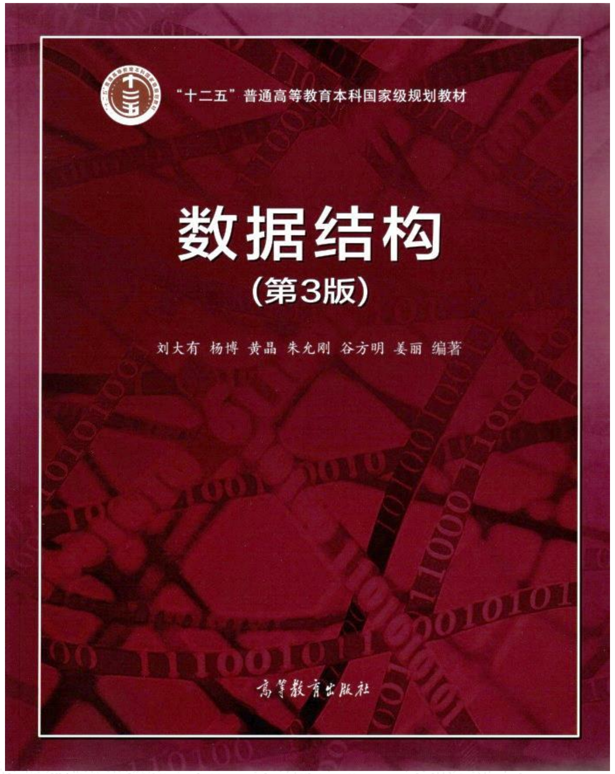



# 字符串匹配

- 模式匹配基本概念
- 朴素模式匹配
- KMP算法
- BM算法
- 字符串模糊匹配

数据之法
结构之美
算法之道

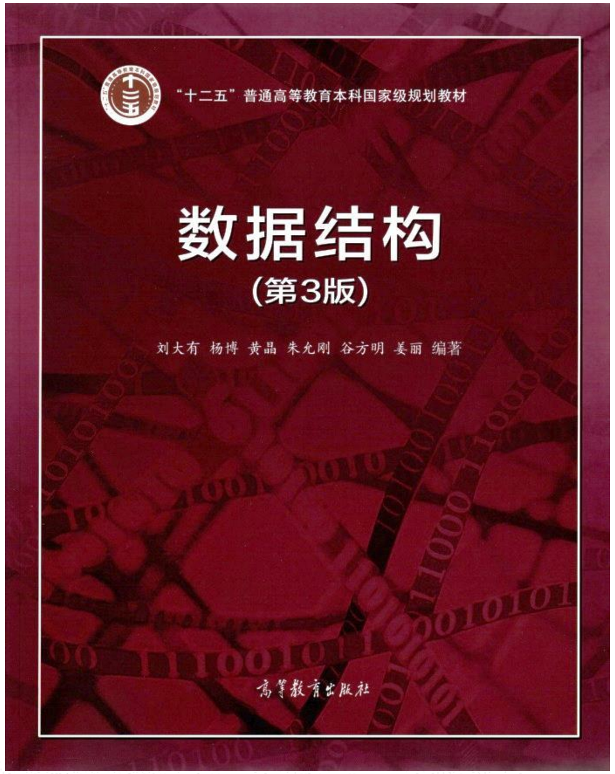
Last updated on 2022.10

zhuyungang@jlu.edu.cn

The best way to learn swimming is swimming, the best way to learn programming is programming.

# 慕课自学内容

- 字符串的定义与字符串类




# 字符串匹配

- **模式匹配基本概念**
- 朴素模式匹配
- KMP算法
- BM算法
- 字符串模糊匹配

数据之法
结构之美
算法之道

Last updated on 2022.10

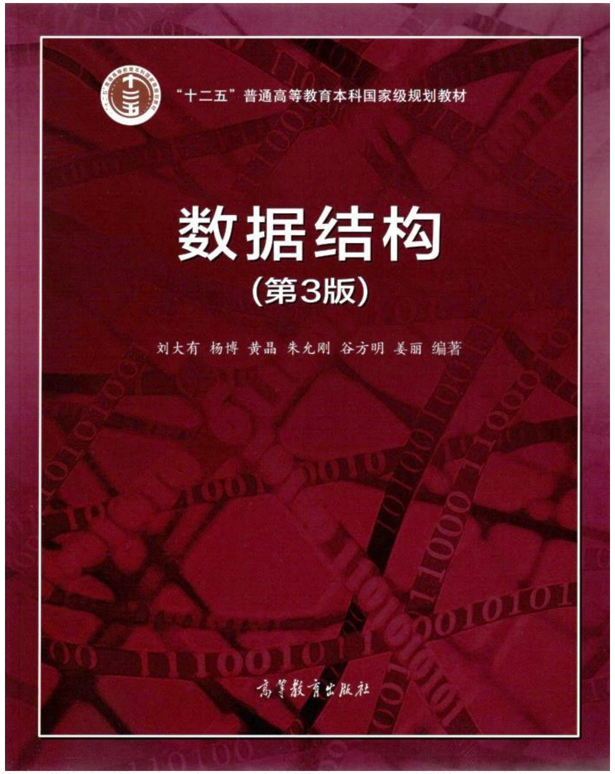
zhuyungang@jlu.edu.cn

# 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，$m \le n$。从S的第一个字符开始，搜索P，如果找到，返回P的第一个字符在S中的位置；如果没找到（即S中不包含P），则返回-1。

例：S= "*abaaabab*", P = "*abab*"

- 模式匹配应用：文本编辑器中常用的"查找"、"替换"




# 字符串匹配

- 模式匹配基本概念
- **朴素模式匹配**
- KMP算法
- BM算法
- 字符串模糊匹配

数据之法
结构之美
算法之道

Last updated on 2022.10

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 |

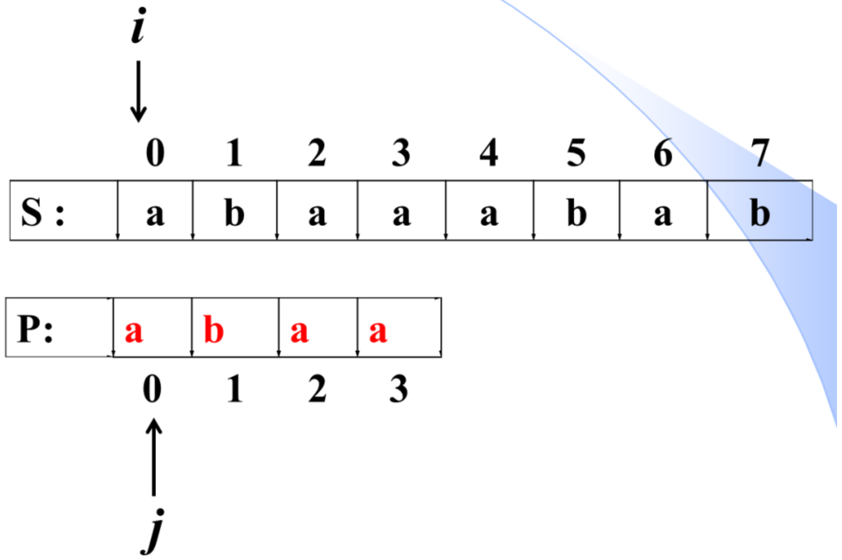
第一次匹配（看P在不在S的第0个位置）

第二次匹配（看P在不在S的第1个位置）

第三次匹配（看P在不在S的第2个位置）

…… ……

# 朴素模式匹配算法




**算法StringMatching( S, P, n, m )**

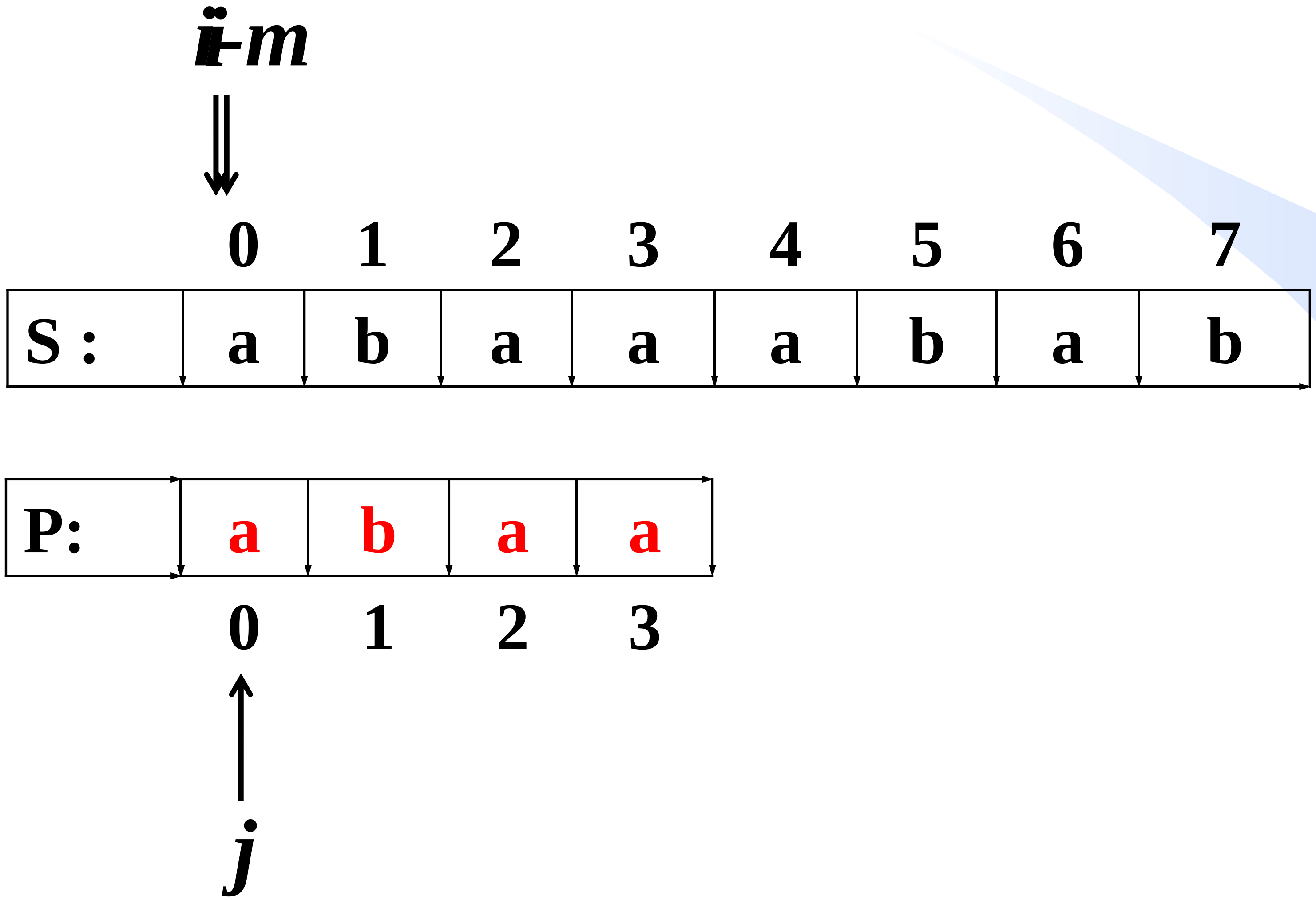
// S为n个字符的目标串，

// P为m个字符的模式串，

// 返回P在S中的位置

```
    i←0 .        //通过i扫描S
    WHILE  i ≤ n  DO
    (     //看P在不在S的第i个位置，从Si开始与P逐字符比对
        j←0 .  //通过j扫描P，
        WHILE j < m AND Si = Pj  DO
        (        i←i+1 . j←j+1.
        )
        IF j = m THEN RETURN i-m .  // 匹配成功
        ……
```

