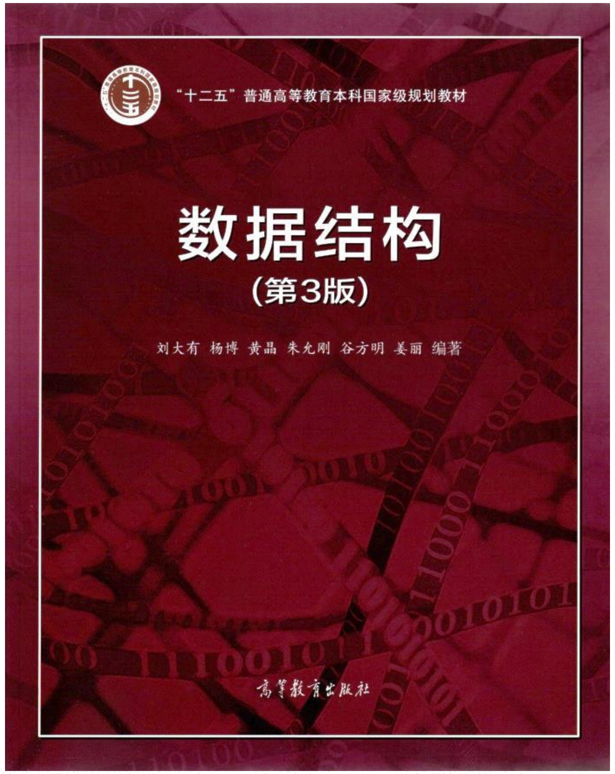


字符串匹配

- 模式匹配基本概念
- 朴素模式匹配
- KMP算法
- KMP算法的改进

数据之法
结构之美
算法之道

Last updated on 2023.10

zhuyungang@jlu.edu.cn

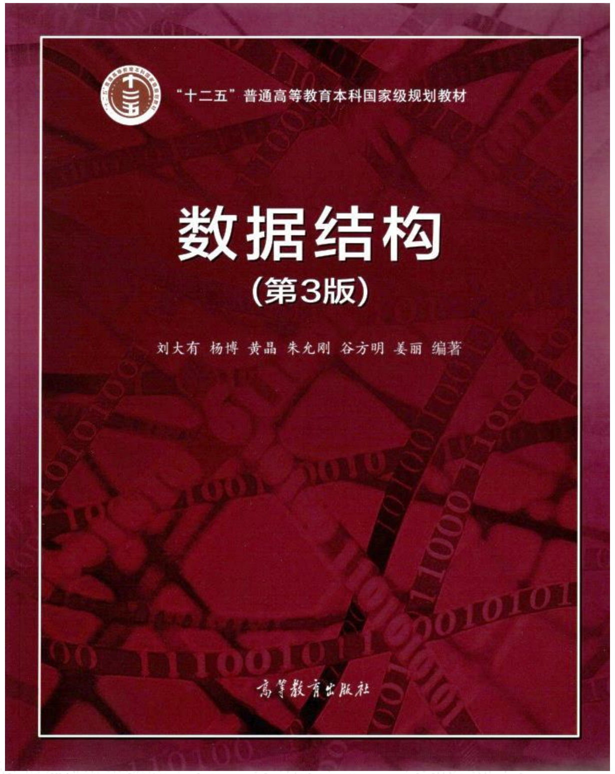
Computer programming is an art, because it applies knowledge to the world, because it requires skill and ingenuity①, and especially because it produces objects of beauty. A programmer who views himself as an artist will enjoy what he does and will do it better.

——Donald Knuth
图灵奖获得者
斯坦福大学教授
美国科学院院士
美国工程院院士

① *ingenuity*: 聪明才智, 独创力, 心灵手巧

慕课自学内容（必看，计入期末成绩）

自学内容	视频时长
字符串的定义与字符串类	7分56秒



字符串模式匹配

- **模式匹配基本概念**
- 朴素模式匹配算法
- KMP算法
- KMP算法的改进

数据之法
结构之美
算法之道

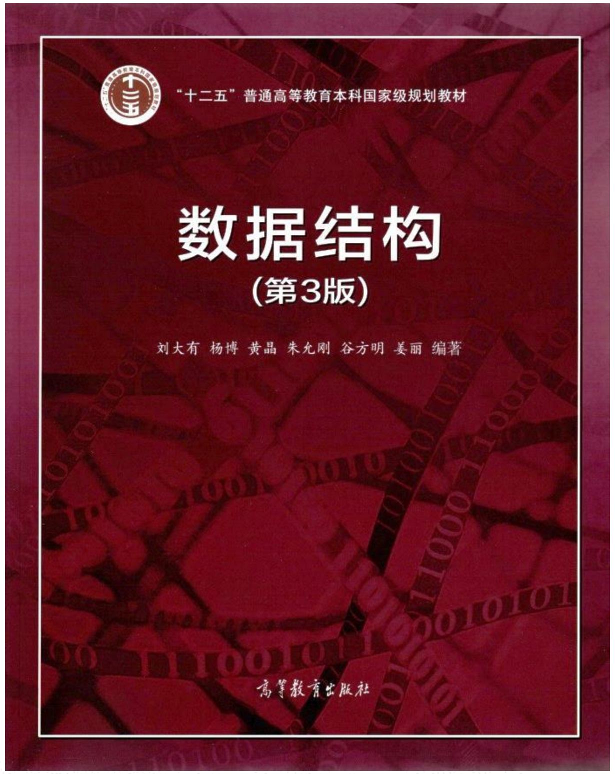
zhuyungang@jlu.edu.cn

字符串模式匹配问题定义

- 在**目标串**中寻找**模式串**出现的**首位置**。
- 给定两个字符串S和P，目标串S有n个字符，模式串P有m个字符，$m \le n$。判断P是否为S的子串，如果是则返回P的第一个字符在S中的首位置；如果不是则返回-1 。【大厂面试题LeetCode28】

例：S= "abaabab", P = "abab"

- 模式匹配应用：文本编辑器中常用的"查找"、"替换"



字符串模式匹配

- 模式匹配基本概念
- **朴素模式匹配算法**
- KMP算法
- KMP算法的改进

数据之法
结构之美
算法之道

Last updated on 2023.10

zhuyungang@jlu.edu.cn

朴素模式匹配算法（Brute Force算法，暴力算法）

	0	1	2	3	4	5	6	7
S	*a*	*b*	*a*	*a*	*a*	*b*	*a*	*b*

P	*a*	*b*	*a*	*b*
	0	1	2	3

第 0 趟匹配（看*P*是否在*S*的第0个位置）

朴素模式匹配算法（Brute Force算法，暴力算法）

	0	1	2	3	4	5	6	7
S	a	b	a	a	a	b	a	b

P	a	b	a	b
	0	1	2	3

第 0 趟匹配（看P是否在S的第0个位置）

第 1 趟匹配（看P是否在S的第1个位置）

朴素模式匹配算法（Brute Force算法，暴力算法）

	0	1	2	3	4	5	6	7
S	*a*	*b*	*a*	*a*	*a*	*b*	*a*	*b*

	a	*b*	*a*	*b*
P	0	1	2	3

第 0 趟匹配（看*P*是否在*S*的第0个位置）

第 1 趟匹配（看*P*是否在*S*的第1个位置）

第 2 趟匹配（看*P*是否在*S*的第2个位置）

… …

朴素模式匹配算法（Brute Force算法，暴力算法）

	0	1	2	3	4	5	6	7
S	*a*	*b*	*a*	*a*	*a*	*b*	*a*	*b*

P	*a*	*b*	*a*	*b*
	0	1	2	3

第 0 趟匹配（看*P*是否在*S*的第0个位置）

第 1 趟匹配（看*P*是否在*S*的第1个位置）

第 2 趟匹配（看*P*是否在*S*的第2个位置）

…… ……

朴素模式匹配算法（Brute Force算法，暴力算法）

	0	1	2	3	4	5	6	7
S	*a*	*b*	*a*	*a*	*a*	*b*	*a*	*b*
P					*a*	*b*	*a*	*b*
					0	1	2	3

