实用算法设计——查找

主讲：娄文启

louwenqi@ustc.edu.cn

5 查找

假定：待查找的数据已在内存中。

重点：理解KMP算法和BM算法的思想。能利用KMP算法和BM算法解决实际的子串查找需求。

难点：理解KMP算法为何比朴素匹配算法快；BM算法为何比KMP算法快。

基础：C语言编程；线性结构的定义及基本操作的实现。

5 查找

5.1 基于Hash表的查找

5.2 蛮力查找（顺序查找）

5.3 基于有序表的二分查找

5.4 字符串的查找

5.5 基于树的查找

- 字符串

- 模式匹配算法
  1. 朴素的模式匹配算法
  2. KMP模式匹配算法
  3. BM模式匹配算法

重难点

- 理解字符串是特殊的线性表，其特殊性在哪？
- 朴素的模式匹配算法为什么慢？
- **KMP**算法
  - 与朴素的模式匹配算法相比，**KMP**算法为什么快？
  - **KMP**算法中**Next**数组的设计初衷是？
  - 如何计算**Next**数组？如何利用**Next**数组来实现快速匹配？
  - **Next**数组存在什么问题，以致于后面提出了改进版的**NextVal**数组
  - 如何计算**NextVal**数组？如何利用**NextVal**数组来实现快速匹配？
- **BM**算法：
  - 理解两类启发式规则：“坏字符”和“好后缀”
  - 如何计算目标串上查找指针的跳跃举例**dist[i]**，以定位下轮匹配的右对齐的位置？
    - 如何计算**MatchJump**数组？
    - 如何计算**CharJump**数组

- 字符串：由零个或多个字符组成的有限序列。
  - 字符串s="$a_1a_2...a_n$"(n≥0)
  - 串的长度：串中的字符数目n
  - 空串：零个字符的串。即串的长度为0.
  - 空格串：只包含空格的串。
- 逻辑结构：可认为是特殊的线性表。
  - 串中的元素只能是字符类型。
  - 字符串：关注子串的操作：查找/插入/删除/替换子串.
    - 线性表：关注单个元素的操作
- 存储结构：推荐顺序存储方式。
  - 例如，存储串"good"，需要5个字节，但是其长度为4。

g	o	o	d	\0

模式匹配算法

- 字符串查找场景：
  - 例如，在文本“**fffffab cfe defe**”中查找字符串“**ff**”
  - 本质：在字符线性表（主串或目标字符串**T**）中查找匹配的子表（子串或模式字符串**P**）。

- 字符串查找（或模式匹配）算法：
  1.朴素的模式匹配算法
  2.KMP算法（Knuth-Morris-Pratt)
  3.BM算法（Boyer-Moore)

1. 朴素的模式匹配算法

示例1：

假设我们要从下面的主串T="goodgoogle"中，找到P="google"这个子串的位置。我们通常需要下面的步骤。

1．主串T第一位开始，T与P 前三个字母都匹配成功，但T第四个字母是 d 而P的是 g。第一位匹配失败。如图 5-6-1 所示，其中竖直连线表示相等，闪电状弯折连线表示不等。

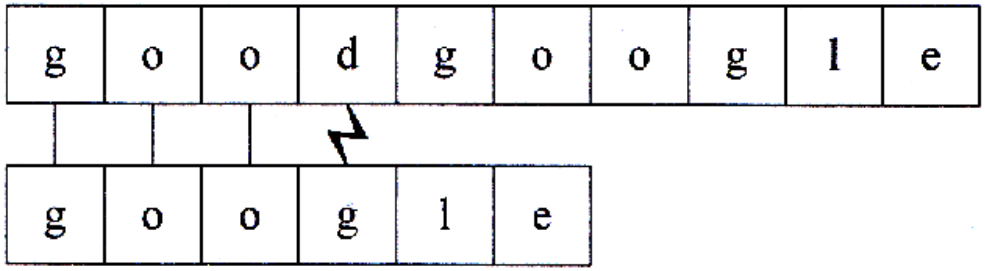


图 5-6-1

2．主串 T 第二位开始，主串 T 首字母是 o，要匹配的 P 首字母是 g，匹配失败，如图 5-6-2 所示。

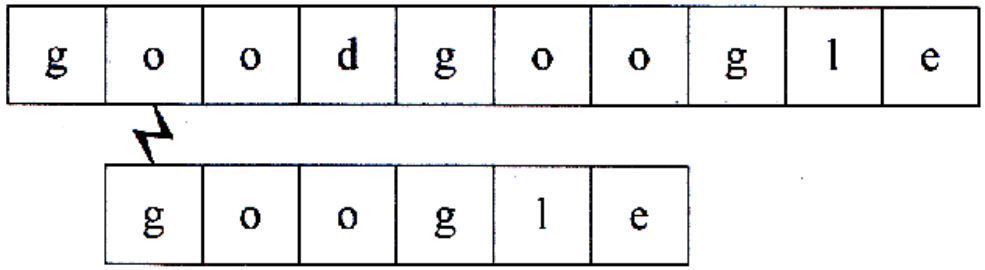


图 5-6-2

3．主串 T 第三位开始，主串 T 首字母是 o，要匹配的 P 首字母是 g，匹配失败，如图 5-6-3 所示。

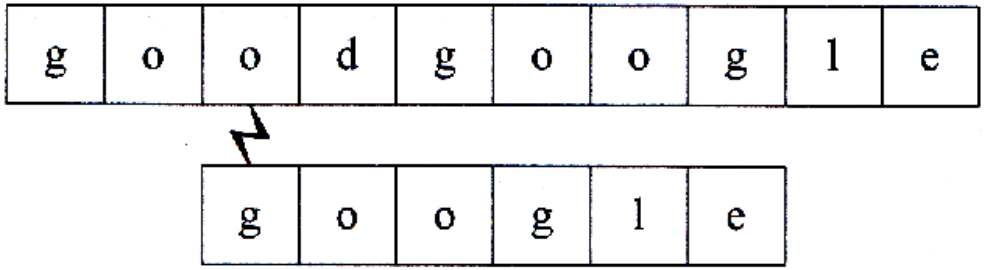


图 5-6-3

4．主串 T 第四位开始，主串 T 首字母是 d，要匹配的 P 首字母是 g，匹配失败，如图 5-6-4 所示。

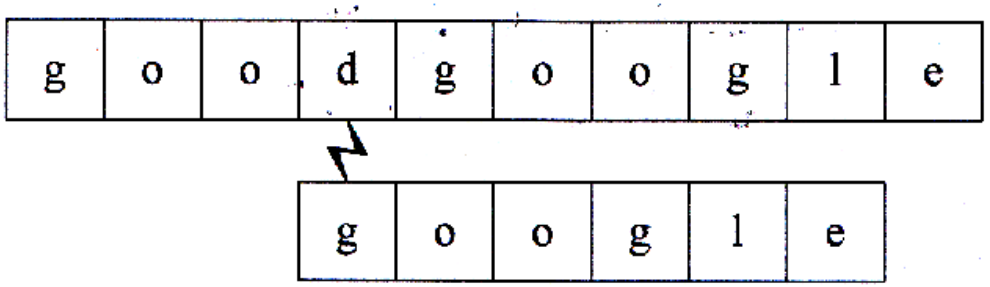


图 5-6-4

5．主串 T 第五位开始，T 与 P，6 个字母全匹配，匹配成功，如图 5-6-5 所示。

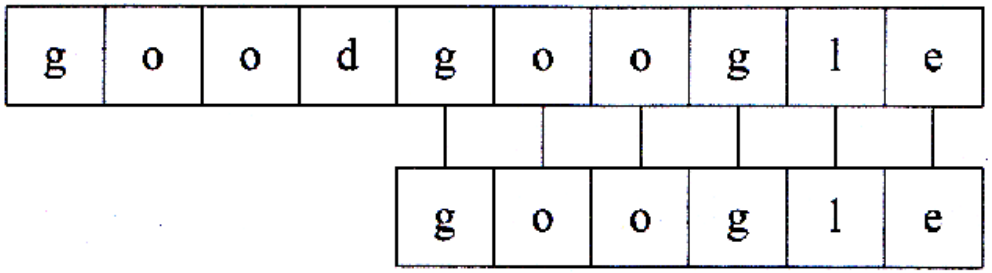


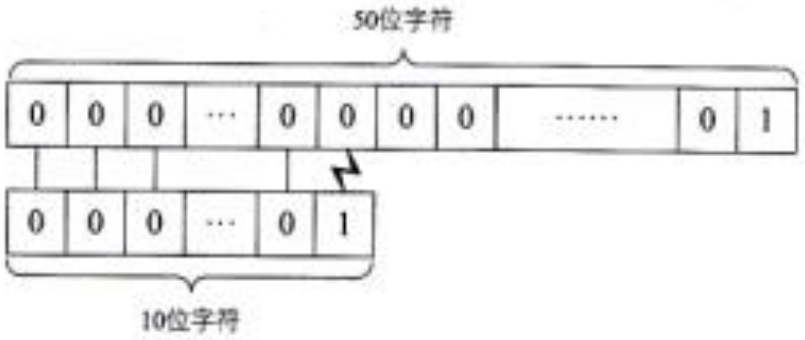
图 5-6-5

假设目标串T长度为n，模式串P长度为m。

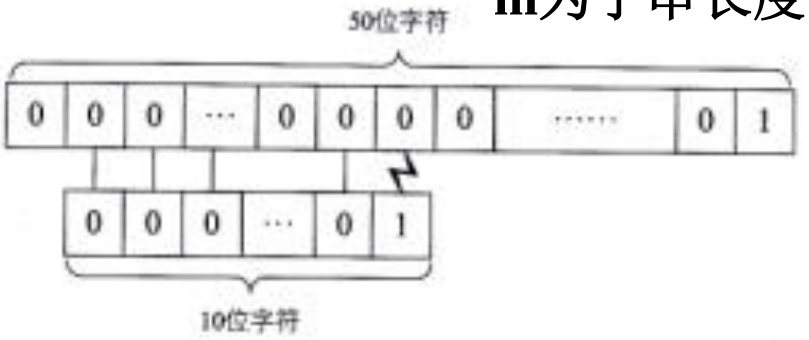
朴素的模式匹配算法的最坏情况下近似时间复杂度为O((n-m+1)*m).

（假设n为主串长度，m为子串长度）

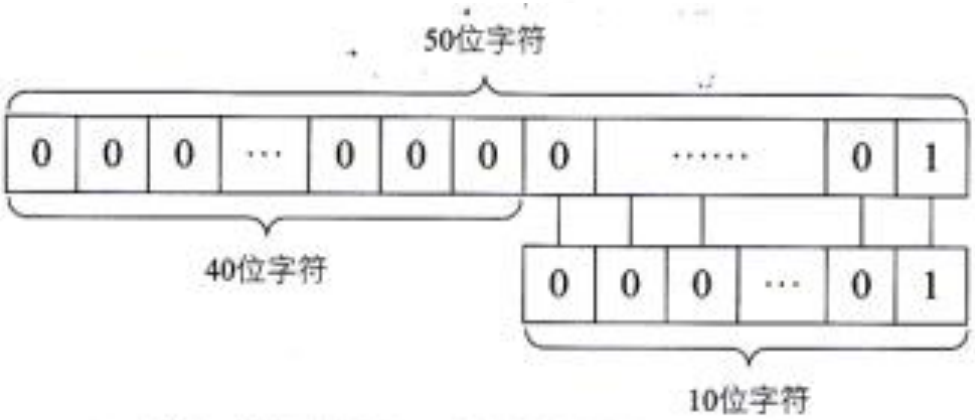
示例2：



T在第一位置判断了10次发现字符串不匹配



T在第二位置再次判断了10次发现字符串不匹配



T在第41个位置判断了10次发现字符串终于匹配成功。
期间进行了（50-10+1）×10次判断操作

- **假设目标串T长度为n，模式串P长度为m。假定T中确实存在子串P（即假定模式匹配成功），则：**
  - 最好的情况下的近似时间复杂度为O(m)
    - 比如，在"googlegood"中查找"google".
  - 若每次不成功的匹配都发生在串P的首字符处，则平均情况下的近似时间复杂度为O(n+m)
  - 最坏的情况下的近似时间复杂度为O((n-m+1)*m).
    - 比如，在“000…00001"中查找“0001".

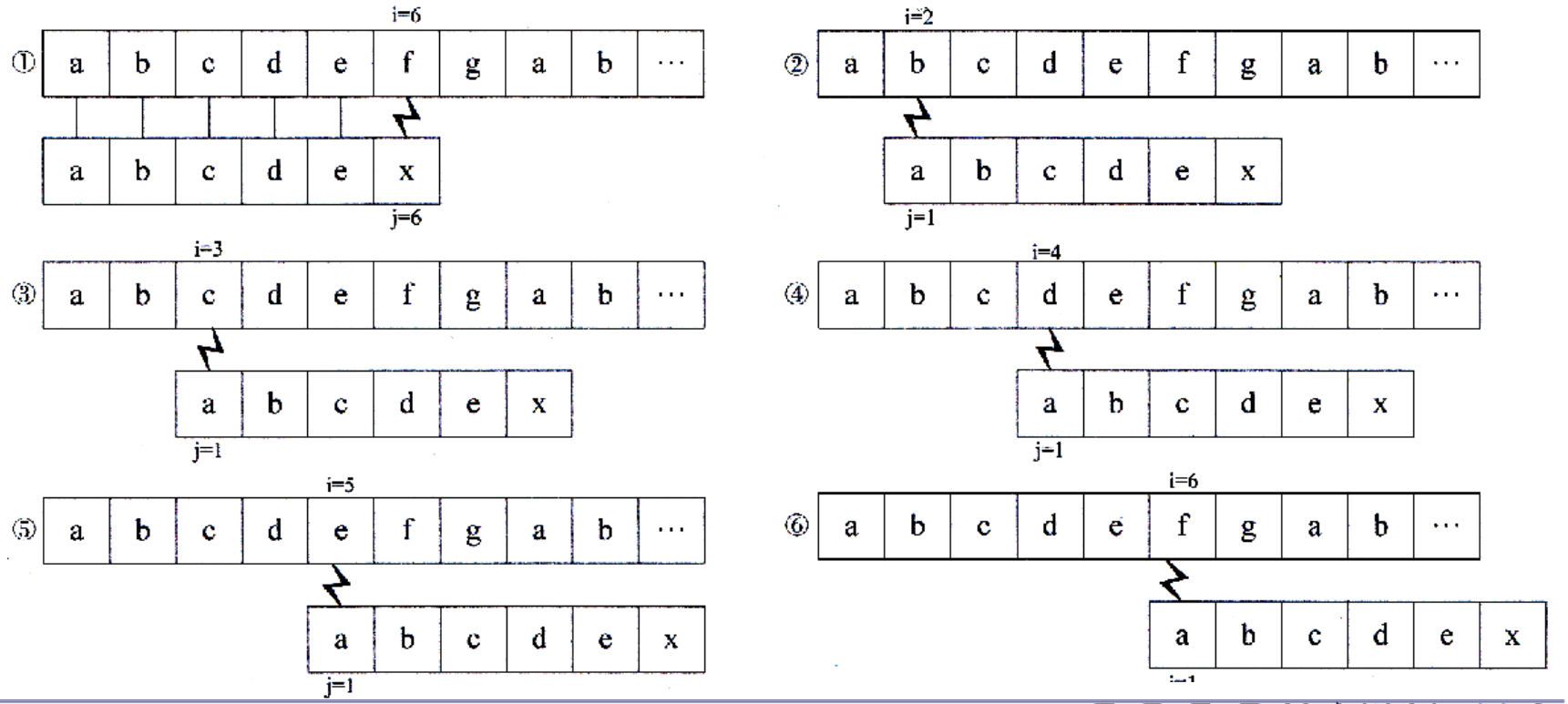
朴素的模式匹配算法 -> KMP, BM

- 朴素的模式匹配算法：
  - 需进行回溯
- 字符串查找算法中，最著名的两个算法：KMP算法（Knuth-Morris-Pratt)和BM算法（Boyer-Moore)
  - 它们都是精确字符串匹配算法（区别于模糊匹配）。
  - 目标字符串中无需进行回溯。（都比朴素的模式匹配算法快）
  - 模式字符串的移动方向：从目标字符串的第一个字符开始，朝 目标字符串的尾部方向移动搜索匹配子串。
  - 每轮匹配时，字符匹配的方向：
    - KMP算法：采用从左向右进行字符的匹配比较。
    - BM算法：采用从右向左进行字符的匹配比较。

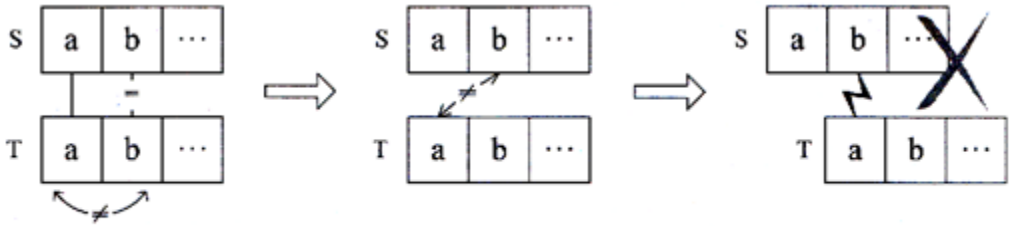
2. KMP算法

- KMP算法的设计初衷：
  - 希望能在匹配过程中，目标串T中的字符匹配是一直向右前进的，而不会出现向左的回溯
  - 理论上可以避免吗？
  - 如何避免？（利用模式串P自身的重复模式）

例1：模式串P中无重复模式



②③④⑤的判断都是多余



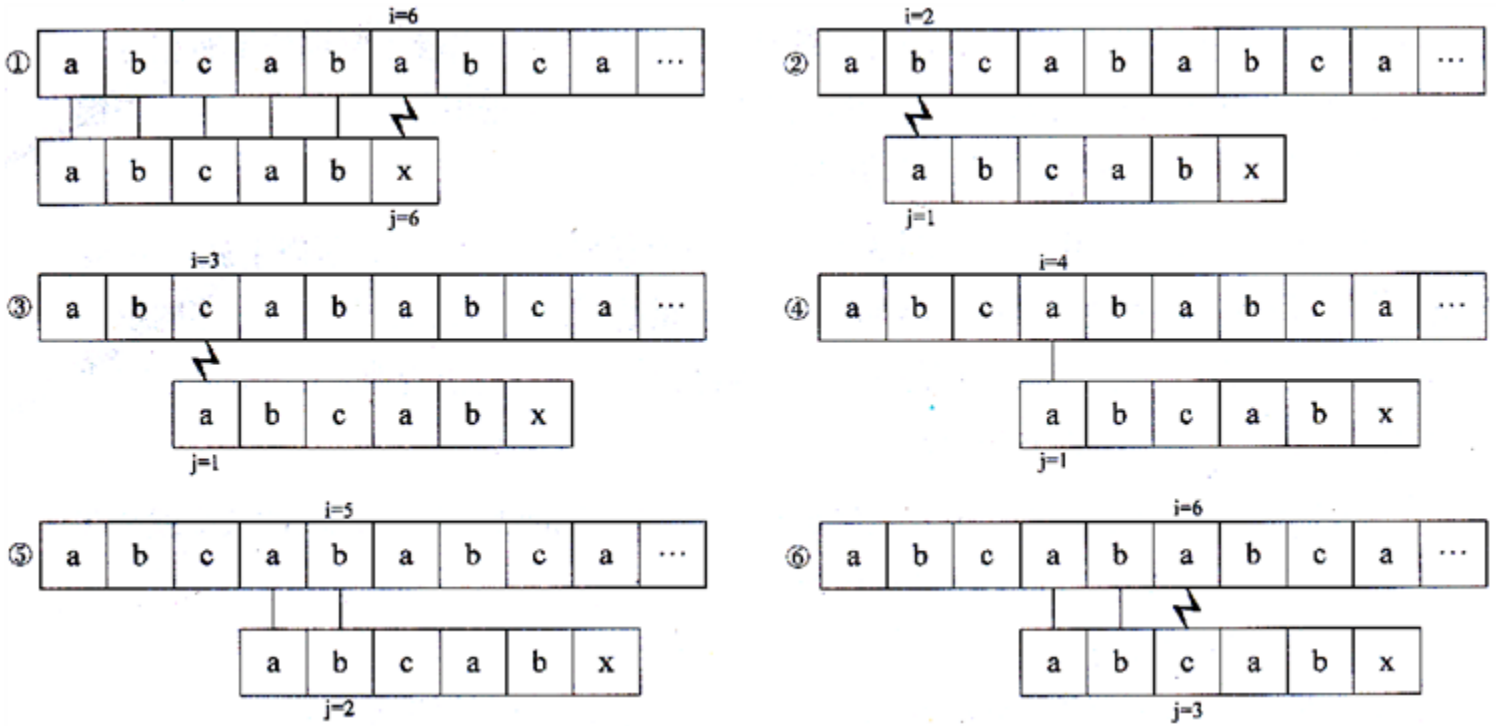
T[1]=a,T[2]=b,S[2]=b
显然T[1]≠T[2]，T[2]=S[2]

