Previous Chapter Return to Table of Contents Next Chapter

CHAPTER 34: STRING MATCHING

Finding all occurrences of a pattern in a text is a problem that arises frequently in text-editing programs. Typically, the text is a document being edited, and the pattern searched for is a particular word supplied by the user. Efficient algorithms for this problem can greatly aid the responsiveness of the text-editing program. String-matching algorithms are also used, for example, to search for particular patterns in DNA sequences.

We formalize the ***string-matching problem*** as follows. We assume that the text is an array *T*[1 . . *n*] of length *n* and that the pattern is an array *P*[1 . . *m*] of length *m*. We further assume that the elements of *P* and *T* are characters drawn from a finite alphabet **Σ**. For example, we may have **Σ** = {`0, 1`} or **Σ** = {`a`, `b`, . . . , `z`}. The character arrays *P* and *T* are often called ***strings*** of characters.

We say that pattern *P* ***occurs with shift s*** in text *T* (or, equivalently, that pattern *P* ***occurs beginning at position s* + 1** in text *T*) if 0 ≤ *s* ≤ *n* - *m* and *T*[*s* + 1 . . *s* + *m*] = *P*[1 . . *m*] (that is, if *T*[*s* + *j*] = P[*j*], for 1 ≤ *j* ≤ *m*). If *P* occurs with shift *s* in *T*, then we call *s* a ***valid shift***; otherwise, we call *s* an ***invalid shift***. The string-matching problem is the problem of finding all valid shifts with which a given pattern *P* occurs in a given text *T*. Figure 34.1 illustrates these definitions.

This chapter is organized as follows. In Section 34.1 we review the naive brute-force algorithm for the string-matching problem, which has worst-case running time *O*((*n* - *m* + 1)*m*). Section 34.2 presents an interesting string-matching algorithm, due to Rabin and Karp. This algorithm also has worst-case running time *O*((*n* - *m* + 1)*m*), but it works much better on average and in practice. It also generalizes nicely to other pattern-matching problems. Section 34.3 then describes a string-matching algorithm that begins by constructing a finite automaton specifically designed to search for occurrences of the given pattern *P* in a text. This algorithm runs in time *O*(*n* + *m* |Σ|). The similar but much cleverer Knuth-Morris-Pratt (or KMP) algorithm is presented in Section 34.4; the KMP algorithm runs in time *O*(*n* + *m*). Finally, Section 34.5 describes an algorithm due to Boyer and Moore that is often the best practical choice, although its worst-case running time (like that of the Rabin-Karp algorithm) is no better than that of the naive string-matching algorithm.

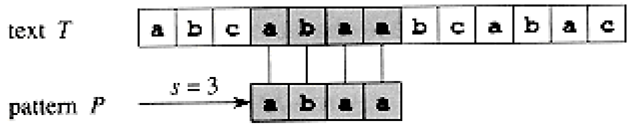


Figure 34.1 The string-matching problem. The goal is to find all occurrences of the pattern P = abaa in the text T = abcabaabcabac. The pattern occurs only once in the text, at shift s = 3. The shift s = 3 is said to be a valid shift. Each character of the pattern is connected by a vertical line to the matching character in the text, and all matched characters are shown shaded.

Notation and terminology

We shall let **Σ*** (read "sigma-star") denote the set of all finite-length strings formed using characters from the alphabet **Σ**. In this chapter, we consider only finite-length strings. The zero-length ***empty string***, denoted ε, also belongs to **Σ***. The length of a string *x* is denoted |*x*|. The ***concatenation*** of two strings *x* and *y*, denoted *xy*, has length |*x*| + |*y*| and consists of the characters from *x* followed by the characters from *y*.

We say that a string *w* is a ***prefix*** of a string *x*, denoted $w \sqsubset x$, if $x = wy$ for some string *y* ∈ **Σ***. Note that if $w \sqsubset x$, then |*w*| ≤ |*x*|. Similarly, we say that a string *w* is a ***suffix*** of a string *x,* denoted $w \sqsupset x$, if $x = yw$ for some $y \in \Sigma^*$. It follows from $w \sqsupset x$ that $|w| \le |x|$. The empty string ε is both a suffix and a prefix of every string. For example, we have `ab` $\sqsubset$ `abcca` and `cca` $\sqsubset$ `abcca`. It is useful to note that for any strings x and y and any character a, we have $x \sqsupset y$ if and only if $xa \sqsupset ya$. Also note that $\sqsubset$ and $\sqsupset$ are transitive relations. The following lemma will be useful later.

Lemma 34.1

Suppose that x, y, and z are strings such that $x \sqsupset z$ and $y \sqsupset z$. If $|x| \le |y|$, then $x \sqsupset y$. If $|x| \ge |y|$, then $y \sqsupset x$. If $|x| = |y|$, then $x = y$.

Proof See Figure 34.2 for a graphical proof.

For brevity of notation, we shall denote the k-character prefix $P[1 \,.\,.\, k]$ of the pattern $P[1 \,.\,.\, m]$ by P_k. Thus, $P_0 = \varepsilon$ and $P_m = P = P[1 \,.\,.\, m]$. Similarly, we denote the k-character prefix of the text T as T_k. Using this notation, we can state the string-matching problem as that of finding all shifts s in the range $0 \le s \le n - m$ such that $P \sqsupset T_{s-m}$.

In our pseudocode, we allow two equal-length strings to be compared for equality as a primitive operation. If the strings are compared from left to right and the comparison stops when a mismatch is discovered, we assume that the time taken by such a test is a linear function of the number of matching characters discovered. To be precise, the test "$x = y$" is assumed to take time $\Theta(t + 1)$, where t is the length of the longest string z such that $z \sqsubset x$ and $z \sqsubset y$.

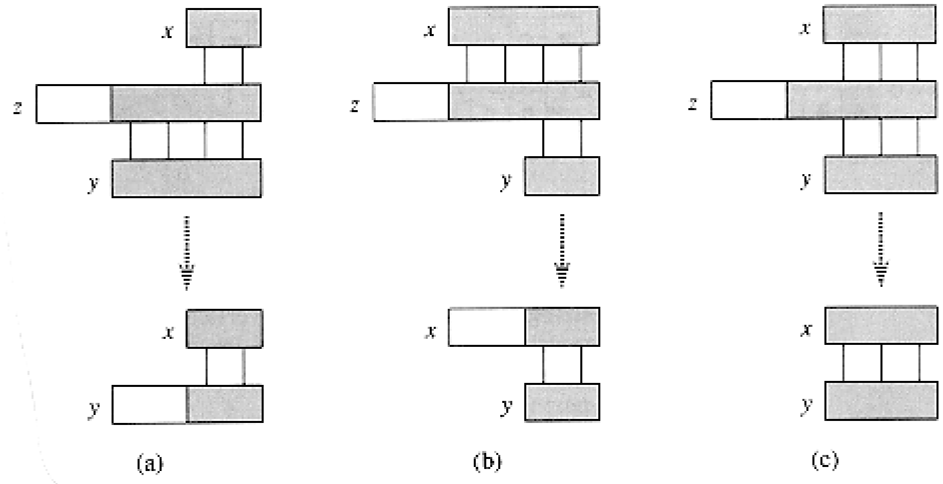


Figure 34.2 A graphical proof of Lemma 34.1. We suppose that $x \sqsupset z$ and $y \sqsupset z$. The three parts of the figure illustrate the three cases of the lemma. Vertical lines connect matching regions (shown shaded) of the strings. (a) If |x| ≤ |y|, then $x \sqsupset y$. (b) If |x| ≥ |y|, then $y \sqsupset x$. (c) If |x| = |y|, then x = y.

34.1 The naive string-matching algorithm

The naive algorithm finds all valid shifts using a loop that checks the condition $P[1 \,.\,.\, m] = T[s + 1 \,.\,.\, s + m]$ for each of the $n - m + 1$ possible values of s.

```
NAIVE-STRING-MATCHER(T, P)
```

```
1  n ← length[T]
2  m ← length[P]
3  for s ← 0 to n - m
4        do if P[1 . . m] = T[s + 1 . . s + m]
5              then print "Pattern occurs with shift" s
```

The naive string-matching procedure can be interpreted graphically as sliding a "template" containing the pattern over the text, noting for which shifts all of the characters on the template equal the corresponding characters in the text, as illustrated in Figure 34.3. The **for** loop beginning on line 3 considers each possible shift explicitly. The test on line 4 determines whether the current shift is valid or not; this test involves an implicit loop to check corresponding character positions until all positions match successfully or a mismatch is found. Line 5 prints out each valid shift *s*.

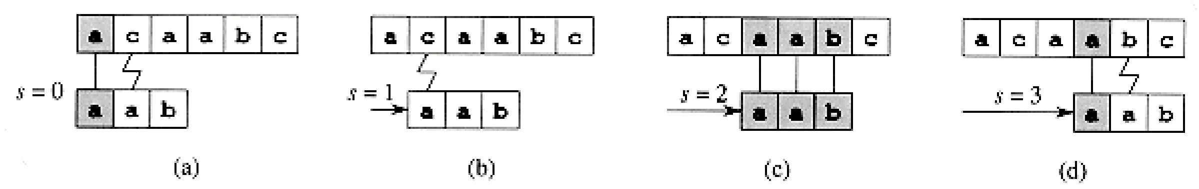


Figure 34.3 The operation of the naive string matcher for the pattern P = aab and the text T = acaabc. We can imagine the pattern P as a "template" that we slide next to the text. Parts (a)-(d) show the four successive alignments tried by the naive string matcher. In each part, vertical lines connect corresponding regions found to match (shown shaded), and a jagged line connects the first mismatched character found, if any. One occurrence of the pattern is found, at shift s = 2, shown in part (c).

Procedure NAIVE-STRING-MATCHER takes time $\Theta((n - m + 1)m)$ in the worst case. For example, consider the text string a^n (a string of *n* a's) and the pattern a^m. For each of the $n - m + 1$ possible values of the shift *s*, the implicit loop on line 4 to compare corresponding characters must execute *m* times to validate the shift. The worst-case running time is thus $\Theta((n - m + 1)m)$, which is $\Theta(n^2)$ if $m = \lfloor n/2 \rfloor$.

As we shall see, NAIVE-STRING-MATCHER is not an optimal procedure for this problem. Indeed, in this chapter we shall show an algorithm with a worst-case running time of $O(n + m)$. The naive string-matcher is inefficient because information gained about the text for one value of *s* is totally ignored in considering other values of *s*. Such information can be very valuable, however. For example, if *P* = aaab and we find that *s* = 0 is valid, then none of the shifts 1, 2, or 3 are valid, since *T*[4] = b. In the following sections, we examine several ways to make effective use of this sort of information.

Exercises

34.1-1

Show the comparisons the naive string matcher makes for the pattern *P* = 0001 in the text *T* = 000010001010001.

34.1-2

Show that the worst-case time for the naive string matcher to find the *first* occurrence of a pattern in a text is $\Theta((n - m + 1)(m - 1))$.

34.1-3

Suppose that all characters in the pattern *P* are different. Show how to accelerate NAIVE-STRING-MATCHER to run in time *O*(*n*) on an *n*-character text *T*.

34.1-4

Suppose that pattern *P* and text *T* are *randomly* chosen strings of length *m* and *n*, respectively, from the *d*-ary alphabet $\Sigma_d = \{0, 1, \ldots, d - 1\}$, where $d \geq 2$. Show that the *expected* number of character-to-character comparisons made by the implicit loop in line 4 of the naive algorithm is

$$(n - m + 1)\frac{1 - d^{-m}}{1 - d^{-1}} \leq 2(n - m + 1) .$$

(Assume that the naive algorithm stops comparing characters for a given shift once a mismatch is found or the entire pattern is matched.) Thus, for randomly chosen strings, the naive algorithm is quite efficient.

