第三章　串

- 串的定义
- 串的逻辑结构和基本操作
- 串的存储结构
- 串的实现
- 串的模式匹配算法
- 进阶导读
- 本章小结

3.1 串的定义

- 假设V是程序设计语言所采用的字符集，由字符集V上的字符所组成的任何有限序列，称为字符串（或简称为串）：

$$s=\text{“}a_1a_2\ldots a_n\text{”}\quad (n\geq 0)$$

其中：

  - s是串的名；
  - 两个双引号之间的字符序列“$a_1a_2\ldots a_n$”是串的值；
  - $a_i\in V\ (1\leq i\leq n)$是字符集上的字符。
- 串中字符的数目n称为串的长度。长度为0的串称为空串。
- 一个串的子串是这个串中的任一连续子序列。包含子串的串相应地称为主串。
- 通常称字符在序列中出现的序号为该字符在串中的位置。相应地，子串在主串中的位置则以该子串的第一个字符在主串中的位置来表示。

3.2 串的逻辑结构和基本操作

- 在逻辑结构方面，串与线性表极为相似，区别在于串的数据对象约束为字符集。

- 字符串的逻辑表示：
  - 数据对象：$D=\{a_i|a_i\in$字符集$, i=1, 2, \ldots, n, n\geq 0\}$
  - 数据关系：$R=\{<a_{j-1}, a_j>|(a_{j-1}\in D)\wedge(a_j\in D), 2\leq j\leq n\}$

- 在基本操作方面，串与线性表差别很大。
  - 线性表：大多以“单个元素”为操作对象，如：在线性表中查找某个元素、在某个位置上插入一个元素或删除一个元素等；
  - 串：通常以“串的整体”作为操作对象，如：在串中查找某个子串、在串的某个位置上插入一个子串或删除一个子串等。
- 字符串数据结构需支持的基本操作：

1. ***Assign*** 字符串的初始化，用字符串常量为当前字符串初始化。
2. ***GetLen*** 获得字符串的长度。
3. ***IsEmpty*** 判断当前字符串是否为空串。
4. ***Empty*** 清空当前字符串，即使当前串为空串。
5. ***Comp*** 比较两个字符串是否相等。
6. ***Concat*** 将两个字符串拼接在一起。
7. ***SubString*** 获得位置***npos***开始，长度为***nCount***的子串。
8. ***Find*** 获得字符串中子串的出现位置。

- 除上述基本操作外，字符串上的操作还包括***Insert***、***Delete***、***Replace***等。相对串上的基本操作，这些操作相对比较复杂。

3.3 串的存储结构

1. 数组存储
2. 块链存储

3.3.1 串的数组存储表示

(1) 静态数组存储方法

- 为字符串变量分配一个固定长度的存储空间
- 一般用定长数组加以实现

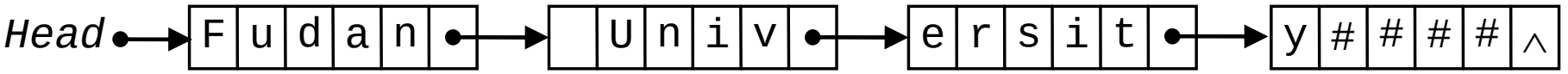
```
const int nMaxLen=1024;  //字符串的最大长度
class SString {
private:
    int nLen;  //字符串的当前长度
    char ch[nMaxLen+1];  //字符串存储空间
};
```

(2) 动态数组存储方法

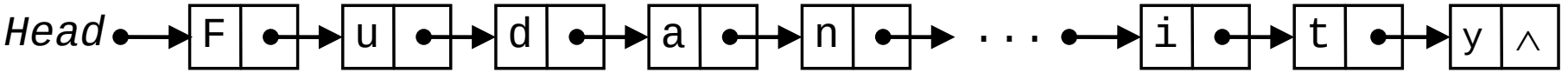
- 使用C++提供的**new**操作符分配一块连续的堆空间

```
class Dstring {
private:
    char *ch;  //指向当前字符串的指针
    int nLen;  //字符串长度
};
```

3.3.2 串的块链存储表示



(a) 结点大小为5



(b) 结点大小为1

图3-1 字符串“*Fudan University*”的块链存储方式

- 结点大小的选择和子串存储方式是影响效率的重要因素

$$\text{存储密度} = \frac{\text{串值所占的存储位}}{\text{实际分配的存储位}}$$

- 块链式存储结构对字符串的连接等操作的实现比较方便，但总体来说操作复杂，占用的存储空间大。
- 块链存储结构字符串的操作实现，与线性表的链表实现类似。

3.4 串的实现

- 程序3-1提供了字符串的类定义，封装了3.2小节描述的8个基本操作
- **程序3-1 字符串的类定义**

```
#ifndef DSTRING__INCLUDED_
#define DSTRING__INCLUDED_
const int nInitLen=1024; //初始最大长度
class DString {
  public:
      DString( );
      //构造函数：为数组分配nInitLen+1大小的空间；初始化串为空串；长度为0
      ~DString( ); //析构函数：释放字符串所占的内存
      DString(const DString &strSrc);
      //重载构造函数：用已知的字符串初始化当前字符串
      DString(const char *chSrc); //重载构造函数：用chSrc初始化当前字符串
      int GetLen( ) const { return nLen; } //内联函数：计算字符串的长度
      int IsEmpty( ) { return nLen ? 0 : 1; } //内联函数：判断当前字符串是否为空串
      void Empty( ) { nLen=0; ch[0]='0'; } //内联函数：清空当前字符串
      DString GetSub(int nPos, int nCount) const;
      //获得当前字符串位置nPos开始的长度为nCount的子串
      char operator [ ] (int nPos) const; //操作符重载：获得指定下标的字符
      Dstring & operator = (const DString &str); //操作符重载：字符串赋值
      Dstring & operator += (const DString &str); //操作符重载：字符串合并
      int operator == (const DString &str) const; //操作符重载：字符串等值判断
      int Find(DString &strSub) const; //字符串的模式匹配，找子串：精确匹配
  private:
      int nLen; //字符串实际长度
      char *ch; //字符串存储所在的数组
};
#endif //#define DSTRING__INCLUDED_
```

- 程序3-2 字符串类的实现

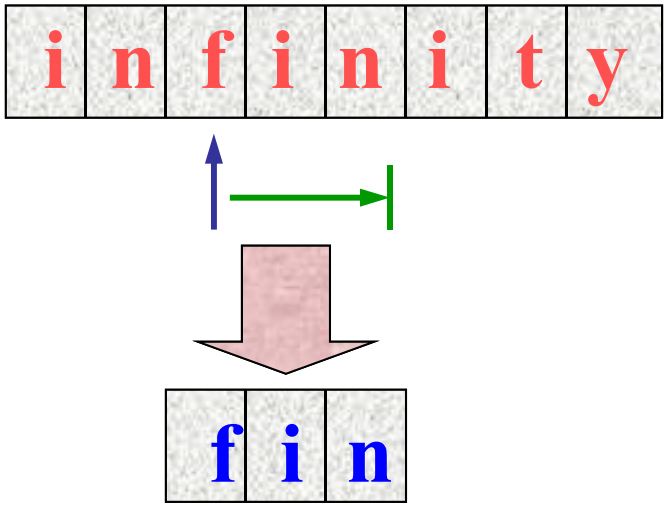
```
1.  DString::DString( ) {
2.      //为动态数组ch开辟nInitLen+1的存储空间
3.      ch=new char[nInitLen+1];
4.      if (!ch) { cerr<<"Allocate Error!\n";  return; }
5.      //初始化当前字符串：长度为0，空串
6.      nLen=0;
7.      ch[0]='\0';
8.  }

1.  DString::~DString( ) {
2.      //释放动态数组的存储空间
3.      delete [ ] ch;
4.  }
```

```
1.  DString::DString(const DString &strSrc) {
2.      nLen=strSrc.GetLen( );  //设置字符串长度
3.      //为动态数组开辟max(strSrc.nLen, nInitLen)+1的存储空间
4.      if (nLen>nInitLen)
5.          ch=new char[nLen+1];
6.      else
7.          ch=new char[nInitLen+1];
8.      if (!ch) { cerr<<"Allocate Error!\n";  return; }
9.      strcpy(ch, strSrc.ch);  //复制字符串序列
10. }
```

```
1.  DString::DString(const char *chSrc) {
2.      nLen=strlen(chSrc);  //设置字符串长度
3.      //为动态数组开辟max(strlen(chSrc), nInitLen)+1的存储空间
4.      if (nLen>nInitLen)
5.          ch=new char[nLen+1];
6.      else
7.          ch=new char[nInitLen+1];
8.      if (!ch) { cerr<<"Allocate Error!\n";  return; }
9.      strcpy(ch, chSrc);  //复制字符串序列
10.  }
```