因此T[1]≠S[2]

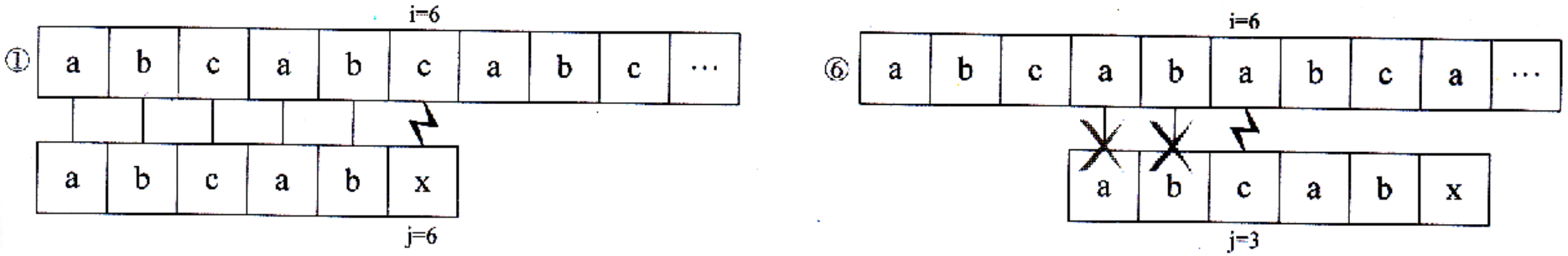
因此当T位于第二位置的
判断就根本不需要进行了

例2：模式串P中有重复模式

假设T=“abcabcabc”，P=“abcabx”。

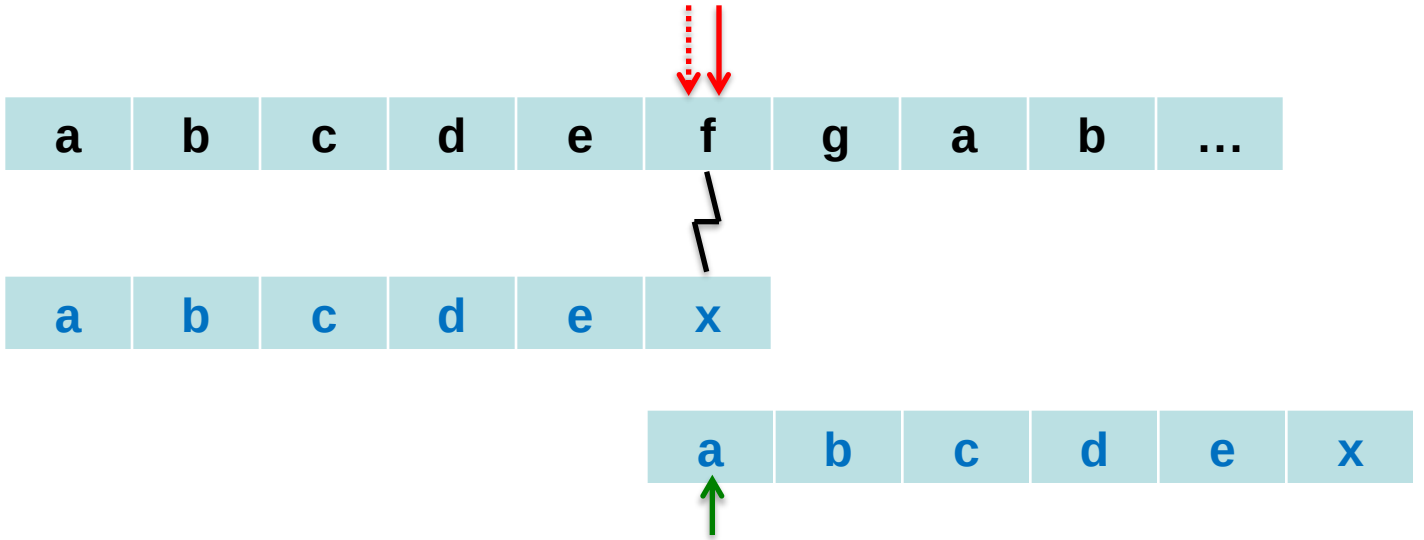


步骤是②③④⑤都是多余的！⑥中的前2次字母的匹配也是多余的！



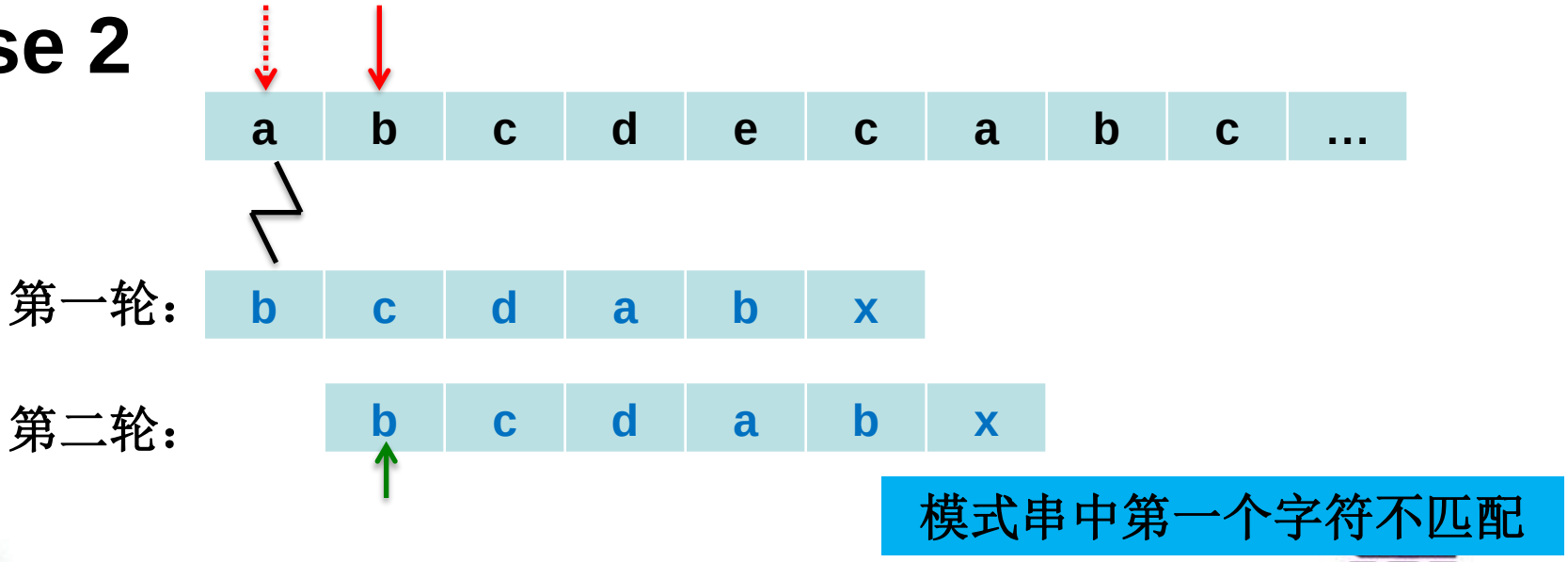
- 如何避免目标串中字符匹配的回溯问题？（利用模式串P自身的重复模式）
  - 假定目标串为T，模式串为P。若T[i]≠P[j]导致当前轮的匹配失败，则下一轮匹配时应该比较T和P中的哪两个字符？
    - 每轮中，T与P都是从左至右逐字符比较。
  - 关注：
    - 匹配失败，是否发生在P的第一个字符处？
    - P中是否有重复模式？

