第三章　字符串

3.1　字符串的概念及其运算

3.2　字符串的实现

3.3　模式匹配（子串查找）

3.1 字符串及其运算

字符串：简称串，是特殊的线性表，元素是字符。主要特征是**支持一组以串为对象的操作**

“字符”是个抽象概念，**字符集**是全体字符的集合

人们常考虑**计算机的标准字符集**上的字符串，实际上可以以**任意数据元素**的集合作为字符集

长度：串中字符个数。长度为零的串称为**空串**

在任意字符集里，只有唯一的一个空串

子串：串s_1中若干连续字符组成的串s_2称为s_1的子串，也称s_1是s_2的主串，一个子串可能主串里多次出现

子串在主串中的位置：即子串的**第一个**字符在主串中的序号（下标 +1）

严格说，是子串的某个出现在主串里的位置

相等：两个串长度相等，且对应位置的字符都相等

顺序：两个串可以按字典序比较

字典序的基础是**字符序**（字符集上的一个**全序**）。对串 $s_1 = a_0 a_1 \cdots a_{n-1}, \quad s_2 = b_0 b_1 \cdots b_{m-1}$

$S_1 < S_2$：

若存在 k 使 $a_i = b_i$ $(i = 0,1, \ldots k\text{-}1)$，但是 $a_k < b_k$

或者$n < m$，且$a_i = b_i$ $(i = 0,1, \ldots n\text{-}1)$

抽象模型

字符集：　$\mathbb{C} \qquad c \in \mathbb{C}$

字符串集：

$$\mathbb{C}^0 \stackrel{\text{def}}{=} \{\varepsilon\}$$ 长度为0的串只有一个

$$\mathbb{C}^1 \stackrel{\text{def}}{=} \mathbb{C}$$ 长度为1的串的集合

$$\mathbb{C}^{n+1} \stackrel{\text{def}}{=} \mathbb{C}^n \times \mathbb{C}$$ 长度为n+1的串集合

$$\mathbb{S} \stackrel{\text{def}}{=} \bigcup_{n \in \mathbb{N}} \mathbb{C}^n$$ 所有有限长字符串的集合

运算

串拼接：　$\hat{} : \mathbb{S} \times \mathbb{S} \to \mathbb{S}$　　$s \hat{} t$ 的前一段是 s，后一段是 t

串长度：　$\# : \mathbb{S} \to \mathbb{N}$　　$\# s = n \text{ iff } s \in \mathbb{C}^n$

$s, s_i \in \mathbb{S}$

法则：

$$\varepsilon \hat{} s = s = s \hat{} \varepsilon$$
$$(s_1 \hat{} s_2) \hat{} s_3 = s_1 \hat{} (s_2 \hat{} s_3)$$
$$\# \varepsilon = 0$$
$$\#(s_1 \hat{} s_2) = \# s_1 + \# s_2$$

- 空串是拼接操作的“单位元”（么元），拼接操作有结合律，无交换律
- 串集合加上拼接操作，构成一个半群
- 它有单位元，因此是一个么半群

关于串的理论有许多研究工作。基于串和串中的替换，研究者开发了post系统（一种计算模型），串重写系统（rewriting system）等

基本运算

- 创建空串
  String createNullStr(void)
- 判断一个串是否空
  int isNullstr(String s)
- 求串的长度
  int length(String s)
- 将两个串拼接成一个新串
  String concat(String s1, String s2)
- 求串s中第i个字符开始连续j个字符构成的子串
  String subStr(String s, int i, int j)
- 求串s2在串s1中第一次出现的位置
  int index(String s1, String s2)

其他操作

按照字典序比较字符串 s1 和 s2，当 s1 小于、等于或者大于 s2 时返回 –1、0 或者 1:
int compStr(String s1, String s2);

在字符串 s 里把 s1 的所有出现代换为 s2:
void replaceSubStr(String s, String s1, String s2);

删除串中从第 m 个字符开始的 n 个字符:
void eraseSubStr(String s, int m, int n);

删除字符串里的所有字符:
void clearStr(String s);

