《多媒体系统导论》期末速通教程

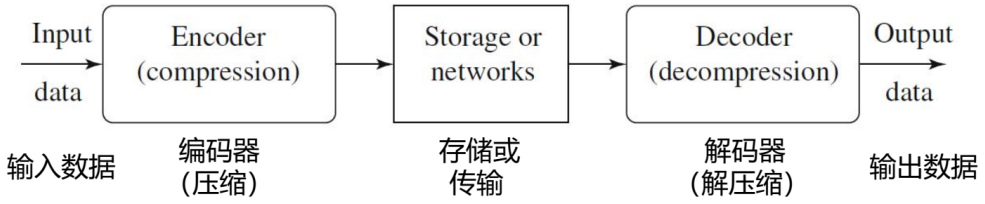
3. 图像的压缩

3.1 信息论基础

[压缩]

(1) 定义: 有效地减少表示某种信息所需的比特数的编码过程.

(2) 通用的压缩方案:



(3) 图像常用的无损压缩和有损压缩:

$$
\text{图像的压缩}\begin{cases}\text{无损压缩}\begin{cases}\text{游程编码}\\ \text{变长编码}\begin{cases}\text{Shannon-Fano 算法}\\ \text{Huffman 编码}\end{cases}\\ \text{LZW 压缩}\\ \text{算术编码}\\ \text{无损 JPEG}\end{cases}\\ \text{有损压缩：JPEG}\end{cases}
$$

[压缩率]

定义: 设压缩前某数据所需的总位数为 B_0 , 压缩后所需的总位数为 B_1 , 则 压缩率 $=\dfrac{B_0}{B_1}$.

[注] 在失真容忍范围内, 压缩率越高越好.

[熵]

(1) 定义: 有符号集 $S=\{s_1,\cdots,s_n\}$ 的信号源的熵 $\eta=H(S)=\sum\limits_{i=1}^{n}p_i\cdot\log_2\dfrac{1}{p_i}=-\sum\limits_{i=1}^{n}p_i\cdot\log_2 p_i$, 其中 $p_i\ \ (1\le i\le n)$ 为 S 中符号 s_i 出现的概率.

(2) $\log_2\dfrac{1}{p_i}$ 称为 s_i 在 S 中的**自信息量**, 表示编码 s_i 所需的最小位数.

① 高频出现的符号自信息量小, 分配短码字.

② 低频出现的符号自信息量大, 分配长码字.

(3) 熵度量系统的无序性, 熵越大, 系统越无序.

(4) 熵反映 S 的平均信息量.

(5) 熵表示对 S 中每个符号编码所需的平均位数的下界, 即 $\eta\le\overline{l}$, 其中 $\overline{l}$ 是编码器码字的平均长度.

理想的编码方法应尽量接近下界.

[**例**] 某电报有 4 种字符 $S=\{A,B,C,D\}$, 每种字符出现的概率分别为 $\frac{1}{2}$、$\frac{1}{6}$、$\frac{1}{6}$、$\frac{1}{6}$. 采用熵编码时,求编码该电报所需的最小位数.

[**解**] 编码 A 需 $-\log_2\frac{1}{2}=1$ 位, 编码 B、C、D 各需 $-\log_2\frac{1}{6}\approx 2.5850$ 位.

编码该电报需 $H(S)=\frac{1}{2}\times 1+3\times\frac{1}{6}\times 2.5850\approx 1.7924$ 位.

3.2 游程编码

[**游程编码**]

例: 将 "dfffffeeeeettttrrrrttttt" 编码为 "d1f5e5t4r4t5" .

3.3 变长编码

3.3.1 Shannon-Fano 算法

[**Shannon-Fano 算法**]

(1) 自顶向下.

(2) 步骤:

① 将符号按出现频数非升序排列.

② 迭代地将符号分为两部分, 两部分的符号频数之和相近, 直至所有部分都只包含一个符号.

③ 按根节点到叶子节点的路径分配码字.

(3) 结果可能不唯一, 不同结果的编码长度可能不同.

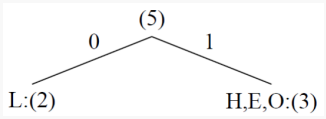
(4) 是前缀编码, 即一个符号的码字不可能是另一符号的码字的前缀.

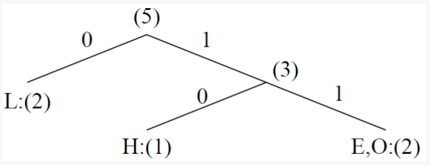
[**例**] 用 Shannon-Fano 算法编码串 "HELLO" .

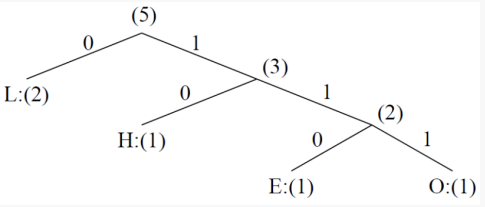
[**解 1**] 符号表:

符号	H	E	L	O
出现次数	1	1	2	1

Shannon-Fano 算法:







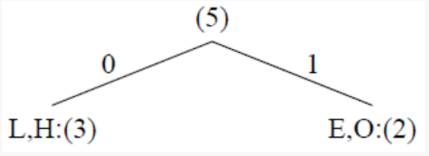
符号	出现次数	$\log_2 \frac{1}{p_i}$	码字	码字位数
L	2	1.32	0	$2 \times 1 = 2$
H	1	2.32	10	2
E	1	2.32	110	3
O	1	2.32	111	3

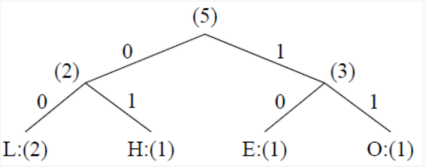
$$\text{总位数} = 2+2+3+3 = 10\,,\ \text{平均位数} = \frac{10}{5} = 2\,.$$

$$\text{熵}\ \eta = p_L \cdot \log_2 \frac{1}{p_L} + p_H \cdot \log_2 \frac{1}{p_H} + p_E \cdot \log_2 \frac{1}{p_E} + p_O \cdot \log_2 \frac{1}{p_O}$$

$$= 0.4 \times 1.32 + 3 \times 0.2 \times 2.32 \approx 1.92\,,\ \text{即 Shannon-Fano 算法的平均位数接近下界.}$$

[**解 2**]





符号	出现次数	$\log_2 \frac{1}{p_i}$	码字	码字位数
L	2	1.32	00	$2 \times 2 = 4$
H	1	2.32	01	2
E	1	2.32	10	2
O	1	2.32	11	2

$$\text{总位数} = 4+2+2+2 = 10\,,\ \text{平均位数} = \frac{10}{5} = 2\,.$$

3.3.2 Huffman 编码

[**Huffman 编码**]

(1) 自底向上.

(2) 步骤:

① 将符号按出现频数非降序排列.

② 每次取出现频数最小的两符号, 将频数之和作为父节点. 重复该过程直至所有符号都连接.

③ 按根节点到叶子节点的路径分配码字.

(3) 性质:

① 是前缀编码, 即一个符号的码字不可能是另一符号的码字的前缀.

② 是最优性-最小冗余编码:

(i) 两出现频数最低的字符的码长相同, 码字最后一位不同.

(ii) 频率高的字符码字短.

(iii) 平均编码长度 $\overline{l}$ 满足 $\eta \leq \overline{l} < \eta + 1$.

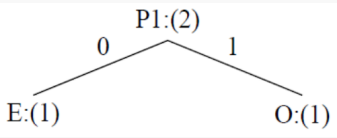
(4) 缺点:

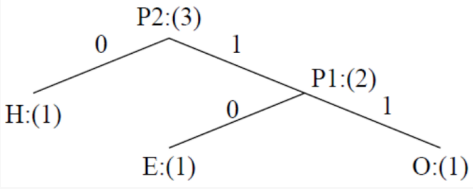
① 局限于单个字符编码.

