# 第2章 无失真信源编码

## 2.4 算术编码

（第6讲2007.10.9.）

- 计划学时：2.5学时
- 要求掌握的主要内容：

1. 深刻理解算术编码原理。

2. 熟练掌握作图与公式相结合计算积累概率的方法。

3. 了解算术编码实用技术上的困难与解决方案。

- 重点难点：

重点----算术编码原理。

难点---- 计算积累概率的递推算法。

- 外语关键词：

算术编码：**Arithmetic Coding**

序列概率：**Symbol String Probability**

积累概率：**Cumulative Probability**

递推公式：**Recursion Formula**

作图法：**Diagraph Method**

流编码技术：**Stream Coding Technology**

计算精度：**Precision in Calculation**

进位问题：**Problem of Carry**

# [温旧引新]

- 变长码编码原理-----概率匹配原则：

$$\log_r(1/p_i) \leqslant l_i \leqslant 1+\log_r(1/p_i)$$

经常出现的符号采用较短的码字表示，不常出现的符号采用较长的码字表示，使平均码长最短。

- 信源序列的概率：

无记忆信源：单个符号概率之积；

有记忆信源：各阶条件概率之积；

# 2.4 算术编码

- 从结构上分，编码有两种方式：

（1）分组编码方式(块码)：为信源的每个字符(或字符组)建立编码的码字。

（2）序列编码方式(流码)：直接为整个信源消息序列寻找编码序列。

- 等长码和变长码的编码，都是“块码”，本节讨论的算术编码属于“流码”，直接把信源发出的非等概序列变换成等概序列。
- 由于等概序列单位码元信息荷载量大，携带同样多的信息需要的代码少，编码后序列长度就会变短。

# 2.4.1 有限长序列的算术编码原理

## 1．二元等概序列的积累概率：

- 二元等概无记忆信源发出长度为$L$的随机序列：

$$S_i=a_{L-1}a_{L-2}\cdots\cdots a_2a_1a_0$$

其中$a_r$ $(r=0,1\ \cdots\cdots L-1)$取1或取0，概率均为1/2；

- 事实上，序列$S_i$可以是$L$位自然码从000……0到111……1之间$2^L$个码字的任何一个。比如$L=3$，$S_i$是000，001，……111之中的任何一个。

## (1) 序列概率：

$p(S_i) = p(a_{L-1}a_{L-2}\cdots\cdots a_2a_1a_0)$ -----由于无记忆

$= p(a_{L-1})p(a_{L-2})\cdots\cdots p(a_1)p(a_0)$

$= (1/2)^L = 2^{-L}$ ------由于等概。

- 结论：长度为$L$的二元等概无记忆信源序列$(S_i)$，共$2^L$个，概率都相等，为$2^{-L}$；
- 把这些序列的概率按照自然码顺序排在数轴上，因总概率为1，故恰好排满[0, 1]区间。

## (2)积累概率：

- 定义序列$S_i$的积累概率$F(S_i)$是按照自然码排序顺列在$(S_i)$前面的所有序列的概率之和：

$$F(S_i)= p(S_0)+ p(S_1)+ p(S_2)+\cdots\cdots+ p(S_{i-1})$$

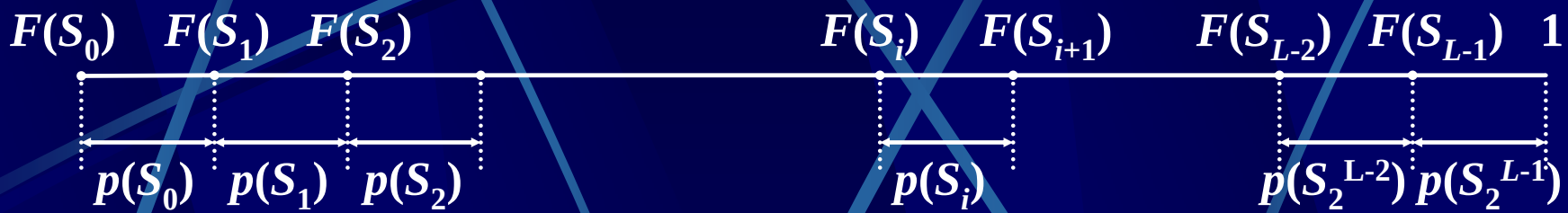
比如，$F(S_0)=0$, $F(S_1)=p(S_0)$，$F(S_2)=p(S_0)+p(S_1)$，

$$F(S_3)= p(S_0)+p(S_1)+p(S_2)，\ \cdots\cdots$$

- 序列概率与积累概率之间的关系为：

$$F(S_{i+1})=F(S_i)+p(S_i) \quad 或 \quad p(S_i)=F(S_{i+1})-F(S_i)$$

等概序列的积累概率在[0，1]区间的分布为




- 如果把这段数轴看作一把概率标尺，积累概率则给出了这把等分度标尺的刻度值，序列概率给出了最小分度的长度。$F(S_i)$总是$p(S_i)$ 段的左端点。
- 序列长度为$L$时，等概标尺的最小刻度为$2^{-L}$。
- 如 $L$=3 时； 对于 $(S_5)=(101)_2$， ∵ $p(S_i)\equiv 1/8$，
∴ $F(S_5)= p(S_0)+p(S_1)+p(S_2)+p(S_3)+p(S_4)=5/8$;

- 用二进制小数表示：

$$F(S_5)= 5\times 1/8= (101)_2 \times 2^{-3} =(0.101)_2$$

- 结论：(1) 等溉序列的概率在数轴上均匀分布，最小分度为$2^{-L}$；(2)等溉序列的积累概率是该序列为有效数字的二进制纯小数。

[例]已知$L$=5; $S_i$=11101; 求序列概率与积累概率。

解： $p(S_i)\equiv 2^{-L}=1/32=(0.00001)_2$ ，

$F(S_i)=(0.11101)_2=1/2+1/4+1/8+1/32=29/32$;

## 2．二元非等概序列的积累概率：

假设信源无记忆：$p(0)= p_0 =1/4$，$p(1)= p_1 = 3/4$；

### (1)序列概率：

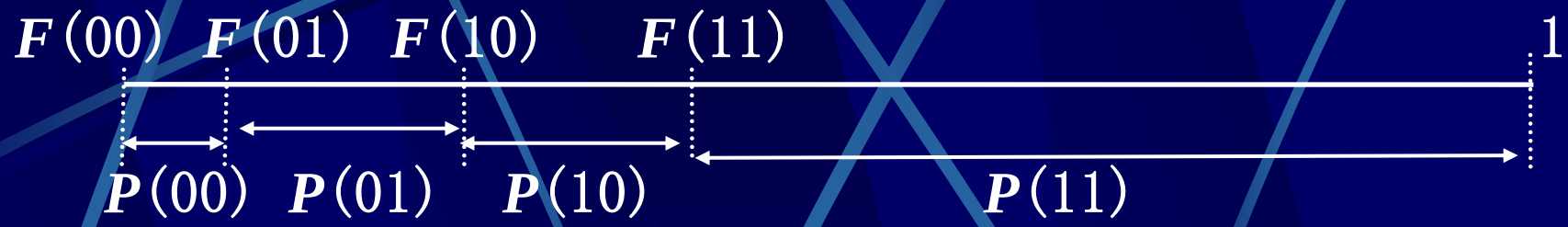
如$S_i$=11101; $p(S_i)= p_1^4 p_0 =(3/4)^4(1/4)=81/1024$;

显然序列中1与0的个数不同，序列概率便不同。各个$p(S_i)$在概率数轴上是非均匀分布的：

如 $L$=2时，$S_0$=00, $S_1$=01, $S_2$=10, $S_3$=11；则：

$p(00)=1/16$，$p(01)= p(10)=3/16$，$p(11)=9/16$;

非等概序列的积累概率在[0，1]区间的分布为：




不难求出：

$F(00)=0, F(01)=1/16, F(10)=1/4, F(11)=7/16;$

表达成为二进制小数，4个积累概率分别是：

$F(00)=0, F(01)=0.0001, F(10)=0.01, F(11)=0.0111$：

显然都不是原序列小数点的简单移动。

- 结论：(1) 非等溉序列的概率在数轴上非均匀分布；(2)等溉序列的积累概率与原序列没有小数点简单移位的规律。

[例]已知$L=5$; $S_i=11101$; 求序列概率与积累概率。

解： $p(S_i)= p_1^4 p_0 = 81/1024$； ，

$$F(S_i) =1-p(11111)-p(11110)-p(11101)=$$

$$=1-(3/4)^5-(3/4)^4(1/4)\times 2=619/1024$$

$$=(0.1001101011)_2$$

## 3. 算术编码原理：

- 如果把上例求出的非等概序列$S_i$=11101的积累概率$(0.1001101011)_2$它看作是另一个等概序列$C_i$=1001101011的积累概率值，那么通过积累概率这座桥梁，就建立起了非等概序列$S_i$与等概序列$C_i$之间的对应关系。
- 算术编码就是寻找与信源序列积累概率相同的等概序列的过程。
- 或者说，算术编码是用等分度概率尺去测量非均匀分布的积累概率数轴的结果。

