Multimedia Computing

Fundamentals of Data Compression

10101110110000

Motivation

- **Text**
  - 1 page with 80 characters/line and 64 lines/page and 1 byte/char results in 80 * 64 * 1 * 8 = **40kbit/page**
- **Still image**
  - 24 bits/pixel, 512 x 512 pixel/image results in 512 x 512 x 24 = **6M bit/image**
- **Audio**
  - CD quality, sampling rate 44.1 KHz, 16 bits per sample results in 44.1 x 16 = **706 kbit/s** (If stereo: **1.412 Mbit/s**)
- **Video**
  - Full-size frame 1024 x 768 pixel/frame, 24 bits/pixel, 30 frames/s results in 1024 x 768 x 24 x 30 = **566 Mbit/s.**
  - More realistic: 360 x 240 pixel/frame, 360 x 240 x 24 x 30 = **60 Mbit/s**
- **Multimedia streams must be compressed for storage and transmission.**

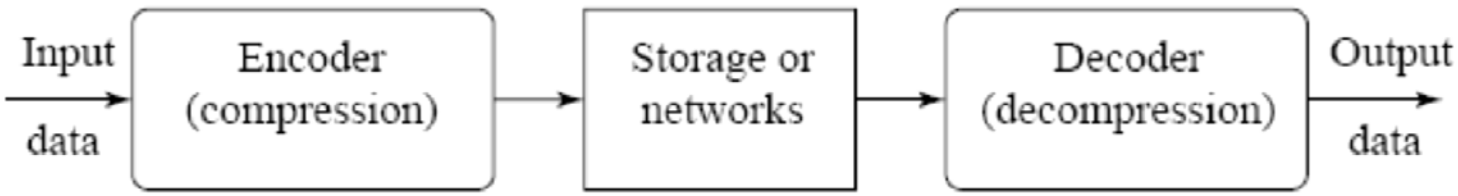
Binary & Decimal Representation

- $15 = 1 \times 10^1 + 5 \times 10^0$
  - Thus the decimal representation of 15 is 15.
- $15 = 8 + 4 + 2 + 1$

  $= 1 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$
  - Thus the binary representation of 15 is 1111.
- Questions
  - What is the binary value of 19?
  - What is the decimal value of 10011?

Compression and Decompression

- **Compression:** the process of coding that will effectively reduce the total number of bits needed to represent certain information.



A General Data Compression Scheme.

Compression Ratio

- If the compression and decompression processes induce no information loss, then the compression scheme is **lossless**; otherwise, it is **lossy**.
- **Compression ratio:**

