



数据结构与算法

第三章 字符串

张铭

<http://db.pku.edu.cn/mzhang/DS/>

北京大学信息科学与技术学院
“数据结构与算法”教学小组

©版权所有，转载或翻印必究



主要内容

- 3.1 字符串抽象数据类型
- 3.2 字符串的存储结构和类定义
- 3.3 字符串运算的算法实现
- 3.4 字符串的模式匹配



3.1 字符串抽象数据类型

- 3.1.1 基本概念
- 3.1.2 String抽象数据类型



3.1.1 基本概念

- **字符串**，由0个或多个字符的顺序排列所组成的复合数据结构，简称“**串**”。
- **串的长度**：一个字符串所包含的字符个数。
 - **空串**：长度为零的串，它不包含任何字符内容。



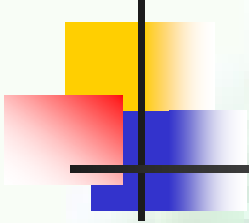
3.1.1.1 字符串常数和变量

- 字符串常数
 - 例如: `"\n"`
- 字符串变量



3.1.1.2 字符

- **字符(char)**：组成字符串的基本单位。
- 在C和C++中
 - 单字节（8 bits）
 - 采用**ASCII**码对**128**个符号（字符集**charset**）进行编码



3.1.1.4 C++ 标准string

- 标准字符串：将C++的 `<string.h>` 函数库作为字符串数据类型的方案。
 - 例如： `char S[M];`
- 串的结束标记： `'\0'`
 - `'\0'` 是ASCII码中8位BIT全0码，又称为 **NULL** 符。



3.1.1.4 C++ 标准string(续)

- 1. 串长函数

int strlen(char *s);

- 2. 串复制

char *strcpy(char *s1, char*s2);

- 3. 串拼接

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

- 4. 串比较

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



3.1.1.4 C++ 标准string(续)

- 5. 输入和输出函数
- 6. 定位函数

char * strchr(char *s, char c);

- 7. 右定位函数

char * strrchr(char *s, char c);



3.1.2 String抽象数据类型

- 字符串类（**class String**）：
 - 不采用**char S[M]**的形式
 - 而采用一种动态变长的存储结构。

`class String`

`//字符串 类`

`//它的存储结构和实现方法使用了C++标准string(简称标准串),
//为了区别,类String所派生创建的实例对象,简称‘本串’,或‘实例串’
//在程序首,要#include <string.h>和#include <iostream.h>及
// 及 #include <stdlib.h>, 以及#include <assert.h>`

`{`

`//1. 字符串的数据表示:`

`//字符串 S 通常用顺序存放,用数组S[]存储,元素的类型为char`

`//字符串为变长,使用变量size记录串的当前长度`

`// 2. 使用变量访问字符串:`

`//字符串变量能参与运算,例如S1 + S2表示两个字符串首尾拼接在一起`

`//用数组str[]存储字符串,在内部可以用str[i]访问串的第i个字符,`

`// 3. 字符串类的运算集: 请参看下面的成员函数`

```
private:
```

```
    char *str;        //私有的指针变量，用于指向存储向量str[size+1]
```

```
    int size;         //本串的当前实际长度
```

```
public:
```

```
    String(char *s = ''); //创建一个空的字符串
```

```
    String(char *s); // 创建新字符串，并将标准字符串s拷贝为初值
```

```
    ~String()         // 销毁本串，从计算机存储空间删去本串
```

```
//下面是函数的定义，包括赋值函数 = 拼接函数 + 和比较函数 < 等
```

```
    String& operator= (char *s) ; //赋值操作=，标准串s拷贝到本串
```

```
    String& operator= (String& s) ; //赋值操作=，串s复制到本串
```

```
    String operator+ (char *s) ; //拼接函数+，本串拼接标准串s
```

```
    String operator+ (String& s) ; //拼接函数+，本串拼接串s
```

```
    friend String operator+ (char *s1, String& s) ;
```

```
//友函数作为拼接函数+ 其返回值是一个实例串，等于标准串str拼接串s
```

```

//‘关系’函数，用于比较相等、大、小，例如
int operator< (char *s) ;//比较大小，本串小于标准串s则返回非0
int operator< (String& s) ;//比较大小，本串小于串s则返回非0
friend int operator< (char *s1, String& s) ; //友函数用于比较，
//      , 标准串s1小于串s, 则返回非0
//‘输入输出’函数 >>和<< 以及 读子串等，例如友函数
friend istream& operator>> (istream& istr, String& s) ;
friend ostream& operator<< (ostream& ostr, String& s) ;
// ‘子串函数’：插入子串、寻找子串、提取子串、删除子串等，例如
String Substr(int index, int count) ; //它们的功能参见下文
//‘串与字符’函数：按字符定位等，例如
int Find(char c, int start) ;//在本串中寻找字符c, 从下标start开始找，
// 寻找到c后, 返回字符c在本串的下标位置

//其他函数：求串长、判空串、清为空串、
int strlen() ; //返回本串的当前串长
int IsEmpty() ; //判本串为空串？
void clear() ; //清本串为空串
};