- 编码方法：(1)计算信源序列的积累概率；(2)去掉积累概率的小数点；(3)截取码字。
- 序列$S_i$=11101的自信息$I(S_i)=-\log p(S_i)=3.66$ bit，
- 等概序列$C_i$的信息熵是 1 bit/符号，只须4位符号就足以承载3. 66 bit 的信息。
- 可以对$C_i$=1001101011四舍五入截取4位代码1010，作为信源序列$S_i$的编码，就比原来少用1位代码。
- 压缩比$K$=4/5=0.8，效率 $\eta$=3.66/4=0.915;

## 4. 唯一可译性的讨论：

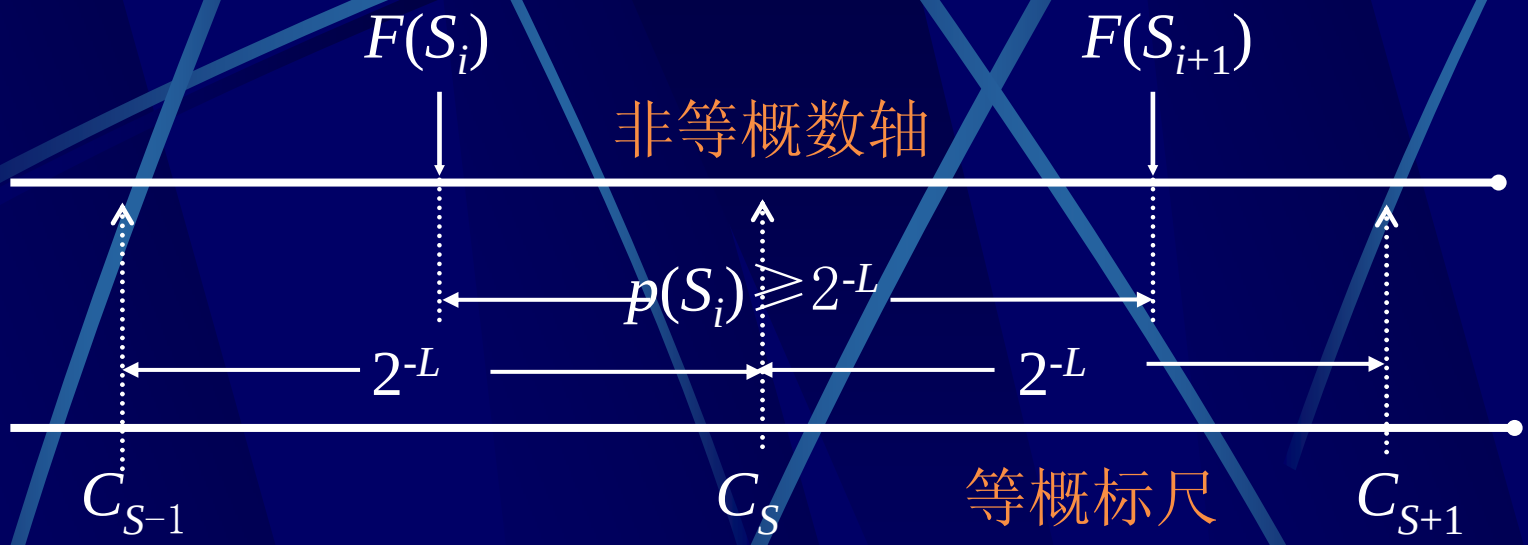
- 可以证明这样得到的码字一定是唯一的，没有第二个信源序列会对应着同一个码字。

- 只要按照概率匹配原则：$L \geqslant I(S_i)=-\log p(S_i)$ 来截取码长，就有：$p(S_i) \geqslant 2^{-L}$

  即：$F(S_{i+1}) - F(S_i) \geqslant 2^{-L}$

- $F(S_{i+1})$与$F(S_i)$是信源最近邻的两个序列的积累概率，$2^{-L}$是等概序列的最小分度，连最近邻的两个序列的积累概率也不能放在同一个最小分度内，表明必然有不同的码字来对应。

- $F(S_{i+1})$和$F(S_i)$位于概率标尺两个不同的分度格中




- 练习：计算等概与非等概（$p_0$ =1/4, $p_1$ = ¾）两种情况下S=1101的序列概率与积累概率。

# 2.4.2 无限长序列的算术编码原理

- 对有限长序列进行算术编码实际上还是块码。
- 对无限长序列，随着序列的延伸，$S_i$在变化，$p(S_i)$ 与$F(S_i)$ 也都在不断变化，我们不能等序列全部输入完结再编码，作为“流码”，是边输入、边编码、边输出的。

## 1．递推公式：

设当前序列为$S_i$，当又发出一个符号$a_r$时，序列延伸为 $S_{i+1}=S_i a_r$（表示接在后面）

- 对于无记忆信源，延伸序列的序列概率为：

$$p(S_{i+1})=p(S_i a_r)=p(S_i)p(a_r)=p(S_i)p_r \qquad (1)$$

式中：
$$p(a_r)=p_r=\begin{cases}p_0 & \text{当} a_r=0\\ p_1 & \text{当} a_r=1\end{cases}$$

由于：$p(S_j0)+p(S_j1)=p(S_j)p_0+p(S_j)p_1=p(S_j)(p_0+p_1)=p(S_j)$

所以：$$F(S_i0)=\sum_{j=0}^{i-1}[p(S_j0)+p(S_j1)]=\sum_{j=0}^{i-1}p(S_j)=F(S_i)$$

$$F(S_i1)=F(S_i0)+p(S_i0)=F(S_i)+p(S_i)p_0$$

引入：

$$F_r = \begin{cases} F_0 = 0 & \text{当} a_r = 0 \\ F_1 = p_0 & \text{当} a_r = 1 \end{cases}$$

两种情况就能合并，得到延伸序列的积累概率：

$$F(S_i a_r) = F(S_i) + p(S_i) F_r \qquad (2)$$

下面以$S$=（11101……）来演绎这个递推过程：

假定$p(0) = p_0 = 1/4$，$p(1) = p_1 = 3/4$

初态$S_0$=｛$\phi$｝表示空序列，这时：$p(S_0)$=1，　$F(S_0)$=0；

若发出第1个符号$a_r$=1，则$S_1 = \phi a_r = a_r = 1$，计算过程如下：

- S1=1: $p(S_1)= p(S_0)\, p_1=3/4;$

  $F(S_1)=F(S_0)+ p(S_0)\, p_0=1/4;$
- S2=11: $p(S_2)= p(S_1)\, p_1=(3/4)\times (3/4)=9/16;$

  $F(S_2)=F(S_1)+ p(S_1)\, p_0=1/4+ (3/4)\times (1/4) =7/16;$
- S3=111: $p(S_3)= p(S_2)\, p_1=(9/16)\times (3/4)=27/64;$

  $F(S_3)=F(S_2)+ p(S_2)\, p_0=7/16+ (9/16)\times (1/4) =37/64;$
- S4=1110: $p(S_4)= p(S_3)\, p_0=(27/64)\times (3/4)=27/256;$

  $F(S_4)=F(S_3) =37/64;$
- S5=11101: $p(S_5)= p(S_4)\, p_1=(27/256)\times (3/4)=81/1024;$

  $F(S_5)=F(S_4)+ p(S_4) p_0=37/64+ (27/256)\times (1/4) =619/1024;$

## 2. 图解分析：

- 当序列无限延伸时，序列概率$p(S_i)$和积累概率$F(S_i)$在概率数轴上的行为如何呢？

（1）序列概率$p(S_i)$是概率数轴上某一区间的长度，积累概率$F(S_i)$永远是该区间的左端点。

（2）序列$S_i$每延伸一个符号$a_r$，即序列多一位，概率区间$p(S_i)$就被细分一次，细分是按$p_0:p_1$进行的，左边是$p(S_i0)$，右边是$p(S_i1)$。究竟在那边，取决于符号$a_r$是0还是1。

