第四章　字符串

张铭　主讲

采用教材：张铭，王腾蛟，赵海燕 编写
高等教育出版社，2008. 6 （“十二五”国家级规划教材）

http://jpk.pku.edu.cn/course/sjjg/
https://www.icourse163.org/course/PKU-1002534001

主要内容

- 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
  - 朴素算法
  - KMP算法

4.1 字符串基本概念

- 字符串是特殊的线性表，即元素为字符（char) 的线性表
- 简称 “串”，零个或多个字符/符号构成的有限序列
- n（≥0）个字符的有限序列，一般记作S：“$c_0c_1c_2...c_{n-1}$”
  - S是**串名字**
  - “$c_0c_1c_2...c_{n-1}$”是**串值**
  - c_i 是串中的**字符**
  - **n是串长**：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

字符/符号

- **字符**(char)：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：Σ = {0,1}
- 生物信息中DNA字符集：Σ = {A,C,G,T}
- 英语语言： Σ = {26个字符，标点符号}
- 简体中文标准字符集 GB2312 ：Σ = {6763个汉字，标点符号，… }
- ……

字符编码

- **单字节**（8 bits）
  - 采用 ASCII 码对 128 个符号进行编码
  - 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），UCS（Universal Character Set）

4.1 字符串基本概念

字符的编码顺序

- 为便于字符串间比较和运算，字符编码一般遵循 **“偏序编码规则”**
  - e.g.， 数字0-9连续编码，A-Z连续编码
  - encode（‘0’）+1 = encode（‘1’）
  - encode（‘A’）+1 = encode（‘B’）
  - ……

- **字符偏序**：根据字符的自然含义，某些字符间可两两比较次序
  - 字典序
  - 中文字符串有些特例，如“笔划”序

字符串的数据类型

- 因语言而不同
- 简单类型
  - 复合类型

- 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

4.1 字符串基本概念

子串 (Substring)

- 一个字符串中**任意个连续的字符**组成的子序列称为该串的子串
  - 两个字符串

$$s_1 = a_0a_1a_2...a_{n-1};\quad s_2 = b_0b_1b_2...b_{m-1};\quad 0 \le m \le n$$

若存在整数 i (0 ≤ i ≤n-m)，使得 $b_j = a_{i+j},\ j = 0, 1, ..., m-1$

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**

  - 空串是任意串的子串；**任意串S 都是其自身的子串**
  - **真子串**：非空且不为自身的子串
- 子串函数
  - 提取、插入、寻找、删除 …

字符串中的字符

- 重载下标运算符[ ]

```
char& string::operator [] (int n);
```

- 按字符定位下标

```
int string::find(char c,int start=0);
```

- 反向寻找，定位尾部出现的字符

```
int string::rfind(char c, int pos=0);
```

4.1 字符串基本概念

字符串抽象数据类型

```
.......
int length();                                  //  返回串的长度
int isEmpty();                                 //  判断串是否为空串
void clear();                                  //  清空串
int find(const char c, const int s);           //  从 s 开始搜索串寻找一个给定字符
string substr(const int s, const int len);     //  从 s开始提取一个长度为len的子串
string insert(const char c, const int index);  //  往串中给定位置插入一个字符
string append(const char c);                   //  在串尾添加字符
string concatenate(const char *s);             //  把串s连接在本串后面
int    strcmp(const char *s1, const char *s2); //  串比较
char *strcpy(char *s1, const char *s2);        //  串复制
```

主要内容

- 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
  - 朴素算法
  - KMP算法

4.2 字符串的存储结构和实现

- 标准字符串
  - 存储结构（顺序存储）
  - 标准字符串的运算实现
- 字符串类的**存储**结构
- 字符串类的**运算**实现

4.2 字符串的存储与实现

符串的顺序存储

- 串长变化不大的字符串采用的定长存储方式，有三种方案：

(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) 特殊标记串的结束

- '\0' 是ASCII码中8位全0码，又称为 NULL 符
- C/C++ 语言的的标准字符串（（#include <string.h>）采用

补充：较早期串的存储

- **顺序**
  - 字编址(压缩、非压缩)

    Pascal 中一般采用压缩的字编址形式， packed array
  - 字节编址
- **索引**
  - 有较多子串的命名串常量
  - 以串名为关键码组织符号表
- **链接**
  - 一般用单链(因为顺序处理)
  - 一个结点中存储多个字符
  - 插入、删除方便，但存储密度小

C/C++的标准字符串

- 字符串变量定义为字符数组
  - char s[M];
  - 字符串的结束标记： ASCII码中8位全0码 ‘\0’ ，亦称NULL
  - 字符串的实际长度为 M-1
    e.g.，char s1[6] = “value”；
- 注意： s1 = s2
- 函数库<string.h> 提供字符串处理函数

C/C++标准字符串运算

- 串长函数 int strlen(char *s);
- 串复制 char *strcpy(char *s1, char*s2);
- 串拼接 char *strcat(char *s1, char *s2);
- 串比较 int strcmp(char *s1, char *s2);
- 字符定位 char *strchr(char *s, char c);
  char *strrchr(char *s, char c);
- 子串抽取 int *strstr(char* s2, char* s1)

C++字符串类 class String

- 字符串长度限制
  - **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）
- 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
  - 实例化 STL 的 basic_string 得到：

