



第四章 字符串

宋国杰

gjsong@pku.edu.cn



一类内容受限的线性表

课程内容

4.1 基本概念

4.2 存储结构

4.3 模式匹配--KMP

4.1 基本概念

➤ 字符串

➡ 由 0 或多个字符顺序排列所组成的有限序列，称“串”

➡ 串长度：字符串所包含的字符个数

– 空串：长度为零的串 “”；空格串：“_”

➤ 字符串常数：双撇号“*xxxx*”作左右括号，中间字符个数有限

➡ 如 “SUNDAY”，“123”，“字符串” 等

➤ **字符(char)**：组成字符串的基本单位，取值依赖于字符集 Σ （同线性表，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： $\Sigma = \{26\text{个字符}, \text{标点符号}\}$
- ➡

➤ **字符编码**

- ➡ 字符类型是单字节 (8 bits) 类型
- ➡ 采用ASCII码对128个符号（字符集charset）进行编码
- ➡ 国标编码GB2312、通用文字符号编码标准UNICODE

网页怎么又乱码了?



“怎么比较字符串大小？”

- 为了字符串间比较和运算的便利，字符编码表一般遵循“**偏序编码规则**”
- **字符偏序**：根据字符的自然含义，某些字符间两两可以比较次序
 - 大多数情况下就是字典序
 - 中文字符串有些特例，例如“笔划”序

`encode('0')+1=encode('1')`

`“monday” < “sunday”`

注：encode()是符号的ASCII编码(序号)的映射函数

子串

➤ 假设 s_1, s_2 是两个串

➡ $s_1 = a_0a_1a_2\dots a_{n-1}$ 和 $s_2 = b_0b_1b_2\dots b_{m-1}$, ($0 \leq m \leq n$)

➡ 若存在整数 i ($0 \leq i \leq n-m$), 使得 $b_j = a_{i+j}$, $j = 0, 1, \dots, m-1$ (连续相等)

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 S 本身的子串


➤ 子串处理函数

➡ 提取子串

➡ 插入子串

➡ 删除子串

➡ ...




➤ 若字符串 $s = \text{"software"}$ ，则其子串（真子串）
的数目为多少？

答案：37 (35)

抽象数据类型

```
class string {  
    private:                // 字符串的存储结构在具体实现时定义  
        char *str           // 字符串的数据表示  
        int size           // 串的当前长度  
    public:                 // 字符串的运算集  
        string(char *s = ""); // 创建一个空字符串  
        string(char *s);      // 创建一个初值为s的字符串  
        ~string ();           // 消除该串实例  
        int length();         // 返回串的长度  
        int isEmpty();        // 判断串是否为空串  
};
```



```
void clear();           // 把串清空
string append(char c);  // 在串尾添加字符
string concatenate(char *s); // 把串s连接在本串后面
string copy(char*s);    // 将一个串s拷贝到本串
string insert(char c, int index); // 往串中给定位置插字符
// 从位置start开始搜索串寻找一个给定字符
int find(char c, int start);
// 从位置s开始提取一个长度为len的子串
string substr(int s, int len);
}
```

4.2 字符串的存储实现

1. 静态存储

- ➡ 采用char S[M]的形式（**字符数组**）
- ➡ 长度固定
- ➡ 不是面向对象数据类型，存在局限性

2. 动态存储

- ➡ 采用String类
- ➡ 一种动态变长的存储结构

1. 静态存储：C++标准字符串

➤ 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

➡ 采用 `char S[M];` 的形式定义字符串变量

➡ M是常量，保持不变

➤ 串的结束标记：'\0'

➡ '\0'是ASCII码中8位BIT全0码，又称为NULL

➡ 长度为M的字符串实际最大容量的为(M-1)

`char s1[7]="value." ;`

V	a	l	u	e	.	<u>\0</u>
---	---	---	---	---	---	-----------

`char s2[9];`

<u>\0</u>								
-----------	--	--	--	--	--	--	--	--

s1=s2?

标准字符串函数

- **串长函数** // 返回字符串s的长度;

`int strlen(char *s);`

- **串复制** // 将s2值复制给s1, 返回指针指向s1;

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

- **串拼接** // 将串s2拼接到s1的尾部

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

- **串比较**

`int strcmp(char *s1, char *s2); (=, >, <)`

- **(左)定位函数** // c在s中第一次出现的位置;

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

- **右定位函数** // 逆向寻找c在s中第一次出现的位置;

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

举例

e.g, 字符串s :

s	H	e	l	l	o		w	o	r	l	d	\0
	0	1	2	3	4	5	6	7	8	9	10	11
					↑			↑				
					strchr(s,'o')			strrchr(s,'o')				

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

算法实现

➤ 字符串的比较

```
int strcmp( char *d, char *s) {           // int i =0;

    while (s[i] != '\0' && d[i] != '\0' ) {

        if (d[i] > s[i])

            return 1;

        else if (d[i] < s[i])

            return -1;

        i ++;

    }
```




```
if( d[i] == '\0' && s[i] != '\0')  
    return -1;  
  
else if (s[i] == '\0' && d[i] != '\0')  
    return 1;  
  
return 0;  
  
}
```

➤ 寻找字符

`char * strchr(char *d , char ch)` // 查找ch在d中第一次出现的位置

```
{
```

```
    i = 0;
```

```
    while (d[i] != '\0' && d[i] != ch ) i++; // 跳过不是ch的字符
```

```
    if (d[i] == '\0')
```

```
        return '\0'; // 表示失败
```

```
    else
```

```
        return &d[i] ;
```

```
}
```

➤ 反向寻找字符

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

```
{
```

```
    i = 0;
```

```
    while (d[i] != '\0' ) i++;
```

//找串尾

```
    while (i >= 0 && d[i] != ch ) i- - ;
```

//跳过不是ch的字符

```
    if (i < 0)
```

```
        return '\0' ;
```

```
    else
```

```
        return &d[i] ;
```

```
}
```

