Chapter 32

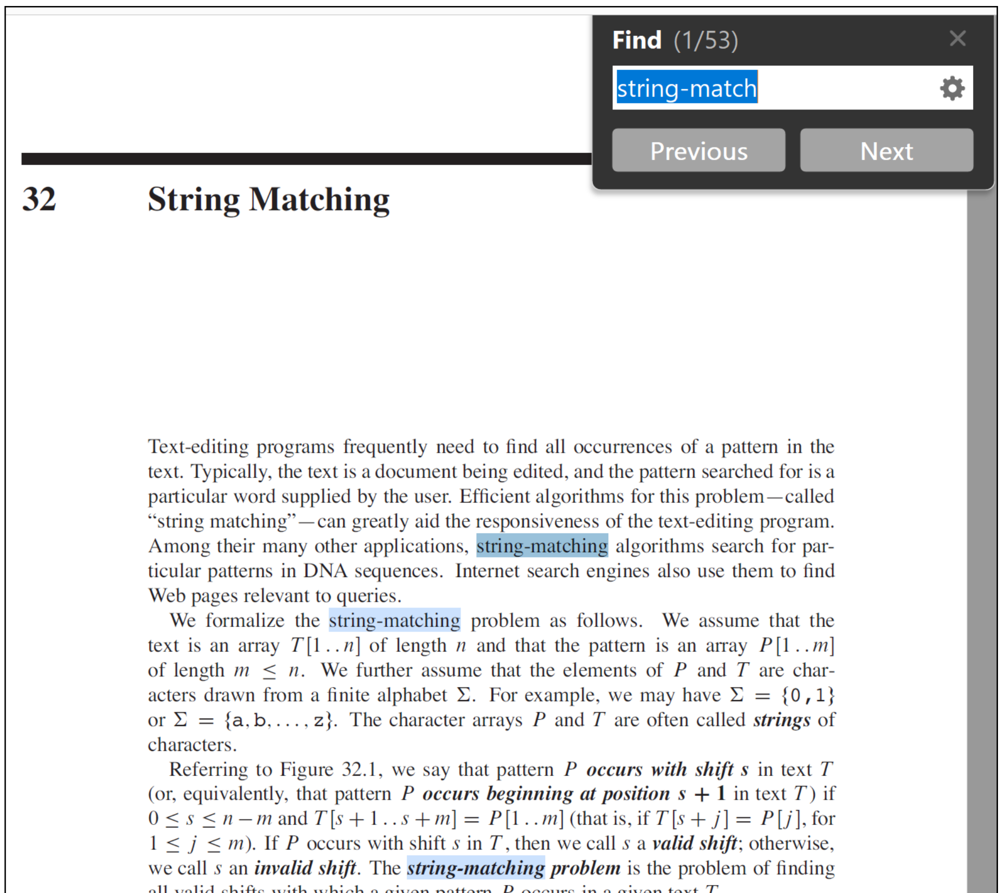
字符串匹配

VII Selected Topics

32 字符串匹配

在文本编辑程序中，查找文本中某一模式的所有出现位置是一个常见的问题。

char *strstr(char *text, char *pattern);



32 String Matching

Text-editing programs frequently need to find all occurrences of a pattern in the text. 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—called "string matching"—can greatly aid the responsiveness of the text-editing program. Among their many other applications, string-matching algorithms search for particular patterns in DNA sequences. Internet search engines also use them to find Web pages relevant to queries.

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 $m \le n$. We further assume that the elements of P and T are characters drawn from a finite alphabet Σ. For example, we may have $\Sigma = \{0,1\}$ or $\Sigma = \{a, b, \ldots, z\}$. The character arrays P and T are often called ***strings*** of characters.

Referring to Figure 32.1, 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 $0 \le s \le n - m$ and $T[s+1..s+m] = P[1..m]$ (that is, if $T[s+j] = P[j]$, for $1 \le j \le m$). 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.

32 字符串匹配

char *strstr(char *text, char *pattern);

Google 学术搜索

Fast pattern matching in strings

文章

找到约 168,000 条结果 （用时**0.08**秒）

时间不限
2021以来
2020以来
2017以来
自定义范围...

按相关性排序
按日期排序

不限语言
中文网页
简体中文网页

类型不限
评论性文章

☐ 包括专利
☑ 包含引用

创建快讯

Fast pattern matching in strings
DE Knuth, JH Morris, Jr, VR Pratt - SIAM journal on computing, 1977 - SIAM
An algorithm is presented which finds all occurrences of one given string within another, in running time proportional to the sum of the lengths of the strings. The constant of proportionality is low enough to make this algorithm of practical use, and the procedure can …
☆ 保存 99 引用 被引用次数：4221 相关文章 所有 20 个版本 »

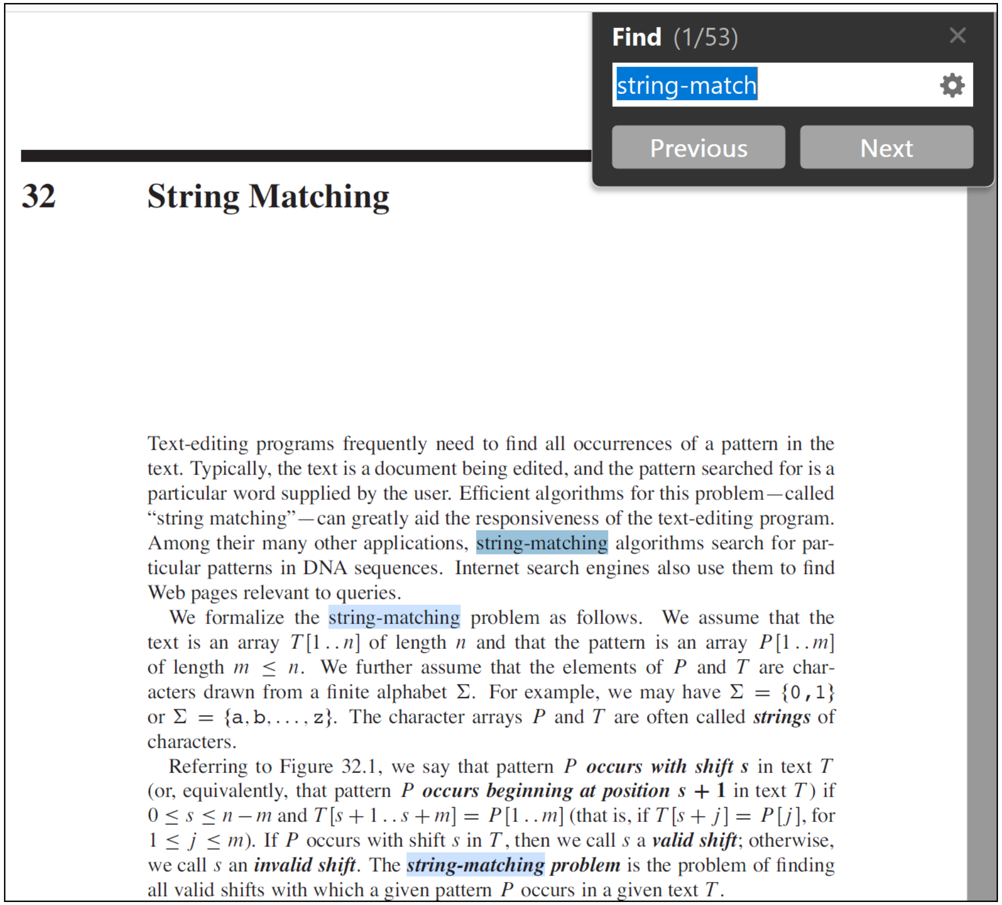
[HTML] Fast pattern-matching on indeterminate strings
J Holub, WF Smyth, S Wang - Journal of Discrete Algorithms, 2008 - Elsevier
In a string x on an alphabet Σ, a position i is said to be indeterminate iff x [i] may be any one of a specified subset {λ 1, λ 2,…, λ j} of Σ, 2⩽ j⩽| Σ|. A string x containing indeterminate positions is therefore also said to be indeterminate. Indeterminate strings can arise in DNA …
☆ 保存 99 引用 被引用次数：67 相关文章 所有 12 个版本

Fastest pattern matching in strings
L Colussi - Journal of Algorithms, 1994 - Elsevier
An algorithm is presented that substantially improves the algorithm of Boyer and Moore for pattern matching in strings, both in the worst case and in the average. Both the Boyer and Moore algorithm and the new algorithm assume that the characters in the pattern and in the …
☆ 保存 99 引用 被引用次数：61 相关文章 所有 4 个版本

Pattern matching in strings
AV Aho - Formal Language Theory, 1980 - Elsevier
… In this way an algorithm can construct from the pattern whatever Pattern Matching in

32 字符串匹配

- 解决这个问题的有效算法可以极大地提高文本编辑程序的响应能力。
- 字符串匹配算法也被用于搜索DNA序列中的特定模式等。



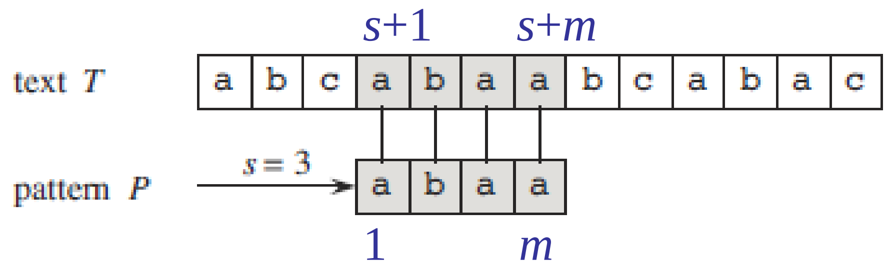
32 String Matching

Text-editing programs frequently need to find all occurrences of a pattern in the text. 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—called "string matching"—can greatly aid the responsiveness of the text-editing program. Among their many other applications, string-matching algorithms search for particular patterns in DNA sequences. Internet search engines also use them to find Web pages relevant to queries.

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 $m \le n$. We further assume that the elements of P and T are characters drawn from a finite alphabet Σ. For example, we may have $\Sigma = \{0,1\}$ or $\Sigma = \{a, b, \ldots, z\}$. The character arrays P and T are often called ***strings*** of characters.

Referring to Figure 32.1, 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 $0 \le s \le n - m$ and $T[s+1..s+m] = P[1..m]$ (that is, if $T[s+j] = P[j]$, for $1 \le j \le m$). 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.

32 字符串匹配



该模式只在文本出现了一次, 即移位 $s = 3$ 处。
移位 $s = 3$ 被称为一个有效移位。

字符串匹配问题:

- 文本: $T[1 .. n]$, 模式: $P[1 .. m]$, $m \le n$.
- 有限字母表: Σ.
  For example, $\Sigma = \{0, 1\}$ or $\Sigma = \{a, b, \ldots, z\}$.
- $P_i \in \Sigma$, $T_i \in \Sigma$.
- P occurs with shift s in T if $0 \le s \le n-m$ and
  $T[s+1 .. s+m] = P[1 .. m]$ (that is, if $T[s+j] = P[j]$, for $1 \le j \le m$).
  (或者说, P从 T 的 s+1 处开始出现).
- 有效移位 s: 若 P 在 T 移位 s 处出现 ; 否则, s 是无效移位。
- 找出在给定的 T 中出现给定 P 的所有有效移位。

- Σ^* :使用字母表 Σ 的字符组成的所有有限长度字符串的集合。例如：

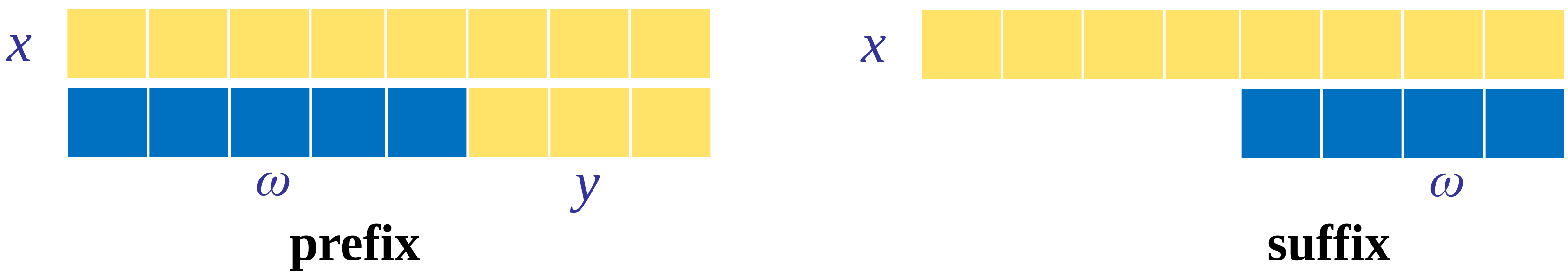
  $\Sigma = \{a, b, c\}$

  $\Sigma^* = \{\varepsilon, a, b, c, ab, bc, ac, abc, acb, aabbc, \ldots\ldots\}$

- ε : 长度为零的空字符串, 也属于 Σ^*.
- $|x|$: x 的长度。
- 两个字符串 x 和 y 的连接, 记为 xy, 长度为 $|x| + |y|$,由 x 中的字符后跟随 y 中的字符组成。

符号和术语

- $\omega \sqsubset x$: 字符串 ω 是 x 的**前缀**, 若 $x = \omega y$ 对于某个 $y \in \Sigma^*$.

