

面向21世纪课程教材
普通高等教育“十一五”国家级规划教材
北京市高等教育精品教材
教育部普通高等教育精品教材

算法与数据结构

第三讲：字符串

张乃孝等编著

《算法与数据结构—C语言描述》

3 字符串

- 3.1 字符串及其抽象数据类型
- 3.2 字符串的实现
- 3.3 模式匹配

3.1 字符串及其抽象数据类型

- 基本概念

- 字符串,简称串,一种特殊的线性表,表中的每个元素都是一个字符。
- 一个串可以记为 $s = "s_0 s_1 \dots s_{n-1}"$ ($n \geq 0$)
 - s 是串的名字;
 - 字符序列 $s_0 s_1 \dots s_{n-1}$ 是串的值;
 - 字符个数称为该串的长度。
- 长度为0的串称为空串,写成 $s = ""$,注意与空白字符构成的串 $s = " "$ 相区分。

子串、主串与子串位置

- 子串

- 字符串s1中任意个连续的字符组成的子序列s2被称为是s1的**子串**，而称s1是s2的**主串**。
 - 空串是任意串的子串；
 - 除s外，s的其他子串称为s的**真子串**。
- 子串在主串中的**位置**：该子串的第一个字符在主串中的位置。

A= " PEKINGUNIVERSITY "

1 7

B= " UNIVERSITY "

字符串的相等关系与字典序关系

- 两个字符串**相等**:
 - 两个字符串的长度相等;
 - 且各个对应位置上的字符都相同。
- 如果整个字符集上有全（线）序关系，则两个字符串之间有如下**字典序关系**:
 - 设 $A = a_0a_1 \cdots a_{n-1}$, $B = b_0b_1 \cdots b_{m-1}$, 则 $A < B$:
 - 若存在 k 使 $a_i = b_i$ ($i = 0, 1, \cdots k-1$) , 但是 $a_k < b_k$,
 - 或者 $n < m$, 且 $a_i = b_i$ ($i = 0, 1, \cdots n-1$)。

抽象数据类型

**ADT String is
operations**

String createNullStr (void)

创建一个空串。

int IsNullStr (String s)

判断串s是否为空串，若为空串，则返回1，否则返回0。

int length (String s)

返回串s的长度。

String concat (String s1, Sting s2)

返回将串s1和s2拼接在一起构成的一个新串。

String subStr (String s, **int** i, **int** j)

在串s中，求从串的第i个字符开始连续j个字符所构成的子串。

int index (String s1, String s2)

如果串s2是s1的子串，则可求串s2在串s1中第一次出现的位置。

end ADT String

3.2 字符串的实现

顺序表示

字符串的顺序表示，就是把串中的字符，顺序地存储在—组地址连续的存储单元中。其类型定义为：

```
struct SeqString {           /* 顺序串的类型 */
    int  MAXNUM;             /* 串允许的最大字符个数 */
    int  n;                  /* 串的长度，n≤MAXNUM */
    char *c;
};
typedef struct SeqString *PSeqString;
```

顺序表示举例

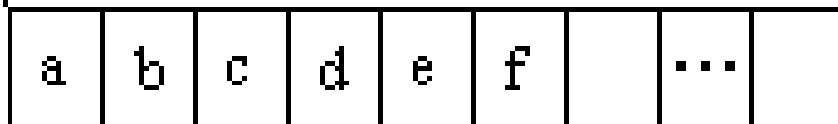
- 串 $s = \text{"abcdef"}$ ，用顺序表示方式，假设 s 是 `struct SeqString` 类型的变量，那么它的元素在数组中的存放方式如下图所示：

`s.n = 6`

`s.c`

元素

下标



a	b	c	d	e	f		...
---	---	---	---	---	---	--	-----

0 1 2 3 4 5 6

创建空顺序串

```
PSeqString createNullStr_seq( int m ) {  
    PSeqString pstr = (PSeqString)malloc(sizeof(struct  
        SeqString));  
    if (pstr!=NULL) {  
        pstr->c= (char* )malloc(sizeof (char)*m);  
        if(pstr->c) {  
            pstr->n=0;  pstr->MAXNUM=m;  
            return  pstr;  
        }  
        else free  (pstr);  
    }  
    printf("Out of space!!\n");  
    return NULL;  
}
```

