

数据结构与算法

第 4 章 字符串

主讲：赵海燕

北京大学信息科学技术学院
“数据结构与算法” 教学组

国家精品课 “数据结构与算法”
<http://www.jpk.pku.edu.cn/pkujpk/course/sjjg/>

张铭，王腾蛟，赵海燕
高等教育出版社，2008. 6，“十一五”国家级规划教材

内容提要

- 字符串基本概念
- 字符串的存储结构
- 字符串运算的算法实现
- 字符串的模式匹配

「字符串基本概念

- 字符编码
- 字符编码顺序
- 字符串抽象数据类型

「字符串基本概念」

- 简称“串”，零个或多个字符/符号的顺序排列构成的数据结构
- 特殊的线性表，即元素为字符（char）的线性表
 - $n (\geq 0)$ 个字符的有限序列，一般记作 $S : "c_0c_1c_2\dots c_{n-1}"$
 - ◆ 其中， S 是串名字，“ $c_0c_1c_2\dots c_{n-1}$ ”是串值
 - ◆ c_i 串中的字符
 - 串的长度：串所包含的字符个数， n
 - ◆ 空串：长度为零的串，不包含任何字符内容

「字符/符号」

- 组成字符串的基本单位
- 取值依赖于字符集 Σ （结点的有限集合）
 - 二进制字符集： $\Sigma = \{0, 1\}$
 - 生物信息中DNA字符集： $\Sigma = \{A, C, G, T\}$
 - 英语语言： $\Sigma = \{26\text{个字符, 标点符号, ...}\}$
 - 简体中文标准字符集 GB2312： $\Sigma = \{6763\text{个汉字, 标点符号, ...}\}$
 -

字符编码

■ ASCII编码

- 单字节 (8 bits)
- 对128个符号 (字符集charset) 进行编码
- 在C和C++中均采用

■ 其他编码方式

- ANSI编码 (本地化, GB2312、BIG5、JIS等, 不同ANSI编码间互不兼容)
- UNICODE (国际化, 各种语言中的每一个字符具有唯一的数字编号, 便于跨平台的文本转换)

「字符的编码顺序

- 为便于字符串间比较和运算，字符编码表一般遵循约定俗成的 “**偏序编码规则**”
- **字符偏序**：根据字符的自然含义，**某些字符间可以**两两比较次序
 - 字典序
 - 中文字符串有些特例，例如 “笔划” 序

「字符串长度」

- 理论上，一个字符串的长度可任意且有限的，但在实际语言中总有一定的长度
 - **定长**: 具有一个固定的最大长度，所用内存量始终如一
 - **变长**: 根据实际需要**伸缩**。尽管命名为变长，但实际长度也有限（取决于可用的内存量）

子串

- 假设两个串 s_1, s_2

$$s_1 = a_0a_1a_2 \cdots a_{n-1}$$

$$s_2 = b_0b_1b_2 \cdots b_{m-1}$$

其中 $0 \leq m \leq n$, 若存在整数 $i (0 \leq i \leq n-m)$, 使得

$$b_j = a_{i+j}, j = 0, 1, \dots, m-1$$

同时成立, 则称串 s_2 是串 s_1 的子串, 或称 s_1 包含串 s_2

- 真子串: 非空且不为自身的子串。空串是任意串的子串
- 任意串S都是其本身的子串

- 子串函数

- 提取、插入、寻找、删除 ...

「字符串数据类型」

- 字符串常数和变量
 - 字符串常数 (string literal)
 - ◆ 例如：“\n” , “a” , “student” ...
 - 字符串变量
- 因语言而不同
 - 简单类型
 - 复合类型

C/C++的标准字符串

- 将 `<string.h>` 函数库作为字符串数据类型的方案
 - 例如： `char s[M];` 定义了字符串变量s
e.g., `char s1[7] = "value";`
- 串的结束标记： `'\0'`
 - `'\0'` 是 ASCII 码中 8 位全 0 码， 又称为 `NULL` 符， 专门用于结束标志
 - 字符串的实际长度为 `M-1`
- 注意： `s1 = s2`

C/C++的标准字符串

- 串长函数
- 串复制
- 串拼接
- 串比较
- 输入和输出函数
- 定位函数
- 右定位函数

int strlen(char *s);
char *strcpy(char *s1, char*s2);
char *strcat(char *s1, char *s2);
int strcmp(char *s1, char *s2);
cin>> cout<<
char * strchr(char *s, char c);
char * strrchr(char *s, char c);

C++的字符串类String

- class String
 - 适应字符串长度动态变化的复杂性
 - 不再直接以字符数组char S[M]的形式出现，而采用一种动态变长的存储结构

C++字符串类的部分操作

| 操作类别 | 方法 | 描述 |
|-------|-------------|------------------------|
| 子串 | substr () | 返回一个串的子串 |
| 拷贝/交换 | swap () | 交换两个串的内容 |
| | copy () | 将一个串拷贝到另一个串中 |
| 赋值 | assign () | 把一个串、一个字符、一个子串赋值给另一个串中 |
| | = | 把一个串或一个字符赋值给另一个串中 |
| 插入/追加 | insert() | 在给定位置插入一个字符、多个字符或串 |
| | += | 将一个字符或串追加到另一个串后 |
| | append () | 将一个或多个字符、或串追加在另一个串后 |
| 拼接 | + | 通过将一个串放置在另一个串后面来构建新串 |
| 查询 | find () | 找到并返回一个子序列的开始位置 |
| 替换/清除 | replace () | 替换一个指定字符或一个串的字串 |
| | clear () | 清除串中的所有字符 |
| 统计 | size () | 返回串中字符的数目 |
| | length () | 返回size () |
| | max_size () | 返回串允许的最大长度 |

「字符串的存储结构和实现

■ 存储结构

- 字符串的顺序存储
- 字符串类 class String 的存储结构

■ 运算实现

- 标准串的运算实现
- 字符串类的运算实现

「字符串的顺序存储

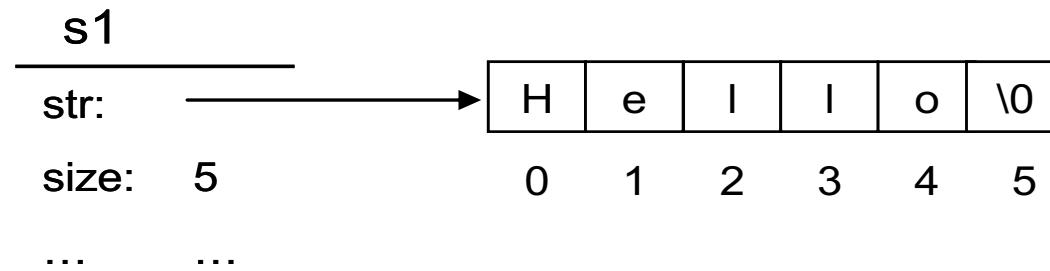
- 对于串长变化不大的字符串，可有三种处理方案：
 - (1) 用 $s[0]$ 作为记录串长的存储单元
 - ◆ 缺点：限制了串的最大长度不能超过256
 - (2) 另辟空间存储串的长度
 - ◆ 缺点：串的最大长度一般是静态给定的，而非动态申请
 - (3) 用特殊标记'\0' (C/C++)
 - ◆ 例如：C的string函数库 (#include <string.h>) 所采用

class String的存储结构

```
private:          // 具体实现的字符串存储结构
    char *str;   // 字符串的数据表示
    int size;    // 串的当前长度
```

例如，

```
String s1 = "Hello";
```



「字符串运算的算法实现」

1. 串长函数

`int strlen(char *s);`

2. 串复制

`char *strcpy(char *d, char*s);`

3. 串拼接

`char *strcat(char *s1, char *s2);`

4. 串比较

`int strcmp(char *s1, char *s2);`

5. 寻找字符

`char * strchr(char *d, char ch)`

`char * strrchr(char *d, char ch)`

6. 抽取子串

`int *strstr(char* s2, char* s1)`

「标准串运算的实现

```
// 字符串的复制
char *strcpy(char *d, char *s) {
    int i = 0;
    while (s[i] != '\0')  {
        d[i] = s[i];  i++;
    }
    d[i] = '\0';
    return d;
}
```

// 问题 ?

「标准串运算的实现

// 字符串的比较

```
int strcmp(const char *s1, const char *s2) {
    int i = 0;
    while (s2[i] != '\0' && s1[i] != '\0') {
        if (s1[i] > s2[i])
            return 1;
        else if (s1[i] < s2[i])
            return -1;
        i++;
    }
    if (s1[i] == '\0' && s2[i] != '\0')
        return -1;
    else if (s2[i] == '\0' && s1[i] != '\0')
        return 1;
    return 0;
}
```

「标准串运算的实现

// 字符串比较 更简便的算法

```
int strcmp_1(char *d, char *s) {
    int i;
    for (i=0; d[i]==s[i]; ++i ) {
        if (d[i]=='\0' && s[i]=='\0')
            return 0;          // 两个字符串相等
    }
    // 不等，比较第一个不同的字符
    return (d[i]-s[i])/abs(d[i]-s[i]);
}
```

「标准串运算的实现

```
// 求字符串的长度
int strlen(char d[ ]) {
    int i = 0;
    while (d[i] != 0)
        i++;
    return i;
}
```

「标准串运算的实现

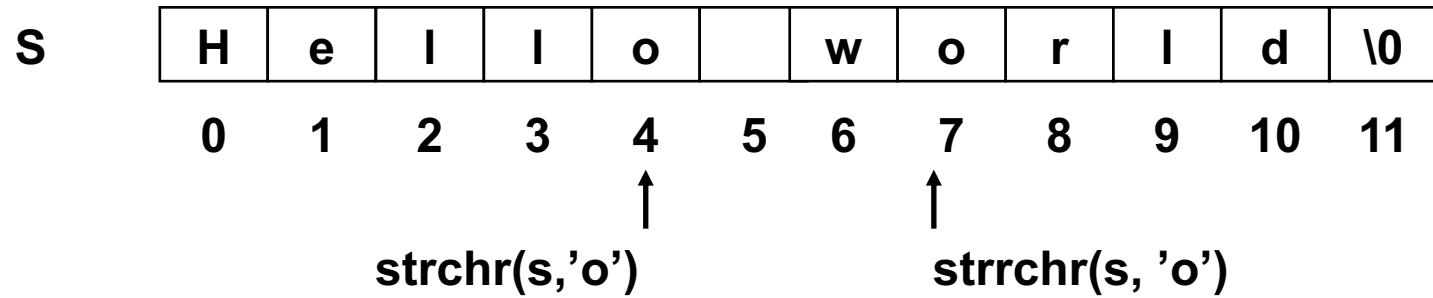
```
// 寻找字符
char * strchr(char *d, char c) {
    // 按照数组指针d依次寻找字符c，若找到c，则将指针位置返回，
    // 没有找到c，则为0值
    i = 0;
    // 循环跳过那些不是c的字符
    while (d[i] != 0 && d[i] != c)
        i++;
    // 当本串不含字符c，则在串尾结束；成功寻找到c，返回该位置指针
    if (d[i] == 0)
        return 0;
    else
        return &d[i];
}
```

