第3章 字符串

串(字符串)是一种特殊的线性表,它的数据元素仅由一个字符组成。串是计算机在处理非数值对象时常见的一种数据类型。考虑到串自身所具有的一些特性,以及串在各类算法设计实践及程序设计竞赛中的广泛应用,本章在简要介绍串的基本概念、存储结构及基本运算的基础上,重点介绍串的模式匹配算法。随着人工智能技术的不断发展和突破,模式匹配算法作为人工智能的重要组成部分,在数据挖掘、计算机视觉、自然语言处理等领域发挥着巨大的作用,以此来解决实际问题,提高生产效率以及解决人类面临的共性问题。


观看视频

3.1 串类型定义

串(String)是由零个或多个任意字符组成的有限序列。一般记作 s ="$s_1 s_2 \cdots s_n$",其中 s 是串名;双引号为串的定界符,双引号内的内容($s_1 s_2 \cdots s_n$)为串值,引号本身不是串的内容;s_i($1 \leqslant i \leqslant n$)是串的元素;$n$ 为串的长度,表示串中所包含的字符个数,当 $n = 0$ 时,称为空串,通常记为 Φ。

串中任意多个连续的字符组成的子序列称为该串的子串,Φ 为特殊的子串。包含子串的串称为主串。子串的第一个字符在主串中的序号称为子串的位置。当且仅当两个串的长度相等且对应字符都相同时,称为两个**串相等**。

ADT String{
数据对象: D = {a_i | $a_i \in$ ElemSet, i = 1,2,3, …, n, n≥0}
数据关系: R = {< a_{i-1}, a_i > | a_{i-1}, $a_i \in$ D, i = 1,2,3, …, n}
基本操作集如下。
(1) StrLength(s): 求出串 s 的长度。
(2) StrAssign(s1,s2): 将 s2 的串值赋值给 s1。
(3) StrConcat(s1,s2,s): s1 与 s2 连接以后的结果存于 s 中,原 s1、s2 的值不变。或 StrConcat(s1, s2): 将 s2 的内容连接于 s1 之后,s1 改变,s2 不变。
(4) SubStr(s,i,len): 返回从串 s 的第 i 个字符开始的长度为 len 的子串。len = 0 或 i > StrLength(s) 得到的是 Φ,i≤StrLength(s)且 len > StrLength(s)则取第 i 个字符开始到串的最后一个字符的子序列作为返回值。
(5) StrCmp(s1,s2): 比较两个串 s1、s2,若 s1 = s2,返回值 0; 若 s1 < s2,返回值< 0; 若 s1 > s2,返回值 > 0。
(6) StrIndex(s,t): 找子串 t 在主串 s 中首次出现的位置。若 t∈s,则操作返回 t 在 s 中首次出现的位置,否则返回值为 - 1。
(7) StrInsert(s,i,t): 将串 t 插入串 s 的第 i 个字符位置上,1≤i≤StrLength(s) + 1。
(8) StrDelete(s,i,len): 删除串 s 中从第 i 个字符开始的长度为 len 的子串,1≤i≤StrLength(s),

```
    0≤len≤StrLength(s) - i + 1。
(9) StrRep(s,t,r): 用子串 r 替换串 s 中出现的所有子串 t。
}ADT String
```

以上是串的几个基本操作。其中前 5 个操作是最为基本的，它们不能用其他操作来合成，因此通常将这 5 个基本操作称为最小操作集。

3.2 串的表示和实现


观看视频

3.2.1 串的定长顺序存储结构及其基本运算实现

1. 串的定长顺序存储结构

按预定义的大小，用一组地址连续的存储单元为每个串变量分配一个固定长度的存储区以存储串值中的字符序列。例如：

```
#define MAXSIZE  256
char  s[MAXSIZE];
```

则串的最大长度不能超过 256 个字符。

定长顺序表示带来的一个问题是如何掌握串的实际长度。以下是 3 种常用标识方法。

(1) 用一个指针来指向最后一个字符，其串描述如下。

```
typedef struct
{ char  data[MAXSIZE];
  int   curlen;
} SeqString;
```

定义一个串变量：

```
SeqString s;
```

这种存储方式可以直接得到串的长度 s.curlen + 1，如图 3.1 所示。

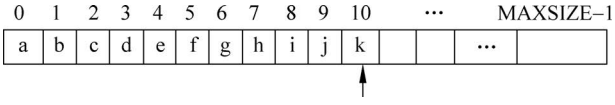


图 3.1 串的顺序存储方式 1

(2) 在串尾存储一个不会在串中出现的特殊字符作为串的终结符，以此表示串的结尾。例如，C 语言中处理定长串的方法就是用“\0”来表示串的结束。这种存储方法不能直接得到串的长度，而是用判断当前字符是否是“\0”来确定串是否结束，从而可求得串的长度，如图 3.2 所示。

