+m-1]$$

Shifting one position to the right in the text string simply corresponds to replacing k by

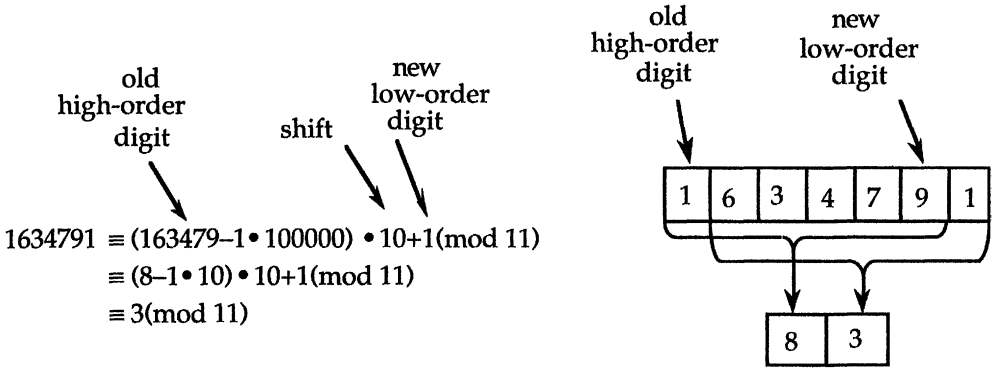
$$(k - \text{text}[i] \times d^{m-1}) \times d + \text{text}[i+m]$$

Consider the example shown in Figure 1 of the KR algorithm based on [Cormen et al., 90]. Each character is a decimal digit, and the hashed value in computed by modulo 11. In Figure 1a the same text string with values computed modulo 11 for each possible position of a section of length 6. Assuming the pattern k=163479, we look for sections whose value modulo 11 is 8, since h(k)=163479 mod 11=8. Two such sections for 163479 and 123912 are found. The first, beginning at text position 8,

is indeed an occurrence of the pattern, and the second, beginning at text position 14, is spurious. In Figure 1b we are computing the value for a section in constant time, given the value for the previous section. The first section has value 163479. Dropping the high-order digit 1 gives us the new value 634791. All computations are performed modulo 11, so the value of the first section is 8 and the value computed of the new section is 3.



(a)



(b)

Figure 1. Illustrations of the KR algorithm

Evaluations of single keyword matching algorithms

Rivest ([Rivest, 77]) has shown that any algorithm for finding a keyword in a string must examine at least $n-m+1$ of the characters in the string in the worst case, and Yao ([Yao, 79]) has shown that the minimum average number of characters needed to be examined in looking for a pattern in a random text string is $O(n\lceil \log_A m \rceil / m)$ for $n>2m$, where A is the alphabet size. The upper bound and lower bound time complexities of single pattern matching are discussed in [Galil et al., 91] and [Galil et al., 92].

The minimum number of character comparisons needed to determine all occurrences of a keyword is an interesting theoretical question. It has been considered by [Galil, 79] and [Guibas et al., 80], and

they have discussed some improvements to the BM algorithm for its worst-case behavior. Apostolico et al. ([Apostolico et al., 84] and [Apostolico et al., 86]) have presented a variant of the BM algorithm in which the number of character comparisons is at most $2n$, regardless of the number of occurrences of the pattern in the string. Sunday ([Sunday, 90]) has devised string matching methods that are generally faster than the BM algorithm. His faster method uses statistics of the language being scanned to determine the order in which character pairs are to be compared. In the paper [Smith, 91] the peformances of similar, language-independent algorithms are examined. Results comparable with those of language-based algorithms can be achieved with an adaptive technique. In terms of character comparisons, a faster algorithm than Sunday's is constructed by using the larger of two pattern shifts. Evaluating the theoretical time complexity of the BF, KMP, and BM algorithms, based on empirical data presented, Smit ([Smit, 82]) has shown that, in a general text editor operating on lines of text, the best solution is to use the BM algorithm for patterns longer than three characters and the BF algorithm in the other cases. The KMP algorithm may perform significantly better than the BF algorithm when comparing strings from a small alphabet, for example, binary strings. Some experiments in a distributed environment are presented in [Moller et al., 84]. Considering the length of patterns, the number of alphabets, and the uniformity of texts, there is a trade-off between time (on the average) and space in the original BM algorithm. Thus, the original algorithm has been analyzed extensively, and several variants of it have been introduced. An average-case analysis of the KR method is discussed in [Gonnet et al., 90]. Other theoretical and experimental considerations for single keyword matching are in [Arikawa, 81], [Collussi et al., 90], [Li, 84], [Liu, 81], [Miller et al., 88], [Slisenko, 80], [Waterman, 84], and [Zhu et al., 87]. Preprocessing and string matching techniques for a given text and pattern are discussed in [Naor, 91].

For incorrect preprocessing of the pattern based on the KMP algorithm, a corrected version can be found in [Rytter, 80]. For m being similar to n, Iyenger et al. ([Iyenger et al., 80]) have given a variant of the BM algorithm. A combination of the KMP and BM algorithms is presented in [Semba, 85], and the worst cost is proportional to $2n$.

Horspool ([Horspool, 80]) has presented a simplification of the BM algorithm, and based on empirical results has shown that this simpler version is as good as the original one. The simplified version is obtained by using only the match heuristic. The main reason behind this simplification is that, in practice, the occurrence table does not make much contribution to the overall speed. The only purpose of this table is to optimize the handling of repetitive patterns (such as *xabcyyabc*) and so to avoid the worst-case running time, $O(mn)$. Since repetitive patterns are not common, it is not worthwhile to expend the considerable effort needed to set up the table. With this, the space depends only on the size of the alphabet (almost always fixed) and not on the length of the pattern (variable).

Baeza-Yates ([Baeza-Yates, 89b]) has improved the average time of the BM algorithm using extra space. This improvement is accomplished by applying a transformation that practically increases the size of the alphabet in use. The improvement is such that for long patterns an algorithm more than 50 percent faster than the original can be obtained. In this paper different heuristics are discussed that improve the search time based on the probability distribution of the symbols in the alphabet used. Schaback ([Schaback, 88]) has also analyzed the expected performance of some variants of the BM algorithm.

Horspool's implementation performs extremely well when we search for a random pattern in a random text. In practice, however, neither the pattern nor the text is random; there exist strong dependencies between successive symbols. Raita ([Raita, 92]) has suggested that it is not profitable to compare the pattern symbols strictly from right to left; if the last symbol of the pattern matches the corresponding text symbol, we should next try to match the first pattern symbol, because the dependencies are weakest between these two. His resulting code runs 25 percent faster than the best currently known routine.

Davies et al. ([Davies et al., 86]) have described four algorithms (BF, KMP, BM, and KR) of varying complexity used for pattern matching; and have investigated their behavior. Concluding from the

empirical evidence, the KMP algorithm should be used with a binary alphabet or with small patterns drawn from any other alphabet. The BM algorithm should be used in all other cases. Use of the BM algorithm may not be advisable, however, if the frequency at which the pattern is expected to be found is small, since the preprocessing time is in that case significant; similarly with the KMP algorithm, so the BF algorithm is better in that situation. Although the KR algorithm is linear in the number of references to the text string per characters passed, its substantially higher running time makes it unfeasible for pattern matching in strings. The advantage of this algorithm over the other three lies in its extension to two-dimensional pattern matching. It can be used for pattern recognition and image processing and thus in the expanding field of computer graphics. The extension will be discussed in Section 4.

Related problems

Cook ([Cook, 71]) has shown that a linear-time pattern matching algorithm exists for any set of strings that can be recognized by a two-way deterministic push-down automaton (2DPDA), even though the 2DPDA may spend more than linear time recognizing the set of strings. The string matching capabilities of other classes of automata, especially k-head finite automata, have been of theoretical interest to [Apostolico et al., 85], [Chrobak et al., 87], [Galil et al., 83], and [Li et al., 86].

Schemes in [Bean et al., 85], [Crochemore et al., 91], [Duval, 83], [Guibas et al., 81a], [Guibas et al., 81b], and [Lyndon et al., 62] have added new vigor to the study of periods and overlaps in strings and to the study of the combinatorics of patterns in strings. [Crochemore et al., 91] presents a new algorithm that can be viewed as an intermediate between the standard algorithms of the KMP and the BM. The algorithm is linear in time and uses constant space like the algorithm of [Galil et al., 83]. The algorithm relies on a previously known result in combinatorics on words, the critical factorization theorem, which relates the global period of a word to its local repetitions of blocks. The following results are presented in [Crochemore et al., 91].

1. It is linear in time $O(n+m)$, as KMP and BM, with a maximum number of letter comparisons bounded by $2n+5m$ compared to $2n+2m$ for KMP and $2n+f(m)$ for BM, where f depends on the version of their algorithm.
2. The minimum number of letter comparisons used during the search phase (executing the preprocessing of the pattern) is $2n/m$ compared to n for KMP and n/m for BM.
3. The memory space used, additional to the locations of the text and the pattern, is constant instead of $O(m)$ for both KMP and BM.

Parallel approaches of string matching are discussed in [Galil, 85], and an O(log log n) time parallel algorithm improving Galil's method is presented in [Breslauer et al., 90]. The paper [Breslauer et al., 92] describes the parallel complexity of the string matching problem using p processors for general alphabets. The other parallel matching algorithms are discussed in [Barkman et al., 89], [Kedam et al., 89], and [Viskin, 85].

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 a b*. 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, 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

$$\begin{array}{l} \qquad\qquad\quad a\ b\ c\ a\ b\ c\ a\ c\ a\ b \\ a\ b\ c\ a\ b\ c\ a\ c\ a\ b \\ \qquad\qquad\quad \uparrow \end{array}$$

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 $pattern[m+1] = \text{“@”}$ 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 $text[n+1] = \text{“}\perp\text{”}$ (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] := '⊥' ; text[n+2] := a;
        j := k := 1;
get started: comment j = 1;
        while text[k] ≠ 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 *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

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

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 *next*$[m+1]$ untouched, never changing it to -1, and to define integer arrays *head*$[1:m]$ and *link*$[1:n]$ initially zero, and to insert the code

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

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

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

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

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

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

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

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

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

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

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

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 $k \geqq 0$, where $|\alpha_1\alpha_2| = p$ and $\alpha_2 \neq \varepsilon$. 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 $pattern[1] \ldots pattern[j-1] = \alpha$ ends with $pattern[1] \ldots pattern[i-1] = \beta$, we have

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

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

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

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

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

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
```

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:*

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

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

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 (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 $q \geqq 1$,

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

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 $m=2$, $n=2q+1$, $\sigma(2, 2q+1)=(ab)^qa$, 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. 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 $\alpha\alpha^R$. □

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 $\beta = \beta_{r+1} \ldots \beta_s$ 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* $= \alpha$ 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.) □

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 $1, 2, 4, \ldots, 2^k, \ldots$, 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 $(K+K_1)+(2K+K_1)+(4K+K_1)+\cdots+(|\alpha|K+K_1) < 2Kn+K_1 \log_2 n$ 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

$$S \to aS, \qquad \text{for each } a \text{ in the alphabet,}$$
$$S \to \alpha,$$

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 ≦ 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

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

(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×m−k;
j := m; t := m+1;
while j > 0 do
    begin
        f[j] := t;
        while t ≦ m and pattern[j] ≠ 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

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

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 $k = 1, 2, \ldots$. 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 $k \to k'$ 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 $k_i \to k'$ 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

state q	known characters	$t[q]$	$s[q,x], q'[q,x]$ $x=a$	$x=b$	$x=c$	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

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

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 ≦ 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. 2
[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.

Programming Techniques
G. Manacher, S.L. Graham Editors

A Fast String Searching Algorithm

Robert S. Boyer
Stanford Research Institute
J Strother Moore
Xerox Palo Alto Research Center

An algorithm is presented that searches for the location, "*i*," of the first occurrence of a character string, "*pat*," in another string, "*string*." During the search operation, the characters of *pat* are matched starting with the last character of *pat*. The information gained by starting the match at the end of the pattern often allows the algorithm to proceed in large jumps through the text being searched. Thus the algorithm has the unusual property that, in most cases, not all of the first *i* characters of *string* are inspected. The number of characters actually inspected (on the average) decreases as a function of the length of *pat*. For a random English pattern of length 5, the algorithm will typically inspect *i*/4 characters of *string* before finding a match at *i*. Furthermore, the algorithm has been implemented so that (on the average) fewer than *i* + *patlen* machine instructions are executed. These conclusions are supported with empirical evidence and a theoretical analysis of the average behavior of the algorithm. The worst case behavior of the algorithm is linear in *i* + *patlen*, assuming the availability of array space for tables linear in *patlen* plus the size of the alphabet.

Key Words and Phrases: bibliographic search, computational complexity, information retrieval, linear time bound, pattern matching, text editing

CR Categories: 3.74, 4.40, 5.25

Reprinted from ***Communications of the ACM***, Volume 20, No. 10, October 1977, pp. 62-72.

Authors' present addresses: R.S. Boyer, Computer Science Laboratory, Stanford Research Institute, Menlo Park, CA 94025. This work was partially supported by ONR Contract N00014-75-C-0816; JS. Moore was in the Computer Science Laboratory, Xerox Palo Alto Research Center, Palo Alto, CA 94304, when this work was done. His current address is Computer Science Laboratory, SRI International, Menlo Park, CA 94025.

1. Introduction

Suppose that *pat* is a string of length *patlen* and we wish to find the position *i* of the leftmost character in the first occurrence of *pat* in some string *string*:

