

第4章 串

基本学习要点如下:

- 理解串和一般线性表之间的差异。
- 重点掌握在顺序串上和链串上实现串的基本运算算法。
- 掌握串的模式匹配算法。
- 灵活运用串这种数据结构解决一些综合应用问题。

4.1 串的基本概念

关联的知识点



串的概念



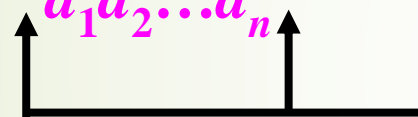
串抽象数据类型的描述方法

串（或字符串），是由零个或多个字符组成的有限序列。
含零个字符的串称为空串，用 Φ 表示。

串中所含字符的个数称为该串的长度（或串长）。


串的逻辑表示：

“ $a_1a_2\cdots a_n$ ”



双引号不是串的内容，起标识作用

每个 a_i ($1 \leq i \leq n$) 代表一个字符。




当且仅当两个串的长度相等并且各个对应位置上的字符都相同时，这两个串才是相等的。

如：

“abcd”≠“abc”

“abcd”≠“abcde”

但所有空串是相等的。



一个串中任意个连续字符组成的子序列（含空串）称为该串的子串。

例如，“abcde”的子串有：

“”、“a”、“ab”、“abc”、“abcd”和“abcde”等

真子串是指不包含自身的所有子串。



思考题：

串和线性表有什么异同？

【例（补充）】 问题：“abcde”有多少个真子串？

解：

动画演示

空串数:1 👉 “”

含1个字符的子串数:5 👉 “a”， “b”， “c”， “d”， “e”

含2个字符的子串数:4 👉 “ab”， “bc”， “cd”， “de”

含3个字符的子串数:3 👉 “abc”， “bcd”， “cde”

含4个字符的子串数:2 👉 “abcd”， “bcde”

共有 $1+2+3+4+5=15$ 个真子串。

推广：含有 n 个互不相同字符的串有 $n(n+1)/2$ 个真子串。

案例引入



案例4.1：病毒感染检测

研究者将人的DNA和病毒DNA均表示成由一些字母组成的字符串序列。

然后检测某种病毒DNA序列是否在患者的DNA序列中出现过，如果出现过，则此人感染了该病毒，否则没有感染。

例如，假设病毒的DNA序列为baa，患者1的DNA序列为aaabbba，则感染，患者2的DNA序列为babbbba，则未感染。

（注意，人的DNA序列是线性的，而病毒的DNA序列是环状的）

病毒感染检测输入数据.txt - 记事本

文件(F) 编辑(E) 格式(O) 查看(V)

```
10
baa      bbaabbba
baa      aaabbbba
aabb     abceaabb
aabb     abaabcea
abcd     cdabbbab
abcd     cabbbbab
abcde    bcdebdba
acc      bdedbcda
cde      cdcdcdcc
cced     cdccdcce
```

病毒感染检测输出结果.txt - 记事本

文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

```
baa      bbaabbba      YES
baa      aaabbbba      YES
aabb     abceaabb      YES
aabb     abaabcea      YES
abcd     cdabbbab      YES
abcd     cabbbbab      NO
abcde    bcdebdba      NO
acc      bdedbcda      NO
cde      cdcdcdcc      YES
cced     cdccdcce      YES
```

串抽象数据类型=逻辑结构+基本运算（运算描述）

串的基本运算如下：

- **StrAssign(&s,cstr):** 将字符串常量cstr赋给串s，即生成其值等于cstr的串s。
- **StrCopy(&s,t):** 串复制。将串t赋给串s。
- **StrEqual(s,t):** 判串相等。若两个串s与t相等则返回真；否则返回假。
- **StrLength(s):** 求串长。返回串s中字符个数。
- **Concat(s,t):** 串连接:返回由两个串s和t连接在一起形成的新串。
- **SubStr(s,i,j):** 求子串。返回串s中从第 i ($1 \leq i \leq n$) 个字符开始的、由连续 j 个字符组成的子串。

- **InsStr(s1,i,s2):** 插入。将串s2插入到串s1的第 i ($1 \leq i \leq n+1$) 个字符中, 即将s2的第一个字符作为s1的第 i 个字符, 并返回产生的新串。
- **DelStr(s,i,j):** 删除。从串s中删去从第 i ($1 \leq i \leq n$) 个字符开始的长度为 j 的子串, 并返回产生的新串。
- **RepStr(s,i,j,t):** 替换。在串s中, 将第 i ($1 \leq i \leq n$) 个字符开始的 j 个字符构成的子串用串t替换, 并返回产生的新串。
- **DispStr(s):** 串输出。输出串s的所有元素值。



串处理应用程序



各种串的基本运算算法



4.2 串的存储结构

关联的知识点

串的顺序存储结构及顺序串算法设计方法

串的链式存储结构及链串算法设计方法



串可以采用哪些存储结构 ?

4.2.1 串的顺序存储及其基本操作实现

在顺序串中，串中的字符被依次存放在一组连续的存储单元里。一般来说，一个字节（8位）可以表示一个字符（即该字符的ASCII码）。

因此，一个内存单元可以存储多个字符。例如，一个32位的内存单元可以存储4个字符（即4个字符的ASCII码）。

串的顺序存储有两种方法：一种是每个单元只存一个字符，这称为非紧缩格式（其存储密度小）；另一种是每个单元存放多个字符，这称为紧缩格式（其存储密度大）。

1001	A			
1002	B			
1003	C			
1004	D			
1005	E			
1006	F			
1007	G			
1008	H			
1009	I			
100a	J			
100b	K			
100c	L			
100d	M			
100e	N			

非紧缩格式示例

1001	A	B	C	D
1002	E	F	G	H
1003	I	J	K	L
1004	M	N		

紧缩格式示例

对于非紧缩格式的顺序串，其类型定义如下：

```
#define MaxSize 100
typedef struct
{   char data[MaxSize];
    int length;
} SqString;
```

其中**data**域用来存储字符串，**length**域用来存储字符串的当前长度，**MaxSize**常量表示允许所存储字符串的最大长度。在C语言中每个字符串以'\0'标志结束。

顺序串中实现串的基本运算如下。

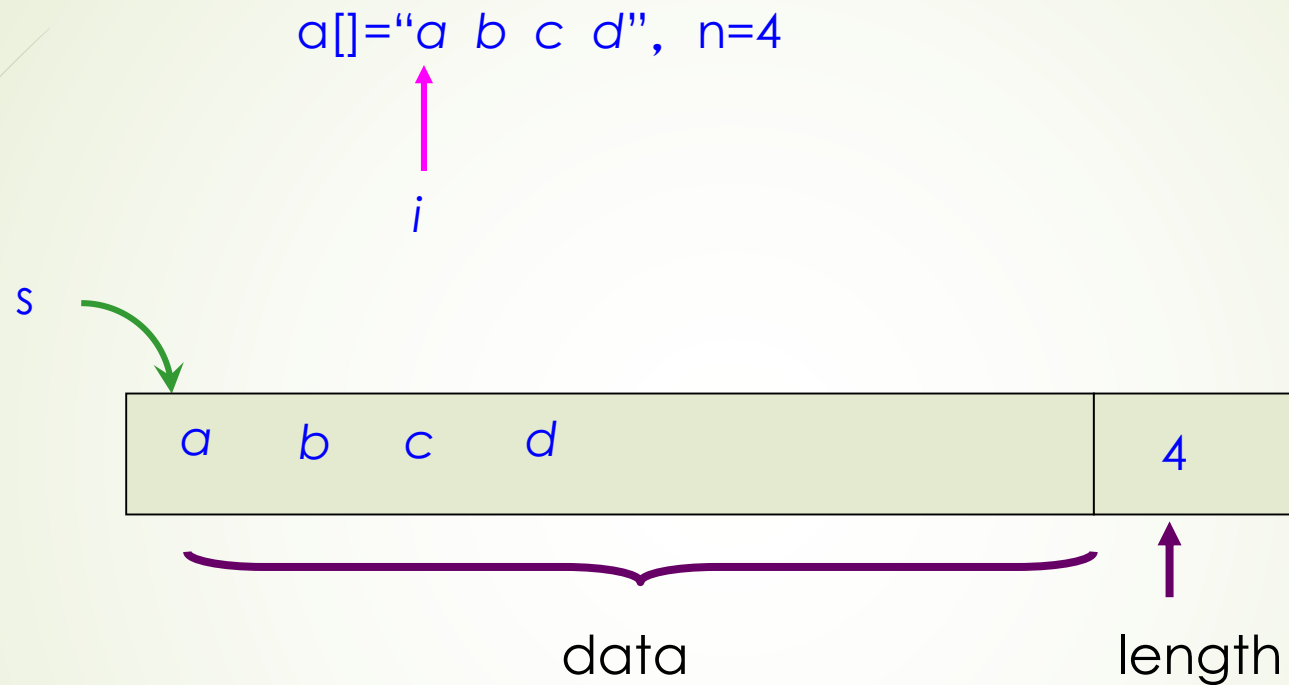
(1) StrAssign(s,cstr)