```

3.1.2.3 赋值操作符、拼接操作符和比较操作符

- 赋值操作符 =
- 拼接操作符 +
- 比较操作符 < <= >
>= != 和 ==

3.1.2.4 输入输出操作符

<< 和 >>

- 输入操作符 >>
- 输出操作符 <<

3.1.2.5 处理子串(Substring) 的函数

■ 简称“子串函数”

- 提取子串
- 插入子串
- 寻找子串
- 删除子串
- ...



3.1.2.6 字符串中的字符

- 重载下标操作符[]
char& operator[] (int n);
- 按字符定位下标
int Find(char c,int start);
- 反向寻找，定位尾部出现的字符
int FindLast(char c);

3.2 字符串的存储结构 和类定义



- 3.2.1 字符串的顺序存储
- 3.2.2 字符串类class String的存储结构





3.2.1 字符串的顺序存储

- 用**S[0]**作为记录串长的存储单元
 - 缺点：串的最大长度不能超过**256**
- 另辟一个存储的地方存储串的长度
 - 缺点：串的最大长度静态给定
- 用一个特殊的末尾标记'**\0**'
 - 例如：**C++**语言的**string**函数库（**#include <string.h>**）采用这一存储结构

3.2.2 字符串类class String 的存储结构(续)

- 微软VC++的CString类介绍
 - 自动的动态存储管理，串的最大长度不超过2GB
 - 容器型
 - 不必使用new和delete
- 使用特点：
 - 变量申明
 - `CString s6('x', 6);` `// s6 = "xxxxxx"`
 - `CString city = "Philadelphia";` `// 串常数作为初值`
 - 赋值语句
 - `s1 = s2 = "hi there";`
 - 变量比较 `if(s1 == s2)`
 - 方法调用 `s1.MakeUpper();`
 - 内部字符比较 `if(s2[0] == 'h')`



3.3 字符串运算的算法实现

- 1. 串长函数

```
int strlen(char *s);
```

- 2. 串复制

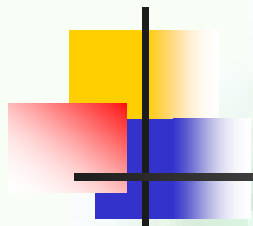
```
char *strcpy(char *s1, char*s2);
```

- 3. 串拼接

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

- 4. 串比较

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



```
int strcmp_1(char *d, char *s) {  
    for (int 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]);  
}
```



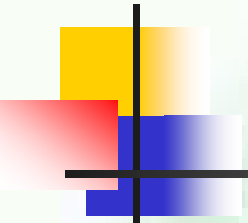
3.3 字符串运算的算法实现

- 3.3.1 C++标准串运算的实现
- 3.3.2 String串运算的实现



3.4 字符串的模式匹配

- 模式匹配(pattern matching)
 - 一个目标对象**S**（字符串）
 - 一个模板（**pattern**）**P**（字符串）
- 任务：求出**S**中第一个与**P**全同匹配的子串（简称为“**配串**”），返回其首字符位置



$$\begin{array}{ccccccc}
 \textcolor{red}{S} & s_0 & s_1 & \cdots & s_i & s_{i+1} & s_{i+2} \cdots s_{i+m-2} & s_{i+m-1} & \cdots & s_{n-1} \\
 & & & & \parallel & \parallel & \parallel & \parallel & \parallel & \\
 \textcolor{red}{P} & & & & p_0 & p_1 & p_2 & \cdots & p_{m-2} & p_{m-1}
 \end{array}$$