（3）一旦$a_r$给定，序列概率所在的小区就被确定，以后再继续延伸，只会在本小区内细分定位，不会跑出该小区。

（4）序列越长，概率区间$p(S_i)$的宽度就越窄，数值就越小；而积累概率$F(S_i)$的小数位数就越多。

（5）序列无限延伸意味着概率轴区间无限细分，序列概率$p(S_i)$像一个无理数，无限接近地趋于数轴某个位置。

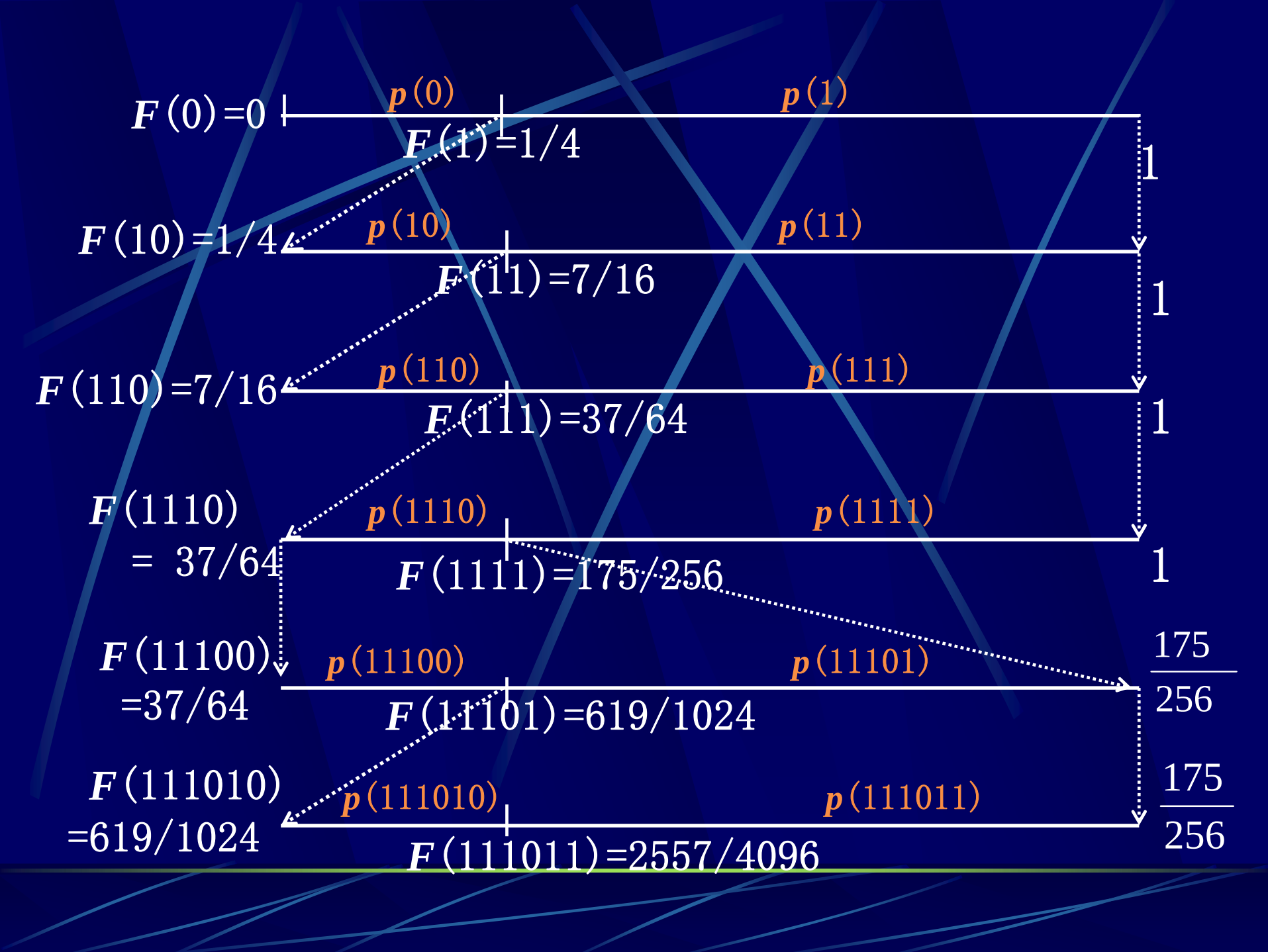
（6）正如测量微小长度需要刻度更精细的量具一样，为了便于作图，便于演示与计算，往往把序列概率小区“放大”后，再继续细分。

下面以 $S$=(11101……）来演绎这个递推过程：

仍然假定 $p(0)= p_0 =1/4$, $p(1)= p_1 =3/4$

初态$S_0$=｛$\phi$｝表示空序列，这时：$p(S_0)$=1， $F(S_0)$=0；

发出第1个符号$a_r$=1, [0, 1]取间分成1:3两段，$p(S_1)$进入右区。


F(0)=0
p(0)
p(1)
F(1)=1/4
1
F(10)=1/4
p(10)
p(11)
F(11)=7/16
1
F(110)=7/16
p(110)
p(111)
F(111)=37/64
1
F(1110)
= 37/64
p(1110)
p(1111)
F(1111)=175/256
1
F(11100)
=37/64
p(11100)
p(11101)
F(11101)=619/1024
175
256
F(111010)
=619/1024
p(111010)
p(111011)
F(111011)=2557/4096
175
256

[例1]已知二元无记忆信源发信概率为$p_0$=0.2, $p_1$=0.8，计算：

（1）信源发出序列S=1101的序列概率与积累概率；

（2）当信源又继续发出两个符号为11或10时，其序列概率与积累概率分别为多少？

（3）试为110111寻求算术编码。

解：(1)序列概率$p(S)= p_0 p_1^3$=0.2×0.8$^3$=0.1024;

积累概率$F(S)$= 1−$p$(1101)−$p$(1110)−$p$(1111)=

=1−0.1024 ×2−0.8$^4$=0.3856;

(2)利用递推公式，先延伸1个符号，计算$S1$：

$p(S1)= p(S)\, p_1$= 0.1024 ×-0.8=0.08192;

$F(S1)= F(S)+ p(S)\, p_0$= 0.3856+ 0.1024 ×0.2=0.40608;

已经求出$p(S1)$=0.08192; $F(S1)$= 0.40608;

当再延伸第2个符号为0时：

$p(S10)= p(S1)\ p_0$= 0.08192 ×0.2= 0.016384;

$F(S10)= F(S1)$= 0.40608;

当再延伸第2个符号为1时：

$p(S11)= p(S1)\ p_1$= 0.08192 ×0.8= 0.065536;

$F(S11)= F(S1)+ p(S1)\ p_0$= 0.40608+ 0.08192 ×0.2=

=0.422464=$(0.0110110\ldots\ldots)_2$;

(3) ∵ $-\log p(110111)= -\log 0.065536 \approx 3.93$

∴C (110111)=0111

# 2.4.3 流编码技术

理论上递推算法可以一直进行下去，但是，随着序列的不断延伸，$p(S_i)$的数值越来越小，$F(S_i)$的小数位数越来越多，计算量不断增加，任何计算机的容量和精度 (容许小数位数) 都是有限的，由此会带来很多实际困难需要解决。

## 1. 计算复杂性问题：

计算复杂主要来自乘法运算。如果把符号概率$p_0$用与它接近的$2^{-m}$代替，那么乘以$p_0$就变成小数点左移$m$位，使计算大大简化。计算表明，当$p_0$界于$2^{-m}$和$2^{-m-1}$时，这种代替引入的误差不超过5%；

## 2. 计算精度问题：

- 随着序列的延伸，积累概率小数变得很长，一旦超过计算机容许的位数，使相邻区间难于区别，编码失去唯一性。而序列概率变得很小，甚至出现机器零，造成无码可编的情况。

- 考虑到当序列很长时，积累概率小数的高位部分已基本不变，可以让不变的部分移到小数点左面，而让低位（仍处于不断变化的部分）移到计算机方便的位置继续计算。而$p(S_i)$的高位小数都是0，有效数字并不会因为移位而改变。

- 这种做法正如图解示例中，每次细分区间后，都把子区间又扩大到同样大小，以便下次细分时作图更精确一些，这样做不会影响计算结果。

## 3. 进位问题：

从小数高位移出来的数据，如果是1，则暂不要立刻输出，这是因为万一后面出现了进位，前面数据又可能需要修改。为此，需要一个计数器记录1的个数。并设想小数点前面本来就有一个0，算法如下：

（1）移出1，则计数器加1，暂不输出编码；（等待状态）

（2）移出0，则可以输出最前面的那个0和计数器中的Q个1，只留下最后的这个0等待即可；

（3）来了进位，则输出一个1和 Q-1个0；（因为计数器中已经等待了Q个1，一旦有进位，则它们都被置成0，最前面的0被置1。需要留下最后那个0，其余都可输出）。

（4）每次输出后，计数器都清零。

# 2.4.4 编、译码示例

- 设： $p_0=1/4=(0.01)_2$，$p_1=3/4=(0.11)_2$ ；

## 1. 编码过程：

假设计算过程只保留2位小数，一旦发现 $p(S)<1$，就移动小数点1位或2位，使$p(S)$不小于 1。

例如，编码输入序列为11111100……，编码过程见表所示。编码输出码字为10101000……；

- 初 态 ： $S_0$= 空 串 ，$p_0=(0.01)_2$，

  $p(S_0)=1$， $F(S_0)=0$，

- 输入“1”：

  $p(S_1)= p(S_0 1)= p(S_0)p_1$

  $=0.11$

  $F(S_1)= F(S_0)+ p(S_0)p_0$

  $=0.01$

- 由于$p(S_1)$已小于1，须将小数点右移1位，并保留2位小数得到：

  $p(S_1)=1.10$，$F(S_1)=0.10$

| 编码过程 | | | |
|---|---|---|---|
| 输入序列 | $p(S)$ | $F(S)$ | 编码输出 |
| $S_0$ | 1 | 0 | |
| 1 | 0.11 | 0.01 | |
| (移1位) | 1.10 | 0.10 | |
| 1 | 1.00 | 0.11 | |
| 1 | 0.11 | 1.00 | |
| (移1位) | 1.10 | 10.00 | 1 |
| 1 | 1.00 | 10.01 | |
| 1 | 0.11 | 10.10 | |
| (移1位) | 1.10 | 101.00 | |
| 1 | 1.00 | 101.01 | |
| 0 | 0.01 | 101.01 | |
| (移2位) | 1.00 | 10101.00 | 01 |
| 0 | 0.01 | 10101.00 | |
| (移2位) | 1.00 | 1010100 | 010 |