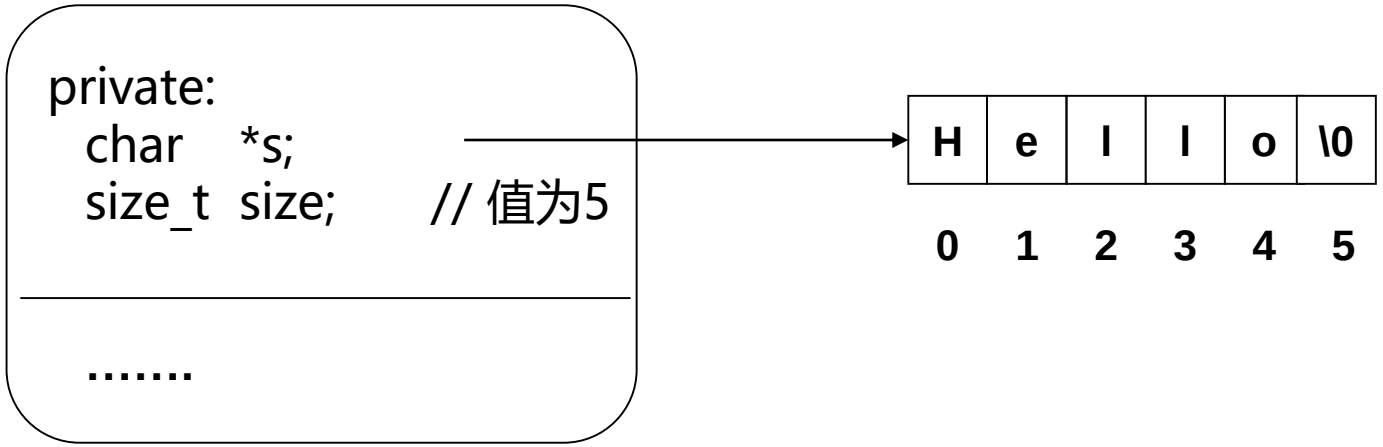
```
typedef basic_string <char> string;
```

字符串类的存储结构

```
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“) ;

s1



4.2 字符串的存储与实现

class String 存储结构

```
private:                                    // 具体实现的字符串存储结构
        char *str;                                  // 串表示
        int    size;                                // 串长度

public:                                             //  成员函数
        String(char *s);                            // 构造子
        ~String();                                  // 析构子
        String operator=(String& s);                // 赋值
        String operator+(String)  ;                 // 拼接
        String substr(int index, int cout);         // 子串
        int find(char c, int start);                // 查找
        ……
```

4.2 字符串的存储与实现

字符串类String运算

```
// 构造算子(constructor)
String::String(char *s)  {
    // 确定新字符串需要的空间，初值为char * s，其长度由标准字符串函数 strlen(s)确定
    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);
}
```

C++字符串类的常用操作

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
	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 () / length()	返回串中字符的数目
	max_size ()	返回串允许的最大长度

4.2 字符串的存储与实现

标准串运算实现

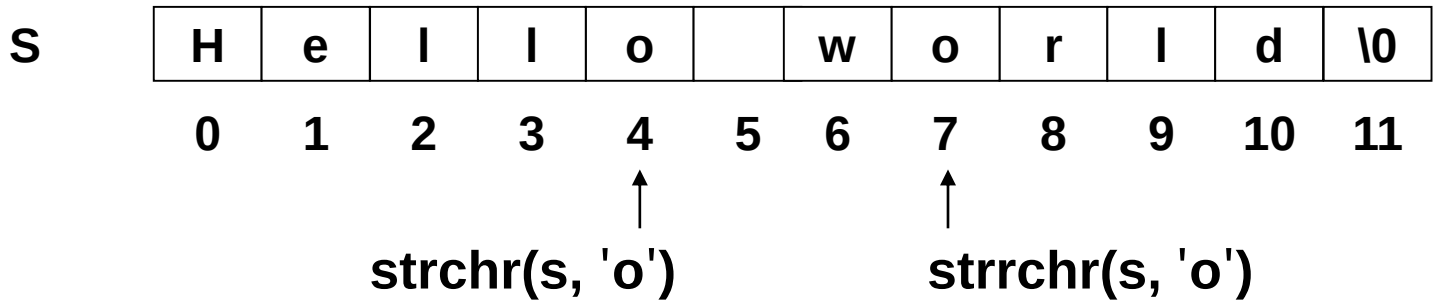
```
// 字符串比较
int strcmp(const char *s1, const char *s2)
 {
    int i = 0;
    while (s2[i] != '\0'  &&  s1[i] != '\0' ) {
          if (s1[i] > s2[i])
           return 1;
          else if (s1[i] < s2[i])
           return -1;
          i++;
    }
    if (s1[i] == '\0' && s2[i] != '\0')
           return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

```
// 字符串比较
int strcmp_1(char *s1, char *s2)
{
    int i;
    for (i=0; s1[i] == s2[i]; ++i ) {
        if (s1[i] == '\0' && s2[i] == '\0')
            return 0;          // 两个字符串相等
    }
    // 不等，比较第一个不同的字符
    return (s1[i]-s2[i])/abs(s1[i]-s2[i]);
}
```

4.2 字符串的存储与实现

标准串运算实现

e.g,



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

4.2 字符串的存储与实现

标准串运算实现

```
//在串s中寻找字符c
char * strchr(char *s , char c)
{   // s中找到c则返回所在位置，否则返回NULL
    i = 0;
    // 循环跳过 非c 字符
    while (s[i] != '\0' && s[i] != c )
                    i++;
    // 当s不含字符c则在串尾结束;
    // 成功找到c则相应位置
    if (s[i] == '\0')
          return  0;
    else
          return  &s[i] ;
}
```

```
//在串s中反向寻找字符c
char * strrchr(char *s, char c)
{   // s中找到c则返回指针位置，否则返回NULL
    i = 0;
    while (s[i] != '\0' )  i++;
    // 循环跳过非c字符
    while (s[--i] != ' \0' && s[i] != c )  ;
    // 若没找到c则在串尾结束;
    // 成功则返回相应位置
    if (s[i] == '\0')
          return  0;
    else
          return  &s[i] ;
}
```

主要内容

- 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**
  - **朴素算法**
  - **KMP算法**

4.3 字符串的模式匹配

- 模式匹配(pattern matching)
  - 在 **目标文本 T** 中寻找和定位
    一个**给定模式 P** (pattern) 的过程
- 应用
  - 文本编辑时的特定词、句的查找
    - UNIX/Linux: sed, awk, grep
  - 生物信息学和 DNA 测序
  - 确认是否具有某种结构
  - …

4.3 字符串的模式匹配

基本概念

- 精确匹配（Exact String Matching）：若目标 T 中存在至少一处与模式 P 完全相同的子串，则称为匹配**成功**，否则匹配**失败**
  - 单选 “Set” ；
  - 多选 “S?t”
  - 正则表达式（通配符 * ？ 等等）
- 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**
  - **字符串相似度**通常定义串变换所需**基本操作数目**
  - 基本操作包括**插入**、**删除** 和 **替换** 三种操作 （上网查“编辑距离 “）