$$C_R = B_0/B_1$$

B_0: number of bits before compression

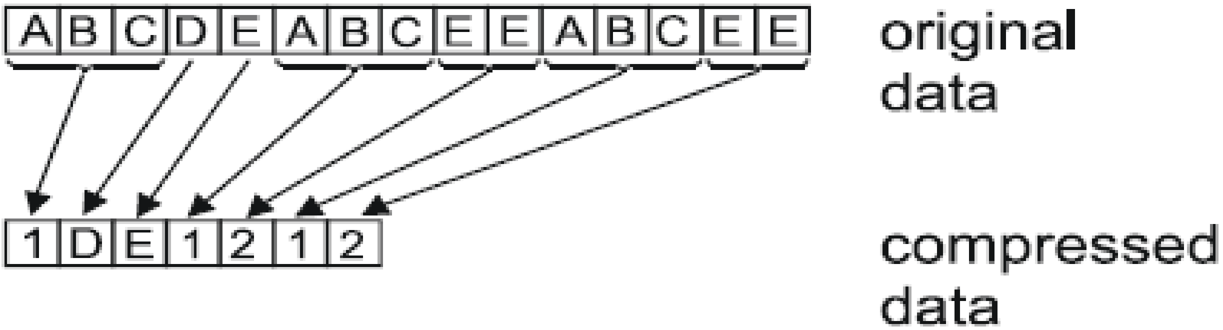
B_1: number of bits after compression

Lossless and Lossy Compression

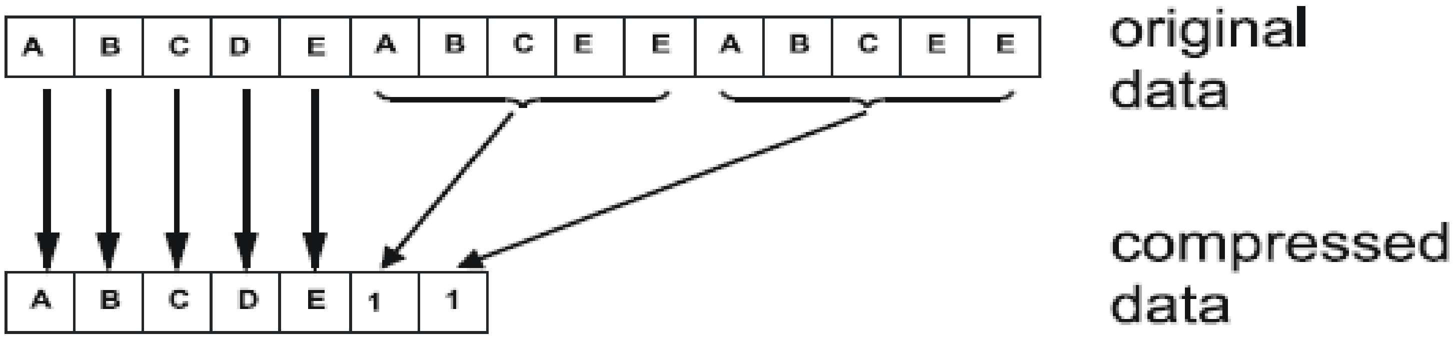
- **Lossless Compression**
  - The original data can be reconstructed perfectly from the compressed data.
  - Compression ratio usually ranges from 2:1 to 50:1.
  - Example technique: Huffman coding
- **Lossy Compression**
  - There is a difference between the original data and the reconstructed data after compression.
  - Higher compression ratios are possible than with lossless compression (typically up to 100:1).
  - Example: JPEG, JPEG2000, MP3, MPEG2, H.264

Simple Lossless Compression Examples

- Example 1: ABC→1; EE→2



- Example 2: ABCEE→1



Outline

- Basics of Information Theory
- Run-Length Coding
- Variable-Length Coding
- Arithmetic Coding

Probability Distribution Function (PDF)

- The PDF for a discrete random variable is a function that assigns a probability to each value of the random variable.
- In discrete case, the PDF value p_i of a variable S is the frequency of $S=i$, i.e. $p_i = prob(S=i)$.
- The PDF of a discrete image is usually called histogram.