- **Case 1**



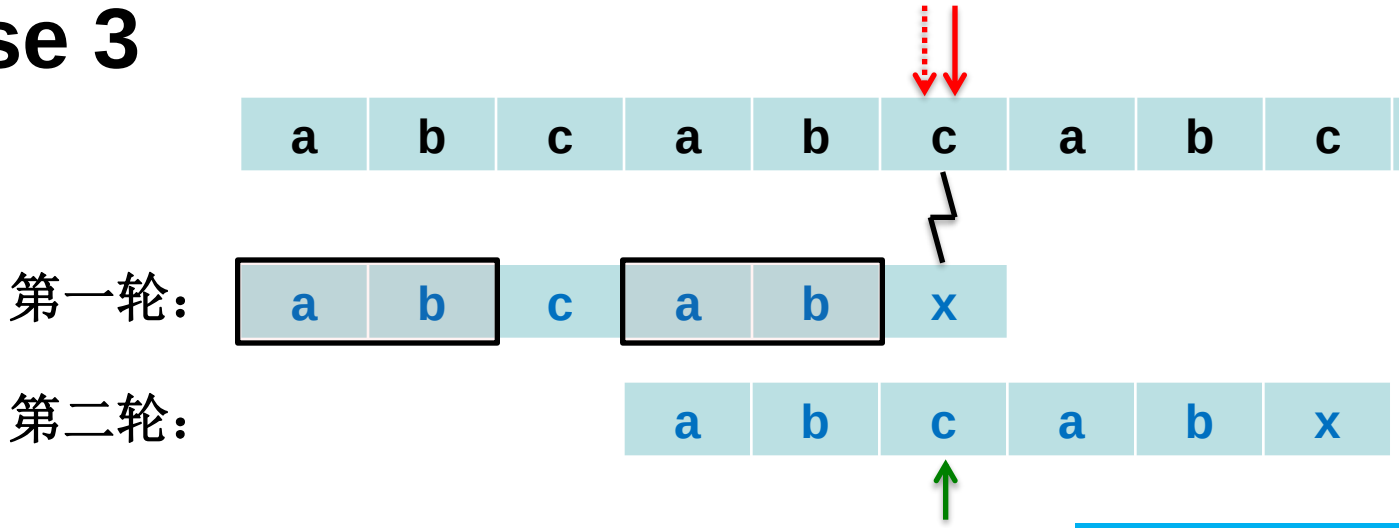
模式串中无重复模式

- **Case 2**



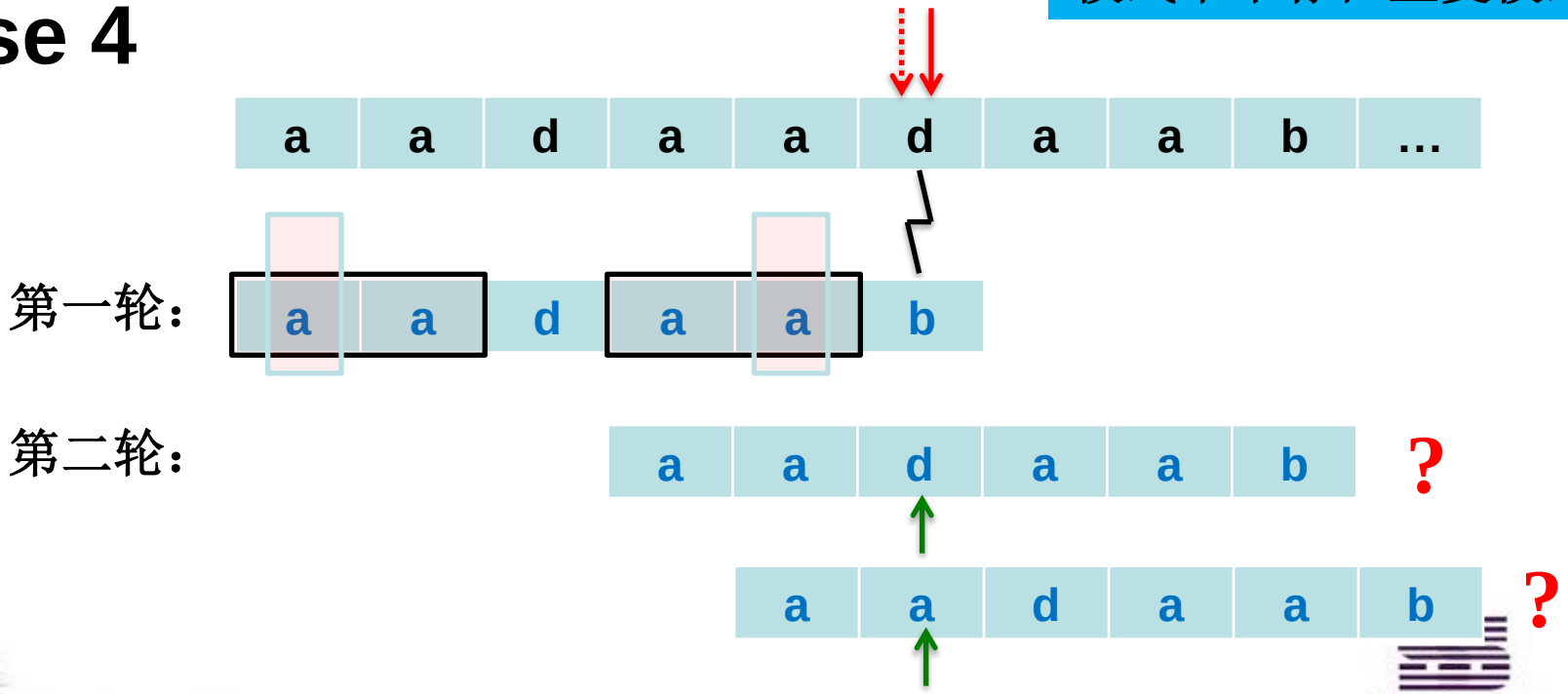
模式串中第一个字符不匹配

- **Case 3**

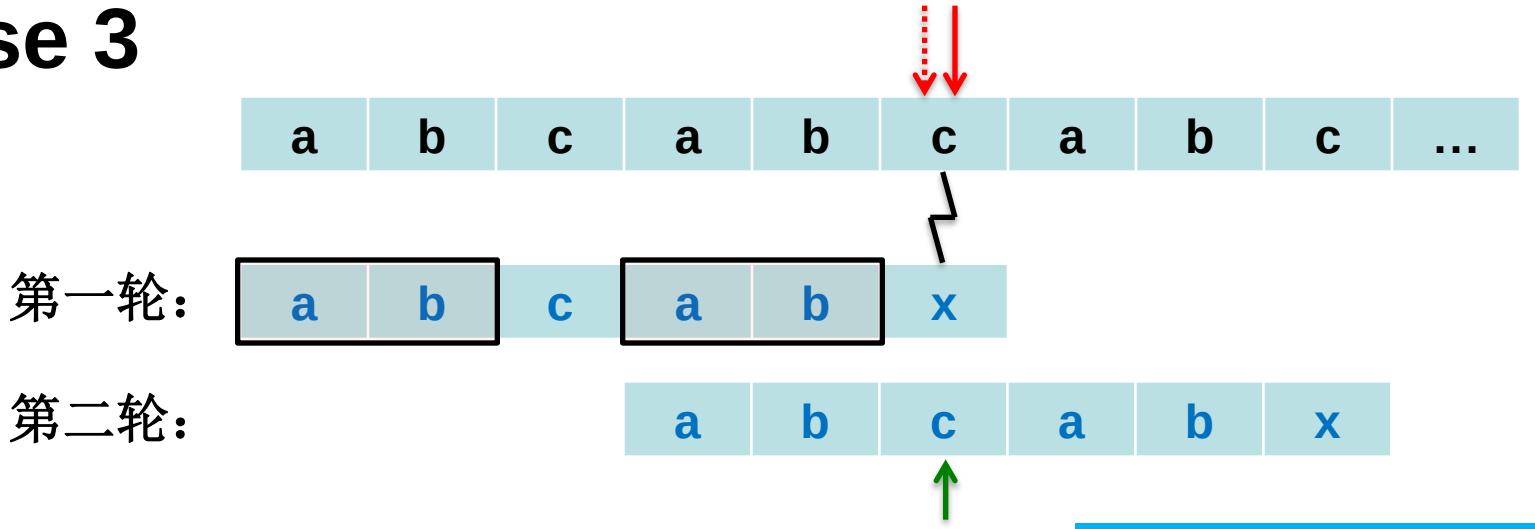


模式串中存在重复模式

- **Case 4**

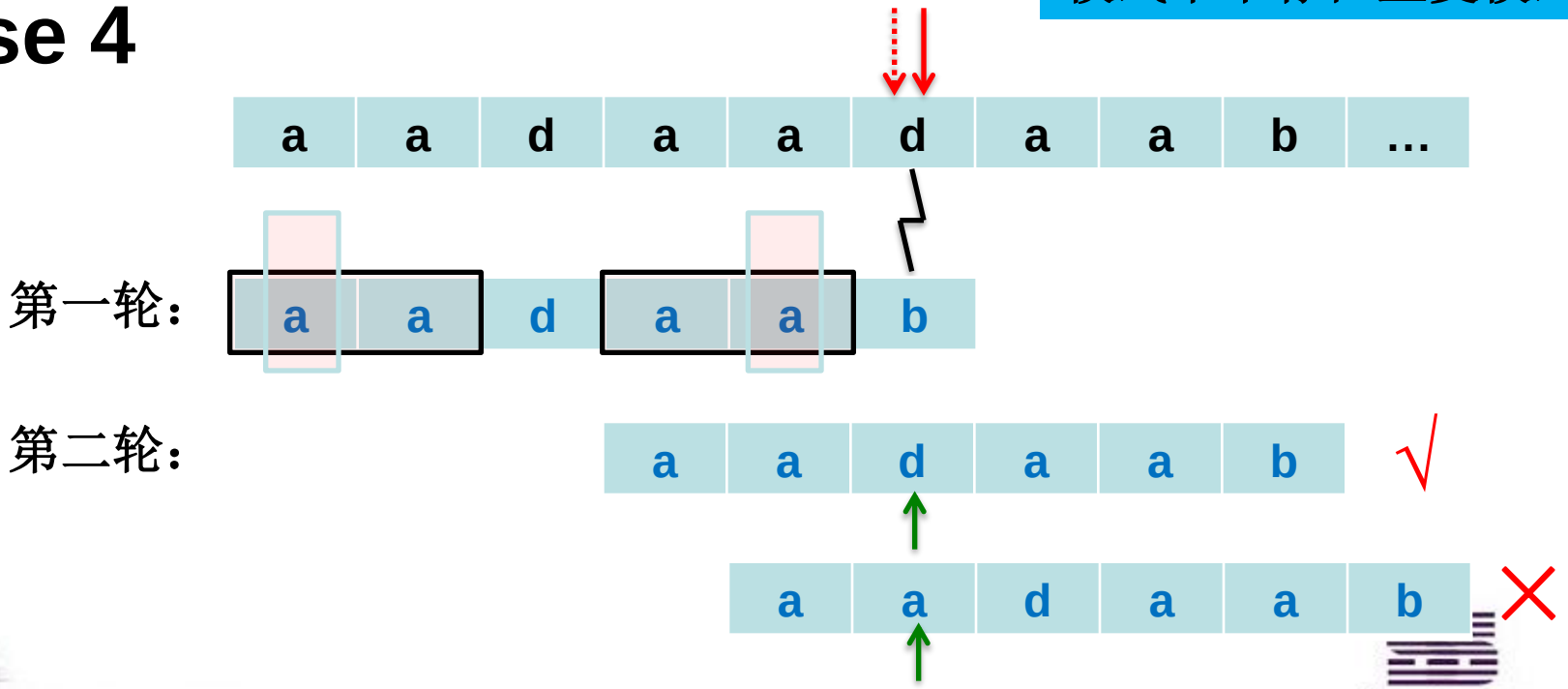


- **Case 3**



模式串中存在重复模式

- **Case 4**

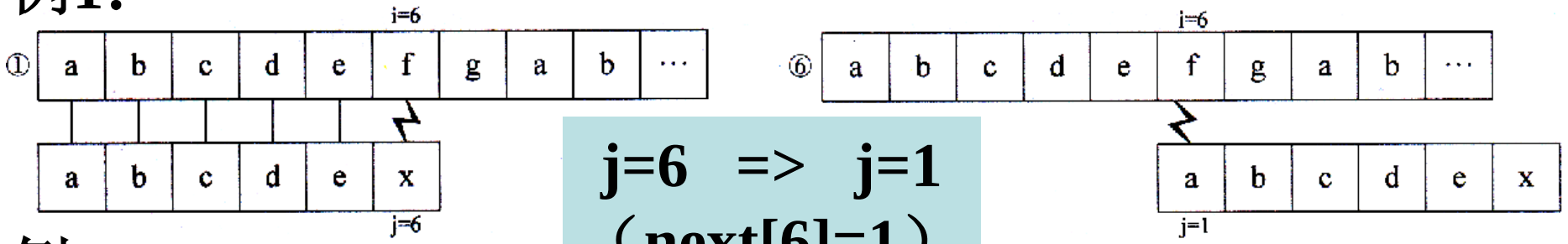


- 如何避免目标串中字符匹配的回溯问题？（利用模式串P自身的重复模式）
  - 假定目标串为T，模式串为P。若T[i]≠P[j]导致当前轮的匹配失败，则下一轮匹配时应该比较T和P中的哪两个字符？
    - 每轮中，T与P都是从左至右逐字符比较。
  - 案例分析总结：
    - 若当前轮匹配在进行第一个字符比较时就失败，那么下一轮应该是比较T[i+1]和P[1]
    - 若P中在当前轮成功匹配的子串的后缀与子串的前缀无重复模式，那么下一轮应该是比较T[i]和P[1]
    - 若P中在当前轮成功匹配的子串的后缀与子串的前缀有重复模式，那么下一轮应该是比较T[i]和P[next[j]]
      - next[j] = ？

next数组

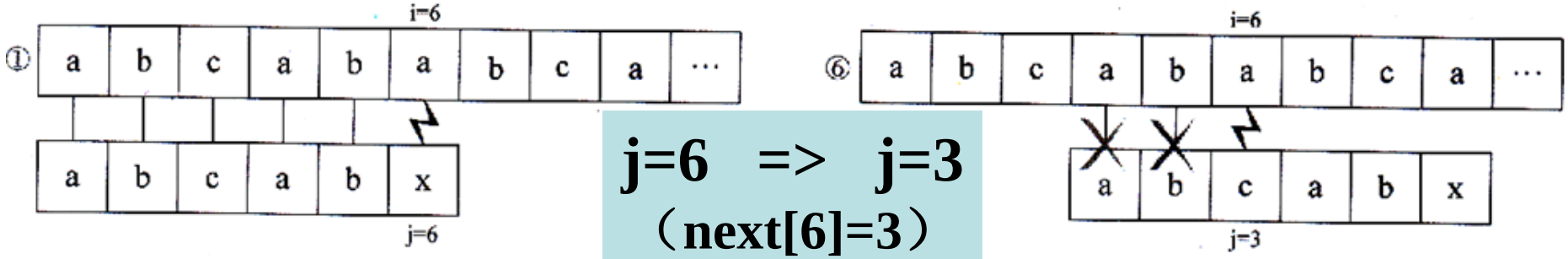
注意：在KMP算法中，模式串P的下标从1开始，而不是从0开始。为什么呢？

例1：



j=6 => j=1
（next[6]=1）

例2：



j=6 => j=3
（next[6]=3）

我们把 T 串各个位置的 j 值的变化定义为一个数组 next，那么 next 的长度就是 T 串的长度。于是我们可以得到下面的函数定义：

$$next[j]=\begin{cases}0, \text{当 } j=1 \text{时}\\ Max\{k \mid 1<k<j, \text{且} 'p_1 \cdots p_{k-1}'='p_{j-k+1} \cdots p_{j-1}'\} \text{当此集合不空时} \quad (\text{隐含} j>2)\\ 1, \text{其他情况}\end{cases}$$

Next[j]表征：下一轮匹配时比较P与T的哪两个字符？

KMP算法的主要思想

- 将模式串P自身的重复规律保存到next数组中

$$next[j]=\begin{cases}0, \text{当 } j=1 \text{时}\\ Max\{k \mid 1<k<j, \text{且} 'p_1 \cdots p_{k-1}'='p_{j-k+1} \cdots p_{j-1}'\} & \text{当此集合不空时}\\ 1, \text{其他情况}\end{cases}$$

- 匹配过程：若某轮匹配失败，则利用next数组分别计算下一轮匹配时目标串和模式串的开始位置
  - 若是T[i]≠P[j]导致当前轮的匹配失败，则按照下列规则开始下一轮匹配：
    - 若next[j] ≠ 0，则将T[i..]与P[next[j]..]匹配；
    - 若next[j]==0，则将T[(i+1)..]与P[1..]匹配 。

next数组的计算示例

1．P="abcdex"（如表 5-7-1 所示）

表 5-7-1

j	123456
模式串	abcdex
next[j]	011111

1）当 j=1 时，next[1]=0；

2）当 j=2 时，j 由 1 到 j－1 就只有字符 "a"，属于其他情况 next[2]=1；

3）当 j=3 时，j 由 1 到 j－1 串是 "ab"，显然 "a" 与 "b" 不相等，属其他情况，next[3]=1；

4）以后同理，所以最终此 T 串的 next[j]为 011111。

2．**P**="abcabx"（如表 5-7-2 所示）

表 5-7-2

j	123456
模式串	abcabx
next[j]	011123

1）当 j=1 时，next[1]=0；

2）当 j=2 时，同上例说明，next[2]=1；

3）当 j=3 时，同上，next[3]=1；

4）当 j=4 时，同上，next[4]=1；

5）当 j=5 时，此时 j 由 1 到 j－1 的串是“a*bca*”，前缀字符“a”与后缀字符“a”相等（前缀用下划线表示，后缀用斜体表示），因此可推算出 k 值为 2（由 ‘$p_1 \cdots p_{k-1}$’ = ‘$p_{j-k+1} \cdots p_{j-1}$’，得到 $p_1=p_4$）因此 next[5]=2；

6）当 j=6 时，j 由 1 到 j－1 的串是“ab*cab*”，由于前缀字符“ab”与后缀“ab”相等，所以 next[6]=3。

3．P="ababaaaba"（如表 5-7-3 所示）

表 5-7-3

j	123456789
模式串	ababaaaba
next[j]	011234223

1）当 j=1 时，next[1]=0；

2）当 j=2 时，同上 next[2]=1；

3）当 j=3 时，同上 next[3]=1；

4）当 j=4 时，j 由 1 到 j－1 的串是“a*b*a”，前缀字符“a”与后缀字符“a”相等，next[4]=2；

5）当 j=5 时，j 由 1 到 j－1 的串是“ab*ab*”，由于前缀字符“ab”与后缀“ab”相等，所以 next[5]=3；

6）当 j=6 时，j 由 1 到 j－1 的串是“ab*aba*”，由于前缀字符“aba”与后缀“aba”相等、所以 next[6]=4；

7）当 j=7 时，j 由 1 到 j－1 的串是“ababa*a*”，由于前缀字符“ab”与后缀“aa”并不相等，只有“a”相等，所以 next[7]=2；

8）当 j=8 时，j 由 1 到 j－1 的串是“ababaa*a*”，只有“a”相等，所以 next[8]=2；

9）当 j=9 时，j 由 1 到 j－1 的串是“abab*aab*”，由于前缀字符“ab”与后缀“ab”相等，所以 next[9]=3。

4．P=“aaaaaaaab”（如表 5-7-4 所示）

表 5-7-4

j	123456789
模式串	aaaaaaaab
next[j]	012345678

1）当 j=1 时，next[1]=0；

2）当 j=2 时，同上 next[2]=1；

3）当 j=3 时，j 由 1 到 j－1 的串是“a*a*”，前缀字符“a”与后缀字符“a”相等，next[3]=2；

4）当 j=4 时，j 由 1 到 j－1 的串是“a*aa*”，由于前缀字符“aa”与后缀“aa”相等，所以 next[4]=3；

5）……

6）当j=9时，j由1到j-1的串是“aaaaaaaa”,由于前缀“aaaaaaa”与后缀“aaaaaaa”相等，所以next[9]=8.

next数组的应用

- 例：在目标串“**aaaaaaacaaaaaaaab**”中查找模式串“**aaaaaaaab**”的位置。
- **Step1**：计算模式串的**next**数组：

j	123456789
模式串	aaaaaaaab
next[j]	012345678

- **Step2**：匹配过程。（**9**轮）

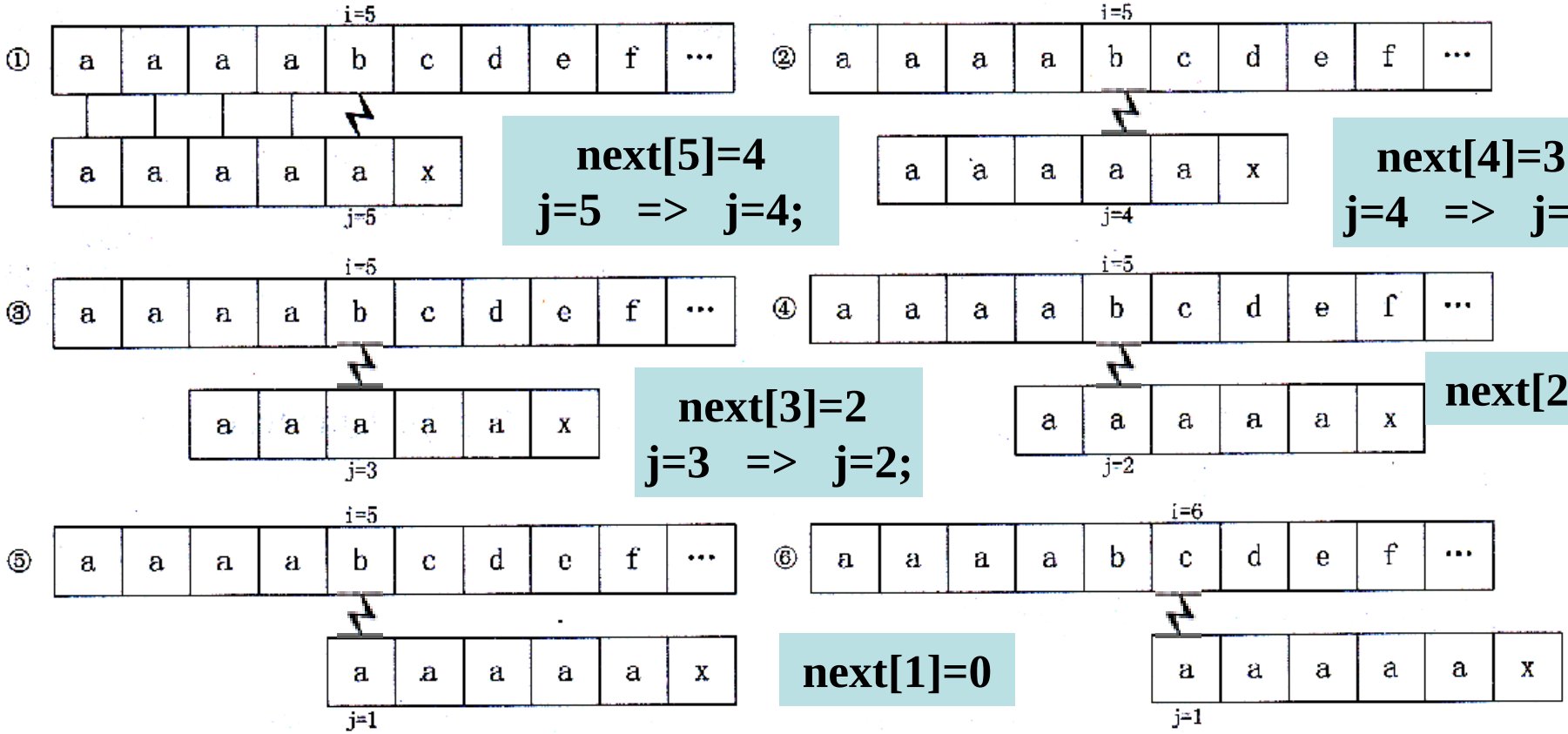
KMP算法的实现

```
class Solution {
public:
    int strStr(string s, string p) {
        int n,m;
        n = s.size();
        m = p.size();
        vector<int> next;
        int i=0,j=-1;
        next.push_back(-1);
        while(i < m){
            if(j==-1 or p[i] == p[j]){
                i++;
                j++;
                next.push_back(j);
            }else {
                j = next[j];
            }
    }}};
```

```
i=0;j=0;
while( i < n and j < m){
    if(j==-1 ||s[i] == p[j]){
        i++;
        j++;
    }else{ j = next[j]; }
}

return (j==m) ? i-m : -1;
// return 0 ;
```

Q: next数组还有改进空间吗？



步骤②③④⑤都是多余的！

我们发现，当中的②③④⑤步骤，其实是多余的判断。由于 T 串的第二、三、四、五位置的字符都与首位的“a”相等，那么可以用首位 next[1]的值去取代与它相等的字符后续 next[j]的值，这是个很好的办法。因此我们对求 next 函数进行了改良。

KMP算法的改进

- 什么情况下有改进的空间？
  - 假设**T[i] ≠P[j]**导致失配 。若**P[j]==P[next[j]]**，此时若向右移动模式串**P**，将**T[i]**与**P[next[j]]**对齐进行比较必然是无意义的，因为此时**T[i]**必定**≠P[next[j]]**。
- 如何改进？用**nextval**数组代替**next**数组。
  - **nextval[1]=0;**
  - **for(j>1;j<=n;j++)**
    - 若**P[j]==P[next[j]]**，则**nextval[j]=nextval[next[j]];**
    - 若**P[j]≠P[next[j]]**，则**nextval[j]=next[j];**
- 若某轮匹配失败，则利用**nextval**数组计算下一轮匹配时的目标串和模式串的开始位置（类似**next**数组的应用）

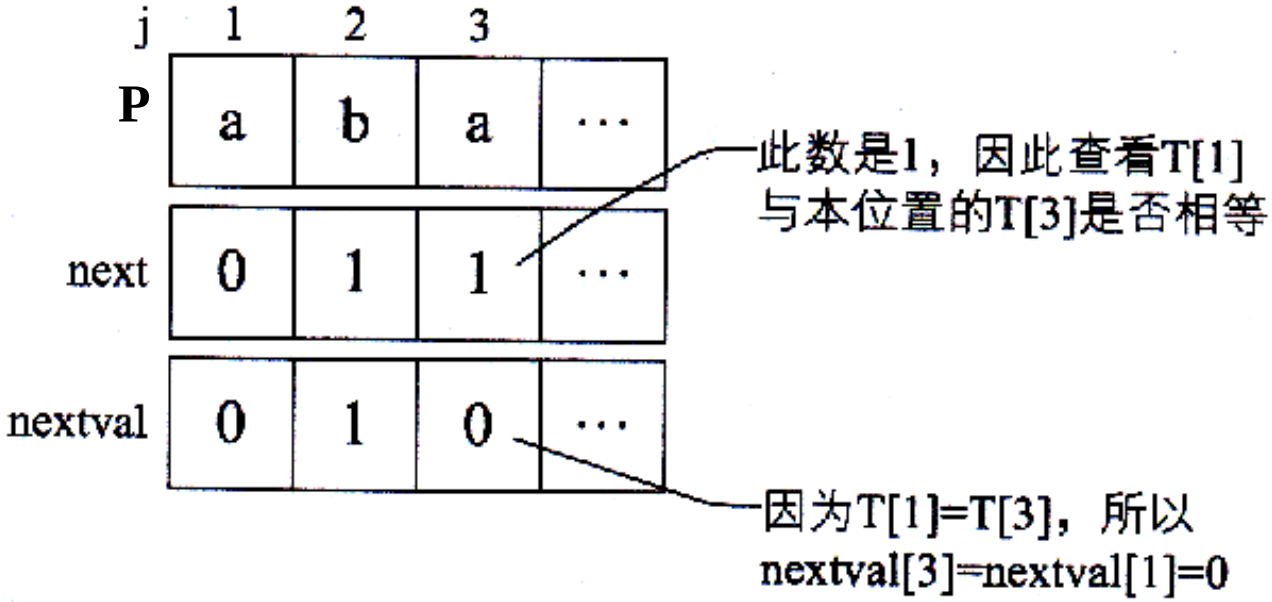

j 1 2 3
P a b a …
next 0 1 1 …
nextval 0 1 0 …
此数是1，因此查看T[1]
与本位置的T[3]是否相等
因为T[1]=T[3]，所以
nextval[3]=nextval[1]=0

图 5-7-7

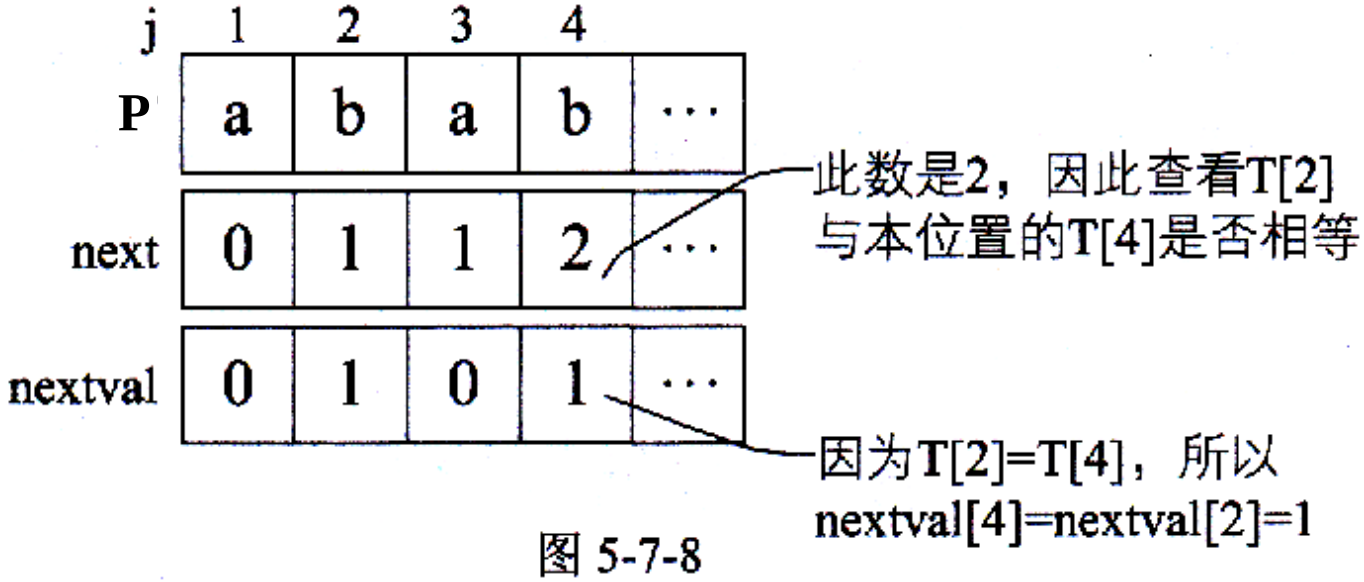

j 1 2 3 4
P a b a b …
next 0 1 1 2 …
nextval 0 1 0 1 …
此数是2，因此查看T[2]
与本位置的T[4]是否相等
因为T[2]=T[4]，所以
nextval[4]=nextval[2]=1