2. 动态存储：字符串类

➤ 字符串类（`class String`）

- 适应字符串长度动态变化的复杂性
- 不再以字符数组`char S[M]`的形式出现，而采用一种动态变长的存储结构
- 不适合采用动态链表的形式

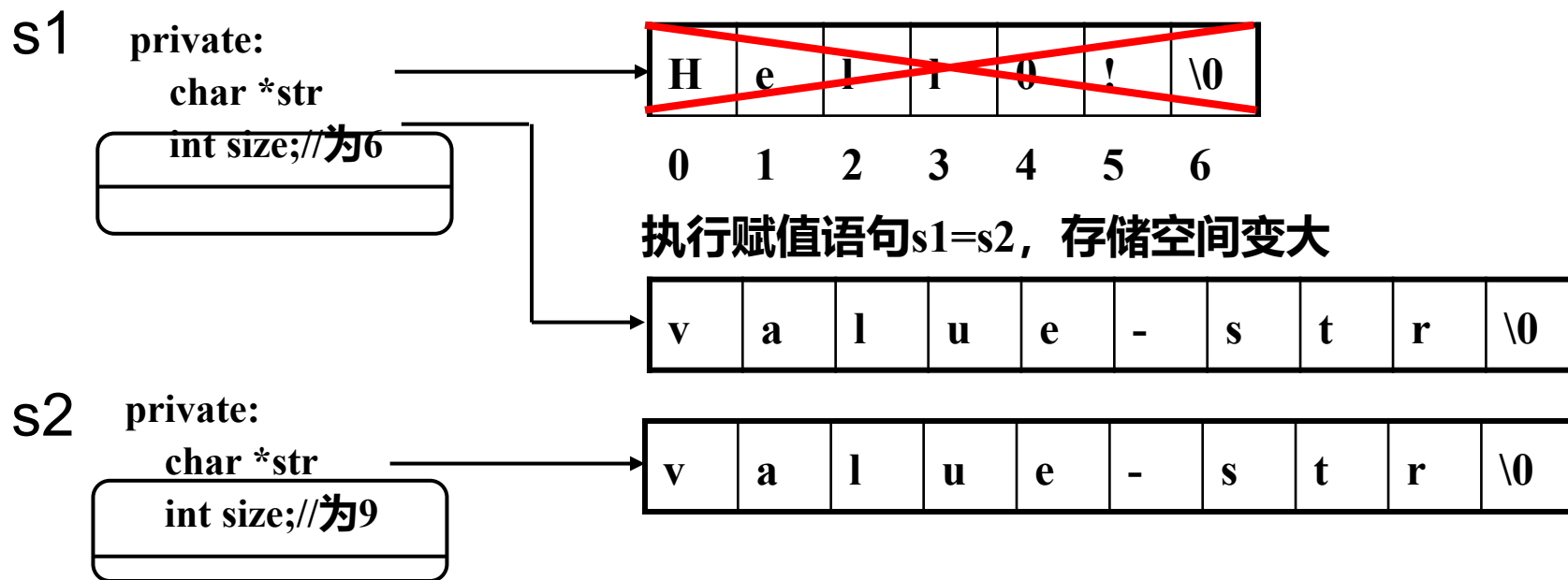
构造与赋值算子

➤ 构造函数String::String(char *s)

➡ String s1="hello!"

➤ 赋值算子String String::operator=(String& s)

➡ String s2="value-str"; s1=s2



String串运算的实现

➤ 构造算子(constructor)

```
String::String(char *s)
```

```
{ //先要确定新创建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。确定s  
  的长度，用标准字符串函数strlen(s)计算长度
```

```
  size = strlen(s);
```

```
  //然后在动态存储区开辟一块空间存储初值s，把结束字符也包括进来;
```

```
  str = new char [size +1];
```

```
  //开辟空间不成功时，运行异常，退出
```

```
  assert(str != NULL);
```

```
  //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 );
```

```
}
```

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];
```

```
        assert(str != NULL); //若开辟动态存储空间失败, 则退出正常运行
```

```
        size = s.size;
```

```
    }
```

```
    strcpy(str , s.str );
```

```
    return *this; //返回本实例, 作为String类的一个实例
```

```
}
```

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 ;           //自index到串尾的有效长度;
    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) // 注意不是 index >= size - 1
        return temp;
    if(count > left ) //若count超过自index以右的实际子串长度，则count变小
        count = left;
```




```
temp.str = new char [count+1];
```

```
p = temp.str; //p的内容是指针，指向目前暂无内容的字符数组首字符处
```

```
q = &str[index]; //q的内容是一指针，指向本实例串的str数组的下标index
```

```
//q取出所指内容后，指针加1，用p所指的字符单元接受拷贝，指针加1
```

```
for (i = 0; i < count; i++)
```

```
    *p++ = *q++;
```

```
*p = ' \0'; //循环结束后，让temp.str的结尾为' \0'
```

```
temp.size = count;
```

```
return temp;
```

```
}
```

String类部分算子列表

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
	=	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

4.3 字符串的模式匹配

➤ 模式匹配 (Pattern Matching)

- ➡ 一个目标对象T (字符串)
- ➡ 一个模板 (pattern) P (字符串)

➤ 任务

- ➡ 用给定的模板P, 在目标字符串T中搜索与模板P全相同的一个子串, 并返回P 和 T匹配的第一个子串的首字符位置

➤ 举例

- ➡ T = “e~~asdk~~nje~~asdk~~”, P=“asdk”
- ➡ 返回1

意义

- 是计算机科学中最古老、研究最广泛的问题之一
- 有着大量的实际应用
 - ➡ 生物信息学、信息检索、拼写检查、数据压缩检测等
- 例如
 - ➡ 在信息检索中，一个挑战性的任务是，搜索出由用户自定义的模式对应文本中的匹配位置，这种模式很可能带有通配符。
 - ➡ 在生物信息学中，相似基因序列的比对。
- 大数据的搜索代价不容小觑！