将一个字符串常量赋给串s，即生成一个其值等于cstr的串s。

```
void StrAssign(SqString &s, char cstr[]) //s为引用型参数
{
    int i;
    for (i=0; cstr[i]!='\0'; i++)
        s.data[i]=cstr[i];
    s.length=i;
}
```

↑
建立顺序串的算法。

建立顺序串动画演示



顺序串建立完毕。

(2) StrCopy(s,t)

将串t复制给串s。

```
void StrCopy(SqString &s, SqString t) //s为引用型参数
{
    int i;
    for (i=0; i<t.length; i++)
        s.data[i]=t.data[i];
    s.length=t.length;
}
```

(3) StrEqual(s,t)

判串相等：若两个串s与t相等返回真（1）；否则返回假（0）。

```
bool StrEqual(SqString s, SqString t)
{
    bool same=true;
    int i;
    if (s.length!=t.length) //长度不相等时返回0
        same=false;
    else
        for (i=0;i<s.length;i++)
            if (s.data[i]!=t.data[i])
                {
                    same=false;
                    break;
                }
    return same;
}
```

(4) StrLength(s)

求串长：返回串s中字符个数。

```
int StrLength(SqString s)
{
    return s.length;
}
```


(5) Concat(s,t)



串连接：返回由两个串s和t连接在一起形成的新串。

s为"abcd", t为"123"



`s1=Concat(s,t)`

s1为"abcd123"



```
SqString Concat(SqString s, SqString t)
{
    SqString str;
    int i;
    str.length=s.length+t.length;
    for (i=0;i<s.length;i++) //s.data[0..s.length-1]→str
        str.data[i]=s.data[i];
    for (i=0;i<t.length;i++) //t.data[0..t.length-1]→str
        str.data[s.length+i]=t.data[i];
    return str;
}
```

(6) SubStr(s,i,j)



求子串：返回串s中从第i ($1 \leq i \leq \text{StrLength}(s)$) 个字符开始的、由连续j个字符组成的子串。参数不正确时返回一个空串。

s为" a**bcd**123"



$s1 = \text{SubStr}(s, 2, 4)$

s1为" bcd1"



```
SqString SubStr(SqString s,int i,int j)
{
    SqString str;
    int k;
    str.length=0;
    if (i<=0 || i>s.length || j<0 || i+j-1>s.length)
        return str;    //参数不正确时返回空串
    for (k=i-1;k<i+j-1;k++) //s.data[i..i+j]→str
        str.data[k-i+1]=s.data[k];
    str.length=j;
    return str;
}
```

(7) InsStr(s1,i,s2)



将串s2插入到串s1的第i ($1 \leq i \leq \text{StrLength}(s1)+1$) 个字符中, 即将s2的第一个字符作为s1的第i个字符, 并返回产生的新串。参数不正确时返回一个空串。

s1为" abcd", s2为" 123"



`s3=InsStr(s1,2,s2)`

s3为" a123bcd"



```
SqString InsStr(SqString s1,int i,SqString s2)
{  int j;  SqString str;
   str.length=0;
   if (i<=0 || i>s1.length+1) //参数不正确时返回空串
       return str;
   for (j=0;j<i-1;j++)          //将s1.data[0..i-2]→str
       str.data[j]=s1.data[j];
   for (j=0;j<s2.length;j++) //s2.data[0..s2.length-1]→str
       str.data[i+j-1]=s2.data[j];
   for (j=i-1;j<s1.length;j++) //s1.data[i-1..s1.length-1]→str
       str.data[s2.length+j]=s1.data[j];
   str.length=s1.length+s2.length;
   return str;
}
```

(8) DelStr(s,i,j)



从串s中删去第i ($1 \leq i \leq \text{StrLength}(s)$) 个字符开始的长度为j的子串，并返回产生的新串。参数不正确时返回一个空串。

s为" a**bcd**123"



$s1 = \text{DelStr}(s, 2, 4)$

s1为" a23"



```
SqString DelStr(SqString s,int i,int j)
{  int k; SqString str;
   str.length=0;
   if (i<=0 || i>s.length || i+j>s.length+1)
       return str;           //参数不正确时返回空串
   for (k=0;k<i-1;k++)      //s.data[0..i-2]→str
       str.data[k]=s.data[k];
   for (k=i+j-1;k<s.length;k++)
       //s.data[i+j-1..s.length-1]→str
       str.data[k-j]=s.data[k];
   str.length=s.length-j;
   return str;
}
```

(9) RepStr(s,i,j,t)

在串s中，将第i ($1 \leq i \leq \text{StrLength}(s)$) 个字符开始的j个字符构成的子串用串t替换，并返回产生的新串。参数不正确时返回一个空串。

s为"abcde fg", t为"123"

$s1 = \text{RepStr}(s, 2, 4, t)$

s1为" a123fg"

```
SqString RepStr(SqString s,int i,int j,SqString t)
{  int k; SqString str; str.length=0;
   if (i<=0 || i>s.length || i+j-1>s.length)
       return str;      //参数不正确时返回空串
   for (k=0;k<i-1;k++)    //s.data[0..i-2]→str
       str.data[k]=s.data[k];
   for (k=0;k<t.length;k++)
       //t.data[0..t.length-1]→str
       str.data[i+k-1]=t.data[k];
   for (k=i+j-1;k<s.length;k++)
       //s.data[i+j-1..s.length-1]→str
       str.data[t.length+k-j]=s.data[k];
   str.length=s.length-j+t.length;
   return str;
}
```

(10) DispStr(s)

输出串s的所有元素值。

```
void DispStr(SqString s)
{   int i;
    if (s.length>0)
    {   for (i=0;i<s.length;i++)
        printf("%c",s.data[i]);
        printf("\n");
    }
}
```

【例4.1:p103】设计顺序串上实现串比较运算Strcmp(s,t)的算法。



解：本例的算法思路如下：

(1) 比较s和t两个串共同长度范围内的对应字符:

- ① 若s的字符 $>$ t的字符，返回1；
- ② 若s的字符 $<$ t的字符，返回-1；
- ③ 若s的字符 $=$ t的字符,按上述规则继续比较。

(2) 当(1)中对应字符均相同时，比较s和t的长度:

- ① 两者相等时，返回0；
- ② s的长度 $>$ t的长度，返回1；
- ③ s的长度 $<$ t的长度，返回-1。



```
int Strcmp(SqString s, SqString t)
{
    int i, comlen;
    if (s.length < t.length) comlen = s.length; // 求s和t的共同长度
    else comlen = t.length;
    for (i = 0; i < comlen; i++) // 在共同长度内逐个字符比较
        if (s.data[i] > t.data[i])
            return 1;
        else if (s.data[i] < t.data[i])
            return -1;
    if (s.length == t.length) // s == t
        return 0;
    else if (s.length > t.length) // s > t
        return 1;
    else return -1; // s < t
}
```

【例4.2:p104】以顺序串作为串的存储结构，设计一个算法求串s中出现的最长的连续相同字符构成的平台。

解：用index保存最长的平台在s中的开始位置，max保存其长度，先将它们初始化为0。扫描串s，计算局部重复子串的长度length，若较max大，则更新max，并用index记下其开始位置。

0	1	2	3	4	5	6	7	8	9	10
a	b	c	c	c	a	d	a	a	a	a
↑	↑	↑		↑			↑			↑
i	i	i		i						

index=2 max=3

index=7 max=4


```
void LongestString(SqString s,int &index,int &max)
{  int length=1,i=0,start=0;  //length保存平台的长度
  index=0,max=0;  //index保存最长平台在s中的开始位置, max保存其长度
  while (i<s.length-1)
  {  if (s.data[i]==s.data[i+1])
    {  i++;
      length++;
    }
    else  //上一个平台结束
    {  if (max<length)  //当前平台长度大, 则更新max
      {  max=length;
        index=start;
      }
      i++;start=i;  //初始化下一个平台的起始位置和长度
      length=1;
    }
  }
  if (max<length)  //当前平台长度大, 则更新max
  {  max=length;
    index=start;
  }
}
```

