Image Compression-II

Contents

- ✓ Lossless Compression Techniques
  - Shannon-Fano Coding
  - Huffman Coding
  - Arithmetic Coding

Uniquely Decodable Codes

Variable length code assigns a bits of string (codeword) of variable length to every message value

e.g. a = 1, b = 01, c = 101, d = 011

What if you get the sequence of bits 1011 ?

Is it aba, ca, or, ad?

Uniquely decodable code is a variable length code in which bit strings can always be uniquely decomposed into its codewords.

Prefix Codes

Prefix code is a variable length code in which no codeword is a prefix of another word.

It is possible to decode each message in an encoded string without reference to succeeding code symbols.

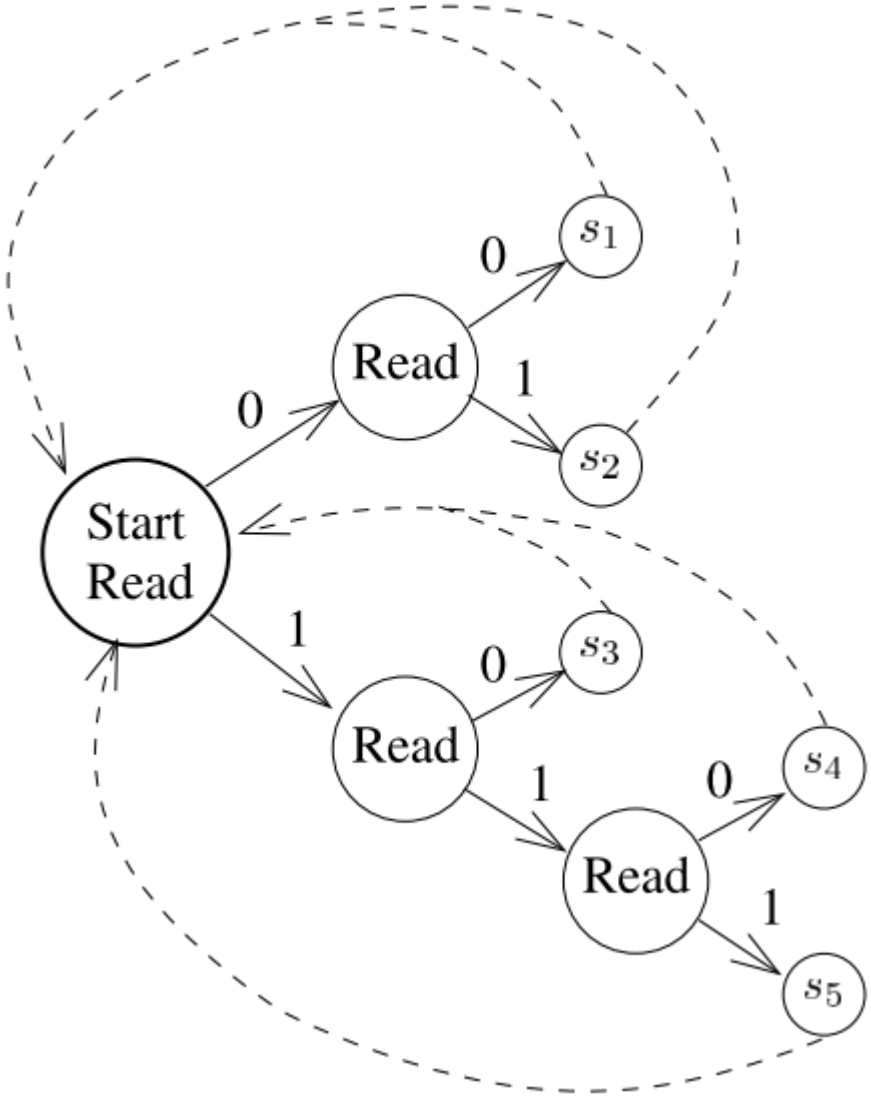
Example

Source	Code 1	Code 2
a_1	0	0
a_2	10	01
a_3	110	011
a_4	1110	0111

Code1 is Prefix code.

Decoding Prefix Codes

Source	Code
s_1	00
s_2	01
s_3	10
s_4	110
s_5	111



An example of Prefix code is Huffman code.