PDF: an example

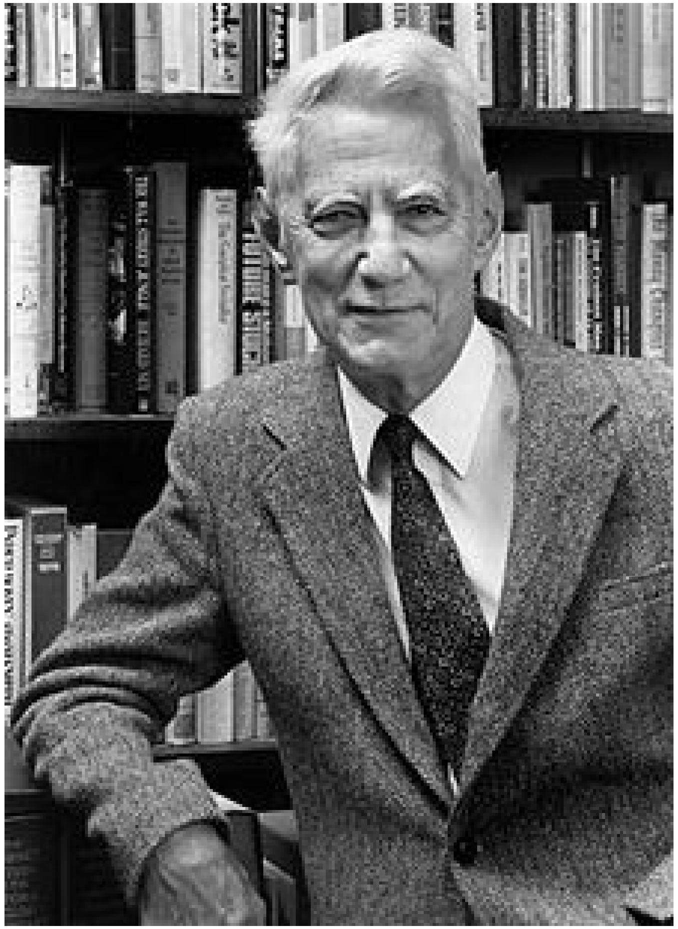
$$S = [1, 2, 1, 4, 3, 2, 3, 1]$$

$$p_1 = prob(S = 1) = 3/8;$$
$$p_2 = prob(S = 2) = 2/8;$$
$$p_3 = prob(S = 3) = 3/8;$$
$$p_4 = prob(S = 4) = 1/8;$$

i	1	2	3	4
p_i	3/8	2/8	2/8	1/8

The father of information theory

- **Claude Elwood Shannon** (April 30, 1916 – February 24, 2001), an American electrical engineer and mathematician, has been called "the father of information theory".



Entropy

- The *entropy* of an information *source* with alphabet $S = \{s_1, s_2, \ldots, s_n\}$ is:

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

- p_i – probability that symbol s_i will occur in S.
- $\log_2(1/p_i)$ – indicates the amount of information (*self-information* as defined by Shannon) contained in s_i, which corresponds to the number of bits needed to encode s_i.

Example

$$S = [1, 2, 1, 4, 3, 2, 3, 1]$$

i	1	2	3	4
p_i	3/8	2/8	2/8	1/8

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

$$= -\left(\frac{3}{8}\log_2\frac{3}{8} + \frac{2}{8}\log_2\frac{2}{8} + \frac{2}{8}\log_2\frac{2}{8} + \frac{1}{8}\log_2\frac{1}{8} \right) = 1.9056$$

The result means that we can use 1.9bit (in practice 2bits) in average to encode the symbols in S.

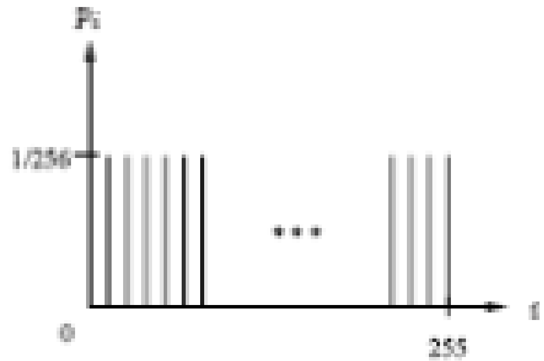
Entropy and Code Length

- It can be seen that the entropy is a weighted-sum of terms $Log_2(1/p_i)$; hence it represents the *average* amount of information contained per symbol in the source *S*.
- The entropy specifies the Lower Bound for the average number of bits to code each symbol in *S*, i.e.,

$$\eta \leq len$$

len - the average length (measured in bits) of the codewords produced by the encoder.

Another example



- Suppose the histogram of an image is with *uniform* PDF of gray-level intensities, i.e., $p_i=1/256$ for all i. Hence, the entropy of this image is:

$$\eta = -\sum_{i=1}^{n} p_i \log_2 p_i = -256 \times \frac{1}{256} \log \frac{1}{256} = 8$$

This implies that this image needs to be encoded with at least 8bits. It cannot be further (losslessly) compressed.

Outline

- Basics of Information Theory
- Run-Length Coding
- Variable-Length Coding
- Arithmetic Coding

Run Length Coding (RLC)

- **Principle:** Replace all repetitions of the same symbol in the text ("runs") by a repetition counter and the symbol.
- Example

  Text:

  AAAABBBAABBBBBCCCCCCCCDABCBAABBBBCCD

  Encoding:

  4A3B2A5B8C1D1A1B1C1B2A4B2C1D
- As we can see, we can only expect a good compression rate when long runs occur frequently.