求顺序表示的串的子串

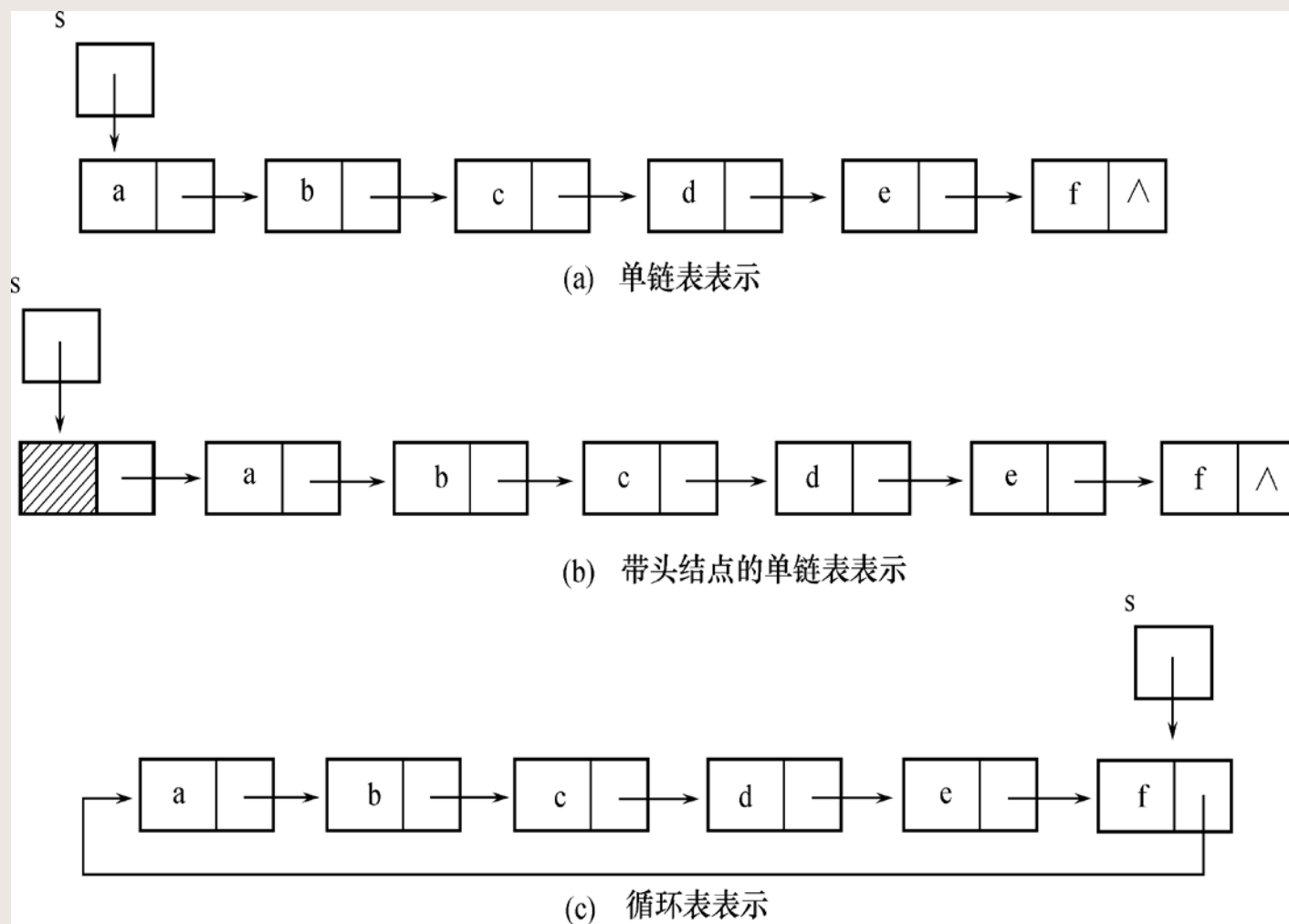
```
PSeqString subStr_seq(PSeqString s, int i, int j) {
PSeqString s1;
    int k;
    s1 = createNullStr_seq( j); /* 创建一空串 */
    if (s1==NULL) return NULL ;
    if ( i>0 && i<=s->n && j>0 ) {
        if ( s->n<i+j-1 )    j = s->n-i+1;
        /*若从i开始取不了j个字符, 则能取几个就取几个*/
        for (k=0;k<j;k++)    s1->c[k]=s->c[i+k-1];
        s1->n=j;
    }
    return s1 ;
}
```

