串

（1）掌握串的特性和存储设计。

（2）掌握串的操作方法。

（3）掌握串匹配的BF算法、KMP算法中next[]函数值的计算方法。

串(String)

定义

- **串**：是由n(n≥0)个字符组成的有限序列。一般记为：S="$a_1a_2...a_n$"
  - S是串名
  - a_i（1≤i≤n）可以是英文字母、数字字符或其他字符，其值均取自于某个字符集
  - 串中字符的个数n称为串的长度；n=0时是空串。
- **子串**：串中任意个连续的字符组成的子序列称为该串的子串
  - 空串是任何串的子串.
  - 一个串也可以看成是自身的子串
  - 子串个数公式：**n(n+1)/2+1**
- **真子串**：除本身之外的其它子串，真子串个数公式：**n(n+1)/2**
- **主串**：包含子串的串称为主串。
- **字符的定位**：一个字符在串中的序号称为该子符在串中的位置。例如，S="$a_1a_2...a_n$"中，则字符a_3的位置是3。
- **子串的定位**：子串在主串中的位置指的是该子串的第一个字符在主串中第一次出现的位置。例如：S="$a_1a_2...a_n$"中，则子串$a_3a_4...a_n$的位置是3。

串的比较

- ASCII码由8bit组成一个字符，共可形成2^8=256个字符（仅英语）
- Unicode码由16bit组成一个字符，共可表示2^{16}=65536个字符（全世界的字符，前256个字符不动，高位用0补充）
- 给定两个串： X="$x_1x_2...x_n$"，Y="$y_1y_2...y_m$"，则：当n=m 且 $x_1=y_1$,$x_2=y_2$, ..., $x_n= y_m$时， 称X=Y。
- 当下列条件之一成立时，称X<Y。
  - n<m，且$x_i=y_i$（i=1,2, ...,n）
  - 存在某个k≤min(m,n)，使得xi=yi（i=1,2, ...,k-1），$x_k<y_k$
  - 例如，有下面一些串：
    - S1="child";
    - S2="chalde";
    - S3="student";
    - S4="student ";
    - S5="c";
    - S6="ABCDEF";
    - S7="ABCEF"
    - S8="cnt"

- S9="cnt asf"
- 则有: **S1>S2，S3<S4, S5<S9, S5<S8, S6<S7**

空格串与空串

- 由一个或多个空格所组成的串称为**空格串**，空格串的串长不为0。例如：S = " "。
- 完全没有内容的串称为**空串**，空串的串长为0。例如： S == NULL

注意事项

- 串是一种线性表。
- 串和线性表的不同点：
  - 串的数据元素是字符，即每个数据元素都是**一个**字符。
  - 线性表的操作主要是针对某个数据元素进行的，而串的操作则主要是针对串的**整体或某一部分子串**进行的 。

抽象数据类型ADT

```
ADT String{
     Data:
         串中的数据元素仅由一个字符组成，相邻元素具有前驱和后继的关系。
    Operation:
         StrAssign(&S, chars)
                  初始条件：chars是字符串常量。
                  操作结果：把chars赋为串S的值。
         StrCopy(&S,T)
                  初始条件：串T存在。
                  操作结果：由串T复制得串S。
         StrEmpty(S)
                  初始条件：串S存在。
                  操作结果：若S为空串，则返回true；否则返回false。
         StrLength(S)
                  初始条件：串S存在。
                  操作结果：返回S的元素个数，即串的长度。
         StrCompare(S, T)
                  初始条件：串S和T存在。
                  操作结果：若S<T，则返回值<0；若S=T，则返回值=0；若S>T，则返回值>0。
         StrConcat(&S, T)
                  初始条件：串S和T存在。
                  操作结果：用S返回由S和T联接而成的新串。
         SubString(S, &Sub, pos, len)
                  初始条件：串S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len的字串。
         Index(S, T, &pos)
                  初始条件：串S和T存在，且T是非空串，1≤pos≤StrLength(S)。
                  操作结果：若主串S中存在和串T值相同的子串，则由pos返回它在主串S中第一次出现的
位置，函数值为true；否则函数值为false。
         StrInsert(&S, pos, T)
                  初始条件：串S和T存在，1≤pos≤StrLength(S)+1。
                  操作结果：在串S的第pos个字符之前插入串T。
         StrDelete(&S, pos, len)
                  初始条件：串S存在，1≤pos≤StrLength(S)-len+1。
```

```
                操作结果：从串S中删除第pos个字符起长度为len的子串。
        Replace(&S，T，V)
                初始条件：串S，T和V存在，T是非空串。
                操作结果：用V替换主串S中出现的所有与T相等的不重叠的子串。
        StrTraveres_Sq(S)
                初始条件：串S已存在。
                操作结果：依次输出串S中的每个字符。
        DestroyString(&S)
                初始条件：串S存在。
                操作结果：串S被撤销。
}ADT String。
```