为使模式 $\textcolor{red}{P}$ 与目标 $\textcolor{red}{S}$ 匹配，必须满足

$$p_0 p_1 p_2 \cdots p_{m-1} = s_i s_{i+1} s_{i+2} \cdots s_{i+m-1}$$



3.4 字符串的模式匹配

- 3.4.1 朴素模式匹配
- 3.4.2 字符串的特征向量N
- 3.4.3 KMP模式匹配算法

朴素模式匹配

S= a b a b a b a b a b a b b

P= a b a b a b ~~b~~

~~a~~ b a b a b b

a b a b a b ~~b~~

~~a~~ b a b a b b

a b a b a b ~~b~~

~~a~~ b a b a b b

a b a b a b b

back

next



朴素模式匹配代码（简洁）

```
int FindPat_2(String S, String P, int startindex) {  
    //g为S的游标，用模板P和S第g位置子串比较，  
    //若失败则继续循环  
    for (int g= startindex; g <= S.strlen() - P.strlen();  
        g++) {  
        for (int j=0; ((j<P.strlen()) && (S[g+j]==P[j]));  
            j++) ;  
        if (j == P.strlen())  
            return g;  
    }  
    return(-1); // for结束，或startindex值过大,则匹配失败  
}
```



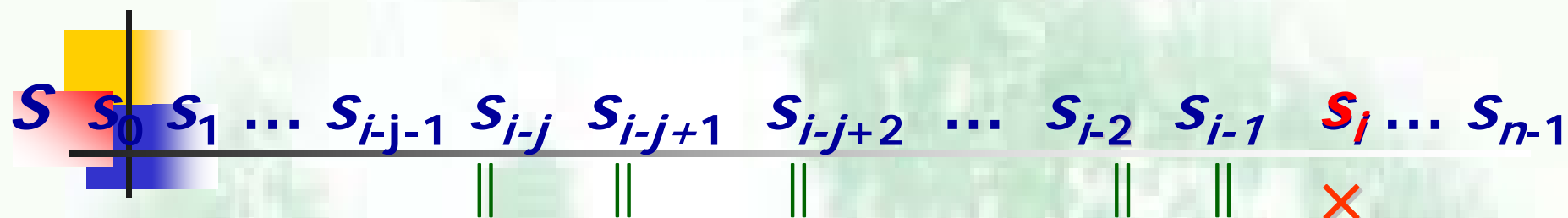


朴素模式匹配算法代价

例如, **aaaaaaaaaab**
aaaaaab

- 目标**S**的长度为**n**, 模板**P**长度为**m**, $m \leq n$
- 在最坏的情况下, 每一次循环都不成功, 则一共要进行比较 $(n-m+1)$ 次
- 每一次“相同匹配”需要**P**和**S**逐个字符比较的时间, 最坏情况下, 共**m**次
- 整个算法的最坏时间开销估计为 $O(m \cdot n)$

KMP算法思想



P
 $p_0 \quad p_1 \quad p_2 \quad \dots \quad p_{j-2} \quad p_{j-1} \quad p_j$
 则有 $s_{i-j} s_{i-j+1} s_{i-j+2} \dots s_{i-1} = p_0 p_1 p_2 \dots p_{j-1}$ (1)

朴素下一趟 $p_0 \quad p_1 \quad p_2 \quad \dots \quad p_{j-2} \quad p_{j-1}$
 如果 $p_0 p_1 \dots p_{j-2} \neq p_1 p_2 \dots p_{j-1}$ (2)