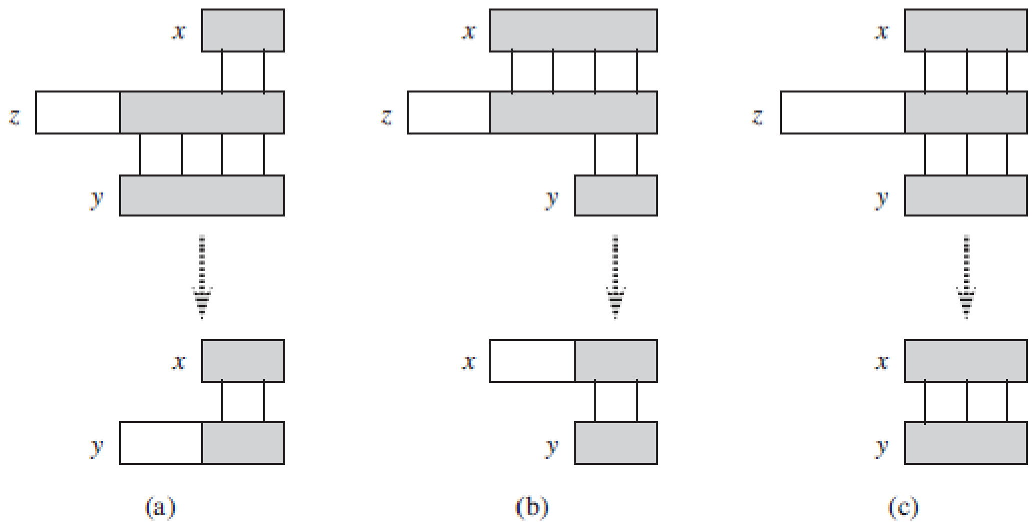


- $\omega \sqsupset x$: ω 是 x 的**后缀** , 若 $x = y\omega$ 对于某个 $y \in \Sigma^*$.
  - 若 $\omega \sqsubset$ x 或者 $\omega \sqsupset x$, 那么 $|\omega| \le |x|$。
  - 空字符串 ε 既是每个字符串的后缀，也是每个字符串的前缀。
  - 例如，**ab ⊏ abcca** ， **cca ⊐ abcca** 。
  - 对于任意字符串 x 和 y 以及任意字符 a, 我们有 $x \sqsupset y$ 当且仅当 $xa \sqsupset ya$.
  - ⊏ 和 ⊐是传递关系。

Lemma 32.1: (重叠后缀引理)

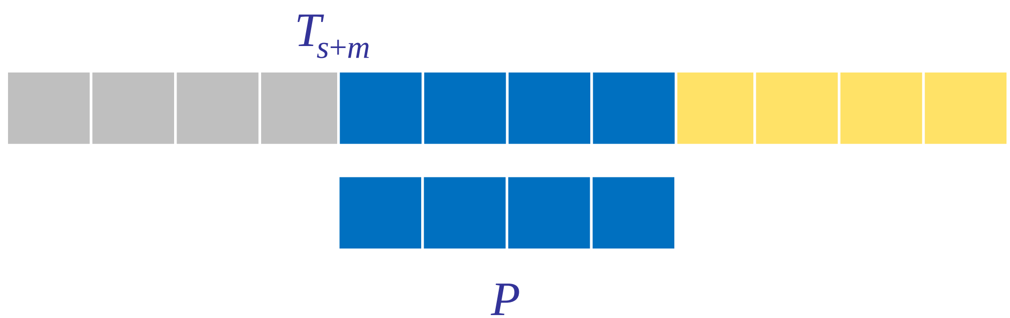
假设 *x*, *y*, *z* 是字符串且 x ⊐z, y ⊐z。若 |x| ≤ |y|, 那么 x ⊐y。若 |x| ≥ |y|, 则 y ⊐x。若 |x| = |y|, 则 x = y。

***Proof* 证明如图**



符号和术语

- 为了表示法的简洁，我们表示模式 $P[1\,..\,m]$k个字符的前缀 $P[1\,..\,k]$为 P_k。
  - $P_0 = \varepsilon$ 且 $P_m = P = P[1\,..\,m]$.
- 类似地，我们将文本 T 的k字符前缀表示为 T_k.
- 使用这种表示法，我们可以说明字符串匹配问题：**找到范围 $0 \le s \le n\text{-}m$ 内所有的位移 s 使 $P \sqsupset T_{s+m}$.**

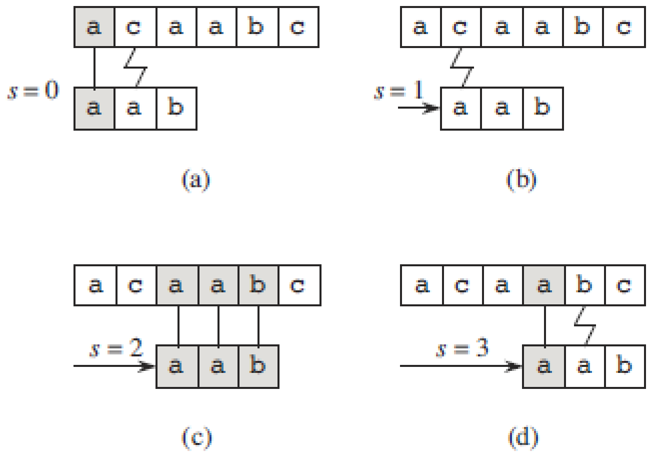


- 原语操作:比较字符。

32.1 朴素字符串匹配算法

朴素算法对s的n-m+1个可能值通过循环检查条件 $P[1 \,..\, m] = T[s+1 \,..\, s+m]$来找到所有的有效移位。

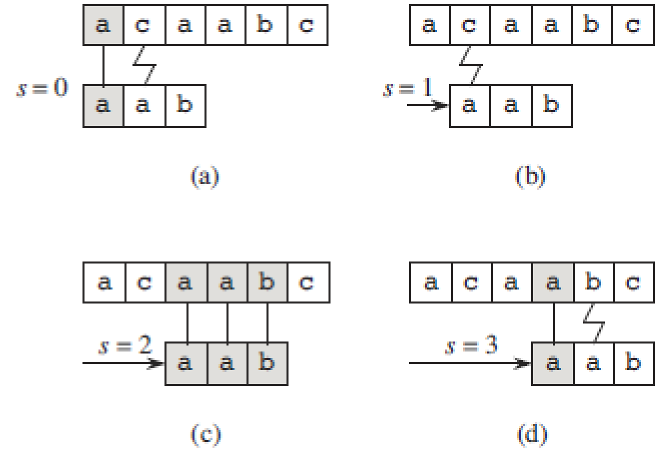
NAIVE-STRING-MATCHER(T, P)
1 $n \leftarrow$ length[T]
2 $m \leftarrow$ length[P]
3 for $s \leftarrow 0$ to n-m
4   if $P[1 \,..\, m] = T[s+1 \,..\, s+m]$
5   print "Pattern occurs with shift" s



- 该过程可以图形化地解释为在文本上滑动模式的“模板”。
- 第3行显式地考虑每个可能的移位。
- 在第4行上的测试确定当前移位是否有效; 这个测试包含一个隐式的循环。

32.1 朴素字符串匹配算法

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

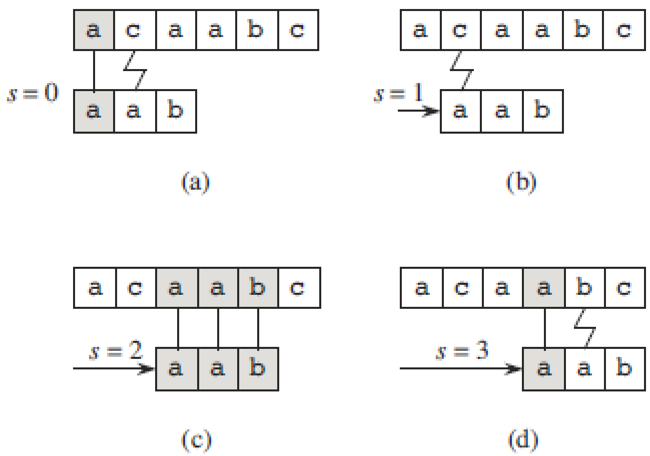



```
char *my_strstr(const char *T, const char *P)
{
    if(T == NULL)       return NULL;
    int n = strlen(T), m = strlen(P), s, i;
    for(s=0; s<=n-m; s++)    {
        for(i=0; i<m; i++)
            if(P[i] != T[s+i]) break;
        if(i == m) return T+s;
    }
    return NULL;
}
```

```
char *my_strstr(const char *T, const char *P)
{
    return strstr(T, P);  // 用库函数实现
}
```

- 代码：自己实现strstr（找第一次匹配的位置）
- 所有匹配（伪代码部分）都找出来，怎么实现？
- 怎么实现strrstr?（最后一次匹配的位置）

32.1 朴素字符串匹配算法



- **Running time ?**

$O((n\text{-}m+1)\mathrm{m})$

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

32.1 朴素字符串匹配算法

Exercise 32.1-2, 32.1-4

*32.2 The Rabin-Karp algorithm

T

a	b	c	a	d	d	b	c	a	a
	b	c	a	d					

P

R-K algorithm performs well in practice and that also generalizes to other algorithms for related problems, such as two-dimensional pattern matching.
用了Hash和数论的方法。
对T_s计算p时，还有更有效的方法（后一个m个T_s的p值跟前面计算的结果有关系，可充分利用前面的计算信息，加快计算速度）。
计算$p(P)$，可以用Horner’s rule.

编码规则: {a, b, c, d} ⇨ {0, 1, 2, 3},

则 $p(P) = 1*4^3 + 2*4^2 + 0*4^1 + 3*4^0 = 99$

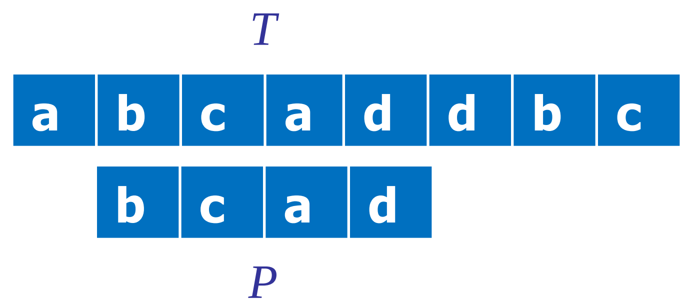
$(12345 = 1*10^4 + 2*10^3 + 3*10^2 + 4*10^1 + 5*10^0)$

若 $p(P) = p(T_s)$ ($T_s = T[s+1 .. s+m]$, s = 0, 1, ..), 匹配成功。

或者如果 $p(P)$ mod $q = p(T_s)$ mod q, 检查是否 $P = T_s$

预处理时间 $\Theta(m)$,
最差运行时间 $\Theta((n-m+1)m)$.

*32.2 The Rabin-Karp algorithm



coding rule: {a, b, c, d} ⇨ {0, 1, 2, 3},

then $p(P) = 1*4^3 + 2*4^2 + 0*4^1 + 3*4^0 = 99$

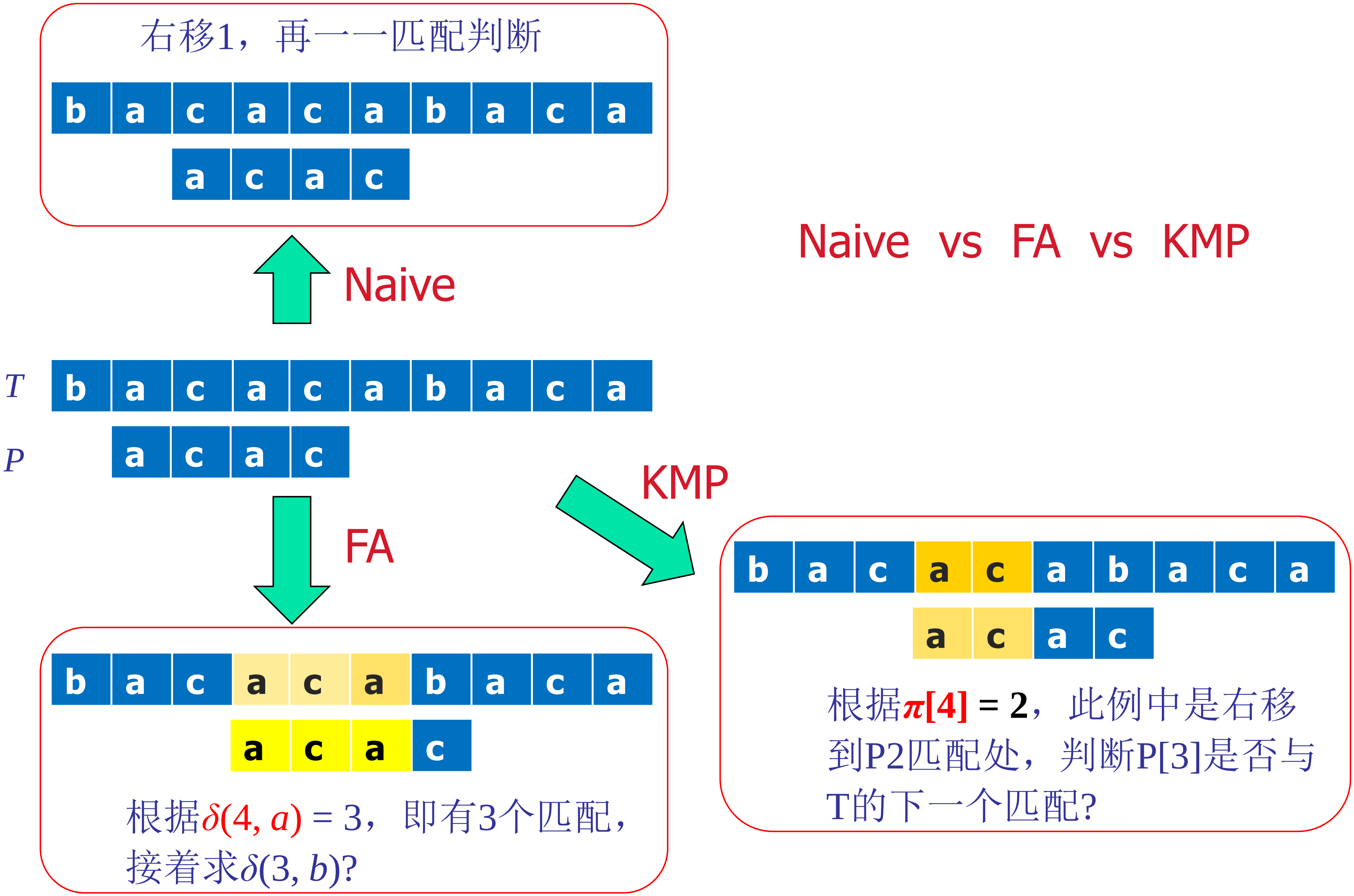
$((12345)_{10} = 1*10^4 + 2*10^3 + 3*10^2 + 4*10^1 + 5*10^0)$

