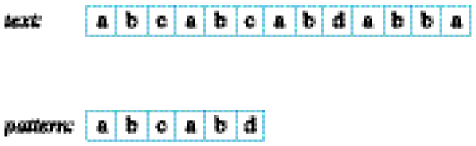
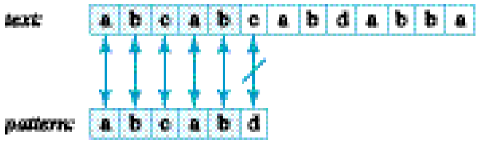
Pattern Matching

In Sec. 5.4 we considered the pattern-matching problem that occurs in many applications, such as text editing and text processing, a good deal of work has gone into designing more efficient algorithms. Here we describe in detail one such algorithm, called the *Knuth-Morris-Pratt algorithm*. Another efficient approach is the Boyer-Moore method described in several algorithms texts.

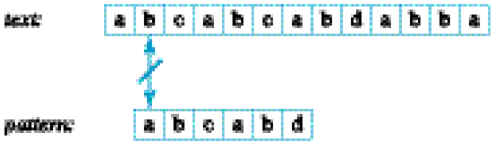
We review first the brute force method described in Sec. 5.4 and how it can be improved. We consider again the example of finding the first occurrence of the pattern “abcabd” in the line of text“abcabcabdabba”:



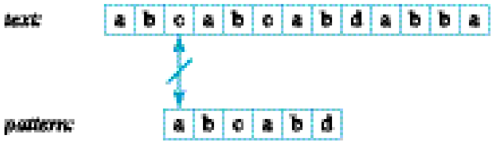
We begin matching characters in *pattern* with those in *text* and find that the first five characters match, but the next characters do not:



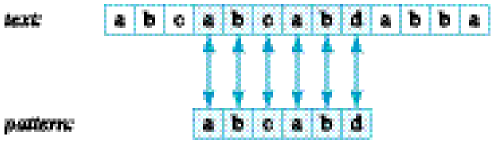
When such a mismatch occurs, we must backtrack to the beginning of *pattern*, shift one position to the right in *text*, and start the search over again:



This time a mismatch occurs immediately, and so we backtrack once more to the beginning of *pattern*, shift another position to the right in *text*, and try again:

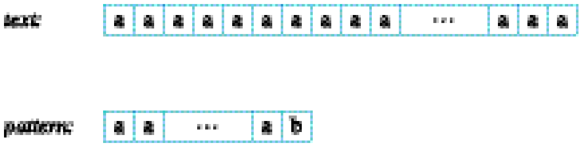


Another mismatch of the first characters occurs, so we backtrack and shift once again:

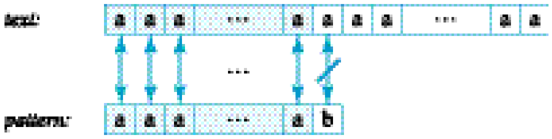


On the next search, all of the characters in *pattern* match the corresponding characters in *text*, and thus the pattern has been located in the line of text.

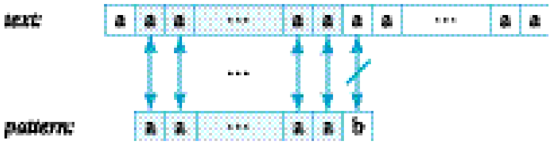
In this example, only three backtracks were required, and two of these required backing up only one character. The situation may be much worse, however. To illustrate, suppose that the text to be searched consists of one hundred characters, each of which is the same, and the pattern consists of 49 of these same characters followed by a different character, for example,



In beginning our search for *pattern* in *text*, we find that the first 49 characters match but that the last character in *pattern* does not match the corresponding character in *text*:

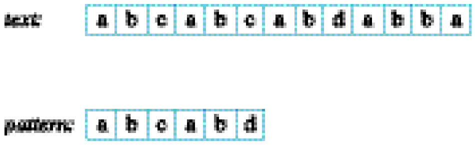


We must therefore backtrack to the beginning of *pattern*, shift one position to the right in *text*, and repeat the search. As before, we make 49 successful comparisons of characters in *pattern* with characters in *text* before a mismatch occurs:

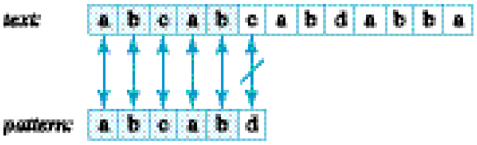


This same thing happens again and again until eventually we reach the end of *text* and are able to determine that *pattern* is not found in *text*. After each unsuccessful scan, we must backtrack from the last character of *pattern* way back to the first character and restart the search one position to the right in *text*.

