第四章 串

线性表——具有相同类型的数据元素的有限序列。

限制插入、删除位置

栈——仅在表的一端进行插入和删除操作
队列——在一端进行插入操作，而另一端进行删除操作

限制数据元素的类型

串——数据元素为字符的有限序列。

4.1 串类型的定义

串(字符串)：是零个或多个字符组成的有限序列。记作：　*S*="$a_1a_2a_3$…"，其中*S*是串名，$a_i(1\leq i\leq n)$是单个，可以是字母、数字或其它字符。

串值：双引号括起来的字符序列是串值。

串长：串中所包含的字符个数称为该串的长度。

空串(空的字符串)：长度为零的串称为空串，它不包含任何字符。

空格串(空白串)：构成串的所有字符都是空格的串称为空白串。

注意：空串和空白串的不同，例如"　"和""分别表示长度为1的空白串和长度为0的空串。

子串(substring)：串中任意个连续字符组成的子序列称为该串的子串，包含子串的串相应地称为主串。

子串的序号：将子串在主串中首次出现时的该子串的首字符对应在主串中的序号，称为子串在主串中的序号（或位置）。

例如，设有串A和B分别是：

A="这是字符串"，B="是"

则B是A的子串，A为主串。B在A中出现了两次，其中首次出现所对应的主串位置是3。因此，称B在A中的序号为3 。

空串是任意串的子串，任意串是其自身的子串。

串相等：如果两个串的串值相等(相同)，称这两个串相等。换言之，只有当两个串的长度相等，且各个对应位置的字符都相同时才相等。

通常在程序中使用的串可分为两种：串变量和串常量。

串常量和整常数、实常数一样，在程序中只能被引用但不能不能改变其值，即只能读不能写。通常串常量是由直接量来表示的，例如语句错误("溢出")中"溢出"是直接量。

串变量和其它类型的变量一样，其值是可以改变。

a= "Welcome to Beijing"
b= "Welcome"
c= "Bei"
d= "welcometo"

两个串相等：两个串的长度相等，并且各个对应的字符也都相同。

例如，有下列四个串a，b，c，d：
a= "program"
b= "Program"
c= "pro"
d= "programer "

串的基本操作：

（1） 创建串 StringAssign (s,string_constant)

（2）判断串是否为空 StringEmpty(s)

（3）计算串长度 StrLength(s)

（4）串连接 Concat(s1,s2)

（5）求子串 SubString(s1,s2,start,len)

（6）子串的定位 Index(s1,s2)

（7）子串的插入和删除

串的抽象数据类型定义

ADT String{

数据对象：

D = { $a_i|a_i \in$ CharacterSet, i=1,2,…,n, $n \geq 0$ }

数据关系：R = {< a_{i-1}, a_i >| a_{i-1}, $a_i \in D$, i=2,3,…,n }

基本操作：

StrAssign(t , chars)

初始条件： chars是一个字符串常量。

操作结果：生成一个值为chars的串t 。

StrConcat(s, t)

初始条件：串s, t 已存在。

操作结果：将串t联结到串s后形成新串存放到s中。

StrLength(t)

初始条件：字符串t已存在。

操作结果：返回串t中的元素个数，称为串长。

SubString (s, pos, len, sub)

初始条件：串s, 已存在, 1≦pos≦StrLength(s)且

0≦len≦StrLength(s) –pos+1。

操作结果：用sub返回串s的第pos个字符起长度为len的子串。

……

} ADT String

4.2 串的存储表示和实现

串是一种特殊的线性表，其存储表示和线性表类似，但又不完全相同。串的存储方式取决于将要对串所进行的操作。串在计算机中有3种表示方式：

- 定长顺序存储表示：将串定义成字符数组，利用串名可以直接访问串值。用这种表示方式，串的存储空间在编译时确定，其大小不能改变。
- 堆分配存储方式：仍然用一组地址连续的存储单元来依次存储串中的字符序列，但串的存储空间是在程序运行时根据串的实际长度动态分配的。
- 块链存储方式：是一种链式存储结构表示。

4.2.1 串的定长顺序存储表示

串的顺序存储结构，是用一组连续的存储单元来存放串中的字符序列。定长顺序存储结构，是直接使用固定长度的字符数组来存储字符序列。

定长顺序存储结构定义为：

```
#define MAX_STRLEN  256
typedef  struct
  {  char  str[MAX_STRLEN] ;
     int  length;
  } StringType ;
```