「标准串运算的实现

```
// 反向寻找字符
char * strrchr(char *d, char c) {
    // 按照数组指针d, 从尾部反着寻找字符c, 若找到则将指针位置返回,
    // 如果没有找到, 则为0值
    i = 0;
    while (d[i] != '\0')           // 找串尾
        i++;
    // 循环跳过那些不是c的字符
    while (d[--i] != '\0' && d[i] != c) ;
    // 若串不含字符c则在串尾结束; 成功寻找到c则返回该位置指针
    if (d[i] == '\0')
        return 0;
    else
        return &d[i];
}
```

「标准串运算的实现」

e.g. 字符串s :



寻找字符o, `strchr(s, 'o')`结果返回 4;

反方向寻找 o, `strrchr(s, 'o')`结果返回 7

「字符串类String运算的实现」

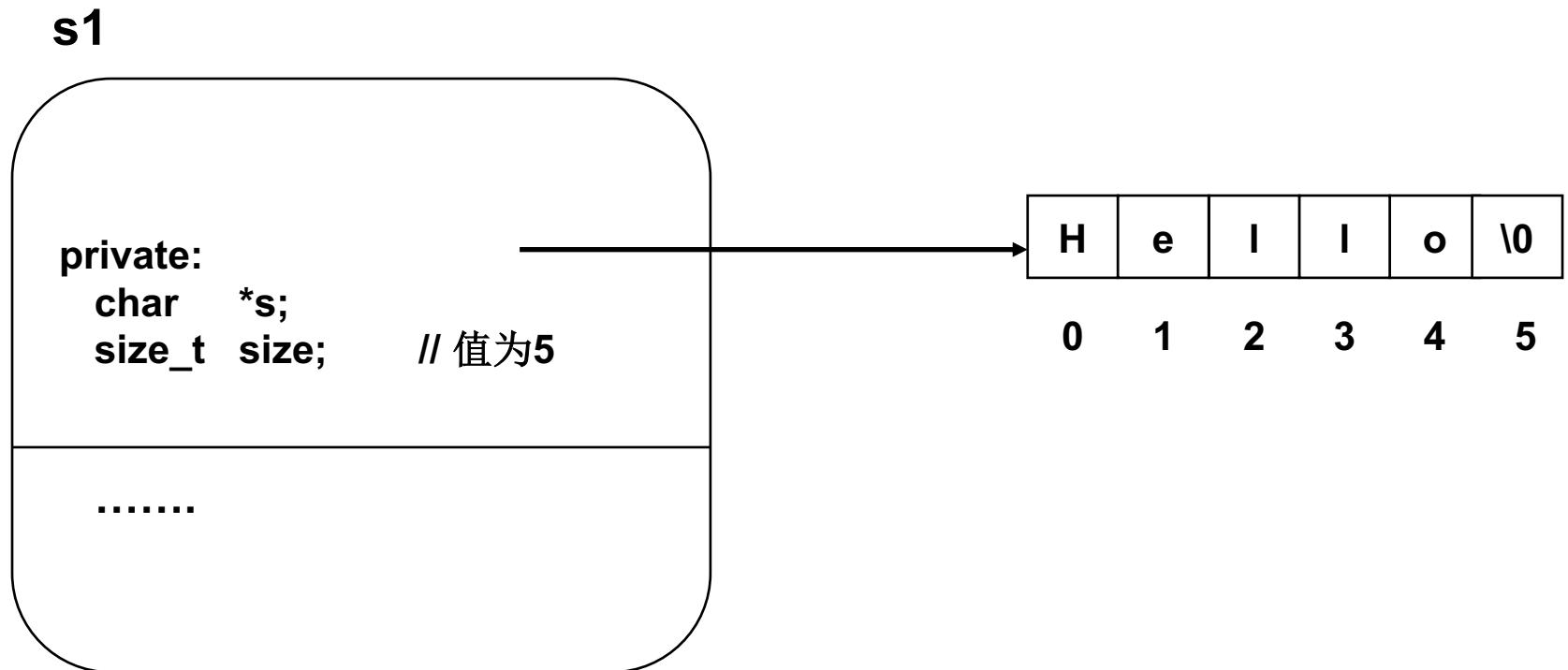
```
// 创建算子(constructor)
String::String(char *s) {
    // 先要确定新创字符串实际需要的存储空间， s的类型为(char *),
    // 作为新创字符串的初值。用标准字符串函数 strlen(s)确定s的长度
    size = strlen(s);

    // 在动态存储区域开辟一块空间， 用于存储初值s， 包括结束符
    str = new char [size+1];
    // 开辟空间不成功时， 运行异常， 退出
    assert(str != '\0');

    // 用标准字符串函数strcpy， 将s完全复制到指针str所指的存储空间
    strcpy(str, s);
}
```

String串的创建运算

```
String s1 = "Hello" ;
```



「String串运算的实现

```
// 析构函数  
String::~String() {  
    // 必须释放动态存储空间  
    delete [] str;  
}
```

「String串运算的实现

// 赋值算子

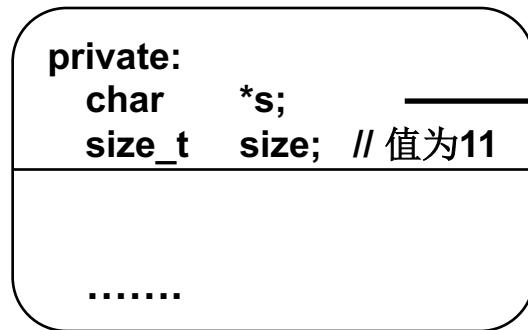
```
String String::operator= (String& s) {
    // 参数 s 将被赋值到本串。若本串的串长和s的串长不同，则应该
    // 释放本串的str存储空间，并开辟新的空间
    if (size != s.size) {
        delete [] str;           // 释放原存储空间
        str = new char [s.size+1];
        // 若开辟动态存储空间失败，则退出正常运行
        assert(str != 0);
        size = s.size;
    }
    strcpy(str, s.str);
    // 返回本实例，作为String类的一个实例
    return *this;
}
```

String的赋值运算

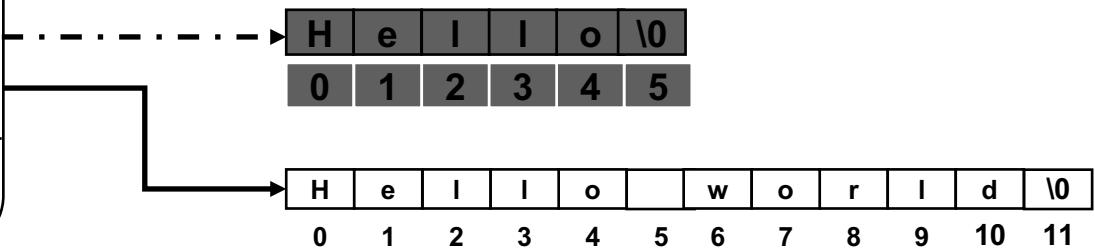
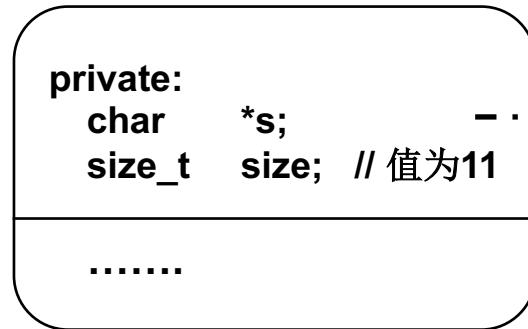
String s2 = "Hello world";

赋值语句： s1 = s2;

s2



s1



「String串运算的实现

// 抽取子串函数

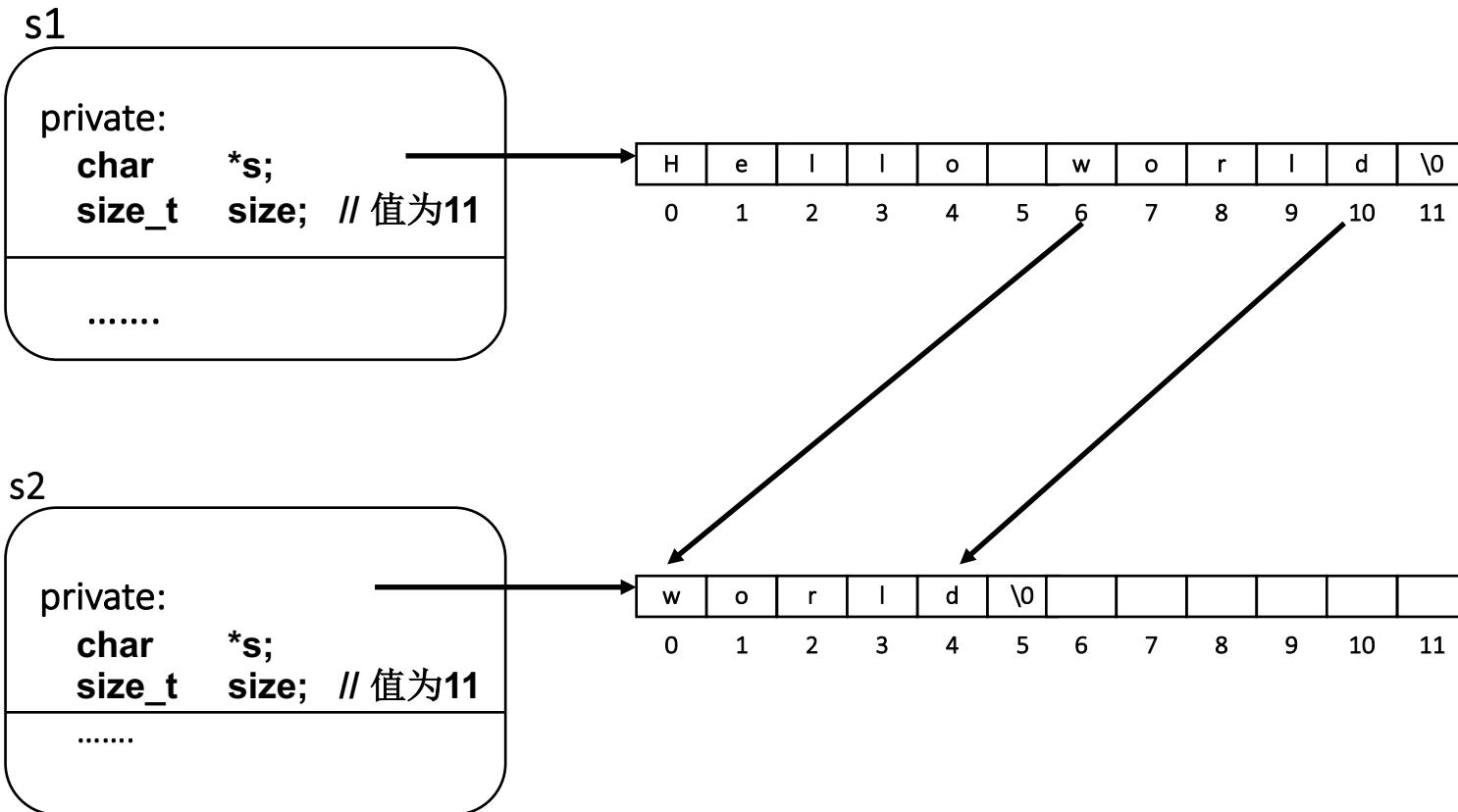
```
String String::Substr(int index , int count ) {  
    // 取出一个子串返回，自下标index开始，长度为count  
  
    int i;  
    // 本串自下标index开始向右数直到串尾，长度为left  
    int left = size - index ;  
    String temp;  
    char *p, *q;  
    // 若下标index值太大，超过本串实际串长，则返回空串  
    if (index >= size)  
        return temp;  
    // 若count超过自index以右的实际子串长度，则把count变小  
    if (count > left )      count = left;  
    // 释放原来的存储空间  
    delete [] temp.str;
```

「String串运算的实现

```
// 若开辟动态存储空间失败，则退出
temp.str = new char [count+1];
assert(temp.str != 0);
// p的内容是一个指针，指向目前暂无内容的字符数组的首字符处
p = temp.str;
// q的内容是一个指针，指向本实例串的str数组的下标index字符
q = &str[index];
// 用q指针取出它所指的字符内容后，指针加1
// 用p该指针所指的字符单元接受拷贝，该指针也加1
for (i =0; i < count; i++)
    *p++ = *q++;
// 循环结束后，让temp.str的结尾为' \0'
*p = 0;
temp.size = count;
return temp;
}
```

String抽取子串

$s2 = s1. substr(6, 5) ;$



字符串模式匹配

■ 模式匹配(pattern matching)

- 一个**目标对象T** (字符串)
- 一个**模式P** (pattern) (字符串)

在**目标T** 中寻找一个**给定模式P** 的过程

■ 应用

- 文本编辑时特定词、句的查找
 - ◆ UNIX/Linux: sed, awk, grep
- DNA信息的提取
- 确认是否具有某种结构
- ...

