Chapter 32

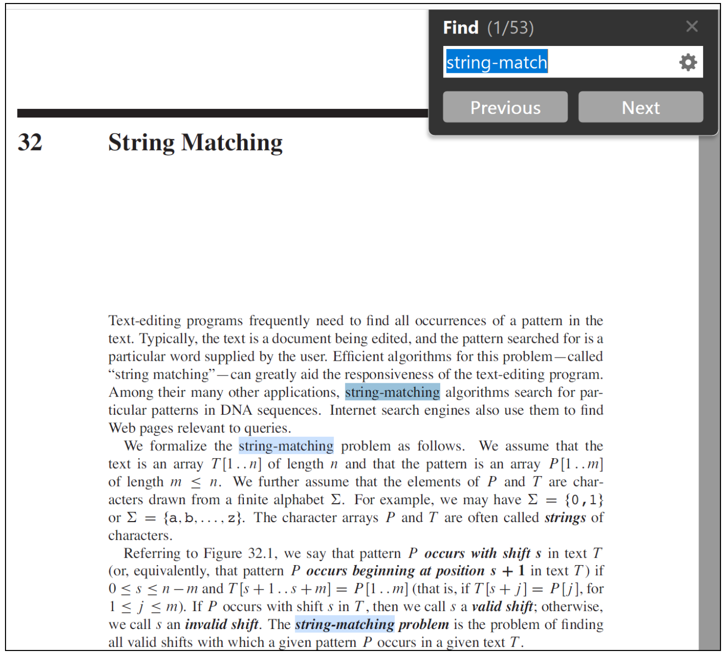
String Matching

songyou@buaa.edu.cn

VII Selected Topics

32 String Matching

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

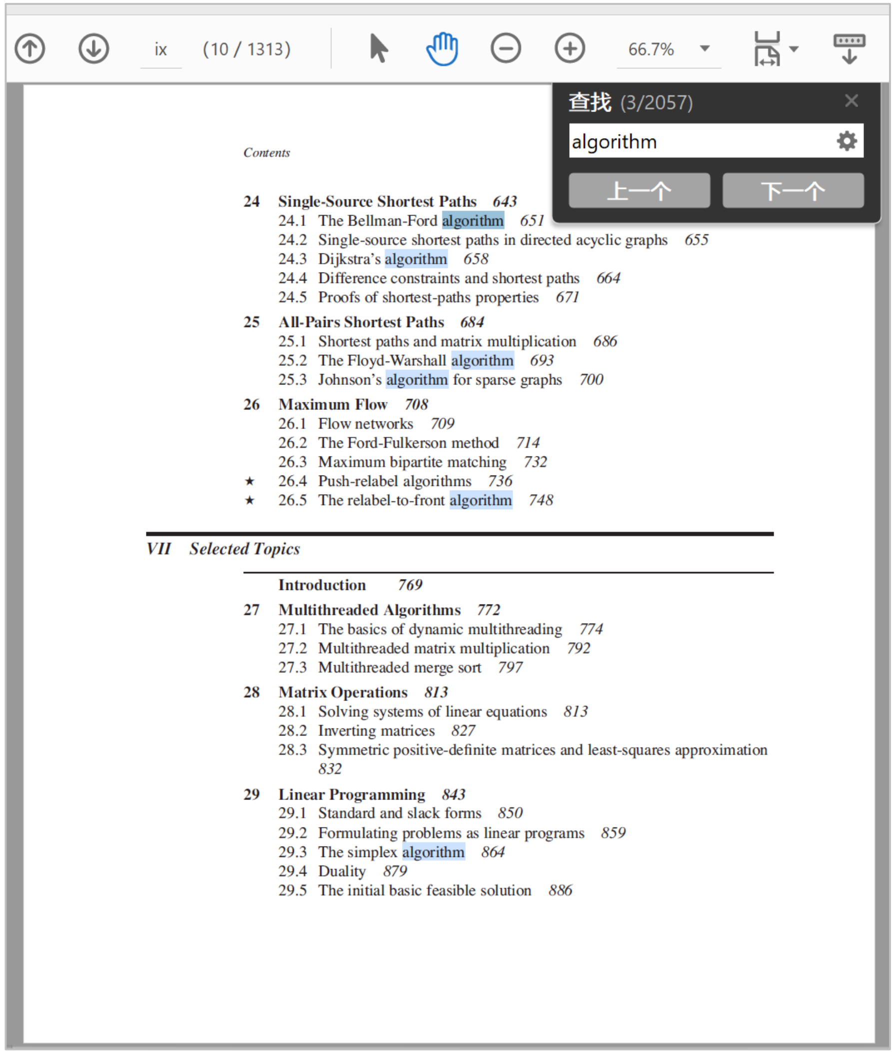


32 **String Matching**

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.



Contents

Finding all occurrences of a pattern in a text is a problem that arises frequently in text-editing programs.

在文本中搜索某个模版出现的所有位置

32 String Matching

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

Fast pattern matching in strings

找到约 270,000 条结果 （用时0.12秒）

Fast pattern matching in strings
DE Knuth, JH Morris, Jr, VR Pratt - SIAM journal on computing, 1977 - SIAM
… Finally, 8 discusses still more recent work on pattern matching. … Theidea behind this approach to pattern matching is perhaps easiest to grasp if we imagine placing the pattern over the …
☆ 保存 99 引用 被引用次数：4686 相关文章 所有 17 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 …
☆ 保存 99 引用 被引用次数：70 相关文章 所有 4 个版本

Pattern matching in strings
AV Aho - Formal Language Theory, 1980 - Elsevier
… match measured as a function of the lengths of p and x. We will assume the pattern is given before the input string … input string of length n, we can use a NDFA to do pattern matching in …
☆ 保存 99 引用 被引用次数：110 相关文章 所有 2 个版本

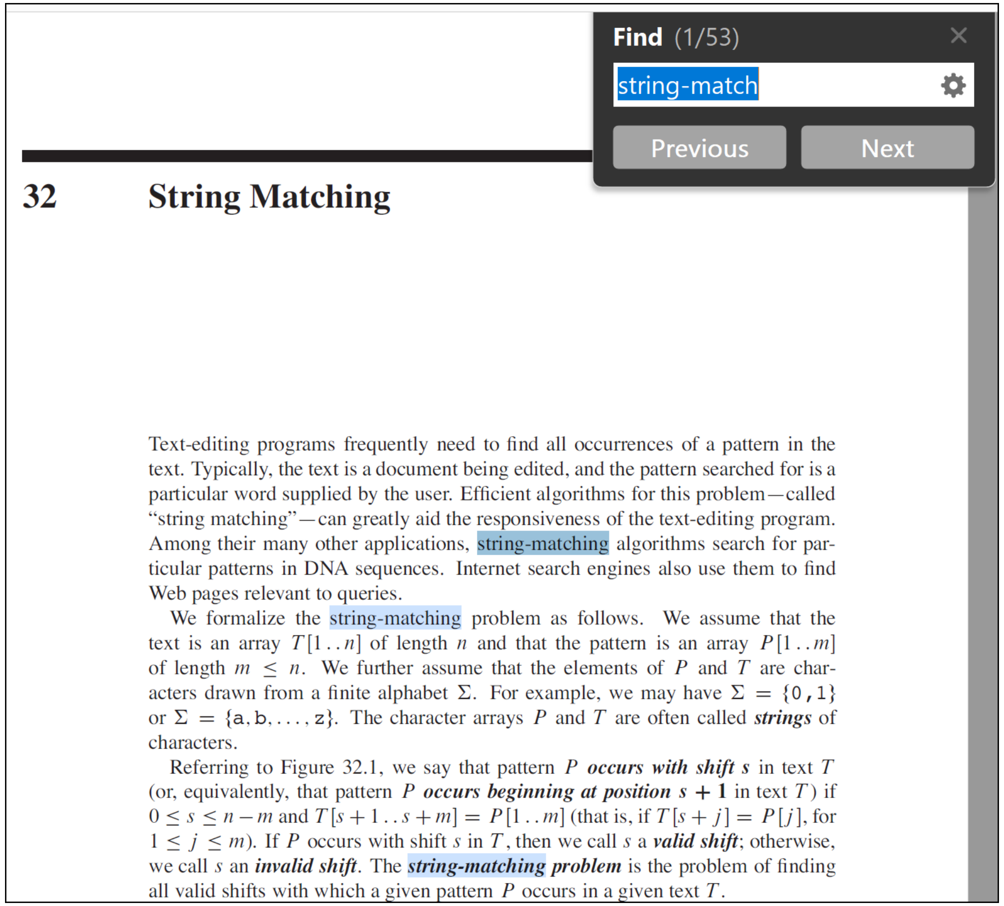
- 这篇文章提出了KMP算法。
- 在网络上搜索关键字（海量数据里匹配关键字，如查找这篇文章，本身就是一个字符串匹配过程）

32 String Matching

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

Applications:

- text-editing program
- search for particular patterns in DNA sequences
- …



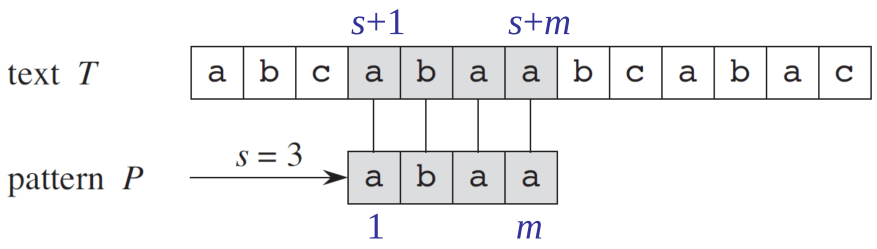
32 **String Matching**

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 String Matching



The pattern occurs only once in the text, at shift s = 3.

The shift s = 3 is said to be a valid shift.

String-matching problem:

- Text: $T[1 .. n]$, Pattern: $P[1 .. m]$, $m \leq n$.
- Finite alphabet: Σ , for example, $\Sigma = \{0, 1\}$ or $\Sigma = \{a, b, . . . , z\}$.
- $P_i \in \Sigma$, $T_i \in \Sigma$.
- P occurs with shift s (偏移量，转移，漂移，位移) in T if $0 \leq s \leq n-m$ and $T[s+1 .. s+m] = P[1 .. m]$ (that is, if $T[s+j] = P[j]$, for $1 \leq j \leq m$). (or, equivalently, that P occurs beginning at position s+1 in T).
- Valid shift s: if P occurs with shift s in T; otherwise, s is an invalid shift.
  有效偏移，P 在 T 中出现，偏移量为 s
- Finding all valid shifts with which a given P occurs in a given T .

Notation and terminology

- Σ^* : the set of all finite-length strings formed using characters from the alphabet Σ. 有限长度的字符串结合

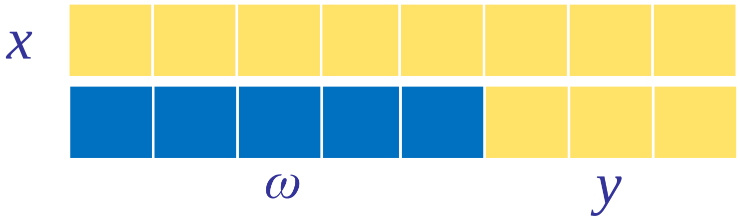
  Example:

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

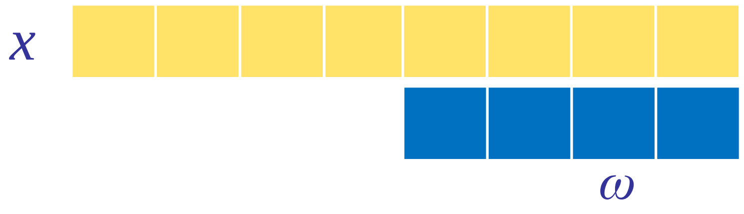
  $\Sigma^* = \{\varepsilon, a, b, c, ab, bc, ac, abc, acb, aabbc, \ldots\ldots\}$
- ε : The zero-length empty string, also belongs to Σ^*.
- $|x|$: The length of x.
- The concatenation of two strings x and y, denoted xy, has length $|x| + |y|$ and consists of the characters from x followed by the characters from y.

Notation and terminology

- $\omega \sqsubset x$: string ω is a prefix of x, if $x = \omega y$ for some $y \in \Sigma^*$.



prefix (ω 是 x 的前缀)



suffix

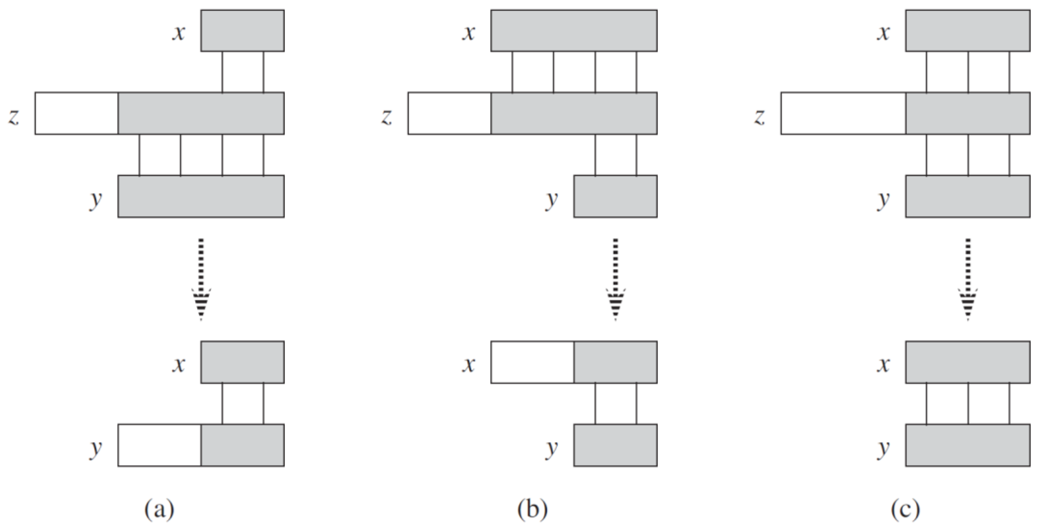
- $\omega \sqsupset x$: ω is a suffix of x, if $x = y\omega$ for some $y \in \Sigma^*$.
  - If $\omega \sqsubset x$ or $\omega \sqsupset x$, then $|\omega| \leq |x|$.
  - The empty string ε is both a suffix and a prefix of every string.
  - For example, we have ab $\sqsubset$ abcca and cca $\sqsupset$ abcca.
  - For any strings x and y and any character a, we have $x \sqsupset y$ if and only if $xa \sqsupset ya$.
  - $\sqsubset$ and $\sqsupset$ are transitive relations.