第 0 趟匹配（看*P*是否在*S*的第0个位置）

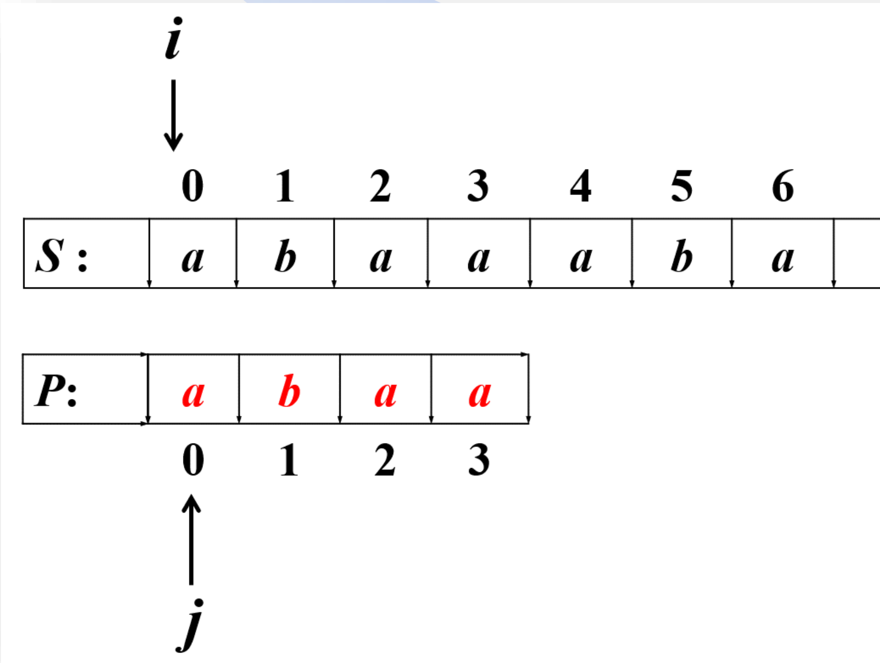
第 1 趟匹配（看*P*是否在*S*的第1个位置）

第 2 趟匹配（看*P*是否在*S*的第2个位置）

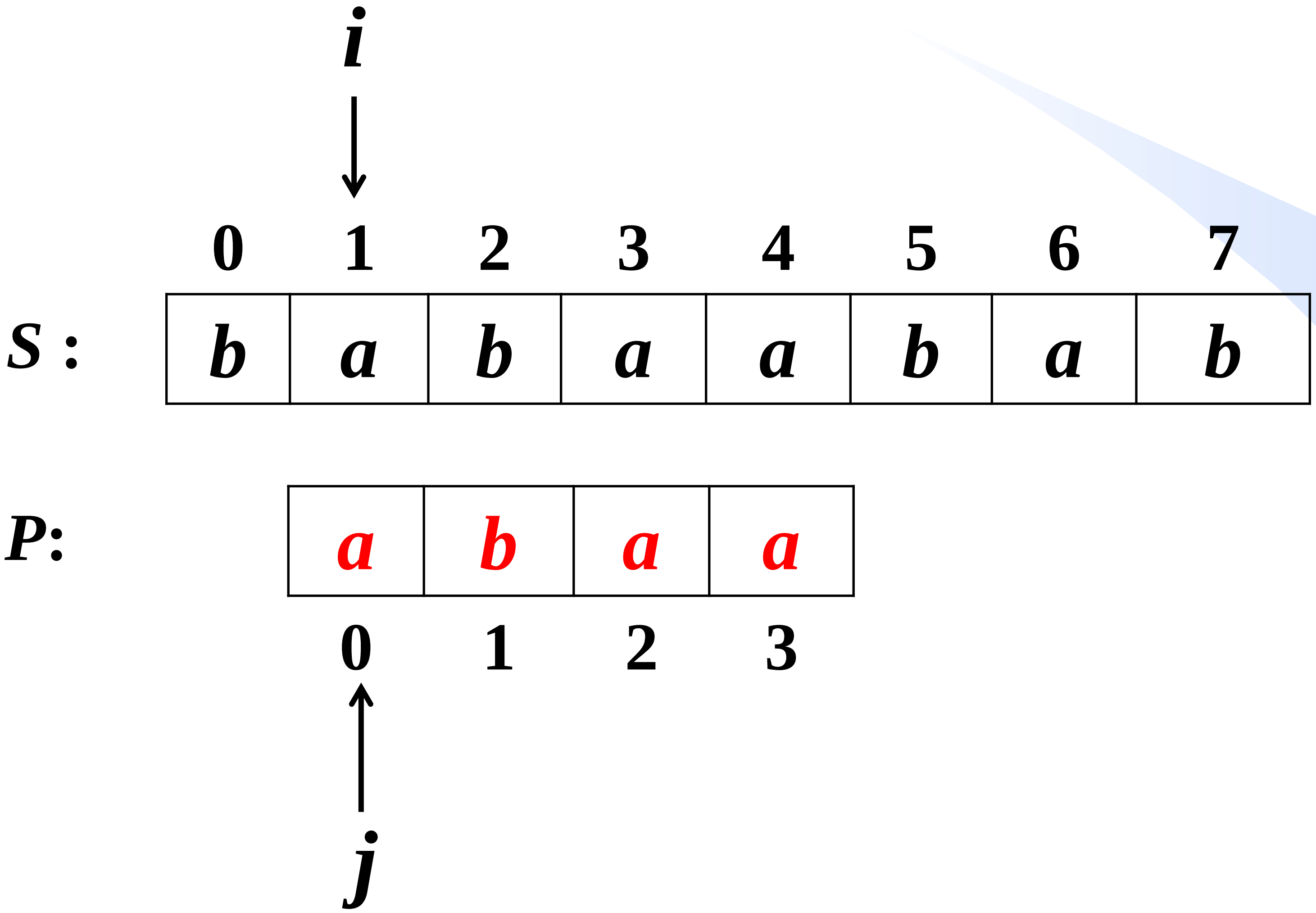
… …

朴素模式匹配算法

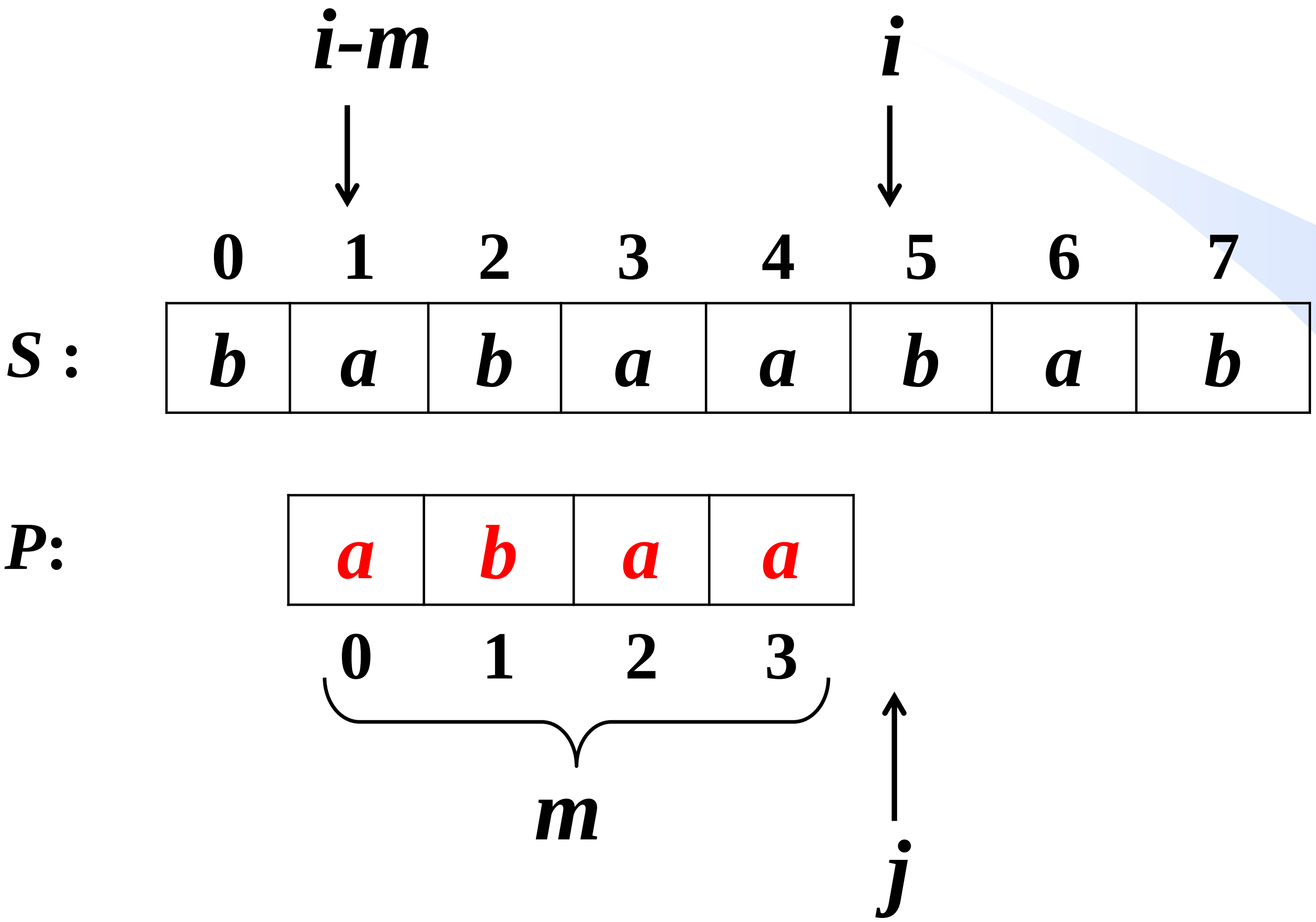
```
int StringMatching(char *s, char *p){ //返回p在s中的位置
    int n=strlen(s), m=strlen(p);  //n为s的长度,m为p的长度
    int i = 0;        //通过i扫描s
    while(i<=n-m){ //看p是否在s的第i个位置,从s[i]开始与p逐字符比对
        int j = 0;   //通过j扫描p
        while(j < m && s[i] == p[j]) {
            i++; j++;
        }
        if(j==m) return i-m; // 匹配成功
        ……未完待续
```



匹配成功的情况



匹配成功的情况



朴素模式匹配算法

```
int StringMatching(char *s, char *p){ //返回p在s中的位置
    int n=strlen(s), m=strlen(p);  //n为s的长度,m为p的长度
    int i = 0;       //通过i扫描s
    while(i<=n-m){ //看p是否在s的第i个位置,从s[i]开始与p逐字符比对
        int j = 0;   //通过j扫描p
        while (j < m && s[i] == p[j]) {
            i++; j++;
        }
        if (j == m) return i - m; // 匹配成功
        i = i - j + 1;    //本趟匹配失败，指针i回退到原位置+1
    }
    ……未完待续
```

本趟匹配失败的情况

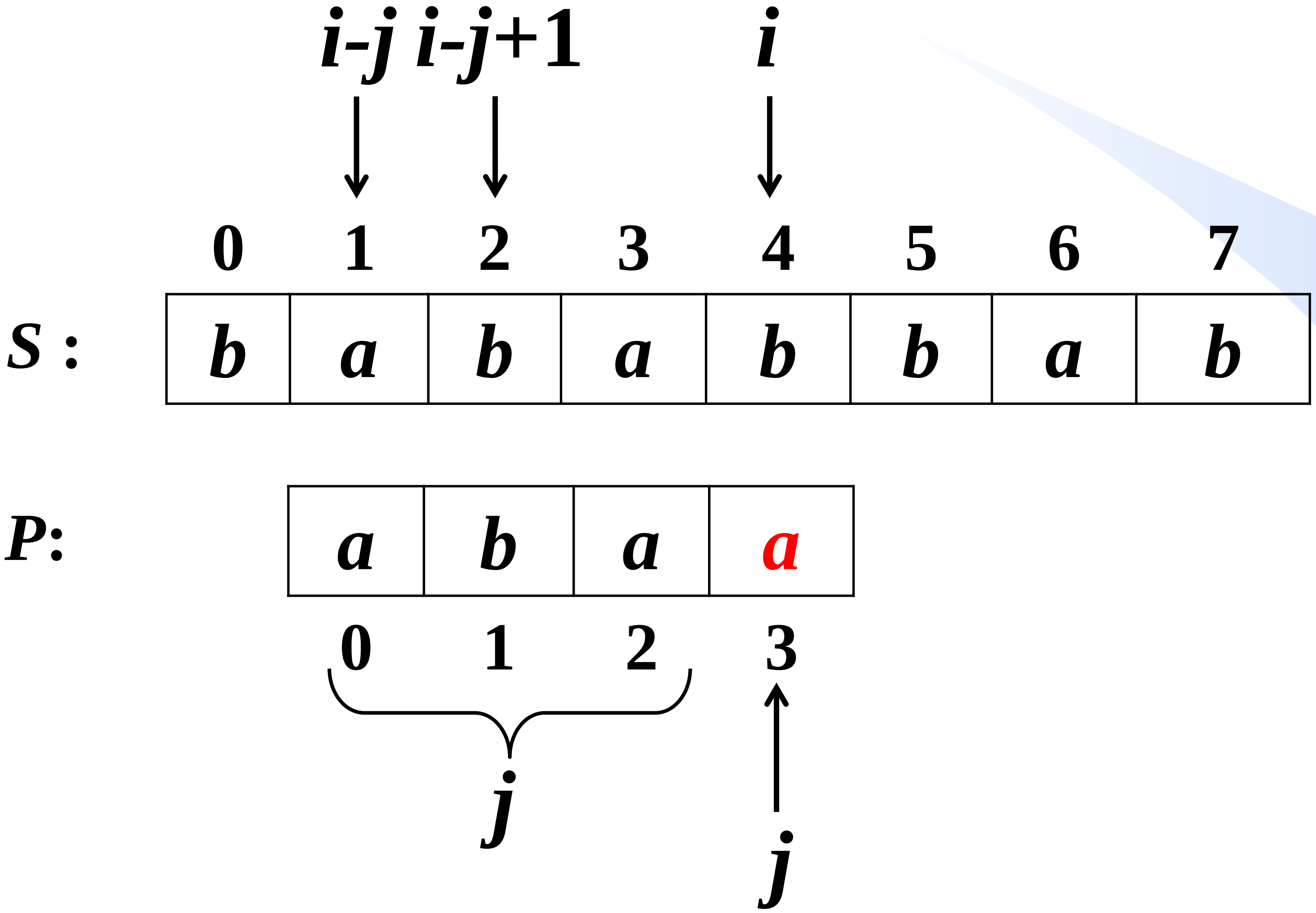
i

	0	1	2	3	4	5	6	7
S :	*b*	*a*	*b*	*a*	*b*	*b*	*a*	*b*

P:	*a*	*b*	*a*	*a*
	0	1	2	3

j

本趟匹配失败的情况



本趟匹配失败的情况

$i\text{-}j$ $i\text{-}j+1$

	0	1	2	3	4	5	6	7
S :	*b*	*a*	*b*	*a*	*b*	*b*	*a*	*b*

P:	*a*	*b*	*a*	*a*
	0	1	2	3

j

本趟匹配失败的情况

$i \leftarrow i\text{-}j+1$

S :

0	1	2	3	4	5	6	7
b	*a*	*b*	*a*	*b*	*b*	*a*	*b*

P:

a	*b*	*a*	*a*
0	1	2	3

j

每次失配后：
模式串右移1位
指针 *i* 回退到*i-j*+1
指针 *j* 回退到0

朴素模式匹配算法

```
int StringMatching(char *s, char *p){ //返回p在s中的位置
    int n=strlen(s), m=strlen(p);  //n为s的长度,m为p的长度
    int i = 0;       //通过i扫描s
    while(i<=n-m){ //看p是否在s的第i个位置,从s[i]开始与p逐字符比对
        int j = 0;   //通过j扫描p
        while (j < m && s[i] == p[j]) {
            i++; j++;
        }
        if (j == m) return i - m; // 匹配成功
        i = i - j + 1;    //本趟匹配失败，指针i回退到原位置+1
    }
    return -1; // p不在s中
}
```

关键运算：字符比较

最坏时间复杂度

$O(n \cdot m)$

朴素模式匹配算法——平均情况时间复杂度

假定目标串S长度为n，模式串P长度为m，证明朴素模式匹配算法在平均时间情况下的字符比较次数不超过$2n$.【东京大学考研题】

- 考察第i趟匹配：看P是否在S的第i个位置。
- 假定字符集包含k种字符，则P_i等于S中某个字符的概率为$\frac{1}{k}$，P_i不等于S中某个字符的概率为$1-\frac{1}{k}$

S	…	s_i	s_{i+1}	s_{i+2}	…	s_{i+m-2}	s_{i+m-1}	…
P		p_0	p_1	p_2	…	p_{m-2}	p_{m-1}	

朴素模式匹配算法——平均情况时间复杂度

	字符比较次数	发生概率
第1种输入	1	$1-\frac{1}{k}$

S	…	s_i	s_{i+1}	s_{i+2}	…	s_{i+m-2}	s_{i+m-1}	…
		$\neq$						
P		p_0	p_1	p_2	…	p_{m-2}	p_{m-1}	

朴素模式匹配算法——平均情况时间复杂度

	字符比较次数	发生概率
第2种输入	2	$\frac{1}{k}\left(1-\frac{1}{k}\right)$

S	…	s_i	s_{i+1}	s_{i+2}	…	s_{i+m-2}	s_{i+m-1}	…
		$=$	$\neq$					
P		p_0	p_1	p_2	…	p_{m-2}	p_{m-1}	

朴素模式匹配算法——平均情况时间复杂度

	字符比较次数	发生概率
第3种输入	3	$\frac{1}{k^2}\left(1-\frac{1}{k}\right)$

S	…	s_i	s_{i+1}	s_{i+2}	…	s_{i+m-2}	s_{i+m-1}	…
		$=$	$=$	$\neq$				
P		p_0	p_1	p_2	…	p_{m-2}	p_{m-1}	

朴素模式匹配算法——平均情况时间复杂度

	字符比较次数	发生概率
第m-1种输入	m-1	$\frac{1}{k^{m-2}}\left(1-\frac{1}{k}\right)$

S	…	s_i	s_{i+1}	s_{i+2}	…	s_{i+m-2}	s_{i+m-1}	…
		=	=	=	=	≠		
P		p_0	p_1	p_2	…	p_{m-2}	p_{m-1}	

朴素模式匹配算法——平均情况时间复杂度

	字符比较次数	发生概率
第m种输入	m	$\frac{1}{k^{m-1}}\left(1-\frac{1}{k}+\frac{1}{k}\right)$

S	…	s_i	s_{i+1}	s_{i+2}	…	s_{i+m-2}	s_{i+m-1}	…
		=	=	=	=	=	≠或=	
P		p_0	p_1	p_2	…	p_{m-2}	p_{m-1}	

朴素模式匹配算法——平均情况时间复杂度

第i趟匹配：看P是否在S的第i个位置。假定字符集包含k种字符

	字符比较次数	发生概率
第1种输入	1	$1-\frac{1}{k}$
第2种输入	2	$\frac{1}{k}\left(1-\frac{1}{k}\right)$
…	…	…
第m-1种输入	m-1	$\frac{1}{k^{m-2}}\left(1-\frac{1}{k}\right)$
第m种输入	m	$\frac{1}{k^{m-1}}\left(1-\frac{1}{k}+\frac{1}{k}\right)$

朴素模式匹配算法——平均情况时间复杂度

$$T_i=\left(1-\frac{1}{k}\right)+\frac{2}{k}\left(1-\frac{1}{k}\right)+\frac{3}{k^2}\left(1-\frac{1}{k}\right)+\cdots+\frac{m-1}{k^{m-2}}\left(1-\frac{1}{k}\right)+\cdots+\frac{m}{k^{m-1}}\left(1-\frac{1}{k}+\frac{1}{k}\right)$$

$$T_i=\left(1-\frac{1}{k}\right)\left[1+\frac{2}{k}+\frac{3}{k^2}+\cdots+\frac{m}{k^{m-1}}\right]+\frac{m}{k^m}$$

$$令\ \ S=\left(1+\frac{2}{k}+\frac{3}{k^2}+\cdots+\frac{m}{k^{m-1}}\right)①$$

$$则\frac{S}{k}=\left(\frac{1}{k}+\frac{2}{k^2}+\frac{3}{k^3}+\cdots+\frac{m}{k^m}\right)②$$

$$令①-②有\left(1-\frac{1}{k}\right)S=\left(1+\frac{1}{k}+\frac{1}{k^2}+\cdots+\frac{1}{k^{m-1}}-\frac{m}{k^m}\right)$$