「字符串模式匹配」

- 精确匹配 (Exact String Matching) : 结果或成或败的匹配, 即, 若在目标 T 中至少存在一处与模式 P 完全相同的子串, 则称为匹配成功, 否则匹配失败
 - 单选的, "Set" ;
 - 多选的, $S?t$ "
 - 正则表达式
- 近似匹配 (Approximate String Matching) : 若模式 P 与 目标 T (或其子串) 存在 某种程度 的相似, 则认为匹配成功
 - 衡量字符串相似度的常定义为一个串转换成另一个串所需的基本操作数目: 基本操作包括字符串的插入、删除和替换三种操作

「字符串模式匹配

■ 模式单选精确匹配

- 用给定的模式 P ，在目标字符串 T 中搜索与模式 P 全同的一个子串，并求出 T 中第一个和 P 全同匹配的子串（简称“配串”），返回其首字符位置

$T: t_0 t_1 \dots t_i t_{i+1} t_{i+2} \dots t_{i+m-2} t_{i+m-1} \dots t_{l-1}$

|| || || || ||

$P: p_0 p_1 p_2 \dots p_{m-2} p_{m-1}$

为使模式 P 与目标 T 匹配，必须满足

$$p_0 p_1 p_2 \dots p_{m-1} = t_i t_{i+1} t_{i+2} \dots t_{i+m-1}$$

单模式匹配算法

| <i>Algorithm</i> | <i>Preprocessing time</i> | <i>Matching time</i> |
|--|------------------------------|---|
| Naïve string search algorithm | $O(\text{no preprocessing})$ | $\Theta(nm)$ |
| Rabin-Karp string search algorithm | $\Theta(m)$ | average $(n+m)$, worst $\Theta(nm)$ |
| Finite automaton | $\Theta(m \Sigma)$ | $\Theta(n)$ |
| Knuth-Morris-Pratt algorithm | $\Theta(m)$ | $\Theta(n)$ |
| Boyer-Moore string search algorithm | $\Theta(m + \sigma)$ | average (n/m) , worst $\Theta(mn)$ |
| Bitap algorithm (<i>shift-or</i> , <i>shift-and</i> , <i>Baeza-Yates-Gonnet</i>) | $\Theta(m + \Sigma)$ | $\Theta(n)$ |
| Shift Or / shift and Algorithm | $\Theta(m + \Sigma)$ | $\Theta(n)$ |

字符串模式匹配

■ 目标

- 在大文本（诸如，句子、段落，或书本）中定位（查找）特定的模式
 - ◆ 对于大多数的算法而言，匹配的主要考虑在于其速度和效率
 - ◆ 有相当数目的算法用于解决模式匹配问题，典型算法
 - 朴素（“Brute Force”/“Naive”）
 - Knuth-Morrit-Pratt (KMP 算法)
 - Boyer-Moore 模式匹配算法 (BM算法)

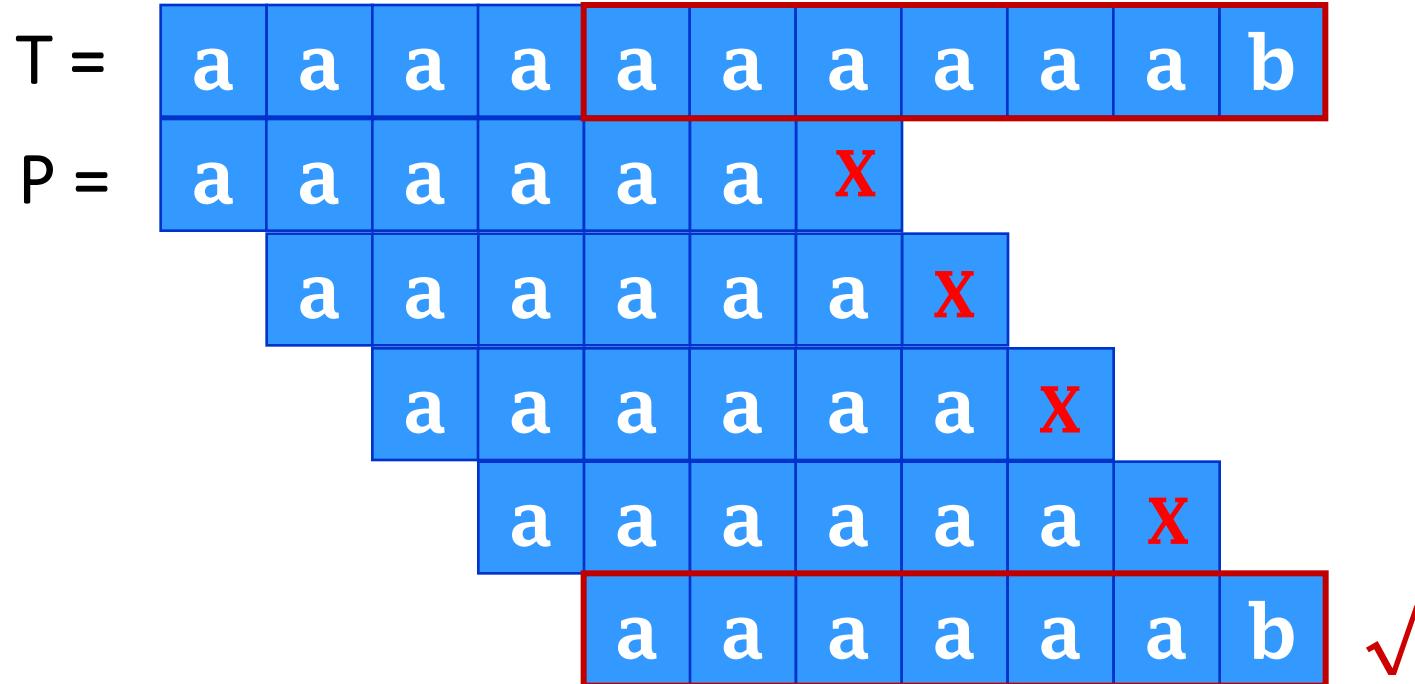
朴素模式匹配

- “Naive”，也称“Brute Force”
 - 穷举
- $T = t_0t_1t_2\dots t_n$, $P = p_0p_1\dots p_{m-1}$: j, i 分别为 T 和 P 中字符的下标指针
 - 尝试所有可能情况
 1. 匹配成功 ($p_0 = t_j, p_1 = t_{j+1}, \dots, p_{m-1} = t_{j+m-1}$)
即满足: $T.substr(j, m) == P.substr(0, m)$
 2. 一趟失配 ($p_i \neq t_j$) 时, 将 P 右移后再行比较, 开始下一趟的匹配

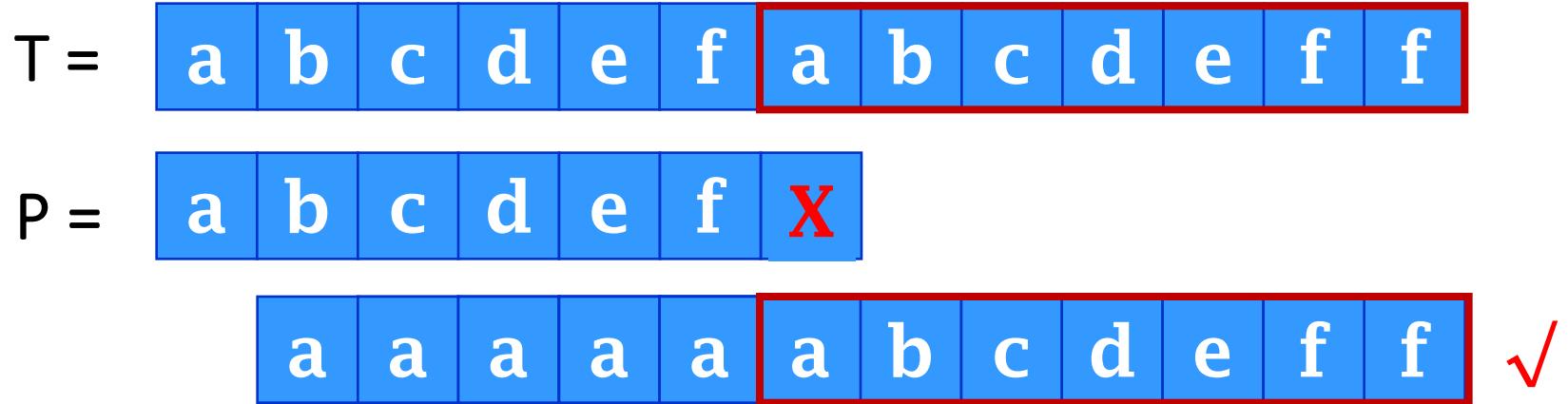
朴素模式匹配：示例

| | | | | | | | | | | | | | |
|-------|----------|---|---|---|---|---|----------|---|---|---|---|---|---|
| $T =$ | a | b | a | b | a | b | a | b | a | b | a | b | b |
| $P =$ | a | b | a | b | a | b | X | | | | | | |
| | X | b | a | b | a | b | b | | | | | | |
| | a | b | a | b | a | b | X | | | | | | |
| | X | b | a | b | a | b | b | | | | | | |
| | a | b | a | b | a | b | X | | | | | | |
| | X | b | a | b | a | b | b | | | | | | |
| | a | b | a | b | a | b | X | | | | | | |
| | X | b | a | b | a | b | b | | | | | | |
| | a | b | a | b | a | b | X | | | | | | |