The source of inefficiency in this algorithm is the backtracking required whenever a mismatch occurs. To illustrate how it can be avoided, consider again the first example in which *pattern* = “abcabd” and *text* = “abcabcabdabba”:



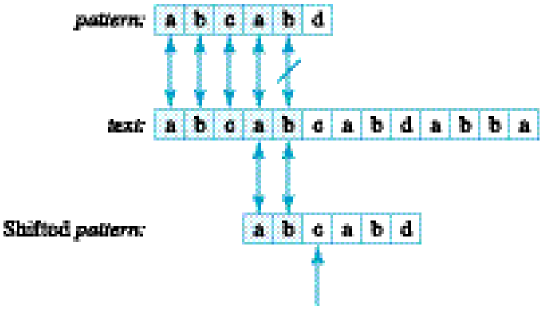
In the first scan, we find that *pattern*[0] = *text*[0], *pattern*[1] = *text*[1], *pattern*[2] = *text*[2], *pattern*[3] = *text*[3], and *pattern*[4] = *text*[4], *but pattern*[5] ≠ *text*[5]:



Examining *pattern*, we see that *pattern*[0] ≠ *pattern*[1]; thus *pattern*[0] cannot possibly match *text*[1] because *pattern*[1] did. Similarly, since *pattern*[0] is different from *pattern*[2], which matched *text*[2], neither can *pattern*[0] match *text*[2]. Consequently, we can immediately “slide” *pattern* three positions to the right, eliminating the backtracks to positions 1 and 2 in *text*. Moreover, examining the part of *pattern* that has matched a substring of *text*,

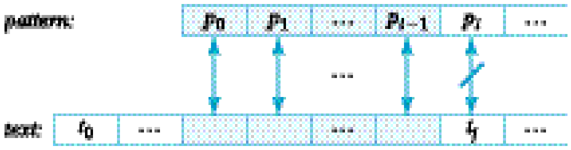
abcab

we see that we need not check *pattern*[0] and *pattern*[1] again since they are the same as *pattern*[3] and *pattern*[4], respectively, which have already matched characters in *text*.

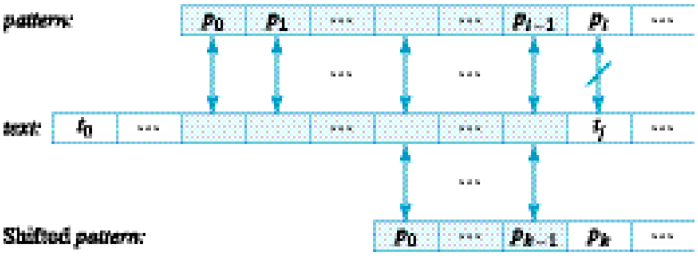


This partial match means that we can continue our search at position 5 in *text* and position 2 in *pattern*; no backtracking to examine characters before that in the position where a mismatch occurred is necessary at all!

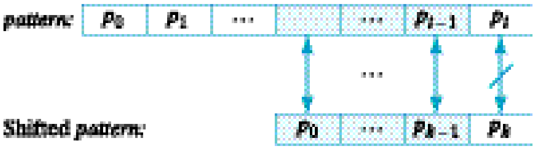
Now consider the general problem of finding the index of a pattern $p_0\, p_1 \cdots p_m$ in a text $t_0 t_1 \cdots t_n$ and suppose that these strings are stored in the arrays *pattern* and *text* so that *pattern*[*i*] = p_i and *text*[*j*] = t_j. Also suppose that in attempting to locate *pattern* in *text* we have come to a point where the first *i* characters in *pattern* have matched characters in *text*, but a mismatch occurs when *pattern*[*i*] is compared with *text*[*j*].



To avoid backtracking, we must shift *pattern* to the right so that the search can continue with *text*[*j*] and *pattern*[*k*], for some $k < i$.



It is clear from this diagram that in order to do this, the first *k* characters of *pattern* must be identical to the *k* characters that precede *pattern*[*i*] and that *pattern*[*k*] must be different from *pattern*[*i*] so that *pattern*[*k*] has a chance of matching *text*[*j*]:



Let us denote this value *k* by *next*[*i*]. Thus, as we have just observed, *next*[*i*] is the position in *pattern* at which the search can continue by comparing *pattern*[*next*[*i*]] with the character *text*[*j*] that did not match *pattern*[*i*]; that is, we slide *pattern* to the right to align *pattern*[*next*[*i*]] with *text*[*j*] and continue searching at this point. If no such *k* exists, we take *next*[*i*] = –1 to indicate that the search is to resume with *pattern*[0] and *text*[*j* + 1]. (In this case, we can think of sliding *pattern* to the right to align the nonexistent character in position –1 with *text*[*j*] and resume the search.)