# 朴素模式匹配算法



# 朴素模式匹配算法

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

| P: | a | b | a | a |
|---|---|---|---|---|
| | 0 | 1 | 2 | 3 |

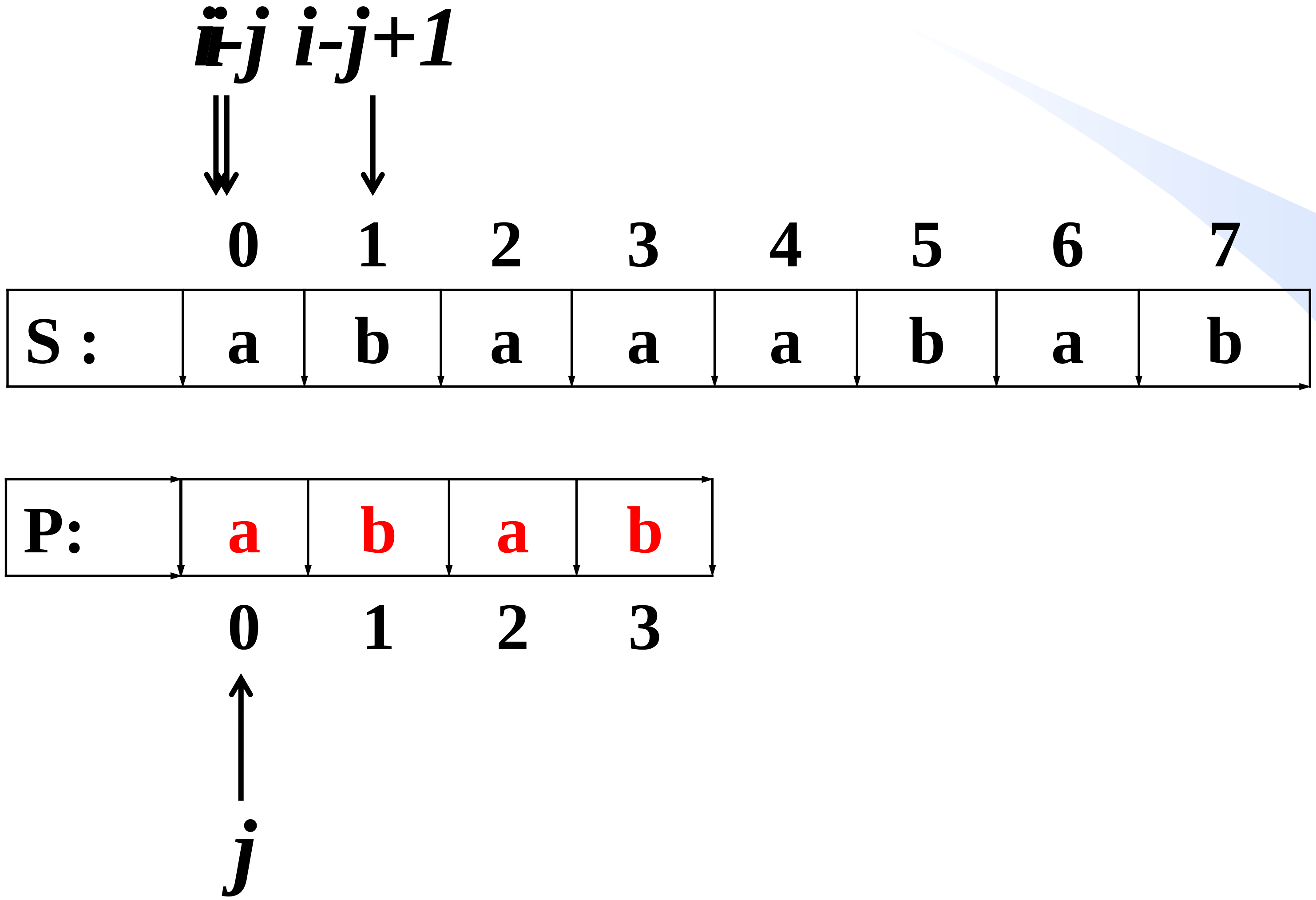
**算法StringMatching( S, P, n, m )**

// S为n个字符的目标串，P为m个字符的模式串，

// 返回P在S中的位置

```
i←0 .        //通过i扫描S
WHILE  i ≤ n  DO
(     //看P在不在S的第i个位置, 从Si开始与P逐字符比对
    j←0 .  //通过j扫描P,
    WHILE j < m AND Si = Pj  DO
    (        i←i+1 . j←j+1.
    )
    IF j = m THEN RETURN i-m .  // 匹配成功
    i←i –j +1 //本次匹配失败，i回退到原位置+1
)……
```

# 朴素模式匹配算法



# 朴素模式匹配算法

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

$i \leftarrow i-j+1$

| | 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 |

$j$

# 朴素模式匹配算法

**算法StringMatching( S, P, n, m )**

**// S为n个字符的目标串，P为m个字符的模式串，**
**// 返回P在S中的位置**

```
i←0 .        //通过i扫描S
WHILE  i ≤ n  DO
(     //看P在不在S的第i个位置, 从Si开始与P逐字符比对
      j←0 .  //通过j扫描P,
      WHILE j < m AND Si = Pj  DO
      (        i←i+1 . j←j+1.
      )
       IF j = m THEN RETURN i-m .  // 匹配成功
       i←i –j +1 //回退到原位置+1
)
RETURN – 1. ■
```

*i* ↓

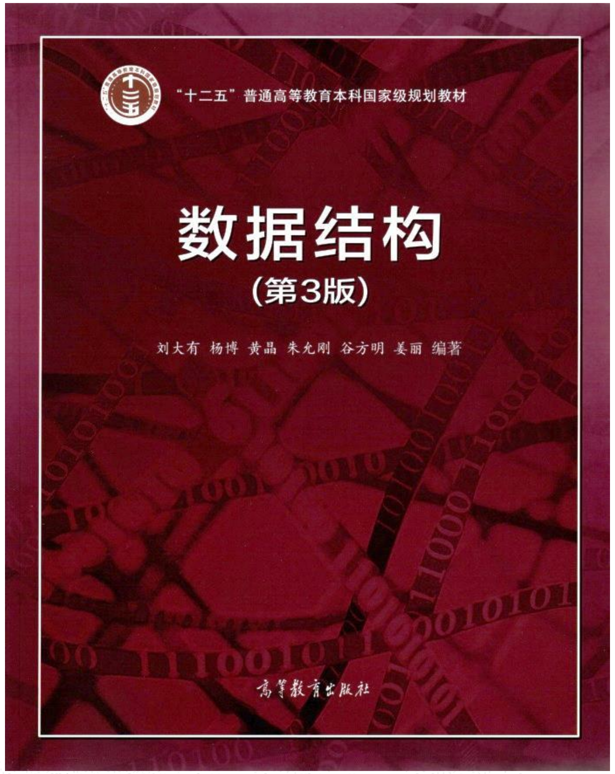
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| S : | a | b | a | a | a | b | a | b |

| P: | a | b | a | a |
|---|---|---|---|---|
| | 0 | 1 | 2 | 3 |

↑ *j*

# 朴素模式匹配算法时间复杂性

- 朴素模式匹配算法的优点是过程简单，缺点是效率较低。
- 关键运算：字符比较
- 时间复杂性为O(nm).




# 字符串匹配

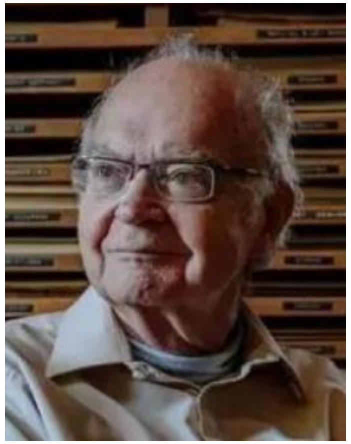
- 模式匹配基本概念
- 朴素模式匹配
- **KMP算法**
- BM算法
- 字符串模糊匹配

数据之法
结构之美
算法之道



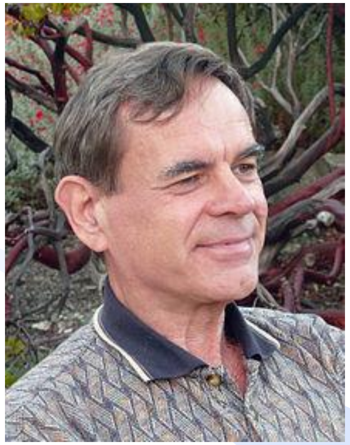
zhuyungang@jlu.edu.cn

# 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



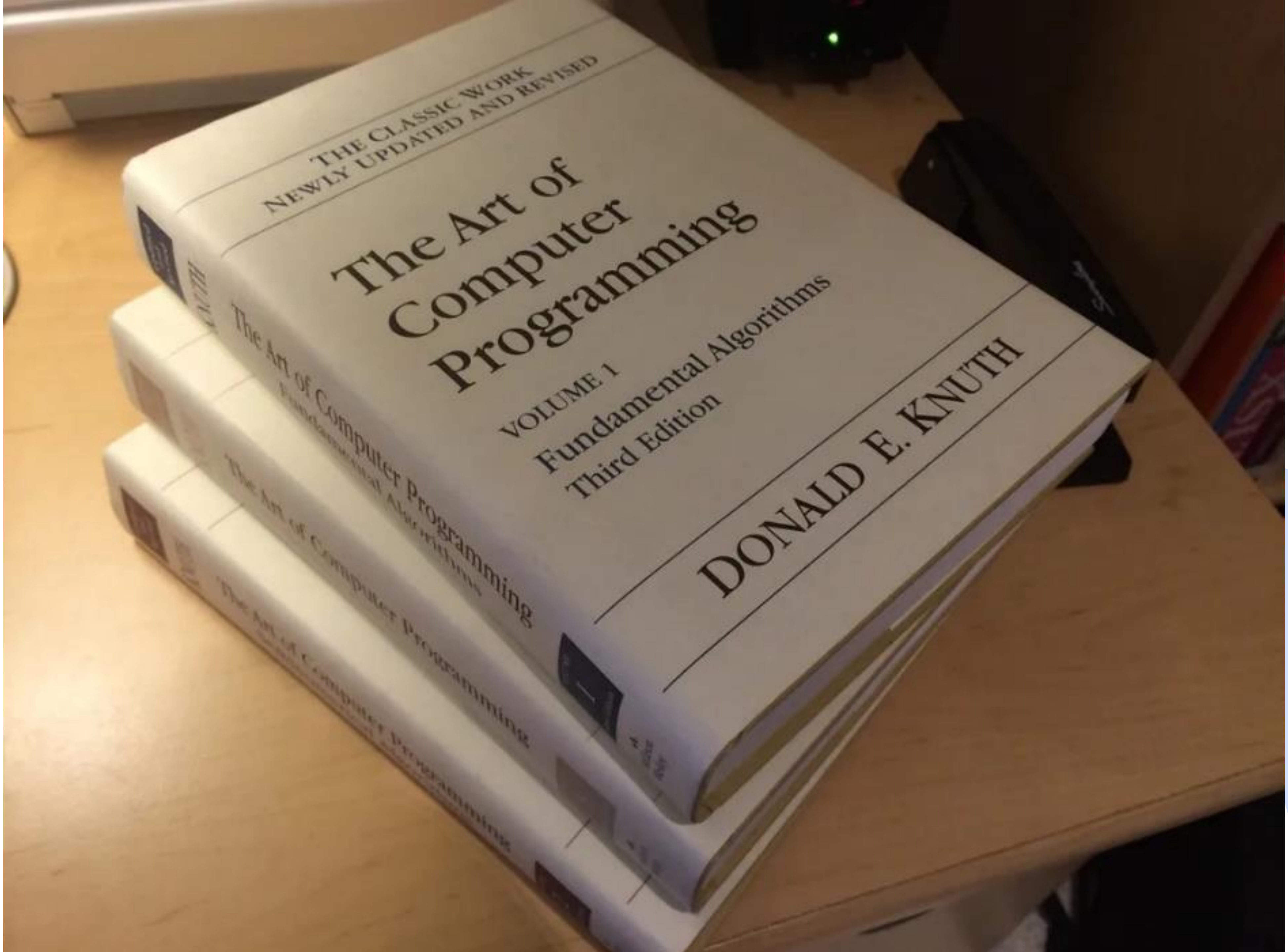
**Vaughan Pratt**
斯坦福大学教授
**ACM Fellow**