朴素模式匹配：示例



朴素模式匹配：示例



朴素模式匹配算法：其一

```
int NaiveStrMatching_1(string S, string P, int startindex) {
    // 从S末尾倒数一个模式长度位置
    int LastIndex = S.length() - P.length();
    int count = P.length();
    // 开始匹配位置startindex的值过大， 匹配无法成功
    if (LastIndex < startindex)
        return (-1);
    // g为S的游标， 用模式P和S第g位置子串比较， 若失败则继续循环
    for (int g = startindex; g <= LastIndex; g++) {
        if (P == S.substr(g, count))
            return g;
    }
    // 若for循环结束，则整个匹配失败， 返回值为负，
    return (-1);
}
```

朴素匹配算法：其二

```
#include "String.h"
#include <assert.h>
int NaiveStrMatching (String T, String P) {
    int i = 0;                                // 模式的下标变量
    int j = 0;                                // 目标的下标变量
    int pLen = P.length( );                    // 模式的长度
    int tLen = T.length( );                    // 目标的长度
    if (tLen < pLen)                         // 如果目标比模式短，匹配无法成功
        return (-1);
    while ( i < pLen && j < tLen)           // 反复比较对应字符来开始匹配
        if (T[j] == P[i])
            i++, j++;
        else {
            j = j - i + 1;
            i = 0;
        }
    if ( i >= pLen)
        return (j - pLen+1);
    else return (-1);
}
```

朴素匹配算法：效率分析

- 若目标 T 的长度为 n , 模式 P 长度为 m , 且 $m \leq n$
 - 最差情况下, 每趟循环都不成功, 则共要进行比较 $(n-m+1)$ 趟匹配
 - 每一趟 “相同匹配” 比较所耗费的时间, 是 P 和 T 的字符逐个比较的时间, 最坏情况下, 比较 m 次
 - 故, 整个算法的最坏时间开销估计为 $O(m \bullet n)$

朴素匹配算法：最差情况

- 模式与目标的每一个长度为m的子串进行比较

AAAAAA_{AAAAB}AAAAAAAAAAAAAA

AAAAB 5次比较

A_{AAAAA}AAAAA_{AAAAB}AAAAAA

AAAAB 5次比较

AA_{AAAAA}AAAAA_{AAAAB}AAAAAA

AAAAB 5次比较

AAA_{AAAAA}AAAAA_{AAAAB}AAAAAA

AAAAB 5次比较

.....

AAAAAAAAAAAAAA_{AAAAA}

AAAAB 5次比较

- 目标形如 a^n , 模式形如 $a^{m-1}b$
- 总比较次数：
 - ✓ $n-m+1$ 趟
 - ✓ 每趟 m 次比较
 - 共计 $O(n-m+1)$
- 时间复杂度： $O(mn)$

朴素模式匹配算法： 最佳情况 – 匹配成功

- 在目标的前 m 个位置上找到模式，设 $m = 5$

AAAAAA
AAAAAAAAAAAAAAB

AAAAA

5次比较

- 总比较次数： m （只1趟， m 次比较）
- 时间复杂度： $O(m)$

「朴素匹配算法：

最佳情况 – 匹配失败

- 总在第一个字符上不匹配

| | | | | | | | | | | |
|-------|-------|-------|-------|-------|-------|--------|---------|---------|---------|----|
| A | AAAAA | AAAAA | AAAAA | AAAAA | AAAAA | AAAAA | AAAAA | AAAAA | AAAAA | AH |
| O | OOO | OOO | OOO | OOO | OOO | OOO | OOO | OOO | OOO | H |
| 1次比较 | | | | | | | | | | |
| A | A | AA | AAA | AAAA | AAAAA | AAAAAA | AAAAAAA | AAAAAAA | AAAAAAA | AH |
| O | O | OO | OOO | OOOO | OOOOO | OOOOAA | OOOOAAA | OOOOAAA | OOOOAAA | H |
| 1次比较 | | | | | | | | | | |
| A | AA | AAA | AAA | AAA | AAA | AAA | AAA | AAA | AAA | AH |
| O | O | OO | OOO | OOOO | OOOOO | OOOOAA | OOOOAAA | OOOOAAA | OOOOAAA | H |
| 1次比较 | | | | | | | | | | |
| A | AAA | AAA | AAA | AAA | AAA | AAA | AAA | AAA | AAA | AH |
| O | O | OO | OOO | OOOO | OOOOO | OOOOAA | OOOOAAA | OOOOAAA | OOOOAAA | H |
| 1次比较 | | | | | | | | | | |
| | | | | | | | | | | |
| A | AAAA | AAAA | AAAA | AAAA | AAAA | AAAA | AAAA | AAAA | AAAA | AH |
| O | O | OO | OOO | OOOO | OOOOO | OOOOAA | OOOOAAA | OOOOAAA | OOOOAAA | H |
| 1次比较 | | | | | | | | | | |

- 总比较次数：
 - $n-m+1$
 - $n-m+1$ 趟
 - 1次比较/趟
- 时间复杂度：
 - $O(n)$

思考

- 朴素模式匹配效率低下的原因？

「朴素匹配算法的问题」

| | | | | | | | | | | | | | |
|-------|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $T =$ | a | b | a | b | a | b | a | b | a | b | a | b | b |
| $P =$ | a | b | a | b | a | b | X | | | | | | |
| | X | b | a | b | a | b | b | | | | | | |
| | a | b | a | b | a | b | a | b | X | | | | |
| | X | b | a | b | a | b | a | b | b | | | | |
| | a | b | a | b | a | b | a | b | a | b | X | | |
| | X | b | a | b | a | b | a | b | b | b | | | |
| | a | b | a | b | a | b | a | b | a | b | b | | b |

「朴素算法的问题：回溯

- 朴素算法之所以效率低下，在于匹配过程中**目标串**多有**回溯**，回溯是否**必要**？
- e.g.,
 - 由1) 可知： $P_6 \neq T_6$ ， $P_0 = T_0$, $P_1 = T_1$, 同时由 $P_0 \neq P_1$ 可得知 $P_0 \neq T_1$ 故将 P 右移一位后的第2) 趟匹配的第1次比较一定不等；冗余的比较
 - 那么把P右移几位合适？既能**消除冗余比较**又保证不丢失配串呢？

T a b a b a b a b a b a b b
P a b a b a b b

1) $P_6 \neq T_6$ P右移一位

T a b a b a b a b a b a b b
P a b a b a b b

2) $P_0 \neq T_1$ P右移一位

T a b a b a b a b a b a b b
P a b a b a b b

3) $P_6 \neq T_8$ P右移一位

T a b a b a b a b a b a b b
P a b a b a b b

.....

「无回溯匹配算法

- 关键在于匹配过程中，一旦 p_j 和 t_i 比较不等时，即， $\text{substr}(P, 0, j-1) = \text{substr}(T, i-j+1, j-1)$ $\&\&$
 $p_j \neq t_i$

要能立即确定模式 相对目标 右移的位数 和 继续比较的字符，即该用 P 中的哪个字符和 t_i 继续进行比较？

如何定位？保留之前比较的结果？

若把这个字符记为 p_k ，显然有 $k < j$ (??)，且不同的 j ，其 k 值可能不同

KMP算法

- Knuth-Morris-Pratt (KMP) 发现每个字符对应的该k值仅依赖于模式P本身，与目标串T无关
 - 理论：1970年，S. A. Cook在进行抽象机的理论研究时证明了最差情况下模式匹配可在 $O(N+M)$ 时间内完成
 - 理论指导下的实践：D. E. Knuth 和V. R. Pratt以Cook理论为基础，构造了一种在 $O(N+M)$ 时间内进行模式匹配的方法
 - 实践出发的发现：与此同时，J. H. Morris在开发文本编辑器时为了避免检索文本时的回溯也得到了同样的算法

KMP算法思想

$T \ t_0 \ t_1 \dots t_{i-j-1} \ t_{i-j} \ t_{i-j+1} \ t_{i-j+2} \dots t_{i-2} \ t_{i-1} \textcolor{red}{t}_i \dots \ t_{n-1}$
|| || || || || \times

$P \ p_0 \ p_1 \ p_2 \dots p_{j-2} \ p_{j-1} \textcolor{red}{p}_j$

则有 $t_{i-j} \ t_{i-j+1} \ t_{i-j+2} \dots \ t_{i-1} = p_0 \ p_1 \ p_2 \dots p_{j-1}$ (1)

朴素下一趟 $p_0 \ p_1 \ \dots \ p_{j-2} \ p_{j-1}$

若 $p_0 \ p_1 \ \dots \ p_{j-2} \neq p_1 \ p_2 \ \dots \ p_{j-1}$ (2)

则立刻可以断定

$p_0 \ p_1 \ \dots \ p_{j-2} \neq t_{i-j+1} \ t_{i-j+2} \ \dots \ t_{i-1}$

(朴素匹配的)下一趟一定不匹配，可跳过不做

$p_0 \ p_1 \ \dots \ p_{j-2} \ p_{j-1}$

KMP算法思想

同样，若 $p_0 p_1 \dots p_{j-3} \neq p_2 p_3 \dots p_{j-1}$

则再下一趟也失配，因为有

$$p_0 p_1 \dots p_{j-3} \neq t_{i-j+2} t_{i-j+3} \dots t_{i-1}$$

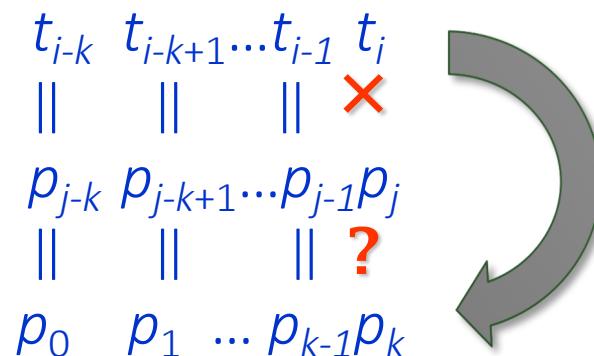
直到对于某一个 “ k ” 值(首尾串长度)，使得

$$p_0 p_1 \dots p_k \neq p_{j-k-1} p_{j-k} \dots p_{j-1}$$

且

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

模式右滑 $j-k$ 位



则 $p_0 p_1 \dots p_{k-1} = t_{i-k} t_{i-k+1} \dots t_{i-1}$

KMP算法的关键：模式本身

| | | | | | | | | |
|-----------|-------------|-------------|-----|-----------|-------------|-----|-----------|-------|
| t_{i-j} | t_{i-j+1} | t_{i-j+2} | ... | t_{i-k} | t_{i-k+1} | ... | t_{i-1} | t_i |
| | | | | | | | | ✗ |
| p_0 | p_1 | p_2 | ... | p_{j-k} | p_{j-k+1} | ... | p_{j-1} | p_j |
| | | | | | | | ? | |
| | p_0 | p_1 | ... | p_{k-1} | p_k | | | |



模式右移 $j-k$ 位

$$p_0 p_1 \dots p_{k-1} = t_{i-k} t_{i-k+1} \dots t_{i-1}$$

$t_i \neq p_j$, $p_j == p_k$?