则立刻可以断定

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

(朴素匹配的)下一趟一定不匹配，可以跳过去

$p_0 \quad p_1 \quad p_2 \quad \dots \quad p_{j-2} \quad p_{j-1}$



同样，若 $p_0 p_1 \dots p_{j-3} \neq p_2 p_3 \dots p_{j-1}$
 则再下一趟也不匹配，因为有

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

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

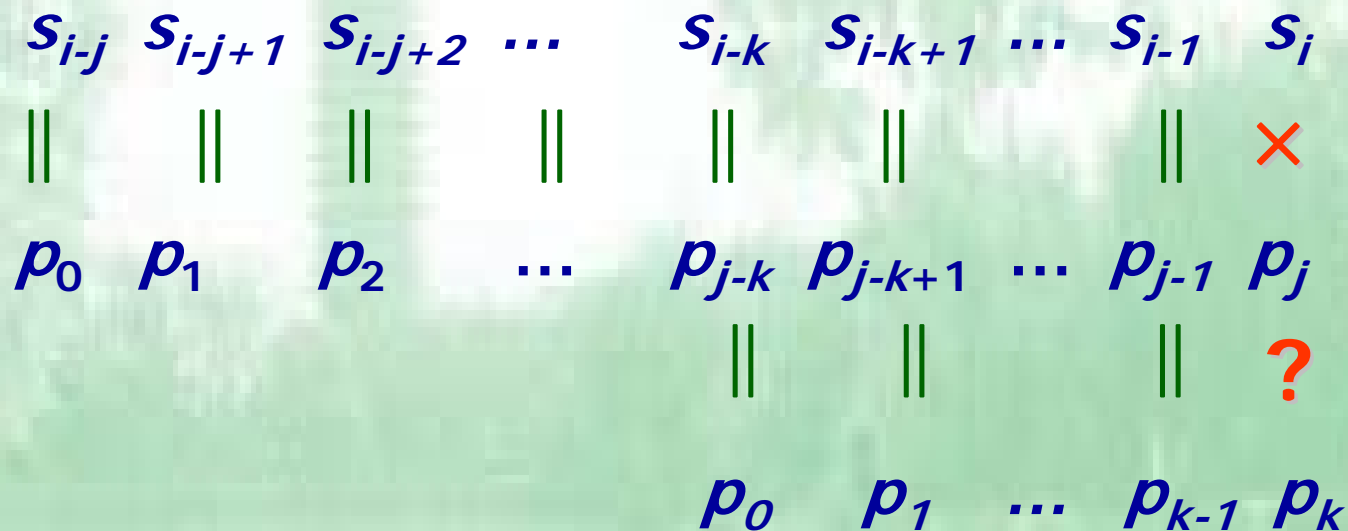
$$\begin{array}{ccccccc} s_{i-k} & s_{i-k+1} & \dots & s_{i-1} & s_i \\ \parallel & \parallel & & \parallel & \times \end{array}$$

$$\begin{array}{ccccccc} p_{j-k} & p_{j-k+1} & \dots & p_{j-1} & p_j \\ \parallel & \parallel & & \parallel & ? \end{array}$$

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

← back 则 $p_0 p_1 \dots p_{k-1} = s_{i-k} s_{i-k+1} \dots s_{i-1}$ next →

模式右滑 $j-k$ 位

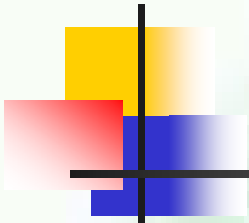


$$p_0 \ p_1 \ \dots \ p_{k-1} = s_{i-k} \ s_{i-k+1} \ \dots \ s_{i-1}$$

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

back

next



3.4.2 字符串的特征向量N

- 设模板P由m个字符组成：
记为 $P = q_0 q_1 q_2 q_3 \dots q_{m-1}$
- 特征向量N用于表示模板P的字符分布特征
 - $N = n_0 n_1 n_2 n_3 \dots n_{m-1}$
 - m个非负整数