分类

➤ 精确匹配 (Exact String Matching)

- ➡ 如果在目标T中至少一处存在模式P，则称为匹配成功。
 - 单选: “Set”
 - 多选: “S?t”
 - 正则表达式: 通常被用来检索、替换那些符合某个模式(规则)的文本

➤ 举例

- ➡ $L(a|b) = \{a, b\}$
- ➡ $L((a|b)(a|b)) = \{aa, ab, ba, bb\}$
- ➡ $L(a^*) = \{\epsilon, a, aa, aaa, aaaa, \dots\}$
- ➡ $L((a|b)^*) = \{\epsilon, a, b, aa, ab, ba, bb, aaa, aab, \dots\}$
- ➡ $L(a|a^*b) = \{a, b, ab, aab, aaab, \dots\}$

分类

➤ 近似匹配 (Approximate String Matching)

➡ 如果模式P与目标T的某个子串某种程度相似，则认为匹配成功

- 常用衡量字符串相似度的方法是根据一个串转换成另一个串所需的基本操作数目来确定。 (**编辑距离 Edit Distance**)
- 基本操作由字符串的**插入、删除和替换**三种操作来组成

➤ 举例

- ➡ $d(\text{"abc"}, \text{"abd"}) = 1$
- ➡ $d(\text{"abc"}, \text{"ab"}) = 1$
- ➡ $d(\text{"abc"}, \text{"abcdf"}) = 2$
- ➡ $d(\text{"serverU"}, \text{"ser-u"}) = 4$

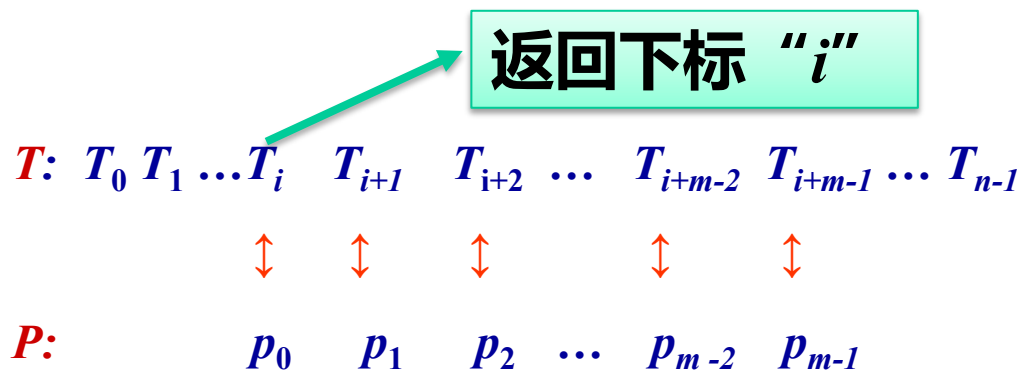
在数据挖掘领域的应用

➤ 序列模式挖掘

DATABASE			FREQUENT SET (minsup = 75%)	
SID	EID	Items		
1	10	A B	F1	
	20	B	A	4
	30	A B	B	4
2	20	A C	F2	
	30	A B C	AB	3
	50	B	A→A	4
3	10	A	A→B	4
	30	B	B→A	3
	40	A	B→B	3
4	30	A B	F3	
	40	A	AB→B	3
	50	B		

字符串的模式匹配

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



- 为使模式 P 与目标 T 匹配，必须满足

$$p_0 p_1 p_2 \dots p_{m-1} = T_{i+0} T_{i+1} T_{i+2} \dots T_{i+m-1}$$

模式匹配算法

算法	预处理时间	匹配时间
Naïve string search algorithm	0 (no preprocessing)	$\Theta(n*m)$
Rabin-Karp string search algorithm	$\Theta(m)$	average $(n+m)$, worst $\Theta(n*m)$
Finite automaton	$\Theta(m \Sigma)$	$\Theta(n)$
Knuth-Morris-Pratt algorithm	$\Theta(m)$	$\Theta(n)$
Boyer-Moore algorithm	$\Theta(m)$	average (n/m) , worst $\Theta(n)$
Bitap algorithm	$\Theta(m+ \Sigma)$	$\Theta(n)$
Shift Or Algorithm	$\Theta(m + \Sigma)$	$\Theta(n)$

朴素算法

➤ 朴素模式匹配算法 (Brute Force)

- ➡ 从主串T的第一个字符起和模式串P的第一个字符进行比较
- ➡ 若相等，则继续比较T和P的后续字符
- ➡ 否则，从主串T的第“2”个字符起再重新和模式P的第“1”个字符进行比较

朴素模式匹配过程

1) 从首位置开始匹配

$T_0 \quad T_1 \quad T_2 \quad \dots \quad T_{m-1} \quad T_m \quad \dots \quad T_{n-1}$

$\uparrow \quad \uparrow \quad \uparrow \quad \quad \quad \uparrow$

$P_0 \quad P_1 \quad P_2 \quad \dots \quad P_{m-1}$

2) 模式右移一位开始匹配

$T_0 \quad T_1 \quad T_2 \quad \dots \quad T_{m-1} \quad T_m \quad \dots \quad T_{n-1}$

$\quad \uparrow \quad \uparrow \quad \uparrow \quad \quad \quad \uparrow$

$\quad P_0 \quad P_1 \quad P_2 \quad \dots \quad P_{m-1}$

3) 匹配成功:

$T(j \dots j+m-2) = P(0 \dots m-1)$

$T_0 \quad T_1 \quad \dots \quad T_j \quad T_{j+1} \quad T_{j+2} \quad \dots \quad T_{j+m-2} \quad T_{j+m-1} \quad \dots \quad T_{n-1}$

$\quad \quad \quad \parallel \quad \parallel \quad \parallel \quad \quad \quad \parallel \quad \parallel$