关键： 对每个 p_j , 如何求 k 值

「字符串特征向量

- 设模式P由m个字符组成，记为

$$P = p_0 p_1 p_2 p_3 \dots p_{m-1}$$

- 令**特征向量 N** 用来表示模式P 的**字符分布特征**，简称**N向量**

- 和P同长，由m个特征数 $n_0 \dots n_{m-1}$ 整数组成，记为

$$N = n_0 n_1 n_2 n_3 \dots n_{m-1}$$

N在很多文献中也称为**next**数组，每个 n_i 对应**next**数组中的元素**next[i]**

KMP匹配算法示例

P =

| | | | | | | |
|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| a | b | a | b | a | b | b |

N = -1 0 0 1 2 3 4

T =

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|----|----|----|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| a | b | a | b | a | b | a | b | a | b | a | b | b |

P =

| | | | | | | |
|---|---|---|---|---|---|---|
| a | b | a | b | a | b | X |
|---|---|---|---|---|---|---|

 i=6, j=6, N[j]=4

| | | | | | | |
|---|---|---|---|---|---|---|
| a | b | a | b | a | b | X |
|---|---|---|---|---|---|---|

 i=8, j=6, N[j]=4

| | | | | | | |
|---|---|---|---|---|---|---|
| a | b | a | b | a | b | X |
|---|---|---|---|---|---|---|

 i=10, j=6, j'=4

| | | | | | | |
|---|---|---|---|---|---|---|
| a | b | a | b | a | b | b |
|---|---|---|---|---|---|---|

 ✓

KMP模式匹配算法

```
int KMPStrMatching(string T, string P, int *N, int start) {
    int j= 0;                                // 模式的下标变量
    int i = start;                            // 目标的下标变量
    int pLen = P.length( );                   // 模式的长度
    int tLen = T.length( );                   // 目标的长度
    if (tLen - start < pLen)                // 若目标比模式短， 匹配无法成功
        return (-1);
    while ( j < pLen && i < tLen) {          // 反复比较， 进行匹配
        if (j == -1 || T[i] == P[j])
            i++, j++;
        else j = N[j];
    }
    if (j >= pLen)
        return (i-pLen);                    // 注意下标的计算
    else return (-1);
}
```

「字符串特征向量

■ 相关概念

□ 模式 P 的 前缀子串（首串）

- ◆ P的前 t 个字符: $p_0p_1p_2\dots p_{k-1}$

□ 模式 P 的 左子串（尾串）

- ◆ 相对于串中的某个位置而言的

- ◆ j位置的左子串: $p_{j-k}\dots p_{j-2}p_{j-1}$, 第j位置前的 k个字符

■ 模式P第 j 个位置的特征数 n_j

□ 最长的 (k 最大的) 能够与前缀子串匹配的左子串 (简称第 j 位的最长前缀串)

□ k 即 p_j 的特征数 n_j

「字符串特征向量N：构造方法

- 模式P 第j个位置的特征数 n_j , 首尾串最长的k
 - 首串: $p_0 p_1 \dots p_{k-2} p_{k-1}$
 - 尾串: $p_{j-k} p_{j-k+1} \dots p_{j-2} p_{j-1}$

$$\text{next}[j] = \begin{cases} -1, & j = 0 \text{时候} \\ \max \{k: 0 < k < j \text{ } \& \text{ } P[0\dots k-1] = P[j-k\dots j-1]\}, & \text{首尾配串最长} k \\ 0, & \text{其他} \end{cases}$$

「求特征向量算法框架」

一般，模式第一个字符的特征数设置为 **-1**，以尽可能地减少匹配时的**冗余比较**：

特征数 n_j ($j > 0, 0 \leq n_{j+1} \leq j$) 可**递归定义**为：

- ① $n_0 = -1$ ，对于 $j \geq 0$ 的 n_{j+1} ，若已知前一位位置 n_j 的特征数 $n_j = k$ ；
- ② If ($k \geq 0 \ \&\& \ p_j \neq p_k$)，令 $k = n_k$ ；循环**步骤②**直到条件不满足
- ③ $n_{j+1} = k + 1$ ； // 此时， $k == -1 \ || \ p_j == p_k$

「字符串的特征向量N：非优化版

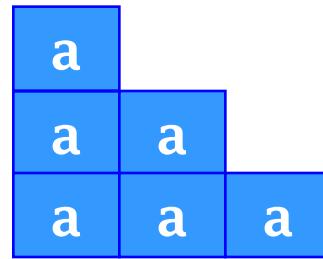
```
int findNext(string P) {  
    int j, k;  
    int m = P.length();  
    assert(m > 0);  
    int *next = new int[m];  
    assert(next != 0);  
    next[0] = -1;  
    j = 0; k = -1;  
    while (j < m-1) {  
        while (k >= 0 && P[k] != P[j])  
            k = next[k];  
        j++; k++;  
        next[j] = k;  
    }  
    return next;  
}
```

「求特征向量示例

$N = \boxed{-1} \boxed{0} \boxed{1} \boxed{2} \boxed{3} \boxed{0} \boxed{1} \boxed{2} \boxed{3} \boxed{4}$

$P = \begin{matrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ \boxed{a} & \boxed{a} & \boxed{a} & \boxed{a} & \boxed{b} & \boxed{a} & \boxed{a} & \boxed{a} & \boxed{a} & \boxed{c} \end{matrix} \quad j = \boxed{9} \quad k = \boxed{0}$

首/尾串→



首/尾串→

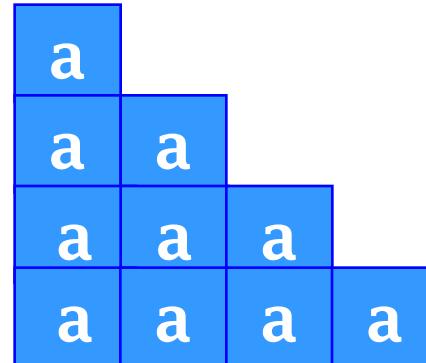
首/尾串→

首/尾串→

首/尾串→

首/尾串→

首/尾串→



求特征向量示例

$$j = \boxed{12} \quad k = \boxed{0}$$

$P = \begin{array}{cccccccccccccc} & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\ \hline a & b & a & b & a & b & a & c & a & b & a & a & a & a \end{array}$

首/尾串 \rightarrow $\begin{array}{ccccc} a & b & a & b & a \end{array} \quad \begin{array}{ccccc} a & b & a & a & a \end{array}$

$N = \begin{array}{cccccccccccccc} -1 & 0 & 0 & 1 & 2 & 3 & 4 & 5 & 0 & 1 & 2 & 3 & 1 \end{array}$

「字符串的特征向量N：优化版

```
int findNext(string P) {  
    int j, k;  
    int m = P.length( );  
    int *next = new int[m];  
    next[0] = -1;  
    j = 0; k = -1;  
    while (j < m-1) {  
        while (k >= 0 && P[k] != P[j])  
            k = next[k];  
        j++; k++;  
        if (P[k] == P[j])  
            next[j] = next[k];  
        else next[j] = k;  
    }  
    return next;  
}
```

// m为模式P的长度
// 动态存储区开辟整数数组
// 若写成 $j < m$ 会越界
// 若不等，采用 KMP 找首尾子串
// k 递归地向前找
// 取消 if 判断，则不优化
// 前面找 k 值，没有受优化的影响

「字符串的特征向量

优化与否的next数组对比

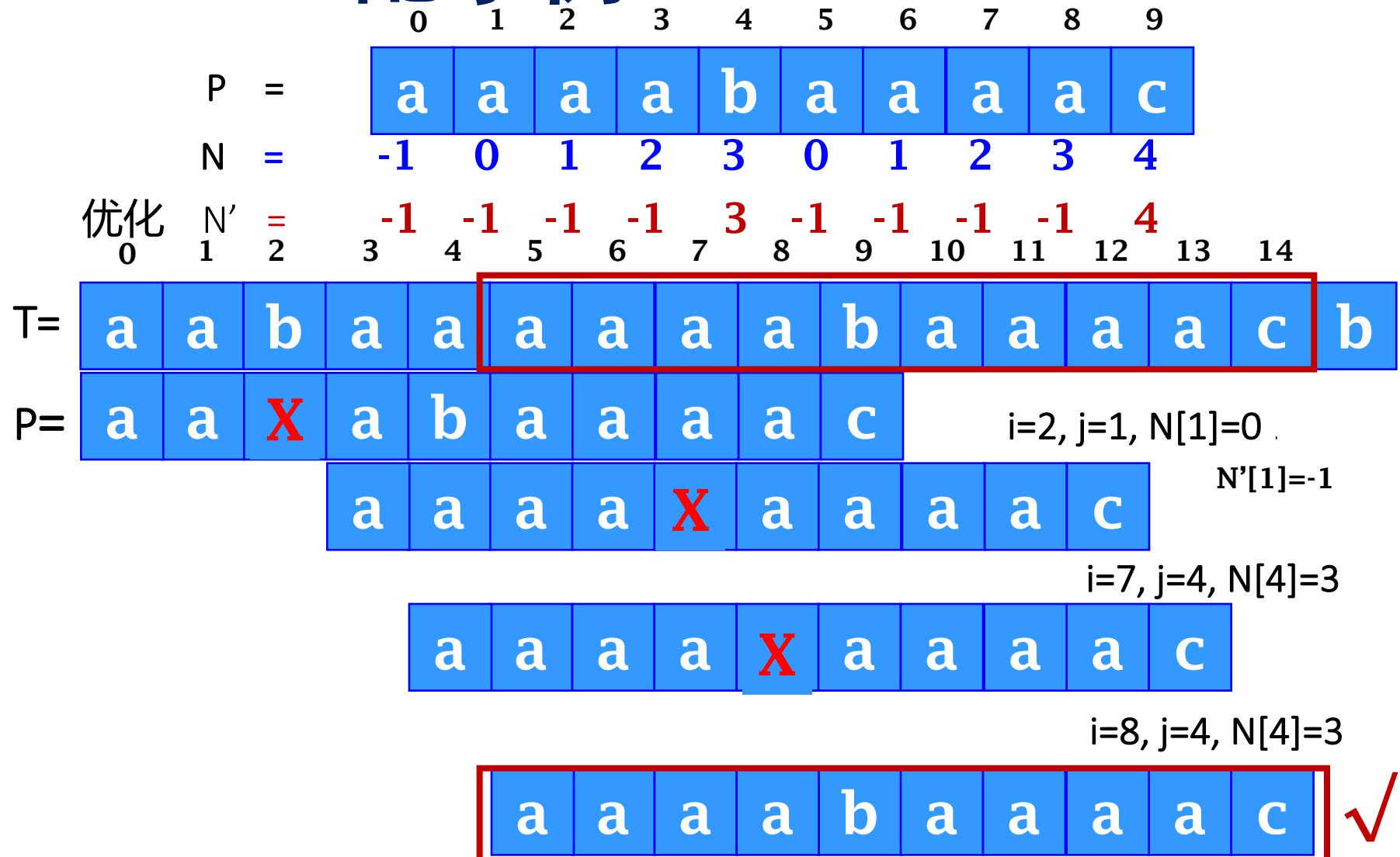
| 序号 j | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | |
|---------------|----|--------|--------|------|--------|------|--------|------|------|------|
| P | a | b | c | a | a | b | a | b | c | |
| k | | 0 | 0 | 0 | 1 | 1 | 2 | 1 | 2 | 非优化版 |
| $p_k == p_j?$ | | \neq | \neq | $==$ | \neq | $==$ | \neq | $==$ | $==$ | |
| n_j | -1 | 0 | 0 | -1 | 1 | 0 | 2 | 0 | 0 | 优化版 |