Efficient randomized pattern-matching algorithms

RM Karp, MO Rabin - IBM journal of research and development, 1987 - ieeexplore.ieee.org

We present randomized algorithms to solve the following string-matching problem and some of its generalizations: Given a string X of length n (the pattern) and a string Y (the text), find the first occurrence of X as a consecutive block within Y. The algorithms represent strings of …

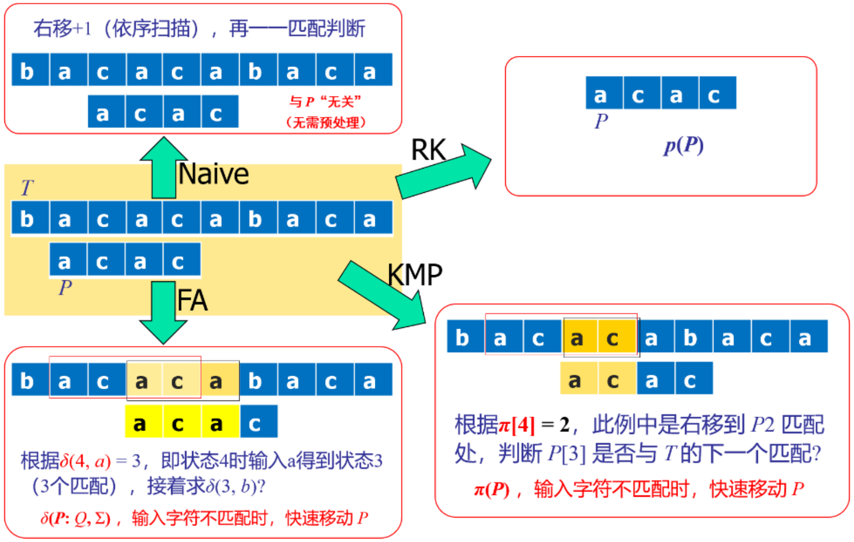
Naive vs FA vs KMP



Naive vs RK vs FA vs KMP

Algorithm	Preprocessing time	Matching time
Naive	0	$O((n-m+1)m)$
Rabin-Karp	$\Theta(m)$	$O((n-m+1)m)$
Finite automaton	$O(m\|\Sigma\|)$	$\Theta(n)$
Knuth-Morris-Pratt	$\Theta(m)$	$\Theta(n)$

	关键	特征
FA	求 $\delta(P: Q, \Sigma)$	输入字符$T[i]$不匹配时，快速移动 P，每个$T[i]$匹配一次
KMP	求 $\pi(P)$	输入字符$T[i]$不匹配时，快速移动 P，每个$T[i]$匹配一次



32.3 有限自动机的字符串匹配

- **许多字符串匹配算法构建一个有限自动机，扫描文本T以寻找模式P的所有出现。**
- **这些字符串匹配自动机非常高效：**
  - **他们只检查每个文本字符一次；**
  - **每个文本字符占用恒定的时间。**
- **匹配时间为 $\Theta(n)$.**
  - **如果Σ很大，预处理时间(通过模式构建自动机)可能很大。（对英文文本来说，小写26，大写26，数字10，共62，Σ不算大）**
  - **32.4节描述了解决这个问题的巧妙方法。**

有限自动机

- **一个有限自动机 M 是一个五元组** $M = (Q, q_0, A, \Sigma, \delta)$ **,** 其中
  - Q 是一个有限状态集,
  - $q_0 \in Q$ 是初始状态,
  - $A \subseteq Q$ 是一组特殊的接受状态,
  - Σ 是输入的有限字母表,
  - δ 是一个从 $Q \times \Sigma$ 到 Q 的函数,称为 M 的转移函数。
- 有限状态机从状态 q_0 开始，每次读入一个输入字符串的字符。如果自动机处于状态 q 且读取到了字符 a, 它将从状态 q 转移(“进行转换”)到状态 $\delta(q, a)$. 若它的当前状态 q 是 A 的成员, 自动机 M 被认为**接受**了目前读取的字符串。 不被接受的输入称为被**拒绝**。

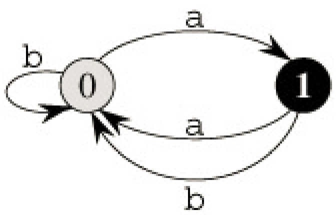
$$q_0 \longrightarrow \overset{a_1}{\boxed{\mathbf{M}}} \longrightarrow q_1 \longrightarrow \overset{a_2}{\boxed{\mathbf{M}}} \longrightarrow q_2 \longrightarrow \cdots \longrightarrow q_i \longrightarrow \overset{a_{i+1}}{\boxed{\mathbf{M}}} \longrightarrow q_{i+1}$$

有限自动机:一个例子

- **图32.6:一个简单的两状态有限自动机，状态集*Q* = {0, 1}, 初始状态 q_0 = 0, 输入字母表Σ = {a, b}。**

state	input a	input b
0	1	0
1	0	0

(a)



(b)

- (a) 转移函数 δ 的表格表示。
- (b) 一个等价的状态转换图。

a b a a a
<0, 1, 0, 1, 0, 1>

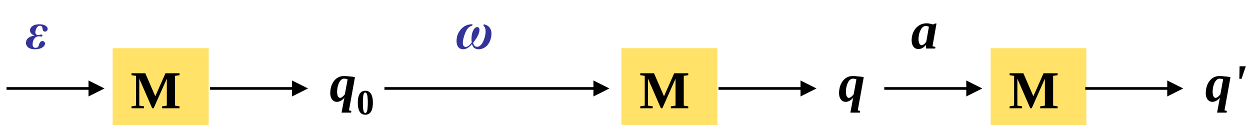
- 状态1是唯一的接受状态(显示为黑色)。 有向边表示状态转移例如，标记为b的状态1到状态0的边表示 δ(1, b) = 0。这个自动机接受以奇数个a结尾的字符串。例如, 这个自动机为输入 abaaa (包括初始状态)输入的状态序列是 <0, 1, 0, 1, 0, 1>, 因此它接受这个输入。对于输入 abbaa, 状态序列是 <0, 1, 0, 0, 1, 0>, 所以拒绝这个输入。

有限自动机:终态函数

- 有限自动机 M 引入了一个函数 Φ, 称为 ***终态函数***, 从 Σ^* 到 Q 使 $\Phi(\omega)$ 是 M 扫描完字符串 ω 后的最终状态. (Σ^* : 字母表 Σ 中的字符组成的所有有限字符串的集合。)
- 因此，M 接受字符串 ω 当且仅当$\Phi(\omega) \in A$。
- 函数 Φ 由递归关系定义

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

$$\Phi(\omega a) = \delta(\Phi(\omega), a) \text{ for } w \in \Sigma^*, a \in \Sigma \,.$$

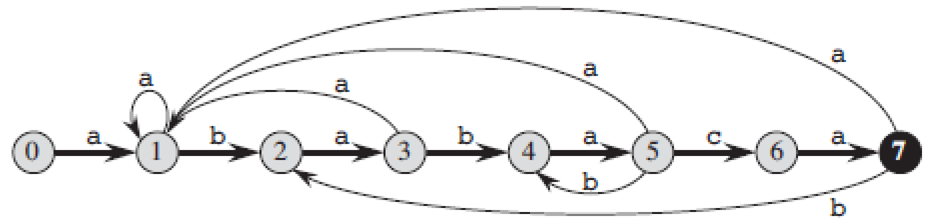


$q_0 = \Phi(\varepsilon)$ $\qquad$ $q = \Phi(\omega)$ $\qquad$ $q' = \Phi(\omega a) = \delta(\Phi(\omega), a)$

字符串匹配自动机

- 每个模式 P 都有一个字符串匹配自动机。
- 在使用该自动机搜索文本字符串之前，必须在预处理步骤中从模式构造该自动机。
- 图32.7说明了模式

P = ababaca的这种构造。



(a)

		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		

(b)

Figure 32.7

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

(c)

ω $\qquad\qquad$ ω

aba $\qquad\qquad$ abaa

P : ababaca $\qquad$ ababaca

$\delta(2, a) = 3$ $\qquad$ $\delta(3, a) = 1$

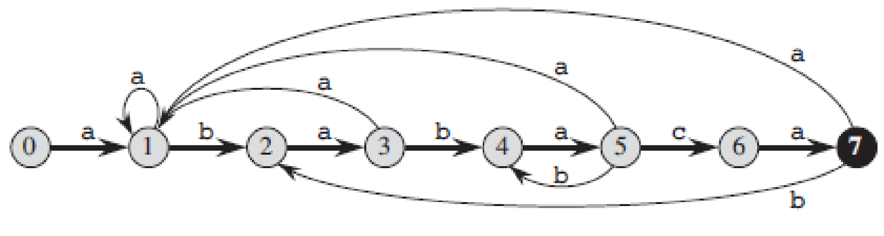
读入T_k，看T_k的后缀跟P的前缀P_p的匹配情况，匹配数为m，则字符串匹配。

字符串匹配自动机:一个例子

- (a) 字符串匹配自动机的状态转移图接受以字符串 ababaca 结尾的所有字符串。状态 0 是初始状态, 状态7(显示为黑色)是唯一的接受状态。标记为α的状态i到状态j的有向边表示 $\delta(i, \alpha) = j$。向右的边构成了自动机的“脊柱”, 显示为加粗的线, 对应模式和输入字符之间的成功匹配。

向左的边对应失败的匹配。

一些对应失败匹配的边没有显示出来;按照惯例，如果对于某些 $\alpha \in \Sigma$ 状态 i 没有标记为 α 的输出边, 则$\delta(i, \alpha) = 0$。



(a)

Figure 32.7

字符串匹配自动机:一个例子

(b) 对应的转移函数 δ, 模式串 P = ababaca. 模式和输入字符间的成功匹配对应的条目显示为阴影。

state	input a	input b	input 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	

(b)

(c) 自动机对文本 T = abababacaba 的操作。

自动机处理T_i后，其状态为$\Phi(T_i)$ 。

发现了该模式的一次出现，在位置9终止。

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

(c)

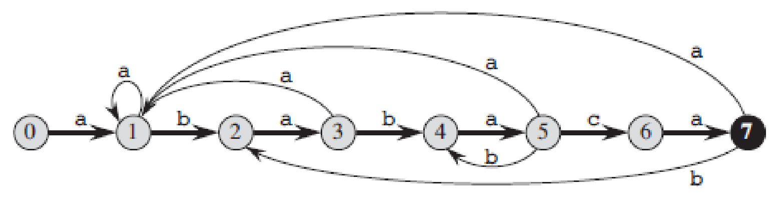
核心：如何构建 $\delta(q, a)$?

字符串匹配自动机:一个例子

自动机对文本 T = abababacaba 的操作。

自动机处理T_i后，其状态为$\Phi(T_i)$ 。

$\delta(0, \text{a}) = 1, \delta(3, \text{b}) = 4, \delta(4, \text{c}) = 0, \delta(5, \text{b}) = 4$?



(a)

state	input 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	

(b)

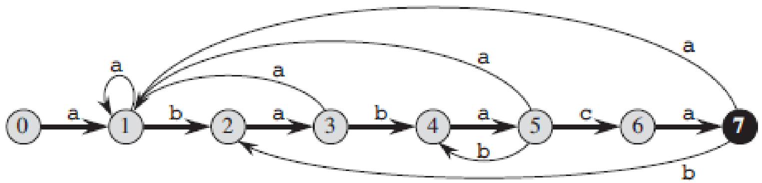
i	—	1	2	3	4	5	6	7	8	9	10	11
$T[i]$	—	a	b	a	b	a	b	a	c	a	b	a
state $\phi(T_i)$	0	1	2	3	4	5	4	5	6	7	2	3

(c)

字符串匹配自动机:一个例子

自动机对文本 T = abababacaba 的操作。

自动机处理T_i后，其状态为$\Phi(T_i)$ 。



(a)

$\delta(0, a) = 1$, $\delta(3, b) = 4$, $\delta(4, c) = 0$, $\delta(5, b) = 4$?

	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	

(b)

T a b a b a b a c a b a ...

P a b a b a c a

...

a ...

