字符串

在非数值处理、事务处理等问题常涉及到一系列的字符操作。计算机的硬件结构主要是反映数值计算的要求，因此，字符串的处理比具体数值处理复杂。下面讨论串的存储结构及几种基本的处理。

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，$a_i(1 \le i \le n)$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。

串类型的定义

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。

串类型的定义

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）。

串类型的定义

例如，设有串A和B分别是：

A="xxf2aaa55a10a1xxf2aaa55a10a1xxx"

B="aaa55a10a1"

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4。因此，称B在A中的序号为4 。

特别地，空串是任意串的子串，任意串是其自身的子串。

串类型的定义

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。串变量和其它类型的变量一样，其值是可以改变。

串的ADT定义

ADT String{

- 数据对象：$D = \{ a_i | a_i \in CharacterSet,\ i = 1,2,\dots,n, n \geq 0 \}$
- 数据关系：$R = \{ < a_{i-1}, a_i > | a_{i-1}, a_i \in D,\ i = 2,3,\dots,n \}$
- 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 。

串的ADT定义

- 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。

串的ADT定义

- SubString (s, pos, len, sub)
- 初始条件：串s已存在, 满足

$1 \leq pos \leq StrLength(s)$ 且 $0 \leq len \leq StrLength(s) - pos + 1$ 。

- 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

串的存储表示与实现

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。

串的存储表示与实现

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。

串的定长顺序存储表示

串的顺序存储是用一组连续的存储单元来存放串中的字符序列。所谓定长顺序存储结构，是直接使用定长的字符数组来定义，数组的上界预先确定。

定长顺序存储结构定义为：

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

串的联结操作

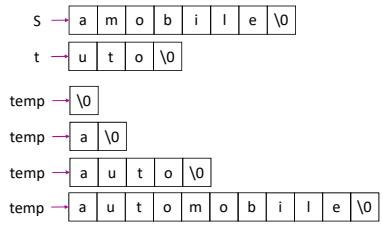
```
bool  StrConcat(StringType  &s, StringType t)
/*  将串t联结到串s之后，结果仍然保存在s中  */
{
   int i, j;
   if ((s.length + t.length)>MAX_STRLEN)
      Return 0; /*联结后长度超出范围 */
   for (i = 0; i<t.length; i++)
      s.str[s.length + i] = t.str[i];   /*  串t联结到串s之后  */
   s.length = s.length + t.length;  /* 修改联结后的串长度 */
   return 1;
}
```


求子串操作

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


插入串

在字符串s的第一位后面插入字符串t：



插入串的实现

```
//插入串
bool insertString(StringType t1, StringType t2, int i,
StringType &S)
{
   StringType temp;
   if (i<0 && i>StrLength(t1))
      return 0;
   if (!StrLength(t1))
      StrConcat(s, t2);
   else if (StrLength(t2))
   {
      SubString(t1, 1, i, &S);
      SubString(t1, i + 1,StrLength(t1) - i, &temp);
      StrConcat(S, t2);
      StrConcat(S, temp);
   }
   return 1;
}
```


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。

串的堆分配存储表示

串的堆分配存储结构定义

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

向系统请求存储空间：

```
T.ch = (char *)malloc(sizeof(char)*T.length)
```

堆存储表示串的联结实现

```
bool StrConcat(HString  &T, HString *s1, HString *s2)
/*  用T返回由s1和s2联结而成的串  */
{
    int k, j, t_len;
    if (T.ch)  free(T);     /*  释放旧空间   */
    t_len = s1->length + s2->length;
    if (!(T.ch = (char *)malloc(sizeof(char)*t_len)))
    {
        printf("系统空间不够，申请空间失败！\n");
        return 0;
    }
    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[k] = s2->ch[j];    /*  将串s2复制到串T中 */
    free(s1->ch);
    free(s2->ch);
    return 1;
}
```


串的链式存储表示

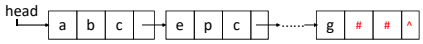
串的链式存储结构和线性表的串的链式存储结构类似，采用单链表来存储串，结点的构成是：

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。

串的链式存储表示

然而，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。如图所示，是块大小为3的串的块链式存储结构示意图。