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

**FAST PATTERN MATCHING IN STRINGS***

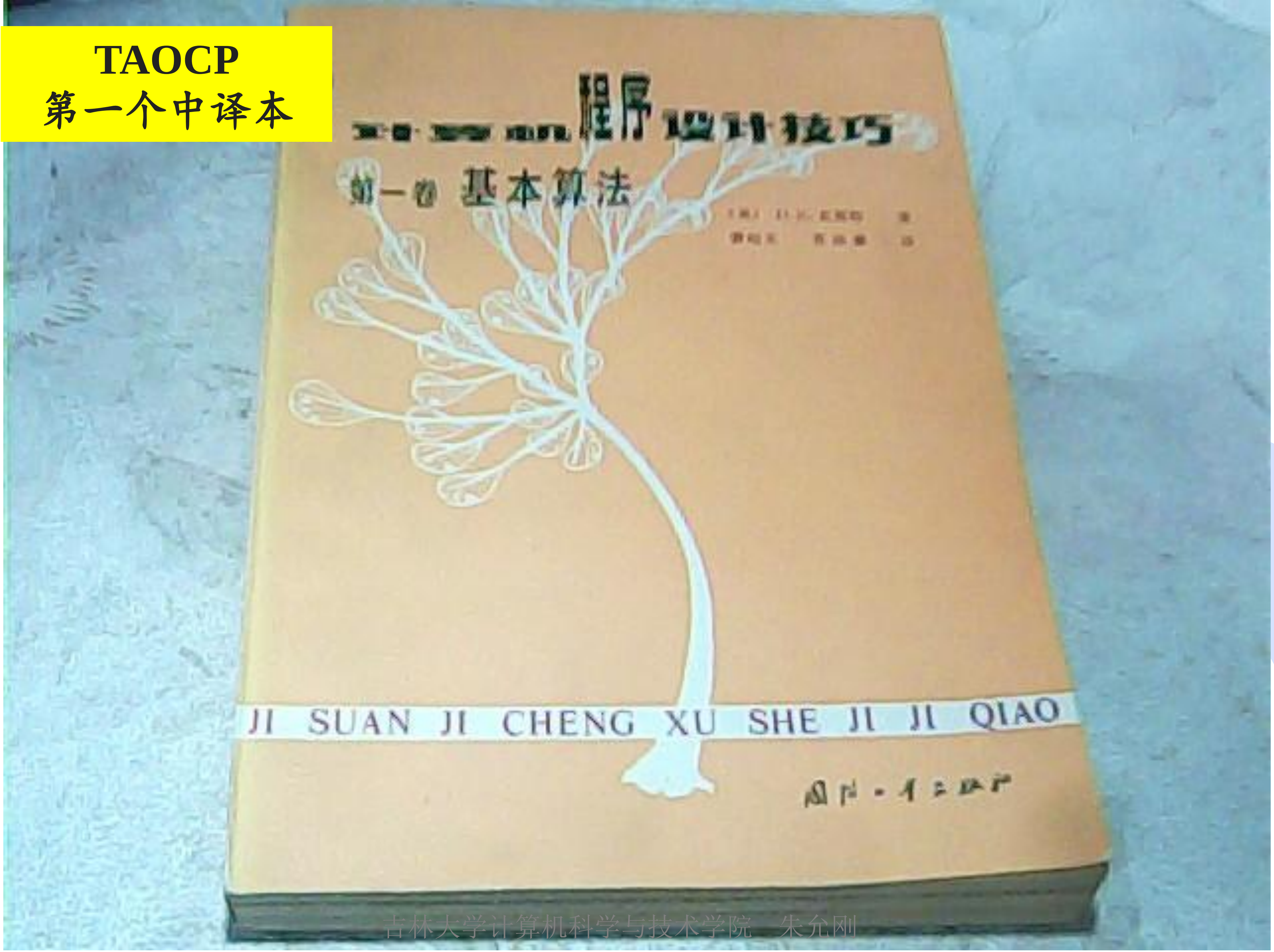
DONALD E. KNUTH†, JAMES H. MORRIS, JR.‡ AND VAUGHAN R. PRATT¶

**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
The Art of Computer Programming
Fundamental Algorithms

**TAOCP**

**第一个中译本**



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

# 朴素模式匹配算法

**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**

*i* ↓

| | 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 |

↑ *j*

# 朴素模式匹配算法

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

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

| | 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 |

$j$

# 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

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

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

**相等前后缀*abc*，下一次比对：*P*右移4位**

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

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

# 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: $\qquad\qquad\qquad\qquad\quad a\ b\ c\ \boxed{a}\ b\ c\ a\ \ldots$

**漏掉了可能的成功匹配位置！**

# KMP算法

$$\begin{array}{lccccccccc|c|ccc}
S: & a & b & c & a & b & c & a & b & c & a & b & c & x \\
P: & a & b & c & a & b & c & a & b & c & x & & & 
\end{array}$$

**方案2：更长的相等的前后缀*abcabc*，*P*右移3位**

$$\begin{array}{lccccccccc|c|ccc}
S: & a & b & c & a & b & c & a & b & c & a & b & c & x \\
P: & & & & a & b & c & a & b & c & a & b & c & x
\end{array}$$

**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。**

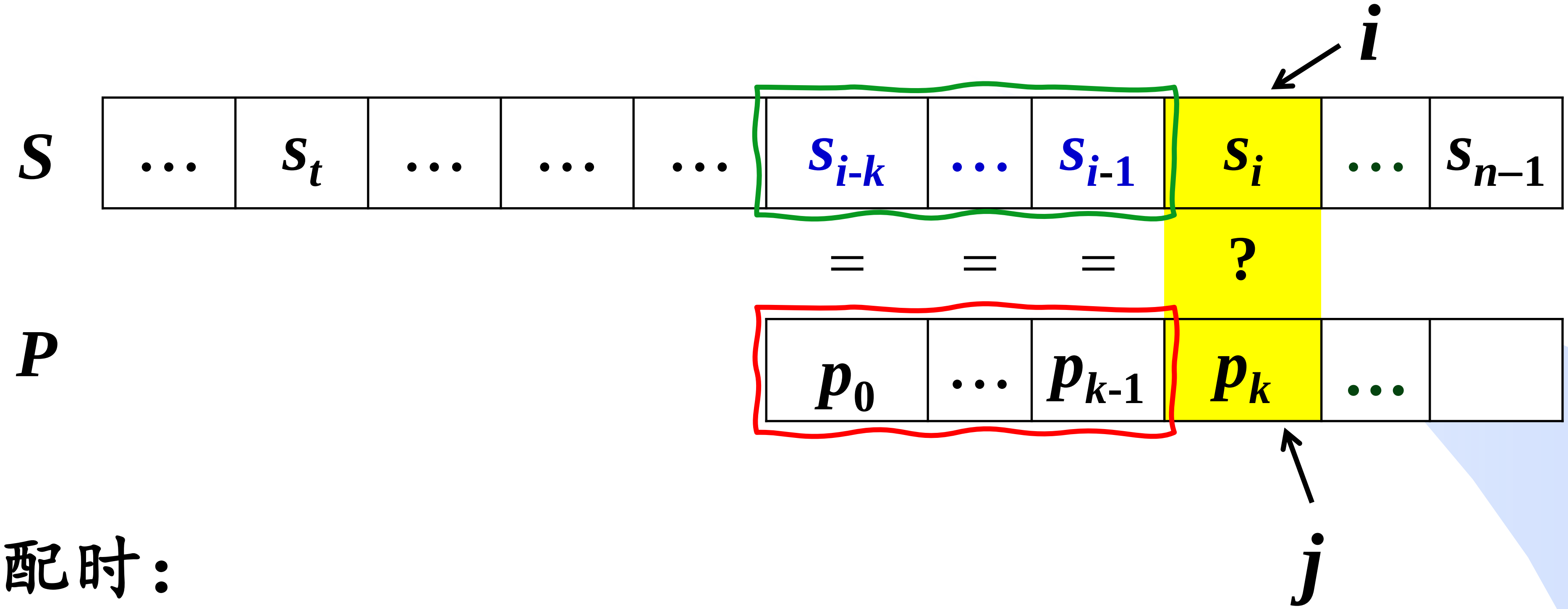
- 一般情况：设目标$S=s_0, s_1, \ldots, s_{n-1}$，模式串$P=p_0, p_1, \ldots, p_{m-1}$
- 假设当前正进行某次匹配，有 $s_{i\text{-}k}\ldots s_{s\text{-}1} = p_{j\text{-}k}\ldots p_{j\text{-}1}$，但 $s_i \neq p_j$，即在$p_j$的位置失配。

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

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

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

# $k$：$p[0…j\text{-}1]$的最长相等前后缀长度




下次匹配时：

- 右移P使$p[0…k\text{-}1]$ 与 $s[i–k …i\text{-}1 ]$对齐。
- 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。

# 在$p_j$处匹配失败后

- $P$的下一个匹配位置（$P$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）$next(j)$ = $p_0$ …$p_{j-1}$的最长相等前后缀的长度$k$。

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $S$ | … | $s_t$ | … | … | … | $s_{i\text{-}k}$ | … | $s_{i\text{-}1}$ | $s_i$ | … | $s_{n-1}$ |
| | | = | = | = | = | = | = | = | ≠ | | |
| $P$ | | $p_0$ | … | $p_{k\text{-}1}$ | … | $p_{j\text{-}k}$ | … | $p_{j\text{-}1}$ | $p_j$ | | |

# 在$p_j$处匹配失败后

- $P$的下一个匹配位置（$P$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）$next(j)$ = $p_0 \ldots p_{j-1}$的最长相等前后缀的长度$k$。

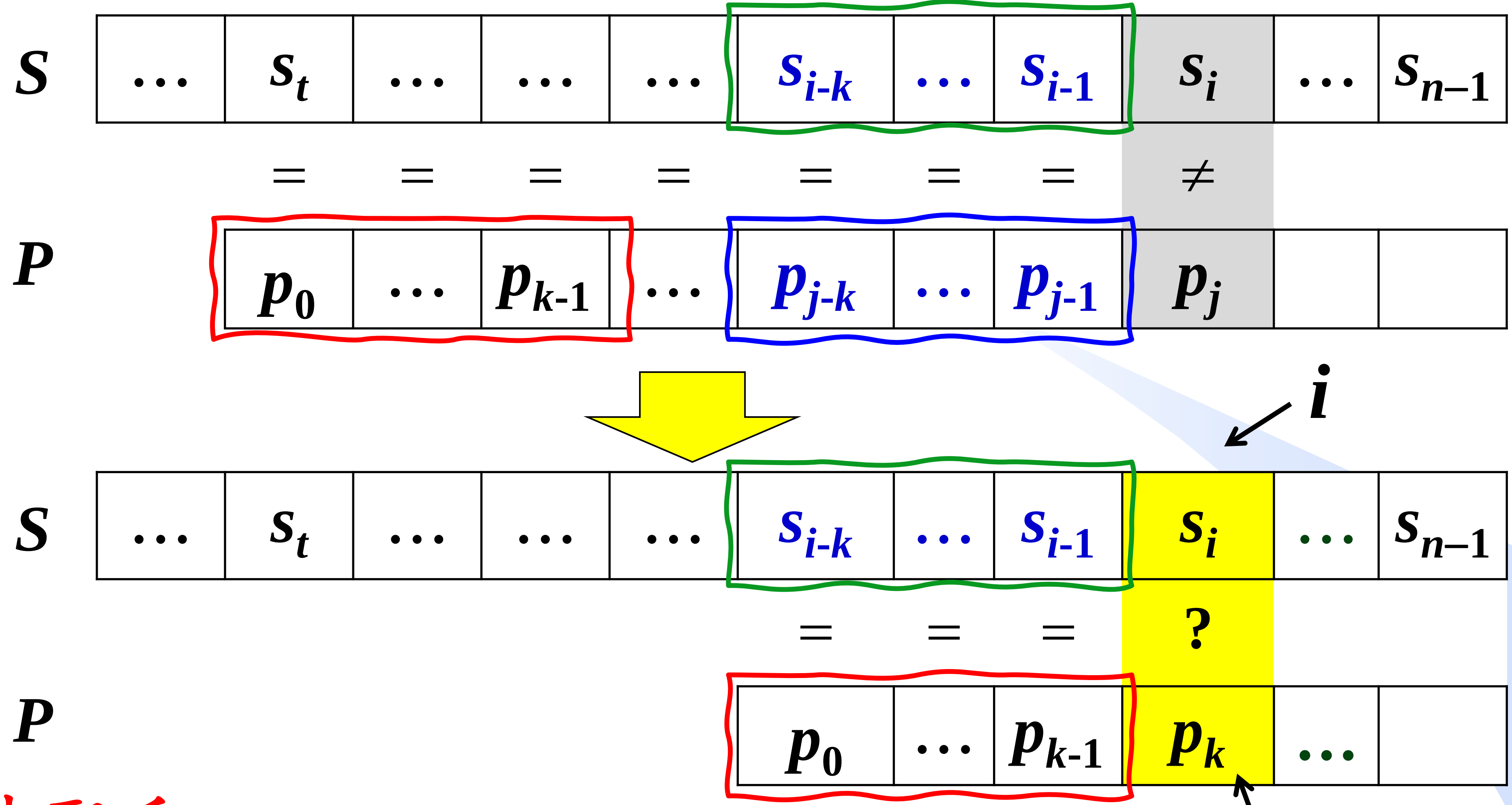
| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $S$ | … | $s_t$ | … | … | … | $s_{i-k}$ | … | $s_{i-1}$ | $s_i$ | … | $s_{n-1}$ |
| | | | | | | = | = | = | ? | | |
| $P$ | | | | | | $p_0$ | … | $p_{k-1}$ | $p_k$ | … | |