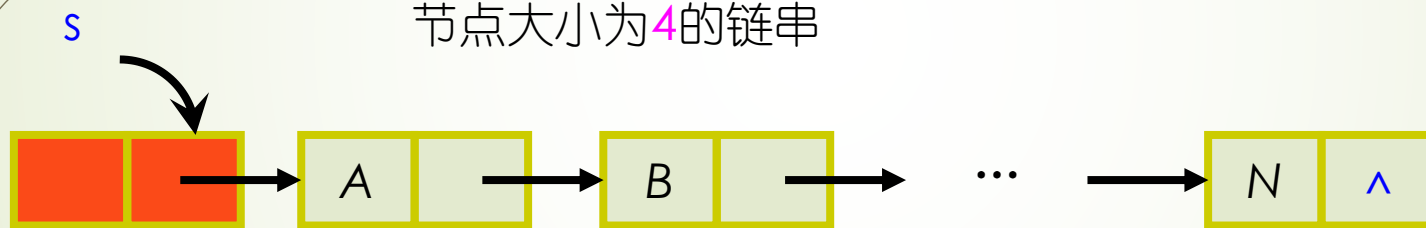
4.2.2 串的链式存储及其基本操作实现

链串的组织形式与一般的链表类似。主要的区别在于，链串中的一个节点可以存储多个字符。通常将链串中每个节点所存储的字符个数称为节点大小。

以下两图分别表示了同一个串"ABCDEFGHJKLMNOP"的节点大小为4（存储密度大）和1（存储密度小）的链式存储结构。

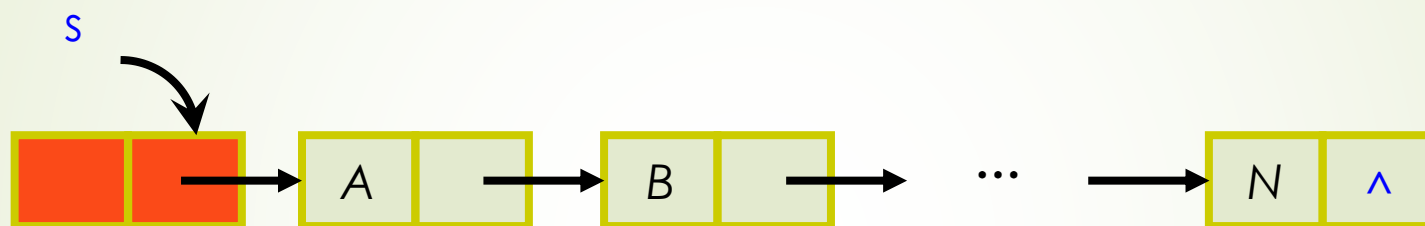


节点大小为4的链串



节点大小为1的链串

链串节点大小的选择与顺序串的格式选择类似。节点大小越大，则存储密度越大。但存储密度越大，一些操作（如插入、删除、替换等）有所不便，且可能引起大量字符移动，因此它适合于在串基本保持静态使用方式时采用。节点大小越小（如节点大小为1时），运算处理越方便，但存储密度下降。为简便起见，这里规定链串节点大小均为1。



链串的节点类型定义如下：

```
typedef struct snode
{   char data;
    struct snode *next;
} LiString;
```

下面讨论在链串上实现串基本运算的算法。

(1) StrAssign(s,cstr)

将一个字符串常量cstr赋给串s，即生成一个其值等于cstr的串s。以下采用尾插法建立链串s。

```
void StrAssign(LiString *&s, char cstr[])
{
    int i; LiString *r, *p;
    s = (LiString *) malloc(sizeof(LiString));
    r = s;          // r始终指向尾节点
    for (i = 0; cstr[i] != '\0'; i++)
    {
        p = (LiString *) malloc(sizeof(LiString));
        p->data = cstr[i];
        r->next = p; r = p;
    }
    r->next = NULL;
}
```

思路：尾插法建立单链表

(2) StrCopy(s,t)

将串t复制给串s。以下采用尾插法建立复制后的链串s。

```
void StrCopy(LiString *&s, LiString *t)
{
    LiString *p=t->next, *q, *r;
    s=(LiString *)malloc(sizeof(LiString));
    r=s;                                //r始终指向尾节点
    while (p!=NULL)                    //将t的所有节点复制到s
    {
        q=(LiString *)malloc(sizeof(LiString));
        q->data=p->data;
        r->next=q; r=q;
        p=p->next;
    }
    r->next=NULL;
}
```

思路：尾插法建立单链表

(3) StrEqual(s,t)

判串相等：若两个串s与t相等则返回真；否则返回假。

```
bool StrEqual(LiString *s, LiString *t)
{
    LiString *p=s->next, *q=t->next;
    while (p!=NULL && q!=NULL && p->data==q->data)
    {
        p=p->next;
        q=q->next;
    }
    if (p==NULL && q==NULL)
        return true;
    else
        return false;
}
```

(4) StrLength(s)

求串长：返回串s中字符个数。

```
int StrLength(LiString *s)
{   int i=0;
    LiString *p=s->next;
    while (p!=NULL)
    {   i++;
        p=p->next;
    }
    return i;
}
```


(5) Concat(s,t)

串连接：返回由两个串s和t连接在一起形成的新串。以下采用尾插法建立链串str并返回其地址。

s为"abcd", t为"123"



s1=Concat(s,t)

s1为"abcd123"

```
LiString *Concat (LiString *s, LiString *t)
{
    LiString *str, *p=s->next, *q, *r;
    str=(LiString *)malloc(sizeof(LiString));
    r=str;
    while (p!=NULL)    //将s的所有节点复制到str
    {
        q=(LiString *)malloc(sizeof(LiString));
        q->data=p->data;
        r->next=q; r=q;
        p=p->next;
    }
    p=t->next;
    while (p!=NULL)    //将t的所有节点复制到str
    {
        q=(LiString *)malloc(sizeof(LiString));
        q->data=p->data;
        r->next=q; r=q;
        p=p->next;
    }
    r->next=NULL;
    return str;
}
```

(6) SubStr(s,i,j)

求子串：返回串s中从第i ($1 \leq i \leq \text{StrLength}(s)$) 个字符开始的、由连续j个字符组成的子串，参数不正确时返回一个空串。以下采用尾插法建立链串str并返回其地址。

s为" a**bcd**123"



$s1 = \text{SubStr}(s, 2, 4)$

s1为" bcd1"

思路：尾插法建立单链表，没有破坏s单链表

```
LiString *SubStr(LiString *s,int i,int j)
{
    int k;
    LiString *str,*p=s->next,*q,*r;
    str=(LiString *)malloc(sizeof(LiString));
    str->next=NULL;
    r=str;          //r指向新建链表的尾节点
    if (i<=0 || i>StrLength(s) || j<0 || i+j-1>StrLength(s))
        return str; //参数不正确时返回空串
    for (k=0;k<i-1;k++)
        p=p->next;
    for (k=1;k<=j;k++) //将s的第i个节点开始的j个节点复制到str
    {
        q=(LiString *)malloc(sizeof(LiString));
        q->data=p->data;
        r->next=q;r=q;
        p=p->next;
    }
    r->next=NULL;
    return str;
}
```

(7) InsStr(s,i,t)

将串t插入到串s的第i ($1 \leq i \leq \text{StrLength}(s)+1$) 个字符位置，并返回产生的新串，参数不正确时返回一个空串。

s1为"abcd", s2为"123"




`s3=InsStr(s1,2,s2)`

s3为"a123bcd"

```
LiString *InsStr(LiString *s,int i,LiString *t)
{   int k;
    LiString *str,*p=s->next,*p1=t->next,*q,*r;
    str=(LiString *)malloc(sizeof(LiString));
    str->next=NULL;
    r=str;          //r指向新建链表的尾节点
    if (i<=0 || i>StrLength(s)+1)
        return str; //参数不正确时返回空串
```

思路：尾插法建立单链表，没有破坏s和t单链表



```
for (k=1;k<i;k++)    //将s的前i个节点复制到str
{
    q=(LiString *)malloc(sizeof(LiString));
    q->data=p->data;
    r->next=q;r=q;
    p=p->next;
}
while (p1!=NULL) //将t的所有节点复制到str
{
    q=(LiString *)malloc(sizeof(LiString));
    q->data=p1->data;
    r->next=q;r=q;
    p1=p1->next;
}
while (p!=NULL)    //将*p及其后的节点复制到str
{
    q=(LiString *)malloc(sizeof(LiString));
    q->data=p->data;
    r->next=q;r=q;
    p=p->next;
}
r->next=NULL;
return str;
```

```
}
```

(8) DelStr(s,i,j)

从串s中删去从第i ($1 \leq i \leq \text{StrLength}(s)$) 个字符开始的长度为j的子串, 并返回产生的新串, 参数不正确时返回一个空串。以下采用尾插法建立链串str并返回其地址。

s为" a**bcd**123"





$s1 = \text{DelStr}(s, 2, 4)$

s1为" a23"


```
LiString *DelStr(LiString *s,int i,int j)
{   int k;
    LiString *str,*p=s->next,*q,*r;
    str=(LiString *)malloc(sizeof(LiString));
    str->next=NULL;
    r=str;          //r指向新建链表的尾节点
    if (i<=0 || i>StrLength(s) || j<0 || i+j-1>StrLength(s))
        return str; //参数不正确时返回空串
```

思路：尾插法建立单链表，没有破坏s单链表



```
for (k=0;k<i-1;k++)    //将s的前i-1个节点复制到str
{   q=(LiString *)malloc(sizeof(LiString));
    q->data=p->data;
    r->next=q;r=q;
    p=p->next;
}
for (k=0;k<j;k++) //让p沿next跳j个节点
    p=p->next;
while (p!=NULL)    //将*p及其后的节点复制到str
{   q=(LiString *)malloc(sizeof(LiString));
    q->data=p->data;
    r->next=q;r=q;
    p=p->next;
}
r->next=NULL;
return str;
}
```

(9) RepStr(s,i,j,t)


在串s中，将第i ($1 \leq i \leq \text{StrLength}(s)$) 个字符开始的j个字符构成的子串用串t替换，并返回产生的新串，参数不正确时返回一个空串。以下采用尾插法建立链串str并返回其地址。

s为" a**bcde**fg", t为" 123"