```
pat:          AT-THAT
string:  ... WHICH-FINALLY-HALTS.--AT-THAT-POINT ...
i:                                 ↑
```

The obvious search algorithm considers each character position of *string* and determines whether the successive *patlen* characters of *string* starting at that position match the successive *patlen* characters of *pat*. Knuth, Morris, and Pratt [4] have observed that this algorithm is quadratic. That is, in the worst case, the number of comparisons is on the order of *i* * *patlen*.[1]

Knuth, Morris, and Pratt have described a linear search algorithm which preprocesses *pat* in time linear in *patlen* and then searches *string* in time linear in *i* + *patlen*. In particular, their algorithm inspects each of the first *i* + *patlen* − 1 characters of *string* precisely once.

We now present a search algorithm which is usually "sublinear": It may not inspect each of the first *i* + *patlen* − 1 characters of *string*. By "usually sublinear" we mean that the expected value of the number of inspected characters in *string* is $c * (i + patlen)$, where $c < 1$ and gets smaller as *patlen* increases. There are patterns and strings for which worse behavior is exhibited. However, Knuth, in [5], has shown that the algorithm is linear even in the worst case.

The actual number of characters inspected depends on statistical properties of the characters in *pat* and *string*. However, since the number of characters inspected on the average decreases as *patlen* increases, our algorithm actually speeds up on longer patterns.

Furthermore, the algorithm is sublinear in another sense: It has been implemented so that on the average it requires the execution of fewer than *i* + *patlen* machine instructions per search.

The organization of this paper is as follows: In the next two sections we give an informal description of the algorithm and show an example of how it works. We then define the algorithm precisely and discuss its efficient implementation. After this discussion we present the results of a thorough test of a particular machine code implementation of our algorithm. We compare these results to similar results for the Knuth, Morris, and Pratt algorithm and the simple search algorithm. Following this empirical evidence is a theoretical analysis which accurately predicts the performance measured. Next we describe some situations in which it may not be advantageous to use our algorithm. We conclude with a discussion of the history of our algorithm.

[1] The quadratic nature of this algorithm appears when initial substrings of *pat* occur often in *string*. Because this is a relatively rare phenomenon in string searches over English text, this simple algorithm is *practically* linear in *i* + *patlen* and therefore acceptable for most applications.

2. Informal Description

The basic idea behind the algorithm is that more information is gained by matching the pattern from the right than from the left. Imagine that *pat* is placed on top of the left-hand end of *string* so that the first characters of the two strings are aligned. Consider what we learn if we fetch the *patlen*th character, *char*, of *string*. This is the character which is aligned with the *last* character of *pat*.

Observation 1. If *char* is known not to occur in *pat*, then we know we need not consider the possibility of an occurrence of *pat* starting at *string* positions 1, 2, . . . or *patlen*: Such an occurrence would require that *char* be a character of *pat*.

Observation 2. More generally, if the last (rightmost) occurrence of *char* in *pat* is $delta_1$ characters from the right end of *pat*, then we know we can slide *pat* down $delta_1$ positions without checking for matches. The reason is that if we were to move *pat* by less than $delta_1$, the occurrence of *char* in *string* would be aligned with some character it could not possibly match: Such a match would require an occurrence of *char* in *pat* to the right of the rightmost.

Therefore unless *char* matches the last character of *pat* we can move past $delta_1$ characters of *string* without looking at the characters skipped; $delta_1$ is a function of the character *char* obtained from *string*. If *char* does not occur in *pat*, $delta_1$ is *patlen*. If *char* does occur in *pat*, $delta_1$ is the difference between *patlen* and the position of the rightmost occurrence of *char* in *pat*.

Now suppose that *char* matches the last character of *pat*. Then we must determine whether the previous character in *string* matches the second from the last character in *pat*. If so, we continue backing up until we have matched all of *pat* (and thus have succeeded in finding a match), or else we come to a mismatch at some new *char* after matching the last m characters of *pat*.

In this latter case, we wish to shift *pat* down to consider the next plausible juxtaposition. Of course, we would like to shift it as far down as possible.

Observation 3(*a*). We can use the same reasoning described above—based on the mismatched character *char* and $delta_1$—to slide *pat* down k so as to align the two known occurrences of *char*. Then we will want to inspect the character of *string* aligned with the last character of *pat*. Thus we will actually shift our attention down *string* by $k + m$. The distance k we should slide *pat* depends on where *char* occurs in *pat*. If the rightmost occurrence of *char* in *pat* is to the right of the mismatched character (i.e. within that part of *pat* we have already passed) we would have to move *pat* backwards to align the two known occurrences of *char*. We would not want to do this. In this case we say that $delta_1$ is "worthless" and slide *pat* forward by $k = 1$ (which is always sound). This shifts our attention down *string* by $1 + m$. If the rightmost occurrence of *char* in *pat* is to the left of the mismatch, we can slide forward by $k = delta_1(char) - m$ to align the two occurrences of *char*. This shifts our attention down *string* by $delta_1(char) - m + m = delta_1(char)$.

However, it is possible that we can do better than this.

Observation 3(*b*). We know that the next m characters of *string* match the final m characters of *pat*. Let this substring of *pat* be *subpat*. We also know that this occurrence of *subpat* in *string* is preceded by a character (*char*) which is different from the character preceding the terminal occurrence of *subpat* in *pat*. Roughly speaking, we can generalize the kind of reasoning used above and slide *pat* down by some amount so that the discovered occurrence of *subpat* in *string* is aligned with the rightmost occurrence of *subpat* in *pat* which is not preceded by the character preceding its terminal occurrence in *pat*. We call such a reoccurrence of *subpat* in *pat* a "plausible reoccurrence." The reason we said "roughly speaking" above is that we must allow for the rightmost plausible reoccurrence of *subpat* to "fall off" the left end of *pat*. This is made precise later.

Therefore, according to Observation 3(b), if we have matched the last m characters of *pat* before finding a mismatch, we can move *pat* down by k characters, where k is based on the position in *pat* of the rightmost plausible reoccurrence of the terminal substring of *pat* having m characters. After sliding down by k, we want to inspect the character of *string* aligned with the last character of *pat*. Thus we actually shift our attention down *string* by $k + m$ characters. We call this distance $delta_2$, and we define $delta_2$ as a function of the position j in *pat* at which the mismatch occurred. k is just the distance between the terminal occurrence of *subpat* and its rightmost plausible reoccurrence and is always greater than or equal to 1. m is just $patlen - j$.

In the case where we have matched the final m characters of *pat* before failing, we clearly wish to shift our attention down *string* by $1 + m$ or $delta_1(char)$ or $delta_2(j)$, according to whichever allows the largest shift. From the definition of $delta_2$ as $k + m$ where k is always greater than or equal to 1, it is clear that $delta_2$ is at least as large as $1 + m$. Therefore we can shift our attention down *string* by the maximum of just the two *delta*s. This rule also applies when $m = 0$ (i.e. when we have not yet matched any characters of *pat*), because in that case $j = patlen$ and $delta_2(j) \geq 1$.

3. Example

In the following example we use an "↑" under *string* to indicate the current *char*. When this "pointer" is pushed to the right, imagine that it drags the right end of *pat* with it (i.e. imagine *pat* has a hook on its right end). When the pointer is moved to the left, keep *pat* fixed with respect to *string*.

```
pat:          AT-THAT
string:  ... WHICH-FINALLY-HALTS.--AT-THAT-POINT ...
                    ↑
```

Since "F" is known not to occur in *pat*, we can appeal to Observation 1 and move the pointer (and thus *pat*) down by 7:

```
pat:                  AT-THAT
string:  ... WHICH-FINALLY-HALTS.--AT-THAT-POINT ...
                        ↑
```

Appealing to Observation 2, we can move the pointer down 4 to align the two hyphens:

```
pat:                      AT-THAT
string:  ... WHICH-FINALLY-HALTS.--AT-THAT-POINT ...
                            ↑
```

Now *char* matches its opposite in *pat*. Therefore we step left by one:

```
pat:                      AT-THAT
string:  ... WHICH-FINALLY-HALTS.--AT THAT POINT ...
                            ↑
```

Appealing to Observation 3(a), we can move the pointer to the right by 7 positions because "L" does not occur in *pat*.[2] Note that this only moves *pat* to the right by 6.

```
pat:                          AT-THAT
string:  ... WHICH-FINALLY-HALTS.--AT-THAT-POINT ...
                                ↑
```

Again *char* matches the last character of *pat*. Stepping to the left we see that the previous character in *string* also matches its opposite in *pat*. Stepping to the left a second time produces:

```
pat:                          AT-THAT
string:  ... WHICH-FINALLY-HALTS.--AT-THAT-POINT ...
                              ↑
```

Noting that we have a mismatch, we appeal to Observation 3(b). The $delta_2$ move is best since it allows us to push the pointer to the right by 7 so as to align the discovered substring "AT" with the beginning of *pat*.[3]

```
pat:                              AT-THAT
string:  ... WHICH-FINALLY-HALTS.--AT-THAT-POINT ...
                                  ↑
```

This time we discover that each character of *pat* matches the corresponding character in *string* so we have found the pattern. Note that we made only 14 references to *string*. Seven of these were required to confirm the final match. The other seven allowed us to move past the first 22 characters of *string*.

[2] Note that $delta_2$ would allow us to move the pointer to the right only 4 positions in order to align the discovered substring "T" in *string* with its second from last occurrence at the beginning of the word "THAT" in *pat*.

[3] The $delta_1$ move only allows the pointer to be pushed to the right by 4 to align the hyphens.

4. The Algorithm

We now specify the algorithm. The notation *pat*(*j*) refers to the *j*th character in *pat* (counting from 1 on the left).

We assume the existence of two tables, $delta_1$ and $delta_2$. The first has as many entries as there are characters in the alphabet. The entry for some character *char* will be denoted by $delta_1(char)$. The second table has as many entries as there are character positions in the pattern. The *j*th entry will be denoted by $delta_2(j)$. Both tables contain non-negative integers.

The tables are initialized by preprocessing *pat*, and their entries correspond to the values $delta_1$ and $delta_2$ referred to earlier. We will specify their precise contents after it is clear how they are to be used.

Our search algorithm may be specified as follows:

```
        stringlen ← length of string.
        i ← patlen.
top:    if i > stringlen then return false.
        j ← patlen.
loop:   if j = 0 then return j + 1.
        if string(i) = pat(j)
          then
          j ← j − 1.
          i ← i − 1.
          goto loop.
          close;
        i ← i + max(delta1 (string(i)), delta2 (j)).
        goto top.
```

If the above algorithm returns false, then *pat* does not occur in *string*. If the algorithm returns a number, then it is the position of the left end of the first occurrence of *pat* in *string*.

The $delta_1$ table has an entry for each character *char* in the alphabet. The definition of $delta_1$ is:

$delta_1(char)$ = If *char* does not occur in *pat*, then *patlen*; else *patlen* − *j*, where *j* is the maximum integer such that *pat*(*j*) = *char*.

The $delta_2$ table has one entry for each of the integers from 1 to *patlen*. Roughly speaking, $delta_2(j)$ is (a) the distance we can slide *pat* down so as to align the discovered occurrence (in *string*) of the last *patlen* − *j* characters of *pat* with its rightmost plausible reoccurrence, plus (b) the additional distance we must slide the "pointer" down so as to restart the process at the right end of *pat*. To define $delta_2$ precisely we must define the rightmost plausible reoccurrence of a terminal substring of *pat*. To this end let us make the following conventions: Let \$ be a character that does not occur in *pat* and let us say that if *i* is less than 1 then *pat*(*i*) is \$. Let us also say that two sequences of characters $[c_1 \ldots c_n]$ and $[d_1 \ldots d_n]$ "unify" if for all *i* from 1 to *n* either $c_i = d_i$ or c_i = \$ or d_i = \$.

Finally, we define the position of the rightmost plausible reoccurrence of the terminal substring which starts at position *j* + 1, *rpr*(*j*), for *j* from 1 to *patlen*, to be the greatest *k* less than or equal to *patlen* such that

[*pat*(*j* + 1) . . . *pat*(*patlen*)] and [*pat*(*k*) . . . *pat*(*k* + *patlen* − *j* − 1)] unify and either $k \leq 1$ or $pat(k-1) \neq pat(j)$.[4] (That is, the position of the rightmost plausible reoccurrence of the substring *subpat*, which starts at *j* + 1, is the rightmost place where *subpat* occurs in *pat* and is not preceded by the character *pat*(*j*) which precedes its terminal occurrence—with suitable allowances for either the reoccurrence or the preceding character to fall beyond the left end of *pat*. Note that *rpr*(*j*) may be negative because of these allowances.)

Thus the distance we must slide *pat* to align the discovered substring which starts at *j* + 1 with its rightmost plausible reoccurrence is *j* + 1 − *rpr*(*j*). The distance we must move to get back to the end of *pat* is just *patlen* − *j*. $delta_2(j)$ is just the sum of these two. Thus we define $delta_2$ as follows:

$$delta_2(j) = patlen + 1 - rpr(j).$$

To make this definition clear, consider the following two examples:

j:	1	2	3	4	5	6	7	8	9
pat:	A	B	C	X	X	X	A	B	C
$delta_2(j)$:	14	13	12	11	10	9	11	10	1

j:	1	2	3	4	5	6	7	8	9
pat:	A	B	Y	X	C	D	E	Y	X
$delta_2(j)$:	17	16	15	14	13	12	7	10	1

5. Implementation Considerations

The most frequently executed part of the algorithm is the code that embodies Observations 1 and 2. The following version of our algorithm is equivalent to the original version provided that $delta_0$ is a table containing the same entries as $delta_1$ except that $delta_0(pat(patlen))$ is set to an integer *large* which is greater than *stringlen* + *patlen* (while $delta_1(pat(patlen))$ is always 0).

```
        stringlen ← length of string.
        i ← patlen.
        if i > stringlen then return false.
fast:   i ← i + delta0(string(i)).
        if i ≤ stringlen then goto fast.
undo:   if i ≤ large then return false.
        i ← (i − large) − 1.
        j ← patlen − 1.
slow:   if j = 0 then return i + 1.
        if string(i) = pat(j)
          then
          j ← j − 1.
          i ← i − 1.
          goto slow.
          close;
        i ← i + max(delta1(string(i)), delta2(j)).
        goto fast.
```

[4] Note that when *j* = *patlen*, the two sequences [*pat*(*patlen* + 1) . . . *pat*(*patlen*)] and [*pat*(*k*) . . . *pat*(*k* − 1)] are empty and therefore unify. Thus, *rpr*(*patlen*) is simply the greatest *k* less than or equal to *patlen* such that $k \leq 1$ or $pat(k-1) \neq pat(patlen)$.

Of course we do not actually have two versions of $delta_1$. Instead we use only $delta_0$, and in place of $delta_1$ in the *max* expression we merely use the $delta_0$ entry unless it is *large* (in which case we use 0).

Note that the *fast* loop just scans down *string*, effectively looking for the last character *pat*(*patlen*) in *pat*, skipping according to $delta_1$. ($delta_2$ can be ignored in this case since no terminal substring has yet been matched, i.e. $delta_2(patlen)$ is always less than or equal to the corresponding $delta_1$.) Control leaves this loop only when *i* exceeds *stringlen*. The test at *undo* decides whether this situation arose because all of *string* has been scanned or because *pat*(*patlen*) was hit (which caused *i* to be incremented by *large*). If the first case obtains, *pat* does not occur in *string* and the algorithm returns false. If the second case obtains, then *i* is restored (by subtracting *large*) and we enter the *slow* loop which backs up checking for matches. When a mismatch is found we skip ahead by the maximum of the original $delta_1$ and $delta_2$ and reenter the *fast* loop. We estimate that 80 percent of the time spent in searching is spent in the *fast* loop.

The *fast* loop can be coded in four machine instructions:

```
fast:  char ← string(i).
       i ← i + delta0(char).
       skip the next instruction if i > stringlen.
       goto fast.
undo: . . .
```

We have implemented this algorithm in PDP-10 assembly language. In our implementation we have reduced the number of instructions in the *fast* loop to three by translating *i* down by *stringlen*; we can then test *i* against 0 and conditionally jump to *fast* in one instruction.

On a byte addressable machine it is easy to implement "*char* ← *string*(*i*)" and "*i* ← *i* + $delta_0$(*char*)" in one instruction each. Since our implementation was in PDP-10 assembly language we had to employ byte pointers to access characters in *string*. The PDP-10 instruction set provides an instruction for incrementing a byte pointer by one but not by other amounts. Our code therefore employs an array of 200 indexing byte pointers which we use to access characters in *string* in one indexed instruction (after computing the index) at the cost of a small (five-instruction) overhead every 200 characters. It should be noted that this trick only makes up for the lack of direct byte addressing; one can expect our algorithm to run somewhat faster on a byte-addressable machine.

6. Empirical Evidence

We have exhaustively tested the above PDP-10 implementation on random test data. To gather the test patterns we wrote a program which randomly selects a substring of a given length from a source string. We used this program to select 300 patterns of

length *patlen*, for each *patlen* from 1 to 14. We then used our algorithm to search for each of the test patterns in its source string, starting each search in a random position somewhere in the first half of the source string. All of the characters for both the patterns and the strings were in primary memory (rather than a secondary storage medium such as a disk).

We measured the cost of each search in two ways: the number of references made to *string* and the total number of machine instructions that actually got executed (ignoring the preprocessing to set up the two tables).

By dividing the number of references to *string* by the number of characters $i - 1$ passed before the pattern was found (or *string* was exhausted), we obtained the number of references to *string* per character passed. This measure is independent of the particular implementation of the algorithm. By dividing the number of instructions executed by $i - 1$, we obtained the average number of instructions spent on each character passed. This measure depends upon the implementation, but we feel that it is meaningful since the implementation is a straightforward encoding of the algorithm as described in the last section.

We then averaged these measures across all 300 samples for each pattern length.

Because the performance of the algorithm depends upon the statistical properties of *pat* and *string* (and hence upon the properties of the source string from which the test patterns were obtained), we performed this experiment for three different kinds of source strings, each of length 10,000. The first source string consisted of a random sequence of 0's and 1's. The second source string was a piece of English text obtained from an online manual. The third source string was a random sequence of characters from a 100-character alphabet.

In Figure 1 the average number of references to *string* per character in *string* passed is plotted against the pattern length for each of three source strings. Note that the number of references to *string* per character passed is less than 1. For example, for an English pattern of length 5, the algorithm typically inspects 0.24 characters for every character passed. That is, for every reference to *string* the algorithm passes about 4 characters, or, equivalently, the algorithm inspects only about a quarter of the characters it passes when searching for a pattern of length 5 in an English text string. Furthermore, the number of references per character drops as the patterns get longer. This evidence supports the conclusion that the algorithm is "sublinear" in the number of references to *string*.

For comparison, it should be noted that the Knuth, Morris, and Pratt algorithm references *string* precisely 1 time per character passed. The simple search algorithm references *string* about 1.1 times per character passed (determined empirically with the English sample above).

Fig. 1.

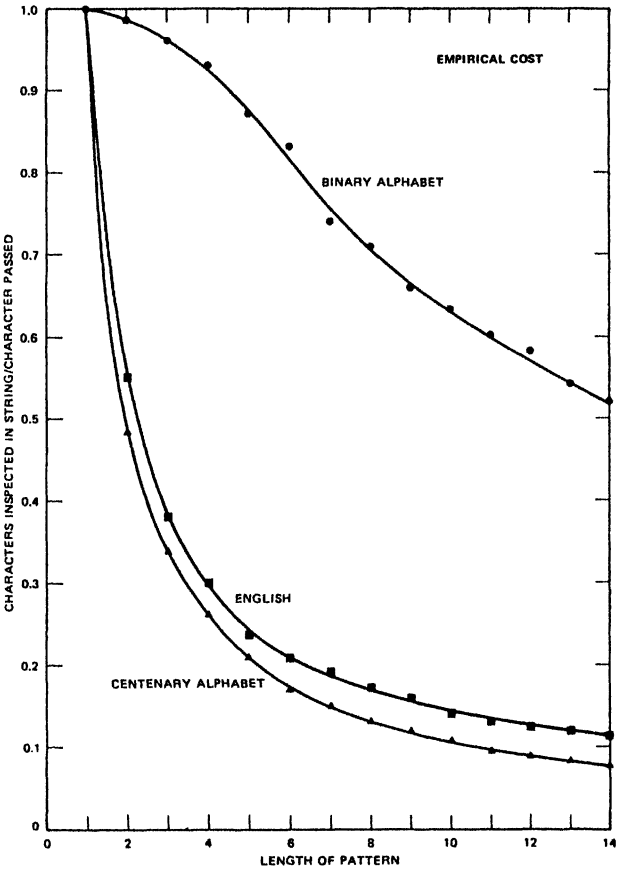
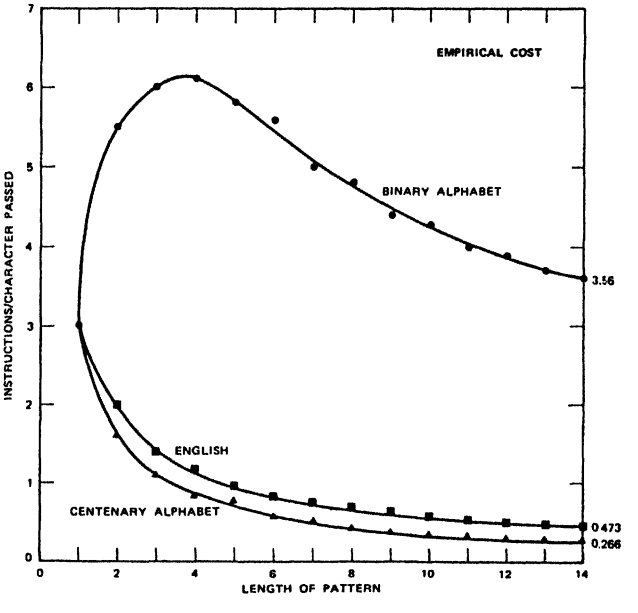


Fig. 2



In Figure 2 the average number of instructions executed per character passed is plotted against the pattern length. The most obvious feature to note is that the search speeds up as the patterns get longer. That is, the total number of instructions executed in order to pass over a character decreases as the length of the pattern increases.

Figure 2 also exhibits a second interesting feature of our implementation of the algorithm: For sufficiently

large alphabets and sufficiently long patterns the algorithm executes fewer than 1 instruction per character passed. For example, in the English sample, less than 1 instruction per character is executed for patterns of length 5 or more. Thus this implementation is "sublinear" in the sense that it executes fewer than i + *patlen* instructions before finding the pattern at i. This means that no algorithm which references each character it passes could *possibly* be faster than ours in these cases (assuming it takes at least one instruction to reference each character).

The best alternative algorithm for finding a single substring is that of Knuth, Morris, and Pratt. If that algorithm is implemented in the extraordinarily efficient way described in [4, pp. 11–12] and [2, Item 179],[5] then the cost of looking at a character can be expected to be at least $3 - p$ instructions, where p is the probability that a character just fetched from *string* is equal to a given character of *pat*. Hence a horizontal line at $3 - p$ instructions/character represents the best (and, practically, the worst) the Knuth, Morris, and Pratt algorithm can achieve.

The simple string searching algorithm (when coded with a 3-instruction fast loop[6]) executes about 3.3 instructions per character (determined empirically on the English sample above).

As noted, the preprocessing time for our algorithm (and for Knuth, Morris, and Pratt) has been ignored. The cost of this preprocessing can be made linear in *patlen* (this is discussed further in the next section) and is trivial compared to a reasonably long search. We made no attempt to code this preprocessing efficiently. However, the average cost (in our implementation) ranges from 160 instructions (for strings of length 1) to about 500 instructions (for strings of length 14). It should be explained that our code uses a block transfer instruction to clear the 128-word $delta_1$ table at the beginning of the preprocessing, and we have counted this single instruction as though it were 128 instructions. This accounts for the unexpectedly large instruction count for preprocessing a one-character pattern.

7. Theoretical Analysis

The preprocessing for $delta_1$ requires an array the size of the alphabet. Our implementation first initializes all entries of this array to *patlen* and then sets up $delta_1$ in a linear scan through the pattern. Thus our preprocessing for $delta_1$ is linear in *patlen* plus the size of the alphabet.

At a slight loss of efficiency in the search speed one could eliminate the initialization of the $delta_1$ array by storing with each entry a key indicating the number of times the algorithm has previously been called. This approach still requires initializing the array the first time the algorithm is used.

To implement our algorithm for extremely large alphabets, one might implement the $delta_1$ table as a hash array. In the worst case, accessing $delta_1$ during the search itself could require order *patlen* instructions, significantly impairing the speed of the algorithm. Hence the algorithm as it stands almost certainly does not run in time linear in i + *patlen* for infinite alphabets.

Knuth, in analyzing the algorithm, has shown that it still runs in linear time when $delta_1$ is omitted, and this result holds for infinite alphabets. Doing this, however, will drastically degrade the performance of the algorithm on the average. In [5] Knuth exhibits an algorithm for setting up $delta_2$ in time linear in *patlen*.

From the preceding empirical evidence, the reader can conclude that the algorithm is quite good in the average case. However, the question of its behavior in the worst case is nontrivial. Knuth has recently shed some light on this question. In [5] he proves that the execution of the algorithm (after preprocessing) is linear in i + *patlen*, assuming the availability of array space linear in *patlen* plus the size of the alphabet. In particular, he shows that in order to discover that *pat* does not occur in the first i characters of *string*, at most $6 * i$ characters from *string* are matched with characters in *pat*. He goes on to say that the constant 6 is probably much too large, and invites the reader to improve the theorem. His proof reveals that the linearity of the algorithm is entirely due to $delta_2$.

We now analyze the average behavior of the algorithm by presenting a probabilistic model of its performance. As will become clear, the results of this analysis will support the empirical conclusions that the algorithm is usually "sublinear" both in the number of references to *string* and the number of instructions executed (for our implementation).

The analysis below is based on the following simplifying assumption: Each character of *pat* and *string* is an independent random variable. The probability that a character from *pat* or *string* is equal to a given character of the alphabet is p.

Imagine that we have just moved *pat* down *string* to a new position and that this position does not yield a match. We want to know the expected value of the ratio between the cost of discovering the mismatch and the distance we get to slide *pat* down upon finding the mismatch. If we define the cost to be the total number of references made to *string* before discovering the mismatch, we can obtain the expected value of the average number of references to string per character

[5] This implementation automatically compiles *pat* into a machine code program which implicitly has the skip table built in and which is executed to perform the search itself. In [2] they compile code which uses the PDP-10 capability of fetching a character and incrementing a byte address in one instruction. This compiled code executes at least two or three instructions per character fetched from *string*, depending on the outcome of a comparison of the character to one from *pat*.

[6] This loop avoids checking whether *string* is exhausted by assuming that the first character of *pat* occurs at the end of *string*. This can be arranged ahead of time. The loop actually uses the same three instruction codes used by the above-referenced implementation of the Knuth, Morris, and Pratt algorithm.

passed. If we define the cost to be the total number of machine instructions executed in discovering the mismatch, we can obtain the expected value of the number of instructions executed per character passed.

In the following we say "only the last m characters of *pat* match" to mean "the last m characters of *pat* match the corresponding m characters in *string* but the $(m + 1)$-th character from the right end of *pat* fails to match the corresponding character in *string*."

The expected value of the ratio of cost to characters passed is given by:

$$\left(\sum_{m=0}^{patlen-1} cost(m) * prob(m)\right) \Bigg/ \left(\sum_{m=0}^{patlen-1} prob(m) * \left(\sum_{k=1}^{patlen} skip(m,k) * k\right)\right)$$

where $cost(m)$ is the cost associated with discovering that only the last m characters of *pat* match; $prob(m)$ is the probability that only the last m characters of *pat* match; and $skip(m,k)$ is the probability that, supposing only the last m characters of *pat* match, we will get to slide *pat* down by k.

Under our assumptions, the probability that only the last m characters of *pat* match is:

$$prob(m) = p^m(1 - p)/(1 - p^{patlen}).$$

(The denominator is due to the assumption that a mismatch exists.)

The probability that we will get to slide *pat* down by k is determined by analyzing how i is incremented. However, note that even though we increment i by the maximum *max* of the two *deltas*, this will actually only slide *pat* down by $max - m$, since the increment of i also includes the m necessary to shift our attention back to the end of *pat*. Thus when we analyze the contributions of the two *deltas* we speak of the amount by which they allow us to slide *pat* down, rather than the amount by which we increment i. Finally, recall that if the mismatched character *char* occurs in the already matched final m characters of *pat*, then $delta_1$ is worthless and we always slide by $delta_2$. The probability that $delta_1$ is worthless is just $(1 - (1 - p)^m)$. Let us call this $probdelta_1worthless(m)$.

The conditions under which $delta_1$ will naturally let us slide forward by k can be broken down into four cases as follows: (a) $delta_1$ will let us slide down by 1 if *char* is the $(m + 2)$-th character from the righthand end of *pat* (or else there are no more characters in *pat*) and *char* does not occur to the right of that position (which has probability $(1 - p)^m$ * (if $m + 1 = patlen$ then 1 else p)). (b) $delta_1$ allows us to slide down k, where $1 < k < patlen - m$, provided the rightmost occurrence of *char* in *pat* is $m + k$ characters from the right end of *pat* (which has probability $p * (1 - p)^{k+m-1}$). (c) When $patlen - m > 1$, $delta_1$ allows us to slide past $patlen - m$ characters if *char* does not occur in *pat* at all (which has probability $(1 - p)^{patlen-1}$ given that we know *char* is not the $(m + 1)$-th character from the right end of *pat*). Finally, (d) $delta_1$ never allows a slide longer than $patlen - m$ (since the maximum value of $delta_1$ is *patlen*).

Thus we can define the probability $probdelta_1(m, k)$ that when only the last m characters of *pat* match, $delta_1$ will allow us to move down by k as follows:

$$\begin{aligned} probdelta_1(m, k) = {} & \text{if } k = 1 \\ & \quad \text{then} \\ & \quad (1 - p)^m * (\text{if } m + 1 = patlen \text{ then } 1 \text{ else } p); \\ & \text{elseif } 1 < k < patlen - m \text{ then } p * (1 - p)^{k+m-1}; \\ & \text{elseif } k = patlen - m \text{ then } (1 - p)^{patlen-1}; \\ & \text{else (i.e. } k > patlen - m) \ 0. \end{aligned}$$

(It should be noted that we will not put these formulas into closed form, but will simply evaluate them to verify the validity of our empirical evidence.)

We now perform a similar analysis for $delta_2$; $delta_2$ lets us slide down by k if (a) doing so sets up an alignment of the discovered occurrence of the last m characters of *pat* in *string* with a plausible reoccurrence of those m characters elsewhere in *pat*, and (b) no smaller move will set up such an alignment. The probability $probpr(m, k)$ that the terminal substring of *pat* of length m has a plausible reoccurrence k characters to the left of its first character is:

$$\begin{aligned} probpr(m, k) = {} & \text{if } m + k < patlen \\ & \quad \text{then } (1 - p) * p^m \\ & \quad \text{else } p^{patlen-k} \end{aligned}$$

Of course, k is just the distance $delta_2$ lets us slide provided there is no earlier reoccurrence. We can therefore define the probability $probdelta_2(m, k)$ that, when only the last m characters of *pat* match, $delta_2$ will allow us to move down by k recursively as follows:

$$probdelta_2(m, k) = probpr(m, k)\left(1 - \sum_{n=1}^{k-1} probdelta_2(m, n)\right).$$

We slide down by the maximum allowed by the two *deltas* (taking adequate account of the possibility that $delta_1$ is worthless). If the values of the *deltas* were independent, the probability that we would actually slide down by k would just be the sum of the products of the probabilities that one of the *deltas* allows a move of k while the other allows a move of less than or equal to k.

However, the two moves are not entirely independent. In particular, consider the possibility that $delta_1$ is worthless. Then the *char* just fetched occurs in the last m characters of *pat* and does not match the $(m + 1)$-th. But if $delta_2$ gives a slide of 1 it means that sliding these m characters to the left by 1 produces a match. This implies that all of the last m characters of *pat* are equal to the character $m + 1$ from the right. But this character is known not to be *char*. Thus *char* cannot occur in the last m characters of *pat*, violating the hypothesis that $delta_1$ was worthless. Therefore if $delta_1$ is worthless, the probability that $delta_2$ specifies a skip of 1 is 0 and the probability that it specifies one of the larger skips is correspondingly increased.

This interaction between the two *delta*s is also felt (to a lesser extent) for the next m possible $delta_2$'s, but we ignore these (and in so doing accept that our analysis may predict slightly worse results than might be expected since we allow some short $delta_2$ moves when longer ones would actually occur).

The probability that $delta_2$ will allow us to slide down by k when only the last m characters of *pat* match, assuming that $delta_1$ is worthless, is:

$$probdelta_2'(m,k) = \text{if } k = 1 \text{ then } 0 \text{ else } probpr(m,k)\left(1 - \sum_{n=2}^{k-1} probdelta_2'(m,n)\right).$$

Finally, we can define $skip(m, k)$, the probability that we will slide down by k if only the last m characters of *pat* match:

$$\begin{aligned} skip(m,k) = \text{if } k = 1 \text{ then } & probdelta_1(m,1) * probdelta_2(m,1) \\ \text{else } & probdelta_1 worthless(m) * probdelta_2'(m,k) \\ & + \sum_{n=1}^{k-1} probdelta_1(m,k) * probdelta_2(m,n) \\ & + \sum_{n=1}^{k-1} probdelta_1(m,n) * probdelta_2(m,k) \\ & + probdelta_1(m,k) * probdelta_2(m,k). \end{aligned}$$

Now let us consider the two alternative *cost* functions. In order to analyze the number of references to *string* per character passed over, $cost(m)$ should just be $m + 1$, the number of references necessary to confirm that only the last m characters of *pat* match.

In order to analyze the number of instructions executed per character passed over, $cost(m)$ should be the total number of instructions executed in discovering that only the last m characters of *pat* match. By inspection of our PDP-10 code:

$$cost(m) = \text{if } m = 0 \text{ then } 3 \text{ else } 12 + 6m.$$

We have computed the expected value of the ratio of cost per character skipped by using the above formulas (and both definitions of *cost*). We did so for pattern lengths running from 1 to 14 (as in our empirical evidence) and for the values of p appropriate for the three source strings used: For a random binary string p is 0.5, for an arbitrary English string it is (approximately) 0.09, and for a random string over a 100-character alphabet it is 0.01. The value of p for English was determined using a standard frequency count for the alphabetic characters [3] and empirically determining the frequency of space, carriage return, and line feed to be 0.23, 0.03, and 0.03, respectively.[7]

In Figure 3 we have plotted the theoretical ratio of references to *string* per character passed over against

Fig. 3.

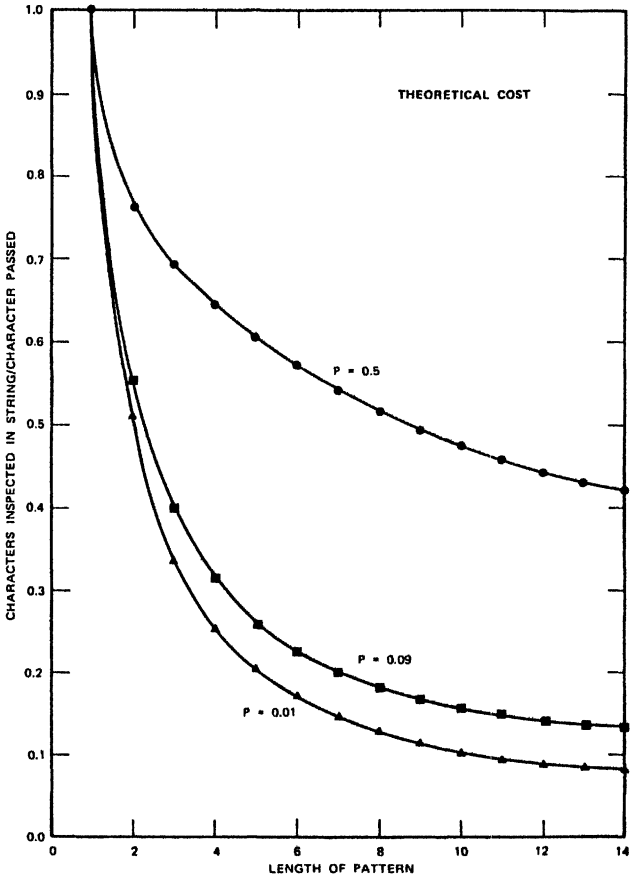
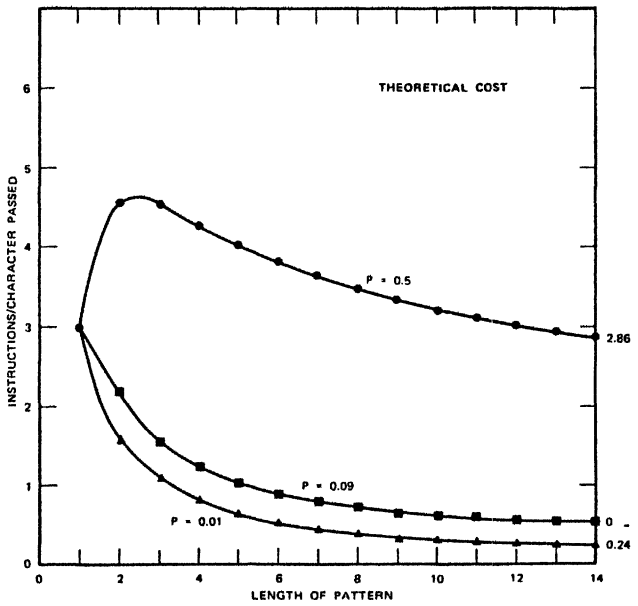


Fig. 4.



the pattern length. The most important fact to observe in Figure 3 is that the algorithm *can be expected* to make fewer than i + *patlen* references to *string* before finding the pattern at location i. For example, for English text strings of length 5 or greater, the algorithm may be expected to make less than $(i + 5)/4$ references to *string*. The comparable figure for the Knuth,

[7] We have determined empirically that the algorithm's performance on truly random strings where $p = 0.09$ is virtually identical to its performance on English strings. In particular, the reference count and instruction count curves generated by such random strings are almost coincidental with the English curves in Figures 1 and 2.

Morris, and Pratt algorithm is of course precisely i. The figure for the intuitive search algorithm is always greater than or equal to i.

The reason the number of references per character passed decreases more slowly as *patlen* increases is that for longer patterns the probability is higher that the character just fetched occurs somewhere in the pattern, and therefore the distance the pattern can be moved forward is shortened.