RLC for Binary Files

- When dealing with binary files where a run of “1”s is always followed by a run of “0”s and vice versa, RLC is efficient for compression.

```
000000000000000000000000000011111111111111000000000  28 14 9
000000000000000000000000001111111111111111110000000  26 18 7
000000000000000000000001111111111111111111111110000  23 24 4
000000000000000000000011111111111111111111111111000  22 26 3
000000000000000000001111111111111111111111111111110  20 30 1
000000000000000000011111110000000000000000001111111  19 7 18 7
000000000000000000011111000000000000000000000011111  19 5 22 5
000000000000000000011100000000000000000000000000111  19 3 26 3
000000000000000000011100000000000000000000000000111  19 3 26 3
000000000000000000011100000000000000000000000000111  19 3 26 3
000000000000000000011100000000000000000000000000111  19 3 26 3
000000000000000000001111000000000000000000000001110  20 4 23 3 1
000000000000000000000011100000000000000000000111000  22 3 20 3 3
011111111111111111111111111111111111111111111111111  1 50
011111111111111111111111111111111111111111111111111  1 50
011111111111111111111111111111111111111111111111111  1 50
011111111111111111111111111111111111111111111111111  1 50
011111111111111111111111111111111111111111111111111  1 50
011000000000000000000000000000000000000000000000011  1 2 46 2
```

Outline

- Basics of Information Theory
- Run-Length Coding
- Variable-Length Coding
- Arithmetic Coding

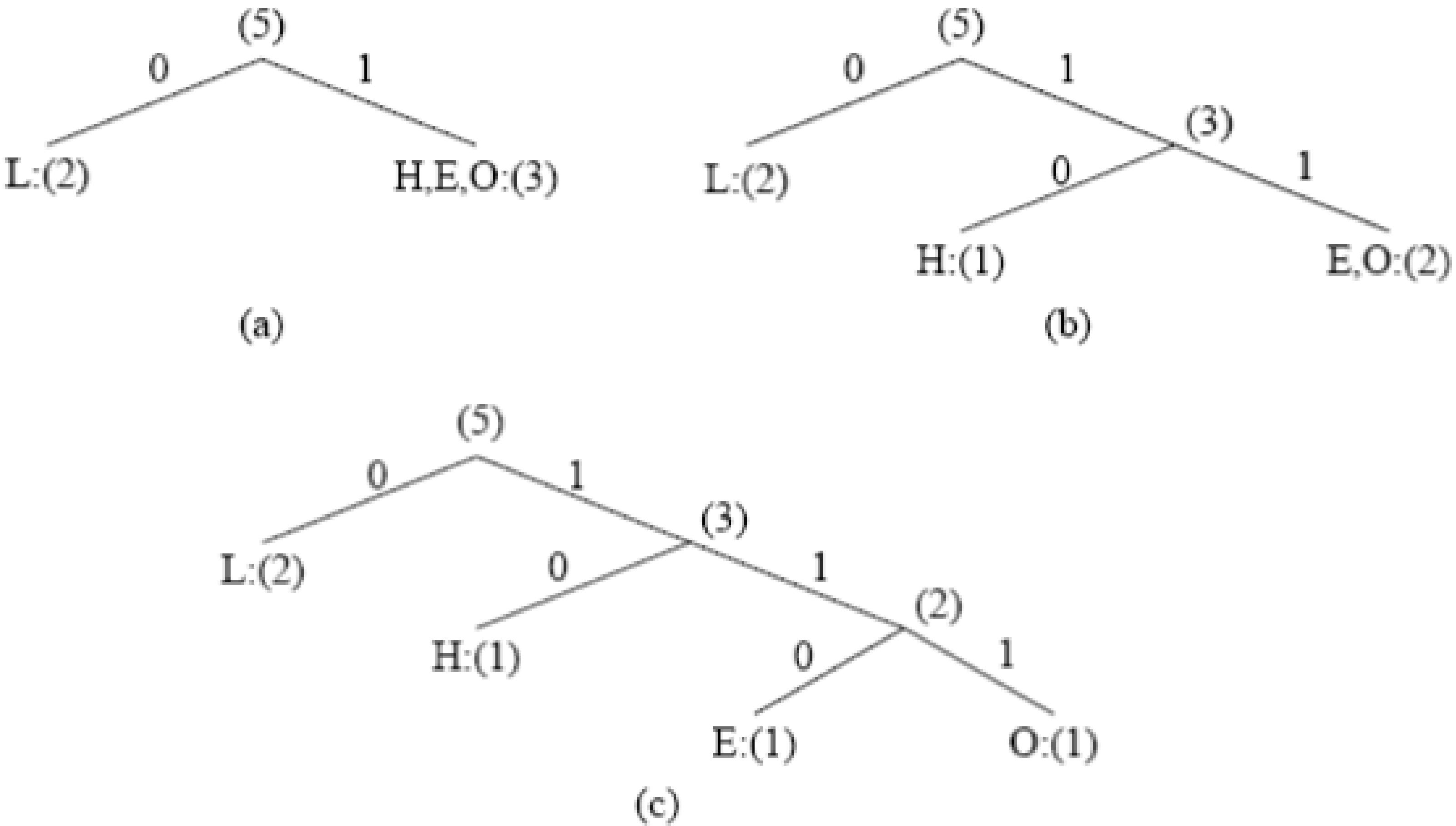
Variable-Length Coding (VLC)