`s1=RepStr(s,2,4,t)`

s1为" a123fg"



```
LiString *RepStr(LiString *s,int i,int j,LiString *t)
{   int k;
    LiString *str,*p=s->next,*p1=t->next,*q,*r;
    str=(LiString *)malloc(sizeof(LiString));
    str->next=NULL;
    r=str;                //r指向新建链表的尾节点
    if (i<=0 || i>StrLength(s)
        || j<0 || i+j-1>StrLength(s))
        return str;      //参数不正确时返回空串
```

思路：尾插法建立单链表，没有破坏s和t单链表



```
for (k=0;k<i-1;k++)          //将s的前i-1个节点复制到str
{
    q=(LiString *)malloc(sizeof(LiString));
    q->data=p->data;q->next=NULL;
    r->next=q;r=q;
    p=p->next;
}
for (k=0;k<j;k++)            //让p沿next跳j个节点
    p=p->next;
while (p1!=NULL)              //将t的所有节点复制到str
{
    q=(LiString *)malloc(sizeof(LiString));
    q->data=p1->data;q->next=NULL;
    r->next=q;r=q;
    p1=p1->next;
}
while (p!=NULL)                //将*p及其后的节点复制到str
{
    q=(LiString *)malloc(sizeof(LiString));
    q->data=p->data;q->next=NULL;
    r->next=q;r=q;
    p=p->next;
}
r->next=NULL;
return str;
}
```

(10) DispStr(s)

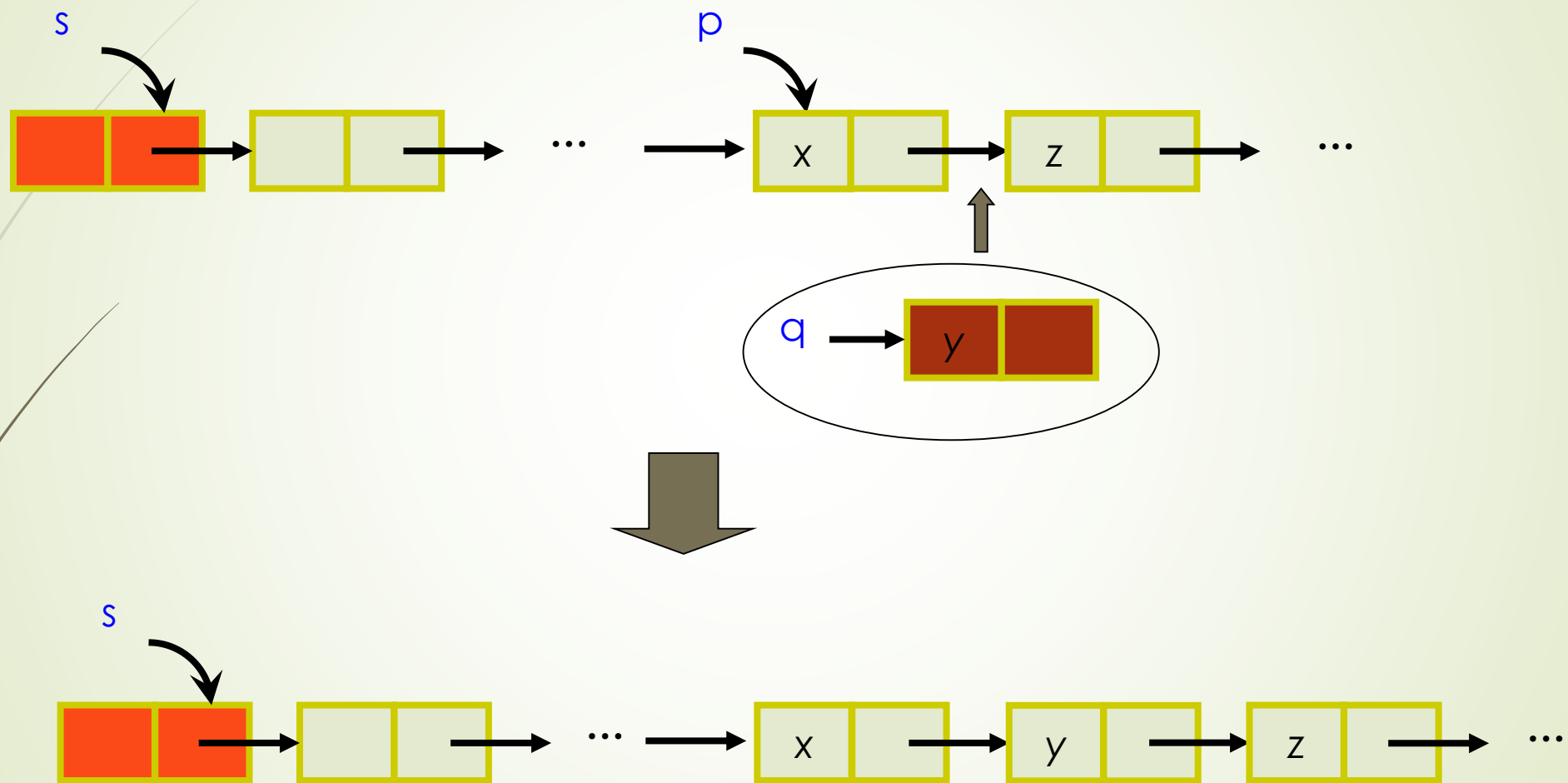
输出串s的所有元素值。



```
void DispStr(LiString *s)
{
    LiString *p=s->next;
    while (p!=NULL)
    {
        printf("%c",p->data);
        p=p->next;
    }
    printf("\n");
}
```

【例4.3:p110】在链串中,设计一个算法把最先出现的子串"ab"改为"xyz"。

解：在串s中找到最先出现的子串“ab”，p指向data域值为‘a’的结点，其后为data域值为‘b’的结点。将它们的data域值分别改为‘x’和‘z’，再创建一个data域值为‘y’的结点，将其插入到*p之后。

“ab”改为“xyz”的动画演示





```
void Repl (LiString *&s)
{   LiString *p=s->next,*q;
    int find=0;
    while (p->next!=NULL && find==0)           //查找'ab'子串
    {   if (p->data=='a' && p->next->data=='b') //找到子串
        {   p->data='x';p->next->data='z';      //替换为xyz
            q=(LiString *)malloc(sizeof(LiString));
            q->data='y';q->next=p->next;p->next=q;
            find=1;
        }
        else p=p->next;
    }
}
```