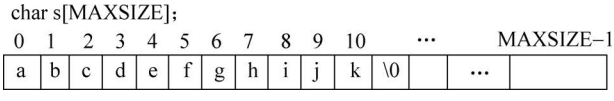


图 3.2 串的顺序存储方式 2

(3) 设定长串存储空间 char s[MAXSIZE + 1]，用 s[0]存放串的实际长度，串值存放在 s[1]～s[MAXSIZE]，字符的序号和存储位置一致，应用更为方便，如图 3.3 所示。

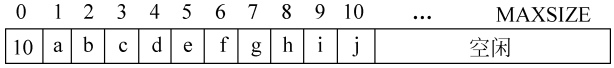


图 3.3　串的顺序存储方式 3

2. 定长顺序串的基本运算

下面简要介绍定长顺序串连接、求子串、串比较算法，其插入和删除等运算基本与顺序表相同，在此不再赘述。

1）串连接

把两个串 s1 和 s2 首尾连接成一个新串 s，即 s≤s1 + s2。

```
int StrConcat1(s1,s2,s)
  char s1[],s2[],s[];
{ int i = 0, j, len1, len2;
  len1 = StrLength(s1); len2 = StrLength(s2)
  if  (len1 + len2 > MAXSIZE - 1)  return  0;        /* s 长度不够 */
  j = 0;
  while(s1[j]!= '\0')  { s[i] = s1[j];i++; j++; }   /* 复制 s1 的内容到 s */
  j = 0;
  while(s2[j]!= '\0')  { s[i] = s2[j];i++; j++; }   /* 复制 s2 的内容到 s */
  s[i] = '\0';                                       /* 在 s 串中建立结束标志 */
  return 1;
}
```

2）求子串

```
int StrSub(char *t, char *s, int i, int len)
  /* 用 t 返回串 s 中第 i 个字符开始的长度为 len 的子串 1≤i≤串长 */
  { int slen;
    slen = StrLength(s);
      if (i < 1 || i > slen || len < = 0)      /* i 位置值不正确或 len≤0 时返回空串或不操作 */
        return 0;
    for (j = 0; j < len; j++)
      t[j] = s[i + j - 1];                          /* 取出长度≤len 的子串 t */
    t[j] = '\0';                                    /* 建立结束标志 */
    return 1;
  }
```

3）串比较

```
int StrComp(char *s1, char *s2)
{ int i = 0;
  while (s1[i] == s2[i] && s1[i]!= '\0')  i++;       /* 对应字符相同且 s1 串不是结束标志 */
  return (s1[i] - s2[i]);           /* 若同时结束则相等，否则对应字符的大小即为比较结果 */
}
```

3.2.2　串的堆存储结构及其基本运算实现


观看视频

1. 串名的存储映像

串名的存储映像是串名-串值内存分配对照表（索引表）。假设 s1 ="abcdef"，s2 ="hij"，常见的串名-串值存储映像索引表有如下几种。

1）带串长度的索引表

如图 3.4 所示，索引项的结点类型定义如下。

```
typedef  struct
  { char  name[MAXNAME];                    /*串名*/
    int length;                             /*串长*/
    char *stradr;                           /*起始地址*/
  } LNode;
```

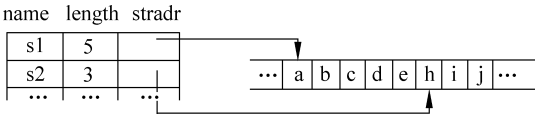


图 3.4　带串长度的索引表

2）带末尾指针的索引表

如图 3.5 所示，索引项的结点类型定义如下。

```
typedef  struct
  { char  name[MAXNAME];                    /*串名*/
    char *stradr, *enadr;                   /*起始地址,末尾地址*/
  } ENode;
```

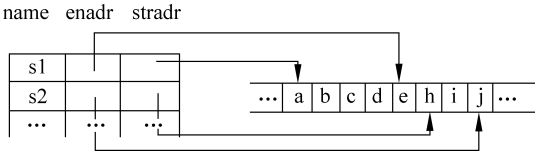


图 3.5　带末尾指针的索引表

3）带特征位的索引表

当一个串的存储空间需求不超过一个指针的存储空间时，可以直接将该串存储在索引项的指针域内，这样既节约了存储空间，又提高了查找速度，但同时要增加一个特征位 tag 以区分指针域存放的是指针还是串。

如图 3.6 所示，索引项的结点类型定义如下。

```
typedef  struct
  { char  name[MAXNAME];
    int tag;                                /*特征位*/
    union                                   /*起始地址或串值*/
      {char *stradr;
       char value[4];
      };
  } TNode;
```