In Figure 4 we have plotted the theoretical ratio of the number of instructions executed per character passed versus the pattern length. Again we find that our implementation of the algorithm *can be expected* (for sufficiently large alphabets) to execute fewer than i + *patlen* instructions before finding the pattern at location i. That is, our implementation is usually "sublinear" even in the number of instructions executed. The comparable figure for the Knuth, Morris, and Pratt algorithm is at best $(3 - p) * (i + patlen - 1)$.[8] For the simple search algorithm the expected value of the number of instructions executed per character passed is (approximately) 3.28 (for $p = 0.09$).

It is difficult to fully appreciate the role played by $delta_2$. For example, if the alphabet is large and patterns are short, then computing and trying to use $delta_2$ probably does not pay off much (because the chances are high that a given character in *string* does not occur anywhere in *pat* and one will almost always stay in the *fast* loop ignoring $delta_2$).[9] Conversely, $delta_2$ becomes very important when the alphabet is small and the patterns are long (for now execution will frequently leave the *fast* loop; $delta_1$ will in general be small because many of the characters in the alphabet will occur in *pat* and only the terminal substring observations could cause large shifts). Despite the fact that it is difficult to appreciate the role of $delta_2$, it should be noted that the linearity result for the worst case behavior of the algorithm is due entirely to the presence of $delta_2$.

Comparing the empirical evidence (Figures 1 and 2) with the theoretical evidence (Figures 3 and 4, respectively), we note that the model is completely accurate for English and the 100-character alphabet. The model predicts much better behavior than we actually experience in the binary case. Our only explanation is that since $delta_2$ predominates in the binary alphabet and sets up alignments of the pattern and the string, the algorithm backs up over longer terminal substrings of the pattern before finding mismatches. Our analysis ignores this phenomenon.

However, in summary, the theoretical analysis supports the conclusion that on the average the algorithm is sublinear in the number of references to *string* and, for sufficiently large alphabets and patterns, sublinear in the number of instructions executed (in our implementation).

8. Caveat Programmer

It should be observed that the preceding analysis has assumed that *string* is entirely in primary memory and that we can obtain the ith character in it in one instruction after computing its byte address. However, if *string* is actually on secondary storage, then the characters in it must be read in.[10] This transfer will entail some time delay equivalent to the execution of, say, w instructions per character brought in, and (because of the nature of computer I/O) all of the first i + *patlen* − 1 characters will eventually be brought in whether *we* actually reference all of them or not. (A representative figure for w for paged transfers from a fast disk is 5 instructions/character.) Thus there may be a hidden cost of w instructions per character passed over.

According to the statistics presented above one might expect our algorithm to be approximately three times faster than the Knuth, Morris, and Pratt algorithm (for, say, English strings of length 6) since that algorithm executes about three instructions to our one. However, if the CPU is idle for the w instructions necessary to read each character, the actual ratios are closer to $w + 3$ instructions than to $w + 1$ instructions. Thus for paged disk transfers our algorithm can only be expected to be roughly $4/3$ faster (i.e. 5 + 3 instructions to 5 + 1 instructions) if we assume that we are idle during I/O. Thus for large values of w the difference between the various algorithms diminishes if the CPU is idle during I/O.

Of course, in general, programmers (or operating systems) try to avoid the situation in which the CPU is idle while awaiting an I/O transfer by overlapping I/O with some other computation. In this situation, the chances are that our algorithm will be I/O bound (we will search a page faster than it can be brought in), and indeed so will that of Knuth, Morris, and Pratt if $w > 3$. Our algorithm will require that fewer CPU cycles be devoted to the search itself so that if there are other jobs to perform, there will still be an overall advantage in using the algorithm.

---

[8] Although the Knuth, Morris, and Pratt algorithm will fetch each of the first i + *patlen* − 1 characters of *string* precisely once, sometimes a character is involved in several tests against characters in *pat*. The number of such tests (each involving three instructions) is bounded by $\log_\Phi(patlen)$, where Φ is the golden ratio.

[9] However, if the algorithm is implemented without $delta_2$, recall that, in exiting the *slow* loop, one must now take the *max* of $delta_1$ and *patlen* − j + 1 to allow for the possibility that $delta_1$ is worthless.

[10] We have implemented a version of our algorithm for searching through disk files. It is available as the subroutine FFILEPOS in the latest release of INTERLISP-10. This function uses the TENEX page mapping capability to identify one file page at a time with a buffer area in virtual memory. In addition to being faster than reading the page by conventional methods, this means the operating system's memory management takes care of references to pages which happen to still be in memory, etc. The algorithm is as much as 50 times faster than the standard INTERLISP-10 FILEPOS function (depending on the length of the pattern).

There are several situations in which it may not be advisable to use our algorithm. If the expected penetration i at which the pattern is found is small, the preprocessing time is significant and one might therefore consider using the obvious intuitive algorithm.

As previously noted, our algorithm can be most efficiently implemented on a byte-addressable machine. On a machine that does not allow byte addresses to be incremented and decremented directly, two possible sources of inefficiency must be addressed: The algorithm typically skips through *string* in steps larger than 1, and the algorithm may back up through *string*. Unless these processes are coded efficiently, it is probably not worthwhile to use our algorithm.

Furthermore, it should be noted that because the algorithm can back up through *string*, it is possible to cross a page boundary more than once. We have not found this to be a serious source of inefficiency. However, it does require a certain amount of code to handle the necessary buffering (if page I/O is being handled directly as in our FFILEPOS). One beauty of the Knuth, Morris, and Pratt algorithm is that it avoids this problem altogether.

A final situation in which it is unadvisable to use our algorithm is if the string matching problem to be solved is actually more complicated than merely finding the first occurrence of a single substring. For example, if the problem is to find the first of several possible substrings or to identify a location in *string* defined by a regular expression, it is much more advantageous to use an algorithm such as that of Aho and Corasick [1].

It may of course be possible to design an algorithm that searches for multiple patterns or instances of regular expressions by using the idea of starting the match at the right end of the pattern. However, we have not designed such an algorithm.

9. Historical Remarks

Our earliest formulation of the algorithm involved only $delta_1$ and implemented Observations 1, 2, and 3(a). We were aware that we could do something along the lines of $delta_2$ and Observation 3(b), but did not precisely formulate it. Instead, in April 1974, we coded the $delta_1$ version of the algorithm in Interlisp, merely to test its speed. We considered coding the algorithm in PDP-10 assembly language but abandoned the idea as impractical because of the cost of incrementing byte pointers by arbitrary amounts.

We have since learned that R.W. Gosper, of Stanford University, simultaneously and independently discovered the $delta_1$ version of the algorithm (private communication).

In April 1975, we started thinking about the implementation again and discovered a way to increment byte pointers by indexing through a table. We then formulated a version of $delta_2$ and coded the algorithm more or less as it is presented here. This original definition of $delta_2$ differed from the current one in the following respect: If only the last m characters of *pat* (call this substring *subpat*) were matched, $delta_2$ specified a slide to the second from the rightmost occurrence of *subpat* in *pat* (allowing this occurrence to "fall off" the left end of *pat*) but without any special consideration of the character preceding this occurrence.

The average behavior of that version of the algorithm was virtually indistinguishable from that presented in this paper for large alphabets, but was somewhat worse for small alphabets. However, its worst case behavior was quadratic (i.e. required on the order of $i * patlen$ comparisons). For example, consider searching for a pattern of the form $CA(BA)^r$ in a string of the form $((XX)^r(AA)(BA)^r)^*$ (e.g. r = 2, *pat* = "CABABA," and *string* = "XXXXAABABAXXXXXAABABA . . ."). The original definition of $delta_2$ allowed only a slide of 2 if the last "BA" of *pat* was matched before the next "A" failed to match. Of course in this situation this only sets up another mismatch at the same character in *string*, but the algorithm had to reinspect the previously inspected characters to discover it. The total number of references to *string* in passing i characters in this situation was $(r + 1) * (r + 2) * i/(4r + 2)$, where $r = (patlen - 2)/2$. Thus the number of references was on the order of $i * patlen$.

However, on the average the algorithm was blindingly fast. To our surprise, it was several times faster than the string searching algorithm in the Tenex TECO text editor. This algorithm is reputed to be quite an efficient implementation of the simple search algorithm because it searches for the first character of *pat* one full word at a time (rather than one byte at a time).

In the summer of 1975, we wrote a brief paper on the algorithm and distributed it on request.

In December 1975, Ben Kuipers of the M.I.T. Artificial Intelligence Laboratory read the paper and brought to our attention the improvement to $delta_2$ concerning the character preceding the terminal substring and its reoccurrence (private communication). Almost simultaneously, Donald Knuth of Stanford University suggested the same improvement and observed that the improved algorithm could certainly make no more than order $(i + patlen) * log(patlen)$ references to *string* (private communication).

We mentioned this improvement in the next revision of the paper and suggested an additional improvement, namely the replacement of both $delta_1$ and $delta_2$ by a single two-dimensional table. Given the mismatched *char* from *string* and the position j in *pat* at which the mismatch occurred, this table indicated the distance to the last occurrence (if any) of the substring $[char, pat(j + 1), \ldots, pat(patlen)]$ in *pat*. The revised paper concluded with the question of whether this improvement or a similar one produced an algorithm

which was at worst linear and on the average "sub-linear."

In January 1976, Knuth [5] proved that the simpler improvement in fact produces linear behavior, even in the worst case. We therefore revised the paper again and gave $delta_2$ its current definition.

In April 1976, R.W. Floyd of Stanford University discovered a serious statistical fallacy in the first version of our formula giving the expected value of the ratio of cost to characters passed. He provided us (private communication) with the current version of this formula.

Thomas Standish, of the University of California at Irvine, has suggested (private communication) that the implementation of the algorithm can be improved by fetching larger bytes in the *fast* loop (i.e. bytes containing several characters) and using a hash array to encode the extended $delta_1$ table. Provided the difficulties at the boundaries of the pattern are handled efficiently, this could improve the behavior of the algorithm enormously since it exponentially increases the effective size of the alphabet and reduces the frequency of common characters.

Acknowledgments. We would like to thank B. Kuipers, of the M.I.T. Artificial Intelligence Laboratory, for his suggestion concerning $delta_2$ and D. Knuth, of Stanford University, for his analysis of the improved algorithm. We are grateful to the anonymous reviewer for *Communications* who suggested the inclusion of evidence comparing our algorithm with that of Knuth, Morris, and Pratt, and for the warnings contained in Section 8. B. Mont-Reynaud, of the Stanford Research Institute, and L. Guibas, of Xerox Palo Alto Research Center, proofread drafts of this paper and suggested several clarifications. We would also like to thank E. Taft and E. Fiala of Xerox Palo Alto Research Center for their advice regarding machine coding the algorithm.

Received June 1975; revised April 1976

References

1. Aho, A.V., and Corasick, M.J. Fast pattern matching: An aid to bibliographic search. *Comm. ACM 18*, 6 (June, 1975), 333-340.
2. Beeler, M., Gosper, R.W., and Schroeppel, R. Hakmem. Memo No. 239, M.I.T. Artificial Intelligence Lab., M.I.T., Cambridge, Mass., Feb. 29, 1972.
3. Dewey, G. *Relativ Frequency of English Speech Sounds*. Harvard U. Press, Cambridge, Mass., 1923, p. 185.
4. Knuth, D.E., Morris, J.H., and Pratt, V.R. Fast pattern matching in strings. TR CS-74-440, Stanford U., Stanford, Calif., 1974.
5. Knuth, D.E., Morris, J.H., and Pratt, V.R. Fast pattern matching in strings. (to appear in *SIAM J. Comput.*).

Algorithms for Pattern Matching

G. DAVIES AND S. BOWSHER
Faculty of Mathematics, The Open University, Walton Hall, Milton Keynes, MK7 6AA, U.K.

SUMMARY

This paper describes four algorithms of varying complexity used for pattern matching, and investigates their behaviour. The algorithms are tested using patterns of varying length from several alphabets. It is concluded that although there is no overall 'best' algorithm, the more complex algorithms are worth considering as they are generally more efficient in terms of number of comparisons made and execution time.

KEY WORDS String searching Pattern matching Text editing Information retrieval

INTRODUCTION

Pattern matching is an integral part of many text editing, data retrieval, symbol manipulation, and word and data processing problems. 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 the position at which the pattern occurs as a contiguous substring of the text.

In general, given a string of text, $S_1, S_2 \ldots S_n$, it is required to find an occurrence, if it exists, of a pattern. $P_1P_2 \ldots P_m$ in the string; such an occurrence is identified by the value i such that the characters $S_iS_{i+1} \ldots S_{i-1+m}$ match the characters in the pattern $P_1P_2 \ldots P_m$. There may be more than one occurrence of the pattern in the string and hence more than one value for i.

This paper investigates four algorithms for finding the value of i for the first (leftmost) occurrence of the patterns.

Algorithm 1 is the 'brute-force' method, the obvious solution to the pattern matching problem, where the algorithm considers each character position of the text string being searched and determines whether the successive 'pattern-length' characters of the 'string' starting at that position match the successive 'pattern-length' characters of the pattern.

Algorithm 2 is based on work done by Knuth, Morris and Pratt.[1] It involves the preprocessing of the pattern to be located to create a table which is then used to tell us where to resume matching after a character mismatch has occurred.

Algorithm 3 was developed by R. S. Boyer and J. Strother Moore.[2] The algorithm uses the fact that more information can be gained by matching the pattern from the right than from the left. It also involves preprocessing of the pattern to produce tables that are used to compute the failure jumps, i.e. how far to skip along in the text after a mismatch has occurred.

Algorithm 4 is an algorithm developed by O. M. Rabin and R. M. Karp.[3] It is another brute-force approach to string searching which uses a large memory to advantage by treating each possible m-character section (where m is the pattern length) of the text as a

Reprinted from ***Software – Practice and Experience***, Volume 16, No. 6, June 1986, pp. 575-601.

key in a standard hash table. The algorithm simply computes the hash function for each of the possible m-character sections of the text and checks if it is equal to the hash function of the pattern. It is not necessary to keep a whole hash table in memory since only one key is being sought. The algorithm really only finds an m-character section in the text which has the same hash value as the pattern, so to be extra sure a direct comparison of the text and pattern is made.

AIMS

The purpose of the investigation was to examine four different algorithms of varying complexity used for pattern matching in strings. Four algorithms are discussed and both their theoretical and actual behaviours are looked at.

The investigation aimed to compare and contrast the four algorithms and to answer such questions as:

1. How do the actual performances of the four algorithms compare under test conditions?
2. How does the predicted theoretical behaviour compare with the actual performance on test data?
3. Do variables such as the size of the alphabet or the size of the pattern being searched for have any effect on the performances of the algorithms. (At the outset it was intuitively expected that the alphabet size would be a major influencing factor on the performances of the different algorithms; the smaller the size of the alphabet the higher the probability of matching any two random characters from that alphabet, and thus it was expected that if the pattern size was of a reasonable length the simple brute-force approach would suffice because the computational effort required to produce failure vectors for algorithms 2 and 3 could not be justified against the simple approach. Similarly the size of the pattern being searched for was expected to play a large part).

THE ALGORITHMS USED

All the algorithms under consideration were coded in the C programming language and designed to run under UCB UNIX Version 2.8. For analysis purposes only (and not for efficiency) the algorithms were implemented in such a way that the program tries to match the pattern only up to the first occurrence of the pattern.

All the algorithms used could easily be extended to find all occurrences of the pattern in the input text, since they scan sequentially through the text and can be restarted at the point directly after the beginning of the match to find the next match. They could also be extended to take as input a file of any size and to report each line in the file in which the pattern occurs.

In all cases the C optimizer was used to produce efficient executable code.

The analysis assumes that the string and pattern are entirely in primary memory and that each character can be obtained in one instruction after computing its byte address. However, if the string is actually on secondary storage then the characters must be read in.

Throughout the discussions of the algorithms the following notation is assumed:

pattern	—	pattern being search for
pattern[j]	—	jth character of the pattern

string	—	text string being searched
string[k]	—	kth character of the string
patlen, m	—	number of characters in pattern

NB. In all descriptions of the algorithms when the pattern is shifted by x places or the text is moved along x places, this actually means that the pointer into the string is incremented by x positions.

Algorithm 1

This is the obvious or brute-force approach to solving the problem of pattern matching and is well documented. The essence of this algorithm is to superimpose the pattern on top of each substring of the text string of patlen proceeding from left to right, and to check the characters of the pattern, character by character against the substring of the text string beneath it. As soon as a mismatch is detected, the pattern is shifted to the right one character. The algorithm tries to search at every starting position of the text, abandoning the search as soon as an incorrect character is found; this occurs if an initial substring of the pattern occurs in the string, and is known as a false start, e.g.