P **a b a b a c a**

从空字符开始，从文本串 T 的第一字符依序扫描，输入 a…时（…表示还有很多字符），P 的前1个跟其匹配，即 $\delta(0, a) = 1$；【*从T一个一个地扫描，跟KMP有相似性】

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

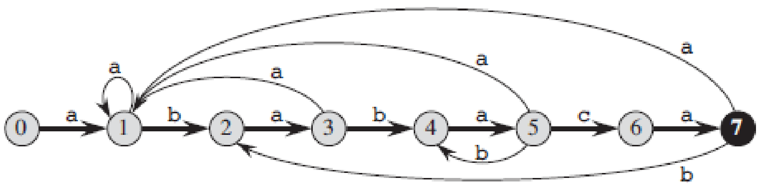
(c)

字符串匹配自动机:一个例子

自动机对文本 T = abababacaba 的操作。

自动机处理T_i后，其状态为$\Phi(T_i)$ 。

δ(0, a) = 1, **δ(3, b) = 4**, δ(4, c) = 0, δ(5, b) = 4 ?



(a)

T	a	b	a	b	a	b	a	c	a	b	a ...
P	a	b	a	b	a	c	a				

	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	

(b)

a b a ...

a b a b ...

P **a b a b**

状态3时（有3个匹配），输入b，即文本串 T 为abab…时（…表示还有很多字符），P 的前4个跟其匹配，即 δ(3, b) = 4

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

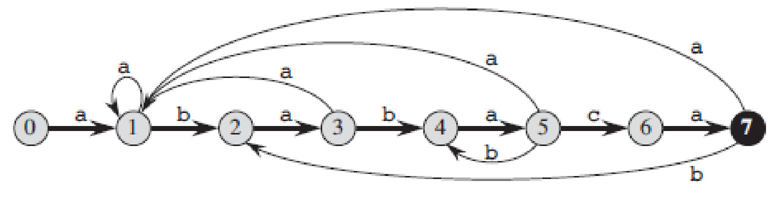
(c)

字符串匹配自动机:一个例子

自动机对文本 T = abababacaba 的操作。

自动机处理T_i后，其状态为$\Phi(T_i)$ 。

δ(0, a) = 1, δ(3, b) = 4, **δ(4, c) = 0**, δ(5, b) = 4 ?



(a)

T a b a b a b a c a b a …

P a b a b a c a

a b a b …

a b a b c …

P **a b a b a**

a b a b

a b a

……

状态4时（有4个匹配），输入c，即文本串 T 为ababc…时（…表示还有很多字符），P 的前5个跟其不匹配，即δ(4, c) != 5；
把 P 按字符右移1位（寻找新的可能匹配），P 的前4个跟其不匹配，即δ(4, c) != 4；
把 P 按字符右移2位，P 的前3个跟其不匹配，即δ(4, c) != 3；
以此类推。【*有点像KMP了】

state	input 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	

(b)

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

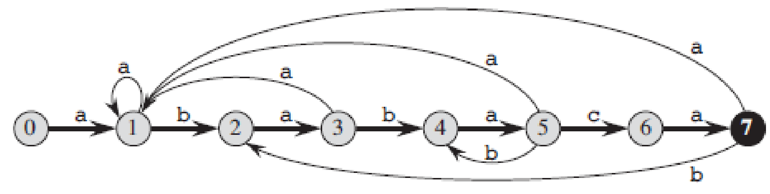
(c)

字符串匹配自动机:一个例子

自动机对文本 T = abababacaba 的操作。

自动机处理T_i后，其状态为$\Phi(T_i)$ 。

$\delta(0, a) = 1$, $\delta(3, b) = 4$, $\delta(4, c) = 0$, **$\delta(5, b) = 4$** ?



(a)

T a b a b a b a c a b a …

P a b a b a c a

a b a b a …

a b a b a b …

P **a b a b a c**

a b a b a

a b a b

每次把 P 右移1位后，都从 P 的第一个字符开始匹配，跟naive方法一样？

	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	

(b)

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

(c)

字符串匹配自动机:**后缀函数**

- 对应 P 的后缀函数 σ :
从 Σ^* 到 $\{0, 1, . . . , m\}$ 的映射使 $\sigma(x)$ 是 x的后缀且是P的最长前缀的长度:

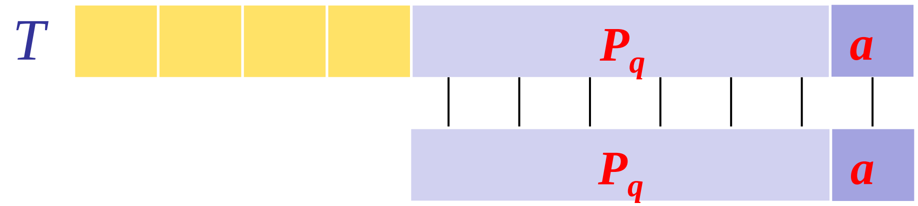
$$\sigma(x) = \max\ \{k : P_k \sqsupset x\}.$$

- 后缀函数σ的定义很好，因为空字符串$P_0 = \varepsilon$ 是每个字符串的后缀。例如,
  - 对于模式 P = ab, 我们有 $\sigma(\varepsilon) = 0$, $\sigma(\mathrm{ccaca}) = 1$, $\sigma(\mathrm{ccab}) = 2$。
- 对于长度为m的模式P，我们有 $\sigma(x) = m$ 当且仅当 $P \sqsupset x$。
- 从后缀函数的定义来看, 若 $x \sqsupset y$, 则$\sigma(x) \le \sigma(y)$。

字符串匹配自动机

$M = (Q, q_0, A, \Sigma, \delta)$

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



- 我们定义对应给定模式$P[1\ ..\ m]$**字符串匹配自动机**如下。
  - 转移函数 δ 由以下等式定义, 对于任意状态 q 和字符 a:

$$\boldsymbol{\delta(q, a) = \sigma(P_q a).} \qquad \textbf{(32.3)}$$

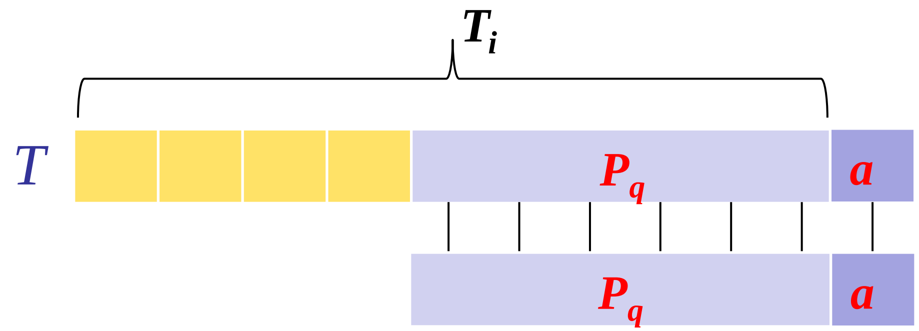
  - 其中,状态集 Q 是 $\{0, 1, . . . , m\}$, 起始状态 q_0 是状态 0, 状态 m 是唯一接受状态 A。

$\delta(q, a) = \sigma(P_q a)$ 的定义合理，后面将证明，$\delta(q, a) = \sigma(P_q a) = \sigma(T_i a)$, 即，扫描 T_i 后，匹配为 P_q，接着读入a，对 $T_i a$ 的匹配与对 $P_q a$ 的匹配是一样的。由于 $P_q a$ 的长度比 $T_i a$ 短，处理起来就简单得多。

字符串匹配自动机

$M = (Q, q_0, A, \Sigma, \delta)$

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



We define the *machine* ***M*** :

$Q = \{0, 1, \ldots, m\}$; $q_0 = 0$; $A = \{m\}$;

$$\boldsymbol{\delta(q, a) = \sigma(P_q a).} \qquad \textbf{(32.3)}$$

- 直观上, 自动机 ***M*** 保持一个不变量:

$$\boldsymbol{\Phi(T_i) = \sigma(T_i)}, \qquad \textbf{(where, } \boldsymbol{\Phi(T_i) = q = \sigma(T_i)}\textbf{).} \qquad \textbf{(32.4)}$$

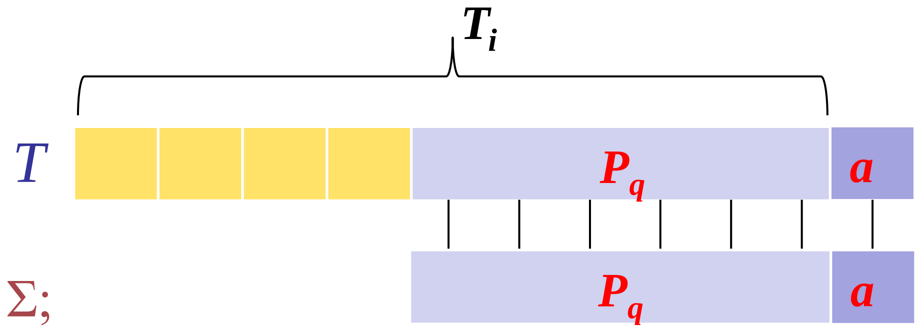
自动机 M 扫描字符串 T 的过程中，扫描到前缀子串 T_i 时状态为 q（为 T_i 的后缀函数$\sigma(T_i)$），接着扫描下一个字符 $T[i+1]$（记为a），状态转移到$\delta(q, a) = \sigma(P_q a)$，这就是扫描到前缀子串 T_{i+1} 时状态（为 T_{i+1} 的后缀函数$\sigma(T_{i+1})$）

$$\boldsymbol{\Phi(T_{i+1}) = \Phi(T_i a) = \delta(\Phi(T_i), a) = \delta(q, a) = \sigma(P_q a) \stackrel{?}{=} \sigma(T_i a) = \sigma(T_{i+1})} \qquad \textbf{(32.4)*}$$

[(32.3) maintains the invariant, or, it is rationale for defining (32.3).]

字符串匹配自动机

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



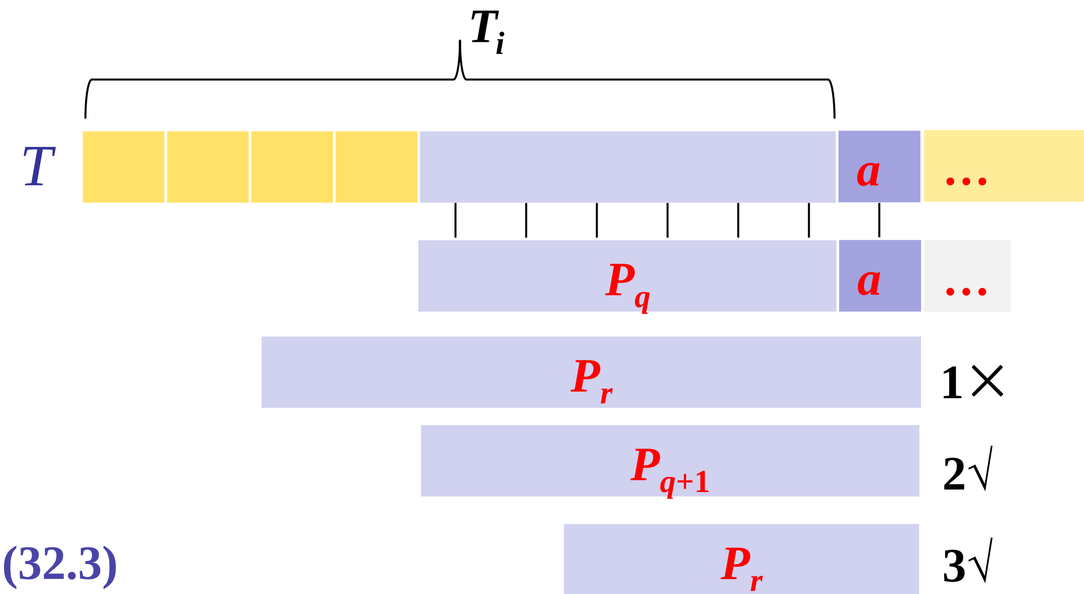
We define the *machine* M :

$Q = \{0, 1, \ldots, m\}$; $q_0 = 0$; $A = \{m\}$; Σ;

$$\delta(q, a) = \sigma(P_q a). \qquad (32.3)$$

- $\Phi(T_{i+1}) = \Phi(T_i a) = \delta(\Phi(T_i), a) = \delta(q, a) = \sigma(P_q a) \overset{?}{=} \sigma(T_i a) = \sigma(T_{i+1})$ **(32.4)**
  [(32.3) 维持不变式,或者说, 它是定义 (32.3)的根本原因。]
- 我们可以证明 $\sigma(T_i a) = \sigma(P_q a)$ [*Lemma 32.3*]. (32.A)
  Maintain: 因此, 设置(32.3)保持着所需的不变式 (32.4)。
- **Compute:** 通过 (32.A), 为了计算P的前缀且是 $T_i a$ 的最长后缀的长度, 我们可以计算P的前缀且是$P_q a$的最长后缀的长度。
- 在每个状态, 自动机只需要直到目前为止已读内容的后缀也是P的最长前缀的长度。

字符串匹配自动机



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

We define the *machine* $\boldsymbol{M}$:

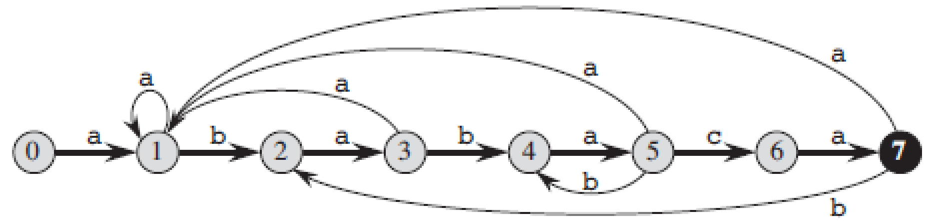
$Q = \{0, 1, \ldots, m\};$

$q_0 = 0;\ A = \{m\};\ \Sigma;$

$\boldsymbol{\delta(q, a) = \sigma(P_q a)}.$ **(32.3)**

- $\boldsymbol{\Phi(T_{i+1}) = \Phi(T_i a) = \delta(\Phi(T_i), a) = \delta(q, a) = \sigma(P_q a) = \sigma(T_i a) = \sigma(T_{i+1})}$ **(32.4)**
  [(32.3)维持不变式,或者说, 它是定义 (32.3)的根本原因。]
- **If** $\boldsymbol{\sigma(T_i) = \sigma(P_q) = q}$**, then** $\boldsymbol{\sigma(T_i a) = \sigma(P_q a)}$ **[*Lemma 32.3*].** **(32.A)**
  Proof
  情况1是不可能的。若 1 满足， $\boldsymbol{\sigma(T_i) > q}$ **, 与假设矛盾。**
  显然，若 $\boldsymbol{P[q+1] = a}$ **, 它是情形2，否则就是情形3。**
- ***Form 32.3* 和 *32.A* 表示自动机扫描字符 $T[i]$ 后处于状态 $\sigma(T_i)$ 。因为 $\sigma(T_i) = m$ 当且仅当 $P \sqsupset T_i$, 自动机处于接受状态 m 当且仅当模式 P 刚刚被扫描。**

字符串匹配自动机



(a)

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

We define the *machine* ***M*** :

$Q = \{0, 1, . . . , m\};$

$q_0 = 0; A = \{m\}; \Sigma;$

$\boldsymbol{\delta(q, a) = \sigma(P_q a).}$ **(32.3)**

state	input 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	

(b)

Figure 32.7

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

(c)

- 例如，在图32.7的字符串匹配自动机中, δ(5, **b**) = 4.
  - 我们这样做是因为，如果自动机在状态 q = 5时读到一个b, 则 P_q**b** = **ababab**, P 的最长前缀(也是**ababab**的后缀)是P_4 = **abab**。

字符串匹配自动机:**程序**

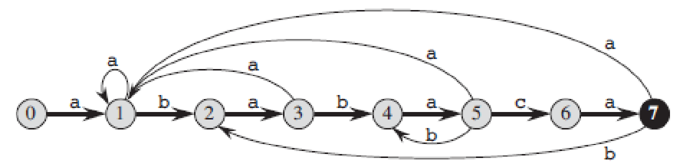
$\sigma(x) = \max\ \{k : P_k \sqsupset x\}.$

A string-matching automaton

$\boldsymbol{M} = (Q, q_0, A, \Sigma, \delta)$:

$Q = \{0, 1, . . . , m\};\ \ q_0 = 0;\ \ A = \{m\};$

$$\boldsymbol{\delta(q, a) = \sigma(P_q a) = \sigma(T_{i-1} a)}\ . \qquad \textbf{(32.3)}$$



(a)

state	input 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	

(b)

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

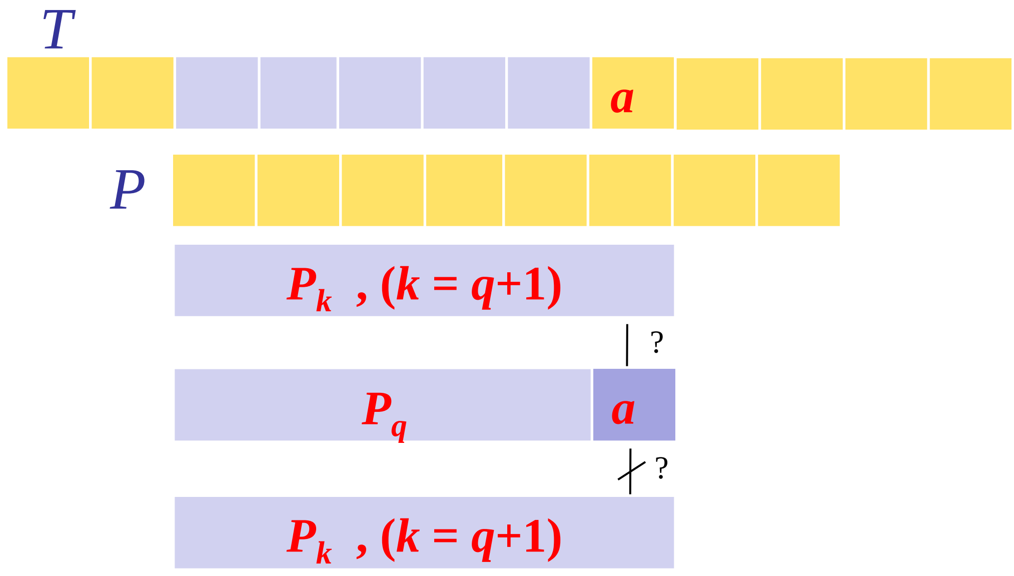
(c)

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

运行时间？

- 匹配时间为 $\boldsymbol{\Theta(n)}$.
- 但是它不包括计算转移函数 δ 所需的预处理时间。

计算转移函数



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

A string-matching automaton

$\boldsymbol{M} = (Q, q_0, A, \Sigma, \delta)$:

$Q = \{0, 1, \ldots, m\}$; $q_0 = 0$; $A = \{m\}$;

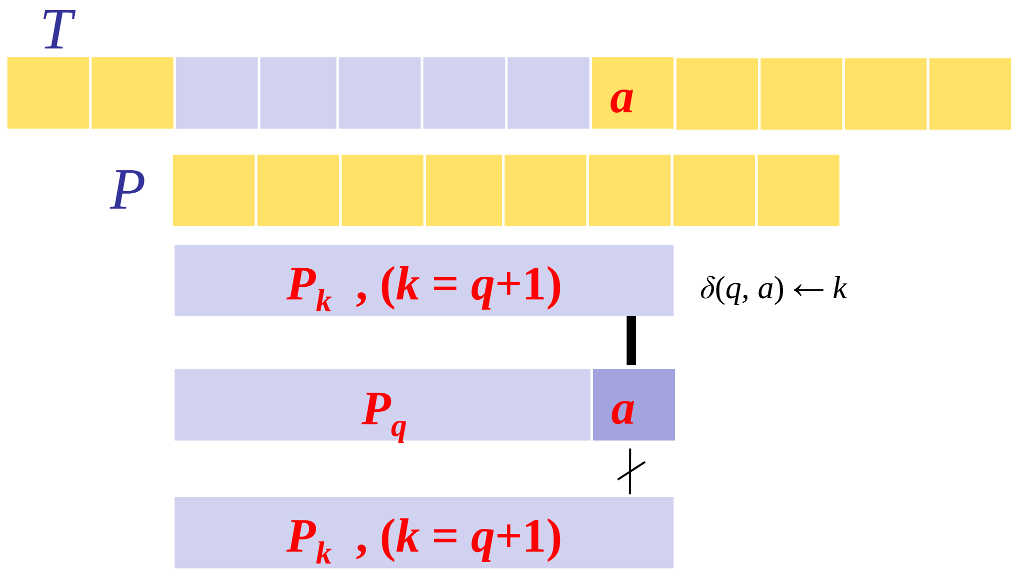
$$\boldsymbol{\delta(q, a) = \sigma(P_q a)} \ . \qquad \textbf{(32.3)}$$

Computing δ from a given pattern $P[1 .. m]$:

```
COMPUTE-TRANSITION-FUNCTION(P, Σ)
  m ← length[P]
  for q ← 0 to m
      for each character a ∈ Σ
          k ← min(m, q + 1)
          while P_k !⊐ P_q a
              k--
          δ(q, a) ← k
  return δ
```

- **P_q 时（即T与P的前q个字符匹配时），输入第 q+1（即第k个）字符 α 时：**
  1. **$k = q+1$ (超过m时，取m)**
  2. **$P_k \sqsupset P_q a$ ？**

计算转移函数



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

A string-matching automaton

$M = (Q, q_0, A, \Sigma, \delta)$:

$Q = \{0, 1, . . . , m\}$; $q_0 = 0$; $A = \{m\}$;

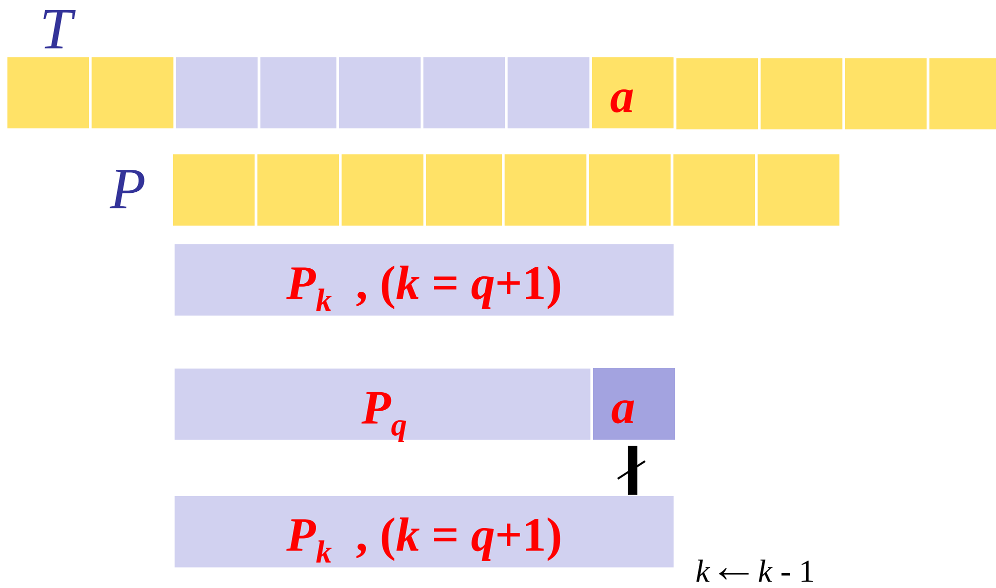
$$\boldsymbol{\delta(q, a) = \sigma(P_q a)} \ . \qquad \textbf{(32.3)}$$

Computing δ from a given pattern $P[1 .. m]$:

```
COMPUTE-TRANSITION-FUNCTION(P, Σ)
m ← length[P]
for q ← 0 to m
    for each character a ∈ Σ
        k ← min(m, q + 1)
        while P_k !⊐ P_q a
            k--
        δ(q, a) ← k
return δ
```

- **P_q 时（即T与P的前q个字符匹配时），输入第 q+1（即第k个）字符 α 时：**
  1. **$k = q+1$ (超过m时，取m)**
  2. **$P_k \sqsupset P_q a$?**
  3. **若2成立, 则 $P_q\alpha == P_k$ ， 即，对在T的继续扫描过程中，若扫描的下一个字符α==$P[k]$，则匹配字符增加1；**

计算转移函数



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

A string-matching automaton

$\boldsymbol{M} = (Q, q_0, A, \Sigma, \delta)$:

$Q = \{0, 1, \ldots, m\}$; $q_0 = 0$; $A = \{m\}$;

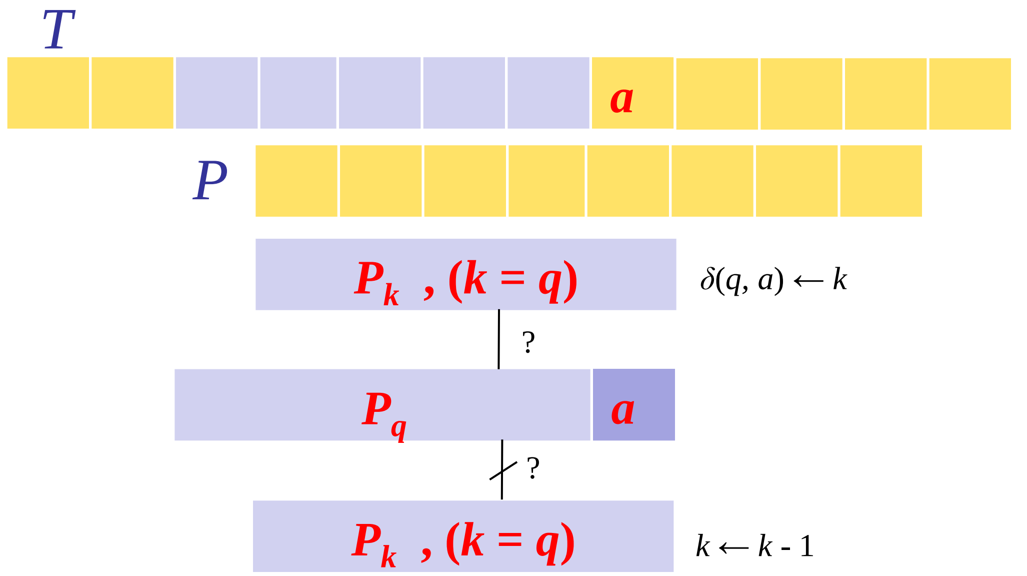
$\boldsymbol{\delta(q, a) = \sigma(P_q a)}$. **(32.3)**

Computing δ from a given pattern $P[1 .. m]$:

```
COMPUTE-TRANSITION-FUNCTION(P, Σ)
m ← length[P]
for q ← 0 to m
    for each character a ∈ Σ
        k ← min(m, q + 1)
        while P_k !⊐ P_q a
            k--
        δ(q, a) ← k
return δ
```

