

# 第三章

## 无失真数据压缩

# 主要内容

- **3.1 数据压缩的理论极限与基本途径**
- **3.2 统计编码方法**
  - **3.2.1 霍夫曼编码**
  - **3.2.2 算术编码**
  - **3.2.3 词典编码**
  - **3.2.4 游程编码**
  - **3.2.5 Golomb编码**

# 3.1 数据压缩的理论极限与基本途径

- 3.1.1 离散无记忆信源及其压缩基本途径
- 3.1.2 联合信源及其压缩基本途径
- 3.1.3 随机序列信源及其压缩基本途径

# 3.1.1 离散无记忆信源及其压缩基本途径

## 数据压缩的基本途径之一——概率匹配

### ■ 离散无记忆信源的编码理论

### ■ 二进制编码的平均码长

$$l = E(L_j) = \sum_{j=1}^m p_j L_j \geq H(X)$$

熵是离散无记忆信源编码进行无失真编码的下界。

### ■ 对离散无记忆平稳信源，必须：

- ① 准确得到字符概率；
- ② 对各字符的编码长度都达到它的自信息量。

## 3.1.2 联合信源及其压缩基本途径

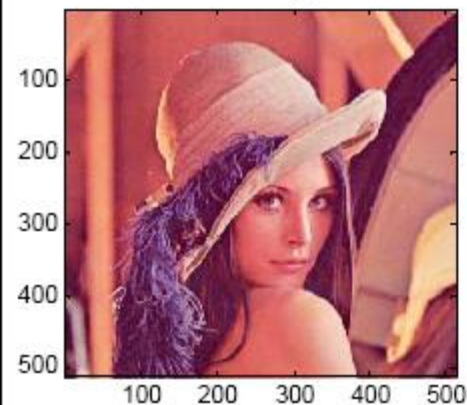
### ■ 对联合信源：

- ① 其冗余度隐含在信源间的**相关性**中，通常不宜直接对各相关分量进行编码；
- ② 尽量去除各分量间的相关性，再对各独立的分类进行编码。

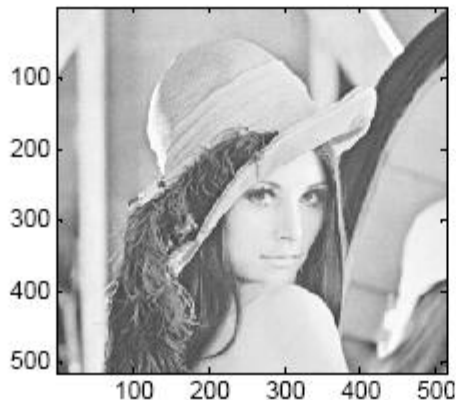
例：彩色空间转换、变换编码

# 例：颜色变换

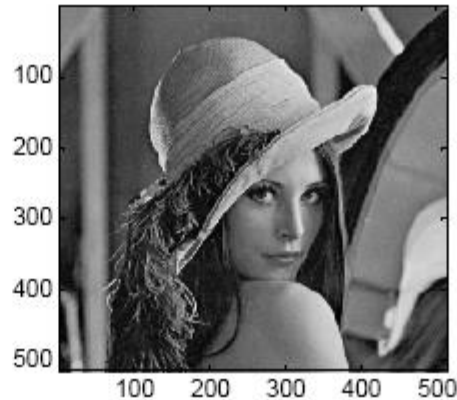
Standard Image 'Lena'



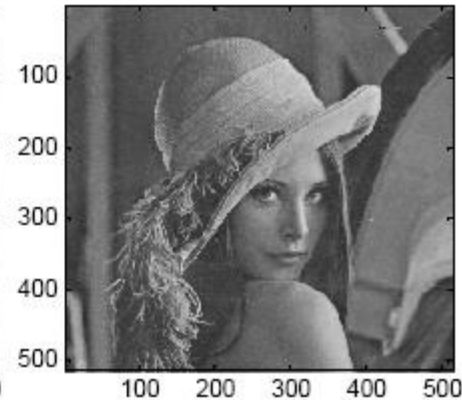
Red R



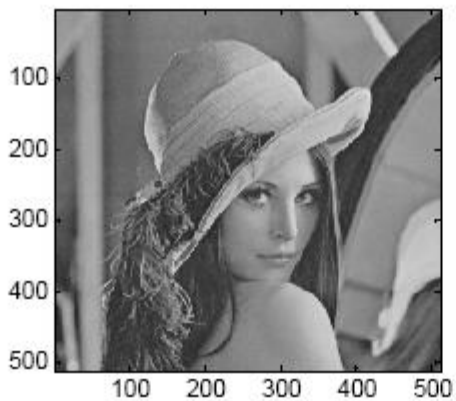
Green G



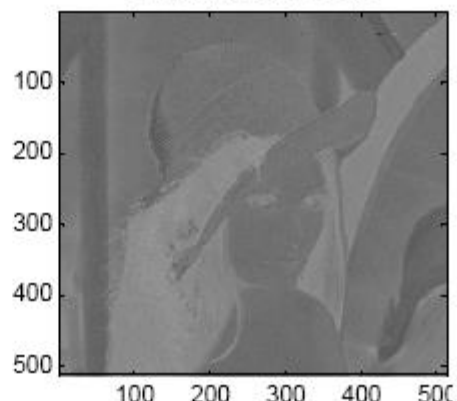
Blue B



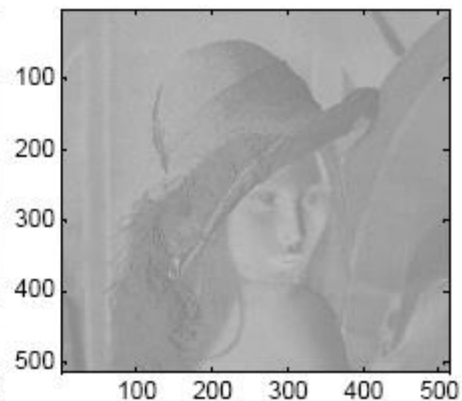
Luminance Y



Chrominance Cb

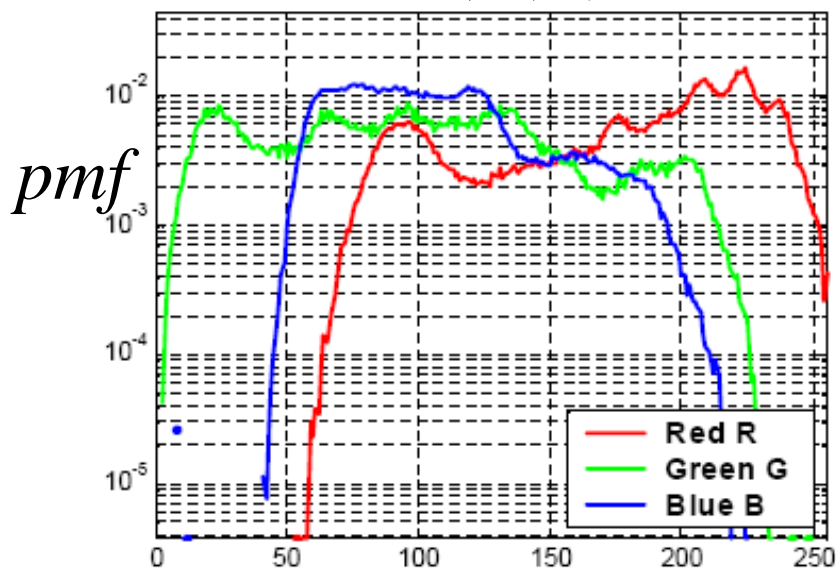


Chrominance Cr



# 例：颜色变换

*RGB*像素值

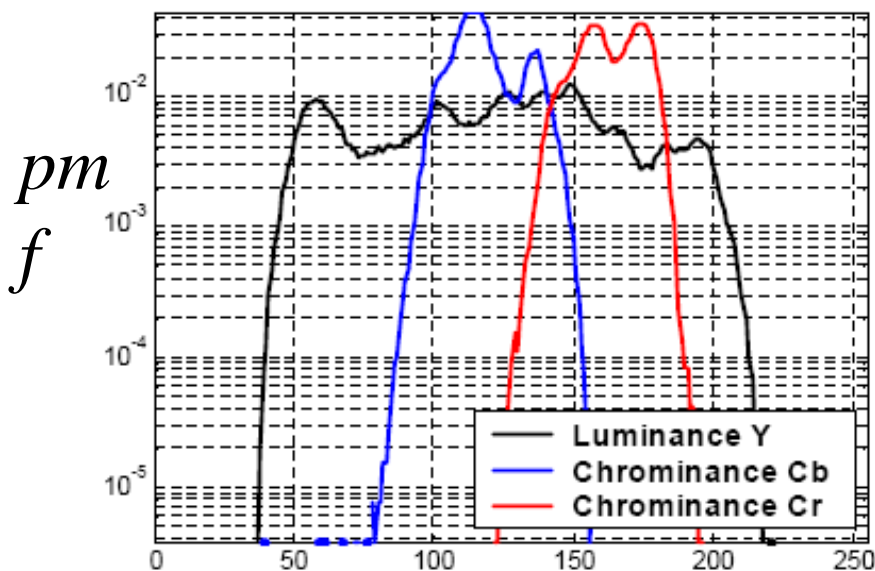


$$H(R)=7.25 \text{ bpp}$$

$$H(G)=7.59 \text{ bpp}$$

$$H(B)=6.97 \text{ bpp}$$

*YCbCr*像素值



$$H(Y)=7.23 \text{ bpp}$$

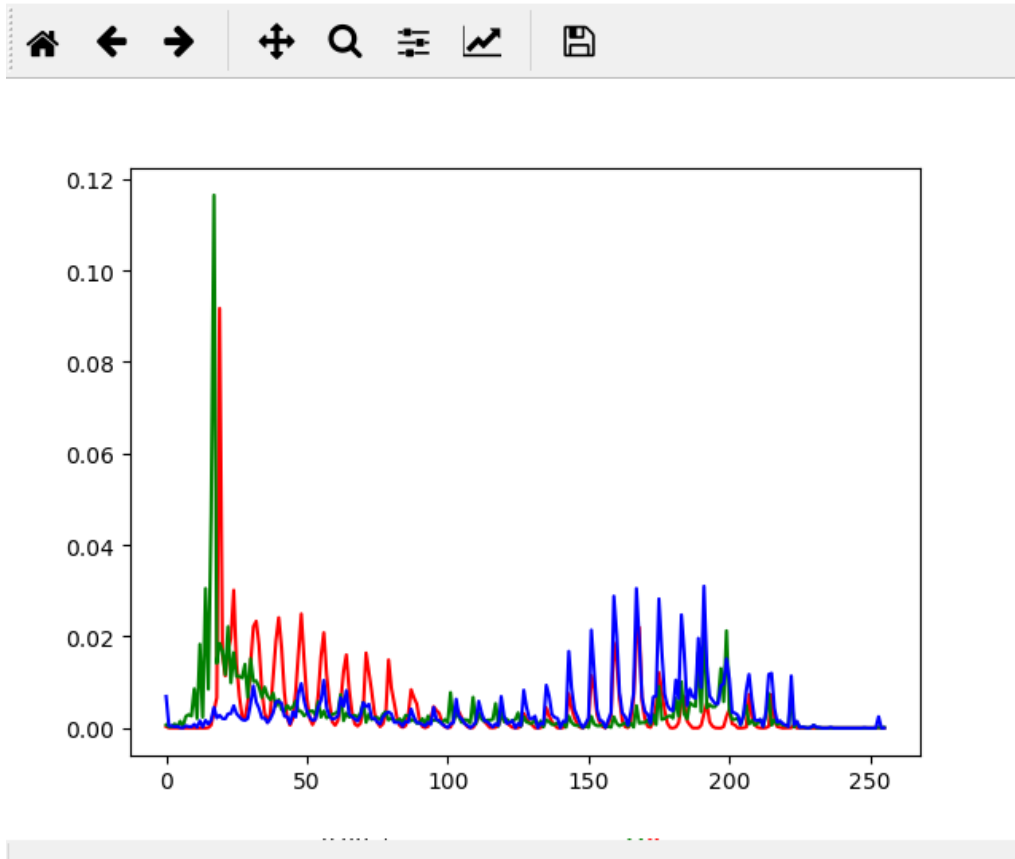
$$H(Cb)=5.47 \text{ bpp}$$

$$H(Cr)=5.42 \text{ bpp}$$

# 例：颜色变换



Figure 1





### 3.1.3 随机序列信源及其压缩基本途径

□ 数据压缩的基本途径之三 — 利用条件概率

对于有记忆信源：

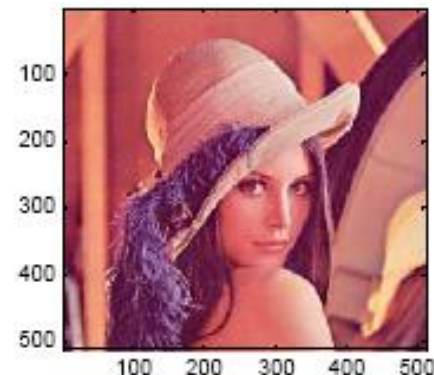
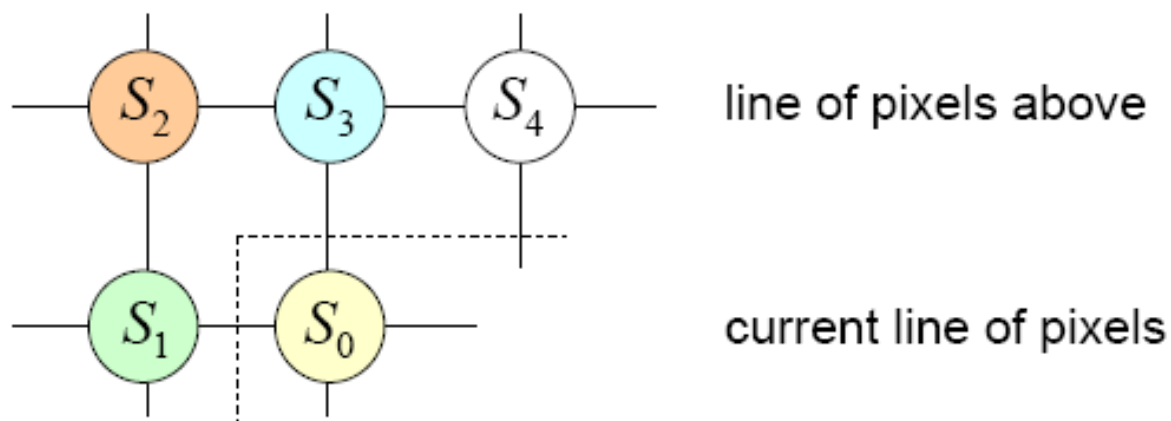
- ① 理论上，可以通过它的条件概率计算极限熵；
- ② 实践中，可利用条件概率进行编码，阶越高越有利。

预测编码的基础

# 例：语音编码

- 1、等长PCM编码：传送  $H_0$
- 2、如果测得各信源符号出现的概率，并利用无记忆信源的统计编码，就可能降到  $H_1$ 
  - 信息变差  $I_{01} = H_0 - H_1$  就是统计了信源符号出现概率获得的信息
- 3、进一步考虑利用前一取样值进行预测（DPCM），可望降到  $H_2$
- 若利用前M个取样值，则可能降到  $H_{M+1}$

# 例：图像的条件熵



component	$H(S_0)$	$H(S_0   S_1)$	$H(S_0   S_3)$	$H(S_0   S_2)$
Y	7.23	4.67	4.32	4.86
Cb	5.47	3.80	3.58	3.85
Cr	5.42	3.69	3.55	3.82

- 对实际图像，通常2阶熵比一阶熵小得多，3阶熵又比二阶熵小，而更高阶熵的减小就不明显了，但计算量却增加很快。
- 因此在实际分析中，以能看到  $H_n(X) \sim n$  曲线出现平稳趋势即可。

