

KMP算法

11. 串

(a) ADT

邓俊辉

deng@tsinghua.edu.cn

定义

❖ 由来自**字母表 Σ** 的字符所组成的**有限序列**:

$$S = [a_0 \ a_1 \ a_2 \ \dots \ a_{n-1}] \in \Sigma^*$$



❖ 既然如此，为何不直接用**序列**来实现串？



❖ 通常，**字符的种类不多**，而串长 $= n \gg |\Sigma|$

❖ 比如： 英文文章 $(['A' - 'Z'] \cup ['a' - 'z'] \cup \{', ', '.', ',', '\', ... \})^*$

C/C++程序 $(\{95\text{个可打印字符}\} \cup \{\text{LF, CR}\})^*$

天然蛋白质 $\{\text{21种氨基酸}\}^*$

DNA $\{\text{A, C, G, T}\}^*$

RNA $\{\text{A, C, G, U}\}^*$

二进制 $\{\text{0, 1}\}^*$

术语

❖ 相等: $S[0, n) = T[0, m)$

 长度相等 ($n = m$) , 且对应的字符均相同 ($S[i] = T[i]$)

❖ 子串: $S.substr(i, k) = S[i, i + k), 0 \leq i < n, 0 \leq k$

 亦即, 从 $S[i]$ 起的连续 k 个字符 $[0, i)$ $[i, i + k)$ $[i + k, n)$

❖ 前缀: $S.prefix(k) = S.substr(0, k) = S[0, k), 0 \leq k \leq n$

 亦即, S 中最靠前的 k 个字符 $[0, k)$ $[k, n)$

❖ 后缀: $S.suffix(k) = S.substr(n - k, k) = S[n - k, n), 0 \leq k \leq n$

 亦即, S 中最靠后的 k 个字符 $[0, n - k)$ $[n - k, n)$

❖ 联系: $S.substr(i, k) = S.prefix(i + k).suffix(k)$

❖ 空串: $S[0, n = 0)$, 也是任何串的子串、前缀、后缀

ADT**length()**[0, **n**)**charAt(i)**

[0, i)

[i]

(i, n)

substr(i, k)

[0, i)

[i, i + k)

[i + k, n)

prefix(k)

[0, k)

[k, n)

suffix(k)

[0, n - k)

[n - k, n)

concat(T)**S****T****equal(T)****S****T****indexOf(P)****S**[**k** , k + m)

P[0, m)

实例

- ❖ "data structures".length() = 15
- "data structures".charAt(5) = 's'
- "data structures".prefix(4) = "data"
- "data structures".suffix(10) = "structures"
- "data structures".concat(" & algorithms") = "data structures & algorithms"
- "algorithms".equal("data structures") = false
- "data structures and algorithms".indexOf("string") = -1
- "data structures and algorithms".indexOf("algorithm") = 20
- ❖ <string.h> 中的对应功能: strlen()、strcpy()、 strcat()、 strcmp()、 strstr()
- ❖ 以下，直接利用字符数组实现字符串，转而重点讨论串匹配算法

11. 串

(b1) 串匹配

邓俊辉

deng@tsinghua.edu.cn

串匹配

% grep <pattern> <text>

文本 T = now is the time for all good people to come

模式 P = people

❖ 记 $n = |T|$ 和 $m = |P|$, 通常有 $n \gg m \gg 2$ //比如, 100,000 >> 100 >> 2

❖ Pattern matching

detection: P是否出现?

location: 首次在哪里出现?

//本章主要讨论的问题

counting: 共有几次出现?

//find /c "2013" students.txt

enumeration: 各出现在哪里?

//find "2013" students.txt

串匹配

❖ **歧义:** T = " 1 0 0 1 **1 0 1 1** 0 **1 0 1** **1** **0 1 1** 1 0 0 1 "

P = " **1 0 1 1** "

❖ **应用:** 文本编辑器、数据库检索、C++模板匹配、模式识别、搜索引擎、...

❖ **应用:** 生物序列分析 (biological sequence analysis)

通常不能完全匹配

——alignment: 最**接近**的匹配在什么位置?

HBA_HUMAN vs. HBB_HUMAN

G	S	A	Q	V K	G	H G K K V	A	D	A	L	T	N	A	V	A H	V	D	D	M	P	N	A	L	S	A	L S	D	L H	A	H
G	N	P	K	V K	A	H G K K V	L	G	A	F	S	D	G	L	A H	L	D	N	L	K	G	T	F	A	T	L S	E	L H	C	D

算法评测

❖ 如何客观地 **测量与评估** 串匹配算法的**性能**？具体采用什么**标准与策略**？

❖ **随机T + 随机P？不妥！**

❖ 以 $\Sigma = \{0, 1\}^*$ 为例

$$|\{ \text{长度为 } m \text{ 的 } P \}| = 2^m$$

$$|\{ \text{长度为 } m \text{ 且在 } T \text{ 中出现的 } P \}| = n - m + 1 < n$$

匹配成功的概率 $= n/2^m \ll 100,000 / 2^{100} < 10^{-25}$

如此，将无法对算法做充分测试

❖ **随机T**，对成功、失败的匹配**分别** 测试

成功：在T中，随机取出长度为m的**子串**作为P；分析平均复杂度

失败：采用**随机**的P；统计平均复杂度

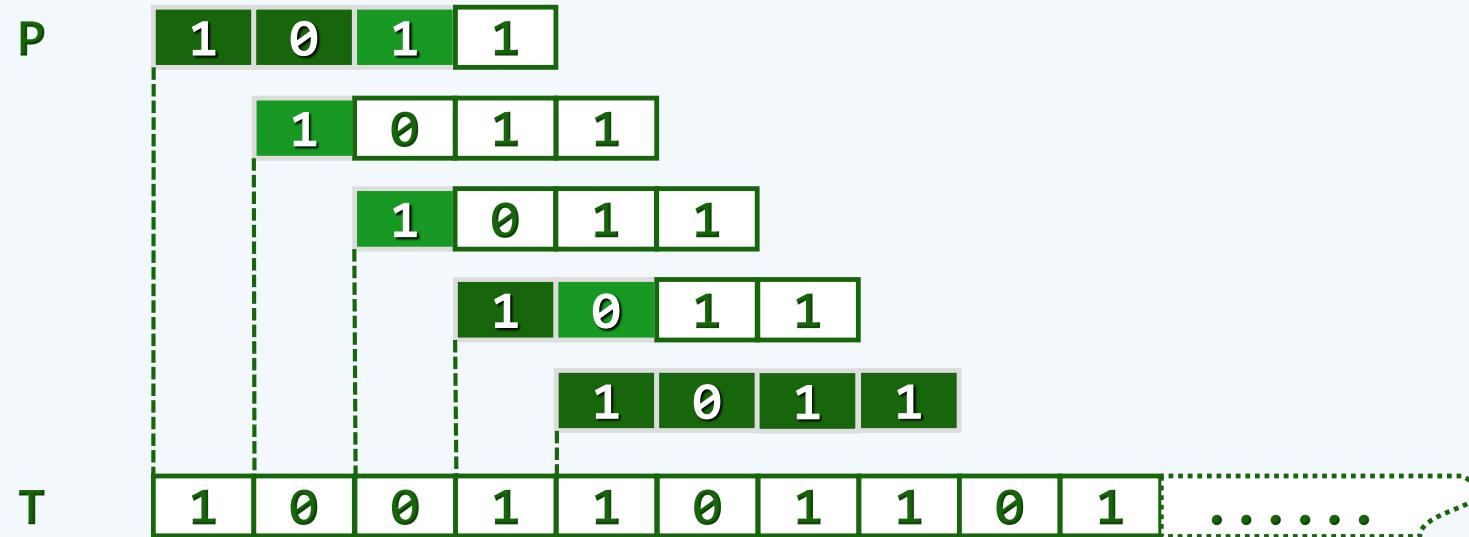
11. 串

(b2) 蛮力匹配

邓俊辉

deng@tsinghua.edu.cn

❖ 自左向右，以字符为单位，依次移动模式串
直到在某个位置，发现匹配



❖ 如何： 确定串T和P每次做比对的字符位置？并
在发现某对字符失配后，调整位置以继续比对？

实现

❖ 实现1： i和j分别指向T[]和P[]中待比对的字符

```
if ( T[i] == P[j] ) { i++; j++; } //若匹配，则i和j同时右移  
else { i -= j - 1; j = 0; } //若失配，则T回退、P复位
```

❖ 实现2： i + j和j分别指向T[]和P[]中待比对的字符

```
if ( T[i + j] == P[j] ) { j++; } //若匹配，则i + j和j同时右移  
else { i++; j = 0; } //若失配，则i右移，j回溯
```

❖ 这两种实现方法，各有什么优缺点？

版本1

```
❖ int match( char * P, char * T ) {
```

```
    size_t n = strlen(T), i = 0;
```

```
    size_t m = strlen(P), j = 0;
```

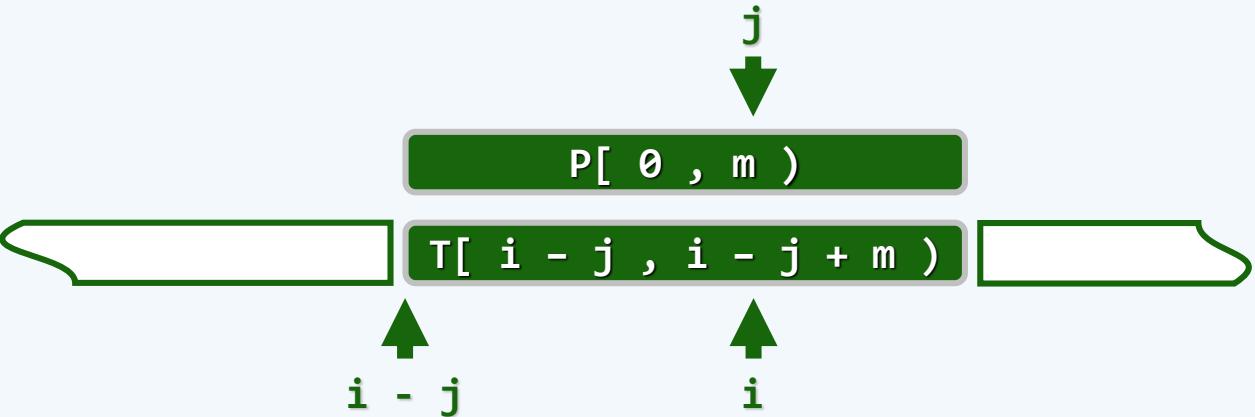
```
    while ( j < m && i < n ) //自左向右逐个比对字符
```

```
        if ( T[i] == P[j] ) { i++; j++; } //若匹配，则转到下一对字符
```

```
        else { i -= j - 1; j = 0; } //否则，T回退、P复位
```

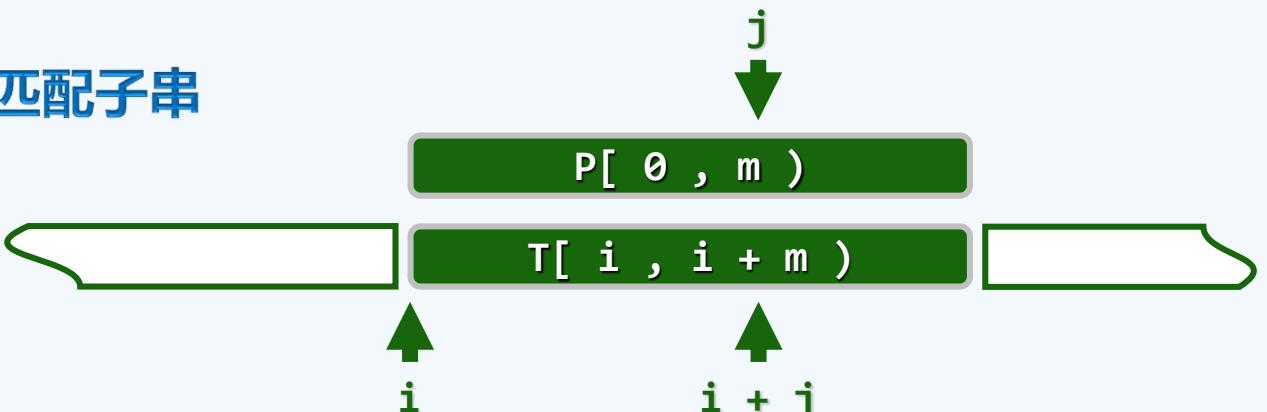
```
    return i - j;
```

```
} //如何通过返回值，判断匹配结果？
```



版本2

```
❖ int match( char * P, char * T ) {  
    size_t n = strlen(T), i = 0; //T[i]与P[0]对齐  
    size_t m = strlen(P), j; //T[i + j]与P[j]对齐  
    for ( i = 0; i < n - m + 1; i ++ ) { //T从第i个字符起, 与  
        for ( j = 0; j < m; j ++ ) //P中对应的字符逐个比对  
            if ( T[i + j] != P[j] ) break; //若失配, P整体右移一个字符, 重新比对  
        if ( m <= j ) break; //找到匹配子串  
    }  
    return i;  
} //如何通过返回值, 判断匹配结果?
```



复杂度

❖ 最好情况 (只经过一轮比对, 即可确定匹配) : #比对 = m = O(m)

❖ 最坏情况 (每轮都比对至P的末字符, 且反复如此)

每轮循环: #比对 = m - 1(成功) + 1(失败) = m

循环次数 = n - m + 1

一般地有 m << n

故总体地, #比对 = m × (n - m + 1) = O(n × m)

❖ 最坏情况, 真会出现?