- P_q 时（即T与P的前q个字符匹配时），输入第 q+1（即第k个）字符 α 时：
  1. $k = q+1$ (超过m时，取m)
  2. $P_k \sqsupset P_q a$ ？
  3. 若2成立, 则 $P_q\alpha == P_k$ ，即，对在T的继续扫描过程中，若扫描的下一个字符α==$P[k]$，则匹配字符个数为k;
  4. 若 2不成立, 即 $P_q\alpha$!= P_k ，即，对在T的继续扫描过程中，若扫描的下一个字符α != $P[k]$，模版右移(k--)，goto step 2;

计算转移函数



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

A string-matching automaton

$M = (Q, q_0, A, \Sigma, \delta)$:

$Q = \{0, 1, \ldots, m\}$; $q_0 = 0$; $A = \{m\}$;

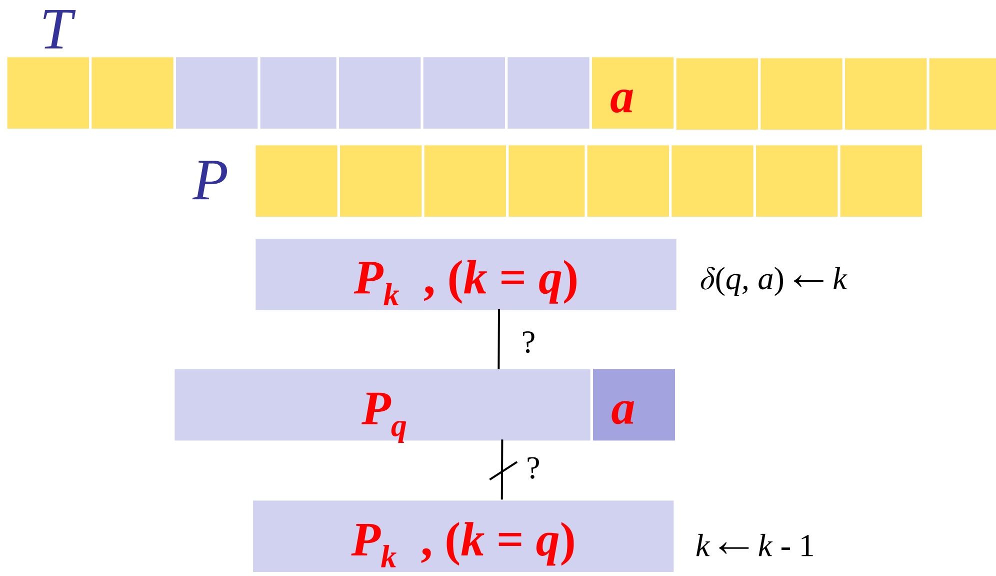
$\delta(q, a) = \sigma(P_q a)$. (32.3)

Computing δ from a given pattern $P[1 .. m]$:

```
COMPUTE-TRANSITION-FUNCTION(P, Σ)
m ← length[P]
for q ← 0 to m
    for each character a ∈ Σ
        k ← min(m, q + 1)
        while P_k !⊐ P_q a
            k--
        δ(q, a) ← k
return δ
```

- **P_q 时（即T与P的前q个字符匹配时），输入第 q+1（即第k个）字符 α 时：**
  1. $k = q+1$ (超过m时，取m)
  2. $P_k \sqsupset P_q a$?
  3. 若2成立, 则 $P_q\alpha == P_k$ ， 即，对在T的继续扫描过程中，若扫描的下一个字符α==$P[k]$，则匹配字符个数为k;
  4. 若 2不成立, 即 $P_q\alpha$!= P_k ， 即，对在T的继续扫描过程中，若扫描的下一个字符α != $P[k]$，模版右移(k--)，goto step 2;

计算转移函数



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

A string-matching automaton

$M = (Q, q_0, A, \Sigma, \delta)$:

$Q = \{0, 1, \ldots, m\}$; $q_0 = 0$; $A = \{m\}$;

$$\delta(q, a) = \sigma(P_q a)\ . \qquad (32.3)$$

Computing δ from a given pattern $P[1 .. m]$:

```
COMPUTE-TRANSITION-FUNCTION(P, Σ)
m ← length[P]
for q ← 0 to m                                   ... O(m)
    for each character a ∈ Σ                    ... O(|Σ|)
        k ← min(m, q + 1)
        while P_k !⊐ P_q a                       ... O(m)
            k--                                  ... O(m)
        δ(q, a) ← k
return δ
```

- **Running time ?**

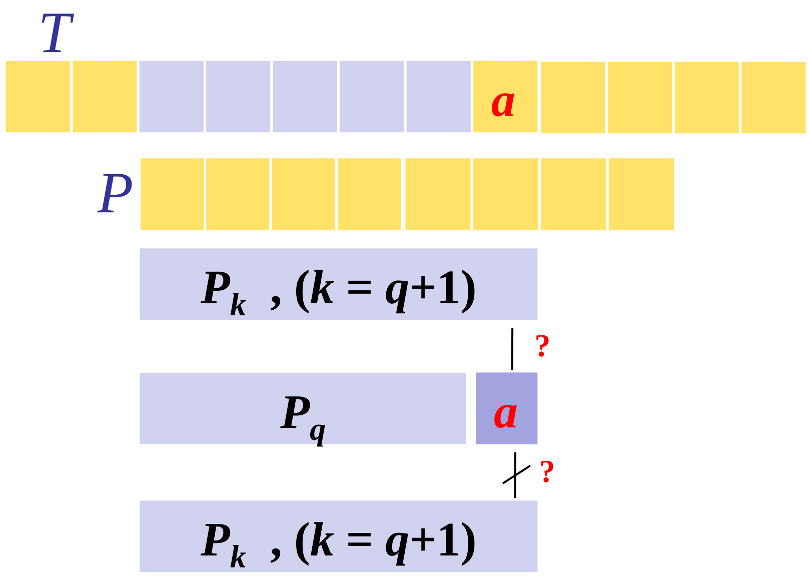
  $O(m^3|\Sigma|)$

- **Much faster procedures exist**

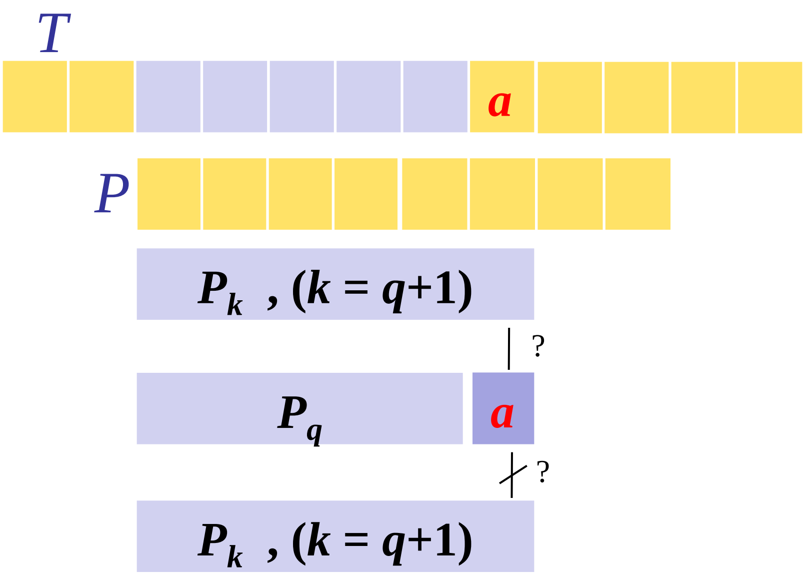
  ☆ $O(m\ |\Sigma|)$, Exercise 32.4-8.

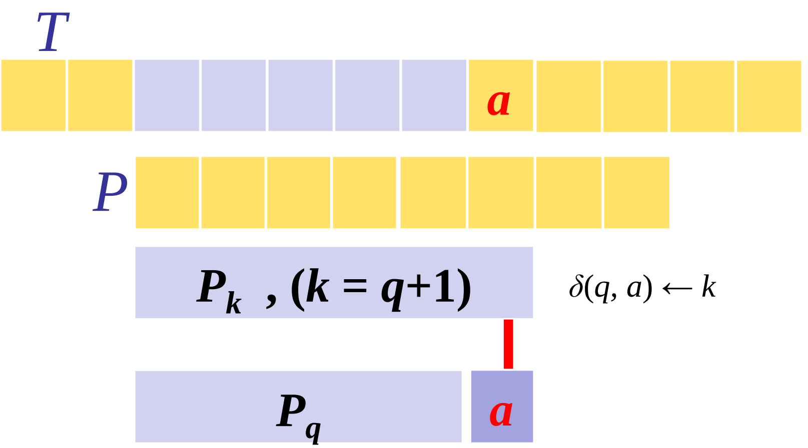
  ☆ $O(m)$, chapter 32.4, KMP.