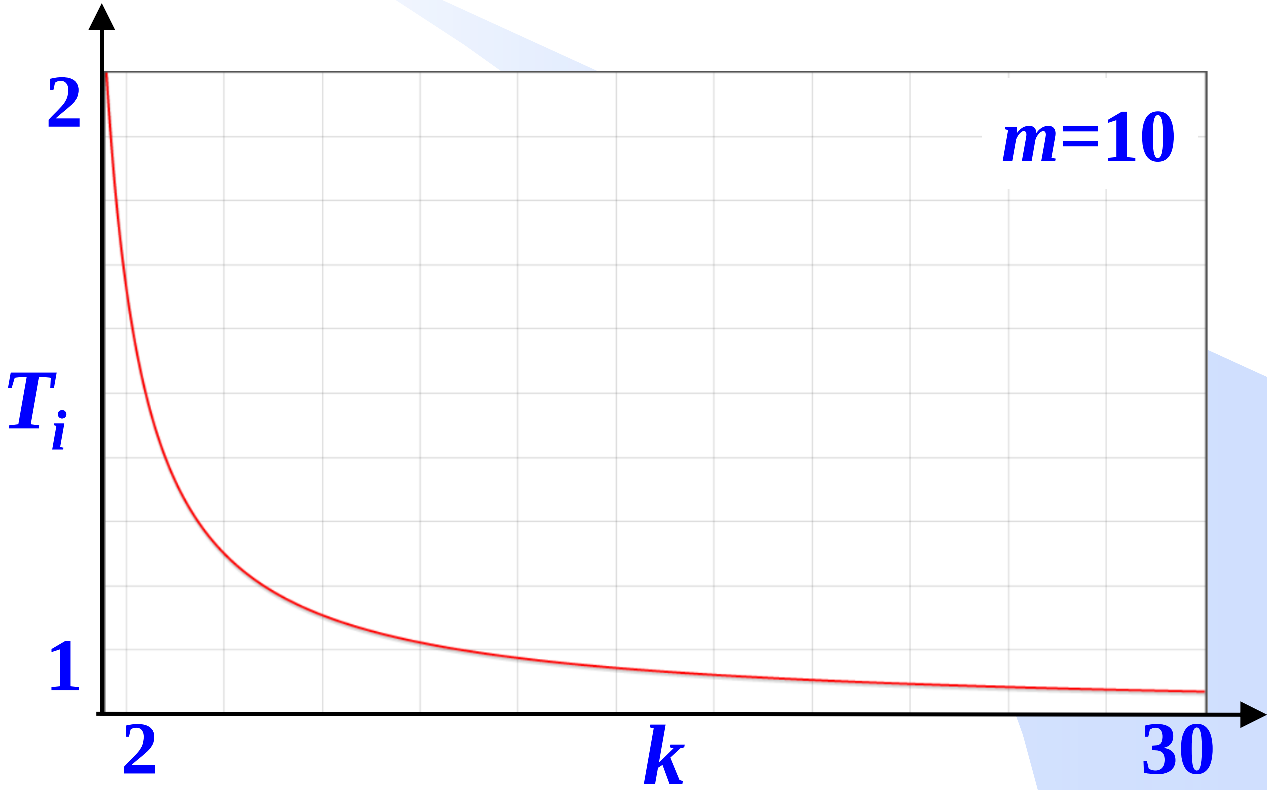
朴素模式匹配算法——平均情况时间复杂度

- 第i趟匹配：看P是否在S的第i个位置
- 由于字符集包含k种字符，故$k \geq 2$

$$T_i = 1 + \frac{1}{k} + \frac{1}{k^2} + \cdots + \frac{1}{k^{m-1}}$$

$$= \frac{1-\left(\frac{1}{k}\right)^m}{1-\frac{1}{k}} \leq 2$$

$$\therefore T \leq nT_i \leq 2n$$

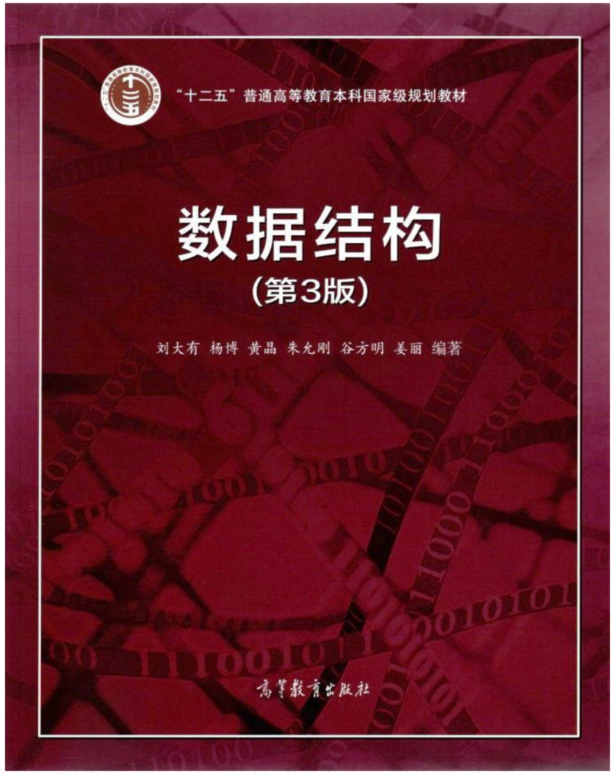


字符集包含的字符种类越多，平均时间复杂度越低

朴素模式匹配算法——总结

- 思路简单、实现简单。
- 缺点：最坏情况下时间复杂度高。
- 字符集越大，单次比对成功的概率越小，时间越接近线性。
- 字符集越小，单次比对成功的概率越高，越接近最坏情况，算法时间性能越差。

S	…	s_i	s_{i+1}	s_{i+2}	…	s_{i+m-2}	s_{i+m-1}	…
P		p_0	p_1	p_2	…	p_{m-2}	p_{m-1}	



字符串匹配

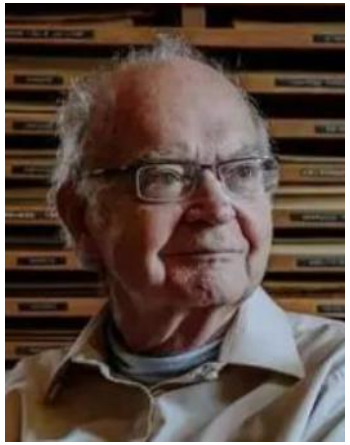
- 模式匹配基本概念
- 朴素模式匹配
- **KMP算法**
- KMP算法的改进