提取子串的算法示例

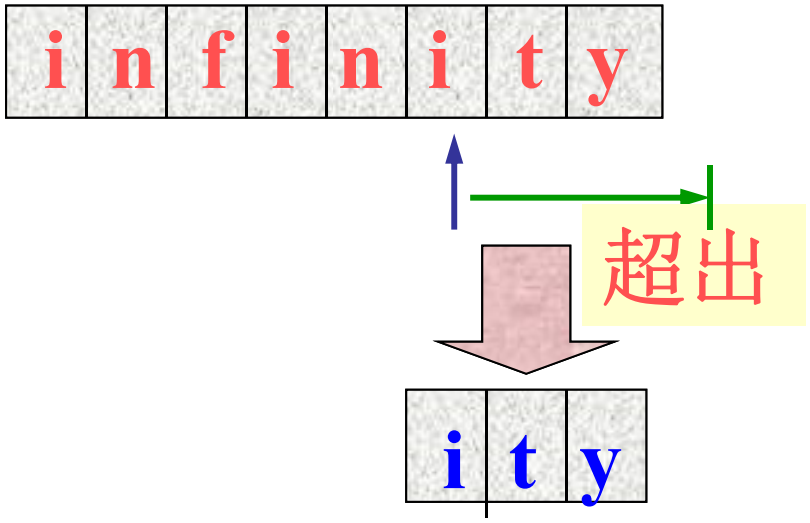
npos=2，nCount=3



npos+nCount-1

≤nLen-1

npos=5，nCount=4



npos+nCount-1

≥nLen

```
1.  //获取子串
2.  DString DString::GetSub(int nPos, int nCount) const {
3.      DString tmpString;
4.      //定义临时字符串，作为保存子串的临时对象
5.      char *pch; //字符串指针，指向要获取的子串
6.      //判断函数的输入参数是否符合逻辑：
7.      //给定子串的在当前串中的位置不能小于0，
8.      //且子串长度不能小于0，且子串的不能超出当前串的右边界
9.      if (nPos<0||nCount<0||nPos+nCount-1>=nInitLen) {
10.         tmpString.nLen=0; tmpString.ch[0]='\0';
11.     }
12.     else { //将子串拷贝到tmpString对象中
13.        //if (nPos+nCount-1<nLen)  nCount=nLen-nPos;
14.         tmpString.nLen=nCount;
15.         pch=ch+nPos;
16.         memcpy(tmpString.ch, pch, nCount);
17.         tmpString.ch[nCount]='\0';
18.     }
19.     return tmpString; //返回获取到的子串
20.     //若函数的参数不合逻辑，返回的是空串
21. }
```

```
1.  //串赋值
2.  Dstring & DString::operator = (const DString &str) {
3.      if (&str!=this) {
4.      //删除当前动态数组，并根据str的长度为当前串开辟存储空间
5.          delete [ ]ch; //释放原有的字符串存储空间
6.          nLen=str.nLen; //复制字符串的长度
7.          if (nLen>nInitLen) //如果新的长度大于默认的初始化长度
8.                                  //按新长度申请存储空间
9.              ch=new char[nLen+1];
10.         else { //否则，按默认长度申请存储空间
11.             ch=new char[nInitLen+1];
12.             //若申请空间不成功，则提示错误信息，并返回空指针
13.             if (!ch) { cerr<<"Allocate Error!\n"; return 0;}
14.              //复制字符串序列
15.             strcpy(ch, str.ch);
16.     }
17.     return *this;
18. }
```

例：串 st="university",

使用示例　newSt=st;

数组赋值　newSt="university"

```
1.  Dstring & DString::operator += (const DString &str) {
2.  //当前串与串str拼接，拼接的结果写入当前串
3.      char *temp=ch;
4.      int n=nLen+str.nLen;
5.      int m=(nInitLen>=n) ? nInitLen:n;
6.      //申请新的存储空间，并进行字符串拼接
7.       ch=new char[m+1];
8.      if (!ch) { cerr<<"Allocate Error!\n";  return 0; }
9.      nLen=n;
10.     strcpy(ch, temp);
11.     strcat(ch, str.ch);
12.     delete [ ] temp;
13.     return *this;
14.  }
```

例：串 st1="beijing ",
st2="university",

使用示例 st1+=st2;

连接结果 st1="beijing university"
st2="university"

```
1.  int DString::operator == (const DString &str) const {
2.  //字符串等值判断：相等则返回0
3.      return strcmp(ch, str.ch);
4.   }

1.  int DString::Find(DString &strSub) const {
2.  //字符串精确匹配：Brute-Force方法
3.      int n=GetLen( );
4.      int m=strSub.GetLen( );
5.      for (int j=0; j<=n-m; ++j) {
6.          for (int i=0; i<m && strSub[i]==ch[i+j]; ++i);
7.              if (i>=m)
8.                  return j;
9.      }
10.     return -1;
11. }
```

```
1.  char DString::operator [ ] (int nPos) const {
2.  //获取当前串的第i个字符
3.      if (nPos<0 && nPos>=nLen)
4.      { cerr<<"nPos Out Of Bounds!"<<endl;  return 0; }
5.      return ch[nPos];
6.   }
```

例： 串 st="university",

使用示例　newChar=st[1];

提取字符　newChar='n'

随堂练习

例1：设S为一个长度为n的字符串，其中的字符各不相同，求解S中互异的非平凡子串（非空且不同于S本身）的个数。

例2：若串S="software"，求解其子串（不含空串）数目。

例3：如果字符串的一个子串（其长度大于1）的各个字符均相同，则称之为等值子串。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。

例4：编写一个算法frequency，统计在一个输入字符串中各个不同字符出现的频度。用适当的测试方法来验证这个算法。

例1：设S为一个长度为n的字符串，其中的字符各不相同，求解S中互异的非平凡子串（非空且不同于S本身）的个数。

除长度为n的子串外，长度为n-1的不同子串个数为2，长度为n-2的不同子串个数为3，依此类推，直至长度为1的不同子串个数为n，即S的非平凡子串个数为：2+3+…+n=n(n+1)/2-1。

例2：若串S=”software”，求解其子串（不含空串）数目。

已知S串长度为8，则其子串个数为1+2+3+…+8=8(8+1)/2=36。

例3： 先从键盘上接受字符串并送入字符串数组 S，然后扫描字符串数组 S。设变量 head 指向当前发现的最长等值子串的串头，max 记录此子串的长度。扫描过程中，若发现等值子串则用 count 变量记录其长度，如果它的长度大于原最长等值子串的长度，则对 head 和 max 进行更新。重复上述过程直到 S 的末尾。最后，根据扫描所得的结果输出最长等值子串或输出等值子串不存在信息。

```
void Equstring (char s[ ])
{
    for(k=0; ; k++)
    {
        scanf((“%c”, &S[k]);
        if (S[k]==’!’)
            break;
    }
    for (i=0, j=1, head=0, max=1; S[i]!=’!’ && S[j]!=’!’; i=j, j++)
    {
        count=1;
        while (S[i]==S[j])
        {
            j++;
            count++;
        }
        if (count>max)
        {
            head=i;
            max=count;
        }
    }
    if (max>1)
        for (k=head; k<(head+max); k++)
            printf(“%c”, S[k]);
    else
        printf(“There is no equivalent substring in S!”);
}
```

例4：

```
void frequency(String &s) {
    int LenOfChars=0;  //不相同的字符个数
    int freqs= new int[maxLen+1];
    char *temp= new char[maxLen+1];
    for (int j=0; j<s->length( ); j++) {
      for (int k=0; k<LenOfChars; k++){
        if (temp[k] == s[j]){
          freqs[k]++;  break;
        }
      }
     if ( k = = LenOfChars) {
        temp[LenOfChars] = s[j];
        freqs[LenOfChars++] = 1;
     }
  }
  cout << "Results are: " << endl;
  for( int j=0; j<LenOfChars; j++)
     cout << temp[j] << " is: "  <<freqs[i] << endl;
  cout << "End of Print!" << endl;
}
```