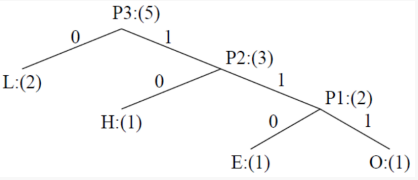
② 需要欲压缩数据的先验统计信息.

[**例**] 对串 "HELLO" 作 Huffman 编码.

[**解**]







3.3.3 扩展 Huffman 编码

[**扩展 Huffman 编码**]

(1) 步骤:

① 将符号表 $S=\{s_1,\cdots,s_n\}$ 中的任 k 个字符的任一顺序分为一组, 得到扩展的符号集 $S^{(k)}$, $|S^{(k)}|=n^k$.

② 对 $S^{(k)}$ 作朴素的 Huffman 编码.

(2) 朴素的 Huffman 编码即 $k=1$ 的情况.

(3) 平均编码长度 $\overline{l}$ 满足 $\eta\leq\overline{l}<\eta+\dfrac{1}{k}$, 比朴素的 Huffman 编码稍好.

(4) 缺点:

① $n>>1$, $k\geq 3$ 时, $|S^{(k)}|$ 过大.

② 需要欲压缩数据的先验统计信息.

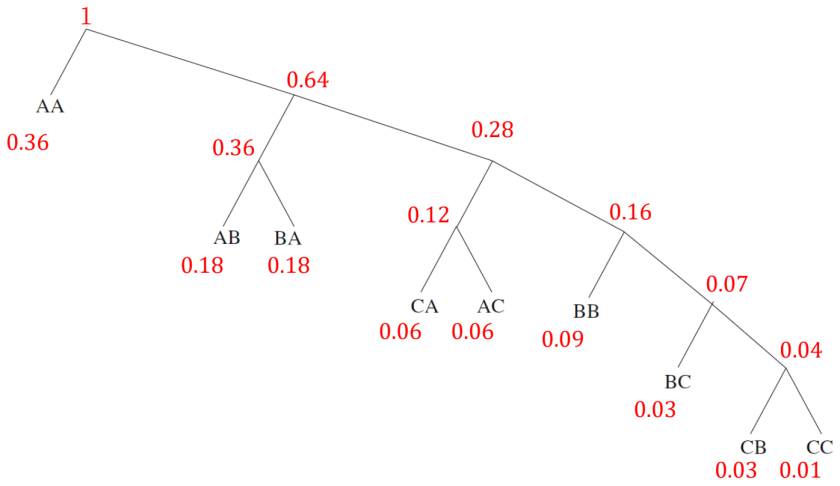
[**例**] 作扩展 Huffman 编码, 取 $k=2$.

符号	A	B	C
概率	0.6	0.3	0.1

[**解**] 扩展符号表:

符号组	AA	AB	BA	CA	AC	BB	BC	CB	CC
概率	0.36	0.18	0.18	0.06	0.06	0.09	0.03	0.03	0.01

扩展 Huffman 编码:



符号组	**概率**	**码字**	**码字长度**
AA	0.36	0	1
AB	0.18	100	3
BA	0.18	101	3
CA	0.06	1100	4
AC	0.06	1101	4
BB	0.09	1110	4
BC	0.03	11110	5
CB	0.03	111110	6
CC	0.01	111111	6

平均码长 $=\dfrac{0.36\times 1+0.18\times 3+\cdots+0.01\times 6}{2}=1.3350$, 此处除以 2 是因为要求每个字符的平均码长.

3.3.4 自适应 Huffman 编码

[**自适应 Huffman 编码**]

(1) 不基于先验概率, 而基于当前收到的实际数据. 统计信息随数据流的到达而动态收集和更新, 即概率和码字动态变化.

(2) 步骤:

① 初始化码字: 为符号分配共识的码字 (常按 ASCII 码顺序分配) , 不含先验信息.

② 更新字符频数.

③ 更新树结构.

(3) 更新树结构:

① 兄弟特性: 同层的节点从左到右频数不减.

② 更新: 同层节点违反兄弟特性时, 当前节点与最远的相同频数的节点交换. 若交换的节点为树中节点, 则也交换子树.

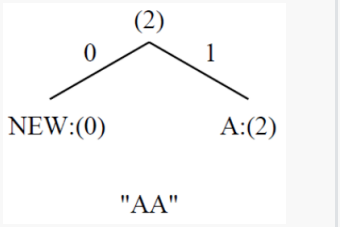
③ 更新完一层的节点后 push up , 递归地更新上层.

[**例**] 对串 "AADCCDD" 作自适应 Huffman 编码.

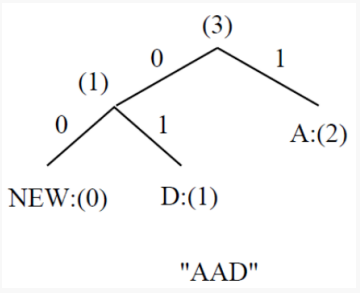
[**解**] 初始码表:

符号	NEW	A	B	C	D
初始编码	0	001	010	011	100

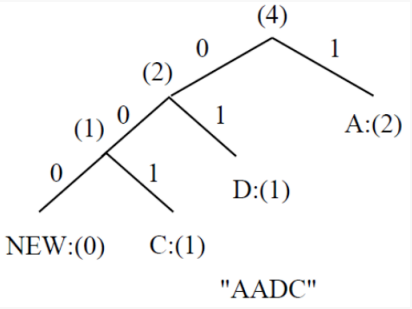
自适应 Huffman 编码:



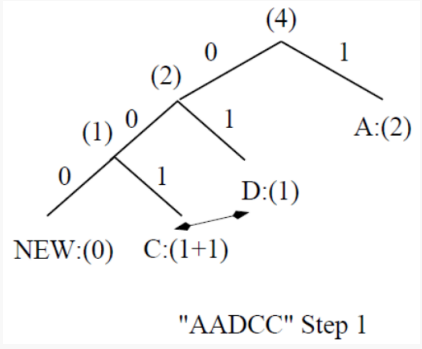
"AA"



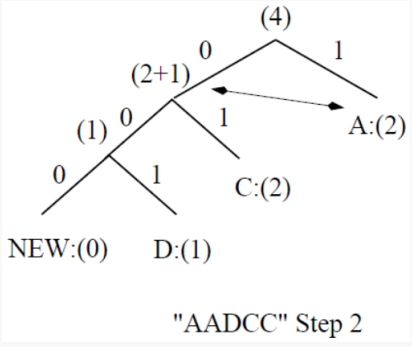
"AAD"



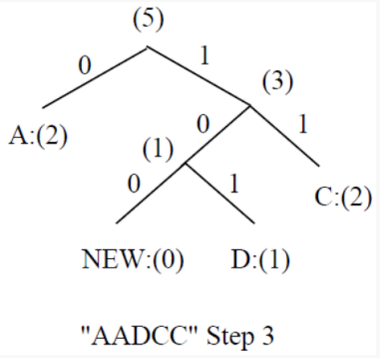
"AADC"



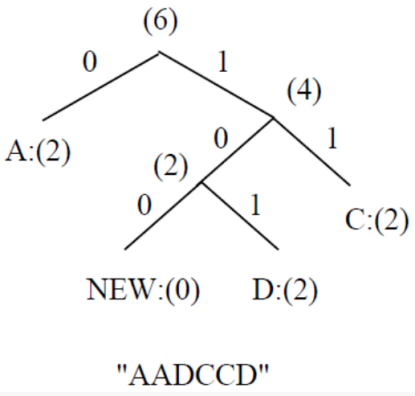
"AADCC" Step 1



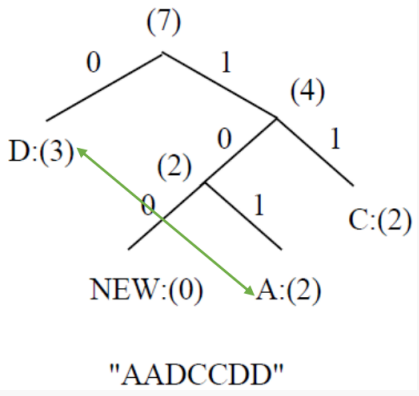
"AADCC" Step 2



"AADCC" Step 3



"AADCCD"