Shannon Fano Coding

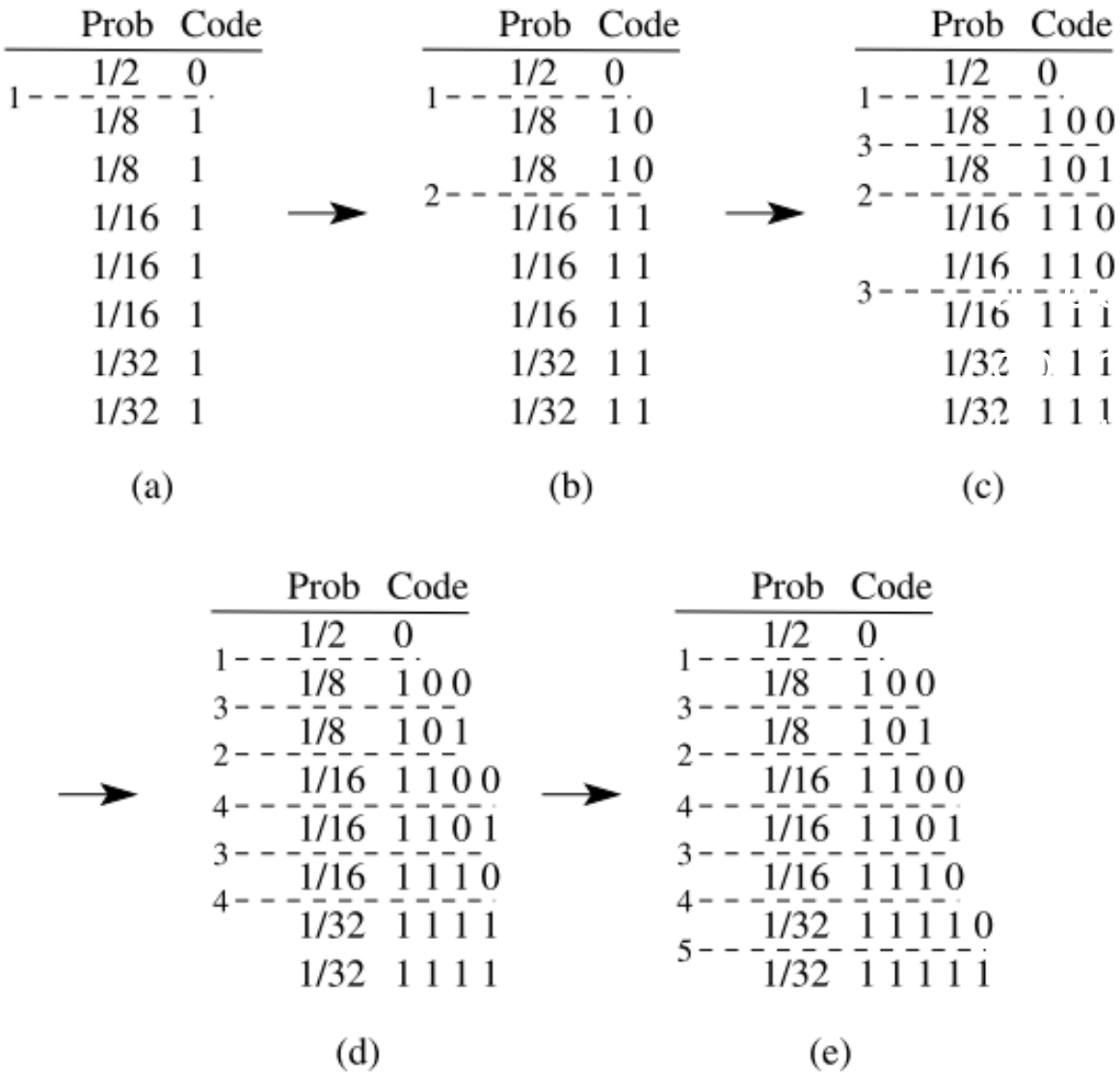
Shannon Fano Coding

Shannon Fano Coding is a top-down binary tree method.

Algorithm:

1. The set of source symbols are sorted in the order of non-increasing probabilities.
2. Source symbol is interpreted as the root of a tree.
3. The list is divided into two groups such that each group has nearly equal total probabilities.
4. The code word of the first group is appended with 0.
5. The code word of the second group is appended with 1.
6. Steps 3-5 are repeated for each group until each group contains only one symbol.

Shannon Fano Coding



Huffman Coding

Huffman Coding

The Huffman code, developed by D. Huffman in 1952, is a minimum length code.

This means that given the statistical distribution of the gray levels (the histogram), the Huffman algorithm will generate a code that is as close as possible to the minimum bound, the entropy.

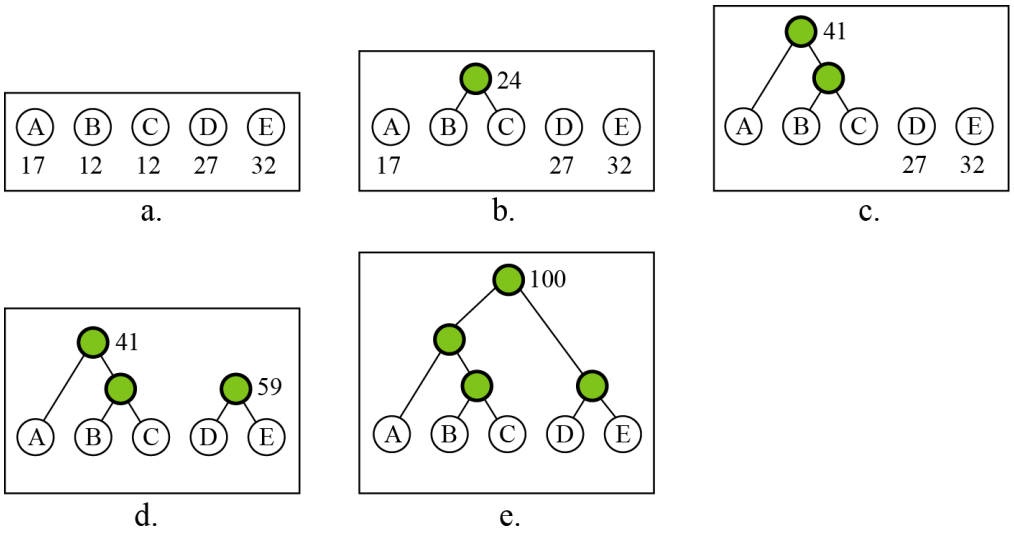
The method results in an unequal (or variable) length code, where the size of the codewords can vary.