最小操作子集

- 一般来说，串赋值（StrAssign），串拷贝（StrCopy），串比较（StrCompare），求串长（StrLength），串联接（StrConcat），求子串（SubString）**等操作**可以构成最小操作子集，即这些操作不能用其他串操作来实现。反之，其他串操作（除串撤销DestroyString外）均可在这个最小操作子集上实现。
- 例如：串置换（Replace）操作就可通过调用串的定位操作、串插入操作和串删除操作来实现。
- 想想插入串，删除串以及复制串操作可以用以下哪几种操作组合实现？

序号	基本操作举例	C语言函数举例
1	StrAssign(s1,s2)	strcpy(s1,s2)
2	StrLength(s1)	strlen(s1)
3	StrConcat(s1,s2)	strcmp(s1,s2)
4	StrAssign(s1,s2)	strcat(s1,s2)
5	Index(s1,s2)	strstr(s1,s2)

顺序串

定义

- **顺序串**就是串以顺序存储结构存储
- **顺序串**可用高级语言的字符数组来实现
- **顺序串**按其存储分配的不同可分为**静态存储分配的顺序串**和**动态存储分配的顺序串。**

顺序串的数据结构：静态存储分配的顺序串

- 静态存储分配的顺序串，都是用定长字符数组存储串值，串的操作比较简单，但由于串值空间的大小已经确定，所以对串的某些操作，如插入、连接、置换等带来不便。方法有二：

1. 直接使用定长的字符数组来定义：

```
typedef char SqString[MaxStrSize + 1];
```

2. 类似静态顺序表的定义：

```
typedef struct{
        char str[MaxStrSize];  //顺序串的最大容量
        int length;            //顺序串的当前长度
     }SSqString;               //静态顺序串类型
```

顺序串的数据结构：动态存储分配的顺序串

- 由于串变量之间的长度相差较大，因此静态分配固定空间大小不合理，宜采用动态存储进行创建。
- 动态存储分配的顺序串，串值空间的大小是在程序执行时动态分配而得，这对串的插入、连接、置换等操作非常有利 ，因此在串处理的应用程序中也常被选用。

```
typedef struct {
    char  *str;                    // 先存放非空串的首地址，不分配内存
    int length;                    // 存放串的当前长度
}DSqString;                        //待到程序执行时，再根据插入、删除等操作动态增补空间。
```

注意事项

- 动态分配的顺序串完全可用动态存储分配的顺序表SqList来表示。
- 动态存储分配的顺序串具有存储结构的优点（随机存取，操作简单）

串赋值

把一个字符串常量赋值给顺序串S,成功返回true,否则返回FALSE，赋值前需要判空。

```
[In DSqString.h]
bool StrAssign_Sq(DSqString &S,   char *chars)
{
     int i,j;
     char *c;
    for(i=0,c=chars; *c;   i++,c++);   // c和chars的长度一样了
    if(i==0) { S.str = NULL;    S.length=0; }  //S是空串
    else {//如果S不是空串
        if(!(S.str=(char *)malloc(i*sizeof(char))))  //申请空间
                 {return false;}       //空间申请失败
        for(j=0;j<i;j++)           //否则，一个一个装填
        S.str[j]=chars[j];
        S.length=i;
    }
    return true;
}// StrAssign_Sq
```