Notation and terminology

Lemma 32.1: (Overlapping-suffix lemma) 重叠后缀引理

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 $x = y$.

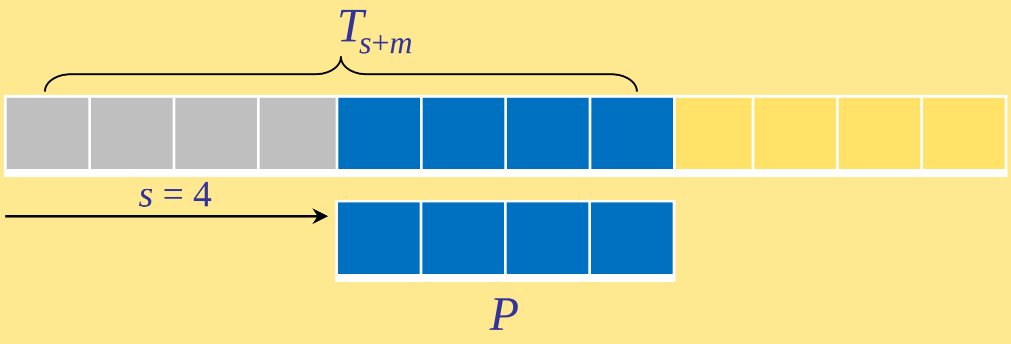
Proof See Fig for a graphical proof.



Notation and terminology

- For brevity of notation, we 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
- string-matching problem:
  finding all shifts s in the range $0 \le s \le n\text{-}m$ such that $P \sqsupset T_{s+m}$

P 是否为文本 T 的前缀 T_x 的后缀



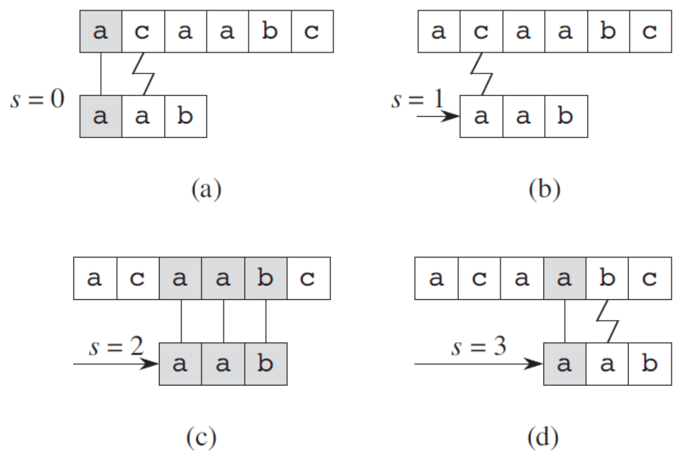
字符串匹配过程：从头到尾依序扫描文本 T，扫描到的字符串都是 T 的前缀 T_x，若有 P 匹配，则此时 P 为 T_x 的一个后缀。也就是，求文本 T 的前缀 T_x 的 P 后缀。

- Primitive operation: comparing characters

32.1 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      if P[1 .. m] = T[s+1 .. s+m]
5           print "Pattern occurs with shift" s
```



- The procedure can be interpreted graphically as sliding a “template” containing the pattern over the text. 在文本中滑动模版，滑动过程中比较是否匹配
- 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. 第4行包括一个隐式的循环

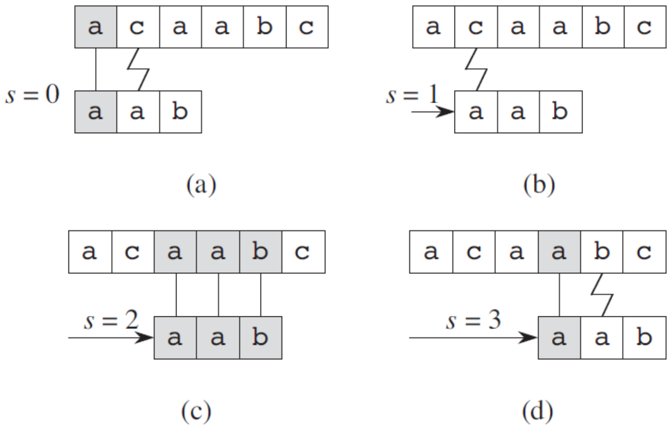
32.1 The naive string-matching algorithm

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

// 返回首次匹配位置，用库函数实现
strstr(T, P);

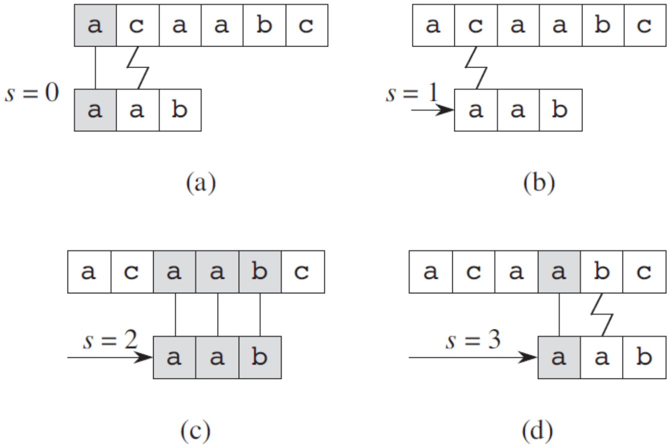
思考题：

- 所有匹配（按伪代码规则）都找出来，怎么实现（伪代码是子串有重叠的情况）？（若子串不重叠，即，若 T = aaaaa, P = aa，匹配偏移量为 0 和 2，怎么实现？）
- 怎么实现strrstr?（最后一次匹配的位置）




```
// 返回首次匹配位置，自定义函数实现
char * __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;
}
```

32.1 The naive string-matching algorithm



Running time ?