3.5 串的模式匹配算法

- 串的模式匹配算法解决的是在长串中查找短串的一个、多个或所有出现的问题。
- 通常，我们称长串为***text***，称短串为***pattern***。
- 一个长度为***m***的**pattern**可被表述为***x*=*x*[0..*m*-1]**；长度为***n***的**text**可被表述为***y*=*y*[0..*n*-1]**；而模式匹配的任务是找到***x***在***y***中的出现。
- 模式匹配过程中，程序会查看**text**中长度为***m***的窗口，即用**pattern**串和**text**的窗口中的子串进行比对。比对完成后，将窗口向右滑动，并不断重复这一过程。直到根据需要找到所需匹配为止。这种机制被称为滑动窗口机制。
- 本节所讨论的若干串模式匹配算法都是基于滑动窗口机制的算法。

- 定义 在串中寻找子串（第一个字符）在串中的位置。
- 词汇 在模式匹配中，子串称为模式，串称为目标。
- 示例 目标 T：“Beijing”

  模式 P：“jin”

  匹配结果 = 3

模式匹配的哲理

- 词汇　目标：“外面的世界”
  模式：“小小的我”

- 定义　在外面的世界中寻找
  小小的我的“位置”。

BF(Brute-Force)算法

- **BF**算法是最基本的串模式匹配算法。程序3-2的***Find***函数就是利用**BF**算法实现的查找最左匹配的实现。
- 其过程是：依次比对**pattern**和滑动窗口中的对应位置上的字符；比对完成后将滑动窗口向右移动**1**，直到找到所需的匹配为止。

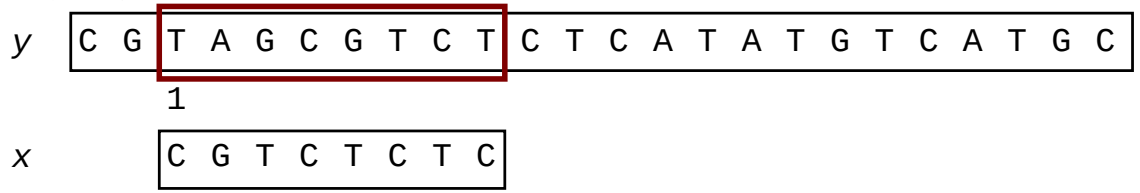
第一轮比较：

比较窗口右移 1 个位置

第二轮比较：

比较窗口右移 1 个位置

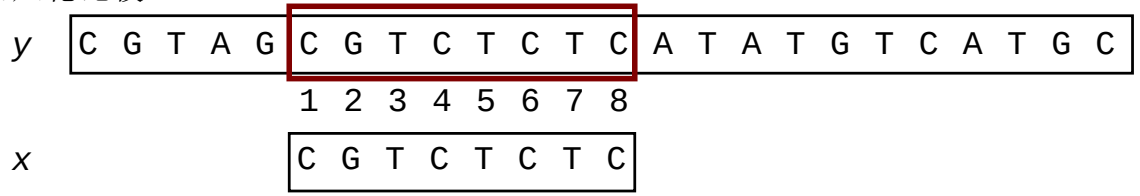
第三轮比较：



比较窗口右移 1 个位置

...

第六轮比较：



找到子串，程序结束

Brute-Force算法示例（图3-2）

第1趟 *T*　a b b a b a

P　a b a

第2趟 *T*　a b b a b a

P　a b a

第3趟 *T*　a b b a b a

P　a b a

第4趟 *T*　a b b a b a

P　a b a

√

穷举模式匹配过程

用*P*中的字符依次与*T*中的字符做比较

```
int AString :: Find(AString &pat) const {
//穷举的模式匹配
    char *p=pat.ch, *s=ch;  int i=0;
    while (*p!='\0' && *s!='\0')
    //当两串未检测完
       if (*p++ != *s++)  //比较串字符，不等
          { i++;  s=ch+i;  p=pat.ch; }
          //对应字符不相等，对齐目标的下一位置
          //继续比较
    if (*p=='\0')  return i;  //相等
    else return -1;
}
```

- 穷举的模式匹配算法分析（时间代价）

  设模式串*P*有*m*个字符，而目标串*T*有*n*个字符：

  最好情况—*T*的前*m*个字符与*P*匹配，可在*m*次比较后找到匹配结果，算法复杂度为$O(m)$。

  最坏情况—假设未进行比较最后一个字符之类的优化，且第一个字符总是能匹配上，但却永远没有匹配上的模式。如：*P*="*abc*", *T*="*aaaaaaaa*" (*m*=3, *n*=8)，则模式*P*="*abc*"的*m* (*m*=3)个字符必须与*T*中的文本块一共比较*n*-*m*+1次，一般情况下必须比较*m*个字符共(*n*-*m*+1)次，即一共进行*m*(*n*-*m*+1)次。

  而 $m(n-m+1)<=m(n-m+m)=mn$

  算法复杂度的估计值为$O(mn)$。

- 算法速度慢的原因在于回溯

  每趟重新比较时，目标串的检测指针要回退。

BF算法的特点总结

(1) 不需要预处理过程；

(2) 只需固定的存储空间；

(3) 滑动窗口的移动每次都为**1**；

(4) 字符串的比对可按任意顺序进行（从左到右、从右到左、或特定顺序均可）；

(5) 算法的时间复杂性为**O(*m*×*n*)**；

(6) 最大比较次数为**(*n*-*m*+1)×*m***。

KR(Karp-Rabin)算法（略）

- 作为最朴素的字符串匹配算法，**BF**算法的效率并不理想。其主要原因有二：
  - 其一是子串与滑动窗口内的子串逐个字符匹配所引发的效率问题；
  - 其二是该算法太健忘，前一次匹配的信息其实可以有部分应用到后一次匹配中的，而**BF**算法只是简单的把这个信息扔掉，重头再来。
- **KR**算法优化“滑动窗口内容逐一匹配”
- **KR**算法：将滑动窗口内***m***个字符的比较变为一个哈希值的比较。
  - 通过对字符串进行哈希运算，然后比较子串哈希值与滑动窗口内子串的哈希值；仅当这两个哈希值相等时，再来比较窗口内的子串是否相等。

- 程序3-3 Karp-Rabin算法框架

```
1.  function RabinKarp(string s[0..n-1], string sub[0..m-1])
2.  hsub=hash(sub[0..m-1]); //计算子串的哈希值
3.  hs=hash(s[0..m-1]); //计算窗口内子串的哈希值
4.  for i from 0 to n-m
5.  //依次比较窗口内子串与给定子串是否相同
6.      if hs=hsub
7.          if s[i..i+m-1]=sub
8.              return i; //如果相同则记录位置并退出函数
9.          hs=hash(s[i+1..i+m]); //否则，当前窗口右移
10.     return not found;
11.     //没有发现，记录没有发现给定子串，退出
```

哈希算法

- **KR**算法的效率取决于哈希函数的的选取。通常，**KR**算法使用$h(x)=x \bmod q$作为哈希函数。
- 滑动窗口的哈希值计算公式：

$$hash(w[0..m-1])=$$
$$(w[0]\times 2^{m-1}+ w[1]\times 2^{m-2}+\ldots+ w[m-1]\times 2^{0}) \bmod q$$

其中，q为一个大整数。

- 当滑动窗口右移时，需要对当前窗口内的子串重新利用哈希函数计算，其计算公式为：

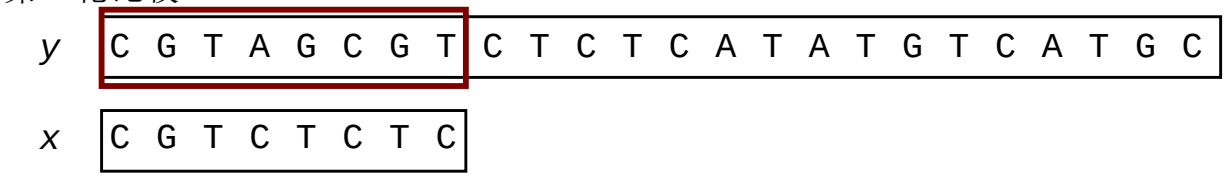
$$rehash(a, b, h)=((h-a\times 2^{m-1})\times 2+b) \bmod q$$