```
string  : string
string  : A string searching example consisting of
            ↑      ↑                      ↑
```

where ↑ signifies a false start.

The pointer into the pattern is incremented four times on execution of the algorithm, once for each s and twice for the first st; these are the false starts. This then involves the backing up of the pointers into the pattern and string, which could add complications, for instance when a large file is being read in from some external device. The backtracking involved when a partially successful search path fails necessitates a lot of storage and bookkeeping and tends to execute slowly.

The actual implementation of the algorithm checks each possible position in the text at which the pattern could match, to see whether it does in fact match. The program keeps one pointer (k) into the text string and another pointer (j) into the pattern. If j and k point to the same characters, then they are both incremented to point to the next character in both the pattern and the string. If j and k point to mismatching characters then j is reset to correspond to moving the pattern to the right by one position for matching against text. If the end of the text string is reached then there is no match.

Informal outline of the algorithm

```
j=0; k=0;
while not at end of pattern or string
    if pattern[j] = string[k]
        then
            k++; j++;
        else
            k=k-j+1; j=0;
    endif
endwhile
```

Obviously this algorithm can be very inefficient; consider trying to match the pattern aaa...ab of length m in a string of n a's:

```
pattern  :  aaa...ab

string   :  aaa.........aaaa
```

On trying to match the two character strings, the first $m-1$ letters of the pattern will match the characters in the string, but the final, mth, letter of the pattern will not; a mismatch will occur. Thus the whole pattern will be moved along by one position and the matching will be resumed.

```
pattern  :   aaa....ab

string   :  aaa.........aaaa
```

Similarly, the first $m-1$ characters of the pattern will match with the string again until mismatch occurs on the mth character of the pattern. Continuing this process we see there will be $n-m+1$ times that the string and the pattern have $m-1$ as in common. Thus the algorithm will require at least $(n-m+1)(m-1)$ comparison operations, i.e. $O(mn)$, and thus the number of direct character comparisons in the worst case is quadratic.

The quadratic nature of this algorithm, however, appears only when initial substrings of the pattern occur in the text string being searched, which appears to be a fairly rare phenomenon in English text, and so in the majority of cases the running time is expected to be less than this. Thus an optimized version of the brute force approach often provides a good 'standard'. This investigation aimed to show how the algorithm actually behaves under (simulated) realistic conditions.

Summarizing, the theoretical behaviour of this algorithm in the worst case is quadratic, $O(mn)$, whereas the average theoretical behaviour is better than this but is dependent on statistical properties of the pattern and the text string being searched.

Algorithm 2

The basic idea behind algorithm 2 is to take advantage of the fact that when a mismatch occurs the false start consists of characters that are already known since they are already in the pattern. This information can then be used to prevent backing up through the text string over all those known characters. 'Shifts' can be precomputed specifying how much to move the given pattern when a mismatch occurs, by preprocessing the actual pattern (by shifting the pattern against itself) to form a table. To illustrate this approach, consider Knuth, Morris and Pratt's example, searching for the pattern abcabcacab in the text string babcbabcabcaabcabcabcacabc. Initially the pattern is placed at the extreme left and scanning starts at the leftmost character of the input text:

```
pattern  :  abcabcacab

string   :  babcbabcabcaabcabcabcacabc
            ↑
```

where ↑ indicates the current text character.

Since the string pointer points to the character b, which does not match the a in the pattern, the pattern is shifted one place to the right and the next string character is inspected:

```
pattern  :   abcabcacab

string   :  babcbabcabcaabcabcabcacabc
             ↑
```

There is a match, so the pattern remains where it is while the next several characters are scanned until we come to a mismatch:

```
pattern  :   abcabcacab

string   :  babcbabcabcaabcabcabcacabc
                ↑

            mismatch
```

The first three pattern characters are matched, but there is a mismatch on the fourth; thus it is known that the last four characters of the input text string have been abc? where ? is not equal to a. There is no need to remember the previously scanned characters since the position in the pattern yields enough information to recreate them. In this example as long as the ? character is not equal to a the pattern can be immediately shifted four more places to the right, since a shift of one, two or three positions could not possibly lead to a match. Thus the situation appears as

```
pattern  :       abcabcacab

string   :  babcbabcabcaabcabcabcacabc
                ↑
```

The characters are scanned again and another partial match occurs; a mismatch occurs on the eighth pattern character:

```
pattern  :       abcabcacab

string   :  babcbabcabcaabcabcabcacabc
                       ↑

                   mismatch
```

Similarly we know that the last eight string characters were abcabca? where ? is not equal to c but might be equal to b. Thus the pattern should be shifted three places to the right:

```
pattern  :          abcabcacab

string   :  babcbabcabcaabcabcabcacabc
                       ↑

                   mismatch
```

Comparing the two characters we get a mismatch, so we shift the pattern a further four places:

```
pattern  :              abcabcacab

string   :  babcbabcabcaabcabcabcacabc
                        ↑
```

This produces another partial match; scanning is continued until mismatch occurs, again, on the eighth pattern character:

```
pattern  :              abcabcacab

string   :  babcbabcabcaabcabcabcacabc
                                ↑

                             mismatch
```

The pattern is then shifted a further three places to the right:

```
pattern  :                 abcabcacab

string   :  babcababcabcaabcabcabcacabc
                           ↑
```

This time a match is produced and, on scanning, the full pattern is found; the pattern match has succeeded.

This worked example illustrates that, by knowing the characters in the pattern and the position in the pattern where a mismatch occurs with a character in the text string, it can be determined where in the pattern to continue the search for a match without moving backwards in the input text. It also shows that the process of pattern matching will run much more efficiently if we have an auxilliary table which will provide us with the information of how far to slide the pattern when a mismatch has been detected at the jth character of the pattern. Thus the vector NEXT can now be introduced; NEXT[j] is the character position in the pattern which should be checked next after a mismatch has occurred at the jth character of the pattern. Hence the pattern is actually being slid j–NEXT[j] places relative to the text string.

When the pattern pointer (j) and the string pointer (k) point to mismatching characters, then we know that the last j characters of the text string up to and including string[k] were pattern[1] . . pattern[j−1] x where x ≠ pattern[j]. NEXT[j] is the largest i<j such that the last i characters of the text string were pattern[1]. .pattern[i−1] x and pattern[i] ≠ pattern[j]. If no such i exists NEXT[j]=0.

The NEXT vector can easily be calculated by using a secondary vector f. The jth entry f[j], j>1, is defined as the largest i less than j such that pattern[1]. .pattern[i−1] = pattern[j−i+1]. .p[j−1]. This holds for i=1, and f[j] is always greater than or equal to 1 when j>1. f[1] is defined to be 0. Thus the last i−1 (already matched) characters of the pattern preceding the mismatched character match the subpattern pattern[1]. pattern[i−1].

The values of the vector f can be determined by sliding a copy of the first j−1 characters of the pattern over itself; sliding from left to right with the first character of the copy over the second character of the pattern, stopping when either all overlapping characters match or there are none overlapping. If a mismatch is detected at the jth

character of the pattern the value of f[j] is exactly the number of overlapping characters plus one.

The values of the vector NEXT can then be calculated using the following relation:

$$NEXT[j] = \begin{cases} f[j] & \text{if pattern[j]} \neq \text{pattern[f[j]]} \\ NEXT[f[j]] & \text{if pattern[j]} = \text{pattern[f[j]]} \end{cases}$$

The NEXT vector, however, can be calculated in such a way that the values of f[j] need not be stored in memory (see below). The above example has the following 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
f[j]	=	0	1	1	1	2	3	4	5	1	2

NEXT[j]=0 means that the pattern is to be slid all the way past the current text character, that is the pointer to the text string should be incremented. By definition NEXT[1]=0.

On mismatch there is no need to back up the string pointer. If NEXT[j]≠0 then we simply leave the string pointer unchanged and set the pattern pointer to NEXT[j]. Thus the vector NEXT provides a way of eliminating the back up of the text string pointer.

The inclusion of the NEXT vector can be seen in the informal outline of algorithm 2 below:

```
j=0; k=0;
while not at end of pattern or string
    if pattern[j] = string [k]
        then
            j++; k++;
        else
            j=NEXT[j];
            if j=0
                k++;
endwhile
```

The initialization of the NEXT vector is similar to the actual algorithm above except that it matches the pattern against itself:

```
x=1; y=0; NEXT[1]=0;
while not at end of pattern
    if pattern [x] ≠ pattern[y]
        then
            y = NEXT[y];
        x++; y++;
    if pattern[x] = pattern[y]
        then
            NEXT[x] = NEXT[y];
        else
            NEXT[x] = y;
endwhile
```

After the NEXT vector has been set up, then at each step of the scanning process either the text string pointer or the pattern pointer is moved. Assuming that the text string is of length n, both the pattern and string pointers can move at most n times, and so at most $2n$ steps are needed to perform the pattern matching operation. It has been shown by Knuth, using the same argument as above, that the NEXT vector can be set up in $O(m)$ steps, where m is the pattern length, since the NEXT vector is found by shifting the pattern against itself. Thus the whole algorithm is of $O(m+n)$, i.e. linear.

Algorithm 2 is very good if it us used to search for highly self-repetitive patterns in self-repetitive text, but intuitively this does not seem to be a very common phenomenon. Thus this investigation seeks to find if this algorithm is (significantly) faster and/or more efficient in character comparisons than the brute-force method in (simulated) realistic conditions.

The major virtue of this algorithm is that it proceeds sequentially through the input text and never needs to back up. This makes the algorithm convenient for use on a large file being read in from some external device, because it avoids rescanning the input text string and possible complicated buffering operations.

Summarizing, the expected theoretical behaviour of algorithm 2 is linear, $O(m+n)$, and can therefore be used as a standard to which the other algorithms can be compared.

Algorithm 3

This algorithm was developed in 1974 by R. S. Boyer and J. S. Moore and the description that follows is essentially theirs. The underlying idea behind algorithm 3 is that a faster searching method can be developed by scanning the pattern from right to left when trying to match it against the text. This is only possible if backing up, that is moving backwards as well as forwards through the character array, is not a problem. The algorithm decides which characters to compare next based on the character that caused the mismatch in the text string as well as the pattern. The next characters to compare are found by appealing to a number of observations (see below).

The essence of this algorithm is to superimpose the pattern $p[1]\ldots p[m]$ on the string $s[1]\ldots s[n]$

$$\begin{array}{lllll} p[1] & p[2] & \ldots & p[m] & \\ s[k-m+1] & s[k-m+2] & \ldots & s[k] & \ldots\ s[n] \end{array}$$

and to match the characters of the pattern against those of the underlying segment of the string in right to left order while the pattern is being moved to the right.

Observation 1

If the rightmost character of the pattern $p[m]$ does not match the underlying character $s[k]$ of the string, and it is also the case that the character corresponding to $s[k]$ does not occur anywhere in the pattern, then the pattern can be slid, in its entirety, over to the right all the way past $s[k]$ since no character of the pattern superimposed above $s[k]$ would ever match. Thus the situation, from above, would be:

$$\begin{array}{llllll} & p[1] & p[2] & \ldots & p[m] & \\ s[k] & s[k+1] & s[k+2] & \ldots & s[k+m] & \ldots\ s[n] \end{array}$$

Matching can then be resumed. Since there was no need to compare any of the $s[k-m+1] \ldots s[k-1]$ with $p[1] \ldots p[m-1]$, then $(m-1)$ character comparisons were omitted.

If however $s[k]$ does match with $p[m]$, then the pattern pointer can be shifted by the minimum amount consistent with a match:

$$\begin{array}{ll} p[1] \quad \ldots p[m-1] & p[m] \\ s[k-m+1] \ldots s[k-1] & s[k] \ldots s[n] \end{array}$$

$$\uparrow$$

Current text character

Observation 2

Generally, if the rightmost occurrence of the character corresponding to the character $s[k]$ in the pattern is r characters from the right end of the pattern, then we can automatically slide the pattern right r characters without checking for matches, since if the pattern were moved right by any amount less than r positions, then the character $s[k]$ would be aligned with some character that it could not possibly match. Such a match would require an occurrence of the character $s[k]$ in the pattern to the right of the last occurrence. The distance r is a function of the characters in the text string being searched. If the character $s[k]$ does not occur in the pattern, then r is patlen positions, i.e. shift the whole pattern along; this is a direct result ascertainable from observation 1. If the character $s[k]$ does occur in the pattern, then r is defined as the difference between the pattern length and the position of the rightmost occurrence of the character $s[k]$ in the pattern, i.e. simply shift the pattern so that the two known occurrences of the character corresponding to $s[k]$ coincide.

Assuming that the last character of the pattern, $p[m]$, matches the underlying character in the string, $s[k]$ say, it must be determined whether the previous character in the string $s[k-1]$ matches the second to last character in the pattern:

$$\begin{array}{lll} & p[1] \quad \ldots p[m-1] & p[m] \\ s[k-m] & s[k-m+1] \ldots \; s[k-1] & s[k] \end{array}$$

$$\uparrow$$

If the characters match then the process of comparing characters working from right to left through the pattern continues until either all of the pattern has been matched, and therefore the algorithm has succeeded in finding a match, or a mismatch is detected at some character $s[k-h]$ after matching the last h characters of the pattern:

$$\begin{array}{ll} p[1] \quad \ldots p[m-h] \ldots p[m] & \\ s[k-m+1] \quad \ldots s[k-h] \quad \ldots s[k] \ldots s[n] & \end{array}$$

$$\uparrow$$

mismatch

In the second case, it is necessary to shift the pattern down, ideally by as much as possible, and to resume matching at the next plausible position for matching.

Observation 3

If the character $s[k-h]$, which caused the mismatch, occurs in the pattern, the pattern can now be slid down by, say, g positions so as to align the two known occurrences of the character $s[k-h]$. Then the character in the string which is aligned with the last character in the pattern will be compared:

$$\begin{array}{lllll} & p[m-h-g] & \ldots\ p[m-g] & \ldots\ p[m] & \\ s[1]\ \ldots\ . & s[k-h] & \ldots\ s[k] & \ldots\ s[k+g] & \ldots\ .\ s[n] \\ & & & \uparrow & \\ & & & \text{resume matching} & \end{array}$$

(s[k−h] and $p[m-h-g]$ are identical).

Thus our attention is shifted down the string by $g + h$ positions. The distance g by which the pattern can be slid depends on where the character $s[k-h]$ occurs in the pattern. If the rightmost occurrence of the character corresponding to $s[k-h]$ in the pattern is to the right of the mismatched character (e.g. it occurs in that part of the pattern which has already been scanned) then by definition the pattern would have to be moved backwards to align the two known occurrences of the character corresponding to $s[k-h]$. This is obviously to be avoided and thus the pattern is slid down by $g=1$ positions (this is always sound). Thus our attention to the text string is shifted down by $1 + h$ positions. Applying this to the above example ($g=1$):

$$\begin{array}{llll} & \ldots\ p[m-1] & p[m] & \\ s[k-h]\ \ldots\ s[k] & & s[k+1] & \ldots\ s[n] \\ & & \uparrow & \end{array}$$

On the other hand, if the rightmost occurrence of the character equivalent to $s[k-h]$ in the pattern is to the left of the mismatch, then we can slide forward by $g = r - h$ positions, where r comes from observations 1 and 2, to align the two known occurrences of the character causing the mismatch.

Thus observation 3 is concerned with shifting the pattern after mismatch has occurred at a character char, say, to align the two known occurrences of char in the pattern and string and then to resume matching from the rightmost end of the pattern with the corresponding character in the string.

This can be improved upon by considering not only the known (multiple) occurrences of a single character but also known occurrences of substrings of the text string in the pattern:

Observation 4

Having matched the last h characters of the pattern and reached a mismatch:

$$\begin{array}{llll} p[1] & \ldots\ p[m-h] & \ldots\ p[m] & \\ \ldots\ s[k-m+1] & \ldots\ s[k-h] & \ldots\ s[k] & \ldots\ s[n] \\ & \uparrow & & \\ & \text{mismatch} & & \end{array}$$

Then, by definition it is known that the final h characters of the pattern match with the

next h characters of the text string (in left to right reading order). The last h characters of the pattern can be treated as a subpattern,

$$\text{subpatt} = p[m-h+1] \ldots p[m] \text{ , length } h$$

This occurrence of subpatt in the text string is preceded by a character, here $s[k-h]$, which is different from that character preceding the terminal occurrence of subpatt in the pattern, here $p[m-h]$. Then, by using roughly similar reasoning to that used in conjunction with observation 3, the next-to-last (rightmost) occurrence of subpatt can be located, if it exists, in the pattern. The pattern can then be slid down by some amount, g, say, so as to align this rightmost occurrence of the subpattern in the pattern which is not preceded by the character char, which caused the mismatch, with its terminal occurrence in the pattern $s[k-h] \ldots s[k]$. Thus:

$$\begin{array}{llll} p[1] & \ldots p[m-h-g] \ldots p[m-g] & \ldots p[m] & \\ s[k+g-m+1] & \ldots s[k-h] \quad \ldots s[k] & \ldots s[k+g] & \ldots s[n] \\ & & \uparrow & \\ & & \text{resume matching} & \end{array}$$

This reoccurrence of subpatt in the pattern is known as a plausible reoccurrence. Thus now the character pointer to the string can be incremented by an amount $g + h$, so it will align the last character $p[m]$ of the pattern in the new patterns position, ready for the next character comparison.

More precisely if the last h characters of the pattern have been matched before finding a mismatch, then the pattern can be moved down by g characters, where g is based on the position in the pattern of the rightmost plausible occurrence of the terminal substring of the pattern having h characters. After sliding down by g positions, the characters of the text string aligned with the last character of the pattern are compared. Thus in actuality we are moving down the string by $h + g$ characters, which allows for the h matched characters before the mismatch occurred. This distance is called R, and is defined as a function of the position j in the pattern at which mismatch occurred. g is simply the distance between the terminal occurrence of the subpattern and its rightmost plausible reoccurrence (if it exists) and is always greater than or equal to 1. h is simply the length of the pattern minus the index position at which mismatch occurred, i.e. that number of characters which have been matched before a mismatch occurs.

Quite simply by observation 4 the pattern is shifted to the right so that the known occurrences of a subpattern occurring both in the string and (at least twice) in the pattern can be aligned. Scanning is then resumed from the far right of the pattern.

Thus if the final h characters of the pattern have been matched before failing at $p[m-h]$ then we wish to slide the pattern to the right by an appropriate amount, and to increment the string pointer by $1+h$ or $r(\text{string}[k-h])$ or $R(j)$, whichever allows the largest shift (and therefore decreases the number of comparisons required). Matching can then be resumed with the last character of the pattern and the corresponding string character. By definition of R $(= h + g)$ it is obvious that $R(j)$ is always greater than or equal to $1 + h$, $g \geq 1$. Therefore the string pointer can simply be incremented by the maximum of the two shifts r and R.

Worked example

Consider the following example; it is used to illustrate how the preceding observations can be used to decrease (in the general case) the number of direct character comparisons:

```
pattern : AT-THAT

string  : WHICH-FINALLY-HALTS.--AT-THAT-POINT
                 ↑

              mismatch
```

When the action of pattern matching is started the seventh character of the string, F, is compared with the last character of the pattern and fails. Since F is known not to appear anywhere in the pattern, by appealing to observation 1 the string pointer can be automatically incremented by 7:

```
pattern :         AT-THAT

string  :  WHICH-FINALLY-HALTS.--AT-THAT-POINT
                        ↑

                   mismatch
```

The next comparison is of the hyphen in the string with the T in the pattern. They mismatch and by appealing to observation 2 the pattern can be moved down 4 positions to align the two hyphens. Scanning is then resumed from the right end of the pattern, comparing it directly with the corresponding string character directly below the pattern:

```
pattern :             AT-THAT

string  :  WHICH-FINALLY-HALTS.--AT-THAT-POINT
                              ↑
```

The characters match and so the pointer is moved backwards through both the string and pattern character arrays:

```
pattern :            AT-THAT

string  : WHICH-FINALLY-HALTS.--AT-THAT-POINT
                           ↑

                     mismatch
```

A mismatch is detected and by observation 3 the string pointer can be moved to the right by 7 positions, since the character causing the mismatch, L, does not occur anywhere in the pattern:

```
pattern :                  AT-THAT

string  : WHICH-FINALLY-HALTS.--AT-THAT-POINT
                                 ↑
```

Again the corresponding characters of the string and pattern match, so the string pointer is moved backwards. The previous character in the text string matches with its corresponding character in the pattern and so the pointer is moved back again:

```
pattern :                    AT-THAT

string  : WHICH-FINALLY-HALTS.--AT-THAT-POINT
                                 ↑

                               mismatch
```

A mismatch is detected again and by now appealing to observation 4 the text pointer can be moved to the right by 7 places, so as to align the discovered substring AT with the beginning of the pattern:

```
pattern :                          AT-THAT

string  : WHICH-FINALLY-HALTS.--AT-THAT-POINT
                                       ↑
```

Scanning is now resumed, the characters are successfully compared and both the string and pattern pointers are decremented and the whole pattern is worked through character by character. Here it can be seen that each character in the pattern matches the corresponding character in the string and so the pattern has been found.

It is important to note that only 14 references to the text string were made, and of these 7 were required to confirm the actual match. The other 7 comparisons allowed the movement past the first 22 characters of the string, e.g. those preceding the first occurrence of the pattern.

A brief outline of the algorithm can now be sketched:

```
j = k = patlen;
repeat
    while not end of pattern or string
        if pattern[j] = string[k]
            j--; k--;
        else
            k = k + max(r(string[k]),R(j));
            j = patlen;
        endif
    endwhile
until string found or EOF
```

The implementation of algorithm 3 depends on the existence of two precomputed vectors which determine the values of r and R, the failure jumps. A more rigorous definition of the two string pointer incrementing functions can now be made.

The vector for the r values has an entry for each character in the alphabet being used, its entry is denoted for r(char) where char is some valid character. The precise definition is