是的!



❖ Σ 越小, 最坏情况出现的概率越高; m 越大, 最坏情况的后果更加严重

11. 串

(c1) KMP算法：从记忆力到预知力

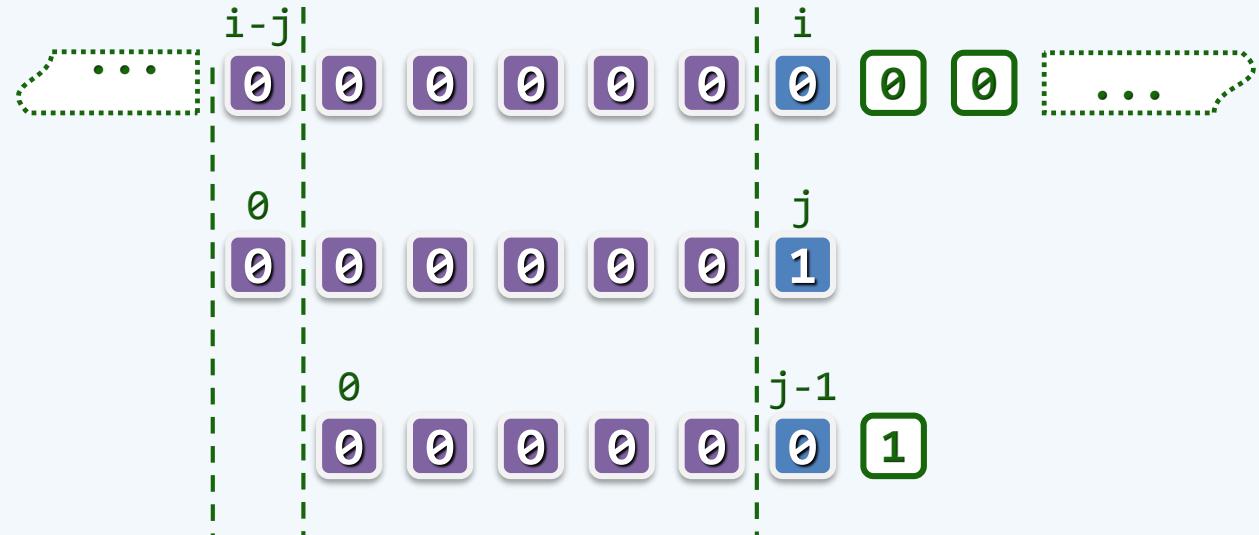
知易者不占，善易者不卜

邓俊辉

deng@tsinghua.edu.cn

蛮力，为何低效

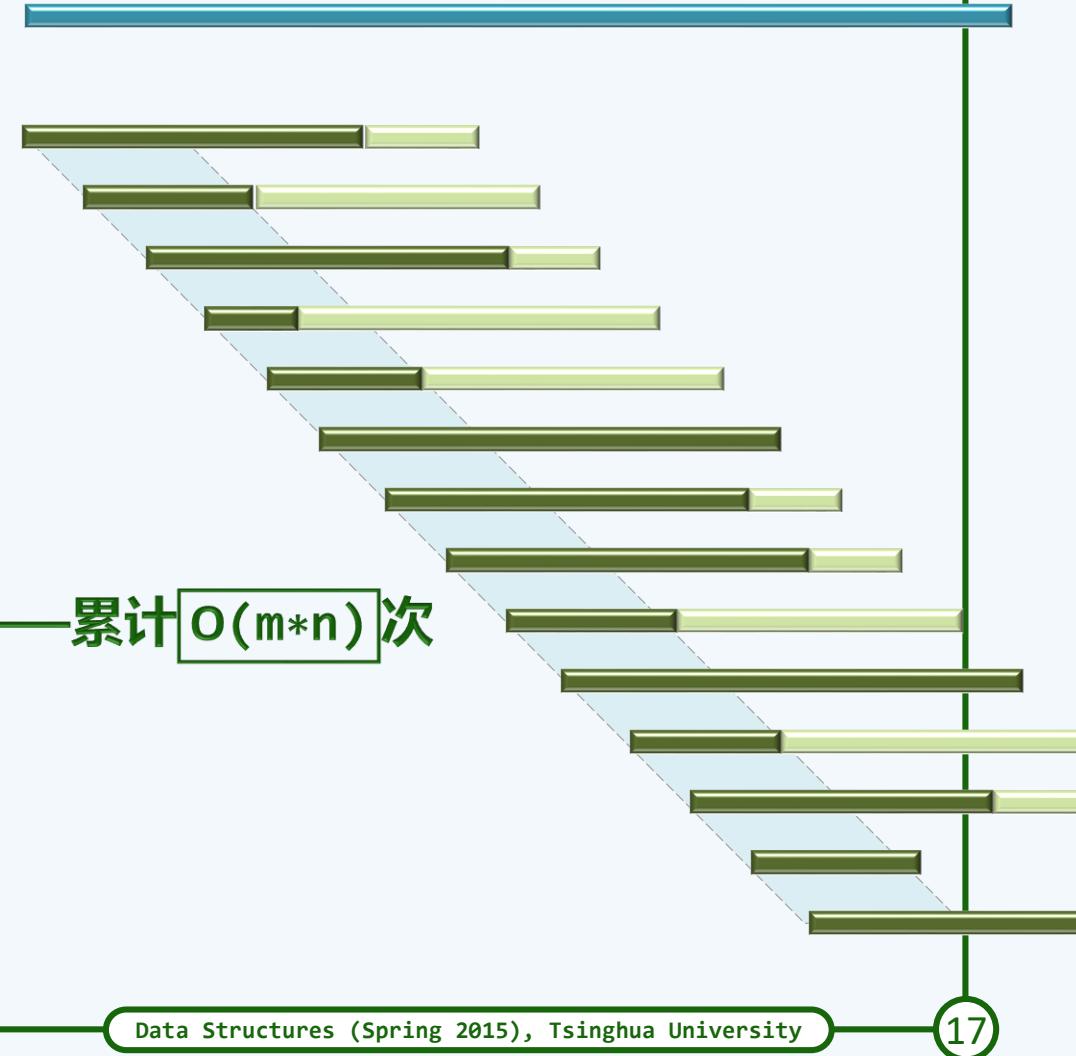
❖ T回退、P复位之后，此前比对过的字符，将再次参与比对



❖ 最坏情况下，T/P中每个字符平均参加 m/n 次比对——累计 $O(m*n)$ 次

❖ 于是，只要局部匹配很多，效率必将很低

❖ 其实，这类比对大多是不必要的，因为...



无论如何，还是不变性

❖ $T[i - j, i] == P[0, j]$

$T[i] == P[j]$

?

$T[i - j, i]$

*

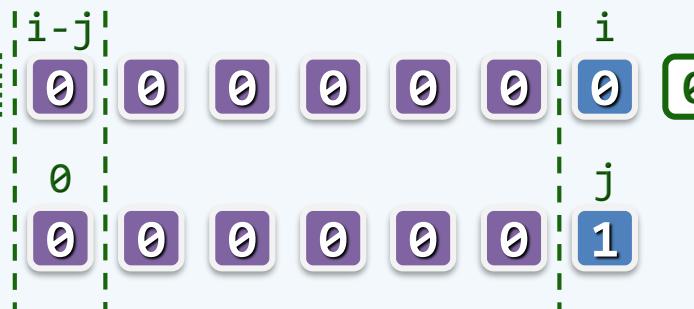
$T(i, n)$

$P[0, j]$

*

$P(j, m)$

❖ 亦即，我们业已掌握 $T[i - j, i]$ 的全部信息——其中的字符各是什么

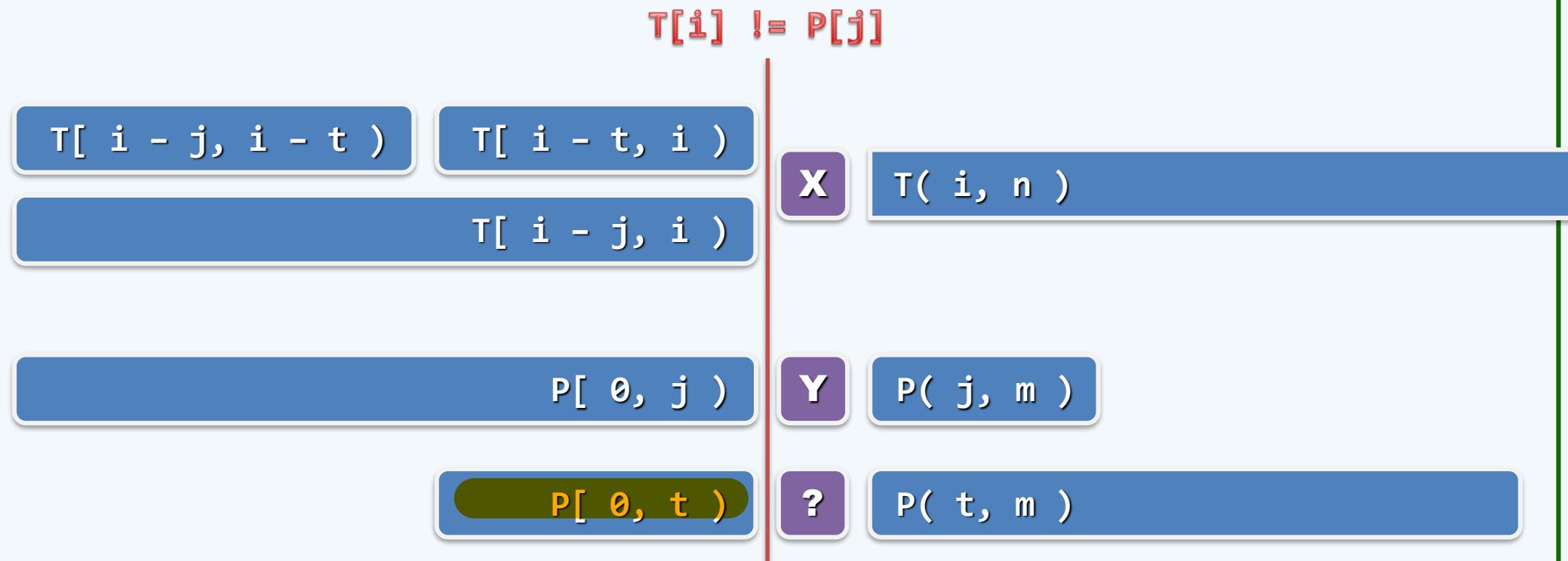


❖ 既如此...