- 又输入“1”：

$p(S_2)=p(S_1)p_1$

$=1.10\times0.11=1.00$

$F(S_2)=F(S_1)+p(S_1)p_0$

$=0.10+1.10\times0.01=0.11$

- 再输入1：

$p(S_3)=p(S_2)p_1=1.00\times0.11=0.11$

$F(S_3)=F(S_2)+p(S_2)p_0$

$=0.11+1.00\times0.01=1.00$

- 以下雷同。是边输入边计算边输出的。

| 编码过程 | | | |
|---|---|---|---|
| 输入序列 | $p(S)$ | $F(S)$ | 编码输出 |
| $S_0$ | 1 | 0 | |
| 1 | 0.11 | 0.01 | |
| (移1位) | 1.10 | 0.10 | |
| 1 | 1.00 | 0.11 | |
| 1 | 0.11 | 1.00 | |
| (移1位) | 1.10 | 10.00 | 1 |
| 1 | 1.00 | 10.01 | |
| 1 | 0.11 | 10.10 | |
| (移1位) | 1.10 | 101.00 | |
| 1 | 1.00 | 101.01 | |
| 0 | 0.01 | 101.01 | |
| (移2位) | 1.00 | 10101.00 | 01 |
| 0 | 0.01 | 10101.00 | |
| (移2位) | 1.00 | 1010100 | 010 |

## 2. 译码原理：

输入的是编码1010100……，以它为有效数字的纯小数C就是积累概率。

编码时，当S后面接0时，进入左边子区间，其码字必然小于$F(S1)$；当S后面接1时，进入右边子区间，其码字必然不小于$F(S1)$；

因此，译码规则为：

（1）若$C<F(S1)$，即$C-F(S)<p(S)p_0$时，可判$a_r=0$；

（2）若$C\geqslant F(S1)$，即$C-F(S)\geqslant p(S)p_0$时，可判$a_r=1$；

## 3. 译码过程：

- 初态：$S_0$=空串，$p(S_0)=1$，$F(S_0)=0$，　$p_0=(0.01)_2$，
- 输入“1”：$C=0.10$时，（保留2位小数）

$$C\text{-}F(S_0)=0.1>p(S_0)p_0=0.01,\ \text{可判}a_r=1;$$

- 递推求$p(S_1)$ 和$F(S_1)$的结果，编码过程已讲，见表左半边。
- 译码过程见表的右半边。

| 编码过程 | | | | 译码过程 | | | | |
|---|---|---|---|---|---|---|---|---|
| 输入序列 | $p(S)$ | $F(S)$ | 编码输出 | 译码输入 | $C\text{-}F(S)$ | 比较 | $p(S)p_0$ | 译码输出 |
| $S_0$ | 1 | 0 | | 1 | 0.1-0 | > | 0.01 | 1 |
| 1 | 0.11 | 0.01 | | | | | | |
| (移1位) | 1.10 | 0.10 | | 0 | 1.00-0.10 | > | 0.01 | 1 |

- 然而$p(S_1)=p(S_0)p_1=0.11$，$F(S_1)=0.01$；应将小数点右移1位，得到$p(S_1)=1.10$和$F(S_1)=0.10$，当然C也移位为1.00；
- 输入“0”：$C\text{-}F(S_1)=0.1>p(S_1)p_0=0.01$，仍判$a_r=1$;
- 以后相同，不在赘述（见表）。

| 编码过程 | | | | 译码过程 | | | | |
|---|---|---|---|---|---|---|---|---|
| 输入序列 | $p(S)$ | $F(S)$ | 编码输出 | 译码输入 | $C\text{-}F(S)$ | 比较 | $p(S)p_0$ | 译码输出 |
| $S_0$ | 1 | 0 | | 1 | 0.1-0 | > | 0.01 | 1 |
| 1 | 0.11 | 0.01 | | | | | | |
| (移1位) | 1.10 | 0.10 | | 0 | 1.00-0.10 | > | 0.01 | 1 |

| 编码过程 | | | | 译码过程 | | | | |
|---|---|---|---|---|---|---|---|---|
| 输入序列 | $p(S)$ | $F(S)$ | 编码输出 | 译码输入 | $C$-$F(S)$ | 比较 | $p(S)p_0$ | 译码输出 |
| $S_0$ | 1 | 0 | | 1 | 0.1-0 | > | 0.01 | 1 |
| 1 | 0.11 | 0.01 | | | | | | |
| (移1位) | 1.10 | 0.10 | | 0 | 1.00-0.10 | > | 0.01 | 1 |
| 1 | 1.00 | 0.11 | | 1 | 1.01-0.11 | > | 0.01 | 1 |
| 1 | 0.11 | 1.00 | | | | | | |
| (移1位) | 1.10 | 10.00 | 1 | 0 | 10.10-10.00 | > | 0.01 | 1 |
| 1 | 1.00 | 10.01 | | 1 | 10.101-10.01 | > | 0.01 | 1 |
| 1 | 0.11 | 10.10 | | | | | | |
| (移1位) | 1.10 | 101.00 | | 0 | 101.01-101.00 | = | 0.01 | 1 |
| 1 | 1.00 | 101.01 | | 0 | 101.010-101.01 | < | 0.01 | 0 |
| 0 | 0.01 | 101.01 | | | | | | |
| (移2位) | 1.00 | 10101.00 | 01 | 0 | 10101.00-10101.00 | < | 0.01 | 0 |
| 0 | 0.01 | 10101.00 | | | | | | |
| (移2位) | 1.00 | 1010100 | 010 | | | | | |

# 小结：

## ❖算术编码的原理：

通过计算积累概率找到相应的等概序列。

根据概率匹配原则在等概序列上截取码字。

## ❖ 流编码原理：

积累概率的递推计算与图示分析；

解决三个具体问题（计算复杂性问题、计算精度问题和进位问题）。

# 课后复习题

- 思考题：

能不能用对多符号信源进行算术编码？怎样进行

- 作业题：

教材第64页习题二第12、13题；

# 第2章 无失真信源编码

## 2.6 通用编码

（第7讲 2007.10.16.）

- 计划学时：2学时

- 要求掌握的主要内容：

  1. 深刻理解L-Z编码的基本原理和意义。
  2. 熟练掌握字典编码有关算法：
  3. 学习通过编程解决编码的思路。

- 重点难点：

  重点----字典编码原理

  难点----字典编码算法

- 外语关键词：

通用编码：**Universal Coding**

字典编码：**Dictionary Coding**

当前字符：**Current Character**

序号：**Serial Number**

内存单元：**Memory Unit**

递归算法：**Recursion Arithmetic**

自适应算法：**Self-adaptive Arithmetic**

# [温旧引新]

- 已经学过的信源编码方法有那些？

  ***Huffman***编码、游程编码、算术编码。

- 他们的基本原理有和共同点？

  都是依据概率匹配原理。

- 如果不清楚信源概率怎么办？

# 2.6 通用编码

- 赫夫曼编码与算术编码都要预先知道信源符号的概率分布。实际问题中往往无法知道或没有必要去统计信源各个符号的概率，希望有一种通用的非概率的编码方法。我们把这种不依靠概率知识就能进行压缩编码的方法叫做通用编码。
- 由于通用，因而普适。它已经成为一种应用广泛的文件压缩技术。
- 现已找到多种通用编码方法，如目前在计算机上常用的**ZIP**、**RAR**等。

- 最有影响力的通用编码是L-Z码。它是以色列的研究人员兰培尔(A.Lamprl)和齐夫(J.Ziv)于20世纪70年代提出的。
- 鉴于实际各类文件中总有许多字词、短语、甚至段落会经常重复出现，就为L-Z码的发明提供了可以利用的契机。
- L-Z码的基本思路是把信源序列分成许多长短不同的小段，凡是后面出现了前面出现过的段落时，就不重复输出，而用代号表达，使文字数量减少，长度变短。

- 尽管L-Z码并没有刻意地统计每个字符的概率，但是编码过程中查看是否有前面出现过的词语，这本身就是一种无意地统计，它属于边统计边编码的自适应编码方法。
- 沿着这个思路，L-Z码不断得到改进：先是L-Z分段编码，后是L-Z指针编码(**LZ77**和LZSS)，最后发展为词典编码(LZ78和**LZW**)。越改方法越简单，越实用，压缩效果也越好。
- 今就指针编码与字典编码展开讲述。它们是通用编码的典型代表，目前已得到广泛应用。