$\quad \quad \quad P_0 \quad P_1 \quad P_2 \quad \dots \quad P_{m-2} \quad P_{m-1}$

4) 匹配失败:

当 T_{n-m+1} 开始与 P_0 比较

$T_0 \quad T_1 \quad \dots \quad T_j \quad T_{j+1} \quad \dots \quad T_{n-m+1} \quad T_{n-m+2} \quad \dots \quad T_{n-1}$

$\quad \quad \quad P_0 \quad P_1 \quad \quad \quad \dots \quad P_{m-2} \quad P_{m-1}$

朴素模式匹配：过程例子

T=	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	b					
			a	b	a	b	a	b	✗				
				✗	b	a	b	a	b	b			
					a	b	a	b	a	b	✗		
						✗	b	a	b	a	b	b	
							a	b	a	b	a	b	b

- 把模式与目标逐一进行比较，直到碰到不匹配的字符为止
- 算法可在第一个匹配或是目标的结束处停止

朴素匹配算法 - I

```
int FindPat_1(string T, string P, int startindex) {  
    int LastIndex = T.length() - P.length();           //从S末尾倒数一个模式长度位置  
    int count = P.length();  
    if (startindex > LastIndex)                          //开始匹配位置startindex值过大，无法匹配  
        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 == T.substr(g, count))  
            return g;  
    }  
    return (-1);  
}
```

朴素匹配算法 - II

```
int FindPat_2 (string T, string P,int startindex) {  
    int LastIndex = T.length() - P.length();    //从T末尾倒数一个模板长度位置  
    if (startindex > LastIndex) return (-1);    //开始匹配位置值过大无法匹配  
    int i = startindex, j = 0;    // i 是T内部游标, j 是P内部游标  
    while (i < T.length() && j < P.length())  
        if (P[j] == T[i])    { i++; j++; }  
        else    { i = i - j + 1; j = 0; }  
    if (j >= P.length())  
        return (i - j);    //若匹配成功, 则返回该T子串的开始位置;  
    else return -1;    //若失败, 函数返回值为负  
}
```

朴素匹配算法 - III

```
int FindPat_3(string T, string P, int startindex) {  
    //g为T的游标, j为P的游标  
    for (int i= startindex; i <= T.length() - P.length(); i++) {  
        for (int j=0; ((j<P.length()) && (T[i+j]==P[j])); j++)  
            ;  
        if (j == P.length())  
            return i;  
    }  
    return(-1); // for结束, 或startindex值过大,则匹配失败  
}
```

朴素匹配算法：最佳情况

- 在目标的前M个位置上找到模式，设 $M = 5$

AAAAA AAAAAAAAAAAAAAAAAAH

AAAAA 5次比较

- 总比较次数：M
- 时间复杂度： $O(M)$

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

- 目标形如 a^n , 模式形如 $a^{m-1}b$

- 总比较次数:

$$M(N-M+1)$$

- 时间复杂度:

$$O(M*N)$$

AAAAA**A**AAAAAAAAAAAAAAAAAAAAAH

AAAA**AH** 5次比较

A**AAAAA**AAAAAAAAAAAAAAAAAAAAAAH

AAAA**AH** 5次比较

AA**AAAAA**AAAAAAAAAAAAAAAAAAAAAAH

AAAA**AH** 5次比较

AAA**AAAAA**AAAAAAAAAAAAAAAAAAAAAAH

AAAA**AH** 5次比较

.....

AAAAAAAAAAAAAAAAAAAA**AAAAAH**

AAAA**AH**

朴素算法性能分析

➤ 分析

- ➡ 假定目标T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，则串p一共右移 $(n-m+1)$ 次
- ➡ 每一次“相同匹配”比较所耗费的时间，是P和T逐个字符比较的时间，最坏情况下，共 m 次。
- ➡ 因此，整个算法的最坏时间开销估计为 $O(m * n)$



一起观察分析下面的例子

➤ 举例

0 1 2 3 4 5 6 7 8 9 10....
T: a b a c a a b a c c a b a c a b a a
P: a b a c a b

1) $P_5 \neq T_5$ P右移一位

T: a b a c a a b a c c a b a c a b a a
P: a b a c a b

2) $P_0 \neq T_1$ P右移一位

T: a b a c a a b a c c a b a c a b a a
P: a b a c a b

3) $P_1 \neq T_3$ P右移一位

T: a b a c a a b a c c a b a c a b a a
P: a b a c a b

4) $P_0 \neq T_3$ P右移, 得到 $P_0 = T_4$

T: a b a c a a b a c c a b a c a b a a
P: a b a c a b

直到 $P_0 = T_4$, 开始 $P_1 = T_5$

➤ 分析

➡ 由右例可知: $P_5 \neq T_5$,

$P_0 P_1 P_2 P_3 P_4 = T_0 T_1 T_2 T_3 T_4$, 同时由 $P_0 \neq T_1$ 可知,
将P右移一位后第2) 趟比较一定不等


➡ 比较操作: $P_0 \neq T_1$ 、 $P_1 \neq T_3$ 、 $P_0 \neq T_3$ 、 $P_0 = T_4$ 都是可以从 $P_0 P_1 P_2 P_3 P_4$ 推断出来的,

➡ $P_1 = T_5$ 才是有效比较, 其余比较都是冗余


消除冗余比较而不丢失配串

希望能达到如此效果

第一次匹配


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

第二次匹配


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

第三次匹配

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

无回溯匹配

- 寻找一个无回溯（**T的游标不后退**）的匹配算法，关键在于匹配过程中，一旦 **P_i 和 T_j 比较不等时**，即

$$P.\text{substr}(0, i) = T.\text{substr}(j-i+1, i) \quad \text{且} \quad P_i \neq T_j$$