其中，h为上一滑动窗口的哈希值；a为即将移出滑动窗口的字符；b是即将移入滑动窗口的字符。

- 程序3-4 **Karp-Rabin**算法（窗口长度为m，q选取2^{m-1}）

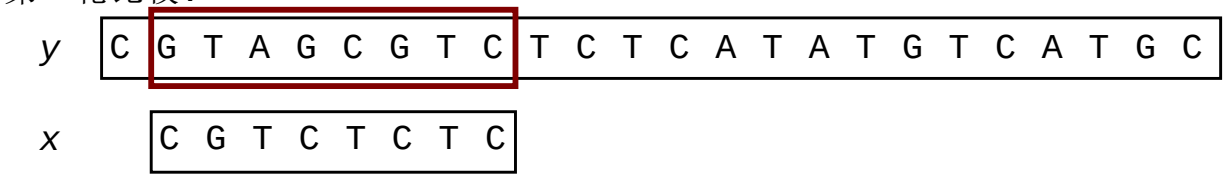
```
1.  #define Rehash(a, b, h) ((((h)-(a)*d)<<1)+(b))
2.  int KR(char *x, int m, char *y, int n) {
3.      int d, hx, hy, i, j;
4.      //预处理，计算d=2^(m-1)，这里的d为算法描述中的q
5.      for (d=i=1; i<m; i++)
6.          d=d<<1;
7.      //分别计算字符串x和y的哈希值
8.      for (hy=hx=i=0; i<m; i++) {
9.          hx=(hx<<1)+x[i];
10.         hy=(hy<<1)+y[i];
11.     }
12.     //查找过程
13.     for (j=0; j<=n-m; j++) {
14.      //如果哈希值相等，再判断字符串是否相等
15.          if (hx==hy && memcmp(x, y+j, m)==0)
16.              return j;  //如字符串相等，返回位置
17.          hy=Rehash(y[j], y[j+m], hy);
18.          //否则，对y上的子串重新进行哈希
19.     }
20.     return -1;
21. }
```

第一轮比较:



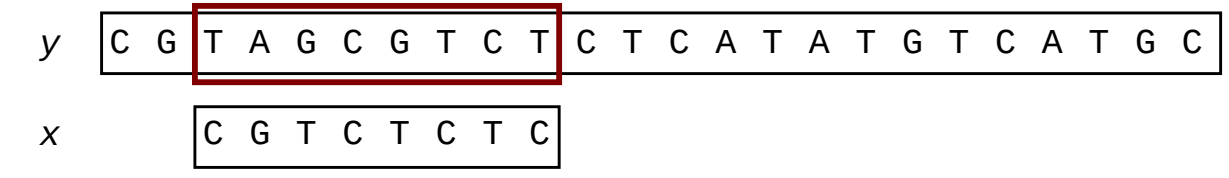
Hash(*y*[0..7])=17910

第二轮比较:



Hash(*y*[1..8])=18735

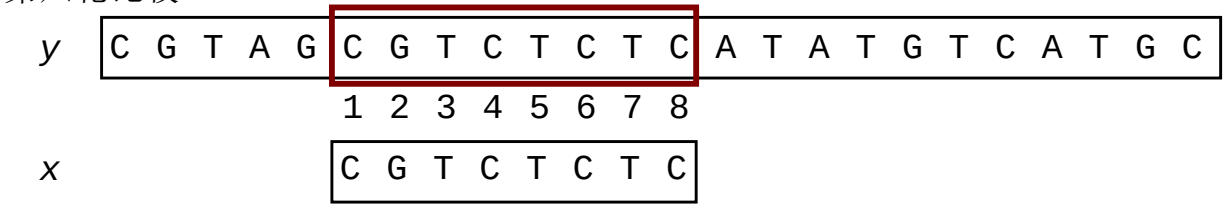
第三轮比较:



Hash(*y*[2..9])=19378

...

第六轮比较:



Hash(*y*[5..12])=Hash(*x*)=18055

找到子串，函数结束

图3-3 Karp-Rabin算法示例

KR算法的特点

(1) 利用哈希的方法；

(2) 预处理需要**O(*m*)**的时间和常数的存储空间；

(3) 最坏情况下，算法时间复杂性为**O(*m*×*n*)**；

(4) 算法时间复杂性的预期为**O(*m*+*n*)**。

KMP (Knuth-Morris-Pratt)算法

- **BF**和**KR**算法的一个共同点是每次比较的窗口向右滑动的距离均为**1**。
- **KMP**算法，旨在利用**pattern**的内容指导滑动窗口向右滑动的距离。

简单模式匹配的缺点：无谓比较

目标 *T*　*S T U D E N S T U D E N T……*

‖ ‖ ‖ ×

模式 *pat*　S T U D E N T

目标 *T*　*S T U D E N S T U D E N T……*

×

模式 *pat*　S T U D E N T……

目标 *T*　*S T U D E N S T U D E N T……*

‖ ‖ ‖ ‖

模式 *pat*　S T U D E N T

改进的模式匹配：

寻找最大“跳跃”

目标 T　$t_0\ t_1\ t_2\ \cdots\cdots\ t_{j-1}\ t_j\ \cdots\cdots\ t_{n-1}$

$\|\ \|\ \|\qquad\quad X$

模式 pat　$p_0\ p_1\ p_2\ \cdots\cdots\ p_{j-1}\ p_j\ \cdots\cdots\ p_{m-1}$

目标 T　$t_0\ t_1\ \ldots\ t_k\ \ldots\qquad\qquad\qquad\ldots\ t_{n-1}$

$\updownarrow\ \updownarrow\qquad\updownarrow\quad\updownarrow$

模式 pat　$p_0\ p_1\ \cdots\cdots\ p_{m-2}\ p_{m-1}$

设目标T="$t_0t_1\cdots t_{n-1}$"，模式P="$p_0p_1\cdots p_{m-1}$"，

用穷举模式匹配法做第s+1趟比较时——

从目标T的第s位置t_s与模式P的第0位置p_0开始进行比较，直到目标T第t_{s+j}"失配"。

$$
\begin{array}{lcccccccccc}
T & t_0 \cdots t_{s-1} & t_s & t_{s+1} & t_{s+2} & \cdots & t_{s+j-2} & t_{s+j-1} & t_{s+j} & \cdots & t_{n-1} \\
 & & \| & \| & \| & & \| & \| & \times & & \\
P & & \{p_0 & [p_1 & p_2 & \cdots & p_{j-2}\} & p_{j-1} &] & p_j & \\
 & & & p_0 & p_1 & \cdots & \cdots & p_{j-2} & & & \\
 & & & & p_0 & p_1 & \cdots & p_{j-3} & & &
\end{array}
$$

则有 $t_s\, t_{s+1}\, t_{s+2} \cdots t_{s+j-1} = p_0\, p_1\, p_2 \cdots p_{j-1}$ (1)

为使模式 P 与目标 T 匹配，必须满足

$$p_0\, p_1\, p_2 \cdots p_{j-1} \cdots p_{m-1} = t_{s+1}\, t_{s+2}\, t_{s+3} \cdots t_{s+j} \cdots t_{s+m}$$

如果 $p_0\, p_1 \cdots p_{j-2} \neq p_1\, p_2 \cdots p_{j-1}$ $(r=2)$ (2)

则立刻可以断定

$$p_0\, p_1 \cdots p_{j-2} \neq t_{s+1}\, t_{s+2} \cdots t_{s+j-1}$$

下一趟必不匹配。

同样，若 $p_0 p_1 \cdots p_{j-3} \neq p_2 p_3 \cdots p_{j-1}$ $(r=3)$

则再下一趟也不匹配，因为有

$$p_0 p_1 \cdots p_{j-3} \neq t_{s+2} t_{s+3} \cdots t_{s+j-1}$$

直到对于某一个“r”值，使得

$$p_0 p_1 \cdots p_{j-r} \neq p_{r-1} p_r \cdots p_{j-1}$$

且 $p_0 p_1 \cdots p_{j-(r+1)} = p_r p_{r+1} \cdots p_{j-1}$

用k替换$j-(r+1)$

$$p_0 p_1 \cdots p_k = p_{j-k-1} p_{j-k} \cdots p_{j-1}$$

则

$$\begin{array}{lllll} p_0 p_1 \cdots p_k = & t_{s+j-k-1} & t_{s+j-k} & \cdots t_{s+j-1} & t_{s+j} \\ & \| & \| & \| & \times \\ & p_{j-k-1} & p_{j-k} & \cdots p_{j-1} & p_j \\ & p_0 & p_1 & \cdots p_k & p_{k+1} \end{array}$$