只要**记忆力**足够强

❖ 在失败之后的下一轮比对中...

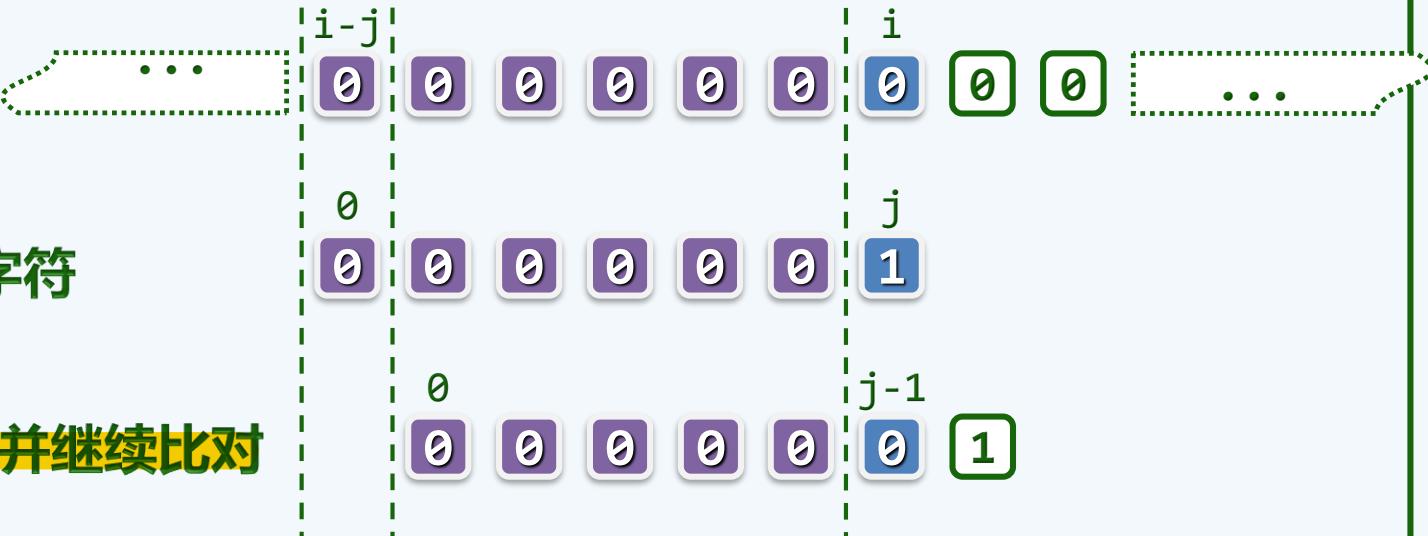
❖ $T[i-j, i]$ 就**不必再次**接受比对，而是可以直接地...



将记忆力，转化为预知力

❖ 如此，**i将完全不必回退！**

- 比对成功，则与**j**同步前进一个字符
- 否则，**j更新为某个更小的t，并继续比对**

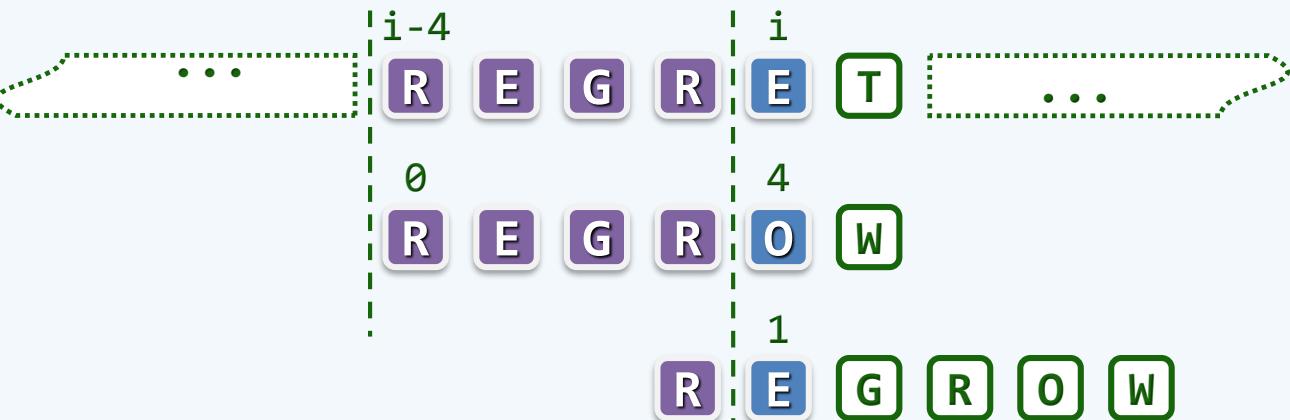


❖ 即便是更为复杂的情况，依然可行

❖ 优化 = P可快速右移 + 避免重复比对

❖ 为确定**t**，需花费多少时间和空间？

更重要地，可否在**事先**就确定？



11. 串

(c2) KMP算法：查询表

邓俊辉

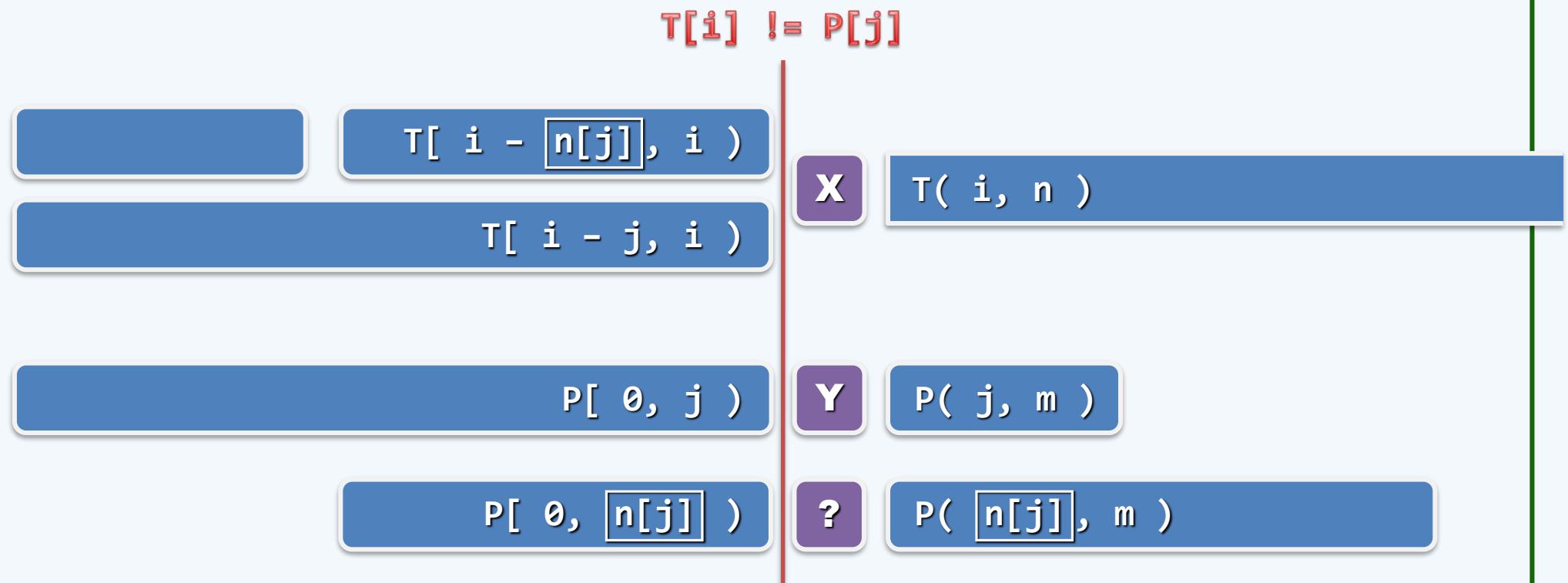
好记性不如烂笔头

deng@tsinghua.edu.cn

事先确定 t

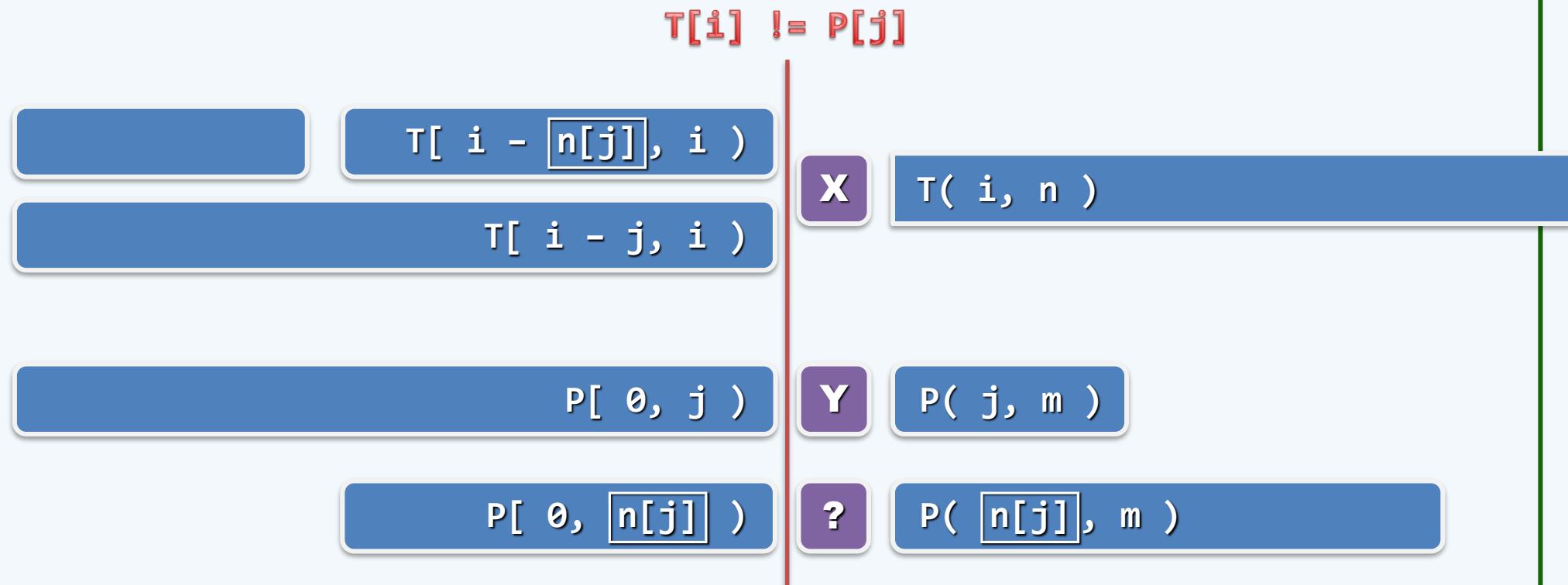
不仅可以事先确定，而且仅根据 P 即可确定（与 T 无关！）

根据失败位置 $P[j]$ ，无非 m 种情况...