34.1-5

Suppose we allow the pattern *P* to contain occurrences of a ***gap character*** ◇ that can match an *arbitrary* string of characters (even one of zero length). For example, the pattern ab◇ba◇c occurs in the text `cabccbacbacab` as

$$\texttt{c}\underbrace{\texttt{ab}}_{\texttt{ab}}\underbrace{\texttt{cc}}_{\diamond}\underbrace{\texttt{ba}}_{\texttt{ba}}\underbrace{\texttt{cba}}_{\diamond}\underbrace{\texttt{c}}_{\texttt{c}}\texttt{ab}$$

and as

$$\texttt{c}\underbrace{\texttt{ab}}_{\texttt{ab}}\underbrace{\texttt{ccbac}}_{\diamond}\underbrace{\texttt{ba}}_{\texttt{ba}}\underbrace{\qquad}_{\diamond}\underbrace{\texttt{c}}_{\texttt{c}}\texttt{ab} .$$

Note that the gap character may occur an arbitrary number of times in the pattern but is assumed not to occur at all in the text. Give a polynomial-time algorithm to determine if such a pattern *P* occurs in a given text *T*, and analyze the running time of your algorithm.

34.2 The Rabin-Karp algorithm

Rabin and Karp have proposed a string-matching algorithm that performs well in practice and that also generalizes to other algorithms for related problems, such as two-dimensional pattern matching. The worst-case running time of the Rabin-Karp algorithm is *O*((*n* - *m* + 1)*m*), but it has a good average-case running time.

This algorithm makes use of elementary number-theoretic notions such as the equivalence of two numbers modulo a third number. You may want to refer to Section 33.1 for the relevant definitions.

For expository purposes, let us assume that $\Sigma = \{0, 1, 2, \ldots, 9\}$, so that each character is a decimal digit. (In the general case, we can assume that each character is a digit in radix-*d* notation, where $d = |\Sigma|$.) We can then view a string of *k* consecutive characters as representing a length-*k* decimal number. The character string `31415` thus corresponds to the decimal number 31,415. Given the dual interpretation of the input characters as both graphical symbols and digits, we find it convenient in this section to denote them as we would digits, in our standard text font.

Given a pattern *P*[1 . . *m*], we let *p* denote its corresponding decimal value. In a similar manner, given a text *T*[1 . . *n*], we let t_s denote the decimal value of the length-*m* substring *T*[*s* + 1 . . *s* + *m*], for *s* = 0, 1, . . . , *n* - *m*. Certainly, $t_s = p$ if and only if *T*[*s* + 1 . . *s* + *m*] = *P*[1 . . *m*]; thus, *s* is a valid shift if and only if $t_s = p$. If we could compute *p* in time *O*(*m*) and all of the t_i values in a total of *O*(*n*) time, then we could determine all valid shifts s in time $O(n)$ by comparing p with each of the t_s's. (For the moment, let's not worry about the possibility that p and the t_s's might be very large numbers.)

We can compute p in time $O(m)$ using Horner's rule (see Section 32.1):

$$p = P[m] + 10 (P[m - 1] + 10(P[m - 2] + . . . + 10(P[2] + 10P[1]) . . .)).$$

The value t_0 can be similarly computed from $T[1 \, . . \, m]$ in time O(m).

To compute the remaining values $t_1, t_2, . . . , t_{n-m}$ in time $O(n - m)$, it suffices to observe that t_{s+1} can be computed from t_s in constant time, since

$$t_s + 1 \quad = \quad 10(t_s - 10^m - 1T[s + 1]) + T[s + m + 1].$$

(34.1)

For example, if m= 5 and t_s = 31415, then we wish to remove the high-order digit $T[s + 1]$ = 3 and bring in the new low-order digit (suppose it is $T[s + 5 + 1]$ = 2) to obtain

$$t_s+1 = 10(31415 - 10000.3) + 2$$

$$= 14152 .$$

Subtracting 10^m-1 $T[s+1]$ removes the high-order digit from t_s, multiplying the result by 10 shifts the number left one position, and adding $T[s + m + 1]$ brings in the appropriate low-order digit. If the constant 10^m-1 is precomputed (which can be done in time O(1g m) using the techniques of Section 33.6, although for this application a straightforward $O(m)$ method is quite adequate), then each execution of equation (34.1) takes a constant number of arithmetic operations. Thus, p and t_0, t_1, . . . , t_n-m can all be computed in time $O(n + m)$, and we can find all occurrences of the pattern $P[1 \, . . \, m]$ in the text $T[1 \, . . \, n]$ in time $O(n + m)$.

The only difficulty with this procedure is that p and t_s may be too large to work with conveniently. If P contains m characters, then assuming that each arithmetic operation on p (which is m digits long) takes "constant time" is unreasonable. Fortunately, there is a simple cure for this problem, as shown in Figure 34.4 : compute p and the t_s's modulo a suitable modulus q. Since the computation of p, t_0, and the recurrence (34.1) can all be performed modulo q, we see that p and all the t_s's can be computed modulo q in time $O(n + m)$. The modulus q is typically chosen as a prime such that 10q just fits within one computer word, which allows all of the necessary computations to be performed with single-precision arithmetic.

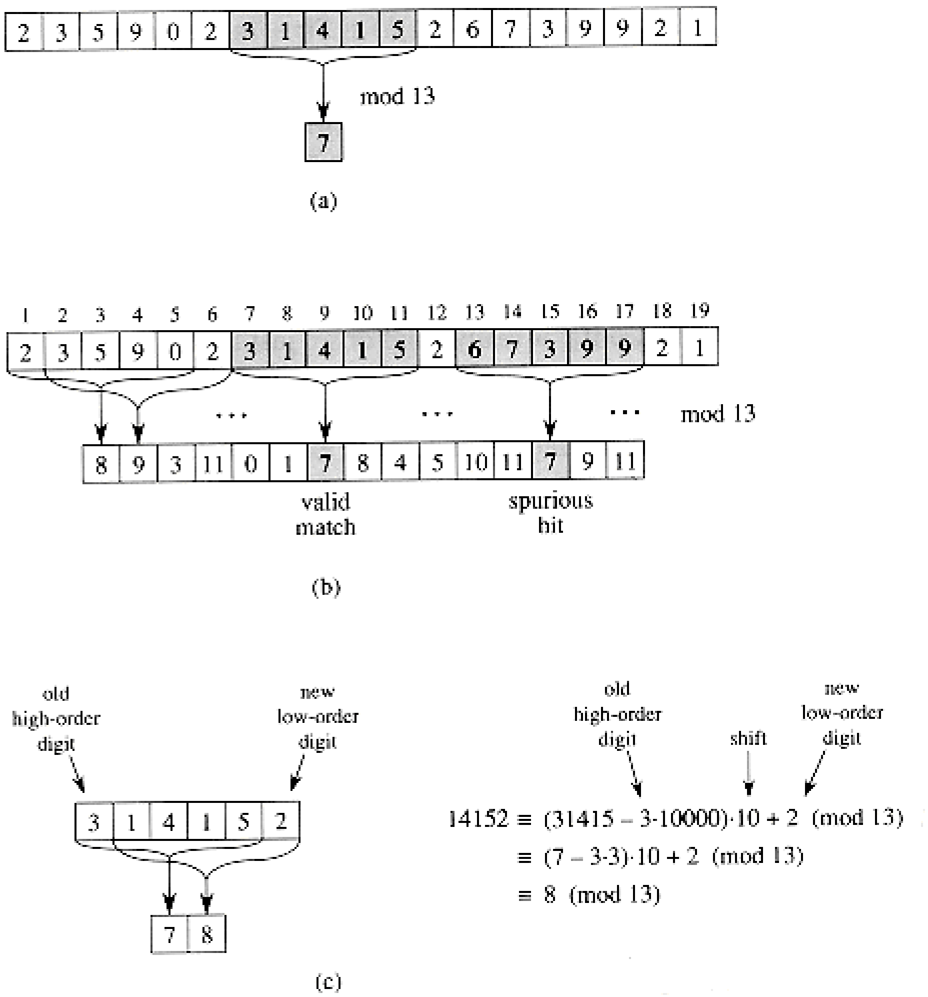


Figure 34.4 The Rabin-Karp algorithm. Each character is a decimal digit, and we compute values modulo 13. (a) A text string. A window of length 5 is shown shaded. The numerical value of the shaded number is computed modulo 13, yielding the value 7. (b) The same text string with values computed modulo 13 for each possible position of a length-5 window. Assuming the pattern P = 31415, we look for windows whose value modulo 13 is 7, since 31415 ≡ 7 (mod 13). Two such windows are found, shown shaded in the figure. The first, beginning at text position 7, is indeed an occurrence of the pattern, while the second, beginning at text position 13, is a spurious hit. (c) Computing the value for a window in constant time, given the value for the previous window. The first window has value 31415. Dropping the high-order digit 3, shifting left (multiplying by 10), and then adding in the low-order digit 2 gives us the new value 14152. All computations are performed modulo 13, however, so the value for the first window is 7, and the value computed for the new window is 8.

In general, with a *d*-ary alphabet {0, 1, . . . ,*d* - 1}, we choose *q* so that *d q* fits within a computer word and adjust the recurrence equation (34.1) to work modulo *q*, so that it becomes

$$t_s+1 = (d(t_s - T[s + 1]h) + T[s + m + 1]) \bmod q \ ,$$

(34.2)

where $h \equiv d^m$-1 (mod *q*) is the value of the digit "1" in the high-order position of an *m*-digit text window.

The ointment of working modulo *q* now contains a fly, however, since $t_s \equiv p$ (mod *q*) does not imply that $t_s = p$. On the other hand, if $t_s \not\equiv p$ (mod *q*), then we definitely have that $t_s \neq p$, so that shift *s* is invalid. We can thus use the test $t_s \equiv p \pmod q$ as a fast heuristic test to rule out invalid shifts s. Any shift s for which $t_s \equiv p \pmod q$ must be tested further to see if s is really valid or we just have a ***spurious hit***. This testing can be done by explicitly checking the condition $P[1 \,.\,.\, m] = T[s + 1 \,.\,.\, s + m]$. If q is large enough, then we can hope that spurious hits occur infrequently enough that the cost of the extra checking is low.

The following procedure makes these ideas precise. The inputs to the procedure are the text T, the pattern P, the radix d to use (which is typically taken to be $|\boldsymbol{\Sigma}|$), and the prime q to use.

```
RABIN-KARP-MATCHER(T, P, d, q)
1  $n \leftarrow length[T]$
2  $m \leftarrow length[P]$
3  $h \leftarrow d^{m}-1^{\,mod}$ q
4  $p \leftarrow 0$
5  $t_0 \leftarrow 0$
6  for $i \leftarrow 1$ to $m$
7        do $p \leftarrow (dp + P[i]) \bmod q$
8           $t_0 \leftarrow (dt_0 + T[i]) \bmod q$
9  for $s \leftarrow 0$ to $n - m$
10       do if $p = t_s$
11             then if $P[1 \,.\,.\, m] = T[s + 1 \,.\,.\, s + m]$
12                     then "Pattern occurs with shift" $s$
13          if $s < n - m$
14            then $t_s$+1 $\leftarrow (d(t_s - T[s + 1]h) + T[s + m + 1]) \bmod q$
```

The procedure RABIN-KARP-MATCHER works as follows. All characters are interpreted as radix-d digits. The subscripts on t are provided only for clarity; the program works correctly if all the subscripts are dropped. Line 3 initializes h to the value of the high-order digit position of an m-digit window. Lines 4-8 compute p as the value of $P[1 \,.\,.\, m] \bmod q$ and t_0 as the value of $T[1 \,.\,.\, m] \bmod q$. The **for** loop beginning on line 9 iterates through all possible shifts s. The loop has the following invariant: whenever line 10 is executed, $t_s = T[s + 1 \,.\,.\, s + m] \bmod q$. If $p = t_s$ in line 10 (a "hit"), then we check to see if $P[1 \,.\,.\, m] = T[s + 1 \,.\,.\, s + m]$ in line 11 to rule out the possibility of a spurious hit. Any valid shifts found are printed out on line 12. If $s < n - m$ (checked in line 13), then the **for** loop is to be executed at least one more time, and so line 14 is first executed to ensure that the loop invariant holds when line 10 is again reached. Line 14 computes the value of $t_{s+}1 \bmod q$ from the value of $t_s \bmod q$ in constant time using equation (34.2) directly.