在第s趟比较“失配”时的模式P从当时位置直接向右“滑动”$j-k-1$位。

因为目标T中t_{s+j}之前已经与模式P之前的字符匹配，则可以直接从T中的t_{s+j}（即上一趟失配的位置）与模式中的p_{k+1}开始，继续向下进行匹配比较。

改进算法中，目标T在第s趟比较失配时，扫描指针s不必回溯。算法下一趟继续从此处开始向下进行匹配比较；而在模式P中，扫描指针应退回到p_{k+1}位置。

KMP (Knuth-Morris-Pratt)算法

- **Knuth-Morris-Pratt (KMP)**算法发现每个字符对应的该***k***值仅依赖于模式***P***本身，与目标对象***T***无关。
  - **1970**年，**S.A. Cook**在进行抽象机的理论研究时证明了在最差情况下模式匹配可在***N+M***时间内完成。
  - 此后，**D.E. Knuth** 和**V.R. Pratt**以此理论为基础，构造了一种方法来在***N+M***时间内进行模式匹配，与此同时，**J.H. Morris**在开发文本编辑器时为了避免在检索文本时的回溯也得到了同样的算法。

*k*的确定方法

当比较到模式第*j*个字符失配时，*k*的值与模式的前*j*个字符有关，与目标无关。

目标 *T*　　*F I F I F I Y U D E N T……*

‖ ‖ ‖ ‖ ×

模式 *pat*　**F I F I Y**

启发：要更好地了解外面的世界，

先要了解小小的我。

失效函数

可以利用失效函数***next (j)***描述。

失效函数又名失败链接值。

哲理：在当前位置遇到失败后，

重新开始的位置。

当模式*P*中的第*j*个字符与目标*T*中相应字符失配时，模式*P*中应该由哪个字符与目标中刚失配的字符对齐继续进行比较。

若设 模式 $P=p_0p_1\cdots p_{m-2}p_{m-1}$

$$next(j)=\begin{cases}-1, & \text{当}j=0\\ k+1, & \text{当}0\le k<j-1\text{且使得}p_0p_1\cdots p_k=p_{j-k-1}p_{j-k}\cdots p_{j-1}\text{的最大整数}\\ 0, & \text{其它情况}\end{cases}$$

示例：确定失效函数*next(j)*

j	0	1	2	3	4	5	6	7
P	***a***	***b***	***a***	***a***	***b***	***c***	***a***	***c***
next(j)	**−1**	**0**	**0**	**1**	**1**	**2**	**0**	**1**

利用失效函数*next(j)*的匹配处理

如果$j=0$，则目标指针加 1，

模式指针回到 p_0。

如果$j>0$，则目标指针不变，

模式指针回到 $p_{next(j)}$。

运用KMP算法的匹配过程

第1趟 目标 a c a b a a b a a b c a c a a b c

模式 a b a a b c a c

× j=1 ⇨ j=next(j)=0

第2趟 目标 a c a b a a b a a b c a c a a b c

模式 a b a a b c a c

× j=0 目标指针加 1

第3趟 目标 a c a b a a b a a b c a c a a b c

模式 a b a a b c a c

× j=5

⇨ j=next(j)=2

第4趟 目标 a c a b a a b a a b c a c a a b c

模式 (a b) a a b c a c √

```
int DString :: FastFind(DString pat) const {
 //带失效函数的KMP匹配算法
    int posP=0, posT=0;
    int lengthP=pat.nLen, lengthT=nLen;
    while (posP<lengthP && posT<lengthT)
        if (pat.ch[posP]==ch[posT]) {
              posP++;  posT++;  //相等继续比较
        }
        else if (posP==0)  posT++;  //不相等
              else posP=pat.next[posP];
    if (posP<lengthP)  return -1;
    else return posT-lengthP;
}
```

```
int DString :: FastFind(DString &pat,
                                    int k, int next[ ]) const {
 //用模式串pat从k开始寻找当前串中匹配的位置
    int posP=0, posT=k;
    int lengthP=pat.nLen, lengthT=nLen;
    while (posP<lengthP && posT<lengthT)
        if (posP==-1 || pat.ch[posP]==ch[posT]) {
              posP++;  posT++;  //相等继续比较
        }
        else posP=pat.next[posP];
    if (posP<lengthP)  return -1;
    else return posT-lengthP;
}
```

- 改进的模式匹配算法可使目标串的检测指针每趟不回退。
- 改进的模式匹配(KMP)算法的时间代价
  - 在带失效函数的匹配算法中，字符比较次数最多为目标串的长度n，则其时间复杂度为$O(n)$；
  - 在失效函数计算过程中，循环次数至多不能超过$m-1$，则其时间复杂度为$O(m)$；
  - 在包括计算失效函数的整个模式匹配过程中，时间复杂度为$O(m+n)$。

字符串特征向量（略）

- 设模式P由m个字符组成，记为

  $P=p_0p_1p_2p_3\cdots p_{m-1}$

- 令特征向量N用来表示模板P的字符分布特征，简称N向量。
  - 和P同长，由m个特征数n_0 …n_{m-1}整数组成，记为：

    $N=n_0n_1n_2n_3\cdots n_{m-1}$

N在大多数文献中称为next数组，每个n_j对应next数组中的元素next[j]。

- 模式P的前缀子串（首串）
  - P的前*t* 个字符：$p_0p_1p_2\cdots p_{t-1}$
- 模式P的左子串（尾串）
  - 在P的第j位置之前的t个字符，称为j位置的左子串：$p_{j-t}\cdots p_{j-2}p_{j-1}$
- 模式P第j个位置的特征数n_j
  - 最长的（*t*最大的）能够与前缀子串匹配的左子串（简称第j位的最长前缀串）
  - *t*即p_j的特征数n_j

- Knuth等人认为合理的右移方案是根据P[0..*j*-1]的**最大前缀**和**最大后缀**能够匹配的内容来决定窗口右移的距离。为此KMP算设计一个next函数:

$$\text{next}(j) = \begin{cases} -1 & \text{如果} j == 0 \\ \max\{k | P[0..k-1] == P[j-k..j-1]\} & \text{如果 } P[0..k-1] == P[j-k..j-1] \;\&\&\; k < j \\ 0 & \text{其他情况} \end{cases}$$

- 特征数n_{j+1} $(0 \le n_{j+1} \le j)$的递归定义如下：
  1. 如果j==0, 令n_0=-1；对于n_{j+1}，假定已知前一位置的特征数 n_j=k；
  2. 如果j>0, k≥0且P_j=P_k，则n_{j+1}=k+1；
  3. 当k≥0, 且$P_j \neq P_k$时，则令k=n_k，并让步骤3循环直到条件不满足（变为P_j==P_k或 k==-1）；
  4. 当k<0（此时，k=-1），则 n_{j+1}=0。

对应的求特征向量算法框架

一般把模式的第一个字符的特征数设置为-1，以尽可能地减少冗余比较：

- 在KMP快速模式匹配中，若用n_i表示第i位的特征值，特征数n_{i+1} $(-1 \le n_i < i)$可以采用如下定义：
  - 1. $n_0=-1$；对于$i>0$的n_{i+1}，假定已知前一位置没有优化以前的特征数 $n_i=k$；
  - 2. 当$k>=0$，且$P_i \neq P_k$时，则令$k=n_k$，并让步骤2循环直到条件不满足（变为$P_i==P_k$，或上一步骤$k==0$而导致$k=n_k$步骤后$k==-1$了）；
  - 3. $n_{i+1}=k+1$（这是没有优化以前的特征数，在优化前传到第一步用于计算下一个特征值）；
  - 4. 若$P_{i+1}==P_{k+1}$，则$n'_{i+1}=n_{k+1}$（此步骤为优化）。

计算next数组

1. 求$p_0, p_1, \cdots, p_{j-1}$中最大相同的前缀与后缀的长度t;
2. next[j]=t;