串赋值

把一个顺序串T复制到另一个顺序串S中，使得S和T具有一样的串值。成功返回true,否则返回false。需要判空

```
[In DSqString.h]
bool StrCopy_Sq(DSqString &S,DSqString T)
{   // 将顺序串T复制到另一个顺序串S中，并返回复制后的顺序串S
    int i;
    if(S.str) free(S.str);                          // 释放S原有空间
```

```
    if(!T.length) { S.str=NULL;  S.length=0;  }   // S置为空串
    else {
        if(!(S.str=(char *)malloc(T.length*sizeof(char)))) // 给串S申请空间
            return false;
        for(i=0;i<T.length;i++)                  // 将串T中的字符复制到串S中
            S.str[i]=T.str[i];
        S.length=T.length;                       // 串S的串长为T.length
    }
    return true;
}// StrCopy_Sq
```

求串长

```
[In DSqString.h]
int StrLength_Sq(DSqString S)
{
    return(S.length);
}// StrLength_Sq
```

串比较

比较顺序串S和T，若S<T,则返回值<0;若S=T,则返回值=0;若S>T，则返回值>0.

```
[In DSqString.h]
int StrCompare_Sq(DSqString S,DSqString T)
{
    int i=0;
    while(i<S.length&&i<T.length)            // 串S和串T对应字符进行比较
    {
        if(S.str[i]>T.str[i]) return 1;
        else if(S.str[i]<T.str[i]) return -1;
        i++;
    }
    if(i<S.length) return 1;
    else if(i<T.length) return -1;
    return 0;
}// StrCompare_Sq
```

串连接

将顺序串T连接在顺序串S之后，需要判空

```
[In DSqString.h]
bool StrConcat_Sq(DSqString &S,DSqString T)
{
   int i;
   if(T.length) {
    if(!(S.str=(char *)realloc(S.str,(S.length+T.length)*sizeof(char))))  // 给串
增补空间
            return false;
        for(i=0;i<T.length;i++)            // 将串T中的字符连接在串S的后面
         S.str[S.length+i]=T.str[i];
```

```
        S.length+=T.length;                 // 串S的串长增加T.length
     }
    return true;
}// StrConcat_Sq
```

取子串

在顺序串S中从第POS个位置开始，取长度为len的子串sub。

```
[In DSqString.h]
bool SubString_Sq(DSqString S,DSqString &Sub,int pos,int len)
{
    int i;
    if(pos<0||pos>S.length-1||len<0||len>S.length-pos)
        return false;                   // 取子串的位置或子串的长度不合理
    if(Sub.str)   free(Sub.str);          // 释放Sub原有空间
    if(!len) { Sub.str=NULL; Sub.length=0; }   // 置Sub为空子串
    else {
        if(!(Sub.str=(char *)malloc(len*sizeof(char))))
            return false;
        for(i=0;i<len;i++)              // 将串S中的len个字符复制到Sub中
            Sub.str[i]=S.str[pos+i];
        Sub.length=len;                 // 子串Sub的串长为len
    }
    return true;
}// SubString_Sq
```

子串定位

串的模式匹配，即在主串S中查找是否存在和串T值相同的子串，若存在则返回true,不存在则返回False.

```
[In DSqString.h]
bool Index_Sq(DSqString S,DSqString T,int i,int &pos)
{
   int j=0;                          // i和j分别扫描主串S和子串T
   while(i<S.length&&j<T.length)
   { if(S.str[i]==T.str[j])          // 对应字符相同，继续比较下一个字符
      { i++;    j++;
      }
      else                           // 主串指针回溯重新开始下一次匹配
      { i=i-j+1;
        j=0;
      }
   }
   if(j==T.length)  { pos=i-T.length; return true;  }
   else return false;
}// Index_Sq
```

