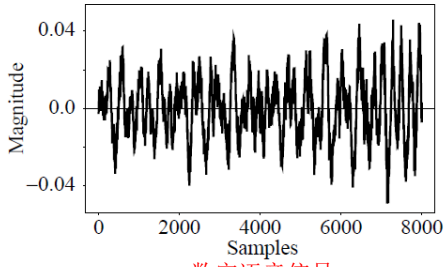
# 回 顾

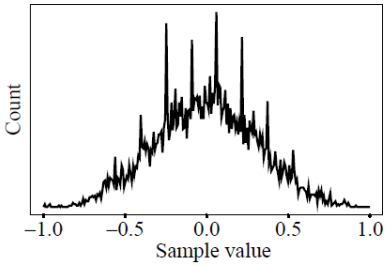
- **语音信号编码**
- 1、话音编码概述
- 2、波形编译码器
  - PCM、DM、ADM、ADPCM的概念
- 3、混合编译码器
  - LPC、CELP的概念
- 4、话音编码标准
  - G.721、G.722、 G.723 、G.726



# 音频数据相关性特征




数字语音信号

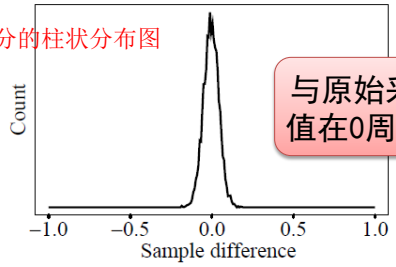



语音信号的柱状分布图

信号差分的柱状分布图

$$\widehat{f_n} = f_{n-1}$$

$$e_n = f_n - \widehat{f_n}$$




与原始采样值相比，差分值在0周围的聚集度更高。

## 无损预测编码

- 假设预测公式如下：

$$\widehat{f_n} = \lfloor \frac{1}{2}(f_{n-1} + f_{n-2}) \rfloor$$

$$e_n = f_n - \widehat{f_n}$$

实际传输的是误差值$e_n$



## DPCM

- 预测器：

$$\hat{f}_n = \text{trunc}\left(\left(\tilde{f}_{n-1} + \tilde{f}_{n-2}\right)/2\right)$$

- 量化器：

$$\tilde{e}_n = Q[e_n] = 16 * \text{trunc}\left((255 + e_n)/16\right) - 256 + 8$$

$$\tilde{f}_n = \hat{f}_n + \tilde{e}_n$$

量化器把误差值平均划分为32块，每一块在原始信号中跨越16个取值范围

# 无损压缩算法

1 数据压缩概述
2 信息论基础
3 RLC编码
4 霍夫曼（Huffman）
5 算术编码
6 词典编码

| | |
|---|---|
| 林福宗： | c4： 无损数据压缩 |
| Ze-Nian Li： | c7.1~c7.6： 无损压缩算法 |



# 1 数据压缩概述

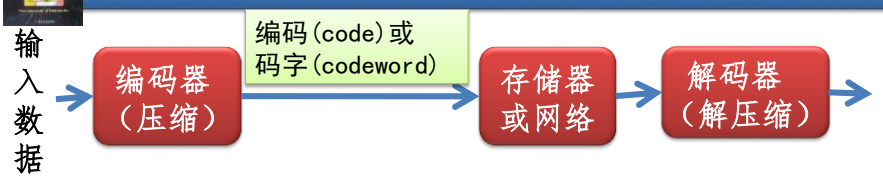
- 采样数据不仅是所代表的原始信息本身，还包含着其它一些没必要保留的（确定的、可推知的）信息，即存在着数据冗余。**M = D - △d**
  - **M**：实际媒体数据（信息）
  - **D**：数字化后的采样数据
  - **△d**：数据冗余量
- 数据压缩：从采样数据中去除冗余，即保留原始信号中变化的、特征性信息，去除重复的、确定的或可推知的信息，在实现更接近实际媒体信息描述的前提下，尽可能地减少描述用的信息量。