单选模式精确匹配

- 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（简称为”配串”），返回其首字符位置

$$
\begin{array}{lccccccccccc}
T & t_0 & t_1 & \cdots & t_i & t_{i+1} & t_{i+2} & \cdots & t_{i+m-2} & t_{i+m-1} & \cdots & t_{n-1} \\
 & & & & \| & \| & \| & & \| & \| & & \\
P & & & & p_0 & p_1 & p_2 & \cdots & p_{m-2} & p_{m-1} & &
\end{array}
$$

若模式 P 与目标 T 的某个子串匹配，必须满足

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

4.3 字符串的模式匹配

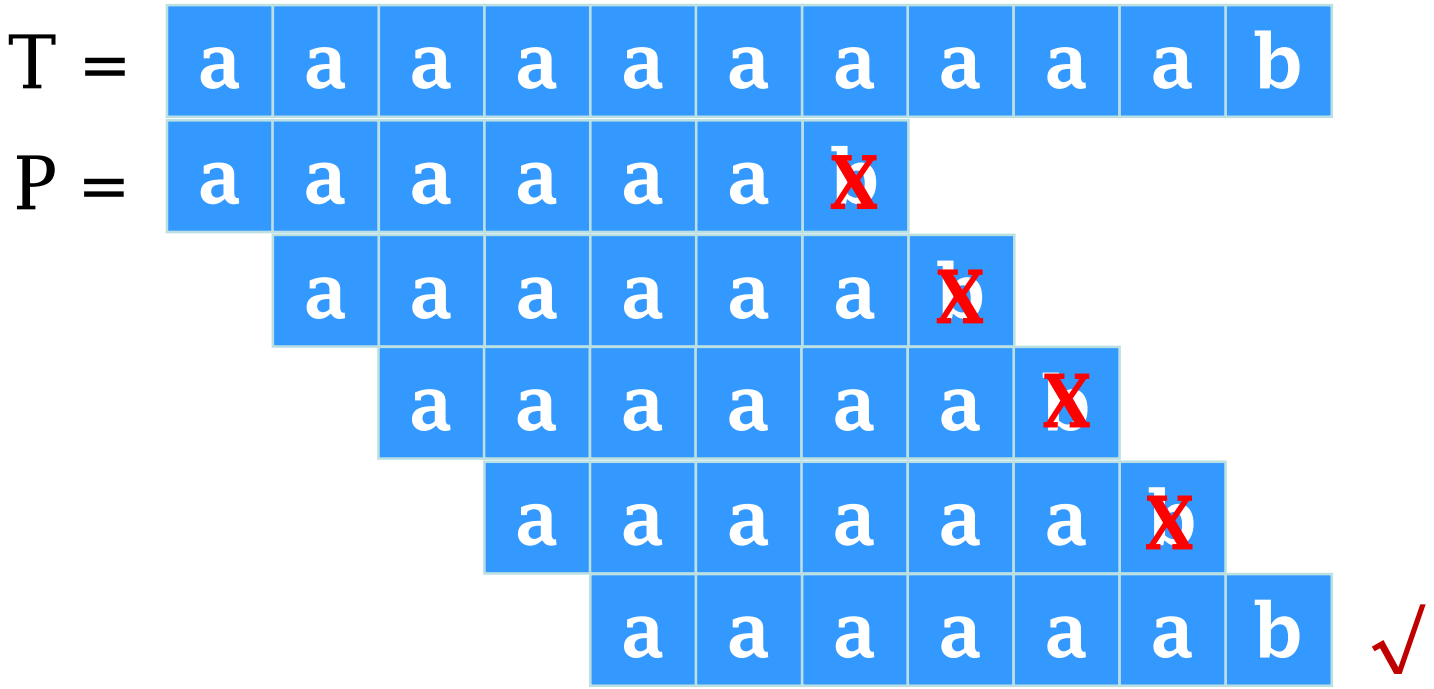
朴素模式匹配

- Naive/Brute Force：蛮力法，即穷举法
  - 尝试所有匹配可能
- **$T = t_0t_1t_2\ldots t_n$，$P = p_0p_2\ldots p_{m-1}$**；i，j分别表示T 和 P 当前字符的下标

1. 将模式从头与目标串的第i（0..n-m+1）个字符开始比较，
   若相等则继续逐个比较后续字符
2. 匹配**成功**（$p_0 = t_k$，$p_1 = t_{k+1}$，…，$p_{m-1} = t_{k+m-1}$），即
   T.substr(k, m) == P
3. 若一趟匹配过程发生**失配**（$p_j \neq t_i$），则将 P整体右移 1位
   开始下一趟的匹配