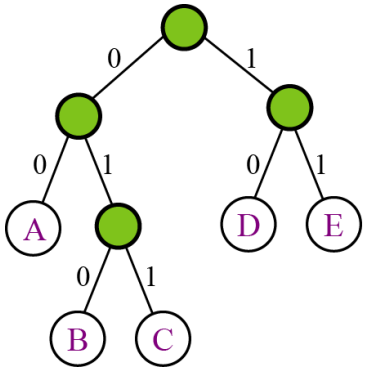
For complex images, Huffman coding alone will typically reduce the file by 10% to 50% (1.1:1 to 1.5:1), but this ratio can be improved to 2:1 or 3:1 by preprocessing for irrelevant information removal.

The Binary Huffman algorithm

1. Find the gray level probabilities for the image by finding the histogram.
2. Order the input probabilities (histogram magnitudes) from smallest to largest.
3. Combine the smallest two by addition. We reorder the probabilities, if necessary.
4. Repeats Step 2-3, until only two probabilities are left.
5. By working backward along the tree, generate code by alternating assignment of 0 and 1.

Binary Huffman Coding



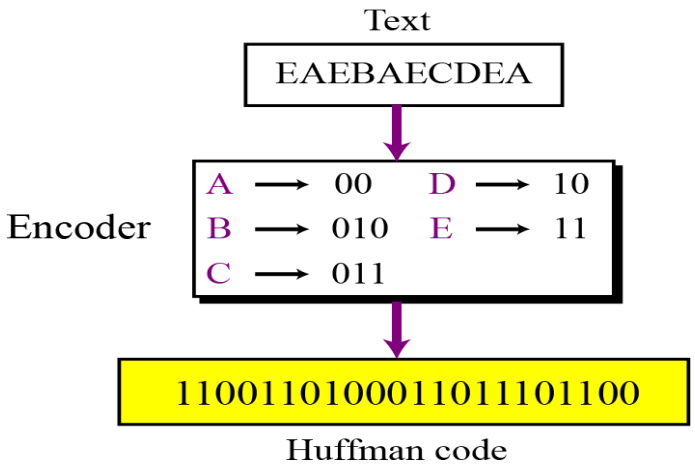


A: 00	D: 10
B: 010	E: 11
C: 011	

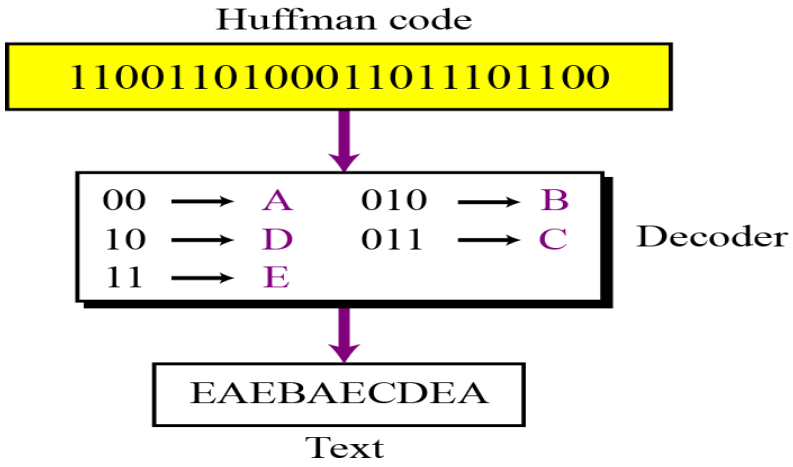
Code

Binary Huffman Coding

Encoding



Decoding



EXAMPLE 10.2.3:

We have an image with 2-bits per pixel, giving four possible gray levels. The image is 10 rows by 10 columns.

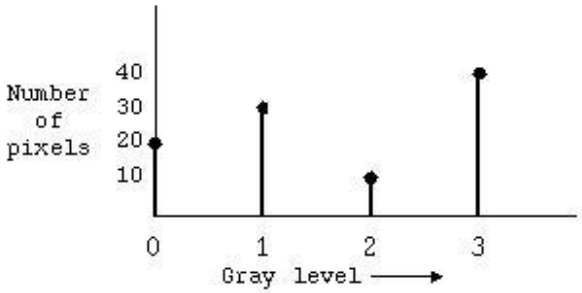
In Step 1 we find the histogram for the image. This is shown in Figure 10.2-2a, where we see that gray level 0 has 20 pixels, gray level 1 has 30 pixels, gray level 2 has 10 pixels, and gray level 3 has 40 pixels with the value. These are converted into probabilities by normalizing to the total number of pixels in the image.

Next, in step 2, the probabilities are ordered as in Figure 10.2-2b.

For step 3, we combine the smallest two by addition.

Step 4 repeats steps 2 and 3, where we reorder (if necessary) and add the two smallest probabilities as in Figure 10.2-2d. This step is repeated until only two values remain.