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

$=$

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| $p_0$ | $p_1$ | … | $p_{k-1}$ | $p_k$ | … | $p_{j-k-1}$ | $p_{j-k}$ | … | $p_{j-1}$ |

**可以预先把P中所有位置的*next*( *j* ) （0 ≤ *j* < *m*）算出来存在数组里， *next*( *j* )保存在数组*next*[ *j* ]中**

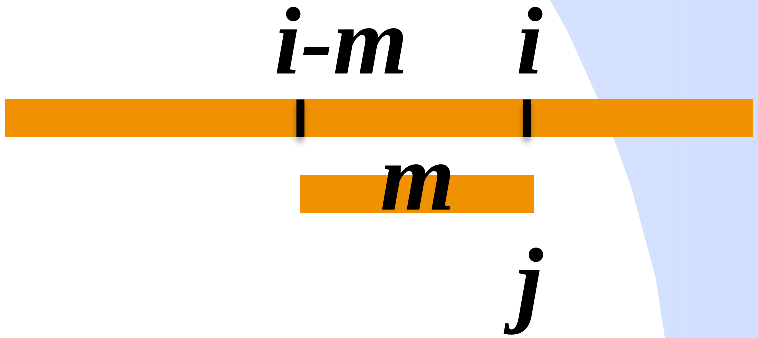



**在$p_j$处失配后：**

- 在$p_0 \ldots p_{j\text{-}1}$中，找最长相等的前后缀；
- $P$右移，使上述前缀和$S$对齐； **$P$的下一个匹配位置是$next[j]$**
- 目标串$S$匹配指针$i$不动，模式串$P$指针$j$指向$next[j]$

# KMP算法

```
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 */
    int next[N];    // N为预先定义的大于m的常数
    buildNext(P, next, m); //计算失败函数
    int i=0, j=0;   // i,j分别为S和P的匹配指针
    while(j<m && i<n) {
        if(S[i]==P[j]) i++, j++;
        else if(j==0) i++; //P[0]处失配, P右移一位, 下次P[0]和s[i+1]比对
        else j=next[j];  //确定P下次匹配位置
    }
    if (j==m) return i-m;   //匹配成功
    return -1;              //匹配失败
}
```



# KMP算法

```
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 */
    int next[N];    // N为预先定义的大于m的常数
    buildNext(P, next, m); //计算失败函数
    int i=0, j=0;   // i,j分别为S和P的匹配指针
    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) return i-m;   //匹配成功
    return -1;              //匹配失败
}
```

# KMP算法

```
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*/
    int next[N];    //N为预先定义的大于m的常数
    buildNext(P, next, m); //计算失败函数
    int i=0, j=0;   //i,j分别为S和P的匹配指针
    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 i-j;
}
```

匹配成功：返回值∈[0,n-m]
匹配失败：返回值＞n-m

- 若匹配成功，则$j=m$，返回值为$i\text{-}m$. 由于$i \le n$，故$i\text{-}j=i\text{-}m \le n\text{-}m$.
- 若匹配失败，则$i=n, j<m$，即$i\text{-}j > n\text{-}m$.

# KMP算法——时间复杂度分析

```
int KMP(char S[], char P[], int n, int m){
    .....暂不考虑失败函数的时间复杂度
    int i=0, j=0;
    while(j<m && i<n) {
        if(j==-1 || S[i]==P[j]) i++, j++;
        else j=next[j];
    }
    return i-j;
}
```

**关键运算**
**元素比较**

**while循环迭代次数不仅取决于*i*，还取决于*j***

**最多执行*n*次**

**使*j*减小，最多执行*n*次**

- 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。故*i*++最多执行*n*次
- *j*++与*i*++同步，故*j*也最多增加*n*次，每次加1.
- *j*初值0，迭代过程中永远不会小于-1.
- 故*j*减少（*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2*n*次，即O(*n*)

# 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 |

# 函数$next(j)$的计算

在 $p_0\, p_1 \ldots p_{j-1}$ 中找出最长相等的前后缀

暴力方案：从长到短考察每个前缀，看能否匹配后缀。

模式串　$a\; b\; a\; a\; b$

| |
|---|
| *abaa* |
| *aba* |
| *ab* |
| *a* |

O(m²)

# 函数$next(j)$的计算

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：

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

用递推法计算next函数，设$next(0) = -1$, $next(1) = 0$，已知$next(j) = k$，求 $next(j+1)$

# 已知 $next(j)=k$, 求 $next(j+1)$

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

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

如果$p_k = p_j$

$p_0 \dots p_j$的最长相等的前后缀长度（$next(j+1)$）比以前加1

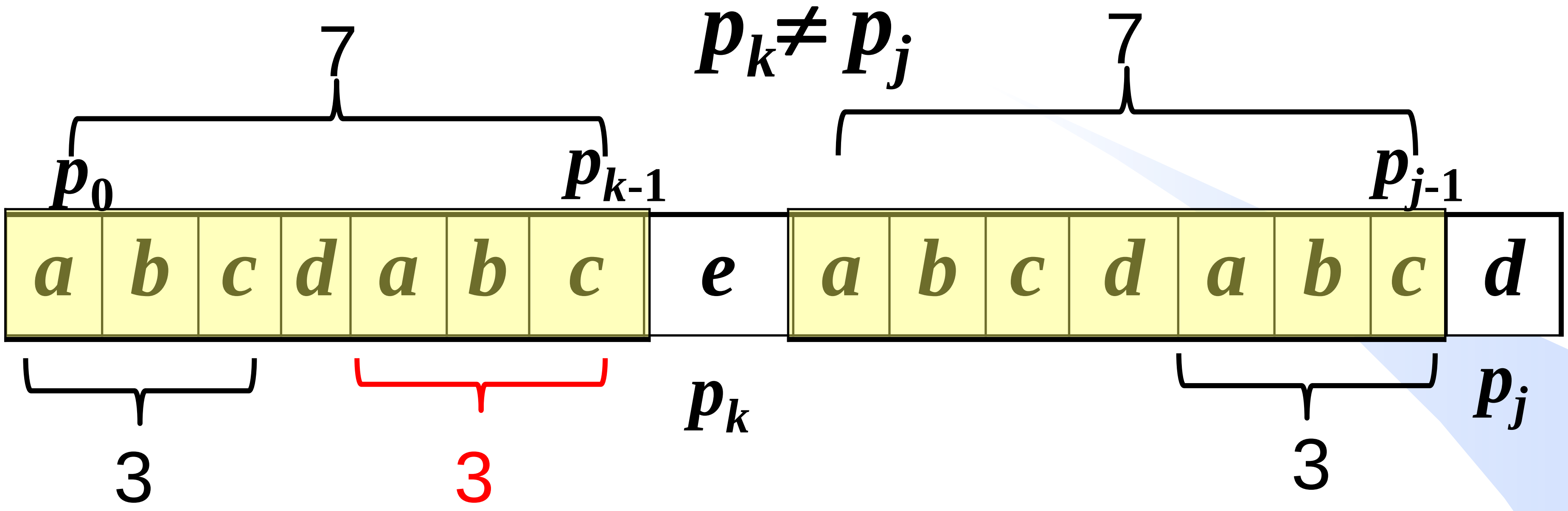
$next(j+1)=k+1$

# 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

这实际上等价于【$p_0 \ldots p_{k-1}$的最长相等前后缀】

# 已知 $next(j)=k$, 求 $next(j+1)$

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

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：即寻找小于$k$的最大$h$，使$p_0 \ldots p_{h-1} = p_{j-h} \ldots p_{j-1} = p_{k-h} \ldots p_{k-1}$，这实际上等价于是找$p_0 \ldots p_{k-1}$的最长相等前后缀$p_0 \ldots p_{h-1}$和$p_{k-h} \ldots p_{k-1}$，其中$h=next(k)$.

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 $next(j+1)$

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

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

若$p_l = p_j$，则 $next(j+1)=l+1$

若$p_l \neq p_j$，则考察再稍短一点（第4长）的相等前后缀

以此类推……


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


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

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


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

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

$l = next[h]$　若 $P[l]=P[j]$ 则 $next[j+1]=l+1$，否则


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

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

$l = next[h]$ 若$P[l]=P[j]$ 则 $next[j+1]=l+1$，否则

$r = next[l]$ 若$P[r]=P[j]$ 则 $next[j+1]=r+1$，否则


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

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

$l = next[h]$ 若$P[l]=P[j]$ 则 $next[j+1]=l+1$，否则

$r = next[l]$ 若$P[r]=P[j]$ 则 $next[j+1]=r+1$，否则

……


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

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

$l = next[h]$ 若$P[l]=P[j]$ 则 $next[j+1]=l+1$，否则

$r = next[l]$ 若$P[r]=P[j]$ 则 $next[j+1]=r+1$，否则

……

$0 = next[s]$ 若$P[0]=P[j]$ 则 $next[j+1]=0+1$，否则


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

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

$l = next[h]$ 若 $P[l]=P[j]$ 则 $next[j+1]=l+1$，否则

$r = next[l]$ 若 $P[r]=P[j]$ 则 $next[j+1]=r+1$，否则

……

$0 = next[s]$ 若 $P[0]=P[j]$ 则 $next[j+1]=0+1$，否则

$-1 = next[0]$ 则 $next[j+1]=0$


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

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

$l$ = $next[h]$ 若$P[l]=P[j]$ 则 $next[j+1]=$

$r$ = $next[l]$ 若$P[r]=P[j]$ 则 $next[j+1]=$

……

0 = $next[s]$ 若$P[0]=P[j]$ 则 $next[j+1]=0+1$，否则

-1 = $next[0]$ 则 $next[j+1]=0$

```
while(P[k]!=P[j])
    k=next[k];
```


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

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

$l = next[h]$ 若$P[l]=P[j]$ 则 $next[j+1]=l+1$

$r = next[l]$ 若$P[r]=P[j]$ 则 $next[j+1]=r+1$

……

$0 = next[s]$ 若$P[0]=P[j]$ 则 $next[j+1]=0+1$，否则

$-1 = next[0]$ 则 $next[j+1]=0$

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


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

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

$l$ = $next[h]$ 若 $P[l]=P[j]$ 则 *nex*

$r$ = $next[l]$ 若 $P[r]=P[j]$ 则 *nex*

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

……

$0$ = $next[s]$ 若 $P[0]=P[j]$ 则 $next[j+1]=0+1$，否则

$-1$ = $next[0]$ 则 $next[j+1]=0$

# *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=-1; next[0]=-1;
    for(int j=0; j<m-1; j++){ //求next[j+1]
        k=next[j]; //此句可省去
        while(k>=0 && P[k]!=P[j])
            k=next[k];
        next[j+1]=++k;
    }
}
```

# 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=-1; next[0]=-1;
    for(int j=0; j<m-1; j++){//求next[j+1], 此刻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

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

关键运算
元素比较

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

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

**KMP算法总时间复杂度O(*n*+*m*)**

- ➢ *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*]）的次数必然小于等于*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2m次，即O(*m*)

# next数组的计算——另一种等价写法

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