### 3.1.3 随机序列信源及其压缩基本途径

□ 数据压缩的基本途径之四 — 利用联合概率

■ 对离散平稳信源：

① 理论上，可以通过联合概率计算极限熵；

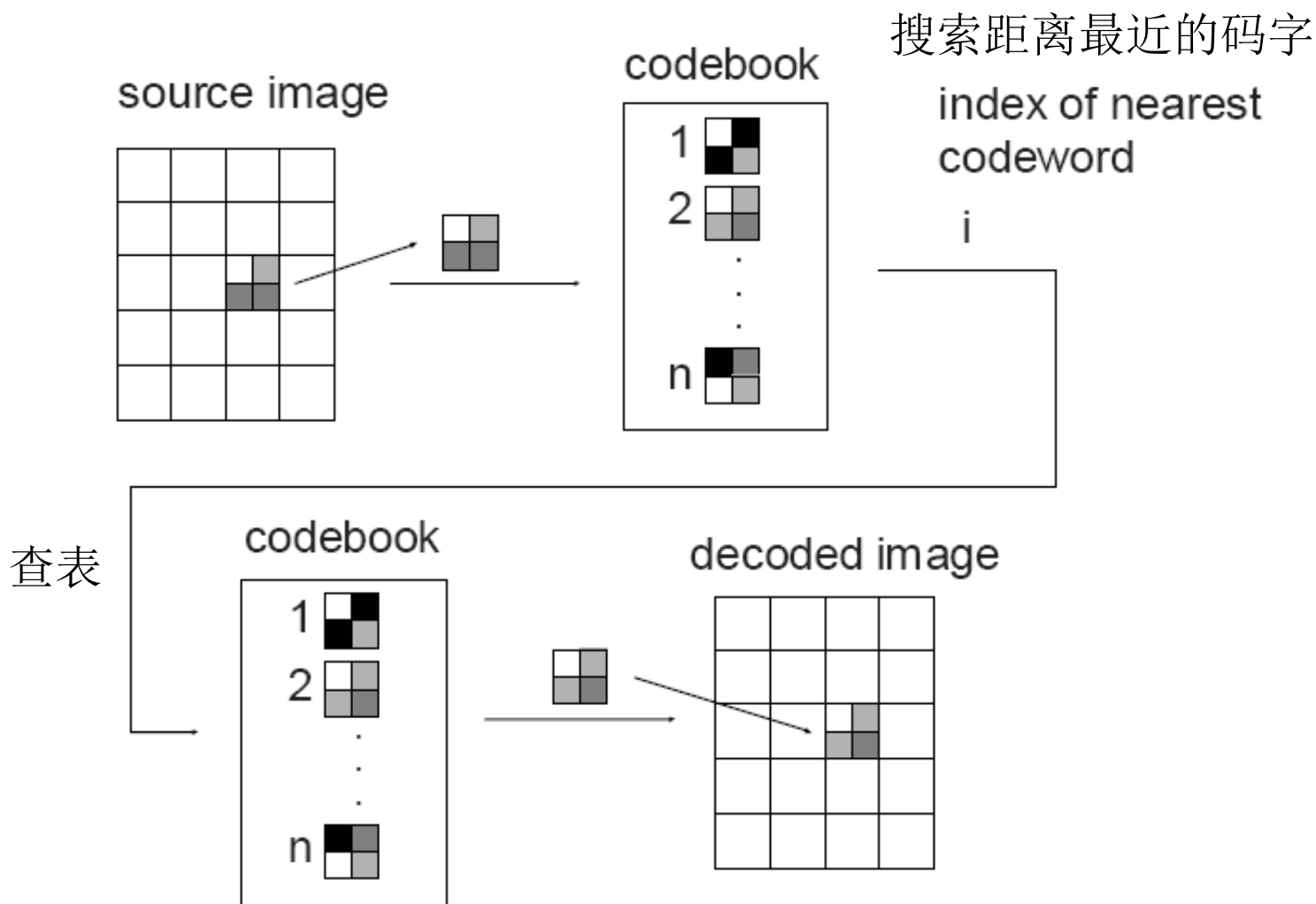
② 实践中，可将多个符号合并成向量，利用其联合概率进行编码，符号越多越高越有利。

$$H_n(X) = \frac{1}{n} H(X) = \frac{1}{n} H(X_1 X_2 \dots X_n)$$

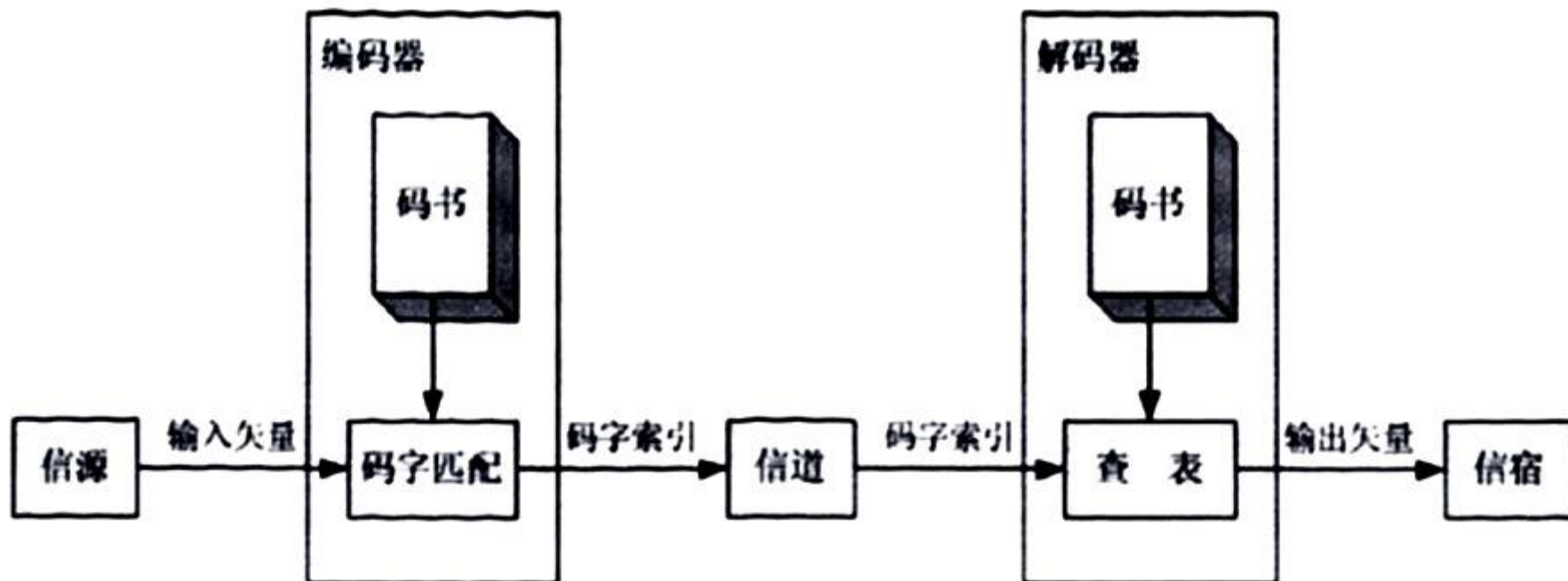
如矢量量化

### 3.1.3 随机序列信源及其压缩基本途径

#### ■ 以图像的矢量量化编码为例



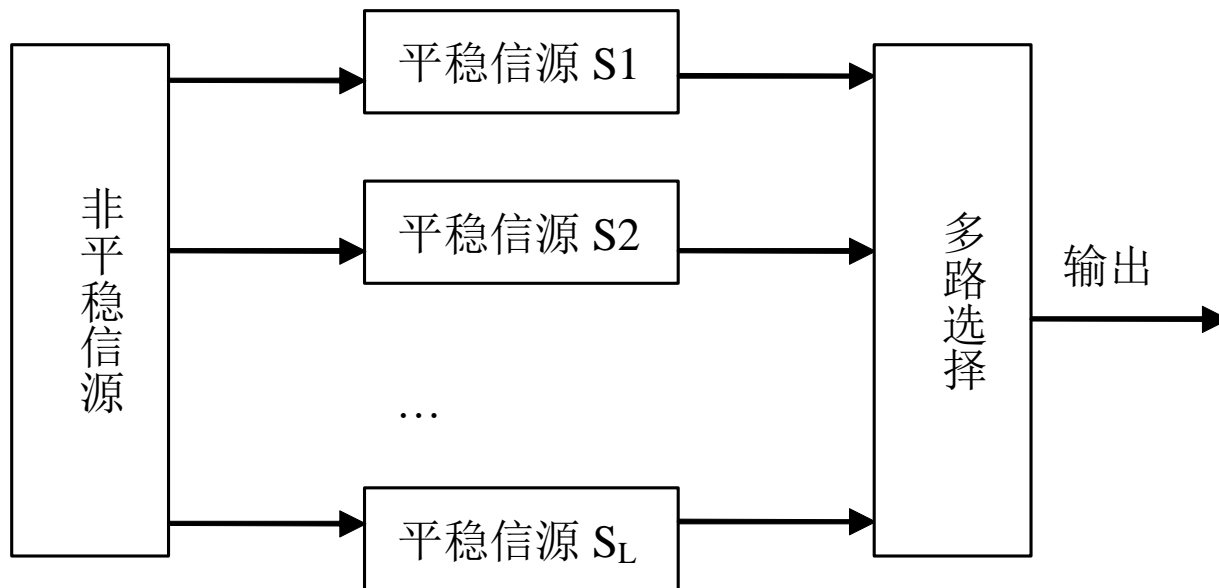
### 3.1.3 随机序列信源及其压缩基本途径



### 3.1.3 随机序列信源及其压缩基本途径

- 语音、图像等信源的统计性质一般是非平稳的，但在一定时间段内作为平稳信源对待还是合理的，如语音的局部平稳段、图像的平坦区域。
- 因此为了处理方便，可以将非平稳信源 $S$ 看成是由多个平稳子带的组合信源。

$S_i$



### 3.1.3 随机序列信源及其压缩基本途径

- 令组合信源中 $L$ 个子信源的符号集相同, 均为  $A_m = \{a_1, a_2, \dots, a_m\}$   
设各子信源在 $\mathbf{S}$ 中出现的概率为  $P_i$ , 此时组合信源模型描述为
$$\begin{cases} S_i(P_i; P(a_1|i), P(a_2|i), \dots, P(a_m|i)) \\ \sum_{i=1}^L P_i = 1 \end{cases}$$
- 该模型对应的熵称为**复合信源的熵**  $H_C$
- 实际信源近似看成是平稳信源, 它的统计是上述模型的平均
$$\begin{cases} S_M(P(a_1), P(a_2), \dots, P(a_m)) \\ P(a_j) = \sum_{i=1}^L P_i P(a_j|i) \end{cases}$$
- 对应的熵为**平均混合信源的熵**  $H_M$   
自适应编码定理: 一个信源复合模型的熵  $H_C$  小于混合模型的熵  $H_M$



### 3.1.3 随机序列信源及其压缩基本途径

- 数据压缩的基本途径之五——对平稳子信源进行编码
  - 对离散非平稳信源：
    - ① 设法将其划分为若干个近似平稳的子信源分别编码；
    - ② 提高编码效率的关键是对平稳子信源的自适应识别。
- 这是子带编码的基础。

# 3.1 数据压缩的理论极限与基本途径

## ■ 3.1.1 离散无记忆信源及其压缩基本途径

- 非等概率→统计匹配：[统计编码](#)

## ■ 3.1.2 联合信源及其压缩基本途径

- 不独立的各分量→独立的各分量：  
[彩色空间转换，变换编码](#)

## ■ 3.1.3 随机序列信源及其压缩基本途径

- 利用条件概率：[预测编码](#)
- 利用联合概率：[对多个符号合并编码，如矢量量化](#)
- 将非平稳信源转换为平稳信源：[分析/综合编码，分成多个平稳子带](#)

## 3.2 统计编码方法

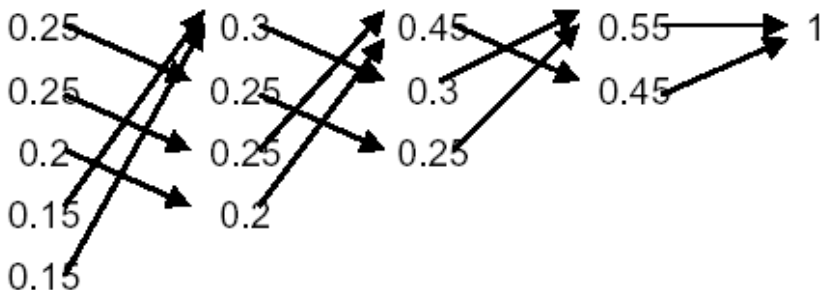
- 3.2.1 霍夫曼编码
- 3.2.2 算术编码
- 3.2.3 词典编码
- 3.2.4 游程编码
- 3.2.5 Golomb编码

## 3.2.1 霍夫曼编码

- 3.2.1.1 霍夫曼编码的实现（数据结构）
- 3.2.1.2 霍夫曼解码
- 3.2.1.3 规范霍夫曼编码
- 3.2.1.4 霍夫曼编码的局限性

Example:

Code word length	codeword	X	Probability			
2	01	1	0.25	0.3	0.45	0.55
2	10	2	0.25	0.25	0.3	0.45
2	11	3	0.2	0.25	0.25	
3	000	4	0.15	0.2		
3	001	5	0.15			



## 3.2.1.1 霍夫曼编码的实现

Huffman编码需要各符号的概率。根据概率来源的不同

- 静态统计模型：根据训练数据得到，然后不同信源利用同一个概率表
  - 无需传送Huffman码表/码树
  - JPEG图像编码器/MPEG/H.263视频编码器
- 半自适应统计模型：对每个输入信源符号，计算其概率
  - 必须传送Huffman码表和压缩流
- 自适应/动态统计模型：
  - 1遍扫描
  - 当频率改变时，改变Huffman码表/树
  - Unix系统对程序的压缩：基于单词的自适应Huffman编码

## 3.2.1.2 霍夫曼解码

- 在开始压缩之前，编码器必须根据符号的概率（或出现的频率）确定码字。解码器利用这些信息对压缩流进行解码，所以应该将**这些信息也写入压缩流中**，以便解码器能解码该压缩流
  - **变长码本身**： 不好，因为码长不同
  - **写入频率/概率**：

将频率/概率标定为整数，然后解码器利用该频率信息构造Huffman树→通常只会在压缩流中增加几百字节

- **写入Huffman树**？

可能比只写频率花费的字节数更多

## 3.2.1.2 霍夫曼解码

- 一. **比特串行解码**（固定输入比特率——可变输出符号率(假定二进制码树在解码器端可得)
  - 实际应用中，从符号-码字的映射表中构建Huffman码树。此表对编码器和解码器都是已知的。
  - **1.** 逐个比特读入输入压缩流，并遍历树，直到到达一个叶节点。
  - **2.** 输入流的每个比特用过后即丢弃。当到达叶节点时，Huffman解码器输出叶节点处的符号，即完成该符号的解码。
  - **3.** 重复步骤1和2，直到所有的输入都解完。
- 由于所有码字长度不同，解码比特率对所有符号并不相同。

## 3.2.1.2 霍夫曼解码

- 二. **基于查找表的解码**: 对所有符号解码率恒定
  - 在解码端从符号-码字映射表中构建查找表. 如果表中的最长码字为 $L$ 比特, 则需要一个 $2^L$ 个入口的查找表. 空间限制: 图象/视频最长的 $L=16-20$
  - **1. 构造查找表**:
    - 设 $S_i$ 对应的码字为 $C_i$ , 假定 $C_i$ 有 $l_i$ 比特. 构造一个 $L$ 比特地址, 前 $l_i$ 个比特是 $C_i$ , 后 $L-l_i$ 个比特取任意0和1的组合. 所以, 对符号 $S_i$ , 有 $2^{L-l_i}$ 个地址.
    - 在每个入口形成 $(S_i, l_i)$



## 3.2.1.2 霍夫曼解码

### ■ 二.基于查找表的解码:

#### ➤ 2.解码步骤:

- 从压缩输入比特流中, 读 $L$ 个比特到缓冲区中;
- 将缓冲区中的 $L$ 比特字作为查找表中的地址, 得到对应的符号 $S_k$ 。令码字长度为 $l_k$ , 即解码一个符号.
- 移出缓冲区中的前 $l_k$ 个比特, 再移入若干比特, 使得缓冲区重新为 $L$ 个比特。
- 重复步骤2和3, 直到所有的符号都解码.

### 3.2.1.3 规范霍夫曼编码

- 并非只有使用二叉树建立的即时码才是 Huffman 编码，只要符合
  - (1) 是即时码
  - (2) 某一字符编码长度和使用二叉树建立的该字符的编码长度相同

这两个条件的编码都可以叫做 Huffman 编码

#### ■ 规范霍夫曼编码的提出

- 小量数据的压缩，霍夫曼码表的存储是问题
- 使用某些强制的约定，仅通过很少的数据便能重构出霍夫曼编码树的结构——**数据结构化问题**

# 3.2.1.3 规范霍夫曼编码

## ■ 编码步骤:

- 统计每个符号的频率并求出该符号所需的位数/编码长度
- 统计从最大编码长度到1的每个长度对应多少个符号，然后为每个符号递增分配编码
  - 码字长度最小的第一个编码从0开始
  - 长度为*i*的第一个码字 $f(i)$ 能从长度为*i-1*的最后一个码字得出, 即:  $f(i) = 2(f(i-1)+1)$ 。假定长度为4的最后一个码字为1001，那么长度为5的第一个码字便为10100
- 编码输出压缩信息

## 3.2.1.3 规范霍夫曼编码

- [例] 符号:  $a\ b\ c\ d\ e\ f\ g\ h\ i\ j\ k\ \dots\ u$   
码长:  $3\ 4\ 4\ 4\ 4\ 4\ 4\ 4\ 4\ 4\ 5\ 5\ \dots\ 5$

$a=000(0); \quad f=0110(6); \quad k=10101(21);$   
 $b=0010(2); \quad g=0111(7); \quad \dots$   
 $c=0011(3); \quad h=1000(8); \quad u=11111(31);$   
 $d=0100(4); \quad i=1001(9);$   
 $e=0101(5); \quad j=10100(20);$

其中  $\text{first}[3] = 0$ ,  $\text{first}[4] = 0010_{\text{b}} = 2$ ,  $\text{first}[5]$   
 $= 10100_{\text{b}} = 20$

$\text{first}[3] = 0, \text{first}[4] = 0010b = 2, \text{first}[5] = 10100b = 20$

输入码流: 0010 0110 10101

```
extern KBitInputStream bs;
```

```
int len = 1;
```

```
int code = bs.ReadBit();
```

```
while(code >= first[len])
```

```
{
```

```
    code << 1;
```

```
    code |= (bs.ReadBit()); // append next input bit to code
```

```
    len++;
```

```
}
```

```
len--;
```

```
// 至此,识别出了一个即时码
```

所有长于 $l$ 位的码字的前 $l$ 位前缀  
大于 $\text{first}[l]$

$\text{index}[3] = 0, \text{index}[4] = 1, \text{index}[5]$   
 $= 9$

下面将 $\text{code}$ 解码为其对应的符号  
 $\text{sym}$

```
int sym_index = index[len] + (code -  
first[len]); // 计算第几个码字
```

```
int sym = table[sym_index];
```

$\text{Table}[0] = a, \text{Table}[1] = b, \text{Table}[2] = c \dots$

# Huffman表存储方式举例说明-JPEG

- FF C4 00 1D **00** **00** **03** **01** **01** **01** **01** **01** **01** **01** **00** **00** **00**  
**00** **00** **00** **04** **05** **06** **03** **02** **01** **00** **09** **07** **08**
- **红色部分** 为哈夫曼表ID和表类型，其值0x00表示此部分数据描述的是DC直流0号表。
- **蓝色部分**（16个字节）为不同位数的码字的数量。这16个数值实际意义为：没有1位的哈夫曼码字；2位的码字有3个；3位-9位的码字各有1个；没有9位或以上的码字。
- **绿色部分**（10个字节）为编码内容。由蓝色部分数据知道，此哈夫曼树有 $0+3+1+1+1+1+1+1+1=10$ 个叶子结点，即本字段应该有10个字节。这段数据表示10个叶子结点按从小到大排列，其权值依次为**04**、**05**、**06**、**03**、**02**、**01**、**00**、**09**、**07**、**08**（16进制）
- 对**大字母表且必须快速解码**时，规范Huffman编码特别有用
  - 解码器很容易逐个读入和检测输入位来识别码长
  - 知道码长，即可进一步解码读出符号

# 3.2.1.3 规范霍夫曼编码

- 建立一种**规范的码字结构**
  - 码字的生成规范，与码长的对应关系明确
- 可以**不依赖任何树结构**进行高速解压缩
- 而且在整个压缩、解压缩过程中需要的空间比传统 Huffman 编码少得多
  - 码表的存储大大降低

# 3.2.1.4 霍夫曼编码的局限性

- Huffman编码的平均码长满足： [gallagher 1978].

$$H(S) \leq L_{avg} < H(S) + p_{max} + 0.086$$

- $p_{max}$ : 信号符号集中的最大概率

- 当符号集较大且概率分布不是很悬殊时，Huffman的平均码长可接近于熵
- 对于概率分布悬殊的消息集合，上限可能很大，意味着编码剩余度较大

## ■ 一个极端的例子:

- 仅有两个信源符号：一个概率很小（接近于0），另一个概率很大（接近于1）
  - 信源熵接近于0，理论上每个符号需要接近0个比特
  - 进行Huffman编码，对于每个符号需要1个比特
    - 编码效率的降低是因为Huffman编码只能为每个信源符号分配整数个比特