插入子串

在顺序串S的第pos个字符之前插入子串T。

```
[In DSqString.h]
bool StrInsert_Sq(DSqString &S,int pos,DSqString T)
    {   // 在顺序串S的第pos个字符之前插入子串T，并返回插入后的顺序串S
     int i;
     if(pos<0||pos>S.length)  return false;     // pos不合理
     if(T.str) {
       if(!(S.str=(char *)realloc(S.str,(S.length+T.length)*sizeof(char))))   //
给串S增补空间
            return false;
       for(i=S.length-1;i>=pos;i--)            // 为插入串T而腾出位置
        S.str[i+T.length]=S.str[i];
       for(i=0;i<T.length;i++)                  // 插入串T
        S.str[pos+i]=T.str[i];
       S.length=S.length+T.length;             // 串S长度增加T.length
      }
    return true;
}// StrInsert_Sq
```

删除子串

从顺序串S的第pos个字符字符开始删除长度为len的子串，并返回删除后的串S

```
[In DSqString.h]
bool StrDelete_Sq(DSqString &S,int pos,int len)
    {
      int i;
      if(pos<0||pos>S.length-1||len<0||pos+len>S.length)  return false;  // pos
和len不合理
      for(i=pos+len;i<S.length;i++)                              // 元素前移，删除子
串
        S.str[i-len]=S.str[i];
      S.str=(char *)realloc(S.str,(S.length-len)*sizeof(char)); // 串S空间减少len
      S.length=S.length-len;                                     // 串S长度减少len
      return true;
}// StrDelete_Sq
```

置换子串

用串V置换主串S中出现的所有与T相等的不重叠子串，并返回置换后的串S

```
[In DSqString.h]
void StrReplace_Sq(DSqString &S, DSqString T,DSqString V)
{
      int  i=0,pos;
      while(Index_Sq(S,T,i,pos))                    // 判断T是否是S的子串
      { StrDelete_Sq(S,pos,T.length);              // 删除子串T
        StrInsert_Sq(S,pos,V);                     // 插入子串V
        i=i+StrLength_Sq(V);
      }
}// StrReplace_Sq
```

串的遍历

用串V置换主串S中出现的所有与T相等的不重叠子串，并返回置换后的串S

```
[In DSqString.h]
void StrTraveres_Sq(DSqString S)
{
    int i;
    for(i=0;i<S.length;i++)
     cout<<S.str[i];
    cout<<endl;
}// StrTraveres_Sq
```

撤销串

释放顺序串S所占的存储空间，顺序串S被撤销。

```
void DestroyString_Sq(DSqString &S)
{
       free(S.str);
       S.str=NULL;         // S.str置空
       S.length=0;
}// DestroyString_Sq
```

链式串

定义

- 串的链式存储结构(一个节点可以存放多个字符)
- 通常把结点大小为1的链式串称为**单链结构**，把结点大小大于1的链式串称为**块链结构**。
- 在块链结构中最后一个结点通常填不满，则用#号把串值填满。

逻辑图示

单链数据结构

用单链结构表示串，其特点是处理灵活，但空间浪费较大，因为串中指针域占有较多的空间。

```
[In SLinkString.h]
typedef struct LNode {
    char str;
    struct LNode *next;
}SNode,*SLinkString;
```

块链数据结构

```
[In SLinkString.h]
typedef struct Chunk{       //可由用户定义的节点大小
      char str[Number]; //一个节点存放Number个字符
      struct Chunk  *next;
}Chunk;                 //定义结点类型
typedef struct{
   Chunk  *head, *tail;     //串的头尾指针
   int length;                 //串的当前长度
}
```