```
r(char) = {s (s=patlen) or
                ( 0<=s<patlen and pattern[patlen-s] = char) }
```

If char does not occur in the pattern than r is the pattern length, otherwise it is the pattern length minus j, where j is the maximum integer such that pattern[j] = char.

This vector is derivable from observations 1 and 2. Assuming that g is the alphabet size the preprocessing for r requires an array of size q. The implementation sets up the vector in a linear scan (right to left) through the pattern. Thus the preprocessing for r is linear in patlen plus the size of the alphabet, $O(q+m)$.

The second vector, for the R values, has as many entries as there are character positions in the pattern.

To define R(j) precisely it is necessary to define the rightmost plausible reoccurrence of a terminal substring of the pattern. The following conventions are used : $ is a character that does not occur in the pattern and if i<0 then pattern[i] is $. Two sequences of characters c[0..n] and d[o..n] 'unify' if for all i from 0 to n either c[i]=d[i] or c[i]=$ or d[i]=$. The position of the rightmost plausible reoccurrence of the terminal substring, rpr(j), which starts at position j+1 is defined to be the greatest k less than or equal to patlen such that (pattern[j+1] .. pattern[patlen]) and (pattern[k]..pattern[k+patlen−j−1]) unify and either k<=0 or pattern[k−1]≠pattern[j], i.e. the position of rpr(j) is the rightmost place where subpatt occurs in the pattern and is not preceded by the character, pattern[j], which precedes its terminal occurrence. This position may be beyond the left end of the pattern, and consequently rpr(j) may be negative.

Thus the distance the pattern must be slid down to align the discovered substring (starting at j+1) with its rightmost plausible reoccurrence is j+1−rpr(j).

The pointer into the string must then be aligned with the last character in the pattern, i.e. moved patlen−j. Thus R(j), which is the sum of these two values, is defined as

$$R(j) = patlen + 1 - rpr(j)$$

In Reference 1 Knuth shows that there is an algorithm for setting up the vector for the R values in time linear in patlen, that is $O(m)$.

Example

pattern	:	b	a	d	b	a	c	b	a	c	b	a
r	:	1	0	8	1	0	2	1	0	2	1	0
R	:	19	18	17	16	15	8	13	12	8	12	1

patlen = 11

In Reference 1 Knuth has also proved that the execution of the Boyer–Moore algorithm in the worst case is linear: $O(n+m)$, assuming the availability of array space linear in patlen plus the size of the alphabet. More specifically he shows that in order to discover that the pattern does not occur in the first p characters of the string a maximum of $6p$ characters from the string are matched with characters in the pattern.

In the average case, however, the algorithm is expected to behave sublinearly, that is the expected value of the number of inspected characters in the string is c*(i+patlen), where i is the index position of the leftmost character in the first occurrence of the pattern in the string, and c<1. By using the two precomputed vectors, which depend on the statistical properties of the characters in the pattern and the string, on average, not all the string characters preceding the first occurrence of the pattern in the string need to be inspected; the earlier worked example illustrates this.

Summarizing, the expected theoretical behaviour of algorithm 3 is sublinear, $<O(n+m)$, whereas the theoretical worst case behaviour is linear, $O(n+m)$. A further modification to this algorithm is described in the paper by Galil.[4]

Algorithm 4

This is an algorithm developed by R. M. Karp and O. M. Rabin. It is a brute-force approach to pattern matching which uses a large memory to advantage by treating each possible m-character section of the text string as a key in a standard hash table. In the actual implementation m is the length of the pattern being searched for. It does not use up as much memory as at first thought because it is not necessary to keep a whole hash table in memory, the problem being set up in such a way that only one key is being sought, that of the pattern. Thus all that is required is that the hash function of each possible m-character section of the text is in turn calculated and this is compared with the hash function of the pattern. This algorithm therefore really only finds the first m-character section in the text which has the same value as that of the pattern, so for preciseness, a direct comparison of that section of text and the pattern has then to be made.

On first thought it seems just as hard to compute the hash function for each m-character section from the text as it does to check to see if each character in the pattern is the same as in the text (algorithm 1). Rabin and Karp solved this problem by taking advantage of a fundamental mathematical property of the mod operation when using the hash function:

$$h(k) = k \bmod q$$

where k is the key to be hashed and q is the table size.

They based their method on computing the hash function for position i in the text string, given its hash value for position $i-1$. To facilitate the application of the hash function on an m-character section of text, each section was transformed into an integer key upon which the hashing function could then be performed. This was achieved by writing the characters as numerals in a base d number system, where d is the number of possible characters. Thus the numeral x corresponding to the m-character section string[i] ... string[$i+m-1$] is

$$x = \text{string}[i] \times d^{m-1} + \text{string}[i+1] \times d^{m-2} + \ldots + \text{string}[i+m-1]$$

This is derived by using the same method as with any numeric system; consider the representation of the number 345 in base 10. This is equivalent to

$$x = 3 \times 10^2 + 4 \times 10^1 + 5$$

Shifting by one position right in the text string, i.e. taking the section string[$i+1$] ... string[$i+m$], the value of x becomes

$$x = \text{string}[i+1] \times \text{d}^{m-1} + \text{string}[i+2] \times d^{m-2} + \ldots + \text{string}[i+m]$$

which is equivalent to replacing x by

$$(x - \text{string}[i] \times d^{m-1}) \times d + \text{string}[i+m]$$

Proof

$$((\text{string}[i] \times d^{m-1} + \ldots + \text{string}[i+m-1]) - \text{string}[i] \times d^{m-1}) \times d + \text{string}[i+m]$$

$$= (\text{string}[i+1] \times d^{m-2} + \ldots + \text{string}[i+m-1]) \times d + \text{string}[i+m]$$

$$= \text{string}[i+1] \times d^{m-1} + \ldots + \text{string}[i+m]$$

A fundamental property of the mod operation is that it can be performed at any time during the above operation and the result will still be the same; if we take the remainder when divided by q after each arithmetic operation then we arrive at the same answer that we would get if all the operations were to be performed and then the remainder when divided by q taken. Thus, assuming that the original value of $h(x) = x \bmod q$ is already known, then by shifting one position right in the text the new hashed value can easily be computed using the above property.

Algorithm 4 can now be outlined:

```
dm=1; h1=0; h2=0;
for i=0 .. patlen-2
    dm = (d*dm) mod q
for i=0 .. patlen-1
    h1 = (h1*d + rk(pattern[i])) mod q
for i=0 .. patlen-1
    h2 = (h2*d + rk(string[i])) mod q
i=0;
while not at end of string and h1!=h2
    h2 = (h2+d*q - rk(string[i])*dm) mod q
    h2 = (h2*d + rk(string[i])) mod q
    i++;
endwhile
```

rk(char) is a function which returns a unique integer for each character in the specified alphabet.

The algorithm first computes a hash value, h1, for the pattern and a hash value, h2, for the first m-character section of the text string and also the computed value $d^{m-1} \bmod q$ in the variable dm. Then the hashed value of each m-character section starting a position i for all possible values of i is computed, taking advantage of the associativity of the mod operation. Each new hash value of h2 is in turn compared with h1. If h1 and h2 have the same values, then the characters are directly compared, as a final check.

The table size q is chosen to be a very large prime, so that synonyms, that is keys which hash to the same value, can be avoided, but it is small enough so that overflow does not occur.

Intuitively the major drawback of this algorithm is that the computation of the hashed

values of the m-character text string sections is quite expensive in terms of machine cycles—each value requires three multiplications, and so this investigation aims to show how its actual performance compares with the other three algorithms. The major advantage of this method is that it can easily be extended to image processing. It facilitates 'pattern matching' in two-dimensional patterns and text, which the other algorithms do not, and thus can be used in the field of computer graphics and pattern recognition.

In the very worst case, where each m-character section produces the same hash value, the algorithm would take $O(nm)$ steps, but because such a large value of q is used with the mod operations in the hashing functions the likelihood of a collision occurring is expected to be very small. Thus in the majority, if not all, of cases where there are no synonyms for the hashed value of the pattern except an identical m-character section, the number of hashing operations performed is obviously linear, and thus algorithm 4 is $O(m+n)$.

TESTING

The four algorithms were tested on various types of test data, with patterns of length 2 to 14. Each separate experiment consisted of finding the first occurrence of the pattern in the text string. Because it was intuitively thought that the performances of the algorithms depended upon the statistical properties of the pattern and the source string from which the test patterns were obtained, experiments were performed on five different source strings, each of length 32,000.

1. Binary strings

The first source string consisted of a random sequence of 0s and 1s. A program was written to randomly generate sequences of binary digits. As the program is called with an integer argument specifying the number of characters to be produced, this program was used to generate both the source string and the patterns.

2. Technical English

The second source string consisted of a piece of technical English from an on-line manual. The patterns were chosen at random from the manual, not all of them occurring in the test file.

3, 4. Random text

The third and fourth source strings consisted of random text from a given character set. A program was written to produce output text in which each character from a given character set has an equally likely chance of occurring. The program takes two arguments, the first specifies the number of characters to be produced, the second specifies the number of distinct characters allowable, the alphabet size. This program was also used to produce both the patterns and the source string.

For the experiments the third source string consisted of characters from a 15 character alphabet, and the fourth source string from a 35 character alphabet.

The range 0–32,767 (the maximum number of 16 bit integer quantities) was divided

into a number of bands equal to the alphabet size; the text was then produced by supplying a unique seed to the system random number generator and the random numbers produced were divided by the number of bands. This was added to a base and, using ASCII codes, a random character was produced.

5. Pseudo-English text

The fifth source string consisted of characters that occur in the same frequency as they do in the English language. A program was written that took as input any specified text file, read it and computed the relative frequencies of each letter occuring in that file. These frequencies were then converted into probabilities, and characters were produced in a similar fashion to the above method. The relative frequencies of punctuation marks as well as letters are taken into account with this method. The limitation of this method to produce text is that the resulting text strings are only as 'English-like' as the input file supplied to the program. This program was used to produce both the source string and the patterns.

Measures of comparison

Each experiment consisted of finding the first occurrence (or non-occurrence) of the pattern in the source string. The pattern lengths were varied from 2 to 14, and five different source files were used. From each experiment the following variables were noted:

(a) the pattern length
(b) the number of comparisons made
(c) the alphabet size from which the strings arose
(d) the index position of the pattern in the string
(e) the user time for the execution of each experiment.

Once the above measures had been collected, the problem of how to compare the performances of the algorithms arose. An independent measure was needed so that the performances could be compared fairly.

The first measure of comparison used was one of those suggested by Boyer and Moore,[2] the number of references made to the text string. To ensure that the measure was independent of the particular implementation of the algorithm concerned, the number of references to the string was divided by the number of characters occurring before the pattern (the index position of the pattern minus one) thus obtaining the number of inspected characters in the text string per character passed. These measures were then averaged for each pattern length over all the samples, for each source string in turn. These results are represented graphically (Figures 1–5).

The second measure of comparison was the actual time spent on execution of the algorithm. A system utility was used to measure the real time, the time spent in the system (e.g. executing system calls such as open or close a file etc.) and the time spent in execution of the command (user time). The CPU times are accurate to 1/60th of a second. As the programs were identical except for the actual code used to pattern match and the same set of patterns and source files were used with all the different algorithms it was deemed reasonable to allow a comparison on the user time. As with the first measure *each* separate experiment was timed and then averaged for each pattern length over each source string. These results are also represented graphically (Figure 6–10).

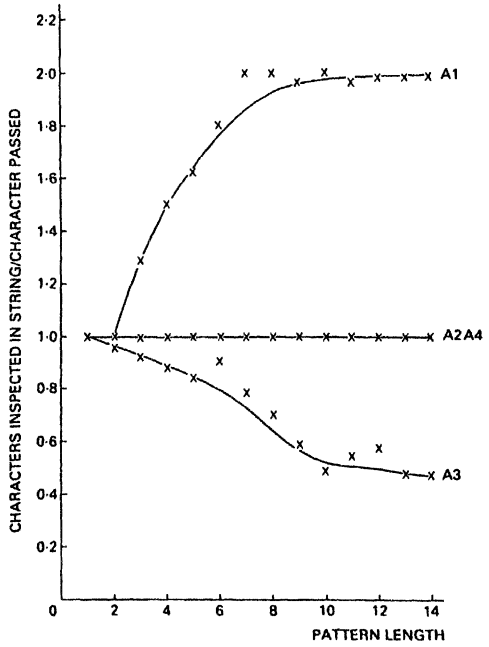


Figure 1. Binary text

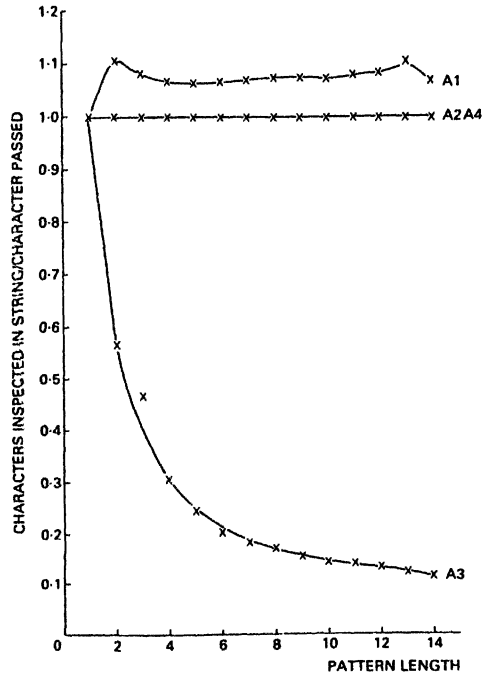


Figure 2. Random (15 character)

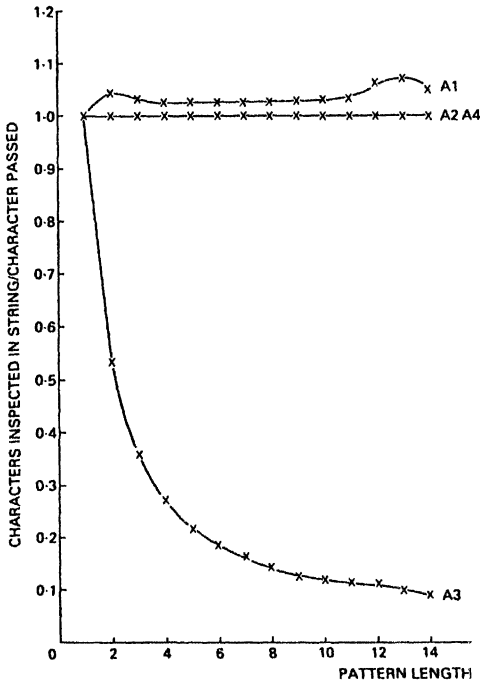


Figure 3. Random (35 character)

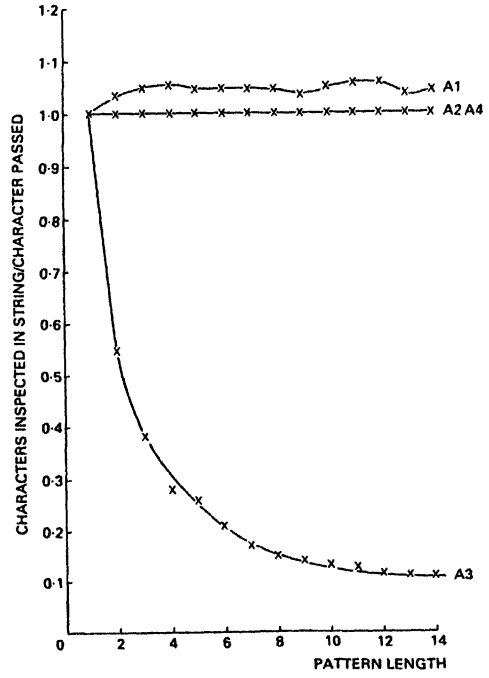


Figure 4. Technical English

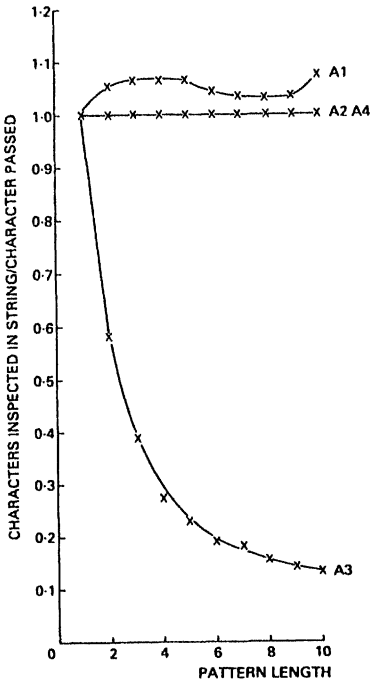


Figure 5. English-like text

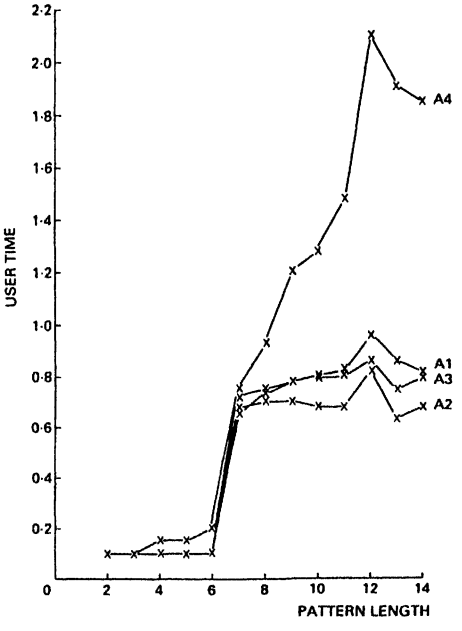


Figure 6. Binary text

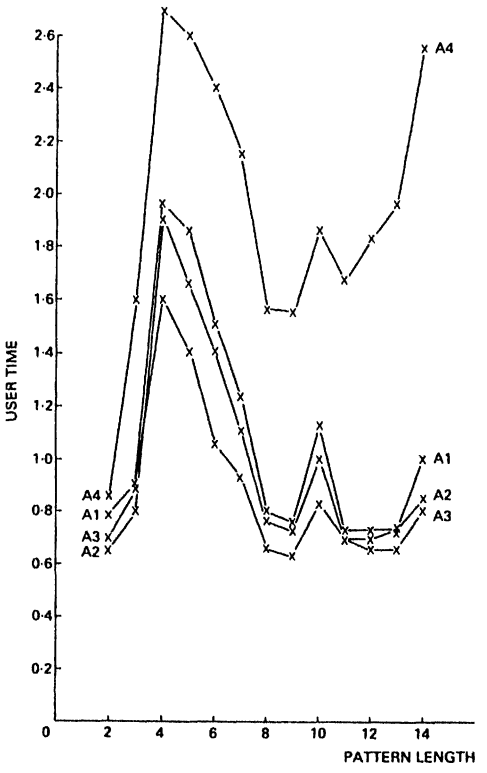


Figure 7. Random (15 character)

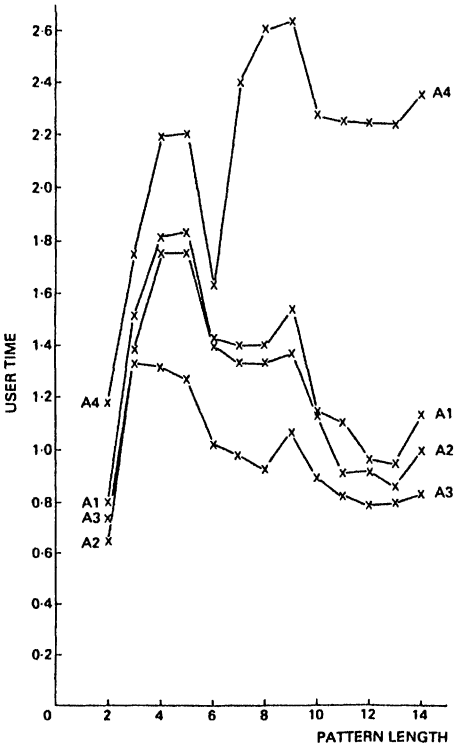


Figure 8. Random (35 character)

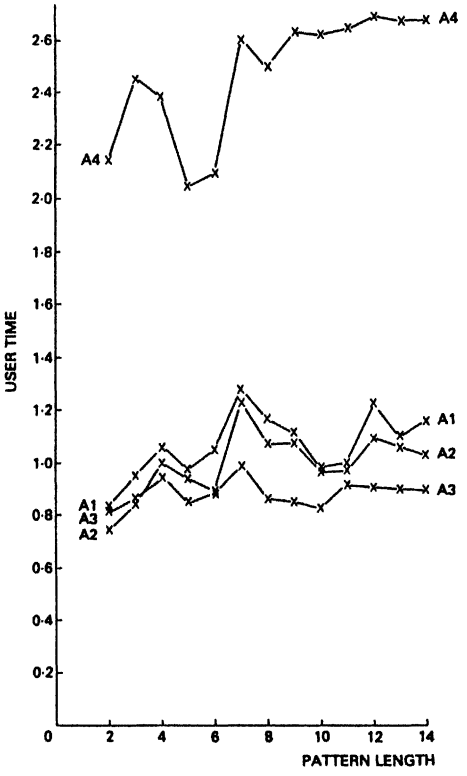


Figure 9. Technical English

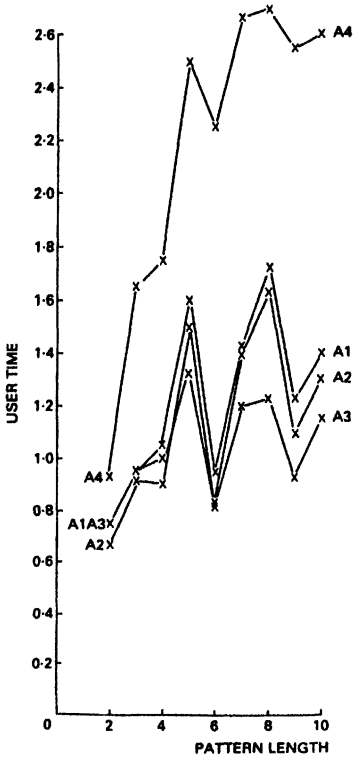


Figure 10. English-like text

Results

The two measures as described were then used to compare the relative performances of the algorithms.

To illustrate the comparative performance for each source string two graphs were drawn. The first plotted the average number of inspected characters in the text string per character passed against the pattern length. The second plotted the average user time for each algorithm against the pattern length.

Additional graphs plotting the number of references per character passed against each pattern length for each source string using algorithms 1 and 3 are also given (Figures 11 and 12).

DISCUSSION OF RESULTS

Referring to the graphs plotting the average number of references to the text string per character passed against the pattern length (Figures 1–5), it can be seen that algorithms 2 and 4 reference the string precisely one time per character passed. This corresponds exactly to their predicted behaviour, and as such they both act as standards to which algorithms 1 and 3 can be compared.

Algorithm 1 references the text string approximately 1·1 times per character passed if the source string is derived from either random text (both 15 and 35 character alphabets) or English (pseudo or technical). However it can be seen that this measure tends to increase if the source string and patterns are obtained from a binary alphabet. For patterns of length greater than six, from the empirical evidence, the average number of references to the string per character passed is approximately 2, double the number made by algorithms 2 and 4. This shows the inefficiency of algorithm 1. It occurs because the probability of an initial substring of the pattern occurring in the text is much greater with a two character alphabet than when using a larger alphabet, and so false starts occur.

The number of references per character passed when using algorithm 3 can be seen to be less than 1. For example, for a pattern of length 5 from technical English text, the algorithm typically inspects approximately 0·26 characters for every character passed. That is, for every reference to the text string the algorithm skips over about 4 characters, or equivalently algorithm 3 inspects only about a quarter of the characters it passes when searching for a pattern of length 5.

If a binary alphabet is used then it can be seen from the empirical evidence that algorithm 3 needs to inspect about 0·84 characters for every character passed, i.e. it needs to inspect three to four times as many characters to find a match as with English text.

It should also be noted that the number of references per character drops as the pattern length increases. The results support the theory that algorithm 3 is sublinear in the number of references to the string.

The reason why the number of references per character passed decreases more slowly as the pattern length increases is that for longer patterns the probability is higher that the character just fetched occurs somewhere in the pattern, and therefore the distance the pattern can be moved forward (if a mismatch occurs) is shortened.

It is also interesting to note that when using algorithm 3 the average number of references per character string passed when a binary alphabet is used is significantly

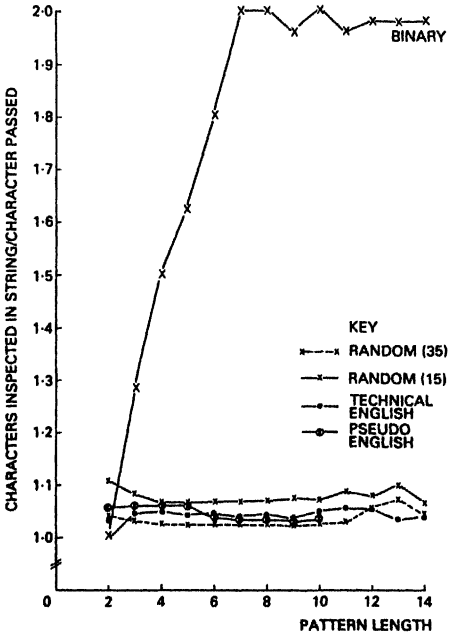


Figure 11. Algorithm 1

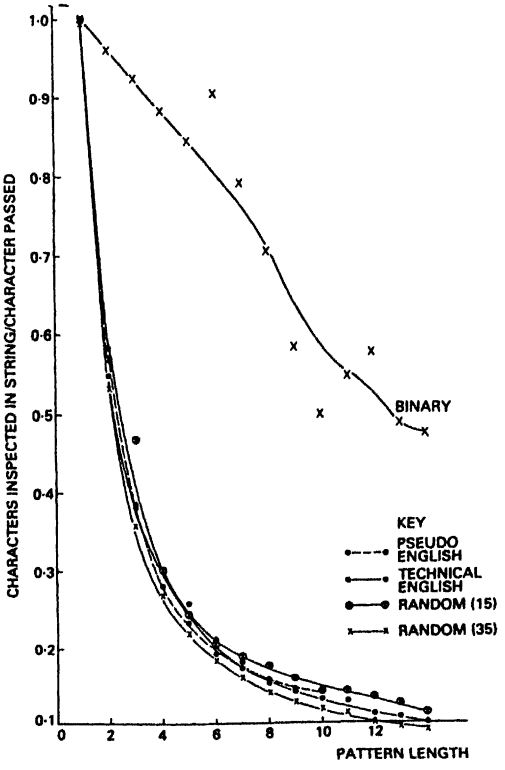


Figure 12. Algorithm 3

higher by a factor of about 4 or 5 than with any other source string for patterns greater than length 5.

In all cases it can be seen that algorithm 3 is much more efficient in terms of direct character comparisons than any of the other three algorithms.

Referring to the graphs plotting the average user time (Figures 6–10), i.e. time spent in execution of the actual code used to implement the algorithm, against the pattern length, it is immediately noticeable that in all cases algorithm 4 takes substantially more time to execute than any of the other three algorithms. This observation agrees with the intuitively expected behaviour—that the computation of the hash values is computationally expensive in terms of machine cycles and so increases the running time of the algorithm. It is clear from looking at the running times that algorithm 4 is not a feasible option if offered a choice of all the algorithms.

It can also be seen that algorithm 1 takes more time than either algorithm 2 or 3, on the whole, for any type of source string.

The comparative increase in the running time of algorithm 3 relative to algorithm 2 on binary strings is explained by the fact that since R (the failure vector concerned with subpatterns) predominates in the binary alphabet and sets up alignments of the pattern and the string, the algorithm backs up over longer terminal substrings of the pattern before finding mismatches. This is because there is a higher probability of subpatterns occurring in the text string.

Thus it can be seen that in the majority of cases, except when using either binary strings or small patterns, algorithm 3 has a faster running time than any of the other three algorithms.

Nothing can be deduced from the absolute shapes of the lines on the user time graph. Information can only be derived from the relative positions of the curves for each algorithm at each pattern length. This is because the patterns were chosen at random and obviously the user time is related to how far into the text the pattern occurs. The times for all the four algorithms can be compared at each pattern length because the same source string and set of patterns were used with each algorithm. The times, which are accurate to 1/60th of a second, also include any time spent preprocessing the pattern, as with algorithms 2 and 3.

CONCLUSIONS

The results show that both the pattern length and the alphabet size from which the strings are taken play an important part when considering which algorithm to use.

From the empirical evidence it can be concluded that algorithm 2 should be used with a binary alphabet or with small patterns drawn from any other alphabet. Algorithm 3 should be used in all other cases. It may not be advisable, however, to use algorithm 3 if the expected penetration at which the pattern will be found is small, since the preprocessing time becomes significant; similarly with algorithm 2, and so algorithm 1 would be better in this situation.

It would also be unadvisable to use algorithm 3 if the string matching problem to be solved is more complicated than finding the first occurrence of a single p ttern. For instance if the problem is to find the first of several possible patterns or to identify a location in the text string defined by a regular expression. This is also because the preprocessing time would be significant.

In no case was algorithm 1 more efficient in terms of either character comparisons

made or running time compared to algorithms 2 and 3. This shows that the computational effort required by algorithms 2 and 3 to preprocess the pattern is justified by an increase in performance.

Although algorithm 4 is linear in the number of references to the text string per character passed the substantially higher running time of this algorithm does not make it a feasible option when considering pattern matching in strings. The advantage of this algorithm over the other three lies in its extension to two-dimensional pattern matching. It can be used for pattern recognition and image processing and thus in the expanding field of computer graphics.

REFERENCES

1. D. E. Knuth, J. H. Morris and V. R. Pratt, 'Fast pattern matching in strings', *SIAM Journal of Computing,* **6**, (2), 323–350 (1977).
2. R. S. Boyer and J. S. Moore, 'A fast string searching algorithm', *Comm. ACM,* **20**, (10), 762–772 (1977).
3. R. Sedgewick, *Algorithms*, Addison-Wesley, pp. 252–253 (1983).
4. Z. Galil, 'On improving the worst-case running time of the Boyer–Moore string matching algorithm', *Comm. ACM,* **22**, (9), 505–508 (1979).