The running time of RABIN-KARP-MATCHER is $\Theta((n - m + 1)m)$ in the worst case, since (like the naive string-matching algorithm) the Rabin-Karp algorithm explicitly verifies every valid shift. If $P = \mathrm{a}^m$ and $T = \mathrm{a}^n$, then the verifications take time $\Theta((n - m + 1)m)$, since each of the $n - m + 1$ possible shifts is valid. (Note also that the computation of d^m-1 mod q on line 3 and the loop on lines 6-8 take time $O(m) = O((n - m + 1\ )m)$.)

In many applications, we expect few valid shifts (perhaps $O(1)$ of them), and so the expected running time of the algorithm is $O(n + m)$ plus the time required to process spurious hits. We can base a heuristic analysis on the assumption that reducing values modulo q acts like a random mapping from $\mathbf{\Sigma}^*$ to $\mathbf{Z}_q$. (See the discussion on the use of division for hashing in Section 12.3.1. It is difficult to formalize and prove such an assumption, although one viable approach is to assume that q is chosen randomly from integers of the appropriate size. We shall not pursue this formalization here.) We can then expect that the number of spurious hits is $O(n/q)$, since the chance that an arbitrary t_s will be equivalent to p, modulo q, can be estimated as $1/q$. The expected amount of time taken by the Rabin-Karp algorithm is then

$$O(n) + O(m(v + n/q)),$$

where v is the number of valid shifts. This running time is $O(n)$ if we choose $q \geq m$. That is, if the expected number of valid shifts is small ($O(1)$) and the prime q is chosen to be larger than the length of the pattern, then we can expect the Rabin-Karp procedure to run in time $O(n + m)$.

Exercises

34.2-1

Working modulo $q = 11$, how many spurious hits does the Rabin-Karp matcher encounter in the text T = 3141592653589793 when looking for the pattern P = 26?

34.2-2

How would you extend the Rabin-Karp method to the problem of searching a text string for an occurrence of any one of a given set of k patterns?

34.2-3

Show how to extend the Rabin-Karp method to handle the problem of looking for a given m X m pattern in an n X n array of characters. (The pattern may be shifted vertically and horizontally, but it may not be rotated.)

34.2-4

Alice has a copy of a long n-bit file $A = \langle a_n$-1, a_n-2, . . . , $a_0\rangle$, and Bob similarly has an n-bit file $B = \langle b_{n-1}, b_{n-2}, . . . ,b_0\rangle$. Alice and Bob wish to know if their files are identical. To avoid transmitting all of A or B, they use the following fast probabilistic check. Together, they select a prime $q > 1000n$ and randomly select an integer x from $\{0, 1, . . . , n - 1\}$. Then, Alice evaluates

$$A(x) = \left(\sum_{i=0}^{n} a_i x^i\right) \bmod q$$

and Bob similarly evaluates $B(x)$. Prove that if $A \neq B$, there is at most one chance in 1000 that $A(x) = B(x)$, whereas if the two files are the same, $A(x)$ is necessarily the same as $B(x)$. (*Hint*: See Exercise 33.4-4.)

34.3 String matching with finite automata

Many string-matching algorithms build a finite automaton that scans the text string T for all occurrences of the pattern P. This section presents a method for building such an automaton. These string-matching automata are very efficient: they examine each text character *exactly once*, taking constant time per text character. The time used--after the automaton is built--is therefore $\Theta(n)$. The time to build the automaton, however, can be large if Σ is large. Section 34.4 describes a clever way around this problem.

We begin this section with the definition of a finite automaton. We then examine a special string-matching automaton and show how it can be used to find occurrences of a pattern in a text. This discussion includes details on how to simulate the behavior of a string-matching automaton on a given text. Finally, we shall show how to construct the string-matching automaton for a given input pattern.

Finite automata

A ***finite automaton*** *M* is a 5-tuple ($Q, q_0, A, \Sigma, \delta$), where

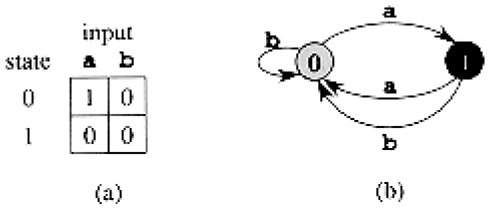
- *Q* is a finite set of **states**,

- $q_0 \in Q$ is the **start state**,

- $A \subseteq Q$ is a distinguished set of **accepting states**,

- Σ is a finite **input alphabet**,

- δ is a function from *Q* X Σ into *Q*, called the **transition function** of *M*.



state	input a	input b
0	1	0
1	0	0

Figure 34.5 A simple two-state finite automaton with state set Q = {0, 1}, start state q_0 = 0, and input alphabet Σ = {a, b}. (a) A tabular representation of the transition function δ. (b) An equivalent state-transition diagram. State 1 is the only accepting state (shown blackened). Directed edges represent transitions. For example, the edge from state 1 to state 0 labeled b indicates δ(1, b) = 0. This automaton accepts those strings that end in an odd number of a's. More precisely, a string x is accepted if and only if x = yz, where y = ε or y ends with a b, and z = a^k, where k is odd. For example, the sequence of states this automaton enters for input abaaa (including the start state) is $\langle$0, 1, 0, 1, 0, 1$\rangle$, and so it accepts this input. For input abbaa, the sequence of states is $\langle$0, 1, 0, 0, 1, 0$\rangle$, and so it rejects this input.

The finite automaton begins in state q_0 and reads the characters of its input string one at a time. If the automaton is in state *q* and reads input character *a*, it moves ("makes a transition") from state *q* to state δ (*q, a*). Whenever its current state *q* is a member of *A*, the machine *M* is said to have ***accepted*** the string read so far. An input that is not accepted is said to be ***rejected***. Figure 34.5 illustrates these definitions with a simple two-state automaton.

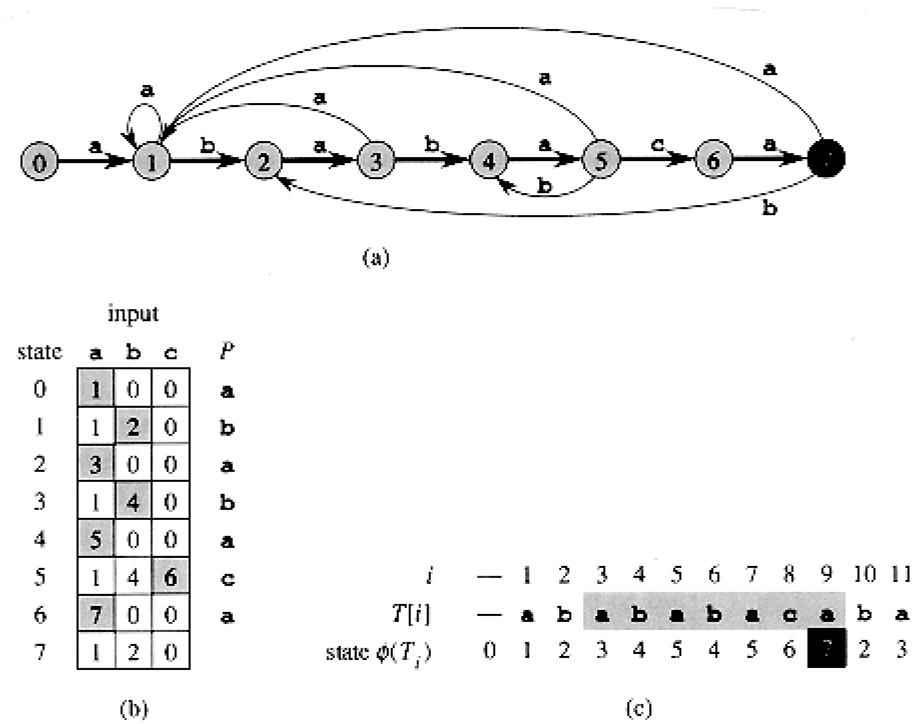
A finite automaton *M* induces a function ø, called the ***final-state function***, from Σ^* to *Q* such that ø(*w*) is the state *M* ends up in after scanning the string *w*. Thus, *M* accepts a string *w* if and only if ø(*w*) $\in$ *A*. The function ø is defined by the recursive relation

$$\phi(\varepsilon) = q_0 ,$$

$$\phi(wa) = \delta(\phi(w), a) \quad for\ w \in \Sigma^*,\ a \in \Sigma.$$

String-matching automata

There is a string-matching automaton for every pattern *P*; this automaton must be constructed from the pattern in a preprocessing step before it can be used to search the text string. Figure 34.6 illustrates this construction for the pattern *P* = `ababaca`. From now on, we shall assume that *P* is a given fixed pattern string; for brevity, we shall not indicate the dependence upon *P* in our notation.



input

state	a	b	c	P
0	1	0	0	a
1	1	2	0	b
2	3	0	0	a
3	1	4	0	b
4	5	0	0	a
5	1	4	6	c
6	7	0	0	a
7	1	2	0	

i	—	1	2	3	4	5	6	7	8	9	10	11
$T[i]$	—	a	b	a	b	a	b	a	c	a	b	a
state $\phi(T_i)$	0	1	2	3	4	5	4	5	6	7	2	3

Figure 34.6 (a) A state-transition diagram for the string-matching automaton that accepts all strings ending in the string ababaca. State 0 is the start state, and state 7 (shown blackened) is the only accepting state. A directed edge from state i to state j labeled a represents δ(i,a) = j. The right-going edges forming the "spine" of the automaton, shown heavy in the figure, correspond to successful matches between pattern and input characters. The left-going edges correspond to failing matches. Some edges corresponding to failing matches are not shown; by convention, if a state i has no outgoing edge labeled a for some a ∈ Σ, then δ(i,a) = 0. (b) The corresponding transition function δ, and the pattern string P = `ababaca`. The entries corresponding to successful matches between pattern and input characters are shown shaded. (c) The operation of the automaton on the text T = `abababacaba`. Under each text character T[i] is given the state ø(T_i) the automaton is in after processing the prefix T_i. One occurrence of the pattern is found, ending in position 9.

In order to specify the string-matching automaton corresponding to a given pattern *P*[1 . . *m*], we first define an auxiliary function **Σ**, called the ***suffix function*** corresponding to *P*. The function **Σ** is a mapping from **Σ*** to {0, 1, . . . , *m*} such that **Σ**(*x*) is the length of the longest prefix of *P* that is a suffix of *x*:

$$\sigma(x) = \max\{k : P_k \sqsupset x\} .$$

The suffix function **Σ** is well defined since the empty string P_0 = ε is a suffix of every string. As examples, for the pattern *P* = `ab`, we have **Σ**(ε) = 0, **Σ**(`ccaca`) = 1, and **Σ**(`ccab`) = 2. For a pattern *P* of length *m*, we have **Σ**(x) = *m* if and only if $P \sqsupset x$. It follows from the definition of the suffix function that if $x \sqsupset y$, then **Σ**(x) ≤ **Σ**(*y*).

