String Matching Algorithms

Naive String–Matching 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
- takes time $O((n-m+1)*m)$, which is $\theta(n^2)$ if $m = \lfloor n/2 \rfloor$

Naive String–Matching Algorithm

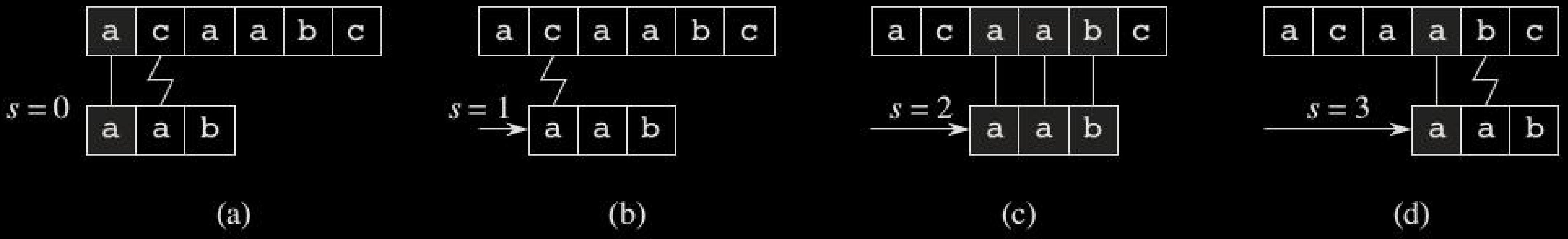


Figure 32.4 The operation of the naive string matcher for the pattern $P = \texttt{aab}$ and the text $T = \texttt{acaabc}$. We can imagine the pattern P as a template that we slide next to the text. **(a)–(d)** The four successive alignments tried by the naive string matcher. In each part, vertical lines connect corresponding regions found to match (shown shaded), and a jagged line connects the first mismatched character found, if any. The algorithm finds one occurrence of the pattern, at shift $s = 2$, shown in part (c).

Naive String–Matching Algorithm

```
NAIVE-STRING-MATCHER(T, P)
n = T.length
m = P.length
for s = 0 to n - m
    if P[1..m] == T[s + 1..s + m]
        print "Pattern occurs with shift" s
```