# 2.6.1 指针编码

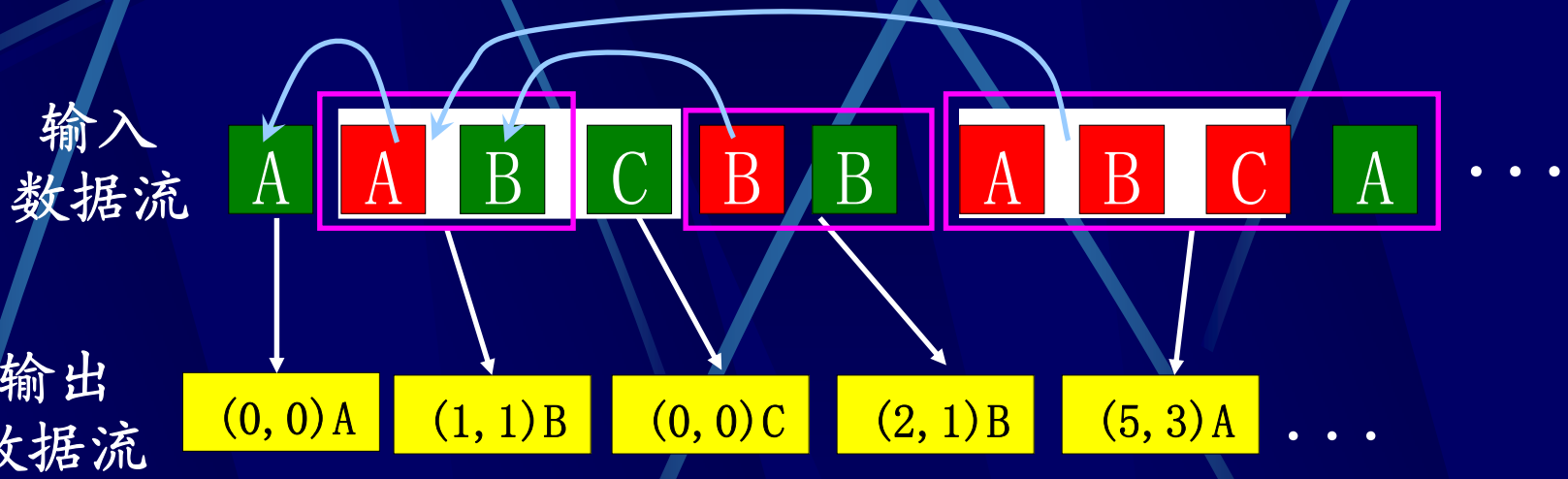
## 1. LZ77算法

- 此算法**1977**年兰培尔和齐夫提出。
- 原理：当待编字符串在早先输出的数据流中已经出现过时，则不必重复输出，而用指向早先那个字符串（称为匹配字符串）的指针（指示匹配字符串的位置）来代替。
- 算法：编码以输入码流的字符顺序进行，依次对当前字符串寻找匹配字符串，直到找不到更长的匹配字符串为止。

- 所找到的最长的匹配字符串，用指针(*x*，*y*)来表示，并用它代替当前待编字符串。其中：*x*表示匹配字符串出现在当前待编字符串之前的位置（按字符个数计算），*y*表示匹配字符串的长度。
- *C*表示当前待编字符串的下一个待编字符。因为当前匹配字符串再接上这个字符后，就成为前面找不到的字符串了。
- 编码格式以每个指针接一个字符为一个单元。整个输出码流的数据流格式为：

$(x_1, y_1)C_1(x_2, y_2)C_2(x_3, y_3)C_3\cdots\cdots$

# LZ77举例：




空指针**(0, 0)** 表示当前字符是第一次出现的字符。

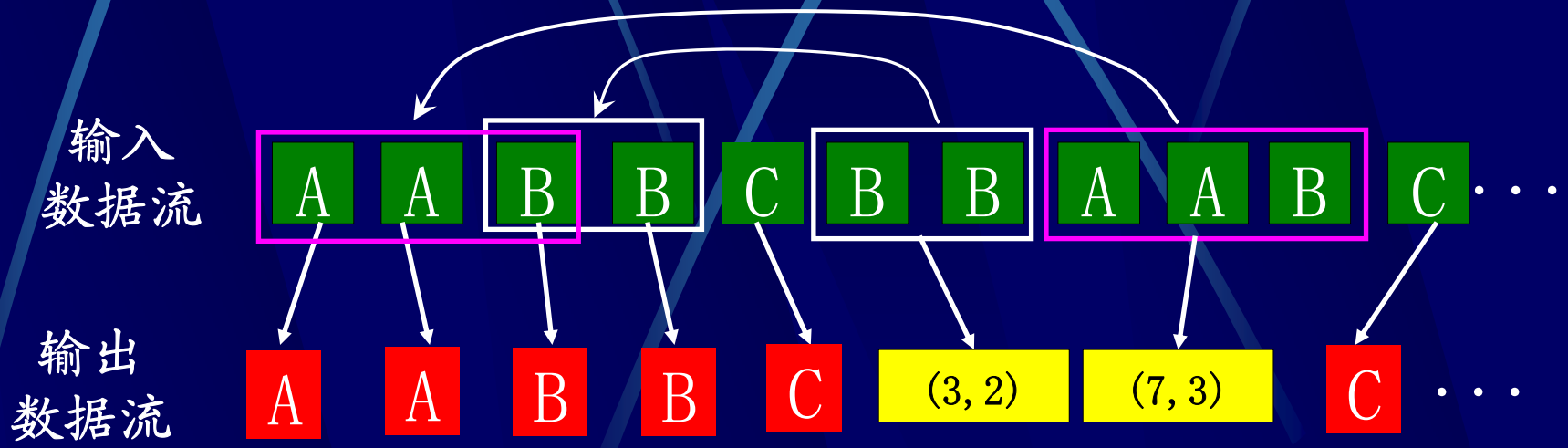
红色代表有匹配字符串，绿色代表无匹配字符串。

# 2．LZSS算法

- **LZSS**算法**1982**年由**Storer**和**Szymanski**改进，比**LZ77**可获得比较高的压缩比，而译码同样简单。
- **LZSS**算法对每一个找到的匹配字符串，都先比较一下它与相应指针代码的长短，若实际匹配字符串比指针更短，则不必用指针来替代匹配字符串，只有匹配字符串比指针更长时，才用指针来替代它。
- 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**。

- **LZSS算法举例：**

只有匹配字符串长度不小于2时，才用指针来代替。



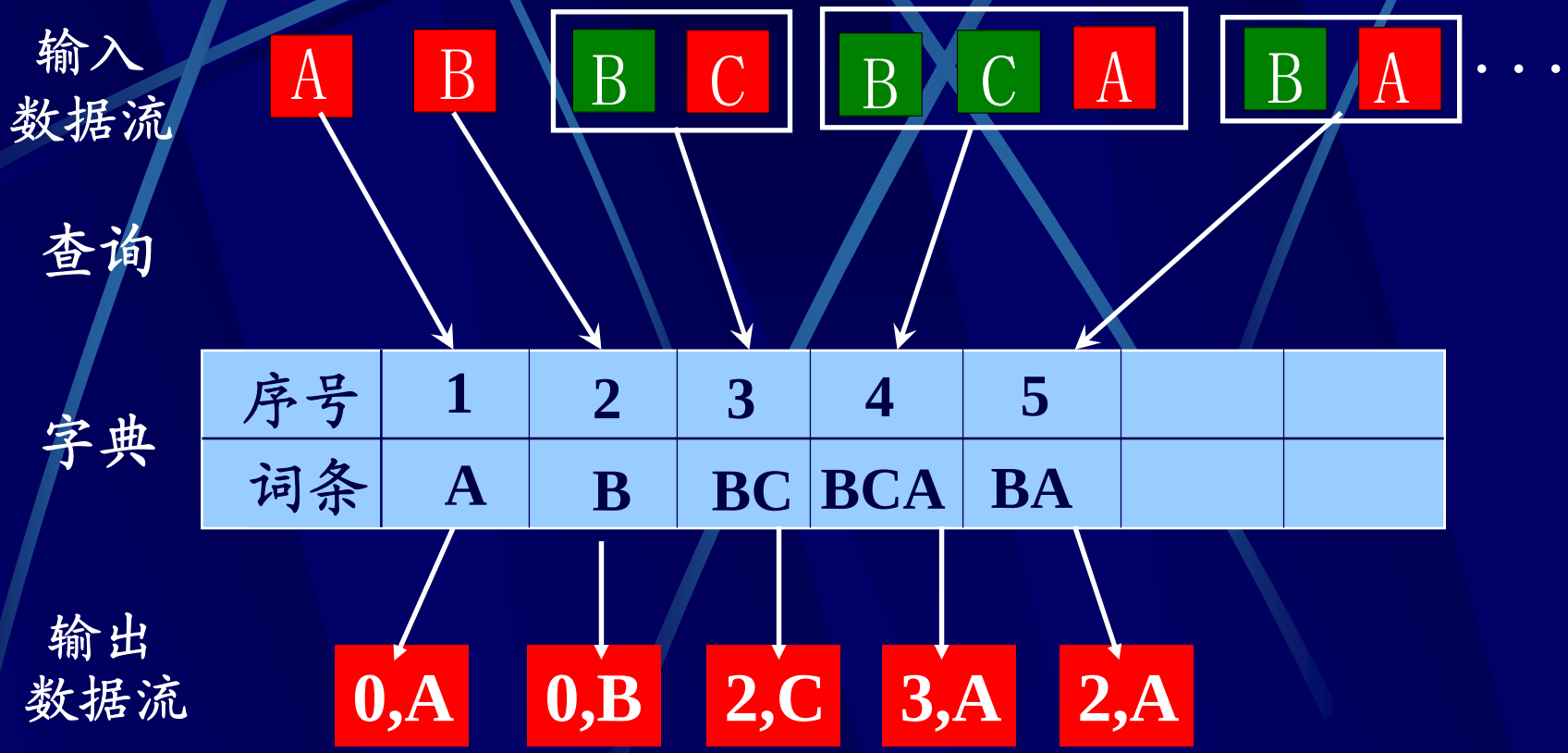
# 2.6.2 词典编码原理

## 1. LZ78词典编码算法：

- 1978年兰培尔和齐夫提出，用计算机进行压缩文件操作时，在内存中临时建立一个“词典”，词典中的“词条”就是读入的字符序列中出现过的各个字符串。
- 以后再遇到该词条时，就用它在词典中的序号(地址)代替。
- 编码过程一方面是查字典：对新输入的字符串寻找词典中最长的匹配词条；另一方面是写字典：把没找到的新词条不断扩充进词典中。
- 编码完成之后，词典(存储空间)不必保留，可从内存删除，译码时能够重新建立这个词典。