# 1 数据压缩概述

- 数据压缩的目的：以减少存储、处理和传输的成本。
- 压缩是信源信号（采样和量化后数字信号），如语音、静止图像、音乐或电视等的有效的数字化表示。
- 多媒体数据的冗余性，如：空间冗余、时间冗余、信息熵冗余(编码冗余)、结构冗余、知识冗余、认知（视觉、听觉）冗余、其他冗余。



# 举例说明




- 对于一个没有中文的文件内存储内容为“ABC”
- ASCII码最高位是0

| | A | B | C | |
|---|---|---|---|---|
| 压缩前： | 01000001 | 01000010 | 01000011 | B0=24位 |
| 压缩后： | 1000001 | 1000010 | 1000011 | B1=21位 |

- 解压缩：最高位加0补齐8位

压缩率=B0/B1=24/21=1.14

## 分2类：无损压缩和有损压缩

- **无损压缩**：使用压缩后的数据进行重构(或者叫做还原，解压缩)，重构后的数据与原来的数据完全相同
  - 压缩比通常2~4，算法有霍夫曼、算术编码、LZW等
- **有损压缩**：使用压缩后的数据进行重构，重构后的数据与原来的数据有所不同，但不影响人对原始资料表达的信息的理解，不造成误解。
  - 压缩比更大，可达30:1，甚至100:1
  - 如变换编码



## 2、信息论基础

- 一个具有符号集$S=\{s_1,s_2,\ldots,s_n\}$的**信息源的熵**定义为：

$$\eta = H(S) = \sum_{i=1}^{n} p_i \log_2 \frac{1}{p_i}$$

- $p_i$ ——$S$中$s_i$出现的概率
- $\log_2 \frac{1}{p_i}$——包含在$s_i$中的信息量（自信息量），与对$s_i$编码所需的位数相等。
- **熵：从数据流中识别出经常出现的符号作为压缩流中的候选短码字。**

# 例1 文字

信源: multimedia

| | |
|---|---|
| $S=\{s_1,s_2,\ldots,s_8\}=$ | { a , d , e , i , l , m , t , u } |
| $f=\{f_1,f_2,\ldots,f_8\}=$ | { 1 , 1 , 1 , 2 , 1 , 2 , 1 , 1 } |
| $p=\{p_1,p_2,\ldots,p_8\}=$ | {1/10,1/10,1/10,2/10,1/10,2/10,1/10,1/10} |
| $I=\{I_1,I_2,\ldots I_8\}=$ | {3.3,3.3,3.3,2.3,3.3,2.3,3.3,3.3} |