Naive String–Matching Algorithm

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	1	0	0	0	1	0	1	0	0	0	1

1	2	3	4
0	0	0	1

Naive String–Matching Algorithm

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	1	0	0	0	1	0	1	0	0	0	1

1	2	3	4
0	0	0	1

Naive String–Matching Algorithm

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	1	0	0	0	1	0	1	0	0	0	1

1	2	3	4
0	0	0	1

Naive String–Matching Algorithm

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	1	0	0	0	1	0	1	0	0	0	1

1	2	3	4
0	0	0	1

Naive String–Matching Algorithm

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	1	0	0	0	1	0	1	0	0	0	1

1	2	3	4
0	0	0	1

Naive String–Matching Algorithm

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	1	0	0	0	1	0	1	0	0	0	1

1	2	3	4
0	0	0	1

Naive String–Matching Algorithm

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	1	0	0	0	1	0	1	0	0	0	1

1	2	3	4
0	0	0	1

Naive String–Matching Algorithm

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	1	0	0	0	1	0	1	0	0	0	1

1	2	3	4
0	0	0	1

Naive String–Matching Algorithm

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	1	0	0	0	1	0	1	0	0	0	1

1	2	3	4
0	0	0	1

Naive String–Matching Algorithm

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	1	0	0	0	1	0	1	0	0	0	1

1	2	3	4
0	0	0	1

Naive String–Matching Algorithm

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	1	0	0	0	1	0	1	0	0	0	1

1	2	3	4
0	0	0	1

Naive String–Matching Algorithm

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	1	0	0	0	1	0	1	0	0	0	1

1	2	3	4
0	0	0	1

Naive String–Matching Algorithm

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	1	0	0	0	1	0	1	0	0	0	1

1	2	3	4
0	0	0	1

Naive String–Matching Algorithm

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	1	0	0	0	1	0	1	0	0	0	1

1	2	3	4
0	0	0	1

Naive String–Matching Algorithm

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	1	0	0	0	1	0	1	0	0	0	1

1	2	3	4
0	0	0	1

Naive String–Matching Algorithm

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	1	0	0	0	1	0	1	0	0	0	1

1	2	3	4
0	0	0	1

Naive String–Matching Algorithm

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	1	0	0	0	1	0	1	0	0	0	1

1	2	3	4
0	0	0	1

Naive String–Matching Algorithm

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	1	0	0	0	1	0	1	0	0	0	1

1	2	3	4
0	0	0	1

Naive String–Matching Algorithm

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	1	0	0	0	1	0	1	0	0	0	1

1	2	3	4
0	0	0	1

Naive String–Matching Algorithm

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	1	0	0	0	1	0	1	0	0	0	1

1	2	3	4
0	0	0	1

Naive String–Matching Algorithm

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	1	0	0	0	1	0	1	0	0	0	1

1	2	3	4
0	0	0	1

Naive String–Matching Algorithm

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	1	0	0	0	1	0	1	0	0	0	1

1	2	3	4
0	0	0	1

Naive String–Matching Algorithm

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	1	0	0	0	1	0	1	0	0	0	1

1	2	3	4
0	0	0	1

Naive String–Matching Algorithm

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	1	0	0	0	1	0	1	0	0	0	1

1	2	3	4
0	0	0	1

Naive String–Matching Algorithm

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	1	0	0	0	1	0	1	0	0	0	1

1	2	3	4
0	0	0	1

Naive String–Matching Algorithm

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	1	0	0	0	1	0	1	0	0	0	1

1	2	3	4
0	0	0	1

Naive String–Matching Algorithm

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	1	0	0	0	1	0	1	0	0	0	1

1	2	3	4
0	0	0	1

Naive String–Matching Algorithm

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	1	0	0	0	1	0	1	0	0	0	1

1	2	3	4
0	0	0	1

Naive String–Matching Algorithm

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	1	0	0	0	1	0	1	0	0	0	1

1	2	3	4
0	0	0	1

Naive String–Matching Algorithm

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	1	0	0	0	1	0	1	0	0	0	1

1	2	3	4
0	0	0	1

Naive String–Matching Algorithm

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	1	0	0	0	1	0	1	0	0	0	1

1	2	3	4
0	0	0	1

Naive String–Matching Algorithm

```
NAIVE-STRING-MATCHER(T, P)
n = T.length
m = P.length
for s = 0 to n − m
    if P[1..m] == T[s + 1..s + m]
        print "Pattern occurs with shift" s
```

Rabin–Karp algorithm

- Performs well in practice and that also generalizes to other algorithms for related problems, such as two-dimensional pattern matching
- uses θ (m) preprocessing time, and its worst-case running time is θ ((n-m+1)*m)

Rabin–Karp algorithm

- Makes use of elementary number–theoretic notions such as the equivalence of two numbers modulo a third number
- Assume that $\Sigma = \{0, 1, 2, \ldots, 9\}$, so that each character is a decimal digit
- Can then view a string of k consecutive characters as representing a length–k decimal number

Rabin–Karp algorithm

- Character string 31415 thus corresponds to the decimal number 31,415
- Given a pattern P [1..m], let p denote its corresponding decimal value.
- In a similar manner, given a text T[1..n], let t_s denote decimal value of the length–m substring T [s+1..s + m]

Rabin–Karp algorithm

- Can compute p in time θ (m) using Horner's rule:

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

- Similarly compute t_0 from T [1..m] in time θ (m)
- t_1 is computed from [2..m+1] and so on

Rabin–Karp algorithm

- Observe that we can compute t_{s+1} from t_s in constant time

$$t_{s+1} = 10(t_s - 10^{m-1}T[s+1]) + T[s+m+1] \, .$$

- 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
- $t_{s+1} = 10(31415 - 10000 * 3) + 2 = 14152$