- 词典初始为空，用序号0表示，新词条从1开始添加。
- 编码从文档头第一个字符开始，首先是寻找当前字符在词典中的匹配词条，若找到，应再纳入下一个字符，继续寻找两字符的匹配字符串；若找到了，则再扩大进一个字符，继续找三字符的匹配字符串；如此进行下去，直到找不到更长的匹配字符串为止。
- 将这个没找到的字符串写入词典，成为下一序号的词条。
- 同时将这个没找到的字符串以(*n*,S)的格式编码输出。*n*为所找到的最长匹配词条在词典中的序号，S为当前待编字符。

# LZ78编码举例：


输入
数据流
A
B
B C
B C A
B A
· · ·
查询
字典
序号 1 2 3 4 5
词条 A B BC BCA BA
输出
数据流
0,A
0,B
2,C
3,A
2,A

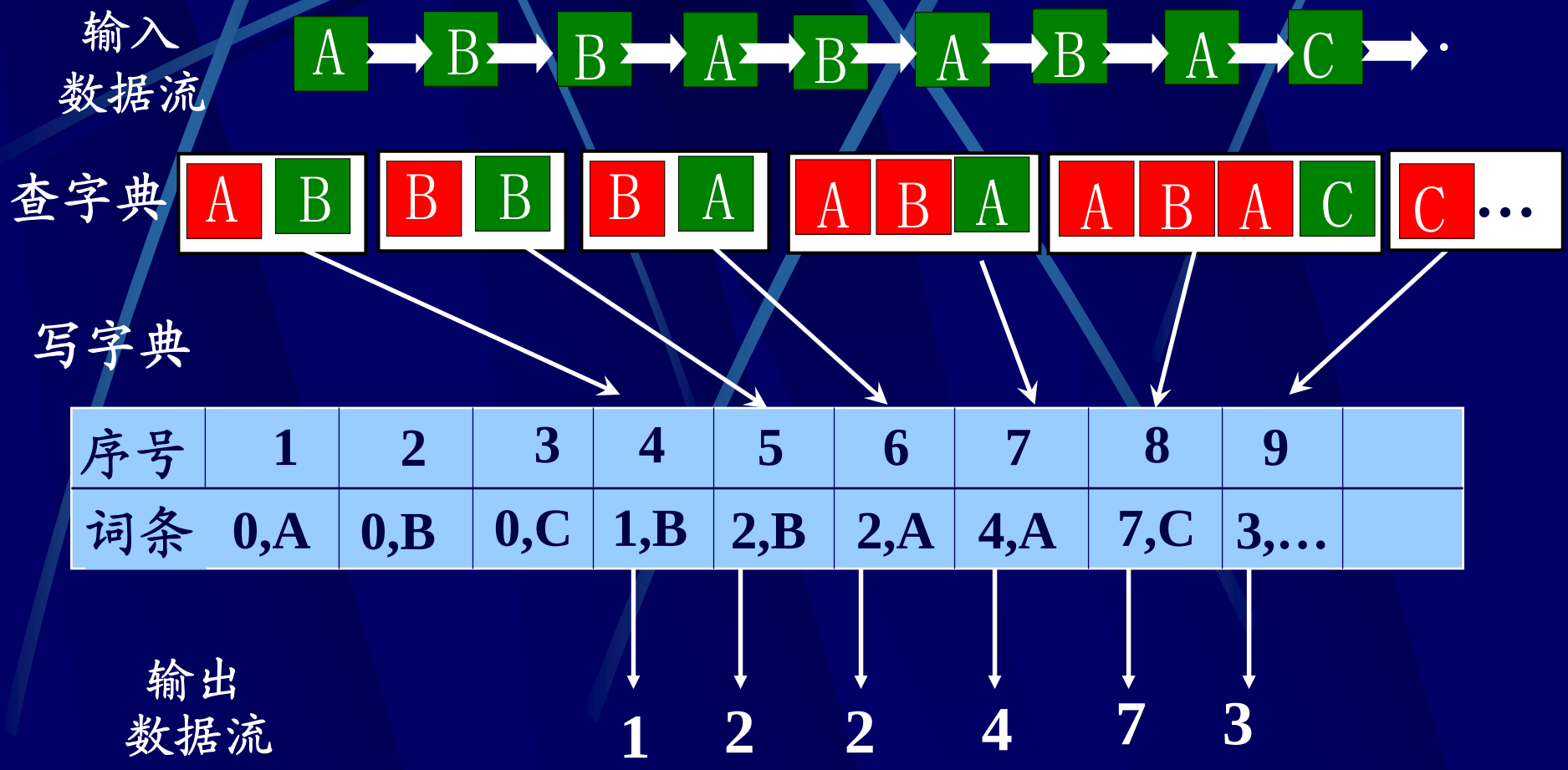
## 2. LZW编码算法：

- 1984年A.Weltch改进了LZ78，给出了实用的编码方法，被称为LZW算法。
- 改进之处在于把基本字符集（256个ASCII字符及扩展符号）预先存入词典，成为初始小词典。以后边编码边扩充。
- 由于有了初始小词典，所以输出代码流中就不必包含字符，可以完全用词条序号表示。这样一来，不仅代码更短，而且保密性更好。
- 另一个进步是词典中的每个词条均由（$n$,$C$）两部分组成，$n$是已查到的匹配字符串的“序号”，$C$是下一个待编的字符。这样就能使所有词条具有统一的格式与大小，便于软件实现。

# 编码方法：

- 因为单个字符在初始词典中都能找到，所以查、写操作从两个字符的字符串开始。
- 设第一个待编字符为$C_P$，它在初始词典中的地址为$n_1$，第二个字符为$C_{P+1}$，就可以查($n_1$ $C_{P+1}$)；若查到它在词典中的地址是$n_2$，则再连接第三个字符$C_{P+2}$查($n_2$ $C_{P+2}$)……，依次类推。
- 一旦发现某个($n_K$ $C_{P+K}$) 在词典中已查不到，则($n_K$ $C_{P+K}$)就是应添入词典尾部的新词条。
- 每轮译码输出的永远是新添词条的序号$n_K$。
- 须注意$C_{P+K}$是下一个待编字符，虽已出现，但尚未被编码。
- 查字典、写字典与编码输出是在同一个过程中完成的。