块大小为3的串的块链式存储结构示意图

串的链式存储表示

串的块链式存储的类型定义包括：

```
//(1) 块结点的类型定义
#define CHUNK_SIZE  4
typedef  struct  Chunk
{
    char  data[CHUNK_SIZE];
    struct  Chunk *next;
}Chunk;

//(2)  块链串的类型定义
typedef  struct
{
    Chunk  head;   //头指针
    int  Strlen;   //当前长度
} Blstring;
```

在这种存储结构下，结点的分配总以完整的结点为单位，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表示串的终结。

当一个块内存放多个字符时，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。

串的模式匹配算法

模式匹配(模范匹配)：模式串在主串中的定位称为模式匹配或串匹配(字符串匹配)。模式匹配成功是指在主串S中能够找到模式串T，否则，称模式串T在主串S中不存在。

模式匹配的应用在非常广泛。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。

串的模式匹配算法

设S为主串，T为模式串，且不妨设：

S="$s_0s_1s_2...s_{n-1}$"， T="$t_0t_1t_2...t_{m-1}$"

串的匹配实际上是对合法的位置$0 \le i \le n-m$依次将主串中的子串$s[i,\dots,i+m-1]$和模式串$t[0,\dots,m-1]$进行比较：

- 若$s[i,\dots,i+m-1]=t[0,\dots,m-1]$：则称从位置$i$开始的匹配成功，亦称模式t在串s中出现；
- 若$s[i,\dots,i+m-1]\neq t[0,\dots,m-1]$：从i开始的匹配失败。位置$i$称为位移，当匹配成功时，$i$称为有效位移；反之为无效位移。

Brute-Force模式匹配算法

基本思想： 分别使用两个计数指针遍历主串S和模式串T，从S的第k个字符起和模式串T的第j=0个字符比较之，

- 若相等，则继续逐个比较后续字符。
- 否则，从主串S的下个字符k+1起再重新和模式串T的第j=0个字符比较。

依次类推，直至模式串T中的每个字符依次和S中的一个连续的字符序列相等，称匹配成功。

Brute-Force算法示例

主串S='a b a b c a b c a c b a b'

模式串T='a b c a c'

k=0

a b a b c a b c a c b a b

a b c a c

j=0

Brute-Force算法示例

主串S='a b a b c a b c a c b a b'

模式串T='a b c a c'

k=1

a b a b c a b c a c b a b

a b c a c

j=1

Brute-Force算法示例

主串S='a b a b c a b c a c b a b'

模式串T='a b c a c'

k=2

a b a b c a b c a c b a b

a b c a c

j=2

Brute-Force算法示例

主串S='a b a b c a b c a c b a b'

模式串T='a b c a c'

k=1

a b a b c a b c a c b a b

a b c a c

j=0

Brute-Force算法示例

主串S='a b a b c a b c a c b a b'

模式串T='a b c a c'

k=2

a b a b c a b c a c b a b

a b c a c

j=0

Brute-Force算法示例

主串S='a b a b c a b c a c b a b'

模式串T='a b c a c'

k=3

a b a b c a b c a c b a b

a b c a c

j=1

Brute-Force算法示例

主串S='a b a b c a b c a c b a b'

模式串T='a b c a c'

k=4

a b a b c a b c a c b a b

a b c a c

j=2

Brute-Force算法示例

主串S='a b a b c a b c a c b a b'

模式串T='a b c a c'

k=5

a b a b c a b c a c b a b

a b c a c

j=3

Brute-Force算法示例

主串S='a b a b c a b c a c b a b'

模式串T='a b c a c'

k=6

a b a b c a b c a c b a b

a b c a c

j=4

Brute-Force算法示例

主串S='a b a b c a b c a c b a b'

模式串T='a b c a c'

k=3

a b a b c a b c a c b a b

a b c a c

j=0

Brute-Force算法示例

主串S='a b a b c a b c a c b a b'

模式串T='a b c a c'

k=4

a b a b c a b c a c b a b

a b c a c

j=0

Brute-Force算法示例

主串S='a b a b c a b c a c b a b'