```
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
```

Exercise 32.1-2, 32.1-4

*32.2 The Rabin-Karp algorithm (拉宾-卡普 算法)

T

a	b	c	a	d	d	b	c
	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和简单数论的方法。对 T_{s+m} 计算 p 时，还有更有效的方法（后一个 T_{s+m} 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**.

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

coding rule (编码规则): {a, b, c, d} ⇨ {0, 1, 2, 3},

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

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

if $p(P) == p(T_{s+m})$ ($T_{s+m} = T[s+1 .. s+m]$, $s = 0, 1, ..$), then match.

or, if ($p(P) \bmod q$) == ($p(T_{s+m}) \bmod q$), *check if* $P == T_{s+m}$

扩展阅读 chapter31
Number-Theoretic Algorithms

preprocessing time: $\Theta(m)$
worst-case running time: $\Theta((n-m+1)m)$

*32.2 The Rabin-Karp algorithm

T

a	b	c	a	d	d	b	c

b	c	a	d

P

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

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

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

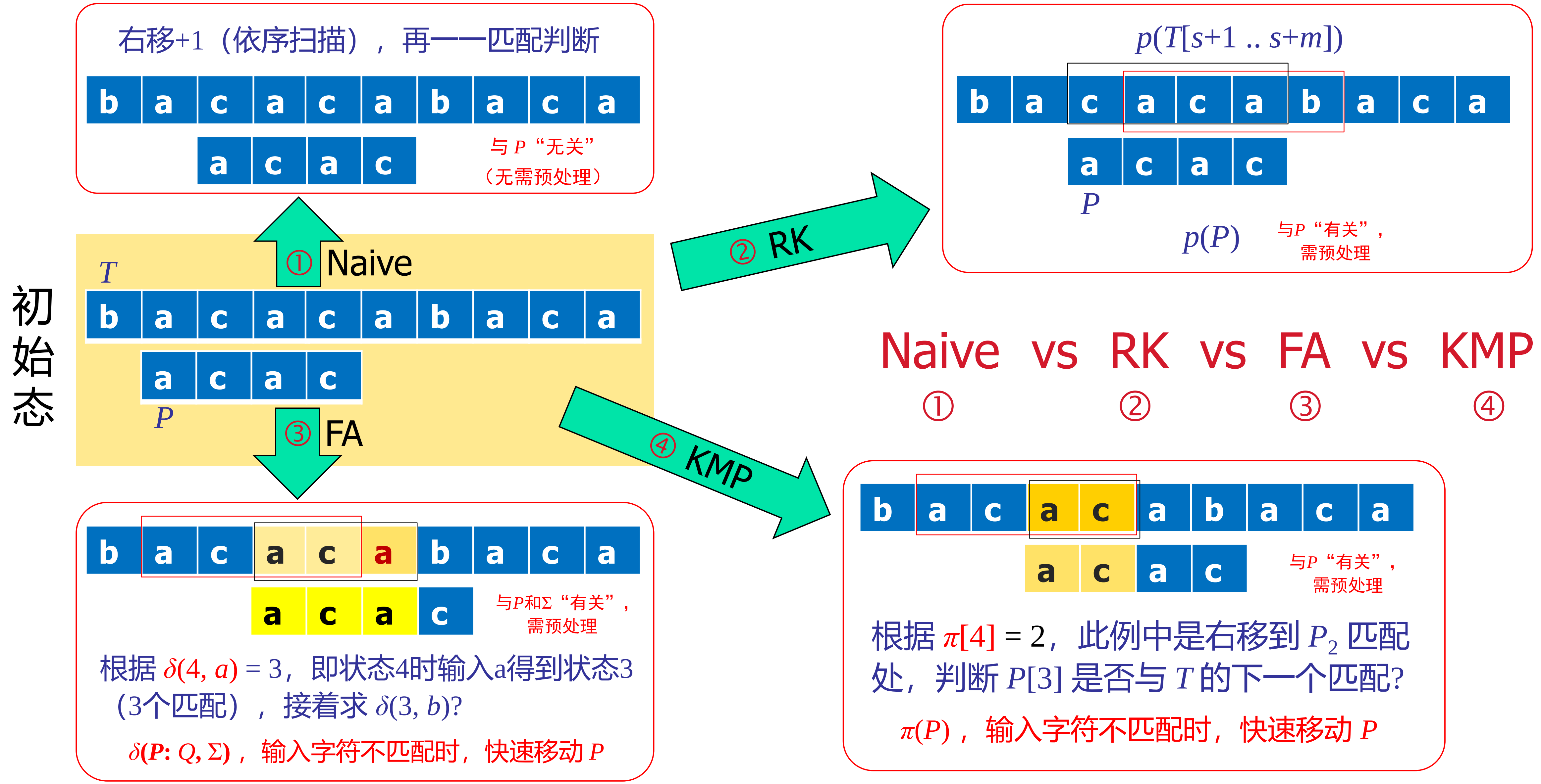
Efficient randomized pattern-matching algorithms

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 …

☆ 保存 ꟹ 引用 被引用次数：2034 相关文章 所有 7 个版本 »

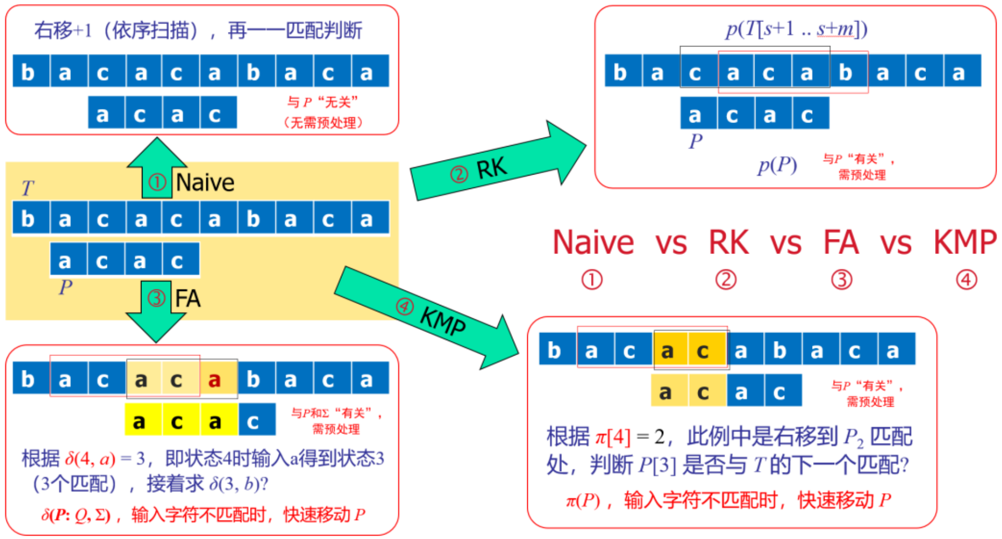
String Matching



String Matching: 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	$O(m\,\lvert\Sigma\rvert)$	$\Theta(n)$
Knuth-Morris-Pratt	$\Theta(m)$	$\Theta(n)$

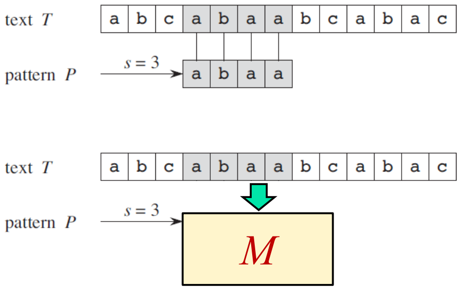
	关键	特征
FA	求 $\delta(P\!: Q, \Sigma)$	输入字符$T[i]$不匹配时，快速移动 P，每个$T[i]$ 匹配一次
KMP	求 $\pi(P)$	输入字符$T[i]$不匹配时，快速移动 P，每个$T[i]$ 可能匹配多次

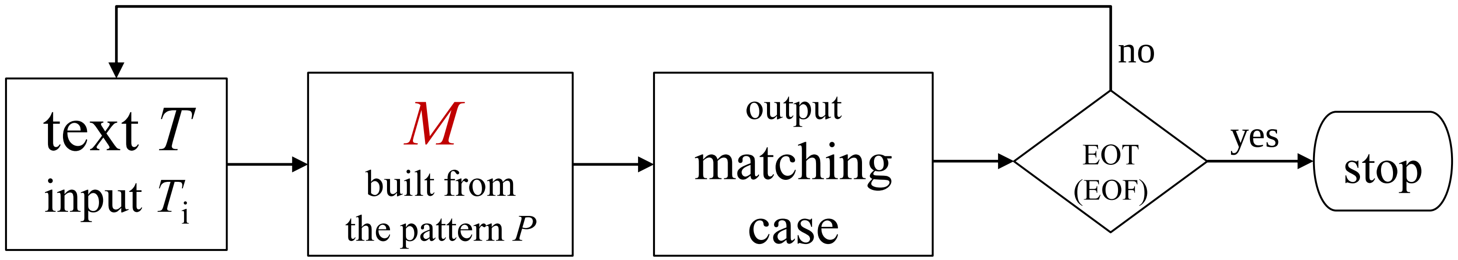


32.3 String matching with finite automata（有限自动机，有穷自动机）

[引用] The design and analysis of computer algorithms
AV Aho, JE Hopcroft - 1974 - Pearson Education India
☆ 保存 99 引用 被引用次数：15831 相关文章 所有 9 个版本 »

- Many string-matching algorithms build a finite automaton (Machine) that scans the text T for all occurrences of the pattern P.
- These string-matching automata are very efficient:
  - they examine each text character *exactly once* ;
  - taking constant time per text character.



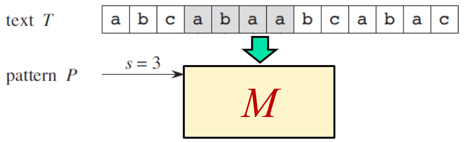


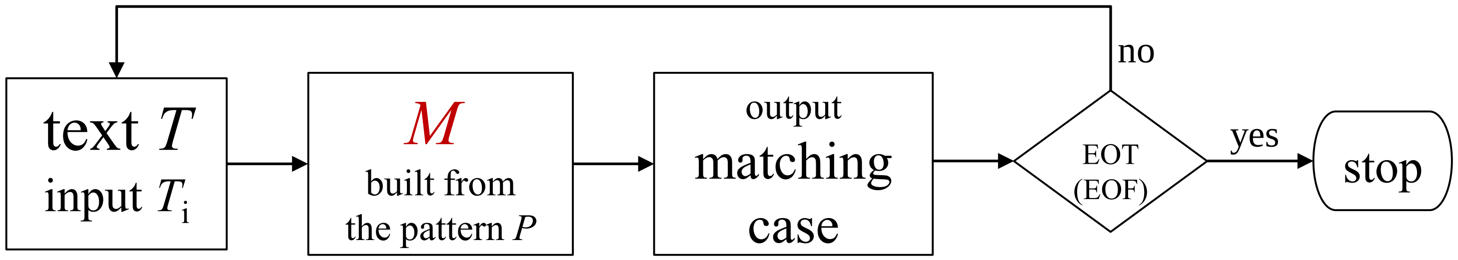
EOT: end of text

32.3 String matching with finite automata（有限自动机，有穷自动机）

The matching time is $\Theta(n)$.

- The preprocessing time (to build the automaton by pattern) can be large if Σ is large.（对西文文本来说，小写26，大写26，数字10，共62， 再加上各种标点符号、或特殊符号、或希腊字母等，Σ 约百余个字符，不算大；若中文，Σ 可以很大）
- Section 32.4 describes a clever way around this problem.

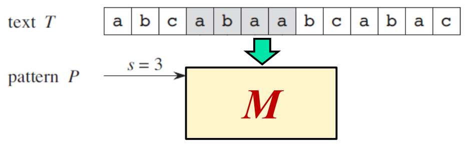




EOT: end of text

Finite automata (Machine)

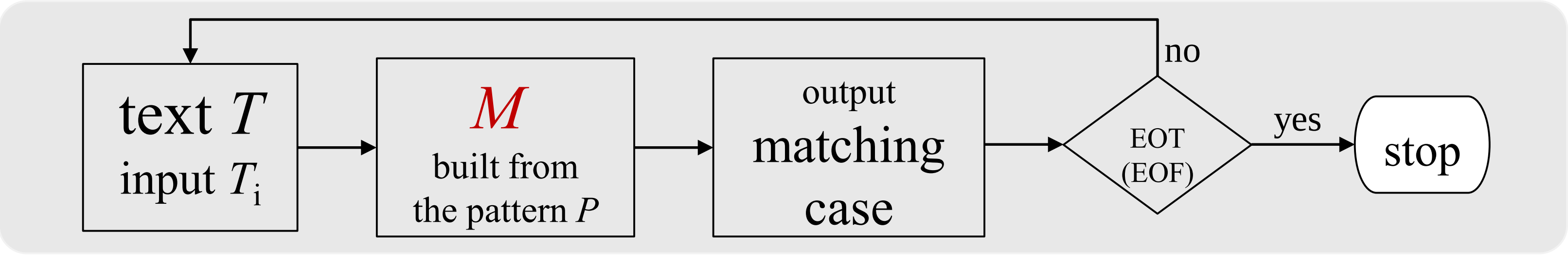
- **A *finite automaton M* is a 5-tuple** $M = (Q, q_0, A, \Sigma, \delta)$ **, where**
  - Q is a finite set of ***states***,
  - $q_0 \in Q$ is the ***start state***,
  - $A \subseteq Q$ is a distinguished set of ***accepting states***,
  - Σ is a finite ***input alphabet***,
  - δ is a function from $Q \times \Sigma$ into Q, called the ***transition function*** of M.



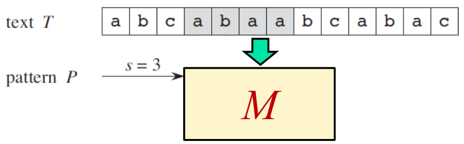
- The ***M*** begins in state q_0 and reads the characters of its input string one at a time. If the ***M*** is in state q and reads input a, it moves ("makes a transition") from state q to $\delta(q, a)$. Whenever its current state q is a member of A, the ***M*** is said to have ***accepted*** the string read so far. An input that is not accepted is said to be ***rejected***.

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

$$\delta(q_0 , a_1) \qquad\qquad \delta(q_i , a_{i+1})$$



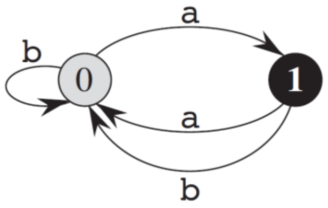
Finite automata: an example



- Figure 32.6: A simple two-state finite automaton with state set $Q = \{0, 1\}$, start state $q_0 = 0$, and input alphabet $\Sigma = \{a, b\}$.

state	input a	input b
0	1	0
1	0	0

(a)



(b)

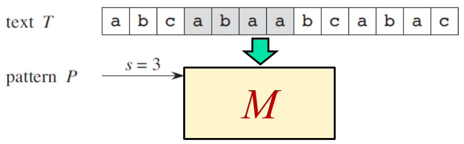
(a) A tabular representation of the transition function δ

(b) An equivalent state-transition diagram 状态转移图

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

- State 1 is the only accepting state (shown blackened). Directed edges represent transitions. For example, the edge from state 1 to 0 labeled b indicates $\delta(1, b) = 0$. This automaton accepts those strings that end in an odd number of a's. For example, the sequence of states this automaton enters for input abaaa (including the start state) is <0, 1, 0, 1, 0, 1>, so it accepts this input. For input abbaa, the sequence of states is <0, 1, 0, 0, 1, 0>, so it rejects this input.

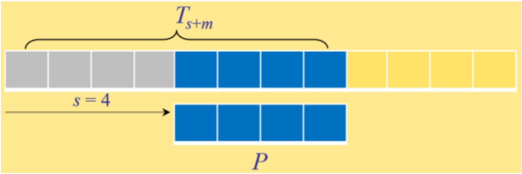
Finite automata: *final-state function*



- A finite automaton M induces a function Φ, called the *final-state function*, from Σ^* to Q such that $\Phi(\omega)$ is the state M ends up in after scanning the string ω.

  终态函数：状态机 M 诱导出的一个函数，M 读入字符串 ω 后得到状态$\Phi(\omega)$

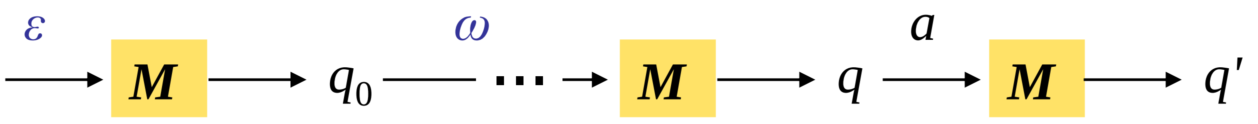
- Thus, M accepts a string ω if and only if $\Phi(\omega) \in A$.
- The function Φ is defined by the recursive relation



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

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

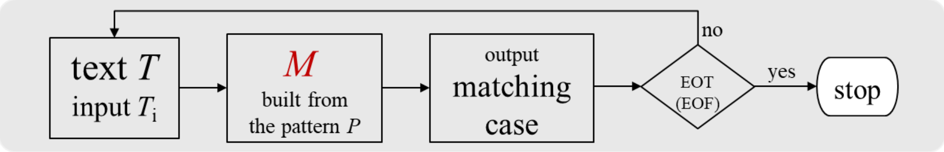
字符串匹配过程：M 从头到尾依序扫描文本 T，扫描到的字符串都是 T 的前缀 T_x，若有 P 匹配，则此时 P 为 T_x的一个后缀。也就是，求文本 T 的前缀的 P 后缀。



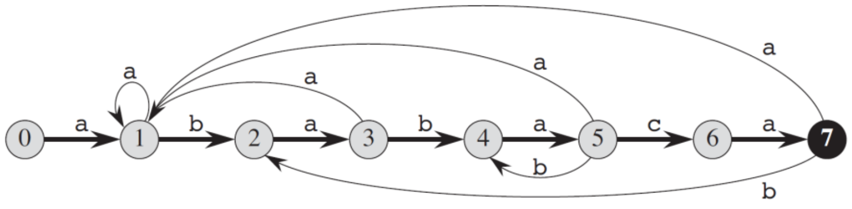
$q_0 = \Phi(\varepsilon)$ $\quad$ $q = \Phi(\omega)$ $\quad$ $q' = \Phi(\omega a) = \delta(\Phi(\omega), a) = \delta(q, a)$

终态函数的意义：扫描字符串 ω 后， ω 的后缀中，有多少个字符跟模版 P 的前缀 P_k 匹配

String-matching automata



- There is a string-matching automaton**(Machine)** 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 illustrates this construction for the pattern P = ababaca .



(a)

state	a	b	c	P
	input			
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)

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

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 ω

aba abaa

P : ababaca ababaca

$\delta(2, \text{a}) = 3$ $\delta(3, \text{a}) = 1$

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

含义：读入 ω 时状态（与 P 中字符匹配个数）为 q，再读入 a 时的状态是什么？

应用：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。

String-matching automata: an example

(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 α represents $\delta(i, \alpha) = j$. The right-going edges forming the “spine” of the automaton, shown heavy, correspond to successful matches between pattern and input characters.

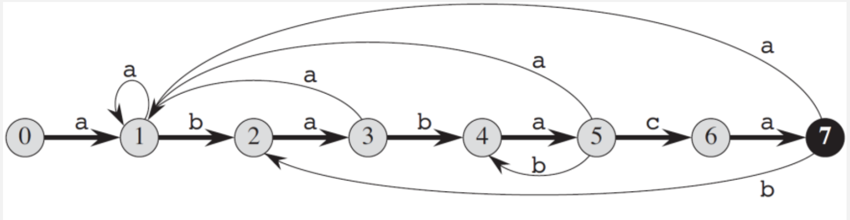
The left-going edges correspond to failing matches.
指向左边的边表示当前输入字符匹配失败（模版 P 需要右移，重新进行匹配分析）。

Some edges corresponding to failing matches are not shown; by convention, if a state i has no outgoing edge labeled α for some $\alpha \in \Sigma$, then $\delta(i, \alpha) = 0$.
对状态 i ，输入时没有输出边，表示当前输入字符串的后缀与P 零匹配，如状态 4 时，输入 b，有 $\delta(4, b) = 0$，省略该输出边（输入 c 同理）。

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

M *of* P = ababaca

(a diagram)



(a)

Figure 32.7

String-matching automata: an example

(b) The corresponding transition function δ (a table), and the pattern string P = ababaca. The entries corresponding to successful matches between pattern and input characters are shown shaded.
以表格形式表示转移函数，输入字符和模版成功匹配的情况以见图中阴影符号。

M of P = ababaca
(a table)

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) The operation of the automaton on the text T = abababacaba.

自动机 M 在文本 T 上的操作情况，处理 T_i 后，其状态为 $\Phi(T_i)$

One occurrence of the pattern is found, ending in position 9.

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)

Key: how to build $\delta(q, a)$?

String-matching automata: an example

The operation of the automaton on the text T = ababababacaba.

自动机 M 在文本 T 上的操作情况，处理 T_i 后，其状态为 $\Phi(T_i)$

One occurrence of the pattern is found, ending in position 9.

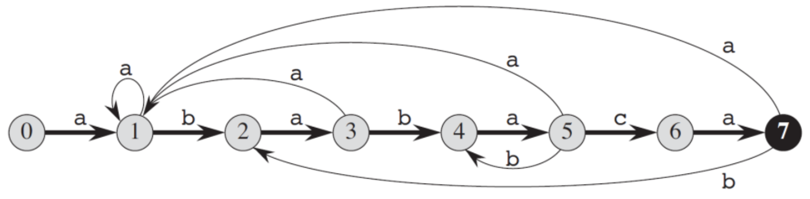
从字符串匹配的扫描过程来理解 δ

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

T a b a b a b a c a b a …

P a b a b a c a

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



(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)

String-matching automata: an example

The operation of the automaton on the text T = ababababacaba.

自动机处理 T_i 后，其状态为 $\Phi(T_i)$

$\delta(0, \text{a}) = 1$?

T a b a b a b a c a b a …

P a b a b a c a

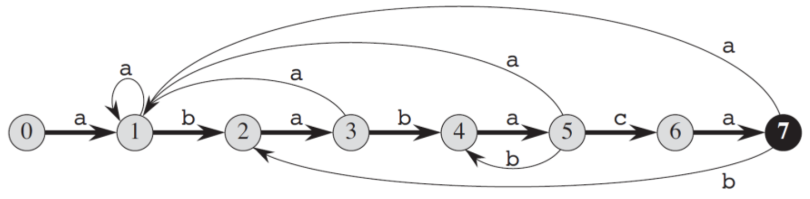
…

T 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 的第1个跟其匹配，即 $\delta(0, \text{a}) = 1$；【*从 T 一个一个地扫描，跟KMP有相似性】

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