数据之法
结构之美
算法之道

Last updated on 2023.10

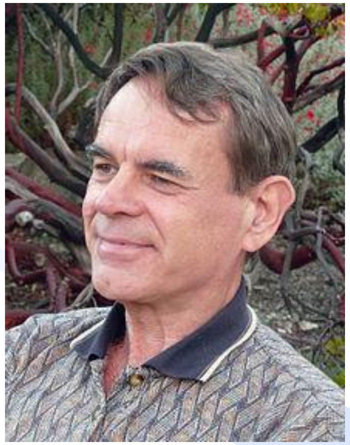
zhuyungang@jlu.edu.cn

快速模式匹配KMP算法

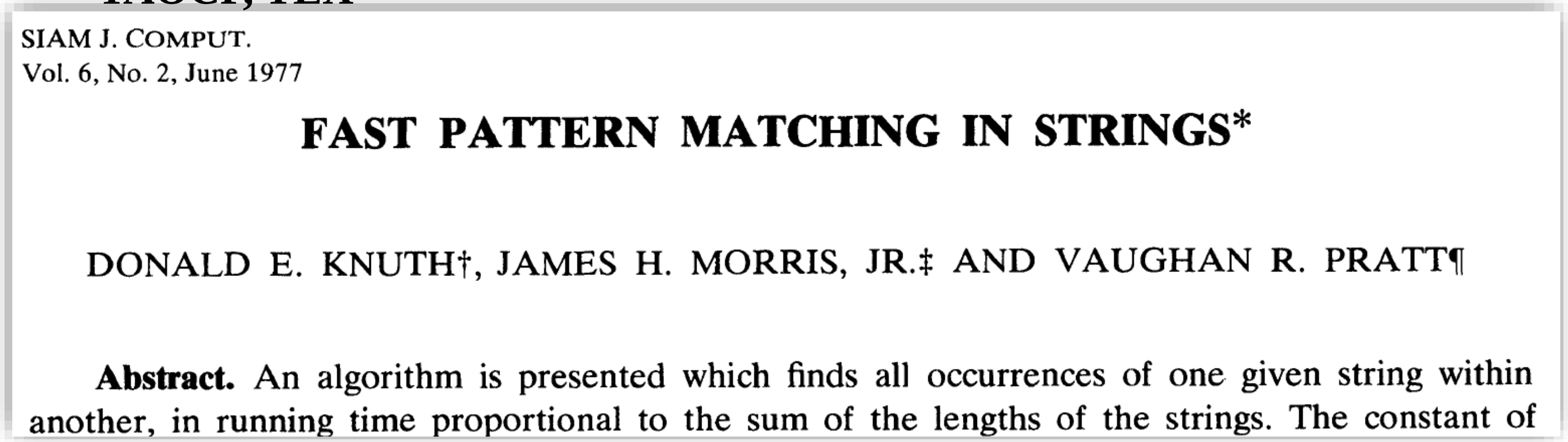


Donald Knuth
斯坦福大学教授
图灵奖获得者
美国科学院院士
美国工程院院士
TAOCP, TEX

James Morris
卡内基梅隆大学教授



Vaughan Pratt
斯坦福大学教授
ACM Fellow

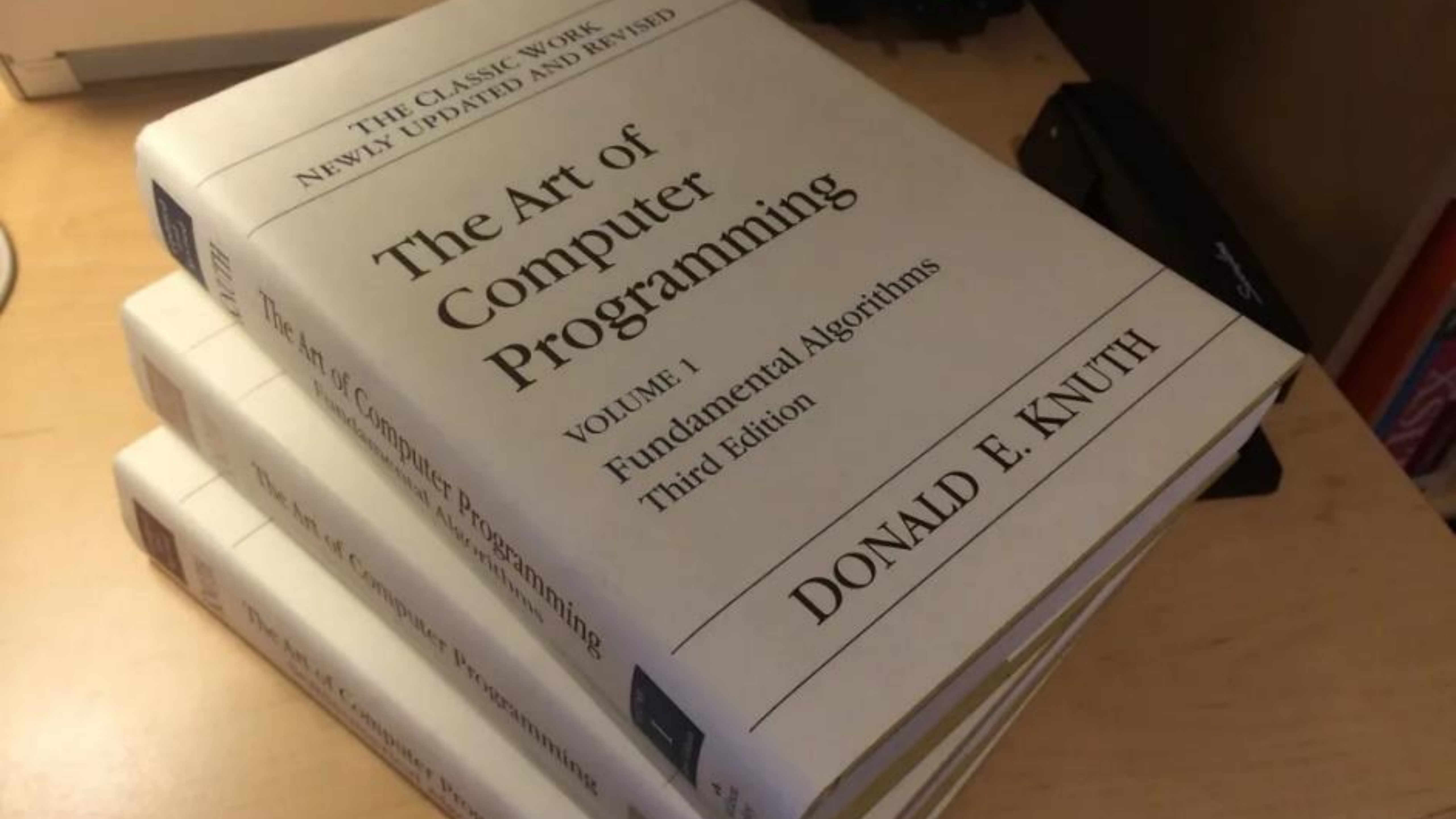


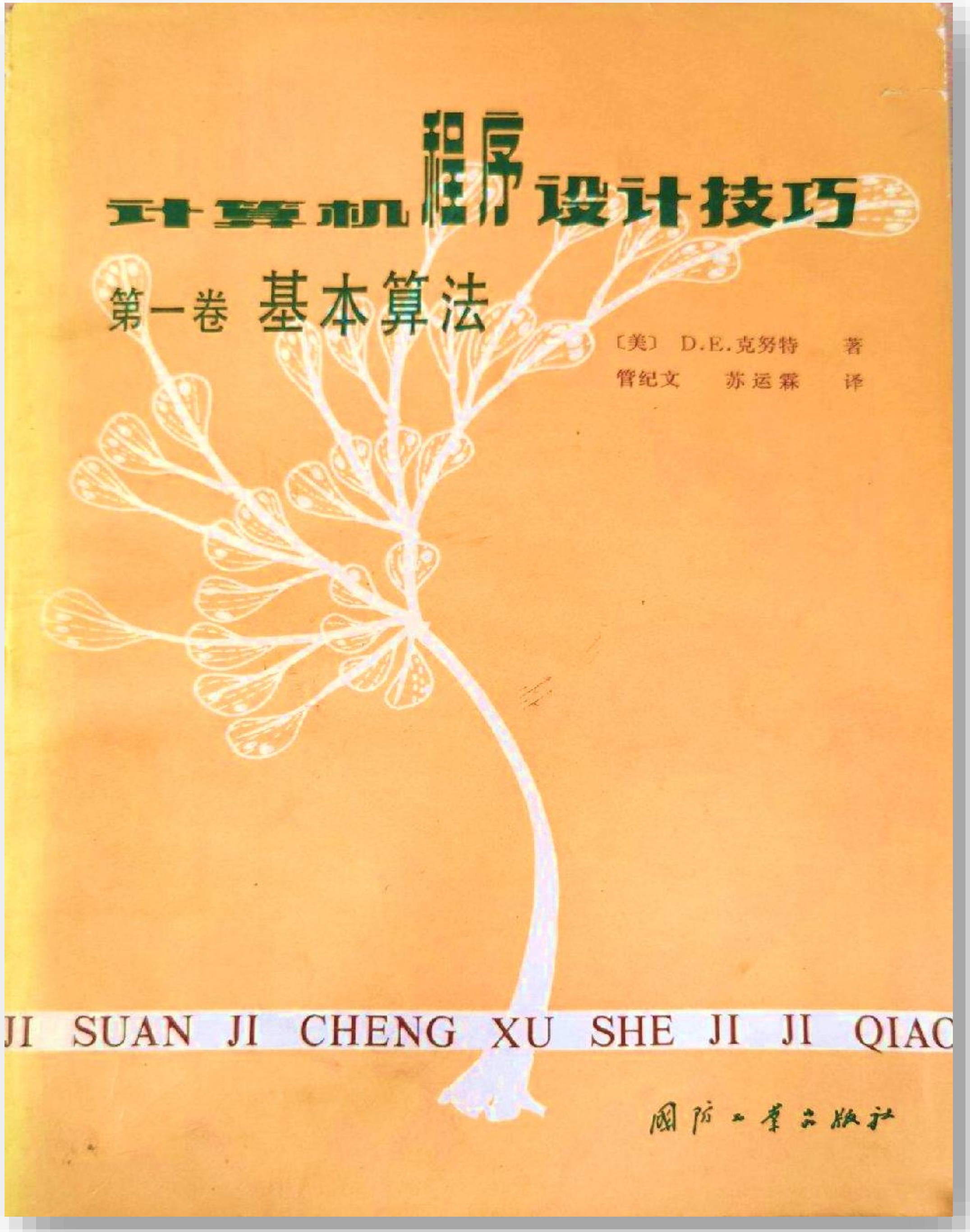
SIAM J. COMPUT.
Vol. 6, No. 2, June 1977

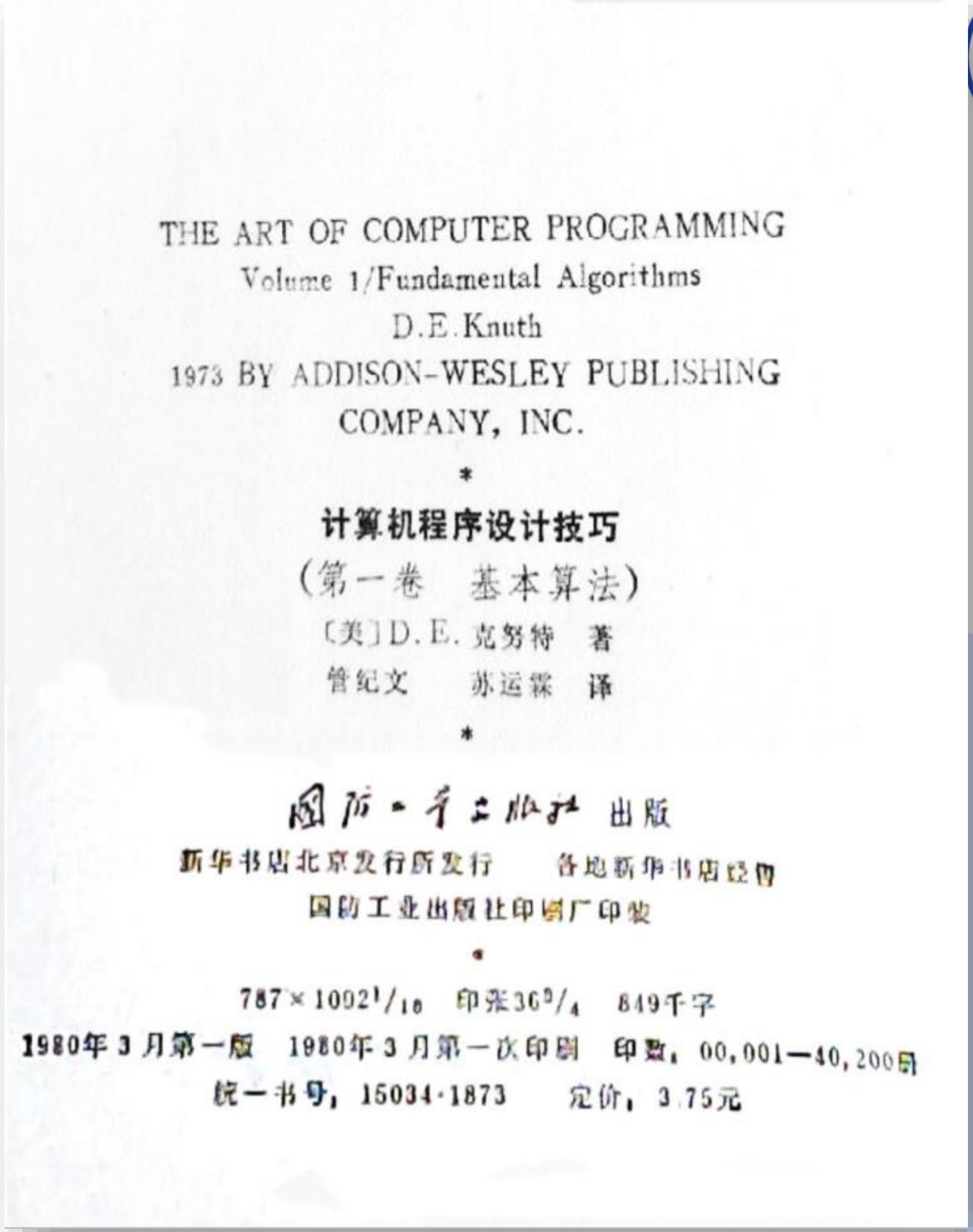
FAST PATTERN MATCHING IN STRINGS*

DONALD E. KNUTH†, JAMES H. MORRIS, JR.‡ AND VAUGHAN R. PRATT¶

Abstract. 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


THE CLASSIC WORK
NEWLY UPDATED AND REVISED
The Art of Computer Programming
VOLUME 1
Fundamental Algorithms
Third Edition
DONALD E. KNUTH
KNUTH
The Art of Computer Programming
Fundamental Algorithms




THE ART OF COMPUTER PROGRAMMING
Volume 1/Fundamental Algorithms
D.E.Knuth
1973 BY ADDISON-WESLEY PUBLISHING COMPANY, INC.

*

计算机程序设计技巧
(第一卷 基本算法)
〔美〕D.E.克努特 著
管纪文 苏运霖 译

*

国防工业出版社 出版
新华书店北京发行所发行 各地新华书店经售
国防工业出版社印刷厂印装

•

787×1092 $^{1}/_{16}$ 印张36 $^{3}/_{4}$ 849千字
1980年3月第一版 1980年3月第一次印刷 印数：00,001—40,200册
统一书号：15034·1873 定价：3.75元

管纪文，原吉林大学计算机科学系主任，中国人工智能学会副理事长，国务院学位委员会学科评议组成员，英国女王大学教授。

苏运霖，原吉林大学计算机科学系教师，吉林省计算机学会秘书长，后调入暨南大学，是暨南大学计算机学科的主要创建者之一。

陆汝钤，中国科学院院士，我国知识工程研究的开拓者和先驱之一，曾任中科院数学所副所长，获吴文俊人工智能最高成就奖、中国计算机学会终身成就奖。

本趟匹配失败的情况

i

↓

	0	1	2	3	4	5	6	7
S :	***b***	***a***	***b***	***a***	***b***	***b***	***a***	***b***

***P*:**	***a***	***b***	***a***	***a***
	0	1	2	3

↑

j

本趟匹配失败的情况

$i \leftarrow i\text{-}j+1$

S:

0	1	2	3	4	5	6	7
b	*a*	*b*	*a*	*b*	*b*	*a*	*b*

P:

a	*b*	*a*	*a*
0	1	2	3

$j \leftarrow 0$

每次失配后：
模式串右移1位
指针 i 回退到 $i\text{-}j+1$
指针 j 回退到0

动机

朴素模式匹配存在的问题（每次匹配失败后）	可能的优化途径
模式串*P*仅右移1位	模式串*P*能否多移几位?
目标串*S*的匹配指针 i 回退	指针 i 可否不回退?
模式串*P*的匹配指针 j 回退至0	指针 j 可否不回退至0?

利用模式串中子串的重复性，加速匹配过程

KMP算法

S: a b c x $\boxed{a\ b\ c}$ $\boxed{x}$ a b c y …

P: $\boxed{a\ b\ c}$ x $\boxed{a\ b\ c}$ $\boxed{y}$

相等前后缀*abc*，下一次比对：*P*右移4位

S: a b c x $\boxed{a\ b\ c}$ $\boxed{x}$ a b c y …

P: $\boxed{a\ b\ c}$ $\boxed{x}$ a b c y

KMP算法

S: $a\ b\ c\ a\ b\ c\ a\ b\ c\ \boxed{a}\ b\ c\ x$

P: $a\ b\ c\ a\ b\ c\ a\ b\ c\ \boxed{x}$

方案1：相等的前后缀*abc*，*P*右移6位

S: $a\ b\ c\ a\ b\ c\ a\ b\ c\ \boxed{a}\ b\ c\ x$

P: $a\ b\ c\ \boxed{a}\ b\ c\ a\ \ldots$

漏掉了可能的成功匹配位置！

KMP算法

S: $a\ b\ c\ a\ b\ c\ a\ b\ c\ \boxed{a}\ b\ c\ x$

P: $a\ b\ c\ a\ b\ c\ a\ b\ c\ \boxed{x}$

方案2：更长的相等的前后缀*abcabc*，*P*右移3位

S: $a\ b\ c\ a\ b\ c\ a\ b\ c\ \boxed{a}\ b\ c\ x$

P: $\qquad\qquad\quad a\ b\ c\ a\ b\ c\ \boxed{a}\ b\ c\ x$

应该保守一些，选择右移位数较少的方案，避免漏掉可能的成功匹配

如何实现？找最长相等的前后缀

KMP算法——基本原理

- 设目标$S = s_0\ s_1 \ldots\ s_{n-1}$，模式串$P = p_0\, p_1 \ldots\ p_{m-1}$
- 假设当前正进行某次匹配，有$s_{i-j} \ldots s_{i-1} = p_0 \ldots p_{j-1}$，但$s_i \neq p_j$，即在$p_j$的位置失配。

S	…	s_{i-j}	…	…	…	s_{i-k}	…	s_{i-1}	s_i	…	s_{n-1}
		=	=	=	=	=	=	=	≠		
P		p_0	…	p_{k-1}	…	p_{j-k}	…	p_{j-1}	p_j		

在失配位置前对应的子串$p[0 \ldots j-1]$中，找最长相等的前后缀，设其长度为k，则有：

$$s[i-k \ldots i-1] = p[j-k \ldots j-1] = p[0 \ldots k-1]$$

KMP算法——基本原理

下次匹配时：

- 右移P使 $p[0…k\text{-}1]$ 与 $s[i–k …i\text{-}1]$对齐，使s_i和 p_k继续比对。
- 目标串匹配指针 i 不动，模式串匹配 j 指针只需回退到k，即模式串右移$j\text{-}k$ 位。

									i		
S	…	$s_{i\text{-}j}$	…	…	…	$s_{i\text{-}k}$	…	$s_{i\text{-}1}$	s_i	…	$s_{n\text{-}1}$
						=	=	=	?		
P						p_0	…	$p_{k\text{-}1}$	p_k	…	
									j		

k是$p_0 … p_{j\text{-}1}$的最长相等前后缀的长度

k标识了模式串P在位置j失配后，下次匹配的位置

在p_j处匹配失败后

P的下一个匹配位置k（P中的哪个字符和S中的失配字符重新匹配）可看做关于j的函数，即 $k = next(j) = p_0 \dots p_{j-1}$的最长相等前后缀的长度。

S	…	s_{i-k}	…	…	…	s_{i-k}	…	s_{i-1}	s_i	…	s_{n-1}
		=	=	=	=	=	=	=	$\neq$		
P		p_0	…	p_{k-1}	p_k	p_{j-k}	…	p_{j-1}	p_j		

在p_j处匹配失败后

***P*的下一个匹配位置*k*（*P*中的哪个字符和*S*中的失配字符重新匹配）可看做关于*j*的函数，** 即 $k = next(j) = p_0 \ldots p_{j-1}$的最长相等前后缀的长度。

S	…	s_{i-k}	…	…	…	s_{i-k}	…	s_{i-1}	s_i	…	s_{n-1}
						=	=	=	?		
P						p_0	…	p_{k-1}	p_k	…	

