数据结构与算法（四）

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 （ “十一五”国家级规划教材）

https://pkumooc.coursera.org/bdsalgo-001

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 快速模式匹配

字符串示例

- s1=“123”
- s2=“ABBABBC”
- s3=“BB”
- s4=“BB ”
- s5=“Hello World!”
- s5=“”

4.1 字符串基本概念

- 字符串，特殊的 线性表，即元素为 字符 的线性表
- n（≥ 0）个字符的有限序列，n ≥ 1时，一般记作

S：“$c_0c_1c_2\ldots c_{n-1}$”

  - S 是串名字
  - “$c_0c_1c_2\ldots c_{n-1}$”是串值
  - c_i 是串中的字符
  - N 是串长（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（注意与 空格串 “　　” 的区别）

字符串是一种特殊的线性结构

- 数据对象
  - 无特殊限制
  - 串的数据对象为字符集
- 基本操作
  - 线性表的大多以“单个元素”为操作对象
  - 串通常以“串的整体”作为操作对象
- 线性表的存储方法同样适用于字符串
  - 应根据不同情况选择合适的存储表示

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（同线性表，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}
  - ……

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
  - GB
  - CJK
  - UNICODE

字符编码顺序

- 为了字符串间比较和运算的便利，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，某些字符间两两可以比较次序
  – 其实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

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

子串(Substring)

- 子串 定义

  假设 s_1，s_2是两个串：

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

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

其中 $0 \le m \le n$, 若存在整数 i ($0 \le i \le n-m$)，使得 $b_j = a_{i+j}$, $j = 0,1,\ldots,m-1$ 同时成立，则称 串 s_2是串 s_1的 子串，s1 为串 s2 的主串，或称 s_1包含串 s_2

- 特殊子串
  - **空串是任意串的子串**
  - 任意串 S 都是 S 本身的子串
  - 真子串：非空且不为自身的子串

字符串的基本运算

C 标准函数库需要 #include <string.h>

- 求串长 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 (注意)
  - int strcmp(char *s1, char *s2);
  - 看 ASCII 码，s1>s2，返回值 > 0；两串相等，返回 0
- 定位 char *strchr(char *s, char c);
- 右定位 char *strrchr(char *s, char c);
- 求子串 char *strstr(const char *str1, const char *str2);

定位函数示例

- 字符串 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 r, strrchr(s,’o’) 结果返回 7

String抽象数据类型

C++标准字符串类库

```
#include <string>
using namespace std;
```

- 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

C++ String 部分操作列表

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

得到字符串中的字符

- 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);
```

思考

- 1. 判断哪些是”software”的子串
  - 空串、software、soft、oft…
  - fare、sfw…
- 2. 若字符串 s =“software”，则其子串的数目为？

主要内容

- 字符串基本概念
- 字符串的存储结构
  - 字符串的顺序存储
  - 字符串类 class String 的存储结构
- 字符串运算的算法实现
  - 字符串运算的实现
  - String 类的实现
- 字符串的模式匹配
  - 朴素算法
  - KMP 快速模式匹配

字符串的顺序存储

- 对串长变化不大的字符串，有三种处理方案

1. 用 S[0] 作为记录串长的存储单元 (Pascal)
   - 缺点：限制了串的最大长度不能超过256
2. 为存储串的长度，另辟一个存储的地方
   - 缺点：串的最大长度一般是静态给定的，不是动态申请数组空间
3. 用一个特殊的末尾标记 ‘\0’ (C/C++)
   - 例如：C/C++ 语言的 string 函数库 （ #include <string.h> ）采用这一存储结构
   - ‘\0’ 的 ASCII 字符表中编号为 0，等价于常量 NULL、数字 0、常量 false

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";
```

```
private:
    char *str;
    int   size;   // 值为5
------------------------------
          ……
```

H	e	l	l	o	\0
0	1	2	3	4	5

字符串运算的算法实现

- 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);

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;
}
```

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;
}
```