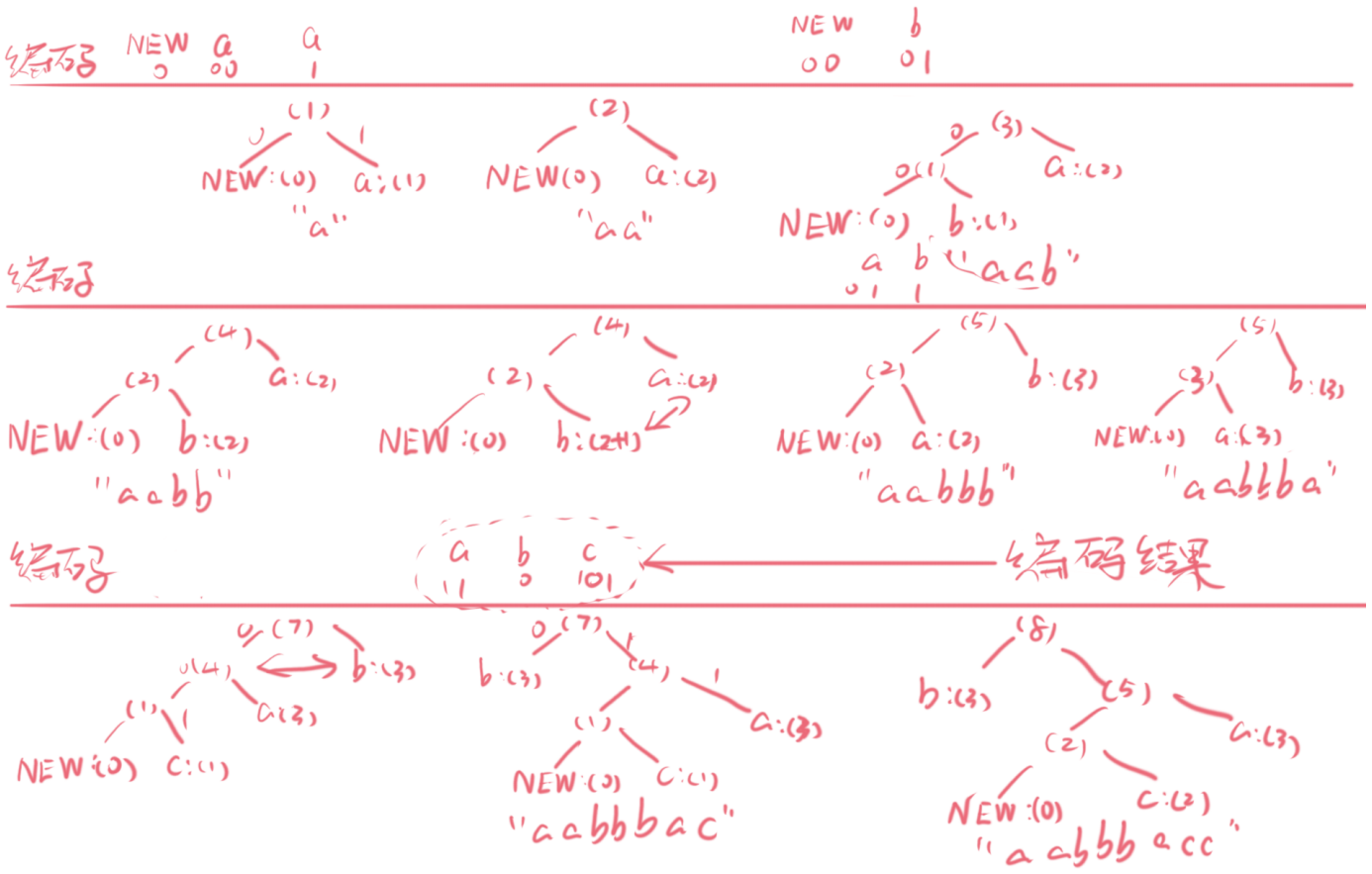
"AADCCDD"

发送给解码器的编码序列:

Symbol	NEW	A	A	NEW	D	NEW	C	C	D	D
Code	0	00001	1	0	00100	00	00011	001	101	101

[**例**] 设字符集 $S=\{a,b,c,d\}$, 约定初始编码 $a=00$, $b=01$, $c=10$, $d=11$. 对串 "aabbbacc" 作自适应 Huffman 编码.

[**解**] 初始编码 $\mathrm{NEW}=0$.



3.4 LZW 压缩

[**LZW 压缩**]

(1) 策略:

① 用固定长度的码字表示变长的串.

② 编解码器动态构建相同的字典.

③ 将越来越长的重复串插入字典, 发送编码而非串.

(2) 步骤:

① 编码:

```
namespace Compression {
    map<vector<int>, int> compressionDictionary;  // 压缩字典 (序列, 码字)
    vector<int> compressionResult;  // 压缩结果

    ...

    void initCompressionDictionary() {
        for (int i = 0; i < MAXV; i++) {
            compressionDictionary[vector<int>(1, i)] = i;
```

```
        }
    }

    void compress() {
        int length = inputSequence.size();
        for (int i = 0, curLength; i < length; i += curLength) {
            // 贪心取在字典中的最长序列
            for (curLength = 1;
                i + (curLength + 1) - 1 < length &&
compressionDictionary.count(vector<int>(inputSequence.begin() + i, inputSequence.begin() +
i + (curLength + 1)));
                curLength++);

            // 编码当前序列
            vector<int> current = vector<int>(inputSequence.begin() + i,
inputSequence.begin() + i + curLength);
            compressionResult.push_back(compressionDictionary[current]);

            // 若有下一个元素，将当前序列连同下一个元素编码后加入字典
            int nextIndex = i + curLength - 1 + 1;
            if (nextIndex < length) {
                vector<int> tmp = current;
                tmp.push_back(inputSequence[nextIndex]);

                if (!compressionDictionary.count(tmp)) {
                    compressionDictionary[tmp] = compressionDictionary.size();
                }
            }
        }
    }

    ...
}
using namespace Compression;
```

② 解码:

```
namespace Decompression {
    map<int, vector<int>> decompressionDictionary;  // 压缩字典（码字，序列）
    set<vector<int>> decompressionSet;  // 已在字典中的序列
    vector<int> decompressionResult;  // 压缩结果

    ...

    void initDecompressionDictionary() {
        for (int i = 0; i < MAXV; i++) {
            decompressionDictionary[i] = vector<int>(1, i);
        }
    }

    void decompress() {
        vector<int> lastOutput;
        for (auto code : compressionResult) {
            vector<int> output;
            if (decompressionDictionary.count(code)) {
```

```
                output = decompressionDictionary[code];
            }
            else {  // 编码器动作先于解码器
                output = lastOutput;
                output.push_back(lastOutput[0]);
            }

            decompressionResult.insert(decompressionResult.end(), output.begin(),
output.end());

            if (lastOutput.size()) {
                vector<int> tmp = lastOutput;
                tmp.push_back(output[0]);

                if (!decompressionSet.count(tmp)) {
                    decompressionDictionary[decompressionDictionary.size()] = tmp;
                    decompressionSet.insert(tmp);
                }
            }

            lastOutput = output;
        }
    }

    ...
}
using namespace Decompression;
```

[**例**] 对串 "0110011" 作 LZW 压缩, 写出压缩和解压过程.

[**解**]

(1) 压缩:

① 初始压缩字典: $0:0$, $1:1$.

② LZW 压缩:

current	next	code	dictionary
0	1	0	$01:2$
1	1	1	$11:3$
1	0	1	$10:4$
0	0	0	$00:5$
01	1	2	$011:6$
1		1	

③ 压缩结果: $0,1,1,0,2,1$.

(2) 解压:

① 初始解压字典: $0:0$, $1:1$.

② LZW 解压:

code	lastOutput	output	dictionary
0		0	
1	0	1	$01:2$
1	1	1	$11:3$
0	1	0	$10:4$
2	0	01	$00:5$
1	01	1	$011:6$

③ 解压结果: $0,1,1,0,01,1$.