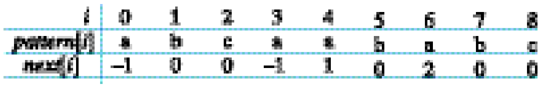
Knuth-Morris-Pratt Pattern Matching Algorithm

```
/*  Algorithm to find a pattern in a text string. The pattern is stored in
    positions 0 through $m$ of the array pattern, and the text is stored in
    positions 0 through $n$ of the array text.


    Input:     Strings text and pattern
    Return:    Location of pattern in text, –1 if not found

----------------------------------------------------------------------------------------------*/
  1. Initialize each of index, i, and j to 0.
     /*index is the beginning position of the substring of text being
        compared with pattern, and indices i and j run through pattern and
        text, respectively. */
  2. While $i = m$ and $j = n$:
        If pattern[i] = text[j] then // match
          Increment each of i and j
        Else do the following: // mismatch
        a. /* Slide pattern to the right the appropriate distance */
           Add $i – next[i]$ to index.
        b. /* Determine where the search is to continue */
           If $next[i] \neq –1$ then
             Set i equal to next[i].
          Else
             Set i equal to 0 and increment j by 1.
  3. If $i > m$ then index is the index of pattern in text; otherwise,
        pattern does not appear in text.
```

To illustrate, consider the following pattern

pattern = abcaababc

and assume that the we are given the following table of *next* values for this pattern:

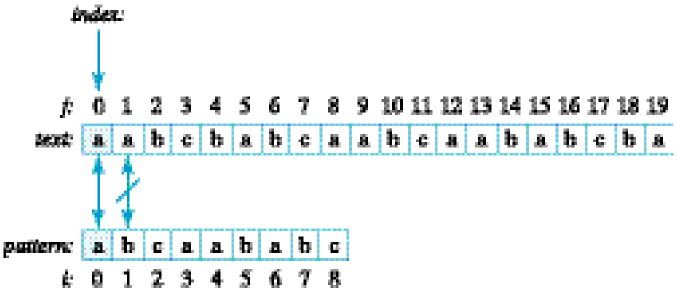


i	0	1	2	3	4	5	6	7	8
pattern[*i*]	a	b	c	a	a	b	a	b	c
next[*i*]	–1	0	0	–1	1	0	2	0	0

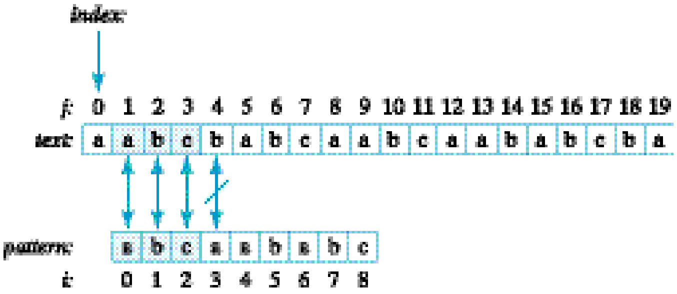
Now suppose that we wish to determine the index of *pattern* in

text = aabcbabcaabcaababcba

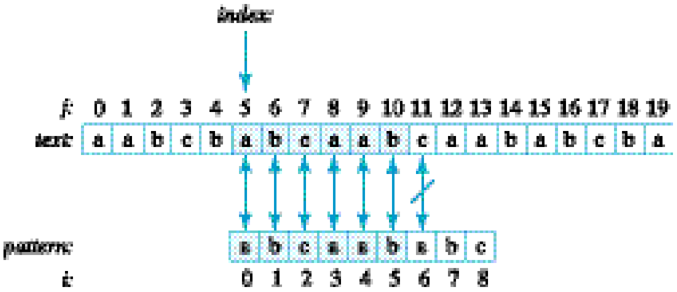
Initially, *index*, *i*, and *j* are 0. Both *text*[0] and *pattern*[0] are 'a', and so both *i* and *j* are incremented to 1. A mismatch now occurs:


index:
j: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
text: a a b c b a b c a a b c a a b a b c b a
pattern: a b c a a b a b c
i: 0 1 2 3 4 5 6 7 8