还可以考虑其他操作，如参考 C 语言的字符串库函数

3.2 字符串的实现

3.2.1 顺序表示

3.2.2 链接表示

串是字符的线性序列，可采用表的实现方式：顺序表示和链接表示，也可以用动态顺序表作为实现方式（实际的字符串库经常采用这种表示方式）

还可以根据串本身的特点和串操作的特点，考虑其他表示方式（当然仍是基于顺序和链接）

关键问题：需要很好支持各种串操作的实现

字符串表示的两个问题

字符串内容的存储。两个极端:

1，连续保存在一块存储区（顺序表实现）

2，一个字符存入一个独立存储块，链接起来（一种链接表示）

也可采用各种中间方式，把一个串中的字符分段保存在一组存储块里，并把这些存储块链接起来。

字符串结束的表示。由于不同的字符串具有不同长度

两种基本方式:

1，用专门的数据域记录字符串长度

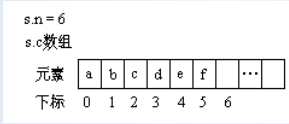
2，用特殊的符号表示字符串结束（例如 C 语言的“字符串”，用空字符表示结束）

3.2.1 顺序表示

➢ 顺序串的定义

```
#define  MAXNUM   80     /* 最大允许字符数 */
typedef struct SeqString {   /* 顺序串的类型 */
    int   n;                 /* 长度 n   MAXNUM */
    char  c[MAXNUM];
} SeqString, *PSeqString;
```

➢ 例



教科书新版采用另一种表示方式:

用字符指针代替字符数组，实际存储区由动态分配获得

- 算法3.1 创建空顺序串
  PSeqString createNullStr_seq(void)
- 算法3.2 创建一个新串，并用C字符串s初始化
  PSeqString createStr_seq(char *s)
- 算法3.3 求顺序表示的串的子串
  PSeqString subStr_seq(PSeqString s, int i, int j)
- 算法3.4 串的比较
  int compStr_seq(PSeqString s, PSeqString s2)

其他操作可以类似定义。

子串匹配问题下面专门讨论。

向下


```
/*创建空顺序串*/
PSeqString createNullStr_seq( void ) {
   PSeqString pst =
      (PSeqString)malloc(sizeof(SeqString));
   if (pst == NULL)
      printf("Out of space!!\n");
   else
      pst->n = 0;
   return (pst);
}
```

这个操作与创建空表完全一样

返回

```
/* 创建一个字符串，用 C 串 s 初始化它 */
PSeqString createStr_seq( char *s ) {
   char *p, *q;
   PSeqString pst = createNullStr_seq();
   if (pst == NULL)  return NULL;

   for (p = q = pst->c; *s != '\0' && p – q < MAXNUM;  )
         *p++ = *s++;            /* 复制s的字符 */
   pst->n = p – q;

   return pst;
}
```

返回


```
/* 求s中第i(i>0)个字符开始连续j个字符构成的子串 */
PSeqString subStr_seq(PSeqString s, int i, int j) {
   int k;
   PSeqString pst = createNullStr_seq( );   /* 创建一空串 */
   if (pst == NULL) return  NULL;
   if ( i > 0 && i <= s->n && j > 0 ) {
      if ( s->n < i+j-1 )  j = s->n-i+1; /*若从i起的字符不够j个*/
      for (k = 0; k < j; ++k)              /* 复制字符 */
         pst->c[k] = s->c[i+k-1];
      pst->n = j;
   }
   return pst;
}
```

返回


```
/* 按照字典序比较串 s1 和 s2 的大小 */
int compStr_seq (PSeqString s1, PSeqString s2 ) {
    int k;
    for (k = 0; k < s1->n && k < s2->n; ++k) {
        if (s->c[k] < s1->c[k])
            return -1;
        else if (s->c[k] > s1->c[k])
            return 1;
    }

    if (s1->n < s2->n) return -1;
    if (s1->n > s2->n) return 1;
    return 0;
}
```

返回

3.2.2 链接表示

➢ 链接串的定义

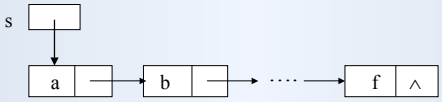
```
typedef  struct  StrNode StrNode, *PStrNode, *LinkString;
struct  StrNode  {
    char  c;
    PStrNode  link;
};
```

数据表示就是以字符为结点元素的链接表。

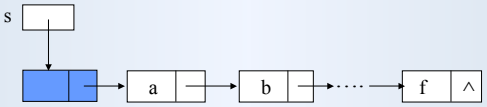
一个结点里存一个字符，操作比较方便，但存储效率低（一个链接指针所占空间通常比一个字符更大）。