链接表示

- 在串的链接表示中，每个结点包含两个字段：字符和指针，分别用于存放字符和指向下一个结点的指针。这样一个串就可用一个单链表来表示，其类型定义为：

```
struct StrNode;                /* 链串的结点 */
typedef struct StrNode *PStrNode; /* 结点指针类型 */
struct StrNode {                /* 链串的结点结构 */
    char    c;
    PStrNode link;
};
typedef struct StrNode *LinkString; /* 链串的类型 */
```

链接表示举例 $s = \text{"abcdef"}$



创建带头结点的空链表

```
LinkString createNullStr_link( void ) {  
/*创建带头结点的空链表*/  
LinkString pst;  
pst = (LinkString)malloc( sizeof(struct StrNode) );  
if (pst!=NULL) pst->link = NULL;  
else printf("Out of space! \n"); /*创建失败*/  
return pst ;  
}
```

求单链表示的串的子串

LinkString subStr_link(LinkString s,int i,int j)

求从s所指的带头结点的链串中第 $i(i>0)$ 个字符开始连续取 j 个字符所构成的子串。

这里首先要为链串结构和头结点申请空间，创建一个空链表，这由前面的算法可以实现。

然后判断所给参数 i ， j 的值是否合理， i ， j 的取值应为 $i>0$ ， $j>0$ 。

接着从 $s \rightarrow \text{head}$ 开始找第 i 个结点，找到后，就从该结点开始，为子串中的结点申请空间，并将元素值复制过去。

LinkString subStr_link (LinkString s,int i,int j)

```
{
    LinkString s1;
    PStrNode p,q,t;
    int k;
    /* 创建一空串 */
    s1 = createNullStr_link( );
    if( s1 == NULL ) {
        printf( "Out of space!\n" );
        return NULL ;
    }
    if (i<1 || j<1 ) return s1 ;
    /* i,j值不合适, 返回空串 */
    p = s; /* p为主串*/
    for (k=1;k<=i;k++)
        /*找第i个结点*/
        if ( p != NULL) p = p->link;
        else return s1 ;
    if (p==NULL) return s1 ;
    t = s1; /* t为子串尾*/

    for (k=1;k<=j;k++)
        /*连续取j个字符*/
        if (p!=NULL) {
            q = (PStrNode)malloc(sizeof(struct
                StrNode));
            if (q==NULL) {
                printf( "Out of space!\n" );
                return s1 ;
            }
            q->c = p->c;
            q->link = NULL;
            t->link = q;
            /* 结点放入子链串中 */
            t = q;
            p = p->link;
        }
    return s1 ;
}
```

字符串的实现：总结

- 无论是顺序表示还是链接表示，都可以看作特殊的线性表的实现方式。
- C语言中的字符串直接有字符数组表示，以“\0”作为串结束的符号，也可以看作一种实现方式。
- 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）。
- 支持不同的字符串操作，可能需要不同的实现。