```
void buildNext(char P[], int next[], int m){
    int j=0,k=next[0]=-1; //此时有k=next[j]
    while(j<m-1){   //求next[j+1], j+1<m
        if(k==-1 || P[k]==P[j])
            next[++j]=++k; //next[j+1]=k+1
        else
            k=next[k];//求p0…pk-1的最长相等前后缀
    }
}
```

# next数组的计算——另一视角

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

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

***P*和自己进行模式匹配**

# next数组的计算——另一视角

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

| | | | | | |
|---|---|---|---|---|---|
| $p_0$ | … | … | $p_{k-1}$ | $p_k$ | … |

***P*和自己进行模式匹配**

# *next*数组的计算——另一视角

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| ***P*** | … | $p_{j\text{-}k}$ | … | … | … | … | … | $p_{j\text{-}1}$ | $p_j$ | … | $p_{m-1}$ |
| | | = | = | = | = | = | = | = | | | |
| ***P*** | | $p_0$ | … | $p_{h\text{-}1}$ | … | $p_{k\text{-}h}$ | … | $p_{k\text{-}1}$ | $p_k$ | | |

***P***和自己进行模式匹配

# next数组的计算——另一视角

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **$P$** | … | $p_{j\text{-}k}$ | … | … | … | $p_{j\text{-}h}$ | … | $p_{j\text{-}1}$ | $p_j$ | … | $p_{m-1}$ |
| | | | | | | = | = | = | | | |
| **$P$** | | | | | | $p_0$ | … | $p_{h\text{-}1}$ | $p_h$ | | |

**$P$和自己进行模式匹配**

# *next*数组的计算——另一视角

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

**KMP: 模式串*P*和*S*匹配**

```
void buildNext(char *P,int next[]){
   int m=strlen(P);
   next[0]=-1;
   int j=0, k=-1; //k=next[j]
   while(j<m-1){
      if(k==-1 || P[k]==P[j])
         j++,k++,next[j]=k;
      else
         k=next[k];
   }
}
```

***next*数组: *P*和自己匹配**

# KMP算法变形：若P在S中多次出现，输出P在S中的所有匹配位置

```
int KMP(char S[], char P[], int n, int m){//S长度n, P长度m
    int next[N];    //数组 next[ ]存放next函数的值
    buildNext(P, next, m); //计算next数组，需要多算一位，算到next[m]
    int i=0,j=0,cnt=0;//i,j为S和P的匹配指针，cnt为P在S中出现次数
    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
        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;
}
```

next数组需要多算一位，算到next[m]

# KMP算法隐藏应用举例

字符串长度为$n$

- 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$\max_i(next[i])$.

$a\ b\ c\ a\ b\ c\ a\ b\ c\ x\ x\ x$

# 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.

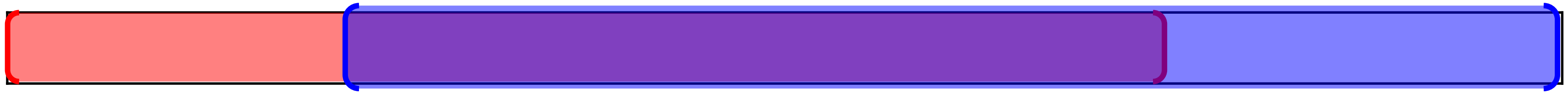
# KMP算法隐藏应用举例

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】

例如：

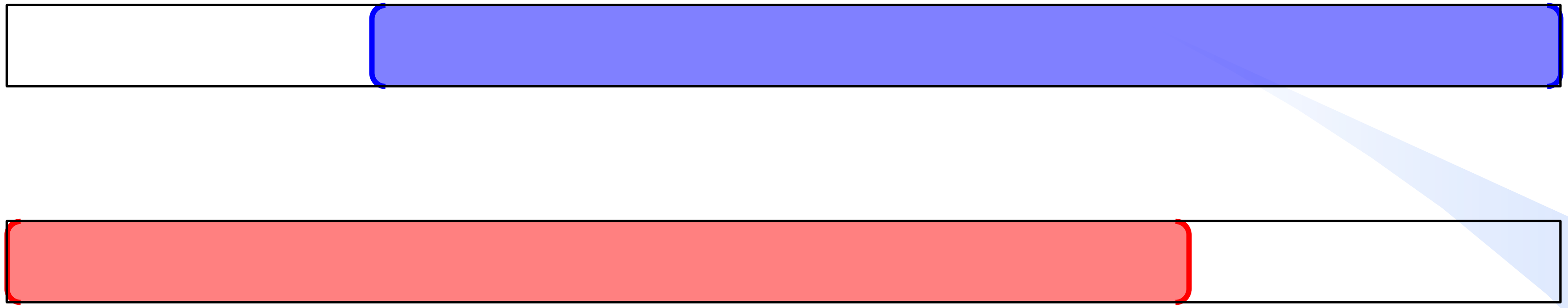
***S*= *a b c a b c a b c a b c*** ，输出**True**

***S*= *a b a*** ，输出**False**

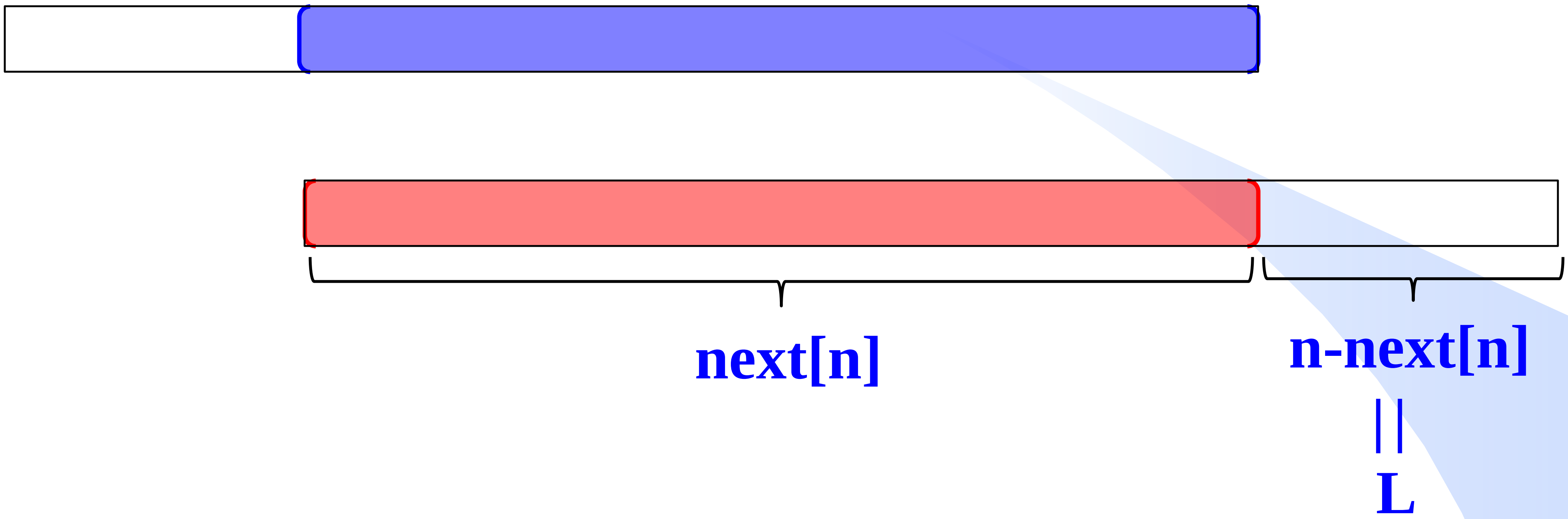


# KMP算法隐藏应用举例

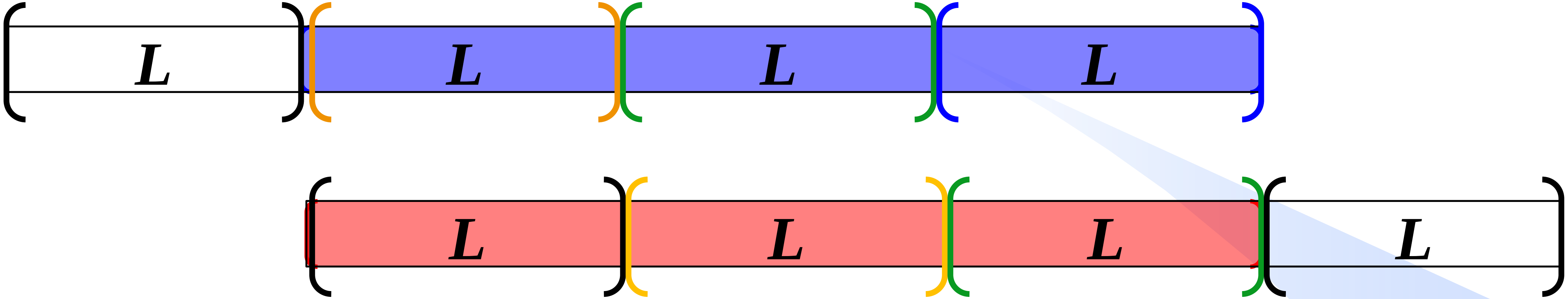
# KMP算法隐藏应用举例



# KMP算法隐藏应用举例



# KMP算法隐藏应用举例




- ➢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

- 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 n=strlen(s);
    int next[N];
    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;
}
```

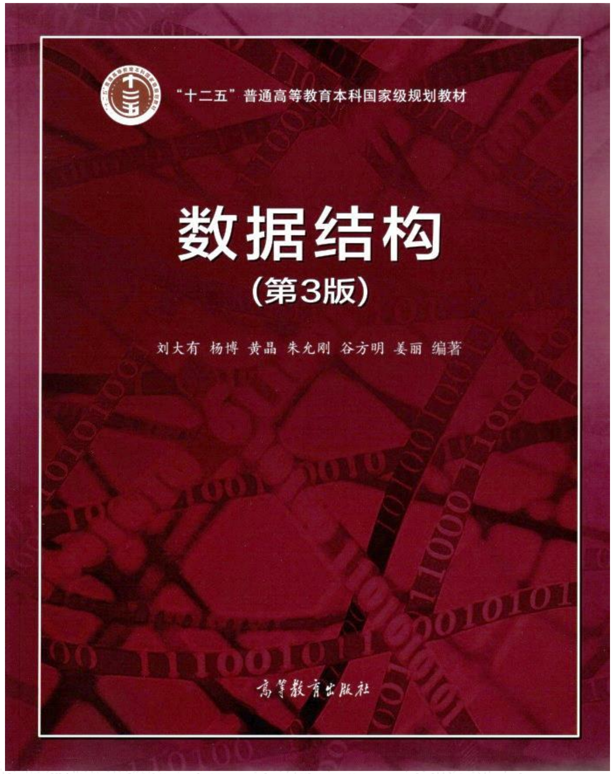
# $next(j)$与教材中失败函数$f(j)$的关系

$next(j)$：　$p_0 \ldots p_{j\text{-}1}$ 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