练习


设计一个程序，计算串str中每一个字符出现的次数。



4.3 串的模式匹配

关联的知识点

- 串的模式匹配的概念
- 串的简单模式匹配算法
- KMP算法及其提高串匹配效率的特点



设有主串s和子串t，子串t的定位就是要在主串s中找到一个与子串t相等的子串。通常把主串s称为目标串，把子串t称为模式串，因此定位也称作模式匹配。

模式匹配成功是指在目标串s中找到一个模式串t；不成功则指目标串s中不存在模式串t。

4.4.1 Brute-Force算法

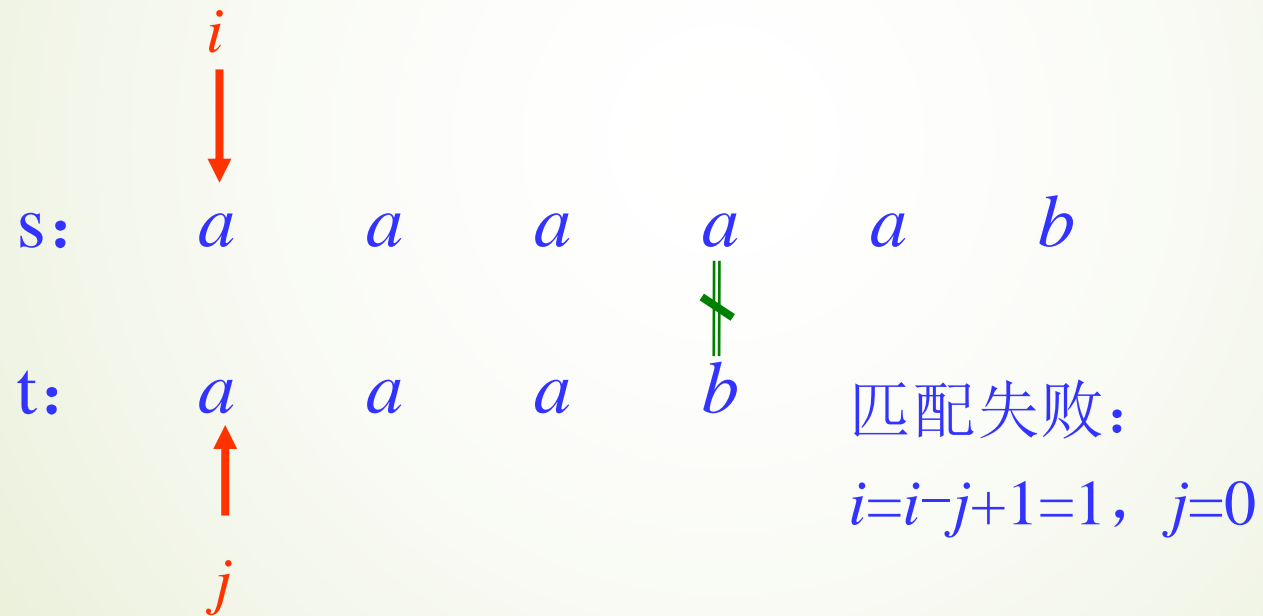
Brute-Force简称为BF算法，亦称简单匹配算法，其基本思路是：

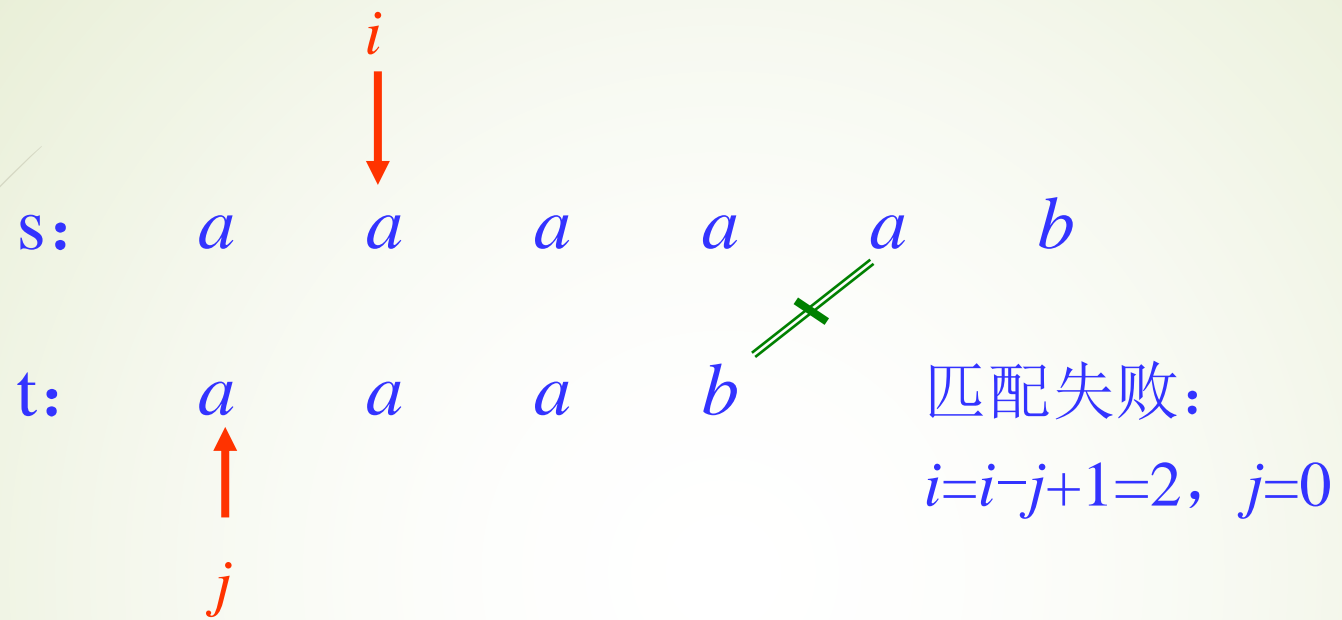
从目标串 $s = "s_0s_1 \dots s_{n-1}"$ 的第一个字符开始和模式串 $t = "t_0t_1 \dots t_{m-1}"$ 中的第一个字符比较，若相等，则继续逐个比较后续字符；否则从目标串 s 的第二个字符开始重新与模式串 t 的第一个字符进行比较。

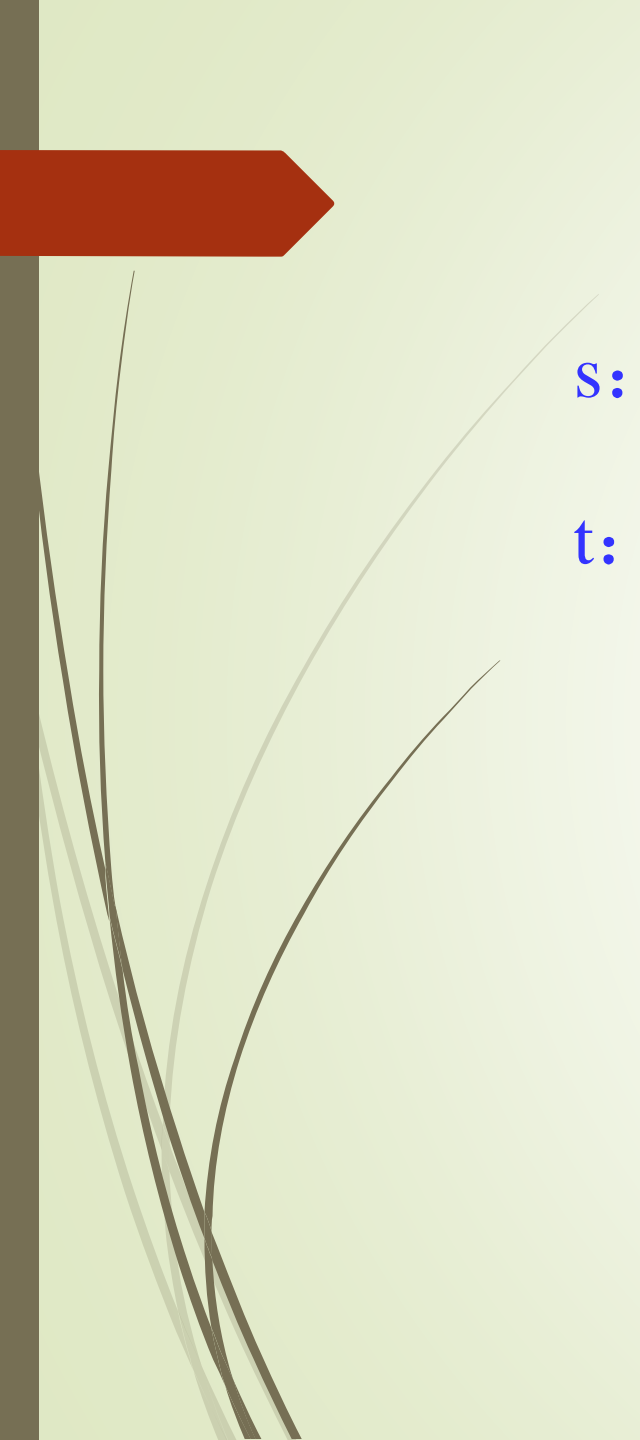
依次类推，若从目标串 s 的第 i 个字符开始，每个字符依次和模式串 t 中的对应字符相等，则匹配成功，该算法返回 i ；否则，匹配失败，函数返回-1。

简单模式匹配算法动画演示

例如，设目标串 s ="aaaaab"，模式串 t ="aaab"。 s 的长度为 n （ $n=6$ ）， t 的长度为 m （ $m=4$ ）。用指针 i 指示目标串 s 的当前比较字符位置，用指针 j 指示模式串 t 的当前比较字符位置。模式匹配过程如下。







s: *a* *a* *a* *a* *a* *b*

t: *a* *a* *a* *b*

 ↑
 j

 ↓
 i

匹配成功:

$i=6, j=4$

返回 $i-t.length=2$

BF算法设计思想

Index(S,T)

- 将主串的第1个字符和模式的第一个字符比较，
若**相等**，继续逐个比较后续字符；
若**不等**，从主串的下一字符起，重新与模式的第一个字符比较。
- 直到主串的一个连续子串字符序列与模式相等 。
返回值为S中与T匹配的子序列**第一个字符的序号**，
即匹配成功。
- 否则，匹配失败，返回值 0