We define the string-matching automaton corresponding to a given pattern P[1 . . *m*] as follows.

- The state set *Q* is {0, 1, . . . , *m*}. The start state q_0 is state 0, and state *m* is the only accepting state.

- The transition function δ is defined by the following equation, for any state *q* and character *a*:

$\delta(q, a) = \Sigma(P_q a)$.

(34.3)

Here is an intuitive rationale for defining δ (q, a) = $\Sigma(P_q\ a)$. The machine maintains as an invariant of its operation that

$\Phi(T_i) = \Sigma(T_i)$;

(34.4)

this result is proved as Theorem 34.4 below. In words, this means that after scanning the first *i* characters of the text string *T*, the machine is in state $\Phi(T_i) = q$, where $q = \Sigma\ (T_i)$ is the length of the longest suffix of T_i that is also a prefix of the pattern *P*. If the next character scanned is $T[i + 1] = a$, then the machine should make a transition to state $\Sigma(T_i + 1) = \Sigma(T_i a)$. The proof of the theorem shows that $\Sigma(T_i a) = \Sigma(P_q a)$. That is, to compute the length of the longest suffix of $T_i a$ that is a prefix of *P*, we can compute the longest suffix of $P_q a$ that is a prefix of *P*. At each state, the machine only needs to know the length of the longest prefix of *P* that is a suffix of what has been read so far. Therefore, setting δ(q, a) = $\Sigma(P_q a)$ maintains the desired invariant (34.4). This informal argument will be made rigorous shortly.

In the string-matching automaton of Figure 34.6, for example, we have δ(5, `b`) = 4. This follows from the fact that if the automaton reads a b in state $q = 5$, then P_q`b` = `ababab`, and the longest prefix of *P* that is also a suffix of `ababab` is P_4 = `abab`.

To clarify the operation of a string-matching automaton, we now give a simple, efficient program for simulating the behavior of such an automaton (represented by its transition function δ) in finding occurrences of a pattern *P* of length *m* in an input text $T[1 \mathinner{..} n]$. As for any string-matching automaton for a pattern of length *m*, the state set *Q* is $\{0,1, \ldots , m\}$, the start state is 0, and the only accepting state is state *m*.

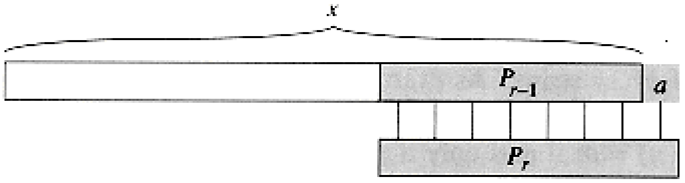


Figure 34.7 An illustration for the proof of Lemma 34.2. The figure shows that r ≤ Σ(x) + 1, where r = Σ (xa).

```
FINITE-AUTOMATON-MATCHER(T, δ, m)
n ← length[T]
q ← 0
for i ← 1 to n
      do q ← δ (q, T[i])
         if q = m
            then s ← i - m
                 print "Pattern occurs with shift" s
```

The simple loop structure of FINITE-AUTOMATON-MATCHER implies that its running time on a text string of length n is $O(n)$. This running time, however, does not include the time required to compute the transition function δ. We address this problem later, after proving that the procedure FINITE-AUTOMATON-MATCHER operates correctly.

Consider the operation of the automaton on an input text $T[1 \mathinner{.\,.} n]$. We shall prove that the automaton is in state $\Sigma(T_j)$ after scanning character $T[i]$. Since $\Sigma(T_i) = m$ if and only if $P \sqsupset T_i$, the machine is in the accepting state m if and only if the pattern P has just been scanned. To prove this result, we make use of the following two lemmas about the suffix function Σ.

Lemma 34.2

For any string x and character a, we have $\Sigma(xa) \le \Sigma(x) + 1$.

Proof Referring to Figure 34.7, let $r = \Sigma(xa)$. If $r = 0$, then the conclusion $r \le \Sigma(x) + 1$ is trivially satisfied, by the nonnegativity of $\Sigma(x)$. So assume that $r > 0$. Now, $P_r \sqsupset xa,$, by the definition of Σ. Thus, $P_{r-1} \sqsupset x,$, by dropping the a from the end of P_r and from the end of xa. Therefore, $r - 1 \le \Sigma(x)$, since $\Sigma(x)$ is largest k such that $P_k \sqsupset x$.

Lemma 34.3

For any string x and character a, if $q = \Sigma(x)$, then $\Sigma(xa) = \Sigma(P_qa)$.

Proof From the definition of Σ, we have $P_q \sqsupset x$. As Figure 34.8 shows, $P_qa \sqsupset xa$. If we let $r = \Sigma(xa)$, then $r \le q + 1$ by Lemma 34.2. Since $P_qa \sqsupset xa,\ P_r \sqsupset xa$, and $|Pr| \le |Pqa|$, Lemma 34.1 implies that $P_r \sqsupset P_qa$. Therefore, $r \le \Sigma(P_qa)$, that is, $\Sigma(xa) \le \Sigma(P_qa)$. But we also have $\Sigma(P_qa) \le \Sigma(xa)$, since $P_qa \sqsupset xa$. Thus, $\Sigma(xa) = \Sigma(P_qa)$.

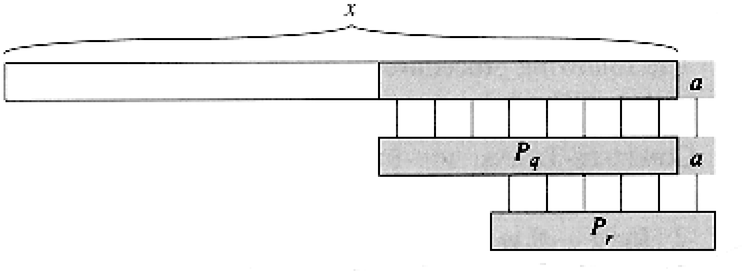


Figure 34.8 An illustration for the proof of Lemma 34.3. The figure shows that r = $\Sigma(P_q a)$, where q = Σ(x) and r = Σ(xa).

We are now ready to prove our main theorem characterizing the behavior of a string-matching automaton on a given input text. As noted above, this theorem shows that the automaton is merely keeping track, at each step, of the longest prefix of the pattern that is a suffix of what has been read so far.

Theorem 34.4

If ϕ is the final-state function of a string-matching automaton for a given pattern P and $T[1 \mathinner{.\,.} n]$ is an input text for the automaton, then

$$\phi(T_i) = \sigma(T_i)$$

for $i = 0, 1, \ldots, n$.

Proof The proof is by induction on *i*. For *i* = 0, the theorem is trivially true, since $T_0 = \varepsilon$. Thus,

$$\phi(T_0) = \sigma(T_0) = 0 .$$

Now, we assume that $\phi(T_i) = \sigma(T_i)$ and prove that $\phi(T_{i+1}) = \sigma(T_{i+1})$. Let *q* denote $\phi(T_i)$, and let *a* denote *T*[*i* + 1]. Then,

$$\begin{aligned}
\phi(T_{i+1}) &= \phi(T_i a) && \text{(by the definitions of } T_{i+1} \text{ and } a\text{)} \\
&= \delta(\phi(T_i), a) && \text{(by the definition of } \phi\text{)} \\
&= \delta(q, a) && \text{(by the definition of } q\text{)} \\
&= \sigma(P_q a) && \text{(by the definition (34.3) of } \delta\text{)} \\
&= \sigma(T_i a) && \text{(by Lemma 34.3 and induction)} \\
&= \sigma(T_{i+1}) && \text{(by the definition of } T_{i+1}\text{) .}
\end{aligned}$$

By induction, the theorem is proved.

By Theorem 34.4, if the machine enters state *q* on line 4, then *q* is the largest value such that $P_q \sqsupset T_i$. Thus, we have *q* = *m* on line 5 if and only if an occurrence of the pattern *P* has just been scanned. We conclude that `FINITE-AUTOMATON-MATCHER` operates correctly.

Computing the transition function

The following procedure computes the transition function δ *from a given pattern* P*[1 . . m].*

```
COMPUTE-TRANSITION-FUNCTION(P, Σ)

1 m ← length[P]

2 for q ← 0 to m

3     do for each character a ∈ *

4            do k ← min (m + 1, q + 2)

5               repeat k ← k - 1

6                  until $P_k \sqsupset P_q a$

7            δ(q, a) ← k

8 return δ
```

This procedure computes δ(q, a*) in a straightforward manner according to its definition. The nested loops beginning on lines 2 and 3 consider all states* q *and characters* a*, and lines 4-7 set* δ(*q, a*) to be the largest *k* such that $P_k \sqsupset P_q a$. The code starts with the largest conceivable value of *k*, which is min(*m, q* + 1), and decreases *k* until $P_k \sqsupset P_q a$.

The running time of COMPUTE-TRANSITION-FUNCTION is $O(m^3 |\Sigma|)$, because the outer loops contribute a factor of $m |\Sigma|$, the inner **repeat** loop can run at most m + 1 times, and the test $P_k \sqsupset P_q a$ on line 6 can require comparing up to m characters. Much faster procedures exist; the time required to compute δ from P can be improved to $O(m|\Sigma|)$ by utilizing some cleverly computed information about the pattern P (see Exercise 34.4-6). With this improved procedure for computing δ, the total running time to find all occurrences of a length-m pattern in a length-n text over an alphabet Σ is $O(n + m|\Sigma|)$.

Exercises

34.3-1

Construct the string-matching automaton for the pattern P = `aabab` and illustrate its operation on the text string T = `aaababaabaababaab`.

34.3-2

Draw a state-transition diagram for a string-matching automaton for the pattern `ababbabbababbababbabb` over the alphabet $\Sigma = \{$`a`, `b`$\}$.

34.3-3

We call a pattern P ***nonoverlappable*** if $P_k \sqsupset P_q$ implies k = 0 or $k = q$. Describe the state-transition diagram of the string-matching automaton for a nonoverlappable pattern.

34.3-4

Given two patterns P and P', describe how to construct a finite automaton that determines all occurrences of *either* pattern. Try to minimize the number of states in your automaton.

34.3-5

Given a pattern P containing gap characters (see Exercise 34.1-5), show how to build a finite automaton that can find an occurrence of P in a text T in $O(n)$ time, where $n = |T|$.

34.4 The Knuth-Morris-Pratt algorithm

We now present a linear-time string-matching algorithm due to Knuth, Morris, and Pratt. Their algorithm achieves a $\Theta(n + m)$ running time by avoiding the computation of the transition function δ altogether, and it does the pattern matching using just an auxiliary function $\pi[1 \mathinner{..} m]$ precomputed from the pattern in time $O(m)$. The array π allows the transition function δ to be computed efficiently (in an amortized sense) "on the fly" as needed. Roughly speaking, for any state $q = 0, 1, \ldots, m,$and any character $a \in \Sigma$, the value $\pi[q]$ contains the information that is independent of a and is needed to compute $\delta(q, a)$. (This remark will be clarified shortly.) Since the array π has only m entries, whereas δ has $O(m |\Sigma|)$ entries, we save a factor of Σ in the preprocessing by computing π rather than δ.