事先确定 t

- 构造查询表 $\text{next}[0, m]$: 在任一位置 $P[j]$ 处失败之后，将 j 替换为 $\text{next}[j]$
- 与其说是借助 强大的记忆，不如说是做好 充分的预案



KMP算法

```
❖ int match( char * P, char * T ) {  
    int * next = buildNext(P); //构造next表  
    int n = (int) strlen(T), i = 0; //文本串指针  
    int m = (int) strlen(P), j = 0; //模式串指针  
    while ( j < m && i < n ) //自左向右, 逐个比对字符  
        if ( 0 > j || T[i] == P[j] ) { //若匹配  
            i++; j++; //则携手共进  
        } else //否则, P右移, T不回退  
            j = next[j];  
    delete [] next; //释放next表  
    return i - j;  
}
```



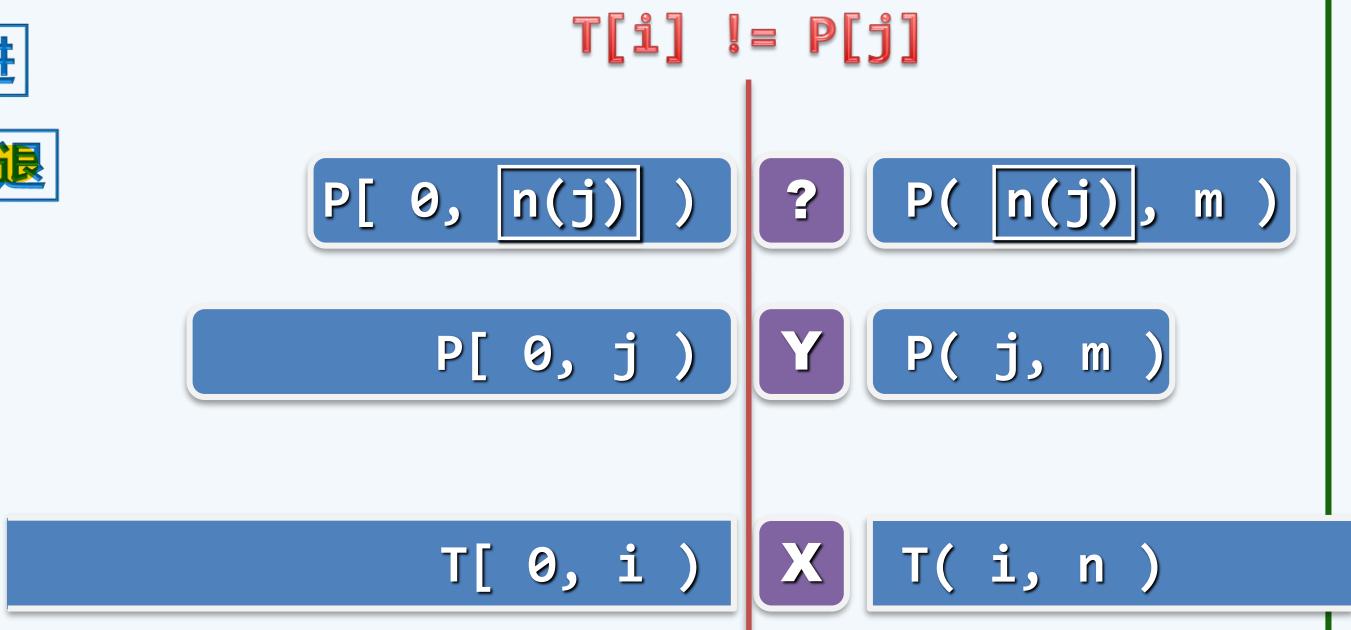
D. E. Knuth



J. H. Morris



V. R. Pratt



FAST PATTERN MATCHING IN STRINGS*

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

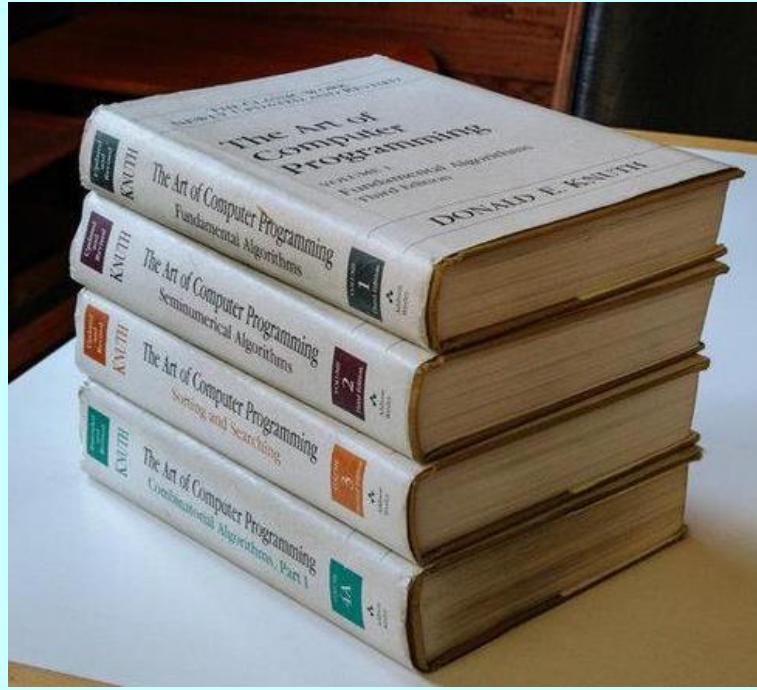
Abstract. An algorithm is presented which finds all occurrences of one given string within another, in running time proportional to the sum of the lengths of the strings. The constant of proportionality is low enough to make this algorithm of practical use, and the procedure can also be extended to deal with some more general pattern-matching problems. A theoretical application of the algorithm shows that the set of concatenations of even palindromes, i.e., the language $\{\alpha\alpha^R\}^*$, can be recognized in linear time. Other algorithms which run even faster on the average are also considered.

Key words. pattern, string, text-editing, pattern-matching, trie memory, searching, period of a string, palindrome, optimum algorithm, Fibonacci string, regular expression

$j = 1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8 \ 9 \ 10$

$pattern[j] = a \ b \ c \ a \ b \ c \ a \ c \ a \ b$

$next[j] = 0 \ 1 \ 1 \ 0 \ 1 \ 1 \ 0 \ 5 \ 0 \ 1$



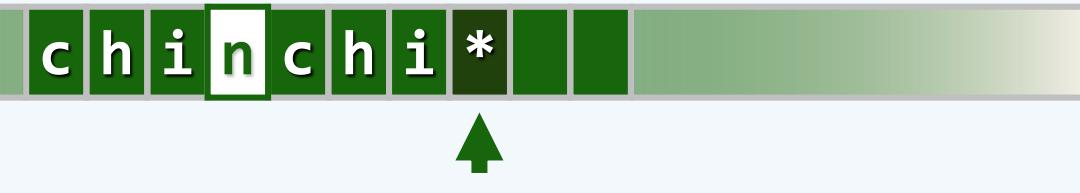
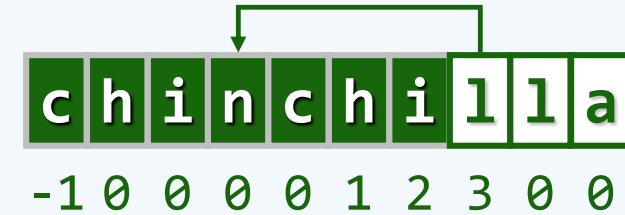
最年轻的图灵奖获得者

(本学期第几位?)

高德纳 (Donald Ervin Knuth, 1938年) , 美国著名计算机科学家, 斯坦福大学电脑系荣誉教授。高德纳教授被誉为现代计算机科学的鼻祖

他是《计算机程序设计艺术》一书的作者, 这部四卷本的作品可谓是他毕生的心血。第一卷出版于1968年, 整部作品(作为一套盒装出售, 售价约250美元), 在2013年被纳入《美国科学家》(American Scientist) 评选的对上一世纪科学界最具影响力著作的榜单——其它入选的著作包括《查尔斯·达尔文自传》的特别版、汤姆·沃尔夫 (Tom Wolfe) 的《真材实料》 (The Right Stuff) 、雷切尔·卡森 (Rachel Carson) 的《寂静的春天》 (Silent Spring) 以及阿尔伯特·爱因斯坦 (Albert Einstein) 和约翰·冯·诺依曼 (John von Neumann) 和理查德·费曼 (Richard Feynman) 的专著。

实例



❖ `int match(char * T) { //对任一模式串 (比如P = chinchilla) , 可自动生成如下代码`

`int n = strlen(T); int i = -1; //文本串对齐位置`

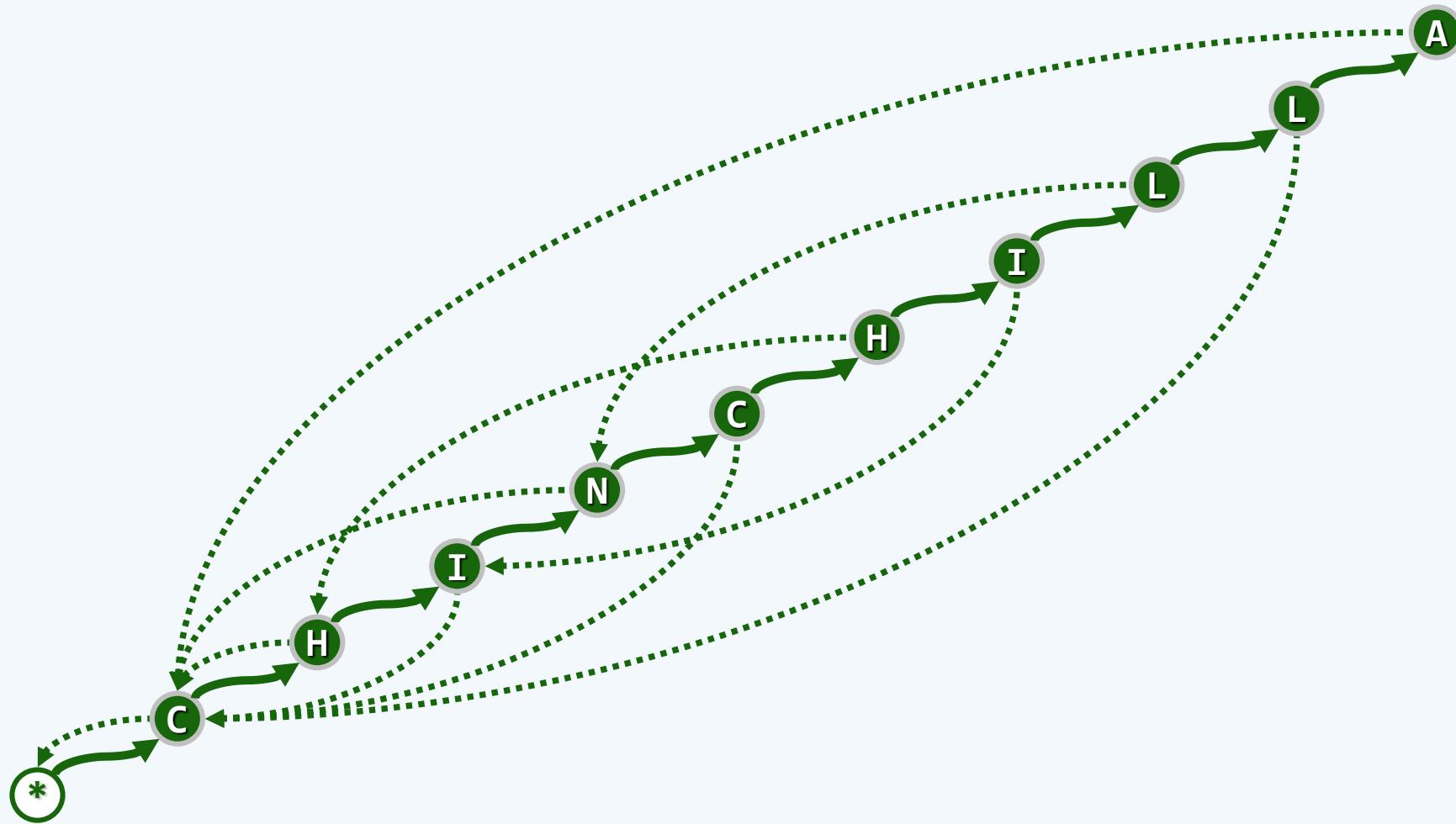
```

s_-: ++i;                                // ↑
s0: (T[i] != 'C') ? goto s_ : if (n <= ++i) return -1; // *      ~ ↑
s1: (T[i] != 'H') ? goto s0 : if (n <= ++i) return -1; // *C     ~ *
s2: (T[i] != 'I') ? goto s0 : if (n <= ++i) return -1; // *CH    ~ *
s3: (T[i] != 'N') ? goto s0 : if (n <= ++i) return -1; // *CHI   ~ *
s4: (T[i] != 'C') ? goto s0 : if (n <= ++i) return -1; // *CHIN  ~ *
s5: (T[i] != 'H') ? goto s1 : if (n <= ++i) return -1; // *CHINC ~ *C
s6: (T[i] != 'I') ? goto s2 : if (n <= ++i) return -1; // *CHINCH ~ *CH
s7: (T[i] != 'L') ? goto s3 : if (n <= ++i) return -1; // *CHINCHI ~ *CHI
s8: (T[i] != 'L') ? goto s0 : if (n <= ++i) return -1; // *CHINCHIL ~ *
s9: (T[i] != 'A') ? goto s0 : if (n <= ++i) return -1; // *CHINCHILL ~ *
                                         // *CHINCHILLA

    return i - [10];
}