如果采用相同位传送：8种状态，首先想到的是3位传送

$$\eta = H(S) = \sum_{i=1}^{8} p_i \log_2 \frac{1}{p_i} = 2.9$$

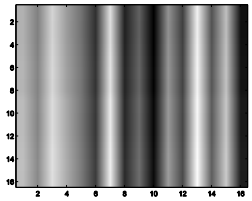
熵：在某种条件下能取得的最小码字平均长度

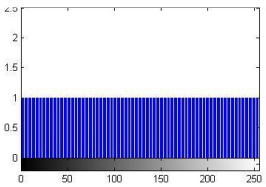




# 例2 图像直方图

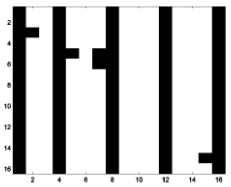
- 一副灰度均匀分布的图：
- $p_i$=1/256

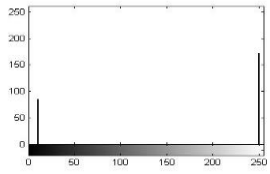




$$\eta = H(S) = \sum_{i=1}^{255} p_i \log_2 \frac{1}{p_i} = 8$$

- 一副只有两个灰度的图：
- $p_1$ =1/3； $p_2$ =2/3





$$\eta = \frac{1}{3}\log_2 3 + \frac{2}{3}\log_2 \frac{3}{2} = 0.92$$

# 无损压缩算法





# 3、RLC(Run-length Coding)编码

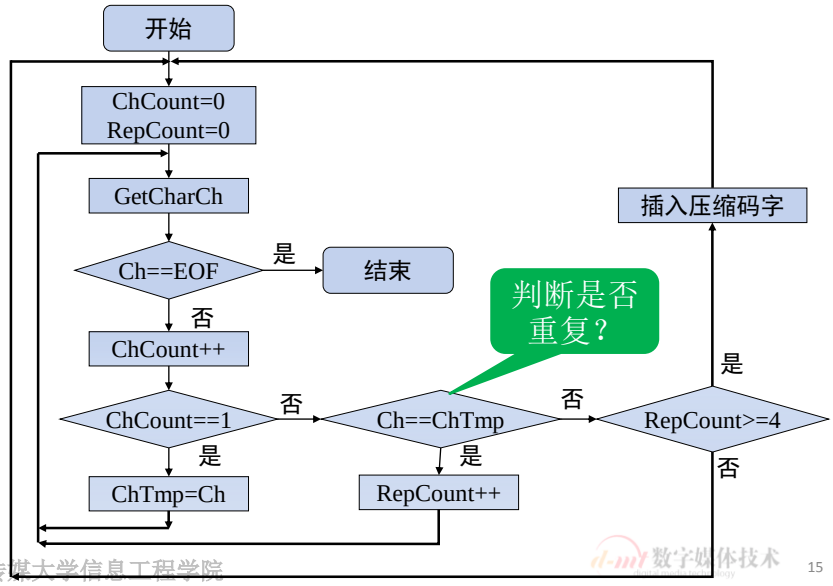
- 利用了信源中的记忆
- 基于简单的编码数据原则：重复的数据值序列（或称为“流”）用一个重复次数和单个数据值来代替。这里，重复的值称为一个“顺串” 或“连续”（run）。

| 控制符 | 重复次数 | 被重复字符 |
|---|---|---|

三字节码字格式

示例：RTAAAASDEEEEE 经RLE压缩后为：RT*4ASD*5E

# RLC编码流程图






# RLC算法应用



- RLC算法是BMP、PCX、TIFF等图像压缩技术的一部分，在PDF文件格式中也得到应用。

# 4 霍（赫）夫曼编码

- 霍夫曼（Huffman）在1952年提出的一种编码方法，是一种熵编码。
- 从下到上的编码方法，属于变长码类。
- 霍夫曼编码可区别的不同码字的生成是基于不同符号出现的不同概率。
- 自含**同步码**，在编码之后的码串中都不需要另外添加标记符号，即在译码时切分符号的特殊代码。
- 基于一种称为“编码树”（coding tree）的技术。
- 得到广泛应用
  - **传真机、JPEG、MPEG**



# 霍夫曼编码算法步骤

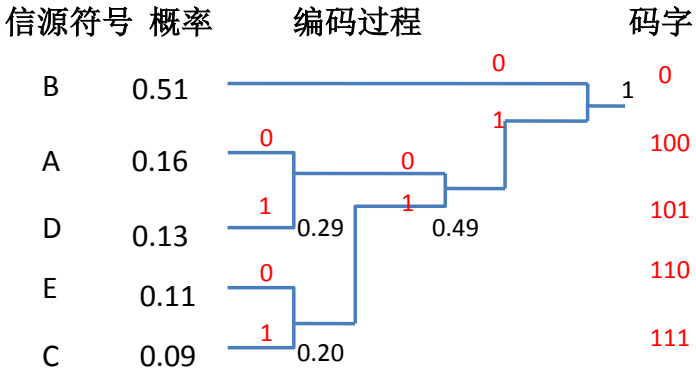
（1）初始化，根据符号**概率**的大小按由大到小顺序对符号进行**排序**。

（2）把**概率最小**的两个符号组成一个新符号（节点），即新符号的概率等于这两个符号概率之和。

（3）重复第2步，直到形成一个符号为止（树），其概率最后等于1。

（4）从编码树的根开始回溯到原始的符号，并将每一下分枝赋值为1，上分枝赋值为0。

# 霍夫曼编码例






# 霍夫曼编码实例

- 【例】有一幅39个像素组成的灰度图像，灰度共有5级，分别用符号A、B、C、D和E表示，39个像素中出现灰度A的像素数有15个，出现灰度B的像素数有7个，出现灰度C的像素数有7个等等，如表所示。如果用3个位表示5个等级的灰度值，也就是每个像素用3位表示，编码这幅图像总共需要117位。