计算next[j]时充分利用了位置j之前的各个字符的next值。

- 程序3-5 KMP算法

```
1.  //预处理：计算Next数组
2.  //带优化
3.  void preprocessing(char *p, int m, int Next[ ]) {
4.      for (int i=0, j=Next[0]=-1; i<m-1; ) {
5.          for ( ; j>-1&& p[i]!=p[j]; )
6.              j=Next[j];
7.          i++;  j++;
8.          if (p[i]==p[j])
9.              Next[i]=Next[j];
10.         else
11.             Next[i]=j;
12.     }
13. }
```

```
1.  int KMP(char *p, int m, char *t, int n)
2.  {
3.      int i, j, Next[SIZE];
4.      //预处理
5.      preprocessing(p, m, Next);
6.      //查找
7.      for (i=j=0; j<n; ) {
8.          for ( ; i>-1 && p[i]!=t[j]; )
9.              i=Next[i];
10.         i++;  j++;
11.         if (i>=m)
12.             return j-i;
13.     }
14.     return -1;
15. }
```

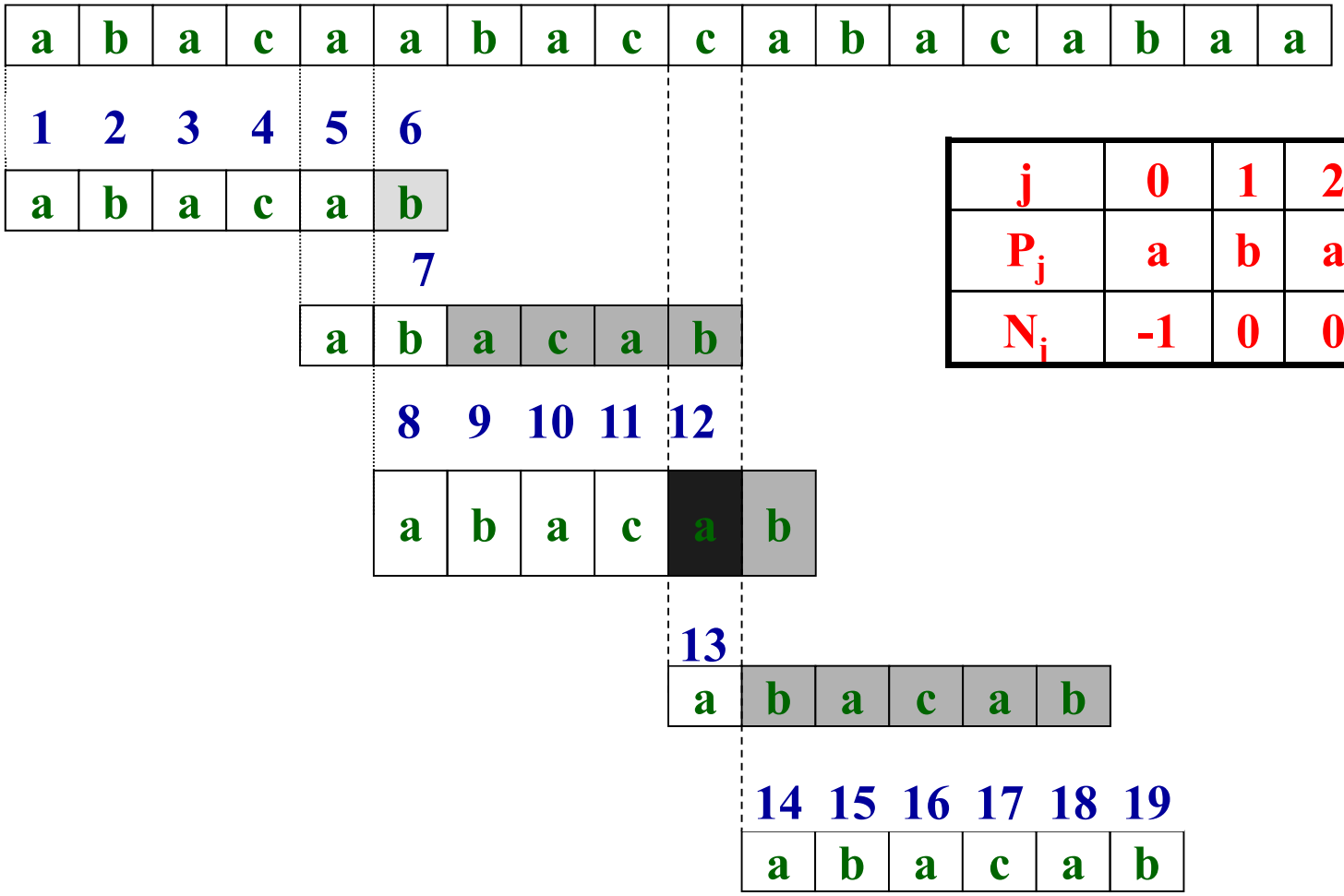
- 程序3-5 KMP算法

```
1.  //预处理：计算Next数组
2.  //带优化
3.  void preprocessing(char *p, int
    m, int Next[ ]) {
4.      for (int i=0, j=Next[0]=-1;
    i<m-1; ) {
5.          for ( ; j>-1&& p[i]!=p[j]; )
6.              j=Next[j];
7.          i++;  j++;
8.          if (p[i]==p[j])
9.              Next[i]=Next[j];
10.         else
11.             Next[i]=j;
12.     }
13. }
```

j	0	1	2	3	4	5
P_j	a	b	a	c	a	b
N_j	-1	0	0	1	0	1

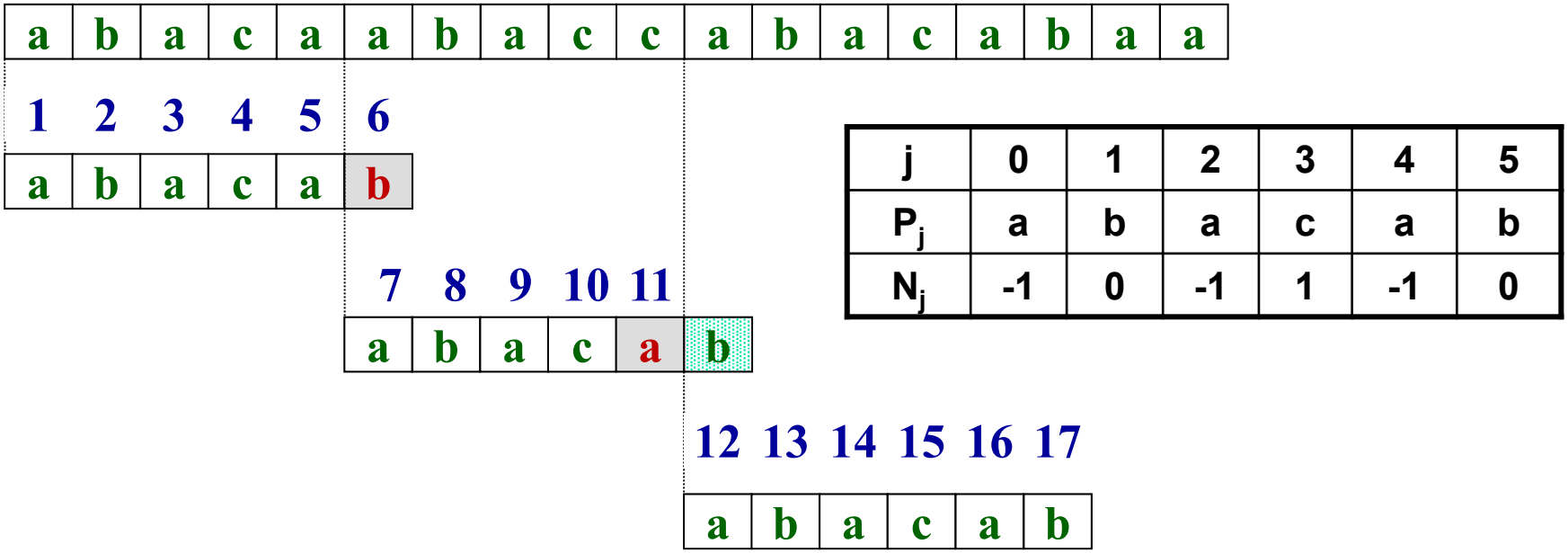
j	0	1	2	3	4	5
P_j	a	b	a	c	a	b
N_j	-1	0	-1	1	-1	0