(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)

String-matching automata: an example

The operation of the automaton on the text T = ababababacaba.

自动机处理 T_i 后，其状态为 $\Phi(T_i)$

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

T a b a b a b a c a b a …

P a b a b a c a

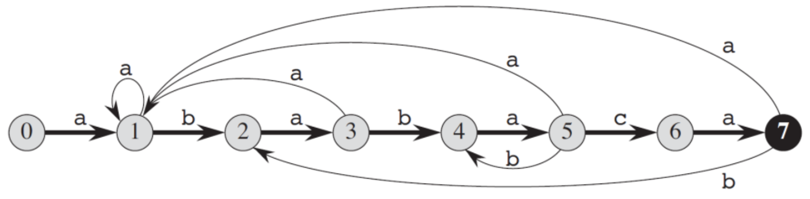
a b a …

T 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 $\delta(3, \text{b}) = 4$

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



(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)

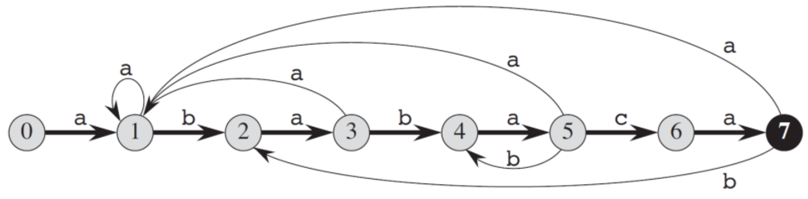
i	—	1	2	3	4	5	6	7	8	9	10	11
$T[i]$	—	a	b	a	b	a	b	a	c	a	b	a
state $\phi(T_i)$	0	1	2	3	4	5	4	5	6	7	2	3

(c)

String-matching automata: an example

The operation of the automaton on the text T = ababababacaba.

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



(a)

自动机处理 T_i 后，其状态为 $\Phi(T_i)$

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)

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

T a b a b a b a c a b a …

P a b a b a c a

a b a b …

T a b a b c …

P a b a b a

a b a b

a b a

… …

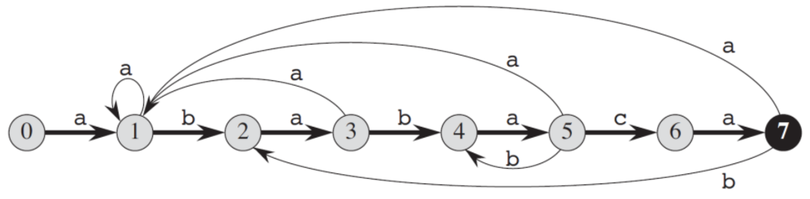
Naive: 状态4时（有4个匹配），输入c，即文本串 T 为ababc…时，P 的前5个跟其不匹配，即$\delta(4, c) != 5$；把 P 按字符右移1位（寻找新的可能匹配），P 的前4个跟其不匹配，即$\delta(4, c) != 4$；把 P 按字符右移2位，P 的前3个跟其不匹配，即$\delta(4, c) != 3$；以此类推。

String-matching automata: an example

The operation of the automaton on the text T = ababababacaba.

自动机处理 T_i 后，其状态为 $\Phi(T_i)$

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



(a)

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

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 …

T a b a b a b …

P a b a b a c

a b a b a

a b a b

这样单步移动不需要

快速移动到这里

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?

肯定不用这样做!

快速右移到 $\delta(q, x)$ 处!

求 $\delta(q, x)$ 是关键!

	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)

String-matching automata: ***suffix function***

- Suffix function σ corresponding to P :
  A mapping from Σ^* to $\{0, 1, \ldots, m\}$ such that $\sigma(x)$ is the length of the longest prefix of P that is a suffix of x （后缀函数西格玛 σ： 字符串 x 的后缀，且是 P 的最长前缀的长度）

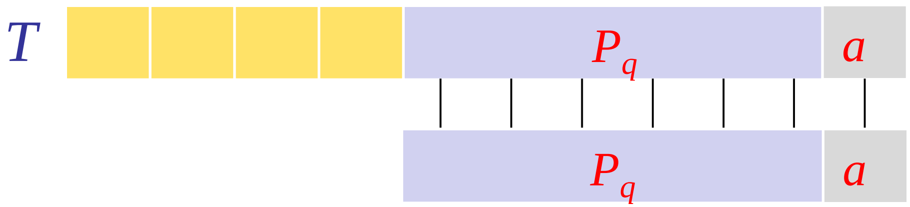
  $$\sigma(x) = \max \{k : P_k \sqsupset x\}.$$

- The suffix function σ is well defined since the empty string $P_0 = \varepsilon$ is a suffix of every string. As examples, （任何字符串，针对模版 P ， 都存在后缀函数）
  - for the pattern P = ab, we have $\sigma(\varepsilon) = 0$, $\sigma(\text{ccaca}) = 1$, and $\sigma(\text{ccab}) = 2$.
- For a pattern P of length m, we have $\sigma(x) = m$ if and only if $P \sqsupset x$.
- From the definition of the suffix function, if $x \sqsupset y$, then $\sigma(x) \leq \sigma(y)$.

String-matching automata

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

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



We define the ***string-matching automaton*** that corresponds to a given pattern $P[1 .. m]$ as follows. （模版 P 的字符串匹配自动机定义如下）

- The transition function δ is defined by the following equation, for any state q and character a: （状态转移函数 δ 定义为后缀函数，如下）

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

- where, the state set Q is $\{0, 1, . . . , m\}$, the start state q_0 is state 0, and state m is the only accepting state 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(**Lemma 32.1**)。 $P_q a$ 的长度比 $T_i a$ 短，处理起来（求δ）就简单得多。换一个角度，自动机跟模版 P 和输入字母表 Σ 相关，即，字母表 Σ 和一个 P 可构造一个自动机。

String-matching automata

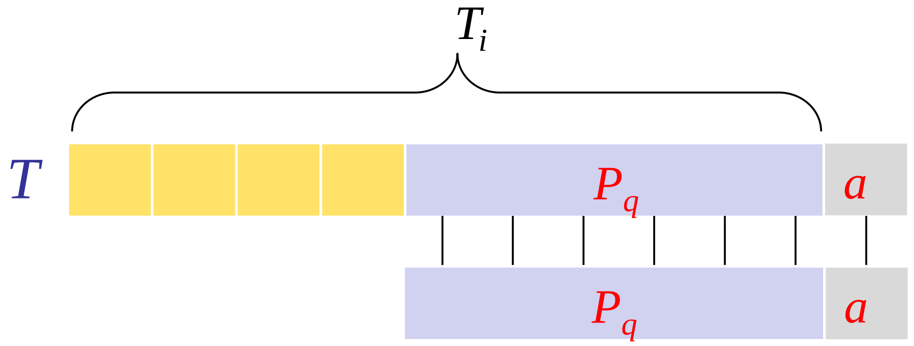
$M = (Q, q_0, A, \Sigma, \delta)$

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

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

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

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



Intuitively, the machine $\boldsymbol{M}$ maintains an invariant:

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

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$\sigma(T_{i+1})$）

扫描过程　从模版 P 着手　匹配情况（状态）

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

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

String-matching automata

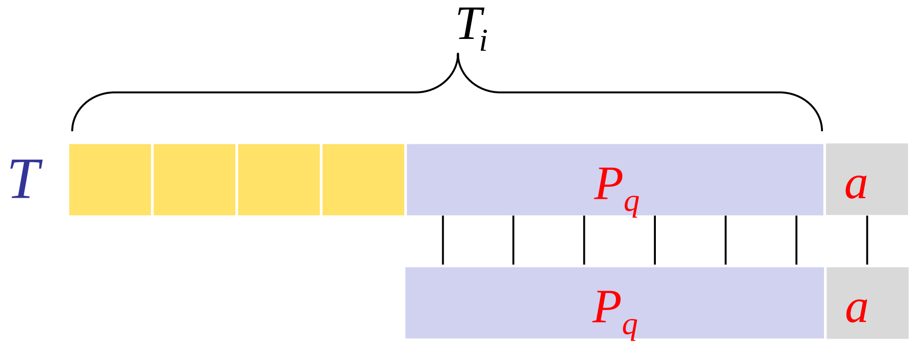
$M = (Q, q_0, A, \Sigma, \delta)$

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

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

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

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



- $\Phi(T_{i+1}) = \Phi(T_i a) = \delta(\Phi(T_i), a) = \delta(q, a) = \sigma(P_q a) \stackrel{?}{=} \sigma(T_i a) = \sigma(T_{i+1})$ (32.4)
  [(32.3) maintains the invariant, or, it is rationale for defining (32.3).]

- *Lemma 32.3*: $\sigma(T_i a) = \sigma(P_q a)$ (32.A)

  this lemma means definition (32.3) maintains the desired invariant (32.4).

- Compute: With (32.A), to compute $\sigma(T_i a)$, we can compute $\sigma(P_q a)$.

String-matching automata

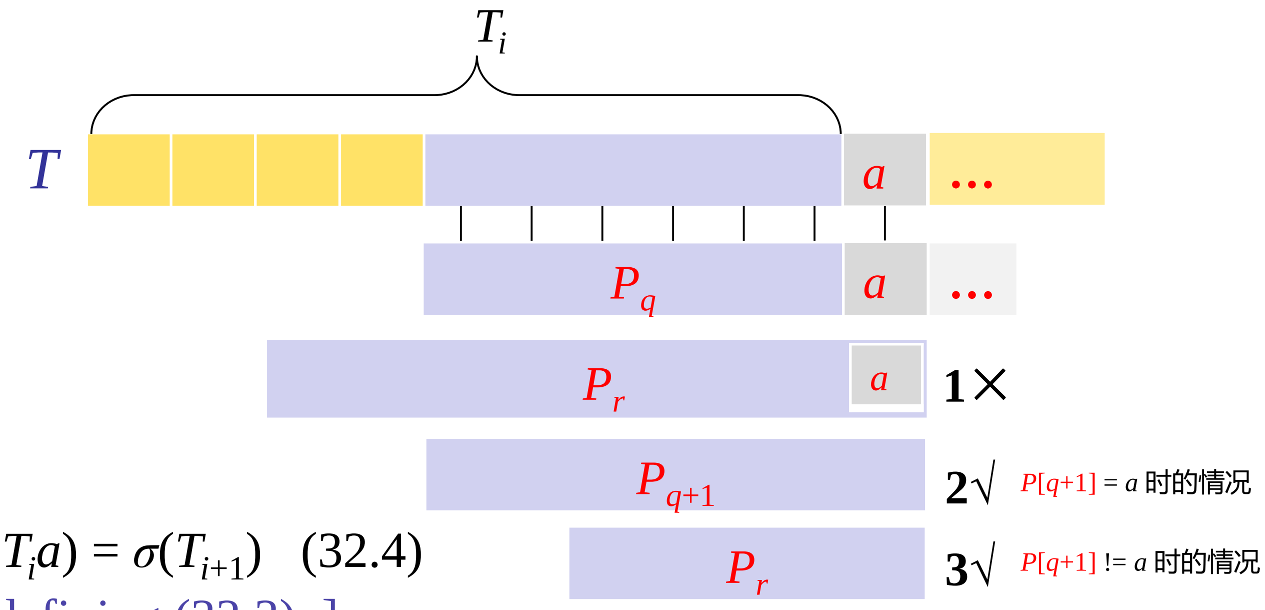
$M = (Q, q_0, A, \Sigma, \delta)$

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

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

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

$\boxed{\delta(q, a) = \sigma(P_q a) \qquad (32.3)}$



- $\Phi(T_{i+1}) = \Phi(T_i a) = \delta(\Phi(T_i), a) = \delta(q, a) = \sigma(P_q a) = \sigma(T_i a) = \sigma(T_{i+1})$ (32.4)
  [(32.3) maintains the invariant, or, it is rationale for defining (32.3).]
- *Lemma 32.3*: If $\sigma(T_i) = \sigma(P_q) = q$, then $\boxed{\sigma(T_i a) = \sigma(P_q a)\ . \qquad (32.A)}$
  Proof
  Situation 1 is impossible（$\sigma(T_i a) > q+1$不可能）. if 1 满足， $\sigma(T_i) > q$, 与假设矛盾。
  Apparently, if $P[q+1] = a$, it is situation 2, $\sigma(T_i a) = \sigma(P_{q+1}) = q+1$; else , situation 3.
- *32.3* and *32.A* show the automaton is in state $\sigma(T_i)$ 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.

String-matching automata

For example, in the string-matching automaton of Figure 32.7 (P = ababaca), $\delta(5, \text{b}) = 4$.

We make this transition because if the automaton reads a b in state $q = 5$, then $P_q\text{b}$ = ababab, and then, $\delta(5, \text{b}) = \sigma(\text{ababab}) = 4$.

P_5	a b a b a
P_5b	a b a b a b
P_4	a b a b
P	a b a b a c a

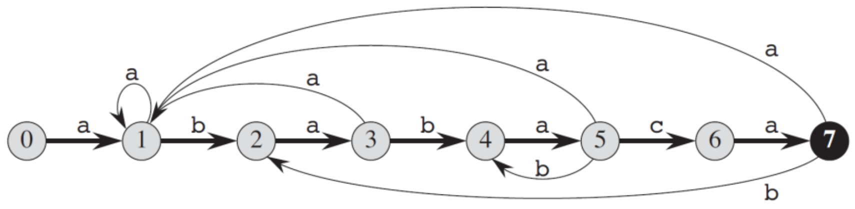
$\sigma(x) = \max \{k : P_k \sqsupset x\}$.

We define the *machine* M :

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

$q_0 = 0$; $A = \{m\}$; Σ;

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

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)

String-matching automata: ***program***

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

A string-matching automaton

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

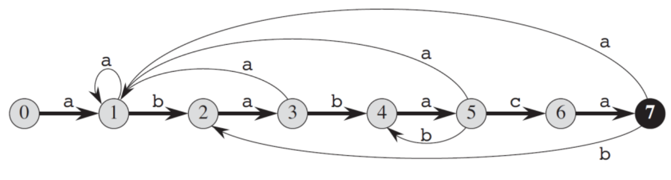
$Q = \{0, 1, \ldots, m\};\ \ q_0 = 0;\ \ A = \{m\};\ \Sigma;$

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

FINITE-AUTOMATON-MATCHER(*T*, *δ*, *m*)

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

状态 q 时，输入 a，通过查转移函数表可知新状态为 $\delta(q, a)$，将新状态赋值给 q，如果 $q == m$，即有 m 个字符匹配，输出一个匹配位置。



(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)

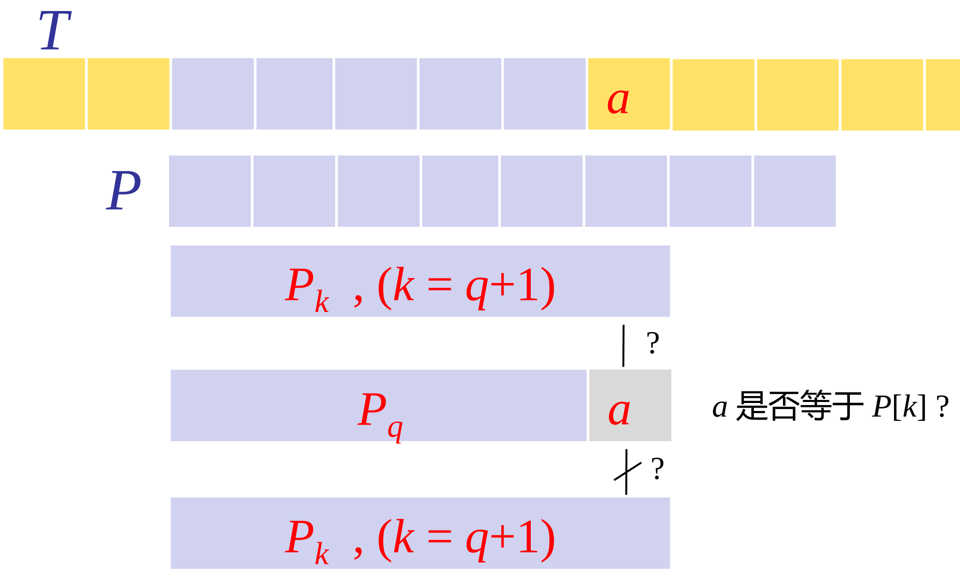
i	—	1	2	3	4	5	6	7	8	9	10	11
$T[i]$	—	a	b	a	b	a	b	a	c	a	b	a
state $\phi(T_i)$	0	1	2	3	4	5	4	5	6	7	2	3

(c)

Running time ?

- The matching time is $\Theta(n)$.
- However, it does not include the preprocessing time required to compute the transition function δ.

Computing the transition function



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

A string-matching automaton

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

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

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

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

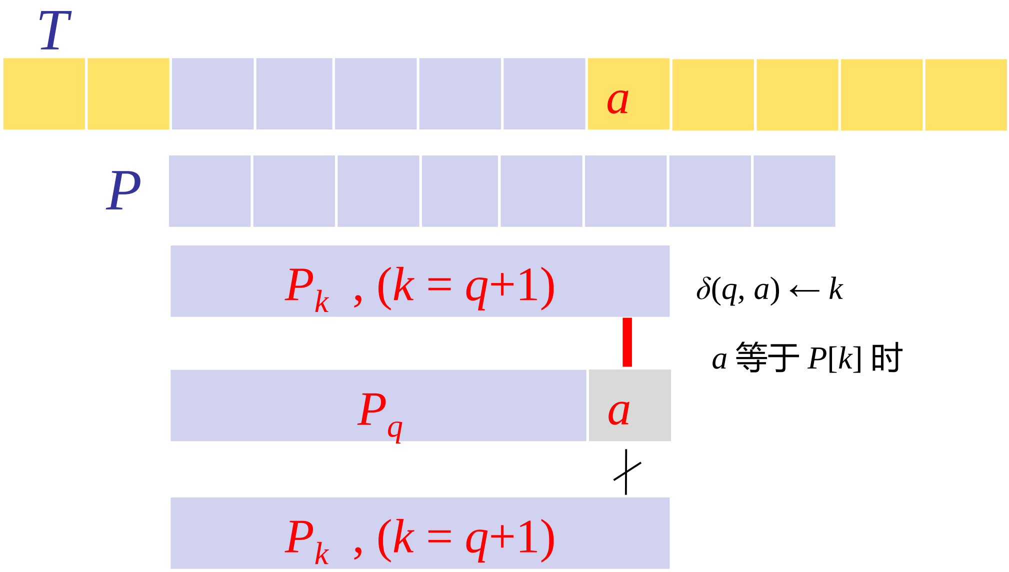
```
COMPUTE-TRANSITION-FUNCTION(P, Σ)
m ← length[P]
for q ← 0 to m
    for each character a ∈ Σ
        k ← min(m, q + 1)  // P 的最大长度为 m
        while P_k !⊐ P_q a
            k--
        δ(q, a) ← k
return δ
```

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 a 时：

1. $k = q+1$ （超过 m 时，即已匹配，即 q=m，取 k=m，因匹配数不大于 m）

2. $P_k \sqsupset P_q a$ ？

Computing the transition function



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

A string-matching automaton

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

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

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

Computing δ from a given pattern $P[1 \mathinner{..} m]$:

```
COMPUTE-TRANSITION-FUNCTION(P, Σ)
m ← length[P]
for q ← 0 to m
    for each character a ∈ Σ
        k ← min(m, q + 1)  // 模版长度为 m，因此匹配数最多为 m
        while P_k !⊐ P_q a
            k--
        δ(q, a) ← k
return δ
```

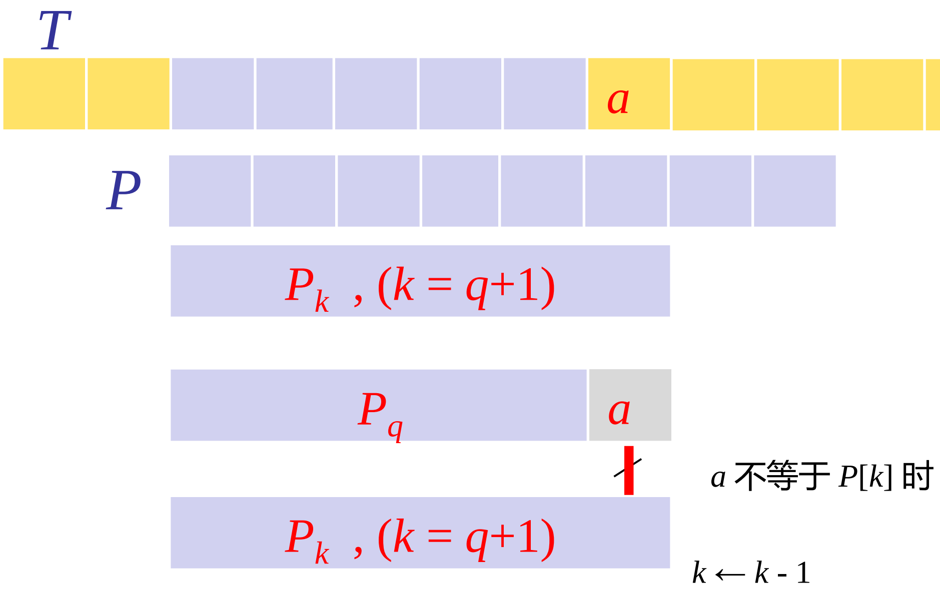
P_q 时（即 T 与 P的前 q 个字符匹配时），输入第 q+1（即第 k 个）字符 a 时：

1. $k = q$+1 （超过 m 时，取 m，匹配数不大于 m）

2. $P_k \sqsupset P_q a$ ？

3. 若2成立，则 $P_q a == P_k$ ，即，对在 T 的继续扫描过程中，若扫描的下一个字符 $a == P[k]$，则匹配字符增加 1（或继续为 m）

Computing the transition function



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

A string-matching automaton

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

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

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

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

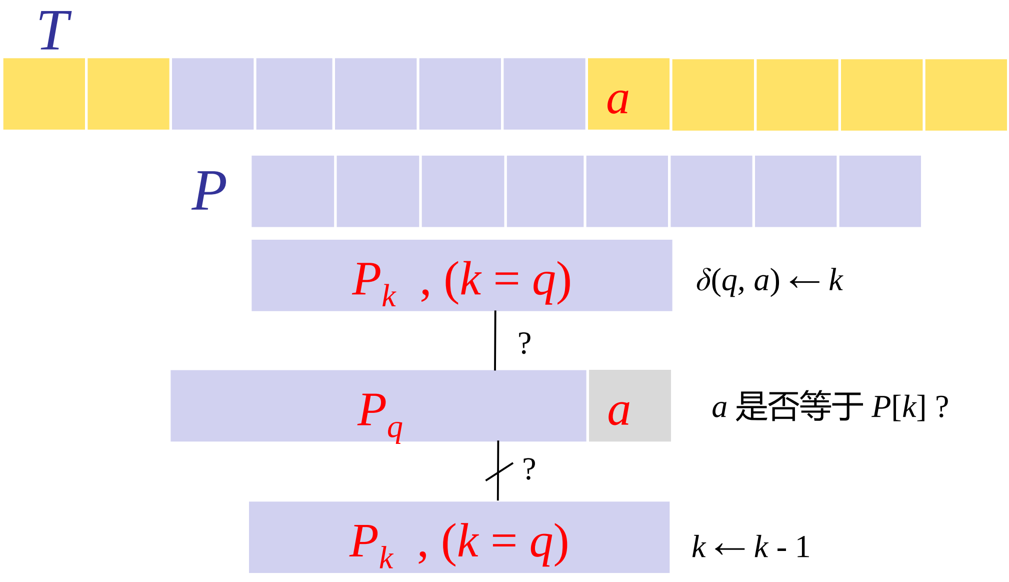
COMPUTE-TRANSITION-FUNCTION(P, Σ)