符号在图像中出现的数目

| 符 号 | A | B | C | D | E |
|---|---|---|---|---|---|
| 出现的次数 | 15 | 7 | 6 | 6 | 5 |

- 按照香农理论，图像的熵为

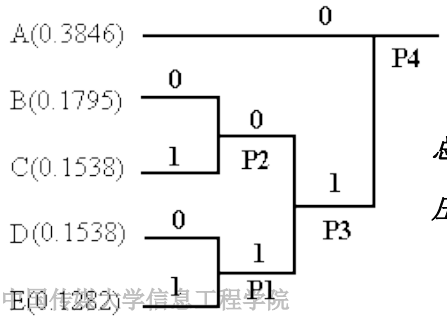
$$H(S) = (15/39)\log_2(39/15) + (7/39)\log_2(39/7) + \ldots + (5/39)\log_2(39/5) = 2.1859$$

压缩比= 3/2.1859 = 1.3724:1

# 霍夫曼编码实例

| 符号 | 出现的次数 | Log2（1/pi） |
|---|---|---|
| A | 15(0.3846) | 1.38 |
| B | 7(0.1795) | 2.48 |
| C | 6(0.1538) | 2.70 |
| D | 6(0.1538) | 2.70 |
| E | 5(0.1282) | 2.96 |

| 符号 | 分配的代码 | 需要的位数 |
|---|---|---|
| A | 0 | 15 |
| B | 100 | 21 |
| C | 101 | 18 |
| D | 110 | 18 |
| E | 111 | 15 |




*总长度：15*1+7*3+6*3+6*3+5*3=87bit*

*压缩比：39*3/87=1.3448*



# 霍夫曼编码优缺点

| 符号 | 分配的代码 |
|---|---|
| A | 0 |
| B | 100 |
| C | 101 |
| D | 110 |
| E | 111 |

- 1、唯一前缀性：任意一个huffman编码都不能作为另一个编码的前缀
  - 码长虽然可变，但不需要附加的同步代码；
  - 如果事先编写出一本解释各种代码意义的“词典”（码簿），就可以根据词典一个码一个码地依次进行译码。
- 2、最优性：一种**最小冗余编码**(对已知任何数据分布模型都是最优的)
  - 两个频率最低的字符具有相同长度的码字，但最后一位不同。
  - 出现频率较高的符号的码字短，出现频率较低的符号的码字长。
  - 信息源S的平均编码长度： $\eta \le \bar{l} < \eta + 1$

# 霍夫曼编码优缺点

- 3、错误传播(error propagation)
  - 霍夫曼码没有错误保护功能
    - 译码，若码串中无错误，能一个接一个地正确译码
    - 若码串中有错误，哪仅是1位出现错误，不但这个码本身译错，更糟糕的是一错一大串。
- 4、霍夫曼码是可变长度码，因此很难随意查找或调用压缩文件中间的内容，然后再译码。
- 5、 构造出的码不唯一。
- 6、对不同的信源编码效率不同。



# 霍夫曼编码算法的改进

- 1、扩展的霍夫曼编码（Extended Huffman code）
  - 当一个符号$s_i$的概率接近1时，log2（$1/p_i$）接近0，这时用1位来表示这个符号是相当昂贵的。
  - 当所有符号的概率都可以表示成$2^{-k}$时，码字的平均长度才能达到最优，$\overline{l} = \eta$
  - 解决：编码符号组而不是单个符号

$$\overline{l} < \eta + \frac{1}{k}$$

# 改进的霍夫曼编码算法



- 2、自适应霍夫曼编码（Adaptive Huffman code）
  - 信源的存储与传输过程中必须首先存储与传输编码表。但有关信源的先验统计知识难于获得。
  - 根据符号概率的变化动态地更新信源的先验统计，并动态地改变码字。
  - 产生的代码比原始霍夫曼编码更有效。



