

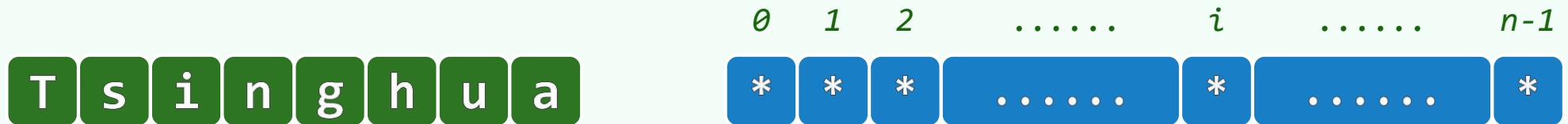
13-A

邓俊辉  
[deng@tsinghua.edu.cn](mailto:deng@tsinghua.edu.cn)

串

ADT

# 术语



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

[ $\theta$ ,  $i$ )

[ $i$ ,  $i+k$ )

[ $i+k$ ,  $n$ )

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

[ $\theta$ ,  $k$ )

[ $k$ ,  $n$ )

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

[ $\theta$ ,  $n-k$ )

[ $n-k$ ,  $n$ )

$$S.substr(i, k) = S.prefix(i + k).suffix(k) = S.suffix(n - i).prefix(k)$$

**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)**

[ 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**

串

## 模式匹配：问题 & 蛮力算法

# 13.-B

一切事情，一旦失败，就显得很愚蠢

长的是磨难，短的是人生

邓俊辉

deng@tsinghua.edu.cn

## 循模式访问：问题特点 + 测试方法

*Text :* a man with money is no **match** against a man on a mission

*Pattern :* **match**

random  $T$  + random  $P$  ?

random  $T$  + substring  $P$  !

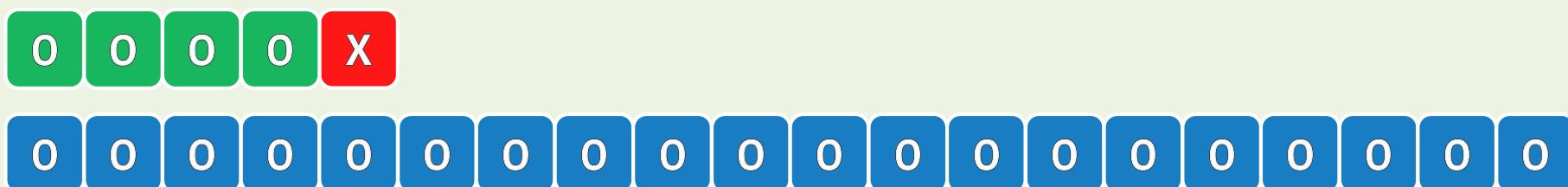
$$n = |T| \quad m = |P|$$

$$2 \ll m \ll n$$

$$s = |\Sigma|$$

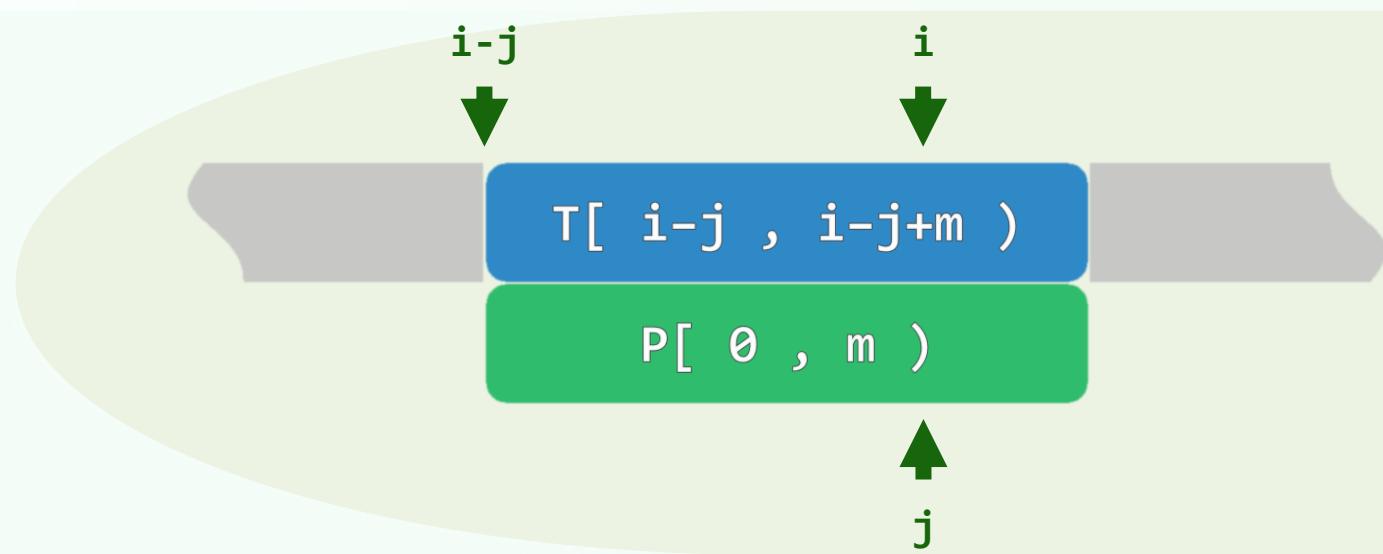
$$Pr_{match} \approx n/s^m \rightarrow 0$$

# 蛮力策略



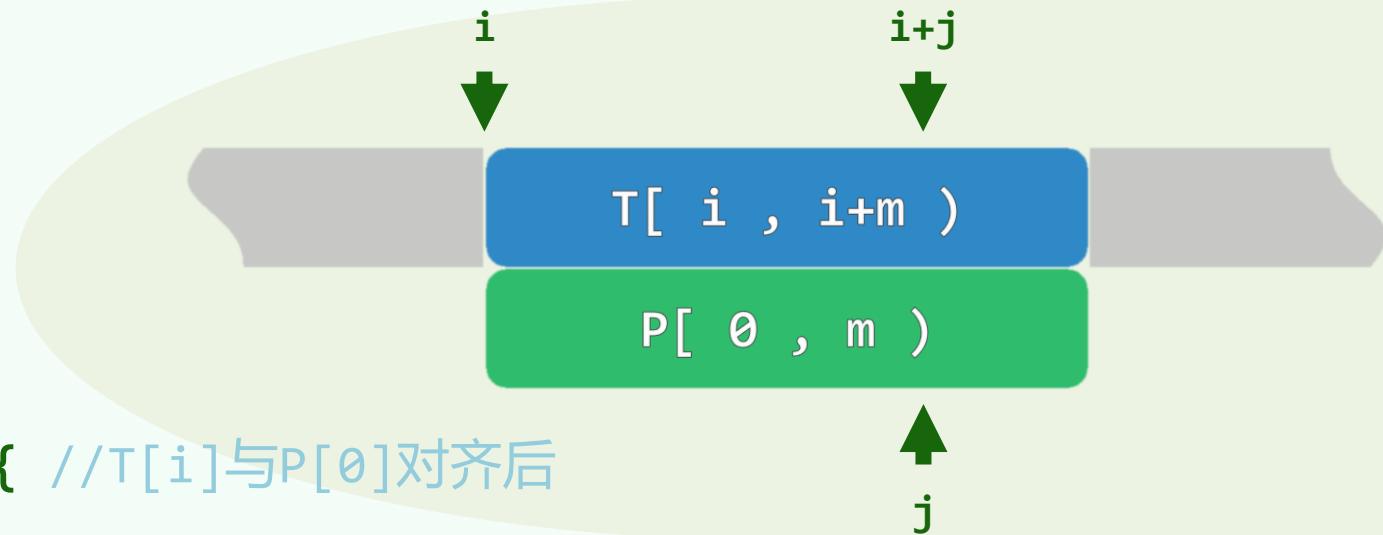
# 版本1

```
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 ) //自左向右逐次比对  
  
        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; //最终的对齐位置：藉此足以判断匹配结果  
}
```



## 版本2

```
int match( char * P, char * T ) {  
    size_t n = strlen(T), i = 0;  
    size_t m = strlen(P), j;  
    for ( i = 0; i < n-m+1; i ++ ) { //T[i]与P[0]对齐后  
        for ( j = 0; j < m; j ++ ) //逐次比对  
            if ( T[i+j] != P[j] ) break; //失配, 转下一对齐位置  
        if ( m <= j ) break; //完全匹配 (Python: 可以写得更简洁、优美)  
    }  
    return i; //最终的对齐位置: 藉此足以判断匹配结果
```



串

KMP算法：记忆法

13-C1

邓俊辉

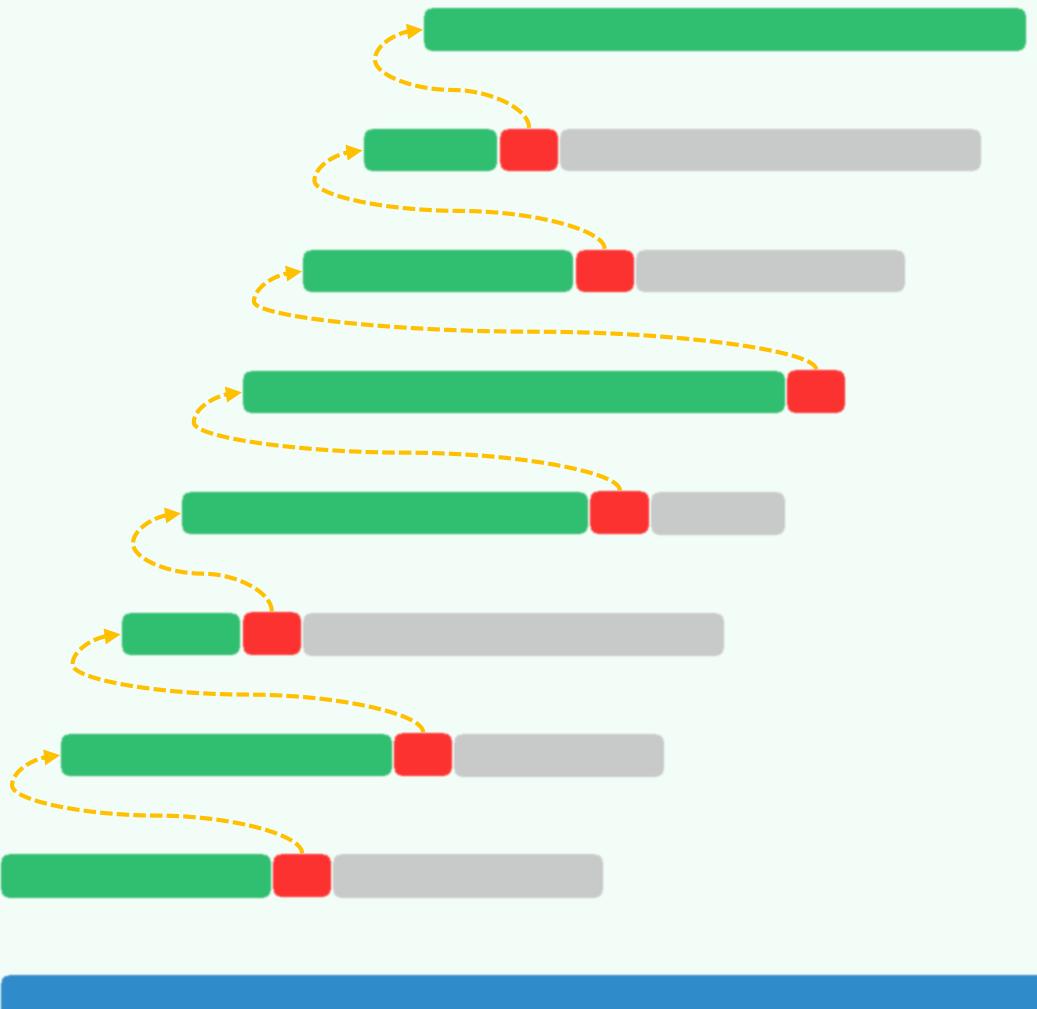
deng@tsinghua.edu.cn

知易者不占，善易者不卜

只有记忆才能建立起身份，即你个人的相同性

丧失了记忆又不自知，那才是人生最快乐的时光

## 低效 ~ 局部匹配



# 不变性：“超强大脑”玩“记忆翻牌消消看”

❖ 在任一时刻，都有

$$T[i-j, i] == P[0, j]$$

❖ 亦即，我们业已掌握 $T[i-j, i]$ 的所有信息

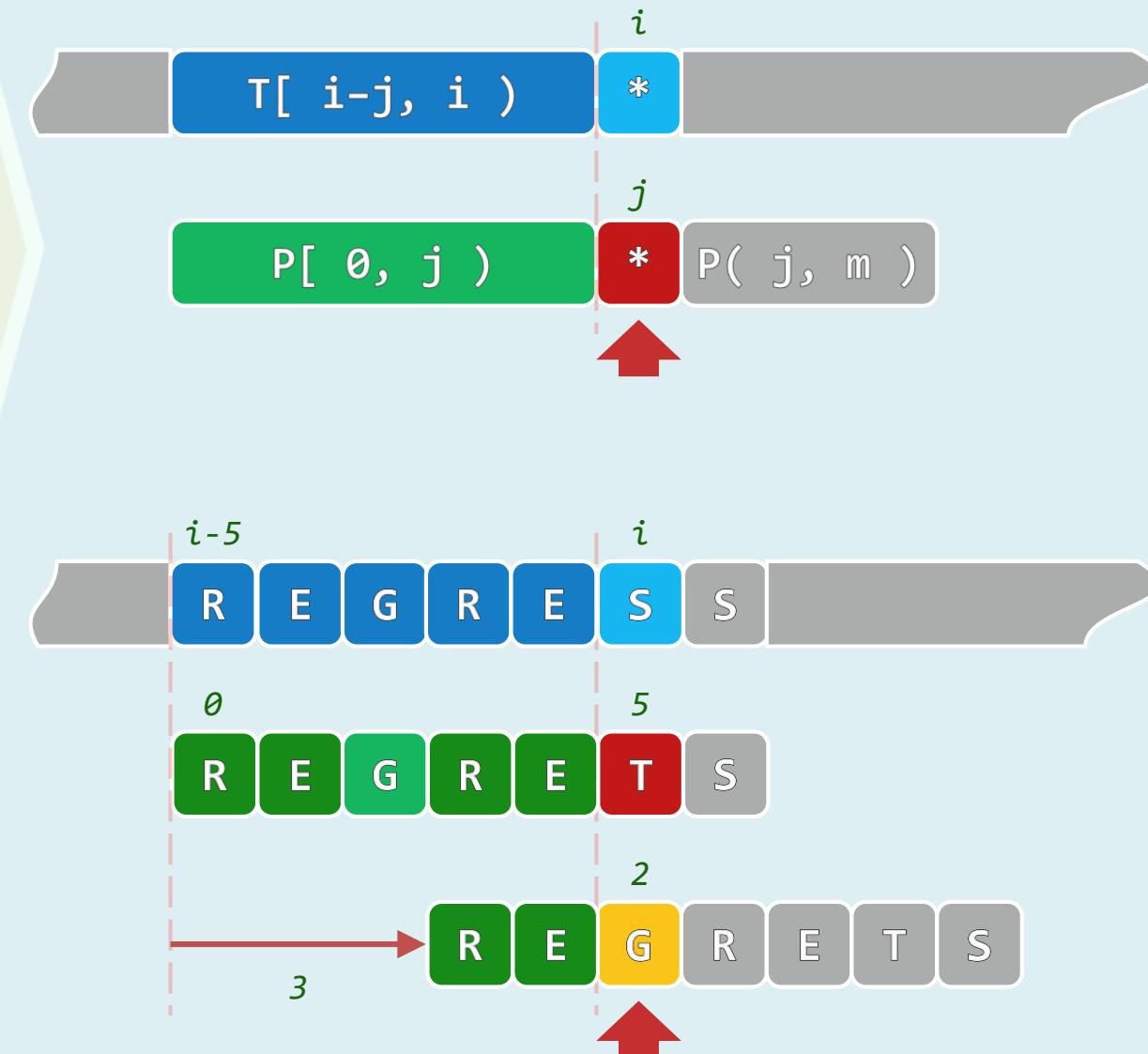
既如此...

❖ 一旦失败，我们就应已知

哪些位置值得/不必对齐

而且...在下一轮比对中...