串的联接操作

```
Status  StrConcat ( StringType  s, StringType t)
  /*   将串t联接到串s之后，结果仍然保存在s中   */
  {  int i,  j ;
     if ((s.length+t.length)>MAX_STRLEN)
        Return ERROR ;  /* 联接后长度超出范围 */
      for (i=0 ; i<t.length ; i++)
        s.str[s.length+i]=t.str[i] ;  /* 串t联接到串s之后 */
     s.length=s.length+t.length ; /* 修改联接后的串长度 */
     return OK ;
  }
```

求子串操作

输入：

输出：

初始条件：

操作结果：

求子串操作

```
Status SubString (StringType s, int pos, int len, StringType *sub)
  { int k,  j ;
     if (pos<1||pos>s.length||len<0||len>(s.length-pos+1)) return ERROR ;  /* 参数非法 */
     sub->length=len-pos+1 ;  /* 求得子串长度 */
     for (j=0, k=pos ; k<=len ; k++, j++)
        sub->str[j]=s.str[k] ;  /* 逐个字符复制求得子串 */
     return OK ;
  }
```


4.2.2 串的堆分配存储表示

实现方法：系统提供一个空间足够大且地址连续的存储空间（称为“堆”）供串使用。可使用C语言的动态存储分配函数malloc()和free()来管理。

特点是：仍然以一组地址连续的存储空间来存储字符串值，但其所需的存储空间是在程序执行过程中动态分配，故是动态的，变长的。

串的堆式存储结构的类型定义

```
typedef  struct
  { char *ch;    /* 若非空，按长度分配，否则为NULL */
     int length;      /* 串的长度 */
  } HString ;
```


串的联接操作

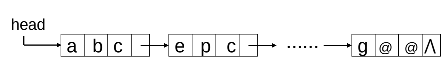
```
Status  Hstring  *StrConcat(HString  *T, HString *s1, HString *s2)
   /* 用T返回由s1和s2联接而成的串 */
   {  int k,  j , t_len ;
      t_len=s1->length+s2->length ;
      if ((p=(char *)malloc(sizeof((char)*t_len))==NULL)
         {  printf(“系统空间不够，申请空间失败！ \n”) ;
             return ERROR ;    }
      for (j=0 ; j<s->length; j++)
         T->ch[j]=s1->ch[j] ;   /* 将串s复制到串T中 */
      for (k=s1->length, j=0 ; j<s2->length; k++, j++)
         T->ch[j]=s1->ch[j] ;   /* 将串s2复制到串T中 */
      return OK ;
   }
```


4.2.3 串的链式存储表示

串的链式存储结构和线性表的串的链式存储结构类似，采用单链表来存储串，结点的构成是：

- **data**域：存放字符，**data**域可存放的字符个数称为结点的大小；
- **next**域：存放指向下一结点的指针。

若每个结点仅存放一个字符，则结点的指针域就非常多，造成系统空间浪费，为节省存储空间，考虑串结构的特殊性，使每个结点存放若干个字符，这种结构称为块链结构。



串的块链式存储结构示意图

串的块链式存储的类型定义包括：

(1) 块结点的类型定义

```
#define BLOCK_SIZE  4
typedef  struct  Blstrtype
   {  char  data[BLOCK_SIZE] ;
       struct  Blstrtype  *next;
   }BNODE ;
```

(2) 块链串的类型定义

```
typedef  struct
 { BNODE  head;    /* 头指针 */
   int  Strlen ;      /* 当前长度 */
 } Blstring ;
```

在这种存储结构下，结点的分配总是完整的结点为单位，因此，为使一个串能存放在整数个结点中，在串的末尾填上不属于串值的特殊字符，以表示串的终结。

当一个块(结点)内存放多个字符时，往往会使操作过程变得较为复杂，如在串中插入或删除字符操作时通常需要在块间移动字符。

4.3 串的模式匹配算法

子串定位运算又称为模式匹配(Pattern Matching)或串匹配(String Matching)，此运算的应用非常广泛。例如，在文本编辑程序中，我们经常要查找某一特定单词在文本中出现的位置。显然，解此问题的有效算法能极大地提高文本编辑程序的响应性能。