- 要能立即**确定右移的位数**，即该用P中的哪个字符和**当前T中失配位 T_j** 进行比较？

如何移位？ 移多少位？

如何利用匹配失败位置前的信息 消除大量不必要的回溯？



KMP算法

- Knuth-Morris-Pratt (KMP)算法发现，P中每个字符对应一个移位值 k ，该值仅依赖于模式P本身，与目标T无关
 - 1970年，S. A. Cook在进行抽象机的理论研究时证明了在最差情况下模式匹配也可在 $N+M$ 时间内完成
 - 此后，D. E. Knuth 和V. R. Pratt以此理论为基础，构造了一种方法来在 $N+M$ 时间内进行模式匹配
 - 与此同时，J. H. Morris在开发文本编辑器时为了避免在检索文本时的回溯也得到了同样的算法

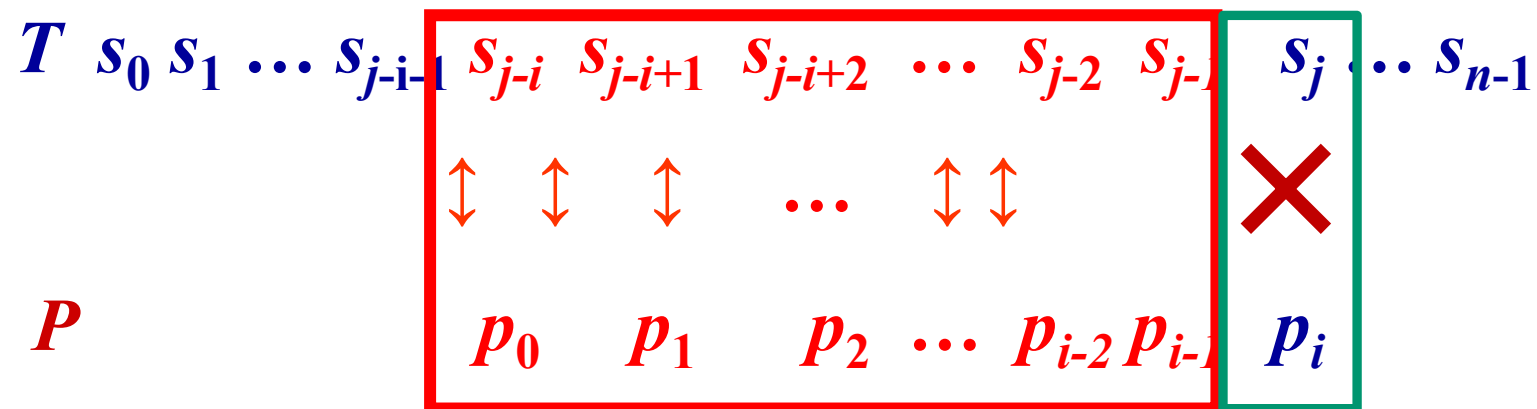
Donald E. Knuth



Professor Emeritus of **The Art of Computer Programming** at
Stanford University

1938年出生于Wisconsin。1960年，当他毕业于凯斯理工学院(Case Institute of Technology)数学系时，因为成绩过于出色，被校方打破历史惯例，同时授予学士和硕士学位。他随即进入大名鼎鼎的加州理工学院数学系，仅用三年时间便取得博士学位，此时年仅25岁。毕业后留校任助理教授，28岁时升为副教授。30岁时，加盟斯坦福大学计算机系，任正教授。从31岁那年起，他开始出版他的历史性经典巨著：**The Art of Computer Programming**。他计划共写7卷，然而仅仅出版三卷之后，已经震惊世界，使他获得计算机科学界的最高荣誉**Turing Award**！此时，他年仅38岁！后来，此书与牛顿的“自然哲学的数学原理”等一起，被评为“世界历史上最伟大的十种科学著作”之一。

一起分享牛人的思路吧



则有

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

如果

$$p_0 p_1 \dots p_{i-2} \neq p_1 p_2 \dots p_{i-1} \quad (2)$$

则立刻可以断定

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

(朴素匹配的)下一趟一定不匹配，可以跳过去

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

则再下一趟也不匹配，因为有

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

直到对于某一个 “k” 值(首尾串长度)，使得

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

则

$p_0 p_1 \cdots p_{k-1}$	s_{j-k}	s_{j-k+1}	\cdots	s_{j-1}	s_j
	\parallel	\parallel	\cdots	\parallel	\nparallel
	p_{i-k}	p_{i-k+1}	\cdots	p_{i-1}	p_i
	\parallel	\parallel	\cdots	\parallel	$?$
	p_0	p_1	\cdots	p_{k-1}	p_k