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;
}
```

更简便的算法

```
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]);
}
```

串运算的实现

```
// 构造函数(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。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，在动态存储区域开辟一块空间，用于存储初值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);
}
```

String串运算的实现

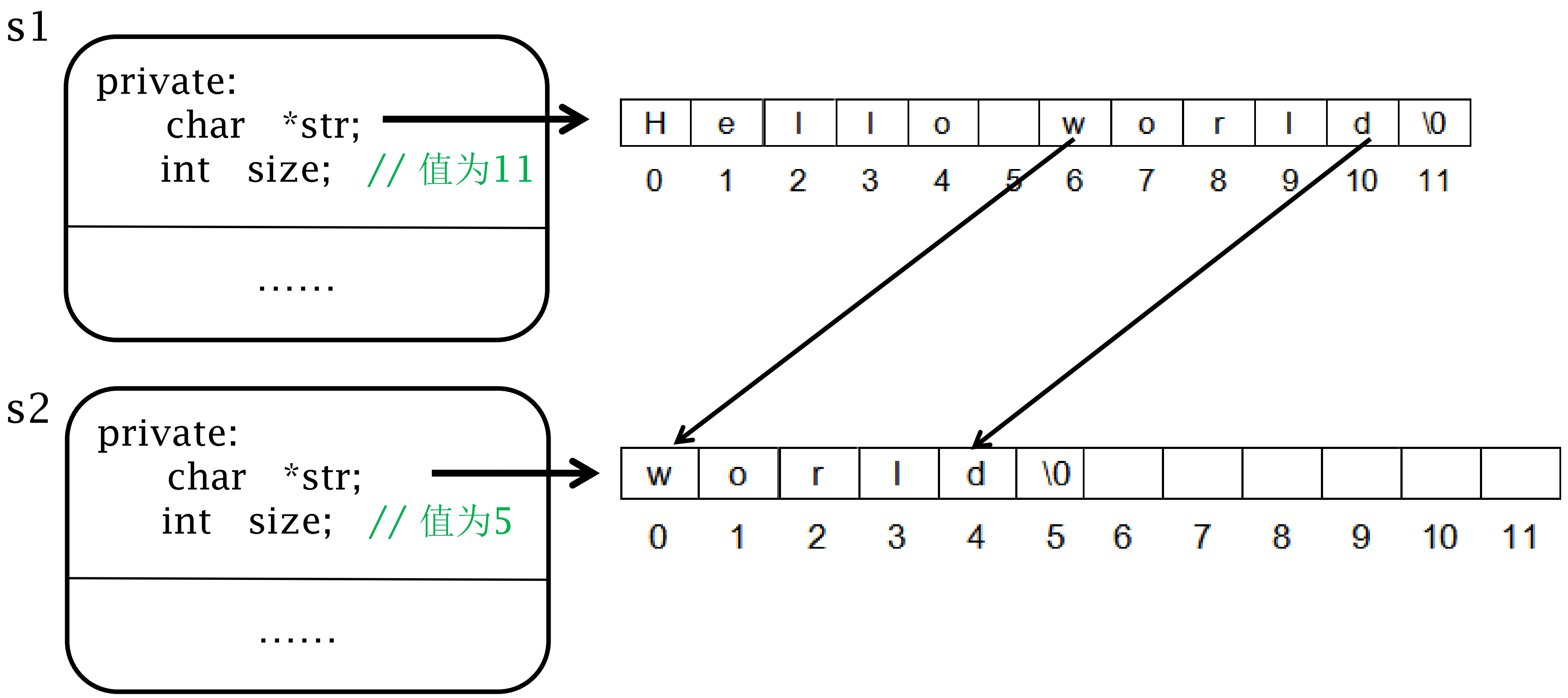
```
// 析构函数
String::~String() {
  // 必须释放动态存储空间
  delete [] str;
}
```

String串运算的实现

