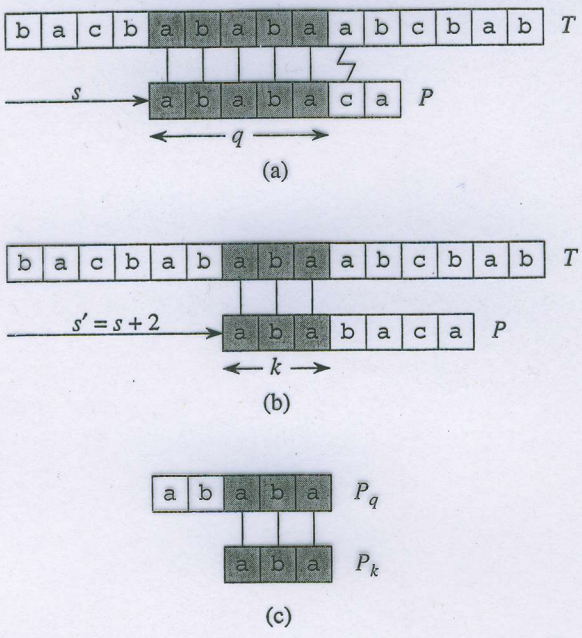
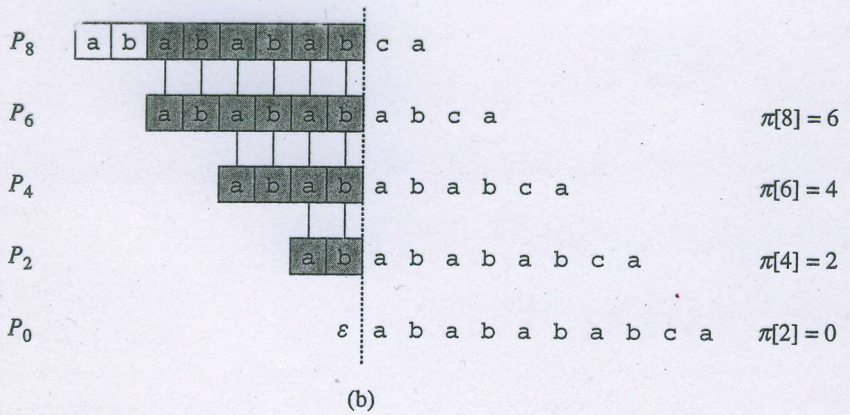


Figure 32.10 The prefix function π. **(a)** The pattern $P =$ `ababaca` is aligned with a text T so that the first $q = 5$ characters match. Matching characters, shown shaded, are connected by vertical lines. **(b)** Using only our knowledge of the 5 matched characters, we can deduce that a shift of $s + 1$ 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 proper suffix of P_5 is P_3. This information is precomputed and represented in the array π, so that $\pi[5] = 3$. Given that q characters have matched successfully at shift s, the next potentially valid shift is at $s' = s + (q - \pi[q])$.

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)



(b)

Figure 32.11 An illustration of Lemma 32.5 for the pattern $P =$ `abababaca` and $q = 8$. **(a)** The π function for the given pattern. Since $\pi[8] = 6$, $\pi[6] = 4$, $\pi[4] = 2$, and $\pi[2] = 0$, by iterating π we obtain $\pi^*[8] = \{6, 4, 2, 0\}$. **(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, $\{k : k < q \text{ and } P_k \sqsupset P_q\} = \{6, 4, 2, 0\}$. The lemma claims that $\pi^*[q] = \{k : k < q \text{ and } P_k \sqsupset P_q\}$ for all q.

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)
 1  n ← length[T]
 2  m ← length[P]
 3  π ← COMPUTE-PREFIX-FUNCTION(P)
 4  q ← 0                                  ▷ Number of characters matched.
 5  for i ← 1 to n                         ▷ Scan the text from left to right.
 6      do while q > 0 and P[q + 1] ≠ T[i]
 7            do q ← π[q]                  ▷ Next character does not match.
 8         if P[q + 1] = T[i]
 9           then q ← q + 1               ▷ Next character matches.
10         if q = m                       ▷ Is all of P matched?
11           then print "Pattern occurs with shift" i − m
12                q ← π[q]                ▷ Look for the next match.
```

```
COMPUTE-PREFIX-FUNCTION(P)
 1  m ← length[P]
 2  π[1] ← 0
 3  k ← 0
 4  for q ← 2 to m
 5      do while k > 0 and P[k + 1] ≠ P[q]
 6            do k ← π[k]
 7         if P[k + 1] = P[q]
 8           then k ← k + 1
 9         π[q] ← k
10  return π
```

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 $\Theta(m)$, using the potential method of amortized analysis (see Section 17.3). 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 $\pi[k] \geq 0$ 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