[**例**] 对串 "ABABBABCABABBA" 作 LZW 压缩, 写出压缩和解压过程.

[**解**]

(1) 压缩:

① 初始字典: $A:1$, $B:2$, $C:3$.

② LZW 压缩:

current	next	code	dictionary
A	B	1	$AB:4$
B	A	2	$BA:5$
AB	B	4	$ABB:6$
BA	B	5	$BAB:7$
B	C	2	$BC:8$
C	A	3	$CA:9$
AB	A	4	$ABA:10$
ABB	A	6	$ABBA:11$
A		1	

③ 压缩结果: $1,2,4,5,2,3,4,6,1$.

(2) 解压:

① 初始字典: $A:1$, $B:2$, $C:3$.

② LZW 解压:

code	lastOutput	output	dictionary
1		A	
2	A	B	$AB:4$
4	B	AB	$BA:5$
5	AB	BA	$ABB:6$
2	BA	B	$BAB:7$
3	B	C	$BC:8$
4	C	AB	$CA:9$
6	AB	ABB	$ABA:10$
1	ABB	A	$ABBA:11$

③ 解压结果: A,B,AB,BA,B,C,AB,ABB,A .

[**例**] 对某串作 LZW 压缩的结果为 $1,2,4,5,2,3,6,10$. 已知初始字典 $A:1$, $B:2$, $C:3$, 作 LZW 解压.

[**解**] ① 初始字典: $A:1$, $B:2$, $C:3$.

② LZW 解压:

code	lastOutput	output	dictionary
1		A	
2	A	B	$AB:4$
4	B	AB	$BA:5$
5	AB	BA	$ABB:6$
2	BA	B	$BAB:7$
3	B	C	$BC:8$
6	C	ABB	$CA:9$
10	ABB	特殊情况: $ABB+A=ABBA$	$ABBA:10$

③ 解压结果: $A,B,AB,BA,B,C,ABB,ABBA$.

3.5 算术编码

[**算术编码**]

(1) 编码:

① 基本步骤:

```
low = 0.0, high = 1.0, range = 1.0;

while (symbol != terminator) {
    get(symbol);
    high = low + range * Range_high(symbol);
    low = low + range * Range_low(symbol);
    range = high - low;
}

output a code so tha low <= code < high;
```

② 其中最后一步:

```
code = 0;
k = 1;
while (value(code) < low) {
    assign 1 to the k-th binary fraction bit;
    if (value(code) > high) {
        replace the k-th bit by 0;
    }
    k++;
}
```

(2) 解码:

```
do {
    find a symbol s s.t. Range_low(s) <= value < Range_high(s);
    output s;

    low = Range_low(s), high = Range_high(s), range = high - low;
    value = (value - low) / range;
} while (symbol != terminator);
```

(3) 编码位数的上下界:

① 熵为对每个符号编码所需的平均位数的理论下界: $\sum_i \log_2 \frac{1}{p_i} = \log_2 \frac{1}{\prod_i p_i} = \log_2 \frac{1}{range}$.

② 最坏上界: $\left\lceil \log_2 \frac{1}{\prod_i p_i} \right\rceil = \left\lceil \log_2 \frac{1}{range} \right\rceil$.

算术编码的最坏上界近似于 Huffman 编码的最好下界.

(4) 算术编码的性能一般优于 Huffman 编码, 因为前者将整个消息视为一个单元, 后者受到必须为每个符号分配整数位的限制.

(5) 缺点:

① 串长时, 区间小, 需高精度.

② 编码器只有得到整个序列才能编码, 解码器只有得到整个序列的码字才能解码.

改进: 缩放增量编码.

[注] 常用的 2 的负指数幂的值:

k	-1	-2	-3	-4	-5	-6	-7	-8
2^k	0.5	0.25	0.125	0.0625	0.03125	0.015625	0.0078125	0.00390625

k	-9	-10
2^k	0.001953125	0.0009765625

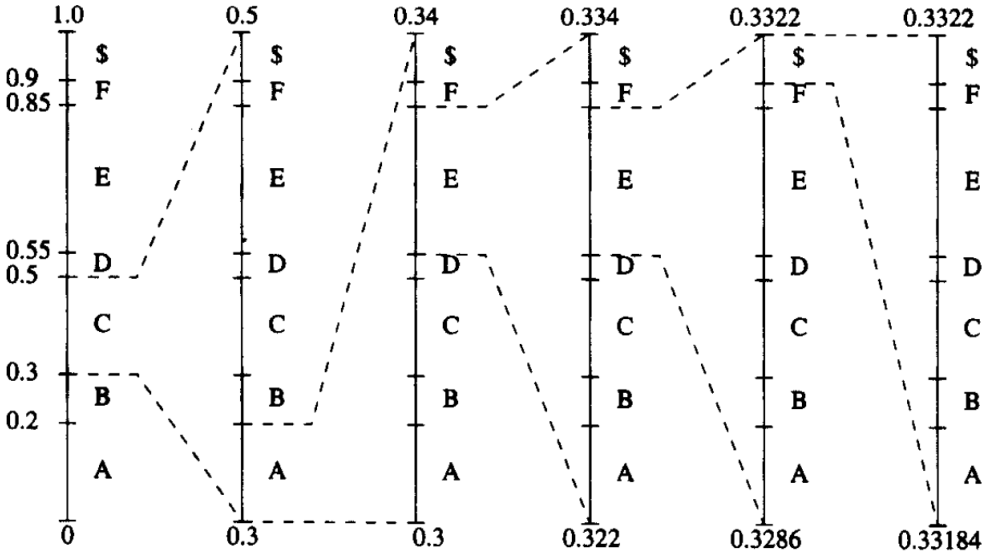
[**例**] 设符号 $\{A, B, C, D, E, F, \$\}$ 的概率分布如下, 其中 $ 是结束符.

符号	A	B	C	D	E	F	$
概率	0.2	0.1	0.2	0.05	0.3	0.05	0.1

对串 "CAEE$" 作算术编码.

[**解**]

(1) 编码:



最终 $low = 0.33184$, $high = 0.3322$, $range = high - low = 0.3322 - 0.33184 = 0.00036$.

(或 $range = p_C \cdot p_A \cdot p_E^2 \cdot p_\$ = 0.2 \times 0.2 \times (0.3)^2 \times 0.1 = 0.00036$.)

下求一个二进制码字 $code$ s.t. $low \le code < high$. 初始时 $code = 0$.

① 将 $code$ 的二进制小数第 1 位置 1 , 则 $code = (0.1)_2 = 0.5 > high$,

故将 $code$ 的二进制小数第 1 位置 0 , 即 $code = (0.0)_2$.

② 将 $code$ 的二进制小数第 2 位置 1 , 则 $code = (0.01)_2 = 0.25 < high$, 且 $0.25 < low$.

③ 将 $code$ 的二进制小数第 3 位置 1 , 则 $code = (0.011)_2 = 0.375 > high$,

故将 $code$ 的二进制小数第 3 位置 0 , 即 $code = (0.010)_2$.

④ 将 $code$ 的二进制小数第 4 位置 1 , 则 $code = (0.0101)_2 = 0.3125 < high$, 且 $0.3125 < low$.

⑤ 将 $code$ 的二进制小数第 5 位置 1 , 则 $code = (0.01011)_2 = 0.34375 > high$,

故将 $code$ 的二进制小数第 5 位置 0 , 即 $code = (0.01010)_2$.

⑥ 将 $code$ 的二进制小数第 6 位置 1 , 则 $code = (0.010101)_2 = 0.32815 < high$,

且 $0.32815 < low$.

⑦ 将 $code$ 的二进制小数第 7 位置 1 , 则 $code = (0.0101011)_2 = 0.3359375 > high$,

故将 $code$ 的二进制小数第 7 位置 0 , 即 $code = (0.0101010)_2$.

⑧ 将 $code$ 的二进制小数第 8 位置 1，则 $code = (0.01010101)_2 = 0.33203125 < high$，

此时 $0.33203125 \geq low$，终止.