```
// 赋值函数
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 != NULL) ;
      size = s.size;
   }
   strcpy(str , s.str );
   // 返回本实例，作为String类的一个实例
   return  *this;
}
```

思考：String抽取子串

- s2 = s1.Substr(6, 5) ;



思考

- 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）
- 设计一个算法来实现字符串逆序存储，要求不另设串存储空间

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 快速模式匹配

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）
  - （pattern）P（字符串）

  在目标 T 中寻找一个给定的模式P的过程
- 应用
  - 文本编辑时的特定词、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
  - …
- 模式集合

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

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

为使模式 *P* 与目标 *T* 匹配，必须满足

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

模式匹配的目标和算法

- 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
- 解决模式匹配问题的算法
  - 朴素（称为“Brute Force”，也称 “Naive”）
  - Knuth-Morrit-Pratt（KMP 算法）
  - ……

朴素模式匹配（穷举法）

- 设 $T = t_0 t_1, t_2, \ldots, t_{n-1}$ ，$P = p_0, p_1, \ldots, p_{m-1}$
  - i 为 T 中字符的下标，j 为 P 中字符的下标
  - 匹配成功 $(p_0 = t_i, p_1 = t_{i+1}, \ldots, p_{m-1} = t_{i+m-1})$
    - 即，T.substr(i, m) == P.substr(0, m)
  - 匹配失败 $(p_j \neq t_i)$ 时，
    - 将 P 右移再行比较
  - 尝试所有的可能情况

朴素匹配例1

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

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

a a a a a a ~~b~~

a a a a a a ~~b~~

a a a a a a ~~b~~

a a a a a a b

朴素模式匹配例2

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	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

朴素匹配例3

T =	a	b	c	d	e	f	a	b	c	d	e	f	f	
P =	a	b	c	d	e	f	~~f~~ x							
		a	a	a	a	a	a	b	c	d	e	f	f	√

朴素模式匹配算法：其一

```
int FindPat_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);
}
```

朴素模式匹配算法：其二

```
int  FindPat_2(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; }
   if ( j  >=  P.length())   // “>”  可以吗？
       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
}
```

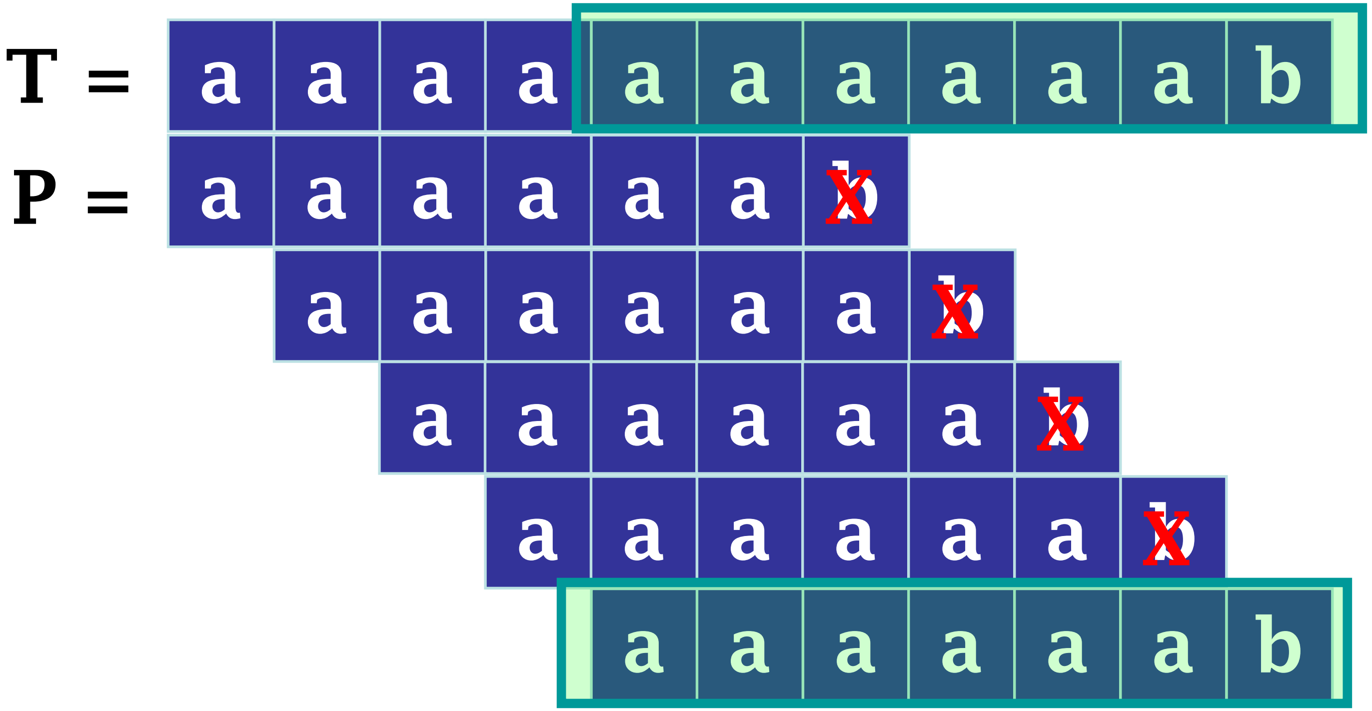
模式匹配原始算法：效率分析

- 假定目标 T 的长度为 n，模式 P 长度为 m，m≤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”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

$$O(m \bullet n)$$

朴素模式匹配算法：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 $a^{n-1}X$
  - 模式形如 $a^{m-1}b$



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

朴素模式匹配算法：
最佳情况-找到模式

- 在目标的前 m 个位置上找到模式，设 m = 5

AAAAAAAAAAAAAAAAAAAAH

AAAAA 5次比较

- 总比较次数：m
- 时间复杂度：O(m)

朴素模式匹配算法：

最佳情况-没找到模式

- 总是在第一个字符上不匹配
- 总比较次数：
  - n – m + 1
- 时间复杂度：
  - O(n)

AAAAAAAAAAAAAAAAAAAAAAH
OOOOH 1次比较
AAAAAAAAAAAAAAAAAAAAAAH
 OOOOH 1次比较
AAAAAAAAAAAAAAAAAAAAAAH
  OOOOH 1次比较
AAAAAAAAAAAAAAAAAAAAAAH
   OOOOH 1次比较

....

AAAAAAAAAAAAAAAAAAAAAAH
OOOOH 1次比较

思考：朴素算法的冗余运算

- 朴素算法之所以较慢的原因是有冗余运算
- e.g.,
  - 由1）可知： $p_5 \neq t_5$， $p_0=t_0$, $p_1 = t_1$,同时由 $p_0 \neq p_1$可得知$p_0 \neq t_2$ 故将 P 右移一位后第2）趟比较一定不等；比较冗余
  - 那么把 P 右移几位可以消除冗余的比较而不丢失配串呢？

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
....

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**

无回溯匹配

- 匹配过程中，一旦 p_j 和 t_i 比较不等时，即