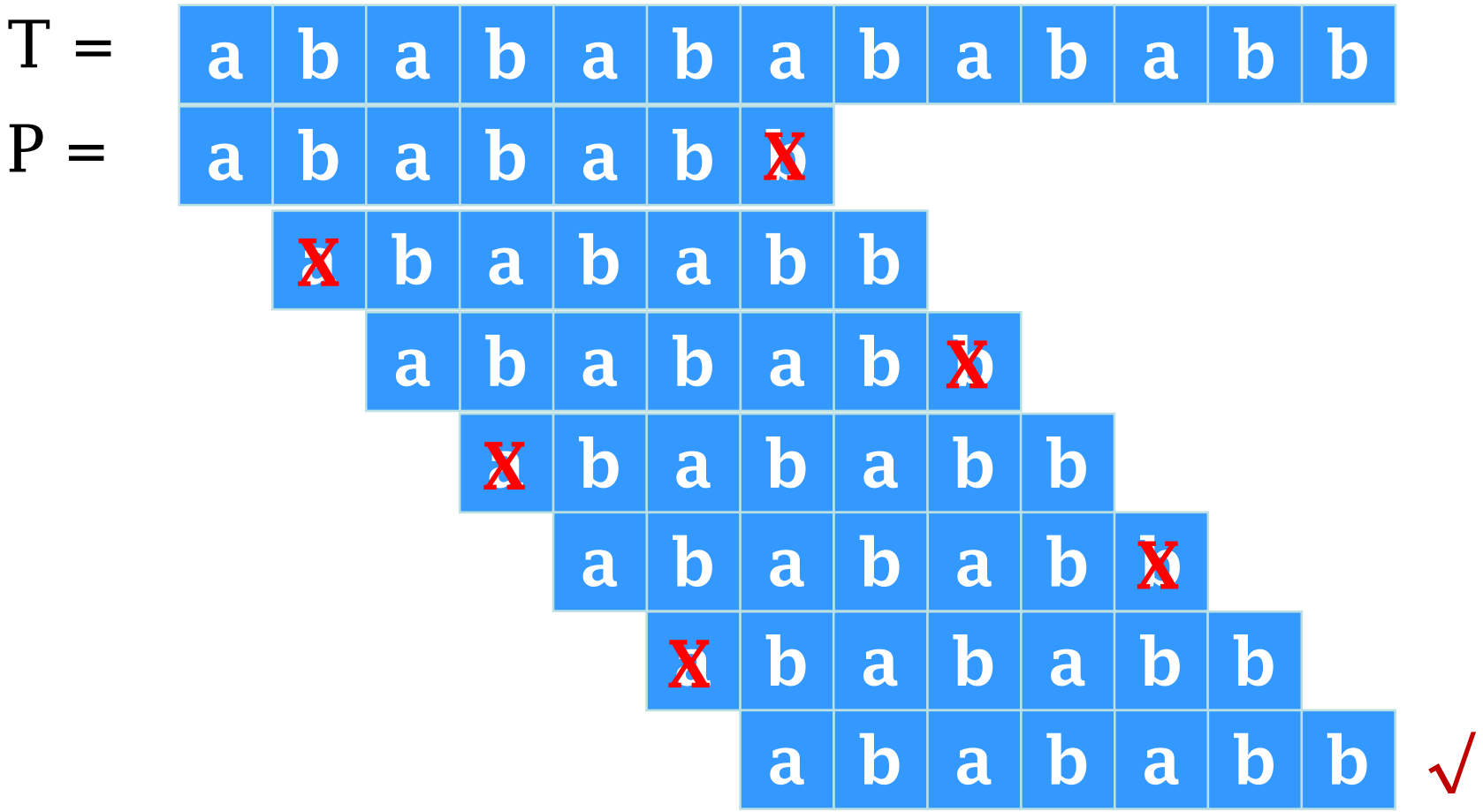
4.3 字符串的模式匹配

朴素模式匹配：示例1

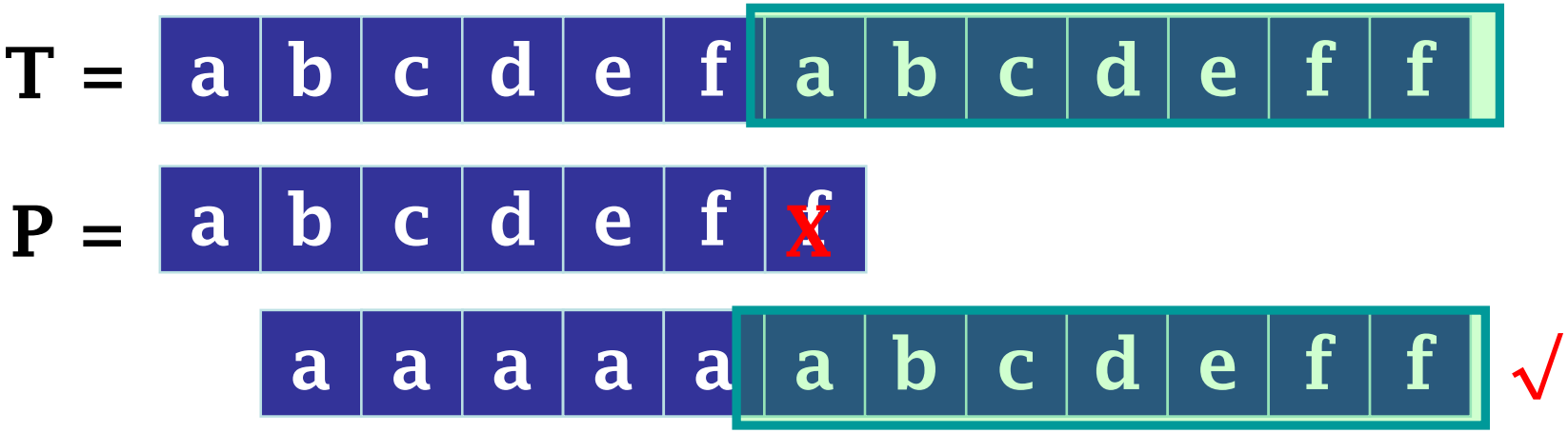


4.3 字符串的模式匹配

朴素模式匹配：示例2



朴素模式匹配：示例3



4.3 字符串的模式匹配

朴素模式匹配算法

```
int  FindPat_1(string T, string P, int startindex) {
   // 从T末尾倒数一个模板长度位置
   int LastIndex = T.length() - P.length();
   // 开始匹配位置startindex的值过大，匹配无法成功
   if  (LastIndex < startindex) return (-1);
   // i 是指向T内部字符的游标，j 是指向P内部字符的游标
   int i = startindex,   j = 0;
   while (i < T.length()  &&  j < P.length())    // “<=”呢?
        if (P[j] == T[i])  {  i++;    j++;   }
        else     {  i = i - j + 1;   j = 0; }          // 重新开始下一趟匹配
   // 若匹配成功，则返回该T子串的开始位置；否则返回失败负值
   if ( j  >=  P.length())                              // “>” 可以吗?
        return (i - j);
   else    return -1;
}
```

朴素模式匹配代码（简洁）

```
int FindPat_3(string T, string P, int startindex) {
    //g为T的游标，用模板P和T第g位置子串比较，
    //若失败则继续循环
    for (int g= startindex; g <= T.length() - P.length(); g++) {
        for (int j=0;  ((j<P.length()) && (T[g+j]==P[j])) ; j++)
            ;
        if (j == P.length())
        return  g;
    }
    return(-1);  // for结束，或startindex值过大,则匹配失败
}
```

4.3 字符串的模式匹配

朴素匹配算法：最差情况

- 目标形如a^n，模式形如$a^{m-1}b$

```
AAAAAAAAAAAAAAAAAAAAAAAA
AAAAB          5次比较
AAAAAAAAAAAAAAAAAAAAAAAA
 AAAAB          5次比较
AAAAAAAAAAAAAAAAAAAAAAAA
  AAAAB          5次比较
AAAAAAAAAAAAAAAAAAAAAAAA
   AAAAB          5次比较
.............
AAAAAAAAAAAAAAAAAAAAAAAA
                    AAAAB     5次比较
```