在串匹配中，一般将主串称为目标串，子串称之为模式串。设*S*为目标串，*P*为模式串，且不妨设：

S= “$s_1s_2\cdots s_n$” *P*= “$t_1\cdots t_m$” ，$n-m+1$个子串

4.3.1 朴素的模式匹配算法

朴素模式匹配算法的基本思想：从主串*S*的第一个字符开始和模式*P* 的第一个字符进行比较，若相等，则继续比较两者的后续字符；否则，从主串*S*的第二个字符开始和模式*P* 的第一个字符进行比较，重复上述过程，直到*P* 中的字符全部比较完毕，则说明本趟匹配成功；或*S*中字符全部比较完，则说明匹配失败。

设*S*为目标串，*P*为模式串，且不妨设：

S=“$s_1s_2\ldots s_n$”， *P*=“$t_1t_2\ldots t_m$”

串的匹配实际上是对合法的位置$1\leq i\leq n-m+1$依次将目标串中的子串模式串进行比较：

- 若$s_{i\ldots i+m-1}=p_{1\ldots m}$：则称从位置*i*开始的匹配成功，亦称模式*t*在目标*s*中出现；
- 若$s_{i\ldots i+m-1}\neq p_{1\ldots m}$：从*i*开始的匹配失败。

No.1 a b a b c a b c a c b a b
a b c a c 第一轮比较，初始： i=1,j=1当 i=3,j=3 时失败

No.2 a b a b c a b c a c b a b
a b c a c 第二轮比较，初始： i=2,j=1当 i=2,j=1 时失败

No.3 a b a b c a b c a c b a b
a b c a c 第三轮比较，初始： i=3,j=1当 i=7,j=5 时失败

No.4 a b a b c a b c a c b a b
a b c a c 第四轮比较，初始： i=4,j=1当i=4,j=1 时失败

No.5 a b a b c a b c a c b a b
a b c a c 第五轮比较，初始： i=5,j=1当i=5,j=1 时失败

No.6 a b a b c a b c a c b a b
a b c a c 第六轮比较，初始： i=6,j=1 成功

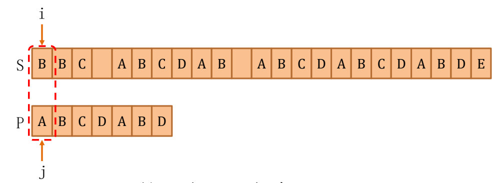

```
int NaiveStrMatching(StringType s, StringType p)
{  int i=0,j=0;             // 目标串和模式串的下标变量
   int n=s.length;
   int m=p.length;
   while (i<n)&&(j<m)
   {  if ( s.str[i]==p.str[j])   //如果相同，则两者++，继续比较
         {  i++; j++;   }
      else                     //否则，指针回溯，重新开始匹配
         { i=i-j+1; j=0;  }
    }
    if (j>=m) return i-j;
     else  return –1;
 }
```

- 假定目标S的长度为n，模式P长度为m，且 $m \le n$
  - 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S逐个字符比较的时间，最坏情况下，共m次

算法的时间复杂度： $O(n \times m)$

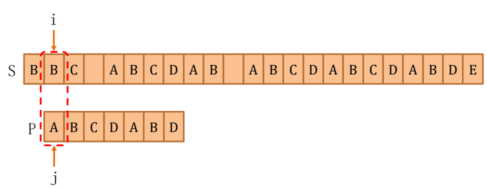
示 例 演 示

- 给定文本串S= “BBC ABCDAB ABCDABCDABDE”和模式串P =“ABCDABD”