```
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 a 时：

1. $k = q+1$ （超过 m 时，取 m，匹配数不大于 m）
2. $P_k \sqsupset P_q a$ ？
3. 若2成立，则 $P_q a == P_k$ ，即，对在 T 的继续扫描过程中，若扫描的下一个字符 $a == P[k]$，则匹配字符增加 1 （或继续为 m）
4. 若2不成立，即 $P_q a \mathrel{!}= P_k$ ，即，对在T 的继续扫描过程中，若扫描的下一个字符 $a \mathrel{!}= P[k]$，模版右移(k--)，goto step 2

Computing the transition function



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

A string-matching automaton

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

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

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

1. $k = q+1$ （超过 m 时，取 m，匹配数不大于 m）
2. $P_k \sqsupset P_q a$?
3. 若2成立，则 $P_q a == P_k$ ，即，对在 T 的继续扫描过程中，若扫描的下一个字符 $a == P[k]$，则匹配字符增加 1 （或继续为 m）
4. 若2不成立，即 $P_q a \mathrel{!=} P_k$ ，即，对在 T 的继续扫描过程中，若扫描的下一个字符 $a \mathrel{!=} P[k]$，模版右移(k--)，goto step 2

Computing the transition function

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

A string-matching automaton

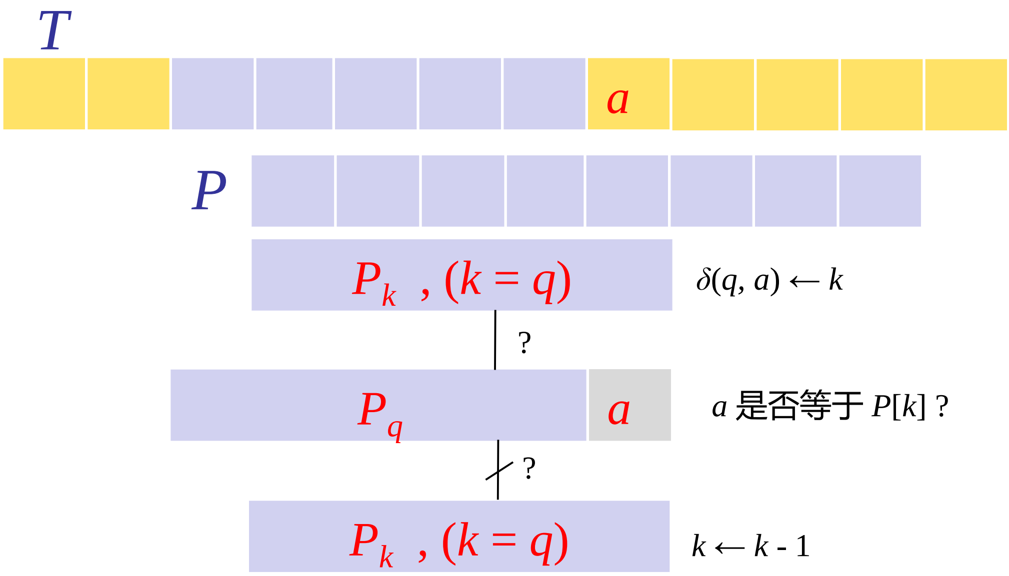
$\boldsymbol{M} = (Q, q_0, A, \Sigma, \delta)$:

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

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

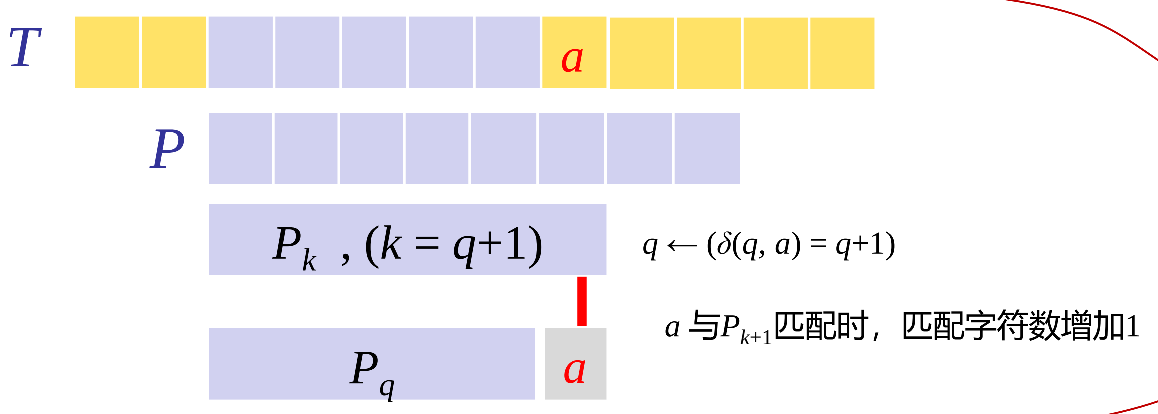
Computing δ from a given pattern $P[1 .. m]$:

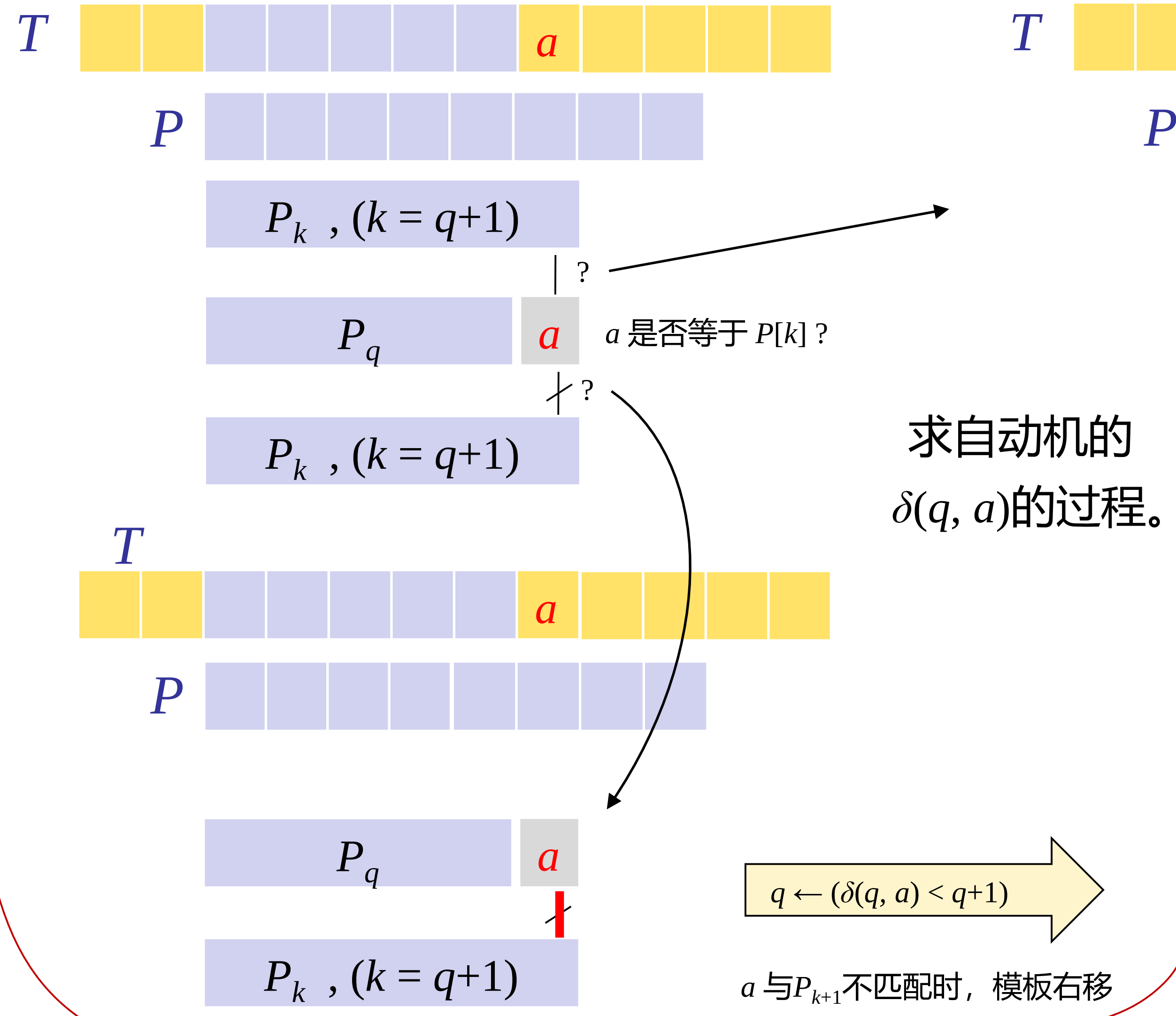
```
COMPUTE-TRANSITION-FUNCTION(P, Σ)
m ← length[P]
for q ← 0 to m
     for each character a ∈ Σ
          k ← min(m, q + 1)
          while P_k !⊐ P_q a
               k--
          δ(q, a) ← k
return δ
```



- Running time ?

32.3 String matching with finite automata（有限自动机，有穷自动机）

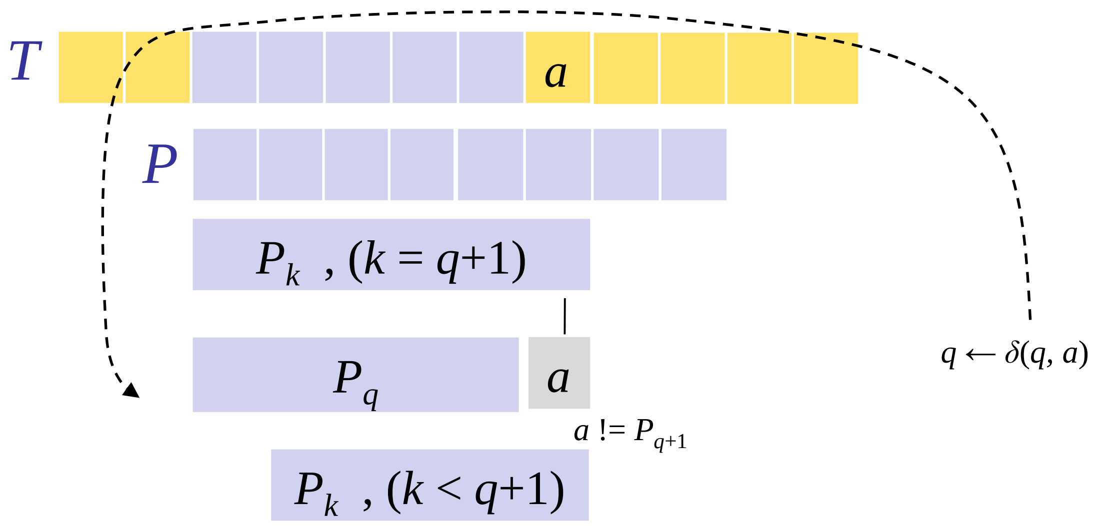




在 T 中的匹配过程：匹配数为 q 时，输入 a ，通过查转移函数 $\delta(q, a)$，可快速知道新的匹配数是多少，即，能够快速移动模版，然后从下一个输入继续。

*32.4 The Knuth-Morris-Pratt algorithm

FA

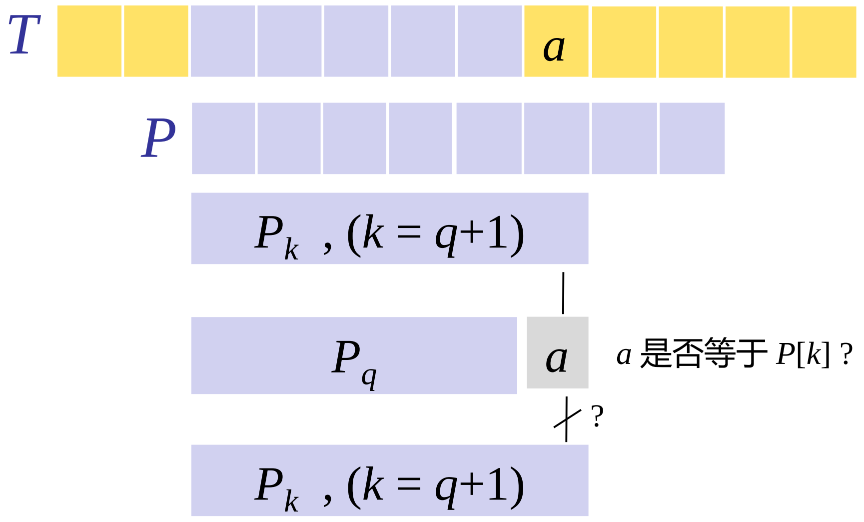


对 T 的扫描过程：

根据输入的 a，然后查表 δ 知道需要转移的位置。

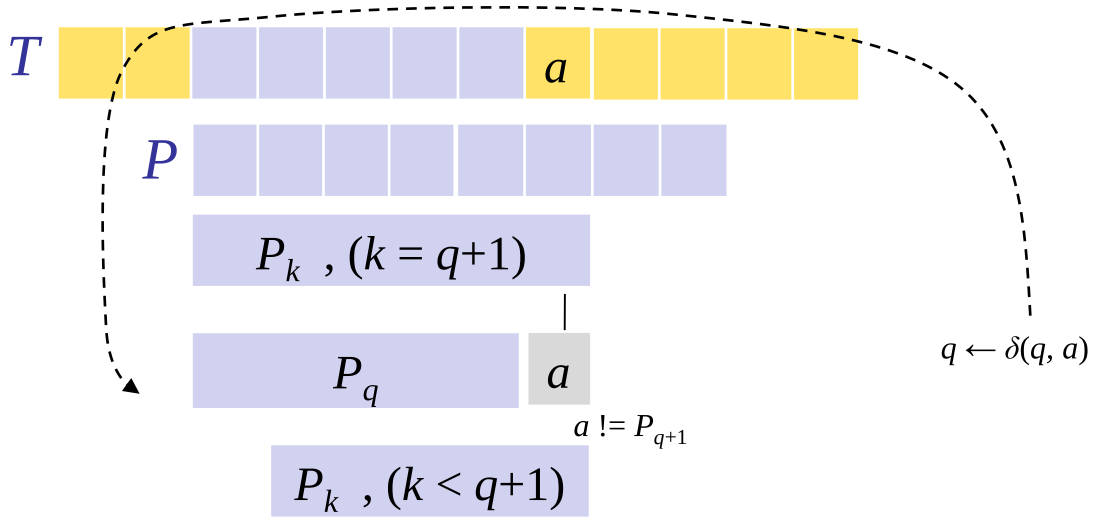
自动机构造完后，从 δ 已经知道输入 a 后 P 应快速右移多少（这种思想跟 KMP 相似，但 FA 的核心在于求 δ 有额外计算开销）。