在的匹配过程中，若在*P*的位置*j*失配，则*P*下次匹配的位置就是*next*(*j*)
next(*j*)：p_j前面的子串的最长相等前后缀的长度

$next$函数（失败函数、前缀函数）

$$next(j)=\begin{cases}-1, & j=0\\ \max\{k \mid p_0\ldots p_{k-1}=p_{j-k}\ldots p_{j-1}\text{且}0<k<j\},\text{存在}k & \\ 0, & \text{其他}\end{cases}$$

$$\underbrace{p_0\ p_1\ \cdots\ p_{k-1}}_{k}\ p_k\ \cdots\ \underbrace{p_{j-k}\ p_{j-k+1}\ \cdots\ p_{j-1}}_{k}$$

$$p_0\ p_1\ \cdots\ p_{k-1} = p_{j-k}\ p_{j-k+1}\ \cdots\ p_{j-1}$$

$next$函数仅和P有关，而与S无关。可以预先把P中所有位置的$next(j)$（$0\le j<m$）算出来存入数组$next[j]$，匹配过程中若在P的位置j失配，则直接令$j\leftarrow next[j]$作为P下次匹配的位置

next函数——例1

a	*x*	*a*	*x*	*a*	*y*	*a*	*x*	*a*	*x*	*b*
										j

$$next\,(\,j\,) = 4$$

next函数——例2

=

a	*b*	*a*	*b*	*a*	*b*	*a*	*b*	*a*	*b*	*x*
										j

$next\,(\,j\,) = 8$

next函数——例3

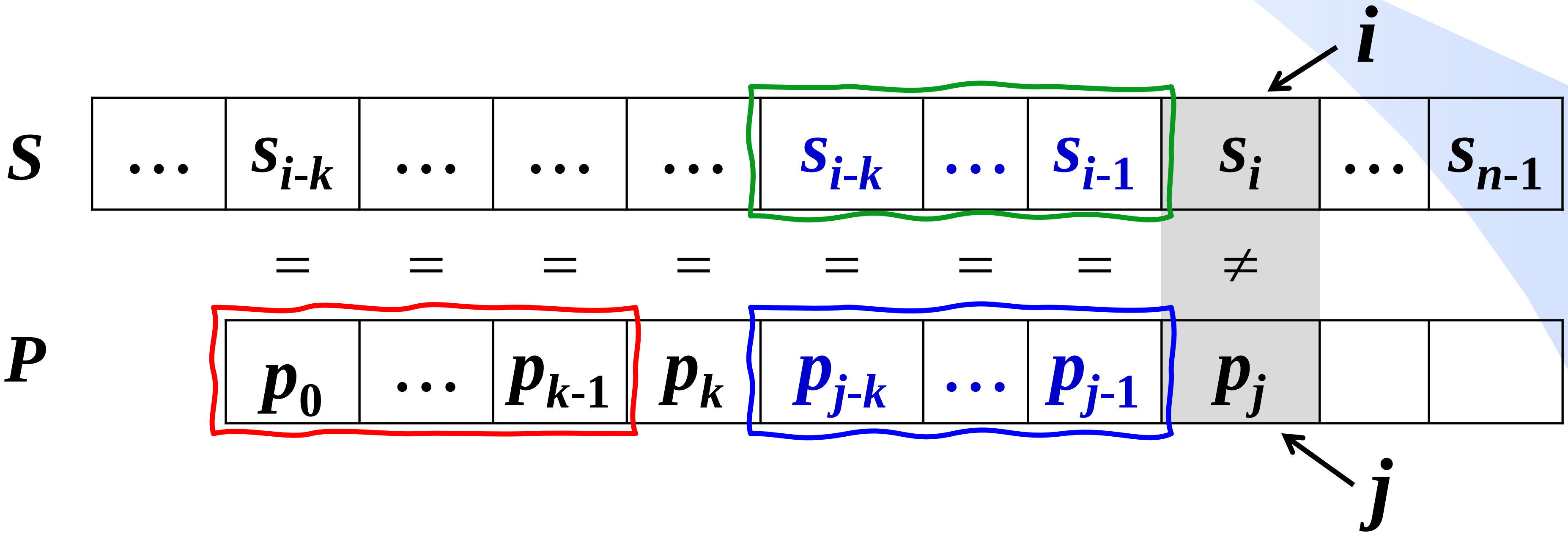
a	*b*	*c*	*d*	*e*	*f*	*g*	*h*	*x*	*y*	*b*
										j

$$next(j) = 0$$

KMP算法——基本过程

通过指针i扫描S，指针j扫描P

- 在p_j (j >0)处失配后：指针i不动，指针$j \leftarrow next[j]$，继续匹配



KMP算法——基本过程

特例：在p_0处失配

➢ 指针 i 右移1位，指针j 置为0，即模式串右移 1 位。

i

	0	1	2	3	4	5	6	7
S:	b	a	b	a	b	b	a	b

	0	1	2	3
P:	b	a	b	b

j

练习

模式串为*abcabcaaa*，使用KMP算法进行模式匹配过程中，若模式串在位置j=7处与目标串匹配失败，则下一次匹配时，应该是模式串的_______位置与目标串失配位置进行匹配。本题下标从0开始。【吉林大学2021级期末考试题】

A. 0　　B. 2　　C. 3　　D. 4

a	*b*	*c*	*a*	*b*	*c*	*a*	*a*	*a*
0	1	2	3	4	5	6	7	8
							j	

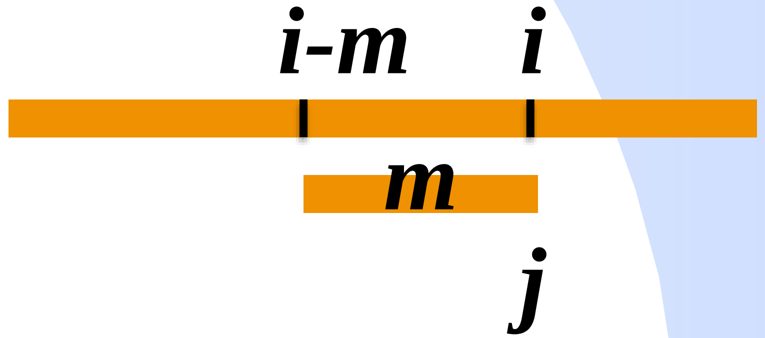
$next(j) = next(7) = 4$

KMP算法

```
const int maxm = 1e4+10;
int KMP(char *S, char *P, int n, int m){
    //目标串S长度为n，模式串P长度为m，返回P在S中的位置
    int next[maxm], i=0, j=0;    //i,j分别为S和P的匹配指针
    buildNext(P, next, m);       //计算失败函数
    while(j<m && i<n) {
        if(j==-1 || S[i]==P[j]) i++, j++;
        else j=next[j];  //确定P下次匹配位置
    }
    return (j==m)? i-m : -1;
}
```

匹配成功

匹配失败



*next*函数——练习

j =	0	1	2	3	4	5	6
P =	a	b	c	a	b	c	d
$next(j)$ =	-1	0	0	0	1	2	3

编程实现 next (j)的计算——暴力方法

- 在 $p_0 p_1 \ldots p_{j\text{-}1}$ 中找最长相等的前后缀
- 暴力方案：从长到短考察每个前缀，看能否匹配后缀。
- 例如：模式串　 *a b a a b*

长度为4的前缀 *a b a a*	*b a a b* 长度为4的后缀
长度为3的前缀 *a b a*	*a a b* 长度为3的后缀
长度为2的前缀 *a b*	*a b* 长度为2的后缀
长度为1的前缀 *a*	*b* 长度为1的后缀

$O(m^2)$

函数*next* (*j*)的计算——高效方法

在 $p_0\ p_1 \ldots p_{j-1}$ 中找出**最长相等前后缀**，即找最大的k，使得：

$$p_0 \ldots p_{k-1} = p_{j-k} \ldots p_{j-1}$$

递推计算*next*函数，令*next*(0) = -1, *next* (1) = 0, 假定已知 *next*(*j*)=*k*, 求 *next* (*j*+1)

已知 $next(j)=k$, 求 $next(j+1)$

$next(j)=k$，即$p_0 \ldots p_{j-1}$的最长相等前后缀为$p_0 \ldots p_{k-1} = p_{j-k} \ldots p_{j-1}$，长度为$k$

p_0	…	…	p_{k-1}	p_k	…	p_{j-k}	…	…	p_{j-1}	p_j

如果$p_k = p_j$

$next(j+1)$即$p_0 \ldots p_j$的最长相等的前后缀长度比以前增加1

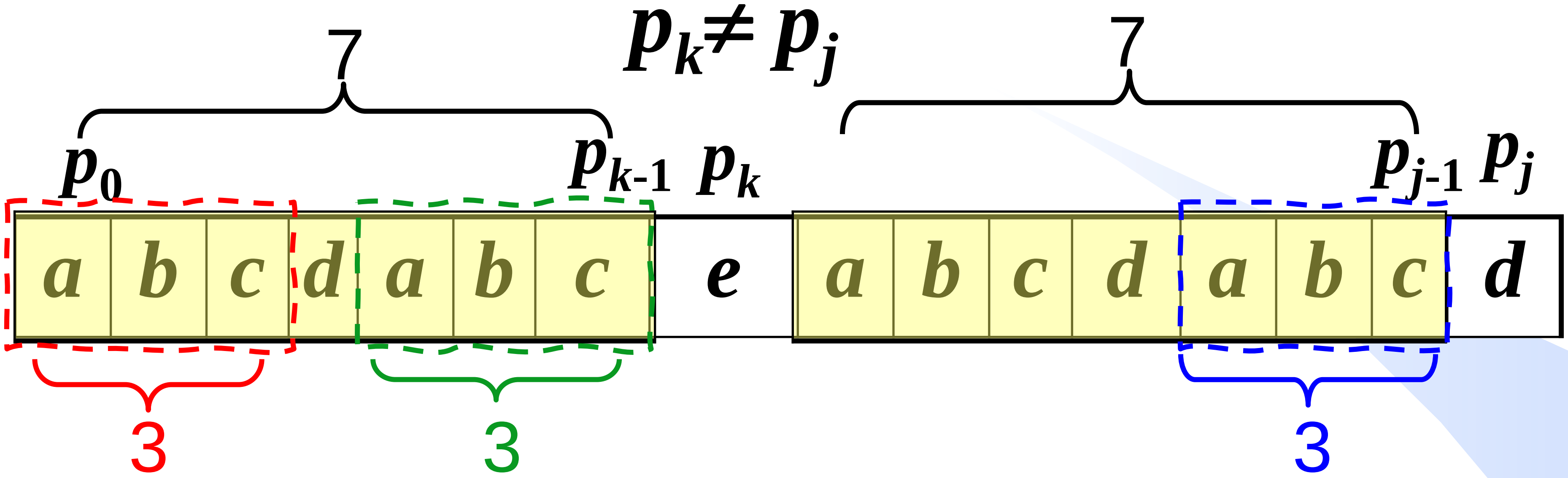
$next(j+1)=k+1$

已知 $next(j)=k$, 求 $next(j+1)$

p_0	…	…	p_{k-1}	$\cancel{p_k}$	…	p_{j-k}	…	…	p_{j-1}	$\cancel{p_j}$

如果$p_k \neq p_j$

已知 $next(j)=k$，求 $next(j+1)$



考察$p_0 \ldots p_{j-1}$中稍短一点（第2长）的相等前后缀，然后比较这两个前后缀的下一个字符

$p_0 \ldots p_{j-1}$的第2长相等前后缀

$p_0 \ldots p_{k-1}$的最长相等前后缀

已知 $next(j)=k$，求 $next(j+1)$

$$\underbrace{p_0 \cdots p_{h-1}}_{h}\ p_h \cdots \underbrace{p_{k-h} \cdots p_{k-1}}_{h}\ p_k\ \cdots\ p_{j-k}\ \cdots\ \underbrace{p_{j-h} \cdots p_{j-1}}_{h}\ p_j$$

若$p_k \neq p_j$，则考察$p_0 \ldots p_{j-1}$的稍短一点（第2长）相等前后缀：即寻找k以内的最大h，使$p_0 \cdots p_{h-1} = p_{j-h} \cdots p_{j-1}$，而$p_{j-h} \cdots p_{j-1} = p_{k-h} \cdots p_{k-1}$，即等价于找$p_0 \ldots p_{k-1}$的最长相等前后缀$p_0 \ldots p_{h-1}$和$p_{k-h} \ldots p_{k-1}$，故$h=next(k)$.

若$p_h = p_j$，则$p_0 \ldots p_j$的最大相等前后缀即$p_0 \ldots p_h$和$p_{j-h} \ldots p_j$，于是$next(j+1)=h+1$.

已知 $next(j)=k$, 求 $next(j+1)$

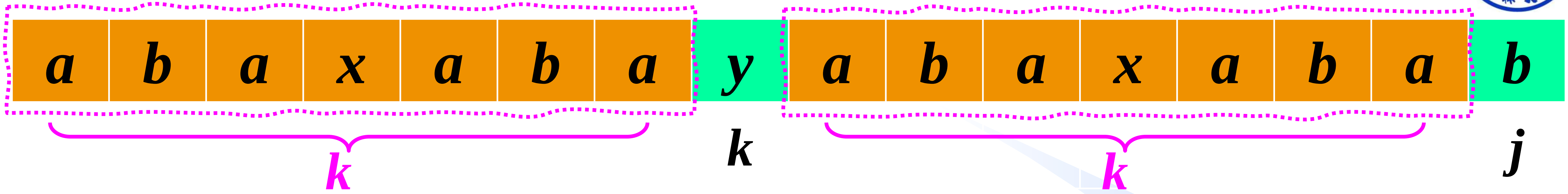
$$\underbrace{p_0 \cdots p_{r-1}}_{r}\ p_r \cdots \underbrace{p_{h-r} \cdots p_{h-1}}_{r}\ p_h\ \cdots\ p_{j-h}\ \cdots\ \cdots\ \underbrace{p_{j-r}\ \cdots\ p_{j-1}}_{r}\ p_j$$

若$p_h \neq p_j$，则考察$p_0 \ldots p_{j-1}$再稍短一点（第3长）的相等前后缀：实际上等价于是找$p_0 \ldots p_{h-1}$的最长相等前后缀$p_0 \ldots p_{r-1}$和$p_{h-r} \ldots p_{h-1}$，故$r = next(h)$

若$p_r = p_j$，则 $next(j+1)=r+1$

若$p_r \neq p_j$，则考察再稍短一点（第4长）的相等前后缀

以此类推……



$k \leftarrow next[j]$ 若$P[k]==P[j]$ 则 $next[j+1] \leftarrow k+1$，否则

a	*b*	*a*	*x*	*a*	*b*	*a*	*y*	*a*	*b*	*a*	*x*	*a*	*b*	*a*	*b*
	h		*h*				*k*						*h*		*j*