The prefix function for a pattern

The prefix function for a pattern encapsulates knowledge about how the pattern matches against shifts of itself. This information can be used to avoid testing useless shifts in the naive pattern-matching algorithm or to avoid the precomputation of δ for a string-matching automaton.

Consider the operation of the naive string matcher. Figure 34.9(a) shows a particular shift *s* of a template containing the pattern *P* = ababaca against a text *T*. For this example, *q* = 5 of the characters have matched successfully, but the 6th pattern character fails to match the corresponding text character. The information that *q* characters have matched successfully determines the corresponding text characters. Knowing these *q* text characters allows us to determine immediately that certain shifts are invalid. In the example of the figure, the shift *s* + 1 is necessarily invalid, since the first pattern character, an a, would be aligned with a text character that is known to match with the second pattern character, a b. The shift *s* + 2 shown in part (b) of the figure, however, aligns the first three pattern characters with three text characters that must necessarily match. In general, it is useful to know the answer to the following question:

Given that pattern characters *P*[1 . . *q*] match text characters *T*[*s* + 1 . . *s* + *q*], what is the least shift *s'* > *s* such that

$$P[1 \, . \, . \, k] = T[s' \; 1 \, . \, . \, s' \; k],$$

(34.5)

where *s*' + *k* = *s* + *q*?

Such a shift *s*' is the first shift greater than *s* that is not necessarily invalid due to our knowledge of *T*[*s* + 1 . . *s* + *q*]. In the best case, we have that *s'* = *s* + *q*, and shifts *s* + 1, *s* + 2, . . . , *s* + *q* - 1 are all immediately ruled out. In any case, at the new shift *s*' we don't need to compare the first *k* characters of *P* with the corresponding characters of *T*, since we are guaranteed that they match by equation (34.5).

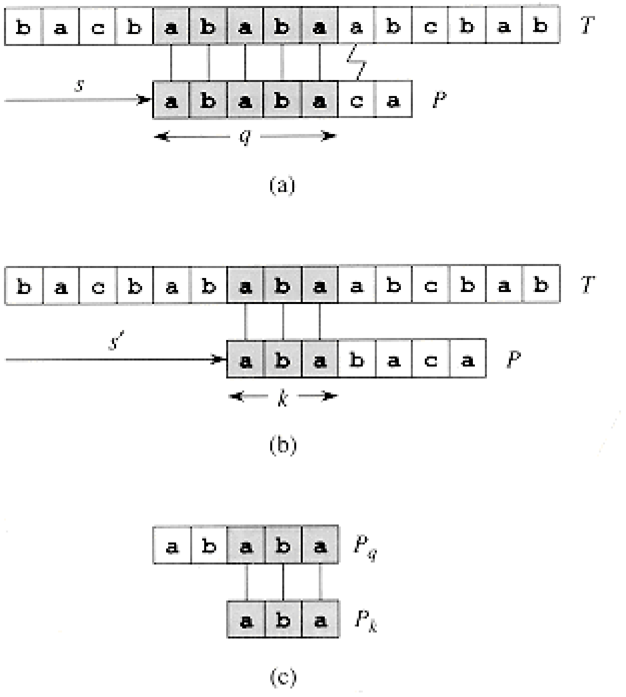


Figure 34.9 The prefix function Π. (a) The pattern P = ababaca is aligned with a text T so that the first q = 5 characters match. Matching characters, shown shaded, are connected by vertical lines. (b) Using only our knowledge of the 5 matched characters, we can deduce that a shift of s + 1 is invalid, but that a shift of s' = s + 2 is consistent with everything we know about the text and therefore is potentially valid. (c) The useful information for such deductions can be precomputed by comparing the pattern with itself. Here, we see that the longest prefix of P that is also a suffix of P_5 is P_3. This infomation is precomputed and represented in the array Π, so that Π[5] = 3. Given that q characters have matched successfully at shift s, the next potentially valid shift is at s' = s + (q - Π[q]).

The necessary information can be precomputed by comparing the pattern against itself, as illustrated in Figure 34.9(*c*). Since $T[s' + 1 \, .. \, s' + k]$ is part of the known portion of the text, it is a suffix of the string P_q. Equation (34.5) can therefore be interpreted as asking for the largest $k < q$ such that $P_k \sqsupset P_q$. Then, $s' = s + (q - k)$ is the next potentially valid shift. It turns out to be convenient to store the number k of matching characters at the new shift **s**′, rather than storing, say, $s' - s$. This information can be used to speed up both the naive string-matching algorithm and the finite-automaton matcher.

We formalize the precomputation required as follows. Given a pattern $P[1 \, .. \, m]$, the ***prefix function*** for the pattern P is the function $\Pi : \{1, 2, \ldots, m\} \rightarrow \{0, 1, \ldots, m - 1\}$ such that

$$\pi[q] = \max\{k : k < q \text{ and } P_k \sqsupset P_q\} \ .$$

That is, $\Pi[q]$ is the length of the longest prefix of P that is a proper suffix of P_q. As another example, Figure 34.10(a) gives the complete prefix function Π for the pattern `ababababca`.

The Knuth-Morris-Pratt matching algorithm is given in pseudocode below as the procedure KMP-MATCHER. It is mostly modeled after FINITE-AUTOMATON-MATCHER, as we shall see. KMP-MATCHER calls the auxiliary procedure COMPUTE-PREFIX-FUNCTION to compute Π.

```
KMP-MATCHER(T, P)
n ← length[T]
m ← length[P]
Π ← COMPUTE-PREFIX-FUNCTION(P)
q ← 0
for i ← 1 to n
    do while q > 0 and P[q + 1] ≠ T[i]
           do q ← Π[q]
       if P[q + 1] = T[i]
          then q ← q + 1
       if q = m
          then print "Pattern occurs with shift" i - m
               q ← Π[q]
```

```
COMPUTE-PREFIX-FUNCTION(P)
m ← length[P]
Π [1] ← 0
k ← 0
for q ← 2 to m
    do while k > 0 and P[k + 1 ≠ P[q]
           do k ← Π[k]
```

```
7         if $P[k + 1] = P[q]$
8            then $k \leftarrow k + 1$
9         $\Pi[q] \leftarrow k$
10 return $\Pi$
```

We begin with an analysis of the running times of these procedures. Proving these procedures correct will be more complicated.

Running-time analysis

The running time of COMPUTE-PREFIX-FUNCTION is $O(m)$, using an amortized analysis (see Chapter 18). We associate a potential of k with the current state k of the algorithm. This potential has an initial value of 0, by line 3. Line 6 decreases k whenever it is executed, since $\Pi[k] < k$. Since $\Pi[k] \geqq 0$ for all k, however, k can never become negative. The only other line that affects k is line 8, which increases k by at most one during each execution of the **for** loop body. Since $k < q$ upon entering the **for** loop, and since q is incremented in each iteration of the **for** loop body, $k < q$ always holds. (This justifies the claim that $\Pi[q] < q$ as well, by line 9.) We can pay for each execution of the **while** loop body on line 6 with the corresponding decrease in the potential function, since $\Pi[k] < k$. Line 8 increases the potential function by at most one, so that the amortized cost of the loop body on lines 5-9 is $O(1)$. Since the number of outer-loop iterations is $O(m)$, and since the final potential function is at least as great as the initial potential function, the total actual worst-case running time of COMPUTE-PREFIX-FUNCTION is $O(m)$.

The Knuth-Morris-Pratt algorithm runs in time $O(m + n)$. The call of COMPUTE-PREFIX-FUNCTION takes $O(m)$ time as we have just seen, and a similar amortized analysis, using the value of q as the potential function, shows that the remainder of KMP-MATCHER takes $O(n)$ time.

Compared to F1NITE-AUTOMATON-MATCHER, by using Π rather than δ, we have reduced the time for preprocessing the pattern from $O(m\ |\Sigma|)$ to $O(m)$, while keeping the actual matching time bounded by $O(m + n)$.

Correctness of the prefix-function computation

We start with an essential lemma showing that by iterating the prefix function Π, we can enumerate all the prefixes P_k that are suffixes of a given prefix P_q. Let

$$\Pi^*[q] = \{q, \Pi[q], \Pi^2[q], \Pi^3[q], \ldots, \Pi^t[q]\} ,$$

where $\Pi^i[q]$ is defined in terms of functional composition, so that $\Pi^0[q] = q$ and $\Pi^{i+1}[q] = \Pi[\Pi^i[q]]$ for $i > 1$, and where it is understood that the sequence in $\Pi^*[q]$ stops when $\Pi^t[q] = 0$ is reached.

Lemma 34.5

Let P be a pattern of length m with prefix function Π. Then, for $q = 1, 2, \ldots, m$, we have $\pi^*[q] = \{k : P_k \sqsupset P_q\}$.

Proof We first prove that

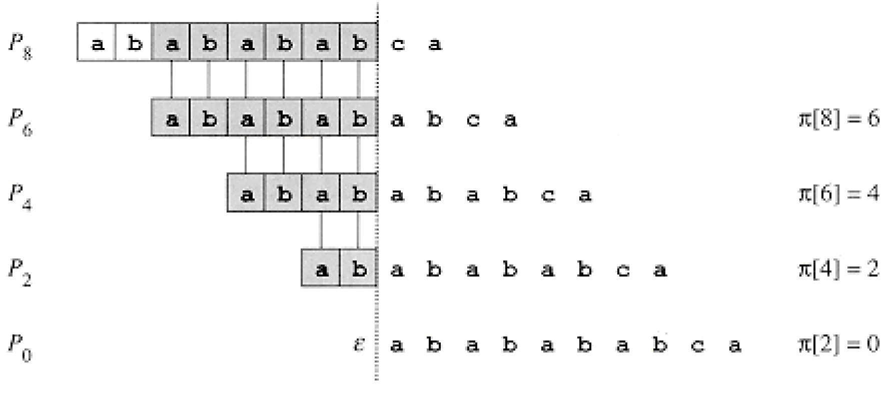
$$i \in \Pi^*[q] \text{ implies } P_i \sqsupset P_q .$$

(34.6)

If $i \in \pi^*[q]$, then $i = \pi^u[q]$ for some u. We prove equation (34.6) by induction on u. For $u = 0$, we have $i = q$, and the claim follows since $P_q \sqsupset P_q$. Using the relation $P_{\pi[i]} \sqsupset P_i$ and the transitivity of $\sqsupset$ establishes the claim for all i in $\pi^*[q]$. Therefore, $\pi^*[q] \subseteq \{k : P_k \sqsupset P_q\}$.

i	1	2	3	4	5	6	7	8	9	10
$P[i]$	a	b	a	b	a	b	a	b	c	a
$\pi[i]$	0	0	1	2	3	4	5	6	0	1

(a)


P_8 a b a b a b a b c a
P_6 a b a b a b a b c a
P_4 a b a b a b a b c a
P_2 a b a b a b a b c a
P_0 ε a b a b a b a b c a
$\pi[8] = 6$
$\pi[6] = 4$
$\pi[4] = 2$
$\pi[2] = 0$

(b)

Figure 34.10 An illustration of Lemma 34.5 for the pattern P = ababababca and q = 8. (a) The π function for the given pattern. Since π[8] = 6, π[6] = 4, π[4] = 2, and π[2] = 0, by iterating π we obtain π*[8] = {8, 6, 4, 2, 0}. (b) We slide the template containing the pattern P to the right and note when some prefix P_k of P matches up with some proper suffix of P_8; this happens for k = 6, 4, 2, and 0. In the figure, the first row gives P, and the dotted vertical line is drawn just after P_8 Successive rows show all the shifts of P that cause some prefix P_k of P to match some suffix of P_8. Successfully matched characters are shown shaded. Vertical lines connect aligned matching characters. Thus, $\{k : P_k \sqsupset P_q\} = \{8, 6, 4, 2, 0\}$. The lemma claims that $\pi^*[q] = \{k : P_k \sqsupset P_q\}$ for all q.