```

自动机



11. 串

(c3) KMP算法：理解next[]表

吴用再使时迁扮作伏路小军，去曾头市寨中，探听他不出何意，所有陷坑，暗暗地记着，离寨多少路远，总有几处。时迁去了一日，都知备细，暗地使了记号，回报军师。

邓俊辉

deng@tsinghua.edu.cn

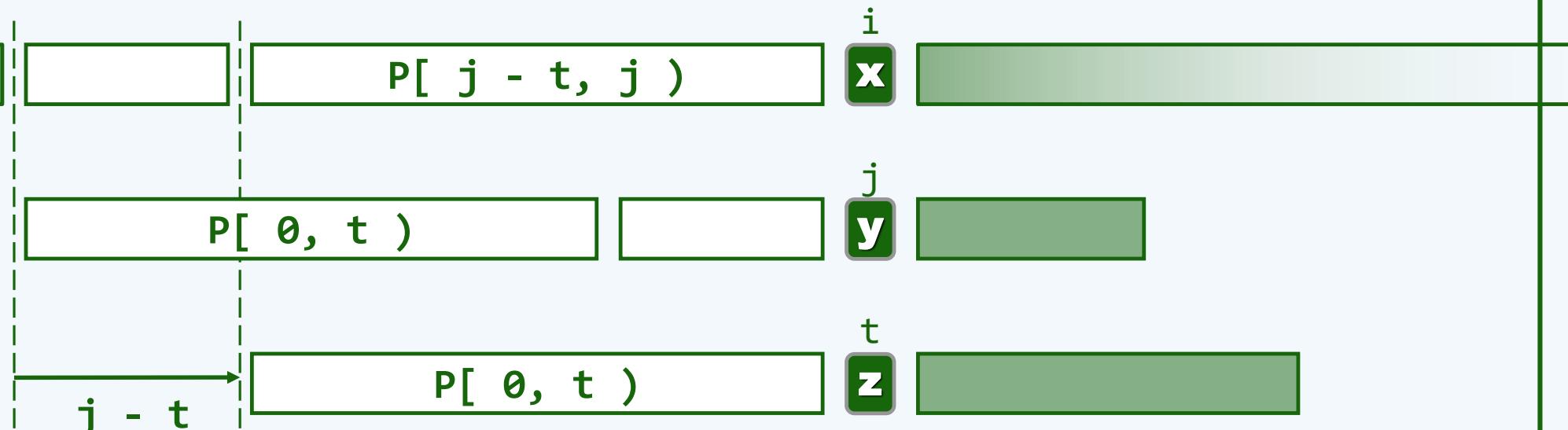
自匹配 = 快速右移

❖ 对任意 j , 考察集合:

$$N(P, j) = \{ \theta \leq t < j \mid P[\theta, t) == P[j-t, j) \}$$

亦即, 在 $P[j]$ 的前缀 $P[\theta, j)$ 中, 所有匹配 真前缀 和 真后缀 的长度

❖ 因此, 一旦 $T[i] \neq P[j]$, 可从 $N(P, j)$ 中取 某个 t , 令 $P[t]$ 对准 $T[i]$, 并继续比对



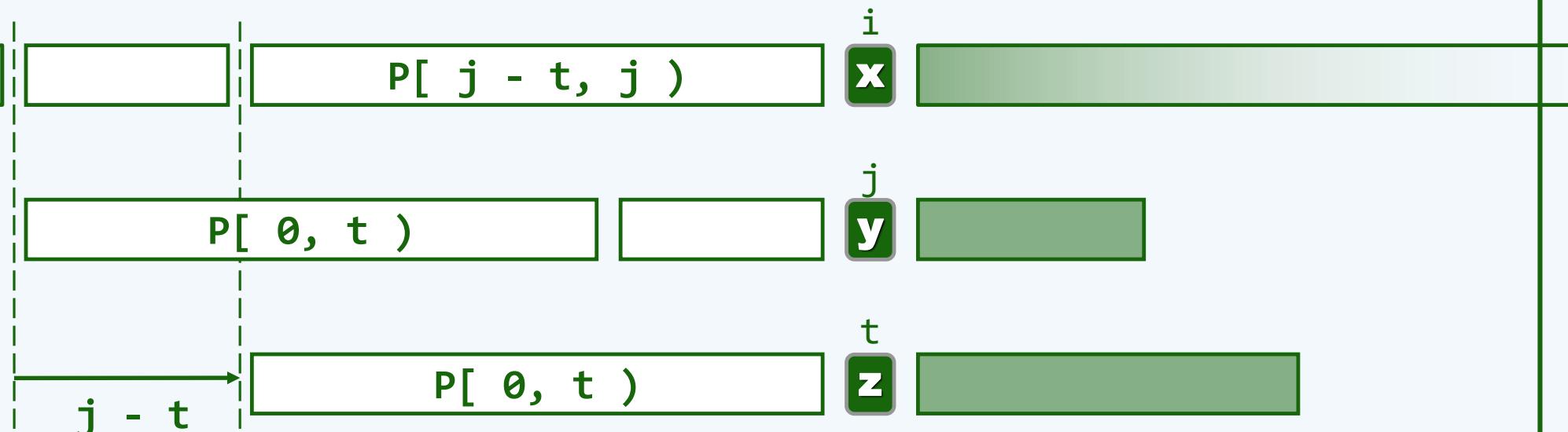
最长自匹配 = 快速右移 + 避免回退

❖ $|N(P, j)| > 1$ 时，难道需要遍历其中的每一个 t ？

❖ 不必！

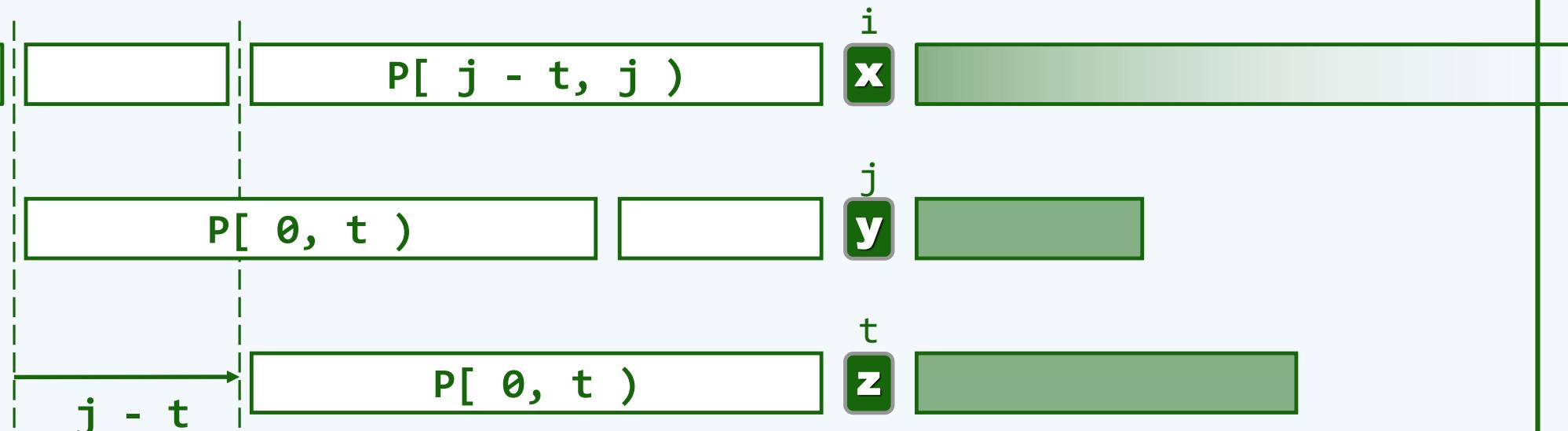
❖ 观察：位移量 = $j - t$ ，与 t 成反比

❖ 因此，若选用最大的 t ，则必然最安全



next[θ]

- ❖ 只要 $j > \theta$, 必有 $\theta \in N(P, j)$ // 空串是任何非空串的真子串
- ❖ 但若 $j = \theta$, 则有 $N(P, \theta) = \emptyset$ // 空串没有真子串
- ❖ 不妨取 $next[\theta] = -1$...
- ❖ 回顾主算法: 行之有效! 如何理解?



11. 串

(c4) KMP算法：构造next[]表

邓俊辉

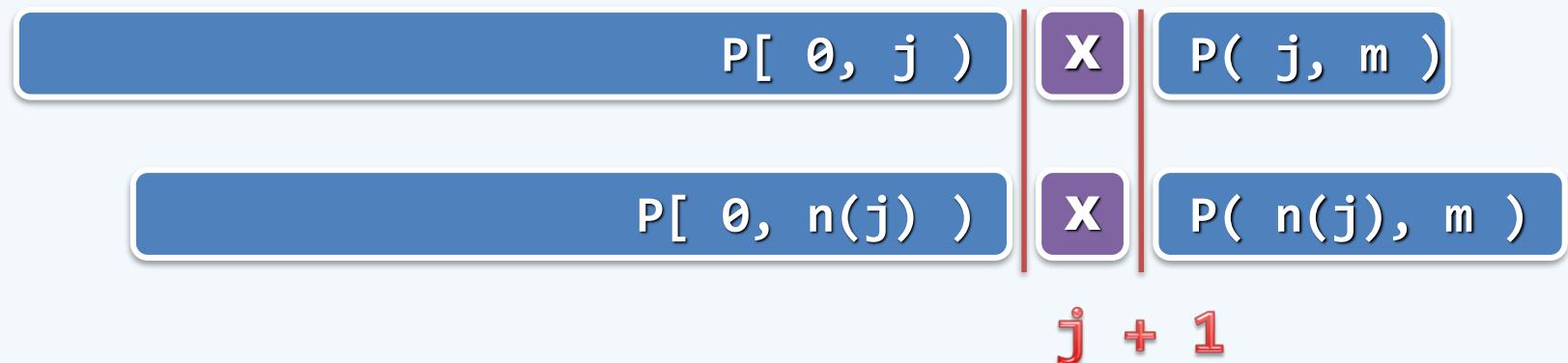
deng@tsinghua.edu.cn

递推

- ❖ 根据已知的 $\text{next}[0, j]$, 如何高效地计算 $\text{next}[j + 1]$?
- ❖ 所谓 $\text{next}(j)$, 即是在 $P[0, j]$ 中, 最大自匹配的真前缀和真后缀的长度
- ❖ 故: $\text{next}[j + 1] \leq \text{next}[j] + 1$

特别地, 当且仅当 $P[j] == P[\text{next}[j]]$ 时取等号

j



- ❖ 一般地, $P[j] != P[\text{next}[j]]$ 时, 又该如何得到 $\text{next}[j + 1]$?