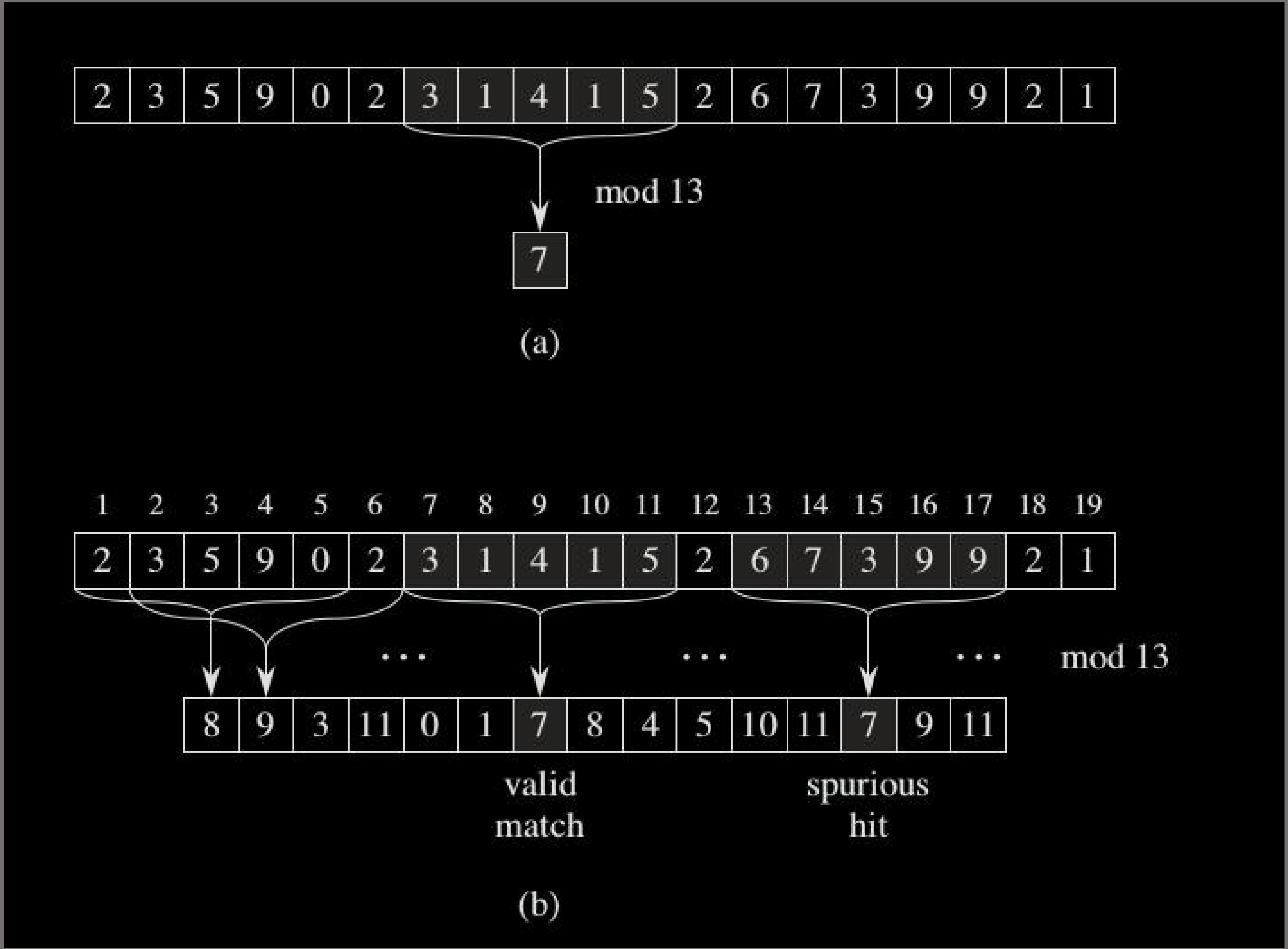
Rabin–Karp algorithm

- We can find all occurrences of the pattern P[1..m] in the text T[1..n] with $\theta(m)$ preprocessing time and $\theta(n - m + 1)$ matching time
- One problem: p and t_s may be too large to work with conveniently
- If P contains m characters, then we cannot reasonably assume that each arithmetic operation on p (which is m digits long) takes "constant time."