「为何优化KMP特征数组？」

| | | | | | | | | | |
|-----|---|---|---|---|---|-------|---|---|---------|
| | 1 | 2 | 3 | 4 | 5 | | | | |
| 目标串 | a | a | a | a | s | a | a | a | a |
| 模式串 | a | a | a | a | X | | | | |

| | | | | | | |
|---------------|----|----|----|----|----|------|
| 序号 j | 0 | 1 | 2 | 3 | 4 | |
| P | a | a | a | a | a | |
| k | | 0 | 1 | 2 | 3 | 非优化版 |
| $p_k == p_j?$ | | == | == | == | == | |
| n_j | -1 | -1 | -1 | -1 | -1 | 优化版 |

KMP匹配示例



KMP算法的效率分析

- 主要代价体现在while循环语句
 - 循环最多执行 $n = T.length()$ 次
 - 由于循环体中j只增不减的特性，语句 “ $j++;$ ” 最多执行 n 次；故 “ $i++;$ ” 语句也最多执行 n 次
 - i 初值为0，使之减少的语句只有 “ $i = N[i];$ ”；由于 $i < N[i]$ ，“ $i = N[i];$ ”每执行一次必然使得 i 减少(至少减1)
 - 故，若 “ $i = N[i];$ ”的执行次数不超过 n 次。
- 故，KMP算法的时间为 $O(n)$
- 同理，求 N 数组的时间为 $O(m)$

「特征向量再考慮

■ 优化的度

- 不足
- 过度

KMP匹配示例

| | | | | | | | | | | |
|-----|----|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| P = | a | a | a | a | b | a | a | a | a | c |
| N = | -1 | 0 | 1 | 2 | X | 0 | 1 | 2 | 3 | 4 |

| | | | | | | | | | | | | | | | |
|-----|---|---|---|---|---|---|---|---|---|---|------------------|----|----|----|----|
| T = | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
| | a | a | b | a | a | a | a | a | a | b | a | a | a | a | c |
| P = | a | a | X | a | b | a | a | a | a | c | i=2, j=1, N[j]=0 | | | | |

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| a | a | a | a | X | a | a | a | a | c |
|---|---|---|---|---|---|---|---|---|---|

i=7, j=4, N[4]=X

错过了 !

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| a | a | a | a | b | a | a | a | a | c |
|---|---|---|---|---|---|---|---|---|---|

思考

- KMP算法是否适合大字符集语言（例如中文）的字符串匹配？
- 什么情况下，优化的KMP算法会有较好的效果？在何种应用中，不进行优化？

「next数组定义区别

2008“十一五”版本，蓝皮书。j位匹配错误，则 $j = \text{next}[j]$

$$\text{next}[j] = \begin{cases} -1, & \text{对于 } j = 0 \\ \max \{k: 0 < k < j \text{ \&& } P[0...k-1] = P[j-k...j-1] \}, & \text{如果 } k \text{ 存在} \\ 0, & \text{否则} \end{cases}$$

2004“十五”版本，红皮书。j位匹配错误，则 $j = \text{next}[j-1]$

$$\text{next}[j] = \begin{cases} 0, & \text{对于 } j = 0 \\ \max \{k: 0 < k < j \text{ \&& } P[0...k] = P[j-k...j] \}, & \text{如果 } k \text{ 存在} \\ 0, & \text{否则} \end{cases}$$

「08版方便于优化next数组

若 $p_k = p_j$ ，因 $p_j \neq t_i$ ， $\therefore p_k \neq t_i$ ；

此时应继续右移，使的 $p_{\text{next}[k]}$ 与 t_i 比较；

也即，可进一步优化为：

```
if (p_k == p_j)
    next[j] = next[k];
else
    next[j] = k;
```

「next数组对比

2008“十一五”版本（含优化）。若j位失配，则 $j = \text{next}[j]$

| 序号 j | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---------------|----|--------|--------|------|--------|------|--------|------|------|
| P | a | b | c | a | a | b | a | b | c |
| k | | 0 | 0 | 0 | 1 | 1 | 2 | 1 | 2 |
| $p_k == p_j?$ | | \neq | \neq | $==$ | \neq | $==$ | \neq | $==$ | $==$ |
| n_j | -1 | 0 | 0 | -1 | 1 | 0 | 2 | 0 | 0 |

2004“十五”版本，红皮书。若j位失配，则 $j = \text{next}[j-1]$

| 序号 j | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|------|---|---|---|---|---|---|---|---|---|
| P | a | b | c | a | a | b | a | b | c |
| k | 0 | 0 | 0 | 1 | 1 | 2 | 1 | 2 | 3 |

KMP算法再回顾

- 预处理模式串，得到相应的next数组（模式的特征向量）
- 自左向右将模式串与目标串逐一比较，一旦发现失配，根据模式失配位置上的字符的特征值，确定模式向右的位移量，开始下一趟匹配，如此直到匹配成功或失败

有无 更高效的 字符串模式匹配 方法？

「Boyer-Moore 模式匹配算法」

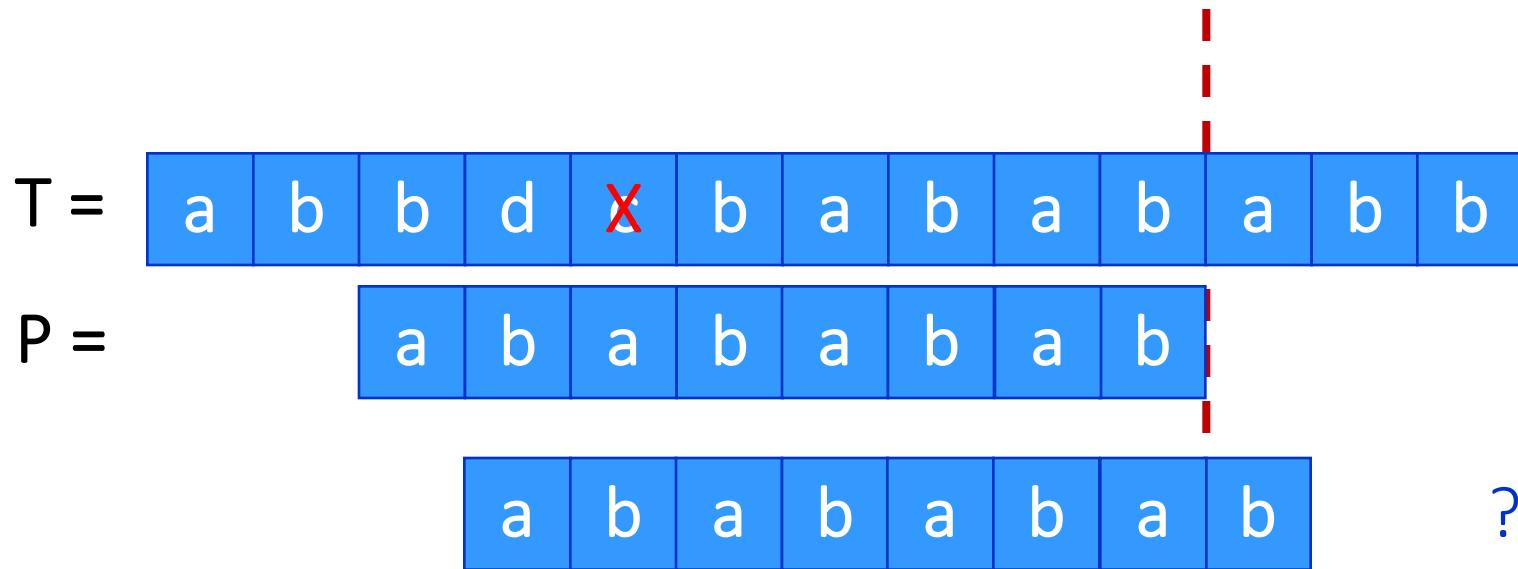
- 简称BM算法，由Robert S. Boyer 和 J. Strother Moore（以及另一独立发表者R. S. Gosper）于1977年设计
- 与KMP类似，需对**模式串预处理**，进行相应的特征分析，并在尔后的模式匹配中充分利用预处理过程中所获得的模式特征来提高匹配速度
- 一般意义上，**BM算法的运行速度随模式长度增长而加快**

「BM 模式匹配算法

■ 基本思想

- 目标串T和模式串P的每趟匹配**自右向左**进行比较
- 匹配过程中若在目标串 T_j ($1 \leq j \leq n$) 处失配，则根据预处理的结果**右移模式串**后开始下一趟自右向左的匹配，运用
 - ◆ 坏字符规则 (the bad character rule)
 - ◆ 好后缀规则 (the good suffix rule)
- 坏字符规则是基础，好后缀规则是对前者的补充和优化