特征 n_i 的含义

- P的**首子串** $q_0q_1q_2\cdots q_{t-1}$
- P的**尾子串** $q_{i-t+1}\cdots q_{i-2}q_{i-1}q_i$
- 特征数 n_i
 - 最长的（ t 最大的）首尾子串能够匹配的 t



特征数 n_i 的递归定义

- ① $n_0 = 0$,
对于 $i \geq 1$ 的 n_i , 假定已知前一位置的特征数 n_{i-1} , 并且 $n_{i-1} = k$;
- ② 如果 $q_i = q_k$, 则 $n_i = k + 1$;
- ③ 当 $q_i \neq q_k$ 且 $k \neq 0$ 时,
则令 $k = n_{k-1}$;
让③循环直到条件不满足;
- ④ 当 $q_i \neq q_k$ 且 $k = 0$ 时, 则 $n_i = 0$;

N =

0 1 2 3 0 1 2 3 4 0

P =

a a a a b a a a a c

前缀 →

a

前缀 →

a

a

前缀 →

a

a

a

前缀 →

a

前缀 →

a

a

前缀 →

a

a

a

前缀 →

a

a

a

a

back

求特征向量N

next

求特征向量N(例2)

$P =$

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

$N =$

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 2 | 3 | 4 | 5 | 0 | 1 | 2 | 3 | 1 | 1 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|

←back

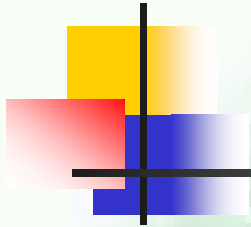
next→



【算法3-15】 计算特征向量N

```
int *Next(String P)
{
    int m = P.strlen();    //m为模板P的长度
    assert( m > 0);        //若m=0, 退出
    int *N = new int[m];    // 动态存储区开辟整数数组
    assert( N != 0);        //若开辟存储区域失败, 退出
    N[0] = 0;

    for( int i = 1 ; i < m ; i++ ) //分析P的每个位置i
    {
        int k = N[i-1]; //第(i-1)位置的最长前缀串长度
```



```
//以下while语句递推决定合适的前缀位置k  
while( k > 0 && P[i] != P[k] )  
    k = N[k-1];
```

```
//根据P[i]比较第k位置前缀字符，决定N[i]  
if(P[i] == P[k])  
    N[i] = k+1;  
else  
    N[i] = 0 ;  
}
```

```
return N;
```

```
}
```





【算法3-16】 KMP模式匹配

```
int KMP_FindPat(String S, String P, int *N, int
    startindex) {
    //假定事先已经计算出P的特征数组N，作为输入参数

    // S末尾再倒数一个模板长度位置
    int LastIndex = S.strlen() - P.strlen();
    if ((LastIndex - startindex) < 0)
        return (-1);    //startindex过大，匹配无法成功

    int i;                // i 是指向S内部字符的游标，
    int j = 0;            // j 是指向P内部字符的游标，
```