## 3.2.1.4 霍夫曼编码的局限性

- 为实现统计匹配，可对扩展信源进行 Huffman 编码，并使平均码长下降

➤ 代价是必须计算出所有N长信源序列的概率分布，并构造相应的完整的码树，相当复杂，码表也相当大

信源符号	发生概率	编码
<b>S1</b>	<b>0.9</b>	<b>0</b>
<b>S2</b>	<b>0.1</b>	<b>1</b>

信源符号	发生概率	编码
<b>S1S1</b>	<b>0.81</b>	<b>0</b>
<b>S1S2</b>	<b>0.09</b>	<b>10</b>
<b>S2S1</b>	<b>0.09</b>	<b>110</b>
<b>S2S2</b>	<b>0.01</b>	<b>111</b>
	平均码长	<b>0.645</b>

## 3.2.2 算术编码

- 3.2.2.1 算术编码的基本思路
- 3.2.2.2 自适应算术编码

## 3.2.2.1 算术编码的基本思路

- 算术编码是从另一种角度对很长的信源符号序列进行有效编码的方法 – 非分组码
  - **Stream-based**，直接将信源符号序列编码为码字序列，突破每个信源符号必须使用整数个比特的限制，从而更加有效
  - 将**编码和建模分离**，可动态修改概率模型而不影响编码器的设计

## 3.2.2 算术编码的基本思路

### ■ 基本思想

- 是**香农编码思想**的扩展
- 把**信源输出序列**和实数段 $[0, 1)$ 中的一个数 $p$ 联系起来
  - 计算信源符号序列的累积概率（Cumulative Probability, CP）函数
  - 每个符号序列对应于累积概率函数上的不同区间
  - 在区间内取一点，将其二进制小数点后 $L$ 位作为这符号序列的码字

## 3.2.2 算术编码的基本思路

将区间 $[0, 1)$ 划分为子区间

### ■ 累积概率（CP）

➤ 和概率论中的累积分布函数稍有不同

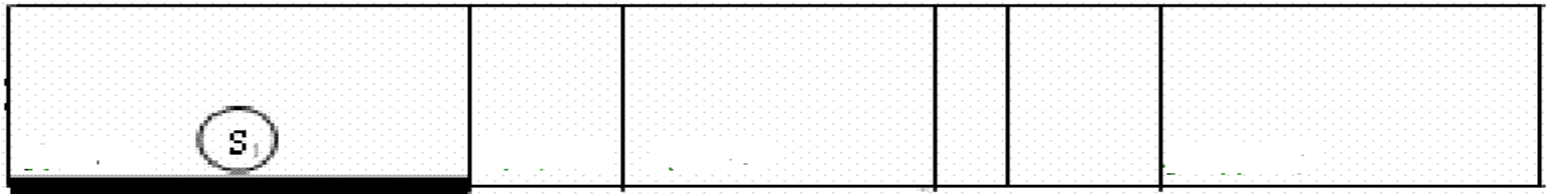
$$CDF(s_i) = \sum_{j=1}^i p(s_j)$$

$$CP(s_i) = \sum_{j=1}^{i-1} p(s_j)$$

其中  $CP(s_1) = 0$

信源符号	发生概率	对应子区间	累积概率 $CP(S_i)$
<b>S1</b>	<b>0.3</b>	<b>[0, 0.3)</b>	<b>0</b>
<b>S2</b>	<b>0.1</b>	<b>[0.3,0.4)</b>	<b>0.3</b>
<b>S3</b>	<b>0.2</b>	<b>[0.4,0.6)</b>	<b>0.4</b>
<b>S4</b>	<b>0.05</b>	<b>[0.6,0.65)</b>	<b>0.6</b>
<b>S5</b>	<b>0.1</b>	<b>[0.65,0.75)</b>	<b>0.65</b>
<b>S6</b>	<b>0.25</b>	<b>[0.75,1)</b>	<b>0.75</b>

0.3      0.4                      0.6   0.65    0.75                      1.0



子区间的长度表示  
该符号的发生概率

子区间的左端点表示该  
符号的累积概率

## 3.2.2 算术编码的基本思路

如何计算信源符号序列的累积概率函数？

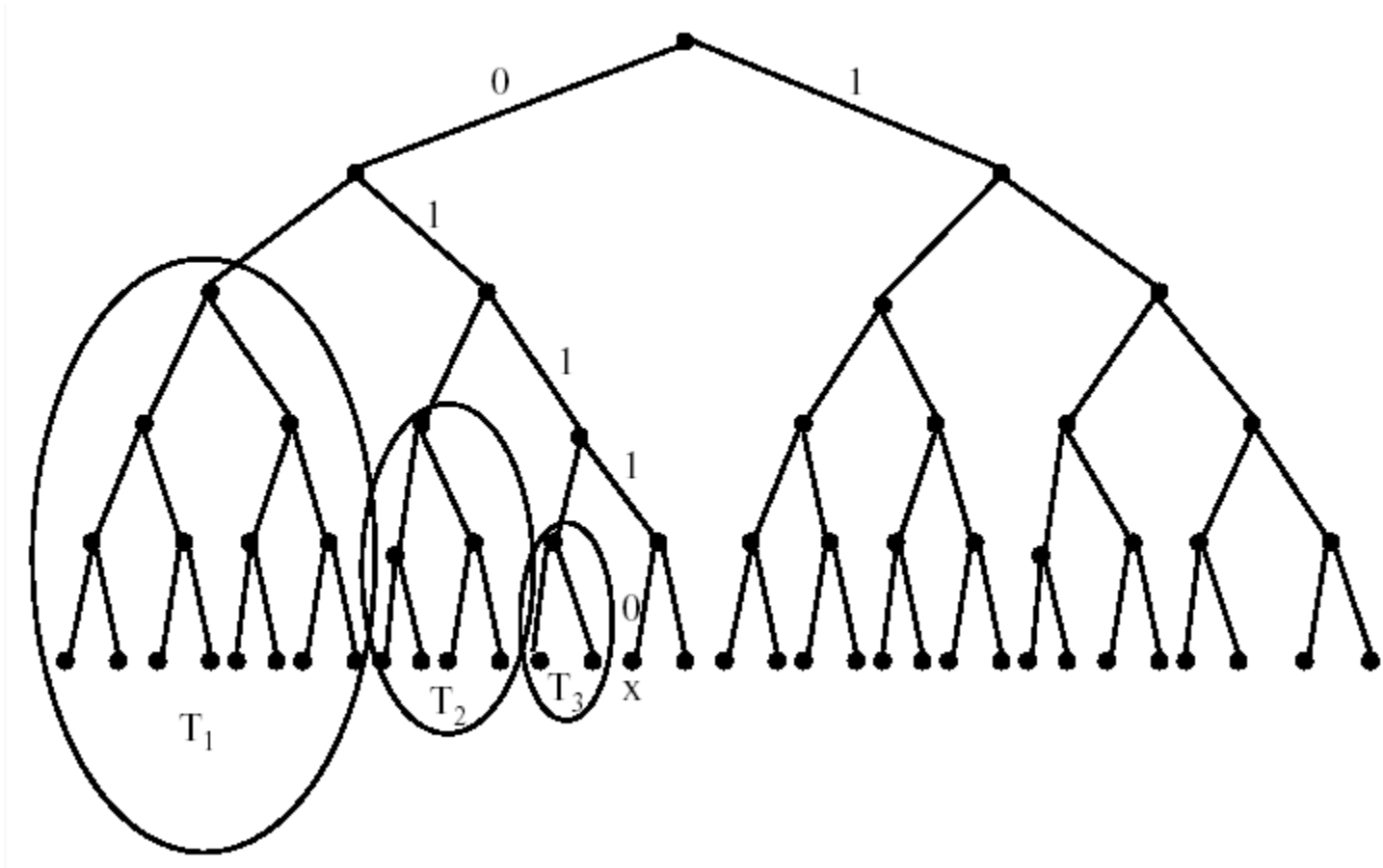
- 假定信源符号集是二进制的，即{0, 1}
- 定义一个符号串  $x > y$ ，须满足

$$x > y \quad \text{if} \quad \sum_i x_i 2^{-i} > \sum_i y_i 2^{-i},$$

i.e., if the corresponding binary number satisfy

$$0.x > 0.y$$

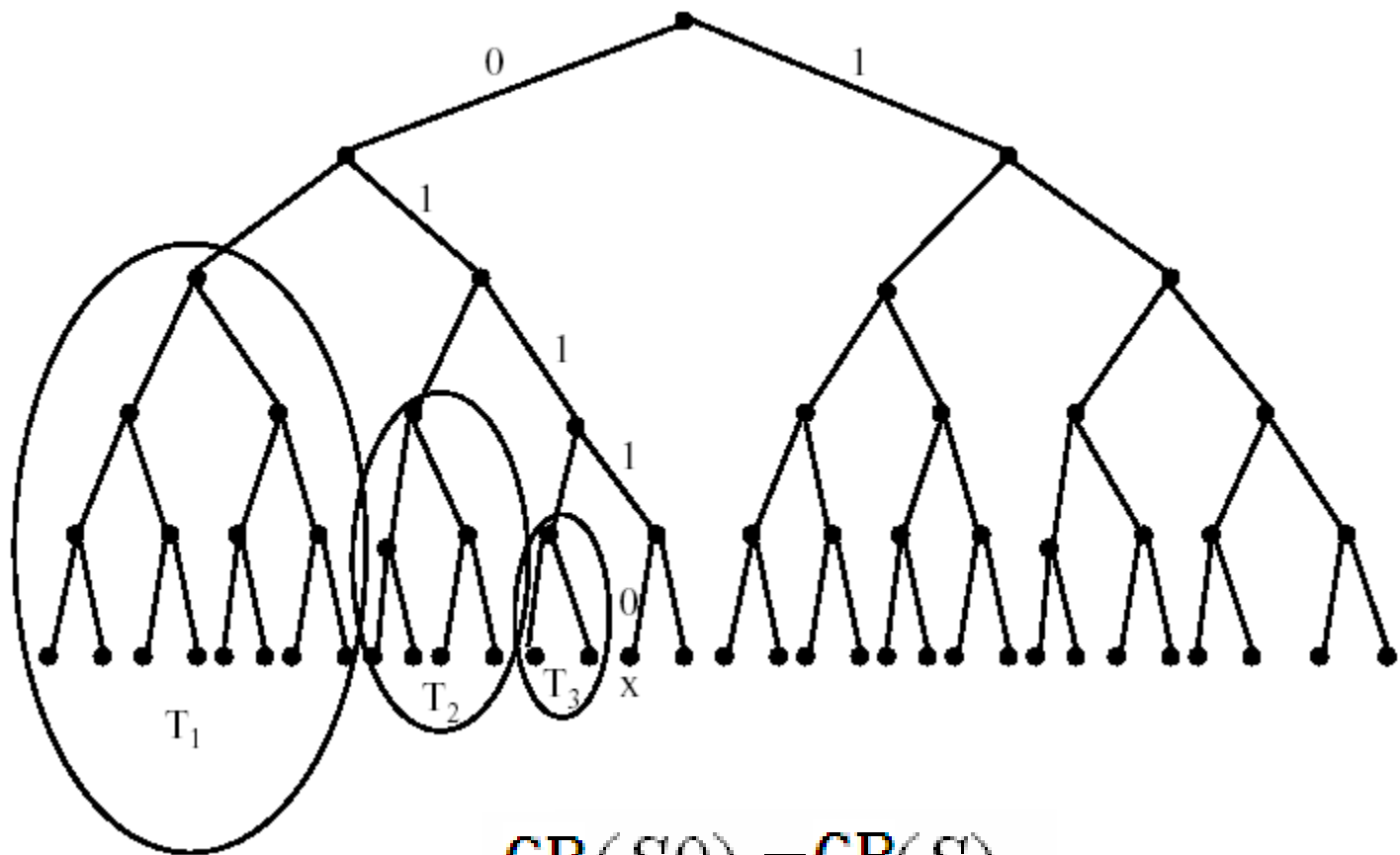
- 将符号串（信源符号序列）视作二叉树中深度为n的树叶节点，树中的每一级对应一个符号



- 在上树中， $x > y$ 意味着在树的同一级上 $x$ 在 $y$ 的右边
- 符号序列 $x$ 的累积概率是其左侧所有树叶节点概率的和
- $X_n$ 左侧所有树叶节点的概率和是 $X_n$ 左侧所有子树的概率和







$$\text{CP}(S0) = \text{CP}(S)$$

$$\text{CP}(S1) = \text{CP}(S) + P(S) \bullet P(0)$$

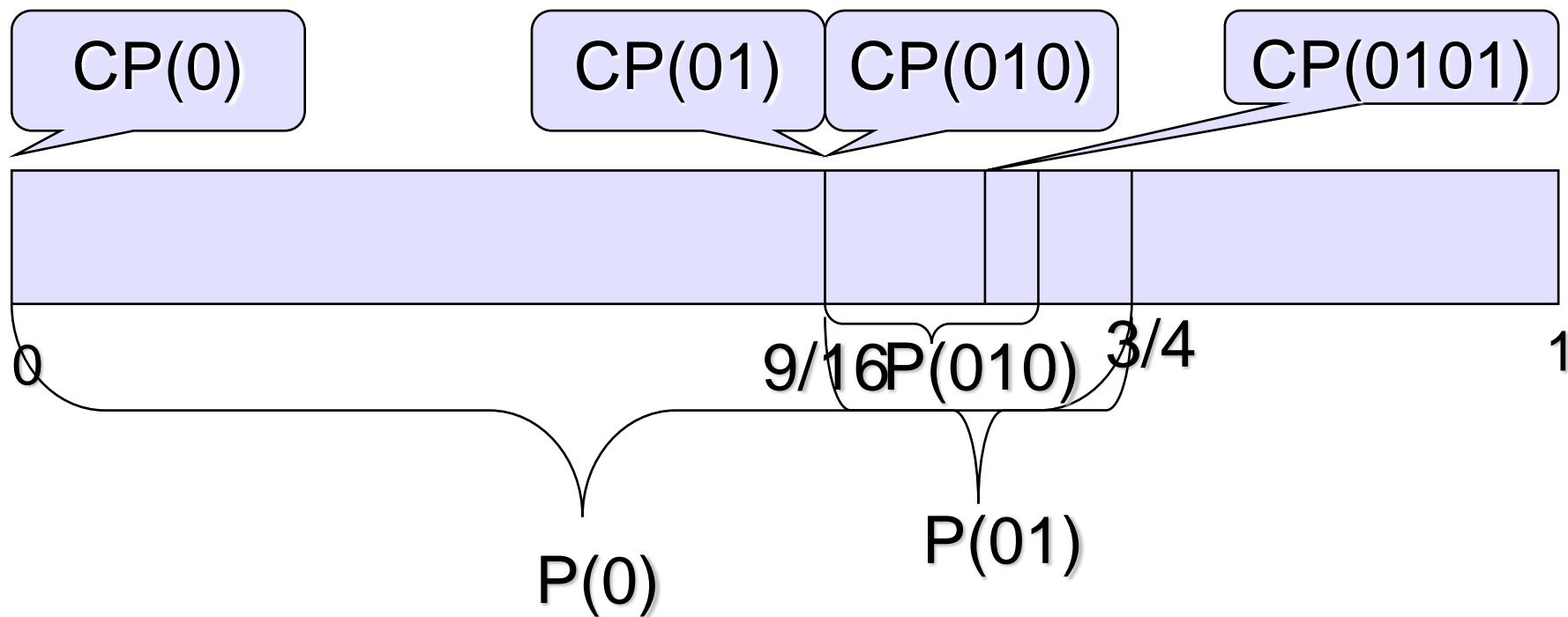
$$\text{CP}(Sr) = \text{CP}(S) + P(S) \bullet \text{CP}(r) \quad (r = 0,1)$$

信源符号序列对应的宽度

$$A(Sr) = P(Sr) = P(S) \bullet P(r) \quad (r = 0,1)$$

算术编码示例：

0 1 0 1



## 3.2.2 算术编码的基本思路

用区间内的一点表示信源序列

- $CP(S)$ 把区间 $[0,1)$ 划分为许多小区间
- 不同的信源序列对应于不同的区间  
 $[CP(S), CP(S)+P(S))$
- 将区间的左端点和右端点写成二进制小数，取小数点后 $L$ 位数字作为信源序列的码字
- $L$ 取多少？

## 3.2.2 算术编码的基本思路

- 将符号序列的累积函数写成二进位的小数，取小数点后L位，若后面还有尾数，就进位到第L位。令L满足：

$$L = \left\lceil \log \frac{1}{P(S)} \right\rceil$$

例：CP (S) = 0.10110001    P(S) = 1/7    则L=3

➤ 得C=0.110，S的码字为110。

这样选取的数值C，一般根据二进位小数截去位数的影响，得： $C - CP(S) \leq \frac{1}{2^L}$

当CP (S) 在L位后没有尾数时，C = CP (S)

而  $P(S) \geq \frac{1}{2^L}$

而信源符号序列S对应区间的上界  $CP(S) + P(S) \geq CP(S) + \frac{1}{2^L} \geq C$

所以，数值C在区间[CP (S) , CP (S) +P (S) ) 内。而信源符号对应的不同区间（左闭右开区间）是不重叠的。所以编得的码是即时码

## 二、具体编码方法

例：二元无记忆信源  $S=\{0, 1\}$ ,  $P(0)=1/4$ ,  $P(1)=3/4$ 。对二元序列  $S=11111100$  进行算术编码。

初始： $P(0)=2^{-2}$ ,  $F(0)=0$ ,  $F(1)=P(0)=2^{-2}$ 。计算机中乘以  $2^{-Q}$  用右移  $Q$  位取代；乘以  $1-2^{-Q}$  用此数减去右移  $Q$  位数来取代。

输入符号	$P(S)=A(S)$	$F(S)$	$L(S)$	$C$
空	1	0	0	
1	0.11	0.01	1	0.1
1	0.1001	0.0111	1	0.1
1	0.011011	0.100101	2	0.11
1	0.0101001	0.10101111	2	0.11
1	0.0011110011	0.1100001101	3	0.111
1	0.001011011001	0.110100100111	3	0.111
0	0.00001011011001	0.110100100111	5	0.11011
0	0.0000001011011001	0.110100100111	7	0.1101010

可得： $S$  的码字为 1101010。此时  $S=11111100$  对应的区间为  $[0.110100100111, 0.1101010101001001)$ 。可见  $C$  是区间内一点。