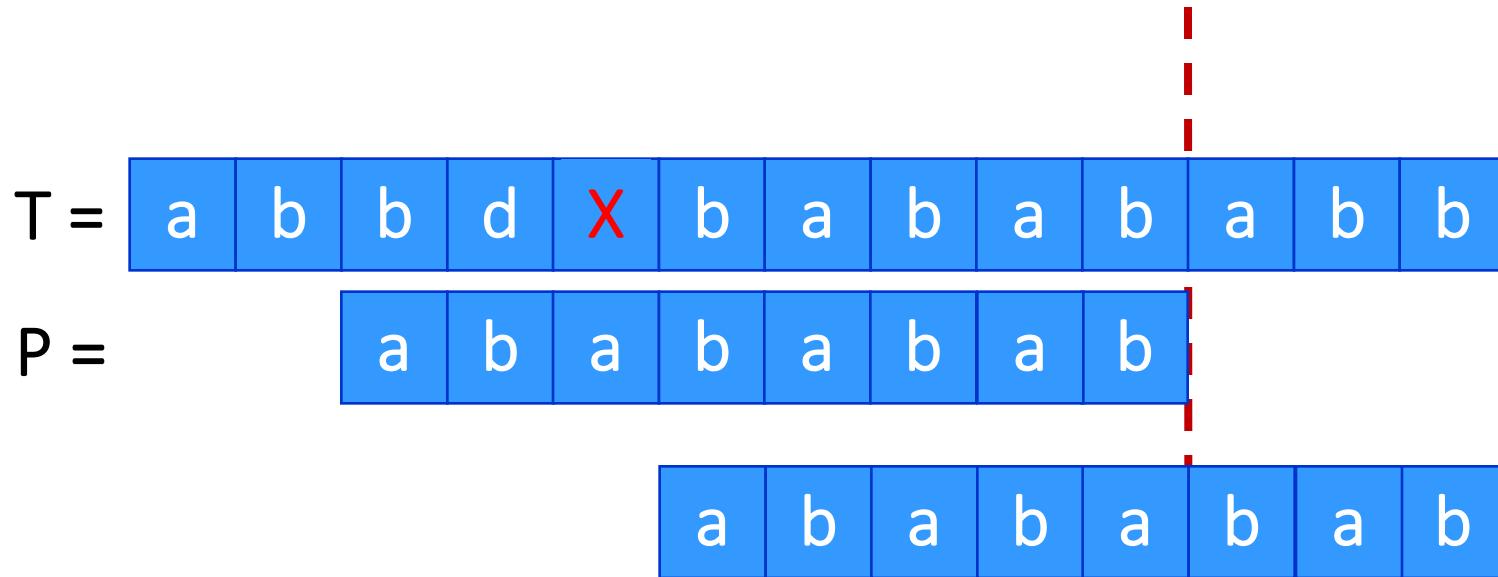
「BM 算法的自右向左」



坏字符： $T[4]=c$

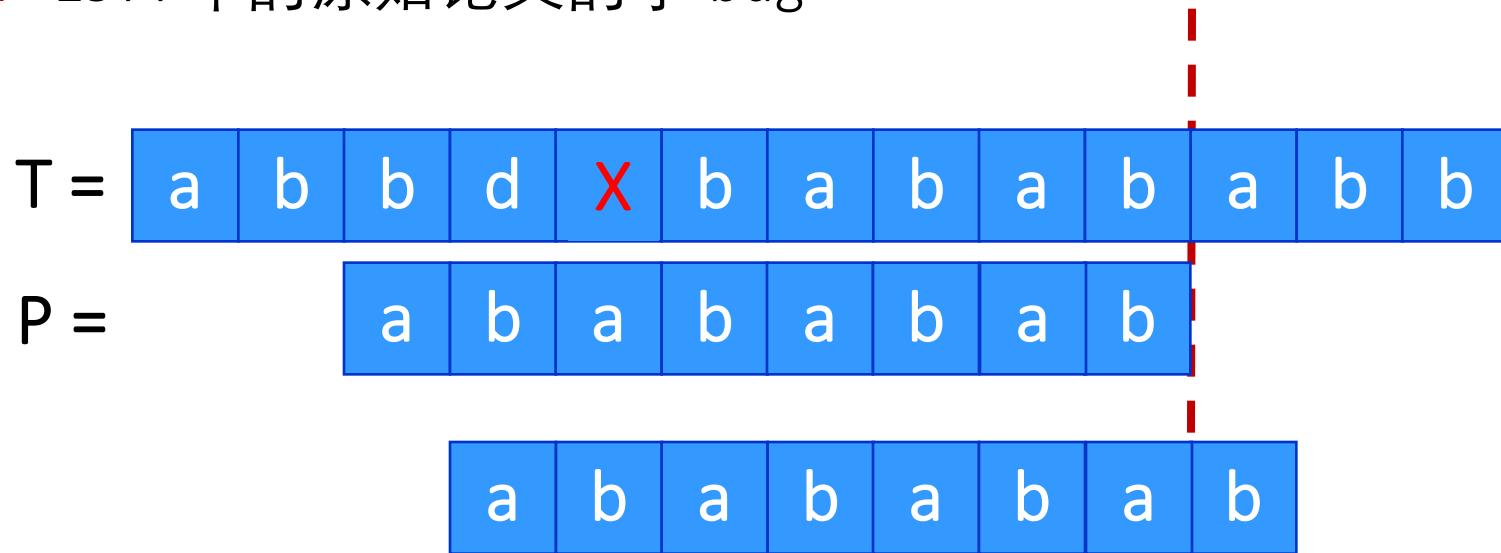
「坏字符规则 #1

- 坏字符没出现 在模式串中：此时可将模式移动到坏字符的下一个字符，继续下一趟比较



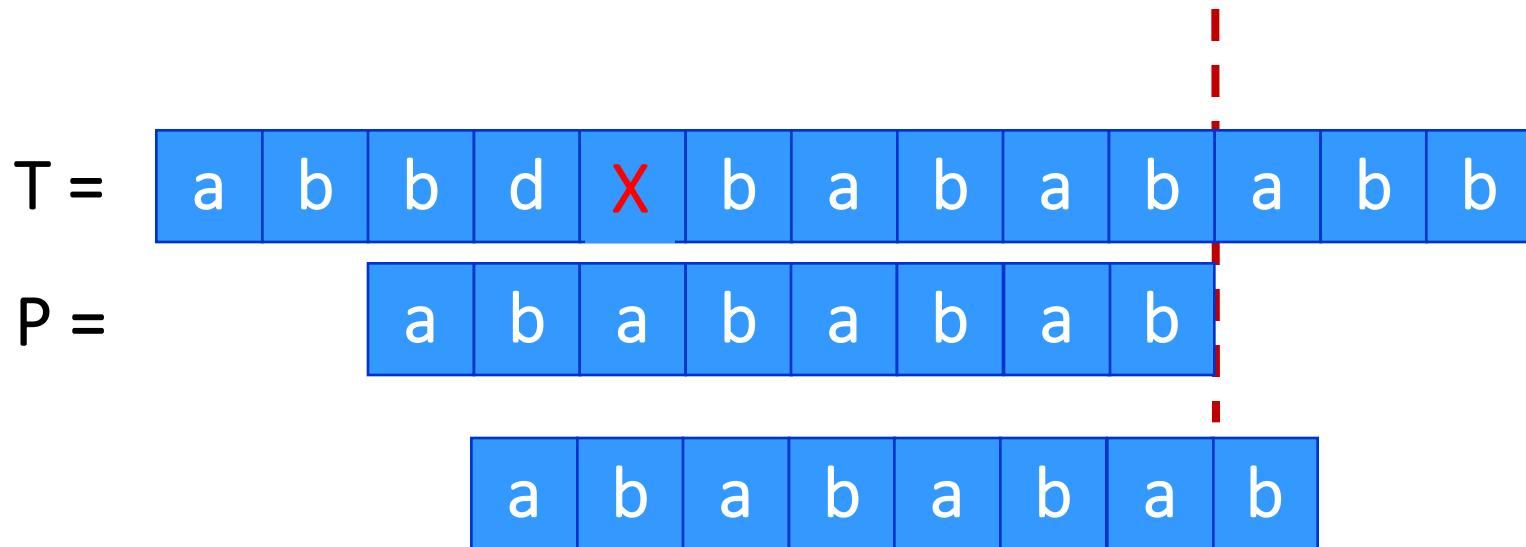
「坏字符规则 #2」

- 坏字符出现在模式串中：此时将模式中第一个出现的坏字符和目标串的坏字符对齐开始下一趟比较（有可能造成模式串的倒退移动）
 - 1977年的原始论文的小 bug



「坏字符 扩展规则 #2

- 坏字符在模式串中出现：与坏字符对应的模式当前位置为 j ，则将模式右移，使得模式中 j 位置前最右出现 的坏字符和 目标串的坏字符对齐 开始下一趟比较（若有）



「坏字符规则」

- 需要预先为坏字符定义一个函数，来指明失配时如何移动模式
 - 设 $\text{Skip}(x)$ 为P右移的距离， m 为模式串P的长度， $\max(x)$ 为字符 x 在P中最右位置

$$\text{skip}(x) = \begin{cases} m; & x \neq P[j] \ (0 \leq j \leq m-1), \text{即 } x \text{ 在 } P \text{ 中未出现} \\ m - \max(x); & \{k \mid P[k] = x, 0 \leq k \leq m-1\}; x \text{ 在 } P \text{ 中出现} \end{cases}$$

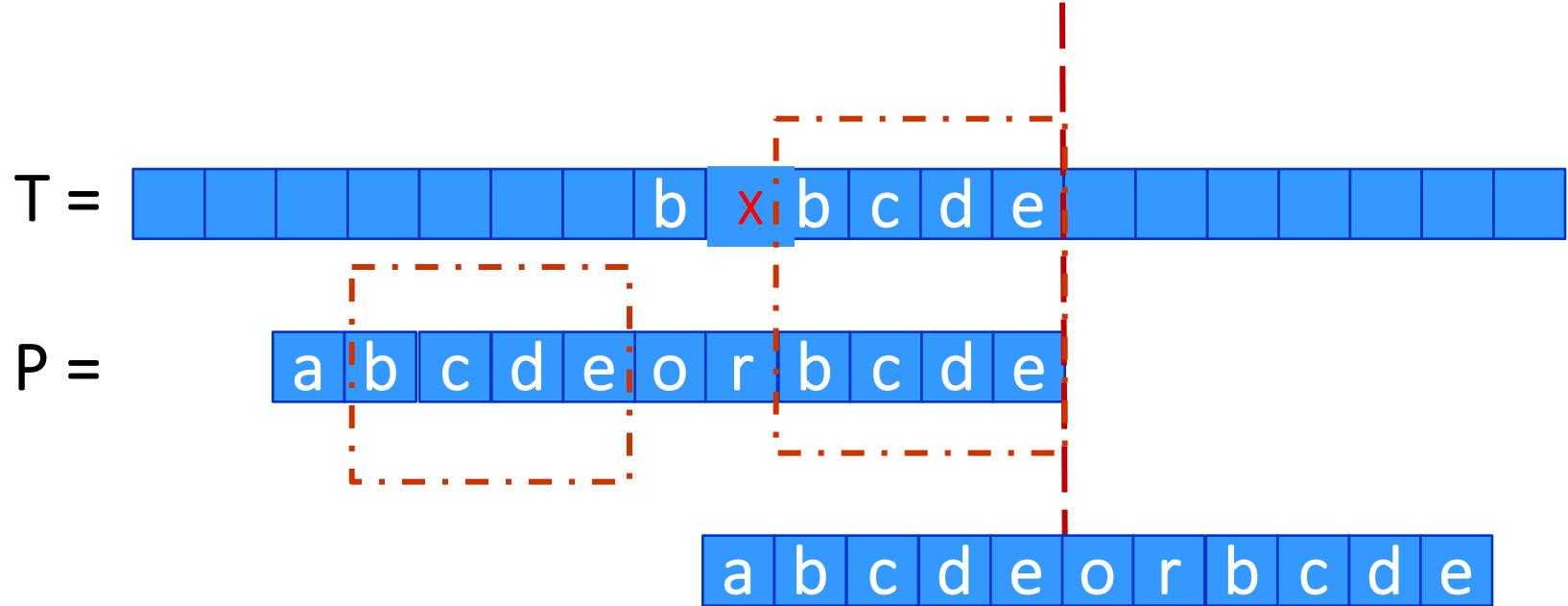
- 简单版本：空间复杂度： $O(|\Sigma|)$ （字符集的大小）
- 扩展版本：空间复杂度： $O(m * |\Sigma|)$ （每个字符在模式每一位置上失配时之前最右出现的字符位置）

「坏字符规则」

- 适合于文本所在字符集较大的情况
 - 大多数情况下坏字符不在模式中出现；
 - 或出现也不在对应的模式位置之前，失配时右移幅度大
- 不适于字符集较小的文本，存在大量相似但不精确的子串
 - DNA ($|\Sigma| = 4$)
 - Protein ($|\Sigma| = 20$)