KMP模式匹配：示例



j	0	1	2	3	4	5
P_j	a	b	a	c	a	b
N_j	-1	0	0	1	0	1

KMP模式匹配：示例



j	0	1	2	3	4	5
P_j	a	b	a	c	a	b
N_j	-1	0	-1	1	-1	0

预处理过程

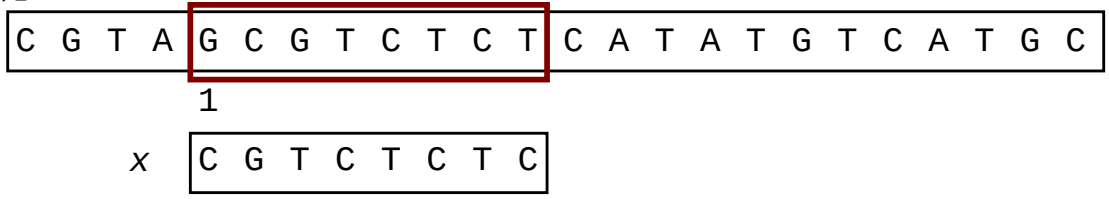
i	0	1	2	3	4	5	6	7	8
x[*i*]	C	G	T	C	T	C	T	C	
Next[*i*]	-1	0	0	-1	1	-1	1	-1	1

查找过程

第一轮：

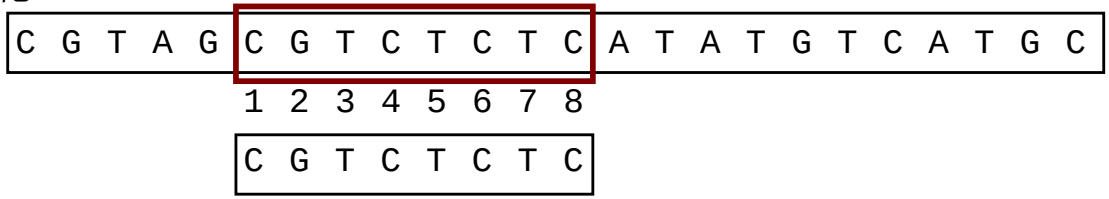
窗口右移 4(i-Next[i]=3-(-1))

第二轮：



窗口右移 1(i-Next[i]=0-(-1))

第三轮：



找到子串，程序结束

图3-4 KMP算法示例

KMP算法的特点

(1) 与**BF**算法类似，按照从左到右的顺序进行比较；

(2) 预处理需要**O(*m*)**的时间和存储空间；

(3) 算法时间复杂性为**O(*m*+*n*)**；

(4) 查找阶段的最大比较次数为**2n-1**。

随堂练习

例1：设串s为“aaab”，串t为“abcabaa”，串r为“abcaabbabcabaacbacba”，试分别计算它们的失效函数f(j)的值。

例2：给定一模式串p=”abcabaa”与目标串t=”abcaabbabcabaacbacba”，求解模式串的失效函数值，并给出利用KMP算法进行模式匹配的每一趟匹配过程的实现。

BM(Boyer-Moore)算法（略）

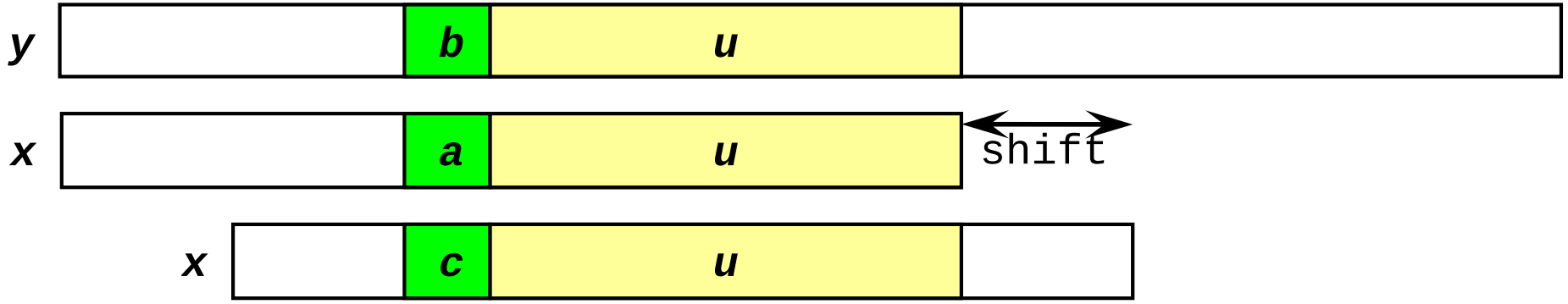
- **BM**算法与**KMP**算法的差别在于：

  (1) 在进行匹配比较时，不是自左向右进行，而是自右向左进行；

  (2) 预先计算出正文中可能的字符在模式中出现的位置相关信息，利用这些信息来减少比较次数。

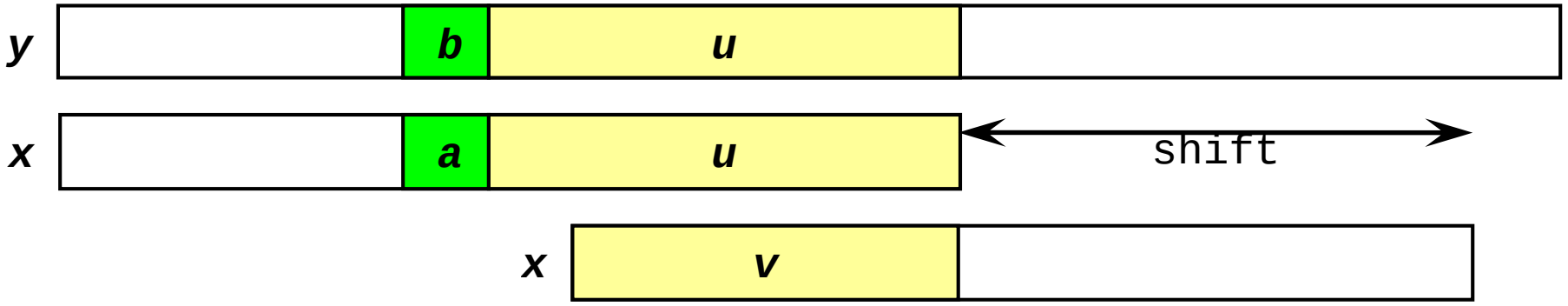
- 根据字符串匹配的思想，窗口向右移动的动作一定发生在匹配失效或完全匹配时。为了右移更大的距离，**BM**算法使用两个预计算函数来指导窗口向右移动的距离，这两个函数分别被称为好后缀移动(***Good-Suffix Shift***)和坏字符移动(***Bad-Character Shift***)。

- 好后缀移动　假定当匹配失效发生时，窗口的左侧位于字符串的第j个位置，且失效发生在第i个位置。由于**BM**算法按照从右向左的顺序对窗口中的字符串进行比对，这时$x[i+1..m-1]=y[j+i+1..j+m-1]=u$，且$a=x[i]\neq y[j+i]=b$。好后缀移动的基本思想是将窗口右移，使$x$中片段$u$除$x[i+1..m-1]$之外的最右出现移动至当前的比对位置（如图3-5所示）。



- 图3-5 好后缀移动（后缀重复出现）

- ***x*[1..*i*]中与*x*[*i*+1..*m*-1]这段相同，且再向前一个字符不同的部分移动到原对齐位置。若*x*中不存在这样的部分，即*x*[*i*+1...*m*-1]这一段在*x*中只出现了一次，则找到最长的（不长于*u*）这样一段字符串*v*：既是*x*最开头的一段，也是*x*最末尾的一段，也就是所谓的相同的真前缀与真后缀，将开头的一段移动到与*y*中相应的字符串对齐的位置（如图3-6所示）。



- 图3-6 好后缀移动（部分后缀重复出现）

- 坏字符移动　　当匹配失效发生时，窗口的左侧位于字符串的第*j*个位置，且失效发生在第*i*个位置。假定*y[j]=b*，这时将窗口移动到下一个*b*的出现位置（如图3-7所示）；如果*x*中不存在*b*，则将窗口向右滑动，使得*x*的左端对齐*y*中*b*的下一个字符。（如图3-8所示）