模式串T='a b c a c'

k=5

a b a b c a b c a c b a b

a b c a c

j=0

Brute-Force算法示例

主串S='a b a b c a b c a c b a b'

模式串T='a b c a c'

k=6

a b a b c a b c a c b a b

a b c a c

j=1

Brute-Force算法示例

主串S='a b a b c a b c a c b a b'

模式串T='a b c a c'

k=7

a b a b c a b c a c b a b

a b c a c

j=2

Brute-Force算法示例

主串S='a b a b c a b c a c b a b'

模式串T='a b c a c'

k=8

a b a b c a b c a c b a b

a b c a c

j=3

Brute-Force算法示例

主串S='a b a b c a b c a c b a b'

模式串T='a b c a c'

k=9

a b a b c a b c a c b a b

a b c a c

j=4

匹配完成。

Brute-Force模式匹配算法分析

该算法简单，易于理解。在一些场合的应用里，如文字处理中的文本编辑，其效率较高。

理解该算法的关键点

当第一次$s_k \neq t_j$时：主串要退回到$k-j+1$的位置，而模式串也要退回到第一个字符（即j=0的位置）。

比如出现$s_k \neq t_j$时：则应该有$s_{k-1} = t_{j-1}$，…，$s_{k-j+1} = t_1$，$s_{k-j} = t_0$。

该算法的时间复杂度为O(n*m)，其中n、m分别是主串和模式串的长度。

```
int IndexString(StringType  s, StringType  t, int pos)
/*   采用顺序存储方式存储主串s和模式t，若模式t在主串s中从第pos位置开始有匹配的子串，返回位置，否则返回-1 */
{
    char *p, *q;
    int  k, j;
    k = pos - 1 ;  j = 0; p = s.str + pos - 1; q = t.str;
    /*  初始匹配位置设置，顺序存放时第pos位置的下标值为pos-1*/
    while (k<s.length) && (j<t.length))
    {
        if (*p == *q)
        { p++; q++; k++; j++; }
        else
        { k = k - j + 1; j = 0; q = t.str; p = s.str + k; }
        /*  重新设置匹配位置    */
    }
    if (j == t.length)
        return(k - t.length); / *匹配，返回位置   * /
    else return(-1);      /*   不匹配，返回-1   */
}
```


Brute-Force算法改进

改进一：如果发现匹配串长度大于主串剩余串长就结束查找

改进二：先拿匹配串最后一个字符和主串对应位置字符比较，如果匹配，再依次匹配

最坏情况：$O(mn)$

Brute-Force改进算法示例

主串S='a b a b c a b c a c b a b'

模式串T='a b c a c'

start=0
endmatch=4

a b a b c a b c a c b a b

a b c a c

endmatch=4，末尾元素匹配成功

Brute-Force改进算法示例

主串S='a b a b c a b c a c b a b'

模式串T='a b c a c'

start=0
endmatch=4

i=start=0

a b a b c a b c a c b a b

a b c a c

j=0

末尾匹配，从前往后依次遍历并比较

Brute-Force改进算法示例

主串S='a b a b c a b c a c b a b'

模式串T='a b c a c'

start=0
endmatch=4

i=1

a b a b c a b c a c b a b

a b c a c

j=1

Brute-Force改进算法示例

主串S='a b a b c a b c a c b a b'

模式串T='a b c a c'

start=0
endmatch=4

i=2

a b a b c a b c a c b a b

a b c a c

j=2

不匹配，endmatch++，start++

Brute-Force改进算法示例

主串S='a b a b c a b c a c b a b'

模式串T='a b c a c'

start=1
endmatch=5

a b a b c a b c a c b a b
a b c a c

不匹配，endmatch++，start++

Brute-Force改进算法示例

主串S='a b a b c a b c a c b a b'

模式串T='a b c a c'

start=2
endmatch=6

a b a b c a b c a c b a b
a b c a c

不匹配，endmatch++，start++

Brute-Force改进算法示例

主串S='a b a b c a b c a c b a b'

模式串T='a b c a c'

start=3
endmatch=7

a b a b c a b c a c b a b
a b c a c

endmatch=7，末尾元素匹配成功

Brute-Force改进算法示例

主串S='a b a b c a b c a c b a b'