# 无损压缩算法

1 数据压缩概述
2 信息论基础
3 RLC编码
4 霍夫曼（Huffman）
**5 算术编码**
6 词典编码

# 5 算术编码

- 算术编码（Arithmetic Coding）是一种更现代的编码方法，在实际中比霍夫曼编码更有效。
- 最初在20世纪七八十年代提出。
- 各种更现代的算术编码被提出，应用于新的多媒体标准中。如JPEG2000中的快速二进制算术编码，H.264和H.265中的CABAC（Context-Adaptive Binary Arithmetic Coding）



# 5 算术编码

- 回忆：扩展Huffman编码符号组而不是单个符号

$$\overline{l} < \eta + \frac{1}{k}$$

  - 导致编码和解码器所需的结果符号表会非常大
- 算术编码的基本原理：
  - 将编码的消息表示成实数0和1之间的一个间隔（Interval），消息越长，编码表示它的间隔就越小，表示这一间隔所需的二进制位就越多。
- 两个基本的参数：
  - 符号的概率和它的编码间隔。

# 算术编码步骤

29

① 编码器在开始时将“当前间隔”[L,H)设置为[0,1)。

② 对每一事件，编码器按步骤（a）和（b）进行处理。

a. 编码器将“当前间隔”分为子间隔，每个事件一个。

b. 一个子间隔的大小与下一个将出现的事件的概率成比例，编码器选择子间隔应与下一个确切发生的事件相对应，并使它成为新的“当前间隔”。

③ 最后输出的“当前间隔”的**下边界**就是该给定事件序列的算术编码。





# 算术编码过程举例

**[例] 假设信源符号为{00,01,10,11}，这些符号的概率分别为{ 0.1,0.4,0.2,0.3 }，根据这些概率可把间隔[0,1)分成4个子间隔：[0,0.1), [0.1,0.5), [0.5,0.7), [0.7,1)，其中[x,y)表示半开放间隔，即包含x不包含y。**

信源符号、概率和初始编码间隔

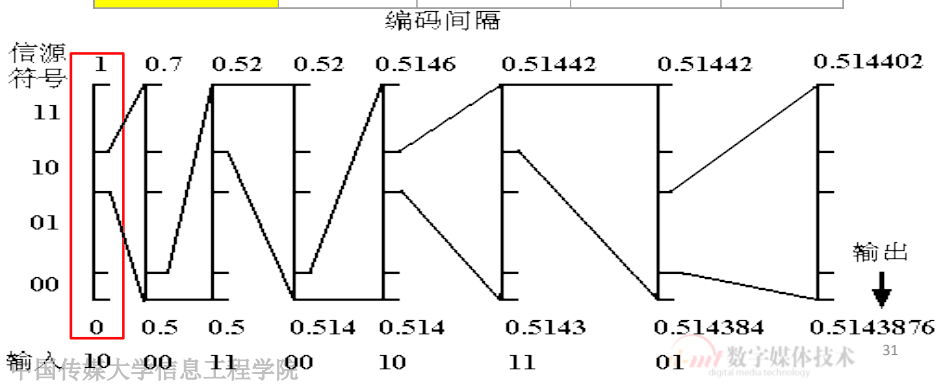
| 符号 | A/00 | B/01 | C/10 | D/11 |
|---|---|---|---|---|
| 概率 | 0.1 | 0.4 | 0.2 | 0.3 |
| 初始编码间隔 | [0，0.1） | [0.1，0.5） | [0.5，0.7） | [0.7，1） |