Figure 10.2-2: Huffman Coding Example



$g_0 = \frac{20}{100} = 0.2$

$g_1 = \frac{30}{100} = 0.3$

$g_2 = \frac{10}{100} = 0.1$

$g_3 = \frac{40}{100} = 0.4$

a) Step 1: Histogram

$g_3 \rightarrow 0.4$

$g_1 \rightarrow 0.3$

$g_0 \rightarrow 0.2$

$g_2 \rightarrow 0.1$

b) Step 2: Order

$0.4 \rightarrow 0.4$

$0.3 \rightarrow 0.3$

$0.2 \rightarrow 0.3$

0.1

c) Step 3: Add

$0.4 \rightarrow 0.4 \rightarrow 0.4$

$0.3 \rightarrow 0.3 \rightarrow 0.6$

$0.2 \rightarrow 0.3$

0.1

$0.4 \rightarrow 0.4 \rightarrow 0.6$

$0.3 \rightarrow 0.3 \rightarrow 0.4$

$0.2 \rightarrow 0.3$

0.1

d) Step 4: Reorder and add until only two values remain

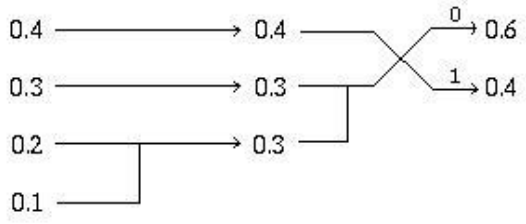
EXAMPLE 10.2.3 (Contd):

Since we have only two left in our example, we can continue to step 5 where the actual code assignment is made. The code assignment is shown in Figure 10.2-3. We start on the right-hand side of this tree and assign 0's and 1's, working our way back to the original probabilities. Figure 10.2-3a shows the first assignment of 0 and 1. A 0 is assigned to the 0.6 branch, and a 1 to the 0.4 branch. In Figure 10.2-3b, the assigned 0 and 1 are brought back along the tree, and wherever a branch occurs the code is put on both branches.

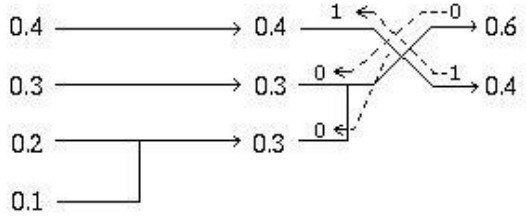
Now (Figure 10.2-3c), we assign the 0 and 1 to the branches labeled 0.3, appending to the existing code. Finally (Figure 10.2-3d), the codes are brought back one more level, and where the branch splits another assignment of 0 and 1 occurs (at the 0.1 and 0.2 branch).

Now we have the Huffman code for this image in Table 10-1.

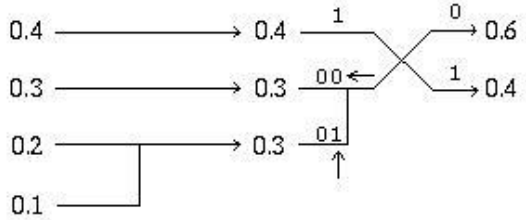
Figure 10.2-3: Huffman Coding Example, Step 5



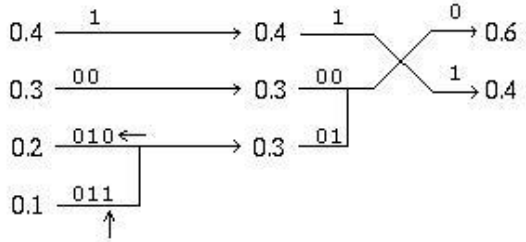
a) Assign 0 and 1 to the right-most probabilities



b) Bring 0 and 1 back along the tree



c) Append 0 and 1 to previously-added branches



d) Repeat the process until the original branch is labeled

TABLE 10-1

Original Gray Level (Natural Code)	**Probability**	**Huffman code**
g_0: 00_2	0.2	010_2
g_1: 01_2	0.3	00_2
g_2: 10_2	0.1	011_2
g_3: 11_2	0.4	1_2

EXAMPLE 10.2.4:

$$Entropy = -\sum_{i=0}^{3} p_i \log_2(p_i)$$

$$= -[(0.2)\log_2(0.2) + (0.3)\log_2(0.3) + (0.1)\log_2(0.1) + (0.4)\log_2(0.4)$$

$$\approx 1.846 \text{ bits/pixel}$$

(*Note* : $\log_2(x)$ *can be found by taking* $\log_{10}(x)$ *and multiplying by 3.322*)

$$L_{ave} = \sum_{i=0}^{L-1} l_i \, p_i$$

$$= 3(0.2) + 2(0.3) + 3(0.1) + 1(0.4)$$

$$= 1.9 \text{ bits/pixel} \quad (\text{Average length with Huffman code})$$

$$Huffman\ Code\ Efficiency = \frac{Entropy}{L_{ave}} = \frac{1.846}{1.9} = 0.9716 = 97.16\%$$

- In the example, we observe a 2.0 : 1.9 compression, which is about a 1.05 compression ratio, providing about 5% compression.

- Huffman code is highly dependent on the histogram, so any preprocessing to simplify the histogram will help improve the compression ratio.

Arithmetic Coding

Arithmetic Coding

We have just seen Huffman coding scheme assigns a code to each symbol individually so that on an average it approaches optimum coding.

For different symbols it may use codes of different lengths and to a given symbol it always assigns a fixed code irrespective of its context.

Arithmetic Coding

So, the length of each codeword is integer with minimum length one, which leads to a larger size of compressed image that it should be.

When symbols are coded independently based on the statistics of their occurrence, It may not produce least length code for a data stream altogether.

Arithmetic Coding

Suppose a data stream has two symbols ‘a’ and ‘b’ with probabilities 0.90 and 0.10, respectively.

So, entropy (H) of the data stream is 0.47.