用链接表示时，也可以采用在一个结点里存放多个字符的方式（采用这种表示，操作的实现复杂一些）

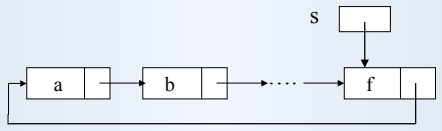
字符串的简单链接表示也可以考虑多种形式：



(a) 不带头结点



(b) 带头结点



(c) 循环表表示

字符串的基本操作很容易写出：

- 创建空串就是创建空链表；
- 求子串就是部分链表结点的复制；
- 下面只介绍几个操作，用带有头结点的链接表实现字符串

```
/*创建带头结点的串，用C 语言字符串 s 初始化*/
LinkString  createStr_link( char *s ) {
   PStrNode p, q;
   LinkString pst = (LinkString)malloc(sizeof(StrNode));
   if (pst == NULL) return NULL;
   for (p = pst; *s != '\0'; ++s) {
      q = (PStrNode)malloc(sizeof(StrNode));
      if (q == NULL) break;
      q->c = *s; p->link = q; p = q;
   }
   p->link = NULL;
   return pst;
}
```



```
/*判断串s是否空串。若为空串则返回1，否则返回0*/
int IsNullStr (LinkString s) {
    return (s->link==NULL);
}
/*返回串s的长度*/
int length (LinkString s) {
    int len = -1;
    LinkString p = s;
    for ( p=s; p!=NULL; p=p->link )
        ++len;
    return len;
}
```



```
/*返回一个新串，其内容是串s1和s2内容的拼接*/
LinkString concat (LinkString s1, LinkString s2 ) {
   LinkString p, q, temp, s3 = createNullStr_link();
   if (s3 == NULL) return NULL;
   for (p = s3, q = s1->link; q != NULL; q = q->link) {
      temp = (LinkString)malloc(sizeof(StrNode));
      if (temp == NULL) { p->link = NULL; return s3; }
      temp->c = q->c; p->link = temp; p = temp;
   }
   for ( q = s2->link; q != NULL; q = q->link ) {
      temp = (LinkString)malloc(sizeof(StrNode));
      if (temp == NULL) break;
      temp->c = q->c; p->link = temp; p = temp;
   }
   p->link = NULL;    return s3;
}
```


上面是基于顺序表和链接的字符串表示和实现

C语言中的字符串直接用字符数组表示，最后用结束符'\0'，串长度通过检查结束符确定，可看作一种实现方式

许多语言提供了标准字符串库，如C语言标准库有一组字符串函数（string.h），一些C语言系统提供的扩展的字符串库，C++ 语言标准库里的字符串库 <string>，Java 标准库的字符串库，许多脚本语言提供了功能丰富的字符串库

许多实际字符串库用动态顺序表结构作为字符串的表示方式。这样既能支持任意长的字符串，又能比较有效地实现各种重要的字符串操作

实际上，支持不同的字符串操作，可能需要不同的实现。例如要保存整本的书，将需要支持若干 MB 或更长的字符串，采用连续存储，也可能带来一些问题

3.3 模式匹配

模式匹配：在主串中做子串的定位

假设有两个串

$t=t_0\ t_1\ t_2 \cdots t_{n-1}$　　目标串

$p=p_0\ p_1\ p_2 \cdots p_{m-1}$　　模式串

通常m << n。模式匹配就是在目标串 t 中查找与模式串 p 相同的子串的过程（这一定义可以推广）

许多计算机应用的最基本功能就是字符串匹配。例：

在文本编辑器或字处理系统里，查找单词或句子（中文字或词语），替换单词或者词组；在程序里找拼写错误的标识符

email 程序的垃圾邮件过滤器，google 等网络搜索系统

生物学：DNA是很长的分子，存在于细胞核中。DNA 内有四种碱基：腺嘌呤（adenine），胞嘧啶（cytosine），鸟嘌呤（guanine），胸腺嘧啶（thymine）。其不同组合形成氨基酸、蛋白质和其他更高级的生命结构