... $j-i$, 1, 2, 3,, $j-1$ j



0, 1, 2, 3, ..., $i-k$, ..., $i-1$

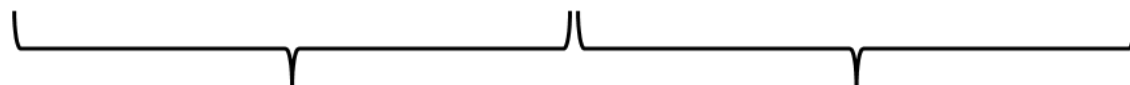
P



P



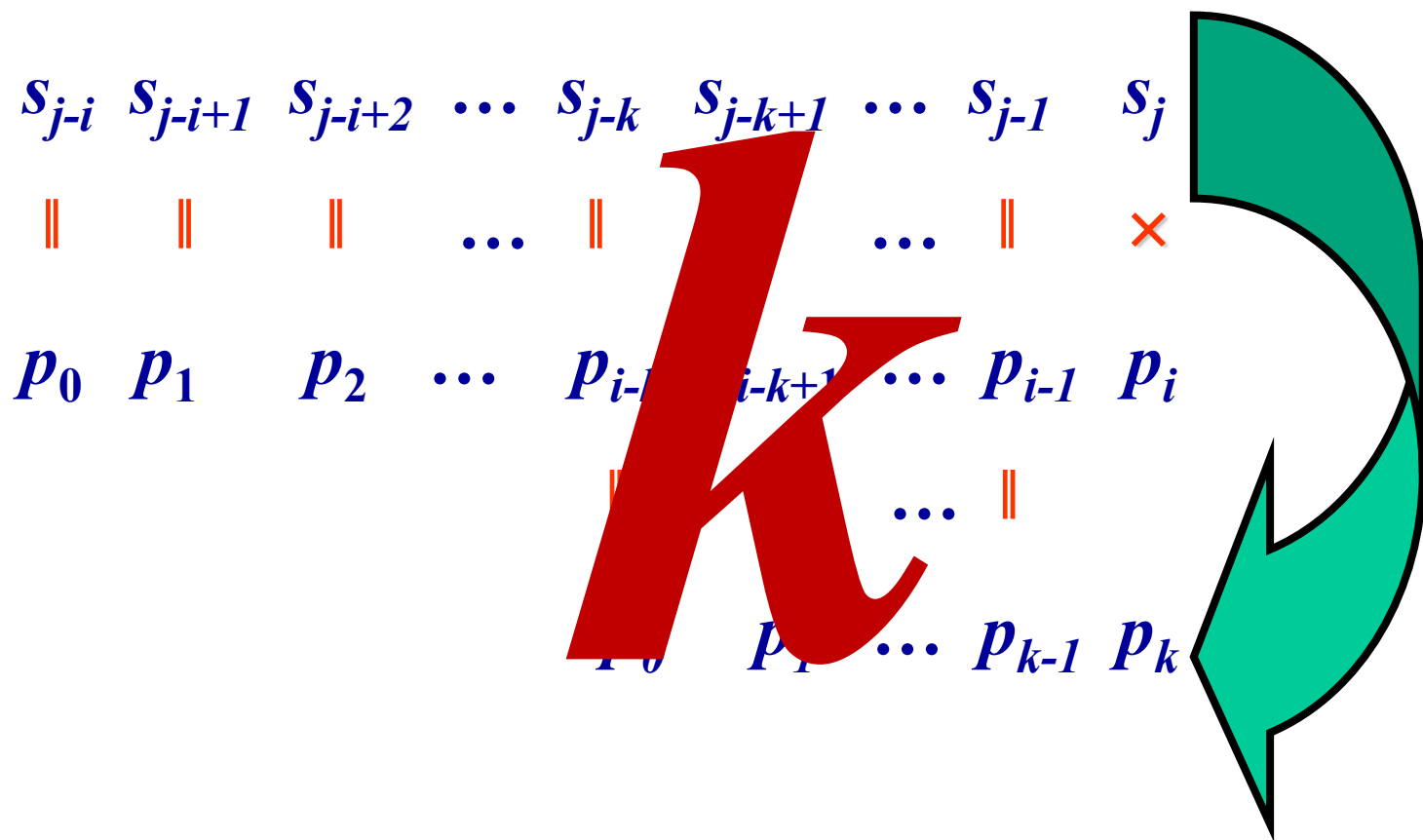
0, 1, 2, 3, ..., $k-1$... $i-1$



移位数: $i-k$

前缀数: k

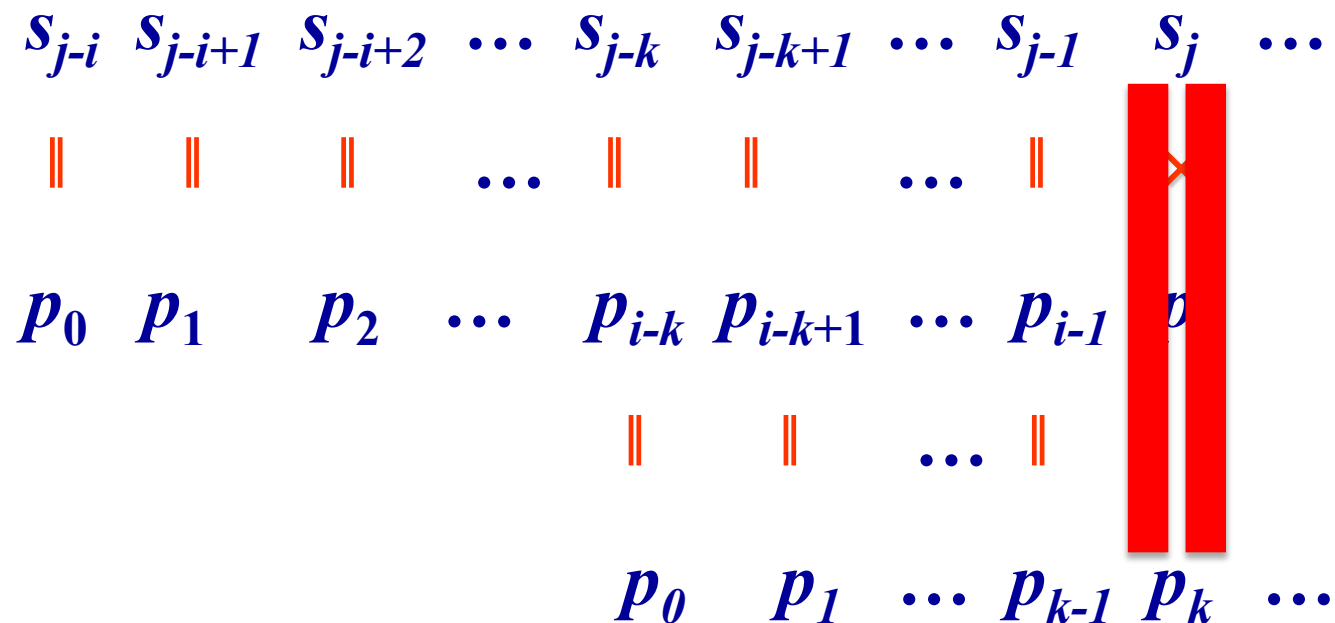
➤ 模式 p 右滑 $i-k$ 位



KMP算法发现每个字符 p_k 对应的该 k 值仅仅依赖于模式 P 本身，而与目标对象 T 无关

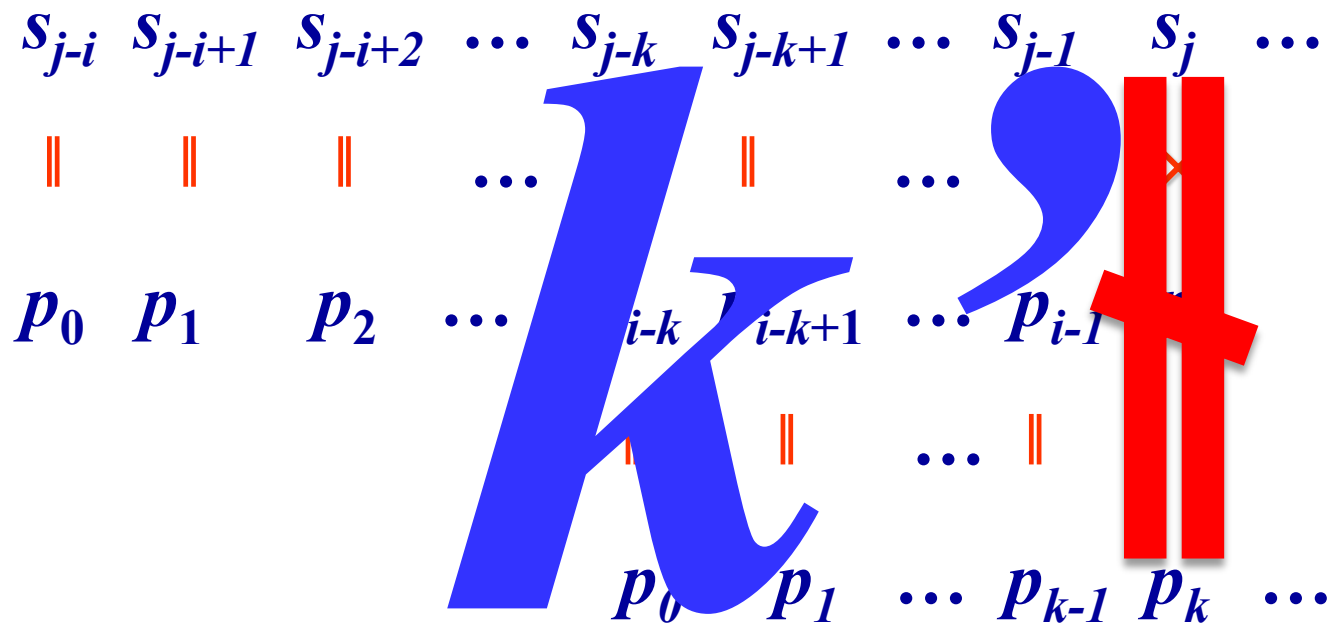
移位后，我们继续匹配

➤ 如果 $s_j = p_k$ ，则继续匹配！



注意字符 p_i, p_k, s_j 之间的关系！！

➤ 如果 $s_j \neq p_k$, 怎么处理?



➤ k' 是什么?

➤ 这个过程会继续下去吗? 何时停止?

每个字符对应的支持快速偏移的 **K** 值

但是，该如何确定和保存？



字符串的特征向量N

➤ 设模板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用来表示模板P的字符分布特征，简称N向量

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

计算特征数 n_i

➤ 前缀定义

- 列出模板P开头的任意 k 个连续字符，把它称为P的前缀子串($k < i$)，记做

$$q_0q_1q_2\cdots q_{k-1}$$

- 在P的第 i 位置的左边，也取出 k 个字符，称为 i 位置的左子串，记做 $q_{i-k}\cdots q_{i-2}q_{i-1}$

➤ 求出最长的（ k 最大的）能够与前缀子串匹配的左子串（简称第 i 位的最长前缀“真子串”）

➤ k 就是要求的特征数 n_i

特征数 n_i 的递归定义

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