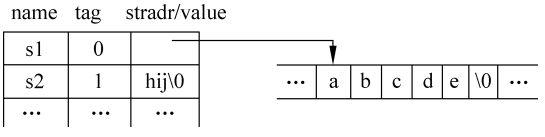


图 3.6　带特征位的索引表

2. 堆存储结构

堆存储结构的基本思想是：在内存中开辟一块地址连续的存储空间作为应用程序中所有串的可利用存储空间(堆空间)，并根据每个串的实际长度动态地为其申请相应大小的存储区域。

如图 3.7 所示，store[SMAX + 1]是一个堆存储结构示意图。阴影部分是已分配的区域，free 为未分配部分的起始地址。当向 store 中存放一个串时，要填上该串的索引项。

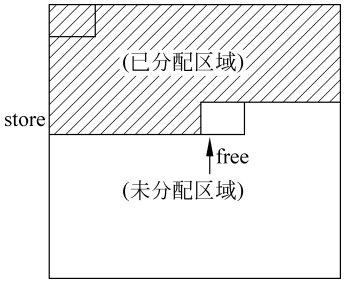


图 3.7　堆结构示意图

3. 基于堆存储结构的基本运算

设堆存储空间为

```
char  store[SMAX + 1];
```

自由区指针：

```
int  free;
```

串的存储映像类型如下。

```
typedef  struct
  { int  length;                                  /* 串长 */
    int  stradr;                                  /* 起始地址 */
  } HString;
```

1) 串常量赋值

```
void StrAssign(HString *s1,char *s2)
    /* 将一个字符型数组 s2 中的字符串送入堆 store 中,free 是自由区的指针 */
  { int i = 0,len;
    len = StrLength(s2);                          /* 计算 s2 的串长 */
    if (len < 0||free + len - 1>SMAX)             /* 空间不够不分配 */
        return 0;
    else {for (i = 0;i < len;i++)
        store[free + i] = s2[i];                  /* 逐个字符地复制 s2 内容到堆空间 */
        s1.stradr = free;                         /* 建立 s2 的索引指针 */
        s1.len. = len;                            /* 记录 s2 的长度 */
        free = free + len;                        /* 修改堆空间中的 free 指针 */
    }
  }
```

2) 赋值一个串

```
void  StrCopy(Hstring *s1,Hstring s2)
  /* 该运算将堆 store 中的一个串 s2 复制到 store 中的一个新串 s1 中 */
{ int i;
  if (free + s2.length - 1 > SMAX)  return  error;   /* 检查 store 的空间是否够,不够给出出错
                                                        信息 */
  else { for(i = 0; i < s2.length;i++)
        store[free + i] = store[s2.atradr + i];       /* 逐个复制 s2 的字符到 s1 中 */
        s1 -> length = s2.length;                     /* s1 的长度定义为 s2 的长度 */
        s1 -> stradr = free;                          /* 设置 s1 的首地址   */
        free = free + s2.length;                      /* 修改 free 指针 */
       }
}
```

3）求子串

```
void  StrSub(Hstring *t, Hstring s,int i,int len)
  /*该运算将串 s 中第 i 个字符开始的长度为 len 的子串送入一个新串 t 中*/
{ int i;
  if (i<0 || len<0 || len>s.len-i+1)  return  error;     /*位置、长度等参数有误,给出出
                                                          错信息*/
  else { t->length=len;                          /*定义 t 的长度*/
         t->stradr=s.stradr+i-1;                 /*定义 t 的起始地址*/
       }
}                                                /*该算法本质上只是建立了一个新的串 t 的索引*/
```

4）串连接

```
void Concat(s1,s2,s)
HString s1,s2;
HString *s;
{ HString t;
  StrCopy(s,s1);                                 /*将 s1 的内容复制到 s*/
  StrCopy(&t,s2);                                /*在前面 s1 内容复制之后将 s2 的内容复制到 s*/
  s->length=s1.length+s2.length;                 /*计算 s 的长度*/
}
```

以上扼要介绍了堆存储结构的处理思想，但诸如废弃串的回归、自由区的管理等很多问题及细节均未涉及。事实上，在常用的高级语言及开发环境中都提供了串的类型及大量的库函数，读者可直接使用，从而使算法的设计和调试更简便、可靠。


观看视频

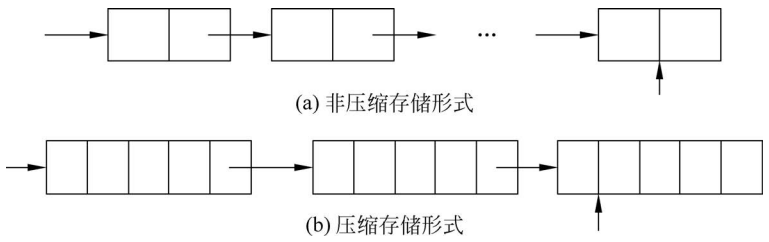
3.2.3 串的链式存储结构及其基本运算实现

1. 串的链式存储结构

串的链式存储结构即用带头结点的单链表形式存储串，称为链式串，其结点的类型定义如下。

```
Typedef struct node
{ Char data;                                     //存放字符
  Struct node *next;                             //指针域
}Linkstring;
```