32.3 有限自动机的字符串匹配

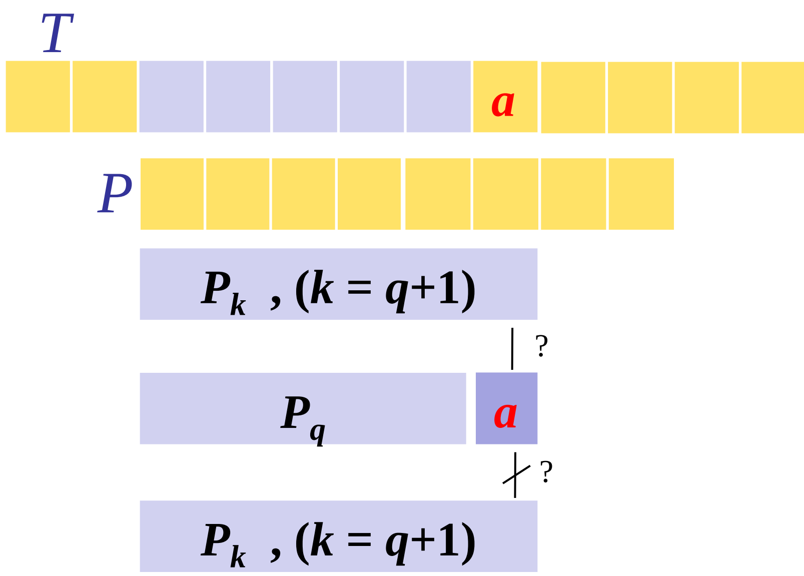


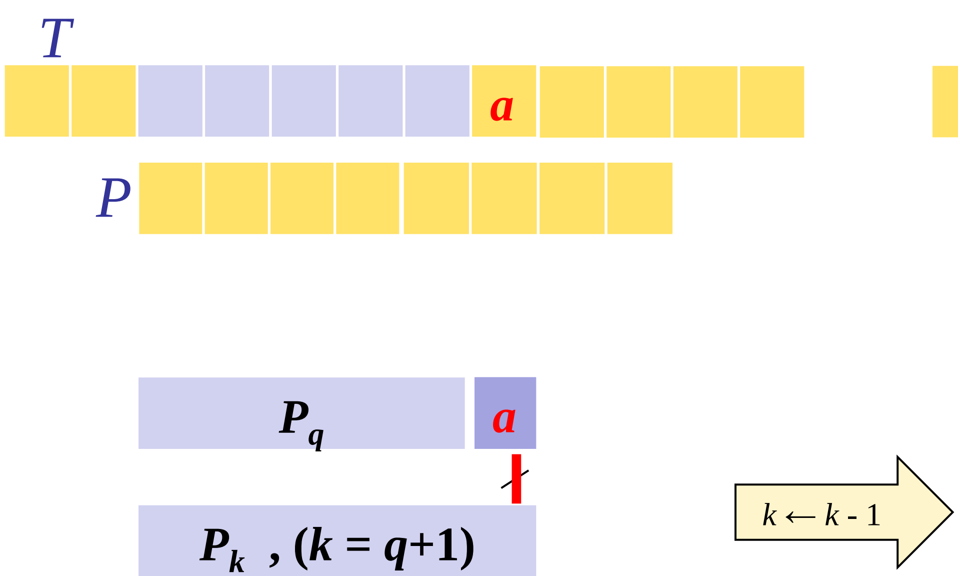
32.3 有限自动机的字符串匹配

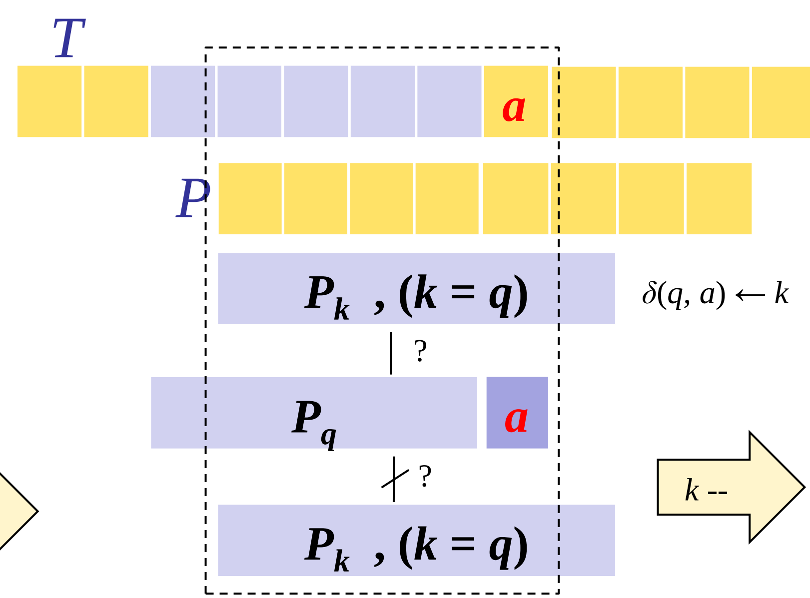




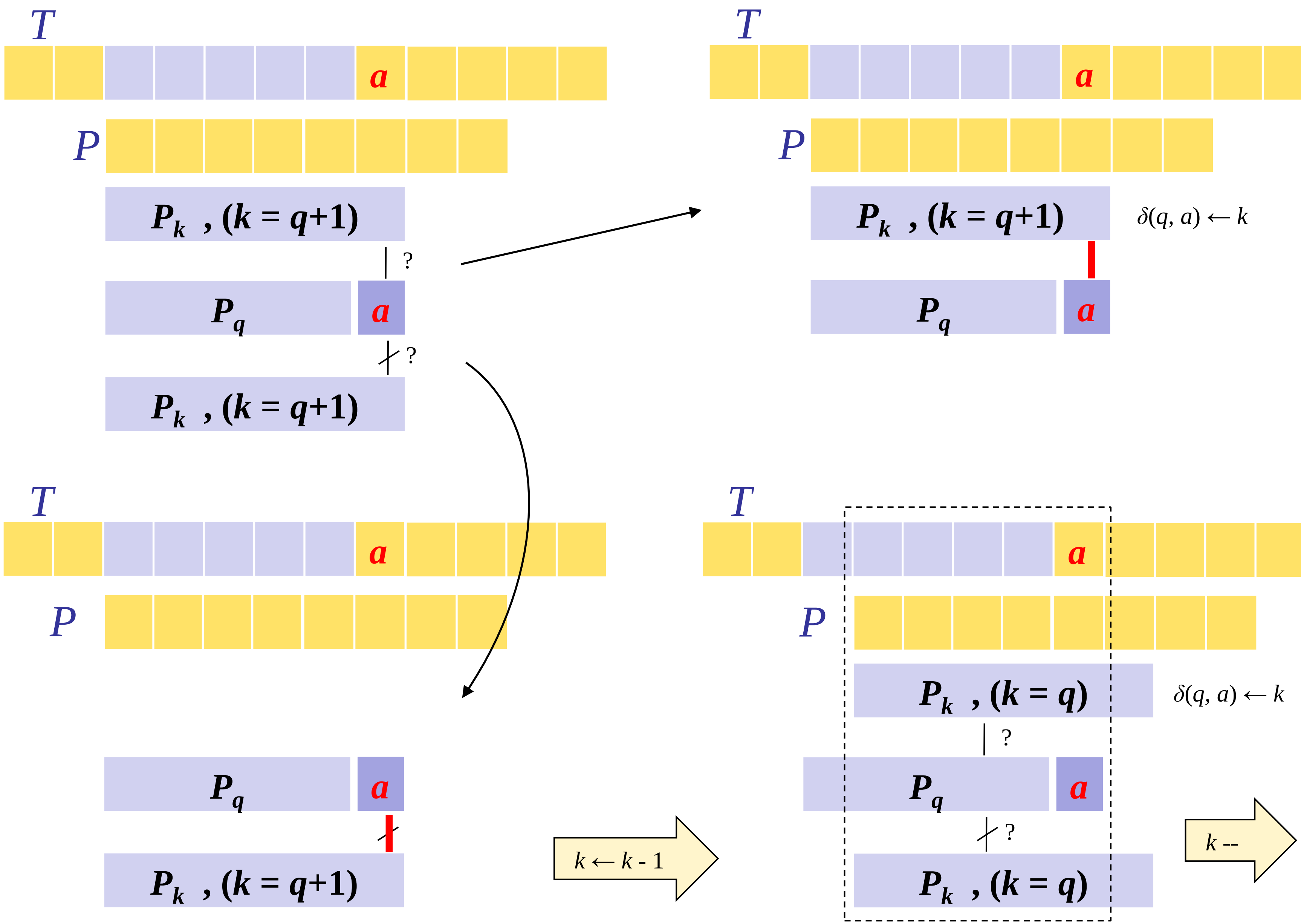
32.3 有限自动机的字符串匹配





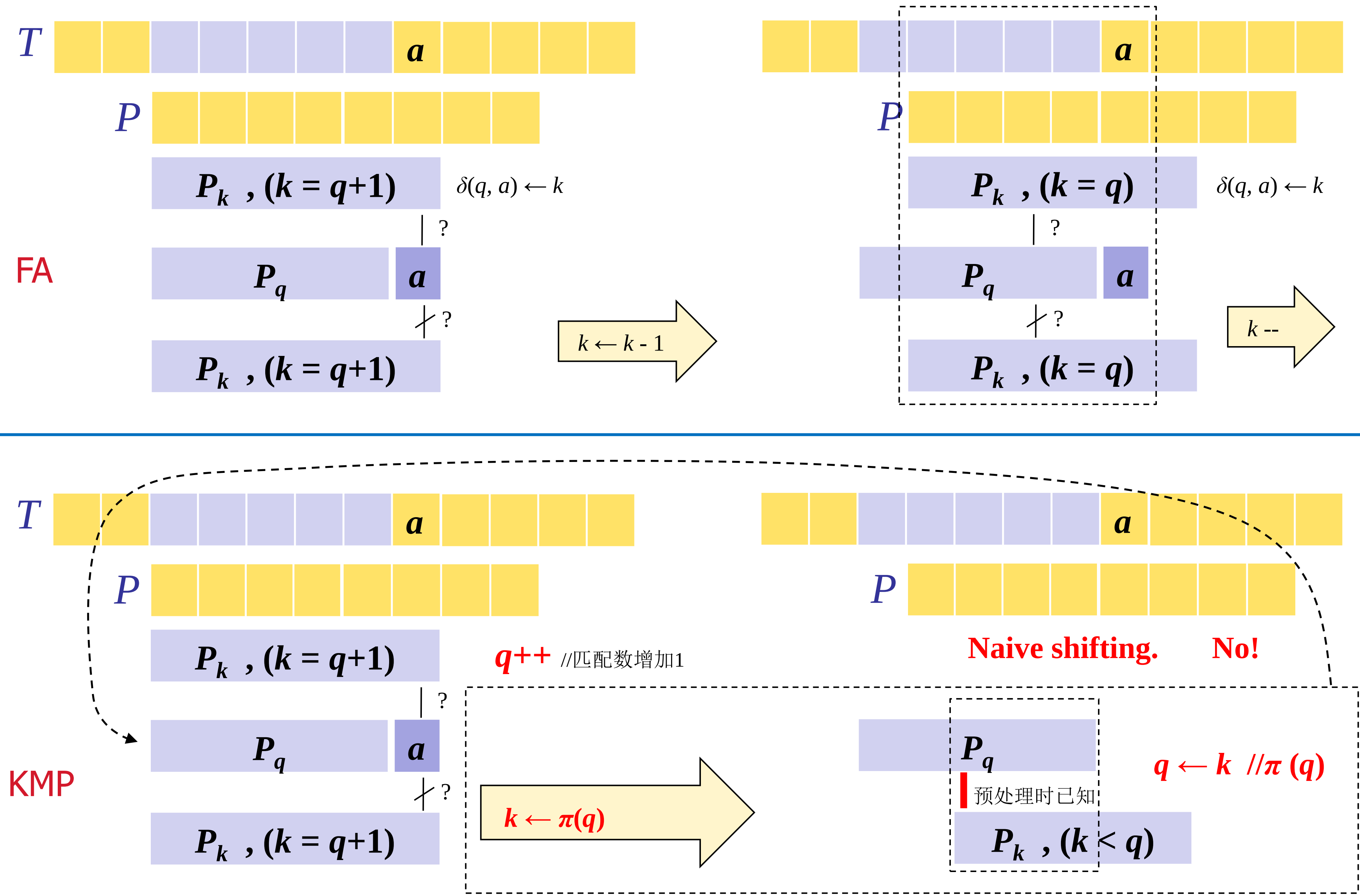


32.3 有限自动机的字符串匹配



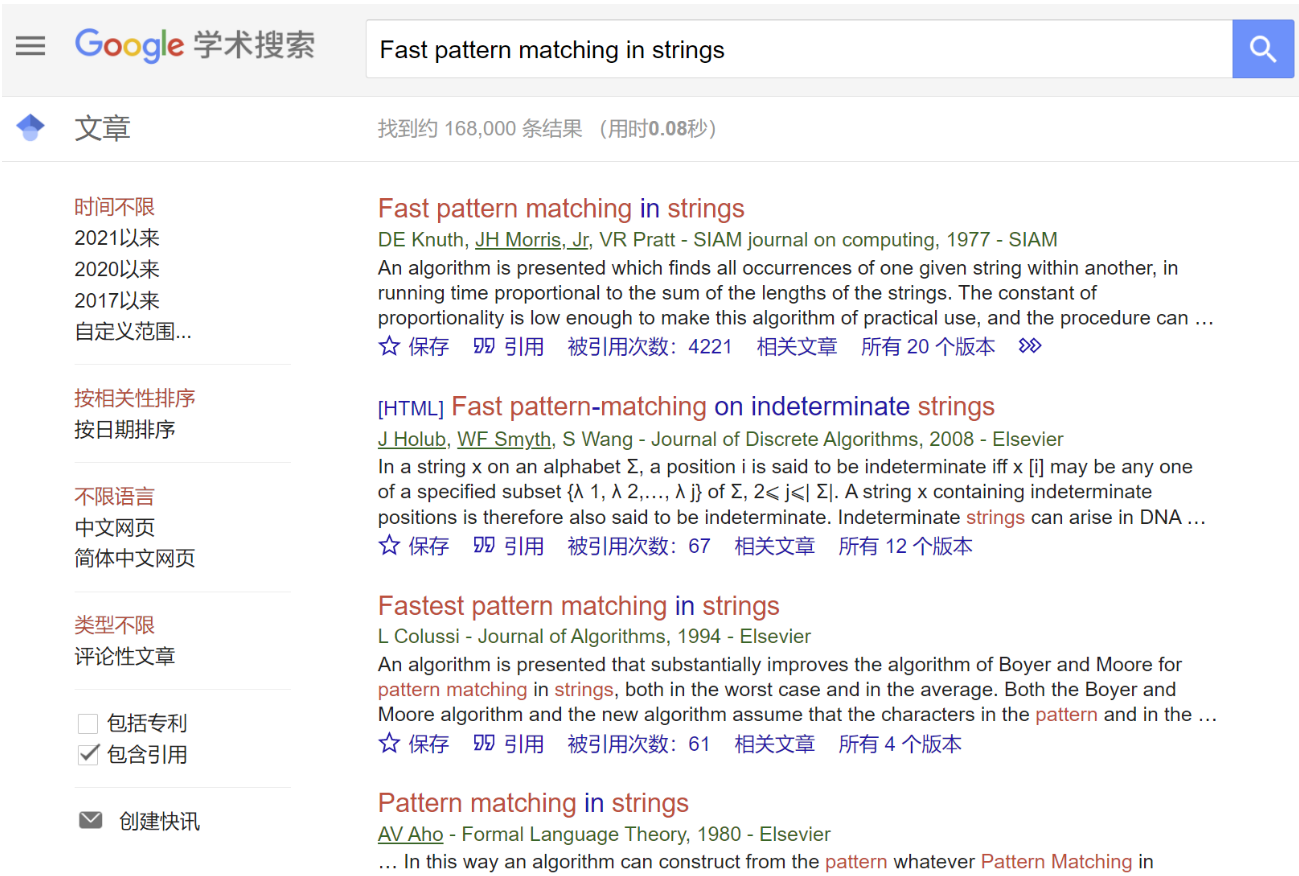
*32.4 The Knuth-Morris-Pratt algorithm

自动机构造完后，从δ已经知道输入a后P应右移多少（此时跟KMP是相似的思想，核心在于求δ有额外计算开销）

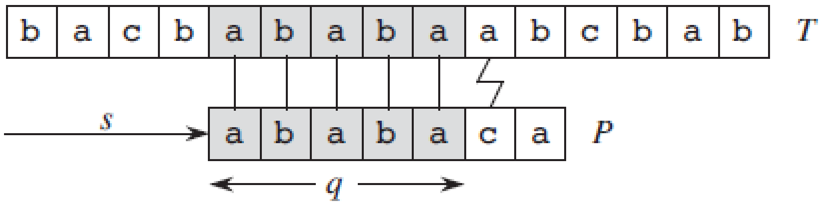


*32.4 The Knuth-Morris-Pratt algorithm

KMP 是由Knuth, Morris, Pratt 提出的一种线性时间的字符串匹配算法。



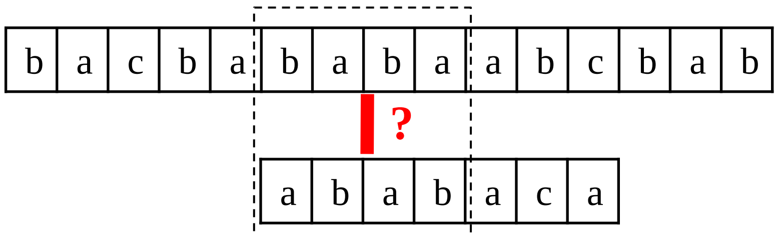
*32.4 The Knuth-Morris-Pratt algorithm



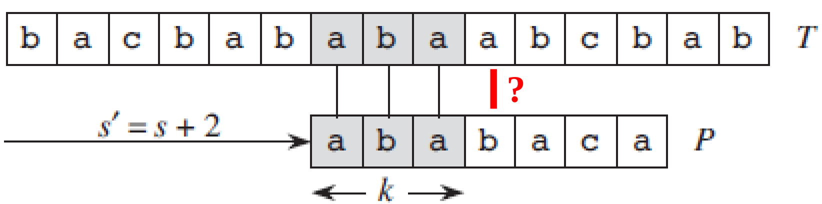
(a)

$P_5 \sqsupset T_{s+5}$, but,

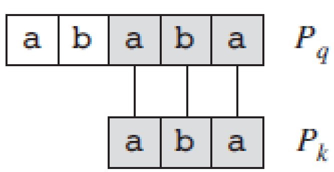
$T[s+5+1]$!= $P[5+1]$ ($q = 5$)



朴素移位, $P_4 \sqsupset T_{s+1+4}$? No!