模式串T='a b c a c'

start=3
endmatch=7

i=start=3

a b a b c a b c a c b a b
a b c a c

j=0

不匹配，endmatch++，start++

Brute-Force改进算法示例

主串S='a b a b c a b c a c b a b'

模式串T='a b c a c'

start=4
endmatch=8

a b a b c a b c a c b a b
a b c a c

不匹配，endmatch++，start++

Brute-Force改进算法示例

主串S='a b a b c a b c a c b a b'

模式串T='a b c a c'

start=5
endmatch=9

a b a b c a b c a c b a b
a b c a c

endmatch=9，末尾元素匹配成功

Brute-Force改进算法示例

主串S='a b a b c a b c a c b a b'

模式串T='a b c a c'

start=5
endmatch=9

i=start=5

a b a b c a b c a c b a b

a b c a c

j=0

末尾匹配，从前往后依次遍历并比较

Brute-Force改进算法示例

主串S='a b a b c a b c a c b a b'

模式串T='a b c a c'

start=5
endmatch=9

i=6

a b a b c a b c a c b a b

a b c a c

j=1

Brute-Force改进算法示例

主串S='a b a b c a b c a c b a b'

模式串T='a b c a c'

start=5
endmatch=9

i=7

a b a b c a b c a c b a b

a b c a c

j=2

Brute-Force改进算法示例

主串S='a b a b c a b c a c b a b'

模式串T='a b c a c'

start=5
endmatch=9

i=8

a b a b c a b c a c b a b

a b c a c

j=3

模式匹配成功，返回start=5

Brute-Force模式匹配算法实现

```
int nfind(char *string, char *pat)
{
    int i, j, start = 0;
    int lasts = strlen(string) - 1;
    int lastp = strlen(pat) - 1;
    int endmatch = lastp;
    //先用endmatch做匹配。剩余串长度小于模式串长度时退出
    for (i = 0; endmatch <= lasts; endmatch++, start++) {
        if (string[endmatch] == pat[lastp])
            for (j = 0, i = start; i<lastp && string[i] == pat[j];
    i++, j++)
                ;
        if (j == lastp)
            return start; //成功找到匹配的串
    }
    return -1;
}
```


模式匹配的KMP算法

该算法是由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，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”尽可能远的一段距离后，继续进行比较。

KMP算法

i=2

a b a b c a b c a c b a b

a b c a c

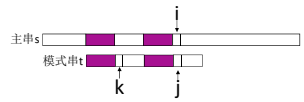
j=2

在i=2和j=2时，匹配失败。重新开始第二次匹配时，不必从i=1，j=0开始。因为$s_1 = t_1$，$t_0 \neq t_1$，必有$s_1 \neq t_0$，由此可知，第二次匹配可以直接从i=2、j=0开始。

在主串s与模式串t的匹配过程中，一旦出现$s_i \neq t_j$，主串s的指针不必回溯，而是直接与模式串的$t_k(0 \leq k < j)$进行比较，而k的取值与主串s无关，只与模式串t本身的构成有关，即从模式串t可求得k值。)

KMP算法

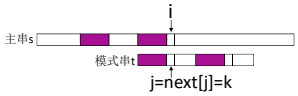
定义$next[j] = k$，表示当模式中第j个字符与主串中的相应字符失配时，在模式中需要重新和主串中该字符进行比较的字符的位置。



紫色部分为模式串j之前的字串中的最长相同真前缀、真后缀，在j处和主串i处的字符失配时只需要从k处继续与主串i处字符的进行比较。

KMP算法

定义$next[j] = k$，表示当模式中第j个字符与主串中的相应字符失配时，在模式中需要重新和主串中该字符进行比较的字符的位置。



j=next[j]，主串i位置不变继续进行比较

Next数组的求解

next[j]的求法为：求模式串t中j之前的子串的最长相同真前缀、真后缀的长度。

对模式串t='a b c a c'

j值	j之前的子串	真前缀	真后缀	最长相同前后缀长度
j= 0	-	-	-	-1
j= 1	a	空	空	0
j= 2	ab	a	b	0
j= 3	abc	a,ab	c,bc	0
j= 4	abca	a,ab,abc	a,ca,bca	1