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

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




# 字符串匹配

- 模式匹配基本概念
- 朴素模式匹配
- KMP算法
- **BM算法**
- 字符串模糊匹配

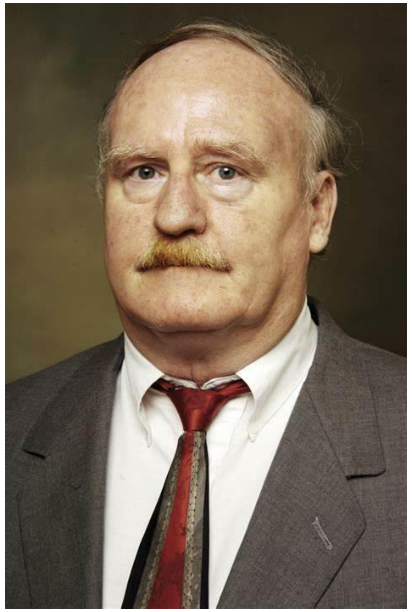
数据之法
结构之美
算法之道

Last updated on 2022.10

zhuyungang@jlu.edu.cn

# BM算法

## BM算法：以终为始，方得始终

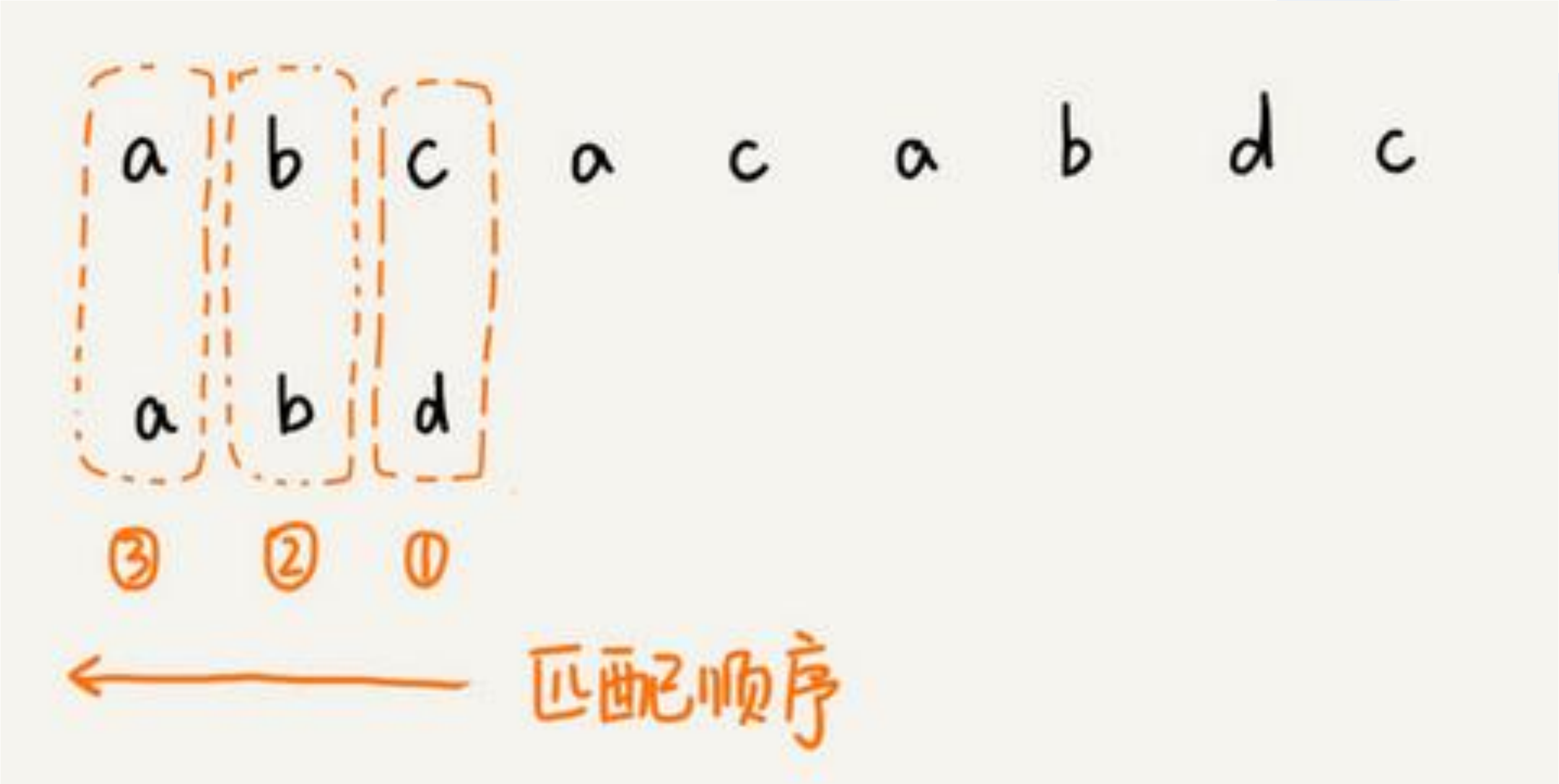


**Robert S. Boyer**
University of Texas at Austin

**J Strother Moore**
University of Texas at Austin
美国工程院院士

# BM算法

➢ 每一趟比对，按照模式串下标从大到小（自右向左）的顺序进行比对



# BM算法

- （1）若P中不含坏字符：

坏字符

```
.  .  .  .  .  .  T  L  E  .  .  .  .  .  .
         N  E  E  D  L  E
```

- 则该坏字符与模式串P中的任何字符都不可能匹配，则P整体移过失配位置，即移动到坏字符后面的位置

```
.  .  .  .  .  .  T  L  E  .  .  .  .  .  .
                     N  E  E  D  L  E
```

# BM算法

➢ （2）若P中含有1个坏字符：

坏字符

还能将P移过坏字符么？至少坏字符本身应得以匹配。

```
. . . . . . N L E . . .
      N E E D L E
```

➢ 找出P中的坏字符，让其与失配位置对齐。

```
. . . . . . N L E . . . . . .
            N E E D L E
```

# BM算法

- （3）若$P$中包含多个坏字符：

坏字符

| . | . | . | . | . | . | N | L | E | . | . | . | . | . | . |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | N | N | E | D | L | E | | | | | | |

…$c$ $c$ $a$ $a$…

$c$ $c$ $a$ $a$

- 让$P$中**最右边**的坏字符与失配位置对齐。

| . | . | . | . | . | . | N | L | E | . | . | . | . | . | . |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | N | N | E | D | L | E | | | |

# BM算法

➢ （4）*P*中包含多个坏字符，最右边的坏字符过于靠右，在失配位置的右侧

坏字符

```
. . . . . . E L E . . . . . .
          N E D D L E
```

➢ 此时将*P*右移1个字符。

```
. . . . . . E L E . . . .
  N E D D L E
```

```
. . . . . . E L E . . . .
            N E D D L E
```

# BM算法

➢ 坏字符策略：

（1）没出现规则：*P*中不包含坏字符，则P整体移过失配位置。

（2）左端出现规则：*P*包含坏字符，且在失配位置的左端，让*P*中最右边的坏字符（失配位置左侧且与失配位置最近的坏字符）与失配位置对齐。

（3）右端出现规则：*P*包含坏字符，且在失配位置的右端，*P*右移1位。

# BM算法时间复杂度

- 若暂且不考虑预处理的时间代价：
- 最好情况O(n/m) **【每比较1次右移m位】**


- P越长，效果越明显。
- 单词匹配概率越小，性能优势越明显。例如字符串S是一段英文文章（包含字符a…z, A…Z, 0…9,标点符号等），这样任给两个字符，相等的概率是较低的。如果字符是0101串，则任给两个字符，其相等概率相对就比较高了。

# BM算法时间复杂度

- 最坏情况O(n×m)【每比较m次右移1位】

S

P


- 退化成朴素匹配算法。

# KMP算法与BM算法

- 当字符集规模较小（组成字符串的字符种类较少）时，字符间相等的概率较高，KMP算法具有稳定的线性时间复杂度，更能体现出优势；采用坏字符策略的BM算法却不能大跨度向右移动。
- 当字符集规模较大时，字符间相等的概率较低，KMP算法依旧保持线性时间复杂度；采用坏字符策略的BM算法会因比对失败的概率增加，大跨度地向右移动。

# BM算法的好后缀策略

S: ... b $\boxed{a\ b}$ ...

P: b a b a b y a b



（1）若P失配位置左方有与好后缀相等的子串，则选取最靠右的子串，让该子串与主串的好后缀对齐。

S: ...b $\boxed{a\ b}$...

P: b a b a b y a b

# BM算法的好后缀策略

S: …d $\boxed{a\ b\ c\ d}$ …

好后缀

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

（2）若P失配位置左方没有与好后缀相等的子串，则找*P*的最长前缀，该前缀与**好后缀的某个后缀**相等，然后该前缀与主串的好后缀对齐。

S: …d $\boxed{a\ b\ c\ d}$ …

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

# BM算法的好后缀策略

$S$:  …d $\boxed{x\ y\ z}$ d a c …

$P$: d a b y $\boxed{x\ y\ z}$  好后缀

（3）若*P*失配位置左方没有与好后缀相等的子串，且找不到与好后缀的某个后缀相等的前缀，意味着*P*一位一位右移的过程中不可能有和*S*的好后缀匹配的子串，故将*P*整体右移至*S*的好后缀后面。

$S$:  …d $\boxed{x\ y\ z}$ d a c …

$P$:  d a b y x y z

# BM算法的坏字符+好后缀策略

每次失配后，

$$P\text{右移的位数} = \max\left\{\begin{array}{c}\text{坏字符策略}\\\text{算出的右移位数}\end{array},\ \begin{array}{c}\text{好后缀策略}\\\text{算出的右移位数}\end{array}\right\}$$

# 字符串匹配时间复杂度

BM

（坏字符）

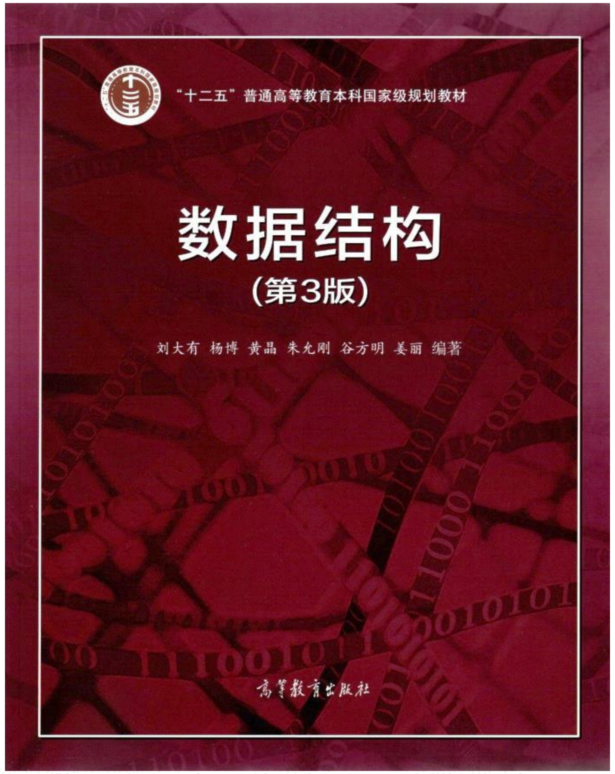
O(n×m)　　O(n+m)　　O(n/m)