编码结果: $(0.01010101)_2 = 0.33203125$.

(2) 解码: 初始时 $value = 0.33203125$.

① 因 $Range\_low(C) = 0.3 \leq value < 0.5 = Range\_high(C)$，故输出 C.

$low = Range\_low(C) = 0.3$，$high = Range\_high(C) = 0.5$，

$range = high - low = 0.2$，$value = \dfrac{value - low}{range} = \dfrac{0.33203125 - 0.3}{0.2} = 0.16015625$.

② 因 $Range\_low(A) = 0.0 \leq value < 0.2 = Range\_high(A)$，故输出 A.

$low = Range\_low(A) = 0.0$，$high = Range\_high(A) = 0.2$，

$range = high - low = 0.2$，$value = \dfrac{value - low}{range} = \dfrac{0.16015625 - 0.0}{0.2} = 0.80078125$.

③ 因 $Range\_low(E) = 0.55 \leq value < 0.85 = Range\_high(E)$，故输出 E.

$low = Range\_low(E) = 0.55$，$high = Range\_high(E) = 0.85$，

$range = high - low = 0.3$，$value = \dfrac{value - low}{range} = \dfrac{0.80078125 - 0.55}{0.3} = 0.8359375$.

④ 因 $Range\_low(E) = 0.55 \leq value < 0.85 = Range\_high(E)$，故输出 E.

$low = Range\_low(E) = 0.55$，$high = Range\_high(E) = 0.85$，

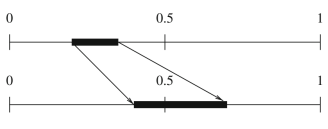
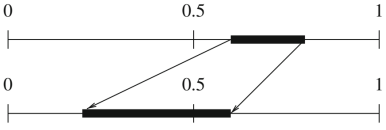
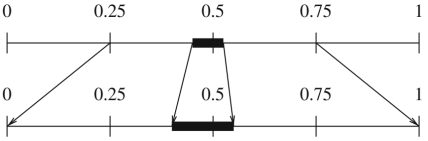
$range = high - low = 0.3$，$value = \dfrac{value - low}{range} = \dfrac{0.8359375 - 0.55}{0.3} = 0.953125$.

⑤ 因 $Range\_low(\$) = 0.55 \leq value < 0.85 = Range\_high(\$)$，故输出 $\$$，终止.

[缩放和增量编码]

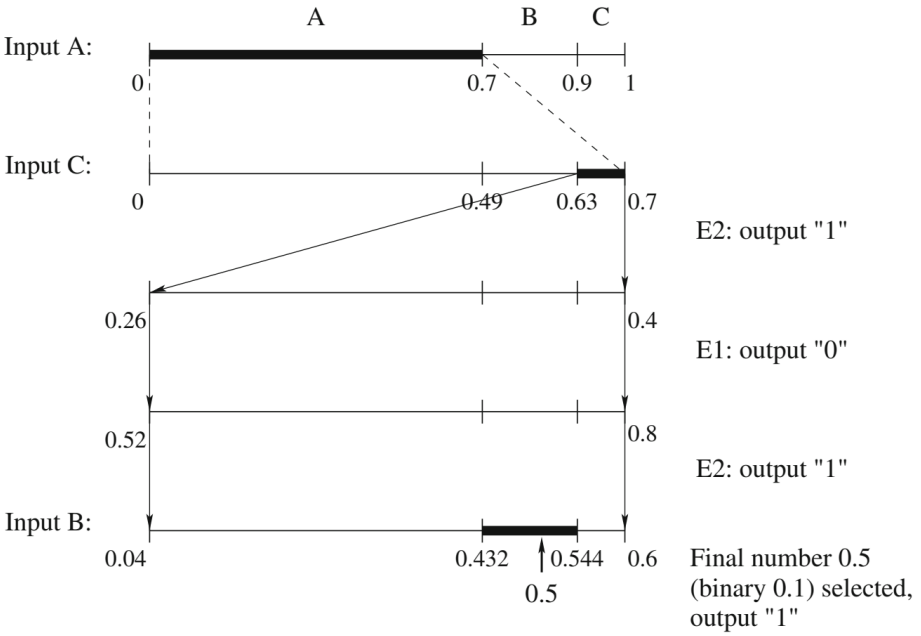
(1) 策略: 注意到表示小区间 $[low, high]$ 中的小数需要很多二进制位, 但二进制数的前缀相同, 可输出公共的前缀, 后续不再考虑这些位.

(2) 三种缩放: 下图中黑色线段表示区间 $[low, high)$ $(low < high)$.

缩放	$E1$ 缩放	$E2$ 缩放	$E3$ 缩放
条件	区间都在左半边, 即 $high \leq 0.5$	区间都在右半边, 即 $low \geq 0.5$	区间横跨中点, 且 $\begin{cases} low \geq 0.25 \\ high \leq 0.75 \end{cases}$
操作	$\begin{cases} low = 2 \cdot low \\ high = 2 \cdot high \end{cases}$ 可通过左移一位实现	$\begin{cases} low = 2 \cdot (low - 0.5) \\ high = 2 \cdot (high - 0.5) \end{cases}$ 可通过左移一位实现	$\begin{cases} low = 2 \cdot (low - 0.25) \\ high = 2 \cdot (high - 0.25) \end{cases}$
示意图			

[**例**] 符号 A、B、C 的概率分别为 0.7、0.2、0.1. 对串 "ACB" 作缩放和增量编码, 假设编码器已知序列长度为 3.

[**解**]



(1) 接收符号 A, 区间变为 $[0, 0.7)$, 无触发缩放.

(2) 接收符号 C,

① 区间变为 $[0.63, 0.7)$, 都在右半边, 触发 $E2$ 缩放, 输出 1,

区间变为 $[2 \times (0.63 - 0.5), 2 \times (0.7 - 0.5)) = [0.26, 0.4)$.

② 区间都在左半边, 触发 $E1$ 缩放, 输出 0,

区间变为 $[2 \times 0.26, 2 \times 0.4) = [0.52, 0.8)$.

③ 区间都在右半边, 触发 $E2$ 缩放, 输出 1,

区间变为 $[2 \times (0.52 - 0.5), 2 \times (0.8 - 0.5)) = [0.04, 0.6)$, 无触发缩放.

(3) 接收符号 B, 区间变为 $[0.432, 0.544)$, 生成二进制码字 $(0.1)_2 = 0.5$, 输出 1.

编码结果: $(1011)_2 = 0.6875$.

3.6 无损图像压缩

3.6.1 图像差分编码

[**图像差分编码**] 对图像 $I(x, y)$, 定义:

(1) 简单差分图像: $d(x, y) = I(x, y) - I(x - 1, y)$.

(2) 用 2D 离散 Laplace 算子定义差分图像:

$$d(x, y) = 4 \cdot I(x, y) - I(x, y - 1) - I(x, y + 1) - I(x + 1, y) - I(x - 1, y).$$

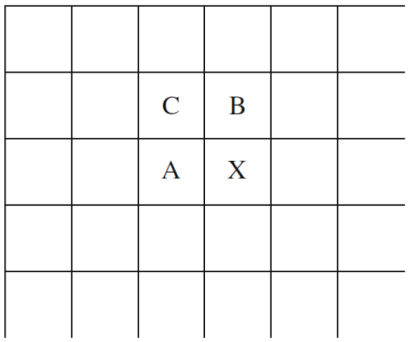
3.6.2 无损 JPEG

[无损 JPEG]

步骤:

① 差分预测: **预测器**将至多三个相邻的像素值组合成当前像素的预测值.

(i) 预测器: 像素 X 的预测器如下.



Predictor	Prediction
P1	A
P2	B
P3	C
P4	$A+B-C$
P5	$A+(B-C)/2$
P6	$B+(A-C)/2$
P7	$(A+B)/2$

(ii) $I(0,0)$ 传原值, 第一行的像素用 $P1$ 预测器, 第一列的像素用 $P2$ 预测器.