```
P.substr(1,j-1) == T.substr(i-j+1,j-1)
```

  但 $p_j \neq t_i$

  - 该用 P 中的哪个字符 p_k 和 t_i 进行比较？
  - 确定右移的位数
  - 显然有 $k < j$, 且不同的 j, 其 k 值不同
- Knuth-Morrit-Pratt (KMP)算法
  - k 值仅仅依赖于模式 P 本身，而与目标对象T无关

KMP算法思想

T = a b c d e f a b c d e f f

P = a b c d e f f X

X a b c d e f f

$T \quad t_0 \; t_1 \; \ldots \; t_{i\text{-}j\text{-}1} \; t_{i\text{-}j} \; t_{i\text{-}j+1} \; t_{i\text{-}j+2} \; \ldots \; t_{i\text{-}2} \; t_{i\text{-}1} \; t_i \; \ldots \; t_{n\text{-}1}$

$\| \quad \| \quad \| \quad \| \quad \| \quad \times$

$P \quad p_0 \; p_1 \; p_2 \; \ldots \; p_{j\text{-}2} \; p_{j\text{-}1} \; p_j$

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

朴素下一趟 $p_0 \quad p_1 \quad \ldots \quad p_{j\text{-}2} \quad p_{j\text{-}1}$

如果 $p_0 \, p_1 \ldots p_{j\text{-}2} \neq p_1 \, p_2 \ldots p_{j\text{-}1}$ (2)

则立刻可以断定

$p_0 \, p_1 \ldots p_{j\text{-}2} \neq t_{i\text{-}j+1} \, t_{i\text{-}j+2} \ldots t_{i\text{-}1}$

(朴素匹配的)下一趟一定不匹配，可以跳过去

$p_0 \quad p_1 \quad \ldots \quad p_{j\text{-}2} \quad p_{j\text{-}1}$

4.3字符串的模式匹配

T = a b c d e f a b c d e f f

P = a b c d e f f ~~X~~

~~X~~ a b c d e f f

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

则再下一趟也不匹配，因为有

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

直到对于某一个“k”值（首尾串长度），使得

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

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

$$\begin{array}{ccccc} t_{i-k} & t_{i-k+1} & \ldots & t_{i-1} & t_i \\ \| & \| & & \| & \times \\ p_{j-k} & p_{j-k+1} & \ldots & p_{j-1} & p_j \\ \| & \| & & \| & ? \\ p_0 & p_1 & \ldots & p_{k-1} & p_k \end{array}$$

模式右滑 **j-k** 位

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

字符串的特征向量N

设模式 P 由 m 个字符组成，记为

$$P = p_0\ p_1p_2p_3\ldots\ldots p_{m-1}$$

令 特征向量 N 用来表示模式 P 的字符分布特征，简称 N 向量由m个特征数 $n_0 \ldots n_{m-1}$ 整数组成，记为

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

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

字符串的特征向量N：构造方法

- 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 … p_{k-2} p_{k-1}
  - 尾串： p_{j-k} p_{j-k+1} … p_{j-2} p_{j-1}

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

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~~ X (应为3)	0	1	2	3	4

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

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

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

错过了！

KMP模式匹配示例

	0	1	2	3	4	5	6
P =	a	b	a	b	a	b	b
N =	-1	0	0	1	2	3	4

	0	1	2	3	4	5	6	7	8	9	10	11	12	
T =	a	b	a	b	a	b	a	b	a	b	a	b	b	
P =	a	b	a	b	a	b	~~b~~ X							i=6,j=6, N[j]=4
			a	b	a	b	a	b	~~b~~ X					i=8,j=6, N[j]=4
					a	b	a	b	a	b	~~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);                     // 注意仔细算下标
   else return (-1);
}
```