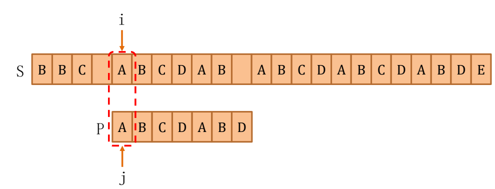


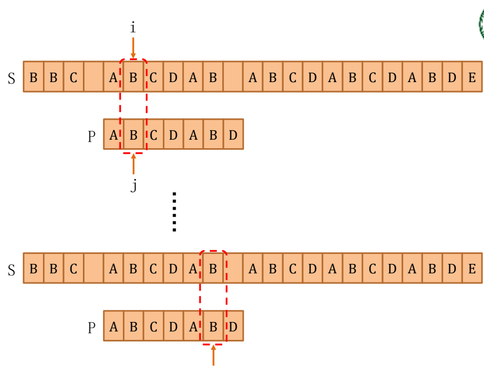
第一次匹配失败

- 开始第二次匹配

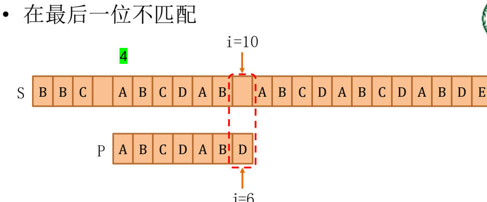


- 直到开始第五次匹配

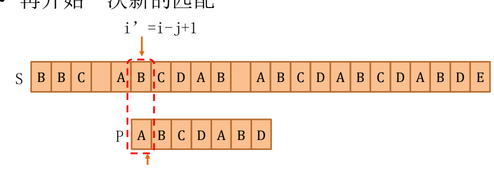


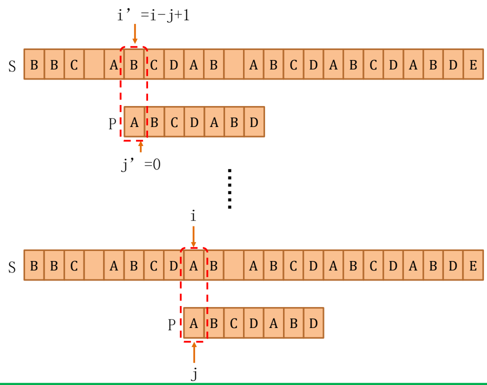


- 在最后一位不匹配

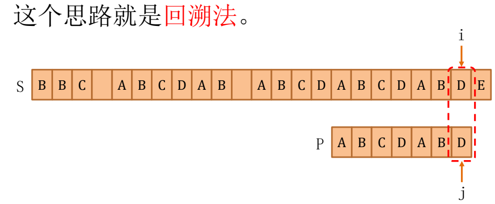


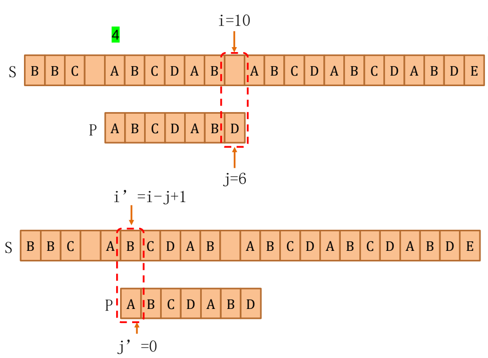
- 再开始一次新的匹配

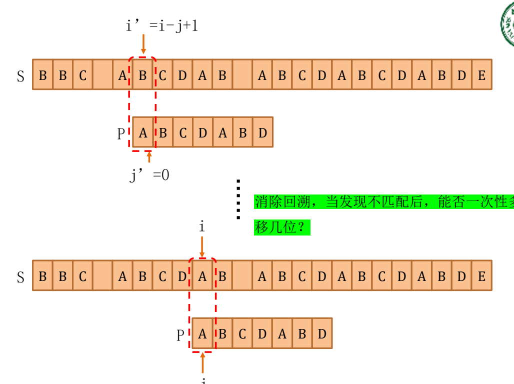




- 逐个匹配，一旦发现不匹配，模式串向后移一位，重新逐个匹配。
- 这个思路就是回溯法。







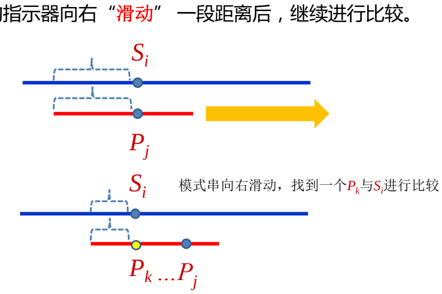
4.3.2 模式匹配的一种改进算法

BF算法：当匹配失败，为了进行下一次的匹配，主串指示器需要进行回溯到*k*-*j*+1的位置，而模式串也要退回到第一个字符（即*j*=0的位置），存在大量的主串指示器的回溯现象。