② 编码: 编码器比较预测值与实际像素值, 用无损压缩算法编码它们间的差值.

3.7 失真度量

[失真度量]

(1) **均方差** $\mathrm{MSE}=\sigma^2=\dfrac{1}{N}\sum\limits_{i=1}^{N}(x_i-y_i)^2$, 其中 x_i 、y_i 、N 分别是输入序列、重现序列和序列长度.

(2) **信噪比** $\mathrm{SNR}=10\log_{10}\dfrac{\sigma_x^2}{\sigma^2}$ (以 dB 为单位时) , 其中 σ_x^2 为原序列的均方.

(3) **峰值信噪比** $\mathrm{PSNR}=10\log_{10}\dfrac{x_{\mathrm{peak}}^2}{\sigma^2}$, 其中 x_{peak} 是原序列的最大值.

[例] 原始数据 $[12,12,12,12,12,8,8,12]$ 经压缩和解压后的数据为 $[12,12,12,8,12,8,12,12]$. 分别求 MSE 、SNR 、PSNR .

[解] $x=[12,12,12,12,12,8,8,12]$, $y=[12,12,12,8,12,8,12,12]$.

$$\mathrm{MSE}=\sigma^2=\frac{1}{N}\sum_{i=1}^{N}(x_i-y_i)^2=\frac{(12-8)^2+(8-12)^2}{8}=4\,.$$

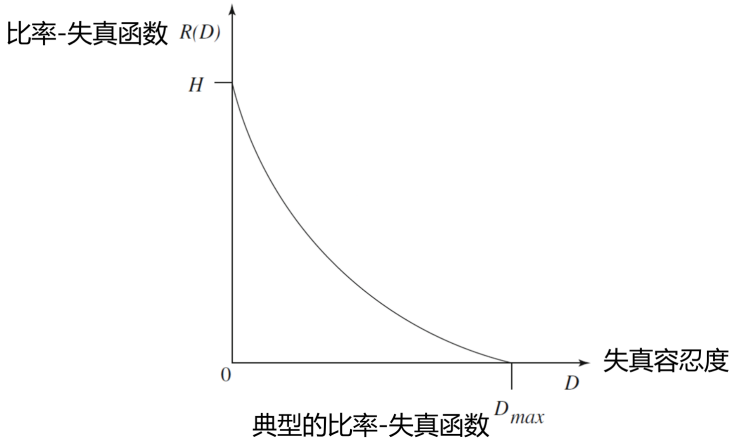
$$\mathrm{SNR}=10\log_{10}\frac{\sigma_x^2}{\sigma^2}=10\log_{10}\frac{\dfrac{6\times12^2+2\times8^2}{8}}{4}\approx14.91\,.$$

$$\mathrm{PSNR}=10\log_{10}\frac{x_{\mathrm{peak}}^2}{\sigma^2}=10\log_{10}\frac{12^2}{4}\approx15.563\,.$$

[比率失真理论]

(1) 比率-失真函数曲线:

D 为失真容忍度, $R(D)$ 表示源数据编码的最低比率.



典型的比率-失真函数

(2) 特殊点:

① $D = 0$, 即无损时的最小比率是源数据的熵.

② $R(D) = 0$, 即比率最小时, 失真达到最大值, 此时完全未进行编码.

3.8 量化

[量化]

(1) 定义: 将不同的输出值的数量降到更小的集合.

(2) 量化是任何有损方案的核心, 是信息损失的主要来源.

(3) 分类:

① 标量量化: 均匀量化、非均匀量化.

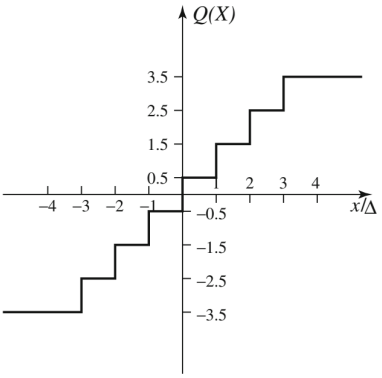
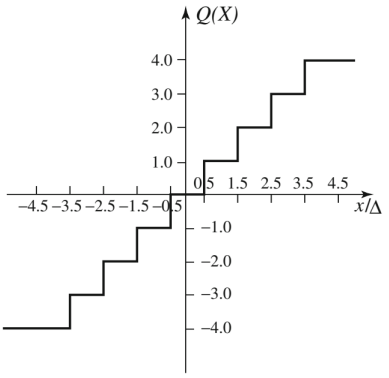
② 向量量化: 如 LZW 压缩、颜色查找表.

[均匀标量量化]

(1) 策略: 将输入值划分为等间隔的区间, 区间端点称为**决策边界**, 区间长度 Δ 称为**步长**.

(2) 目标: 调整步长, 匹配输入, 减小失真.

(3) 分类:

量化器	中高型	中宽型
示意图		
$\Delta = 1$ 时的表达式	$Q_{\text{midrise}}(x) = \lceil x \rceil - 0.5$	$Q_{\text{midtread}}(x) = \lfloor x \rfloor + 0.5$
台阶	偶数级	奇数级

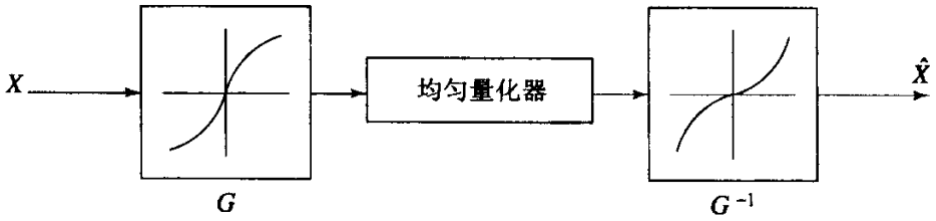
源数据以很小的正负数间的波动表示零值时, 中宽型量化器有效, 因为可精确稳定地表示零值.

[非均匀标量量化]

压缩扩展量化器:

① 策略: 输入通过一个**压缩函数** G 后由一个均匀量化器量化, 随后用**扩展函数** G^{-1} 将量化值映射回去.

② 示意图: 下图中 $\hat{X}$ 是输入 X 的量化值.



③ 对有上界的输入源, 任何非均匀量化器都可用压缩扩展量化器表示.

④ 常用的压缩扩展器:

(i) μ 律压缩扩展器.

(ii) A 律压缩扩展器.

[**向量量化**]

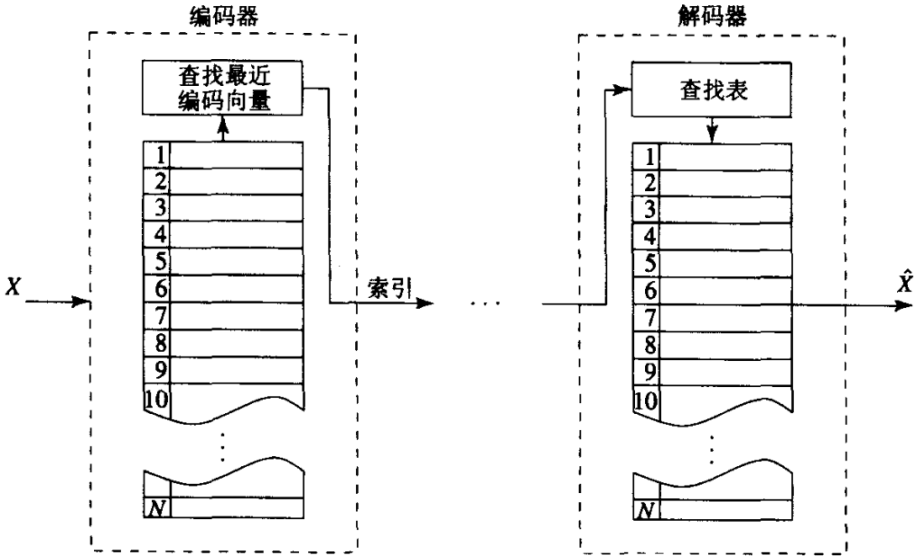
(1) 原因: 根据 Shannon 的信息论: 成组样本的压缩比单个样本的压缩效果好.