- 比较次数：
  - ✓ m 次比较/趟 ： 模式与目标的每一个长度为m的子串进行比较
  - ✓ n-m+1趟
- 时间复杂度：
  - ✓ $O(m \bullet n)$

4.3 字符串的模式匹配

朴素匹配算法：最佳情况 – 匹配成功

- 在目标的前m个位置上找到模式

  m = 5

  AAAAAAAAAAAAAAAAAAAAAAB

  AAAAA　　**5次比较**

- 比较次数：
  - ✓ m 次比较/趟
  - ✓ 1 趟
- 时间复杂度：O(m)

4.3 字符串的模式匹配

朴素匹配算法：最佳情况 – 匹配失败

- 总在模式第一个字符上失配

```
AAAAAAAAAAAAAAAAAAAAH
OOOOH                          1次比较
AAAAAAAAAAAAAAAAAAAAH
 OOOOH                         1次比较
AAAAAAAAAAAAAAAAAAAAH
  OOOOH                        1次比较
AAAAAAAAAAAAAAAAAAAAH
   OOOOH                       1次比较
.....
AAAAAAAAAAAAAAAAAAAAH
                    OOOOH      1次比较
```

- 比较次数
  - ✓ n-m+1 趟
  - ✓ 1 次/趟
- 时间复杂度
  - ✓ O(n)

4.3 字符串的模式匹配

朴素匹配算法时间效率

- 目标 T 的长度为n，模式 P 长度为 m，且 m ≤ n
  - 最差情况下，每一趟匹配都不成功，共进行（n-m+1）趟匹配
  - 一趟匹配最坏情况下比较 m 次： P 和 T 间相应字符逐个比较的时间
  - 整个算法的时间复杂度为

$$O(m \bullet n)$$

4.3 字符串的模式匹配

思 考

1. 字符串“software”的子串有多少个?

- 子串是字符串的**连续片段**
- 例如，software
  - ✓ 子串：空串、software、soft、oft...
  - ✓ 不是子串：fare、sfw...

2. 设 S1, S2 为串，请给出使 S1+S2 == S2+S1 成立的所有可能的条件（其中 + 为连接运算）

3. 朴素的模式匹配算法为什么效率低下？ 哪些地方可以改进?

主要内容

- 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
  - 朴素算法
  - **KMP算法**

4.3 字符串的模式匹配

朴素算法的问题

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

.....

- 由第1趟比较可知：

  $P_6 \neq T_6$； $P_0=T_0$ ， $P_1 = T_1$， ....

- 第2趟匹配将P右移1位，由 $P_0 \neq P_1$可知 $P_0 \neq T_1$

  第1次比较一定不等 —>冗余比较

- 匹配过程一旦发生失配，下一趟匹配可能造成目标串上的回溯

- 回溯是否必要?

4.3 字符串的模式匹配

朴素算法的问题

T $t_0\ t_1 \dots t_{i-j-1}\ t_{i-j}\ t_{i-j+1}\ t_{i-j+2} \dots t_{i-2}\ t_{i-1}\ t_i \dots t_{n-1}$

$\|\quad \|\quad \|\quad \dots \quad \|\quad \|$ ×

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

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)

模式右移1位 $p_0\ p_1 \dots p_{j-2}\ p_{j-1}$

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

则可断定 $p_0\ p_1 \dots p_{j-2} \neq t_{i-j+1}\ t_{i-j+2} \dots t_{i-1}$ (3)

亦即，朴素匹配的下一趟必定失败，可跳过不做

那么，P右移几位合适?

既能消除冗余比较又保证不丢失配串呢?