But, the average code length due to Huffman coding is 1, which is more than double the actual requirement.

Arithmetic Coding

Another problem with Huffman coding is its non-adaptive nature.

One has to scan the data first to collect statistics in order to form the codeword table.

Then actual coding of the symbols is done based on this table, which remains unchanged throughout the coding.

Arithmetic Coding

Arithmetic coding transforms input data into a single floating point number between 0 and 1.

There is not a direct correspondence between the code and the individual pixel values.

No need to keep codebook like Huffman code.

Need only statistics of the source.

As the images are very large and the precision of digital computers is finite, the entire image must be divided into small subimages to be encoded.

Arithmetic coding uses the probability distribution of the data (histogram), so it can theoretically achieve the maximum compression specified by the entropy.

It works by successively subdividing the interval between 0 and 1, based on the placement of the current pixel value in the probability distribution.

Arithmetic Coding (Example)

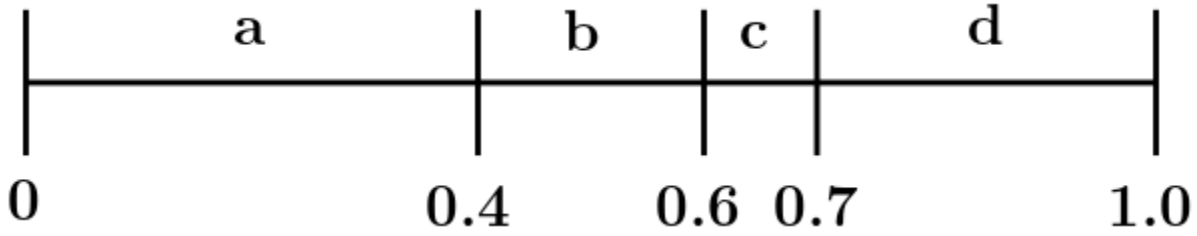
Prob: A source emits four symbols {a,b,c,d} with probabilities 0.4, 0.2, 0.1, and 0.3 respectively. Construct arithmetic coding to encode and decode the word “dad”.

Soln:

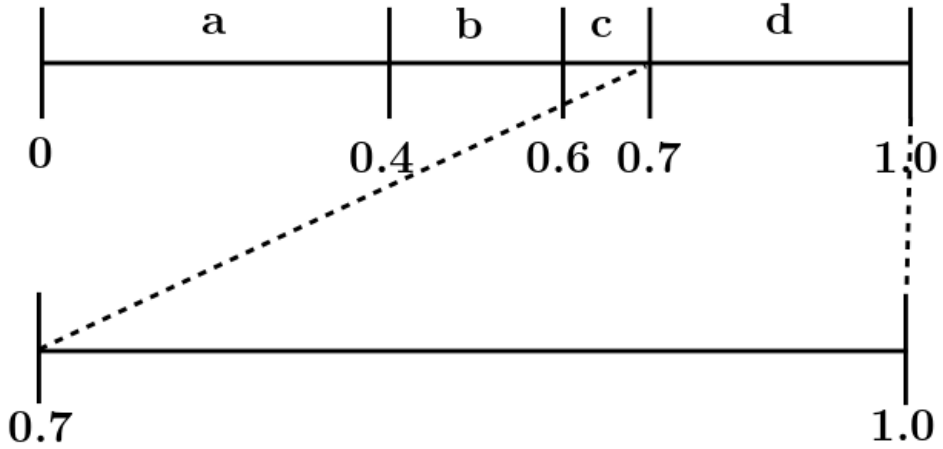
Symbol	a	b	c	d
Probability	0.4	0.2	0.1	0.3
Sub-range	(0.0-0.4)	(0.4-0.6)	(0.6-0.7)	(0.7-1.0)

Arithmetic Coding (Encoder)

Step 1:



Step 2a: The first symbol to be transmitted is d. Hence, the new range is from 0.7 to 1.0



Arithmetic Coding (Encoder)

Step 2b: Find the sub-range for each symbol in between the intervals 0.7 to 1.0.

$$low = low + range \times (low\_range)$$
$$high = low + range \times (high\_range)$$

i. Interval for symbol 'a'

$$low = 0.7 + (1.0 - 0.7) \times 0 = 0.7$$
$$high = 0.7 + (1.0 - 0.7) \times 0.4 = 0.82$$