3.3 模式匹配

`int index(String s1, String s2)`

如果串s2是s1的子串, 则可求串s2在串s1中第一次出现的位置。

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 $m \ll n$ 。

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。

应用：拼写检查、语言翻译、数据压缩、搜索引擎、入侵检测、病毒特征码匹配、DNA序列匹配……

朴素的模式匹配

t

a	b	b	a	b	a
---	---	---	---	---	---

p

a	b	a
---	---	---

t

a	b	b	a	b	a
---	---	---	---	---	---

p

a	b	a
---	---	---

t

a	b	b	a	b	a
---	---	---	---	---	---

p

a	b	a
---	---	---

t

a	b	b	a	b	a
---	---	---	---	---	---

p

a	b	a
---	---	---

朴素的模式匹配算法

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 index( 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 = 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 */  
}
```

算法分析

- 朴素的模式匹配算法直观简单, 易理解
- 在最坏的情况下, 每趟比较都在最后出现不等, 最多比较 $n-m+1$ 趟, 总比较次数为 $m \times (n-m+1) = O(m \times n)$
 - 例如: $t=000000000000000001$, $p=001$
 - 匹配过程有回溯

- 设有主串 $t = \text{"ababbabbabababab"}$, 模式串 $p = \text{"ababa"}$ 。

第一趟匹配

abab**b**abbababa
abab**a**

能否跳到第六趟？

第二趟匹配

ababbabababa
ababa

第三趟匹配

abababbababa
ababab

第四趟匹配

abababbababa
ababa

第五趟匹配

abab**b**abbababa
aababa

第六趟匹配

ababb**a**bababa
ab**a**ba

第七趟匹配

ababbab**b**ababa
ab**a**ba

第八趟匹配

ababbab**b**ababa
ab**a**ba

第九趟匹配


ababbabb**a**ba
ab**a**ba

是否可以
从第六趟
直接到第
九趟?

朴素的模式匹配算法效率不高
影响效率的关键因素是有不必要的回溯
算法中没有利用前面已进行的字符比较得到的信息

提高匹配速度？

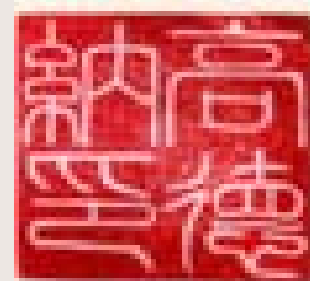
- 如果在匹配过程中一旦 p_i 和 t_j 不相等, 即:
 - $p_0=t_{j-i}, p_1=t_{j-i+1}, \dots, p_{i-1}=t_{j-1}, p_i \neq t_j$,
 - 希望能找到一个大于等于1的右移的位数, 确定 p 和 t 继续比较的字符
 - 希望匹配过程对于 t 是无回溯的:
 - 右移若干位后, 立即用 p 中一个新的字符 p_k 和 $t_j(t_{j+1})$ 继续进行
 - 最好能通过对于 p 的分析得到右移位置 (p_k)



关键问题:
如何找到这
个 k

无回溯的匹配模式 (KMP算法)

- 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。这个算法并不太复杂, 但非常巧妙, 效率较高。



KMP算法的数组next

- 与 p_i 对应的k值与被匹配的目标串无关。通过对模式串 p 的预先分析，可以得到每个 i 对应的 k 值。
- 假设 p 的长度为 m ，现在需要对每个 i ($0 \leq i < m$) 算出一个 k 值并保存起来，以便在匹配中使用。
- 把这 m 个值存入一个数组 **next**，用 $\text{next}[i]$ 表示与 i 对应的 k 值。
- 一种特殊情况：当某 p_i 匹配失败时，用它之前的任何字符与 t_j 比较都无意义，这时应该用 p_0 从头开始与 t_{j+1} 比较。

我们在 $\text{next}[i]$ 里保存 -1 表示这种特殊情况，
显然，对于任何模式都有： $\text{next}[0] = -1$ 。

假设数组next已经建立好

- 无回溯的模式匹配算法的基本思想：匹配中 $p_i \neq t_j$ 时，通过next取得应与目标串当前字符匹配的模式串里的字符下标：
 - 若 $\text{next}[i] \geq 0$ ，右移 $i - \text{next}[i]$ 个字符（也就是说，让i取 $\text{next}[i]$ 的值），下一步用 $p_{\text{next}[i]}$ 与 t_j 比较；
 - 若 $\text{next}[i] = -1$ ，下一步用 p_0 与 t_{j+1} 比较。

模式匹配的核心循环

```
while ( i < p->n && j < t->n )  
    if (p->c[i] == t->c[j]) { ++ i; ++ j; }  
    else { j = j - i + 1; i = 0; }
```

- 假设数组next已经建好:

```
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(蓝色)可以用 || 合并 (执行的操作一样)

```
while ( i < p->n && j < t->n ) {  
    if (i == -1 || p->c[i] == t->c[j]) { i++; j++; }  
    else i = next[i] ;  
}
```