❖  $T[i-j', i]$ 可径直接受，而不必再做比对



# 过目不忘？凡事预则立，不预则废！

❖ 如此，  $i$ 将永远不必回退！

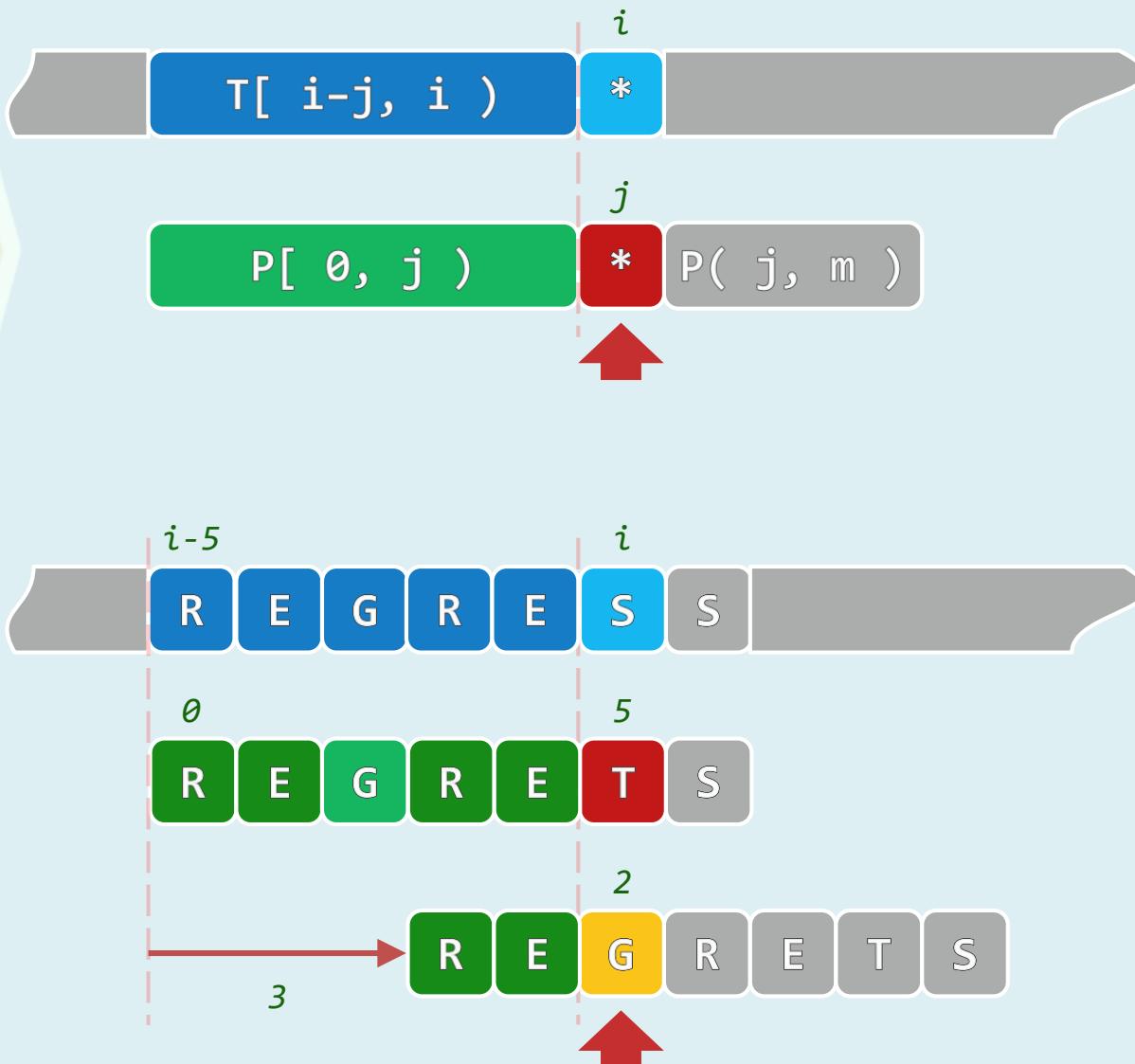
- 比对成功，则与 $j$ 同步前进一个字符
- 否则， $j$ 更新为某更小的 $t$ ，并继续比对

❖ 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？



串

KMP算法：查询表

13-C2

邓俊辉

deng@tsinghua.edu.cn

好记性不如烂笔头

你能看见多远的过去，就能看见多远的未来。

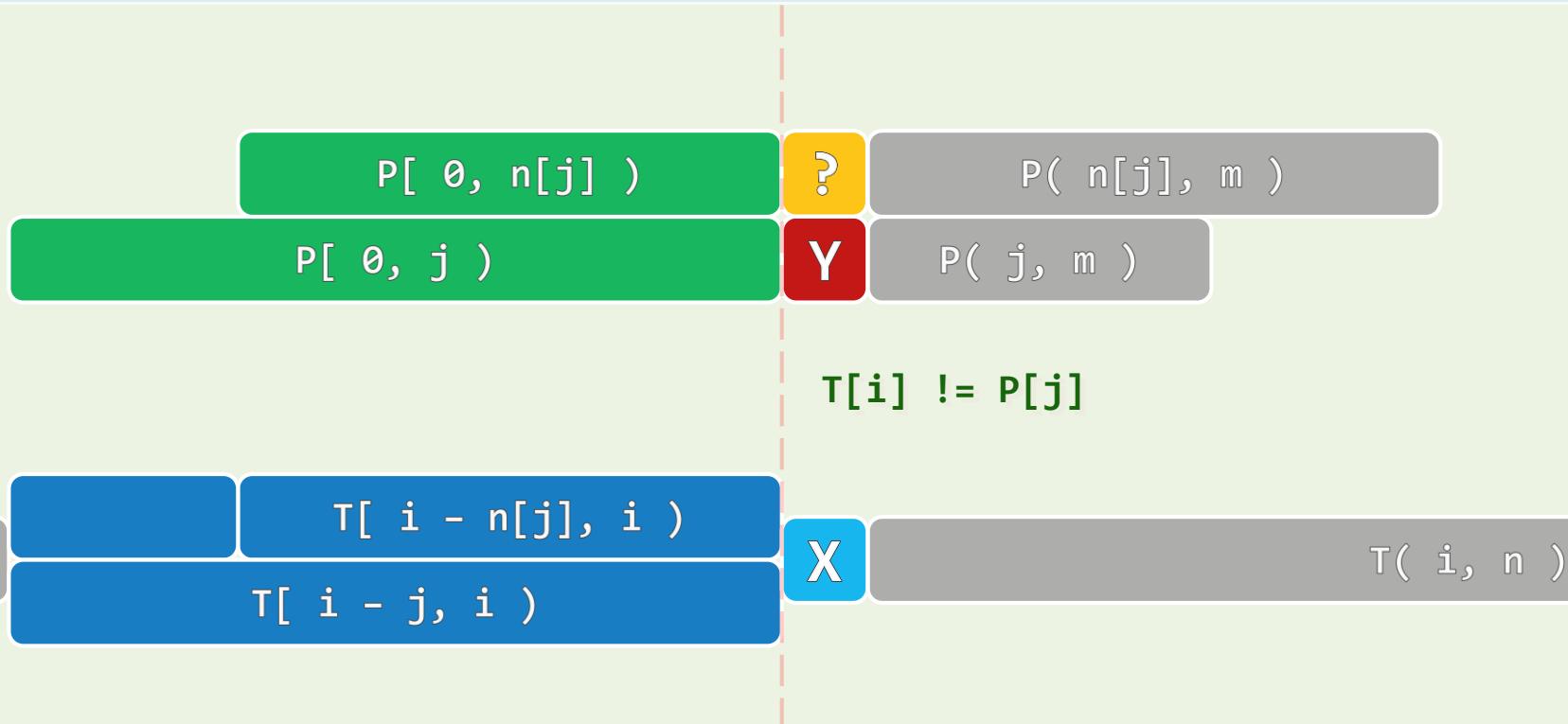
$t$ : 不仅可以事先确定, 而且仅根据  $P[0, j) = T[i-j, i)$  即可确定

❖ 视失败的位置  $j$ , 无非  $m$  种情况

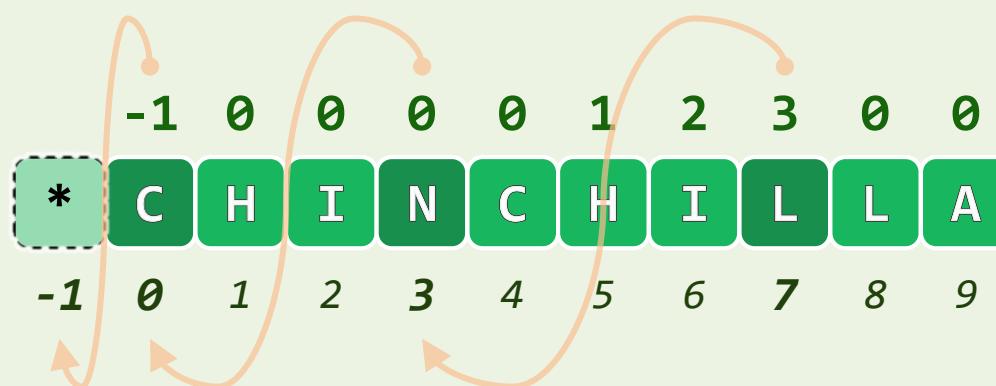
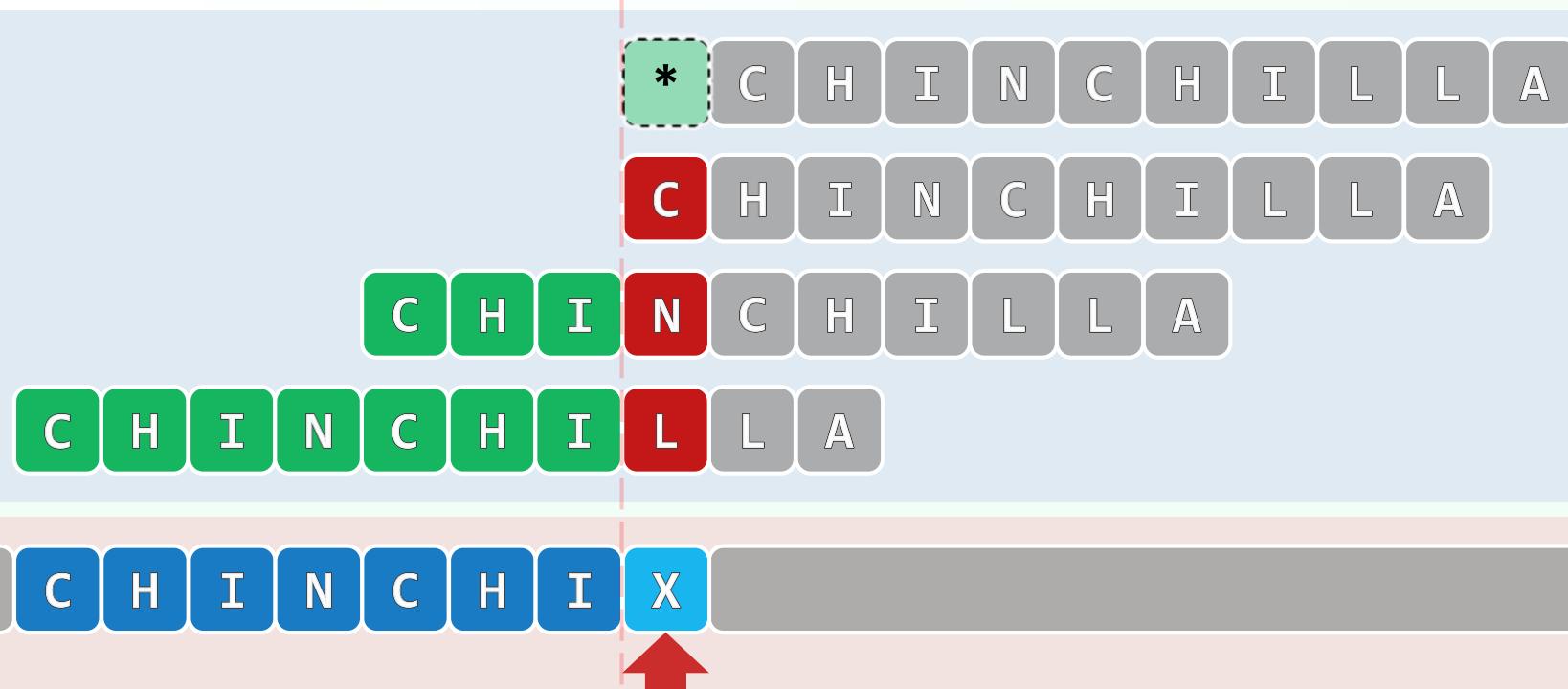
❖ 构造查询表  $\text{next}[0, m)$ , 做好预案

❖ 一旦在  $P[j]$  处失配, 只需

将  $j$  替换为  $\text{next}[j]$ , 继续与  $T[i]$  比对



# 实例



# KMP算法

```
int match( char * P, char * T ) {  
    int * next = buildNext(P);  
    int n = (int) strlen(T), i = 0;  
    int m = (int) strlen(P), j = 0;  
    while ( j < m && i < n )  
        if ( 0 > j || T[i] == P[j] ) {  
            i++; j++;  
        } else  
            j = next[j];  
    delete [] next;  
    return i - j;  
}
```