(2) 策略:

① 将一系列的样本组成向量.

② 向量码的集合组成**码本**.

(3) 示意图: 以 LZW 压缩和颜色查找表为例.



3.9 变换编码

[**变换编码**]

原因:

① 编码向量比编码标量更有效.

② 若 Y 是向量 X 经线性变换 T 得到的结果, 则 Y 的分量相关性更少, 对 Y 编码比对 X 编码更有效.

③ 大部分信息集中在变换后的向量的前几个分量, 其它分量可粗略量化, 甚至置零, 这样带来的信号失真很小.

[离散余弦变换, Discrete Cosine Transform, DCT] 设 $C(x)=\begin{cases}\frac{\sqrt{2}}{2}, & x=0\\ 1, & x\neq 0\end{cases}$.

(1) 1D DCT :

① 正变换: $F(u)=\frac{C(u)}{2}\sum\limits_{i=0}^{7}\cos\frac{(2i+1)\cdot u\cdot\pi}{16}\cdot f(i)$, 其中 $i,u=0,\cdots,7$.

② 逆变换: $\widetilde{f}(i)=\sum\limits_{i=0}^{7}\frac{C(u)}{2}\cos\frac{(2i+1)\cdot u\cdot\pi}{16}\cdot F(u)$, 其中 $i,u=0,\cdots,7$.

(2) 2D DCT :

① 正变换: $F(u,v)=\frac{C(u)\cdot C(v)}{4}\sum\limits_{i=0}^{7}\sum\limits_{j=0}^{7}\cos\frac{(2i+1)\cdot u\cdot\pi}{16}\cdot\cos\frac{(2j+1)\cdot v\cdot\pi}{16}\cdot f(i,j)$, 其中 $i,j,u,v=0,\cdots,7$.

② 逆变换: $\widetilde{f}(i,j)=\sum\limits_{i=0}^{7}\sum\limits_{j=0}^{7}\frac{C(u)\cdot C(v)}{4}\cos\frac{(2i+1)\cdot u\cdot\pi}{16}\cdot\cos\frac{(2j+1)\cdot v\cdot\pi}{16}\cdot F(u,v)$, 其中 $i,j,u,v=0,\cdots,7$.

[例] 设输入信号 $f=[0,10,20,30,40,50,60,70]$. 求该信号的 1D DCT 中的 $F(0)$.

[解] $C(0)=\frac{\sqrt{2}}{2}$.

$$F(0)=\frac{C(0)}{2}\sum_{i=0}^{7}\cos\frac{(2i+1)\cdot u\cdot\pi}{16}f(i)=\frac{\sqrt{2}}{4}\sum_{i=0}^{7}1\cdot f(i)$$

$$=\frac{\sqrt{2}}{4}(0+10+20+30+40+50+60+70)=\frac{140}{\sqrt{2}}\approx 99.1\ .$$

3.10 JPEG 标准

3.10.1 JPEG 的过程

[JPEG 标准]

(1) **JPEG 标准**是一种有损图像压缩方法, 应用了 DCT 变换编码.

(2) JPEG 中应用 DCT 变换的编码效率基于如下特性:

① 图像区域内, 有用的图像内容变化相对缓慢, 即小区域内亮度值的变化不太频繁.

② 人眼对高频分量的损失的感知能力远低于对低频分量的损失的感知能力.

③ 人眼对灰度的敏感度远高于对颜色的敏感度. (故可采用色度二次采样 $4:2:0$.)

(3) 步骤:

① 将 RGB 转化为 YCbCr .

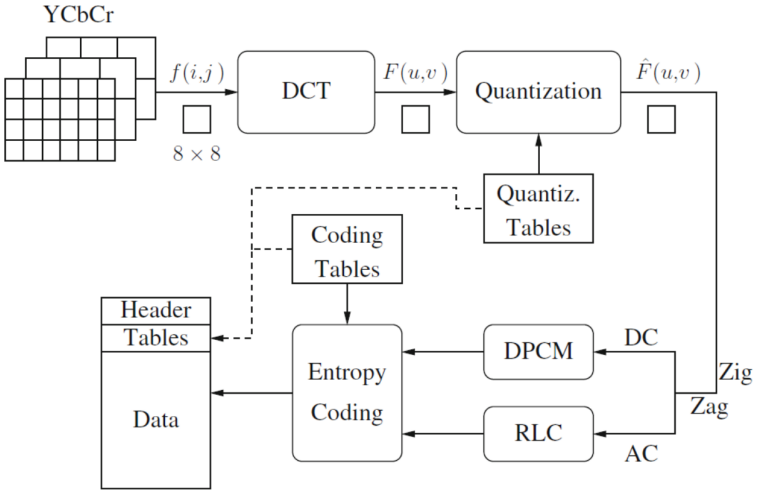
② 对图像块作 DCT .

③ 量化.

④ Z 字型扫描、游程编码.

⑤ 熵编码.

(4) 示意图:



[图像块的 DCT]

(1) 策略: 将图像划分为 8×8 的图像块, 对每个图像块作 2D DCT .

(2) 高压缩率时图像不连贯, 成块状.

原因: 划分图像块时孤立了当前块与周围块的内容.

[量化]

(1) 目的: 减少压缩图像的总位数.

(2) 策略: 对每个图像块量化.

(3) 量化可表示为 $\hat{F}(u,v)=\operatorname{round}\left(\dfrac{F(u,v)}{Q(u,v)}\right)$,

其中 $F(u,v)$ 为 DCT 系数, $Q(u,v)$ 为**量化矩阵**, $\operatorname{round}(x)$ 函数表示对数 x 四舍五入取整.

(4) 亮度和色度采取不同的量化表, 量化表右下角的值更大, 以丢弃更多高频信息.

表 9-1 亮度量化表

16	11	10	16	24	40	51	61
12	12	14	19	26	58	60	55
14	13	16	24	40	57	69	56
14	17	22	29	51	87	80	62
18	22	37	56	68	109	103	77
24	35	55	64	81	104	113	92
49	64	78	87	103	121	120	101
72	92	95	98	112	100	103	99

表 9-2 色度量化表

17	18	24	47	99	99	99	99
18	21	26	66	99	99	99	99
24	26	56	99	99	99	99	99
47	66	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99

(5) 可通过量化矩阵乘比例来改变压缩率.

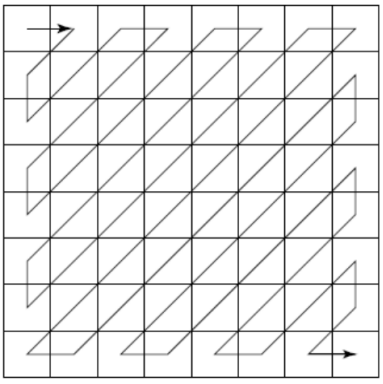
(6) 量化是损失信息的主要原因.

[AC 系数的游程编码, Run-length Coding, RLC]

(1) 目的: 将量化后的 DCT 系数 $\hat{F}(u,v)$ 转化为 $(\mathrm{RunLenth},\mathrm{Value})$, 即 $(\text{跳过 }0\text{ 的个数},\text{下一非零值})$ 的形式.

(2) 采用 Z 字型扫描, 将 $\hat{F}(u,v)$ 矩阵变换为长度为 64 的向量.

① 示意图:



② 目的: Z 字型扫描可能得到一长串的 0, 使得 RLC 更有效.

[**例**] 对 DCT 系数 $\hat{F}(u,v)=\begin{bmatrix}32&6&-1&0&0&0&0&0\\-1&0&0&0&0&0&0&0\\-1&0&1&0&0&0&0&0\\-1&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\end{bmatrix}$ 作 RLC .

[**解**]

(1) 先作 Z 字型扫描得 $(32,6,-1,-1,0,-1,0,0,0,-1,0,0,1,0,\cdots,0)$.

(2) 去掉第一个 DC 分量 32 , 作 RLC 得 $(0,6),(0,-1),(0,-1),(1,-1),(3,-1),(2,1),(0,0)$, 其中 $(0,0)$ 表示块结尾.