### 三、具体解码方法

解码是一系列比较过程。每一步比较  $C-F(S)$  与  $P(S)P(0)$ 。其中：

$S$  为前面已解出的序列串，得  $P(S)$  为序列串  $S$  对应的宽度； $F(S)$  是序列串  $S$  的累积分布函数，即为  $S$  对应区间的下界。 $P(S)P(0)$  是此区间下一个输入为符号“0”所占的子区间宽度。

若  $C-F(S) < P(S) \cdot P(0)$  则解得输出符号为“0”；

若  $C-F(S) > P(S) \cdot P(0)$  则解得输出符号为“1”。

$C$	$F(S)$	$C-F(S)$	$P(S)$	$P(S) \cdot P(0)$	解码输出
-----	--------	----------	--------	-------------------	------

## 3.2.2.1 算术编码的基本思路

- 小结：与Huffman编码相比，算术编码是一种非分组码
  - 不需要对每个信源符号编码并送出结果
  - 用0到1之间的小数表示符号序列
  - 以递推（ [incremental](#) ）的方式实现。当输入符号持续进入到编码器时，此小数会跟着改变，任何待压缩信源符号序列都可以用独一无二的小数表示
  - 根据目前输入的符号，不断缩小编码区间
    - 符号序列的概率越大，对应的区间长度越大，表示其所需要的比特数越少



## 3.2.2.1 算术编码的基本思路

### ➤ 特点

- 当消息长度接近无限时可逼近最优编码
- ✓ 假定编码器和译码器都知道消息的长度，因此译码器的译码过程不会无限制地运行下去。实际上在译码器中需要添加一个专门的终止符，当译码器看到终止符时就停止译码。
- ✓ 由于实际的计算机的精度不可能无限长，运算中出现溢出是一个明显的问题，但多数机器都有16-、32-或者64位的精度，因此这个问题可使用**比例缩放方法**解决。  
算术编码也是一种对错误很敏感的编码方法，如果有一位发生错误就会导致整个消息译错。

## 3.2.2.2 自适应算术编码

- 统计编码技术需要利用信源符号的概率，获得这个概率的过程称为建模。不同准确度（通常也是不同复杂度）的模型会影响算术编码的效率。
- 建模的方式：
  - 静态建模：在编码过程中信源符号的概率不变。但一般来说事先知道精确的信源概率是很难的，而且是不切实际的。
  - 自适应动态建模：信源符号的概率根据编码时符号出现的频繁程度动态地进行修改。当压缩消息时，我们不能期待一个算术编码器获得最大的效率，所能做的最有效的方法是在编码过程中估算概率。
- 算术编码很容易与自适应建模相结合。

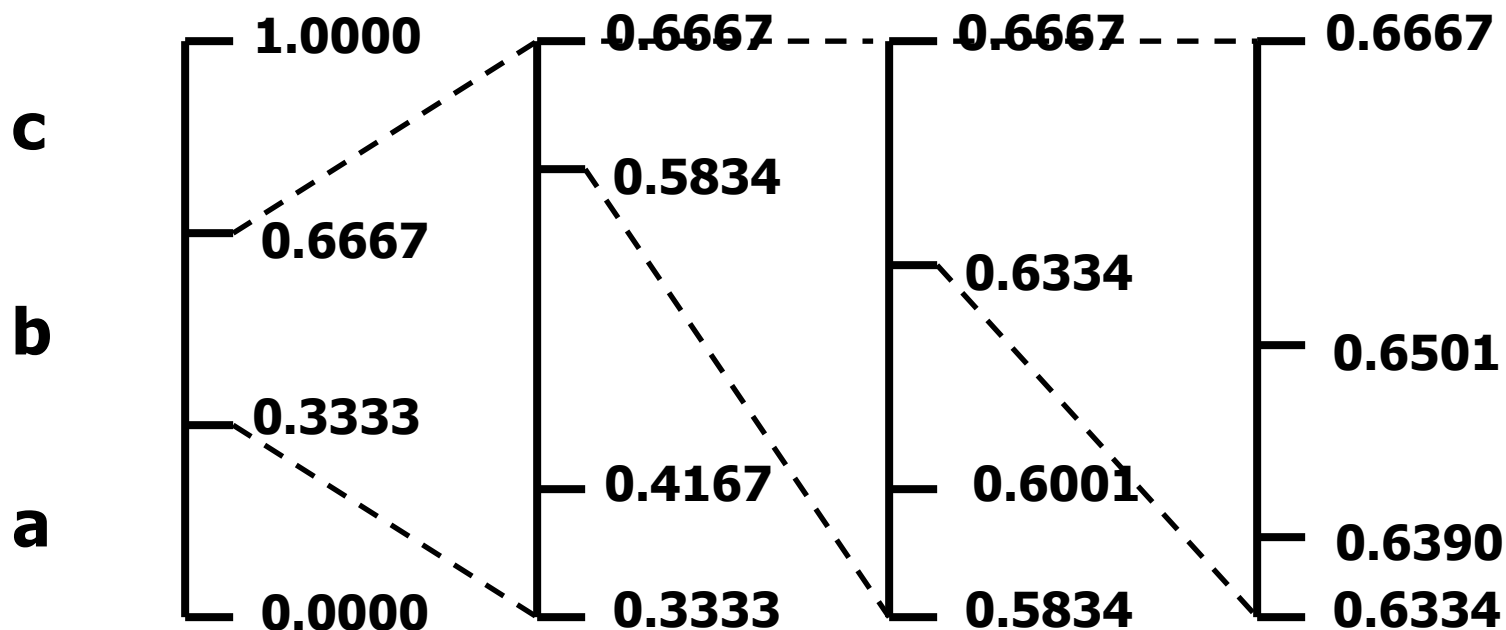
## 3.2.2.2 自适应算术编码

### ■ 自适应算术编码：

- 在编码之前，假设每个信源符号的频率相等（如都等于1），并计算累积频率
- 从输入流中读入一个字符，并对该符号进行算术编码
- 更新该符号的频率，并更新累积频率
- 由于在解码之前，解码器不知道是哪个信源符号，所以概率更新应该在解码之后进行
- 对应的，编码器也应在编码后进行概率更新

## 3.2.2.2 自适应算术编码

输入序列为: bcc.....



c	1/3	1/4	2/5	3/6
b	1/3	2/4	2/5	2/6
a	1/3	1/4	1/5	1/6

# History of arithmetic coding

- The idea of encoding by using cumulative probability in some ordering, and decoding by comparison of magnitude of binary fraction was introduced in Shannon's celebrated paper [1948].
- The recursive implementation of arithmetic coding was devised by Elias (another member in Fano's first information theory class at MIT).  
This unpublished result was first introduced by Abramson as a note in his book on information theory and coding [1963].
- The result was further developed by Jelinek in his book on information theory [1968].
- The growing precision problem prevented arithmetic coding from practical usage, however. The proposal of using finite precision arithmetic was made independently by Pasco [1976] and Rissanen [1976].

# History of arithmetic coding

- Practical arithmetic coding was developed by several independent groups [Rissanen 1979, Rubin 1979, Guazzo 1980].
- A well-known tutorial paper on arithmetic coding appeared in [Langdon 1984].
- The tremendous efforts made in IBM lead to a new form of adaptive binary arithmetic coding known as the Q-coder [Pennebaker 1988].
- Based on the Q-coder, the activities of JPEG and JBIG combined the best features of the various existing arithmetic coders and developed the binary arithmetic coding procedure known as the QM-coder [pennebaker 1992].

# Reading



- W. B. Pennebaker, J. L. Mitchell, G. G. Langdon, Jr., R. B. Arps, “An overview of the basic principles of the Q-Coder adaptive binary arithmetic coder,” IBM J. Res. Develop., vol. 32, no. 6, November 1988.
- Witten, Radford, Neal, Cleary, “Arithmetic Coding for Data Compression” Communications of the ACM, vol. 30, no. 6, pp. 520-540, June 1987.
- Moffat, Neal, Witten, “Arithmetic Coding Revisited,” ACM Transactions on Information Systems, vol. 16, vo. 3, pp. 256–294, July 1998.

## 3.2.3 词典编码

- 3.2.3.1 词典编码的基本思路
- 3.2.3.2 第一类词典编码
  - LZ77, LZSS
- 3.2.3.3 第二类词典编码
  - LZW



# 3.2.3.1 词典编码的基本思路

## ■ 通用编码的概念

许多场合无法观察信源的统计特性，或者可观察但统计特性有变化。对这些数据进行压缩编码的技术统称为通用编码技术。

➤ 常见的数据文件 – 文本文件、图象、源代码文件...

■ 数据本身包含有重复字符这个特性!

## ■ 词典编码的思路:

➤ 如果用一些简单的代号代替这些字符串，就可以实现压缩。字符串与代号的对应表就是词典。

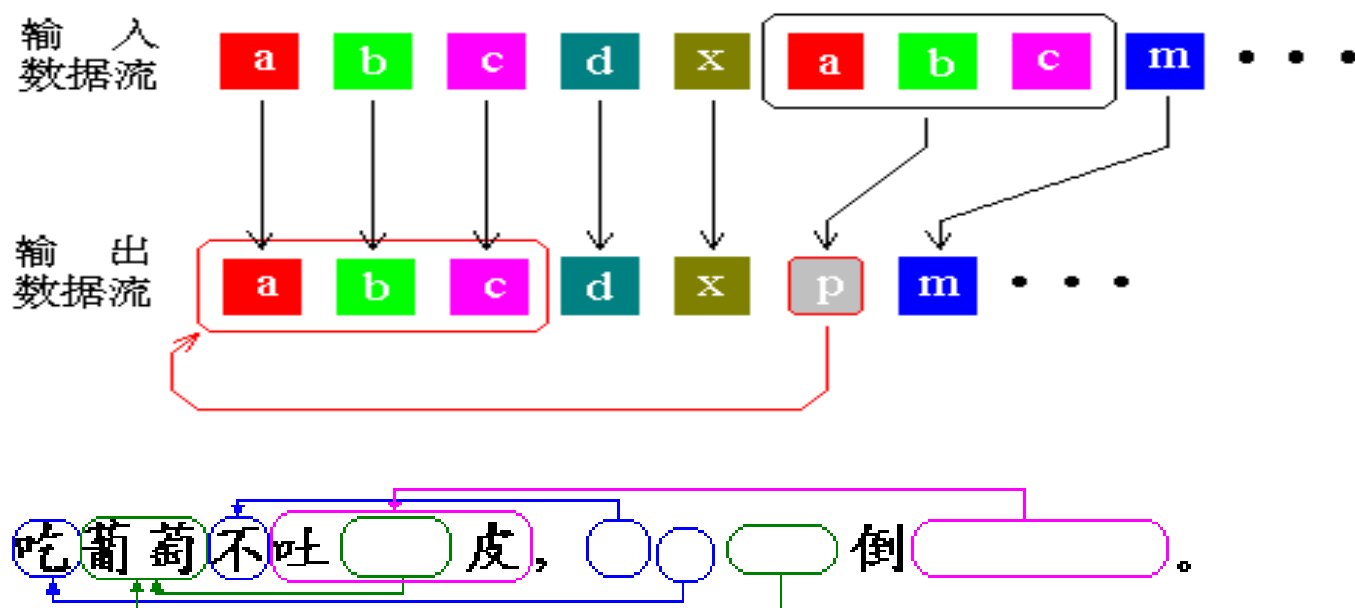
## ✓ 问题:

✓ 词典如何形成的? - 动态, 编解码器的对称性?

✓ 输出代号格式如何定义? 表示的有效性?

## 3.2.3.2 第一类词典编码

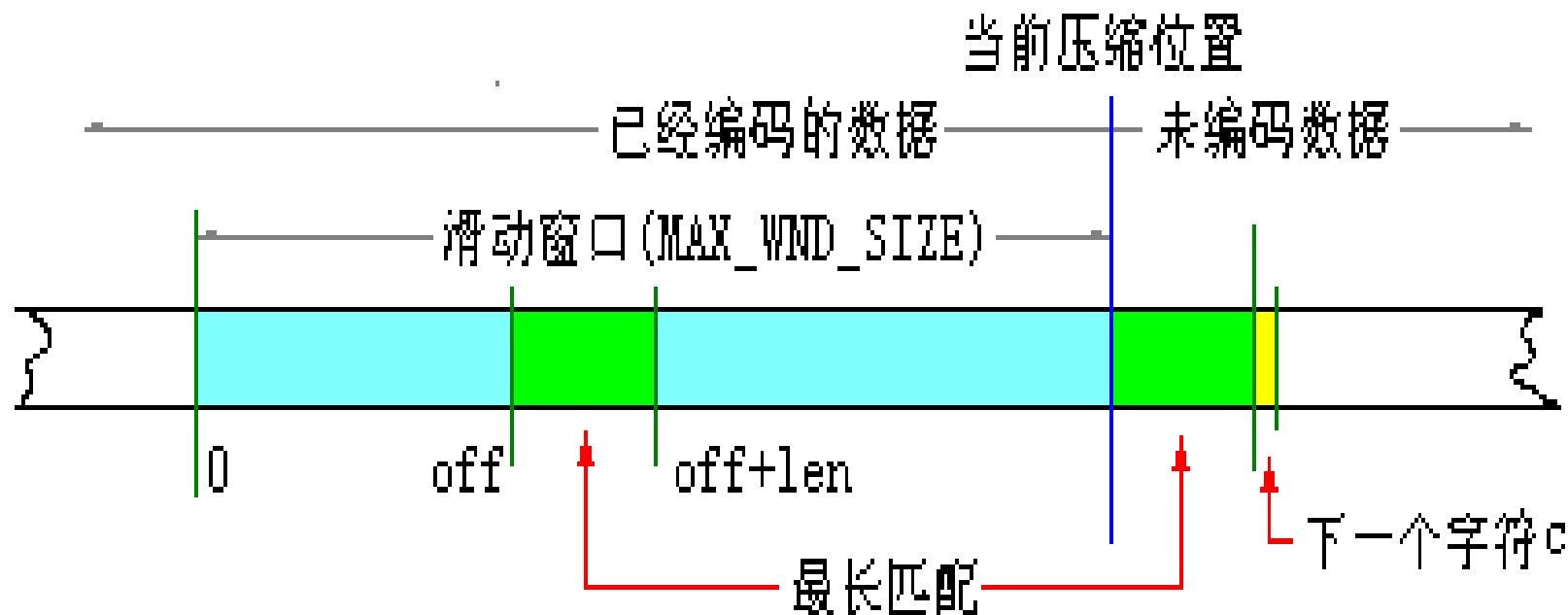
- 第一类词典法的想法是企图查找正在压缩的字符序列是否在以前输入的数据中出现过，然后用已经出现过的字符串替代重复的部分，其输出代号定义为指向早期出现过的字符串的“指针”。



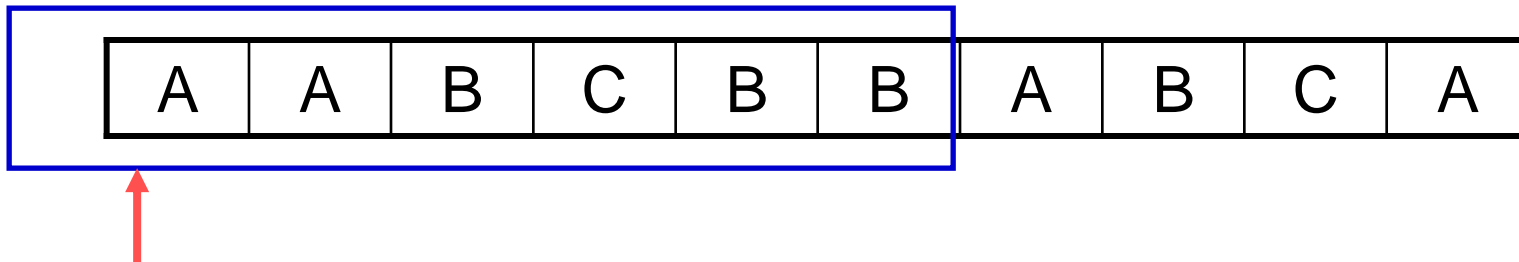
## 3.2.3.2 第一类词典编码 – LZ77

### ■ LZ77 算法又可以称为“滑动窗口压缩”

- 将一个虚拟的，可以跟随压缩进程滑动的窗口作为词典，要压缩的字符串如果在该窗口中出现，则输出其出现位置和长度。



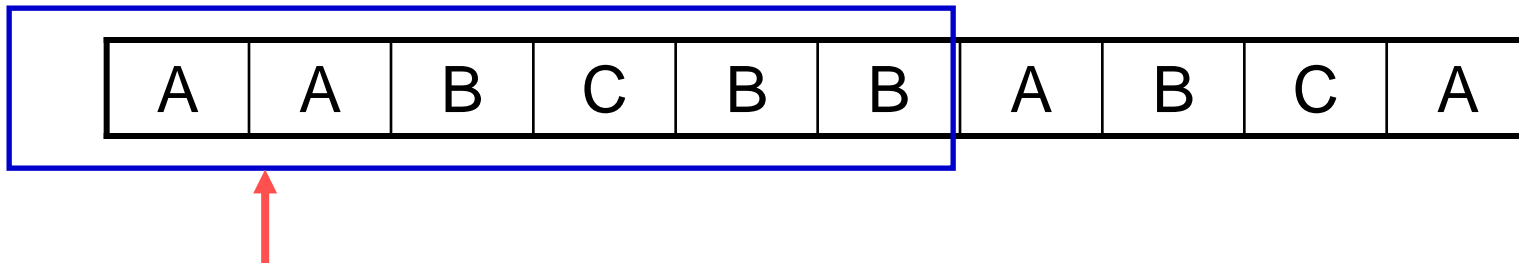
## 3.2.3.2 第一类词典编码 – LZ77



步骤	位置	匹配串	输出
1	1	— —	0,0,A
2			
3			
4			
5			

输出三元符号组 ( **off**, **len**, **c** )。其中 **off** 为窗口中匹配字符串相对窗口边界的偏移，**len** 为可匹配的长度，**c** 为下一个字符，即不匹配的第一个字符。