4.3 字符串的模式匹配

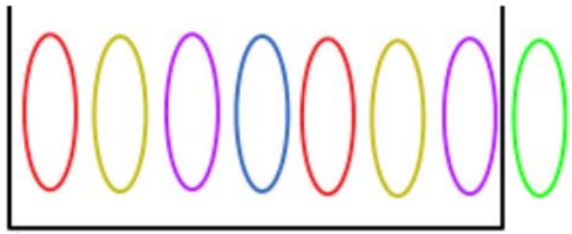
无回溯匹配算法

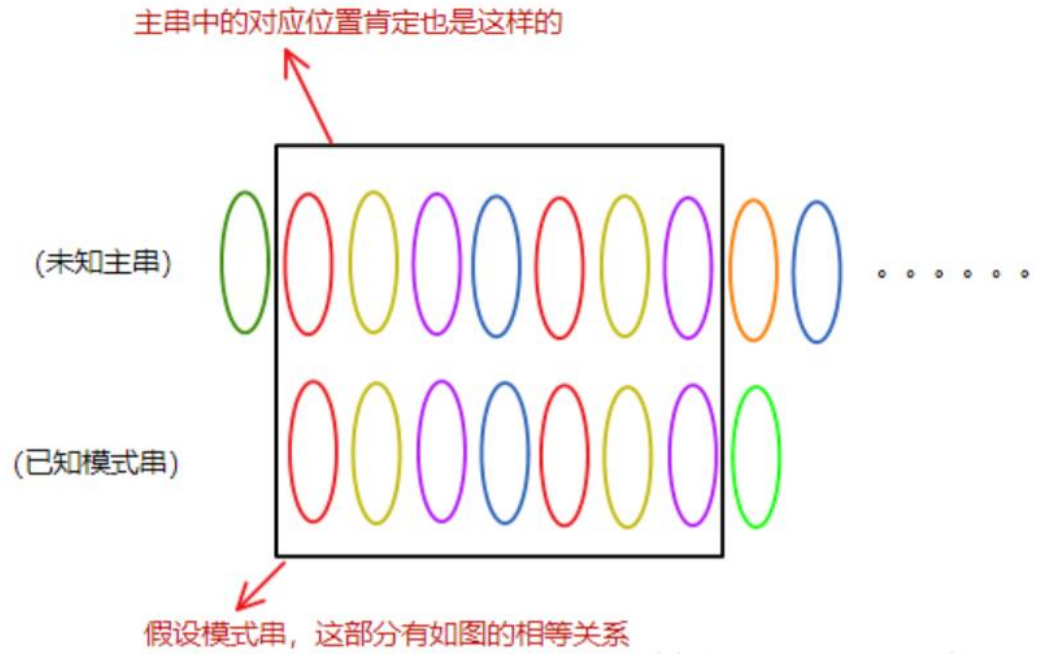
- 关键在于 匹配过程中一旦发现 p_j 和 t_i 失配时，即，

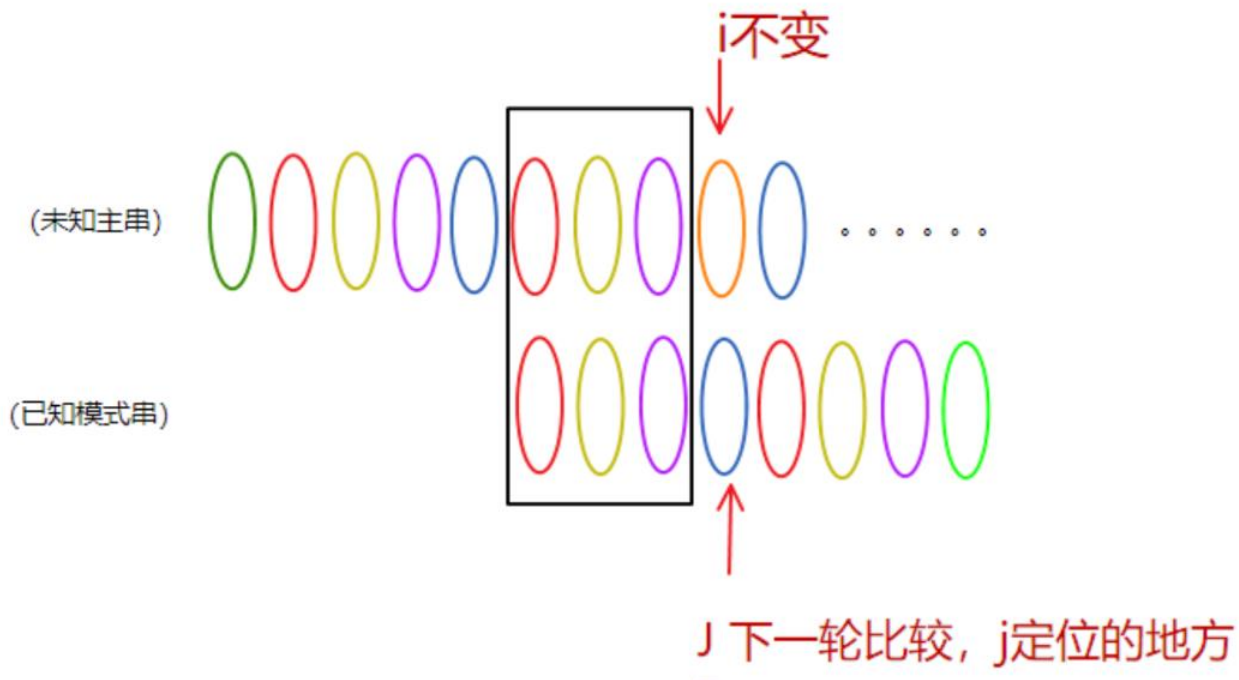
  $$substr(P, 0, j-1) = substr(T, i-j+1, j-1) \ \&\& \ p_j \neq t_i$$

  要能确定模式 **相对目标 右移的位数，** 即，该从 P 中的哪个位置继续和目标字符 t_i 进行比较，而不对目标回溯?