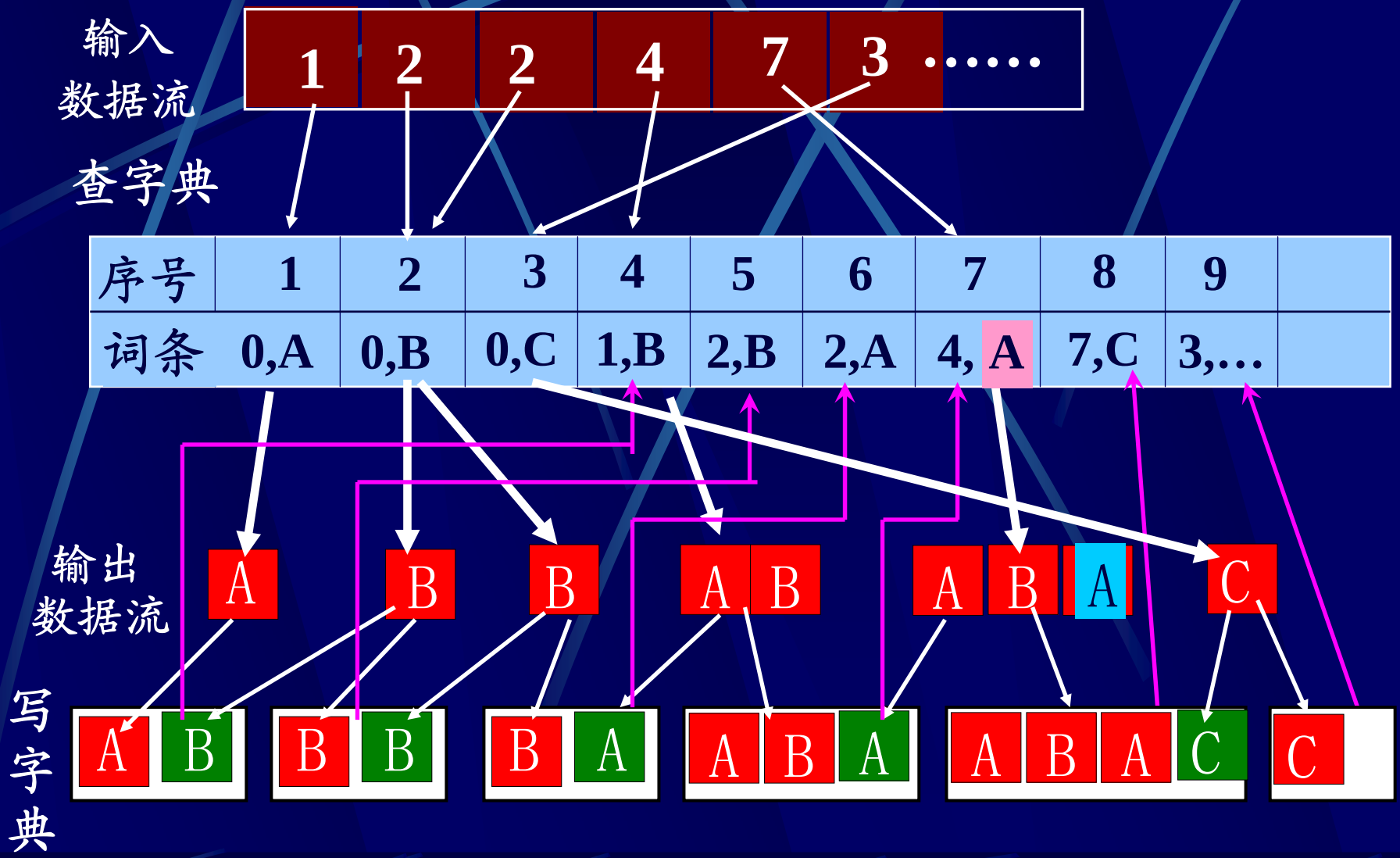
# LZW编码算法举例：


输入
数据流
A B B A B A B A C
查字典
A B
B B
B A
A B A
A B A C
C …
写字典
序号 1 2 3 4 5 6 7 8 9
词条 0,A 0,B 0,C 1,B 2,B 2,A 4,A 7,C 3,…
输出
数据流
1 2 2 4 7 3

## 3. LZW的译码算法：

- 译码很简单，每读入一个代码，就按这个地址去查词典。查到相应词条，即可译码输出。
- 译码时应将序号改写为字符串。序号是已查到的匹配字符串在字典中的地址，而这个地址中的词条又可能指示另一个地址，如此嵌套链接，直到初始字典（序号为0）为止。
- 因为每个词条均由“序号+尾字符”构成，序号是当前读入的序号，而尾字符却是下一轮才会译码输出字符串的首字符，尚属未知。然而，只要延缓一个节拍写字典，就可以用上轮译码的序号+本轮译码输出字符串的首字符，来构成的这个新词条，而这些都已经知道。
- 依次读入各个代码，并作同样处理，直至结束。

# LZW译码算法举例：


输入
数据流
1 2 2 4 7 3 ……
查字典
序号 1 2 3 4 5 6 7 8 9
词条 0,A 0,B 0,C 1,B 2,B 2,A 4, A 7,C 3,…
输出
数据流
A B B A B A B A C
写
字
典
A B
B B
B A
A B A
A B A C
C

## 唯一的例外：

- 特殊情况下，读入的序号可能在字典中查不到。这是因为这个词条还没建立，它正是当前需要添加的词条。
- 可以将次序倒过来，采用先建立词条再查询的方法进行。
- 因为新词条由“上论译码的序号*n* +本轮译码输出字符串的首字符*C*”构成。*C*似乎未知，但可以通过以下推理得到：
- 因为上轮译码输出已经得到，它（的序号）将作为新词条的前半部分被写入词典，而这个新词条又是即将输出的字符串，可见本次欲输出的字符串的首字符就是上轮译码输出的首字符。
- 结论：在这种情况下，新词条是上论译码的序号+上轮译码输出字符串的首字符。

# 2.6.3 LZW编码的软件实现

## 1. 字典的设计

- “字典”是LZW编码的核心。软件中设计字典的容量是4096个词条，其地址就是词条的序号，由0—4095。
- 所谓词条就是字典中尚未保存过的生词，实际上就是文章中新出现的字符串。
- 每个词条在字典中的结构都一样，由两部分构成，前缀是一个数字$n$，它代表当前在字典中找到的最大匹配字符串，也就是序号为$n$的词条。字符串的尾字符就是当前字符$C$。因为前缀再连接上这个字符后，就成了字典中的生词，它就是当前应当写入的新词条。

- 用结构体变量为字典开辟内存。程序为：

```
struct word {
unsigned int n;
unsigned char c;
} wd [4096];
```

- 整数n是16bit，字符c是8bit，每单元24bit；
- 4096个内存单元的地址从000H—FFFH；
- 前256个内存单元的地址从000H—0FFH，为初始小字典。

| 序号 | 前缀$n$ | 尾字符$c$ |
|---|---|---|
| 0 | 4095 | 00H |
| 1 | 4095 | 01H |
| ⋮ | 4095 | (ASCII码) |
| | 4095 | …… |
| 255 | 4095 | FFH |
| 256 | ($n_1$ | $c_1$) |
| 257 | ($n_2$, | $c_2$) |
| ⋮ | 新添加的词条…… | |
| | 新添加的词条…… | |
| 4095 | 最后一个词条 | |

## 2. 初始字典的建立

- ASCII码在各种文档里经常出现，因而作为初始值预先写入，形成初始字典。
- 256个ASCII码被放在0—255的地址上，并且其前缀统统记做4095，表示是单个字符。
- 设置初始字典的程序为：

```
for (i = 0; i<256; i ++) {
        wd[i].n = 4095;
        wd[i].c = i;      //ASCII码值等于序号数
        }
 p = 256;              //字典从地址256开始添加词条
```

## 3. 查字典与添词条

- 从欲压缩的文件中读入的第一个字符，一定是初始字典中的字(其值为n)，所以需要再输入一个字符，赋值给ch。
- 从第二个字符开始，搜索字典:先查前缀，后查尾字。序号不存在，则跳到下一个地址再找；只有序号查到了，才核对尾字符正确与否。
- 查字典的程序为：//应考虑到此段程序也将适用于以后的查询

```
for(i=n+1; i<p; i++){   //字符串的序号一定大于n
    if(wd[i].n!= n) continue ;
    if(wd[i].c == ch)  break ;
  }
```

- 如果尾字符正确，表明以ch为尾字符的字符串被找到，位于i处，用它改写前缀，返回到前面再读入字符查找。

```
if(i!= p) n = i;
```

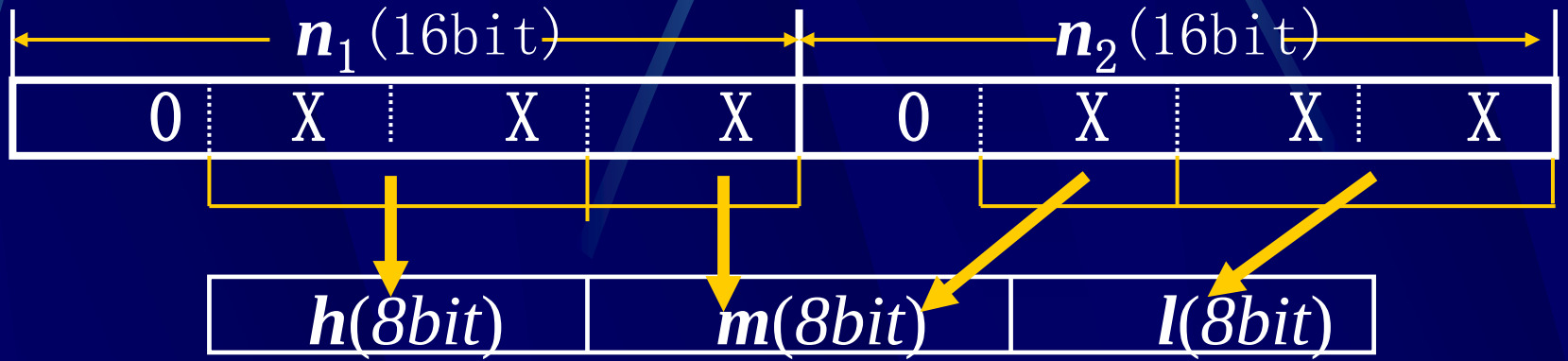
- 如果尾字符不匹配，则当前的前缀就是已找到的最长字符串，将其序号输出。同时把这个没找到的字符串补充入字典，此后恢复初始状态，以便再读字符，进行新的查找：

```
else {
  if(p<4095){
     out(n);            // 输出前缀序号
     wd[p].n = n;       // 写入一个词条
     wd[p].c = ch;
     p++;               // 再添词条应到下一个地址。
     }
   n = ch;       //当前字符作为下一轮要查找的字符串
   }             //的首字符，其码值设为首查序号;
```

# 4. 编码的输出

- 作为压缩结果而输出的数据都是一些序号，而序号$n \leqslant 4095=0x0fff$，有效数字只有12*bit*。如果用整数形式输出，则每个整数需要2个字节，即16*bit*，无疑造成浪费。