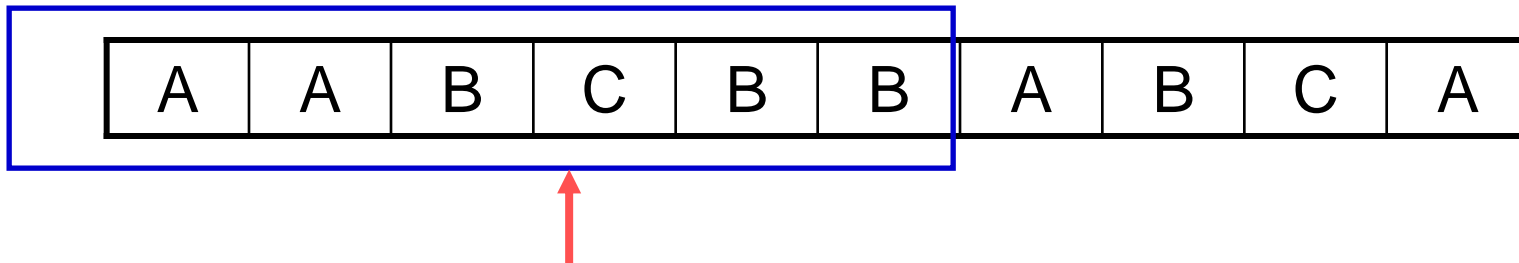
## 3.2.3.2 第一类词典编码 – LZ77



步骤	位置	匹配串	输出
1	1	— —	0,0,A
2	2	A	1,1,B
3			
4			
5			

输出三元符号组 ( **off**, **len**, **c** )。其中 **off** 为窗口中匹配字符串相对窗口边界的偏移，**len** 为可匹配的长度，**c** 为下一个字符，即不匹配的第一个字符。

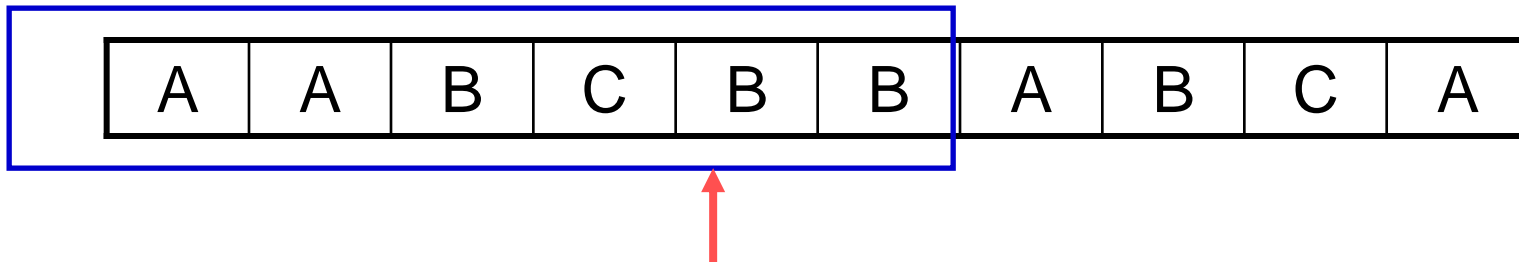
## 3.2.3.2 第一类词典编码 – LZ77



步骤	位置	匹配串	输出
1	1	— —	0,0,A
2	2	A	1,1,B
3	4	— —	0,0,C
4			
5			

输出三元符号组 ( **off**, **len**, **c** )。其中 **off** 为窗口中匹配字符串相对窗口边界的偏移，**len** 为可匹配的长度，**c** 为下一个字符，即不匹配的第一个字符。

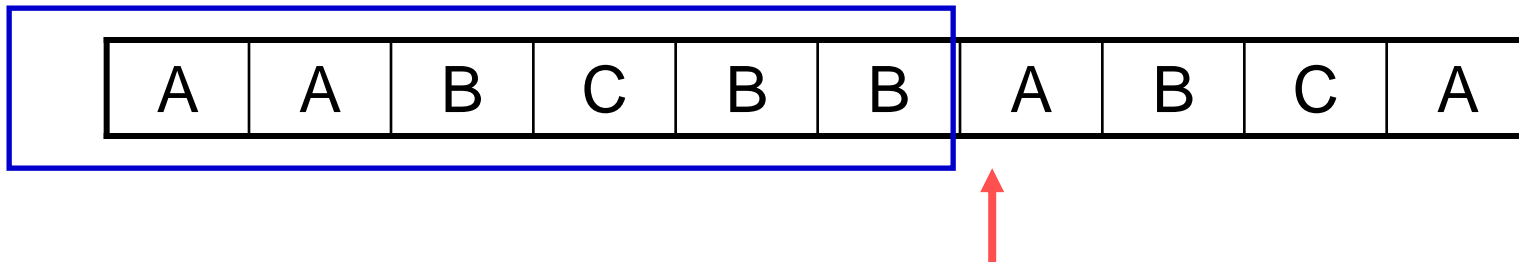
## 3.2.3.2 第一类词典编码 – LZ77



步骤	位置	匹配串	输出
1	1	— —	0,0,A
2	2	A	1,1,B
3	4	— —	0,0,C
4	5	B	2,1,B
5			

输出三元符号组 ( **off**, **len**, **c** )。其中 **off** 为窗口中匹配字符串相对窗口边界的偏移，**len** 为可匹配的长度，**c** 为下一个字符，即不匹配的第一个字符。

## 3.2.3.2 第一类词典编码 – LZ77



步骤	位置	匹配串	输出
1	1	— —	0,0,A
2	2	A	1,1,B
3	4	— —	0,0,C
4	5	B	2,1,B
5	7	ABC	5,3,A

输出三元符号组 ( **off**, **len**, **c** )。其中 **off** 为窗口中匹配字符串相对当前字符位置的偏移，**len** 为可匹配的长度，**c** 为下一个字符，即不匹配的第一个字符。



## 3.2.3.2 第一类词典编码 – LZ77

A	A	B	C	B	B	A	B	C	A
---	---	---	---	---	---	---	---	---	---



步骤	位置	匹配串	输出
1	1	— —	0,0,A
2	2	A	1,1,B
3	4	— —	0,0,C
4	5	B	2,1,B
5	7	ABC	5,3,A

窗口向后滑动 **len + 1** 个字符

## 3.2.3.2 第一类词典编码 – LZ77

### ■ 词典的形成

- 随着压缩的进程滑动词典窗口，使其中总包含最近编码过的信息，是因为对大多数信息而言，要编码的字符串往往在最近的上下文中更容易找到匹配串。
- 使用**固定大小窗口**进行词语匹配，而不是在所有已经编码的信息中匹配，是因为匹配算法的时间消耗往往很多，必须限制词典的大小才能保证算法的效率。
- **解码器**维持一个与编码器窗口大小相同的缓冲区，并在缓冲区中找出匹配串，将匹配串及第3个字段的字符写入输出流，同时移进缓冲区。

## 3.2.3.2 第一类词典编码 – LZ77

### ■ 输出格式的设计

- 三元符号组 ( **off**, **len**, **c** )。其中 **off** 为窗口中匹配字符串相对当前字符位置的偏移，**len** 为可匹配的长度，**c** 为下一个字符，即不匹配的下一个字符。
- 通过输出真实字符解决了在窗口中出现没有匹配串的问题。
  - 这个解决方案包含有冗余信息。冗余信息表现在两个方面，一是空指针，二是编码器可能输出额外的字符，这种字符是指可能包含在下一个匹配串中的字符。

## 3.2.3.2 第一类词典编码 – LZSS

### ■ LZSS算法的思想

- 如果匹配串的长度比指针本身的长度长就输出指针（匹配串长度大于等于MIN\_LENGTH），否则就输出真实字符。
- 需要输出额外的标志位区分是指针还是字符。

# 3.2.3.2 第一类词典编码 – LZSS

输入数据流:

位置	1	2	3	4	5	6	7	8	9	10	11
字符	A	A	B	B	C	B	B	A	A	B	C

步骤	位置	匹配串	输出
1	1	--	A
2	2	A	A
3	3	--	B
4	4	B	B
5	5	--	C
6	6	BB	(3, 2)
7	8	AAB	(7, 3)
8	11	C	C

编码过程

MIN LEN =2

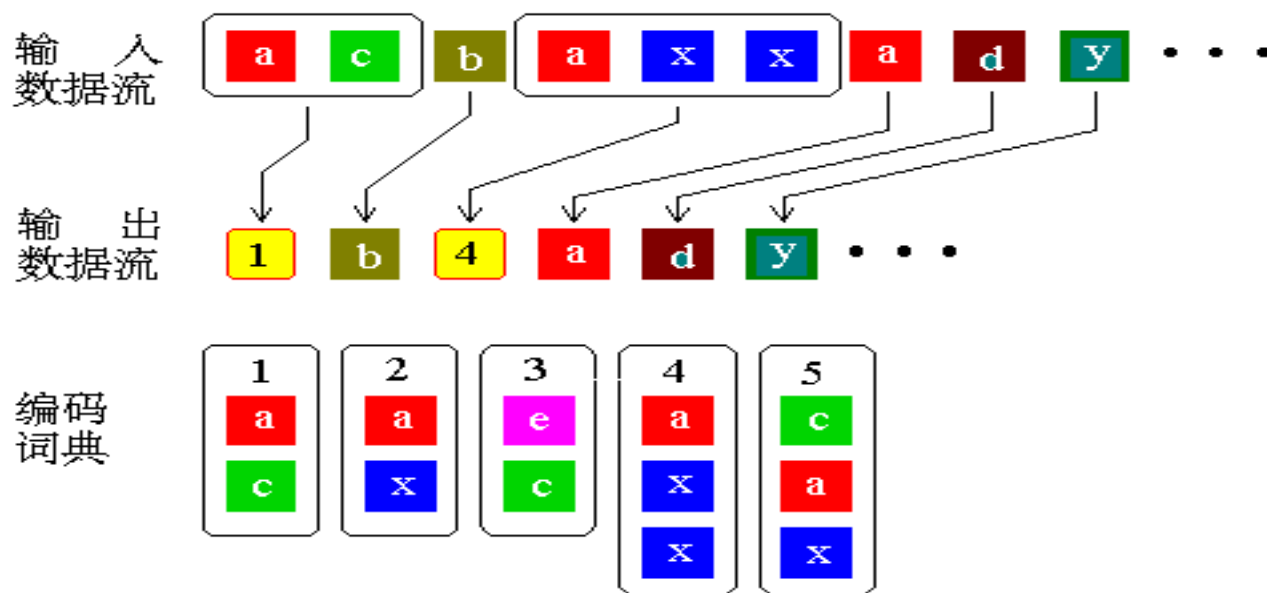
窗口向后滑动 **len** 个字符

## 3.2.3.2 第一类词典编码 – LZSS

- 在相同的计算机环境下，LZSS算法比LZ77可获得更高的压缩比，而译码同样简单。
- 这种算法成为开发新算法的基础，许多后来开发的文档压缩程序都使用了LZSS的思想。例如，PKZip, GZip, ARJ, LHArc和ZOO等等，其差别仅仅是指针的长短和窗口的大小等有所不同。
- LZSS同样可以和熵编码联合使用，例如ARJ就与霍夫曼编码联用，而PKZip则与Shannon-Fano联用，它的后续版本也采用霍夫曼编码。

## 3.2.3.3 第二类词典编码

- 第二类算法的想法是企图从输入的数据中**创建**一个“**短语词典** (dictionary of the phrases)”，这种短语可以是任意字符的组合。编码数据过程中当遇到已经在词典中出现的“短语”时，编码器就输出这个词典中的短语的“索引号”，而不是短语本身。
  - 不再设置搜索缓冲区
  - 编解码器应同步建立词典



### 3.2.3.3 第二类词典编码 - LZW

J. Ziv和A. Lempel在1978年首次发表了介绍第二类词典编码算法的文章。在他们的研究基础上，Terry A. Welch在1984年发表了改进这种编码算法的文章，因此把这种编码方法称为LZW (Lempel-Ziv Welch) 压缩编码。

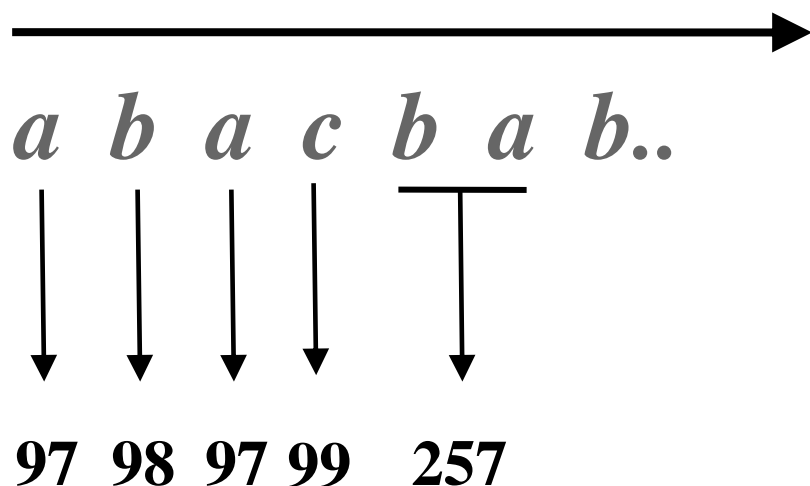
LZW只输出代表词典中的字符串(String)的码字(code word)。这就意味在开始时词典不能是空的，它必须包含可能在字符流出现中的所有单个字符。即在编码匹配时，至少可以在词典中找到长度为1的匹配串。



### 3.2.3.3 第二类词典编码 - LZW

LZW ( Lempel - Ziv - Welch ) 编码算法的思想

算法总体思路:

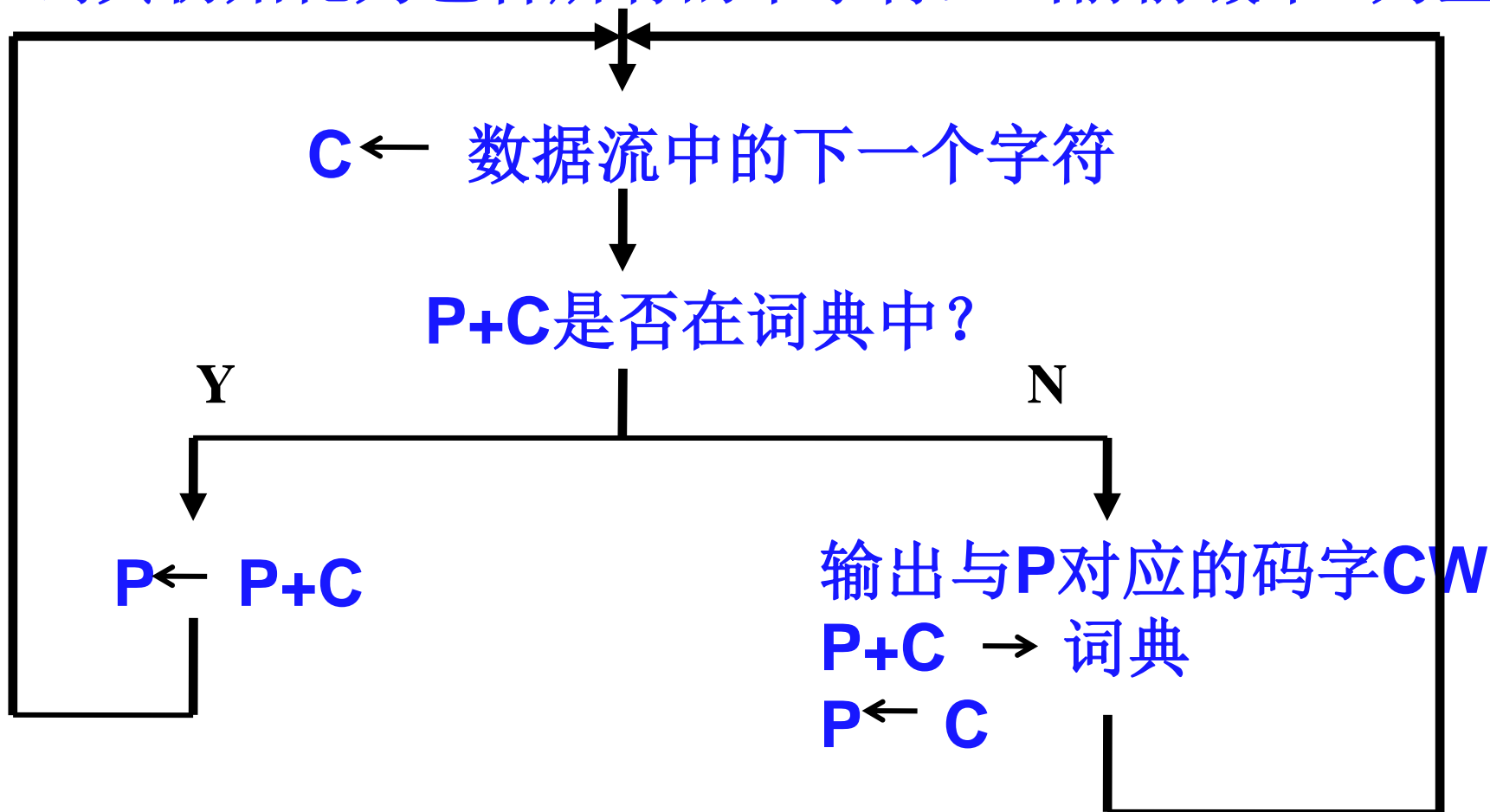


词典			
Index	String	Index	String
0	...	256	<i>ab</i>
...	...	257	<i>ba</i>
65	<i>A</i>	258	<i>ac</i>
...	...	259	<i>cb</i>
97	<i>a</i>	260	
98	<i>b</i>	261	...
99	<i>c</i>	262	
...	...	263	
...	...	264	
254	...	265	
255	...	266	

### 3.2.3.3 第二类词典编码 - LZW

LZW ( Lempel - Ziv - Welch ) 编码算法的思想

词典初始化为包含所有的单字符，当前前缀串**P**为空



## 【例】：LZW编码示例 Step 1

↓  
*a b b a b a b a c*

**P** ← Null

**C** ← “a”

词典			
Index	String	Index	String
0	...	256	
...	...	257	
65	A	258	
...	...	259	
97	<i>a</i>	260	
98	<i>b</i>	261	
99	<i>c</i>	262	
...	...	263	
...	...	264	
254	...	265	
255	...	266	

## 【例】：LZW编码示例 Step 2

*a b b a b a b a c*

↓

*a b*

↓

97

P ← “*b*”

C ← “*b*”

词典			
Index	String	Index	String
0	...	256	<i>ab</i>
...	...	257	
65	A	258	
...	...	259	
97	<i>a</i>	260	
98	<i>b</i>	261	
99	<i>c</i>	262	
...	...	263	
...	...	264	
254	...	265	
255	...	266	

## 【例】：LZW编码示例 Step 3

*a b b a b a b a c*

↓ ↓ ↓

97 98

$P \leftarrow "b"$

$C \leftarrow "b"$

词典			
Index	String	Index	String
0	...	256	<i>ab</i>
...	...	257	<i>bb</i>
65	A	258	
...	...	259	
97	<i>a</i>	260	
98	<i>b</i>	261	
99	<i>c</i>	262	
...	...	263	
...	...	264	
254	...	265	
255	...	266	