串赋值

将字符串chars赋值给链式串S

```
[In SLinkString.h]
void StrAssign_L(SLinkString &S, char * chars) {
    SLinkString p, q;
    S = (SLinkString)malloc(sizeof(LNode));
    if (!(*chars)) S->next = NULL;
    else {
        p = S;
        while (*chars) {
            q = (SLinkString)malloc(sizeof(LNode));
            q->str = *chars;
            p->next = q;
            p = q;
            chars++;
        }
        q->next = NULL;
    }
}
```

串复制

将链式串T复制到S中

```
[In SLinkString.h]
void StrCopy_L(SLinkString &S, SLinkString T) {
    SLinkString p, q, r;
    while (S->next) {
        p = S;
        S = S->next;
        free(p);
    }
    r = T->next;
    p = S;
    while (r) {
        q = (SLinkString)malloc(sizeof(LNode));
        if (!p) {
            cout << "错误" << endl;
            exit(0);
        }
        q->str = r->str;
        p->next = q;
```

```
        p = q;
        r = r->next;
    }
    p->next = NULL;
}
```

求串长

```
[In SLinkString.h]
int StrLength_L(SLinkString S) {
    int n = 0;
    SLinkString p = S->next;    //传的是指针
    while (p) {
        n++;
        p = p->next;
    }
    return n;
}
```

串比较

```
[In SLinkString.h]
int StrCompare_L(SLinkString S, SLinkString T) {
    SLinkString p = S->next, q = T->next;
    while (p&&q) {
        if (p->str > q->str) return 1;
        else if (p->str < q->str) return -1;
        p = p->next;
        q = q->next;
    }
    if (p) return 1;
    else if (q) return -1;
    return 0;
}
```

串连接

判断链栈S是否为空，若为空则返回true,否则返回false

```
[In SLinkString.h]
bool StrConcat_L(SLinkString &S, SLinkString T) {
    SLinkString p, q = S->next, r = T->next;
    while (q->next) q = q->next;
    while (r) {
        p = (SLinkString)malloc(sizeof(LNode));
        if (!p) return false;
        p->str = r->str;
        q->next = p;
        q = p;
        r = r->next;
```

```
    }
    q->next = NULL;
    return true;
}
```

取子串

释放链栈所占空间

```
[In SLinkString.h]
bool SubString_L(SLinkString S, SLinkString &Sub, int pos, int len) {
    SLinkString p, q, r;
    int i;
    if (len<0 || len>StrLength_L(S) - pos + 1) return false; //len值不合理
    p = S->next; i = 1;
    while (p&&i < pos) {
        p = p->next;
        i++;
    }
    if (i != pos) return false; //pos位置不合理
    while (Sub->next) {
        q = Sub;
        Sub = Sub->next;
        free(q);
    }
    r = Sub;
    for (i = 1; i <= len; i++) {
        q = (SLinkString)malloc(sizeof(LNode));
        q->str = p->str;
        r->next = q;
        r = q;
        p = p->next;
    }
    r->next = NULL;
    return true;
}
```

子串定位

```
[In SLinkString.h]
bool Index_L(SLinkString S, SLinkString T, int &pos) {
    int i;
    SLinkString Sub;
    StrAssign_L(Sub, "");   //初始化
    for (i = 1; i < StrLength_L(S) - StrLength_L(T); i++) {
        SubString_L(S, Sub, i, StrLength_L(T));
        //cout << Sub->next->str << endl;
        if (!StrCompare_L(Sub, T)) {
            //cout << Sub->next->str << endl;
            pos = i;
            //cout << i << endl;
            return true;
        }
    }
```

```
    //cout << "fail" << endl;
    return false;
}
```

插入子串

```
[In SLinkString.h]
bool StrInsert_L(SLinkString &S, int pos, SLinkString T) {
    SLinkString p, q, r, h;
    int i = 0;
    p = S;
    while (p&&i < pos-1) {
        p = p->next;
        i++;
    }
    q = p->next;
    if (i != pos-1) return false;
    r = T->next;
    while (r) {
        h = (SLinkString)malloc(sizeof(LNode));
        h->str = r->str;
        p->next = h; h->next = q;
        p = h; r = r->next;
    }
    return true;
}
```