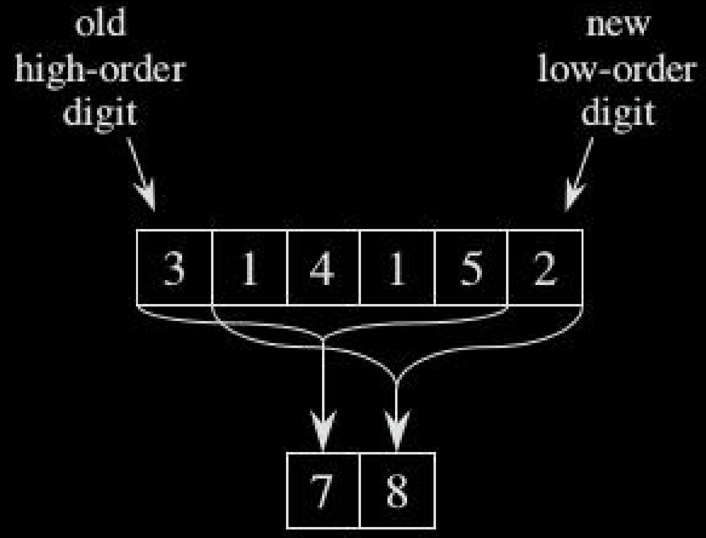
Rabin–Karp algorithm

- Fortunately, we can solve this problem easily, as Figure 32.5 shows: compute p and the t_s values modulo a suitable modulus q
- Choose modulus q as a prime such that 10q just fits within one computer word
- then we can perform all the necessary computations with single–precision arithmetic

Rabin–Karp algorithm



(a)
(b)



$$
\begin{aligned}
14152 &\equiv (31415 - \overset{\text{old high-order digit}}{3}\cdot 10000)\cdot \overset{\text{shift}}{10} + \overset{\text{new low-order digit}}{2} \pmod{13} \\
&\equiv (7 - 3\cdot 3)\cdot 10 + 2 \pmod{13} \\
&\equiv 8 \pmod{13}
\end{aligned}
$$

(c)

Figure 32.5 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, computed modulo 13, yields 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 $31415 \equiv 7 \pmod{13}$. The algorithm finds two such windows, 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)** How to compute 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. Because all computations are performed modulo 13, the value for the first window is 7, and the value for the new window is 8.

Rabin–Karp algorithm

- If q is large enough, then we hope that spurious hits occur infrequently enough that the cost of the extra checking is low.
- The inputs to the procedure are the text T , the pattern P , the radix d to use (which is typically taken to be |Σ|), and the prime q to use

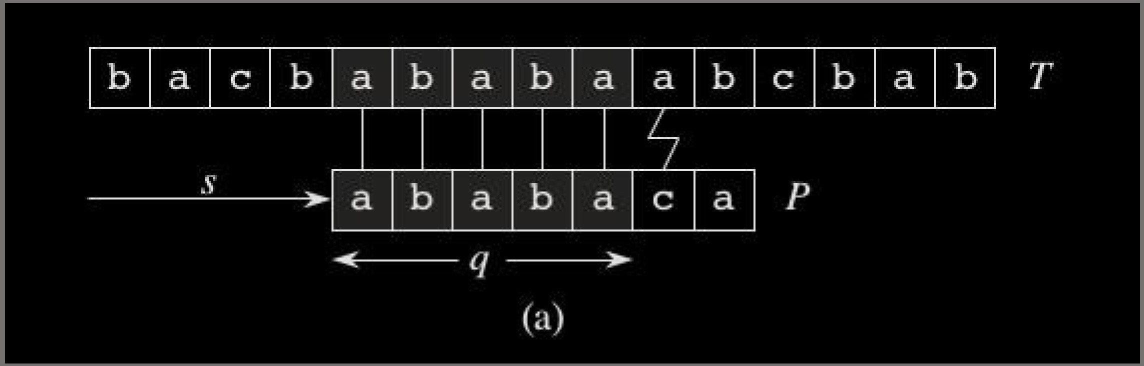
```
RABIN-KARP-MATCHER(T, P, d, q)
n = T.length
m = P.length
h = d^(m-1) mod q
p = 0
t_0 = 0
for i = 1 to m                          // preprocessing
    p = (dp + P[i]) mod q
    t_0 = (d t_0 + T[i]) mod q
for s = 0 to n - m                      // matching
    if p == t_s
        if P[1..m] == T[s+1..s+m]
            print "Pattern occurs with shift" s
    if s < n - m
        t_{s+1} = (d(t_s - T[s+1]h) + T[s+m+1]) mod q
```

The Knuth–Morris–Pratt algorithm

- Linear–time string–matching algorithm
- using just an auxiliary function π , which we precompute from the pattern in time θ (m) and store in an array π[1..m]
- prefix function π for a pattern encapsulates knowledge about how the pattern matches against shifts of itself

The Knuth–Morris–Pratt algorithm

- avoid testing useless shifts in the naive pattern–matching algorithm



(a)

- For this example, q = 5 of the characters have matched successfully, but the 6th pattern character fails to match the corresponding text character

The Knuth–Morris–Pratt algorithm

- Knowing these q text characters allows us to determine immediately that certain shifts are invalid
- In this example, shift s + 1 is necessarily invalid, since first character (a) would be aligned with a text character that we know does not match the first pattern character, but does match the second pattern character (b).