算法

❖ $\text{next}[j + 1]$ 的候选者

依次应该是：

$1 + \boxed{\text{next}[j]}$

$1 + \boxed{\text{next}[\boxed{\text{next}[j]}]}$

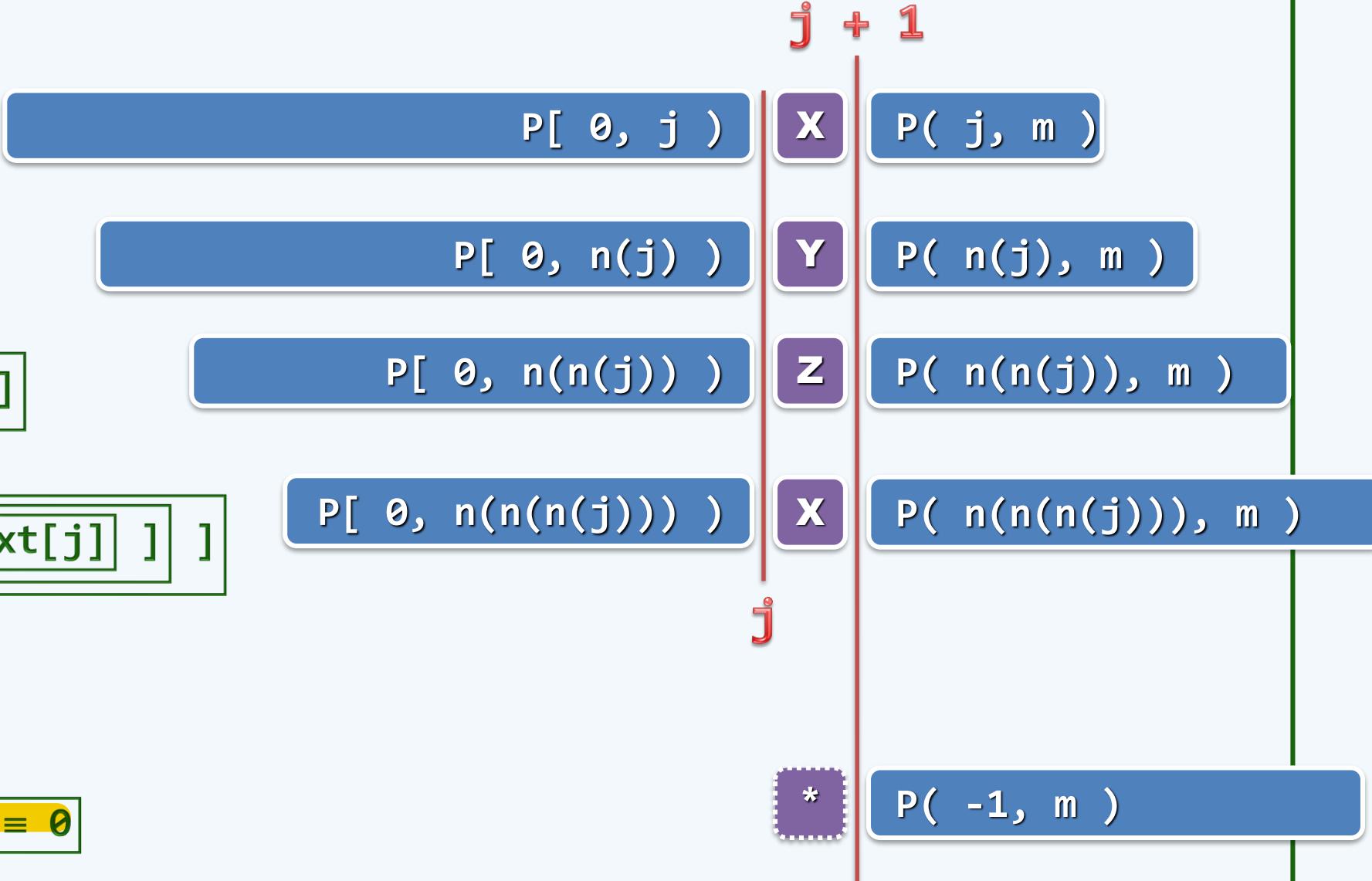
$1 + \boxed{\text{next}[\boxed{\text{next}[\boxed{\text{next}[j]}]}]}$

...

❖ 这个序列严格递减，且

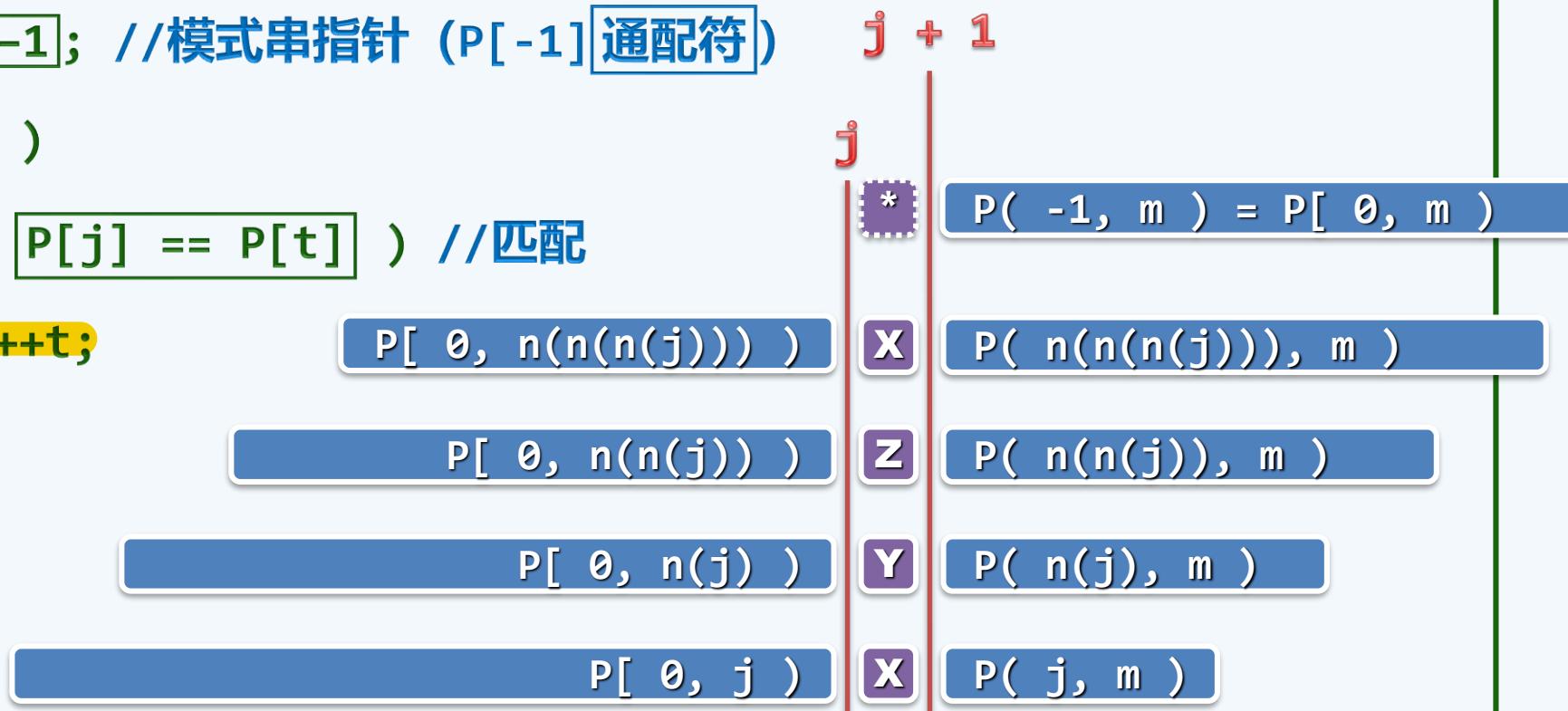
必收敛于 $1 + \text{next}[0] = 0$

❖ 以上递推过程，即是 P 的 **自匹配过程**，故只需对 KMP 框架略做修改...

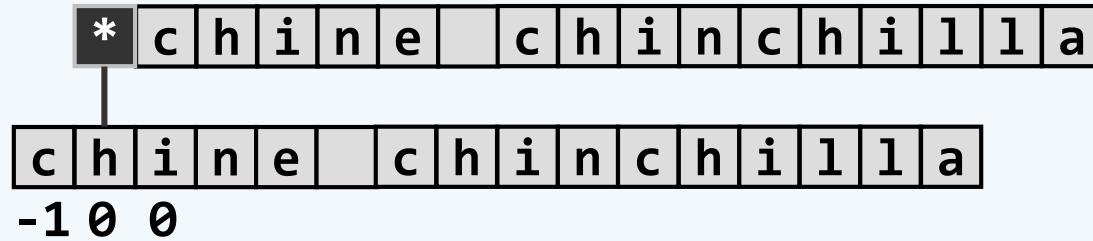
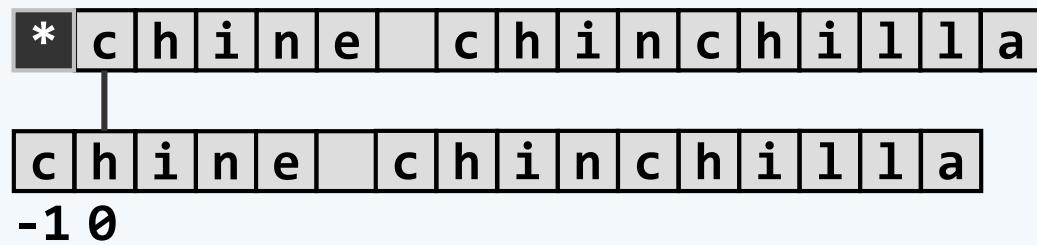
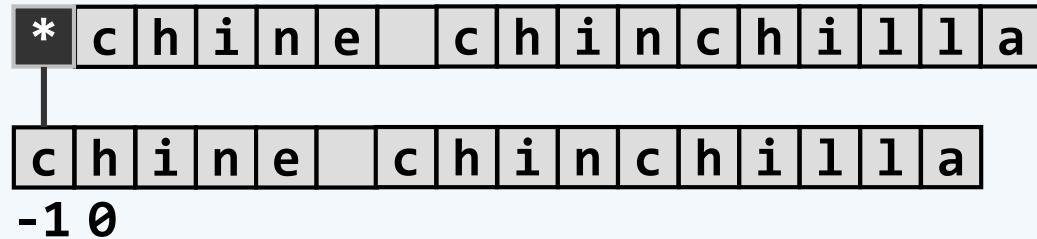