图 5-7-8

nextval数组的计算示例

1．P="ababaaaba"（如表 5-7-5 所示）

表 5-7-5

j	123456789
模式串	ababaaaba
next[j]	011234223
nextval[j]	010104210

先算出 next 数组的值分别为 001234223，然后再分别判断。

1）当 j=1 时，nextval[1]=0；

2）当 j=2 时，因第二位字符“b”的 next 值是 1，而第一位就是“a”，它们不相等，所以 nextval[2]=next[2]=1，维持原值。

3）当 j=3 时，因为第三位字符“a”的 next 值为 1，所以与第一位的“a”比较得知它们相等，所以 nextval[3]=nextval[1]=0；如图 5-7-7 所示。

4）当 j=4 时，第四位的字符“b”next 值为 2，所以与第二位的“b”相比较得到结果是相等，因此 nextval[4]=nextval[2]=1；如图 5-7-8 所示。

5）当 j=5 时，next 值为 3，第五个字符“a”与第三个字符“a”相等，因此 nextval[5]=nextval[3]=0；

6）当 j=6 时，next 值为 4，第六个字符“a”与第四个字符“b”不相等，因此 nextval[6]=4；

7）当 j=7 时，next 值为 2，第七个字符“a”与第二个字符“b”不相等，因此 nextval[7]=2；

8）当 j=8 时，next 值为 2，第八个字符“b”与第二个字符“b”相等，因此 nextval[8]=nextval[2]=1；

9）当 j=9 时，next 值为 3，第九个字符“a”与第三个字符“a”相等，因此 nextval[9]=nextval[3]=1。

2．P=“aaaaaaaab”（如表 5-7-6）

表 5-7-6

j	123456789
模式串 T	aaaaaaaab
next[j]	012345678
nextval[j]	000000008

先算出 next 数组的值分别为 012345678，然后再分别判断。

1）当 j=1 时，nextval[1]=0；

2）当 j=2 时，next 值为 1，第二个字符与第一个字符相等，所以 nextval[2]=nextval[1]=0；

3）同样的道理，其后都为 0……；

4）当 j=9 时，next 值为 8，第九个字符“b”与第八个字符“a”不相等，所以 nextval[9]=8。

例题：计算nextval数组

- 1.

j	123456
模式串	abcabx
next[j]	011123

- 2.

j	123456
模式串	abcdex
next[j]	011111

- 3.

j	123456789
模式串	abababaaba
next[j]	011234223

nextval数组的应用

- 例：在目标串“**aaaaaaacaaaaaaaab**”中查找模式串“**aaaaaaaab**”的位置。
- **Step1**：计算模式串的**next**数组：

j	123456789
模式串	aaaaaaaab
next[j]	012345678
nextval[j]	000000008

- **Step2**：匹配过程。（**2**轮）

KMP算法的性能 vs. 朴素算法

- 近似时间复杂度为O(n+m),其中O(n)表示比较的时间，O(m)表示计算next数组的时间.
  - 目标串中不存在回溯情况
  - 目标串中的每个字符，会比较1~2次；
- 注意：KMP算法中，字符串的下标从1开始，而不是从0开始
- 仅当模式串与目标串之间存在许多“部分匹配”情况下，KMP算法才比朴素模式匹配算法更具优势。
- 若每轮中模式串与目标串之间的不匹配都发生在模式串的第一个字符处，则KMP算法会退化到朴素模式匹配算法。
  - 因为nextval[1]=0;

3. Boyer-Moore查找

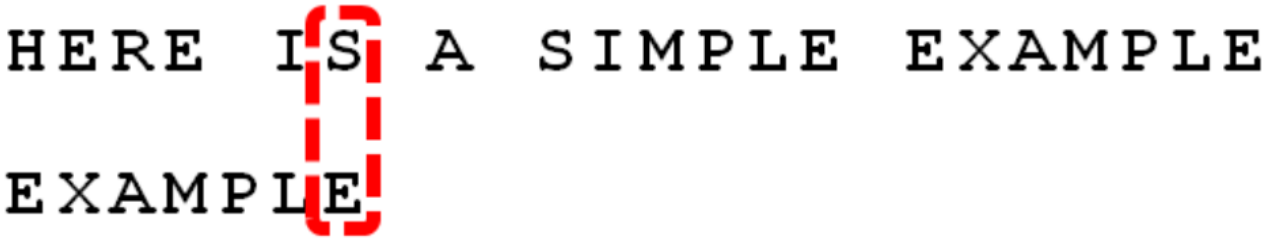
- 是字符串查找算法中最著名的两个算法之一

- **BM算法：**
  - 是一种精确字符串匹配算法（区别于模糊匹配）。
  - 每轮匹配过程中，字符的匹配方向：采用从右向左进行字符的匹配比较。（与KMP算法的主要区别）
    - 若某轮匹配失败，则移动模式串，与目标串的下一轮开始匹配位置进行右对齐，然后开始当前轮的从右至左的逐字符匹配。
    - 故，BM算法中的关键问题是，如何确定目标串中的下一轮匹配的开始位置？即，如何确定目标串中查找指针的移动距离？
  - 采用启发式方法：无需检查目标字符串中的所有字符
    - 利用P中的重复模式 和 T中的失配字符

Boyer-Moore示例

假定字符串为"HERE IS A SIMPLE EXAMPLE"，搜索词为"EXAMPLE"。

1



"S"与"E"不匹配。这时，"S"就被称为"坏字符“, 即不匹配的字符"S"不包含在搜索词"EXAMPLE"之中，

2

模式串移动了7位

Boyer-Moore示例

2

依然从尾部开始比较，发现"P"与"E"不匹配，所以"P"是"坏字符"。但，"P"包含在搜索词"EXAMPLE"之中。所以，将搜索词后移两位，两个"P"对齐。

3

模式串移动了2位

Boyer-Moore示例

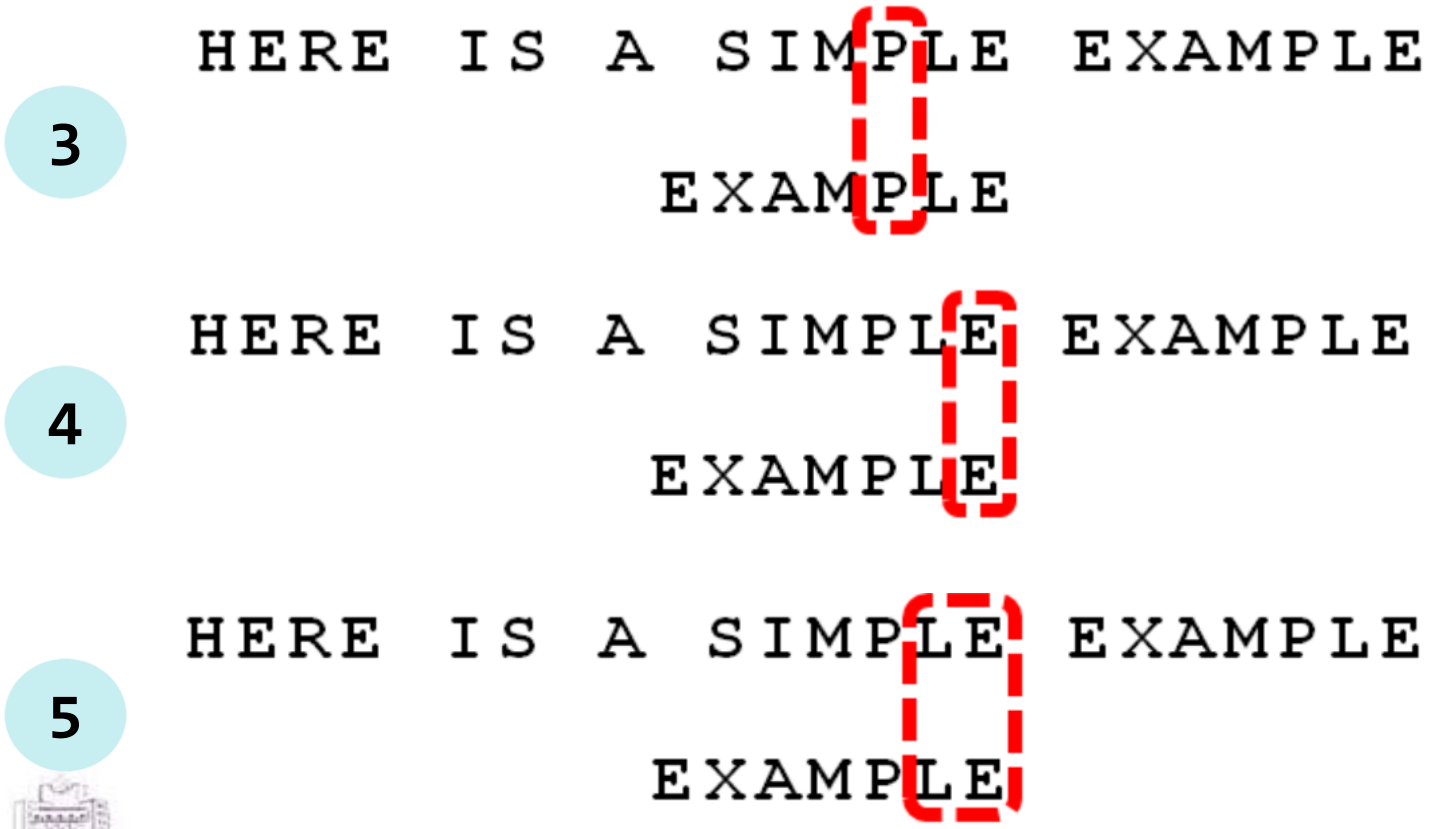
坏字符规则：当文本串中的某个字符跟模式串的某个字符不匹配时，我们称文本串中的这个失配字符为坏字符。

此时**模式串需要向右移动，**

移动的位数 = 坏字符在模式串中的位置 - 坏字符在模式串中最右出现的位置。
此外，如果"坏字符"不包含在模式串之中，则最右出现位置为-1。

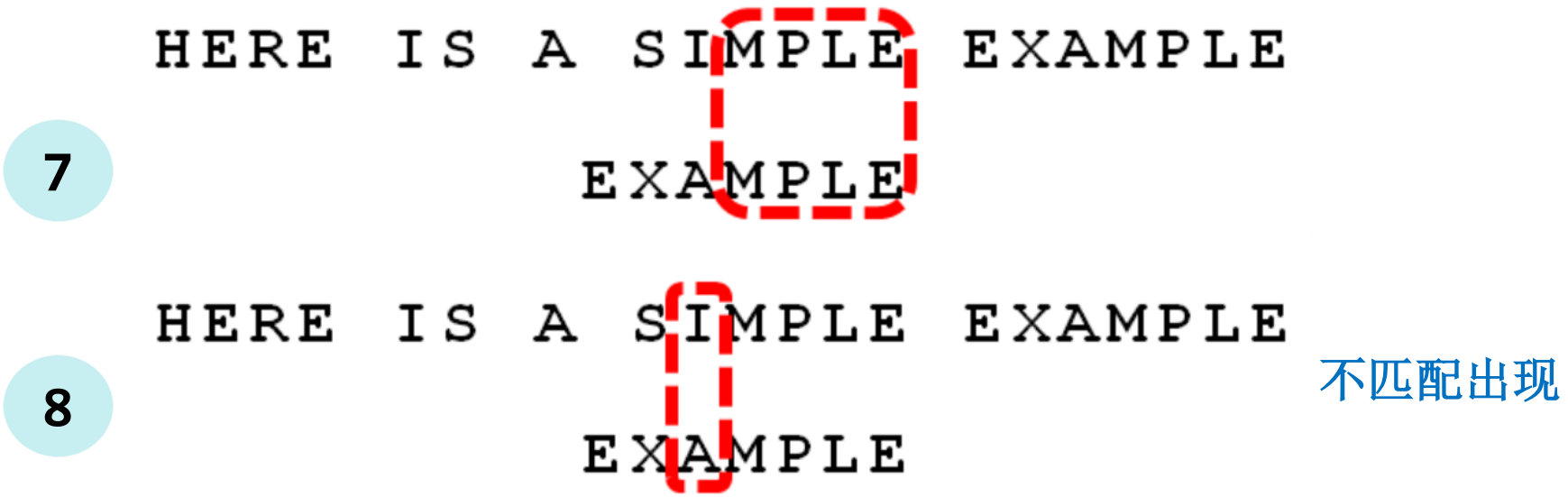
后移位数 = 坏字符的位置 - 搜索词中的上一次出现位置

Boyer-Moore示例



仍然从最右侧开始比较

Boyer-Moore示例

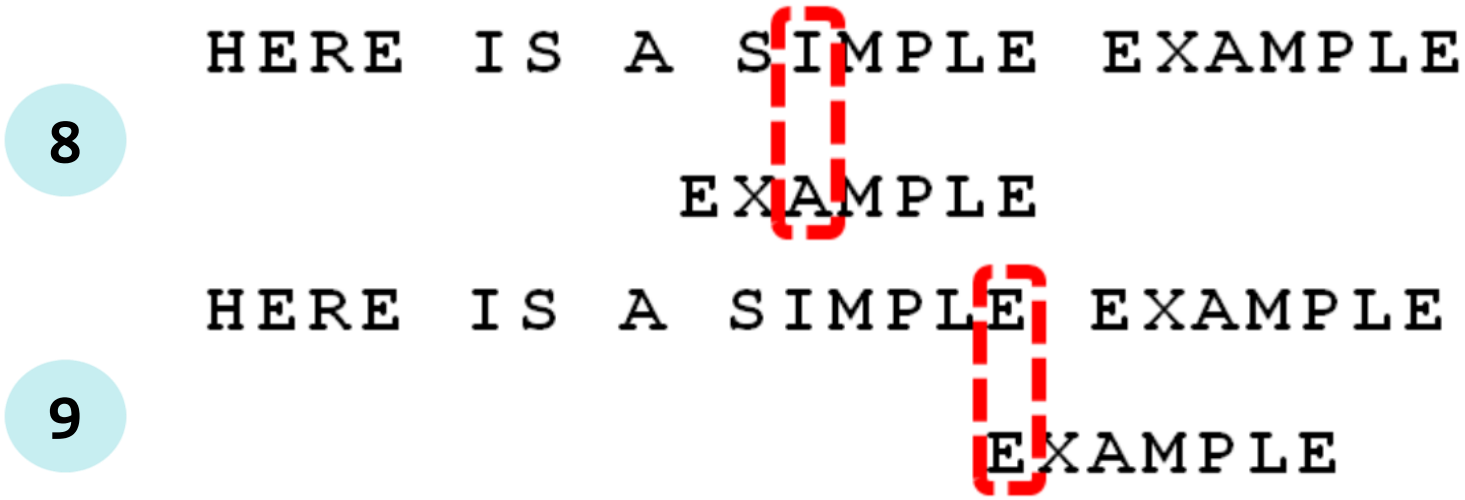


坏字符规则，模式串向右3位（ 2- (-1)）

如考虑已经匹配的“MPLE”后缀，是否有更好的方式

“MPLE”、“PLE”，“LE”，“ E”

Boyer-Moore示例

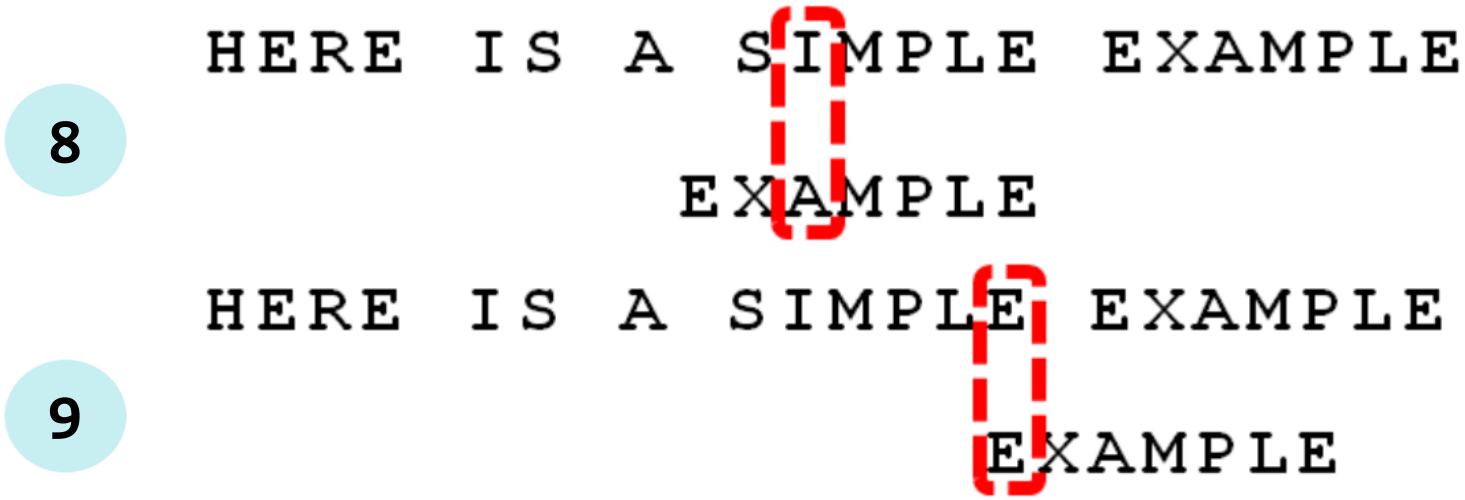


好后缀"的位置以最后一个字符为准。
若"好后缀"在P只出现一次，则它上一次出现位置为 -1

所有的"好后缀"（MPLE、PLE、LE、E）之中，只有"E"在"EXAMPLE"还出现在头部，所以后移 6 - 0 = 6位。

选择最小的移动距离（而非7位）

Boyer-Moore示例



好后缀规则：当字符失配时，

模式串后移位数

= 好后缀在模式串中的位置 - 好后缀在模式串上一次出现的位置

如果好后缀在模式串中没有再次出现，则为-1。

Boyer-Moore示例

9

```
HERE IS A SIMPLE EXAMPLE
                   EXAMPLE
```

10

```
HERE IS A SIMPLE EXAMPLE
                   EXAMPLE
```

- BM算法中的关键问题：当匹配失败时，如何确定目标串中的下一轮匹配的开始位置？即，如何决定目标串中指针向右跳跃的距离dist？
  - 假设出现失配时，T[i]≠P[k]。则 此时坏字符为x(=T[i])，好后缀P'=P[(k+1) … (len(P)-1)]（好后缀，是已匹配的部分字符串）
  - 如何确定目标串中查找指针的移动距离dist[i]？
  - 采用2种启发式方法：无需检查目标串中的所有字符即可查找到是否存在匹配子串。
- 启发式方法#1： 跳过字符（“坏字符”规则）
  - CharJump[x]：依据T中的坏字符x，计算T中查找指针的跳跃距离。
- 启发式方法#2： 重复模式（“好后缀”规则）
  - MatchJump[k]：依据P中的失配位置k，计算T中查找指针的跳跃距离。
- 例1：在目标串“oaks from acorns grow”中查找模式串“corn”的位置。（模式串中无重复模式）
- 例2：在目标串“he went door to door”中查找模式串“door to door”的位置。（模式串具有重复模式）