删除子串

从链式串的第pos个字符串开始删除

```
[In SLinkString.h]
bool StrDelet_L(SLinkString &S, int pos, int len) {
    SLinkString p = S, r, q;
    if (len < 0 || StrLength_L(S) - pos + 1 < len) return false;
    int i = 0;
    while (p&&i < pos-1) {
        p = p->next;
        i++;
    }
    if (i != pos-1) return false;
    q = p->next;
    for (i = 1; i <= len; i++) {
        r = q;
        p->next = q->next;
        q = q->next;
        free(r);
    }
    return true;
}
```

置换子串

```
 [In SLinkString.h]
void StrReplace_L(SLinkString &S, SLinkString T, SLinkString V) {
    int pos = 1;
    while (Index_L(S, T, pos)) {
        StrDelet_L(S, pos, StrLength_L(T));
        StrInsert_L(S, pos, V);
        //pos=pos+StrLength_L(V);
    }
}
```

串的遍历

```
 [In SLinkString.h]
void StrTraveres_L(SLinkString S) {
    SLinkString p = S->next;
    while (p) {
        cout << p->str;
        p = p->next;
    }
    cout << endl;
}
```

串的遍历

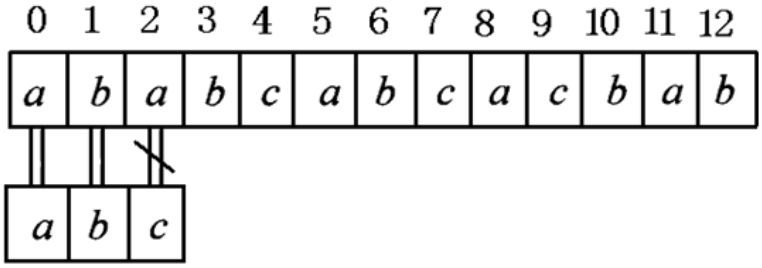
```
 [In SLinkString.h]
void StrTraveres_L(SLinkString S) {
    SLinkString p = S->next;
    while (p) {
        cout << p->str;
        p = p->next;
    }
    cout << endl;
}
```

串的撤销

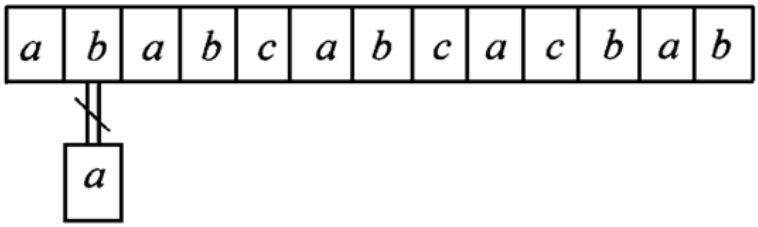
```
[In SLinkString.h]
void DestroyString_L(SLinkString &S) {
    SLinkString p, p1;
    p = S;
    while (p)
    {
        p1 = p;
        p = p->next;
        free(p1);
    }
    S = NULL;
}
```