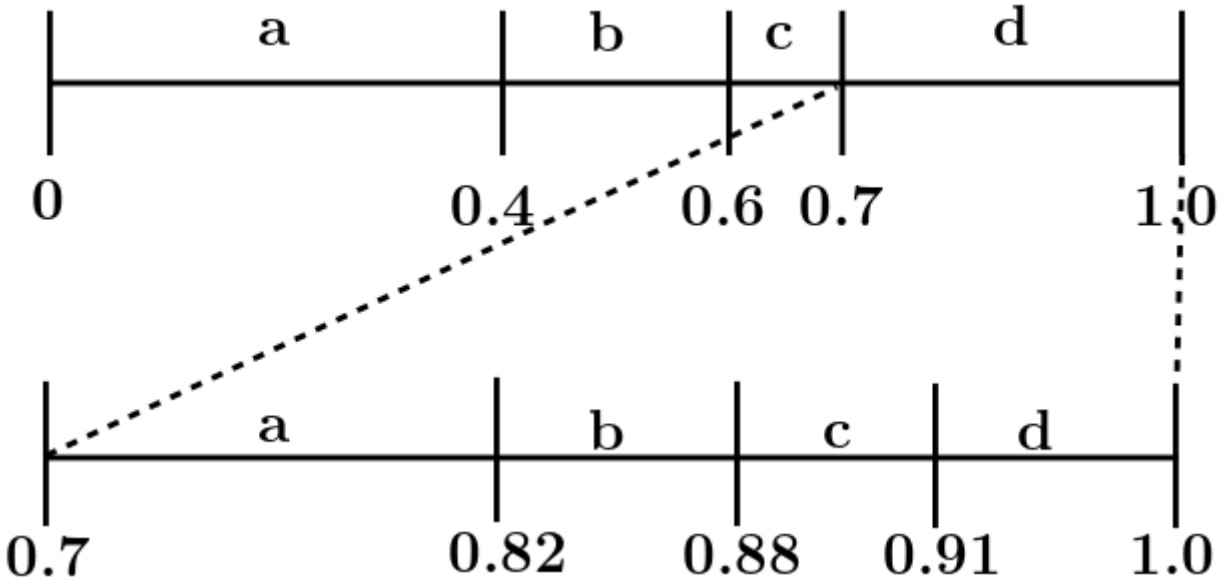
ii. Interval for symbol 'b'

$$low = 0.7 + (1.0 - 0.7) \times 0.4 = 0.82$$
$$high = 0.7 + (1.0 - 0.7) \times 0.6 = 0.88$$

iii. Interval for symbol 'c'

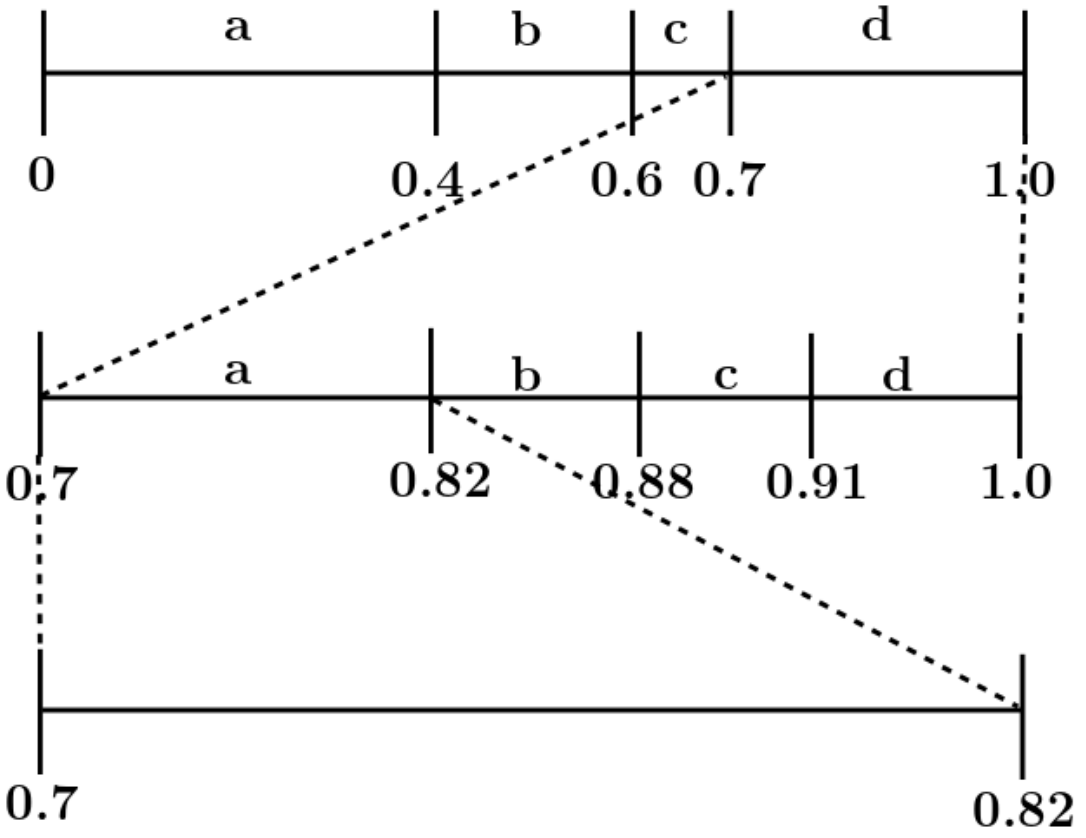
$$low = 0.7 + (1.0 - 0.7) \times 0.6 = 0.88$$
$$high = 0.7 + (1.0 - 0.7) \times 0.7 = 0.91$$

Arithmetic Coding (Encoder)



Arithmetic Coding (Encoder)

Step 3: The symbol to be transmitted is ‘a’.



Arithmetic Coding (Encoder)

Step 3a: Interval for ‘a’.

$$low = 0.7 + (0.82 - 0.7) \times 0.0 = 0.7$$
$$high = 0.7 + (0.82 - 0.7) \times 0.4 = 0.748$$

Step 3b: Interval for ‘b’.