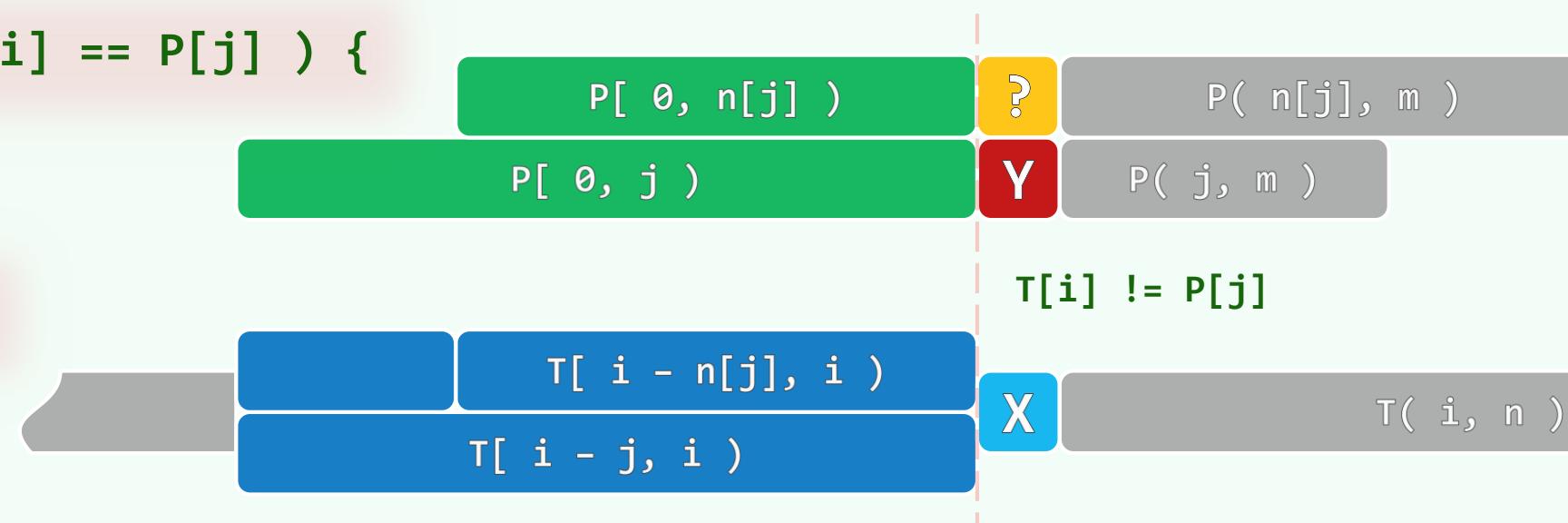
D. E. Knuth



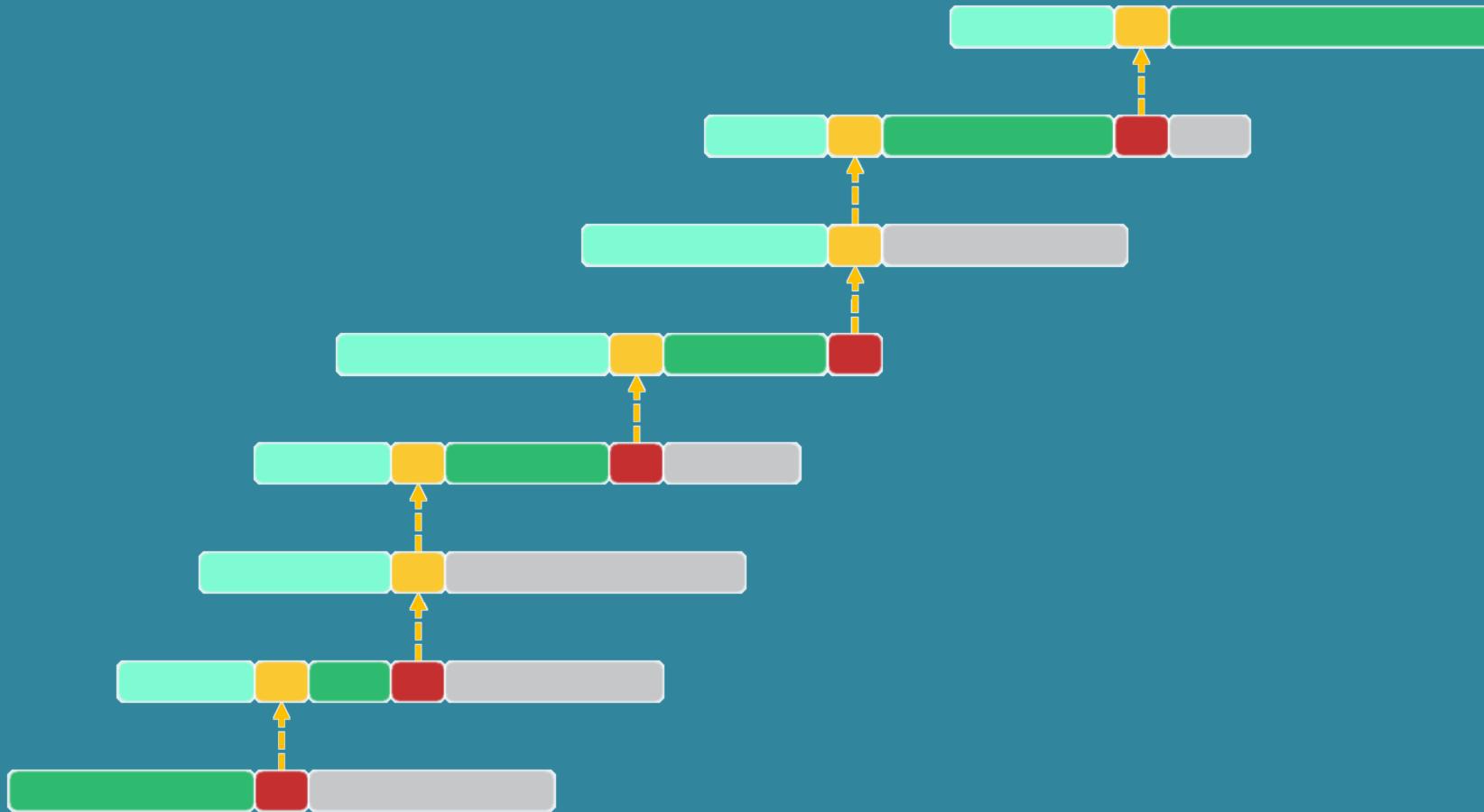
J. H. Morris



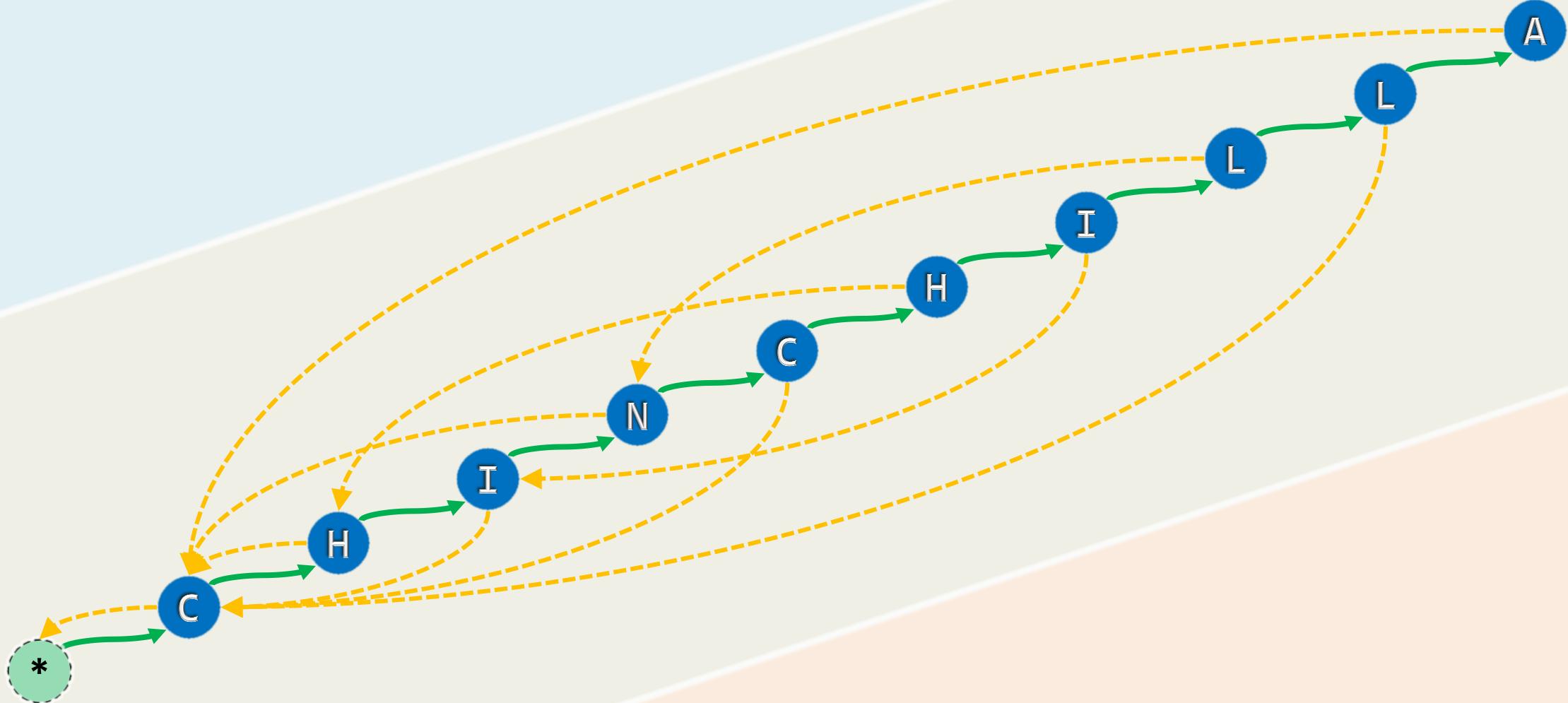
V. R. Pratt



# 快速右移 + 绝不回退



# 每一个P串，都是一台自动机



# 模式串 ~ 匹配算法

```
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 <sub>-</sub> :	++i;	// ↑	
s <sub>0</sub> :	(T[i] != 'C') ? goto s <sub>-</sub> : if (n <= ++i) return -1; // *	~ ↑	
s <sub>1</sub> :	(T[i] != 'H') ? goto s <sub>0</sub> : if (n <= ++i) return -1; // *C	~ *	
s <sub>2</sub> :	(T[i] != 'I') ? goto s <sub>0</sub> : if (n <= ++i) return -1; // *CH	~ *	
s <sub>3</sub> :	(T[i] != 'N') ? goto s <sub>0</sub> : if (n <= ++i) return -1; // *CHI	~ *	
s <sub>4</sub> :	(T[i] != 'C') ? goto s <sub>0</sub> : if (n <= ++i) return -1; // *CHIN	~ *	
s <sub>5</sub> :	(T[i] != 'H') ? goto s <sub>1</sub> : if (n <= ++i) return -1; // *CHINC	~ *C	
s <sub>6</sub> :	(T[i] != 'I') ? goto s <sub>2</sub> : if (n <= ++i) return -1; // *CHINCH	~ *CH	
s <sub>7</sub> :	(T[i] != 'L') ? goto s <sub>3</sub> : if (n <= ++i) return -1; // *CHINCHI	~ *CHI	
s <sub>8</sub> :	(T[i] != 'L') ? goto s <sub>0</sub> : if (n <= ++i) return -1; // *CHINCHIL	~ *	
s <sub>9</sub> :	(T[i] != 'A') ? goto s <sub>0</sub> : if (n <= ++i) return -1; // *CHINCHILL	~ *	
	return i - 10;	// *CHINCHILLA	

```
}
```

串

## KMP算法：理解next[ ]表

13 -

C3

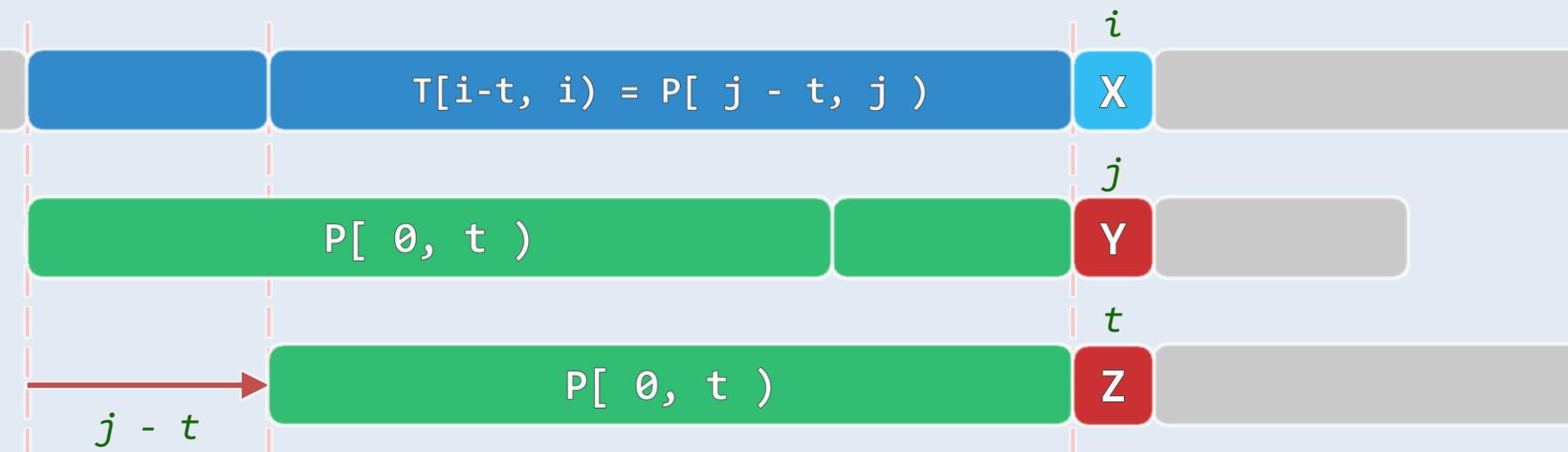
邓俊辉

deng@tsinghua.edu.cn

若想预见数学的未来，正确的方法是研究它的历史和现状。

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。

## 最长自匹配：快速右移 + 绝不回退

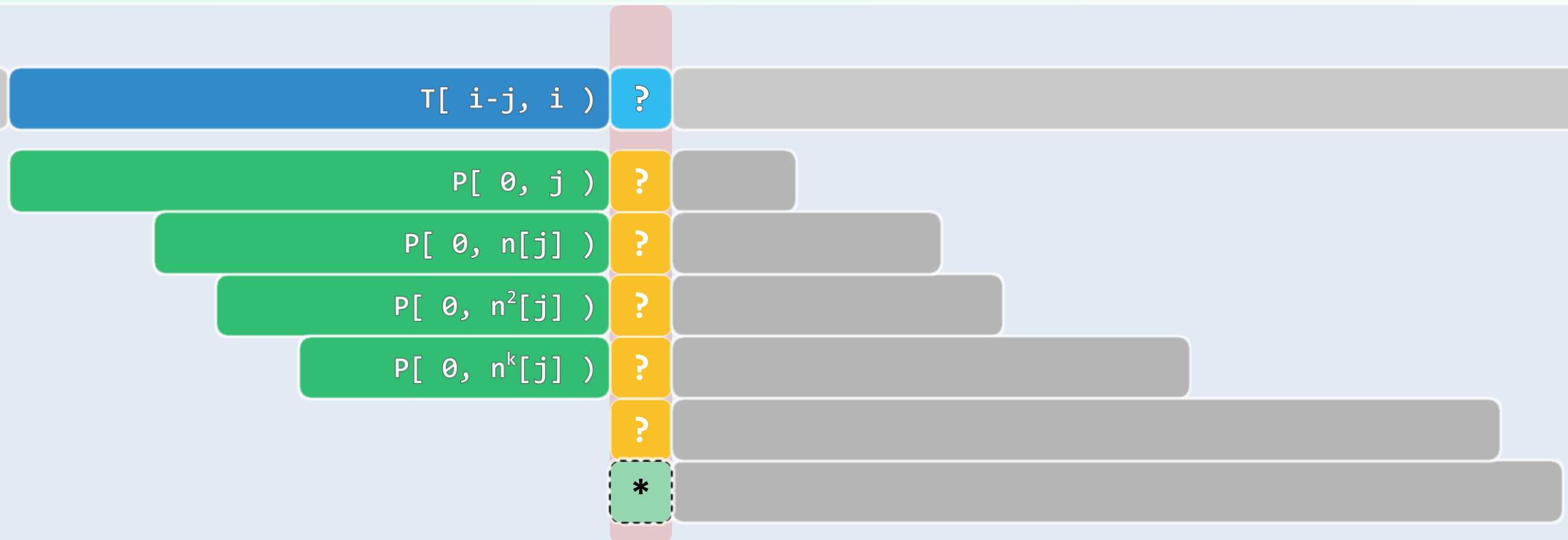


$\forall j \geq 1, \mathbf{N}(P, j) = \{ 0 \leq t < j \mid P[0, t) = P[j - t, j) \} \quad //\text{所有自匹配的长度}$

$0 \in \mathbf{N}(P, j) \neq \emptyset \quad //\text{因总包含0而非空, 故可以}$

$next[j] = \max\{ \mathbf{N}(P, j) \} \quad //\text{取最大长度: 位移最小, 不致回溯}$

## 自匹配：传递链



记:  $next^0[j] = j$ ,  $next^{k+1}[j] = next[next^k[j]]$  ( $k \geq 0$  and  $next^k[j] \geq 0$ )

同一 $T[i]$ 可能依次与 $P$ 中多个字符比对, 其秩是:  $j = next^0[j], next^1[j], next^2[j], \dots, 0, -1$

串

## KMP算法：构造next[ ]表

13-C4

一切都是暂时的，转瞬即逝  
而那逝去的将变为可爱

在这些胡思乱想里，我还怕我的记忆力不听我的使唤，怕记忆力出于疏忽而让我把  
同一件事写上两次。我讨厌在文章里再次认出自己，我炒冷饭向来是违心的。

邓俊辉

deng@tsinghua.edu.cn

# 减而治之：由 $\text{next}[0]$ 、 $\text{next}[1]$ 、...、 $\text{next}[j]$ ，如何得到 $\text{next}[j+1]$ ？

-1 0 0 1 2 3 1 0  
M A M A M M I A

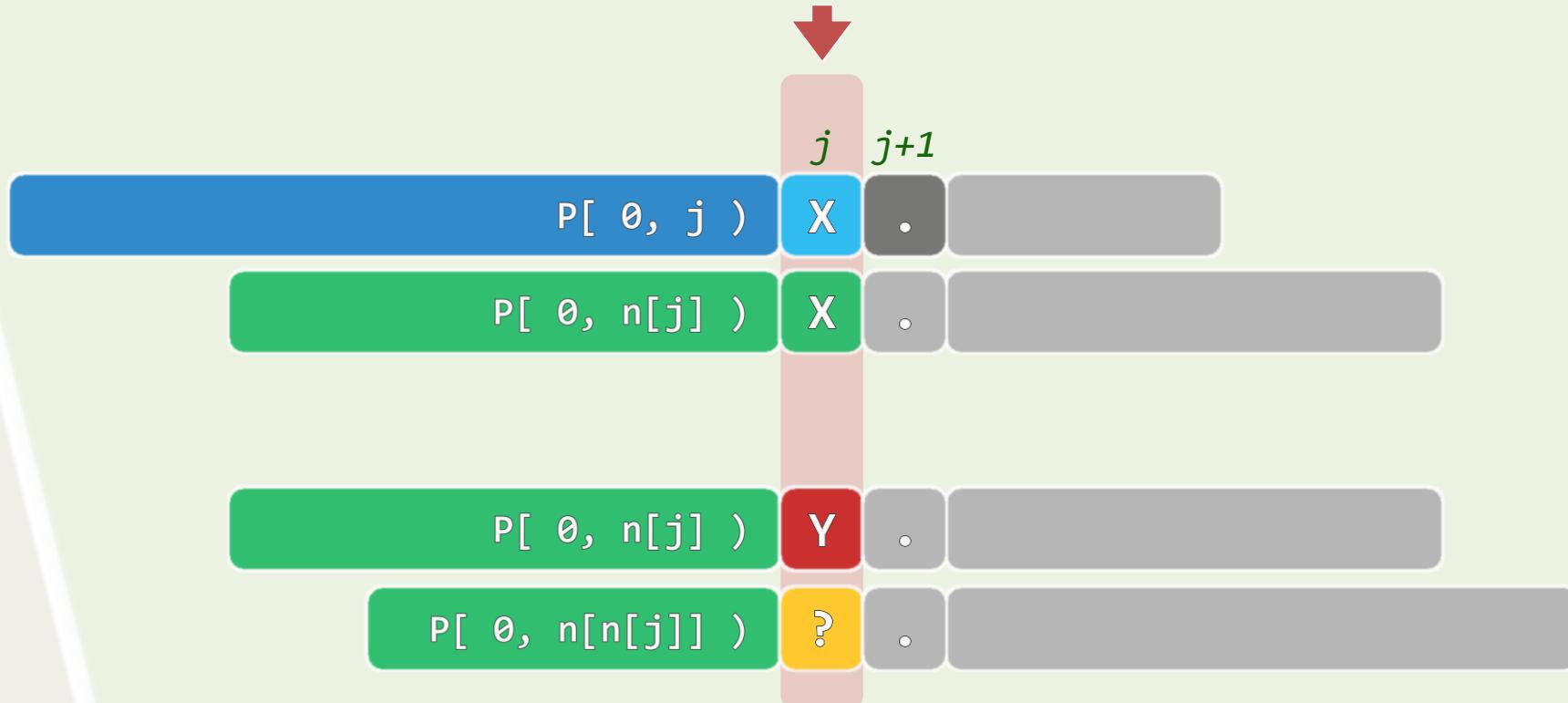
-1 0 0 1  
M A M M A M M I A  
M M A M M M I A

-1 0 0 1 2  
M A M M A M M I A  
M M A M M M I A

-1 0 0 1 2 3  
M A M M A M M I A  
M M A M M M I A

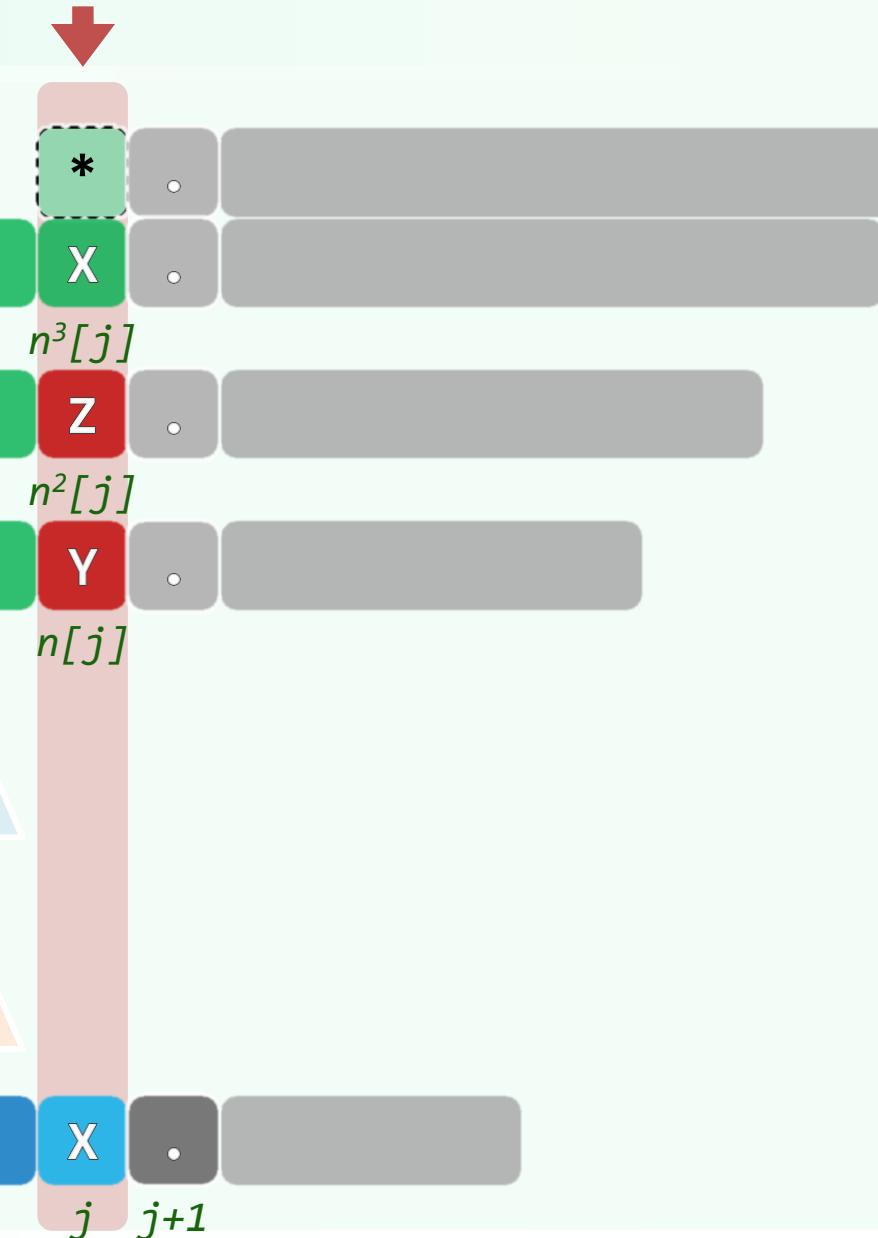
-1 0 0 1 2 3 1  
M A M M A M M I A  
M M A M M M I A  
M A M A M M M I A  
M M A M M M I A

P在 $j+1$ 处的自匹配，只比在 $j$ 处的自匹配多出一个字符...



# 算法实现