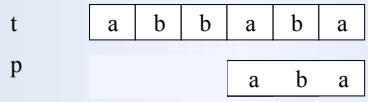
DNA 片段可看作是a,c,g,t构成的模式，如 acgatactagacagt。考查在蛋白质中是否出现某 DNA 片段，可看成与该 DNA 片段的模式匹配问题。DNA 分子可以切断和拼接，切断动作由各种酶完成，酶也是用特定的模式确定剪切位置

由于在计算机科学、生物信息学等许多领域的重要应用，串模式匹配已经变成了一个非常重要的计算问题

- 3.3.1 朴素的模式匹配
- 3.3.2 无回溯的模式匹配（KMP算法）

人们还提出了许多其他模式匹配算法

3.3.1 朴素的模式匹配



朴素的模式匹配过程

朴素匹配算法简单，易理解，但效率低

影响效率的主要原因是执行中有回溯：比较不等时将模式串p右移一个字符，重新从p_0（算法中p->c[0]）开始比较。

算法中没有利用前面已做过的字符比较得到的信息

最坏情况是每趟比较都在最后出现不等情况，最多比较n − m + 1趟，总比较次数为m*(n − m + 1)

由于一般有 m << n，所以算法时间为O(m*n)

最坏情况的实例

主　串：“00000000000000000000000000000000000001”

模式串：“000001”

朴素子串匹配算法

```
/* 求串 p 在串 t 中第一次出现的位置，即 p 的第一个元素在串
t 中的序号 (下标+1) */
int index0( PSeqString t,  PSeqString p ) {
   int i = 0, j = 0; /*初始化*/
   while ( i < p->n && j < t->n )   /*反复比较*/
      if (p->c[i] == t->c[j]) { /* 继续匹配下一个字符 */
         ++ i; ++ j;
      }
      else { /* i, j值回溯，再开始下一位置的匹配 */
         j -= i - 1; i = 0;
      }
   if (i >= p->n)  return( j - p->n + 1);  /* 匹配成功*/
   else return 0;            /* 匹配失败 */
}
```


3.3.2 无回溯的模式匹配

由 D. E. Knuth 和 V. R. Pratt 提出， J. H. Morris 几乎同时发现这一算法。因此又称为 KMP 算法

这是本课程中第一个非平凡的算法，基于对问题的深入分析和理解。这个算法比较复杂，但效率较高

为了理解KMP算法，首先要了解朴素算法的缺陷

现在仔细考察一下朴素匹配算法的执行过程。

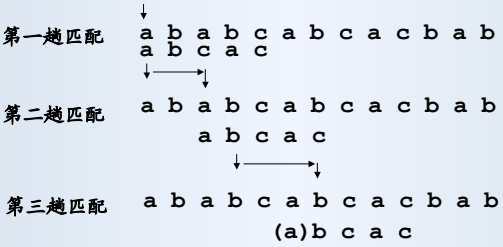
设有主串 *t* =“ababcabcacbab”, 模式串*p* =“abcac”。


```
第一趟匹配   a b a b c a b c a c b a b
             a b c a c
第二趟匹配   a b a b c a b c a c b a b
               a b c a c
第三趟匹配   a b a b c a b c a c b a b
                 a b c a c
第四趟匹配   a b a b c a b c a c b a b
                   a b c a c
第五趟匹配   a b a b c a b c a c b a b
                     a b c a c
第六趟匹配   a b a b c a b c a c b a b
                       a b c a c
```

模式串在匹配前已知，若先对模式串做细致的分析，记录一些有用信息（“静态”预处理），就可能加快匹配过程

KMP 算法的基本想法：每次匹配失败时，利用前面已经做过的匹配中得到的信息，尽可能地把模式串向结束的方向移动，匹配中只做那些不得不做的字符比较，不回溯

KMP算法匹配情况:



第二趟匹配　a b a b c a b c a c b a b
　　a b c a c
　　----------> a b c a c

为正确前移，需要确定匹配失败时前移的位置。对模式串 p 的每个 p_i，要找到某 p_{ki}, $0 \le k_i < i$，若匹配中 $p_i \neq t_j$，下步应该用 p_{ki} 去与 t_j 比较（匹配）