– 出发点：利用前面匹配的结果，进行无回溯匹配

每当一趟匹配过程出现字符不相等时，主串指示器不用回溯，而是利用已经得到的“部分匹配”结果，将模式串的指示器向右“滑动”一段距离后，继续进行比较。

每当一趟匹配过程出现字符不相等时，主串指示器不用回溯，而是利用已经得到的“部分匹配”结果，将模式串的指示器向右“滑动”一段距离后，继续进行比较。



S B B C   A B C D A B   A B C D A B C D A B D E
P A B C D A B D

i=1,j=1,开始比较，失败，模式串第一个元素比较就失败。

S B B C   A B C D A B   A B C D A B C D A B D E
P   A B C D A B D

i=2,j=1,开始比较，失败，模式串第一个元素比较就失败。

S B B C   A B C D A B   A B C D A B C D A B D E
P   A B C D A B D

i=3,j=1,开始比较，失败，模式串第一个元素比较就失败。

S B B C   A B C D A B   A B C D A B C D A B D E
P   A B C D A B D

i=4,j=1,开始比较，失败，模式串第一个元素比较就失败。

S B B C   A B C D A B   A B C D A B C D A B D E
P A B C D A B D

i=5,j=1,开始新的比较，当j=7时失败。

S B B C   A B C D A B   A B C D A B C D A B D E
P A B C D A B D

S B B C   A B C D A B   A B C D A B C D A B D E
P A B C D A B D

S B B C   A B C D A B   A B C D A B C D A B D E
P A B C D A B D

S B B C   A B C D A B   A B C D A B C D A B D E
P A B C D A B D

i=5,j=1,开始新的比较，当j=7时失败。

S B B C   A B C D A B   A B C D A B C D A B D E
P A B C D A B D

利用已经得到的“部分匹配”结果，将模式串的指示器向右“滑动”一段距离后，继续进行比较。

- 进一步思考部分匹配的结果
  - 假定主串中第*i*个字符与模式串第*j*个字符相比较失败，则应有$S_i \neq P_j$，匹配结果如下：

$$S_{i-j+1}\ S_{i-j+2}\cdots\ S_{i-k+1}S_{i-k+2}\cdots S_{i-1}\ S_i$$
$$P_1\ P_2\cdots\ P_{j-k+1}P_{j-k+2}\cdots P_{j-1}\ P_j$$

将模式串向右滑动，找到一个$k<j$, S_i 与 P_k 继续进行比较

则应有：$S_{i-k+1}\ S_{i-k+2}\cdots\ S_{i-1}\ S_i$
$P_1\ P_2\cdots\ P_{k-1}\ P_k$ 成立

$$S_{i-j+1}\ S_{i-j+2}\cdots\ S_{i-k+1}\ S_{i-k+2}\cdots S_{i-1}\ S_i$$
$$P_1\ P_2\cdots\ P_{j-k+1}\ P_{j-k+2}\cdots P_{j-1}\ P_j$$
$$P_1\ P_2\cdots\ P_{k-1}\ P_k$$

因为有：$P_{j-k+1}P_{j-k+2}\cdots P_{j-1}=S_{i-k+1}S_{i-k+2}\cdots S_{i-1}$

– 所以

$$P_{j-k+1}P_{j-k+2}\cdots P_{j-1}=P_1P_2\cdots P_{k-1}$$

*k*值只与模式串有关，与主串无关。

$$S_{i-j+1}\ S_{i-j+2}\cdots\ S_{i-k1+1}\ S_{i-k1+2}\cdots S_{i-1}\ S_i$$
$$P_1\ P_2\cdots\ P_{j-k1+1}\ P_{j-k1+2}\cdots P_{j-1}\ P_j$$
$$P_1\ P_2\cdots\ P_{k1-1}\ P_{k1}$$

K1<*k*2,使得第一个位置不同

$$S_{i-j+1}\ S_{i-j+2}\cdots\ S_{i-k2+1}\ S_{i-k2+2}\cdots\cdots S_{i-1}\ S_i$$
$$P_1\ P_2\cdots\ P_{j-k2+1}\ P_{j-k2+2}\cdots\cdots P_{j-1}\ P_j$$
$$P_1\ P_2\cdots\cdots\ P_{k2-1}\ P_{k2}$$