```
int* buildNext( char* P ) {  
    size_t m = strlen(P), j = 0;  
    int* next = new int[m];  
    int t = next[0] = -1;  
    while ( j < m - 1 )  
        if ( 0 > t || P[t] == P[j] ) { //匹配  
            ++t; ++j; next[j] = t; //则递增赋值  
        } else //否则  
            t = next[t]; //继续尝试下一值得尝试的位置  
    return next;  
}
```



# 13 - C5

串

KMP算法：分摊分析

邓俊辉

deng@tsinghua.edu.cn

失之东隅，收之桑榆

# $\Omega(n*m)$ ?

❖ KMP 的确可以节省多次比对

❖ 然而，就渐近意义而言  
有实质的节省吗？



❖ 每个  $T[i]$  (红) , 都可能参与  $\Omega(m)$  次比对 (黄)

❖ 倘若共有  $\Omega(n)$  个这样的  $T[i]$  ...

❖ 然而更细致的分析将表明

即便在最坏情况下，累计也不过  $2n = \mathcal{O}(n)$  次

$\Theta(n + m)!$

❖ 令:  $k = 2*i - j$  //欠精准, 但还算够用的计步器

`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 ( 0 > j || T[i] == P[j] )`

`{ i++; j++; }` // $k$ 恰好加1

`else`

`j = next[j];` // $k$ 至少加1

❖ 初始:  $k = 0$

算法结束时:  $k = 2*i - j \leq 2(n - 1) - (-1) = 2n - 1$

串

KMP算法：再改进

13-C6

邓俊辉

deng@tsinghua.edu.cn

有颜回者好学，不迁怒，不贰过

母亲心疼地看了我好久，然后叹口气：“好吧，你这个倔强的孩子，那条路很难走，一路小心！”

# 反例

❖ 在  $T[3]$  处

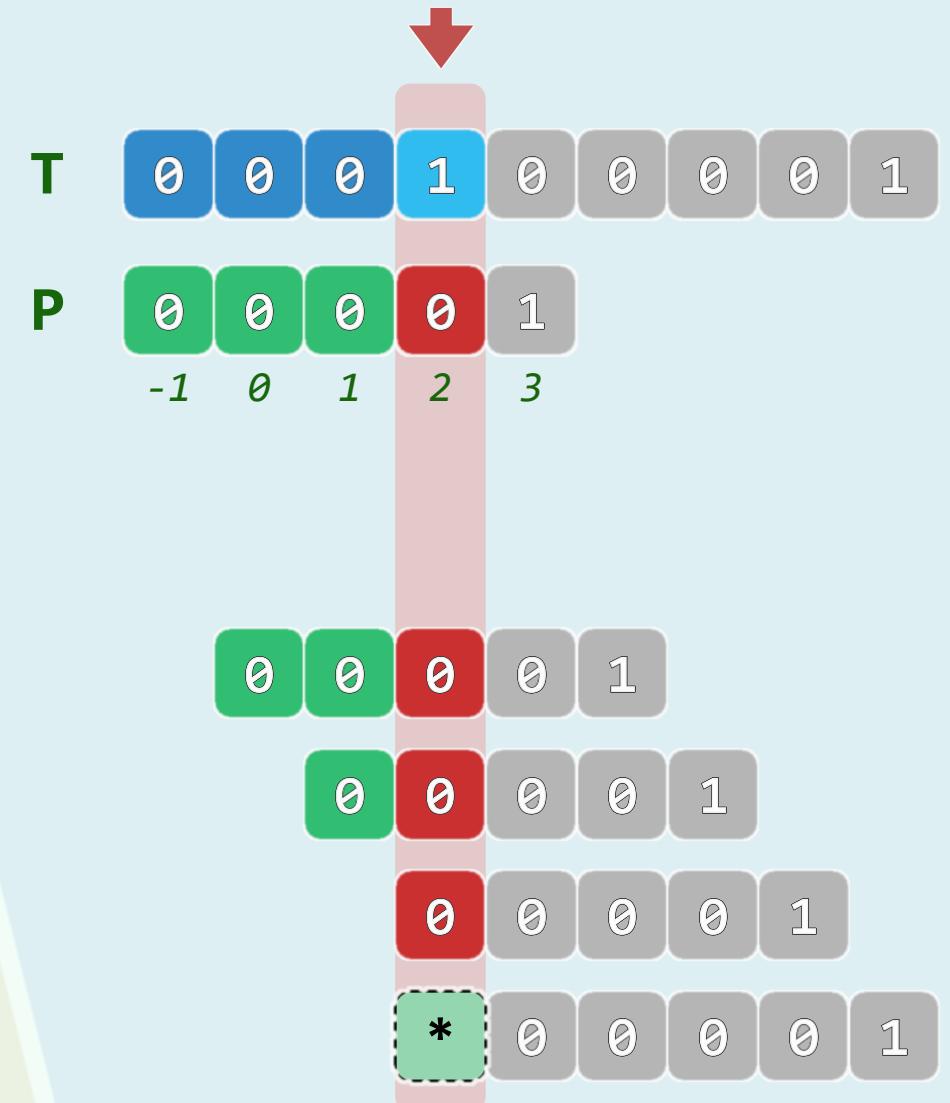
又：与  $P[3]$  比对，失败

双：与  $P[2] = P[\text{next}[3]]$  比对，失败

聂：与  $P[1] = P[\text{next}[2]]$  比对，失败

聂：与  $P[0] = P[\text{next}[1]]$  比对，失败

最终，才前进到  $T[4]$



# 根源

❖ 无需T串，即可在事先确定：

$P[3] =$

$P[2] =$

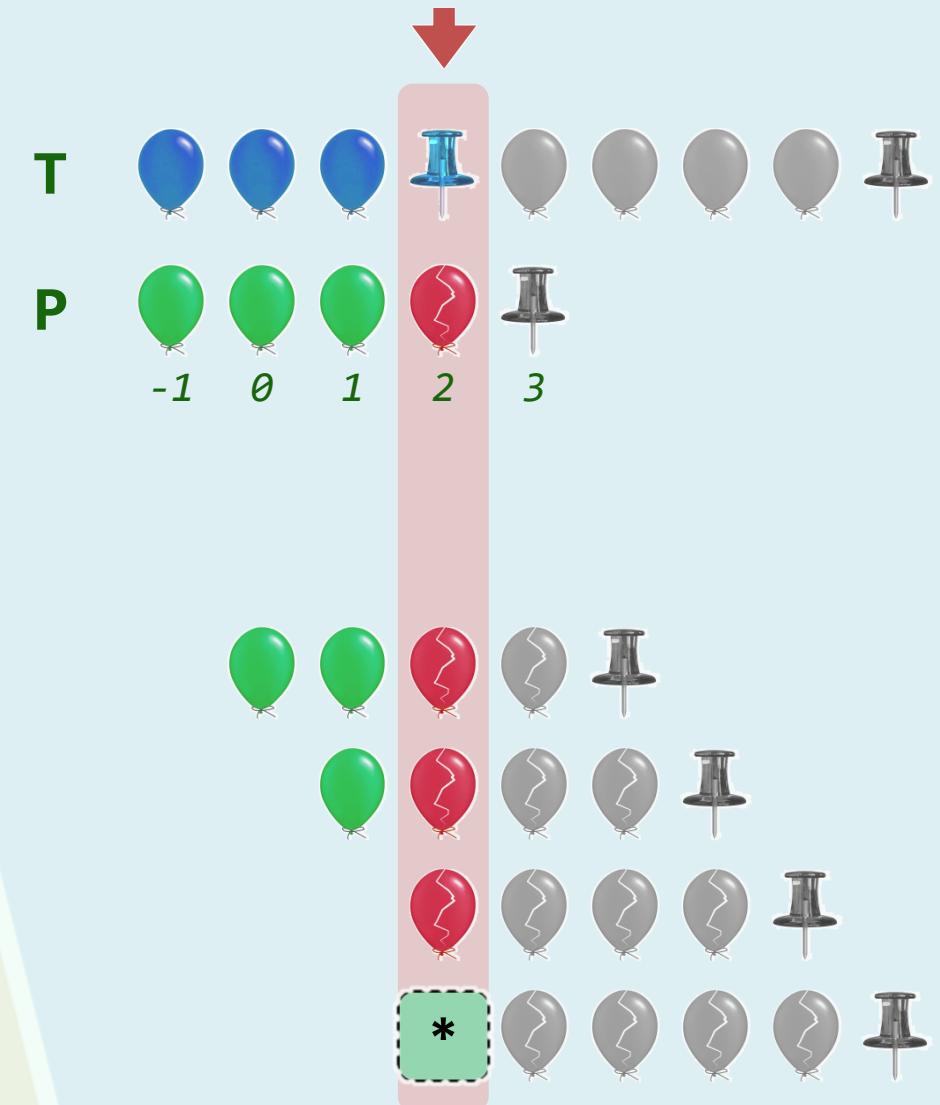
$P[1] =$

$P[0] = 0$

既然如此...

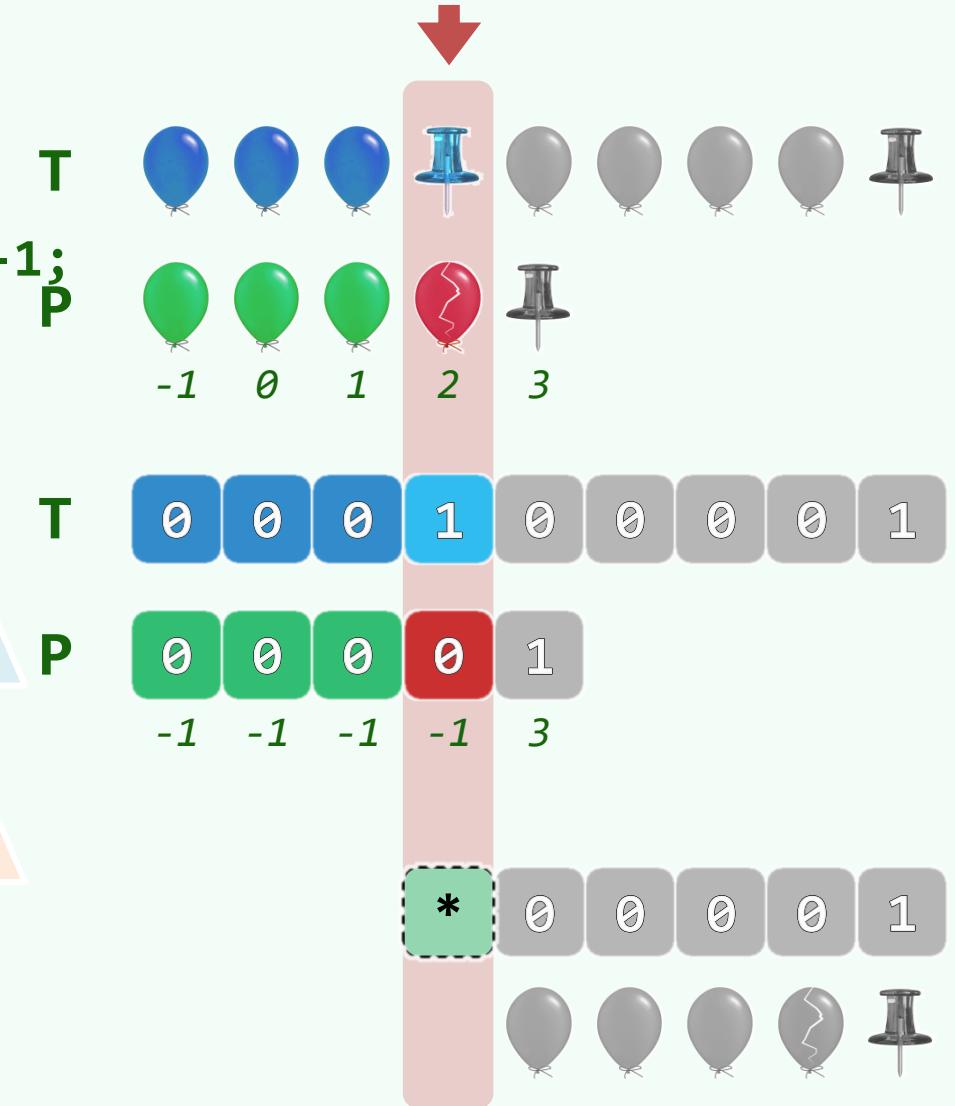
❖ 在发现  $T[3] \neq P[3]$  之后，为何还要一错再错？

❖ 事实上，后三次比对本来都是可以避免的！



# 改进

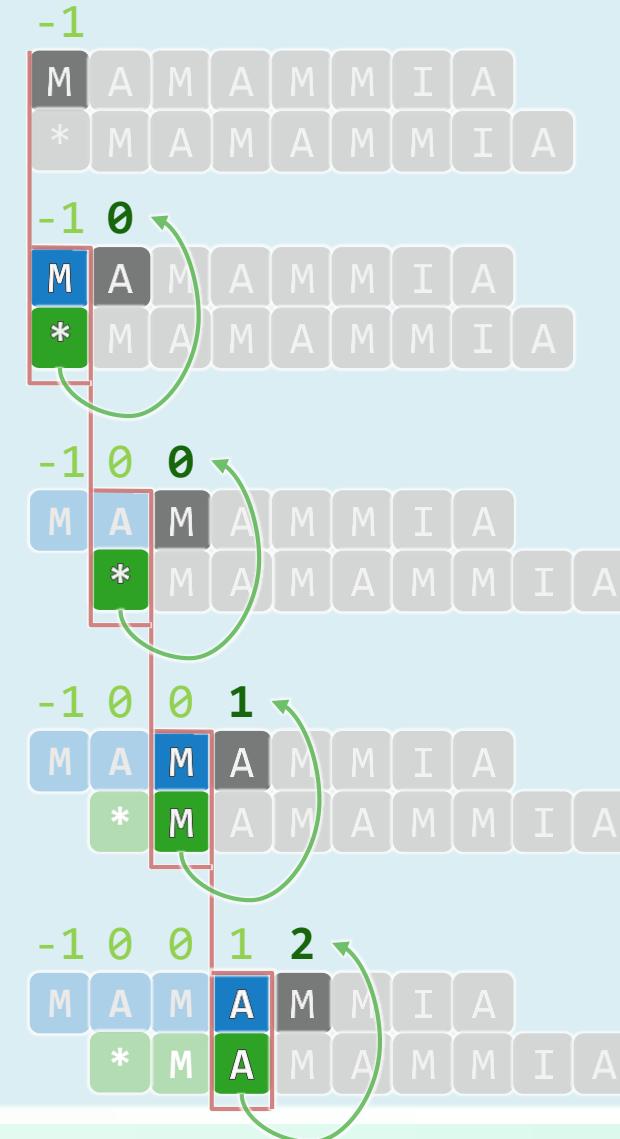
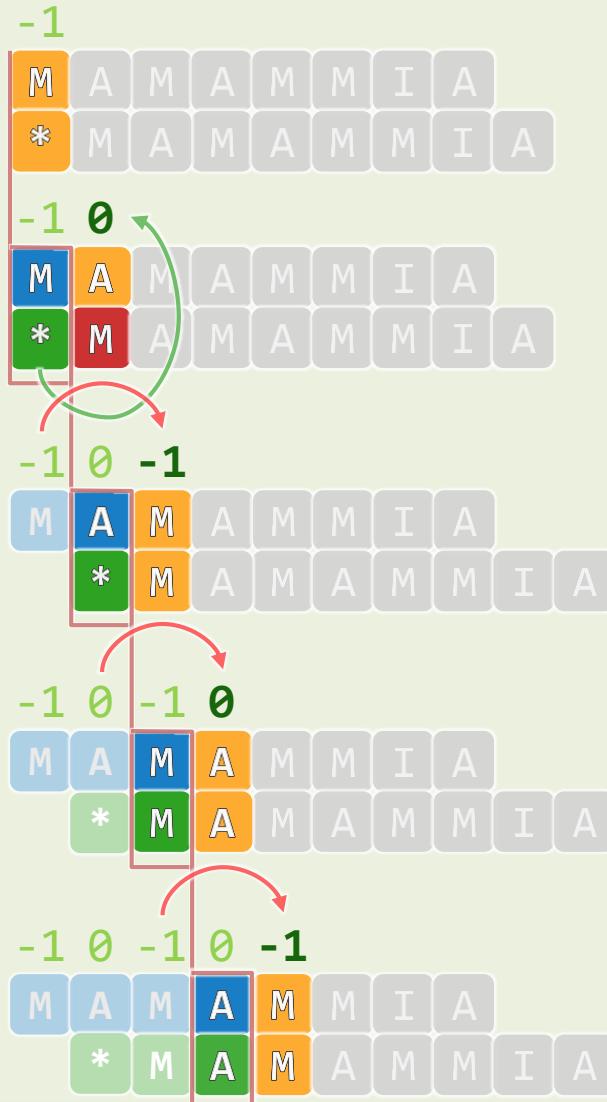
```
int* buildNext( char* P ) {  
    size_t m = strlen(P), j = 0;  
    int* next = new int[m]; int t = next[0] = -1;  
    while ( j < m - 1 )  
        if ( 0 > t || P[t] == P[j] ) {  
            if ( P[++t] != P[++j] )  
                next[j] = t;  
            else //P[next[t]] != P[t] == P[j]  
                next[j] = next[t];  
        } else  
            t = next[t];  
  