T 中一个字符扫描一次。



*32.4 The Knuth-Morris-Pratt algorithm

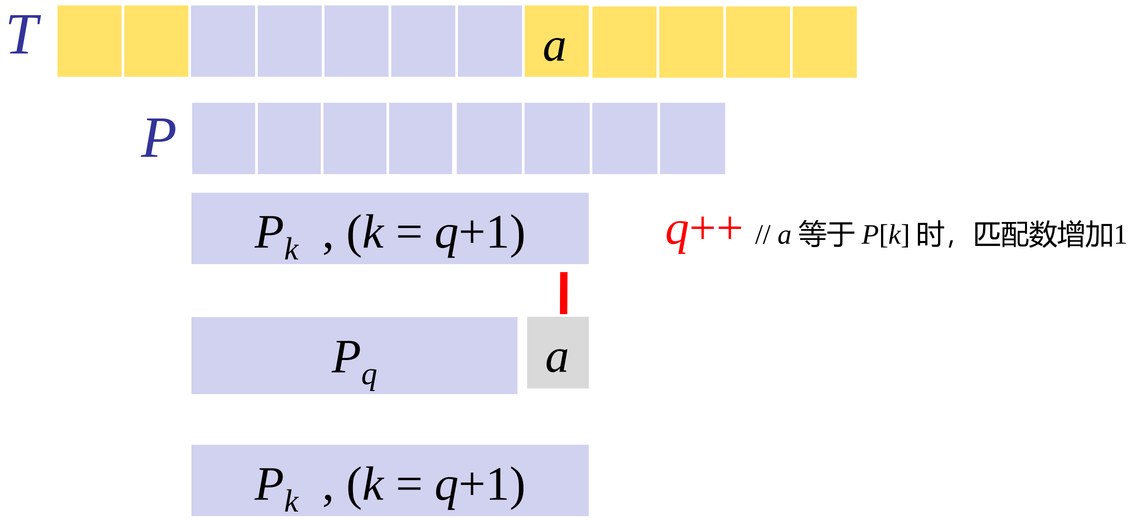
FA



对 T 的扫描过程：

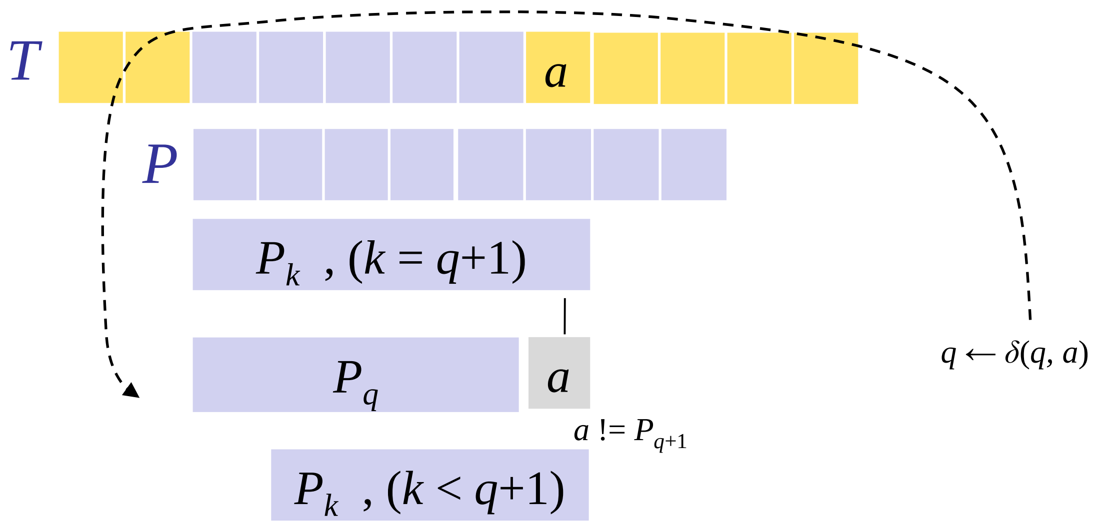
根据输入的 a ，然后查表 δ 知道需要转移的位置。自动机构造完后，从 δ 已经知道输入 a 后 P 应快速右移多少（这种思想跟 KMP 相似，但 FA 的核心在于求 δ 有额外计算开销）。

T 中一个字符扫描一次。



*32.4 The Knuth-Morris-Pratt algorithm

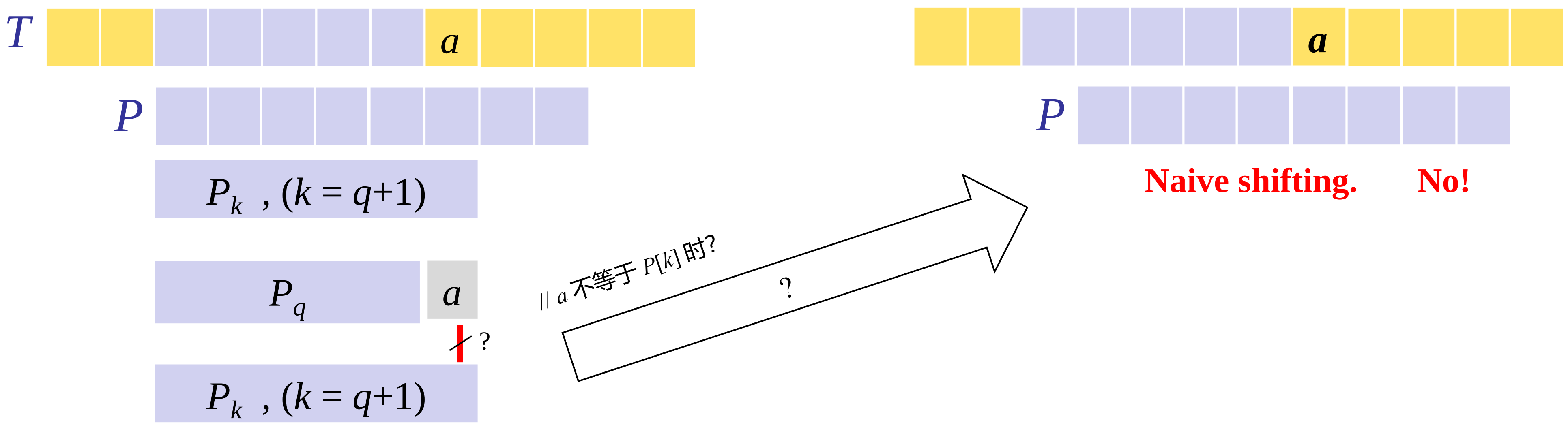
FA



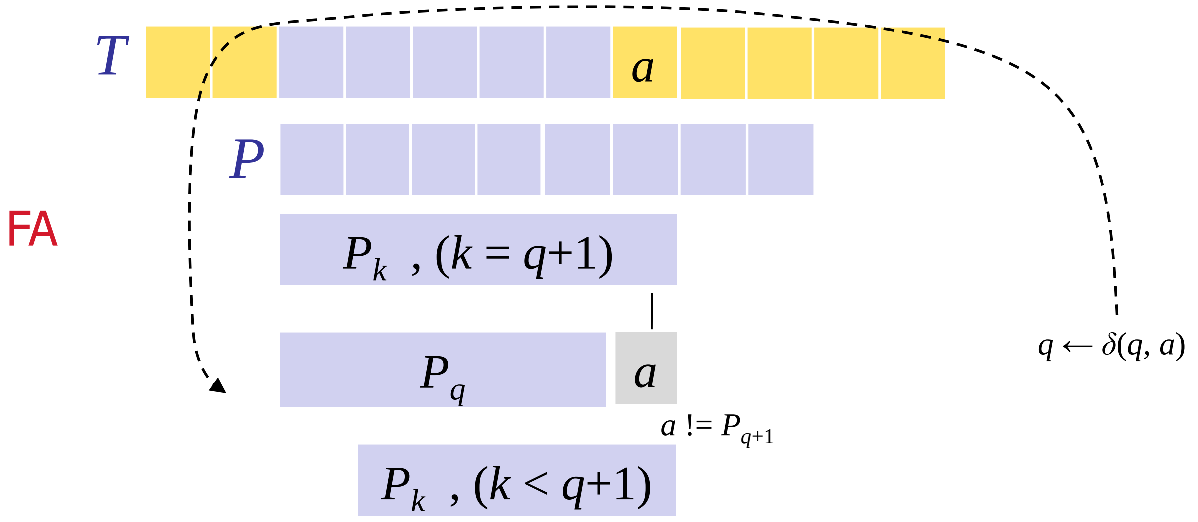
对 T 的扫描过程：

根据输入的 a，然后查表 δ 知道需要转移的位置。自动机构造完后，从 δ 已经知道输入 a 后 P 应快速右移多少（这种思想跟 KMP 相似，但 FA 的核心在于求 δ 有额外计算开销）。

T 中一个字符扫描一次。



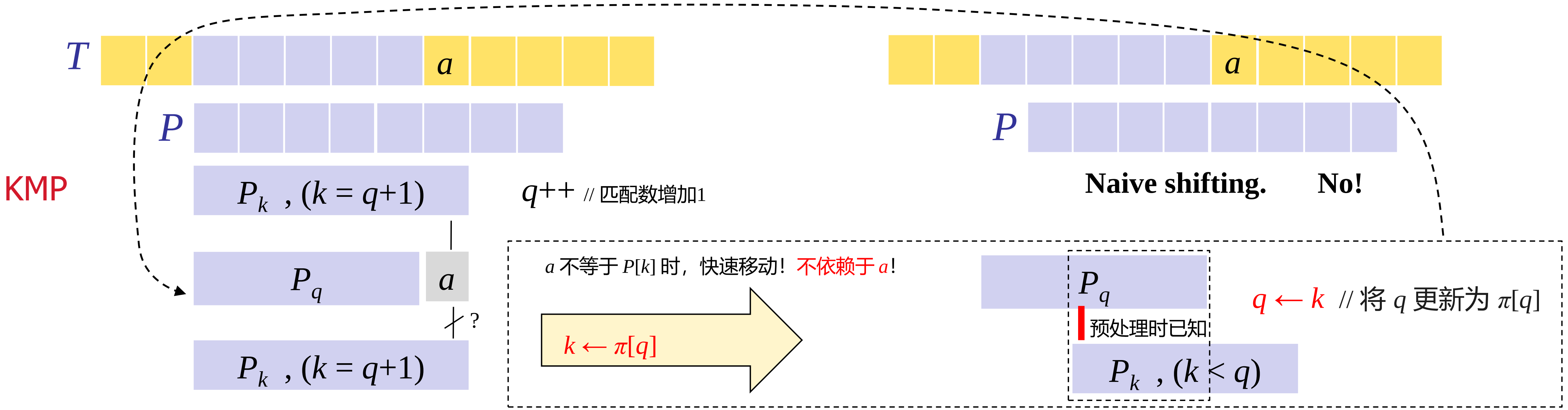
*32.4 The Knuth-Morris-Pratt algorithm



对 T 的扫描过程：

根据输入的 a ，然后查表 δ 知道需要转移的位置。自动机构造完后，从 δ 已经知道输入 a 后 P 应快速右移多少（这种思想跟 KMP 相似，但 FA 的核心在于求 δ 有额外计算开销）。

T 中一个字符扫描一次。



KMP本质：P 的前缀 P_q 与 T 的匹配；P_k (k<q) 是 P_q 的后缀（也是 P_q 的前缀），最大的 k 是多少？**$q \leftarrow \pi(q)$ 后，重新看是否 a == P[q+1] （重复执行此过程）**

⇦

KMP 的前缀函数构造完后，在 T 中无论下一个输入 a 是什么都知道偏移多少位置（从 P_q 移动到 P_k）（偏移跟输入 a 无关，这是KMP跟FA方法的区别）。T 中一个字符 a 可能扫描多次。

*32.4 The Knuth-Morris-Pratt algorithm

KMP is a linear-time string-matching algorithm due to Knuth, Morris, and Pratt.

Fast pattern matching in strings

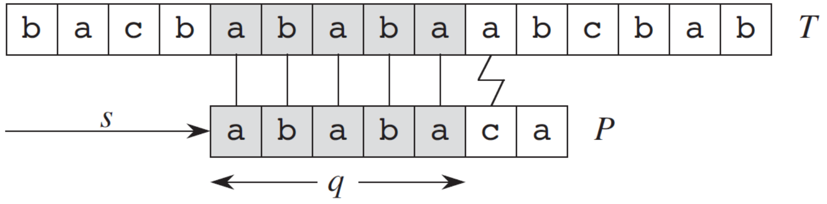
DE Knuth, JH Morris, Jr, VR Pratt - SIAM journal on computing, 1977 - SIAM

… Finally, 8 discusses still more recent work on pattern matching. … Theidea behind this approach to pattern matching is perhaps easiest to grasp if we imagine placing the pattern over the …

☆ 保存 99 引用 被引用次数：4686 相关文章 所有 17 个版本 »

三个作者分别提出，联合发表的一篇文章。
KMP 是一个很伟大的算法。

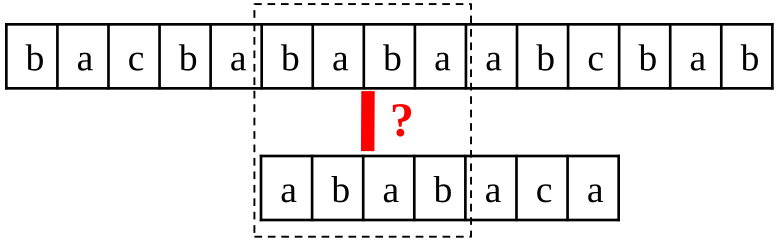
*32.4 The Knuth-Morris-Pratt algorithm



(a)

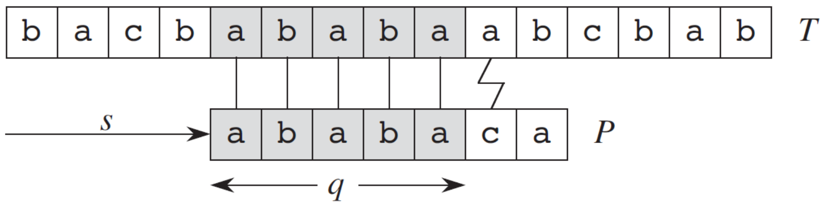
$P_5 \sqsupset T_{s+5}$, but,

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



Naive shifting, $P_4 \sqsupset T_{s+1+4}$? **No!**

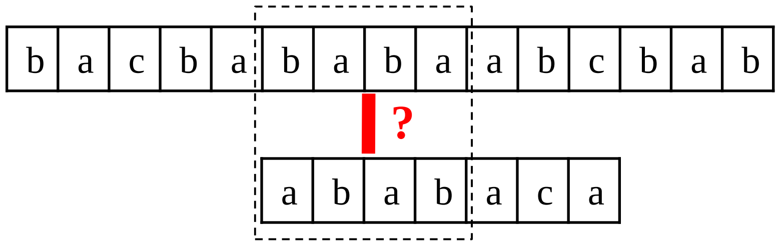
*32.4 The Knuth-Morris-Pratt algorithm



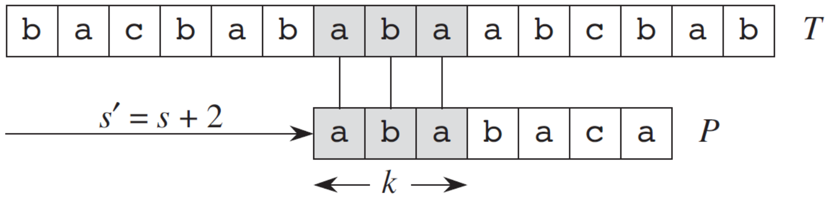
(a)

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

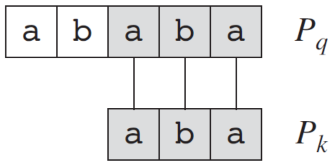
$T[s+5+1]$!= $P[5+1]$ ($q = 5$)



Naive shifting, $P_4 \sqsupset T_{s+1+4}$? **No!**



(b)



(c)

We have already known that P_k is the maximum suffix of P_q , that is, $P_{k'} \sqsupset P_q$ ($k' < q$, and $k = \max(k')$). For this example, q is 5, k is 3. So, we have $P_3 \sqsupset P_5 \sqsupset T_{s+5}$, we just check whether ..