We prove that $\{k : P_k \sqsupset P_q\} \subseteq \pi^*[q]$ by contradiction. Suppose to the contrary that there is an integer in the set $\{k : P_k \sqsupset P_q\} - \pi^*[q]$, and let j be the largest such value. Because q is in $\{k : P_k \sqsupset P_q\} \cap \pi^*[q]$, we have $j < q$, and so we let j' denote the smallest integer in $\pi^*[q]$ that is greater than j. (We can choose $j' = q$ if there is no other number in $\pi^*[q]$ that is greater than j.) We have $P_j \sqsupset P_q$ because $j \in \{k : P_k \sqsupset P_q\}$, $P_{j'} \sqsupset P_q$ because $j' \in \pi^*[q]$; thus, $P_j \sqsupset P_{j'}$ by Lemma 34.1. Moreover, j is the largest such value with this property. Therefore, we must have $\pi[j'] = j$ and thus $j \in \pi^*[q]$. This contradiction proves the lemma.

Figure 34.10 illustrates this lemma.

The algorithm COMPUTE-PREFIX-FUNCTION computes $\pi[q]$ in order for $q = 1, 2, \ldots, m$. The computation of $\pi[1] = 0$ in line 2 of COMPUTE-PREFIX-FUNCTION is certainly correct, since $\pi[q] < q$ for all q. The following lemma and its corollary will be used to prove that COMPUTE-PREFIX-FUNCTION computes $\pi[q]$ correctly for $q > 1$.

Lemma 34.6

Let *P* be a pattern of length *m*, and let Π *be the prefix function for* P. *For* q = *1, 2, . . . ,* m*, if* Π [*q*] > 0, then Π [q*] - 1* ∈ Π *[q - *1].*

Proof If *k* = Π [*q*] > 0, then $P_k \sqsupset P_q$, and thus $P_{k-1} \sqsupset P_{q-1}$ (by dropping the last character from P_k and P_q). By Lemma 34.5, therefore, *k* - 1∈ Π *[*q* - 1].

For *q* = 2, 3, . . . , *m*, define the subset E_q-1 ⊆ Π *[*q* - 1] by

$$E_q\text{-}1 = \{k : k \in \Pi^*[q - 1] \text{ and } P[k + 1] = P[q]\}\ .$$

The set E_q-1 consists of the values *k* for which $P_k \sqsupset P_{q-1}$ (by Lemma 34.5); because *P*[*k* + 1] = *P*[*q*], it is also the case that for these values of *k*, $P_{k+1} \sqsupset P_q$. Intuitively, E_q - 1 consists of those values *k* ∈ Π *[*q* - 1] such that we can extend P_k to P_k+1 and get a suffix of P_q.

Corollary 34.7

Let *P* be a pattern of length *m*, and let Π be the prefix function for *P*. For *q* = 2, 3, . . . , *m*,

$$\pi[q] = \begin{cases} 0 & \text{if } E_{q-1} = \emptyset\ , \\ 1 + \max\{k \in E_{q-1}\} & \text{if } E_{q-1} \neq \emptyset\ . \end{cases}$$

Proof If *r* = Π [*q*], then $P_r \sqsupset P_q$, and so *r* ≥ 1 implies *P*[*r*] = *P*[*q*]. By Lemma 34.6, therefore, if *r* ≥ 1, then

$$r = 1 + \max\ \{k \in \Pi^*[q - 1]: P[k + 1] = P[q]\}\ .$$

But the set maximized over is just E_q-1, so that *r* = 1 + max {*k* ∈ E_q-1} and E_q-1 is nonempty. If *r* = 0, there is no *k* ∈ Π *[*q* - 1] for which we can extend P_k to P_{k+1} and get a suffix of P_q, since then we would have Π [*q*] > 0. Thus, $E_{q-1} = \emptyset$.

We now finish the proof that COMPUTE-PREFIX-FUNCTION computes Π correctly. In the procedure COMPUTE-PREFIX-FUNCTION, at the start of each iteration of the **for** loop of lines 4-9, we have that *k* = Π [*q* - 1]. This condition is enforced by lines 2 and 3 when the loop is first entered, and it remains true in each successive iteration because of line 9. Lines 5-8 adjust k so that it now becomes the correct value of Π [*q*]. The loop on lines 5-6 searches through all values *k* ∈ Π *[*q* - 1] until one is found for which *P*[*k* + 1] = *P*[*q*]; at that point, we have that *k* is the largest value in the set E_q-1, so that, by Corollary 34.7, we can set Π [*q*] to *k* + 1. If no such *k* is found, *k* = 0 in lines 7-9, and Π [*q*] is set to 0. This completes our proof of the correctness of COMPUTE-PREFIX-FUNCTION.

Correctness of the KMP algorithm

The procedure KMP-MATCHER can be viewed as a reimplementation of the procedure FIN1TE-AUTOMATON-MATCHER. Specifically, we shall prove that the code on lines 6-9 of KMP-MATCHER is equivalent to line 4 of FINITE-AUTOMATON-MATCHER, which sets *q* to δ(*q*,*T*[*i*]). Instead of using a stored value of δ(*q*, *T*[*i*]), however, this value is recomputed as necessary from Π . Once we have argued that KMP-MATCHER simulates the behavior of FINITE-AUTOMATON-MATCHER, the correctness of KMP-MATCHER follows from the correctness of FINITE-AUTOMATON-MATCHER (though we shall see in a moment why line 12 in KMP-MATCHER is necessary).

The correctness of KMP-MATCHER follows from the claim that either $\delta(q, T[i]) = 0$ or else $\delta(q, T[i]) - 1 \in \Pi^*[q]$. To check this claim, let $k = \delta(q, T[i])$. Then, $P_k \sqsupset P_q T[i]$ by the definitions of δ and Σ. Therefore, either $k = 0$ or else $k \geq 1$ and $P_{k-1} \sqsupset P_q$ by dropping the last character from both P_k and $P_q T[i]$ (in which case $k - 1 \in \Pi^*[q]$). Therefore, either $k = 0$ or $k - 1 \in \Pi^*[q]$, proving the claim.

The claim is used as follows. Let q' denote the value of q when line 6 is entered. We use the equivalence $\pi^*[q] = \{k : P_k \sqsupset P_q\}$ to justify the iteration $q \leftarrow \Pi[q]$ that enumerates the elements of $\{k : P_k \sqsupset P_{q'}\}$. Lines 6-9 determine $\delta(q', T[i])$ by examining the elements of $\Pi^*[q']$ in decreasing order. The code uses the claim to begin with $q = \Phi(T_i - 1) = \Sigma(T_i - 1)$ and perform the iteration $q \leftarrow \Pi[q]$ until a q is found such that $q = 0$ or $P[q + 1] = T[i]$. In the former case, $\delta(q', T[i]) = 0$; in the latter case, q is the maximum element in E_q', so that $\delta(q', T[i]) = q + 1$ by Corollary 34.7.

Line 12 is necessary in KMP-MATCHER to avoid a possible reference to $P[m + 1]$ on line 6 after an occurrence of P has been found. (The argument that $q = \Sigma(T_i - 1)$ upon the next execution of line 6 remains valid by the hint given in Exercise 34.4-6: $\delta(m, a) = \delta(\Pi[m], a)$ or, equivalently, $\Sigma(Pa) = \Sigma(P_{\Pi[m]}a)$ for any $a \in \Sigma$.) The remaining argument for the correctness of the Knuth-Morris-Pratt algorithm follows from the correctness of FINITE-AUTOMATON-MATCHER, since we now see that KMP-MATCHER simulates the behavior of FINITE-AUTOMATON-MATCHER.

Exercises

34.4-1

Compute the prefix function Π for the pattern ababbabbababbababbabb when the alphabet is $\Sigma = \{$a, b$\}$.

34.4-2

Give an upper bound on the size of $\Pi^*[q]$ as a function of q. Give an example to show that your bound is tight.

34.4-3

Explain how to determine the occurrences of pattern P in the text T by examining the Π function for the string PT (the string of length $m + n$ that is the concatenation of P and T).

34.4-4

Show how to improve KMP-MATCHER by replacing the occurrence of Π in line 7 (but not line 12) by Π', where Π' is defined recursively for $q = 1, 2, \ldots, m$ by the equation

$$\pi'[q] = \begin{cases} 0 & \text{if } \pi[q] = 0 , \\ \pi'[\pi[q]] & \text{if } \pi[q] \neq 0 \text{ and } P[\pi[q] + 1] = P[q + 1] , \\ \pi[q] & \text{if } \pi[q] \neq 0 \text{ and } P[\pi[q] + 1] \neq P[q + 1] . \end{cases}$$

Explain why the modified algorithm is correct, and explain in what sense this modification constitutes an improvement.

34.4-5

Give a linear-time algorithm to determine if a text T is a cyclic rotation of another string T'. For example, arc and car are cyclic rotations of each other.

34.4-6

Give an efficient algorithm for computing the transition function δ for the string-matching automaton corresponding to a given pattern *P*. Your algorithm should run in time *O*(*m*|Σ|). (*Hint*: Prove that δ(q, a*) =* δ*(*π *[*q*],* a *) if* q = m *or* P*[*q + *1]* ≠ a*.)*

* 34.5 The Boyer-Moore algorithm

If the pattern *P* is relatively long and the alphabet Σ is reasonably large, then an algorithm due to Robert S. Boyer and J. Strother Moore is likely to be the most efficient string-matching algorithm.

```
BOYER-MOORE-MATCHER(T, P, Σ)

1 n ← length[T]

2 m ← length[P]

3 λ ← COMPUTE-LAST-OCCURRENCE-FUNCTION(P, m, Σ)

4 γ ← COMPUTE-GOOD-SUFFIX-FUNCTION(P, m)

5 s ← 0

6 while s ≤ n - m

7      do j ← m

8         while j > 0 and P[j] = T[s + j]

9             do j ← j - 1

10        if j = 0

11           then print "Pattern occurs at shift" s

12                s ← s + γ[0]

13           else s ← s + max(γ[j], j - λ[T[s + j]])
```

Aside from the mysterious-looking λ's and γ's, this program looks remarkably like the naive string-matching algorithm. Indeed, suppose we comment out lines 3-4 and replace the updating of *s* on lines 12-13 with simple incrementations as follows:

```
12                s ← s + 1

13           else s ← s + 1
```

The modified program now acts exactly like the naive string matcher: the **while** loop beginning on line 6 considers each of the *n* - *m* + 1 possible shifts *s* in turn, and the **while** loop beginning on line 8 tests the condition *P*[1 . . *m*] = *T*[*s* + 1 . . *s* + *m*] by comparing *P*[*j*] with *T*[*s* + *j*] for *j* = *m*, *m* - 1, . . . , 1. If the loop terminates with *j* = 0, a valid shift *s* has been found, and line 11 prints out the value of *s*. At this level, the only remarkable features of the Boyer-Moore algorithm are that it compares the pattern against the text *from right to left* and that it increases the shift *s* on lines 12-13 by a value that is not necessarily 1.