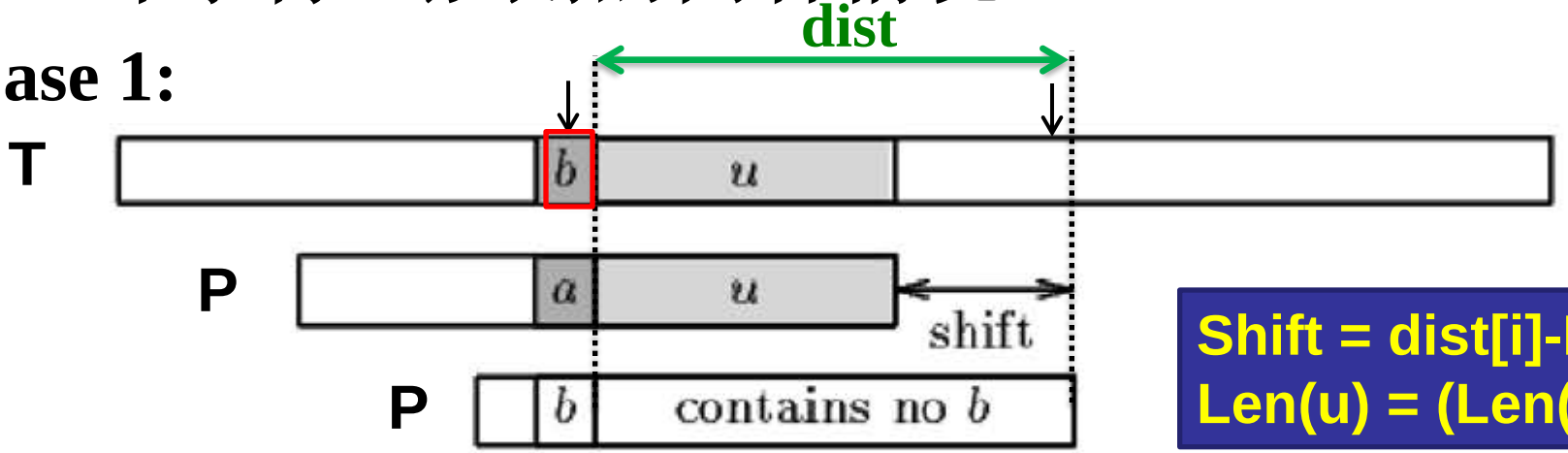
BM算法的基本思想

- 设目标串T，模式串为P。
- **Step1**：对于模式串P，计算CharJump[x]和MatchJump[k]。
- **Step2**: 将T与P的第一个字符对齐。
- **Step3**: T与P进行从右向左的逐字符比较，直至找到一个不匹配字符或者P中所有字符都匹配成功。
- **Step4**: 若出现失配，即存在T[i]≠P[k]，此时坏字符x=T[i]，好后缀P'=P[(k+1) … (len(P)-1)]。按如下规则计算 目标串T中查找指针向右移动dist[i]：
  - 若此时T与P已有部分字符匹配（即存在“好后缀”）时，BM算法将采用2种启发式方法（即坏字符规则 和好后缀规则 ），计算dist[i] =max(CharJump[x], MatchJump[k])。
  - 若不存在“好后缀”，则必定是在模式串P的最后一个字符处出现失配。此时应采用启发式方法#1： 跳过字符规则（“坏字符”规则），计算设置dist[i] =CharJump[x]。
- **Step5**: 若(i+dist[i])≤Len(T)-1，则移动模式字符串P，使之与T[i+dist[i]]右对齐，重复Step3；否则，认为T中不存在与P匹配的子串，返回匹配失败。

“坏字符”规则的两种情况：

dist[i]=CharJump[‘b’]

Case 1:



Shift = dist[i]-Len(u),
Len(u) = (Len(P)-1-k)

Figure 3. The bad-character shift, b occurs in P.

Case 2:

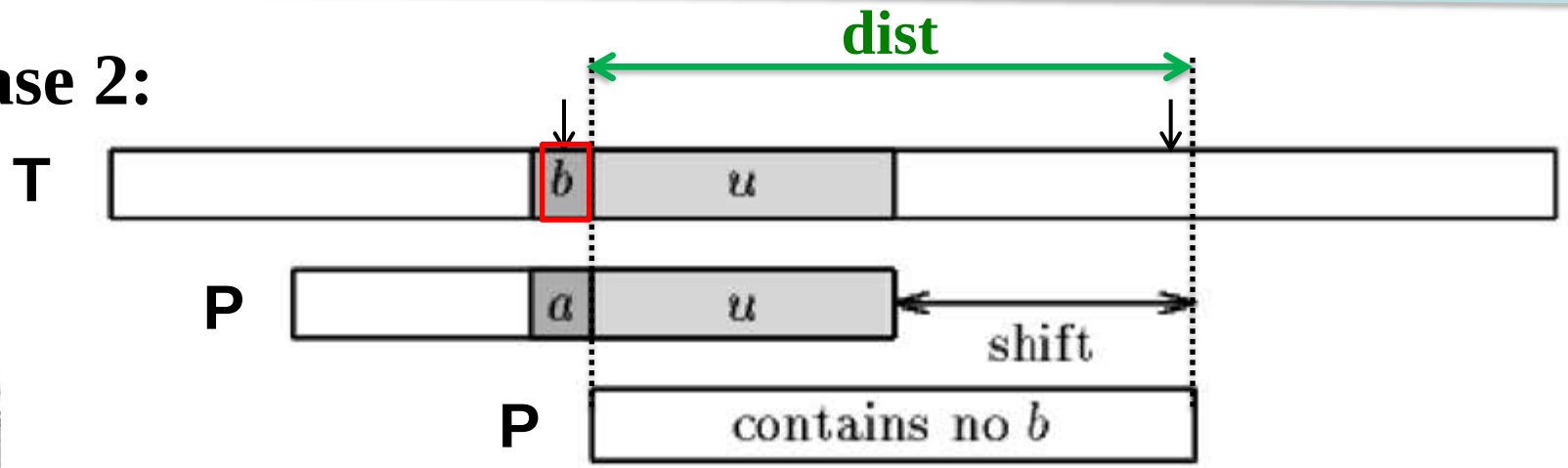
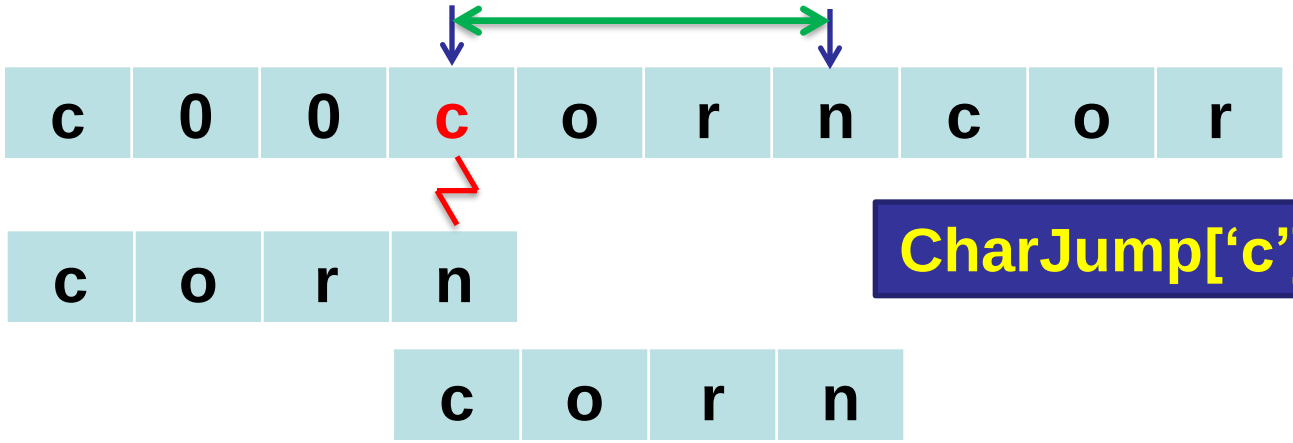


Figure 4. The bad-character shift, b does not occur in P.

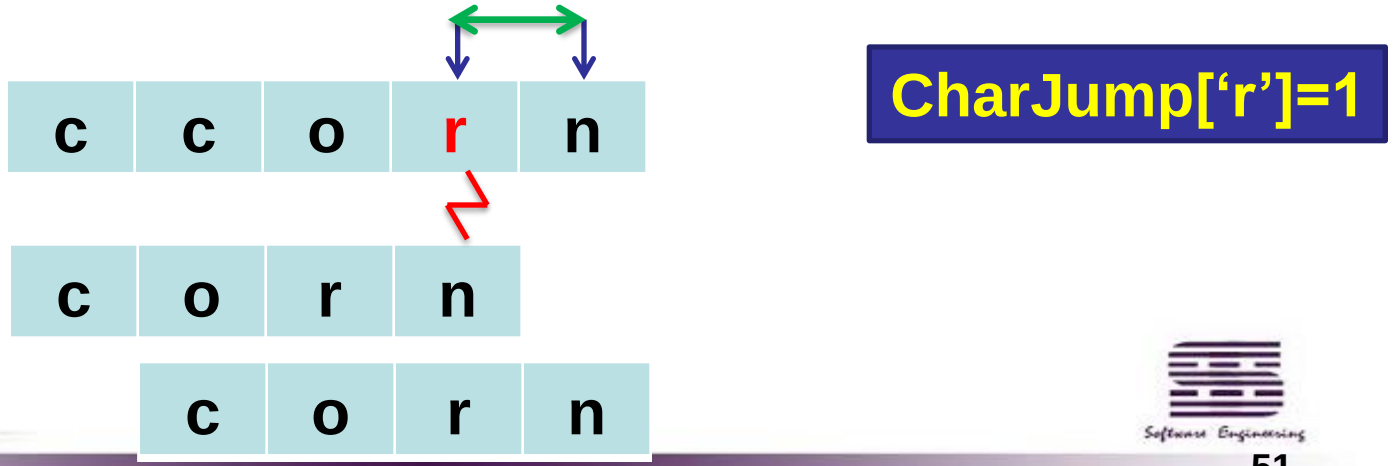
CharJump数组

例1：模式串P='corn'

Case1:

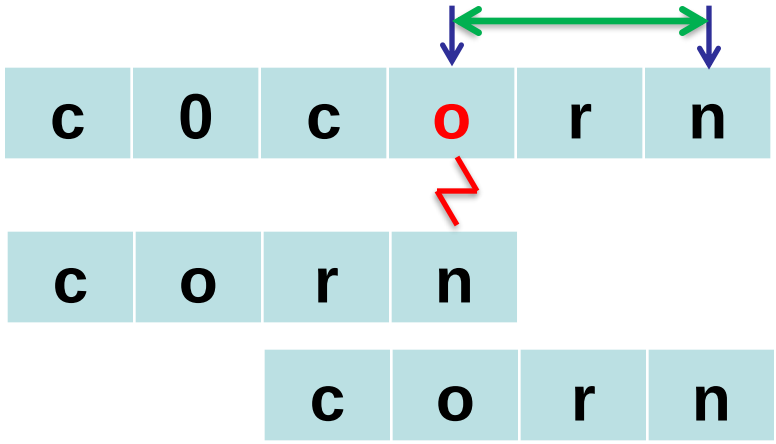


Case2:



CharJump数组

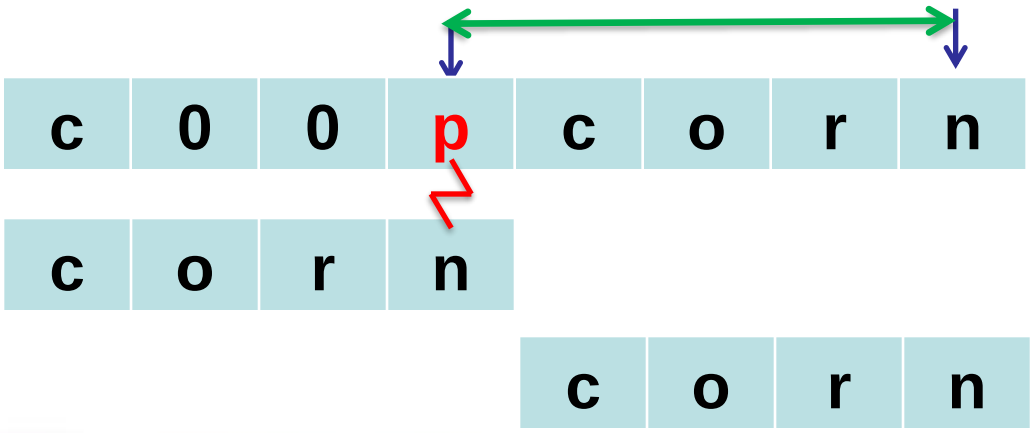
Case3:



CharJump['o']=2

Case4:

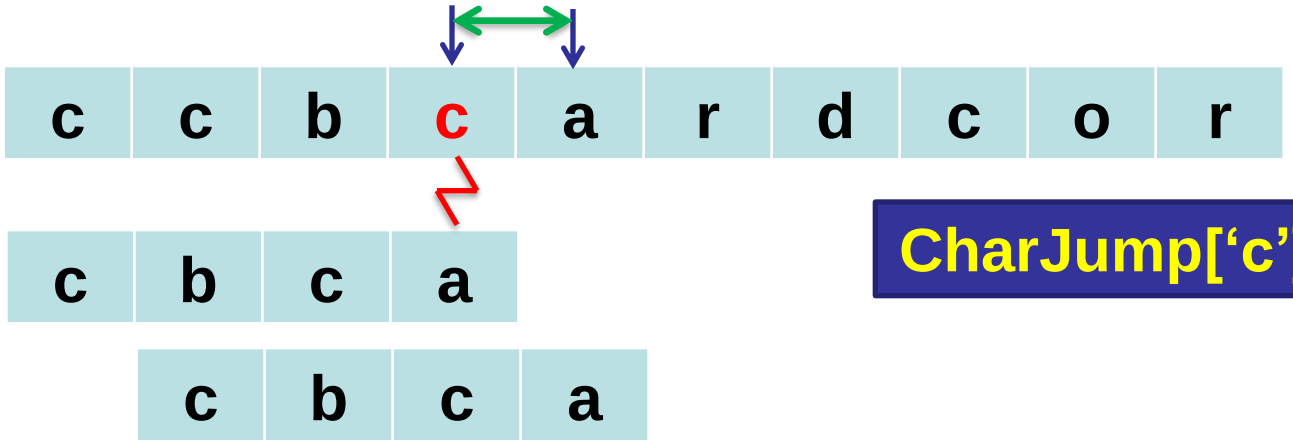
CharJump['p']=4



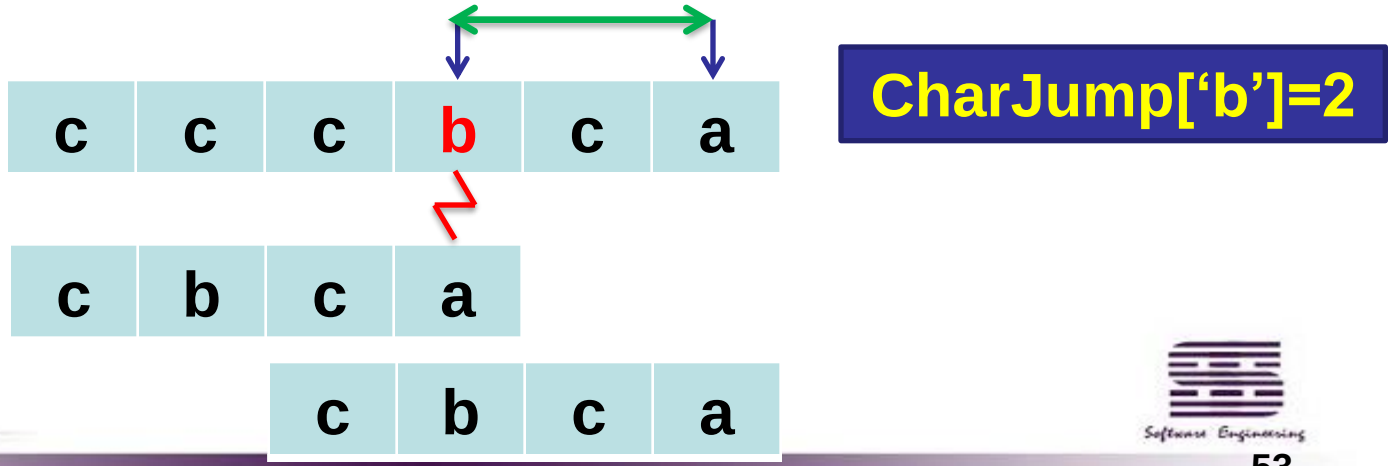
CharJump数组

例2：模式串P='cbca'

Case1:



Case2:



总结：“坏字符”规则

- “坏字符”规则：假设坏字符为x=T[i]，
  - Case1：如果坏字符x在模式P中没有出现，那么从字符x开始的Len(P)个字符显然不可能与P匹配成功，因此，可以使目标串中查找指针直接跳过Len(P) 个字符。
  - Case2：如果坏字符x在模式P中出现(假设P[j]==x)，则将目标串中查找指针移动CharJump[x]，使得下一轮匹配中将字符P[j]与坏字符x进行对齐。
- 关键问题：计算CharJump[x]
  - 若x在P中出现，假设p[j]==x，则CharJump[x]=Len(P)-max(j)-1;
  - 否则， CharJump[x]=Len(P);

  例：若P为“aedabc”，此时模式串中有重复字符'a'. CharJump('a')=2;
- 注意：模式串P右移距离shift=CharJump[x] - (Len(P)-1-k)个字符。

“坏字符”数组

```
#define AlphabetSize (UCHAR_MAX + 1)    /* For portability */
char *BoyerMoore ( const char * String, /* search for this */
                   const char * Text,   /* ...in this text */
                   size_t TextLen )     /* ...up to here.  */
{
   /* array of character mismatch offsets */
   unsigned CharJump[AlphabetSize];
   size_t   PatLen;
   unsigned u;

   /* Set up and initialize arrays */
   PatLen = strlen ( String );

   /* Heuristic #1 -- simple char mismatch jumps ... */
   memset ( CharJump, 0, AlphabetSize * sizeof(unsigned) );
   for ( u = 0 ; u < PatLen; u++ )
      CharJump[((unsigned char) String[u])]
               = PatLen - u - 1;
   ……
}
```