30

# 算术编码过程举例

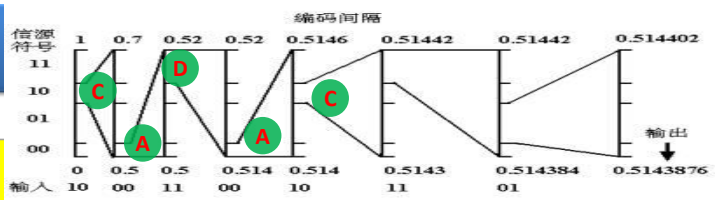
如果二进制消息序列的输入为：**10 00 11 00 10 11 01**。

| 符号 | A/00 | B/01 | C/10 | D /11 |
|---|---|---|---|---|
| 概率 | 0.1 | 0.4 | 0.2 | 0.3 |
| 初始编码间隔 | [0，0.1） | [0.1，0.5） | [0.5，0.7） | [0.7，1） |










| 步骤 | 输入符号 | 编码间隔 | 编码判决 |
|---|---|---|---|
| 1 | C | [0.5，0.7) | 符号的间隔范围[0.5，0.7) |
| 2 | A | [0.5，0.52) | [0.5，0.7)间隔的第一个1/10 |
| 3 | D | [0.514，0.52) | [0.5，0.52)间隔的最后3个1/10 |
| 4 | A | [0.514，0.5146) | [0.514，0.52)间隔的第一个1/10 |
| 5 | C | [0.5143，0.51442) | [0.514，0.5146)间隔的第五个1/10开始，二个1/10 |
| 6 | D | [0.514384，0.51442) | [0.5143，0.51442)间隔的最后3个1/10 |
| 7 | B | [0.5143836，0.514402) | [0.514384，0.51442)间隔的4个1/10，从第1个1/10开始 |
| 8 | 从[0.5143876，0.514402]中选择一个数作为输出：0.5143876 | | |

# 译码过程

| 步骤 | 间隔 | 译码符号 | 译码判决 |
|---|---|---|---|
| 1 | [0.5，0.7) | C | 0.51439在间隔 [0.5，0.7) |
| 2 | [0.5，0.52) | A | 0.51439在间隔 [0.5，0.7)的第1个1/10 |
| 3 | [0.514，0.52) | D | 0.51439在间隔[0.5，0.52)的第7个1/10 |
| 4 | [0.514，0.5146) | A | 0.51439在间隔[0.514，0.52)的第1个1/10 |
| 5 | [0.5143，0.51442) | C | 0.51439在间隔[0.514，0.5146)的第5个1/10 |
| 6 | [0.514384，0.51442) | D | 0.51439在间隔[0.5143，0.51442)的第7个1/10 |
| 7 | [0.51439，0.5143948) | B | 0.51439在间隔[0.51439，0.5143948)的第1个1/10 |
| 8 | | | 译码出来的消息：C A D A C D B |



# 算术编码需要注意的问题



- 计算中有几个问题需要注意：
  - 溢出：由于实际的计算机的精度不可能无限长，运算中容易出现溢出。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。
  - 算术编码器对整个消息只产生一个码字，这个码字是在间隔[0，1)中的一个实数，因此译码器在接受到表示这个实数的所有位之前不能进行译码。
  - 对错误很敏感：如果有一位发生错误就会导致整个消息译错。