The Boyer-Moore algorithm incorporates two heuristics that allow it to avoid much of the work that our previous string-matching algorithms performed. These heuristics are so effective that they often allow the algorithm to skip altogether the examination of many text characters. These heuristics, known as the "bad-character heuristic" and the "good-suffix heuristic," are illustrated in Figure 34.11. They can be viewed as operating independently in parallel. When a mismatch occurs, each heuristic proposes an amount by which *s* can safely be increased without missing a valid shift. The Boyer-Moore algorithm chooses the larger amount and increases *s* by that amount: when line 13 is reached after a mismatch, the bad-character heuristic proposes increasing *s* by $j - \lambda[T[s + j]]$, and the good-suffix heuristic proposes increasing *s* by $\gamma[j]$.

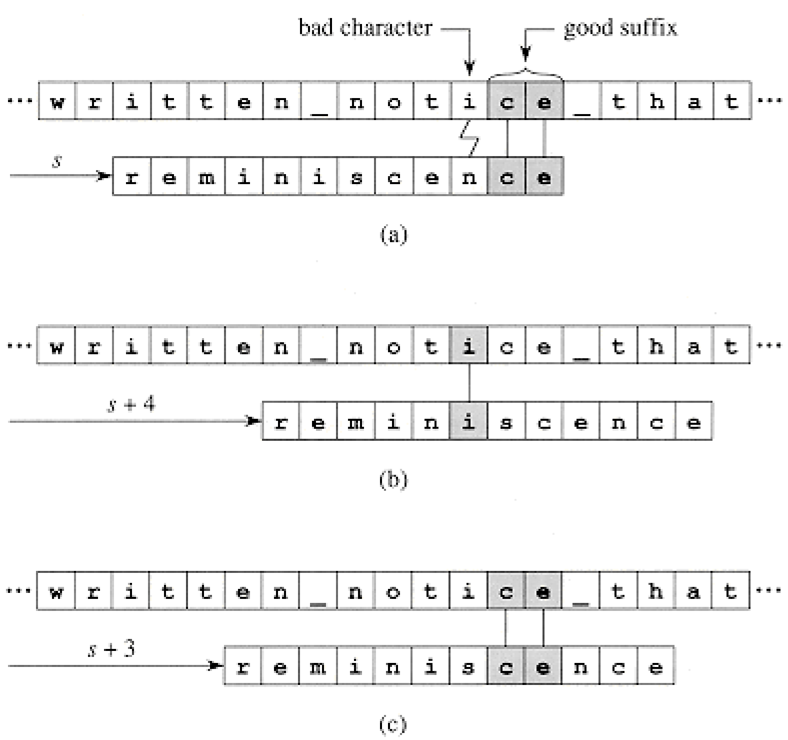

bad character
good suffix
... w r i t t e n _ n o t i c e _ t h a t ...
s
r e m i n i s c e n c e
(a)
... w r i t t e n _ n o t i c e _ t h a t ...
s + 4
r e m i n i s c e n c e
(b)
... w r i t t e n _ n o t i c e _ t h a t ...
s + 3
r e m i n i s c e n c e
(c)

Figure 34.11 An illustration of the Boyer-Moore heuristics. (a) Matching the pattern `reminiscence` against a text by comparing characters in a right-to-left manner. The shift s is invalid; although a "good suffix" ce of the pattern matched correctly against the corresponding characters in the text (matching characters are shown shaded), the "bad character" `i`, which didn't match the corresponding character `n` in the pattern, was discovered in the text. (b) The bad-character heuristic proposes moving the pattern to the right, if possible, by the amount that guarantees that the bad text character will match the rightmost occurrence of the bad character in the pattern. In this example, moving the pattern 4 positions to the right causes the bad text character `i` in the text to match the rightmost `i` in the pattern, at position 6. If the bad character doesn't occur in the pattern, then the pattern may be moved completely past the bad character in the text. If the rightmost occurrence of the bad character in the pattern is to the right of the current bad character position, then this heuristic makes no proposal. (c) With the good-suffix heuristic, the pattern is moved to the right by the least amount that guarantees that any pattern characters that align with the good suffix ce previously found in the text will match those suffix characters. In this example, moving the pattern 3 positions to the right satisfies this condition. Since the good-suffix heuristic proposes a movement of 3 positions, which is smaller than the 4-position proposal of the bad-character heuristic, the Boyer-Moore algorithm increases the shift by 4.

The bad-character heuristic

When a mismatch occurs, the bad-character heuristic uses information about where the bad text character $T[s + j]$ occurs in the pattern (if it occurs at all) to propose a new shift. In the best case, the mismatch occurs on the first comparison ($P[m] \neq T[s + m]$) and the bad character $T[s + m]$ does not occur in the pattern at all. (Imagine searching for a^m in the text string b^n.) In this case, we can increase the shift *s* by *m*, since any shift smaller than $s + m$ will align some pattern character against the bad character, causing a mismatch. If the best case occurs repeatedly, the Boyer-Moore algorithm examines only a fraction $1/m$ of the text characters, since each text character examined yields a mismatch, thus causing *s* to increase by *m*. This best-case behavior illustrates the power of matching right-to-left instead of left-to-right.

In general, the ***bad-character heuristic*** works as follows. Suppose we have just found a mismatch: $P[j] \neq T[s + j]$ for some *j*, where $1 \le j \le m$. We then let *k* be the largest index in the range $1 \le k \le m$ such that $T[s + j] = P[k]$, if any such *k* exists. Otherwise, we let $k = 0$. We claim that we may safely increase *s* by $j - k$. We must consider three cases to prove this claim, as illustrated by Figure 34.12.

- $k = 0$: As shown in Figure 34.12(a), the bad character $T[s + j]$ didn't occur in the pattern at all, and so we can safely increase *s* by *j* without missing any valid shifts.

- $k < j$: As shown in Figure 34.12(b), the rightmost occurrence of the bad character is in the pattern to the left of position *j*, so that $j - k > 0$ and the pattern must be moved $j - k$ characters to the right before the bad text character matches any pattern character. Therefore, we can safely increase *s* by $j - k$ without missing any valid shifts.

- $k > j$: As shown in Figure 34.12(c), $j - k < 0$, and so the bad-character heuristic is essentially proposing to decrease *s*. This recommendation will be ignored by the Boyer-Moore algorithm, since the good-suffix heuristic will propose a shift to the right in all cases.

The following simple program defines $\lambda[a]$ to be the index of the rightmost position in the pattern at which character *a* occurs, for each $a \in \Sigma$. If *a* does not occur in the pattern, then $\lambda[a]$ is set to 0. We call λ the ***last-occurrence function*** for the pattern. With this definition, the expression $j - \lambda[T[s + j]]$ on line 13 of BOYER-MOORE MATCHER implements the bad-character heuristic. (Since $j - \lambda[T[s + j]]$ is negative if the rightmost occurrence of the bad character $T[s + j]$ in the pattern is to the right of position *j*, we rely on the positivity of $\gamma[j]$, proposed by the good-suffix heuristic, to ensure that the algorithm makes progress at each step.)

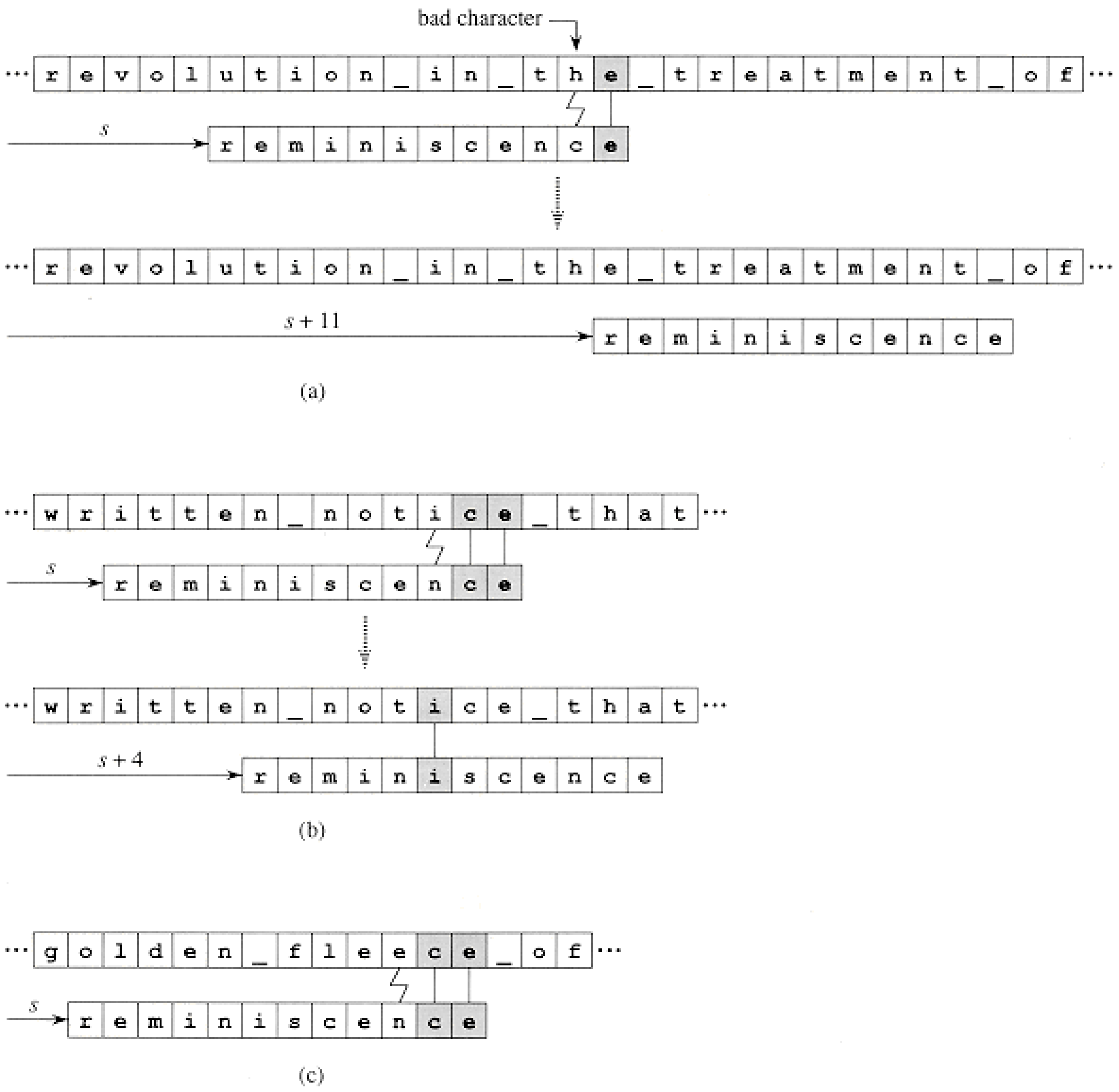

bad character
r e v o l u t i o n _ i n _ t h e _ t r e a t m e n t _ o f
s
r e m i n i s c e n c e
s + 11
(a)
w r i t t e n _ n o t i c e _ t h a t
s + 4
(b)
g o l d e n _ f l e e c e _ o f
(c)

Figure 34.12 The cases of the bad-character heuristic. (a) The bad character `h` occurs nowhere in the pattern, and so the pattern can be advanced j = 11 characters until it has passed over the bad character. (b) The rightmost occurrence of the bad character in the pattern is at position k < j, and so the pattern can be advanced j - k characters. Since j = 10 and k = 6 for the bad character `i`, the pattern can be advanced 4 positions until the `i`'s line up. (c) The rightmost occurrence of the bad character in the pattern is at position k > j. In this example, j = 10 and k = 12 for the bad character `e`. The bad-character heuristic proposes a negative shift, which is ignored.

```
COMPUTE-LAST-OCCURRENCE-FUNCTION(P, m, Σ)

for each character a ∈ Σ

     do λ[a] = 0

for j ←1 to m

     do λ[P[j]] ← j

return λ
```

The running time of procedure COMPUTE-LAST-OCCURRENCE-FUNCTION is $O(|\Sigma| + m)$.