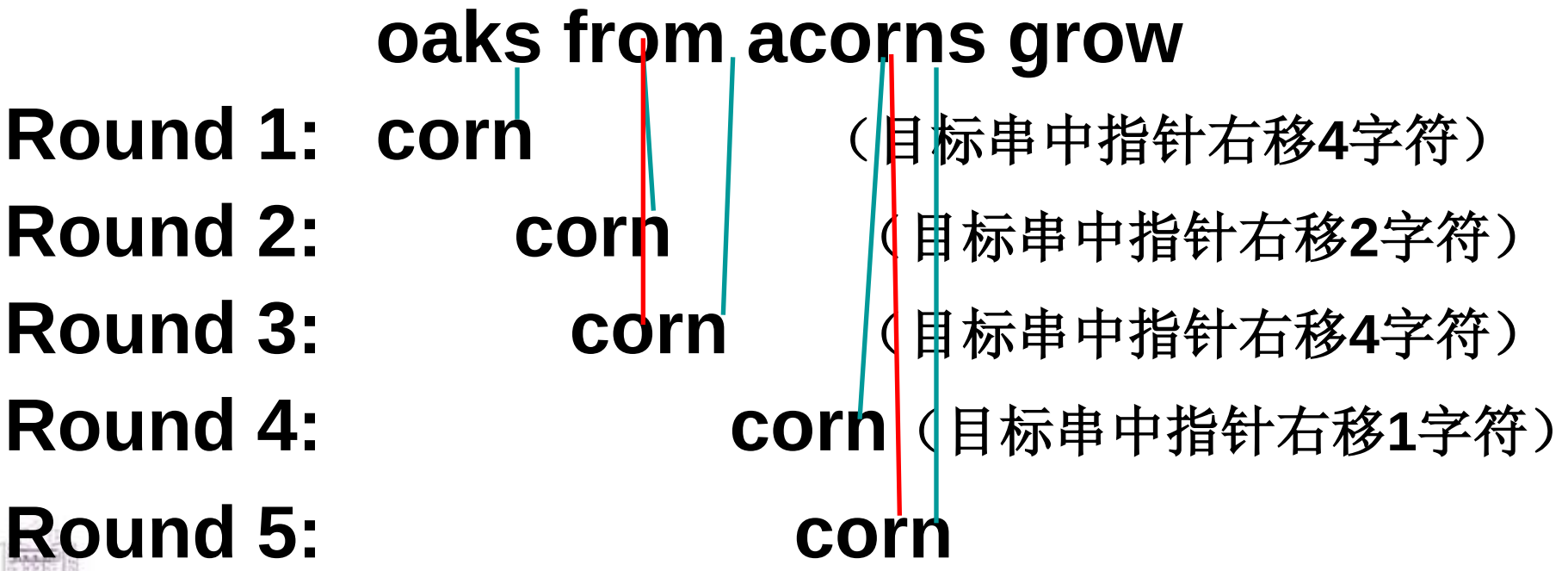
例：若String为"aedabc"，CharJump('a')=2；

练习：计算CharJump

- 若模式串为"abcd"，则
  - CharJump['a']=3
  - CharJump['b']=2
  - CharJump['c']=1
  - CharJump['d']=0
  - CharJump[all others]=4
- 若模式串为"abcadb"，则
  - CharJump['a']=2
  - CharJump['b']=0
  - CharJump['c']=3
  - CharJump['d']=1
  - CharJump[all others]=6

示例：“坏字符”规则的应用

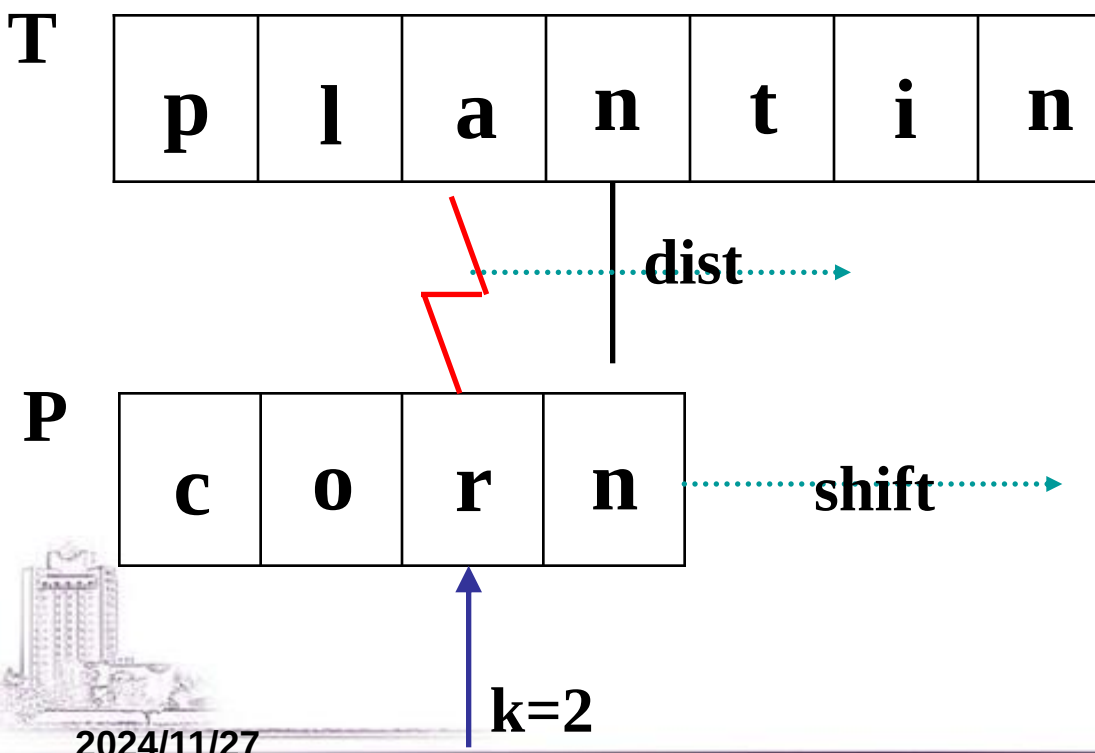
- 例：在“**oaks from acorns grow**”中查找模式串“**corn**”的位置。



CharJump['c']=3; CharJump['o']=2; CharJump['r']=1;
CharJump['n']=0; CharJump[all others]=4

课堂练习：“坏字符”规则的应用

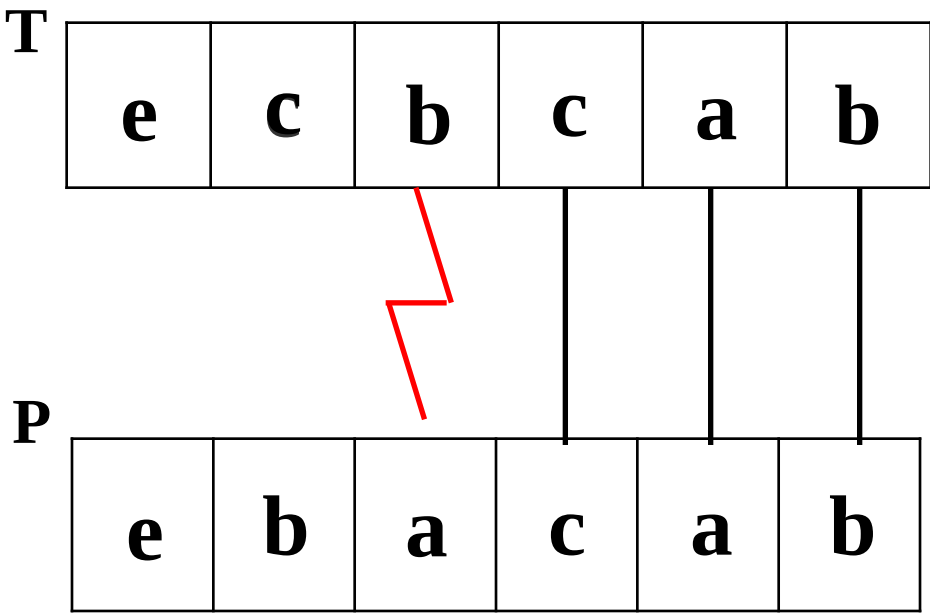
1、在“planting new”中查找子串“corn”的位置。




```
CharJump['c']=3;
CharJump['o']=2;
CharJump['r']=1;
CharJump['n']=0;
CharJump[all
others]=4
```

Round 1

2、在“ecbcabebacab”中查找子串“ebacab”的位置。




```
CharJump['e']=5;
CharJump['c']=2;
CharJump['a']=1;
CharJump['b']=0;
CharJump[all others]=6
```

设文本串T，模式串为P。

首先：将T与P进行左对齐，然后进行从右向左的逐字符比较；直至匹配成功或者找到坏字符（第一个不匹配的字符）

再：

Case1：如果坏字符（记为x）在模式P中没有出现，那么从字符x开始的Len(P)个字符显然不可能与P匹配成功，因此，可以直接跳过Len(P) 个字符以便开始下一轮匹配。

Case2：如果x在模式P中出现，则下一轮匹配开始时按照x在模式串中最右出现的位置进行对齐。（考虑到x可能在p中多次出现）

2、在“ecbcabebacab”中查找子串“ebacab”的位置。

T

e	c	b	c	a	b	e	b	a	c	a	b

移动距离=0（此时指针移动距离是无效值）

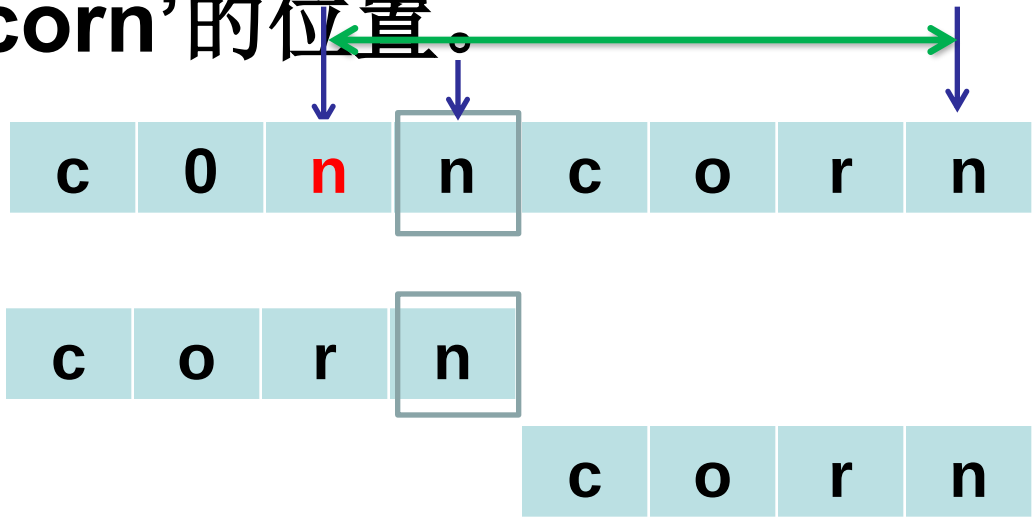
存在好后缀，可以利用“好后缀”规则进行修正，让指针动起来！

P

e	b	a	c	a	b

CharJump(‘b’)=0

3、在目标串‘conncorn’中查找模式串P=‘corn’的位置。



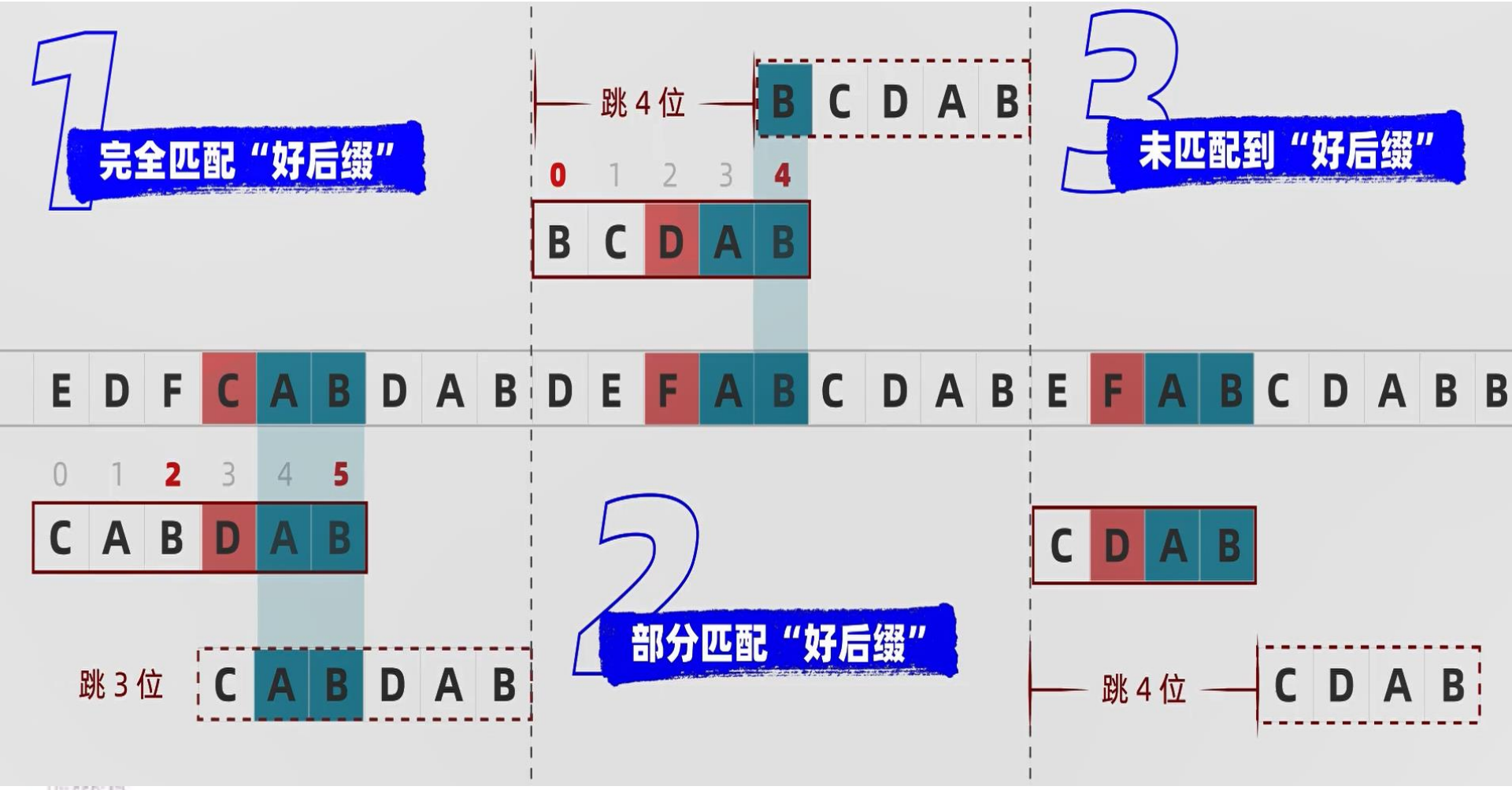
CharJump[‘n’]=0

若利用“坏字符规则，目标串中查找指针的移动距离dist=0，那么此时失配处必定不是在模式串的最后一个字符处，即此时必定存在“好后缀”。

怎么让目标串中查找指针动起来？

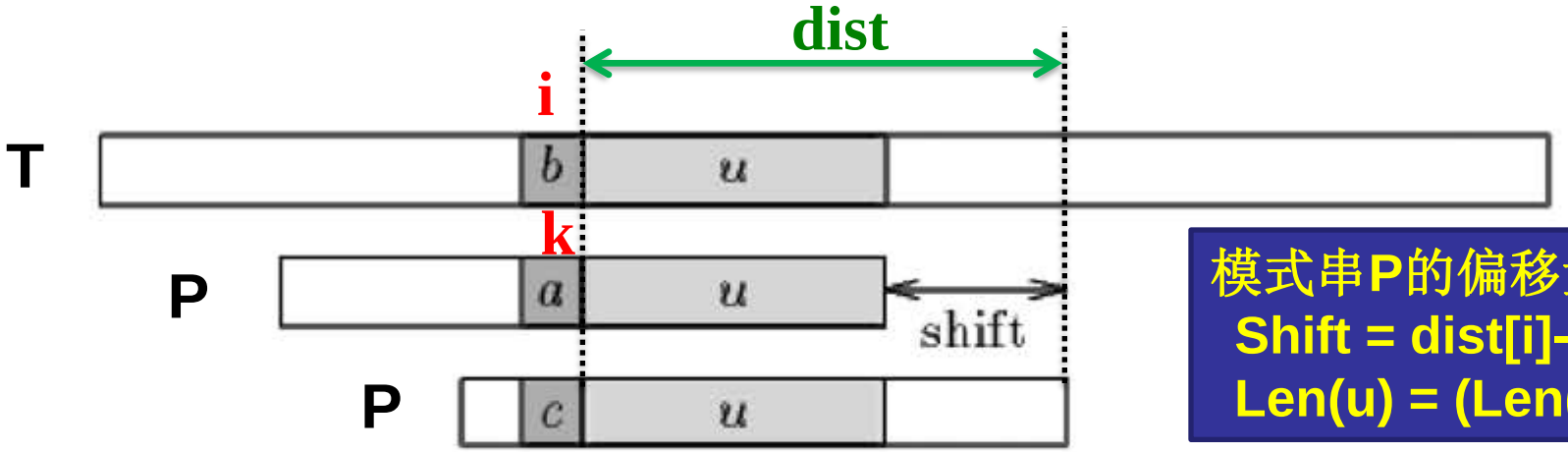
利用“好后缀”规则

“好后缀”规则的三种情况：


1
完全匹配“好后缀”
跳 4 位
B C D A B
0 1 2 3 4
B C D A B
3
未匹配到“好后缀”
E D F C A B D A B
D E F A B C D A B
E F A B C D A B B
0 1 2 3 4 5
C A B D A B
2
部分匹配“好后缀”
C D A B
跳 3 位
C A B D A B
跳 4 位
C D A B

“好后缀”规则的三种情况：

dist[i]=MatchJump[k]



模式串P的偏移量
Shift = dist[i]-Len(u),
Len(u) = (Len(P)-1-k)

Figure 1. The good-suffix shift, *u* re-occurs preceded by a character *c* different from *a*.