$k \leftarrow next[j]$ 若$P[k]==P[j]$ 则 $next[j+1] \leftarrow k+1$，否则

$h \leftarrow next[k]$ 若$P[h]==P[j]$ 则 $next[j+1] \leftarrow h+1$，否则

a	*b*	*a*	*x*	*a*	*b*	*a*	*y*	*a*	*b*	*a*	*x*	*a*	*b*	*a*	*b*
$\underbrace{}_{r}$	*r*	*h*					*k*							$\underbrace{}_{r}$	*j*

$k \leftarrow next[j]$ 若$P[k]==P[j]$ 则 $next[j+1] \leftarrow k+1$，否则

$h \leftarrow next[k]$ 若$P[h]==P[j]$ 则 $next[j+1] \leftarrow h+1$，否则

$r \leftarrow next[h]$ 若$P[r]==P[j]$ 则 $next[j+1] \leftarrow r+1$，否则

……

如何编程实现?
循环

如何编程实现——循环

a	*b*	*a*	*x*	*a*	*b*	*a*	*y*	*a*	*b*	*a*	*x*	*a*	*b*	*a*	*b*
0							*k*							*j*-1	*j*

$k \leftarrow next[j]$ 若$P[k]==P[j]$ 则 $next[j+1] \leftarrow k+1$， 否则

如何编程实现——循环

a	*b*	*a*	*x*	*a*	*b*	*a*	*y*	*a*	*b*	*a*	*x*	*a*	*b*	*a*	*b*

$k \leftarrow$ - - - - - - - - - - - - k $\qquad\qquad$ j

$k \leftarrow next[j]$ 若$P[k]==P[j]$ 则 $next[j+1] \leftarrow k+1$，否则

$k \leftarrow next[k]$ 若$P[k]==P[j]$ 则 $next[j+1] \leftarrow k+1$，否则

如何编程实现——循环

a	*b*	*a*	*x*	*a*	*b*	*a*	*y*	*a*	*b*	*a*	*x*	*a*	*b*	*a*	*b*

k ←----- *k* ←------------- *k* *j*

$k \leftarrow next[j]$ 若 $P[k]==P[j]$ 则 $next[j+1] \leftarrow k+1$，否则

$k \leftarrow next[k]$ 若 $P[k]==P[j]$ 则 $next[j+1] \leftarrow k+1$，否则

$k \leftarrow next[k]$ 若 $P[k]==P[j]$ 则 $next[j+1] \leftarrow k+1$，否则

……

```
k=next[j];
while(k>=0 && P[k]!=P[j])
    k = next[k];
next[j+1] = k+1;
```

next数组的计算

p_0	…	…	p_{k-1}	p_k	…	p_{j-k}	…	…	p_{j-1}	p_j

```
void buildNext(char *P, int next[], int m){
    int k=next[0]=-1;
    for(int j=0; j<m-1; j++){ //求next[j+1], j+1<m
        k=next[j]; //此句可省去
        while(k>=0 && P[k]!=P[j])
            k=next[k]; //求p0…pk-1的最长相等前后缀
        next[j+1]=++k;
    }
}
```

next数组的计算——更简洁版本

p_0	…	…	p_{k-1}	p_k	…	p_{j-k}	…	…	p_{j-1}	p_j

```
void buildNext(char *P, int next[], int m){
    int k=next[0]=-1;
    for(int j=0; j<m-1; j++){ //求next[j+1],此刻必有k=next[j]
        while(k>=0 && P[k]!=P[j])
            k=next[k];
        next[j+1]=++k;
    }
}
```

*next*数组的计算——时间复杂度分析

```
void buildNext(char *P, int next[], int m){
    int k=next[0]=-1;
    for(int j=0; j<m-1; j++){   //求next[j+1]
        while(k>=0 && P[k]!=P[j])
            k=next[k];
        next[j+1]=++k;
    }
}
```

关键运算
字符比较

使*k*减小

使*k*增加

执行次数取决于 k=next[k] 和 next[j+1]=++k 的次数

- 字符比较次数取决于*k*增加次数和*k*减小次数之和。
- ++*k*在for循环内，最多执行*m*次，即*k*最多增加*m*次，每次加1.
- *k*初值-1，迭代过程中永远不会小于-1，故*k*减小（*k* =*next* [*k*]）的次数不会超过*k*增加的次数（即*m*次），否则*k*就比 -1小了。
- 故关键运算不会超过2*m*次，即O(*m*)

KMP算法——时间复杂度分析

关键运算
元素比较

```
int KMP(char *S, char *P, int n, int m){
    int next[N], i=0, j=0;
    buildNext(P, next, m);  //O(m)
    while(j<m && i<n)
        if(j==-1 || S[i]==P[j]) i++, j++;
        else j=next[j];
    return (j==m)? i-m:-1 ;
}
```

while循环迭代次数不仅取决于i，还取决于j

使j增加

使j减小

*KMP*算法总时间复杂度O(n+m)

- 扫描过程中i最多从0至n，i在循环内永远不减小，最多增加n次，每次加1，而j++与i++同步，故j也最多增加n次，每次加1.
- j最小是从-1开始增加，即j++的最小起点是-1，迭代过程中永远不会小于-1，故j减小（j =*next*[j]）的次数不会超过j增加的次数，即n次.
- 故关键运算不会超过2n次，即O(n)

拓展

给定目标串*S*和模式串*P*，输出*P*在*S*中出现的次数，以及*P*在*S*中的所有匹配位置【吉林大学20级数据结构上机考试题】

```
int KMP(char S[], char P[], int n, int m){//S长度n, P长度m
    int next[N],i=0,j=0,cnt=0; // cnt为P在S中出现次数
    buildNext(P, next, m); //计算next数组,需要多算一位到next[m]
    while(j<m && i<n) {
       if(j==-1 || S[i]==P[j]) i++, j++;
       else j=next[j];  //确定P下次匹配位置
       if (j == m) {  //匹配成功
          cnt++; printf("%d\n", i-m); //输出此时P在S中位置
          j=next[j];// “假装”此处失配，重新确定下次匹配位置
       }
    }
    return cnt;
}
```

KMP算法的隐藏应用举例

字符串长度为n

- 最长相等的前后缀长度：$next[n]$.
- 第二长相等的前后缀长度：$next[next[n]]$.
- 最长重复前缀：$next$数组中的最大值$\max_i(next[i])$.

a b c a b c a b c x x x

p_0	…	…	p_{k-1}	…	p_{i-k}	…	p_{i-1}	p_i	…	…

KMP算法的隐藏应用举例

给定长度为n的字符串p，编写算法返回p的最长回文前缀的长度，如果p本身就是回文，则返回p的长度。【搜狗校园招聘笔试题】

- 假定p的最长回文前缀为$p_0 \ldots p_k$，则有$p_0 \ldots p_k = p_k \ldots p_0$

$$\underbrace{p_0 \ldots p_k \ldots p_{n-1}}_{p} \quad \# \quad \underbrace{p_{n-1} \ldots p_k \ldots p_0}_{p^{-1}}$$

KMP算法的隐藏应用举例

给定长度为n的字符串p，在p前面最少添加几个字符可使p变为回文串？并输出该最短回文串。【字节跳动、华为、腾讯、爱奇艺、微软、谷歌面试题LeetCode214】

- 假定p的最长回文前缀为$p_0 \dots p_k$

$$p_{n-1} \cdots p_{k+1}\, \color{red}{p_0 \cdots p_k}\, p_{k+1} \cdots p_{n-1}$$

KMP算法隐藏应用举例——循环节

如果一个字符串*S*可由它的一个最小子串*P*重复*k*次构成，则*P*称为*S*的循环节，*k*称为循环周期。例如：

$$S= a\ b\ c\ a\ b\ c\ a\ b\ c\ a\ b\ c$$

循环节（最小重复单元）为 *a b c*，循环周期为4.