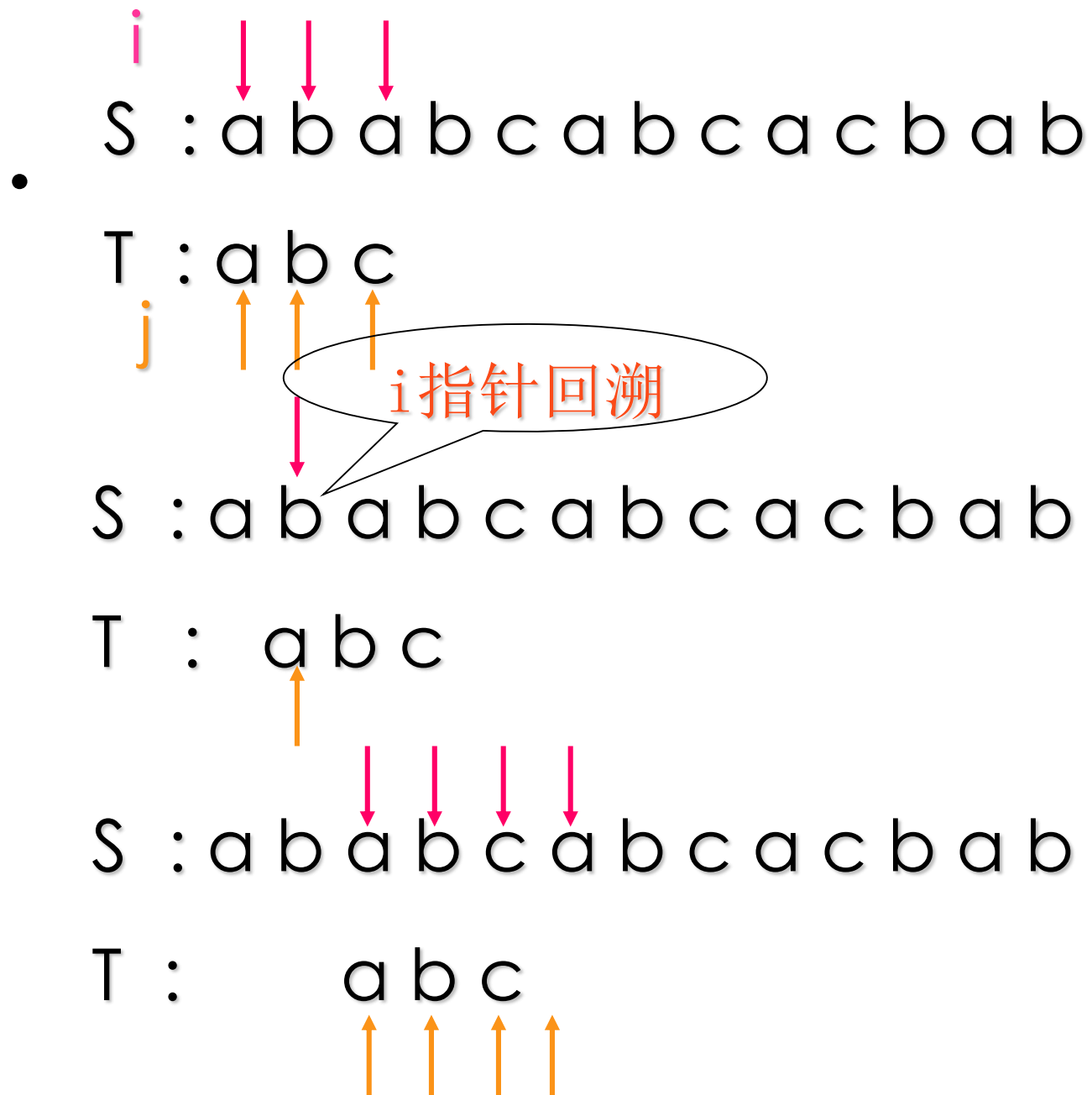
对应的BF算法如下:


```
int index(SqString s, SqString t)
{   int i=0, j=0;
    while (i<s.length && j<t.length)
    {   if (s.data[i]==t.data[j])    //继续匹配下一个字符
        {   i++;                //主串和子串依次匹配下一个字符
            j++;
        }
        else                        //主串、子串指针回溯重新开始下一次匹配
        {   i=i-j+1;              //主串从下一个位置开始匹配
            j=0;                  //子串从头开始匹配
        }
    }
    if (j>=t.length)
        return(i-t.length); //返回匹配的第一个字符的下标
    else
        return(-1);         //模式匹配不成功
}
```



将该算法进行推广，可以指定主串中查找的起始位置pos（下标）。

```
int index(SqString s, SqString t, int pos)
{   int i=pos, j=0;
    while (i<s.length && j<t.length)
    {   if (s.data[i]==t.data[j])    //继续匹配下一个字符
        {   i++;                    //主串和子串依次匹配下一个字符
            j++;
        }
        else                        //主串、子串指针回溯重新开始下一次匹配
        {   i=i-j+1;                //主串从下一个位置开始匹配
            j=0;                    //子串从头开始匹配
        }
    }
    if (j>=t.length)
        return(i-t.length); //返回匹配的第一个字符的下标
    else
        return(-1);         //模式匹配不成功
}
```





这个算法简单，易于理解，但效率不高，主要原因是主串指针 i 在若干个字符序列比较相等后，若有一个字符比较不相等，仍需回溯（即 $i=i-j+1$ ）。

该算法在最好情况下的时间复杂度为 $O(m)$ ，即主串的前 m 个字符正好等于模式串的 m 个字符。在最坏情况下的时间复杂度为 $O(n \times m)$ 。

练习

1.教材P119 4.1

2.采用顺序结构存储串，编写一个算法计算指定子串在一个字符串中出现的次数，如果该子串不出现则为0。

3.采用顺序结构存储串，编写一个算法求串s和串t的一个最长公共子串。

问：怎样才能提高串匹配的效率呢？？？



答：问Knuth吧！！！！



Donald Knuth (1938年 ~)，算法和程序设计技术的先驱者，计算机排版系统TEX和METAFONT的发明者，他因这些成就和大量创造性的影响深远的著作（19部书和160篇论文）而誉满全球。作为斯坦福大学计算机程序设计艺术的荣誉退休教授，他当前正全神贯注于完成其关于计算机科学的史诗性的七卷集。这一伟大工程在1962年他还是加利福尼亚理工学院的研究生时就开始了。他于1974年获得计算机科学界最高奖——**图灵奖**。

4.3.2 KMP算法

KMP算法是D.E.Knuth、J.H.Morris和V.R.Pratt共同提出的，简称KMP算法。该算法较BF算法有较大改进，主要是消除了主串指针的回溯，从而使算法效率有了某种程度的提高。

KMP算法用next数组保存部分匹配信息的动画演示

目标串s="aaaaab", 模式串t="aaab".

0	1	2	3	4	5
a	a	a	a	a	b
↑	↑	↑			
a	a	a	b		
0	1	2	3		

BF算法下一步是从s[1]开始

其实没有必要从s[1]开始, 为什么?

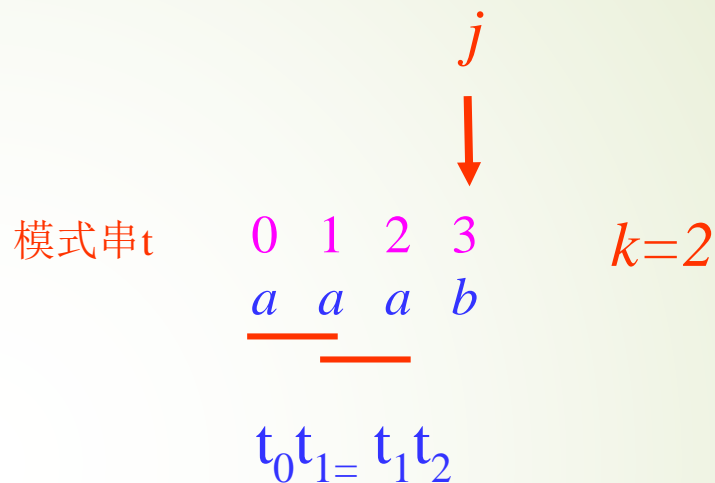
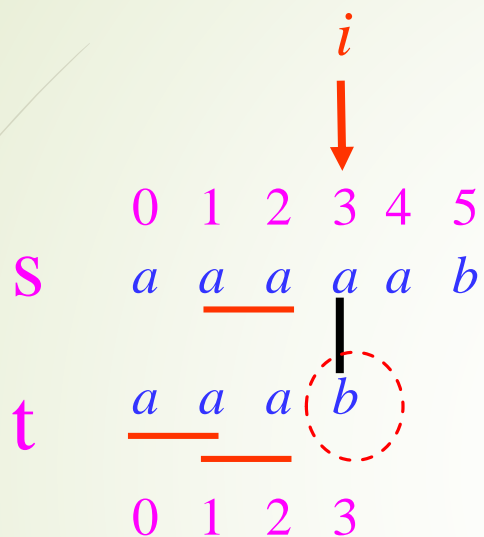
0	1	2	3	4	5
a	a	a	a	a	b
a	a	a	b		
0	1	2	3		



0	1	2	3	4	5
a	a	a	a	a	b
a	a	a	b		
0	1	2	3		

应有一个数组next,
使next[3]=2

模式串中究竟是什么信息呢？



对于模式串 t 中的每个字符 t_j ，存在某个整数 k ($0 < k < j$)，使得模式 t 中 k 所指字符之前的 k 个字符“ $t_0 t_1 \dots t_{k-1}$ ”依次与 t_j 的前面 k 个字符“ $t_{j-k} t_{j-k+1} \dots t_{j-1}$ ”相同，并且与主串 s 中 i 所指字符之前的 k 个字符相同。那么就可以利用这种信息，避免不必要的回溯。

不同于串基本概念
中的真子串

所谓真子串是指模式串 t 存在某个 k ($0 < k < j$)，使得
" $t_0 t_1 \dots t_k$ " = " $t_{j-k} t_{j-k+1} \dots t_j$ "成立。

例如， $t = "abab"$,

即 $t_0 t_1 = t_2 t_3$

也就是说，“ ab ”是真子串。

真子串就是模式串中隐藏的信息，利用它来提高模式匹配的效率。

模式串 $t = \text{"abcac"}$ ，用next数组存放这些“部分匹配”信息。

$\text{next}[j]$ 是指下标为 j 的字符前有多少个真子串的字符。

j	0	1	2	3	4
$t[j]$	a	b	c	a	c
$\text{next}[j]$	-1	0	0	0	1

归纳起来，定义next[j]函数如下：

$$\text{next}[j] = \begin{cases} \max\{k \mid 0 < k < j, \text{且 } "t_0t_1 \dots t_{k-1}" = "t_{j-k}t_{j-k+1} \dots t_{j-1}"\} & \text{当此集合非空时} \\ -1 & \text{当 } j=0 \text{ 时} \\ 0 & \text{其他情况} \end{cases}$$