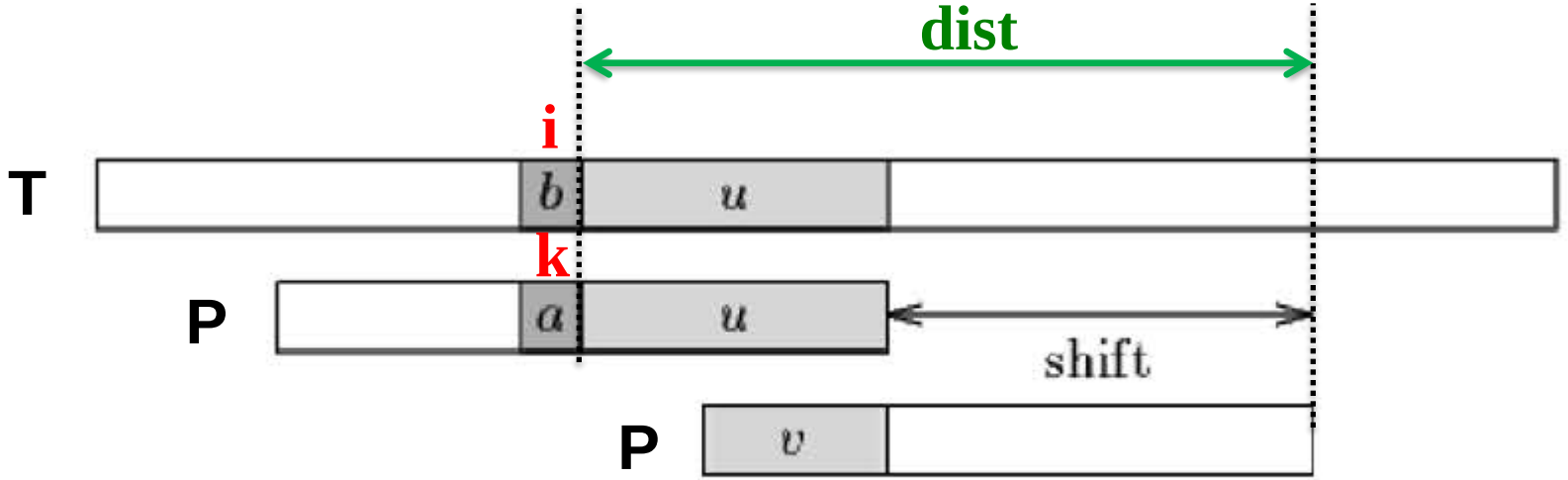


Figure 2. The good-suffix shift, only a suffix of *u* re-occurs in P.

dist[i]= MatchJump[k]

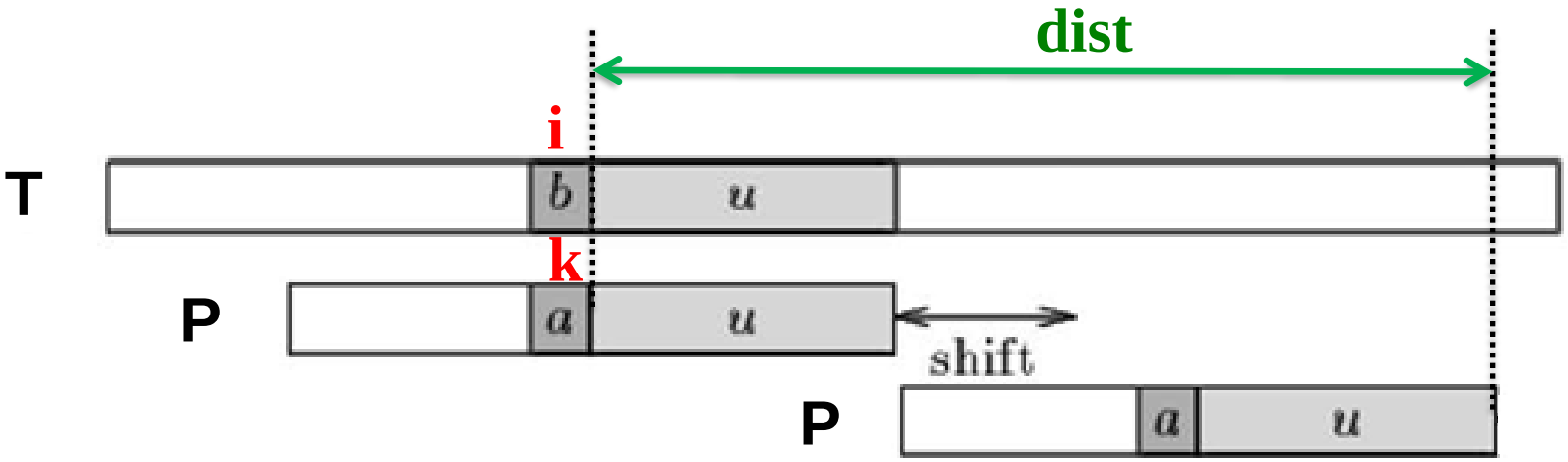


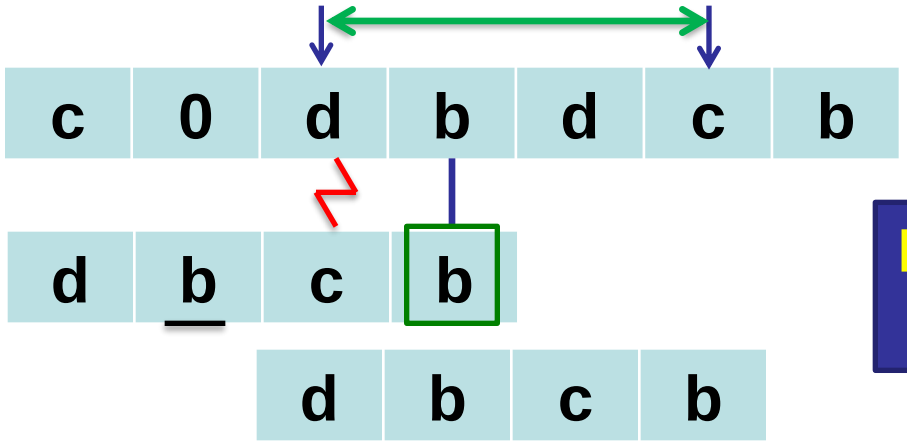
Figure 3. The good-suffix shift, no re-occurs in P

Shift = dist[i]-Len(u),
Len(u) = (Len(P)-1-k)

Q1：坏字符可以怎么被用起来？
Q2：若无“好后缀”，该怎么办？

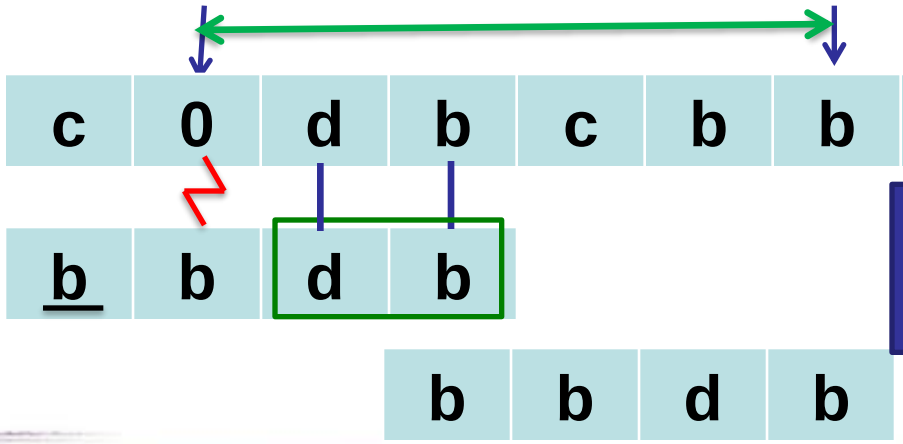
MatchJump数组

Case1:



MatchJump[2]=3
=1+2

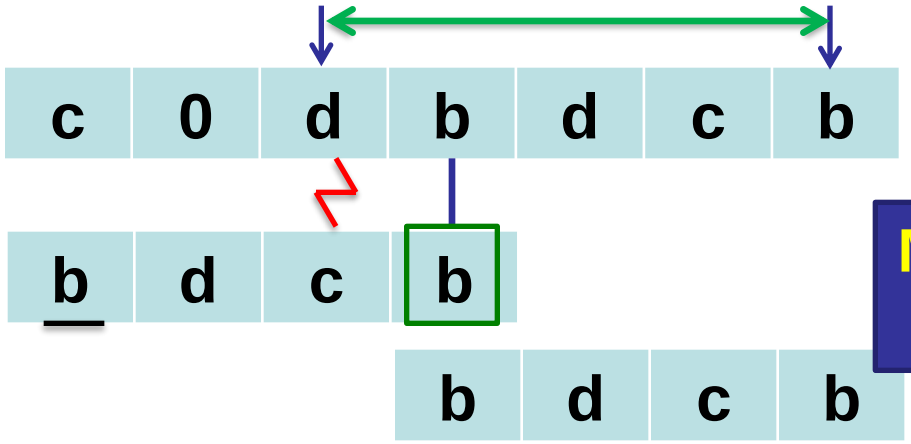
Case2:



MatchJump[2]=5
=2+3

MatchJump数组

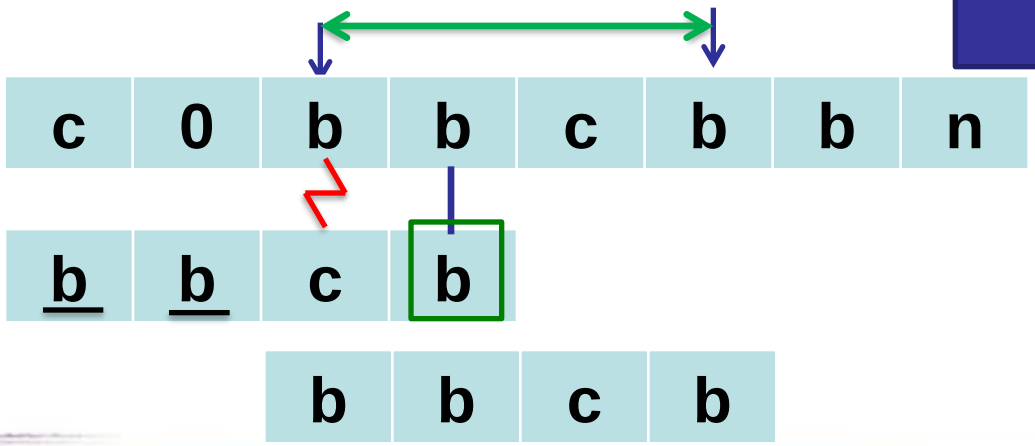
Case3:



MatchJump[2]=4
=1+3

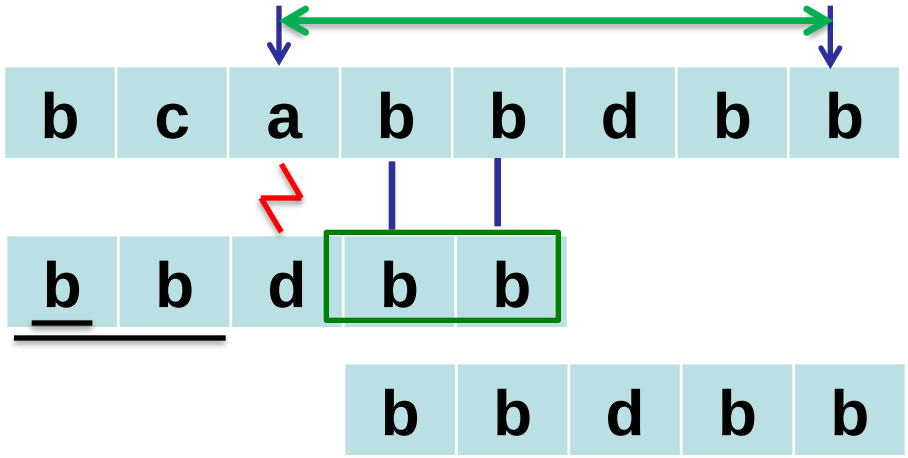
Case4:

MatchJump[2]=3
=1+2



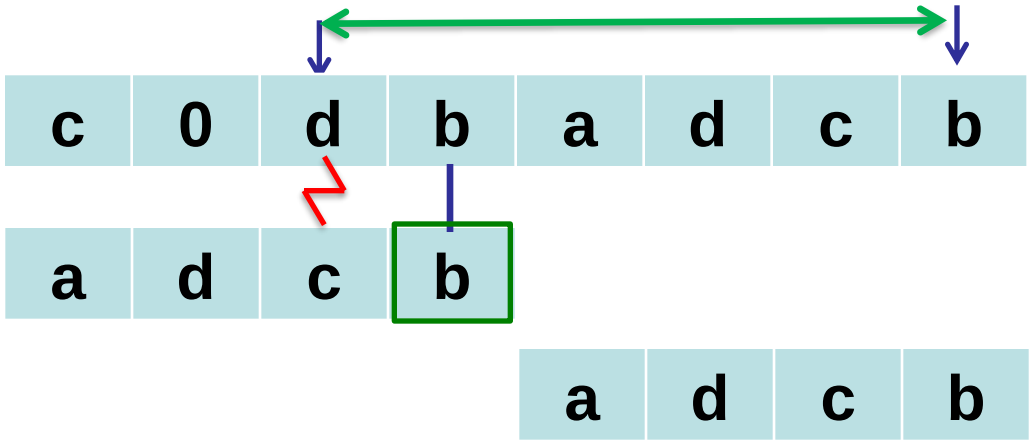
MatchJump数组

Case5:



MatchJump[1]=5
=2+3

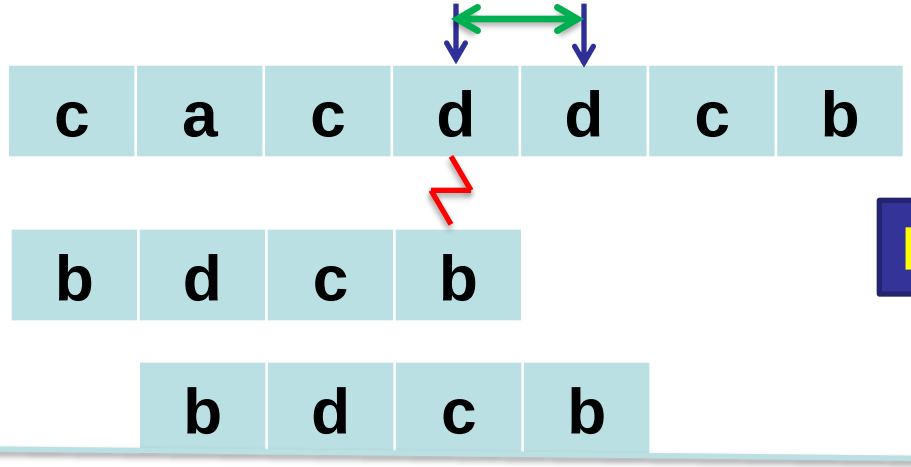
Case6:



MatchJump[2]=5
=1+4

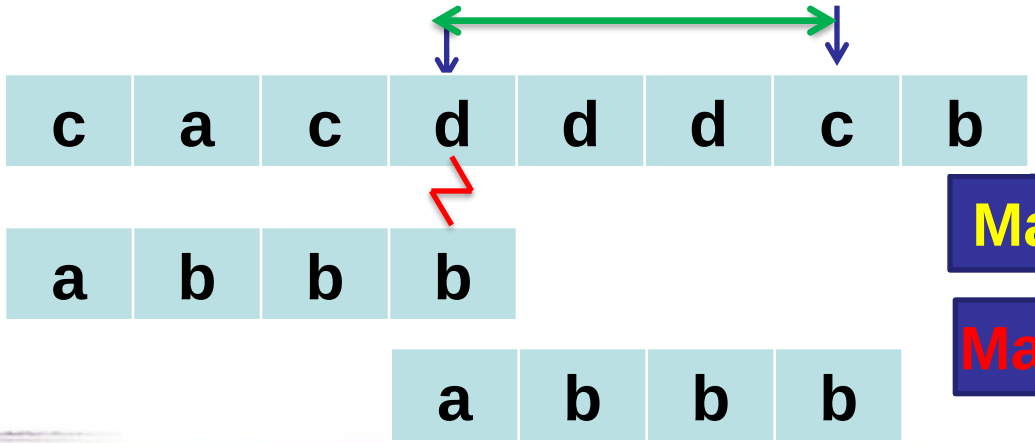
MatchJump数组

Case7:



MatchJump[3]=1

Case8:



MatchJump[3]=1

MatchJump[3]=3?

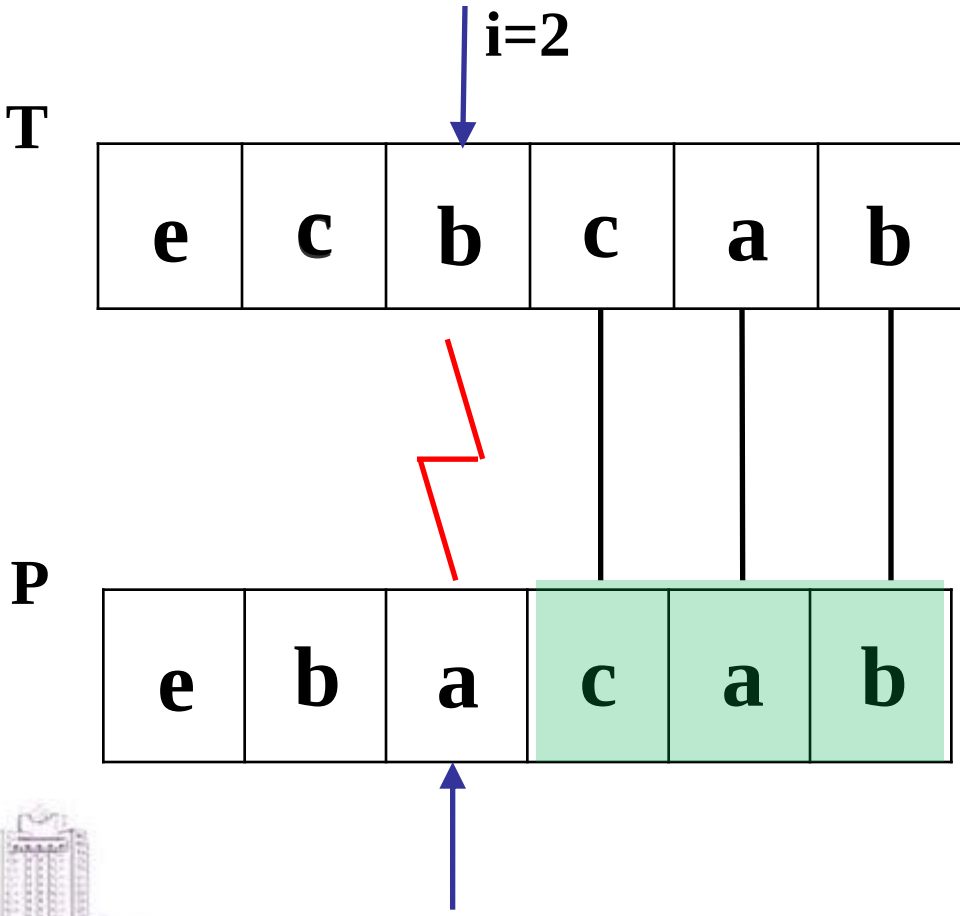
总结：“好后缀”规则

- 关键问题：如何计算目标串中查找指针的移动距离MatchJump?
  - T中查找指针的移动距离dist=模式串P的移动距离shift+好后缀的长度
- Step1: MatchJump数组初始化：
  - MatchJump[k]=2*Len(p)-k-1 = Len(p)-k-1+Len(p);（k∈[0,Len(P)-2]）
  - MatchJump[Len(P)-1] = 1（意味着：此时没有好后缀）
- Step2:若存在好后缀，则依据“好后缀规则”，重新调整MatchJump[k]（k∈[0,Len(P)-2]）（假定：好后缀P’为P[k+1]…P[Len(p)-1]）
  - 情况1：若好后缀P’在P中存在重复模式，且重复模式的前一个字符不等，即P[t +1]…P[Len(p)-1-k+t]== P[k+1]…P[Len(P)-1]且P[t]≠P[k]，则MatchJump(k)=[Len(P)-1-k]+min(k-t)
  - 情况2：若不满足情况1，且P的前缀 为好后缀P’中的某个子后缀P"的重复模式，即(P"=P[t] … P[Len(p)-1]==P[0] … P[Len(p)-1-t]（t>k+1)，则MatchJump(k)=[Len(P)-1-k]+min(t)

（要求计算重复模式的最小间隔，是因为可能有多个重复模式）

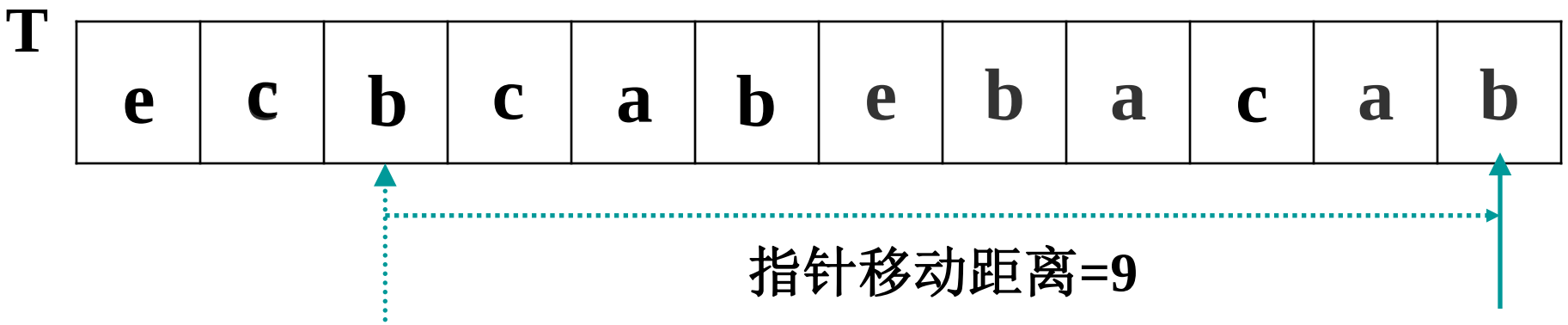
Round 1

在“ecbcabebacab”中查找子串“ebacab”的位置。



- 若利用“坏字符”，则CharJump['b']=0;
- 若利用“好后缀”，则MatchJump(2)=[Len(P)-1-k]+Len(P)=3+6=9.（此时不存在重复模式）
- 查找指针移动距离dist[2]=max(0,9)=9.

