串和线性结构的区别
串的各种概念
串的匹配（KMP算法）（next数组）

2.4 String

- **Definition and operations**
- **Applications**

2.4.1. Definition

- 串：零个或多个字符组成的有限序列。
- 串长度：串中所包含的字符个数。
- 空串：长度为0的串，记为：“ ”。
- 非空串通常记为：　S=" s1 s2 ...... sn "
  - 其中：S是串名，双引号是定界符，双引号引起来的部分是串值，si（1≤i≤n）是一个任意字符。
  - 字符集：ASII码、扩展ASII码、Unicode字符集
- 子串：串中任意个连续的字符组成的子序列。
- 主串：包含子串的串。
- 子串的位置：子串的第一个字符在主串中的序号。

a= “Welcome to Beijing”
b= “Welcome”
c= “Bei”
d= “welcometo”
子串的位置：子串在主串中第一次出现的第一个字符的位置。
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= “program　”

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 Length(s)
- （4）串连接 Concat(s1,s2)
- （5）求子串 SubStr(s1,s2,start,len)
- （6）子串的定位 Index(s1,s2)
- （7）子串的插入和删除

2.4.2 串的模式匹配算法 (Pattern Matching)

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，*T*为模式串，且不妨设：

S= “$s_1s_2\cdots s_{n-1}$”　T= “$t_1\cdots t_{m-1}$”

朴素模式匹配算法(Brute-Force算法）：枚举法

从主串S的第一个字符开始和模式T 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T 的第一个字符进行比较，重复上述过程，直到T 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。

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

No.2 a b a b c a b c a c b a b
a b c a c
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
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
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
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
成功


```
int index-1(sstring s,sstring t,int pos)
{   int i,j,k;
    int n=s.length;
    int m=t.length;
    for(i=1;i<=n-m+1;i++)
     {   j=1;k=i;
         while(j<=m && s.ch[k]==t.ch[j]
          {       k++; j++;    }
         if (j>m)    return i;
      }
      return –1;
   }
```



```
int index-2(sstring s,sstring t,int pos)
{   int i=1,j=1;
    int n=s.length;
    int m=t.length;
    while (i<=n)&&(j<=m)
     {   if ( s.ch[i]==t.ch[j])
           {   i++; j++;    }
          else
           {  i=i-j+1; j=1;  }
       }
       if (j>m) return i-j+2;
        else  return –1;
   }
```


KMP 算 法：改进的模式匹配算法

- 为什么BF算法时间性能低？
  - 在每趟匹配不成功时存在大量回溯，没有利用已经部分匹配的结果。
- 如何在匹配不成功时主串不回溯？
  - 主串不回溯，模式就需要向右滑动一段距离。
- 如何确定模式的滑动距离？
  - 利用已经得到的“部分匹配”的结果
  - 将模式向右“滑动”尽可能远的一段距离(next[j])后，继续进行比较

– 出发点：利用前面匹配的结果，进行无回溯匹配

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

No.2 a b a b c a b c a c b a b
a b c a c
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
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
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
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
成功

KMP 算 法

- 思考：
  – 假定：主串为 $S_1S_2\ldots S_n$
  – 模式串为 $P_1P_2\ldots P_m$
  – 当主串中的第 i 个字符和模式串中的第 j 个字符出现不匹配，主串中的第i个字符应该和模式串中的哪个字符匹配（无回溯，主串指针不回溯，模式串右移动）？

- 进一步思考
  - 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$S_i \neq P_j$

$S_{i-j+1}\ S_{i-j+2}\cdots \quad S_{i-k+1}S_{i-k+2}\cdots S_{i-1}\quad S_i$

$P_1\quad P_2\cdots \quad P_{j-k+1}P_{j-k+2}\cdots P_{j-1}\quad P_j$

$k<j$

$S_{i-k+1}\quad S_{i-k+2}\cdots\quad S_{i-1}\quad S_i$

$P_1\quad P_2\cdots\quad P_{k-1}\quad P_k$ 成立

S_i 与 P_k 进行比较

$S_{i-j+1}\ S_{i-j+2}\cdots \quad S_{i-k+1}\ S_{i-k+2}\cdots S_{i-1}\quad S_i$

$P_1\quad P_2\cdots \quad P_{j-k+1}\ P_{j-k+2}\cdots P_{j-1}\quad P_j$

$P_1\quad P_2\cdots\quad P_{k-1}\quad P_k$

- 而根据已有的匹配，有
  - $P_{j-k+1}P_{j-k+2}\cdots P_{j-1} = S_{i-k+1}S_{i-k+2}\cdots S_{i-1}$
- 因此
  - $P_{j-k+1}P_{j-k+2}\cdots P_{j-1} = P_1P_2\cdots P_{k-1}$
- 因此*k*值只和*P*以及*j*有关，定义为Next[j]

意义：当P_j比较失败时，右移动模式串，让P_k与当前S_i元素进行比较

- Next[j]的定义
- Next[j] =

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

Next数组的实质是赵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
	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

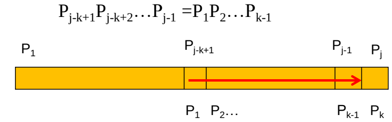
No.2 a b a b c a b c a c b a b
a b c a c
i=7,j=5 时失败

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

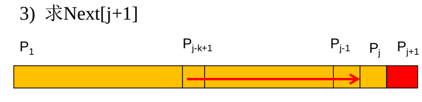

```
int index(sstring s,sstring t,int pos)
{   int i,j;
    int n=s.length;
    int m=t.length;
    while (i<=n)&&(j<=m)
    {  if ( s.ch[i]==t.ch[j])   (j=0) || ( s.ch[i]==t.ch[j])
         {   i++; j++;    }
       else
         { i=i-j+1; j=1; }    j=Next[j];
    }
    if (j>m) return i-j+2;
     else  return –1;
}
```


计算Next数组的方法

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

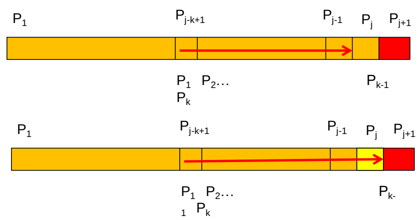


3) 求Next[j+1]



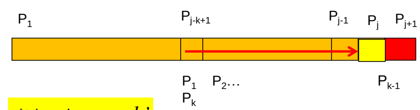
3) 求Next[j+1]

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

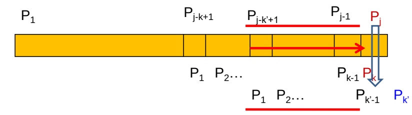


Next[j+1]=Next[j]+1;

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



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(String p,int *pNext)
{
   int i,k; i=1;k=0;Next[1] = 0;
   while(i<Length(p))
   {
        if (k==0)||( p[i]==p[k])
        { j=j+1; k=k+1;      Next[j]=k;    }
      else k=Next[k];
}
```