针对上述串的链式存储结构，可以有非压缩存储形式和压缩存储形式两种解决方案，如图 3.8(a)和图 3.8(b)所示。



(a) 非压缩存储形式

(b) 压缩存储形式

图 3.8　串的链式存储结构

(1) 非压缩存储形式：一个结点只存储一个字符，其优点是操作方便，但存储空间利用率低。

(2) 压缩存储形式：一个结点存储多个字符。这种存储结构提高了存储空间的利用

率,但也增加了实现基本操作算法的复杂性。例如,改变串长的操作可能会涉及结点的增加与删除问题。

2. 链式串基本运算算法实现

1）串赋值运算算法

将一个 C/C++字符数组 t 赋给串 s。

```
Void StrAssign(LinkString *&s, char t[])
{   int i=0;
    LinkString *q, *tc;
    s=(Linkstring *) malloc(sizeof(LinkString));    /*建立头结点*/
    s->next=NULL;                /*初始化链表指针*/
    tc=s;
    while (t[i]!='\0')              /*将整个串逐个字符地申请结点并建立链表及相应的值*/
    {   q=(LinkString *) malloc(sizeof(LinkString));
        q->data=t[i];
        tc->next=q; tc=q;
        i++;
    }
    tc->next=NULL;                      /*终端结点的 next 置 NULL*/
}
```

2）串连接运算算法

将串 t 连接到串 s 之后,返回其结果。

```
LinkString *Concat(LinkString *s, LinkString *t)
{   LinkString *p=s->next, *q, *tc, *r;
    r=(LinkString *) malloc(sizeof(LinkString));   //建立头结点
    r->next=NULL;
    tc=r;                                  //tc 总是指向新链表的最后一个结点
    while (p!=NULL)                        //将 s 串复制给 r
    {   q=(LinkString *) malloc(sizeof(LinkString));
        q->data=p->data;
        tc->next=q; tc=q;
        p=p->next;
    }
    p=t->next;
    while (p!=NULL)                        //将 t 串复制给 r
    {   q=(LinkString *) malloc(sizeof(LinkString));
        q->data=p->data;
        tc->next=q; tc=q;     p=p->next;
    }
    tc->next=NULL;
    return(r);
}
```

3）求子串运算算法

返回串的第 i 个位置开始的 j 个字符组成的串。

```
LinkString *SubStr(LinkString *s, int i, int j)
{   int k=1;
    LinkString *p=s->next, *q, *tc, *r;
    r=(LinkString *) malloc(sizeof(LinkString));    //建立头结点
    r->next=NULL;
    tc=r;                                           //tc 总是指向新链表的最后一个结点
    while (k<i && p!=NULL)
    {   p=p->next; k++;
```

```
    }
    if (p!= NULL)
    {   k = 1;
        while (k <= j && p!= NULL)                    //复制 j 个结点
        {   q = (LinkString * ) malloc(sizeof(LinkString));
            q -> data = p -> data;
            tc -> next = q; tc = q;
            p = p -> next;
            k++;
         }
    tc -> next = NULL;
    }
    return (r);
}
```

3.3 串的模式匹配算法

串的模式匹配，即子串定位，是一种十分重要的串运算，也是 ACM/ICPC 中的高频考点。设主串 s 和子串 t 是给定的两个串，则在 s 中寻找 t 的过程称为模式匹配（Pattern Matching），且称 t 为模式（Pattern）。如果在 s 中找到等于 t 的子串，则称匹配成功，函数返回 t 在 s 中的首次出现的存储位置（或序号），否则匹配失败，返回 -1。为了运算方便，设字符串的长度存放在 0 号单元，串值从 1 号单元存放，这样字符序号与存储位置一致。

观看视频

3.3.1 朴素匹配算法

朴素匹配算法思想如下：首先将 s_1 与 t_1 进行比较，若不同，就将 s_2 与 t_1 进行比较……直到 s 的某一个字符 s_i 和 t_1 相同，再将它们之后的字符进行比较，若也相同，则继续往下比较，当 s 的某一个字符 s_i 与 t 的字符 t_j 不同时，则 s 返回到本趟开始字符的下一个字符，即 s_{i-j+2}，t 返回到 t_1，继续开始下一趟的比较，重复上述过程。若 t 中的字符全部比完毕，则说明本趟匹配成功，本趟的起始位置是 $i-j+1$ 或 $i-t[0]$；否则，匹配失败。