Next数组的求解同样可以作为一个KMP问题来解决。

KMP算法

在已知next[j]值时，KMP算法的思想是：

设主串为s，模式串为t，并设i指针和j指针分别指示主串和模式串中正待比较的字符，设i和j的初值均为0。

- 若有$s_i = t_j$或者j =-1，则i和j分别加1。
- 否则，i不变，j退回到$j = next[j]$的位置，重新进行比较判断。

直至匹配完成。

先使用KMP算法求出模式串t的next值，再进行主串s和模式串t的模式匹配。

Next数组的求解示例

模式串T='a b c a c'

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

j=1

a b c a c 后缀字符串

a b c a c 前缀字符串

k=0

不匹配，转到k=next[k]=-1

Next数组的求解示例

模式串T='a b c a c'

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

j=1

a b c a c 后缀字符串

a b c a c 前缀字符串

k=-1

k=-1时j++,k++,next[j]=k,有next[2]=0

Next数组的求解示例

模式串T='a b c a c'

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

j=2

a b c a c 后缀字符串

a b c a c 前缀字符串

k=0

不匹配，转到k=next[k]=-1

Next数组的求解示例

模式串T='a b c a c'

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

j=2

a b c a c 后缀字符串

a b c a c 前缀字符串

k=-1

k=-1时j++,k++,next[j]=k,有next[3]=0

Next数组的求解示例

模式串T='a b c a c'

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

j=3

a b c a c 后缀字符串

a b c a c 前缀字符串

k=0

成功匹配，j++，k++，next[j]=k=1

KMP算法示例

主串S='a b a b c a b c a c b a b'

模式串T='a b c a c'

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

i=0

a b a b c a b c a c b a b

a b c a c

j=0

匹配成功，i++;j++

KMP算法示例

主串S='a b a b c a b c a c b a b'

模式串T='a b c a c'

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

i=1

a b a b c a b c a c b a b

a b c a c

j=1

匹配成功，i++;j++

KMP算法示例

主串S='a b a b c a b c a c b a b'

模式串T='a b c a c'

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

i=2

a b a b c a b c a c b a b

a b c a c

j=2

匹配失败，j=next[j]=0

KMP算法示例

主串S='a b a b c a b c a c b a b'

模式串T='a b c a c'

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

i=2

a b a b c a b c a c b a b

a b c a c

j=0

匹配成功，i++;j++

KMP算法示例

主串S='a b a b c a b c a c b a b'

模式串T='a b c a c'

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

i=3

a b a b c a b c a c b a b

a b c a c

j=1

匹配成功，i++;j++

KMP算法示例

主串S='a b a b c a b c a c b a b'

模式串T='a b c a c'

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

i=4

a b a b c a b c a c b a b

a b c a c

j=2

匹配成功，i++;j++

KMP算法示例

主串S='a b a b c a b c a c b a b'

模式串T='a b c a c'

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

i=5

a b a b c a b c a c b a b

a b c a c

j=3

匹配成功，i++;j++

KMP算法示例

主串S='a b a b c a b c a c b a b'

模式串T='a b c a c'

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

i=6

a b a b c a b c a c b a b

a b c a c

j=4

匹配失败，j=next[j]=1

KMP算法示例

主串S='a b a b c a b c a c b a b'

模式串T='a b c a c'

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

i=6

a b a b c a b c a c b a b

a b c a c

j=1

匹配成功，i++;j++

KMP算法示例

主串S='a b a b c a b c a c b a b'

模式串T='a b c a c'

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

i=7

a b a b c a b c a c b a b

a b c a c

j=2

匹配成功，i++;j++

KMP算法示例

主串S='a b a b c a b c a c b a b'

模式串T='a b c a c'

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

i=8

a b a b c a b c a c b a b

a b c a c

j=3

匹配成功，i++;j++

KMP算法示例

主串S='a b a b c a b c a c b a b'

模式串T='a b c a c'