(b)

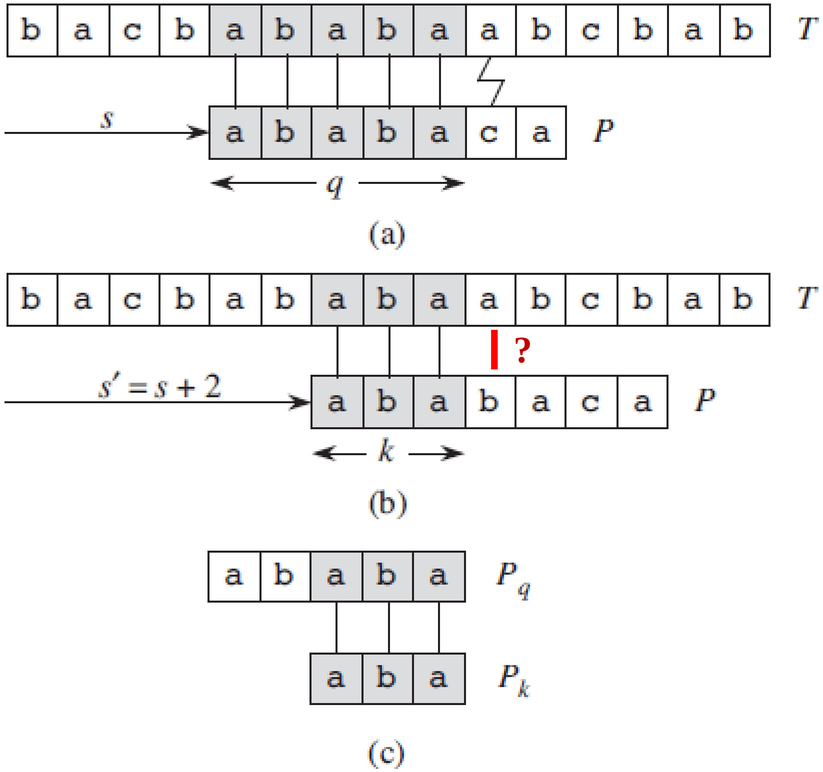


(c)

我们已经知道 P_k 是 P_q 的最大后缀, 即 $P_{k'} \sqsupset P_q$ ($k' < q$, $k = \max(k')$). 对这个例子来说, q 为 5, k 为 3。所以我们得到$P_3 \sqsupset P_5 \sqsupset T_{s+5}$, 我们只需检查是否..

$T[s+5+1]$!= $P[3+1]$ ($q \leftarrow k = 3$)

*32.4 The Knuth-Morris-Pratt algorithm



$P_5 \sqsupset T_{s+5}$, but,

T[s+5+1] != P[5+1] (q = 5)

我们已经知道 P_k 是 P_q 的最大后缀, 即 $P_{k'} \sqsupset P_q$ ($k' < q$, $k = \max(k')$). 对这个例子来说, q 为 5, k 为 3。所以我们得到 $P_3 \sqsupset P_5 \sqsupset T_{s+5}$, 我们只需检查是否 ..

T[s+5+1] != P[3+1] ($q \leftarrow k = 3$)

模式 P 的**前缀函数**为..

$\pi : \{1, 2, \ldots, m\} \rightarrow \{0, 1, \ldots, m\text{-}1\}$ 使

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

P_q 是 P 的前缀，
P_k 是 P_q 的前缀， 且
P_k 是 P_q 的后缀,
最大的 k 即为 $\pi[q]$.

*32.4 The Knuth-Morris-Pratt algorithm

模式 P 的**前缀函数**为..

$\pi : \{1, 2, \ldots, m\} \rightarrow \{0, 1, \ldots, m\text{-}1\}$ 使

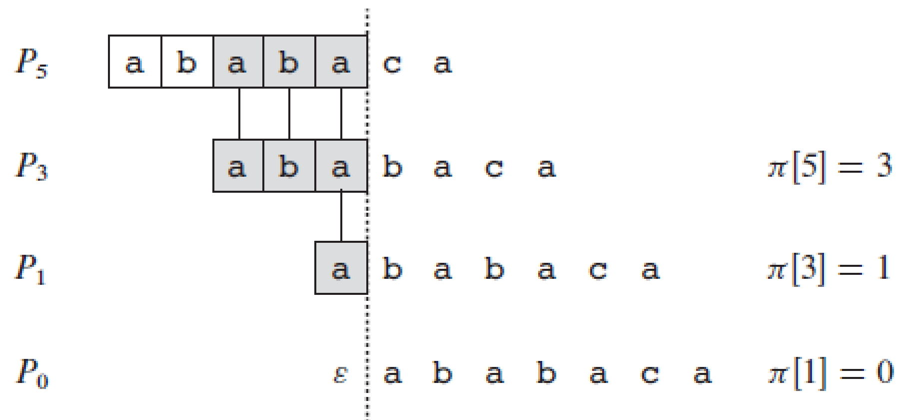
$\boldsymbol{\pi[q] = \max\{k : k < q \text{ and } P_k \sqsupset P_q\}}$.

P

a	b	a	b	a	c	a

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)

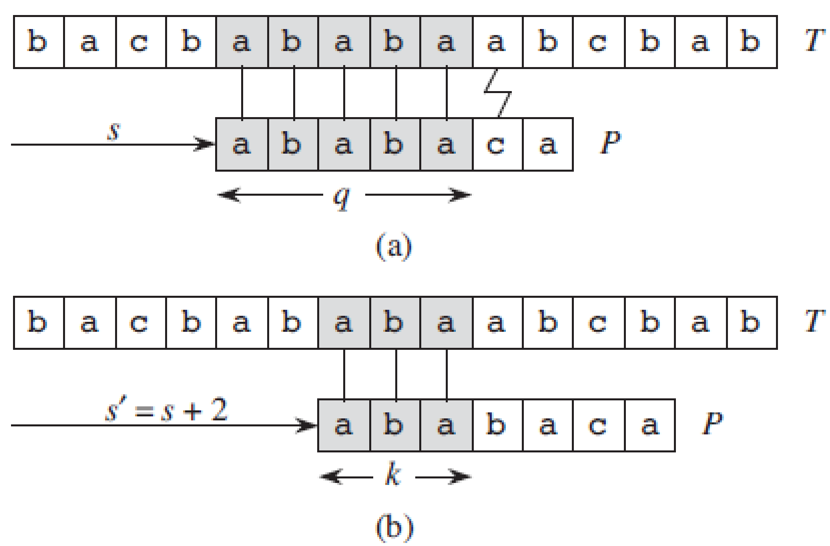
*32.4 The Knuth-Morris-Pratt algorithm

```
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
```

we have $P_q \sqsupset T_{i\text{-}1}$,

check whether $P[q+1] \neq T[i]$

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



*32.4 The Knuth-Morris-Pratt algorithm

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

```
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
```

we have $P_k \sqsupset P[q\text{-}1]$,
check if $P[k+1] \neq P[q]$

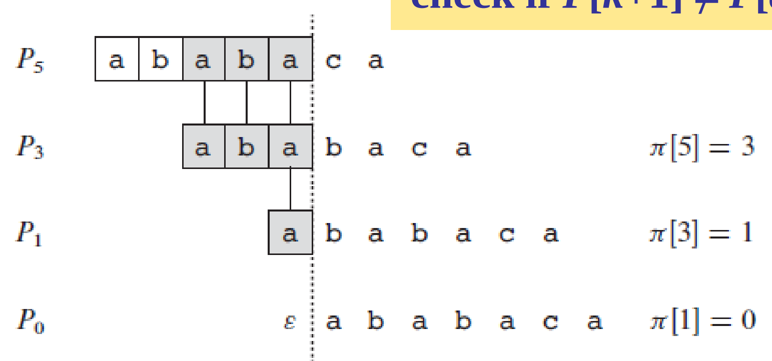
模式*P*的**前缀函数**为..

$\pi : \{1, 2, \ldots, m\} \rightarrow \{0, 1, \ldots, m\text{-}1\}$ 使

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

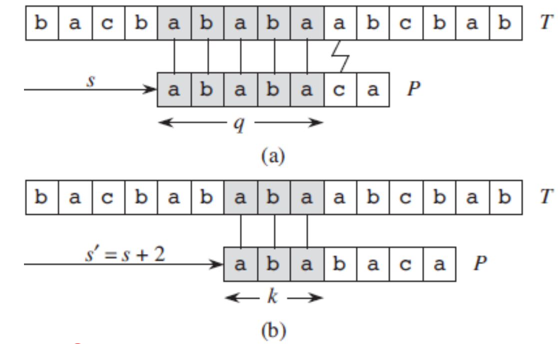
i	1	2	3	4	5	6	7
$P[i]$	a	b	a	b	a	c	a
$\pi[i]$	0	0	1	2	3	0	1

(a)



(b)

*32.4 KMP algorithm

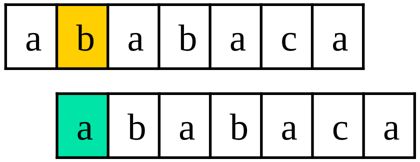



```
COMPUTE-PREFIX-FUNCTION(P)
m = P.length
let π[1..m] be a new array
π[1] = 0
k = 0
for q = 2 to m
    while k > 0 and P[k+1] ≠ P[q]
        k = π[k]
    if P[k+1] == P[q]
        k = k + 1
    π[q] = k
return π
```

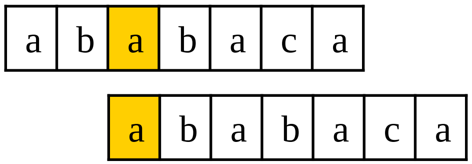
we have $P_k \sqsupset P[q\text{-}1]$,
check if $P[k+1] \neq P[q]$

q = 1: $k \leftarrow 0$, $\pi[1] \leftarrow 0$

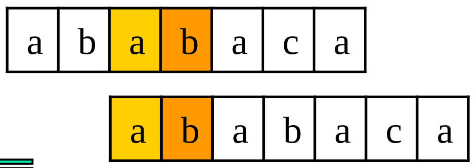
q = 2: $\pi[2] \leftarrow 0$



q = 3: $P[1] == P[3]$
$k \leftarrow 1$, $\pi[3] \leftarrow 1$



q = 4: $P[2] == P[4]$
$k \leftarrow 2$, $\pi[4] \leftarrow 2$



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

模式P的**前缀函数**为..

$\pi : \{1, 2, \ldots, m\} \rightarrow \{0, 1, \ldots, m\text{-}1\}$ 令

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

*32.4 KMP algorithm

accounting: q加1时2个分摊消费（1个用于实际消费，1个是信用），减少时0个分摊消费（数q上有q个信用，最多减少到0，因此信用足够支付减少）

- Running time?

  Amortized analysis (accounting): $\Theta(n)$, $\Theta(m)$

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

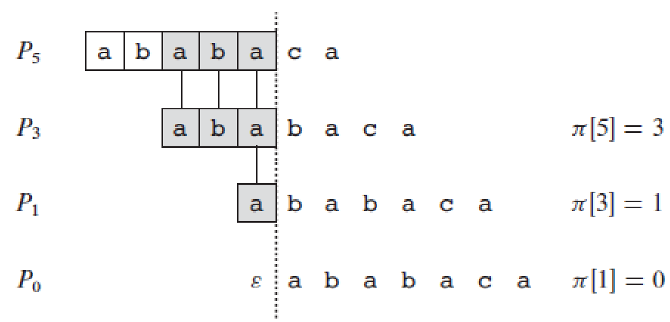
```
COMPUTE-PREFIX-FUNCTION(P)
m = P.length
let π[1..m] be a new array
π[1] = 0
k = 0
for q = 2 to m
    while k > 0 and P[k + 1] ≠ P[q]
        k = π[k]
    if P[k + 1] == P[q]
        k = k + 1
    π[q] = k
return π
```

前缀函数:

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

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)

*32.4 The Knuth-Morris-Pratt algorithm

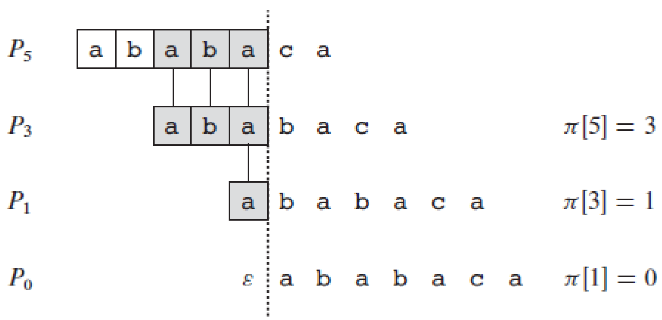
- KMP 算法避免了计算转移函数 δ, 匹配时间是 $\Theta(n)$,只使用了一个辅助函数 π, π由我们在 $\Theta(m)$时间内从模式中预先计算出然后储存在数组$\pi[1 .. m]$中。

前缀函数:

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

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)

Exercises

32.3-1

32.3-2

32.4-1

计算模式**ababbabbabbababbabb**的前缀函数 π