无回溯的模式匹配算法

```
int pMatch (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,返回p中第一个字符在t中的序号*/  
    return 0; /*匹配失败*/  
}
```

next数组的性质

匹配中出现 $p_i \neq t_j$ 时:

$$\begin{array}{ccccccc} t_0 \cdots t_{j-i-1} & t_{j-i} & \cdots & t_{j-1} & t_j & \cdots \\ = & & = & & \neq & \\ p_0 \cdots p_{i-1} & p_i & \cdots & & & \end{array}$$



$$\begin{array}{ccccccc} t_0 \cdots t_{j-i-1} & p_0 & \cdots & p_{i-1} & t_j & \cdots \\ = & & = & & \neq & \\ p_0 \cdots p_{i-1} & p_i & \cdots & & & \end{array}$$



后缀

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

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

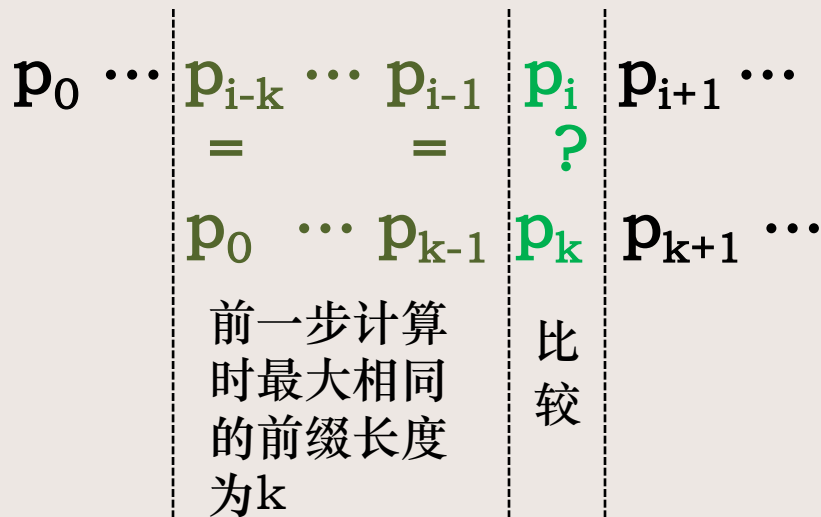
前缀

next数组的存在

- 需要考虑的是模式串 p 的子串 $p_0 \cdots p_{i-1}$ 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 $p_0 \cdots p_{i-1}$ 中最大的相同前缀与后缀(不包括 $p_0 \cdots p_{i-1}$ 本身，但允许为空串)的长度为 k ($0 \leq k \leq i-1$)。当 $p_i \neq t_j$ 时， p 就应右移 $i - k$ 位，随后应比较 p_k 与 t_j ，即 $\text{next}[i]$ 应该为 k 。

next数组计算

- 求数组 next 的问题现在变成：对每个 i , 求出 p 的子串 $p_0 \cdots p_{i-1}$ 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：



检查长度为 $k+1$ 的前后缀是否相等

next数组计算

- 求数组 next 的问题现在变成：对每个 i , 求出 p 的子串 $p_0 \cdots p_{i-1}$ 中最大的相同前缀与后缀的长度。
- 初值：对于任何模式都有 $\text{next}[0] = -1$ 。

如果 $\text{next}[0]$ 到 $\text{next}[i]$ 都已经计算出来，如何计算 $\text{next}[i+1]$ 的值？

- 假设 $\text{next}[i] = k$ ，首先判断 $\text{next}[i+1]$ 的值是否等于 $k+1$ 。

$$\begin{array}{ccccccc} p_0 & \cdots & p_{i-k} & \cdots & p_{i-1} & p_i & p_{i+1} \cdots \\ & & = & & = & = & \\ & & p_0 & \cdots & p_{k-1} & p_k & p_{k+1} \cdots \end{array}$$