为了取得子串第一个字符在主串第一次位置的值，k是模式串“前缀”和“后缀”的最长的共有元素的长度。

- “部分匹配值”就是“前缀”和“后缀”的最长的共有元素的长度。
  - "前缀"指除了最后一个字符以外，一个字符串的全部头部组合
  - "后缀"指除了第一个字符以外，一个字符串的全部尾部组合

模式串的各个子串	前缀	后缀	最大公共元素长度
A	空	空	0
AB	A	B	0
ABC	A,AB	C,BC	0
ABCD	A,AB,ABC	D,CD,BCD	0
ABCDA	A,AB,ABC,ABCD	A,DA,CDA,BCDA	1
ABCDAB	A,AB,ABC,ABCD,ABCDA	B,AB,DAB,CDAB,BCDAB	2
ABCDABD	A,AB,ABC,ABCD,ABCDA ABCDAB	D,BD,ABD,DABD,CDABD BCDABD	0

KMP 算 法

该改进算法是由D.E.Knuth， J.H.Morris和 V.R.Pratt提出来的，简称为KMP算法。其改进在于：

每当一趟匹配过程出现字符不相等时，$S_i \neq P_j$, 主串指示器不用回溯，而是利用已经得到的“部分匹配”结果，将模式串的指示器向右“滑动”一段距离后确定一个k值，k<j, 使得S_i 与 P_k 继续进行比较。k的值只和P以及j有关，定义为Next[j]数组。

① 寻找前缀和后缀最长公共元素长度
② 求next数组

- Next[j]的定义
- Next[j] =

$$\begin{cases} 0, & j=1\text{时} \\ \text{Max}\{k \mid 1<k<j \text{ and } p_1p_2\cdots p_{k-1}=p_{j-k+1}\cdots p_{j-1}\} \\ 1,\text{其它情况} \end{cases}$$

Next数组的实质是找模式串中的最长相同的前缀和后缀。
$p_1p_2\cdots p_{k-1}=p_{j-k+1}\cdots p_{j-1}$

j	1 2 3 4 5 6 7 8
	a b a a b c a c
Next[j]	0 1 1 2 2 3 1 2

j	1 2 3 4 5 6 7 8
	a b a a b c a c
Next[j]	0 1 1

j=3, a b
长度为1的子串，a和b，不相同，第三种情况

j	1 2 3 4 5 6 7 8
	a b a a b c a c
Next[j]	0 1 1 2

j=4, a b a

长度为2的子串，ab和ba，不相同

长度为1的子串，a和a， 相同,k=2

j	1 2 3 4 5 6 7 8
	a b a a b c a c
Next[j]	0 1 1 2 2

j=5, a b a a

长度为3的子串，aba和baa，不相同

长度为2的子串，ab和aa， 不相同

长度为1的子串，a和a， 相同,k=2

j	1 2 3 4 5 6 7 8
	a b a a b c a c
Next[j]	0 1 1 2 2 3

j=6, a b a a b
长度为4的子串，abaa和baab，不相同
长度为3的子串，aba和aab，不相同
长度为2的子串，ab和ab，相同,k=3

j	1 2 3 4 5 6 7 8
	a b a a b c a c
Next[j]	0 1 1 2 2 3 1 2

j	1 2 3 4 5
	a b c a c
Next[j]	0 1 1 1 2

No.1 a b a b c a b c a c b a b
a b c a c
i=3, j=3 时失败，next[3]=1,即让模式串的第一个字符与当前主串进行比较。

No.2 a b a b c a b c a c b a b ➡ a b a b c a b c a c b a b
a b c a c ➡ a b c a c
i=7, j=5 时失败， next[5]=2,即让模式串的第二个字符与当前主串进行比较。

No.3 a b a b c a b c a c b a b ➡ a b a b c a b c a c b a b
a b c a c ➡ a b c a c
成功


```
int KMP_index(StringType s, StringType p)
{   int i=0,j=0;
    int n=s.length;
    int m=p.length;
    while (i<n)&&(j<m)
    {  if ( s.str[i]==p.str[j])  (j=0) || ( s.str[i]==p.str[j])
         {   i++; j++;    }
         else
          { i=i-j+1; i=0;  }   j=Next[j];
    }
    if (j>=m) return i-j;
     else  return –1;
  }
```

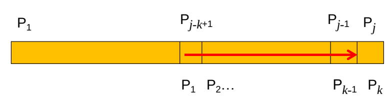

很显然，KMP_index函数是在已知下一个函数值的基础

上执行的，以下讨论如何求next数组?