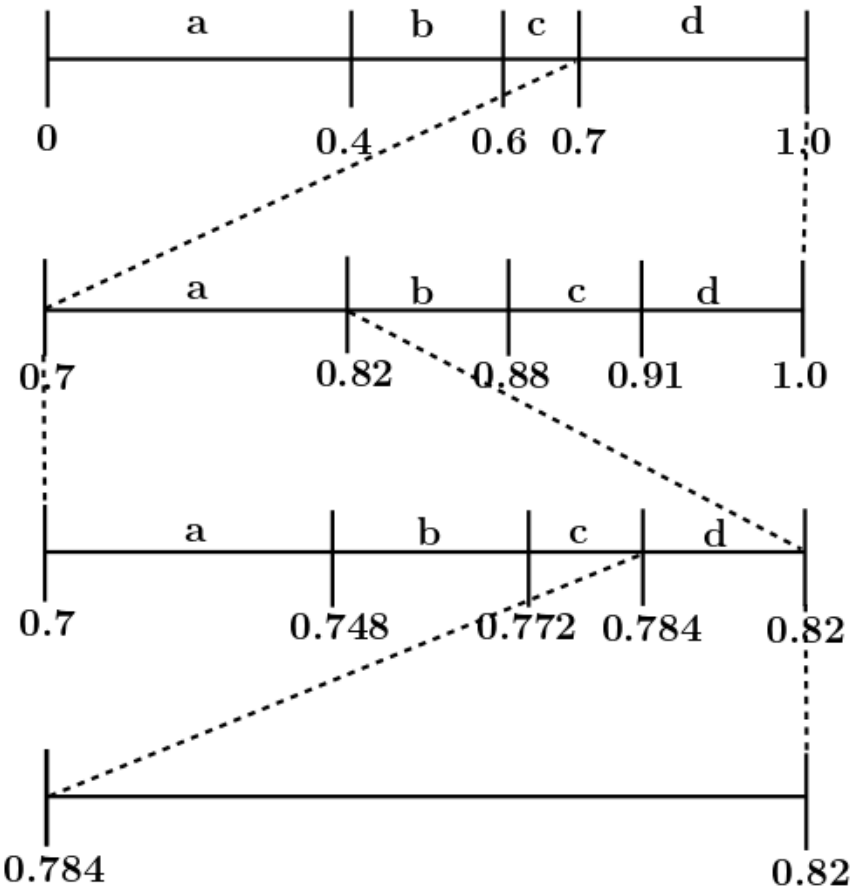
$$low = 0.748 \; and \; high = 0.772$$

Step 3c: Interval for ‘c’.

$$low = 0.772 \; and \; high = 0.784$$

Arithmetic Coding (Encoder)

Step 4: The symbol to be transmitted is ‘d’.

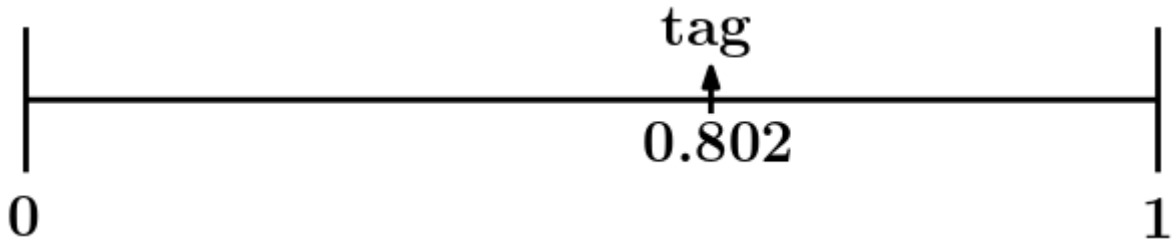


Arithmetic Coding (Encoder)

Step 5: tag value = $\frac{0.784+0.82}{2}$ = 0.802

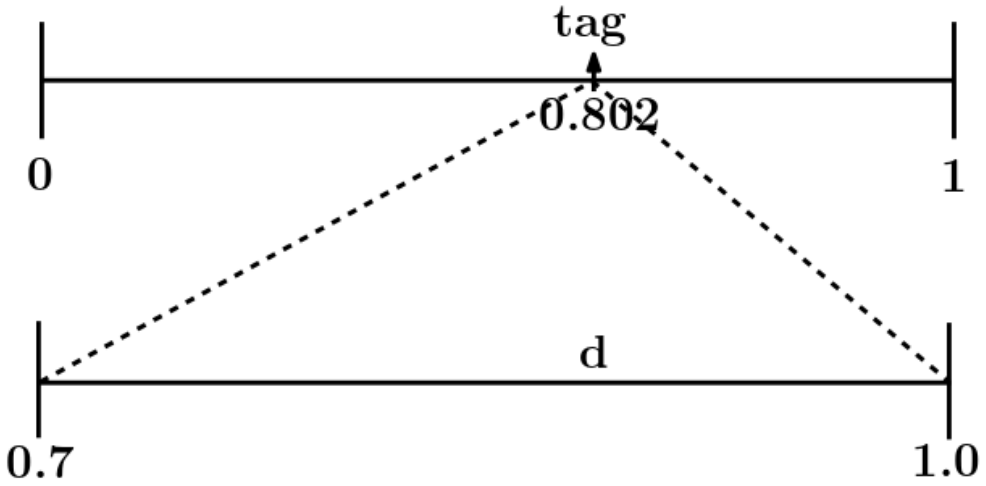
Arithmetic Coding (Decoder)

Step 1: The tag received by the receiver is 0.802



- The tag value is compared with the symbol sub-range. It is in between 0.7 and 1.0. The corresponding decoded symbol is d.