BF算法

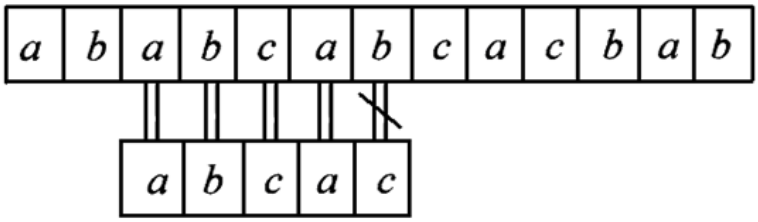
第1趟匹配，i=2，j=2失败
i回溯到1，j回溯到0



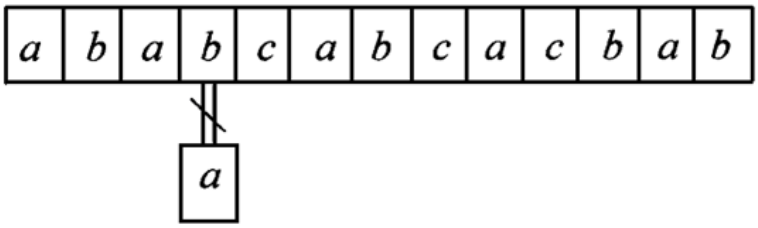
第2趟匹配，i=1，j=0失败
i回溯到2，j回溯到0



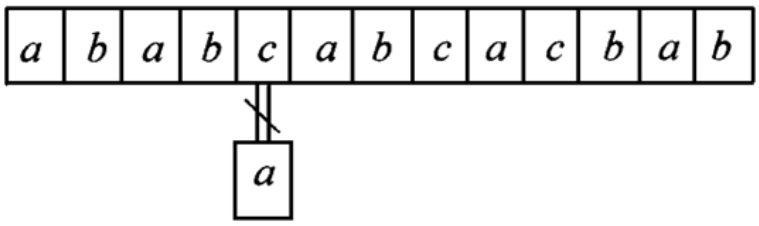
第3趟匹配，i=6，j=4失败
i回溯到3，j回溯到0



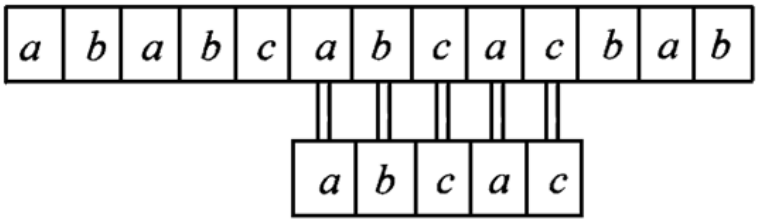
第4趟匹配，i=3，j=0失败
i回溯到4，j回溯到0



第5趟匹配，i=4，j=0失败
i回溯到5，j回溯到0



第6趟匹配，i=10，j=5
T中全部字符都比较完毕
匹配成功



2. 算法分析

设串S长度为n，串T长度为m，在匹配成功的情况下：

⑴最好情况：每趟不成功的匹配都发生在串T的第一个字符。设匹配成功发生在s_i处，则在i-1趟不成功的匹配中共比较了i-1次，第i趟成功的匹配共比较了m次，所以总共比较了i-1+m次，所有匹配成功的可能共有n-m+1种，设s_i从开始与串T匹配成功的概率为p_i，在等概率情况下，平均的比较次数是：

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

⑵最坏情况：每趟不成功的匹配都发生在串T的最后一个字符。设匹配成功发生在s_i处，则i-1趟不成功的匹配共比较了(i-1)×m次，第i趟成功的匹配共比较了m次，所以总共比较了i×m次，由此平均比较的次数是：