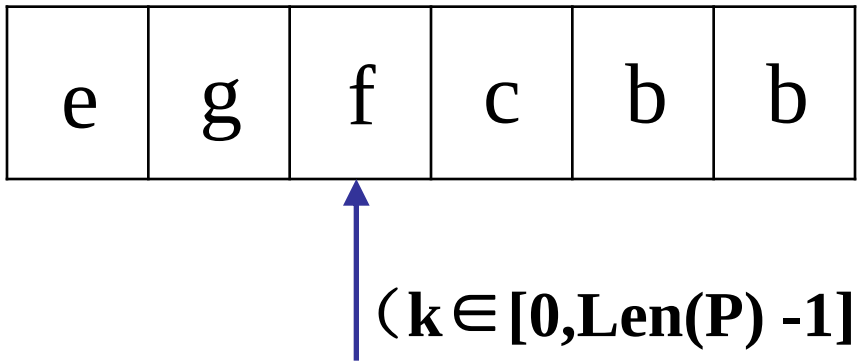
Round 2



从右至左，逐个字符匹配，成功！

示例2：如何计算MatchJump？

- 若P为：

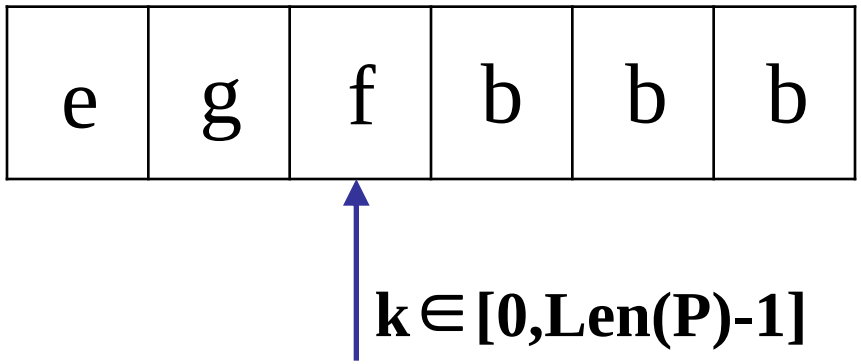


- 分析：模式串p中存在重复模式：后缀子串"b"
- 若好后缀p'中的后缀P''有重复模式，则
- MatchJump(k)=Len(p)-k-1+(t-t')
  - MatchJump(0)=5+6=11
  - MatchJump(1)=4+6=10
  - MatchJump(2)=3+6=9
  - MatchJump(3)=2+6=8
  - MatchJump(4)=1+1=2(（因为好后缀"b"与"cb"中的"b"存在重复， 且P[3]≠ p[4])
  - MatchJump(5)=1

好后缀的长度

示例3：如何计算MatchJump？

- 若P为：

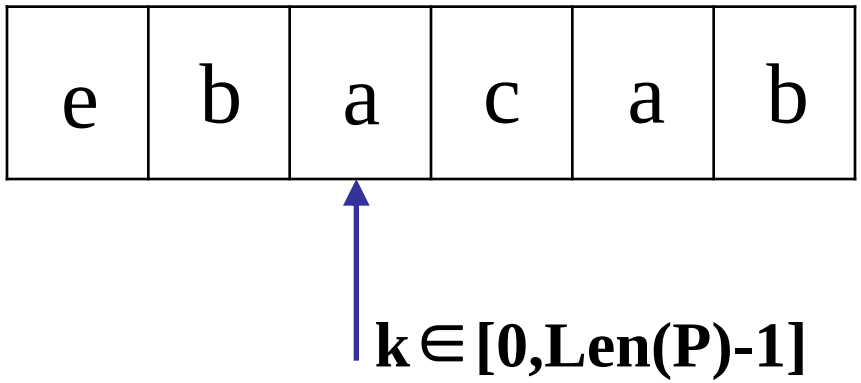


- 分析：模式串p中存在重复模式：后缀子串"bb"、"b"
- 若好后缀p'中的后缀P''有重复模式，则
- MatchJump(k)=Len(p)-k-1+(t-t')
  - MatchJump(0)=5+6=11
  - MatchJump(1)=4+6=10
  - MatchJump(2)=3+6=9
  - MatchJump(3)=2+1=3（因为好后缀"bb"与"fbb"中的"bb"存在重复， 且P[3]≠ p[2])
  - MatchJump(4)=1+2=3（因为好后缀"b"与"fb"中的"b"存在重复，且P[4]≠ p[2])
  - MatchJump(5)=1

好后缀的长度

示例4：如何计算MatchJump？

- 若P为：



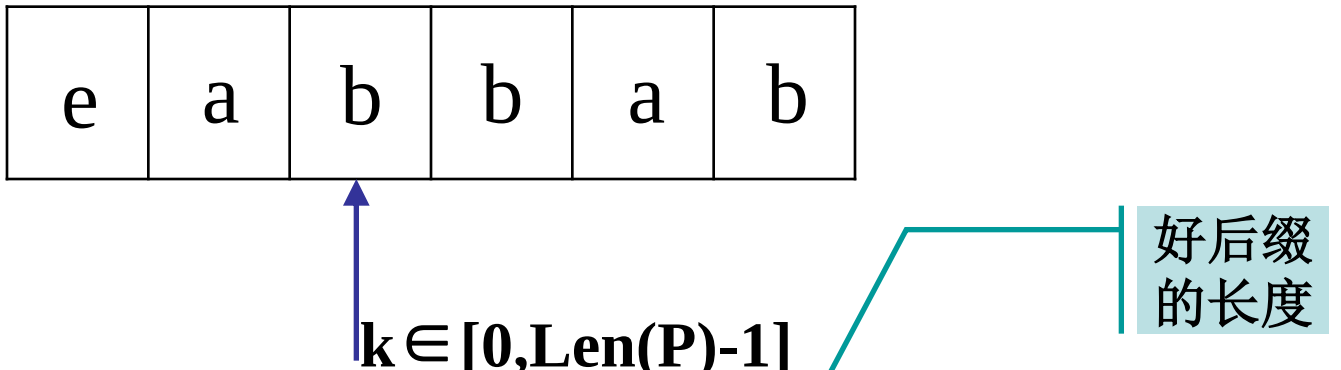
k∈[0,Len(P)-1]

- 分析：模式串p中存在重复模式：后缀子串"b"
- 若好后缀p'中的后缀P''有重复模式，则：
- MatchJump(k)= Len(P)-1-k+(t-t')
  - MatchJump(0)=5+6=11
  - MatchJump(1)=4+6=10
  - MatchJump(2)=3+6=9
  - MatchJump(3)=2+6=8
  - MatchJump(4)=1+4=5（因为好后缀" b"与"eb"中的"b"存在重复， 且P[0]≠ p[4])
  - MatchJump(5)=1

好后缀的长度

示例5：如何计算MatchJump？

- 若P为：



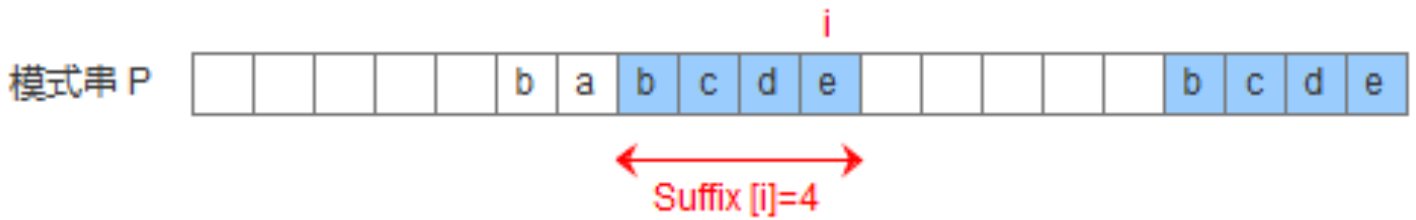
- 分析：模式串p中存在重复模式：后缀子串"ab"、"b"
- 若好后缀p'中的后缀P''有重复模式，则：
- MatchJump(k)= Len(P)-1-k+(t-t')
  - MatchJump(0)=5+6=11
  - MatchJump(1)=4+6=10
  - MatchJump(2)=3+6=6
  - MatchJump(3)=2+3=5（因为好后缀"ab"与"eab"中的"ab"存在重复， 且P[0]≠ p[3])
  - MatchJump(4)=1+2=3（因为好后缀"b"与"bb"中的" b"存在重复， 且P[2]≠ p[4])
  - MatchJump(5)=1

示例6：如何计算MatchJump？

- 若P为"acebabaceb"：
- 分析：模式串p中存在后缀子串"aceb"、"ceb"、"eb"、"b"的重复模式
- 则：
  - MatchJump(0)=9+6=15（P的前缀”aceb”与好后缀中”cebabaceb”的子后缀”aceb”存在重复。好后缀长度为9，重复模式间隔为6）
  - MatchJump(1)=8+6=14
  - MatchJump(2)=7+6=13
  - MatchJump(3)=6+6=12
  - MatchJump(4)=5+6=11
  - MatchJump(5)=4+6=10
  - MatchJump(6)=3+10=13（k=6,意味着在“acebabaceb”带下划线的字符处匹配失败，观察“acebabaceb”，并不满足情况1，（将黄色字符对齐是多余的）按照情况2，找好后缀的子串，仍然没有，故取其初值3+10）
  - MatchJump(7)=2+10=12（与MatchJump(6)的计算原理类似）
  - MatchJump(8)=1+4=5（好后缀为”b”，P中存在1个好后缀的重复模式，如“acebabaceb”，且P[8] ≠P[4]，重复模式间隔为4）
  - MatchJump(9)=1

“好后缀”数组 (shift距离)

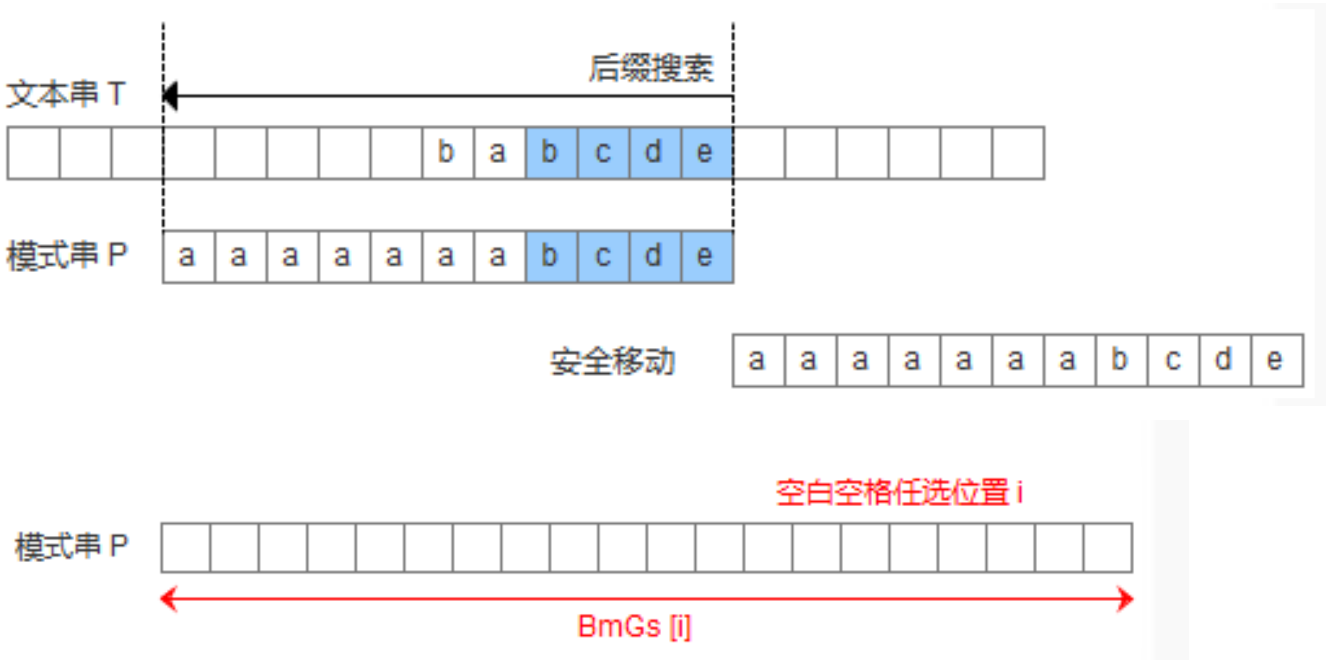
为了实现好后缀规则，需要定义一个数组suffix[]，其中**suffix[i] = s 表示以i为边界，与模式串后缀匹配的最大长度**，即：满足P[i-s, i] == P[m-s, m]的最大长度s。




```
void suffixes(char *x, int m, int *suff){
        suff [m-1]=m;
        for (i=m-2；i>=0；--i){
            q=i;
            while(q>=0&&x[q]==x[m-1-i+q])   --q;
             suff[i]=i-q;   }}
```

“好后缀”数组 (shift距离)

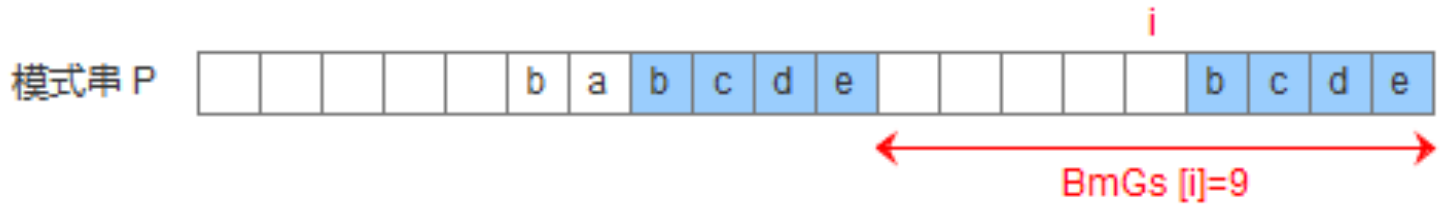
初始化情况（好后缀不重复出现）



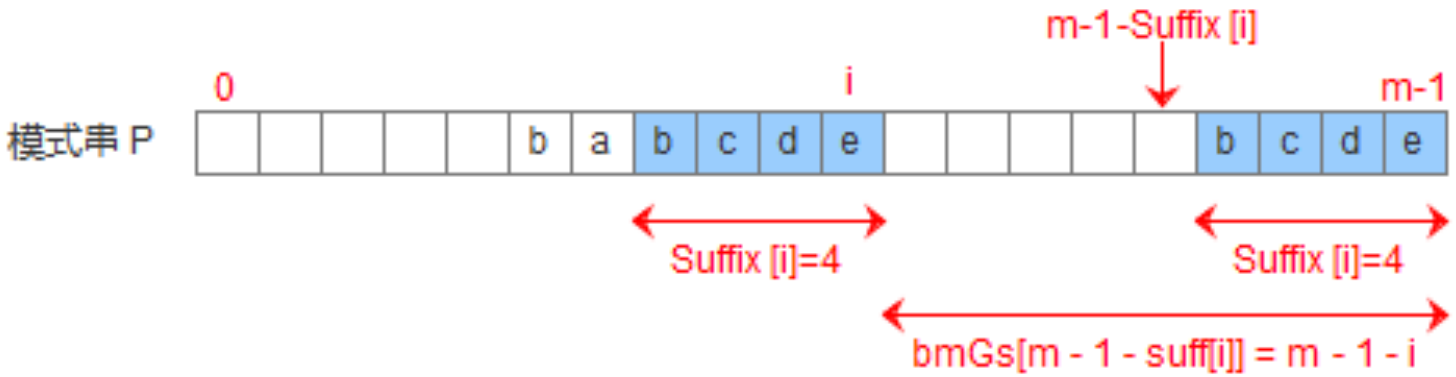
初始化，BmGs[i]均为P串的大小(Len(P))

“好后缀”数组 (shift距离)

有重复后缀 （情况1）



计算视角？：当P串指针为i时，mismatch



“好后缀”数组 (shift距离)

有最大后缀子串（情况2）

对于上述情况，BmGs[2~17] = 18 也就是空白格中的值都应该是18

“好后缀”数组 (shift距离)

有最大后缀子串（情况2），如何计算

字串匹配：x[i-(suff[i]-1) ~ i]＝＝x[m-1- suff[i] ~ m-1],
而前缀必须从0开始：也就是 x[i-(suff[i]-1)~i]=x[0,i]
故suff[i]==i+1时，满足第二种情况。

找到满足suffix[i] = i+1的最大i值

“好后缀”数组 (shift距离)

```
void preBmGs(char *x, int m, int bmGs[]) {
   int i, j, suff[XSIZE];
   suffixes(x, m, suff);
   for (i = 0; i < m; ++i)   bmGs[i] = m; // 初始化情况，无匹配
   j = 0;
   for (i = m - 1; i >= 0; --i)
      if (suff[i] == i + 1)
         for (; j < m - 1 - i; ++j)
            if (bmGs[j] == m) // 值为初值时更改
               bmGs[j] = m - 1 - i; //子缀匹配

   for (i = 0; i <= m - 2; ++i)
      bmGs[m - 1 - suff[i]] = m - 1 - i; // 好后缀匹配
}
```

https://www.cnblogs.com/xubenben/p/3359364.html

BM算法的应用

- 目标串T：GCATCGCAGAGAGTATACAGTACG
- 模式串P：GCAGAGAG

x	A	C	G	其他字符
CharJump(x)	1	6	0	8

- 分析：模式串p中存在后缀子串“G”，“AG”，“GAG”，“AGAG"的重复模式

K	0	1	2	3	4	5	6	7
P[k]	G	C	A	G	A	G	A	G
MatchJump[k]	7+7	6+7	5+7	4+2	3+7	2+4	1+7	1

First attempt

G C A T C G C A G A G A G T A T A C A G T A C G

1

G C A G A G A G

Shift by: 1 (*bmGs*[7]=*bmBc*[A]-8+8)

Second attempt

G C A T C G C A G A G A G T A T A C A G T A C G

3 2 1

G C A G A G A G

Shift by: 4 (*bmGs*[5]=*bmBc*[C]-8+6

Third attempt

G C A T C G C A G A G A G T A T A C A G T A C G

8 7 6 5 4 3 2 1

G C A G A G A G

例题：给定一个非空的字符串 s，检查是否可以通过由它的一个子串重复多次构成。

示例 1:

输入: s = "abab"
输出: true
解释: 可由子串 "ab" 重复两次构成。

示例 2:

输入: s = "aba"
输出: false

示例 3:
输入: s = "abcabcabcabc"
输出: true
解释: 可由子串 "abc" 重复四次构成。 (或子串 "abcabc" 重复两次构成。)

BM算法的性能点评

- **BM**算法的准备时间：与**Patlen+AlphabetSize**成正比。
- **BM**算法利用了**2**个跳转表：**CharJump**和**MatchJump**，跳过了目标字符串中某些字符，不需要对目标字符串中所有字符进行逐一比较。
- 通常模式串越长，**BM**算法速度越快。因为每一次失败的匹配尝试，**BM**算法都能够使用这些信息（好后缀和坏字符）来排除尽可能多的无法匹配的位置。

附：作业7（课后）？

- 利用BM算法和KMP算法完成给定的字符串查找。
- 总结这两类字符串查找算法的特点、并进行性能比较。