$p_i = p_k$ ，直接得到结果 $\text{next}[i+1] = k+1$ （无回溯！）

next数组计算

$$\begin{array}{ccccccc} p_0 & \cdots & p_{i-k} & \cdots & p_{i-1} & p_i & p_{i+1} \cdots \\ & & = & & = & \neq & \\ & & p_0 & \cdots & p_{k-1} & p_k & p_{k+1} \cdots \\ & & & & & ? & \\ & & & & & p_0 & \cdots p_{\text{next}[k]} \cdots \end{array}$$

- 使用next[k]，无回溯的匹配方法，直接考虑已有比较短的前后缀。
- 跳到比较 $p_{\text{next}[k]}$ 和 p_i ，检查长度为next[k+1]的前后缀是否相等（回到前面处理）。

递推计算next数组

- 利用 $\text{next}[0] = -1, \dots, \text{next}[i]$ 求 $\text{next}[i+1]$ 的算法:
 - 1) 假设 $\text{next}[i] = k$, 若 $p_k = p_i$, 则 $p_0 \cdots p_{i-k} \cdots p_i$ 中最大相同前后缀长度为 $\text{next}[i+1] = k+1$ 。
 - 2) 若 $p_k \neq p_i$ 置 k 为 $\text{next}[k]$, 然后转到 1。
(设 $k = \text{next}[k]$, 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 $p_0 \cdots p_{i-k} \cdots p_i$ 中最大相同前后缀的长度为 $k = 0$ (即 $\text{next}[i+1] = 0$)。

计算next数组的算法

```
/*next是指向next数组的指针参数。*/  
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;        / * ? ? * /  
    }  
}
```

对算法的进一步改进

- 当 $p_i \neq t_j$ 时,若 $p_i = p_k$, 那么一定有 $p_k \neq t_j$.所以模式串应再向右移 $k - \text{next}[k]$ 位,下一步用 $p_{\text{next}[k]}$ 与 t_j 比较;
- 对于 $\text{next}[i]=k$ 的改进:
if ($p_k == p_i$) $\text{next}[i] = \text{next}[k]$;
else $\text{next}[i]=k$;
- 这一改进可以避免一些不必要的操作。

计算next数组（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;  
    }  
}
```

算法分析

- 关于算法makeNext设模式串长 m 两重循环, 貌似 $O(m^2)$?
 - 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 $k = \text{next}[k]$ 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 $O(m)$
- 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度是 $O(n)$
- 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)。

例子

- 考虑 $t = \text{"aabcbaabcaabababc"}$
 $p = \text{"abcaababc"}$

下标	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

例子

- 考虑 $t = \text{"aabcbbabcaabcaababc"}$
 $p = \text{"abcaababc"}$

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		\neq	\neq	$=$	\neq	$=$	\neq	$=$	$=$
next[i]	-1	0	0	-1	1	0	2	0	0

第一趟匹配

a**a**bc**b**abcaabcaababc
abcaababc

第二趟匹配

aab**c****b**abcaabcaababc
ab**c**aababc

第三趟匹配

aabcbab**c**aab**c**aababc
abcaab**a**bc

第四趟匹配

aabcbabcaab**c**aababc
ab**c**aababc

小结

- 字符串是由字符作元素组成的线性表。但是作为一种抽象数据类型，有它自己的操作，在对串处理时，要抓住它的特殊性。串和线性表一样有顺序存储和链式存储两种方式。
- 模式匹配是子串在主串中的定位操作，是一个比较常用的操作。朴素的模式匹配算法比较直观，易于理解；但是效率比较低。
- 无回溯的模式匹配算法的技巧性很强，实现无回溯的模式匹配的基础是依靠next数组支持。计算next数组的算法，实质上还是一个无回溯的模式匹配算法。对于这个算法的改进是巧上加巧。

本讲重点

- 算法是人类智慧的结晶。关于算法的研究已经有数千年的历史。公元前三百多年，在Elements一书中，欧几里德就给出著名的最大公因数的求解算法。
- 计算机的出现，使得用机器自动解题的梦想成为现实。计算机的广泛应用也开拓了研究算法的许多新领域和新方法。
- 本章的重点在第3节，详细讨论了无回溯的模式匹配算法。通过本章的学习，希望大家了解计算机科学的本质，提高算法（和程序）设计的兴趣。充分发挥个人的智慧，学好数据结构课。