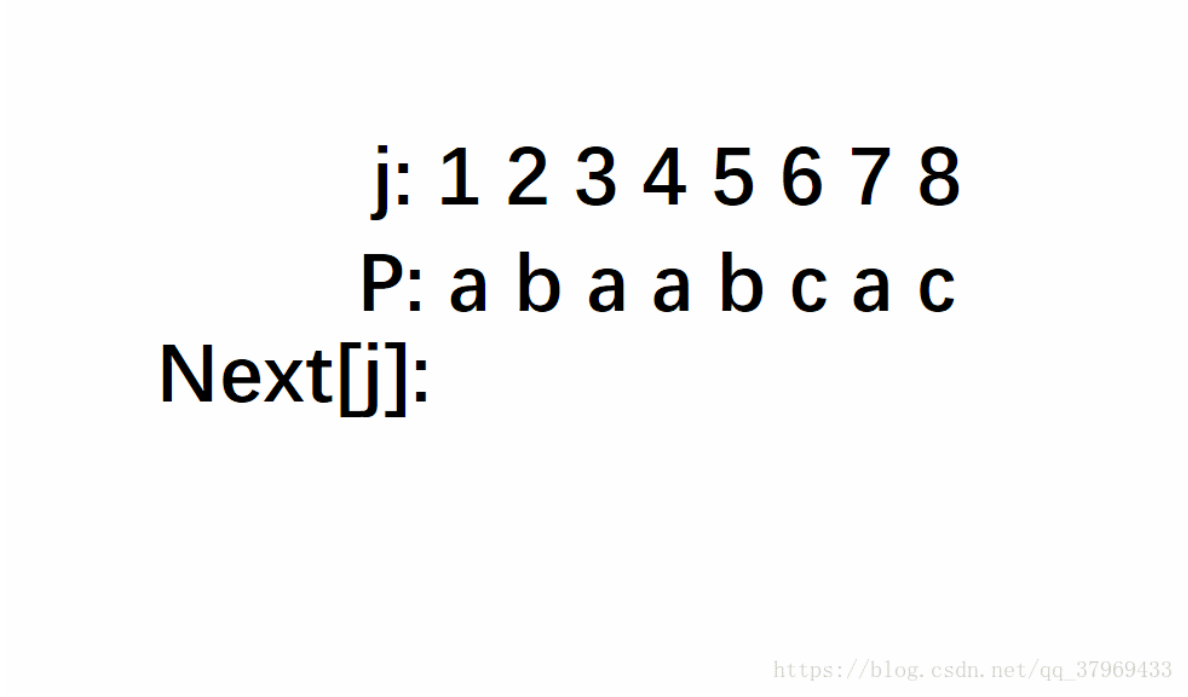
$$\sum_{i=1}^{n-m+1} p_i \times (i \times m) = \sum_{i=1}^{n-m+1} \frac{1}{n-m+1} \times (i \times m) = \frac{m \times (n-m+2)}{2}$$

因此，该算法在最好情况下的时间复杂度为O(n+m)，在最坏情况下的时间复杂度为O(n×m)。由此可知，这个算法虽然简单，易于理解，但效率不高，其主要原因是：主串指针i在若干个字符序列比较相等后只要有一个字符比较不等便需回溯。

KMP算法

- 不需回溯i指针
- 而是利用已经得到的"部分匹配"的结果将模式串向右"滑动尽可能远的一段距离"后，继续做比较
- 使得主串中的每个字符只参加一次比较。

$$next[j] = \begin{cases} 0 & j=1 \\ 1 & \text{首尾重合数为0（不匹配时）} \\ Max\{k \mid 1<k<j \text{ 且 } 'p_1 \ldots p_{k-1}' = 'p_{j-k+1} \ldots p_{j-1}'\} \end{cases}$$




```
# include "stdlib.h"          // 该文件包含malloc()、realloc()和free()等函数
# include "iostream.h"        // 该文件包含标准输入输出流cout和cin
# include "DSqString.h"       // 该文件中包含动态顺序串的相关定义及操作

void get_next(DSqString T,int next[])
{
    int  j=1,k=0;
    next[0]=-1;
    next[1]=0;
    while(j<T.length)
    {
        if(T.str[j]==T.str[k])
        {
            next[j+1]=k+1;
            j++;  k++;
        }
        else if(k==0)
        {
            next[j+1]=0;
            j++;
        }
        else k=next[k];
    }
}// get_next

bool Index_KMP(DSqString S,DSqString T,int next[],int &pos)
{    // 利用模式串T的next函数求T在主串S中的位置
   int i=0,j=0;                          // i和j分别扫描主串S和子串T
   while(i<S.length&&j<T.length)
   {  if(j==-1||S.str[i]==T.str[j])    // 继续比较下一个字符
      {  i++;
  j++;
      }
      else  j=next[j];                    // 模式串向右移动
   }
   if(j==T.length)  { pos=i-T.length; return true;  }
   else return false;
```

```
}// Index_KMP
```