- **如何定位**? 保留之前比较的结果?
  - 若把这个位置记为 k ， 显然有k< j (why?)
  - 不同的 j，其 k 值可能不同




主串中的对应位置肯定也是这样的
(未知主串)
(已知模式串)
假设模式串，这部分有如图的相等关系


i不变
(未知主串)
(已知模式串)
J 下一轮比较，j定位的地方

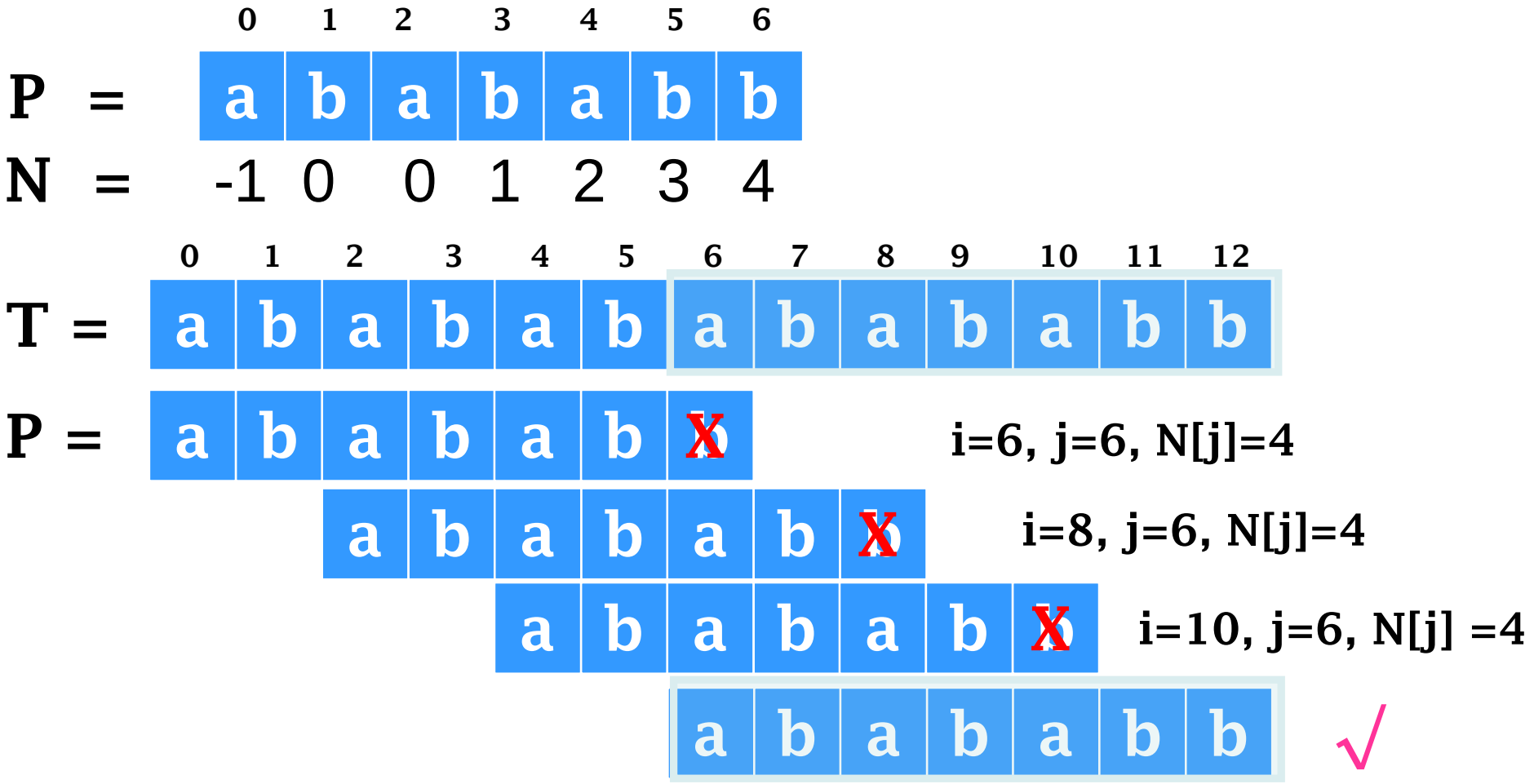
4.3 字符串的模式匹配

KMP算法

- Knuth-Morris-Pratt (KMP) 发现　每个字符对应的 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
  - 实践中的发现：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

4.3 字符串的模式匹配

KMP模式匹配示例



4.3 字符串的模式匹配

	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	1	0	1	2	3	4

X (应为3)

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
T =	a	a	b	a	a	a	a	a	a	b	a	a	a	a	c	b
P =	a	a	X	a	b	a	a	a	a	c						
				a	a	a	a	X	a	a	a	a	c			

i=2, j=1, N[j]=0

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

X

a	a	a	a	b	a	a	a	a	c

错过了！

4.3 字符串的模式匹配

字符串特征向量 N

- 长度为m 的模式 P

$$P = p_0p_1p_2p_3\cdots\cdots p_{m-1}$$

特征向量 N，表示模式 P 的字符分布特征，由 m个特征数 n_j 组成

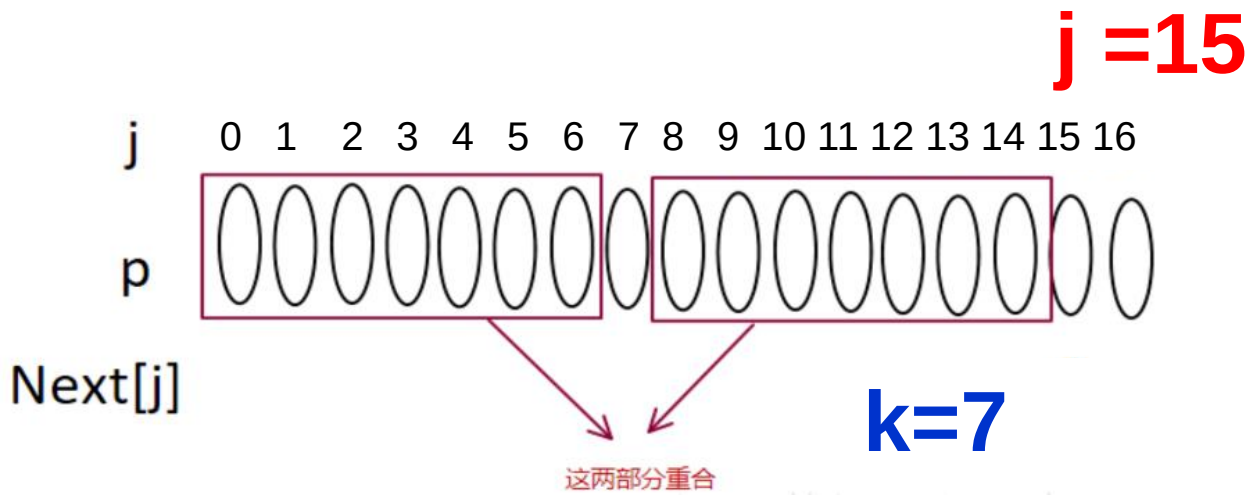
$$N = n_0n_1n_2n_3\cdots\cdots n_{m-1}$$

- 特征向量，简称 N 向量， 在很多文献中也称为 next 数组，每个特征数 n_j 对应 next 数组的一个元素

4.3 字符串的模式匹配

字符串特征向量N：构造方法

- P 第 j 个位置的特征数 n_j，首尾真子串配串中最长，k
  - 首串 $p_0\ p_1\ \cdots\ p_{k-2}\ p_{k-1}$ ：P 的前 k 个字符，也称首子串
  - 尾串 $p_{j-k}p_{j-k+1}\cdots p_{j-2}p_{j-1}$ ： P j 位置前的 k 个字符，也称尾子串



4.3 字符串的模式匹配

特征向量算法框架

- 特征数 n_{j+1} (j > 0, 0≤ n_{j+1}≤ j) 可递归定义为:

① $n_0 = -1$

对于j ≥ 0 的n_{j+1}，若已知前一位置 n_j 的特征数 $n_j = k$;