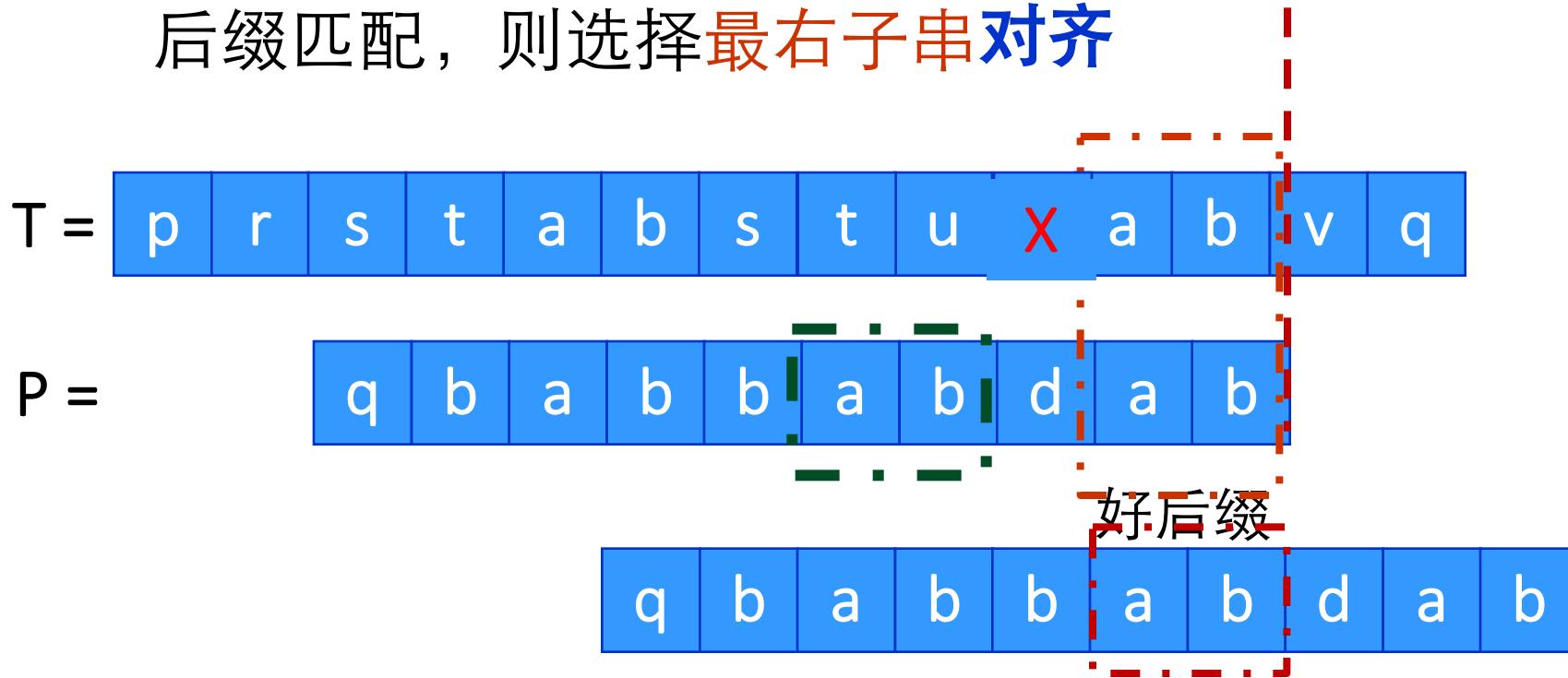
好后缀规则

- 坏字符规则没有考虑匹配过程中已取得的部分匹配结果（而KMP算法则考虑了），由此产生



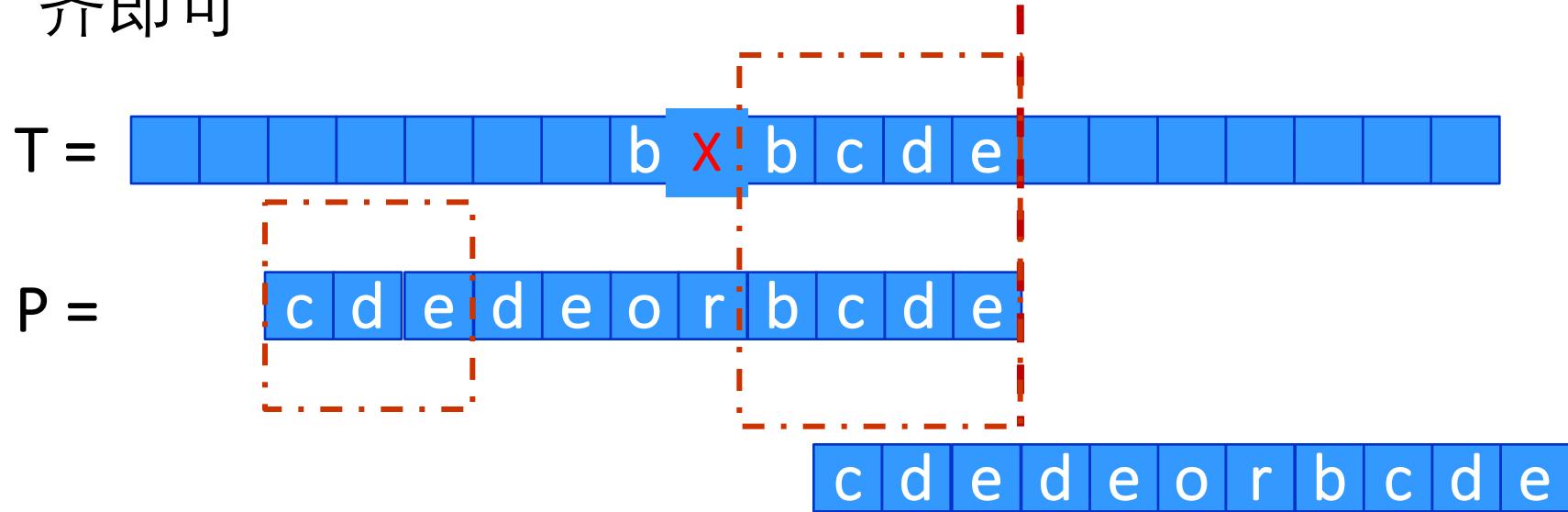
「好后缀规则 #1

- 模式串中有**子串**与**好后缀**匹配：此时移动模式串，使得该子串和好后缀**对齐**即可，若有多**多个子串**与好后缀匹配，则选择**最右子串对齐**



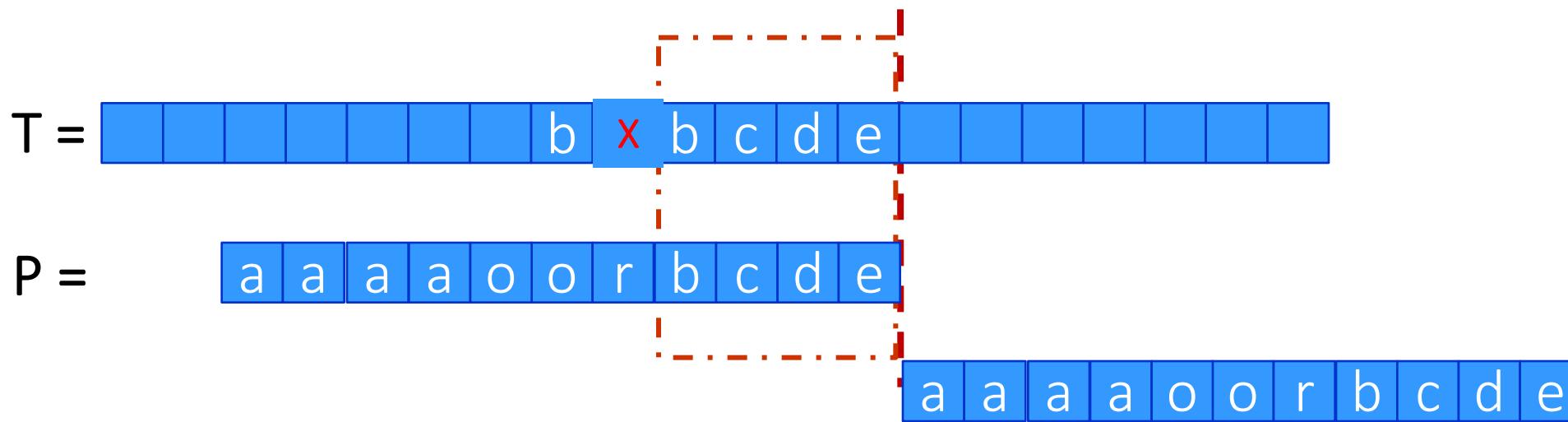
| 好后缀规则 #2

- 若模式串中没有子串与好后缀匹配，此时需寻找模式串的一个最长前缀，使得该前缀等于好后缀的后缀，若寻找到这样的前缀，则将该前缀与好后缀对齐即可



| 好后缀规则 #3

- 若模式串中没有子串与好后缀匹配，且在模式中找不到最长前缀，使得该前缀等于好后缀的后缀。此时直接移动模式到好后缀的下一字符即可



「好后缀规则」

- 好后缀也预先定义一个 函数 ，来指明失配时如何移动模式
 - 设 $\text{shift}(j)$ 为 P 右移的距离， m 为模式串 P 的长度， j 为当前所匹配的字符位置

BM 算法的效率

- 预处理阶段的时间和空间复杂： $O(m+\sigma)$
- 匹配阶段的时间复杂度： $O(mn)$
 - 在匹配的过程中，取 $\text{skip}(x)$ 与 $\text{shift}(j)$ 中的较大者作为跳跃的距离
 - 最差情况：匹配不上，需要 $O(nm)$ 的时间
 - 最佳的性能可达 $O(n/m)$
 - 平均比较次数与模式的长度成反比
- 实际应用中非常高效，尤其在英语文本等的匹配中表现优异，因为BM算法在多数情况下只需比较文本的一小部分
 - 大多数文本编辑器中的search/substitute均采用BM算法

BM算法的改进版

- 基于BM算法有许多改进的版本
 - BMH算法
 - Sunday算法
 - BMB算法
 - BMH2算法
 -

关键在于在一趟匹配失配时**如何加大模式串的右移量**，以在一定程度上提高匹配效率

KMP vs BM

- 分别是前缀匹配和后缀匹配的经典算法
 - 前缀匹配是指模式和目标的比较从左到右，模式串的移动也是从左到右
 - 后缀匹配是指模式和目标的比较从右到左，模式串的移动也从左到右
- 均需对模式进行预处理
 - KMP：特征向量
 - BM：预先定义针对坏字符和好后缀的函数
- 改进版的BM在实际中获广泛应用

单模式的匹配算法小结

| 算法 | 预处理时间效率 | 匹配时间效率 |
|---------------------------------------|------------------------|---------------------------------|
| 朴素匹配算法 | $\Theta(1)$ (无需预处理) | $\Theta(nm)$ |
| KMP算法 | $\Theta(m)$ | $\Theta(n)$ |
| BM算法 | $\Theta(m +)$ | 最优 (n/m) , 最差 $\Theta(nm)$ |
| 位运算算法 <i>(shift-or, shift-and)</i> | $\Theta(m + \Sigma)$ | $\Theta(n)$ |
| Rabin-Karp算法 | $\Theta(m)$ | 平均 $(n+m)$, 最差 $\Theta(nm)$ |
| 有限状态自动机 | $\Theta(m \Sigma)$ | $\Theta(n)$ |

思考

- 请给出BM算法坏字符规则中辅助数组的定义
- 请给出BM算法好后缀规则的定义
- BM算法的正确性如何证明？

「参考资源

■ Pattern Matching Pointer

- <http://www.cs.ucr.edu/~stelo/pattern.html>

■ EXACT STRING MATCHING ALGORITHMS

- <http://www-igm.univ-lv.fr/~lecroq/string/>
- 字符串匹配算法的描述、复杂度分析和C源代码