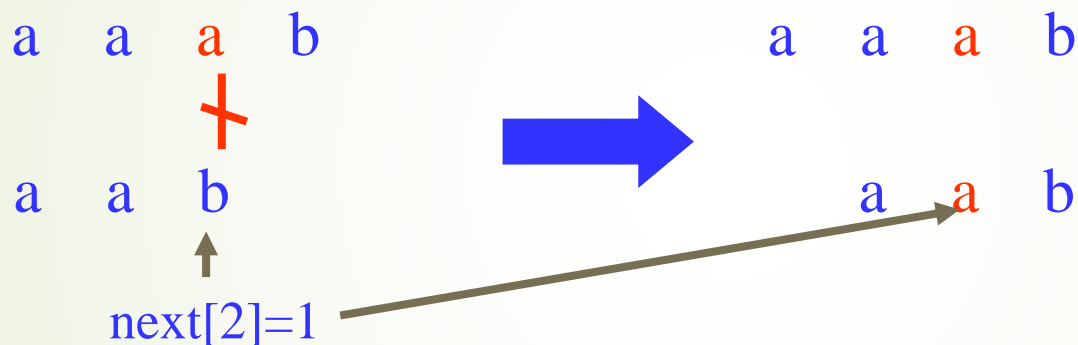
t="abab"对应的next数组如下：

<i>j</i>	0	1	2	3
t[j]	a	b	a	b
next[j]	-1	0	0	1

两个重要的问题

(1) $\text{next}[j]=k$ 表示什么信息？

说明模式串 $t[j]$ 之前有 k 个字符已成功匹配，下一趟应从 $t[k]$ 开始匹配，这就是KMP算法加速匹配的原因。



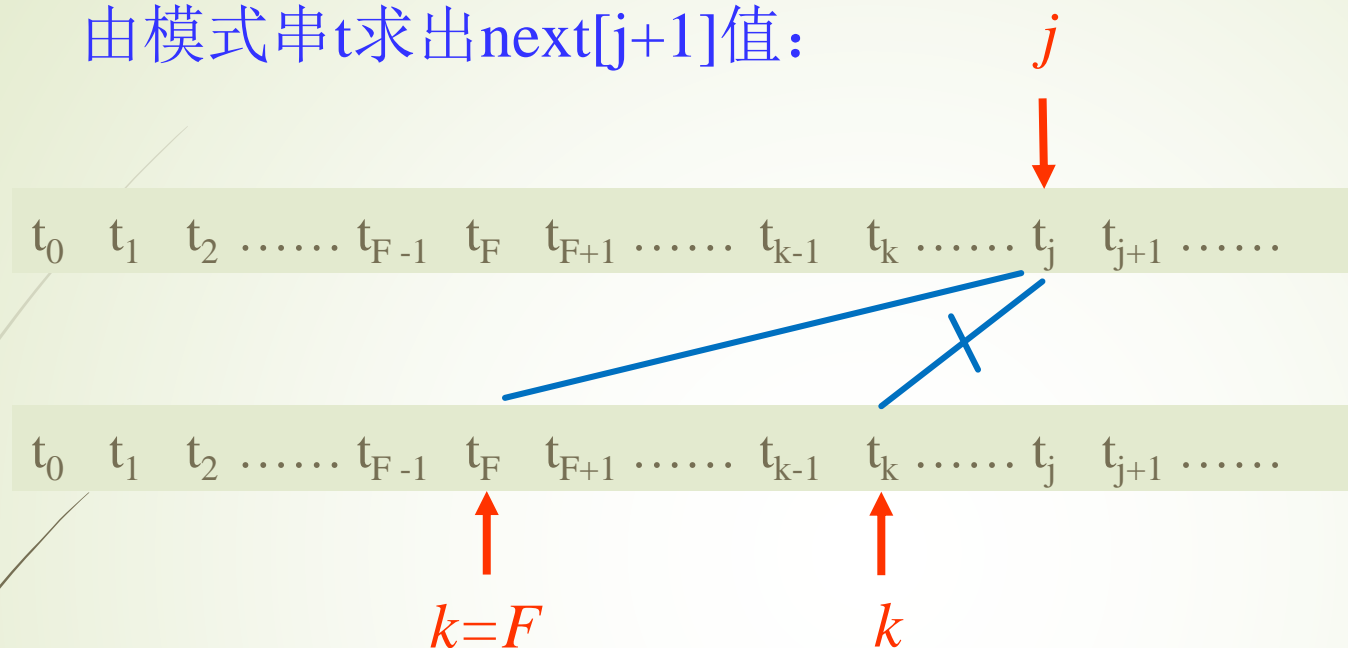
(2) $\text{next}[j]=-1$ 表示什么信息？

说明模式串 $t[j]$ 之前没有任何用于加速匹配的信息， j 退回到-1，此时 i, j 各加1.即从 s_{i+1} 和 t_0 开始比较。

由模式串t求出next值:

```
void GetNext(SqString t,int next[])
{   int j,k;
    j=0;k=-1;next[0]=-1;
    while (j<t.length-1)
    {   if (k==-1 || t.data[j]==t.data[k])
        {   j++;k++;
            next[j]=k;
        }
        else k=next[k];
    }
}
```

由模式串t求出next[j+1]值:



此时:

$$t_0 t_1 \dots t_{k-1} = t_{j-k} \dots t_{j-2} t_{j-1}$$

假如 $k = \text{next}[k] = F$ $t_0 t_1 \dots t_{F-1} = t_{k-F} \dots t_{k-2} t_{k-1}$

当 $t_F == t_j$ 时:

$$t_0 t_1 \dots t_{F-1} t_F = t_{j-F} \dots t_{j-1} t_j$$

$j = j+1, k = F+1, \text{next}[j+1] = F+1$

KMP算法:

```
int KMPIndex(SqString s, SqString t)
{
    int next[MaxSize], i=0, j=0;
    GetNext(t, next);
    while (i<s.length && j<t.length)
    {
        if (j==-1 || s.data[i]==t.data[j])
        {
            i++;
            j++;          //i, j各增1
        }
        else j=next[j];   //i不变, j后退
    }
    if (j>=t.length)
        return(i-t.length); //返回匹配模式串的首字符下标
    else
        return(-1);        //返回不匹配标志
}
```

KMP算法的扩展:

```
int KMPIndex(SqString s, SqString t, int pos )
{
    int next[MaxSize], i=pos, j=0;
    GetNext(t, next);
    while (i<s.length && j<t.length)
    {
        if (j==-1 || s.data[i]==t.data[j])
        {
            i++;
            j++;          //i, j各增1
        }
        else j=next[j];   //i不变, j后退
    }
    if (j>=t.length)
        return(i-t.length); //返回匹配模式串的首字符下标
    else
        return(-1);        //返回不匹配标志
}
```

【P115例4.5】 设主串s=“ababcabcacbab”，模式串t=“abcac”。给出KMP算法进行模式匹配的过程。

解：模式串对应的next数组如表所示

j	0	1	2	3	4
t[j]	a	b	c	a	c
next[j]	-1	0	0	0	1

s: a b a b c a b c a c b a b
 ↓
t: a b c a c
 ↑

匹配失败：
 $i=2$ （不变）
 $j=\text{next}[2]=0$

【P115例4.5】 设主串s=“ababcabcacbab”，模式串t=“abcac”。给出KMP算法进行模式匹配的过程。

解：模式串对应的next数组如表所示

j	0	1	2	3	4
t[j]	a	b	c	a	c
next[j]	-1	0	0	0	1

s: a b a b c a b c a c b a b
 ↓
 i
t: a b c a c
 ↑ ↑
 j j

匹配失败：
 $i=6$ （不变）
 $j=\text{next}[4]=1$

【P115例4.5】 设主串s=“ababcabcacbab”，模式串t=“abcac”。给出KMP算法进行模式匹配的过程。

解：模式串对应的next数组如表所示

j	0	1	2	3	4
t[j]	a	b	c	a	c
next[j]	-1	0	0	0	1

s: a b a b c a b c a c b a b


t: a b c a c

i



j

匹配成功：
返回i-t.length=5



设主串 s 的长度为 n ，子串 t 长度为 m 。

在KMP算法中求next数组的时间复杂度为 $O(m)$ ，在后面的匹配中因主串 s 的下标不减即不回溯，比较次数可记为 n ，所以KMP算法总的时间复杂度为 $O(n+m)$ 。

j	0	1	2	3	4
t[j]	a	a	a	a	b
next[j]	-1	0	1	2	3

第 1 次匹配	s=aaabaaaab t=aaaab	i=3 j=3, j=next[3]=2	失败
第 2 次匹配	s=aaabaaaa t=aaaab	i=3 j=2, j=next[2]=1	失败
第 3 次匹配	s=aaabaaaab t=aaaab	i=3 j=1, j=next[1]=0	失败
第 4 次匹配	s=aaabaaaab t=aaaab	i=3 j=0, j=next[0]=-1	失败
第 5 次匹配	s=aaabaaaab t=aaaab	i=9 j=5, 返回 9-5=4	成功