    return next;  
}
```



# 对比

-1 0 -1 0 -1 3 1 0  
 M A M A M M I A

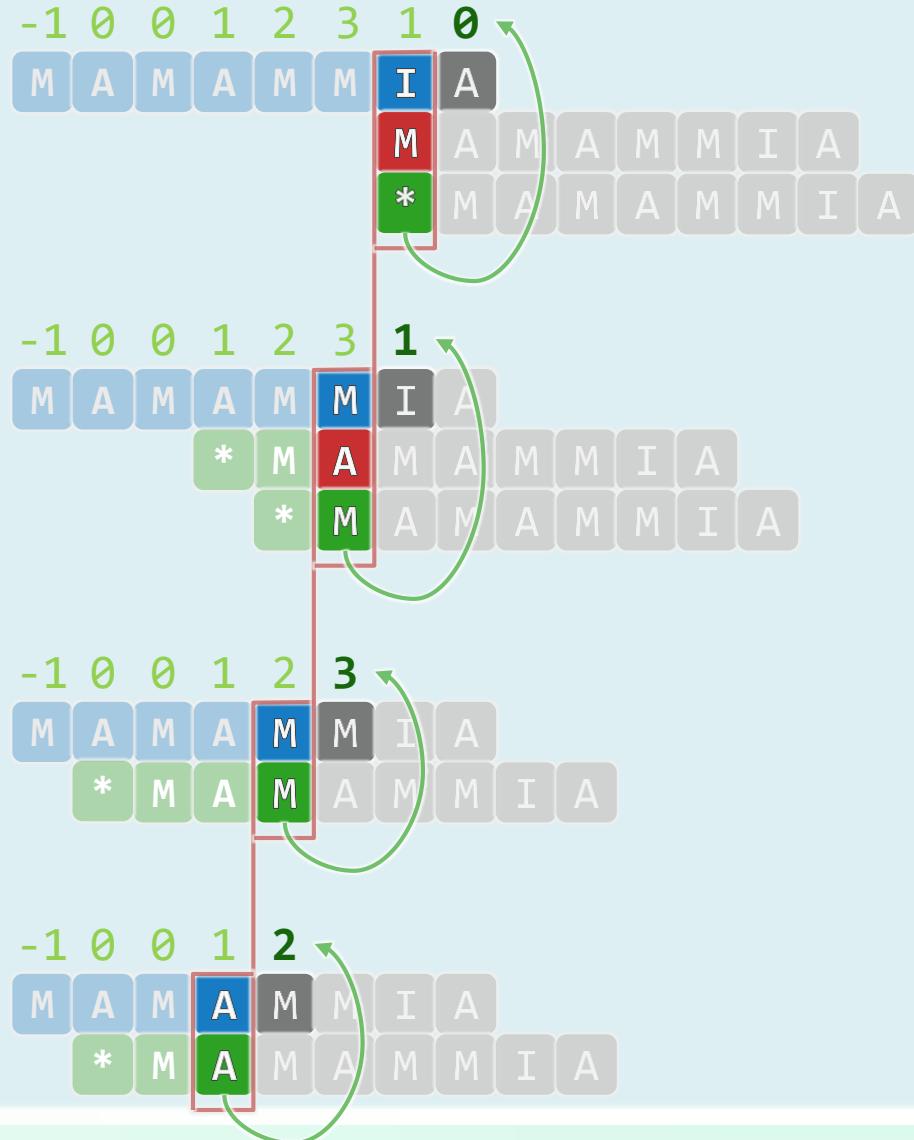
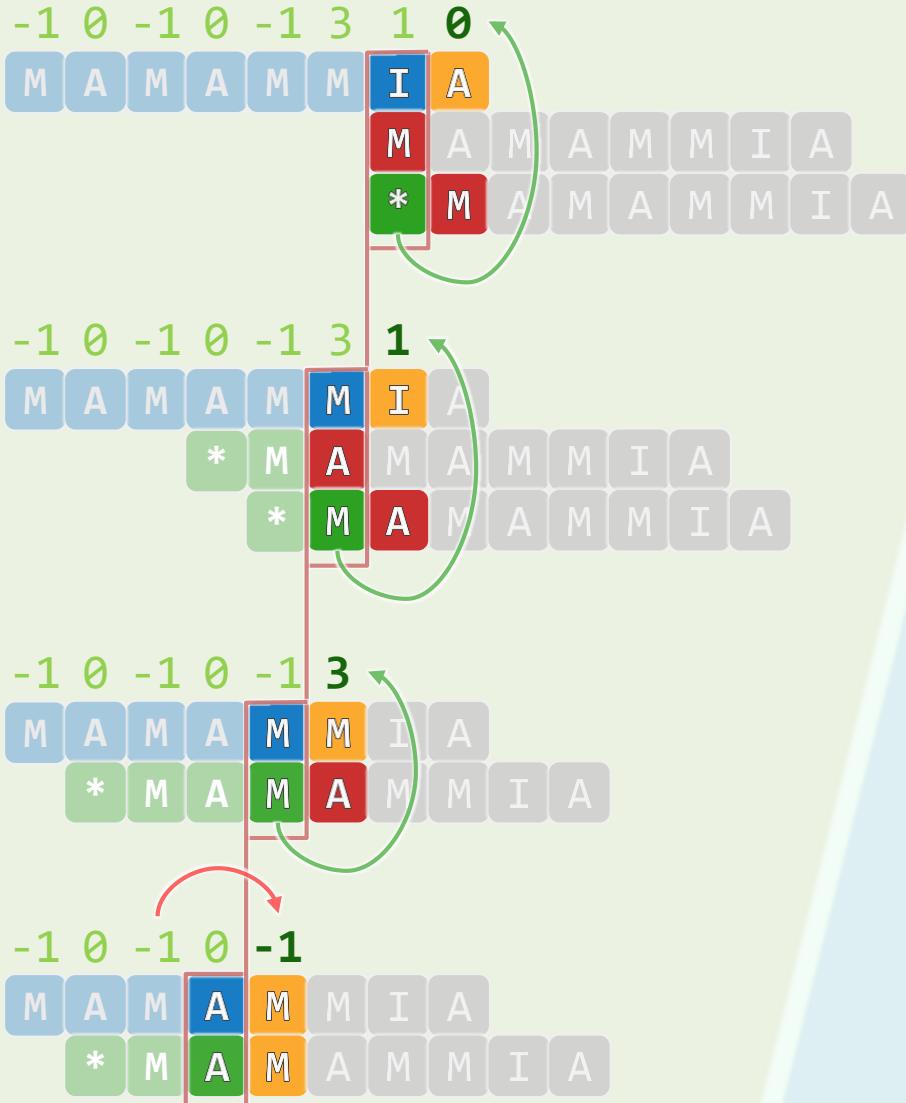
-1 0 0 1 2 3 1 0  
 M A M A M M I A



# 对比

-1 0 -1 0 -1 3 1 0  
 M A M A M M I A

-1 0 0 1 2 3 1 0  
 M A M A M M I A

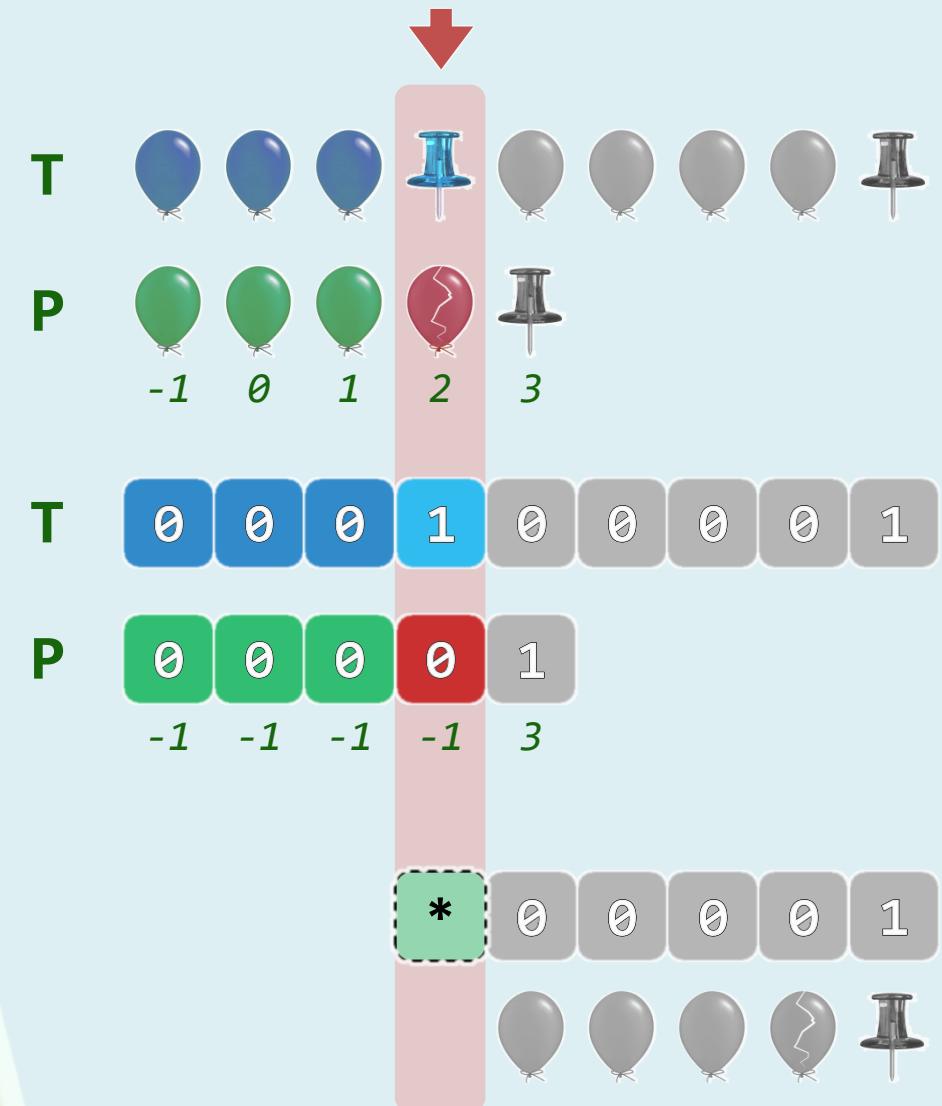


# 小结

- ◆ 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（字符集越小），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...



串

BM算法：BC策略：以终为始

13-D1

It's best to have failure happen early in life. It wakes up the phoenix bird in you so you rise from the ashes.

我劝你还是努力去利用那过去之石，来磨砺这现在之刀吧。因为生活本身就有许多失败，你多得着一次，不就是减少了一次吗？！

邓俊辉

deng@tsinghua.edu.cn

# 善待教训，尽早试错



# 以终为始

❖ 既如此，每一趟比对都更应该

- 从未字符开始
- 自后向前，自右向左

❖  $4 + 4 < 12$



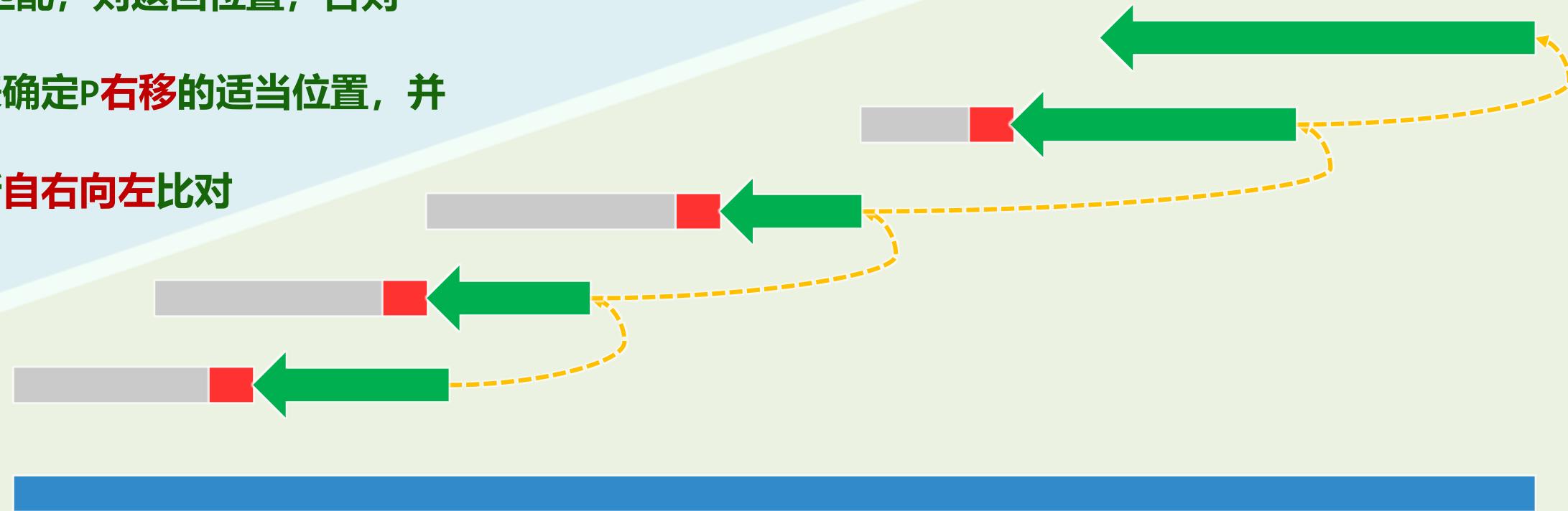
## [Boyer + Moore, 1977] A Fast String Searching Algorithm

❖ 预处理：根据模式串P，预先构造gs[]表和bc[]表

❖ 迭代：自右向左依次比对字符，找到极大的匹配后缀

若完全匹配，则返回位置；否则

- 查表确定P右移的适当位置，并
- 重新自右向左比对



串

BM算法：BC策略：坏字符

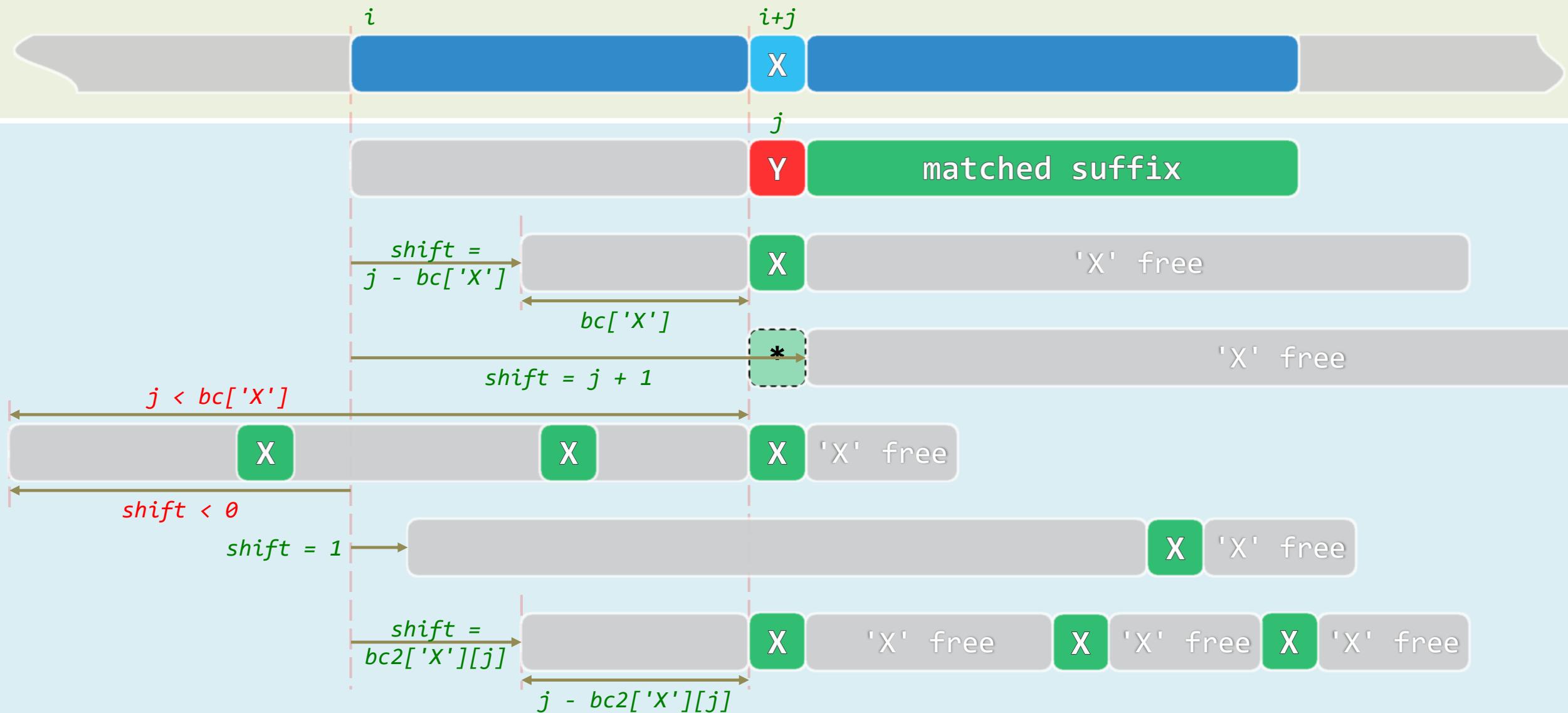
13-D2

邓俊辉

deng@tsinghua.edu.cn

君子求诸己，小人求诸人

# Bad-Character Shift



串

BM算法：BC策略：构造bc[ ]

# 13-D3

邓俊辉

deng@tsinghua.edu.cn

I have learned more from my failures ... failures are much more fun to hear about afterwards; they are not so funny at the time.

# 画家算法

```
int * buildBC( char * P ) {  
    int * bc = new int[ 256 ];  
  
    for ( size_t j = 0; j < 256; j++ ) bc[j] = -1;  
  
    for ( size_t m = strlen(P), j = 0; j < m; j++ )  
        bc[ P[ j ] ] = j; //painter's algorithm
```