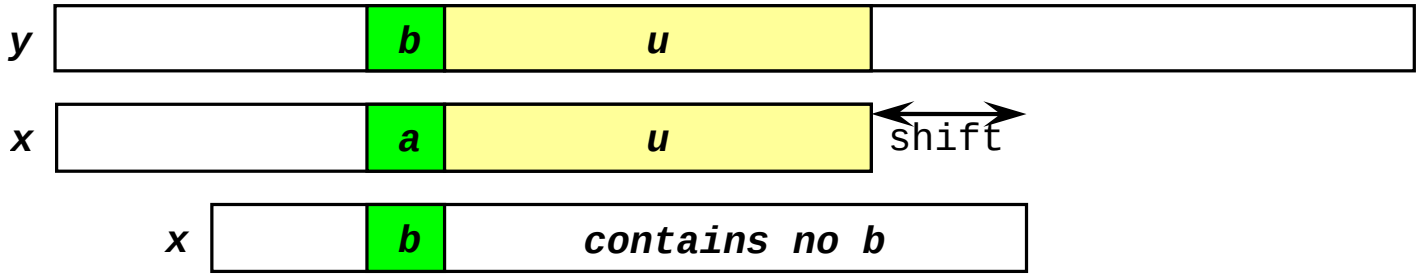


图3-7

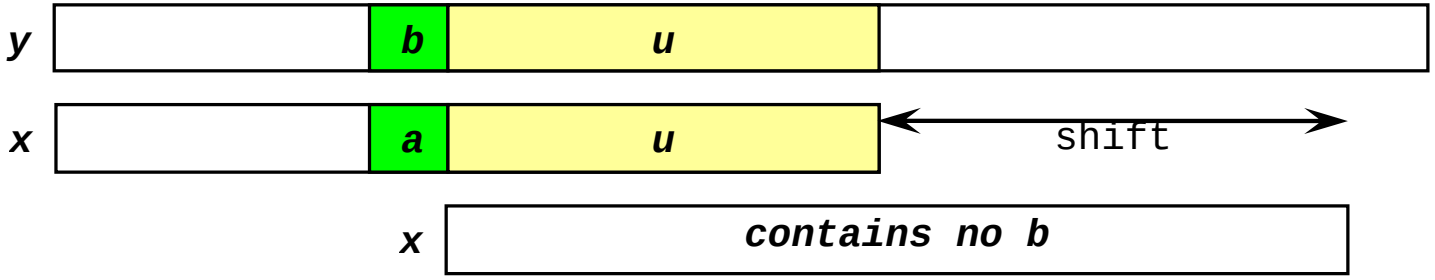


图3-8

程序3-6 BM算法

```
1.  const int ASIZE=200;
2.  const int XSIZE=100;
3.
4.  //预计算bmBc数组（存放坏字符信息）
5.  //BmBc数组的下标是字符，而不是数字
6.  void preBmBc(char *x, int m, int bmBc[ ])
7.  {
8.      int i;
9.      for (i=0; i<ASIZE; ++i)
10.         bmBc[i]=m;
11.     for (i=0; i<m-1; ++i)
12.         bmBc[x[i]]=m-i-1;
13. }
```

```
1.  //在计算BmGc数组时，为提高效率，先计算辅助数组Suff[ ]
2.  void suffixes(char *x, int m, int *suff)
3.  {
4.      int f, g, i; //f为匹配区间的右端标号
5.      //g为匹配区间的左端标号-1（g位置的字符不匹配）
6.      suff[m-1]=m; //模式串中的最末位置的匹配的区间为整个字符串
7.      g=m-1; //将匹配区间的左端标号赋值为m-1
8.      for (i=m-2; i>=0; --i) { //循环计算suff数组
9.      //若i>g，则i落在了当前覆盖最远的匹配区间当中
10.      //可以利用已经计算好的suff[i]值
11.         if (i>g && suff[i+m-1-f]<i-g)
12.             suff[i]=suff[i+m-1-f];
13.         //若i<g，则i落在了当前覆盖最远的匹配区间之外
14.         //需要从第i个字符向前检验，直到不能匹配的字符q为止。
15.         //则suff[i]=f-g.
16.         //该过程相当于右端标号f保持不变，
17.         //每计算一次suff，则更新一次f。
18.         else {
19.             if (i<g)
20.                 g=i;
21.             f=i;
22.             while (g>=0 && x[g]==x[g+m-1-f])
23.                 --g; //直到匹配结束时为止
24.             suff[i]=f-g;
25.         }
26.     }
27. }
```

```
1.  //预计算bmGs数组（存放好后缀信息）
2.  //BmGs数组的下标是数字，表示字符在模式串中的位置
3.  void preBmGs(char *x, int m, int bmGs[ ])
4.  {
5.      int i, j=0, suff[XSIZE];
6.      suffixes(x, m, suff);
7.      for (i=0; i<m; ++i)
8.          bmGs[i]=m;
9.      for (i=m-1; i>=-1; --i)
10.         if (i==-1 || suff[i]==i+1)
11.             for (; j<m-1-i; ++j)
12.                 if (bmGs[j]==m)
13.                     bmGs[j]=m-1-i;
14.     for (i=0; i<=m-1; ++i)
15.         bmGs[m-1-suff[i]]=m-1-i;
16.  }
```

```
1.  int BM(char *x, int m, char *y, int n)
2.  {
3.      int i, j, bmGs[XSIZE], bmBc[ASIZE];
4.      //预处理
5.      preBmGs(x, m, bmGs);
6.      preBmBc(x, m, bmBc);
7.      //查找
8.      j=0;
9.      while (j<=n-m) {  //计算字符串是否匹配到了尽头
10.         for (i=m-1; i>=0&&x[i]==y[i+j]; --i);
11.         if (i<0)
12.             return j;  //找到匹配，  函数结束
13.         else
14.             j+=MAX(bmGs[i], bmBc[y[i+j]]-m+1+i);
15.         //右移窗口
16.     }
17. }
```

预处理

c	A	C	G	T
bmBCs[*i*]	8	2	6	1

i	0	1	2	3	4	5	6	7
x[*i*]	C	G	T	C	T	C	T	C
suff[*i*]	1	0	0	2	0	4	0	8
bmGs[*i*]	7	7	7	2	7	4	7	1

查找

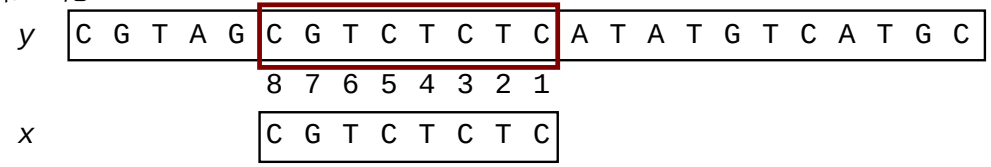
第一轮：

右移 1 (*bmGs*[7]=*bmBc*[A]-7+7)

第二轮：

右移 4 (*bmGs*[7]=*bmBc*[A]-7+7)

第三轮：



找到子串，程序结束

图3-9 BM算法示例

BM算法的特点

(1) 按照从右到左的顺序进行比较；

(2) 预处理需要**O($m+\delta$)**的时间和存储空间；

(3) 匹配时间的复杂性为**O($m\times n$)**；

(4) 最好情况下，时间复杂性为**O(n/m)**；

(5) 对非周期的**pattern**而言，最坏情况下需要**3*n***次字符比较。

3.6 进阶导读

- 在C++的STL库中对字符串进行了封装[1]，微软的ATL/MFC中将其封装为CStringT/CString[2]。最近，英特尔在其最新的CPU中增加了字符串处理的硬件，以加速字符串处理的效率[3]。
- 字符串匹配算法可分为两类：精确匹配和近似匹配。
  - 本章所介绍的模式匹配算法属于精确匹配。

本章小结

- 知识点
  - 字符串基本操作
  - **KMP及其优化算法**

课程习题

- 笔做题——**3.4, 3.5, 3.7**（以作业形式提交）
- 上机题——**3.10**
- 思考题——**3.2, 3.6, 3.9**

Any Suggestion or Question

联系方式：

- 张玥杰

Email: yjzhang@fudan.edu.cn