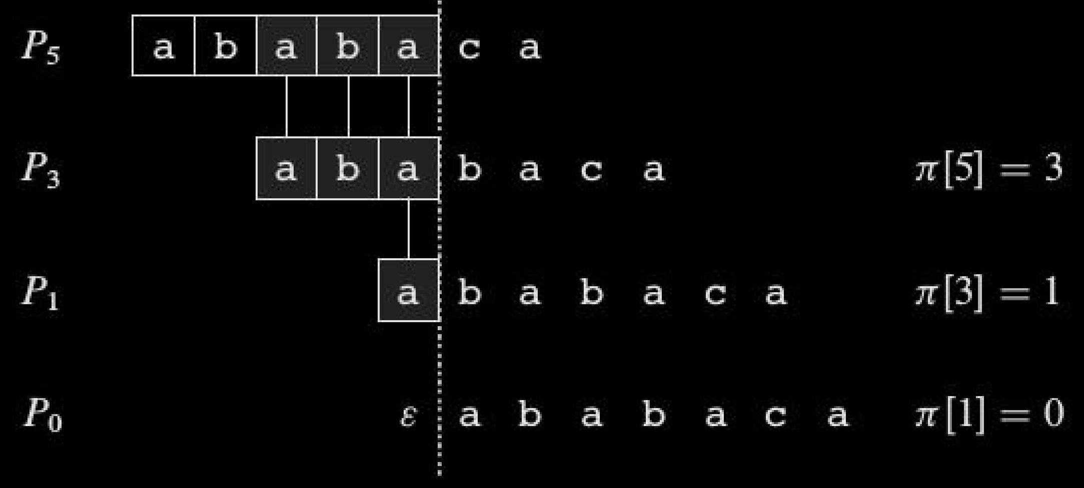
The Knuth–Morris–Pratt algorithm

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

KMP algorithm

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

(a)



(b)

Figure 32.11 An illustration of Lemma 32.5 for the pattern P = ababaca and $q = 5$. **(a)** The π function for the given pattern. Since $\pi[5] = 3$, $\pi[3] = 1$, and $\pi[1] = 0$, by iterating π we obtain $\pi^*[5] = \{3, 1, 0\}$. **(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_5; we get matches when $k = 3$, 1, and 0. In the figure, the first row gives P, and the dotted vertical line is drawn just after P_5. Successive rows show all the shifts of P that cause some prefix P_k of P to match some suffix of P_5. Successfully matched characters are shown shaded. Vertical lines connect aligned matching characters. Thus, $\{k : k < 5 \text{ and } P_k \sqsupset P_5\} = \{3, 1, 0\}$. Lemma 32.5 claims that $\pi^*[q] = \{k : k < q \text{ and } P_k \sqsupset P_q\}$ for all q.

KMP algorithm

```
COMPUTE-PREFIX-FUNCTION(P)
m = P.length
let π[1..m] be a new array
π[1] = 0
k = 0
for q = 2 to m
    while k > 0 and P[k + 1] ≠ P[q]
        k = π[k]
    if P[k + 1] == P[q]
        k = k + 1
    π[q] = k
return π
```

KMP algorithm

```
KMP-MATCHER(T, P)
n = T.length
m = P.length
π = COMPUTE-PREFIX-FUNCTION(P)
q = 0                                   // number of characters matched
for i = 1 to n                          // scan the text from left to right
    while q > 0 and P[q + 1] ≠ T[i]
        q = π[q]                        // next character does not match
    if P[q + 1] == T[i]
        q = q + 1                       // next character matches
    if q == m                           // is all of P matched?
        print "Pattern occurs with shift" i − m
        q = π[q]                        // look for the next match
```

Trace

- T – bacbababaabababacac
- P – ababaca

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

Trace

- P – ababcababcabdababecabdababe

- T – abgababcababcabdababebcedaababcababcabdababe