朴素算法

BM

（坏字符+好后缀）

KMP




# 字符串匹配

- 模式匹配基本概念
- 朴素模式匹配
- KMP算法
- BM算法
- **字符串模糊匹配**

数据之法
结构之美
算法之道



zhuyungang@jlu.edu.cn

# 字符串模糊匹配

- 模糊匹配：降低精确匹配的要求：S和P之间只要有一定程度的相似，就认为匹配成功。
- 衡量两个字符串之间的相似程度。
- 应用：基因测序与基因序列比对

百度为您找到相关结果约26,500,000个

南京疫情中已测序病例病毒基因与俄罗斯入境航班病例一

2021年7月31日 来源：新华网 新华社南京7月30日电（记者郑生竹）30日，南京

任丁洁在南京市新冠肺炎疫情防控新闻发布会上介绍，本次南京疫情早期报告的

海外网 百度快照

南京疫情源头确认:俄罗斯入境的CA910航班,曾被"10次熔

2021年7月31日 南京市疾控中心副主任丁洁通报:我们

情相关的52个病例的病毒基因测序工作,均属于德尔塔

列高度同源,提示是同一个传播链。 本次疫情早期报告

网易 百度快照

## 长春11例病例全基因组测序均为奥密克戎，与吉林市高度同源

播报文章

人民资讯

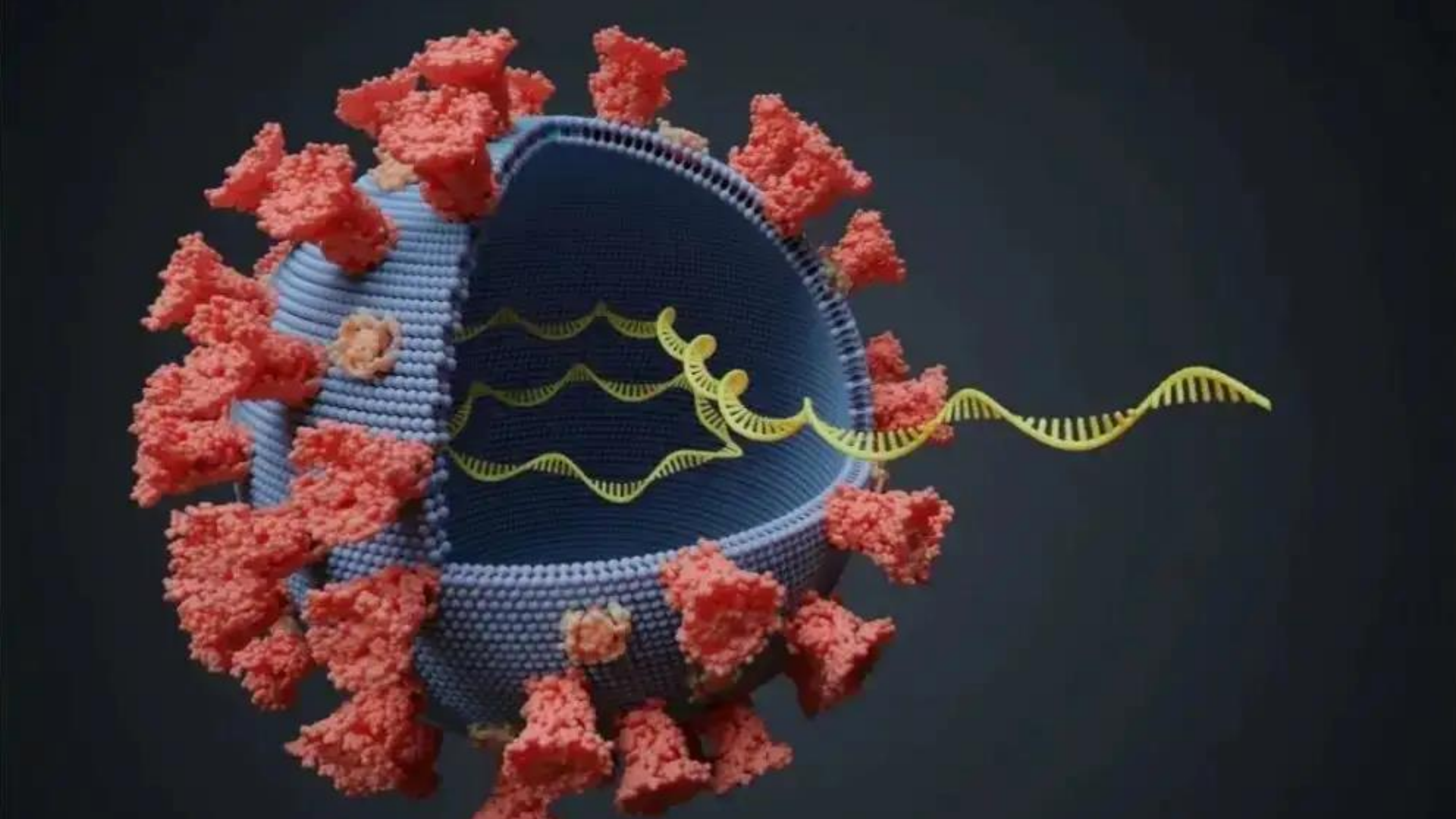
2022-03-09 14:34 | 人民网人民科技官方帐号

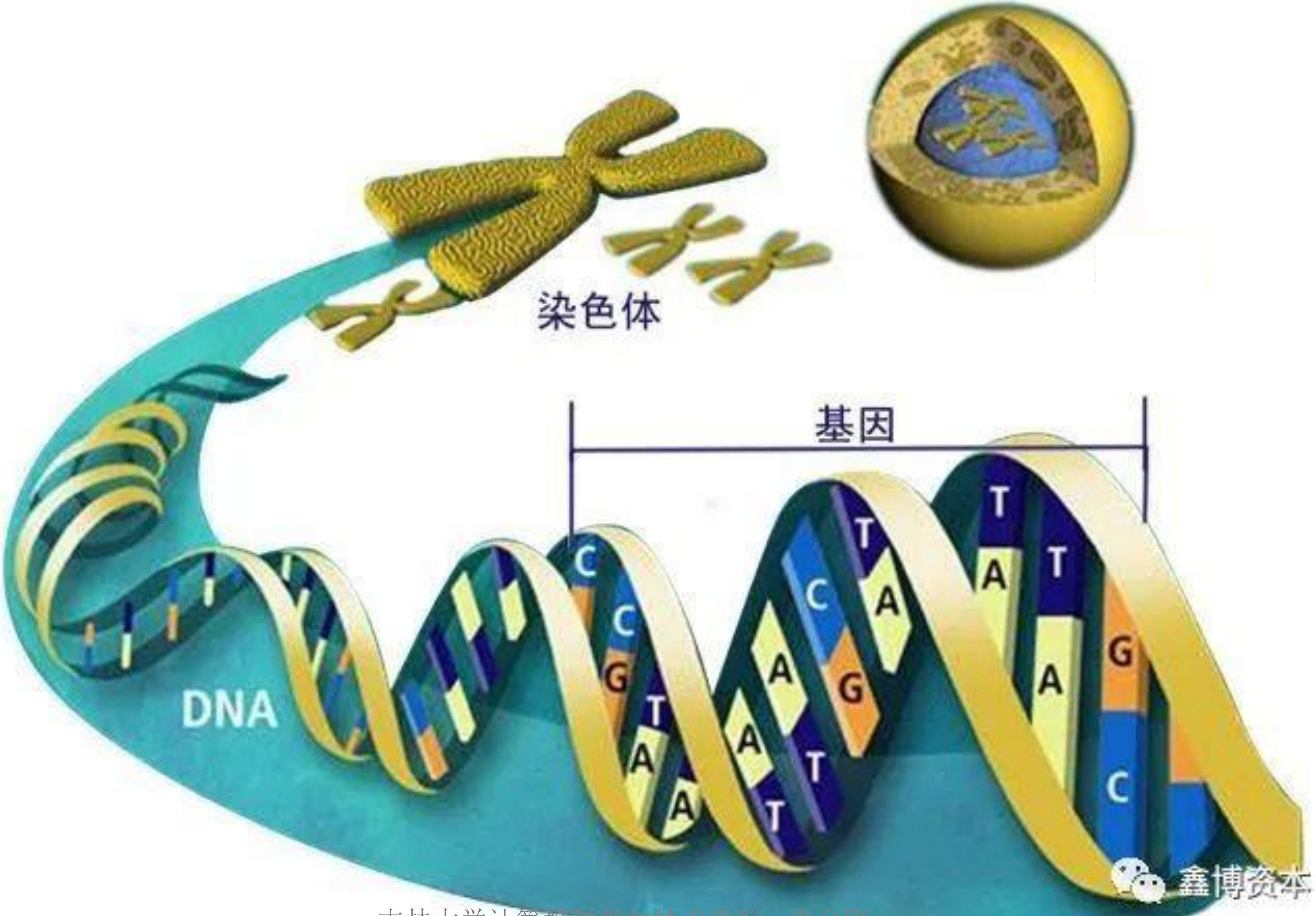
关注

本文转自：央视新闻

今天（3月9日），吉林省长春市召开疫情防控新闻发布会，介绍疫情防控最新情况。

相关负责人介绍，已完成对11例病例的全基因组测序工作，结果显示均为奥密克戎变异株BA.2进化分支，与吉林市病毒测序结果高度同源，推测为同一传播链。




染色体
基因
DNA
C
C
G
T
A
A
C
A
A
T
T
T
A
G
T
A
T
A
G
C
鑫博资本

# 一个新冠病毒个体的基因序列

gttaaaggtttataccttcccaggtaacaaaccaaccaactttcgatctcttgtagatctgttctctaaacgaactttaaaatctgtgtggctgtcactcggctgcatgcttagtgcactcacgca
gtataattaataactaattactgtcgttgacaggacacgagtaactcgtctatcttctgcaggctgcttacggtttcgtccgtgttgcagccgatcatcagcacatctaggtttcgtccgggtgtg
accgaaaggtaagatggagagccttgtccctggtttcaacgagaaaacacacgtccaactcagtttgcctgttttacaggttcgcgacgtgctcgtacgtggctttggagactccgtggagg
aggtcttatcagaggcacgtcaacatcttaaagatggcacttgtggcttagtagaagttgaaaaaggcgttttgcctcaacttgaacagccctatgtgttcatcaaacgttcggatgctcgaac
tgcacctcatggtcatgttatggttgagctggtagcagaactcgaaggcattcagtacggtcgtagtggtgagacacttggtgtccttgtccctcatgtgggcgaaataccagtggcttaccg
caaggttcttcttcgtaagaacggtaataaaggagctggtggccatagttacggcgccgatctaaagtcatttgacttaggcgacgagcttggcactgatccttatgaagattttcaagaaaa
ctggaacactaaacatagcagtggtgttacccgtgaactcatgcgtgagcttaacggaggggcatacactcgctatgtcgataacaacttctgtggccctgatggctaccctcttgagtgca
ttaaagaccttctagcacgtgctggtaaagcttcatgcactttgtccgaacaactggactttattgacactaagaggggtgtatactgctgccgtgaacatgagcatgaaattgcttggtacac
ggaacgttctgaaaagagctatgaattgcagacaccttttgaaattaaattggcaaagaaatttgacaccttcaatggggaatgtccaaattttgtatttcccttaaattccataatcaagactatt
caaccaagggttgaaaagaaaaagcttgatggctttatgggtagaattcgatctgtctatccagttgcgtcaccaaatgaatgcaaccaaatgtgcctttcaactctcatgaagtgtgatcatt
gtggtgaaacttcatggcagacgggcgattttgttaaagccacttgcgaattttgtggcactgagaatttgactaaagaaggtgccactacttgtggttacttaccccaaaatgctgttgttaaa
atttattgtccagcatgtcacaattcagaagtaggacctgagcatagtcttgccgaataccataatgaatctggcttgaaaaccattcttcgtaagggtggtcgcactattgcctttggaggctg
tgtgttctcttatgttggttgccataacaagtgtgcctattgggttccacgtgctagcgctaacataggttgtaaccatacaggtgttgttggagaaggttccgaaggtcttaatgacaaccttctt
gaaatactccaaaaagagaaagtcaacatcaatattgttggtgactttaaacttaatgaagagatcgccattattttggcatctttttctgcttccacaagtgcttttgtggaaactgtgaaaggttt
ggattataaagcattcaaacaaattgttgaatcctgtggtaattttaaagttacaaaaggaaaagctaaaaaaggtgcctggaatattggtgaacagaaatcaatactgagtcctctttatgcatt
tgcatcagaggctgctcgtgttgtacgatcaattttctcccgcactcttgaaactgctcaaaattctgtgcgtgttttacagaaggccgctataacaatactagatggaatttcacagtattcactg
agactcattgatgctatgatgttcacatctgatttggctactaacaatctagttgtaatggcctacattacaggtggtgttgttcagttgacttcgcagtggctaactaacatctttggcactgtttat
gaaaaactcaaacccgtccttgattggcttgaagagaagtttaaggaaggtgtagagtttcttagagacggttgggaaattgttaaatttatctcaacctgtgcttgtgaaattgtcggtggaca
aattgtcacctgtgcaaaggaaattaaggagagtgttcagacattctttaagcttgtaaataaatttttggctttgtgtgctgactctatcattattggtggagctaaacttaaagccttgaatttag
gtgaaacatttgtcacgcactcaaagggattgtacagaaagtgtgttaaat……………………………………………………………………
……………………………………..

来源：中国科学院国家生物信息中心, https://www.ncbi.nlm.nih.gov/nuccore/MT019529.1

# 字符串模糊匹配

- 模糊匹配：降低精确匹配的要求：S和P之间只要有一定程度的相似，就认为匹配成功。
- 如何量化两个字符串之间的相似程度？
- 字符串编辑距离：将一个字符串转化成另一个字符串，需要的最少编辑操作次数。
- 编辑操作：增加一个字符、删除一个字符、替换一个字符。
- 编辑距离越大，说明两个字符串的相似程度越小；编辑距离越小，说明两个字符串的相似程度越大。对于两个完全相同的字符串，编辑距离为0。

# 字符串模糊匹配

- FOOD转换成MONEY：
- 编辑距离4：第1位F替换为M，第3位O替换为N，第4位插入E，第5位D替换为Y。

| F | O | O | | D |
|---|---|---|---|---|
| M | O | N | E | Y |

- 给定两个字符串S[1..m]和字符串P[1..n]，计算S和P的编辑距离。【华为、字节跳动、腾讯、百度、微软、苹果、谷歌、网易、滴滴、快手招聘面试题】

Dist(m, n)

# S和P的编辑距离Dist(m, n)

- 将S变换为P：若$S_m=P_n$

$S_1\ S_2\ \ldots\ S_{m-1}\ \ S_m$

$\downarrow$

$P_1\ P_2\ \ldots\ P_{n-1}\ \ P_n$

- 将S [1..m-1]变换为P[1..n-1]. 即

$$Dist(m, n)=Dist(m-1, n-1)$$

# S和P的编辑距离Dist(m, n)

- **将S变换为P：若$S_m \neq P_n$**

$$\boxed{S_1\ \ S_2\ \ \cdots\ \ S_{m-1}}\ \boxed{S_m}$$

$$\boxed{P_1\ \ P_2\ \ \cdots\ \ P_{n-1}}\ \boxed{P_n}$$

- **策略1：S[1..m-1]变换为P[1..n-1]，将S[m]替换为P[n].　Dist(m, n)=Dist(m-1, n-1)+1**

# S和P的编辑距离Dist(m, n)

- **将S变换为P：若$S_m \neq P_n$**

$S_1 \quad S_2 \quad \ldots \quad S_{m-1} \quad S_m$

$\Downarrow$

$P_1 \quad P_2 \quad \ldots \quad P_{n-1} \quad P_n$

- **策略2：S[1..m]变换为P[1..n-1]，并插入P[n].**

$$\text{Dist}(m, n) = \text{Dist}(m, n-1) + 1$$

# S和P的编辑距离Dist(m, n)

- 将S变换为P：若$S_m \neq P_n$

$$\boxed{S_1 \quad S_2 \quad \cdots \quad S_{m-1}} \quad \cancel{S_m}$$

$$\Downarrow$$

$$\boxed{P_1 \quad P_2 \quad \cdots \quad P_{n-1} \quad P_n}$$

- 策略3：S[1..m-1]变换为P[1..n]，并删去S[m].