## 【例】：LZW编码示例 Step 4

*a b b a b a b a c*

↓ ↓ ↓ ↓

97 98 98

P ← “b”

C ← “a”

词典			
Index	String	Index	String
0	...	256	<i>ab</i>
...	...	257	<i>bb</i>
65	A	258	<i>ba</i>
...	...	259	
97	<i>a</i>	260	
98	<i>b</i>	261	
99	<i>c</i>	262	
...	...	263	
...	...	264	
254	...	265	
255	...	266	

## 【例】：LZW编码示例 Step 5

*a b b a b a b a c*

↓ ↓ ↓ ↓

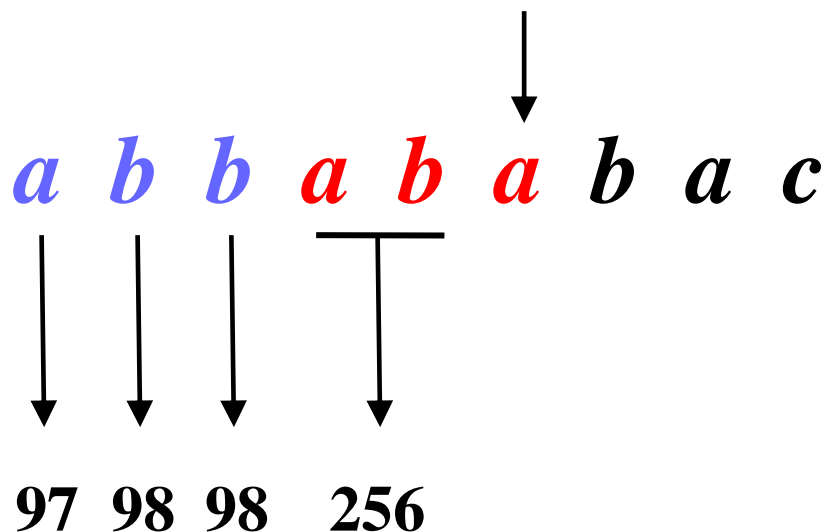
97 98 98

$P \leftarrow "ab"$

$C \leftarrow "b"$

词典			
Index	String	Index	String
0	...	256	<i>ab</i>
...	...	257	<i>bb</i>
65	<i>A</i>	258	<i>ba</i>
...	...	259	
97	<i>a</i>	260	
98	<i>b</i>	261	
99	<i>c</i>	262	
...	...	263	
...	...	264	
254	...	265	
255	...	266	

## 【例】：LZW编码示例 Step 6



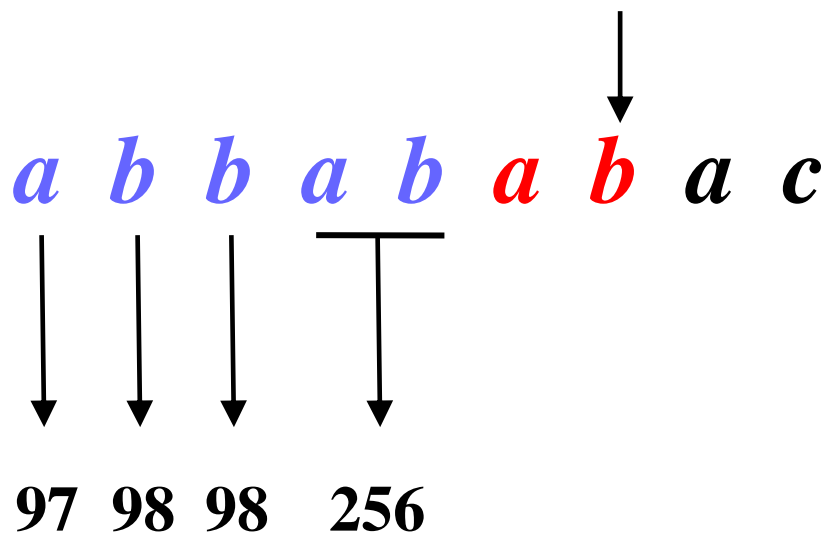
$P \leftarrow "ab"$

$C \leftarrow "a"$

词典			
Index	String	Index	String
0	...	256	<i>ab</i>
...	...	257	<i>bb</i>
65	<i>A</i>	258	<i>ba</i>
...	...	259	<i>aba</i>
97	<i>a</i>	260	
98	<i>b</i>	261	
99	<i>c</i>	262	
...	...	263	
...	...	264	
254	...	265	
255	...	266	



## 【例】：LZW编码示例 Step 7

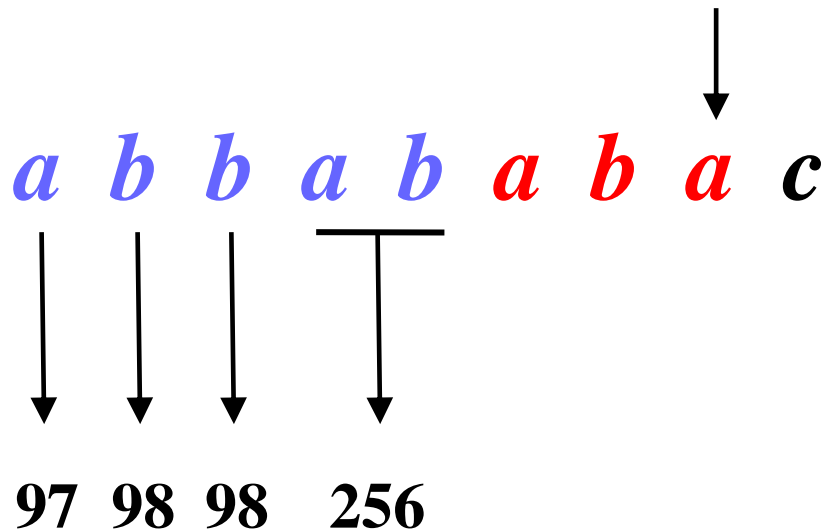


$P \leftarrow "ab"$

$C \leftarrow "b"$

词典			
Index	String	Index	String
0	...	256	<i>ab</i>
...	...	257	<i>bb</i>
65	<i>A</i>	258	<i>ba</i>
...	...	259	<i>aba</i>
97	<i>a</i>	260	
98	<i>b</i>	261	
99	<i>c</i>	262	
...	...	263	
...	...	264	
254	...	265	
255	...	266	

## 【例】：LZW编码示例 Step 8

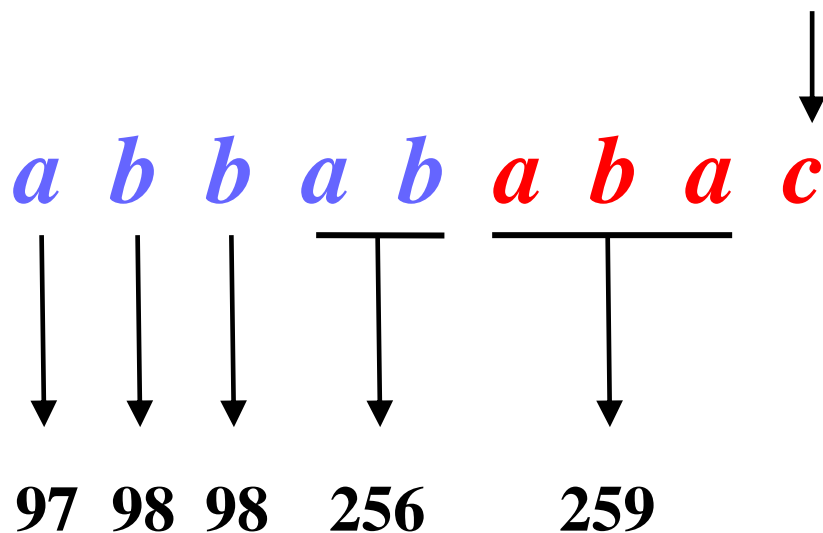


P ← “abä”

C ← “a”

词典			
Index	String	Index	String
0	...	256	<i>ab</i>
...	...	257	<i>bb</i>
65	A	258	<i>ba</i>
...	...	259	<i>aba</i>
97	<i>a</i>	260	
98	<i>b</i>	261	
99	<i>c</i>	262	
...	...	263	
...	...	264	
254	...	265	
255	...	266	

## 【例】：LZW编码示例 Step 9



P ← “*a b a*”

C ← “*c*”

词典			
Index	String	Index	String
0	...	256	<i>ab</i>
...	...	257	<i>bb</i>
65	<i>A</i>	258	<i>ba</i>
...	...	259	<i>aba</i>
97	<i>a</i>	260	<i>abac</i>
98	<i>b</i>	261	
99	<i>c</i>	262	
...	...	263	
...	...	264	
254	...	265	
255	...	266	

## 【例】：LZW编码示例 Step 10

*a b b a b a b a c*

↓ ↓ ↓     ↓     ↓     ↓

97 98 98    256    259    99

将变长的字符串映射为  
定长的码字

输入：8bit x 9 = 72bit

输出：9bit x 6 = 54bit

P “c”                      C ← 结束符

词典			
Index	String	Index	String
0	...	256	<i>ab</i>
...	...	257	<i>bb</i>
65	<i>A</i>	258	<i>ba</i>
...	...	259	<i>aba</i>
97	<i>a</i>	260	<i>abac</i>
98	<i>b</i>	261	
99	<i>c</i>	262	
...	...	263	
...	...	264	
254	...	265	
255	...	266	

# 3.2.3.3 第二类词典编码 - LZW

LZW ( Lempel - Ziv - Welch ) 解码算法

97 98 98 256 259 99

↓ ↓ ↓ ↓ ↓ ↓  
*a b b ab* ?

*cW* ← 码字流中的第一个码字

Dictionary[*cW*] → 字符流

*pW* ← *cW*

Dictionary[*cW*] → 字符流

*P* ← Dictionary[*pW*]

*C* ← Dictionary[*cW*] 的第一个字符

*P+C* → 词典; *pW* ← *cW*

词典

Index	String	Index	String
0	...	256	<i>ab</i>
...	...	257	<i>bb</i>
65	<i>A</i>	258	<i>ba</i>
...	...	259	
97	<i>a</i>	260	
98	<i>b</i>	261	
99	<i>c</i>	262	
...	...	263	
...	...	264	
254	...	265	
255	...	266	

### 3.2.3.3 第二类词典编码 - LZW

$cW \leftarrow$  码字流中的第一个码字

Dictionary[ $cW$ ]  $\rightarrow$  字符流

$pW \leftarrow cW$

While(( $cW \leftarrow$  下一个码字)  $\neq$  NULL)

{ if (  $cW$  在词典中 )

    Dictionary[ $cW$ ]  $\rightarrow$  字符流

$P \leftarrow$  Dictionary[ $pW$ ]

$C \leftarrow$  Dictionary[ $cW$ ] 的第一个字符

$P+C \rightarrow$  词典;     $pW \leftarrow cW$

else

$P \leftarrow$  Dictionary[ $pW$ ]

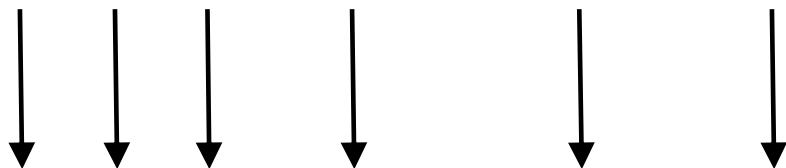
$C \leftarrow$  Dictionary[ $pW$ ] 的第一个字符

$P+C \rightarrow$  词典;  $P+C \rightarrow$  字符流;  $pW \leftarrow cW$  }

## 【例】：LZW解码示例

**cW**

97 98 98 256 259 99



**a**

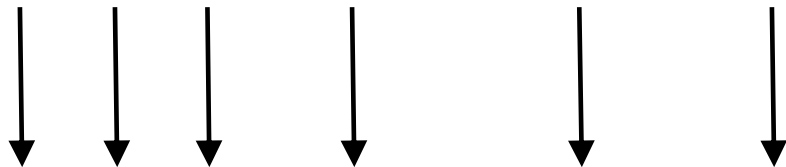
词典

Index	String	Index	String
0	...	256	
...	...	257	
65	A	258	
...	...	259	
97	a	260	
98	b	261	
99	c	262	
...	...	263	
...	...	264	
254	...	265	
255	...	266	

## 【例】：LZW解码示例

pW cW

97 98 98 256 259 99



a b

Dictionary[cW]  $\longrightarrow$  字符流

P  $\longleftarrow$  Dictionary[pW]

C  $\longleftarrow$  Dictionary[cW] 的第一个字符

P+C  $\longrightarrow$  词典

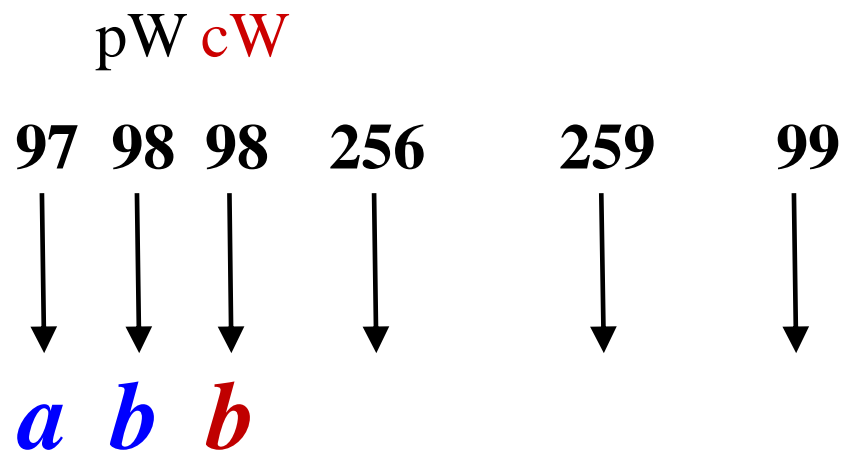
pW  $\longleftarrow$  cW

词典

Index	String	Index	String
0	...	256	ab
...	...	257	
65	A	258	
...	...	259	
97	a	260	
98	b	261	
99	c	262	
...	...	263	
...	...	264	
254	...	265	
255	...	266	



## 【例】：LZW解码示例



Dictionary[cW] → 字符流

P ← Dictionary[pW]

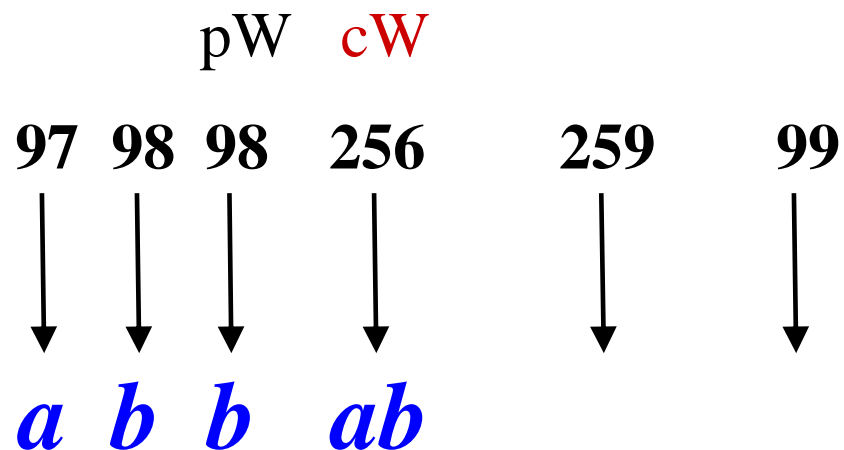
C ← Dictionary[cW] 的第一个字符

P+C → 词典

pW ← cW

词典			
Index	String	Index	String
0	...	256	<i>ab</i>
...	...	257	<i>bb</i>
65	A	258	
...	...	259	
97	<i>a</i>	260	
98	<i>b</i>	261	
99	<i>c</i>	262	
...	...	263	
...	...	264	
254	...	265	
255	...	266	

# 【例】：LZW解码示例



Dictionary[**cW**] → 字符流

**P** ← Dictionary[**pW**]

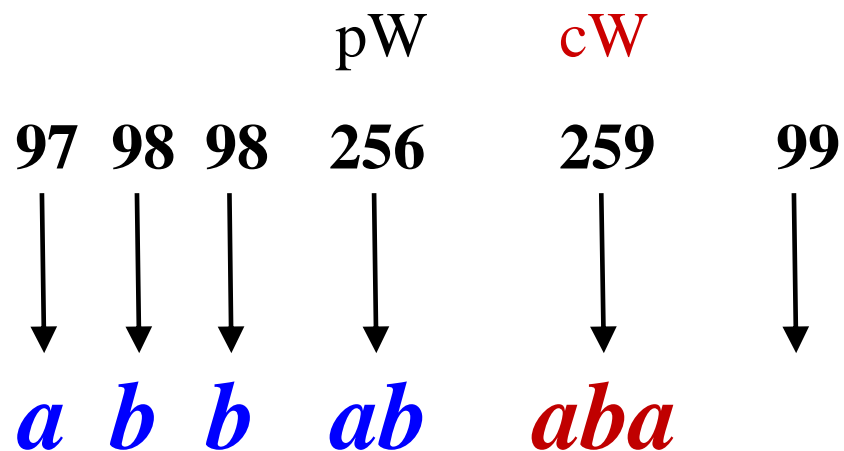
**C** ← Dictionary[**cW**] 的第一个字符...