设主串 s ="ababcabcacbab"，模式 t ="abcac"，匹配过程如图 3.9 所示。

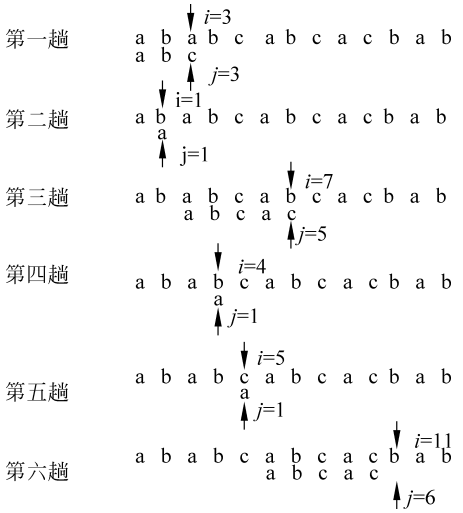


图 3.9 简单模式匹配的匹配过程

依据这个思想，算法描述如下。

```
int  StrIndex_BF(char * s,char * t)
  /* 从串 s 的第一个字符开始找首次与串 t 相等的子串 */
{ int i = 1,j = 1;
  while (i <= s[0] && j <= t[0])                    /* 都没遇到结束符 */
       if (s[i] == t[j])
         { i++;j++; }                               /* 继续 */
       else
         {i = i - j + 2; j = 1;}                    /* 回溯 */
  if (j > t[0])  return (i - t[0]);                 /* 返回存储位置 */
  else  return - 1;                                 /* 匹配失败 */
}
```

该算法简称为 BF 算法。下面分析它的时间复杂度，设串 s 的长度为 n，串 t 的长度为 m。匹配成功的情况下，考虑下面两种极端情况。

(1) 在最好情况下，每趟不成功的匹配都发生在第一对字符比较时。

例如，s ="aaaaaaaaaabc"，t ="bc"。

设匹配成功发生在 s_i 处，则字符比较次数在前面 $i-1$ 趟匹配中共比较了 $i-1$ 次，第 i 趟成功的匹配共比较了 m 次，所以共比较了 $i-1+m$ 次。所有匹配成功的可能共有 $n-m+1$ 种，设从 s_i 开始与 t 串匹配成功的概率为 p_i，在等概率情况下 $p_i = 1/(n-m+1)$，因此最好情况下平均比较的次数是

$$\sum_{i=1}^{n-m+1} p_i \times (i-1+m) = \sum_{i=1}^{n-m+1} \frac{1}{n-m+1} \times (i-1+m) = \frac{n+m}{2}$$

即最好情况下的时间复杂度是 $O(n+m)$。

(2) 在最坏情况下，每趟不成功的匹配都发生在 t 的最后一个字符。

例如，s ="aaaaaaaaaaab"，t ="aaab"。

设匹配成功发生在 s_i 处，则在前面 $i-1$ 趟匹配中共比较了 $(i-1)\times m$ 次，第 i 趟成功的匹配共比较了 m 次，所以共比较了 $i\times m$ 次，因此最坏情况下平均比较的次数是

$$\sum_{i=1}^{n-m+1} p_i \times (i \times m) = \sum_{i=1}^{n-m+1} \frac{1}{n-m+1} \times (i \times m) = \frac{m \times (n-m+2)}{2}$$

即最坏情况下的时间复杂度是 $O(n * m)$。

上述算法中匹配是从 s 串的第一个字符开始的，有时算法要求从指定位置开始，这时算法的参数表中要加一个位置参数 pos：StrIndex(char * s,int pos,char * t)，比较的初始位置定位在 pos 处，BF 算法是 pos = 1 的情况。

3.3.2　KMP 算法


观看视频

BF 算法简单但效率较低，一种对 BF 算法进行了很大改进的模式匹配算法是克努特(Knuth)、莫里斯(Morris)和普拉特(Pratt)同时发现的，简称 KMP 算法。KMP 算法的关键是利用匹配失败后的信息，从失败中提取有效信息，尽量减少模式串与主串的匹配次数，省去中间所有没有意义的匹配过程，从而达到快速匹配的目的。

1. KMP 算法的思想

造成 BF 算法速度慢的原因是回溯，即在某趟的匹配过程失败后，对于 s 串要回到本趟开始字符的下一个字符，t 串要回到第一个字符，而这些回溯并不是必需的。例如图 3.9 所示的匹配过程，在第三趟匹配过程中，$s_3 \sim s_6$ 和 $t_1 \sim t_4$ 是匹配成功的，$s_7 \neq t_5$ 匹配失败，因此有了第四趟，其实这一趟是不必要的。由图可看出，因为在第三趟中有 $s_4 = t_2$，而 $t_1 \neq t_2$，肯定有 $t_1 \neq s_4$。同理，第五趟也是没有必要的，所以从第三趟之后可以直接到第六趟，进一步分析第六趟中的第一对字符 s_6 和 t_1 的比较也是多余的，因为第三趟中已经比过了 s_6 和 t_4，并且 $s_6 = t_4$，而 $t_1 = t_4$，必有 $s_6 = t_1$，因此第六趟的比较可以从第二对字符 s_7 和 t_2 开始进行。这就是说，第三趟匹配失败后，指针 i 不动，而是将模式串 t 向右“滑动”，用 t_2“对准” s_7 继续进行，以此类推。这样的处理方法指针 i 是无回溯的。