实现

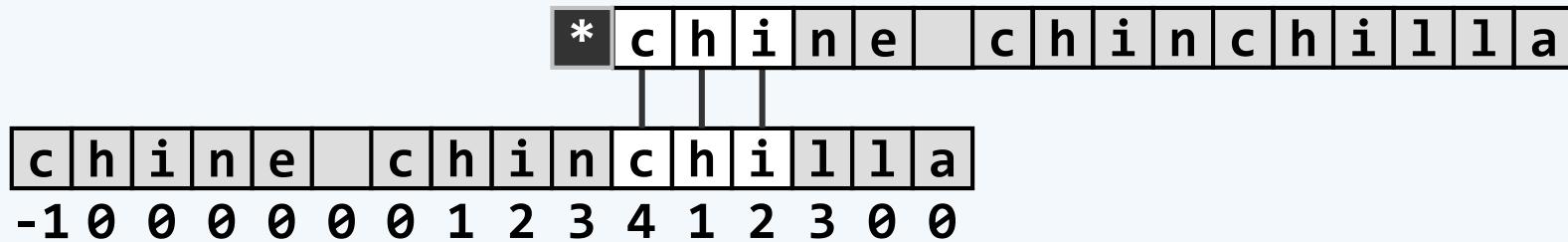
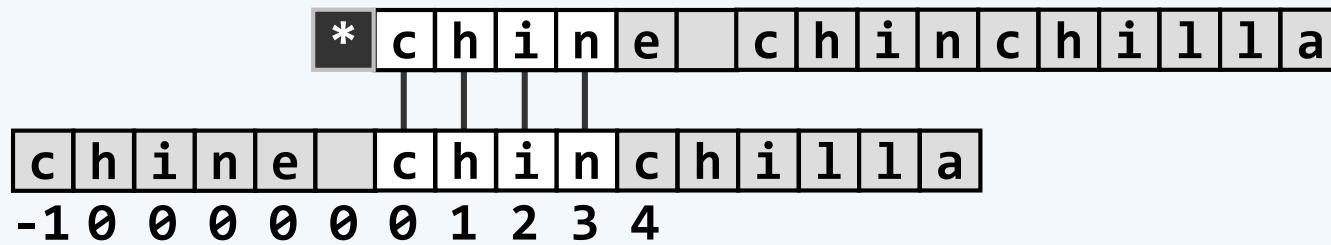
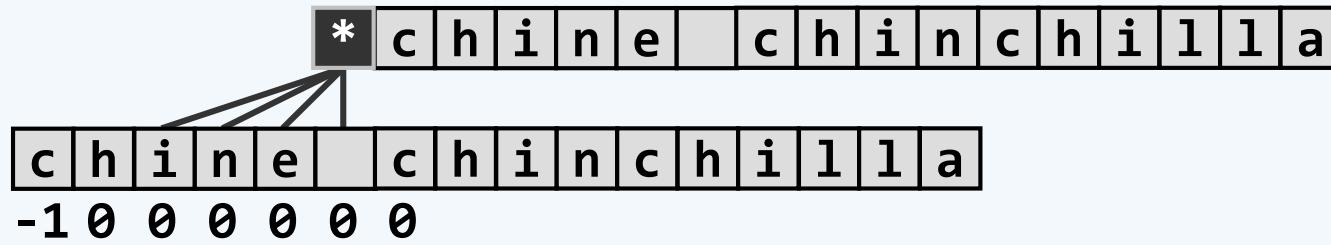
```
❖ int * buildNext( char * P ) { //构造模式串P的next[]表
    size_t m = strlen(P), j = 0; //“主”串指针
    int * N = new int[m]; //next[]表
    int t = N[0] = -1; //模式串指针 (P[-1]通配符)
    while ( j < m - 1 )
        if ( 0 > t || P[j] == P[t] ) //匹配
            N[ ++j ] = ++t;
        else //失配
            t = N[t];
    return N;
}
```



实例



实例



11. 串

(c5) KMP算法：分摊分析

邓俊辉

失之东隅，收之桑榆

deng@tsinghua.edu.cn

$\Omega(n * m)?$

❖ 观察： KMP算法的确可以节省多次比对

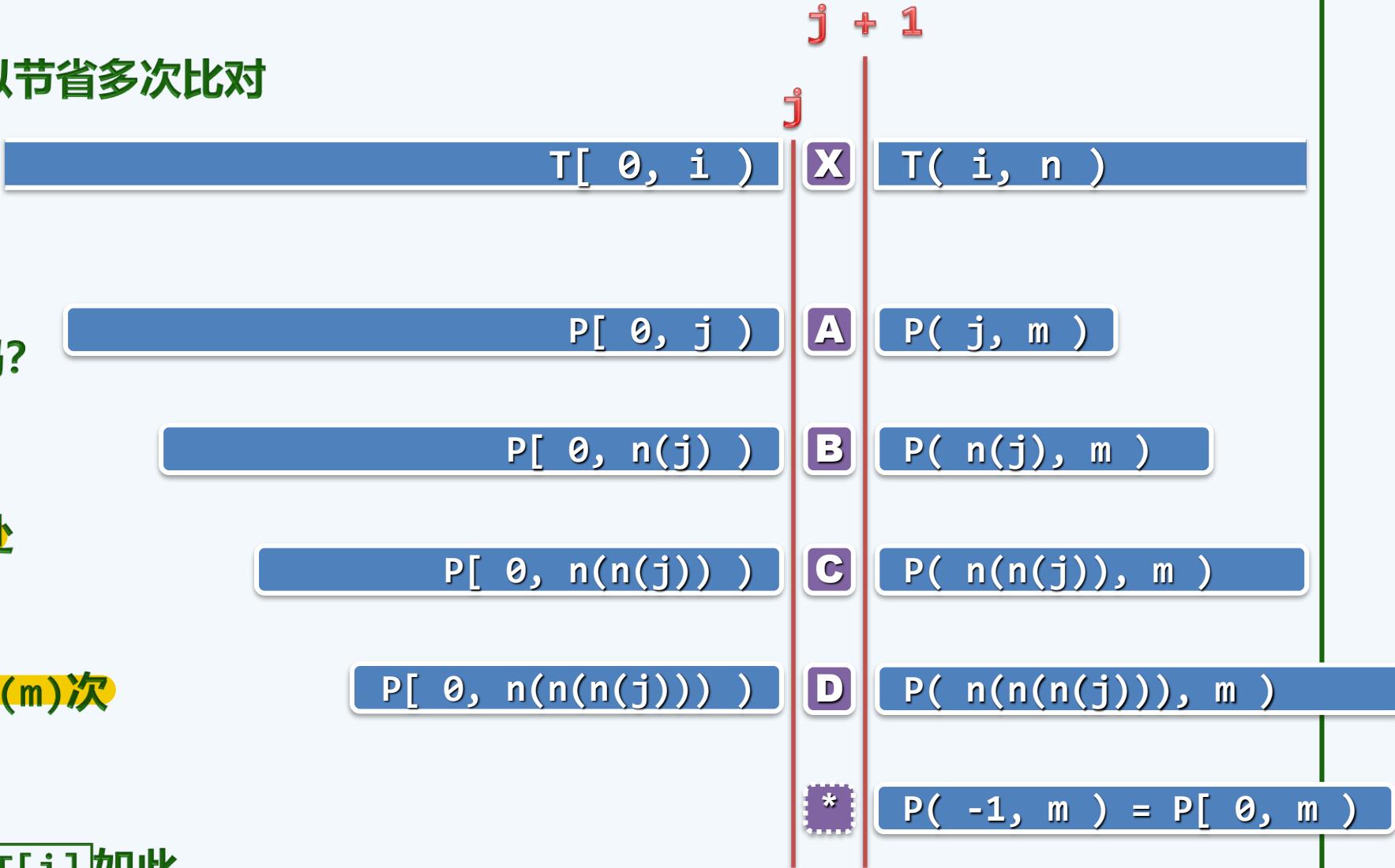
❖ 然而： 就渐进意义而言

有实质的节省吗？

❖ 观察： 在每一个 $T[i]$ 处

P都可能比对 $\Omega(m)$ 次

❖ 于是， 倘若有 $\Omega(n)$ 个 $T[i]$ 如此...



$\Omega(n * m)$?

❖ 难道， 总体复杂度还是

$\Omega(n * m)$?

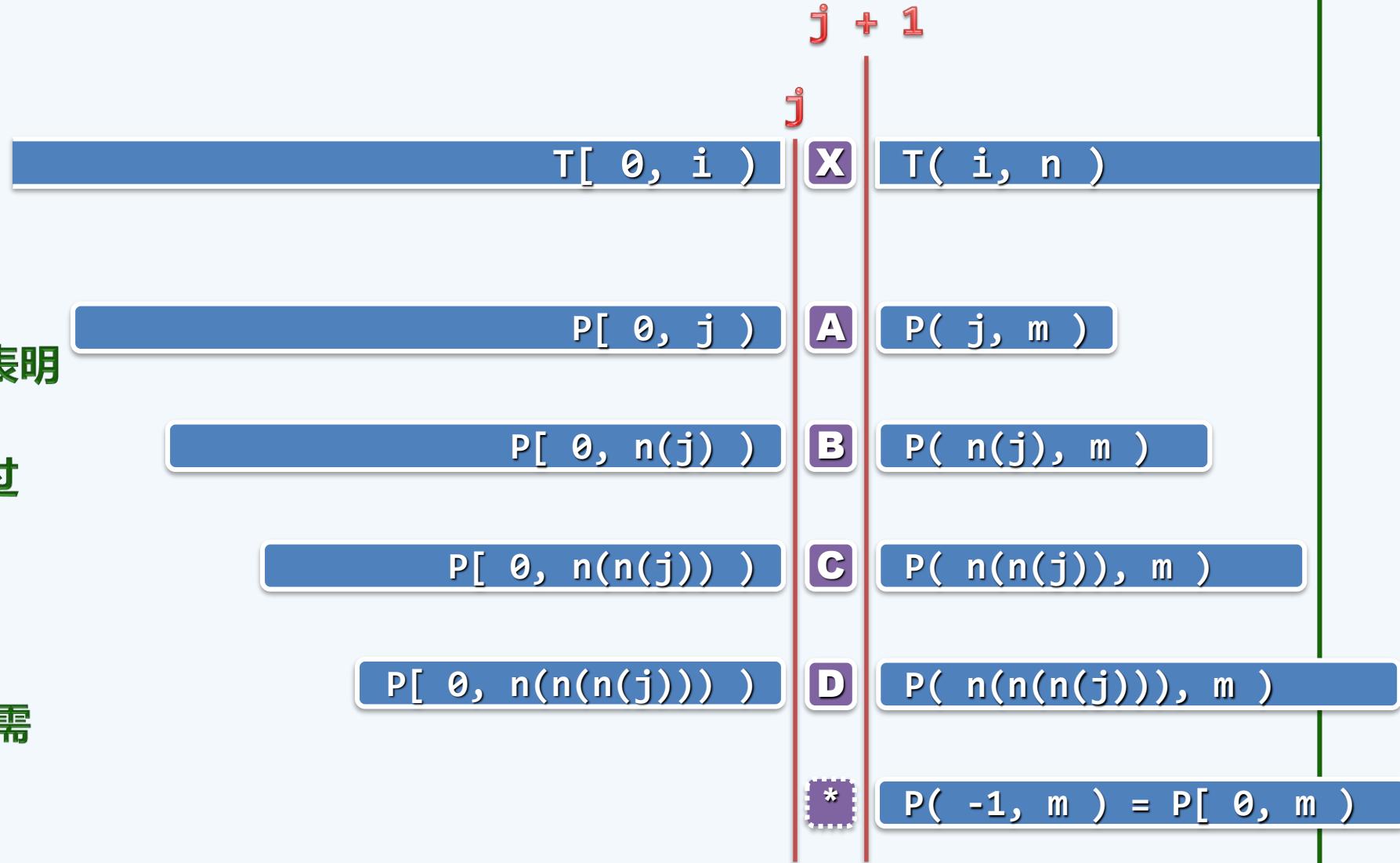
❖ 然而，更细致的分析将表明

即便是最坏情况，也不过

$O(n)$ 时间

❖ 同理，建立next[]也只需

$O(m)$ 时间



$\Theta(n + m)!$

❖ 令 $k = 2*i - j$ 

//具体含义，详见习题[11-4]

while ($j < m \&& i < n$) // k 必随迭代而单调递增，故也是迭代步数的上界

if ($\theta > j \&& T[i] == P[j]$)

{ $i++$; $j++$; } // i 、 j 同时加1，故 k 恰好加1

else

$j = next[j];$ // i 不变， j 至少减1，故 k 至少加1

❖ k 的初值为 θ ；算法结束时，必有：

$$k = 2*i - j \leq 2(n - 1) - (-1) = 2n - 1$$

Next

- 并行编程
- 数据结构讲义

Backup

11. 串

(c6) KMP算法：再改进

邓俊辉

deng@tsinghua.edu.cn

前车之覆，后车之鉴

反例

❖ T =

0	0	0	1	0	0	0	0	1
---	---	---	---	---	---	---	---	---

P =

0	0	0	0	1
---	---	---	---	---

❖ T[3]:

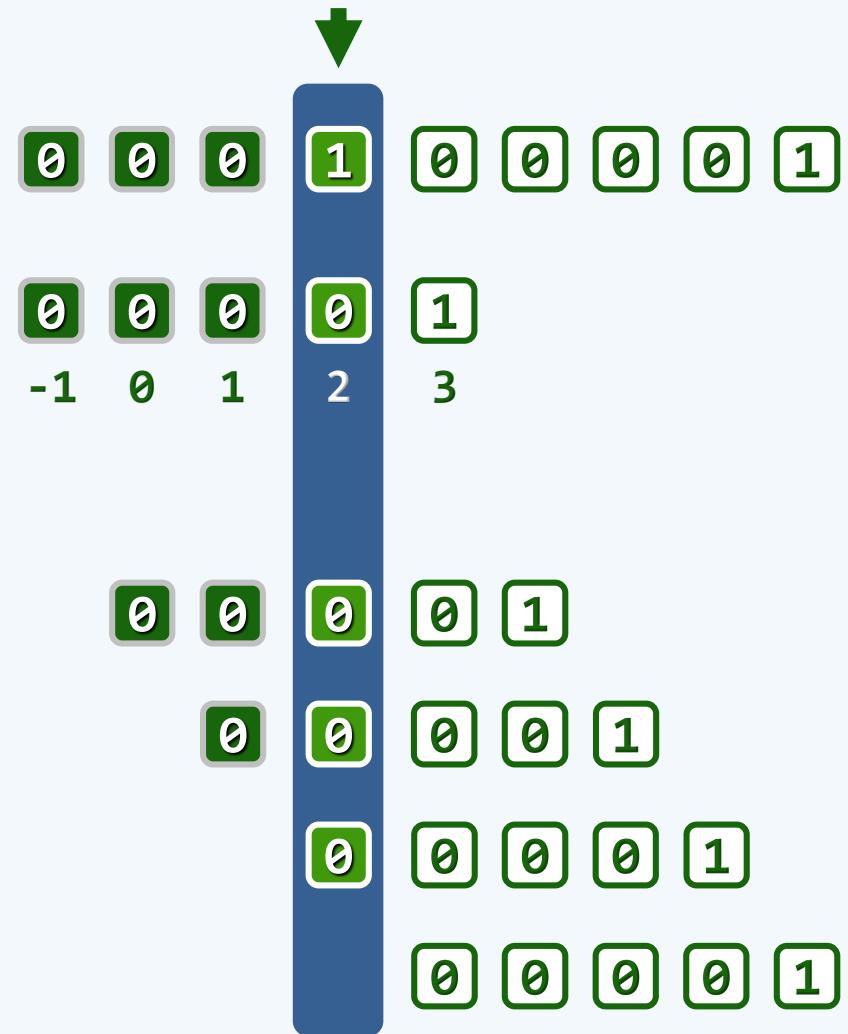
与 P[3] 比对, 失败

与 P[2] = P[next[3]] 继续比对, 失败

与 P[1] = P[next[2]] 继续比对, 失败

与 P[0] = P[next[1]] 继续比对, 失败

最终, 才前进到 T[4]



❖ 无需T串，即可在事先确定：

$P[3] =$

$P[2] =$

$P[1] =$

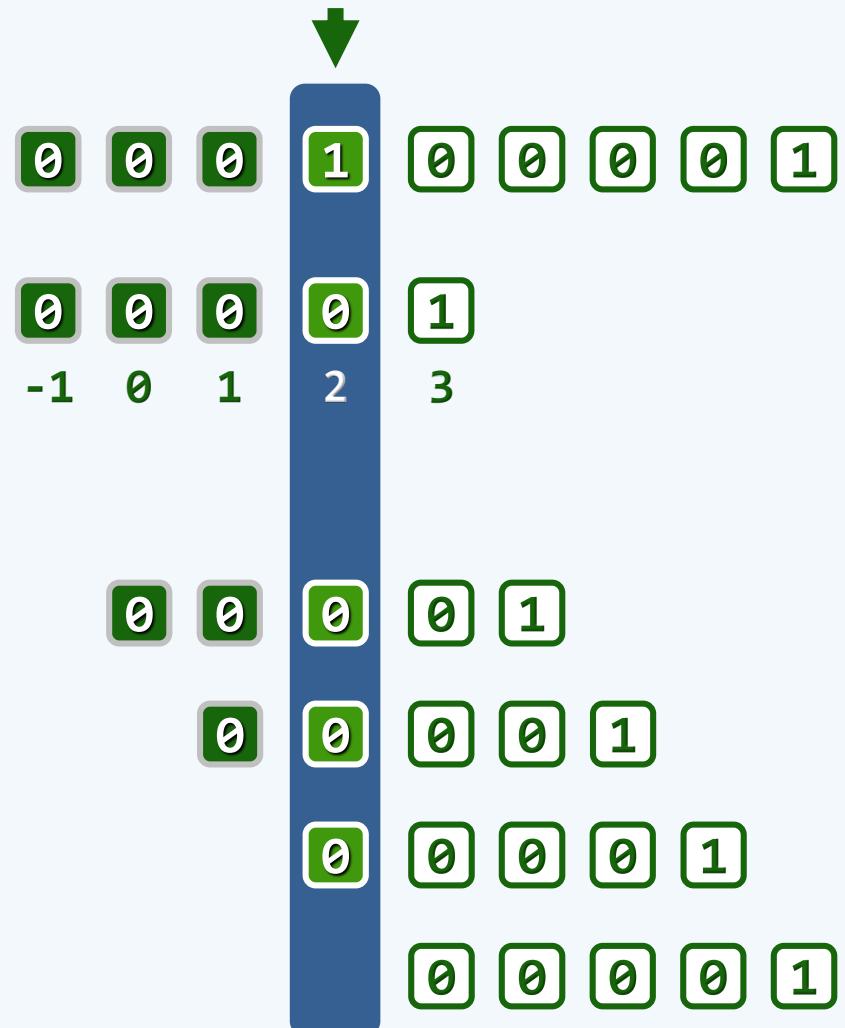
$P[0] = \boxed{0}$

既然如此...

❖ 在发现 $T[3] \neq P[3]$ 之后

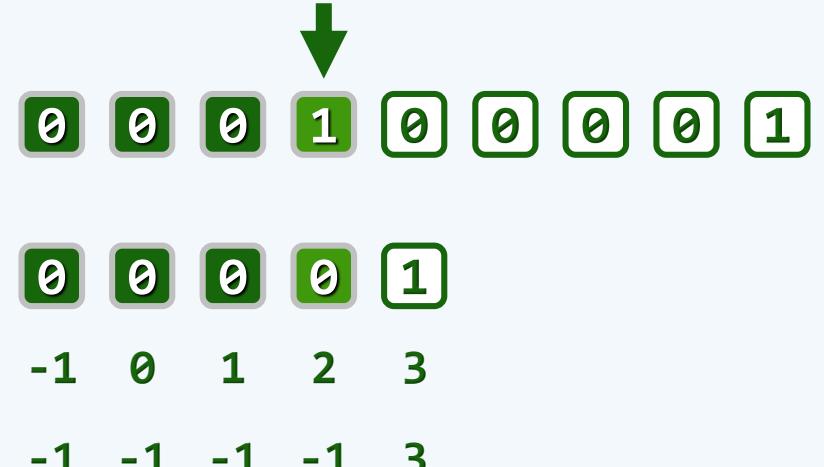
为何还要一错再错？

❖ 事实上，后三次比对本来都是可以避免的！

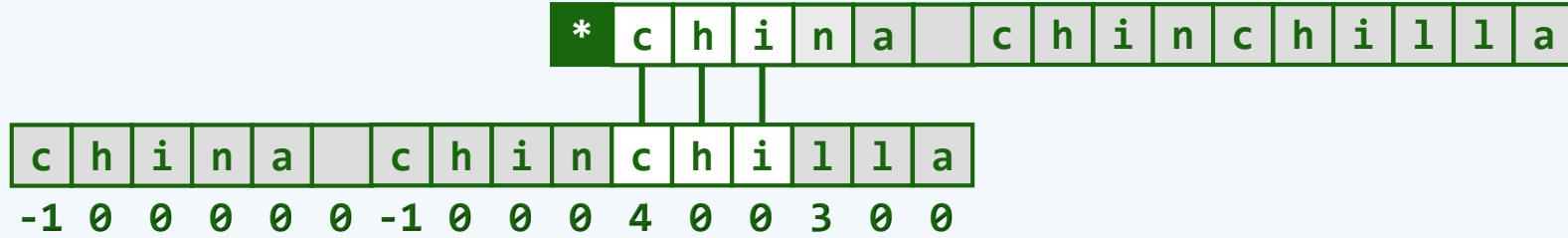
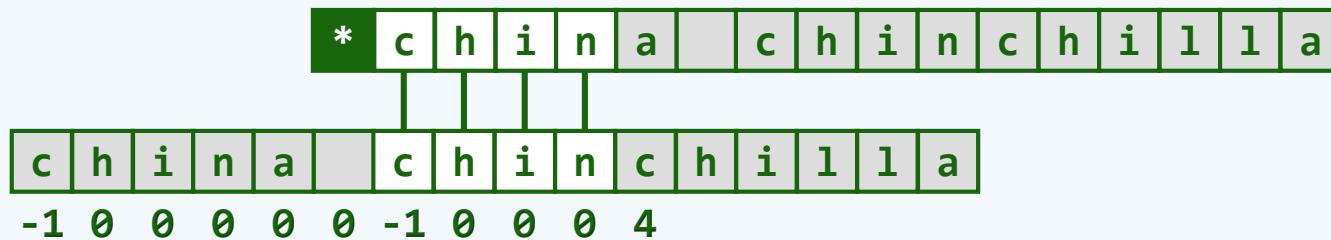
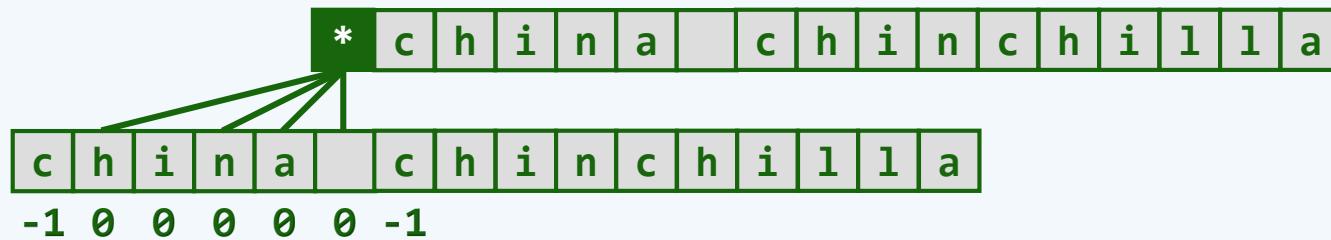


改进

```
❖ int * buildNext( char * P ) {  
    size_t m = strlen(P), j = 0; //“主”串指针  
    int * N = new int[m]; //next表  
    int t = N[0] = -1; //模式串指针  
    while ( j < m - 1 )  
        if ( 0 > t || P[j] == P[t] ) { //匹配  
            j++; t++; N[j] = P[j] != P[t] ? t : N[t];  
        } else //失配  
            t = N[t];  
    return N;  
}
```



实例



❖ 充分利用以往的比对所提供的信息

模式串快速右移，文本串无需回退

❖ 经验：以往成功的比对 —— $T[i - j, i]$ 是什么

教训：以往失败的比对 —— $T[i]$ 不是什么

❖ 特别适用于顺序存储介质

❖ 单次匹配概率越大 ($|\Sigma|$ 越小) 的场合，优势越明显 //比如二进制串
否则，与蛮力算法的性能相差无几...