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

*32.4 The Knuth-Morris-Pratt algorithm

prefix function for the pattern P is the function..

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

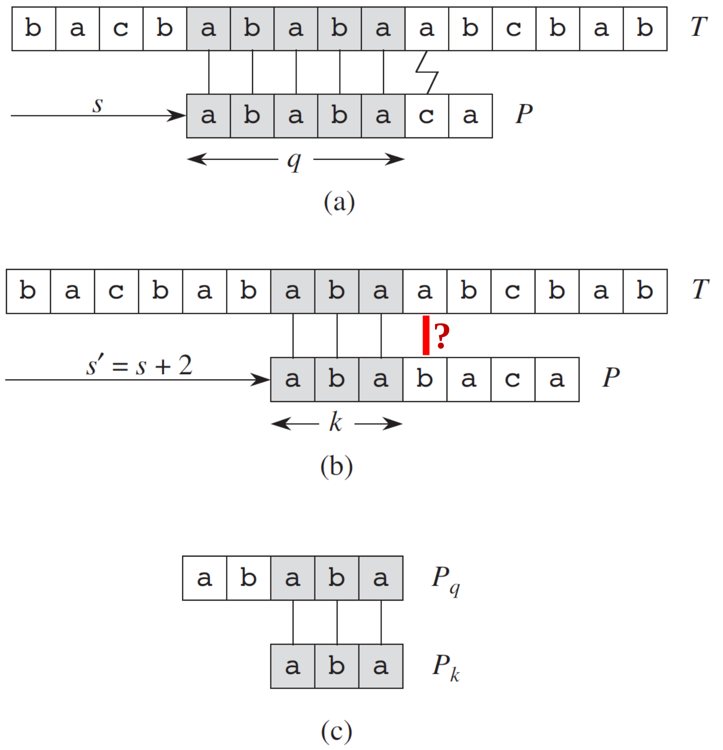
$\pi[q] = \max\{k : k < q \text{ and } P_k \sqsupset P_q\}$

P 的 前缀 P_q 的真后缀是 P 的前缀 P_k （最长的）

✔ P 的 前缀 P_q 的真前缀 P_k 是 前缀 P_q 的后缀（最长的）

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 $\pi[q]$

P = ababaca
P_5 = ababa
P_3 = aba
$\Rightarrow$ $\pi[5] = 3$



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

$T[s+5+1] \mathrel{!}= P[5+1]$ ($q = 5$)

We have already known that P_k is the maximum suffix of P_q, that is, $P_{k'} \sqsupset P_q$ ($k' < q$, and $k = \max(k')$). For this example, q is 5, k is 3. So, we have $P_3 \sqsupset P_5 \sqsupset T_{s+5}$, we just check whether ..

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

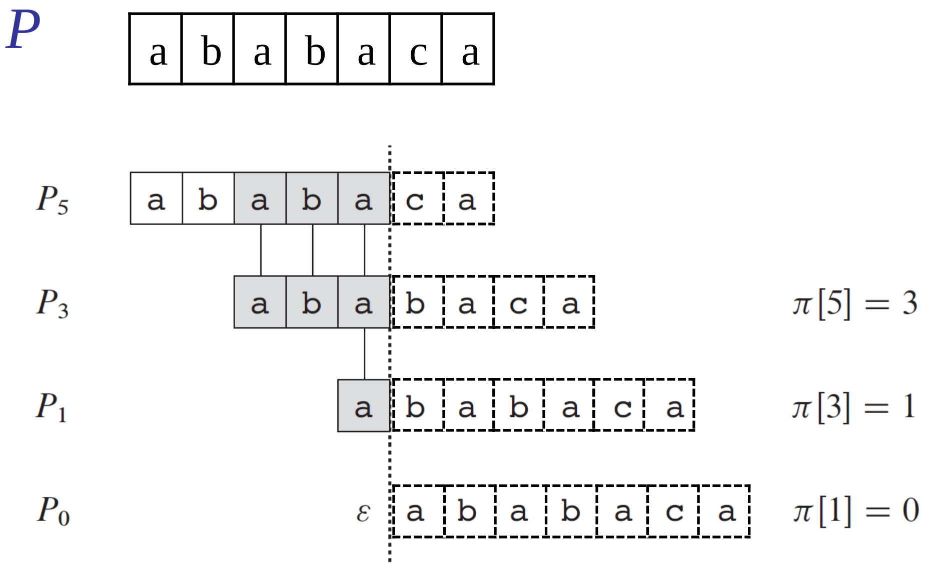
*32.4 The Knuth-Morris-Pratt algorithm

prefix function for the pattern P is the function..

π : {1, 2, …, m} $\rightarrow$ {0, 1, …, m-1} such that $\pi[q] = \max\{k : k < q \text{ and } P_k \sqsupset P_q\}$.

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)

P 的前缀 P_q 的真前缀 P_k 是前缀 P_q 的后缀（最长的）

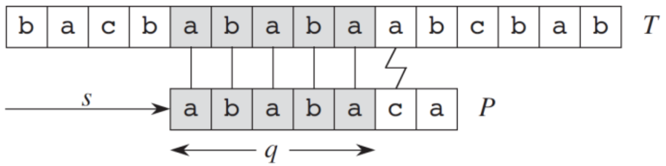
*32.4 The Knuth-Morris-Pratt algorithm

```
KMP-MATCHER(T, P)
n = T.length
m = P.length
π = COMPUTE-PREFIX-FUNCTION(P)                    ?
q = 0                                             已经匹配的字符个数
for i = 1 to n                                    依序扫描文本 T
    while q > 0 and P[q + 1] ≠ T[i]               输入字符 T[i] 跟模版字符不匹配
        q = π[q]                                  根据前缀函数快速移位
    if P[q + 1] == T[i]                           输入字符 T[i] 跟模版字符匹配
        q = q + 1                                 匹配个数增加1
    if q == m                                     匹配个数为 m（模版 P 的长度），找到匹配，输出偏移量
        print "Pattern occurs with shift" i - m
        q = π[q]                                  找下一个匹配（这种情况包括了模版重叠情况，如果不考虑重叠，算法如何改写）
```

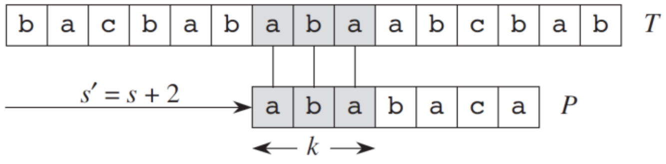
已知 $P_q \sqsupset T[i-1]$

判断 $P[q+1] \neq T[i]$

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



(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
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" i − m
        q = π[q]
```

已知 $P_q \sqsupset T[i\text{-}1]$
判断 $P[q+1] \neq T[i]$

```
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
```

已知 $\pi[q\text{-}1] = k$, 求 $\pi[q]$
即，已知 $P_k \sqsupset P[q\text{-}1]$, 判断 $P[k+1] \neq P[q]$

prefix function for the pattern P is the function..

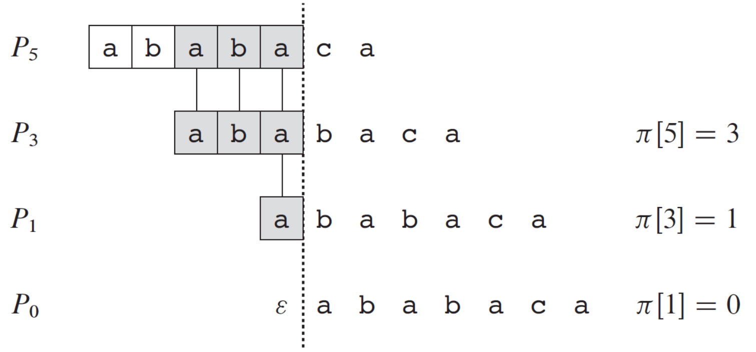
$\pi : \{1, 2, \ldots, m\} \rightarrow \{0, 1, \ldots, m\text{-}1\}$ such that

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

P 的前缀 P_q 的真前缀 P_k 是前缀 P_q 的后缀（最长的）

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

```
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
```

已知 $\pi[q\text{-}1] = k$, 求 $\pi[q]$

即，已知 $P_k \sqsupset P[q\text{-}1]$, 判断 $P[k+1] \neq P[q]$

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

$q = 2$:
∵ k is 0, $P[1] \neq P[2]$
∴ $\pi[2] \leftarrow 0$

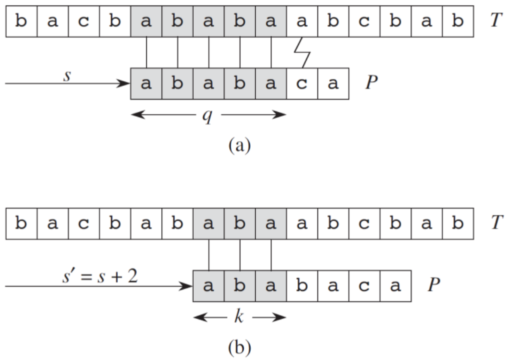
a b a b a c a
a b a b a c a

$q = 3$:
∵ k is 0, $P[1] == P[3]$
∴ $k \leftarrow 1$, $\pi[3] \leftarrow 1$

a b a b a c a
a b a b a c a

$q = 4$:
∵ $k > 0$, $P[2] == P[4]$
∴ $k \leftarrow 2$, $\pi[4] \leftarrow 2$

a b a b a c a
a b a b a c a



prefix function for the pattern P is the function..

$\pi : \{1, 2, \ldots, m\} \to \{0, 1, \ldots, m\text{-}1\}$ such that

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

P 的前缀 P_q 的真前缀 P_k 是前缀 P_q 的后缀（最长的）

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

Running time? chapter17 Amortized analysis (accounting)

```
KMP-MATCHER(T, P)                                   Θ(n)
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" i − m
        q = π[q]
```

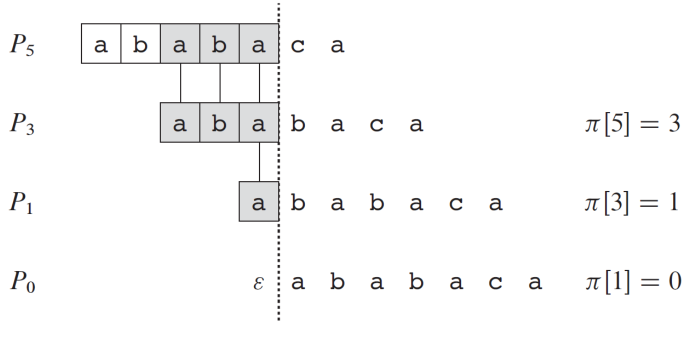
```
COMPUTE-PREFIX-FUNCTION(P)                          Θ(m)
m = P.length
let π[1 . . m] be a new array
π[1] = 0
k = 0
for q = 2 to m
    while k > 0 and P[k + 1] ≠ P[q]
        k = π[k]
    if P[k + 1] == P[q]
        k = k + 1
    π[q] = k
return π
```

prefix function:

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

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

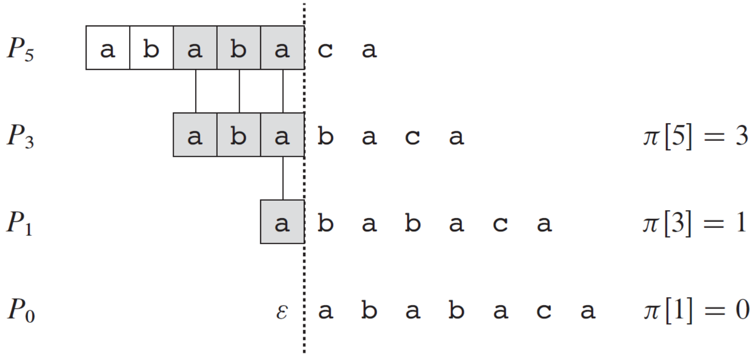
KMP algorithm avoids computing the transition function δ, and its matching time is $\Theta(n)$ using just an auxiliary function π, which we precompute from the pattern in time $\Theta(m)$ and store in an array $\pi[1 \,..\, m]$.

prefix function:

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

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

All