**P+C** → 词典

**pW** ← **cW**

词典			
Index	String	Index	String
0	...	256	<i>ab</i>
...	...	257	<i>bb</i>
65	<i>A</i>	258	<i>ba</i>
...	...	259	
97	<i>a</i>	260	
98	<i>b</i>	261	
99	<i>c</i>	262	
...	...	263	
...	...	264	
254	...	265	
255	...	266	

# 【例】：LZW解码示例



**P** ← Dictionary[**pW**]

**C** ← Dictionary[**pW**] 的第一个字符

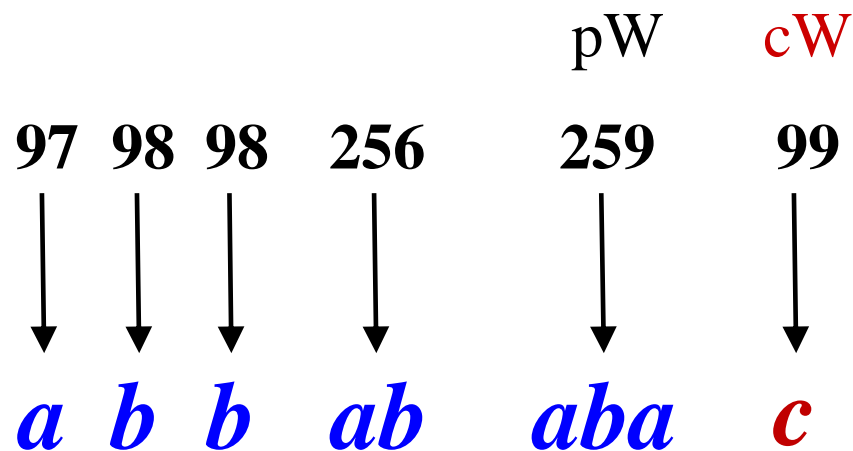
**P+C** → 字符流

**P+C** → 词典

**pW** ← **cW**

词典			
Index	String	Index	String
0	...	256	<i>ab</i>
...	...	257	<i>bb</i>
65	<i>A</i>	258	<i>ba</i>
...	...	259	<i>aba</i>
97	<i>a</i>	260	
98	<i>b</i>	261	
99	<i>c</i>	262	
...	...	263	
...	...	264	
254	...	265	
255	...	266	

## 【例】：LZW解码示例



Dictionary[cW] → 字符流

P ← Dictionary[pW]

C ← Dictionary[cW] 的第一个字符

P+C → 词典

pW ← cW

词典			
Index	String	Index	String
0	...	256	<i>ab</i>
...	...	257	<i>bb</i>
65	<i>A</i>	258	<i>ba</i>
...	...	259	<i>aba</i>
97	<i>a</i>	260	<i>abac</i>
98	<i>b</i>	261	
99	<i>c</i>	262	
...	...	263	
...	...	264	
254	...	265	
255	...	266	

## 3.2.3.3 第二类词典编码 - LZW

### LZW算法的实现

#### 1. 数据结构分析

假设输入的字符序列为 *a a a b b b b b a a b a a b a*

<b>Code</b>	<b>0</b>	<b>1</b>	<b>2</b>	<b>3</b>	<b>4</b>	<b>5</b>	<b>6</b>	<b>7</b>
<b>Key</b>	<i>a</i>	<i>b</i>	<i>aa</i>	<i>aab</i>	<i>bb</i>	<i>bbb</i>	<i>bbba</i>	<i>aaba</i>

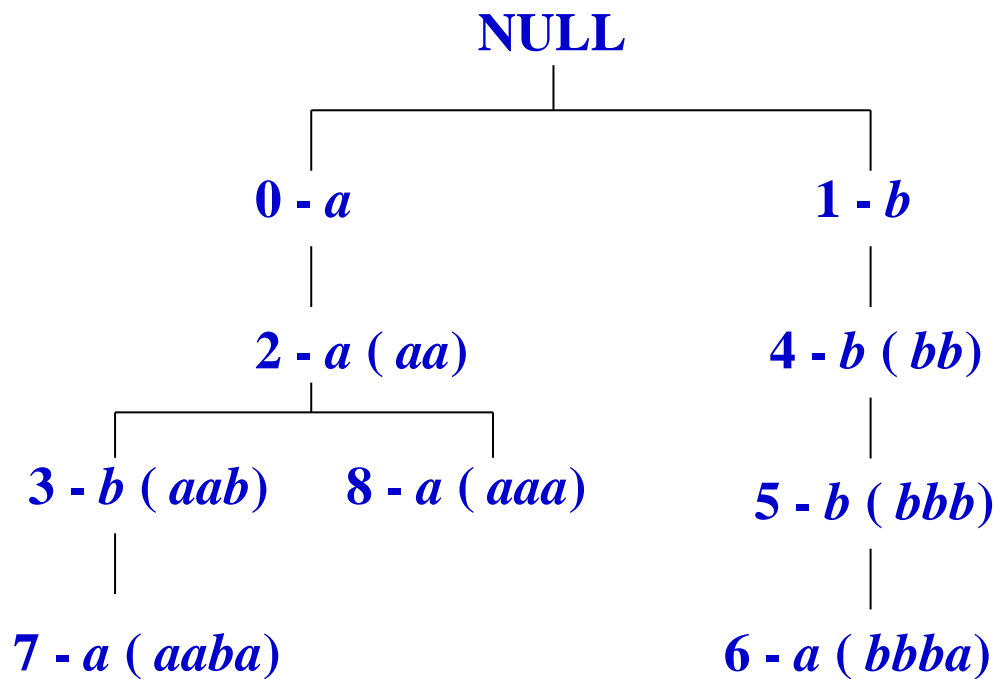
每个长度 $l > 1$ 的关键字的前 $l - 1$ 个字符(称为关键字前缀)都可在字典中找到，因此可以用代码代替其关键字前缀

<b>Code</b>	<b>0</b>	<b>1</b>	<b>2</b>	<b>3</b>	<b>4</b>	<b>5</b>	<b>6</b>	<b>7</b>
<b>Key</b>	<i>a</i>	<i>b</i>	<i>0a</i>	<i>2b</i>	<i>1b</i>	<i>4b</i>	<i>5a</i>	<i>3a</i>

# 3.2.3.3 第二类词典编码 - LZW

## 1. 数据结构分析

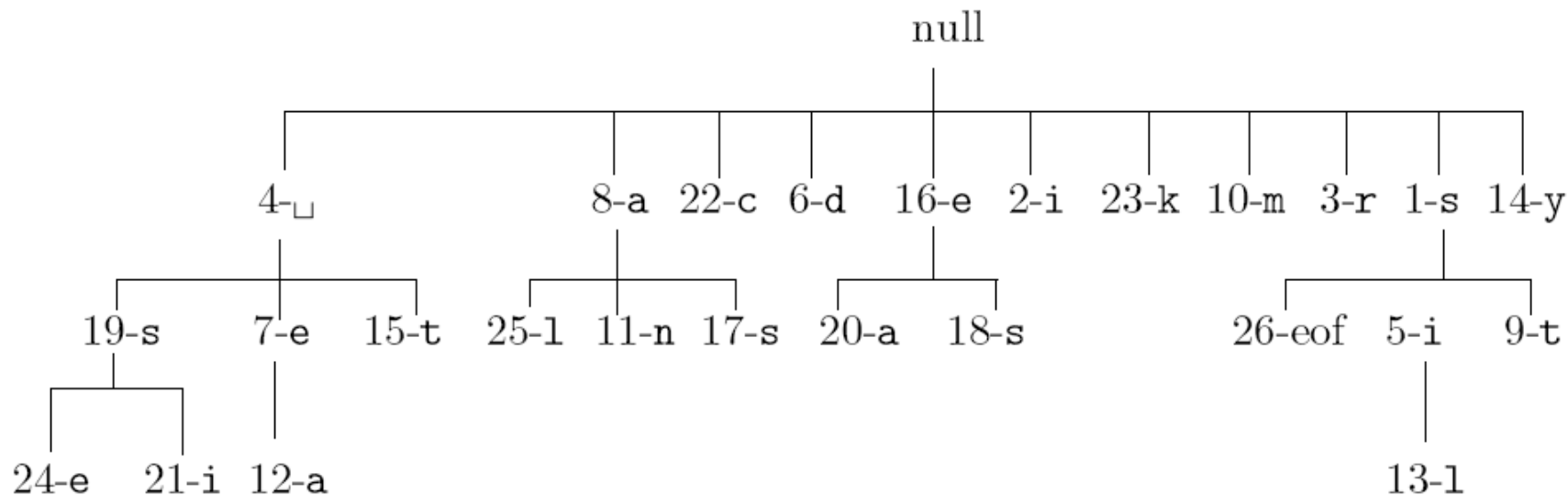
假设输入的字符序列为 *a a a b b b b b a a b a a b a*



数字查找树 (Digital Search Tree)  
或Trie树(trie为retrieve中间4个字符)

# 3.2.3.3 第二类词典编码 - LZW

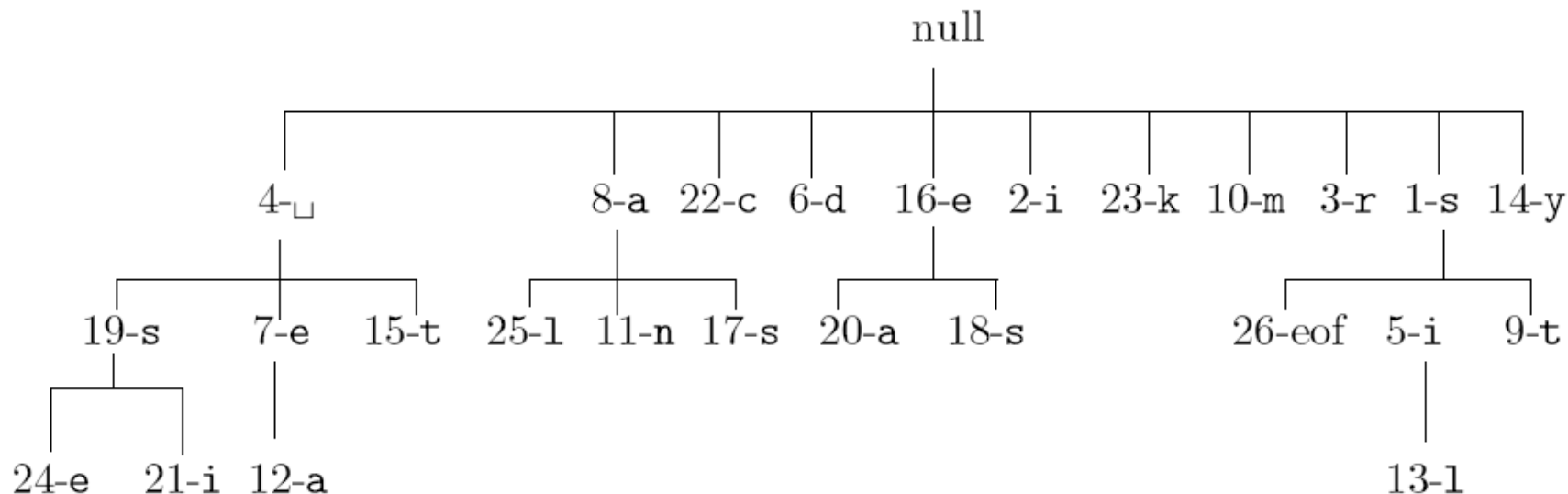
## 1. 数据结构分析



◆ 树是动态建立的，且树中每个节点**可能**存在多个子节点。因此数据结构应该设计成一个节点可拥有**任意**个子节点，但无需为其预留空间

# 3.2.3.3 第二类词典编码 - LZW

## 1. 数据结构分析



◆将树驻留在一个节点数组中，每个节点至少有两个字段：一个字符和指向母节点的指针

◆数据结构中没有指向子节点的指针，需要考虑沿着树从一个节点到其子节点的操作，如何实现？



## 3.2.3.3 第二类词典编码 - LZW

### 1. 数据结构分析

尾缀字符 ( <b>suffix</b> )
母节点 ( <b>parent</b> )
第一个孩子节点( <b>firstchild</b> )
下一个兄弟节点 ( <b>nextsibling</b> )

树用数组dict[ ]表示，数组下标用pointer表示，所以dict[pointer]表示一个节点

dict[pointer].suffix

dict[pointer].parent

dict[pointer].firstchild

dict[pointer].nextsibling

## 3.2.3.3 第二类词典编码 - LZW

### 2. 主要功能模块分析- 初始化词典

```
for( i=0; i<256; i++)  
{  
    dictionary[i].suffix = i;  
    dictionary[i].parent = -1;  
    dictionary[i].firstchild = -1;  
    dictionary[i].nextsibling = i+1;  
}  
dictionary[255].nextsibling = -1;  
next_code = 256;  
string_code = -1;
```

## ■ 查找词典中是否有字符串

/\* \* Input: string represented by string\_code in dictionary,

Output: the index of character+string in the dictionary \* index = -1 if not found \*/

```
int InDictionary( int character, int string_code)
```

```
{
```

```
    int sibling;
```

```
    if( 0>string_code) return character;  如果是单个字符?
```

```
    sibling = dictionary[string_code].firstchild; // 找第一个兄弟节
```

点

```
    while( -1<sibling)
```

```
    {    if( character == dictionary[sibling].suffix)
```

```
        return sibling;
```

```
        sibling = dictionary[sibling].nextsibling;
```

```
    }
```

```
    return -1;
```

```
}
```

```
if( 0<=index)
{ // string+character in dictionary
  string_code = index;
}else
{ // string+character not in dictionary
  output( bf, string_code);
  if( MAX_CODE > next_code)
  { // free space in dictionary
    // add string+character to dictionary
    AddToDictionary( character, string_code);
  }
  string_code = character;
}
```

■ 将新串加入词典

```
void AddToDictionary( int character, int string_code)
{   int firstsibling, nextsibling;
    if( 0>string_code) return;
    dictionary[next_code].suffix = character;
    dictionary[next_code].parent = string_code;
    dictionary[next_code].nextsibling = -1;
    dictionary[next_code].firstchild = -1;
    firstsibling = dictionary[string_code].firstchild;
    if( -1<firstsibling)
    { // the parent has child
        nextsibling = firstsibling;
        while( -1<dictionary[nextsibling].nextsibling )
            nextsibling = dictionary[nextsibling].nextsibling;
        dictionary[nextsibling].nextsibling = next_code;
    }else
    { // no child before, modify it to be the first
        dictionary[string_code].firstchild = next_code;
    }
    next_code ++;
}
```

# 字典结构

- 实用中，每个节点有3个字段：

母节点 (parent)
散列索引 (index)
字符 (symbol)

- 树用数组dict[]表示，数组下标用pointer表示，所以dict[pointer]表示一个节点
  - dict[pointer].parent
  - dict[pointer].index
  - dict[pointer].symbol

# 构造哈希函数

构造哈希函数的目标是使哈希地址尽可能均匀地分布在连续的内存单元地址上，以减少冲突发生的可能性，同时使计算尽可能简单以达到尽可能高的时间效率。根据关键字的结构和分布不同，可构造出与之适应的各不相同的哈希函数。

- 除留余数法采用取模运算（%），把关键字除以某个不大于哈希表表长的整数得到的余数作为哈希地址。哈希函数的形式为：

$$h(\text{key}) = \text{key} \% p$$

- 除留余数法的关键是选好 $p$ ，使得记录集合中的每个关键字通过该孩子数转换后映射到哈希表范围内任意地址上的概率相等，从而尽可能减少发生冲突的可能性。

507683 的  $\rightarrow$  111101111100100011  
二进制

507683%2<sup>5</sup>相当于取低5位二进制数

# 字典结构

- 例：字符串“ababab...” (a: 1 b: 2)
- **Step 0:** 将从3个位置开始的所有字典位置标记为“未使用”

/	/	/	/	/	...
1	2	-	-	-	
a	b				

- **Step 1:** 将第一个字符a输入到变量I。“a”的码字为1，因此I = 1。因为是第一个字符，编码器假定它已在字典中，故无需搜索
- **Step 2:** 将第二个字符b输入到J，所以J = 2。编码器在字典中搜索字符串ab，执行 `pointer:=hash(I,J)`，假设结果为5。因为位置5仍然是空的，字段`dict[pointer].index`标记为“未使用”，因此串ab不在字典中，将其添加到字典中：

- `dict[pointer].parent:=I;`
- `dict[pointer].index:=pointer;`
- `dict[pointer].symbol:=J;`

由于`pointer=5`，将J送入I，因此I = 2。

/	/	/	/	1	...
1	2	-	-	5	
a	b			b	



# 字典结构

- **Step 3:** 将第3个字符**a**输入到J，所以 $J = 1$ 。编码器在字典中搜索字符串**ba**，执行  $\text{pointer} := \text{hash}(I, J)$ ，假设结果为8。因为位置8仍然是空的，字段 $\text{dict}[\text{pointer}].\text{index}$ 标记为“未使用”，因此串**ba**不在字典中，将其添加到字典中：

- $\text{dict}[\text{pointer}].\text{parent} := I$ ;
- $\text{dict}[\text{pointer}].\text{index} := \text{pointer}$ ;
- $\text{dict}[\text{pointer}].\text{symbol} := J$ ;

/	/	/	/	1	/	/	2	/
1	2	-	-	5	-	-	8	-
a	b			b			a	

...

由于 $\text{pointer} = 8$ ，将J送入I，因此 $I = 1$ 。

- **Step 4:** 将第4个字符**b**输入到J，这时 $J = 2$ 。编码器在字典中搜索字符串**ab**，执行  $\text{pointer} := \text{hash}(I, J)$ ，由步骤2可知该结果为5。字段 $\text{dict}[\text{pointer}].\text{index}$ 中含有5，因此字符串**ab**在字典中，将指针的值送入I，因此 $I = 5$ 。

# 字典结构

- **Step 5:** 将第5个字符**a**输入到J，这时 $J = 1$ 。编码器在字典中搜索字符串**aba**，执行  $\text{pointer} := \text{hash}(I, J)$ ，假定该值为8。在字段  $\text{dict}[\text{pointer}].\text{index}$  中含有8，但  $\text{dict}[\text{pointer}].\text{parent}$  为2，而非预期的5，因此散列函数知道这是一个冲突且字符串**aba**不在字典第8项中。将指针+1，直到找到一个 $\text{index}=8$ 且 $\text{parent}=5$ 的词条或找到一个未使用的词条为止。前一种情况字符串**aba**在字典中，可以将其指针送入I；而后一种情况，**aba**不在字典中，编码器将其保存在所指的词条中，并将J送入I。

/	/	/	/	1	/	/	2	5	/
1	2	-	-	5	-	-	8	8	-
a	b			b			a	a	

...