综上所述，希望某趟在 s_i 和 t_j 匹配失败后，指针 i 不回溯，模式串 t 向右“滑动”至某个位置上，使得 t_k 对准 s_i 继续向右进行。显然，现在问题的关键是串 t“滑动”到哪个位置上？不妨设位置为 k，即 s_i 和 t_j 匹配失败后，指针 i 不动，模式串 t 向右“滑动”，使 t_k 和 s_i 对准继续向右进行比较，要满足这一假设，就要有如下关系成立：

$$"t_1\ t_2\ \cdots t_{k-1}" = "s_{i-k+1}\ s_{i-k+2}\ \cdots s_{i-1}" \tag{3.1}$$

式(3.1)左边是 t_k 前面的 $k-1$ 个字符，右边是 s_i 前面的 $k-1$ 个字符。而本趟匹配失败是在 s_i 和 t_j 之处，已经得到的部分匹配结果是

$$"t_1\ t_2\ \cdots t_{j-1}" = "s_{i-j+1}\ s_{i-j+2}\ \cdots s_{i-1}" \tag{3.2}$$

因为 $k < j$，所以有

$$"t_{j-k+1}\ t_{j-k+2}\ \cdots t_{j-1}" = "s_{i-k+1}\ s_{i-k+2}\ \cdots s_{i-1}" \tag{3.3}$$

式(3.3)左边是 t_j 前面的 $k-1$ 个字符，右边是 s_i 前面的 $k-1$ 个字符，通过式(3.1)和式(3.3)得到关系：

$$"t_1\ t_2\ \cdots t_{k-1}" = "t_{j-k+1}\ t_{j-k+2}\ \cdots t_{j-1}" \tag{3.4}$$

结论：某趟在 s_i 和 t_j 匹配失败后，如果模式串中有满足关系(见式(3.4))的子串存在，即模式串中的前 $k-1$ 个字符与模式串中 t_j 字符前面的 $k-1$ 个字符相等时，模式串 t 就可以向右“滑动”致使 t_k 和 s_i 对准，继续向右进行比较即可。


观看视频

2. next 函数

模式串中的每一个 t_j 都对应一个 k 值，由式(3.4)可知，这个 k 值仅依赖于模式串 t 本身字符序列的构成，而与主串 s 无关。用 next[j]表示 t_j 对应的 k 值，根据以上分析，next 函数有如下性质。

(1) next[j]是一个整数，且 $0 \leqslant \text{next}[j] < j$。

(2) 为了使 t 的右移不丢失任何匹配成功的可能，当存在多个满足式(3.4)的 k 值时，应取最大的，这样向右“滑动”的距离最短，“滑动”的字符为 $j - \text{next}[j]$个。

(3) 如果在 t_j 前不存在满足式(3.4)的子串，此时若 $t_1 \neq t_j$，则 $k = 1$；若 $t_1 = t_j$，则 $k = 0$。这时“滑动”的最远，为 $j-1$ 个字符，即用 t_1 和 s_{j+1} 继续比较。

因此，next 函数定义如下：

$$\text{next}[j]=\begin{cases}0, & j=1\\ k & \{k \mid 1\leqslant k<j \text{ 且 } "t_1\,t_2\cdots t_{k-1}"="t_{j-k+1}\,t_{j-k+2}\cdots t_{j-1}"\}\\ 1 & \text{不存在上面的 } k \text{ 且 } t_1\neq t_j\\ 0 & \text{不存在上面的 } k \text{ 且 } t_1=t_j\end{cases}$$

设有模式串 t ="abcaabaabc",则它的 next 函数值为

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


观看视频

3. KMP 算法

在求得模式的 next 函数之后,匹配可按如下方法进行:假设以指针 i 和 j 分别指示主串和模式中的比较字符,令 i 的初值为 pos,j 的初值为 1。若在匹配过程中 $s_i\neq t_j$ ($j=1$),则 i 和 j 分别增 1;若 $s_i\neq t_j$ ($j\neq 1$)匹配失败,则 i 不变,j 退到 next[j]位置再比较,若相等,则指针各自增 1,否则 j 再退到下一个 next 值的位置,以此类推。直至下列两种情况:一种是 j 退到某个 next 值时字符比较相等,则 i 和 j 分别增 1,并继续进行匹配;另一种是 j 退到值为零(即模式的第一个字符失配),则此时 i 和 j 也要分别增 1,表明从主串的下一个字符起和模式重新开始匹配。