➤ 特征数 n_i ($-1 \leq n_i < i$) 是递归定义的, 定义如下:

1. $n_0 \leftarrow -1$, 对于 $i > 0$ 的 n_{i+1} , 假定已知前一位置的特征数 n_i , 并且 $n_i = k$;

2. 如果 $P_i = P_k$, 则 $n_{i+1} \leftarrow k + 1$;

3. 当 $P_i \neq P_k$ 且 $k \neq 0$ 时, 则令 $k \leftarrow n_k$; 循环直到条件不满足;

4. 当 $P_i \neq P_k$ 且 $k = 0$ 时, 则 $n_{i+1} = 0$;

以眼杀人——观察法



a b a a b c a c

a b a a b c a c

-1 0 0 1 1 2 0 1

Exercise

● a b c a b a a b c b c

-1 0 0 0 1 2 1 1 2 0 0

● a b a b a b a b

-1 0 0 1 2 3 4 5

● a a b a a c a a d a

-1 0 1 0 1 2 0 1 2 0

特征向量N：非优化算法

```
int findNext(string P) {  
    int i, k;  
    int m = P.length();           //m为模板P的长度  
    int *next = new int[m];       //动态存储区开辟整数数组  
    next[0] = -1;  
    i = 0; k = -1;  
    while (i < m-1) {              // 若写成 i < m 会越界  
        //如果不等, 采用 KMP 方法自找首尾子串  
        while (k >= 0 && P[k] != P[i])  
            k = next[k];           // k 递归地向前找  
        i++; k++;  
        next[i] = k;  
    }  
    return next;  
}
```