KMP算法隐藏应用举例——循环节

给定一个非空字符串*S*，检查它可否可以通过由它的一个子串重复多次构成。【腾讯、字节跳动、快手、谷歌、携程、招商银行、浦发银行面试题LeetCode459】

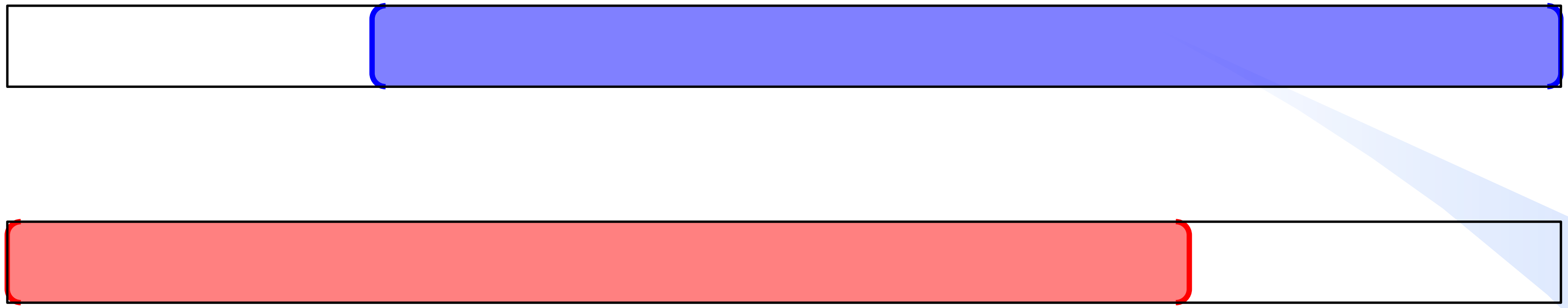
例如：

***S*= *a b c a b c a b c a b c* ，输出True**

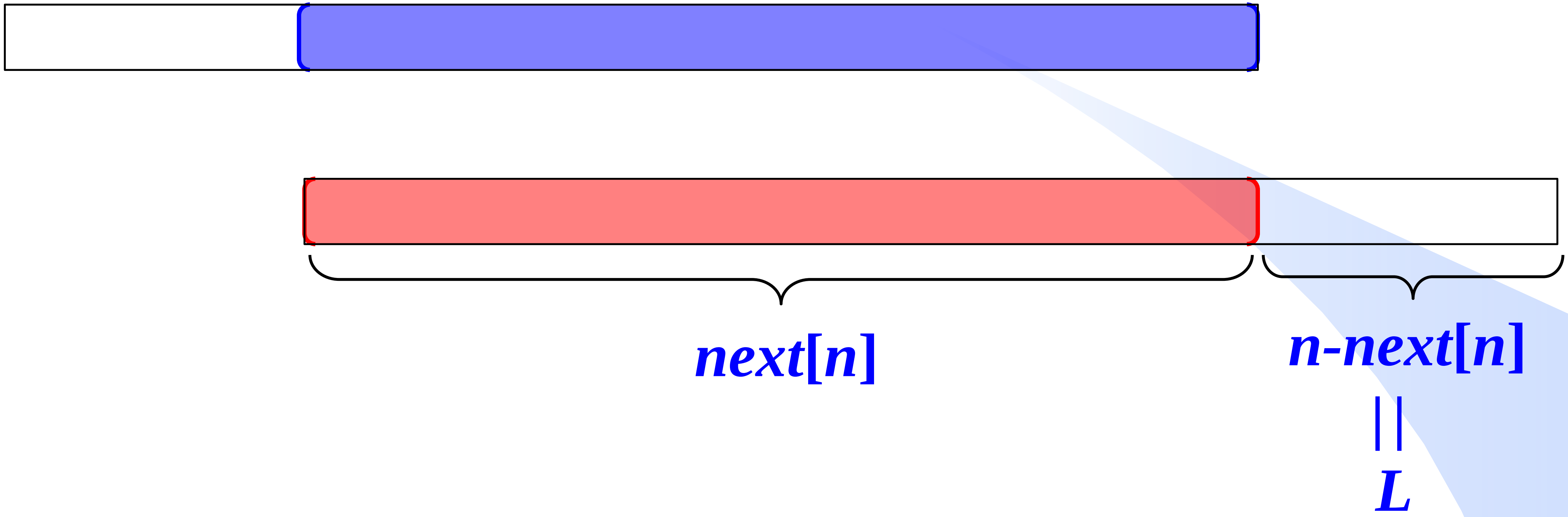
***S*= *a b a* ，输出False**

KMP算法隐藏应用举例——循环节

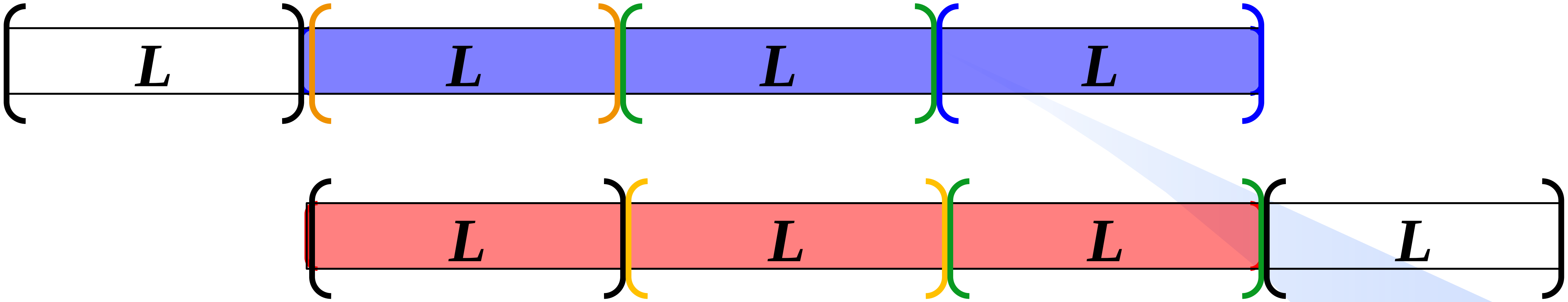
KMP算法隐藏应用举例——循环节



KMP算法隐藏应用举例——循环节



KMP算法隐藏应用举例——循环节



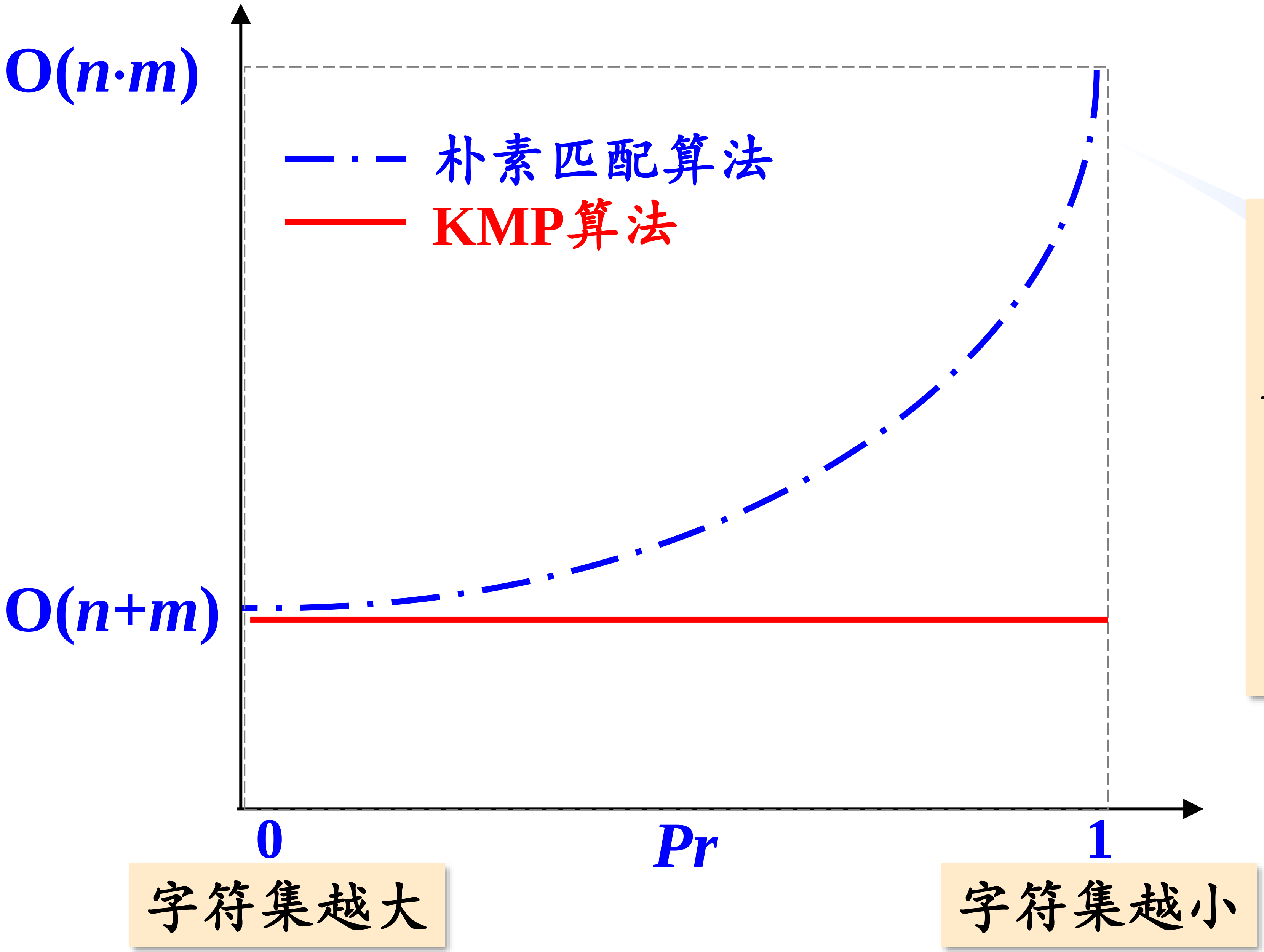
- 如果$next[n]>0$且$n\%L==0$，则S可由循环节完全循环构成：
- 循环节长度为：$L=n-next[n]$，循环周期为： n/L

KMP算法隐藏应用举例——循环节

- 如果*next*[*n*]>0且*n*%*L*==0，则S可由循环节完全循环构成：
- 循环节长度为：*L*=*n*-*next*[*n*]，循环周期为： *n*/*L*

```
const int N=1e4+10;
bool repeatedSubstringPattern(char * s){
    int next[N], n=strlen(s);
    buildNext(s,next,n);  //next数组要多算一位，算到next[n]
    int L=n-next[n];
    if(next[n]>0 && n%L==0) return true;
    return false;
}
```

总结



字符集越小，字符单次成功比对概率 *Pr* 越高，KMP 算法相对于朴素匹配算法优势越大

$$Pr = 1/k$$

k为字符集大小

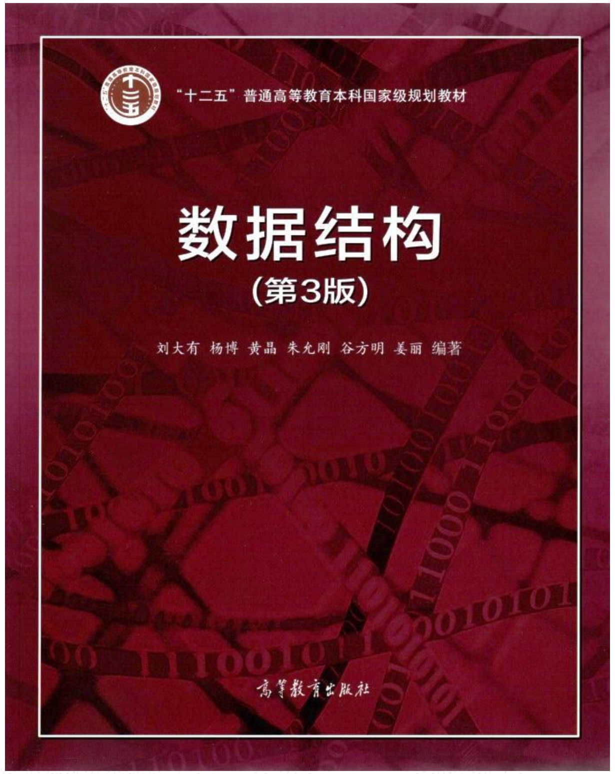
$next(j)$与教材中失败函数$f(j)$的关系

$next(j)$：$p_0 \ldots p_{j\text{-}1}$ 最长相等前后缀的长度

$f(j)$：$p_0 \ldots p_j$ 最长相等前后缀的长度减1

$$f(j) = next(j+1) - 1$$

$$next(j) = \begin{cases} -1, & j = 0 \\ f(j-1) + 1, & j \geq 1 \end{cases}$$



字符串模式匹配

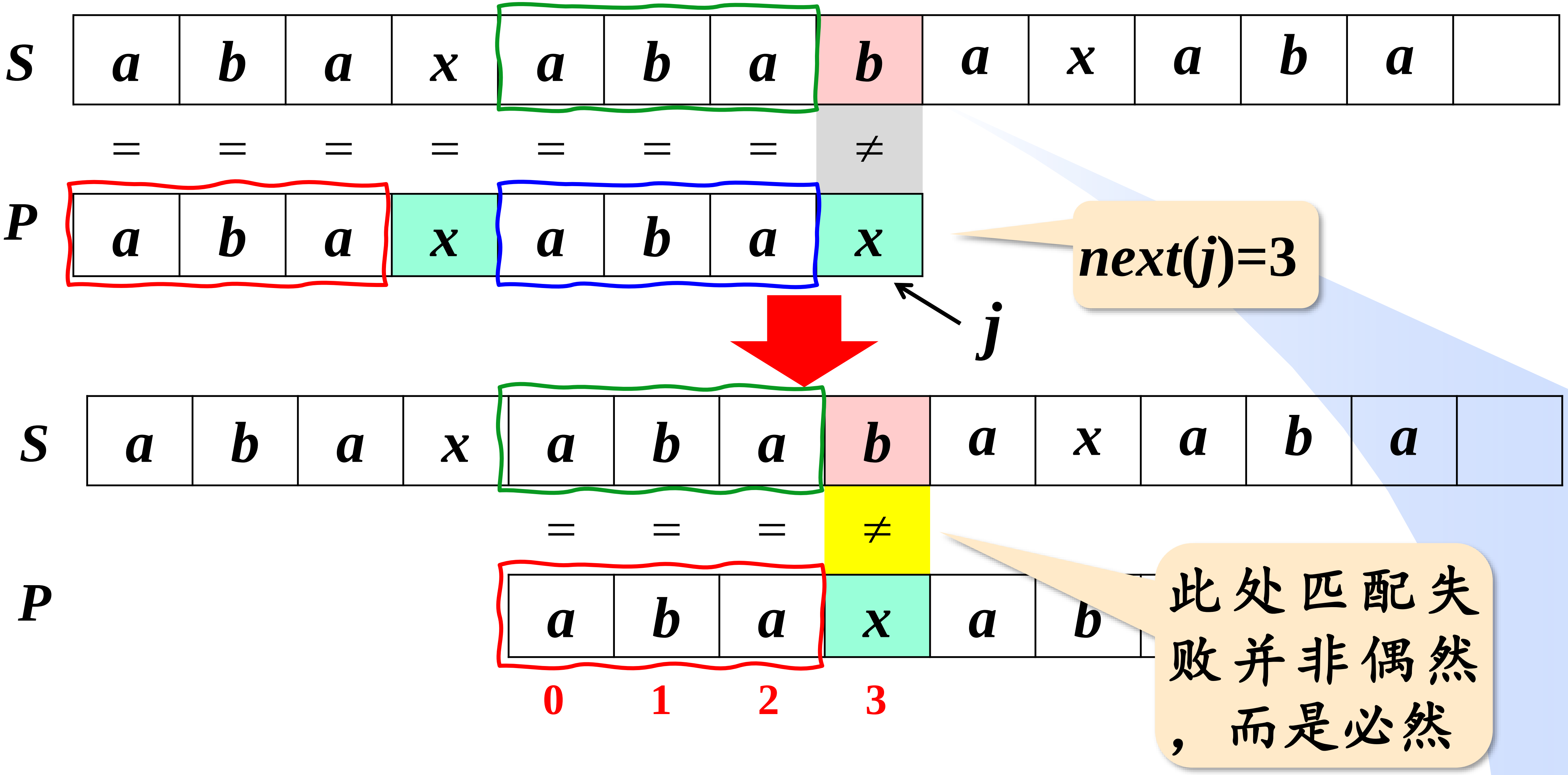
- 模式匹配基本概念
- 朴素模式匹配算法
- KMP算法
- **KMP算法的改进**

数据之法
结构之美
算法之道

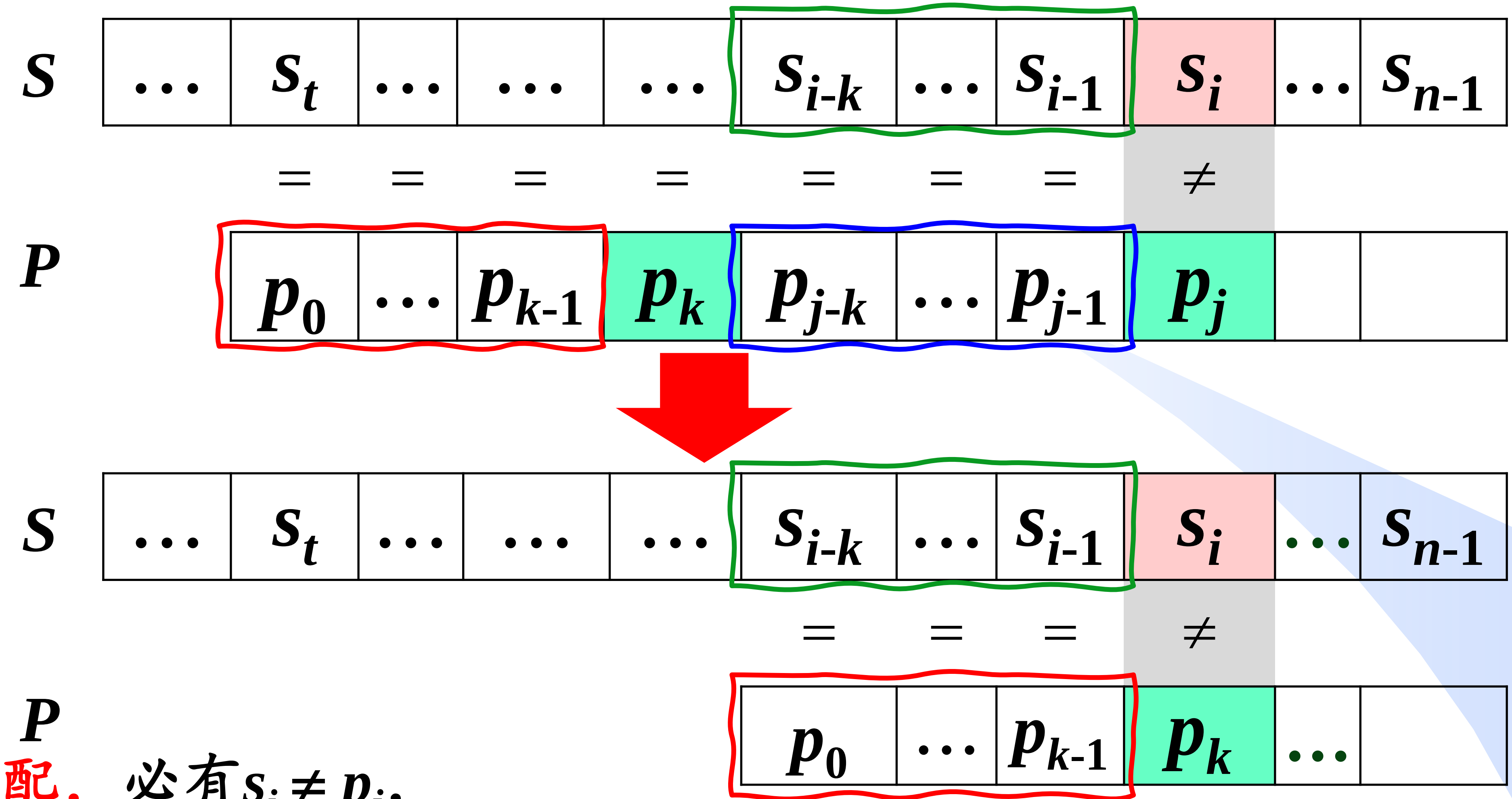
Last updated on 2023.10

zhuyungang@jlu.edu.cn

原始next函数的不足



只利用了过往成功匹配的经验，而失败匹配的教训未充分利用

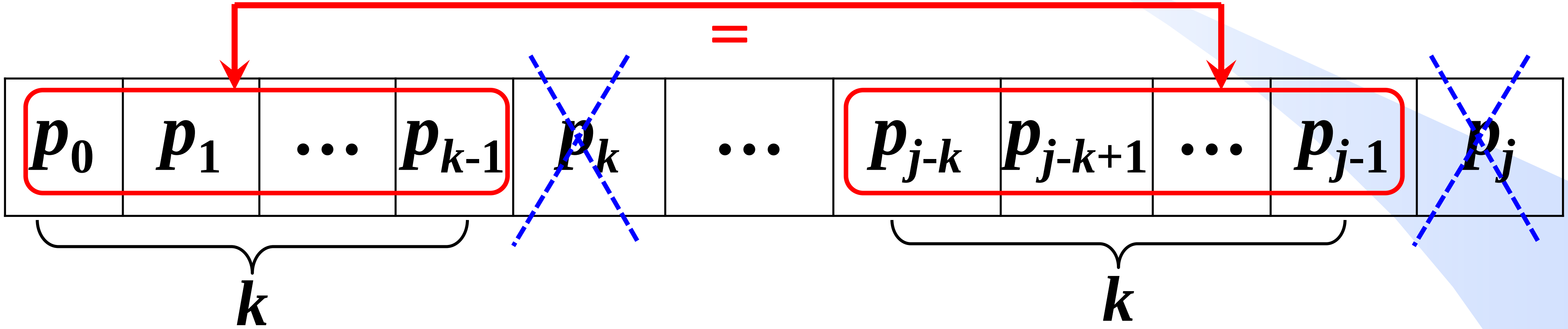


在p_j处失配，必有$s_i \neq p_j$，

- 如果$p_j = p_k$，则必有$s_i \neq p_k$，下趟匹配必然失败；
- 改进方法：在确定下次匹配位置时，除了找失配位置前的最长相等前后缀之外，还应比较其后面的两个字符（即前缀后面的字符p_k和当前失配位置的字符 p_j），确保$p_j \neq p_k$