# 字典结构

- LZW解码器所用的字典数组简单且无需散列
- 解码器首先将数组的前 $n$ 个位置初始化为单个字符（如 $n=256$ ），然后从输入流中读入一个码字，用每个指针来定位字典中的一个字符
- 第一步：读入第一个指针，用它检索字典的第 $I$ 项，并把找到的字符 $I$ 写入解码器输出流中。需要将 $I$ 存入字典，但字符 $x$ 此时仍未知，它将是下一个从字典中取出的串的首字符
- 此后的每一步中，解码器输入下一个指针，用它从字典中取出下一个字符串 $J$ 并记录到解码器输出流中。假设指针为 $8$ ，则解码器检查`dict[8].index`字段，如果它也是 $8$ ，那么节点就是所要找的，否则解码器检查相继的数组位置，直到找到正确的节点

# 字典结构

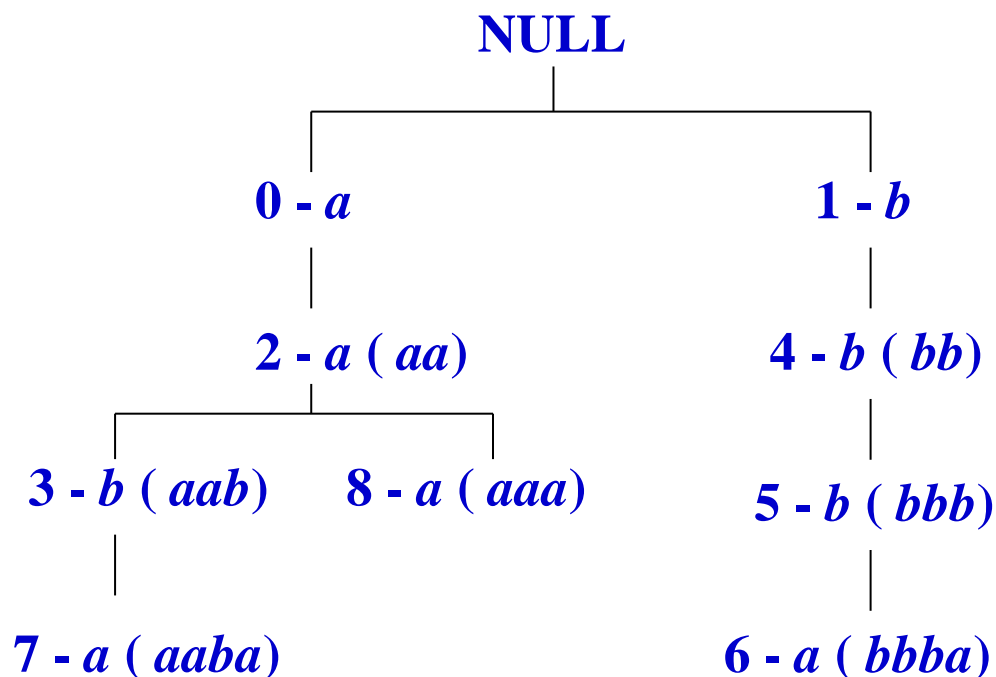
- 一旦找到了所要的树节点，就利用**parent**字段返回到树的上层，并按相反顺序取出字符串中的单个字符，再将江这些字符按正常顺序置于**J**中。解码器把第一个字符**x**从字符串**J**中分离出来，并将字符串**lx**存入下一个可用的数组单元中（字符串**l**是前面找到的，所以只需添加一个带有字符**x**的节点），然后解码器将**J**送入**l**，继续下一步解码
  - 通过**parent**字段追踪，实现对树的上行，无需散列函数

# 3.2.3.3 第二类词典编码 - LZW

## ■ LZW编码的特点和应用

### 1. 与Huffman编码相比的优点

- ◆ LZW只需一遍扫描，具有自适应的特点
- ◆ 算法简单，便于快速实现（数字查找树/键树）



# 3.2.3.3 第二类词典编码 - LZW

## ■ LZW编码的特点和应用

### 2. LZW算法的限制

- ◆ 字符串重复概率低时，影响压缩效率
  - 由输入字符流的统计特性决定，很难解决
- ◆ 词典中的字符串不再出现，影响压缩效率
  - 如果解决？如何更有效地更新词典
- ◆ 从词典中查找词条是算法中最费时的工作

# 3.2.3.3 第二类词典编码 - LZW

## ■ LZW编码的特点和应用

### 3. LZW编码的应用

- ◆ 通用文件压缩: WinZip
- ◆ 动画图像压缩: GIF, TIFF
- ◆ 电子邮件压缩
- ◆ PDF文档压缩
- ◆ 雷达数据压缩

...

# 目前最快的文本压缩LZO

- <http://www.oberhumer.com/opensource/lzo/>
- 号称实时数据压缩
- 被美国**NASA**火星漫步者探测计划使用





## 3.2.4 游程编码

- 3.2.4.1 游程编码的基本思路
- 3.2.4.2 二值图像的游程编码

## 3.2.4.1 游程编码的基本思路

a. Original data

BBBBBBBBBAAAAAAAAAAAAAAAAANMMMMMMMMMM

b. Compressed data

B09A16N01M10

游程长度（**Run Length**）是指由字符构成的数据流中各字符重复出现而形成字符串的长度。

在数据流中用(**Sc**,**X**, **RL**)3个字符给出信息。其中**Sc**是数据集**{X}**中不用的字符，起异字头作用。

讨论：对于原始字符串中**RL**长度和频度都不够显著的信源，游程编码并不适用。

- 对于二值图像可以省去**Sc**
- 对原始信源进行某些预处理，可得到更长的游程

## 3.2.4.2 二值图像的游程编码

### ■ 对二值传真图像的分析

- 每一行往往是由若干个连“0”（白色像素）、连“1”（黑色像素）组成

### ■ 编码思路 - 游程和霍夫曼编码的结合

- 分别对“黑”、“白”的不同游程长度进行Huffman编码，形成黑、白两张码表，编译码通过查表进行。

### ■ 编码规则

- 对0~63的黑白游程，用结尾码表示
- 对64~1728的黑白游程，用构造码+结尾码表示

## 3.2.4.2 二值图像的游程编码

### ■ 编码规则

- 对0~63的黑白游程，用结尾码表示

例：长度为9的白游程：10100

长度为52的黑游程：000000100100

- 对64~1728的黑白游程，用构造码+结尾码表示

例：长度为128的白游程：10010 00110101

- 规定**每行都从白游程开始**，若实际扫描行由黑开始，则需在行首加零长度白游程。**每行结束要加行同步码EOL**

游程长度	白游程码字	黑游程码字	游程长度	白游程码字	黑游程码字
0	00110101	0000110111	32	00011011	000001101010
1	000111	010	33	00010010	000001101011
2	0111	11	34	00010011	000011010010
3	1000	10	35	00010100	000011010011
4	1011	011	36	00010101	000011010100
5	1100	0011	37	00010101	000011010101
6	1110	0010	38	00010111	000011010110
7	1111	00011	39	00101000	000011010111
8	10011	000101	40	00101001	000001101100
9	10100	000100	41	00101010	000001101101
10	00111	0000100	42	00101011	000011011010
11	01000	0000101	43	00101100	000011011011
12	001000	0000111	44	00101101	000001010100
13	000011	00000100	45	00000100	000001010101
14	110100	00000111	46	00000101	000001010110
15	110101	000011000	47	00001010	000001010111
16	101010	0000010111	48	00001011	000001100100
17	101011	0000011000	49	01010010	000001100101
18	0100111	0000001000	50	01010011	000001010010
19	0001100	00001100111	51	01010100	000001010011
20	0001000	00001101000	52	01010101	000000100100
21	0010111	00001101100	53	00100100	000000110111
22	0000011	00000110111	54	00100101	000000111000
23	0000100	00000101000	55	01011000	000000100111
24	0101000	00000010111	56	01011001	000000101000
25	0101011	00000011000	57	01011010	000001011000
26	0010011	000011001010	58	01011011	000001011001
27	0100100	000011001011	59	01001010	000000101011
28	0011000	000011001100	60	01001011	000000101100
29	00000010	000011001101	61	00110010	000001011010
30	00000011	000001101000	62	00110011	000001100110
31	00011010	000001101001	63	00110100	000001100111

## MH 结尾码表

# MH

## 构造码表

游程长度	白游程码字	黑游程码字	游程长度	白游程码字	黑游程码字
64	11011	00000011111	960	011010100	00000011110011
128	10010	000011001000	1024	011010101	00000011110100
192	010111	000011001001	1088	011010110	00000011110101
256	0110111	000001011011	1152	011010111	00000011110110
320	00110110	000000110011	1216	011011000	00000011110111
384	00110111	000000110100	1280	011011001	0000001010010
448	01100100	000000110101	1344	011011010	0000001010011
512	01100101	0000001101100	1408	011011011	0000001010100
576	01101000	0000001101101	1472	010011000	0000001010101
640	01100111	0000001001010	1536	010011001	0000001011010
704	011001100	0000001001011	1600	010011010	0000001011011
768	011001101	0000001001100	1664	011000	0000001100100
832	011010010	0000001001101	1728	010011011	0000001100101
896	011010011	000000111100110	EOL	0000000000001	0000000000001

## **3.2.5 Golomb编码**

- **3.2.5.1 提出的背景**
- **3.2.5.2 一元码**
- **3.2.5.3 Golomb编码**
- **3.2.5.4 指数Golomb编码**

## 3.2.5.1 提出的背景

### ■ 变长编码最致命的缺点

- 对差错极其敏感
- 即时码的特性导致1位出错就可能使解码器失去同步

### ■ 解决思路

- 基于某个预先假定的概率模型设计出最佳变长码，码字具有某种固定的结构
- 在牺牲一定压缩效率的前提下，提高解码器的可靠性

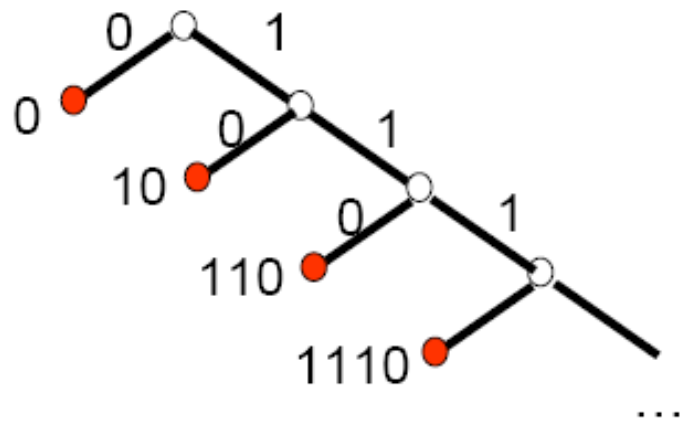


## 3.2.5.2一元码 (Unary Code)

- 对非负整数 $n$ ，用 $n$ 个1和一个0表示（或者 $n$ 个0和一个1）

➤ 为即时码

$n$	Codeword
0	0
1	10
2	110
3	1110
4	11110
5	111110
...	...



- 在什么情况下是最佳码？
  - 当概率为 $1/2$ 、 $1/4$ 、 $1/8$ 、 $1/16$ 、 $1/32$ ...
  - 此时Huffman码就是一元码

## 3.2.5.2一元码 (Unary Code)

### ■ 编码 (Encoding) :

```
UnaryEncode(n) {  
    while (n > 0) {  
        WriteBit(1);  
        n--;  
    }  
    WriteBit(0);  
}
```

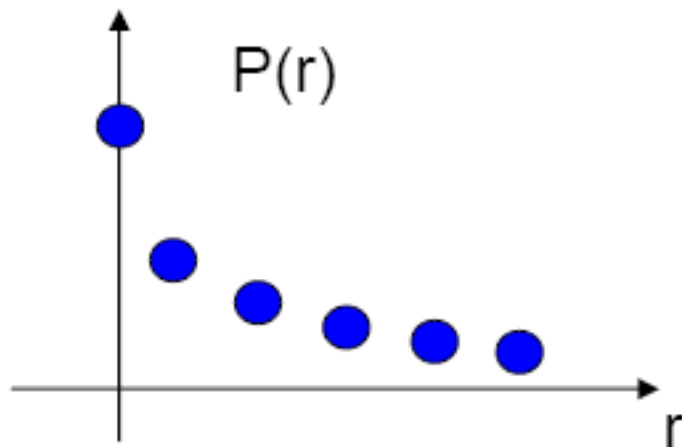
### ■ 解码 (Decoding)

```
UnaryDecode() {  
    n = 0;  
    while (ReadBit(1) == 1) {  
        n++;  
    }  
    return n;  
}
```

### 3.2.5.3 Golomb码

#### ■ 适用的编码对象

- 服从几何分布的正整数数据流，数据流中整数 $n$ 出现的概率为： $P(n)=(1-p)^{n-1}p$
- 例：游程 $r$ 的概率分布
  - $P(r = n) = p^n(1 - p)$       $n$ 个0接着一个1
  - 参数为 $p$ 的单边几何分布



# 3.2.5.3 Golomb码

## ■ 适用的编码对象

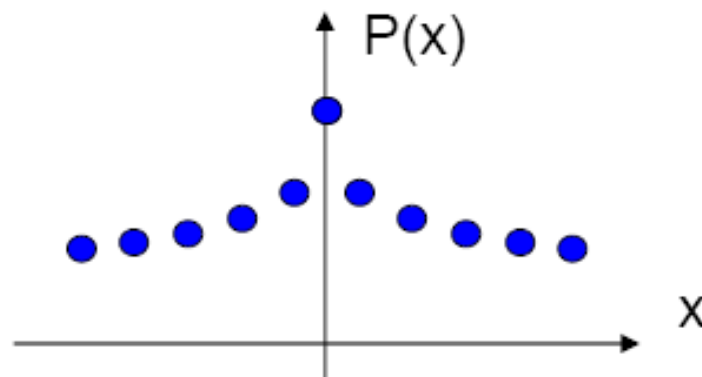
### ■ 例2：预测误差

$$e(n) = x(n) - \text{pred}(x(1), \dots, x(n-1))$$

- 大多数的  $e(n)$  的值都很小，在0附近 → 可用双边几何分布 (GD) 建模

### ■ 双边GD可用单边GD近似：

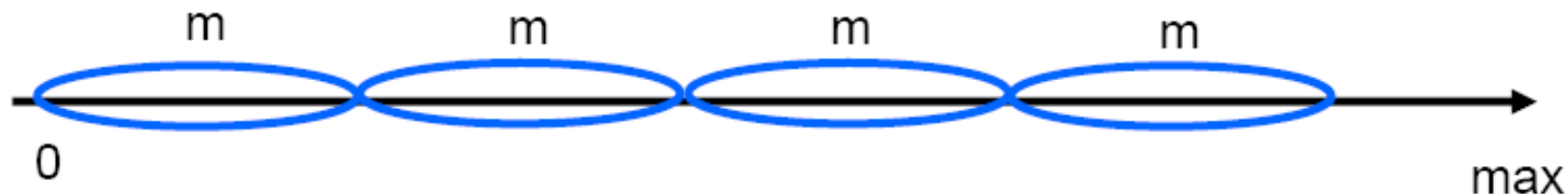
- 减少自适应熵编码中上下文的数目
- 将负数映射成正数：
  - 映射可能不止一个
  - 在JPEG-LS中有详细的规定



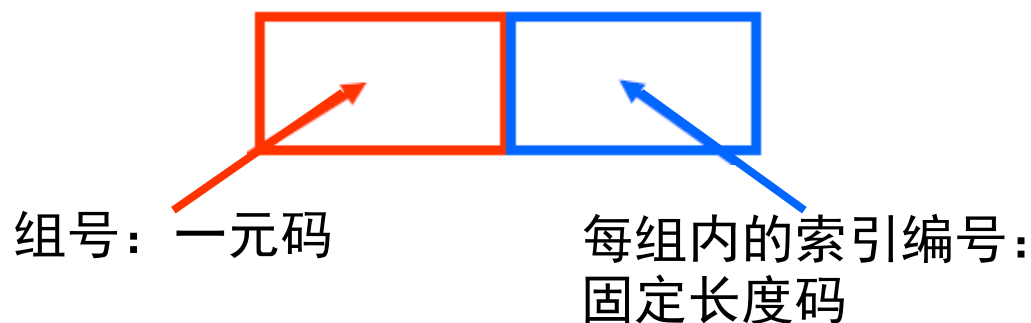
### 3.2.5.3 Golomb码

#### ■ 编码思想 – 多分辨率的方法

- 使用一个可调参数 $m$ 将输出码字分为若干个组，每组内均有码长相近的 $m$ 个码字 –  $\text{Golomb}(m)$



每个码字由两部分组成：



### 3.2.5.3 Golomb码

■ 编码过程 
$$n = qm + r = \left\lfloor \frac{n}{m} \right\rfloor m + r$$

$q$ : 商, 用一元码

$q$       Codeword

0	0
1	10
2	110
3	1110
4	11110
5	111110
6	1111110

... ..

$r$ : 余数, “固定长度”码

当  $m=2^k$  时,  $k$  比特

例:  $m=8$  000,001,...,111

当  $m \neq 2^k$  时,  $k = \lceil \log m \rceil$

$r < 2^k - m$ ,  $k-1$  比特

$r \geq 2^k - m$ ,  $k$  比特, 以  $k$  个 1 结束

例:  $m=5$  00,01,10,110,111

## 3.2.5.3 Golomb码

### ■ $m=5$ 时的Golomb码 (Golomb-5)

n	q	r	code
0	0	0	000
1	0	1	001
2	0	2	010
3	0	3	0110
4	0	4	0111

n	q	r	code
5	1	0	1000
6	1	1	1001
7	1	2	1010
8	1	3	10110
9	1	4	10111

n	q	r	code
10	2	0	11000
11	2	1	11001
12	2	2	11010
13	2	3	110110
14	2	4	110111

### 3.2.5.3 Golomb码 $m=2^k$ 时可以简化为 $G(k)$

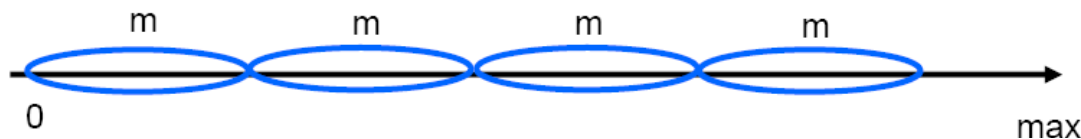
n	k = 0 , m=1	k = 1 ,m=2	k = 2 ,m=4	k = 3,m=8
1	0	0 0	0 00	0 000
2	10	0 1	0 01	0 001
3	110	10 0	0 10	0 010
4	1110	10 1	0 11	0 011
5	11110	110 0	10 00	0 100
6	111110	110 1	10 01	0 101
7	1111110	1110 0	10 10	0 110
8	11111110	1110 1	10 11	0 111
9	111111110	11110 0	110 00	10 000

- $m$ 较小, Golomb码初始很短, 但增长很快, 适合RLE中较小, 即大游程很少的情况
- $m$ 较大, 初始Golomb码很长, 但增长很慢, 适合RLE中较大, 即大游程较多的情况



### 3.2.5.4 指数Golomb码

- **Golomb码** -输出码字分为若干个组，  
每组内均有码长相近的 $m$ 个码字



- **Exp-Golomb码** - 组的大小呈指数增长
- 码字可视为 $G(0)$ 码加上 $q+m$ 位尾码

$q$ :一元码中0的个数;  $m$ : 阶数



- 优点: 可以根据闭合公式解析码字,  
硬件复杂度低

$m=0$

n	code
0	0
1	100
2	101
3	11000
4	11001
5	11010
6	11011
7	1110000
8	1110001
9	1110010
10	1110011
11	1110100
12	1110101
13	1110110
14	1110111
15	111100000