```
    return bc;
```

```
    } //O( s + m )
```



# 13-D4

串

BM算法：BC策略：性能分析

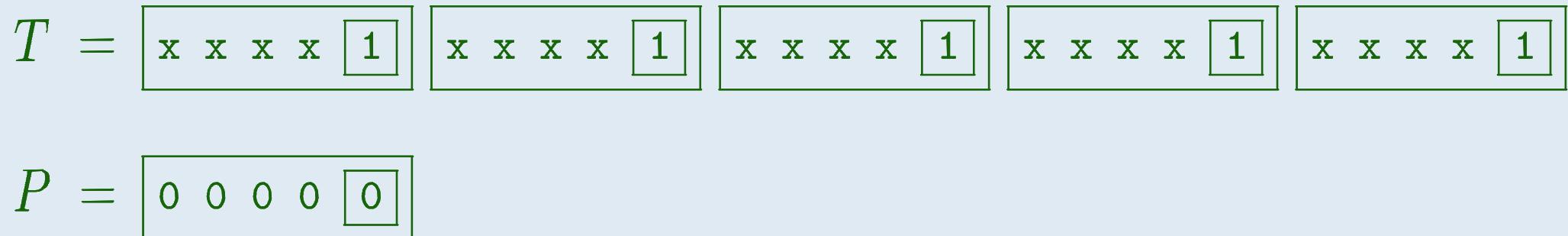
邓俊辉

deng@tsinghua.edu.cn

要不犯十四次，甚至一百四十次错误，就不会得到任何一个真理。

## 最好情况

- ❖  $\mathcal{O}(n/m)$  —— 除法？没错！比如：



- ❖ 一般地：只要 $P$ 不含 $T[i+j]$ ，即可直接移动 $m$ 个字符

仅需单次比较，即可排除 $m$ 个对齐位置

- ❖ 单次匹配概率越小，性能优势越明显 //大字母表：ASCII、UniCode

- ❖  $P$ 越长，这类移动的效果越明显

# 最差情况

- ❖  $\mathcal{O}(n \times m)$  —— 退化为蛮力算法？是的！比如：

$$P = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \end{bmatrix}$$

- ❖ 每轮迭代，都要在扫过整个P之后，方能确定右移一个字符

此时，须经 $m$ 次比较，方能排除单个对齐位置

- ◆ 单次匹配概率越大的场合，性能越接近于蛮力算法 //小字母表Bitmap + DNA

- ◆ 反思：借助以上bc[ ]表，仅仅利用了失配比对提供的信息（教训）！

类比：可否仿照KMP，同时利用起匹配比对提供的信息（经验）？

串

BM算法：GS策略：好后缀

13-

E7

邓俊辉

deng@tsinghua.edu.cn

是谁出的题这么的难

到处全都是正确答案

# 经验 = 匹配的后缀

❖ 首趟比对虽失败，却积累了足够的经验

(匹配的后缀 **ATCH**) // 好后缀

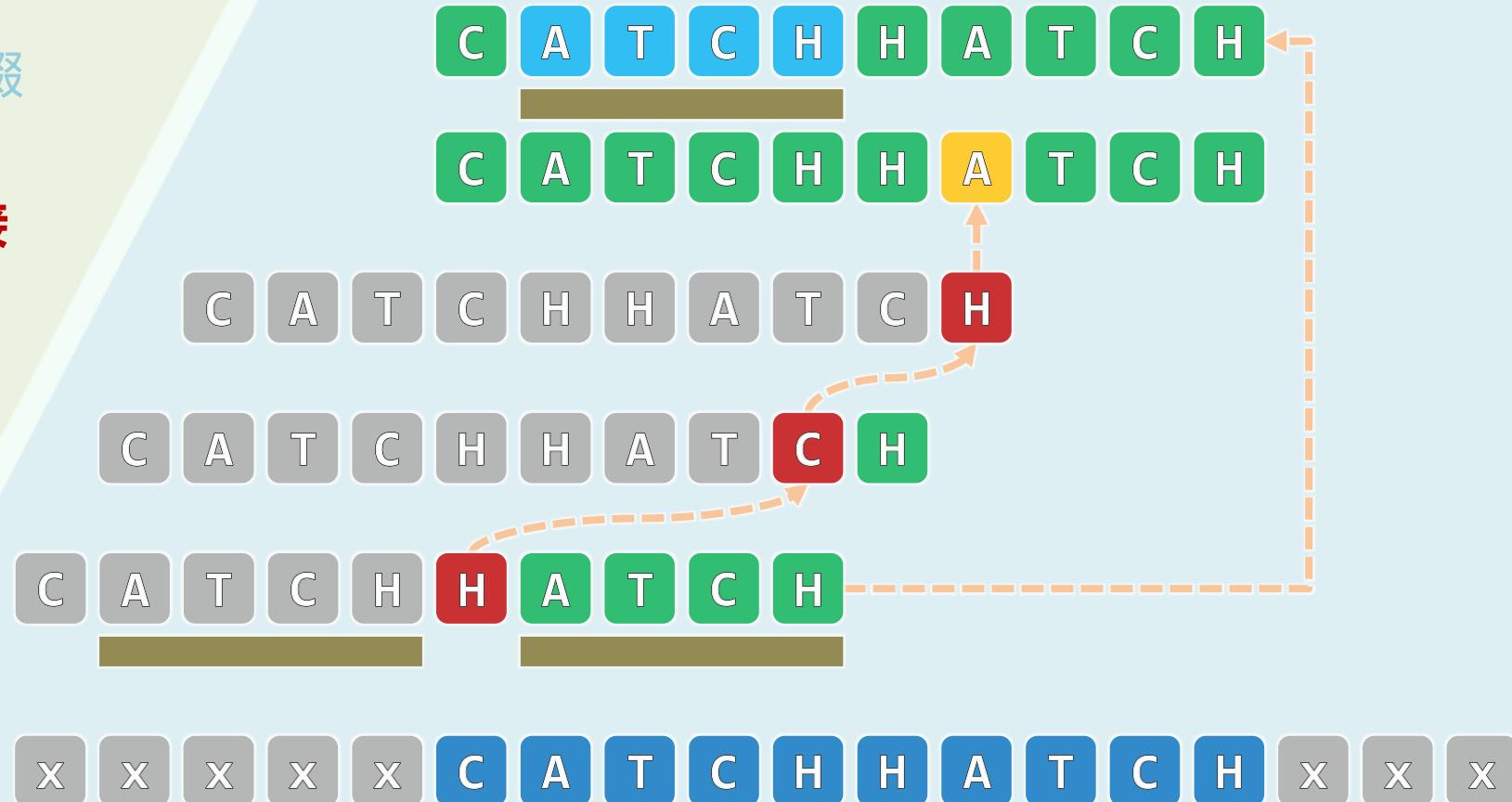
❖ 据此，可省去中间两趟，而直接

转至最后一趟 (P右移5个字符)

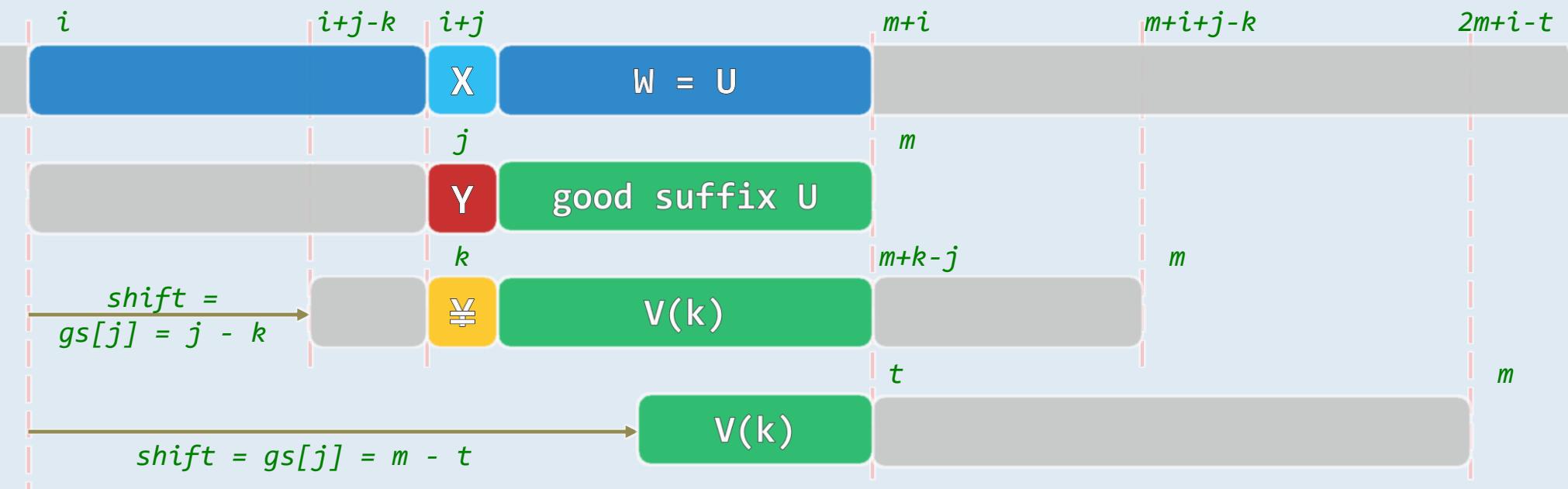
❖ 这一规律与技巧与 **KMP**

如出一辙，只不过

前后颠倒而已...



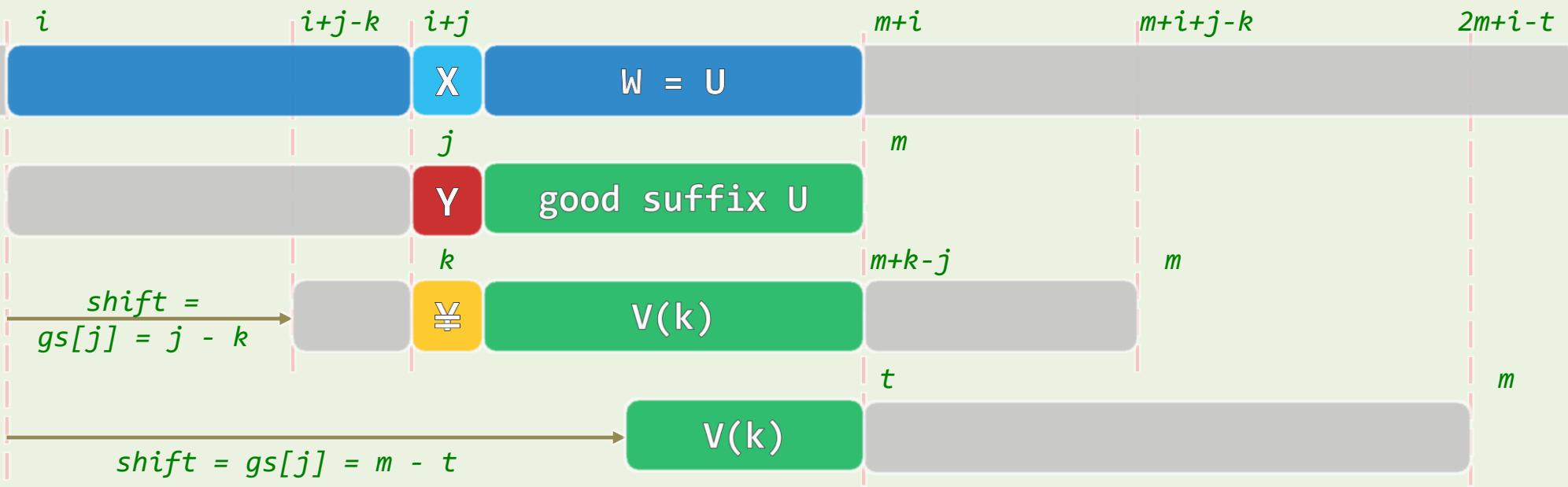
# Good-Suffix Shift



❖ 扫描比对中断于  $T[i + j] = \boxed{X} \neq \boxed{Y} = P[j]$  时,  $U = P(j, m)$  必为好后缀

❖ 故下一对齐位置必须使: 1)  $U$ 重新与  $V(k) = P(k, m + k - j)$  匹配, 且 //经验  
2)  $P[k] = \boxed{\$} \neq \boxed{Y} = P[j]$  //教训

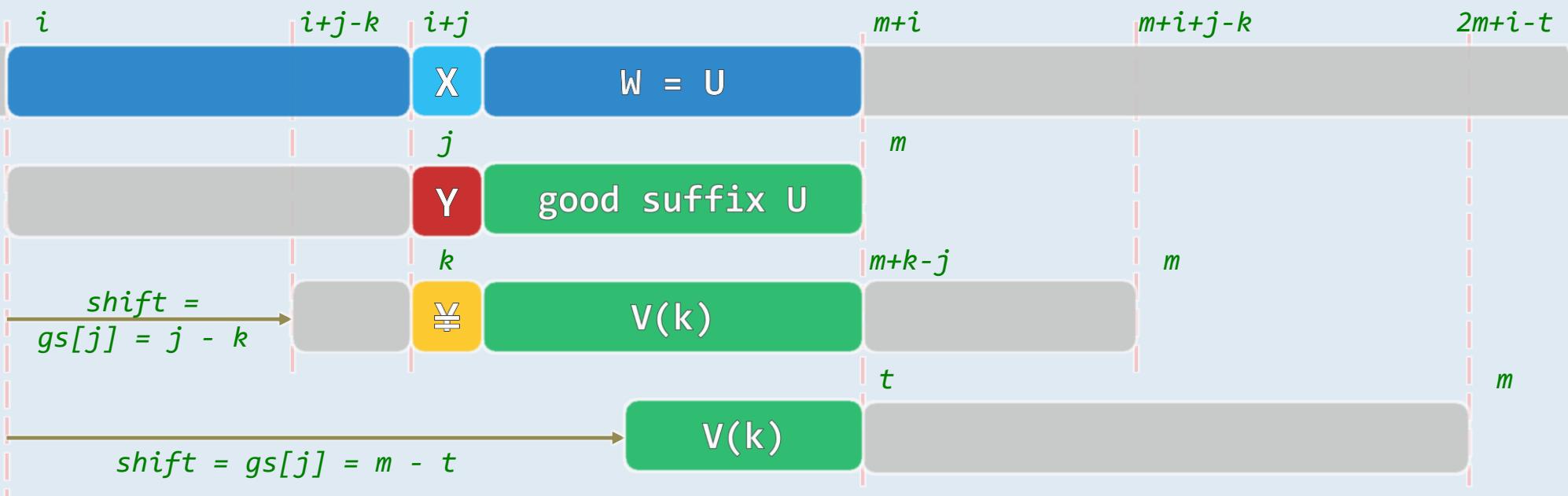
# 完全匹配



❖ 若 $P$ 中的确存在这样的子串 $V(k)$ ，则可

选择其中 $k$ 最大者（尽可能靠后），然后通过右移使之与 $U$ 对齐（移动距离尽可能小）

# 部分匹配



- ❖ 否则，在所有前缀 $P[0, t)$ 中，取与 $U$ 的后缀匹配的最长者 //注意：有可能 $t = 0$
- ❖ 无论如何，位移量仅取决于 $j$ 和 $P$ 本身——亦可预先计算，并制表待查

# 实例



# 13-E2

串

BM算法：GS策略：构造gs表

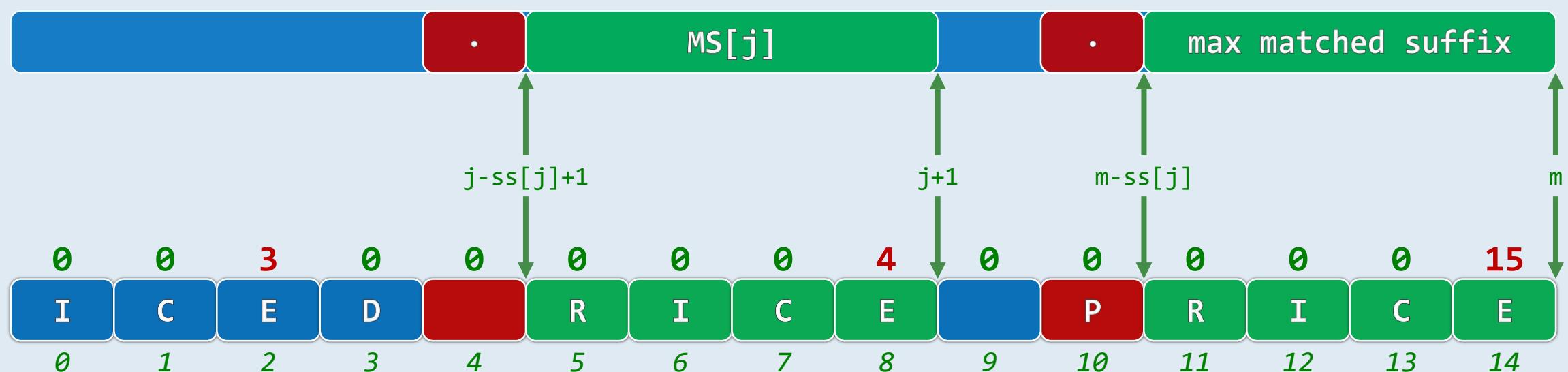
邓俊辉

deng@tsinghua.edu.cn

Wrong cannot afford defeat, but Right can.

## MS[] → ss[]

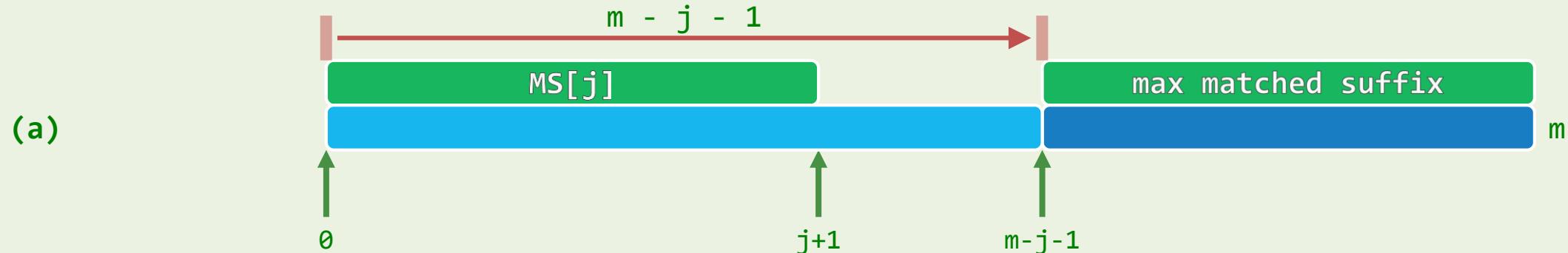
- ❖ 对任一  $0 \leq j < m$  , 令:  $ss[j] = \max\{ 0 \leq s \leq j + 1 \mid P(j - s, j] = P[m - s, m) \}$
- ❖ 于是,  $MS[j] = P(j - ss[j], j]$  就是  $P[0, j]$  所有后缀中, 与P的某一后缀匹配的最长者



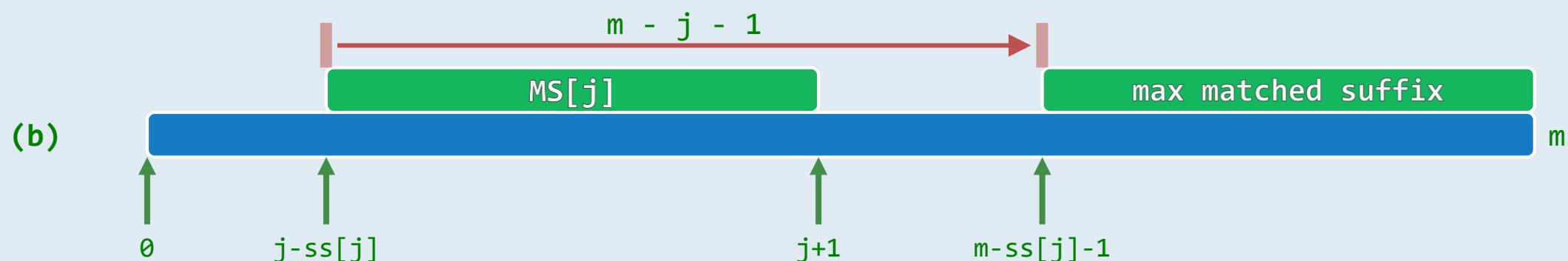
- ❖ 实际上,  $ss[]$  表中蕴含了  $gs[]$  表的所有信息 //无非两种情况...

## $ss[] \rightarrow gs[]$

a) 若  $ss[j] = j + 1$ , 则对于任何  $i < m - j - 1$ ,  $m - j - 1$  必是  $gs[i]$  的一个候选

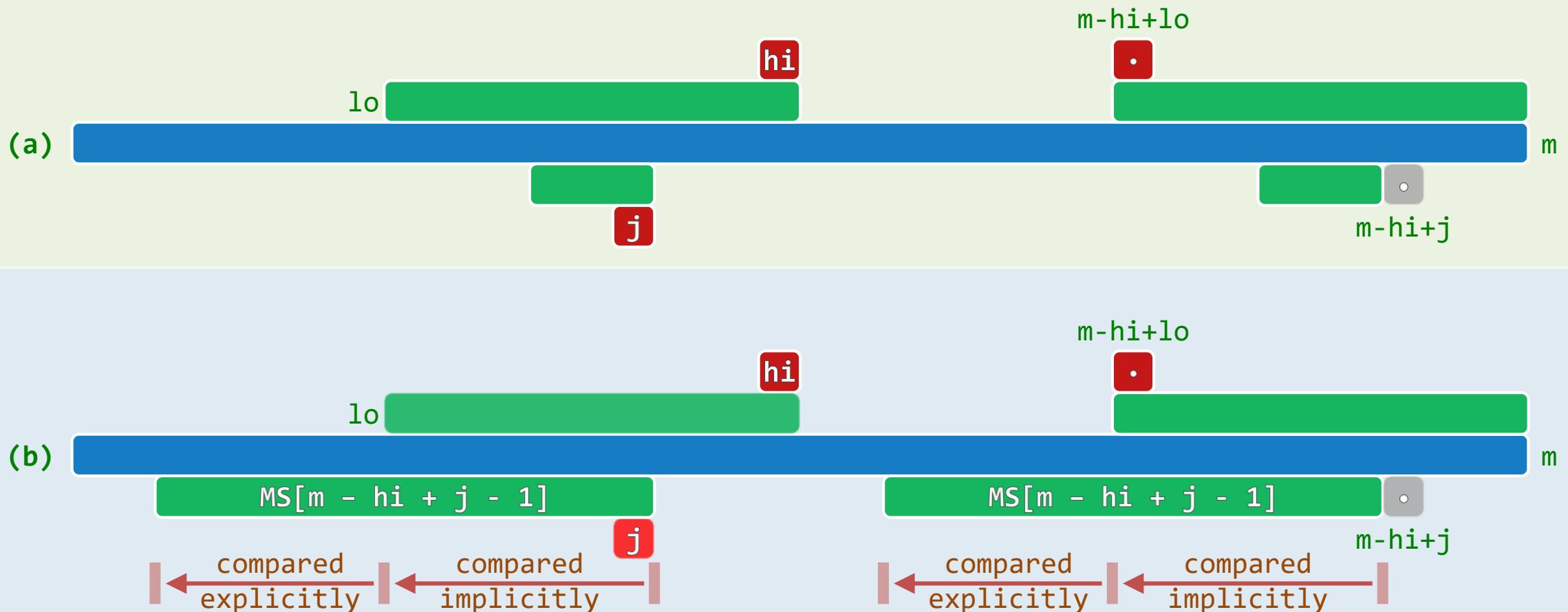


b) 若  $ss[j] \leq j$ , 则  $m - j - 1$  必是  $gs[m - ss[j] - 1]$  的一个候选



## 构造 $ss[]$

❖ 蛮力地对每个字符都扫描一趟，累计 $\Theta(m^2)$ ；自后向前逆向扫描，只需 $\Theta(m)$ 时间 //习题[11-6]



串

BM算法：GS策略：综合性能

13-E3

邓俊辉

deng@tsinghua.edu.cn

叹人生，不如意事，十常八九

婚姻是否能取得最大的幸福，在很多方面要取决于男女双方是不是相配；  
不过，要想在各个方面都相配的话，那是十分愚蠢的。

# 性能

❖ 空间 =  $|bc| + |gs| = \Theta(|\Sigma| + m)$

❖ 预处理:  $\Theta(|\Sigma| + m)$

❖ 查找效率

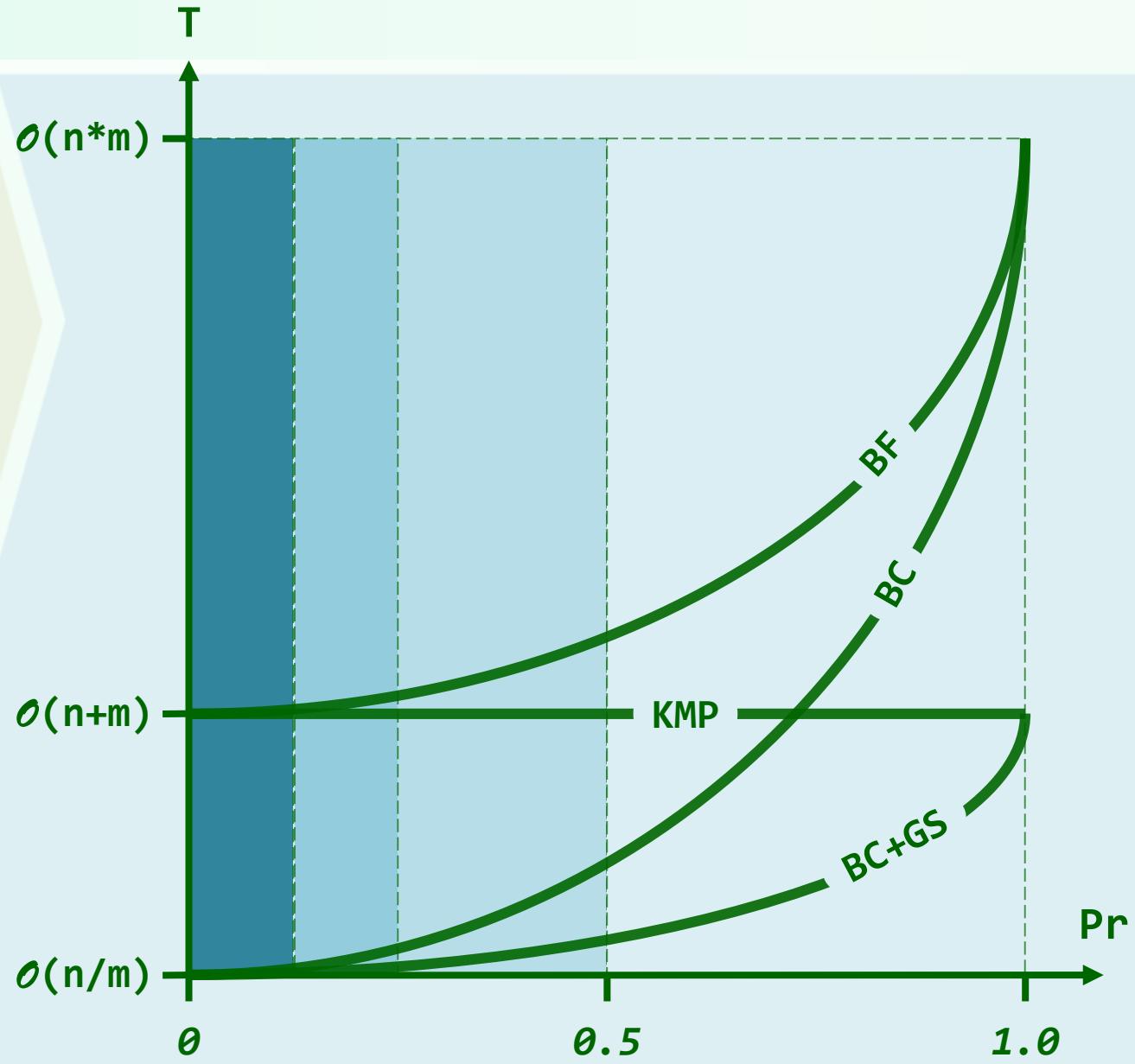
- 最好  $\Theta(n / m)$

- 最差  $\Theta(n + m)$  //分析方法类似KMP

❖ 关键因素

- 单次比对成功的概率

- 通常,  $Pr = 1/s$



# 13 - F1

串

Karp-Rabin算法：串即是数

All things are numbers.

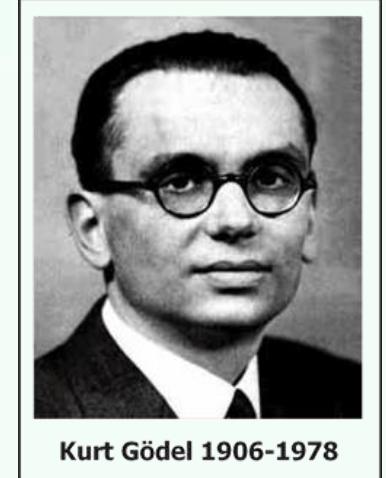
- Pythagoras (570 ~ 495 BC)

God made the integers; all else is the work of man.

- L. Kronecker (1823 ~ 1891)

邓俊辉  
[deng@tsinghua.edu.cn](mailto:deng@tsinghua.edu.cn)

# 凡物皆数：Gödel Numbering



- ❖ 逻辑系统的符号、表达式、公式、命题、定理、公理等，均可表示为自然数
- ❖ 每个有限维的自然数向量（包括字符串），都唯一对应于某个自然数
- ❖ 素数序列： $p(k) = \text{第}k\text{个素数} = 2, 3, 5, 7, 11, 13, 17, 19, \dots$

$$\langle a_1, a_2, \dots, a_n \rangle \Rightarrow p(1)^{1+a_1} \times p(2)^{1+a_2} \times \cdots \times p(n)^{1+a_n}$$

$$\begin{aligned}\langle & [3] & [1] & [4] & [1] & [5] & [9] & [2] & [6] \rangle \\ & 2[4] \times 3[2] \times 5[5] \times 7[2] \times 11[6] \times 13[10] \times 17[3] \times 19[7]\end{aligned}$$

- ❖ "godel" =  $2^{1+7} \times 3^{1+15} \times 5^{1+4} \times 7^{1+5} \times 11^{1+12} = 139869560310664817087943919200000$
- ❖ 若果真如RAM模型所假设的字长无限，则只需一个寄存器即可...

# 凡物皆数：Cantor Numbering

$$cantor_2(i, j) = [(i + j)^2 + 3 \cdot i + j]/2$$

$$cantor_2(2, 3) = [(2 + 3)^2 + 3 \cdot 2 + 3]/2 = 17$$

$$cantor_2(3, 2) = [(3 + 2)^2 + 3 \cdot 3 + 2]/2 = 18$$

0	1	3	6	10	15
2	4	7	11	16	
5	8	12	17		
9	13	18			
14	19				
20					

$$cantor_{n+1}( a_1, \dots; a_{n-1}, a_n, a_{n+1} ) =$$

$$cantor_n( a_1, \dots; a_{n-1}, cantor_2(a_n, a_{n+1}) )$$



Georg Cantor  
(1845-1918)

# 凡物皆数：Cantor Numbering

- ❖ 长度有限的字符串，都可视作 $d=1+|\Sigma|$ 进制的自然数

"decade" = 453145<sub>(10)</sub>

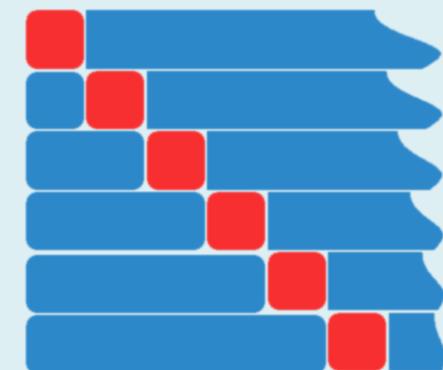
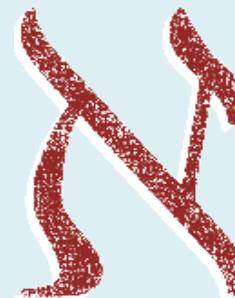
//d = 1 + ('i' - 'a') = 10

- ❖ 长度无限的字符串，都可视作 $[0, 1)$ 内的d进制小数

"bgahbhahbhdei..." = 0.2718281828459...

"fade" = 0.6145 = 0.6144999... = "faddii..."

- ❖ Cantor's Diagonal:  $\aleph_0 < \aleph_1$



# 串亦为数

❖ 十进制串，可直接视作自然数 //指纹 (fingerprint)，等效于多项式法

P = "82818"

T = 271 **82818** 284590452353602874713527

❖ 一般地，随意对字符编号{ 0, 1, 2, …, d - 1 } //设d = |Σ|

于是，每个字符串都对应于一个 d 进制自然数 //尽管不是单射

"CAT" = 2 0 19<sub>(26)</sub> = 1371<sub>(10)</sub> //Σ = { A, B, C, …, Z }

"ABBA" = 0 1 1 0<sub>(26)</sub> = 702<sub>(10)</sub>

❖ P在T中出现 仅当 T中某一子串与P相等 //为什么不是“当”？

❖ 这，不已经就是一个算法了吗？！ //具体如何实现？

❖ 问题似乎解决得很顺利，果真如此简单吗？ //复杂度？

串

Karp-Rabin算法：散列

13-F2

我有些明白了：如果把要指明的恒星与周围恒星的相对位置信息发送出去，接收者把它与星图进行对照，就确定了这颗恒星的位置。

邓俊辉

deng@tsinghua.edu.cn

## 数位溢出

❖ 如果 $|\Sigma|$ 很大，模式串P较长，其对应的指纹将**很长**

比如，若将P视作 $|P|$ 位的 $|\Sigma|$ 进制**自然数**，并将其作为指纹…

❖ 仍以ASCII字符集为例 //  $|\Sigma| = 128 = 2^7$

只要 $|P| > 9$ ，则指纹的长度将至少是： $7 \times 10 = 70$  bits

❖ 然而，目前的字长一般也不过64位 // 存储不便

❖ 而更重要地，指纹的计算与比对，将不能在 $\Theta(1)$ 时间内完成 // RAM!?

准确地说，需要 $\Theta(|P|/64) = \Theta(m)$ 时间；总体需要 $\Theta(n*m)$ 时间 // 与蛮力算法相当

❖ 有何高招？

# 散列压缩

❖ 基本构思：通过对比经压缩之后的**指纹**，确定匹配位置

❖ 关键技巧：通过**散列**，将指纹压缩至存储器支持的范围

比如，采用模余函数： $\text{hash}(\text{key}) = \text{key \% 97}$

❖  $P = [8 \ 2 \ 8 \ 1 \ 8] // \text{hash}(82818) = 77$

❖  $T = 2 \ 7 \ 1 \ [8 \ 2 \ 8 \ 1 \ 8] \ 2 \ 8 \ 4 \ 5 \ 9 \ 0 \ 4 \ 5 \ 2 \ 3 \ 5 \ 3 \ 6$

$[2 \ 7 \ 1 \ 8 \ 2] //22$

$[7 \ 1 \ 8 \ 2 \ 8] //48$

$[1 \ 8 \ 2 \ 8 \ 1] //45$

$[8 \ 2 \ 8 \ 1 \ 8] //77$

# 散列冲突

❖ 注意：hash()值相等，并非匹配的充分条件... //好在必要

因此，通过hash()筛选之后，还须经过严格的比对，方可最终确定是否匹配...

❖ P = **1 8 2 8 4** //hash(18284) = **48**

❖ T = 2 **7 1 8 2 8** **1 8 2 8 4** 5 9 0 4 5 2 3 5 3 6

**2 7 1 8 2** //22

**7 1 8 2 8** //48

• • •

**1 8 2 8 4** //48

❖ 既然是散列压缩，指纹冲突就在所难免——好在，适当选取散列函数，极大降低冲突的概率

# 快速指纹计算

❖ hash()的计算，似乎每次均需 $\Theta(|P|)$ 时间

有可能加速吗？

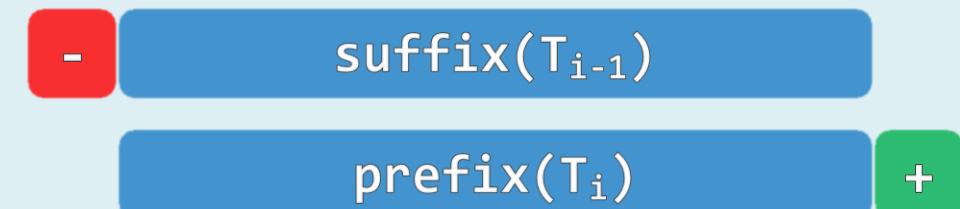
❖ 回忆一下，进制转换算法...

❖ 观察

- 相邻的两次散列之间，存在某种相关性
- 相邻的两个指纹之间，也有某种相关性

❖ 利用上述性质，即可在 $\Theta(1)$ 时间内

由上一指纹得到下一指纹...



串

Karp-Rabin算法：字宽

13 - F3

邓俊辉

deng@tsinghua.edu.cn

God kisses the finite in his love and man the infinite.

当一个人反复思考的时候，就必定会出现悖论，然而不管你们会怎么说，  
我宁愿做一个持有悖论的人，也不愿做心存偏见的人。

$\text{power}_a(n) = a^n$

❖ 平凡实现：