计算示例

➤ $p = \text{"a b c d a a b c a b"}$

下标 i	0	1	2	3	4	5	6	7	8	9
p_i	a	b	c	d	a	a	b	c	a	b
next	-1	0	0	0	0	1	1	2	3	1

如果 $P_k = P_i$ 时，上述算法是否存在进一步优化的空间？

特征向量N：优化算法

```
int findNext*(string P) {  
    int i, k;  
    int m = P.length( );           //m为模板P的长度  
    int *next = new int[m];        //动态存储区开辟整数数组  
    next[0] = -1;  
    i = 0; k = -1;  
    while (i < m-1) {              //若写成i < m 会越界  
        while (k >= 0 && P[k] != P[i]) //找首尾子串  
            k = next[k];             //k递归地向前找  
        i++; k++;  
        if (P[k] == P[i])  
            next[i] = next[k];      //前面找k值, 优化步骤  
        else next[i] = k;  
    }  
    return next;  
}
```


计算示例

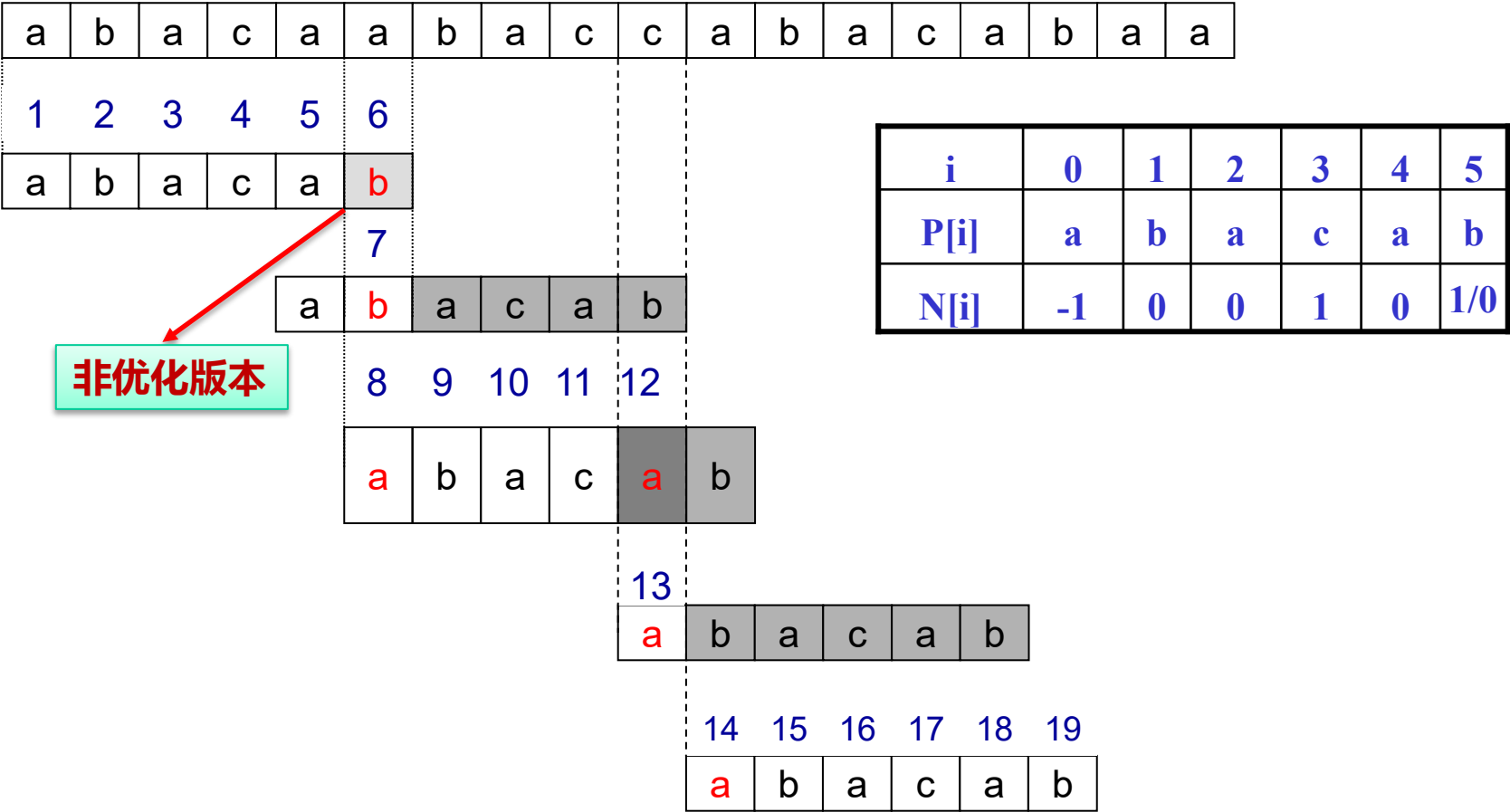
➤ p="abcdaabcab"

下标i	0	1	2	3	4	5	6	7	8	9
p _i	a	b	c	d	a	a	b	c	a	b
next	-1	0	0	0	0	1	1	2	3	1
next*	-1	0	0	0	-1	1	0	0	3	0

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) { // 反复比较对应字符来开始匹配  
        if ( j == -1 || T[i] == P[j])  
            i++, j++;  
        else j = N[j];  
    }  
    if (j >= pLen)  
        return (i-pLen);  
    else return (-1);  
}
```

KMP模式匹配： 示例



KMP算法的效率

➤ KMP算法时间复杂性分析

- while语句中， i 只增不减，所以循环体中 $i++$ 语句执行次数最多 $|N|$ 次，在同一语句中的运算 $j++$ 也不会超过 $|N|$ 次。
 - j 的初值为0，使之减少的语句只有 $j = N[j]$ ；循环体中 $j = N[j]$ 的执行次数不会超过 $i++$ ， $j++$ ；语句的执行次数加1。
 - 所以整个循环体的执行次数至多为 $2|N| + 1$ 次。亦即其时间代价与目标串的长度成线性关系。
- 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 $O(n+m)$

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



再见…

联系信息:

电子邮件: gjsong@pku.edu.cn

电 话: 62754785

办公地点: 理科2号楼2307室