注意：当 p_i 匹配失败时，所有 p_k（$0 \le k < i$）是匹配成功的（否则不会去考虑 p_i 的匹配问题）。也就是说：

t 当前位置之前的 i 个字符就是 p 的前 i 个字符

关键想法：本应根据 t 的情况确定前移方式，但实际上可以根据 p 本身的情况确定，因此可以通过对模式串本身的分析，在匹配之前先做好

结论：与 p_i 对应的 k_i 值与被匹配的串无关。通过对模式串 p 的预先分析，可以得到每个 i 对应的 k_i 值

假设 p 的长度为 m，现在需要对每个 i ($0 \le i < m$) 算出一个 k_i 值并保存起来，以便在匹配中使用

这 m 个值（i 和 k_i 的对应关系）可存入一个数组 next，用 next[i]表示与 i 对应的 k_i 值

还有一种特殊情况：

可能发现当某 p_i 匹配失败时，用它之前的任何字符与 t_i 比较都无意义。这时应该从头开始用 p_0 与 t_{j+1} 比较。我们在 next[i] 里保存 -1 表示这种情况

例：

显然对于任何模式都有：next [0] = -1

假设数组 next 已经建立好了，匹配中 $p_i \neq t_j$ 时，通过它取得应与目标串当前字符匹配的模式串里的字符下标：

- 若next [i] ⩾0，右移 i - next[i] 个字符（也就是说，让i 取 next[i] 的值），下一步用 $p_{next[i]}$ 与 t_j 比较
- 若next [i] = -1，下一步用 p_0 与 t_{j+1} 比较

假设已有 next 数组，模式匹配：

```
while (i < p->n && j < t->n) { /*i, j 是两串的当前位置*/
    if (i == -1) { i++; j++; }
    else if (p->c[i] == t->c[j]) { i++; j++; }
    else  i = next[i] ; /*与朴素的{j = j-i+1; i = 0;}对应*/
}
```

前两个条件可以用 || 合并（执行的操作一样）

无回溯的模式匹配算法

```
int index (PSeqString t, PSeqString p, int *next) {
    int i = 0, j = 0; /*初始化*/
    while (i < p->n && j < t->n) {  /*反复比较*/
        if (i==-1 || p->c[i] == t->c[j]) {
            i++; j++;
        }
        else  i = next[i] ;
    }
    if (i >= p->n)
        return  j - p->n + 1; /*匹配成功*/
    return  -1;          /*匹配失败*/
}
```



```
/* 前面算法的另一等价写法 */
int index (PSeqString t, PSeqString p, int *next) {
    int i = 0, j = 0; /*初始化*/
    while (i < p->n && j < t->n) {  /*反复比较*/
        while (i >= 0 || p->c[i] != t->c[j])
            i = next[i];
        i++;
        j++;
    }
    if (i >= p->n) return  j - p->n + 1; /*匹配成功*/
    return  -1;/*匹配失败*/
}
```

本算法与后面求 next 数组的算法同出一辙，可以一起分析

建立 next 数组

看前面实例：

第二趟匹配　a b a b c a b c a c b a b
　　a b c a c
　　----------> a b c a c

- 应移到的位置，其前缀子串应该和匹配失败的字符前面同样长度的子串完全一样
- 如果在模式串里匹配失败时，前面一段中满足这个条件的位置不止一个，只能移到位置最近的那个位置（应保证不遗漏任何可能的匹配）

前缀：字符串开始的一段　　后缀：字符串最后的一段

next 数组的构造

由前面分析，k_i 值仅依赖于p本身的前i个字符

$t_0 \ldots t_{j-i-1} t_{j-i} \ldots t_{j-1}$ $t_j \ldots$
$\quad\quad\quad\quad \| \quad\quad \| \quad \nparallel$
$\quad\quad\quad\quad p_0 \ldots p_{i-1} p_i \ldots$ } 匹配到 p_i 时失败

$t_0 \ldots t_{j-i-1} p_0 \ldots p_{i-1} t_j \ldots$
$\quad\quad\quad\quad \| \quad\quad \| \quad \nparallel$
$\quad\quad\quad\quad p_0 \ldots p_{i-1} p_i \ldots$ } t 中位置 j 之前的 i 个字符就是 p 的前 i 个字符

前缀 后缀

$t_0 \ldots t_{j-i-1}\ p_0 \ldots p_{k-1} \ldots p_{i-k} \ldots p_{i-1}\ t_j \ldots$