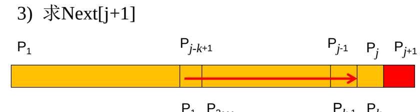
模式串的next[j]值与主串s无关，只与模式串p本身的构成有关，则可把求next数组的问题看成是一个模式匹配问题。

计算Next数组的方法

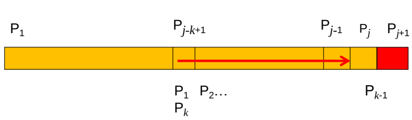
1) Next[1]=0;
2) 设 Next[j]=k； 则意味着
$P_{j-k+1}P_{j-k+2}\ldots P_{j-1} = P_1P_2\ldots P_{k-1}$

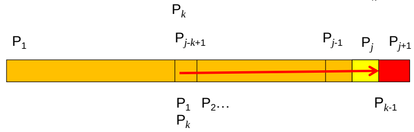


3) 求Next[j+1]



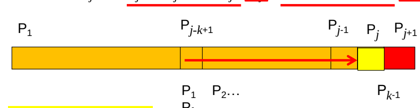
3) 求Next[j+1]
当$P_k=P_j$时 $P_{j-k+1}P_{j-k+2}\ldots P_{j-1}\boxed{P_j}=P_1P_2\ldots\ \ P_{k-1}\boxed{P_k}$



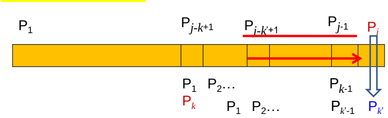


Next[j+1]=Next[j]+1;

当$P_k \neq P_j$时 $P_{j-k+1}P_{j-k+2}\ldots P_{j-1}\boxed{P_j} \neq P_1P_2\ldots\ \ P_{k-1}\boxed{P_k}$



存在一个next[k]=k'



当 $P_j=P_{k'}$， Next[j+1]=Next[k]+1;

同理，若$P_j=P_{k'}$，则将模式串继续向右滑动至模式串的第next[k’] 个字符与P_j对齐，….，依次类推，直到P_j和模式串中某个字符匹配成功或者不存在$k'(1<k'<j)$，则Next[j+1]=1.

Next数组的无回溯匹配计算

```
void Makenext(StringType p,int *pNext)
{
   int i,k; i=1;k=0;Next[1] = 0;
   while(i< p.length)
   {
        if (k==0)||( p.str[i]==p.str[k])
        {  j=j+1; k=k+1;       Next[j]=k;     }
     else k=Next[k];
}
```


KMP算法的时间复杂度分析

- 假设现在文本串S匹配到 i 位置，模式串P匹配到 j 位置，发现如果某个字符匹配成功，模式串首字符的位置保持不动，仅仅是i++、j++；如果匹配失配，i 不变（即 i 不回溯），模式串会跳过匹配过的next [j]个字符。
- 整个算法最坏的情况是，当模式串首字符位于i - j的位置时才匹配成功，算法结束。
- 如果文本串的长度为n，模式串的长度为m，那么匹配过程的时间复杂度为$O(n)$，算上计算next的$O(m)$时间，KMP的整体时间复杂度为$O(m + n)$。

Next数组特殊情况

j	1	2	3	4	5	6	7	8	9	10	11	12	13	
	a	a	a	a	a	a	a	b	b	b	c	a	a	
	0	1	2	3	4	5	6	7	1	1	1	1	2	

通过例子可以发现，当中的2、3、4、5、6步骤，都是多余的判断。由于P串中的这些位置的字符都与Next所指示的字符相等，即使从这个位置开始，也是匹配不成功的结构。

Next数组的改进

方法：如果a位字符与它next值指向的b位字符相等，则a位的nextval就指向b位的nextval值，如果不等，该a位的nextval值就是它自己a位的next值。

j	1 2 3 4 5 6 7 8 9
模式串P	a b a b a a a b a
next[j]	0 1 1 2 3 4 2 2 3
nextval[j]	0 1 0 1 0 4 2 1 0

j	1 2 3 4 5 6 7 8 9
模式串P	a a a a a a a a b
next[j]	0 1 2 3 4 5 6 7 8
nextval[j]	0 0 0 0 0 0 0 0 8