对应的求特征向量算法框架

- 特征数 n_j ($j>0$, $0\le n_{j+1}\le j$)是递归定义的，定义如下：

1. $n_0 = -1$，对于 $j > 0$ 的 n_{j+1}，假定已知前一位置的特征数 n_j，令 $k = n_j$；
2. 当 $k\ge 0$ 且 $p_j \ne p_k$ 时，则令 $k = n_k$；让步骤2循环直到条件不满足
3. $n_{j+1} = k + 1$；// 此时，$k == -1$ 或 $p_j == p_k$

字符串的特征向量N ——非优化版

```
int findNext(string P) {
    int j, k;
    int m = P.length( );                 // 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;
}
```

求特征向量N

	0	1	2	3	4	5	6	7	8	9
N =	-1	0	1	2	3	0	1	2	3	4
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

模式右滑j-k位

$$
\begin{array}{ccccccccc}
t_{i-j} & t_{i-j+1} & t_{i-j+2} & \cdots & t_{i-k} & t_{i-k+1} & \cdots & t_{i-1} & t_i \\
\| & \| & \| & & \| & \| & & \| & \times \\
p_0 & p_1 & p_2 & \cdots & p_{j-k} & p_{j-k+1} & \cdots & p_{j-1} & p_j \\
 & & & & \| & \| & & \| & ? \\
 & & & & p_0 & p_1 & \cdots & p_{k-1} & p_k
\end{array}
$$

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

$$t_i \neq p_j\text{，}\ p_j == p_k?$$

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

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;
}
```

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
$p_k==p_j$?		≠	≠	==	≠	==	≠	==	==	
next[j]	-1	0	0	-1	1	0	2	0	0	优化版

KMP算法的效率分析

- 循环体中” j = N[j];” 语句的执行次数不能超过 n 次。否则，
  - 由于 “j = N[]; ”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]; ”的执行次数超过n次，最终的结果必然使得 j 为**比-1小很多的**负数。这是不可能的（j有时为-1,但是很快+1回到0）。
- 同理可以分析出求N数组的时间为O(m)

 故，KMP算法的时间为O(n+m)

总结：单模式的匹配算法

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)

思考：不同版本特征值定义

j 位匹配错误，则 j=next[j]

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

j 位匹配错误，则 j=next[j-1]

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

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://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