- 把第一个序号的前*8bit*有效数字赋给字符变量*h*，剩下4*bit*与下次要输出序号*n*的前4*bit*有效数字凑成一个8*bit*字符赋给*m*，最后8*bit*有效数字给字符变量*l*。这样一来，原本需要32*bit*表达的两个整数，现在就能用三个8*bit*的字符共24*bit*表示了。



- 输出编码的程序为：

```
void out(int n) {
 if(f == 0){    //如果是前一个序号。f=0表示前一个序号;
     h = n /16;// 相当于右移4位，既把n中间的两个4位，给h;
     m =(n<<4)&0xf0;//最低位的4bit左移4位，右面空出4位;
     f = 1;          //为后一序号设置标记，f=1表示后一序号;
     }               //两个序号的处理方法不同，要区别轮换进行。
else {                                  //下一个序号右移8位，并
     m = m+n/256;                       //把最高4非零值接到m后4位;
     l = n & 0xff;                     //而它的后8位非零值赋给l;
     fwrite(&h,1,1,outfp); //用二进方式写入文件中;
     fwrite(&m,1,1,outfp);
     fwrite(&l,1,1,outfp);
     h = m = l = f =0;
     }
}
```

# 2.6.4 LZW译码

## 1. 读入待解压文件数据

- 如果以字符形式读文件，则可能发生某些ASCII码的误读问题，比如把0A0D当成了回车换行。因此需要以二进制形式读入待解压文件数据。
- 每次读入三个8*bit*，赋给 ***h, m, l*** 三个变量。
- 然后拆分成两个12bit的整数，存入数组E[i]相继的两个单元中，直至读完文件的全部数据。
- E[i]就成为解压缩的数据来源。

- 读入数据的程序为：

```
i=0;
 while(!feof(fp)) {
       fread(&h,1,1,fp);
       fread(&m,1,1,fp);
       fread(&l,1,1,fp);

       E[i]=(h*16)+((m&0xf0)/16);
       E[i+1]=((m&0x0f)*256)+ l;

       h=m=l=0;
       i=i+2;
       }
 k=i-2;                    //给出数组E[i]的长度。
```

## 2. 建立并恢复字典

- 建立初始字典与LZW编码程序段相同，不再赘述。
- 从序号256开始添加新词条。
- 上次读入的序号n就是当前词条的前缀。
- 设c是当前读入待译的序号，p是当前应当添加的词典地址，则在c<p(词典中能找到)的情况下,当前词条的尾字符是本次读入序号的这个字符串的首字符；在c=p（要查找的词条恰好是当前欲建立的词条）的情况下，这个尾字符是上次读入序号的那个字符串的首字符。
- 两种情况都应当按照当前字典上已有的词条的链表来追寻这个首字符。除非它是初始字典中的单字符。

# 恢复字典的程序段如下：

```
p=256;                          //新词条从序号256开始
n=E[0];                         //读入的第一个序号
for(i=1;i<k;i++)   {            // k是序号总的个数
    wd[p].n = n;                //当前词条序号总归是上轮读入的序号
    c=E[i];                     //本轮读入的序号赋给变量c
    if(c= =p) c= n;             //若是c=p，则找上轮序号的首字符
    for(;;){                    //追朔首字符的过程用循环体来处理
       if(c>255) {              //一直追到初始词典为止
          c=wd[c].n;
          continue;
          }
       break;                   //找到首字符便跳出循环
       }
    wd[p].c= c;                 //将此首字符赋给当前词条的尾字符
    n=E[i];                     //将本轮的序号赋给变量n以便下轮使用
    p++;                        //新词条添入地址将移到下一个位置
    }
```

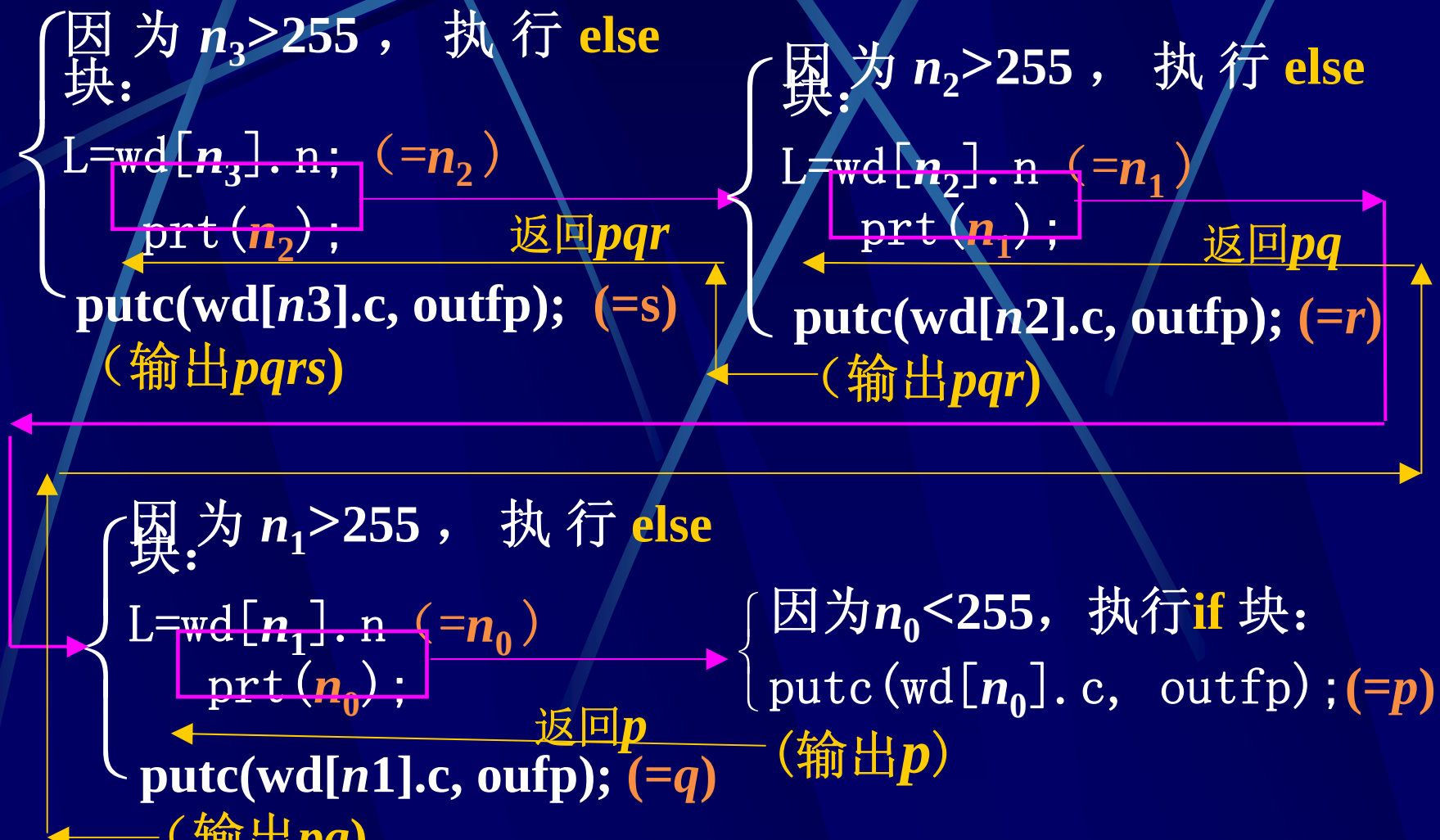
## 3. 解压输出

- 按照E[i]给出的序号，依次一个一个地解压缩。每个序号调用一次解压缩函数prt( )。
- 255以内的序号是单个字符，可直接输出，其序号就是ASCII码值；
- 255以上的序号是字符串，应当按照字典词条所指示的链表来追朔。而追踪的过程是一个递归调用的过程。

- 解压缩函数prt( )的程序段如下：

```
void prt(int M) {
int L;
if(M<256) {
   printf("%c",wd[M].c);
   putc(wd[M].c, outfp);
   }
else  {
   L=wd[M].n;
   prt(L);
   printf("%c",wd[M].c);
   putc(wd[M].c, outfp);
   }
}
```

设：词典中地址$n_3$的词条是$(n_2,s)$，$n_2$的词条是$(n_1,r)$，$n_1$的词条是$(n_0, q)$，$n_0$的词条是$(4095, p)$。递归调用过程是：



# 2.6.4 LZW编、译码的演示

1. 被压缩的原文件：test.txt(3.47KB)
2. LZW压缩软件：LZW1.EXE
3. 压缩后的文件：OUTF.TXT(2.07KB，小40%)
4. 解压缩软件：LZW2.EXE
5. 释放后的文件：REFILE.TXT=test.txt
6. 效果：　REFILE.TXT≦test.txt

# 小结：

❖**L-Z编码的原理：**

从实际文件出发，凡出现过的都记下来，以后再出现时就用序号来代替。

❖**字典编码的算法：**

字典的设计、建立、添加与整理；

边查字典边输出，边搜寻新词边补充；

❖**字典编码的实质-----边统计边编码**

# 课后复习题

- 思考题：

  字典编码没有考虑符号概率，是否违背Shannon关于概率信息的基本原理？

- 作业题：

  教材第63页习题二第15、16题；

- 实践题：编写LZW压缩程序。