```
pow = 1; //O(1)
```

```
while ( 0 < n ) //O(n)
```

```
{ pow *= a; n--; } //O(1+1)
```

❖  $T(n) = 1 + 2n = \mathcal{O}(n)$

线性？**伪线性！**

❖ 所谓输入规模，准确地应定义为

用以**描述输入所需的空间规模**

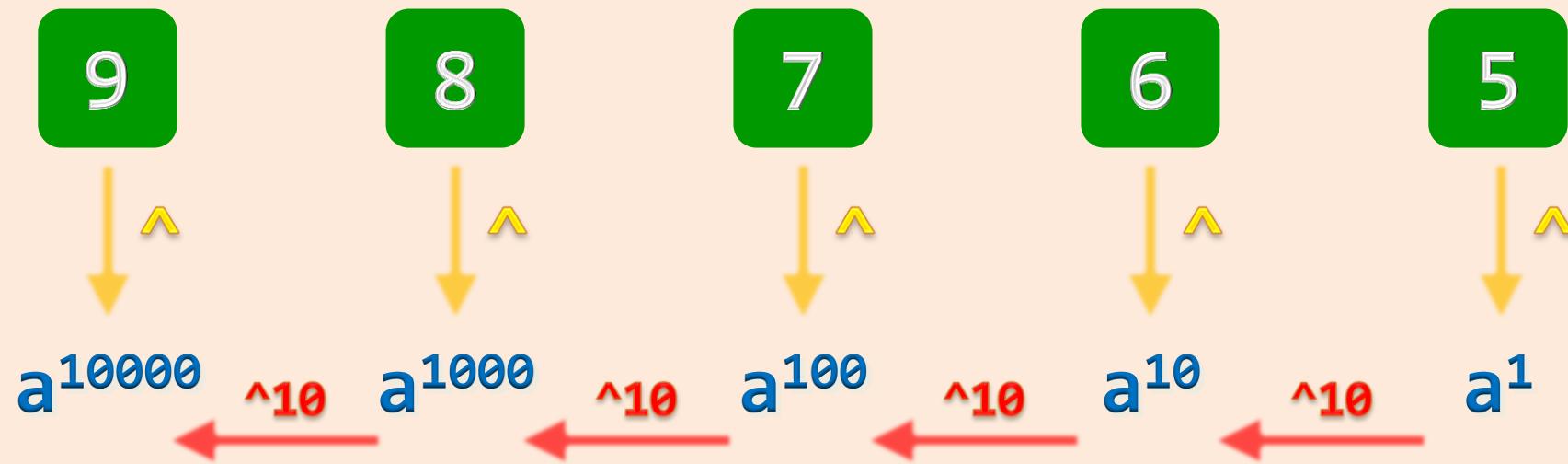
❖ 对于此类数值计算

即是n的二进制位数，亦即n的打印宽度

$$r = \lceil \log_2(n + 1) \rceil = \mathcal{O}(\log n)$$

$T(r) = \mathcal{O}(2^r)$  //指数复杂度！

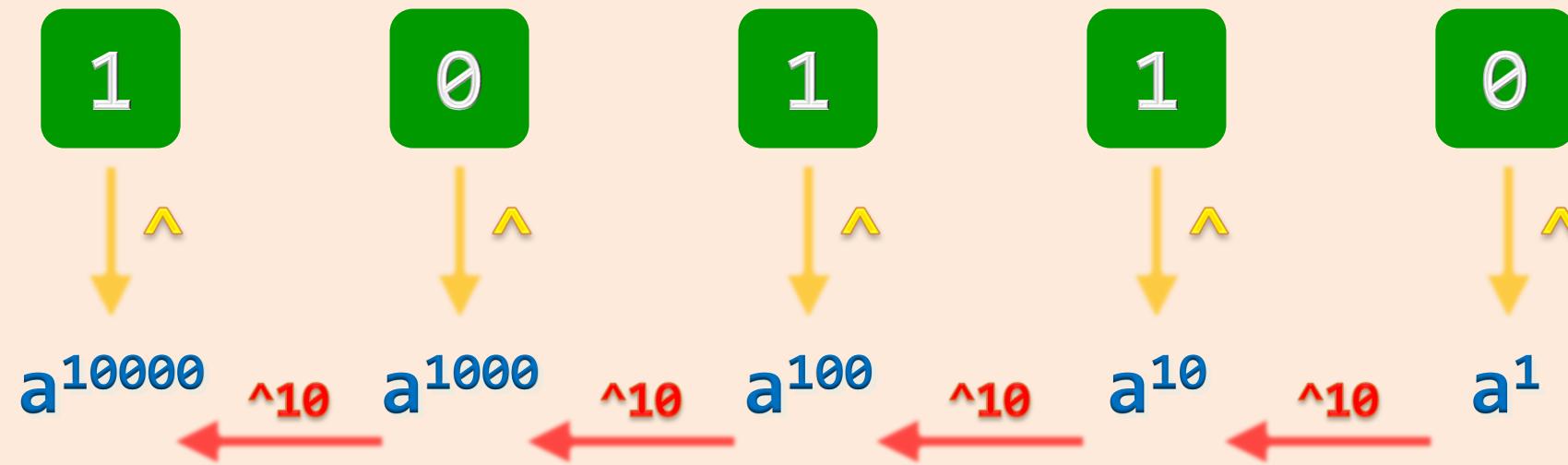
$$a^{9 \cdot 10^4} + 8 \cdot 10^3 + 7 \cdot 10^2 + 6 \cdot 10^1 + 5 \cdot 10^0$$



$$(a^{10^4})^9 \cdot (a^{10^3})^8 \cdot (a^{10^2})^7 \cdot (a^{10^1})^6 \cdot (a^{10^0})^5$$

$a^{10110b}$

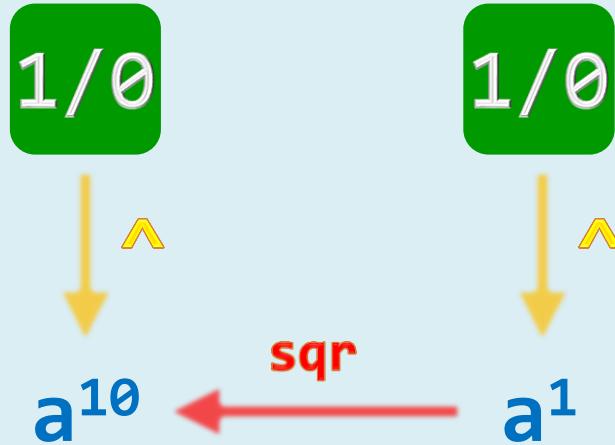
$$a^{1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0}$$



$$(a^{2^4})^1 \cdot (a^{2^3})^0 \cdot (a^{2^2})^1 \cdot (a^{2^1})^1 \cdot (a^{2^0})^0$$

# 从 $\mathcal{O}(n)$ 到 $\mathcal{O}(r = \log n)$