back

next

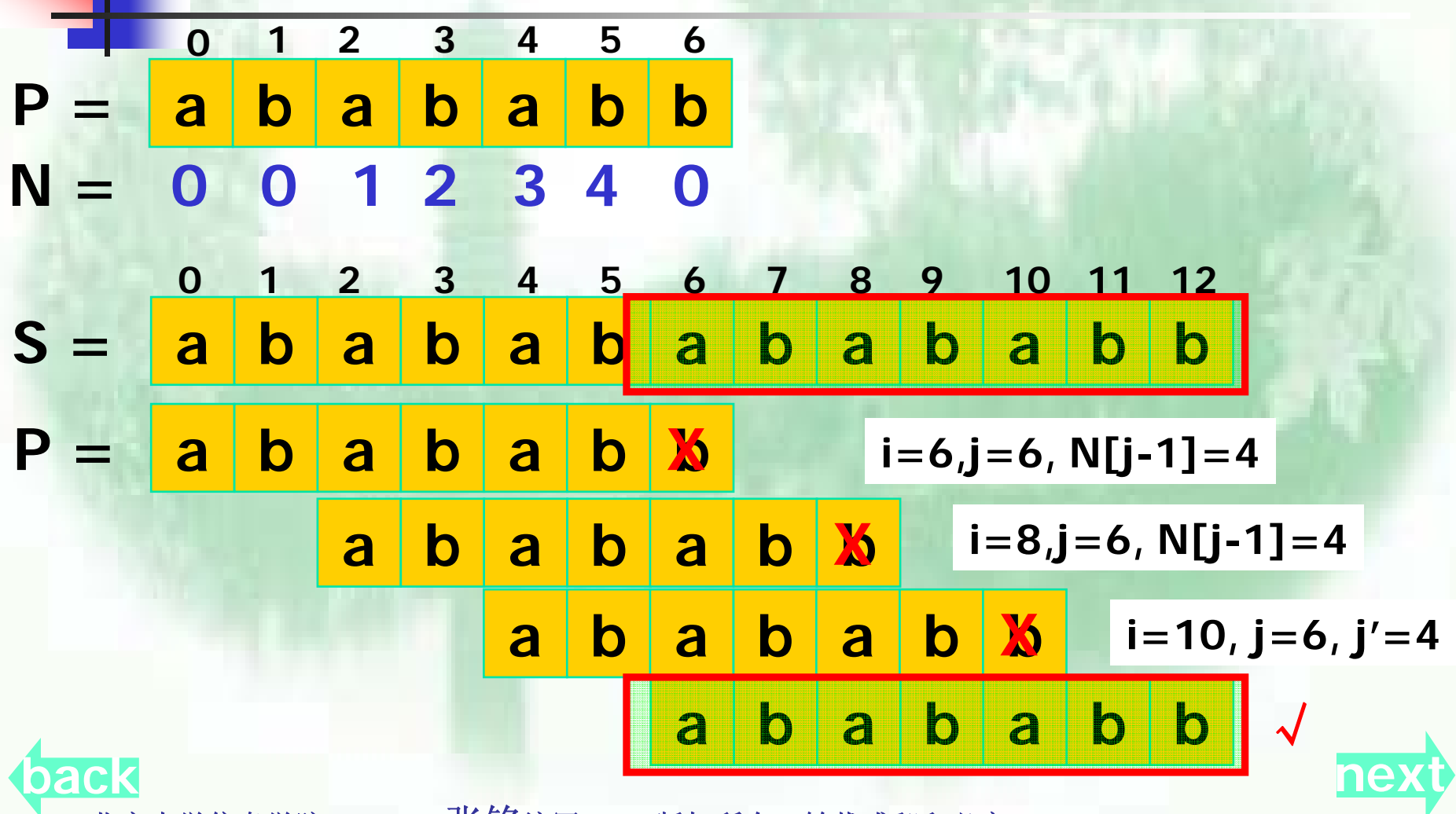
3.4.3 KMP模式匹配算法效率

```
// S游标i循环加1
for (i = startindex; i < S.strlen(); i++) {
    //若当前位置的字符不同，则用N循环求当前的j，
    //用于将P的恰当位置与S的i位置对准
    while (P[j] != S[i] && j > 0)
        j = N[j-1];
    //P[j]与S[i]相同，继续下一步循环
    if (P[j] == S[i]) j++;
    //匹配成功，返回该S子串的开始位置
    if (j == P.strlen())
        return (i - j + 1); }
return (-1); //P和S整个匹配失败，函数返回值为负
```

back

next

KMP模式匹配示例（一）



back

next

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

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | |
|-----|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|---|
| S = | a | a | b | a | a | a | a | a | a | b | a | a | a | a | c | b |

| | | | | | | | | | | | | | | | | |
|-----|---|---|--------------|---|---|---|---|---|---|---|--------------------|--|--|--|--|--|
| P = | a | a | a | a | b | a | a | a | a | c | i=2, j=1, N[j-1]=0 | | | | | |
|-----|---|---|--------------|---|---|---|---|---|---|---|--------------------|--|--|--|--|--|