# [例]算术编译码

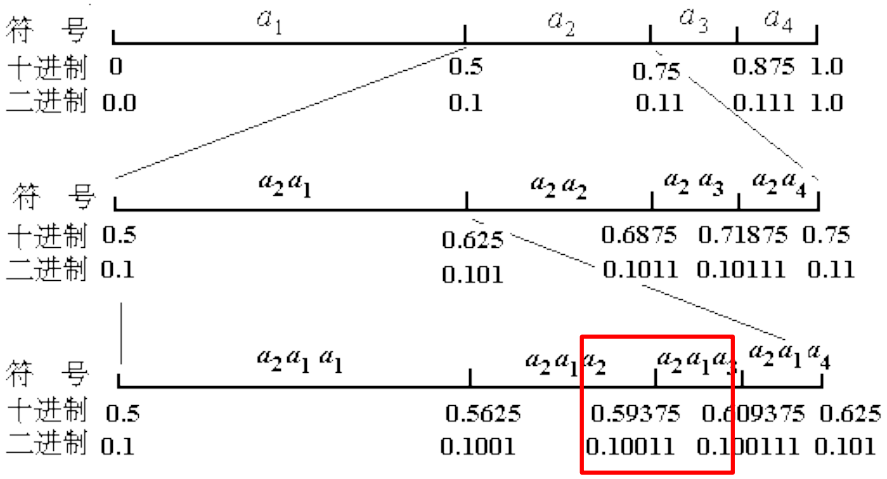
- 假设有4个符号的信源，它们的概率如表：

| 信源符号$a_i$ | $a_1$ | $a_2$ | $a_3$ | $a_4$ |
|---|---|---|---|---|
| 概率$P_i$ | $P_1=0.5$ | $P_2=0.25$ | $P_3=0.125$ | $P_4=0.125$ |
| 初始编码间隔 | [0, 0.5) | [0.5, 0.75) | [0.75, 0.875) | [0.875, 1) |

- 假设输入序列为$X_n$：$a_2,a_1,a_3,\ldots$。它的编码和译码过程。



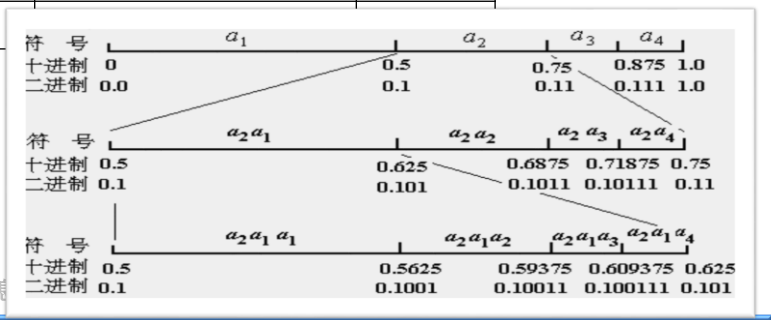
# 输入为$a_2,a_1,a_3,\cdots$




**编码输出的符号是：10011…**

# 10011…译码

| 接收的数字 | 间隔 | 译码输出 |
|---|---|---|
| 1 | [0.5, 1) | - |
| 0 | [0.5, 0.75) | $a_2$ |
| 0 | [0.5, 0.625) | $a_1$ |
| 1 | [0.5625, 0.625) | -- |
| 1 | [0.59375, 0.625) | $a_3$ |
| … | | |






# 无损压缩算法

1 数据压缩概述
2 信息论基础
3 RLC编码
4 霍夫曼（Huffman）
5 算术编码
**6 词典编码**

# 6 词典编码



- 第一类词典法
  - 查找正在压缩的字符序列是否在前面的输入数据中出现过，如果是，则用指向早期出现过的字符串的“指针”替代重复的字符串。
- 第二类算法
  - 从输入的数据中创建一个“短语词典(dictionary of the phrases)”。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。

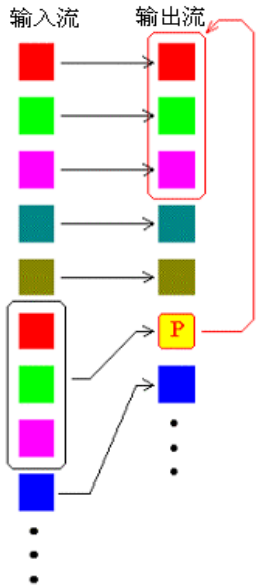




# 第一类词典法编码概念

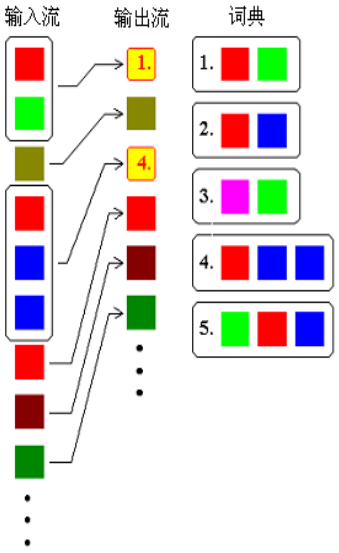


- 基本思想：查找正在压缩的字符序列是否在前面的输入数据中出现过，如果是，则用指向早期出现过的字符串的“指针”替代重复的字符串。
- “词典”是隐含的，指用以前处理过的数据。
- 以Abraham Lempel和Jakob Ziv在1977年开发和发表的算法（称为LZ77算法）为基础。
- 改进算法是由Storer和Szymanski在1982年开发的，称为LZSS算法。