```
int power( int a, int n ) {  
    int pow = 1, p = a; //  $\mathcal{O}(1 + 1)$   
    while (0 < n) { //  $\mathcal{O}(\log n)$   
        if (n & 1) //  $\mathcal{O}(1)$   
            pow *= p; //  $\mathcal{O}(1)$   
        n >>= 1; //  $\mathcal{O}(1)$   
        p *= p; //  $\mathcal{O}(1)$   
    }  
    return pow; //  $\mathcal{O}(1)$   
}
```



- ❖ 输入规模  $= r = \lceil \log_2 (n + 1) \rceil$
- ❖ 运行时间  $= T(r) = 1 + 1 + 4r + 1 = \mathcal{O}(r)$
- ❖ 如此，“实现”了从指数到线性的改进

# 悖论?

❖ 根据以上算法，“可以”在  $\mathcal{O}(\log n)$  时间内计算出  $\text{power}(n) = a^n$

❖ 然而， $a^n$  的二进制展开宽度为  $\Theta(n)$

这意味着，即便是直接打印  $a^n$ ，也至少需要  $\Omega(n)$  时间……哪里错了？

❖ 类似的悖论对  $\text{fib}(n)$  也存在...

❖ 令： $\mathcal{A} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} \text{fib}(0) & \text{fib}(1) \\ \text{fib}(1) & \text{fib}(2) \end{bmatrix}$ ，则： $\mathcal{A}^n = \begin{bmatrix} \text{fib}(n-1) & \text{fib}(n) \\ \text{fib}(n) & \text{fib}(n+1) \end{bmatrix}$

❖ 因此参照上述  $\text{power}()$  算法，似乎也“可以”在  $\mathcal{O}(\log n)$  时间内计算出  $\text{fib}(n)$

❖ RAM模型：常数代价准则 (**uniform cost criterion**)

对数代价准则 (**logarithmic cost criterion**)

串

键树

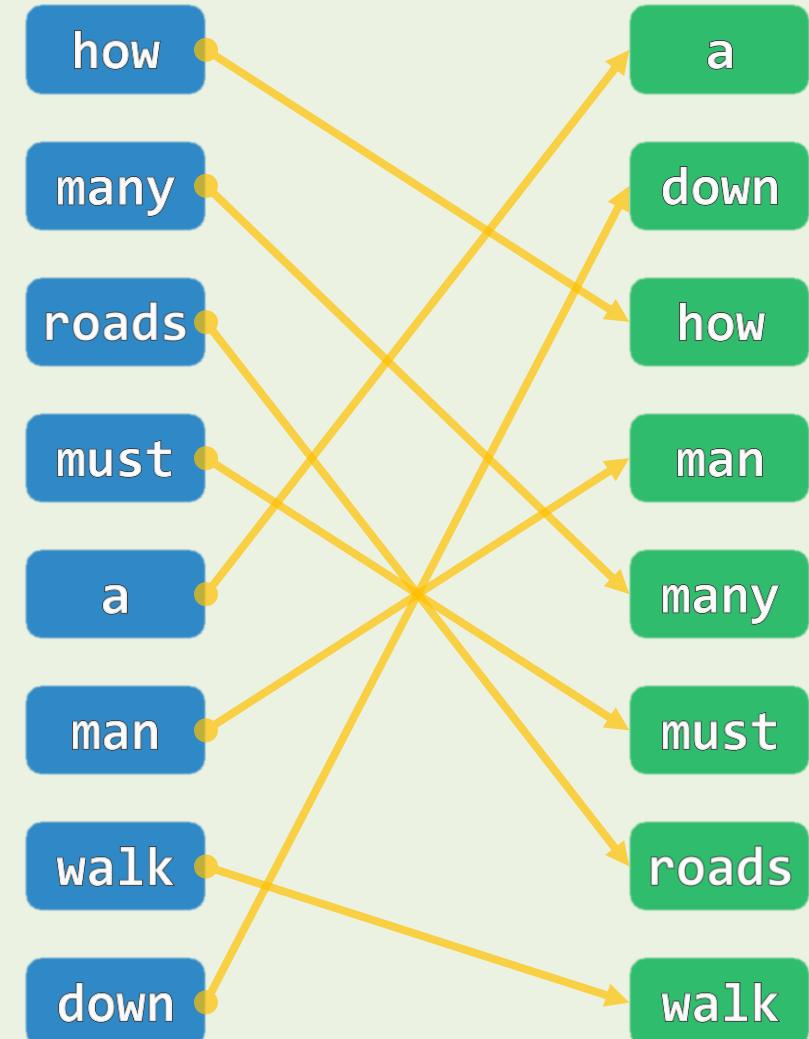
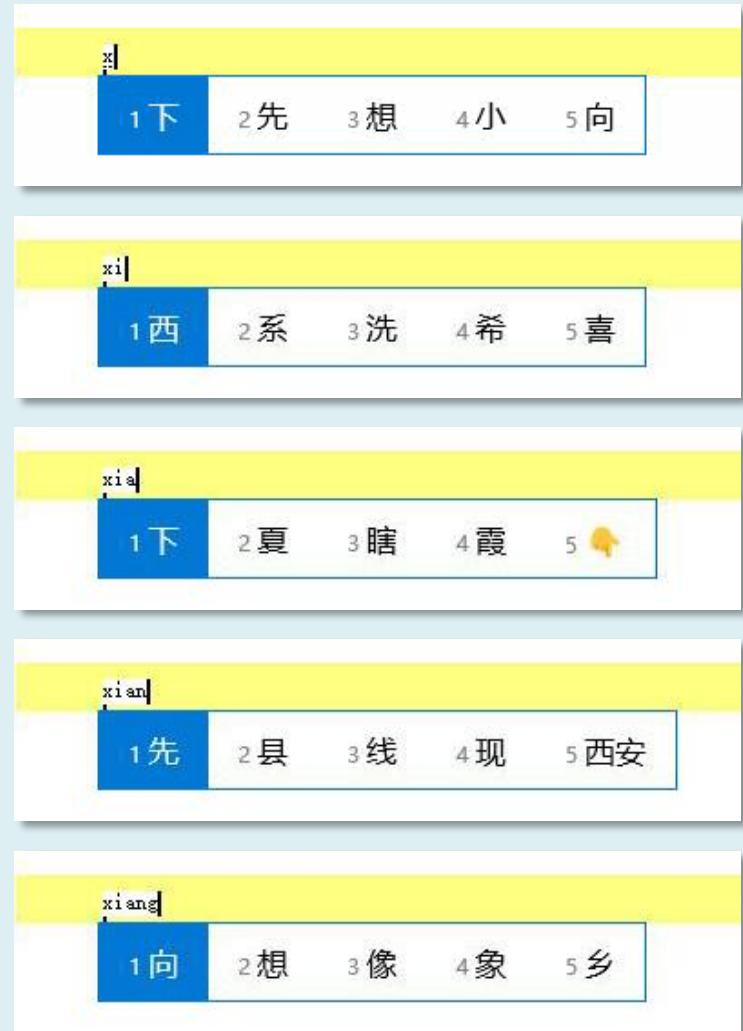
# 13 - G

邓俊辉

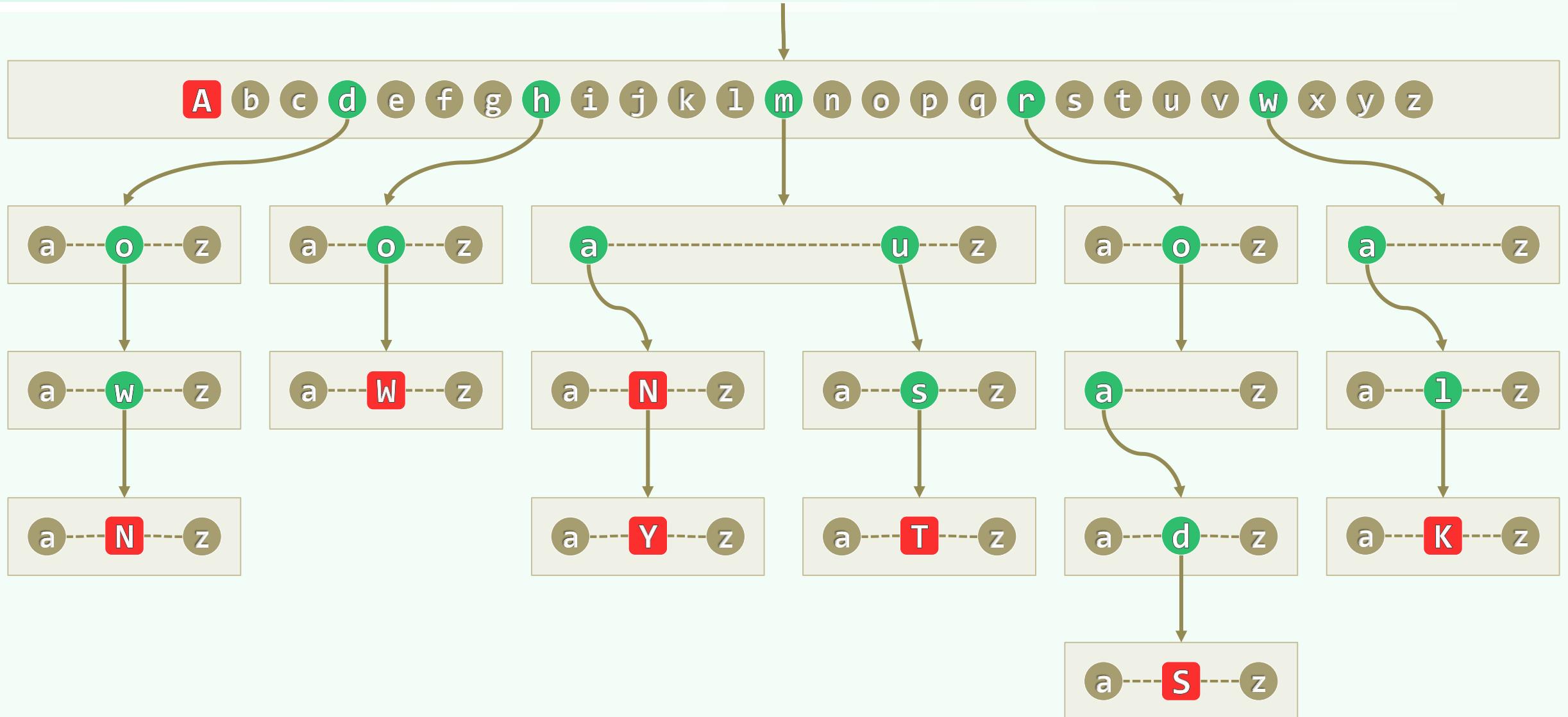
deng@tsinghua.edu.cn

两个吃罢饭，又走了四五十里，却来到一市镇上，地名唤做瑞龙镇，却是个三岔路口。宋江借问那里人道：“小人们欲投二龙山、清风镇上，不知从那条路去？”

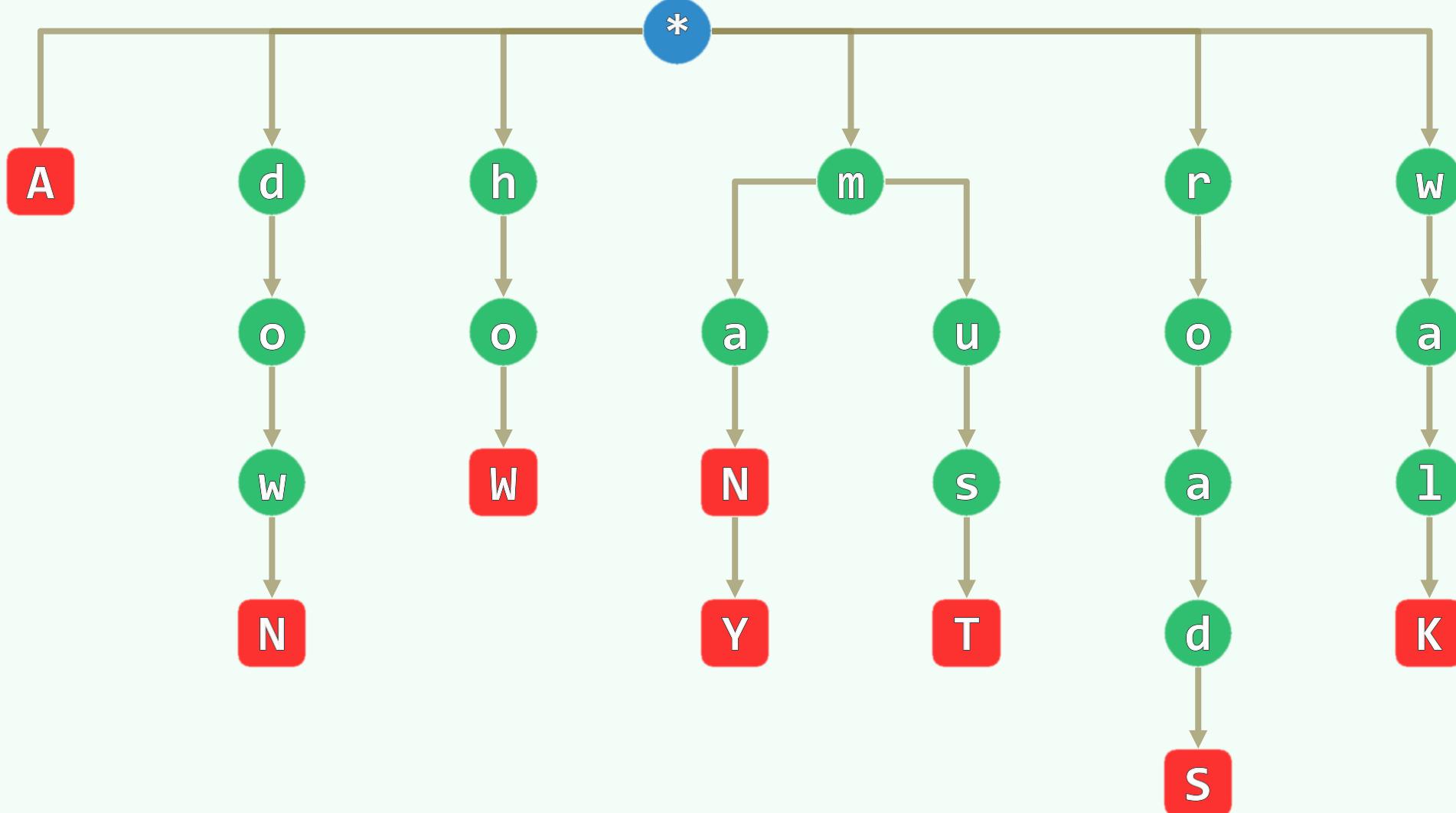
# Trie = Digital Tree = Radix Tree = Prefix Tree



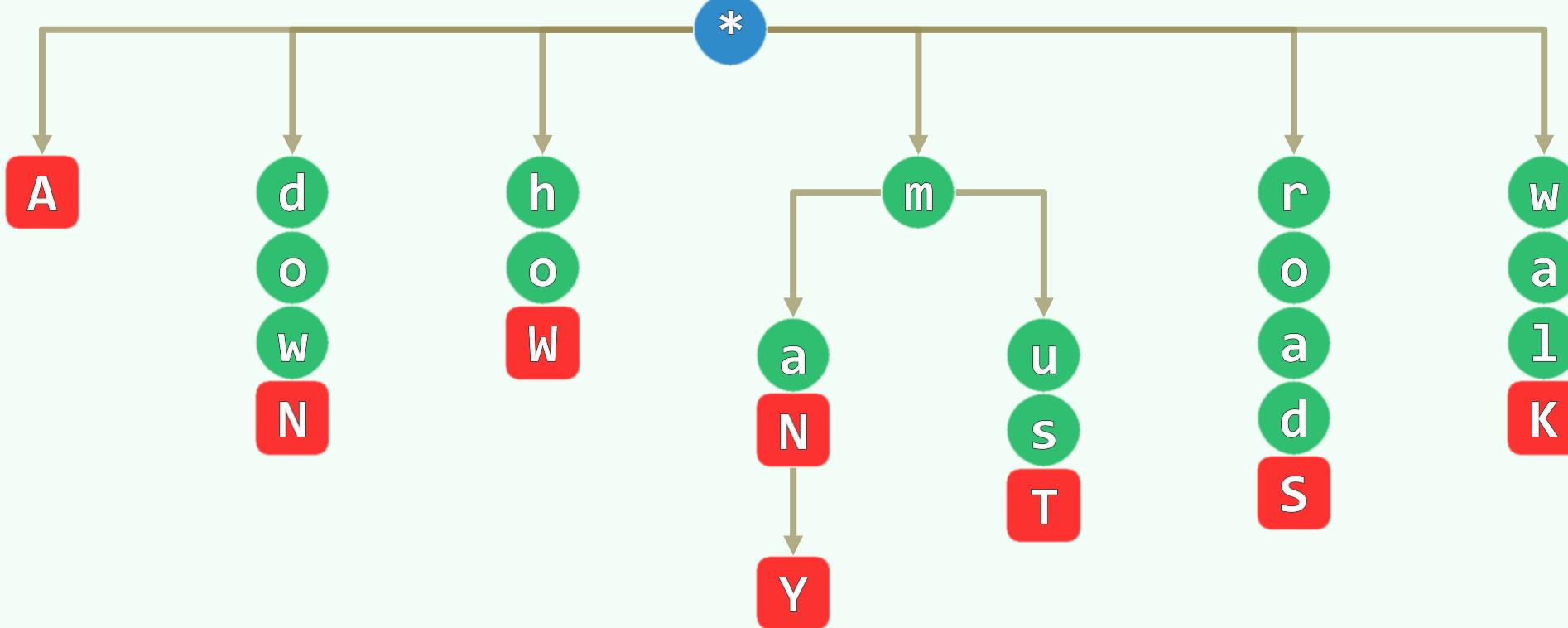
# Trie: how many roads must a man walk down



Trie = reTRIEval



## Trie: PATRICIA Tree



## Trie: Ternary Trie