上述定义的next[]在某些情况下尚有缺陷。

例如，模式“aaaab”在和主串“aaabaaaab”匹配时：

当 $i=3/j=3$ 时， $s.data[3] \neq t.data[3]$ ，由 $next[j]$ 的指示还需进行 $i=3/j=2$ ， $i=3/j=1$ ， $i=3/j=0$ 等3次比较。

实际上，因为模式中的第1、2、3个字符和第4个字符都相等，因此不需要再和主串中第4个字符相比较，而可以将模式一次向右滑动4个字符的位置直接进行 $i=4$ ， $j=0$ 时的字符比较。

这就是说，若按上述定义得到 $\text{next}[j]=k$ ，而模式中 $t_j=t_k$ ，则为主串中字符 s_i 和 t_j 比较不等时，不需要再和 t_k 进行比较，而直接和 $t_{\text{next}[k]}$ 进行比较。换句话说，此时的 $\text{next}[j]$ 应和 $\text{next}[k]$ 相同。

为此将 $\text{next}[j]$ 修正为 $\text{nextval}[j]$ ：

比较 $t.\text{data}[j]$ 和 $t.\text{data}[k]$ ，若不等，则 $\text{nextval}[j]=\text{next}[j]$ ；若相等 $\text{nextval}[j]=\text{nextval}[k]$ 。

由模式串t求出nextval值:

```
void GetNextval(SqString t,int nextval[])
{   int j=0,k=-1;
    nextval[0]=-1;
    while (j<t.length)
    {   if (k==-1 || t.data[j]==t.data[k])
        {   j++;k++;
            if (t.data[j]!=t.data[k])
                nextval[j]=k;
            else
                nextval[j]=nextval[k];
        }
        else
            k=nextval[k];
    }
}
```

修改后的KMP算法:

```
int KMPIndex1(SqString s, SqString t)
{
    int nextval[MaxSize], i=0, j=0;
    GetNextval(t, nextval);
    while (i<s.length && j<t.length)
    {
        if (j==-1 || s.data[i]==t.data[j])
        {
            i++;
            j++;
        }
        else
            j=nextval[j];
    }
    if (j>=t.length)
        return(i-t.length);
    else
        return(-1);
}
```

修改后的KMP算法扩展:

```
int KMPIndex1(SqString s, SqString t, int pos)
{
    int nextval[MaxSize], i=pos, j=0;
    GetNextval(t, nextval);
    while (i<s.length && j<t.length)
    {
        if (j==-1 || s.data[i]==t.data[j])
        {
            i++;
            j++;
        }
        else
            j=nextval[j];
    }
    if (j>=t.length)
        return(i-t.length);
    else
        return(-1);
}
```

<i>j</i>	0	1	2	3	4
<i>t[j]</i>	<i>a</i>	<i>a</i>	<i>a</i>	<i>a</i>	<i>b</i>
next[j]	-1	0	1	2	3
nextval[j]	-1	-1	-1	-1	3

第 1 次匹配

s=aaabaaaab

i=3

t=aaaab

j=3, j=nextval[3]=-1

失败

第 2 次匹配

s=aaabaaaab

i=9

t=aaaab

j=5, 返回 9-5=4

成功

【P118例4.6】 设主串s=“abcaabbabcbacba”，模式串t=“abcabaa”。计算模式串t的nextval函数值，并给出KMP算法进行模式匹配的过程。

解： 模式串对应的next数组如表所示

j	0	1	2	3	4	5	6
t[j]	a	b	c	a	b	a	a
next[j]	-1	0	0	0	1	2	1
Nextval[j]	-1	0	0	-1	0	2	1

s: a b c a a b b a b c a b a a c b a c b a
 ↓
t: a b c a b a a
 ↑

匹配失败：i不变为4
j=nextval[4]=0

【P118例4.6】 设主串s=“abcaabbabcbacba”，模式串t=“abcabaa”。计算模式串t的nextval函数值，并给出KMP算法进行模式匹配的过程。

解： 模式串对应的next数组如表所示

j	0	1	2	3	4	5	6
t[j]	a	b	c	a	b	a	a
next[j]	-1	0	0	0	1	2	1
Nextval[j]	-1	0	0	-1	0	2	1

s: a b c a a b b a b c a b a a c b a c b a

t: a b c a b a a

 ↑ ↓

 j i

匹配失败：i不变为6
j=nextval[2]=0

【P118例4.6】 设主串s=“abcaabbabcbacba”，模式串t=“abcabaa”。计算模式串t的nextval函数值，并给出KMP算法进行模式匹配的过程。

解： 模式串对应的next数组如表所示

j	0	1	2	3	4	5	6
t[j]	a	b	c	a	b	a	a
next[j]	-1	0	0	0	1	2	1
Nextval[j]	-1	0	0	-1	0	2	1

s: a b c a a b b a b c a b a a c b a c b a

 ↑ ↓ ↓

 i i

t: a b c a b a a

 ↑

 j

匹配失败: $j = \text{nextval}[0] = -1$

$i++$ 变为7

$j = j + 1 = 0$

【P118例4.6】 设主串s=“abcaabbabcbacba”，模式串t=“abcabaa”。计算模式串t的nextval函数值，并给出KMP算法进行模式匹配的过程。

解： 模式串对应的next数组如表所示

j	0	1	2	3	4	5	6
t[j]	a	b	c	a	b	a	a
next[j]	-1	0	0	0	1	2	1
Nextval[j]	-1	0	0	-1	0	2	1

s: a b c a a b b a b c a b a a c b a c b a

i
↓

t: a b c a b a a

↑
j

匹配成功： 返回i-t.length=7

说明：

虽然BF算法的时间复杂度是 $O(n \times m)$ ，但在一般情况下，其实际的执行时间近似于 $O(n+m)$ ，因此至今仍被采用。KMP算法仅当模式与主串之间存在许多部分匹配的情况下，才显得比BF算法快得多。但是KMP算法最大特点是主串的指针不需回溯，整个匹配过程中，对主串仅需从头至尾扫描一遍，这对处理从外设输入的庞大文件很有效，可以边读入边匹配而无须回头重读。



思考题：

KMP算法给我们什么启示？

练习

设主串s=“abcaabbcaababababca”，模式串t=“babab”。计算模式串t的nextval函数值，并给出KMP算法进行模式匹配的过程。

案例分析与实现



病毒感染检测

【案例分析】

- 因为患者的DNA和病毒DNA都是由一些字母组成的字符串序列，要检测某种病毒DNA序列是否在患者的DNA序列中出现过，实际上就是字符串的模式匹配问题。
- 可以利用BF算法，也可以利用更高效的KMP算法。
- 但与一般的模式匹配问题不同的是，此案例中病毒的DNA序列是环状的。
- 这样需要对传统的BF算法或KMP算法进行改进。

【案例实现】

- 对于每一个待检测的任务，假设病毒DNA序列的长度是 m ，因为病毒DNA序列是环状的，为了线性取到每个可行的长度为 m 的模式串，可将存储病毒DNA序列的字符串长度扩大为 $2m$ ，将病毒DNA序列连续存储两次。
- 然后循环 m 次，依次取得每个长度为 m 的环状字符串，将此字符串作为模式串，将人的DNA序列作为主串，调用BF算法进行模式匹配。
- 只要匹配成功，即可中止循环，表明该人感染了对应的病毒；否则，循环 m 次结束循环时，可通过BF算法的返回值判断该人是否感染了对应的病毒。

【算法步骤】

- ① 从文件中读取待检测的任务数 num 。
- ② 根据 num 个数依次检测每对病毒DNA和人的DNA是否匹配，循环 num 次，执行以下操作：
 - 从文件中分别读取一对病毒DNA序列和人的DNA序列；
 - 设置标志性变量 $flag$ ，用来标识是否匹配成功，初始为0，表示未匹配；
 - 病毒DNA序列的长度是 m ，将存储病毒DNA序列的字符串长度扩大为 $2m$ ，将病毒DNA序列连续存储两次；
 - 循环 m 次，重复执行以下操作：
 - 依次取得每个长度为 m 的病毒DNA环状字符串；
 - 将此字符串作为模式串，将人的DNA序列作为主串，调用BF算法进行模式匹配，将匹配结果返回赋值给 $flag$ ；
 - 若 $flag$ 非0，表示匹配成功，中止循环，表明该人感染了对应的病毒。
 - 退出循环时，判断 $flag$ 的值，若 $flag$ 非0，输出“YES”，否则，输出“NO”。

本章小结

本章基本学习要点如下：

- (1) 理解串和一般线性表之间的差异。
- (2) 重点掌握在顺序串上和链串上实现串的基本运算算法。
- (3) 掌握串的模式匹配算法。
- (4) 灵活运用串这种数据结构解决一些综合应用问题。



—本章完—