# 第二类词典法编码概念

- 基本思想：从输入的数据中创建一个“短语词典(dictionary of the phrases)”。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。
- A.Lempel和J.Ziv在1978年首次发表了介绍这种编码方法的文章，称为LZ78。
- Terry A.Welch在1984年改进了这种算法。
- 称为LZW(Lempel-Ziv Walch)压缩编码。






# LZW（Lempel-Ziv-Welch）

- 一种自适应的、基于字典的压缩算法。
- 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，LZW不断将越来越长的重复条目插入字典中，然后将字符串的编码而不是字符串本身发送出去。
- LZW编码器和解码器会在接受数据时动态地创建字典，编码器和解码器也会产生相同的字典。

# LZW算法

```
BEGIN
  s=next input character;
  while not EOF
    {c=next input character;
      if s+c exists in the dictionary
         s=s+c;
      else
        {output the code for s;
          add string s+c to the dictionary with a new code;
          s=c;
        }
    }
  output the code for s;
END
```



## 例：对字符串ABBABABAC进行LZW压缩

| 步骤 | 前缀s | 输入c | 输出code | 词典code+string | |
|---|---|---|---|---|---|
| | | | | (1) | A |
| | | | | (2) | B |
| | | | | (3) | C |
| | NULL | A | \ | \ | \ |
| 1 | A | B | (1) | (4) | AB |
| 2 | B | B | (2) | (5) | BB |
| 3 | B | A | (2) | (6) | BA |
| 4 | A | B | \ | \ | \ |
| 6 | AB | A | (4) | (7) | ABA |
| 6 | A | B | \ | \ | \ |
| 7 | AB | A | \ | \ | \ |
| 8 | ABA | C | (7) | (8) | ABAC |
| 9 | C | EOF | (3) | | |

## 例：对字符串ABBABABAC进行LZW压缩

| 步骤 | 前缀s | 输入c | 输出code | 词典code+string |
|---|---|---|---|---|
| 6 | AB | A | (4) | (7) ABA |
| 9 | C | EOF | (3) | |

输出：1 2 2 4 7 3

假设字典每个码字编码用4bit表示，
压缩比：9字节/6*4bit = 3

实现压缩的原因：字符串中具有很多冗余

文本长度很长时才能体现该算法的优越性

## LZW解码算法

```
BEGIN
  s=NIL;
  while not EOF
    {k=next input code;
     entry=dictionary entry for k;
      if (entry==NULL)
          entry=s+s[0];
      output=entry;
      if(s!=NIL)
         add string s+entry(0) to dictionary with a new
  code;
      s=entry;
      }
END
```

## 对码字流(Codestream)解码
1 2 2 4 7 3

| 步骤 | 前缀s | 输入k | Entry/output | 词典code+string | |
|---|---|---|---|---|---|
| | | | | (1) | A |
| | | | | (2) | B |
| | | | | (3) | C |
| 1 | NIL | 1 | A | \ | \ |
| 2 | A | 2 | B | (4) | AB |
| 3 | B | 2 | B | (5) | BB |
| 4 | B | 4 | AB | (6) | BA |
| 6 | AB | 7 | ABA | (7) | ABA |
| 6 | ABA | 3 | C | (8) | ABAC |
| 7 | | EOF | | | |

- 解码后：A B B AB ABA C



## LZW编码特点

- 1、如果待压缩的数据没有任何可重复的结构，那么使用字典条目中新编码的可能性就很小。这时，会导致数据膨胀（data expansion），而非数据压缩。
  - 建立两种状态：压缩和透明的。当检测到数据膨胀时，压缩停止，后一种状态被激活。
- 2、字典大小的限制，会导致条目过大时失去自适应能力，转为静态的、基于字典的技术。
  - 移除一些最近使用较少（LRU）的条目。
- 应用：UNIX compress; GIF图像…