[**例**] 已知 DCT 系数 $F(u,v)=\begin{bmatrix}640&35&44&7&5&1&-7&3\\26&-3&1&5&-4&-5&3&2\\-15&2&-4&3&4&3&1&-4\\5&-4&4&1&-3&2&-3&0\\1&-4&-3&-7&-2&0&1&0\\-2&6&1&1&0&0&0&0\\5&-3&0&3&1&1&0&0\end{bmatrix}$. 求:

[1] 量化后的值 $\hat{F}(u,v)$.

[2] 对 AC 系数作 RLC 的编码结果.

表 9-1　亮度量化表

16	11	10	16	24	40	51	61
12	12	14	19	26	58	60	55
14	13	16	24	40	57	69	56
14	17	22	29	51	87	80	62
18	22	37	56	68	109	103	77
24	35	55	64	81	104	113	92
49	64	78	87	103	121	120	101
72	92	95	98	112	100	103	99

表 9-2　色度量化表

17	18	24	47	99	99	99	99
18	21	26	66	99	99	99	99
24	26	56	99	99	99	99	99
47	66	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99

[**解**]

[1] $\hat{F}(0,0)=\mathrm{round}\left(\dfrac{F(0,0)}{Q(0,0)}\right)=\mathrm{round}\left(\dfrac{640}{16}\right)=40$, $\cdots$.

$$\hat{F}(u,v)=\begin{bmatrix}40&3&4&0&0&0&0&0\\2&0&0&0&0&0&0&0\\-1&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\end{bmatrix}.$$

[2] Z 字型扫描得 $(40,3,2,-1,0,4,0,\cdots,0)$,

去掉第一个 DC 分量 40 , 作 RLC 得: $(0,3),(0,2),(0,-1),(1,4),(0,0)$.

[**DC 系数的 DPCM 编码**]

(1) 策略: 对 DC 系数作 DPCM 编码, 即求 DC 系数构成的序列的差分序列.

(2) 每个 8×8 图像块只有一个 DC 系数, 它表示该图像块的平均亮度.

(3) 不同块间 DC 系数差异较大, 邻近块间 DC 系数相差较小, 故可采用 DPCM 编码.

(4) AC 系数的 RLC 需对每个图像块单独进行, 而 DC 系数的 DPCM 编码只需对整个图像进行一次.

[**熵编码**]

(1) DC 系数:

① 每个 DC 系数经 DPCM 编码后可用 $(\mathrm{Size}, \mathrm{Amplitude})$, 即 $(\text{位数}, \text{二进制})$ 表示, 负值采用绝对值的反码表示.

② 对照表:

Size	Amplitude
1	$-1, 1$
2	$3, -2, 2, 3$
3	$-7 .. -4, 4 .. 7$
4	$-15 .. -8, 8 .. 15$
.	.
.	.
.	.
10	$-1023 .. -512, 512 .. 1023$

③ Size 作 Huffman 编码, 因为值域小, 出现频率高; $\mathrm{Amplitude}$ 不作 Huffman 编码, 因为值域大, Huffman 编码效果差.

(2) AC 系数:

① AC 系数经 RLC 后得到 $(\mathrm{RunLenth}, \mathrm{Value})$ 的形式, 将 Value 作熵编码, 用 $(\mathrm{Size}, \mathrm{Amplitude})$ 表示.

② 为节省空间, 将 $\mathrm{RunLenth}$ 和 Size 各用 4 位表示并合成 $1\ \mathrm{B}$, 得到如下两个符号:

(i) Symbol_1 : $(\mathrm{RunLength}, \mathrm{Size})$, 作 Huffman 编码.

(ii) Symbol_2 : $(\mathrm{Amplitude})$, 不作 Huffman 编码.

③ 4 位的 $\mathrm{RunLenth}$ 只能表示长度 $0 \sim 15$ 的零串. 零串长度超过 15 时, Symbol_1 用特殊的扩展编码 $(15, 0)$. 最坏情况 Symbol_1 结束前需 3 个连续的 $(15, 0)$.

[**例**] 设 DC 系数经 DPCM 编码后的结果为 $[150, -6]$. 作 Huffman 编码, 码表如下:

Size	Amplitude
1	$-1, 1$
2	$3, -2, 2, 3$
3	$-7 .. -4, 4 .. 7$
4	$-15 .. -8, 8 .. 15$
.	.
.	.
.	.
10	$-1023 .. -512, 512 .. 1023$

[**解**]

(1) 150 占 $\lceil \log_2 \lvert 150 \rvert \rceil = 8$ 位, 则 $150 = (10010110)_2$, 编码结果为 $(8, 10010110)$.

(2) -6 占 $\lceil \log_2 \lvert -6 \rvert \rceil = 3$ 位, $6 = (110)_2$, 则 $(-6) = (001)_2$, 编码结果为 $(3, 001)$.

[**例**] 设 DC 系数 $\mathrm{DC} = [130, 135, 141, 180, 182, 179]$. 作 Huffman 编码, 码表如下:

Size	Amplitude
1	$-1, 1$
2	$3, -2, 2, 3$
3	$-7 .. -4, 4 .. 7$
4	$-15 .. -8, 8 .. 15$
.	.
.	.
.	.
10	$-1023 .. -512, 512 .. 1023$

[**解**] 对 DC 系数作 DPCM 得: $d = [130, 5, 6, 39, 2, -3]$.

(1) $130 = (10000010)_2$, 占 8 位, 编码结果为 $(8, 10000010)$.

(2) $5 = (101)_2$, 占 3 位, 编码结果为 $(3, 101)$.

(3) $6 = (110)_2$, 占 3 位, 编码结果为 $(3, 110)$.

(4) $39 = (100111)_2$, 占 6 位, 编码结果为 $(6, 100111)$.

(5) $2 = (10)_2$, 占 2 位, 编码结果为 $(2, 10)$.

(6) $3 = (11)_2$, 则 $(-3) = (00)_2$, 占 2 位, 编码结果为 $(2, 00)$.

[**对比 HEIF 、TPG 与 JPEG**]

(1) 主要特性:

	HEIF	TPG	JPEG
压缩效率	高, 节省约 50% 的空间	高, 比 JPEG 高	较低
图像质量	高, 较少伪影	高, 保持图像质量	一般, 有伪影
多功能性	支持单图像、多图像、序列、元数据	主要支持静态图像	仅支持单图像
兼容性	较差, 需新硬件和软件支持	较差, 推广和支持有限	高, 广泛支持
解码复杂性	高, 需多处理能力	较低, 设计简洁	低, 处理速度快
支持动画	是	否	否

(2) 普遍性:

① JPEG 已普遍支持.

② HEIF 正逐步获得更多支持, 尤其在苹果设备上.

③ TPG 是新兴格式, 需要更多时间推广和获得支持.

3.10.2 JPEG2000

[**JPEG2000 的改进**]

(1) 低位率压缩.

(2) 无损和有损压缩.

(3) 大图像.

(4) 单一的解压体系结构.

(5) 噪声环境中的传输.

(6) 渐进传输.

(7) 感兴趣区域编码.

(8) 计算机生成的影像.

(9) 复合文档.

(10) 多通道.

[**比较 JPEG 与 JPEG2000 的压缩效果**]

(1) JPEG2000 的平均 PSNR 更高.

(2) JPEG2000 压缩的图像的处理痕迹较小.

[**感兴趣区域编码**, **Region-of-Interest Coding**, **ROI**]

(1) JPEG2000 可实现 ROI , 即相比于图像的背景或其它部分, 某些部分可采取高质量编码.

(2) 下面的机制保证了 ROI 会有比背景图案更高的质量:

① MAXSHIFT 方法是一种可伸缩方法, 它可将 ROI 内的系数增加到更高层的位平面中.

③ 在这种嵌入式的编码过程中, 所处理的比特将被放置到图像的非 ROI 部分前面, 故给定一个缩小的码率, ROI 将在图像的其它部分前解码和改善.