The good-suffix heuristic

Let us define the relation *Q* ~ *R* (read "*Q* is similar to *R"*) for strings *Q* and *R* to mean that $Q \sqsupset R$ or $R \sqsupset Q$. If two strings are similar, then we can align them with their rightmost characters matched, and no pair of aligned characters will disagree. The relation "~" is symmetric: *Q* ~ *R* if and only if *R* ~ *Q*. We also have, as a consequence of Lemma 34.1, that

$$Q \sqsupset R \text{ and } S \sqsupset R \text{ imply } Q \sim S . \qquad (34.7)$$

(34.7)

If we find that P[*j*] ≠ *T*[*s* + *j*], where j < *m*, then the ***good-suffix heuristic*** says that we can safely advance *s* by

ϒ [j] = m - max {k: 0 ≤ k < m and P[j + 1 . . m] ~ P_k} .

That is, ϒ[*j*] is the least amount we can advance *s* and not cause any characters in the "good suffix" *T*[*s* + j + 1 . . s + *m*] to be mismatched against the new alignment of the pattern. The function ϒ is well defined for all *j*, since *P*[*j* + 1 . . *m*] ~ P_0 for all *j*: the empty string is similar to all strings. We call ϒ the **good-suffix function** for the pattern *P*.

We now show how to compute the good-suffix function ϒ. We first observe that ϒ[*j*] ≤ *m* - Π [*m*] for all *j*, as follows. If *w* = Π [*m*], then $P_w \sqsupset P$ by the definition of Π. Furthermore, since $P[j+1\,..\,m] \sqsupset P$ for any *j*, we have P_w ~ *P*[*j* + 1 . . *m*], by equation (34.7). Therefore, ϒ[*j*] ≤ *m* - Π *[*m*] for all* j.

We can now rewrite our definition of ϒ as

ϒ[j] = m - max {k : Π[m] ≤ k < m and P[j + 1 . . m] ~ P_k} .

The condition that *P*[*j* + 1 . . *m*] ~ P_k holds if either $P[j+1\,..\,m] \sqsupset P_k$ or $P_k \sqsupset P[j+1\,..\,m]$. But the latter possibility implies that $P_k \sqsupset P$ and thus that *k* ≤ Π [*m*], by the definition of Π. This latter possibility cannot reduce the value of ϒ[*j*] below *m* - Π [*m*]. We can therefore rewrite our definition of ϒ still further as follows:

$$\gamma[j] = m - \max(\{\pi[m]\} \cup \{k : \pi[m] < k < m \text{ and } P[j+1\,..\,m] \sqsupset P_k\}) .$$

(The second set may be empty.) It is worth observing that the definition implies that ϒ[*j*] > 0 for all *j* = 1, 2, . . . , *m*, which ensures that the Boyer-Moore algorithm makes progress.

To simplify the expression for ϒ further, we define *P*' as the reverse of the pattern *P* and Π' as the corresponding prefix function. That is *P'*[*i*] = *P*[*m* - *i* + 1] for *i* = 1, 2, . . . , *m*, and Π'[*t*] is the largest *u* such that *u* < *t* and $P'_u \sqsupset P'_t$.

If *k* is the largest possible value such that $P[j+1\,..\,m] \sqsupset P_k$, then we claim that

Π'[l] = m - j ,

(34.8)

where $l = (m - k) + (m - j)$. To see that this claim is well defined, note that $P[j+1\,..\,m] \sqsupset P_k$ implies that $m - j \le k$, and thus $l \le m$. Also, $j < m$ and $k \le m$, so that $l \ge 1$. We prove this claim as follows. Since $P[j+1\,..\,m] \sqsupset P_k$, we have $P'_{m-j} \sqsupset P'_l$. Therefore, $\Pi'[l] \ge m - j$. Suppose now that $p > m - j$, where $p = \Pi'[l]$. Then, by the definition of Π', we have $P'_p \sqsupset P'_l$ or, equivalently, $P'[1\,..\,p] = P'[l - p + 1\,..\,l]$. Rewriting this equation in terms of P rather than P', we have $P[m - p + 1\,..\,m] = P[m - l + 1\,..\,m - l + p]$. Substituting for $l = 2m - k - j$, we obtain $P[m - p + 1\,..\,m] = P[k - m + j + 1\,..\,k - m + j + p]$, which implies $P[m-p+1\,..\,m] \sqsupset P_{k-m+j+p}$. Since $p > m - j$, we have $j + 1 > m - p + 1$, and so $P[j+1\,..\,m] \sqsupset P[m-p+1\,..\,m]$, implying that $P[j+1\,..\,m] \sqsupset P_{k-m+j+p}$ by the transitivity of $\sqsupset$. Finally, since $p > m - j$, we have $k' > k$, where $k' = k - m + j + p$, contradicting our choice of k as the largest possible value such that $P[j+1\,..\,m] \sqsupset P_k$. This contradiction means that we can't have $p > m - j$, and hus $= m - j$, which proves the claim (34.8).

Using equation (34.8), and noting that $\Pi'[l] = m - j$ implies that $j = m - \Pi'[l]$ and $k = m - l + \Pi'[l]$, we can rewrite our definition of γ still further:

$$\gamma[j] = m - \max(\{\Pi[m]\}$$

$$\cup\ \{m - l + \Pi'[l]: 1 \le l \le m \text{ and } j = m - \Pi'[l]\})$$

$$= \min(\{m - \Pi[m]\}$$

$$\cup\ \{l - \Pi'[l]: 1 \le l \le m \text{ and } j = m - \Pi'[1]\})\ .$$

(34.9)

Again, the second set may be empty.

We are now ready to examine the procedure for computing γ.

```
COMPUTE-GOOD-SUFFIX-FUNCTION(P, m)
Π ← COMPUTE-PREFIX-FUNCTION(P)
P' ← reverse(P)
Π' ← COMPUTE-PREFIX-FUNCTION(P')
for j ← 0 to m
    do γ[j] ← m - Π[m]
for l ← 1 to m
    do j ← m - Π'[1]
       if γ[j] > l - Π'[l]
          then γ[j] ← l - Π'[l]
return γ
```

The procedure COMPUTE-GOOD-SUFFIX-FUNCTION is a straightforward implementation of equation (34.9). Its running time is $O(m)$.

The worst-case running time of the Boyer-Moore algorithm is clearly $O((n - m + 1)m + |\Sigma|)$, since COMPUTE-LAST-OCCURRENCE-FUNCTION takes time $O(m + |\Sigma|)$, COMPUTE-GOOD-SUFFIX-FUNCTION takes time $O(m)$, and the Boyer-Moore algorithm (like the Rabin-Karp algorithm) spends $O(m)$ time validating each valid shift s. In practice, however, the Boyer-Moore algorithm is often the algorithm of choice.

Exercises

34.5-1

Compute the λ and γ functions for the pattern P = `0101101201` and the alphabet $\Sigma = \{0, 1, 2\}$.

34.5-2

Give examples to show that by combining the bad-character and good-suffix heuristics, the Boyer-Moore algorithm can perform much better than if it used just the good-suffix heuristic.

34.5-3

An improvement to the basic Boyer-Moore procedure that is often used in practice is to replace the γ function by γ', defined by

$$\gamma'[j] = m - \max\{k : 0 \le k < m \text{ and } P[j + 1 \,..\, m] \sim P_k \text{ and}$$

$$(k - m + j > 0 \text{ implies } P[j] \ne P[k - m + j])\} .$$

In addition to ensuring that the characters in the good suffix will be mis-matched at the new shift, the γ' function also guarantees that the same pattern character will not be matched up against the bad text character. Show how to compute the γ' function efficiently.

Problems

34-1 String matching based on repetition factors

Let y^i denote the concatenation of string y with itself i times. For example, $(\texttt{ab})^3 = \texttt{ababab}$. We say that a string $x \in \Sigma^*$ has ***repetition factor*** r if $x = y^r$ for some string $y \in \Sigma^*$ and some $r > 0$. Let $p(x)$ denote the largest r such that x has repetition factor r.

a. Give an efficient algorithm that takes as input a pattern $P[1 \,..\, m]$ and computes $\rho(P_i)$ for $i = 1, 2, \ldots, m$. What is the running time of your algorithm?

b. For any pattern $P[1 \,..\, m]$, let $p^*(P)$ be defined as $\max_{1 \le i \le m} \rho(P_i)$. Prove that if the pattern P is chosen randomly from the set of all binary strings of length m, then the expected value of $\rho^*(P)$ *is* $O(1)$.

c. Argue that the following string-matching algorithm correctly finds all occurrences of pattern P in a text $T[1 \,..\, n]$ in time $O(\rho^*(P)n + m)$.

```
REPETITION-MATCHER(P,T)
m ← length[P]
n ← length[T]
k ← 1 + p*(P)
```

```
$q \leftarrow 0$
$s \leftarrow 0$
while $s \le n - m$
     do if $T[s + q + 1] = P[q + 1]$
           then $q \leftarrow q + 1$
                if $q = m$
                   then print "Pattern occurs with shift" $s$
        if $q = m$ or $T[s + q + 1] \ne P[q + 1]$
           then $s \leftarrow s + \max(1, \lceil q/k \rceil)$
                $q \leftarrow 0$
```

This algorithm is due to Galil and Seiferas. By extending these ideas greatly, they obtain a linear-time string-matching algorithm that uses only $O(1)$ storage beyond what is required for P and T.

34-2 Parallel string matching

Consider the problem of string matching on a parallel computer. Assume that for a given pattern, we have a string-matching automaton M with state set Q. Let Φ be the final-state function for M. Suppose that our input ext is $T[1 \,.\,.\, n]$. We wish to compute $\Phi(T_i)$ for $i = 1, 2, \ldots, n$; that is, we wish to compute the final state for each refix. Our strategy is to use the parallel prefix computation described in Section 30.1.2.

For any input string x, define the function $\delta_x : Q \rightarrow Q$ such that if M starts in state q and reads input x, then M ends in state $\delta_x(q)$.

a. Prove that $\delta_y \circ \delta_x = \delta_{xy}$, where $\circ$ denotes functional composition:

$$(\delta_y \circ \delta_x)(q) = \delta_y(\delta_x(q)) \ .$$

b. Argue that $\circ$ is an associative operation.

c. Argue that δ_{xy} can be computed from tabular representations of δ_x and δ_y in $O(1)$ time on a CREW PRAM. Analyze how many processors are needed in terms of $|Q|$.″

d. Prove that $\Phi\ (T_i) = \delta T_i(q_0)$, where q_0 is the start state for M.

e. Show how to find all occurrences of a pattern in a text of length n in $O(\lg n\ )$ time on a CREW PRAM. Assume that the pattern is supplied in the form of the corresponding string-matching automaton.

Chapter notes

The relation of string matching to the theory of finite automata is discussed by Aho, Hopcroft, and Ullman [4]. The Knuth-Morris-Pratt algorithm [125] was invented independently by Knuth and Pratt and by Morris; they published their work jointly. The Rabin-Karp algorithm was proposed by Rabin and Karp [117], and the Boyer-Moore algorithm is due to Boyer and Moore [32]. Galil and Seiferas [78] give an interesting deterministic linear-time string-matching algorithm that uses only $O(1)$ space beyond that required to store the pattern and text.

Go to Chapter 35 Back to Table of Contents