Step 2:

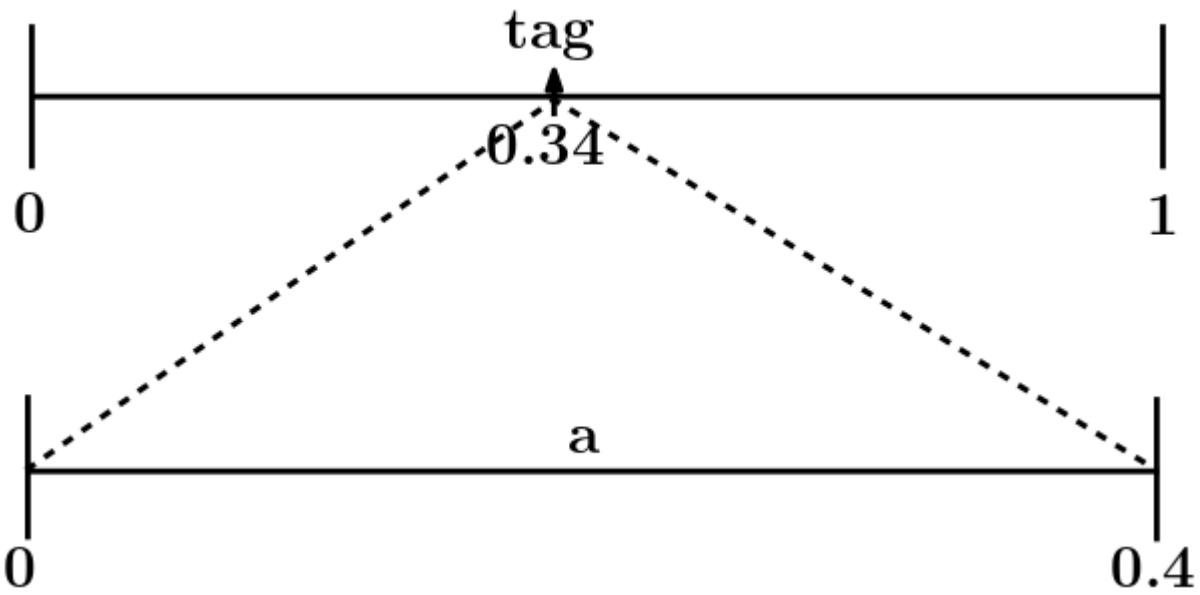


Arithmetic Coding (Decoder)

Step 3: The new tag value-

$$t^* = \frac{tag-low}{range} = \frac{0.802-0.7}{0.3} = 0.34$$

Step 4:

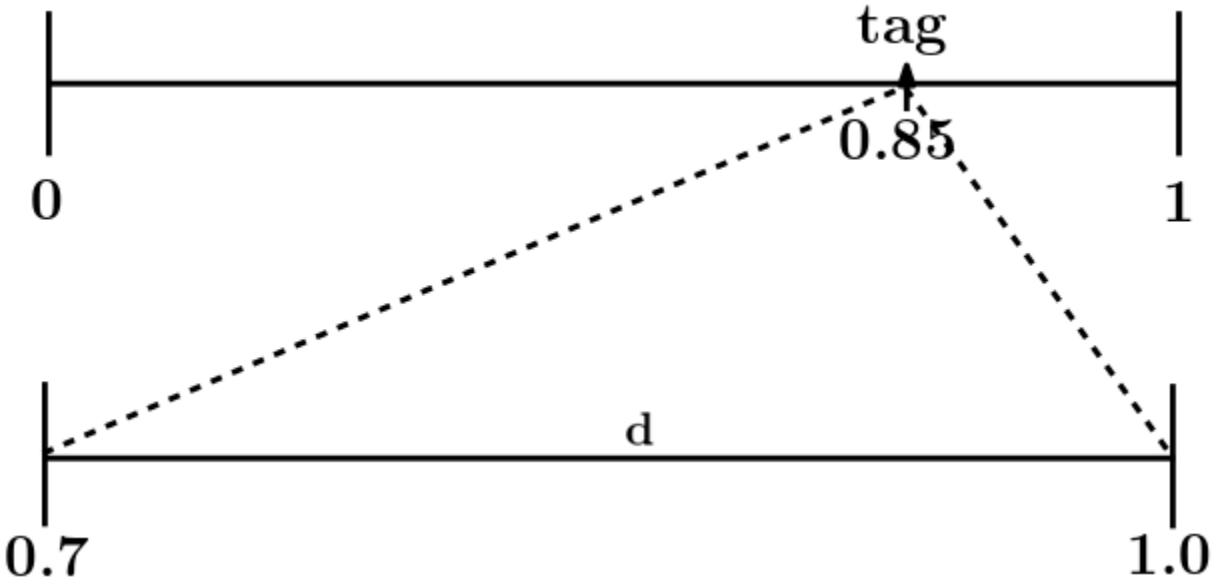


Arithmetic Coding (Decoder)

Step 5: The new tag value-

$$t^* = \frac{tag-low}{range} = \frac{0.34-0}{0.4} = 0.85$$

It is in between 0.7 and 1.0. Decoded symbol is d.



Disadvantage of Arithmetic Coding

The main disadvantage is that it tends to be slow. We have seen that the full precision form of arithmetic coding requires at least one multiplication/division per event and in some implementations up to two multiplications and two divisions per event.

In practice, this technique may be used as part of an image compression scheme, but is impractical to use alone.

It is one of the options available in the JPEG standard.