改进的next(j)函数

额外要求：最长相等前后缀后面的两个字符（即前缀后面的字符p_k和当前位置j的字符p_j）不能相等，即$p_j \neq p_k$



$$next2(j)=\begin{cases} -1, & j=0 \\ \max\{k \mid p_0\dots p_{k-1}=p_{j-k}\dots p_{j-1}\text{且}p_k\neq p_j, 0<k<j\},\text{存在}k \\ 0, & p_0\dots p_{j-1}\text{不存在满足上述条件的前后缀但}p_0\neq p_j \\ -1, & p_0\dots p_{j-1}\text{不存在满足上述条件的前后缀且}p_0=p_j \end{cases}$$

改进的$next(j)$函数

$$next2(j)=\begin{cases}-1, & j=0\\ \max\{k \mid p_0\ldots p_{k-1}=p_{j-k}\ldots p_{j-1}且p_k\neq p_j, 0<k<j\},存在k\\ 0, & p_0\ldots p_{j-1}不存在满足上述条件的前后缀但p_0\neq p_j\\ -1, & p_0\ldots p_{j-1}不存在满足上述条件的前后缀且p_0=p_j\end{cases}$$

S	a	b	c	a	b	c	x
P	a	b	c	x			
P				a	b	c	x

至少可以将P_0移动到失配位置，与S进行下一次比对

改进的next(j)函数

$$next2(j)=\begin{cases}-1, & j=0\\ \max\{k \mid p_0\ldots p_{k-1}=p_{j-k}\ldots p_{j-1}\text{且}p_k\neq p_j, 0<k<j\},\text{存在}k\\ 0, & p_0\ldots p_{j-1}\text{不存在满足上述条件的前后缀但}p_0\neq p_j\\ -1, & p_0\ldots p_{j-1}\text{不存在满足上述条件的前后缀且}p_0=p_j\end{cases}$$

S	a	b	c	b	a	b	c	a
P	a	b	c	a				
P					a	b	c	a

P_j前所有位置都不可能与S匹配，应将P整体移过失配位置

例1

a	*x*	*a*	*x*	*a*	*y*	*a*	*x*	*a*	*x*	*b*

$=$

j

$\neq$

$next2\,(\,j\,) = 4$

例2

a	*x*	*a*	*x*	*b*	*y*	*a*	*x*	*a*	*x*	*b*
										j

例2（续）

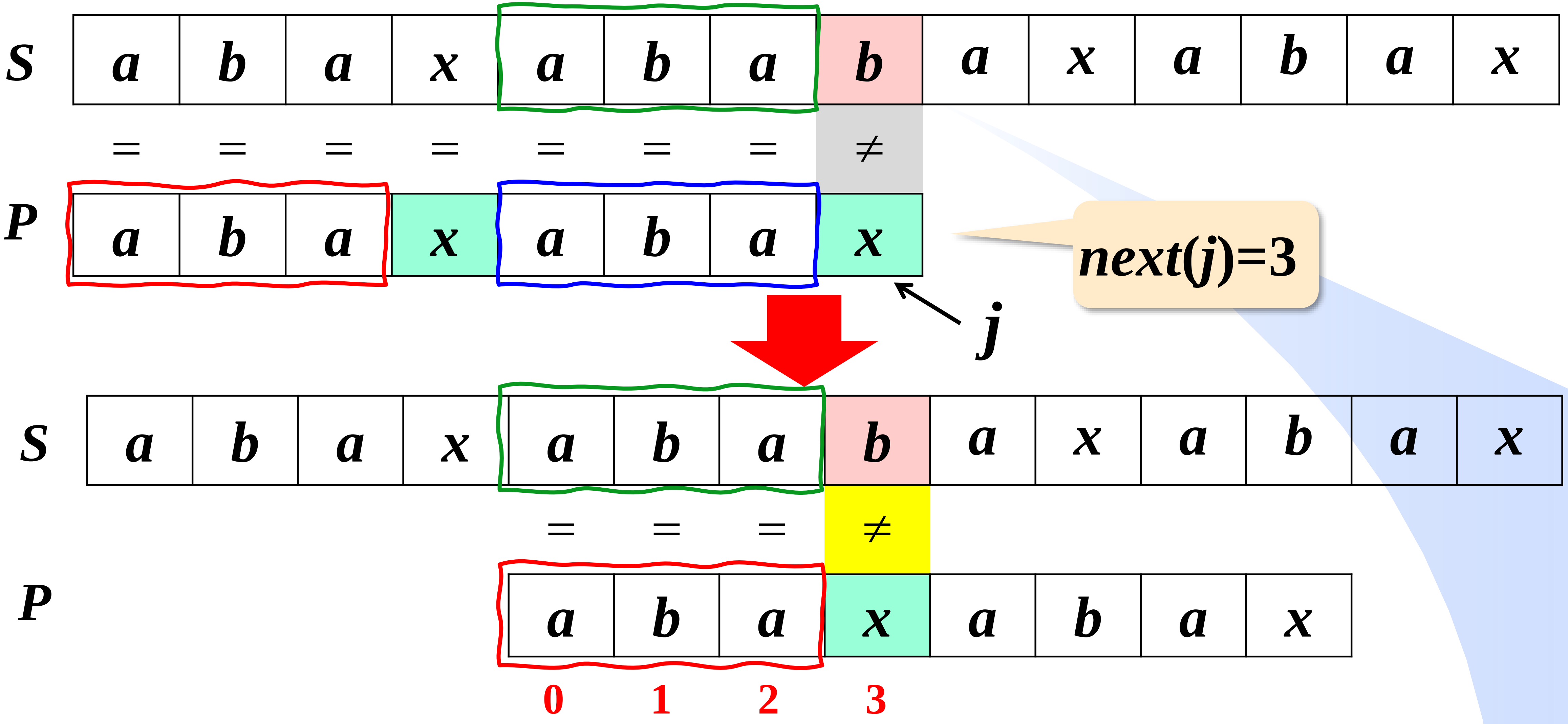
$=$

a	x	a	x	b	y	a	x	a	x	b
										j

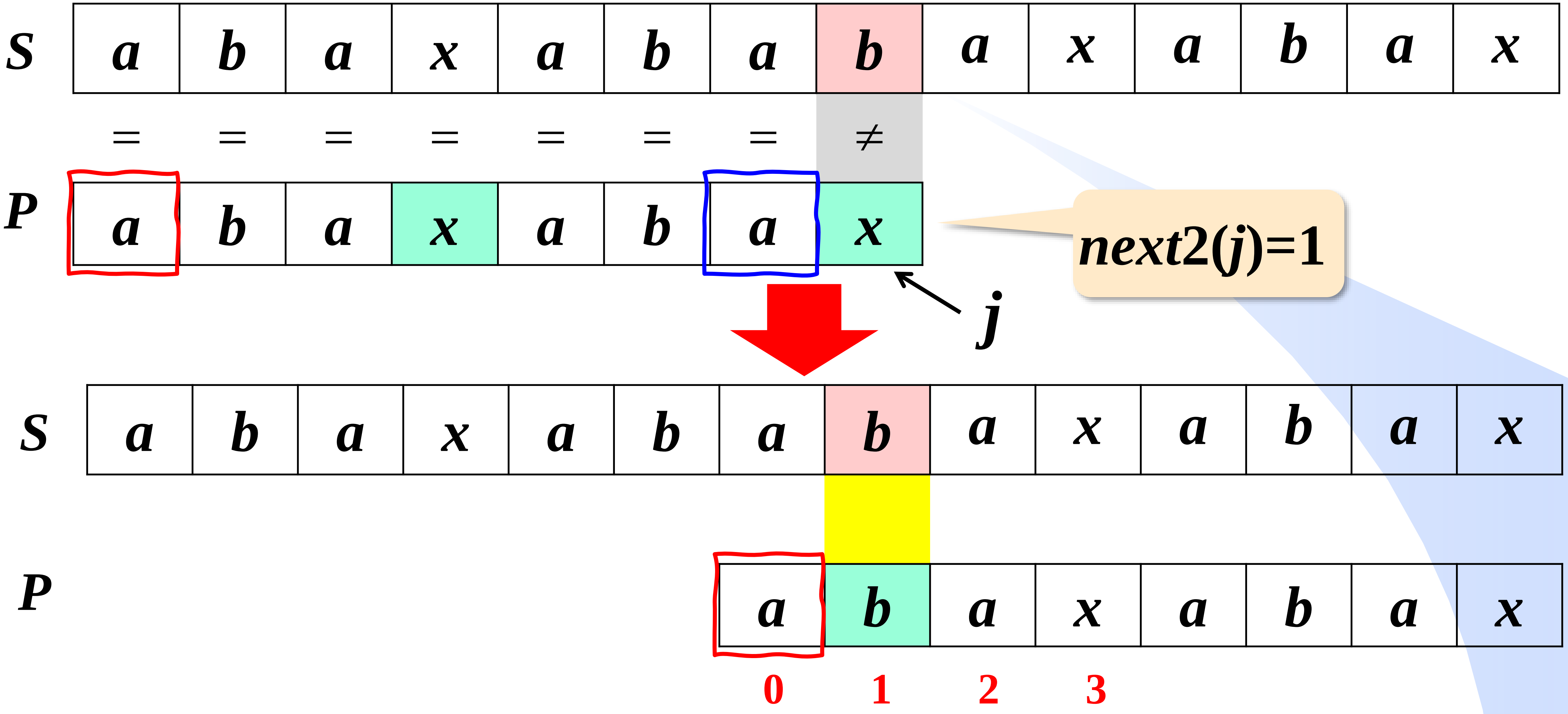
$\neq$

$$next2\,(\,j\,) = 2$$

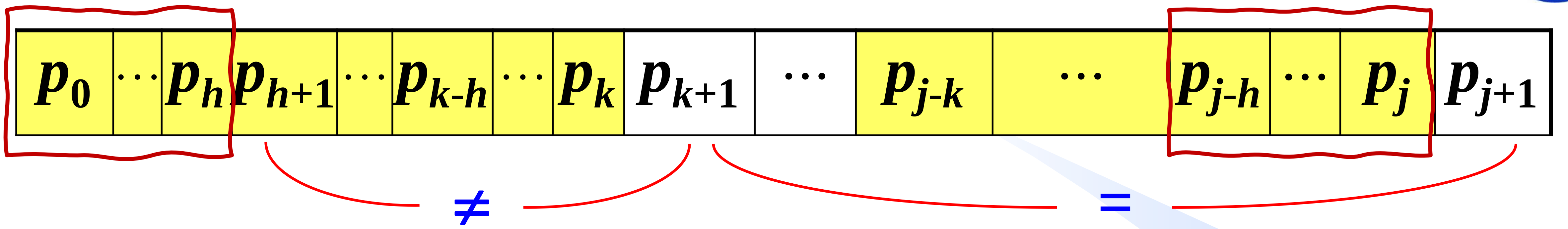
回顾之前的例子——原始的next函数



回顾之前的例子——改进的next函数



改进的next函数




```
void buildNext2(char P[], int next2[], int m) {
    int k = next2[0] = -1;
    for (int j = 0; j<m; j++) {  //求next2[j+1]
        while (k >= 0 && P[k] != P[j])
            k = next2[k];
        if(P[j+1]!=P[k+1]) next2[j+1] = ++k;
        else  next2[j+1] = next2[++k];
    }
}
```

改进的next函数

p_0	…	p_h	p_{h+1}	…	p_{k-h}	…	p_k	p_{k+1}	…	p_{j-k}	…	p_{j-h}	…	p_j	p_{j+1}

$\neq$ $=$

```
void buildNext2(char P[], int next2[], int m) {
    int k = next2[0] = -1;
    for (int j = 0; j<m; j++) {  //求next2[j+1]
        while (k >= 0 && P[k] != P[j])
            k = next2[k];
        next2[j+1]=(P[j+1]!=P[++k])? k:next2[k];
    }
}
```

总结

更加充分地利用以往比对所提供的信息

- 以往成功比对所获得的“经验”：$S_{i-j}\ldots S_{i-1}$是什么
- 以往失败比对所吸取的“教训”：S_i不是什么

S	…	s_{i-j}	…	…	…	s_{i-k}	…	s_{i-1}	s_i	…	s_{n-1}
		$=$	$=$	$=$	$=$	$=$	$=$	$=$	$\neq$		
P		p_0	…	p_{k-1}	p_k	p_{j-k}	…	p_{j-1}	p_j		