设主串 s ="aabcbabcaabcaababc",子串 t ="abcaababc",图 3.10 是一个利用 next 函数进行匹配的过程示意图。

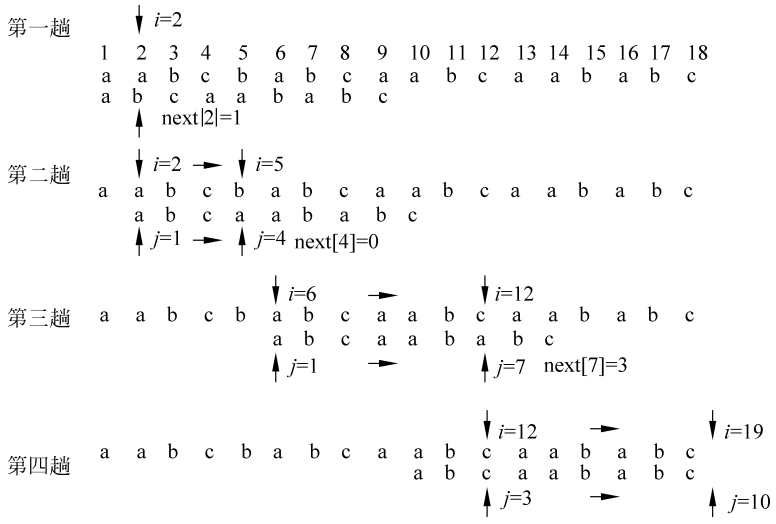


图 3.10　利用模式 next 函数进行匹配的过程示例

在假设已有 next 函数情况下,KMP 算法如下。

```
int StrIndex_KMP(char * s,char * t,int pos)
  /* 从串 s 的第 pos 个字符开始找首次与串 t 相等的子串 */
  { int i = pos,j = 1,slen,tlen;
    while (i <= s[0] && j <= t[0])                    /* 都没遇到结束符 */
```

```
    if (j == 0||s[i] == t[j]) { i++; j++; }
    else  j = next[j];                                  /* 回溯 */
  if (j > t[0])  return  i - t[0];                      /* 匹配成功,返回存储位置 */
     else  return  -1;
}
```

4. 如何求 next 函数

由以上讨论可知，next 函数值仅取决于模式本身而和主串无关。可以从分析 next 函数的定义出发用递推的方法求得 next 函数值。

由定义知：

$$\text{next}[1] = 0 \tag{3.5}$$

设 next$[j]=k$，即有

$$"t_1\ t_2\ \cdots t_{k-1}" = "t_{j-k+1}\ t_{j-k+2}\ \cdots t_{j-1}" \tag{3.6}$$

next$[j+1]$可能有以下两种情况。

第一种情况：若 $t_k = t_j$，则表明在模式串中

$$"t_1\ t_2\ \cdots t_k" = "t_{j-k+1}\ t_{j-k+2}\ \cdots t_j" \tag{3.7}$$

这就是说 next$[j+1]=k+1$，即

$$\text{next}[j+1] = \text{next}[j] + 1 \tag{3.8}$$

第二种情况：若 $t_k \neq t_j$，则表明在模式串中

$$"t_1\ t_2\ \cdots t_k" \neq "t_{j-k+1}\ t_{j-k+2}\ \cdots t_j" \tag{3.9}$$

此时可把求 next 函数值的问题看成一个模式匹配问题，整个模式串既是主串又是模式，而当前在匹配的过程中，已有式(3.6)成立，则当 $t_k \neq t_j$ 时应将模式向右滑动，使得第 next$[k]$个字符和“主串”中的第 j 个字符相比较。若 next$[k]=k'$，且 $t_{k'}=t_j$，则说明在主串中第 $j+1$ 个字符之前存在一个最大长度为 k'的子串，使得

$$"t_1\ t_2\ \cdots t_{k'}" = "t_{j-k'+1}\ t_{j-k'+2}\ \cdots t_j" \tag{3.10}$$

因此

$$\text{next}[j+1] = \text{next}[k] + 1 \tag{3.11}$$

同理，若 $t_{k'} \neq t_j$，则将模式继续向右滑动致使第 next$[k']$个字符和 t_j 对齐，以此类推，直至 t_j 和模式中的某个字符匹配成功或者不存在任何 $k'(1 < k' < k < \cdots < j)$满足式(3.10)，此时若 $t_1 \neq t_{j+1}$，则有

$$\text{next}[j+1] = 1 \tag{3.12}$$

否则若 $t_1 = t_{j+1}$，则有

$$\text{next}[j+1] = 0 \tag{3.13}$$

综上所述，求 next 函数值过程的算法如下。

```
void GetNext(char *t,int next[ ])
  /*求模式 t 的 next 值并存入 next 数组中*/
  { int i = 1,j = 0;
    next[1] = 0;
    while (i < t[0])
      { while (j > 0&&t[i]!= t[j])  j = next[j];
        i++;  j++;
        if (t[i] == t[j])  next[i] = next[j];
```

```
            else  next[i] = j;
        }
    }
```