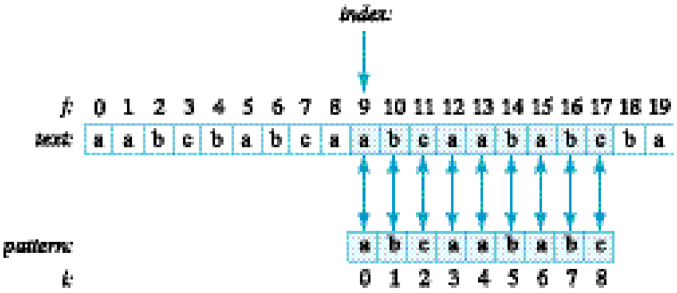
Since *next*[1] = 0, *index* is set to *index* + (1 – *next*[1]) = *index* + (1 – 0) = 1, *i* is set to *next*[1] = 0, *j* retains the value 1, and the search continues by comparing *pattern*[*next*[1]] = *pattern*[0] with *text*[1]. The next mismatch occurs when *i* = 3 and *j* = 4:


index:
j: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
text: a a b c b a b c a a b c a a b a b c b a
pattern: a b c a a b a b c
i: 0 1 2 3 4 5 6 7 8

Since *next*[3] = –1, *index* is set to *index* + (3 – (–1)) = 5, *i* is set to 0, and *j* is incremented to 5. The search then resumes by comparing *pattern*[0] with *text*[5], and continues until the next mismatch, when *j* = 11 and *i* = 6:

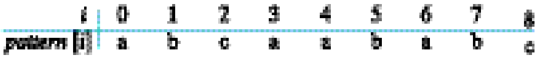

index:
j: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
text: a a b c b a b c a a b c a a b a b c b a
pattern: a b c a a b a b c
i: 0 1 2 3 4 5 6 7 8

Now, *next*[6] = 2, so index is updated to *index* + (6 – 2) = 9, *i* is set equal to *next*[6] = 2, *j* remains at 11, and the search resumes by comparing *pattern*[2] with *text*[11]. This search locates a substring of *text* that matches *pattern*,


index:
j: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
text: a a b c b a b c a a b c a a b a b c b a
pattern: a b c a a b a b c
i: 0 1 2 3 4 5 6 7 8

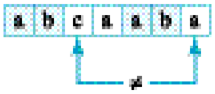
so the algorithm terminates with *index* = 9 as the index of *pattern* in *text*.

To complete our discussion of the Knuth-Morris-Pratt algorithm, we must describe how the table of *next* values is computed. Recall that *next*[*i*] is the length *k* of the longest prefix *pattern*[0], *pattern*[1], . . ., *pattern*[*k* - 1] of *pattern* that matches the *k* characters preceding *pattern*[*i*] but *pattern*[*k*] ≠ *pattern*[*i*]. For example, consider again the pattern “abcaababc”:

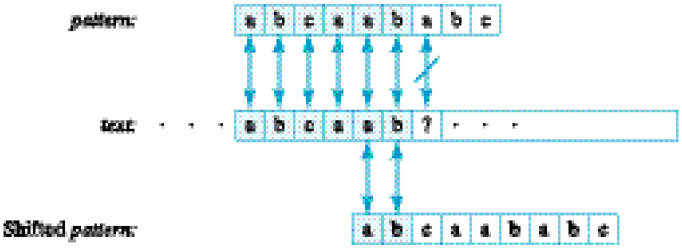


i	0	1	2	3	4	5	6	7	8
pattern [i]	a	b	c	a	a	b	a	b	c

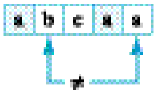
next[6] = 2 since the characters a, b in positions 0 and 1 match those in positions 4 and 5, and *pattern*[2] = ‘c’ is different from *pattern*[6] = ‘a’. The following diagram shows this matching prefix and suffix in abcaaba but that the characters that follow these in positions 2 and 6 do not match:



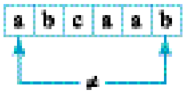
This property of *pattern* guarantees that if a mismatch occurs in comparing some character in *text* with *pattern*[6], the search can continue by comparing *pattern*[*next*[6]] = *pattern*[2] with this character:


pattern: a b c a a b a b c
text: . . . a b c a a b ? . . .
Shifted pattern: a b c a a b a b c

The determination that *next*[4] = 1 is similar to that for *next*[6]; the following diagram displays the matching prefix and suffix in abca and the fact that *pattern*[1] ≠ *pattern*[4]:

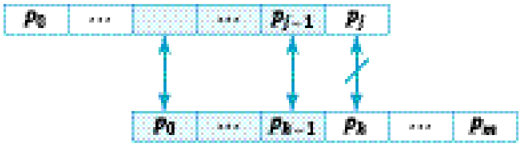


The calculation of *next*[5] requires a bit more care. In looking for a matching prefix and suffix in abcaa, we find one of length 1, that is, *pattern*[0] = *pattern*[4]; however, the characters that follow these matching substrings, *pattern*[1] and *pattern*[5] are not different. Thus *next*[5] is not 1. Since an empty prefix always matches an empty suffix, and since *pattern*[0] = ‘a’ differs from *pattern*[5] = ‘b’, we obtain *next*[5] = 0:



By similar analyses, the remaining values of the *next* table can be verified.