$p_0 \ \ldots p_{k-1} p_k \ldots$

前缀

模式串正确移位，应保证$p_0 \ldots p_{k-1}$与t中对应字符匹配，也就是与$p_{i-k} \ldots p_{i-1}$匹配

易见，现在需要考虑的是模式串 p 前段（子串） $p_0 \ldots p_{i-1}$ 里相同的前缀与后缀:

- 把与当时后缀相同的前缀移来，前面一段保证匹配
- 如果把最大的相同前缀移来，就可保证不遗漏可能的匹配

若$p_0 \ldots p_{i-1}$中最大的相同前缀与后缀（不包括 $p_0 \ldots p_{i-1}$本身，但允许为空串）的长度为 k ($0 \le k < i-1$)。当 $p_i \ne t_j$ 时，p 就应右移 i - k 位，随后应比较 p_k 与 t_j

也就是说： next[i] 应该为k

求数组 next 的问题现在变成：对每个 i，求出 p 的（前缀）子串 $p_0 \ldots p_{i-1}$ 中最大的相同前缀与后缀的长度

KMP 提出了一种巧妙的递推算法

$p_0 \ldots \ldots \ldots p_{k-1} \ldots p_{i-k} \ldots \ldots \ldots p_{i-1}\ p_i\ p_{i+1} \ \ldots$
$\quad\quad\quad\quad\quad\quad\quad \| \quad\quad\quad\quad\quad \| \quad ?$
$\quad\quad\quad\quad\quad\quad\quad p_0 \ldots \ldots \ldots p_{k-1} p_k\ p_{k+1} \ldots$

如果成功， k++; i++;

如不成功，在 $p_0 \ldots \ldots \ldots p_{k-1} \ldots$ 里找 … …

利用next[0]= -1，…，next[i] 求next[i +1] 的算法:

1. 假设 next [i]=k，若$p_k = p_i$，则 $p_0 \ldots p_{i-k} \ldots p_i$ 中最大相同前后缀长度为next[i+1] = k+1
2. 若$p_k \ne p_i$ 置 k 为next[k] 后转到 1。（设 k = next[k]，就是考虑前一个更短的匹配前缀，从那里继续向下检查）
3. 若 k 值（来自next）为-1，就得到$p_0 \ldots p_{i-k} \ldots p_i$中最大相同前后缀的长度为k = 0（即next [i+1] = 0）

计算next数组的算法

/*next是指向next数组的指针参数。数组应足够大，至少应为p->n */

```
void makeNext(PSeqString p, int *next) {
    int i = 0, k = -1;
    next[0]= -1; /* 初始化 */
    while (i < p->n-1) {       /* 计算next[i+1] */
        while (k >= 0 && p->c[i] != p->c[k])
            k = next[k];
        i++; k++;
        next[i] = k;     /* 仅仅多这一句 */
    }
}
```


对算法的进一步考虑和改进

$t_0 \ldots t_{j-i-1} p_0 \ldots p_{j-1} t_j \ldots$
$\quad\quad\quad \| \quad\quad \| \quad \nparallel$
$\quad\quad\quad p_0 \ldots p_{i-1} p_i \ldots$

$t_0 \ldots t_{j-i-1}\ p_0 \ldots p_{i-k+1} \ldots\ p_{i-1}\ t_j \ldots$
$\quad\quad\quad\quad\quad \| \quad\quad\quad \| \quad \nparallel$
$\quad\quad\quad\quad\quad p_0 \quad \ldots \ p_{k-1} p_k \ldots$

当 $p_i \ne t_j$ 时，若$p_i = p_k$，那么一定有 $p_k \ne t_j$ 。所以模式串应再向右移 k-next[k]位，下一步用 $p_{next[k]}$ 与t_j比较。

对于next[i]=k 的改进:

```
if  (pk== pi)  next[i] = next[k];
else  next[i]=k;
```

这一改进可以避免一些不必要的操作（小改进）

计算next数组的算法（改进后）

/* next是指向next数组的指针参数 */

```
makeNext(PSeqString p, int *next) {
    int i = 0, k = −1;
    next[0] = −1;
    while (i < p->n − 1) {    /* 计算next[i+1] */
        while (k >= 0 && p->c[i] != p->c[k])
            k = next[k];
        i++;     k++;
        if(p->c[i] == p->c[k]) next[i] = next[k];
        else  next[i] = k;
    }
}
```

求next数组算法的时间复杂性:

设模式串长m

```
while (i < p->n - 1) {
   while (k >= 0 && p->c[i] != p->c[k])
        k = next[k];
   i++; k++;
   ... ... /* 这部分是常量时间，忽略 */
}
```

两重循环，貌似 $O(m^2)$。

实际不是。

- 外层循环每次将 i 加1，循环体总共执行m-1次
- 外层循环的 k++ 正好执行m-1次。k值从 –1 递增
- 内层循环的 k = next[k] 至少使 k 值减少1，但 k 值不可能小于 -1。因此内层循环最多总共执行 m-1 次
- 因此构造 next 数组的代价是 O(m)

表3.1 计算next数组

下标	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 与 p_i 比较		≠	≠	=	≠	=	≠	=	=
next[i]	-1	0	0	-1	1	0	2	0	0

KMP算法的一个重要优点是执行中不回溯。在处理从外部设备读入的庞大文件时，这种特性很有价值，因为可以一边读入一边匹配，不需要回头重读，因此不需要保存被匹配串

做好next数组之后，无回溯匹配时间复杂性是O(n)

无回溯模式匹配算法的使用

如果需要多次使用一个模式串，相应的 next 数组只需建立一次（如在大文件里反复找一个单词）

这种情况下，可以考虑将 next 数组作为模式串的一个成分（另外定义一个模式串类型）

人们还提出了其他的模式匹配算法

另一个经典算法由 Boyer 和 Moore 提出，采用自右向左的匹配方式。如果字符集比较大且匹配很罕见时（许多实际情况都是如此，例如在文章里找单词，在邮件里找垃圾过滤的关键字），其实际工作速度可能远远高于KMP算法

有兴趣的同学可以自己去找找相关材料

模式匹配的进一步问题

前面讨论的是最简单的模式匹配，其中的模式串就是普通的字符串，匹配就是找与之完全相同的子串

实际中可能需要更一般的匹配概念，例如:

- 查找以 re 开头后跟另外几个字母的单词
- 以 .c 结尾的任意文件名
- 文件里所有形为 href="..." 的段（HTML网页里的链接）
- DNA片段里以某碱基段开始，以另一碱基段结束的片段
- 股票数据中连续上升且其末值大于初值的 25% 的片段
- 计算机可执行文件中的某种片段模式（例如，检查病毒）

需要考虑描述能力更强的模式描述语言，更复杂的匹配算法

一种常见模式描述语言称为“正则表达式”，基本结构包括:

- 通配符：与一类或任意字符匹配，或与任意一段字符匹配
- 顺序组合 αβ：与前一段匹配 α，后一段匹配 β 的序列匹配
- 选择组合 α | β：与匹配 α 或者匹配 β 的序列匹配
- 星号 α*：与 0 段或者任意段与 α 匹配的序列匹配

有很多不同设计，许多脚本语言都提供了正则表达式功能，其中的正则表达式是上面结构的某种变形

C/C++/Java 等语言也有正则表达式程序包，一些常规语言正在或计划把正则表达式纳入标准库

例：（设 ? 与任意字符匹配，# 与任意一段匹配）

a?b#n　　re#ing　　c(a|e)*(s|t)on

采用功能强大的模式描述语言，使我们有可能描述更多的模式，但匹配算法的复杂性也会提高。这方面有许多理论结果

模式语言变得复杂后，匹配算法就可能变成具有指数复杂性的算法，因此就变得不实用了

如果模式语言进一步复杂，模式匹配问题甚至可能变得不可计算。也就是说，根本不可能写出算法

模式匹配问题还有许多扩展:

- 近似匹配
  - 串中数据是通过测量得到的，原本就不准确
  - 并不需要准确的匹配，近似就可以
- 其他模式匹配问题，例如二维或者高维中的模式匹配
- 等等

小　结

串是特殊的线性表，是由字符作元素组成的。它和线性表一样有顺序存储和链式存储两种方式。串作为一种抽象数据类型，有它自己的操作，在对串处理时，要抓住它的特殊要求。

模式匹配是比较复杂的串操作，完成子串在主串中的定位。朴素的模式匹配算法比较直观，易于理解；但无回溯的模式匹配算法的技巧性很强，其中蕴涵了对问题的深刻理解，读懂它需要花费一定时间。从这个算法里可以看到人们如何研究问题，真正弄懂了这个算法，会极大地增强你的学习算法的兴趣。