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

i=7, j=4, N[j-1]=3

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

i=8, j=4, N[j-1]=3

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

✓

← back

next →



KMP算法的效率

■ 两重循环

- **for**循环最多执行 $n = S.\text{strlen}()$ 次
 - 其内部的**while**循环，最长循环次数是 $m = P.\text{strlen}()$ 次。
- 初看起来其时间开销也可能达到 $O(n \times m)$ 。

- 循环体中“ $j=N[j-1];$ ”语句的执行次数不能超过 n 次。否则，
 - 由于“ $j=N[j-1];$ ”每执行一次必然使得 j 减少(至少减1)
 - 而使得 j 增加的操作只有“ $j++$ ”
 - 那么，如果“ $j=N[j-1];$ ”的执行次数超过 n 次，最终的结果必然使得 j 为负数。这是不可能的。
- 同理可以分析出求 $next$ 数组的时间为 $O(m)$
- 因此，**KMP**算法的时间为 $O(n+m)$

另一种 $next$ 数组方法

| | | | | | | | | | |
|---------------|----|--------|--------|------|--------|------|--------|------|------|
| ■ 序号 <i>i</i> | 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 |
| $p_k == p_i?$ | | \neq | \neq | $==$ | \neq | $==$ | \neq | $==$ | $==$ |
| next[i] | -1 | 0 | 0 | -1 | 1 | 0 | 2 | 0 | 0 |

KMP匹配

| | | | | | | | | | |
|---------------|----|--------|--------|-----|--------|-----|--------|-----|-----|
| 序号i | 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 |
| $p_k == p_i?$ | | \neq | \neq | $=$ | \neq | $=$ | \neq | $=$ | $=$ |
| next[i] | -1 | 0 | 0 | -1 | 1 | 0 | 2 | 0 | 0 |

目标

a a b c b a b c a a b c a a b a b c

a ~~b~~ c a a b a b c

a b c a ~~a~~ b a b c

a b c a a b a b c ~~c~~

a b c a a b a b c

$next(1) = 0$

$next(3) = -1$

$next(6) = 2$

✓

back

next

KMP匹配

| | | | | | | | | | |
|-------|---|---|---|---|---|---|---|---|---|
| ■ 序号i | 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 |

第1趟 目标 *a a b c b a b c a a b c a a b a b c*
 模式 *a ~~b~~ c a a b a b c* $next(0) = 0$

第2趟 目标 *a a b c b a b c a a b c a a b a b c*
 模式 *a b c ~~a~~ a b a b c* $next(2) = 0$

第3趟 目标 *a a b c b a b c a a b c a a b a b c*
 模式 *~~a~~ b c a a b a b c* $j = 0$

第4趟 目标 *a a b c b a b c a a b c a a b a b c*
 模式 *a b c a a b ~~a~~ b c* $next(5) = 2$

第5趟 目标 *a a b c b a b c a a b c a a b a b c*
 模式 *(a b) c a a b a b c*

back

next

*next*数组对比

| | | | | | | | | | |
|---------------|----|--------|--------|------|--------|------|--------|------|------|
| ■ 序号i | 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 |
| $p_k == p_i?$ | | \neq | \neq | $==$ | \neq | $==$ | \neq | $==$ | $==$ |
| next[i] | -1 | 0 | 0 | -1 | 1 | 0 | 2 | 0 | 0 |

| | | | | | | | | | |
|-------|---|---|---|---|---|---|---|---|---|
| ■ 序号i | 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 |

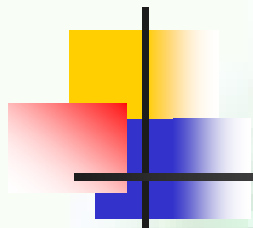
back

next



总结

- 字符串抽象数据类型
- 字符串的存储结构和类定义
- 字符串运算的算法实现
- 字符串的模式匹配
 - 特征向量N及相应的KMP算法还有其他变种、优化
 - <http://www.db.pku.edu.cn/mzhang/site/index.htm>



Thank you!

祝大家学习进步！

<http://db.pku.edu.cn/mzhang/DS/>

张铭: mzhang@db.pku.edu.cn