上述算法的时间复杂度是$O(m)$,所以KMP算法的时间复杂度是$O(n \times m)$,但在一般情况下,实际的执行时间是$O(n+m)$。当然,KMP算法和简单的模式匹配算法相比,设计难度增加很大,我们主要学习该算法的设计技巧。

3.3.3 基于KMP算法的应用举例

下面给出的是一个ACM/ICPC的实战练习题Oulipo,题目请见POJ网。[POJ 3461]

题目要求:给出两个字符串,求出模式串pat在主串text中出现了多少次。

输入

The first line of the input file contains a single number: the number of test cases to follow. Each test case has the following format:

- One line with the word W, a string over {'A', 'B', 'C', …, 'Z'}, with $1 \leqslant |W| \leqslant 10,000$ (here $|W|$ denotes the length of the string W).
- One line with the text T, a string over {'A', 'B', 'C', …, 'Z'}, with $|W| \leqslant |T| \leqslant 1,000,000$.

输出

For every test case in the input file, the output should contain a single number, on a single line: the number of occurrences of the word W in the text T.

输入示例	输出示例
3	1
BAPC	3
BAPC	0
AZA	
AZAZAZA	
VERDI	
AVERDXIVYERDIAN	

参考程序如下:

```
#include<iostream>
#include<cstring>
#include<cstdio>
using namespace std;
const int nMax=10005;
const int mMax=1000005;
char text[mMax],pat[nMax];
int lent,lenp,next[nMax];

void get_next(){                          /*计算模式串的next函数*/
    int i,j=-1;
    next[0]=-1;
    for(i=1;i<=lenp;i++){          //pat[j]可以理解为i的前一个字符的next值所指向的字符
        while (j>-1&&pat[j+1]!=pat[i]) j=next[j];
        if(pat[j+1]==pat[i]) j++;
```

```
            next[i] = j;
        }
    }

    int KMP(){
        int ans = 0,i = 0,j = -1;
        get_next();                          /* 调用 next 函数查找匹配情况 */
        for(i = 0;i < lent;i++){
            while(j!= -1&&pat[j + 1]!= text[i]){
                j = next[j];
            }
            if(pat[j + 1] == text[i]) j = j + 1;
            if(j == lenp - 1) ans++;         //找到一个匹配,计数器加 1
        }
        return ans;
    }

    int main(){
        int t;
        scanf("%d",&t);
        while(t--){
            scanf("%s%s",pat,text);          /* 输入子串和主串 */
            lenp = strlen(pat);              /* 计算模式串长度 */
            lent = strlen(text);             /* 计算主串长度 */
            printf("%d\n",KMP());           /* 求解出现次数并输出 */
        }
        return 0;
    }
```

习题

(1) 若串 s ="software",其子串的个数是多少?

(2) 空串和空格串有何区别? 串中的空格符有何意义? 空串在串处理中有何作用?

(3) 已知模式串"cddcdececdea",计算其对应的 next 数组。

(4) 采用顺序结构存储串,编写一个函数,求串 s 和串 t 的一个最长公共子串。

(5) 如果字符串的一个子串(其长度大于 1)的各字符均相同,则称为等值子串。试设计一个算法,输入字符串 s,以"!"作为结束标志。如果串 s 中不存在等值子串,则输出信息"无等值子串",否则求出(输出)一个长度最大的等值子串。

(6) 回文指一个对称的字符串,即该串从左到右和从右到左读的结果是相同的。请设计算法判断一个字符串 s 是否是回文,是则返回 1,否则返回 0(例如,"abba""abccba"均返回 1,"abab"返回 0)。

(7) 一个文本串可以用事先给定的字母映射表进行加密,例如,假设字母映射表如下:

a b c d e f g h I j k l m n o p q r s t u v w x y z

n g z q t c o b m u h e l k p d a w x f y I v r s j

则字符串"encrypt"被加密成"tkzwsdf",请编写算法分别完成上述问题的加密和解密工作。

ACM/ICPC 实战练习

(1) POJ 1488,UVA 272,TEX Quotes
(2) POJ 3080,ZOJ 2784,Blue Jeans
(3) POJ 1782,ZOJ 2240,Run Length Encoding
(4) POJ 2192,ZOJ 2401,Zipper
(5) POJ 2408,ZOJ 1960,Anagram Groups
(6) POJ 2141,Message Decowding
(7) POJ 1159,Palindrome
(8) POJ 1961,ZOJ 2177,Period
(9) POJ 2752,Seek the Name, Seek the Fame
(10) POJ 1598,ZOJ 1315,Excuses, Excuses!
(11) POJ 1035,ZOJ 2040,Spell checker
(12) POJ 1936,ZOJ 1970,All in All
(13) POJ 2406,ZOJ 1905,Power Strings
(14) POJ 2121,ZOJ 2311,Inglish-Number Translator
(15) POJ 2403,ZOJ 1902,Hay Points