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

i=9

a b a b c a b c a c b a b

a b c a c

j=4

匹配完成。

KMP算法实现

```
//求解Next数组，同样是使用KMP算法，寻找模式串t中位置j前的字串的前缀、后缀
的最大匹配长度
void cal_next(char *str, int *next, int len) {
    next[0] = -1;next[1] = 0;
    int k = 0;
    int j = 1;
    while (j <= len - 1) {
    //str[k]表示前缀，str[j]表示后缀
    if (k == -1 || str[j] == str[k]) {
        ++k;
        ++j;
        next[j] = k;
        }
    else
        k = next[k]; //往前回溯
    }
}
```


KMP算法实现

```
#define Max_Strlen 1024
int next[Max_Strlen];
int KMP_index(StringType  s, StringType  t)
/* 用KMP算法进行模式匹配，匹配返回位置，否则返回-1 */
/*用静态存储方式保存字符串， s和t分别表示主串和模式串
*/
{
   int  i = 0, j = 0;      /*初始匹配位置设置 */
   while (i<s.length) && (j<t.length)
   {
      if ((j == -1) || (s.str[i] == t.str[j]))
         { i++; j++; }
      else j = next[j];
   }
   if (j >= t.length)  return(i - t.length);
   else  return(-1);
}
```

Next数组的求解

next[j]的求法为：求模式串t中j之前的子串的最长相同真前缀、真后缀的长度。

对模式串t='a b c a b a'

j值	j之前的子串	真前缀	真后缀	最长相同前后缀长度
j= 0	-	-	-	-1
j= 1	a	空	空	0
j= 2	ab	a	b	0
j= 3	abc	a,ab	c,bc	0
j= 4	abca	a,ab,abc	a,ca,bca	1
j= 5	abcab	a,ab,abc,abca	a,ca,bca,bcab	2

Next数组的求解同样可以作为一个KMP问题来解决。

KMP算法

在已知next[j]值时，KMP算法的思想是：

设主串为s，模式串为t，并设i指针和j指针分别指示主串和模式串中正待比较的字符，设i和j的初值均为0。

- 若有$s_i = t_j$或者j =-1，则i和j分别加1。
- 否则，i不变，j退回到$j = next[j]$的位置，重新进行比较判断。

直至匹配完成。

先使用KMP算法求出模式串t的next值，再进行主串s和模式串t的模式匹配。

Next数组的求解示例

模式串T='a b c a b a'

j	0	1	2	3	4	5
字符	a	b	c	a	b	a
next[j]	-1	0				

j=1

a b c a b a 后缀字符串

a b c a b a 前缀字符串

k=0

不匹配，转到k=next[k]=-1

Next数组的求解示例

模式串T='a b c a b a'

j	0	1	2	3	4	5
字符	a	b	c	a	b	a
next[j]	-1	0	0			

j=1

a b c a b a 后缀字符串

a b c a b a 前缀字符串

k=-1

k=-1时j++,k++,next[j]=k,有next[2]=0

Next数组的求解示例

模式串T='a b c a b a'

j	0	1	2	3	4	5
字符	a	b	c	a	b	a
next[j]	-1	0	0			

j=2

a b c a b a 后缀字符串

a b c a b a 前缀字符串

k=0

不匹配，转到k=next[k]=-1

Next数组的求解示例

模式串T='a b c a b a'

j	0	1	2	3	4	5
字符	a	b	c	a	b	a
next[j]	-1	0	0	0		

j=2

a b c a b a 后缀字符串

a b c a b a 前缀字符串

k=-1

k=-1时j++,k++,next[j]=k,有next[3]=0

Next数组的求解示例

模式串T='a b c a b a'

j	0	1	2	3	4	5
字符	a	b	c	a	b	a
next[j]	-1	0	0	0	1	

j=3

a b c a b a 后缀字符串

a b c a b a 前缀字符串

k=0

成功匹配，j++，k++，next[j]=k=1

Next数组的求解示例

模式串T='a b c a b a'

j	0	1	2	3	4	5
字符	a	b	c	a	b	a
next[j]	-1	0	0	0	1	2

j=3

a b c a b a 后缀字符串

a b c a b a 前缀字符串

k=0

成功匹配，j++，k++，next[j]=k=2