An algorithm for calculating *next* values using this method is essentially the same as the pattern-matching algorithm except that the pattern is matched against itself. To see this, consider again a general pattern $p_0 p_1 \cdots p_m$. Clearly *next*[0] must have the value –1 since there are no characters that precede *pattern*[0]. Now, if *next*[0], . . ., *next*[*j* - 1] have been determined, these values can be used to calculate *next*[*j*] as follows. We “slide” a copy of *pattern* across itself until we find a prefix (possibly empty) that matches the characters preceding *pattern*[*j*]:



If this prefix has length *k* (possibly 0) and *pattern*[*k*] ≠ *pattern*[*j*], then *next*[*j*] = *k* by definition. However, if *pattern*[*k*] = *pattern*[*j*], then clearly *next*[*j*] has the same value as *next*[*k*], which has already been calculated. This method of calculating *next* values is used in the following algorithm:

Algorithm for next Table

```
/*  Algorithm to compute next values for a pattern stored in
    positions 0 through m of the array pattern.

    Input:      String pattern
    Return:     The array next.

---------------------------------------------------------------------------------*/
```

```
1. Initialize $next[0]$ to –1, $k$ to –1, and $j$ to 0.
2. While $j \le m$:
    a. While ($k \ne -1$) and $pattern[k] \ne pattern[j]$
       Set $k$ equal to $next[k]$.
    b. Increment $k$ and $j$ by 1.
    c. If $pattern[j] = pattern[k]$ then
       Set $next[j]$ equal to $next[k]$.
       Else
       Set $next[j]$ equal to $k$.
```

To illustrate, consider the pattern used earlier

pattern = abcaababc

The following table traces the execution of this algorithm as it calculates the *next* table for this pattern.

Instruction	*k*	*j*	***next*** **Value Computed**
Initially	–1	0	*next*[0] = –1
2a	–1	0	
2b	0	1	
2c	0	1	*next*[1] = 0
2a	*next*[0] = –1	1	
2b	0	2	
2c	0	2	*next*[2] = 0
2a	*next*[0] = –1	2	
2b	0	3	
2c	0	3	*next*[3] = *next*[0] = –1
2a	0	3	
2b	1	4	
2c	1	4	*next*[4] =1
2a	*next*[1] = 0	4	
	0	4	
2b	1	5	
2c	1	5	*next*[5] = *next*[1] =0
2a	1	5	
2b	2	6	
2c	2	6	*next*[6] =2
2a	*next*[2] = 0	6	
	0	6	
2b	1	7	
2c	1	7	*next*[7] = *next*[1] = 0
2a	1	7	
2b	2	8	
2c	2	8	*next*[8] = *next*[2] = 0

The Knuth-Morris-Pratt solution to the pattern-matching problem has an interesting history. A theorem proved by Cook in 1970 states that any problem that can be solved using an abstract model of a computer called a *pushdown automaton* can be solved in time proportional to the size of the problem using an actual computer (more precisely, using a random access machine). In particular, this theorem implies that the existence of an algorithm for solving the pattern-matching problem in time proportional to $m + n$ where m and n are the maximum indices in arrays that store the pattern and text, respectively. Knuth and Pratt painstakingly reconstructed the proof of Cook's theorem and so constructed the pattern-matching algorithm described in this section. At approximately the same time, Morris constructed essentially the same algorithm while considering the practical problem of designing a text editor. Thus we see that not all algorithms are discovered by a "flash of insight" and that theoretical computer science does indeed sometimes lead to practical applications.

EXERCISES

Compute next tables for the patterns in Exercises 1–8.

1. A B R A C A D A B R A
2. A A A A A
3. M I S S I S S I P P I
4. I S S I S S I P P I
5. B A B B A B A B
6. 1 0 1 0 0 1 1
7. 1 0 0 1 0 1 1 1
8. 1 0 0 1 0 0 1 0 0 1

9–16. Construct tables tracing the action of the *next* table algorithm for the patterns in Exercises 1–8.