② 当 k≥0 且 $p_j \neq p_k$ 时，令 $k = n_k$；循环步骤②直到条件不满足

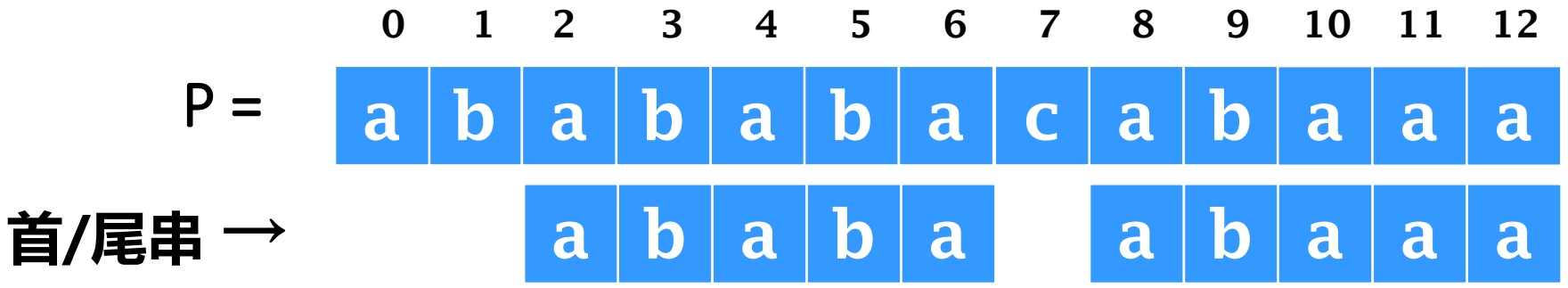
③ $n_{j+1} = k + 1$；　　// 此时，k == -1 || $p_j == p_k$

$$N[j] = \begin{cases} -1, & otherwise \\ \max\{k: 0 < k < j \wedge P[0..k-1] = p[j-k..j-1]\}, & \text{存在最长首尾配串}k \end{cases}$$

4.3 字符串的模式匹配

计算特征向量示例

j = 12　k = 0



N = -1 0 0 1 2 3 4 5 0 1 2 3 1

求特征向量N

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

j= 9 k = 0

首串→ a

首串→ a a

首串→ a a a

首串→ a

首串→ a a

首串→ a a a

首串→ a a a a

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);                          // 注意仔细算下标
    else return (-1);
}
```

字符串的特征向量N ——非优化版

```
int findNext(string P) {
    int j, k;
    int m = P.length( );                    // m为模式P的长度
    assert( m > 0);                         // 若m = 0，退出
    int *next = new  int[m];                // 动态存储区开辟整数数组
    assert( next != 0);                     // 若开辟存储区域失败，退出
    next[0] = -1;
    j = 0;  k = -1;
    while (j < m-1) {
        while (k >= 0 && P[k] != P[j])// 不等则采用 KMP 自找首尾子串
            k = next[k];                    // k 递归地向前找
         j++;   k++;  next[j] = k;
    }
    return next;
}
```

4.3 字符串的模式匹配

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
  = 模式下标 j 初值为0，使之减少的语句只有 “j = N[j];” 因 j < N[j], 每执行一次 “j = N[j];” 必然使得 j 减少 (至少减1)
  = 循环体中 目标的下标 i 只增不减的特性，语句 “i++;” 最多执行 n 次；故 “j++;” 语句也最多执行 n 次
  - 故 “j = N[j]; ” 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)

4.3 字符串的模式匹配

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	3	0	1	2	3	4

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
T =	a	a	b	a	a	a	a	a	a	b	a	a	a	a	c	b
P =	a	a	X	a	b	a	a	a	a	c						
	a	X	a	a	b	a	a	a	a	c						

冗余

KMP匹配

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

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

这行冗余 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

N[1]= 0

N[3] = 0

N[0] = -1

N[6]= 2

√

上面 P[3]==P[0], P[3] ≠ T[4]，再比冗余

模式右滑j-k位

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

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

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

字符串的特征向量N ——优化版

```
int findNext(string P) {
    int j, k;
    int m = P.length( );                    // m为模式P的长度
    int *next = new  int[m];                // 动态存储区开辟整数数组
    next[0] = -1;
    j = 0;  k = -1;
    while (j < m-1) {                       // 若写成 j < m 会越界
        while (k >= 0 && P[k] != P[j])// 若不等，采用 KMP 找首尾子串
            k = next[k];                    // k 递归地向前找
        j++;   k++;
        if (P[k] == P[j])
            next[j] = next[k];              // 前面找 k 值，没有受优化的影响
        else next[j] = k;                   // 取消if判断，则不优化
    }
    return next;
}
```

4.3 字符串的模式匹配

字符串的特征向量

- 优化前后的N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	**-1**	**0**	**0**	**0**	**1**	**1**	**2**	**1**	**2**	优化前
$p_k == p_j$?		**!=**	**!=**	**==**	**!=**	**==**	**!=**	**==**	**==**	
n_j	**-1**	**0**	**0**	**-1**	**1**	0	2	**0**	**0**	优化后

思考：不同版本特征值定义

j 位匹配错误，则 j=next[j]

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

j 位匹配错误，则 j=next[j-1]

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

4.3 字符串的模式匹配

思 考

- **最小覆盖子串：**给定一个字符串s_1，它是由某个字符串s_2不断自我连接形成的某个长串的子串。字符串s_2 的最短长度是多少?

  例如，s_1为cabcabca，s_2最短长度为3，abc、bca、cab都可以利用 abc 不断自我连接得到 abcabcabc （cabcabca是其子串）

- **最小循环节**：给定一个字符串，求它最多由几个相同的子串不重叠组成。例如“abababab”最多可以拆分成由4个“ab”串组成

- **最长重复子串**：对给定的字符串str，返回其最长重复子串及其下标位置。例如，str=”abcdacdac”，子串“ cdac”是str的最长重复子串

4.3 字符串的模式匹配

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	0 (无需预处理)	Θ(n m)
KMP算法	Θ(m)	Θ(n)
BM算法	Θ(m +)	最优 (n/m), 最差 Θ(nm)
位运算算法 (*shift-or, shift-and*)	Θ(m+|Σ|)	Θ(n)
Rabin-Karp算法	Θ(m)	平均 (n+m), 最差Θ(nm)
有限状态自动机	Θ(m |Σ|)	Θ(n)

参考资源

- **Pattern Matching Pointer**
  - http://www.cs.ucr.edu/~stelo/pattern.html
- **EXACT STRING MATCHING ALGORITHMS**
  - http://www-igm.univ-mlv.fr/~lecroq/string/
  - 字符串匹配算法的描述、复杂度分析和C源代码

数据结构与算法

谢谢聆听

国家精品课“数据结构与算法”

http://jpk.pku.edu.cn/course/sjjg/

https://www.icourse163.org/course/PKU-1002534001

张铭，王腾蛟，赵海燕

高等教育出版社，2008.6。“十二五”国家级规划教材