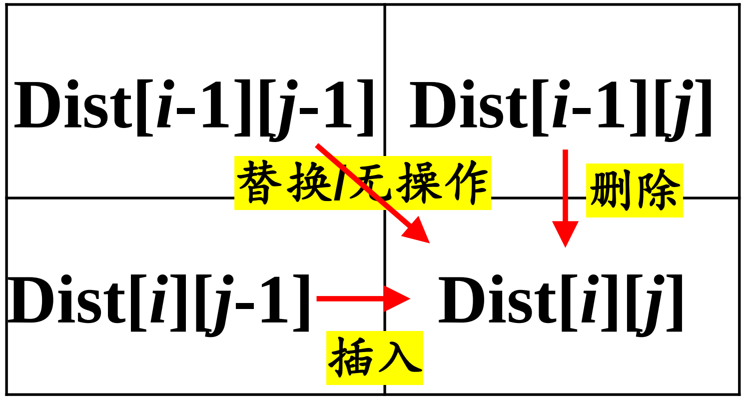
$$Dist(m, n) = Dist(m-1, n) + 1$$

# 字符串S和P的编辑距离

$$Dist(m,n)=\begin{cases} Dist(m-1,n-1), & S_m=P_n \\ \min\{Dist(m-1,n-1), Dist(m,n-1), Dist(m-1,n)\}+1, & S_m\neq P_n \end{cases}$$

## **Dist[i][j]表示$S_1 \ldots S_i$和$P_1 \ldots P_j$的编辑距离**

$$Dist[i][j]=\begin{cases} Dist[i-1][j-1], & S_i=P_j \\ \min\{Dist[i-1][j-1], Dist[i][j-1], Dist[i-1][j]\}+1, & S_i\neq P_j \end{cases}$$



# 字符串S和P的编辑距离——动态规划

$$Dist[i][j]=\begin{cases} Dist[i-1][j-1], & S_i = P_j \\ \min\{Dist[i-1][j-1], Dist[i][j-1], Dist[i-1][j]\}+1, & S_i \neq P_j \end{cases}$$

| | “ ” | B | C | D |
|---|---|---|---|---|
| “ ” | 0 | 1 | 2 | 3 |
| A | 1 | 1 | 2 | 3 |
| B | 2 | 1 | 2 | 3 |
| C | 3 | 2 | 1 | 2 |

# 字符串S和P的编辑距离——动态规划

$$Dist[i][j]=\begin{cases} Dist[i-1][j-1], & S_i = P_j \\ \min\{Dist[i-1][j-1], Dist[i][j-1], Dist[i-1][j]\}+1, & S_i \neq P_j \end{cases}$$

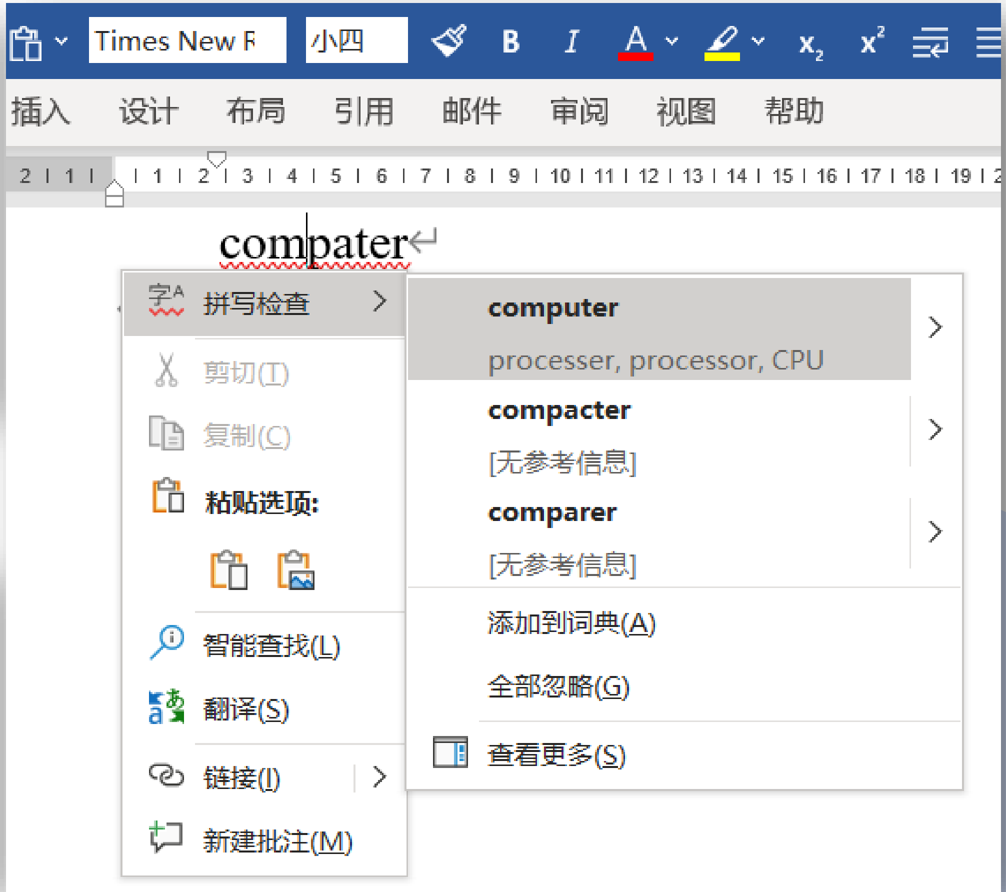
**FOR** ***i*=0 TO *m* DO Dist[*i*][0]←*i*.**
**FOR** ***j*=0 TO *n* DO Dist[0][*j*]←*j*.**
**FOR *i*=1 TO *m* DO**
　　**FOR *j*=1 TO *n* DO**
　　　　**IF $S_i = P_j$ THEN**
　　　　　　**Dist[*i*][*j*]←Dist[*i*-1][*j*-1].**
　　　　**ELSE**
　　　　　　**Dist[*i*][*j*]←min{Dist[*i*-1][*j*-1], Dist[*i*][*j*-1], Dist[*i*-1][*j*]}+1.**

| | |
|---|---|
| Dist[*i*-1][*j*-1] | Dist[*i*-1][*j*] |
| Dist[*i*][*j*-1] | Dist[*i*][*j*] |

**时间复杂度O(*nm*)**

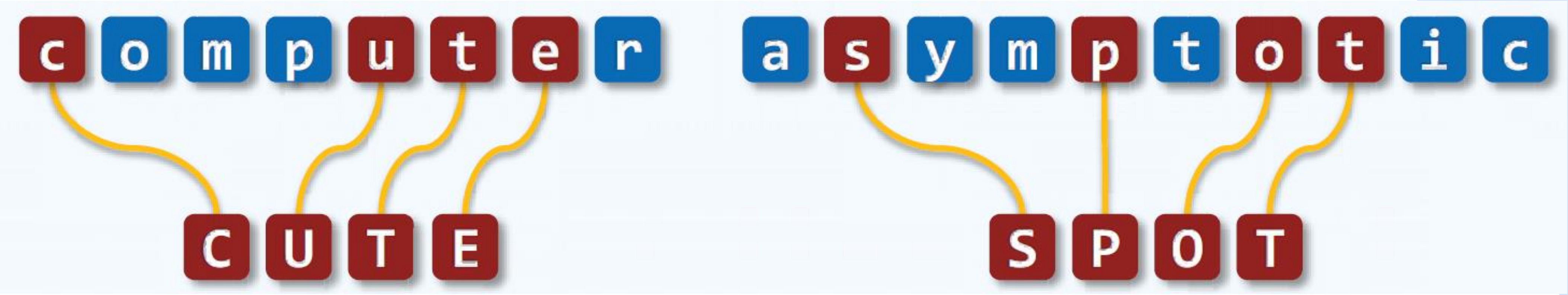
# 字符串S和P的编辑距离





# 最长公共子序列（LCS, Longest Common Sequence）

- 可用两个字符串的最长公共子序列长度衡量其相似程度。
- 给定两个字符串S和P，长度分别为m和n，要求找出它们最长的公共子序列长度。【百度、阿里、华为、字节跳动、腾讯、小米、美团、京东面试题】
- 子序列：由字符串中若干字符按原相对次序构成



# 最长公共子序列（LCS, Longest Common Sequence）

- 最长公共子序列：两个字符串的公共子序列中的最长者。
- 例：

  S = "Hello World"

  P = "loop“
- S和P的最长公共子序列为loo

# 最长公共子序列（LCS, Longest Common Sequence）

- 给定两个字符串S和P，长度分别为*m*和*n*，要求找出它们最长的公共子序列的长度。
- 考察$S_1 \ldots S_i$和$P_1 \ldots P_j$
- LCS(*i*, *j*)表示$S_1 \ldots S_i$和$P_1 \ldots P_j$的最长公共子序列长度。
- 若$S_i = P_j$，则LCS(*i*, *j*)=LCS(*i*-1, *j*-1)+1

$$\boxed{S_1 \quad \cdots \quad S_{i-1}} \qquad S_i$$

$$\boxed{P_1 \quad \cdots \quad P_{j-1}} \qquad P_j$$

# 最长公共子序列（LCS, Longest Common Sequence）

- 若$S_i \neq P_j$， 则$LCS(i, j)=\max\{\boxed{LCS(i\text{-}1, j)}, LCS(i, j\text{-}1)\}$

$$\begin{array}{ccccc} \boxed{S_1 & \cdots & S_{i-1}} & & S_i \\ \end{array}$$

$$\boxed{P_1 \quad \cdots \quad P_{j-1} \quad P_j}$$

# 最长公共子序列（LCS, Longest Common Sequence）

- 若$S_i \neq P_j$，则$LCS(i, j)=\max\{LCS(i-1, j), \boxed{LCS(i, j-1)}\}$

$$\boxed{S_1 \quad \cdots \quad S_{i-1} \quad S_i}$$

$$\boxed{P_1 \quad \cdots \quad P_{j-1}} \quad P_j$$

# 最长公共子序列（LCS, Longest Common Sequence）

- 若$S_i \neq P_j$， 则LCS($i$, $j$)=**max**{LCS($i$-1, $j$), LCS($i$, $j$-1)}

$$
\begin{array}{cccc}
S_1 & \cdots & S_{i-1} & S_i \\
P_1 & \cdots & P_{j-1} & P_j
\end{array}
$$

# 最长公共子序列（LCS, Longest Common Sequence）

$$LCS[i][j]=\begin{cases} LCS[i-1][j-1]+1, & S_i=P_j \\ \max\{LCS[i][j-1],LCS[i-1][j]\}, & S_i\neq P_j \end{cases}$$

***LCS*[*i*][*j*]表示$S_1 \ldots S_i$和$P_1 \ldots P_j$的最长公共子序列长度**

| | |
|---|---|
| **LCS[*i*-1][*j*-1]** | **LCS[*i*-1][*j*]** |
| **LCS[*i*][*j*-1]** $\rightarrow$ | **LCS[*i*][*j*]** |

**时间复杂度O(*nm*)**

# 自愿性质OJ练习题

- ✓ LeetCode 28（KMP算法裸题）
- ✓ POJ 3461（KMP变形，P在S中出现的次数）
- ✓ POJ 2752（相等前后缀数目）
- ✓ LeetCode 459（循环节，注意next数组要多算一位）
- ✓ POJ 2406（循环节）
- ✓ POJ 1961（循环节）
- ✓ LeetCode 72（字符串编辑距离）
- ✓ LeetCode 1143（最长公共子序列）