- **Shannon-Fano Algorithm** -- a top-down approach
  1. Sort the symbols according to the frequency count of their occurrences.
  2. Recursively divide the symbols into two parts, each with approximately the same number of counts, until all parts contain only one symbol.
- Example: coding of "HELLO"

Symbol	H	E	L	O
Count	1	1	2	1

Frequency count of the symbols in "HELLO"

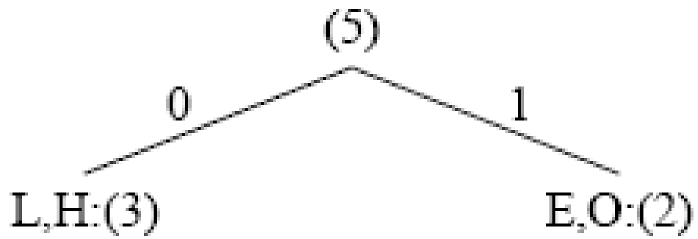
One Coding Tree for HELLO



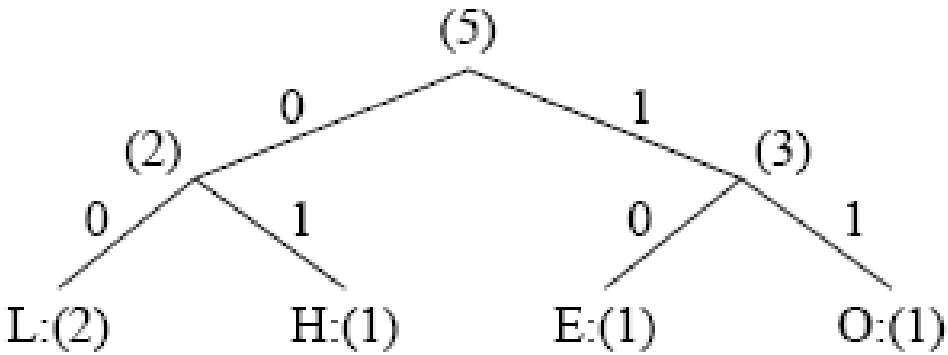
Result of Performing Shannon-Fano on HELLO

Symbol	Count	$\log_2 \frac{1}{p_i}$	Code	# of bits used
L	2	1.32	0	2
H	1	2.32	10	2
E	1	2.32	110	3
O	1	2.32	111	3
TOTAL number of bits:				10

Another coding tree for HELLO



(a)



(b)

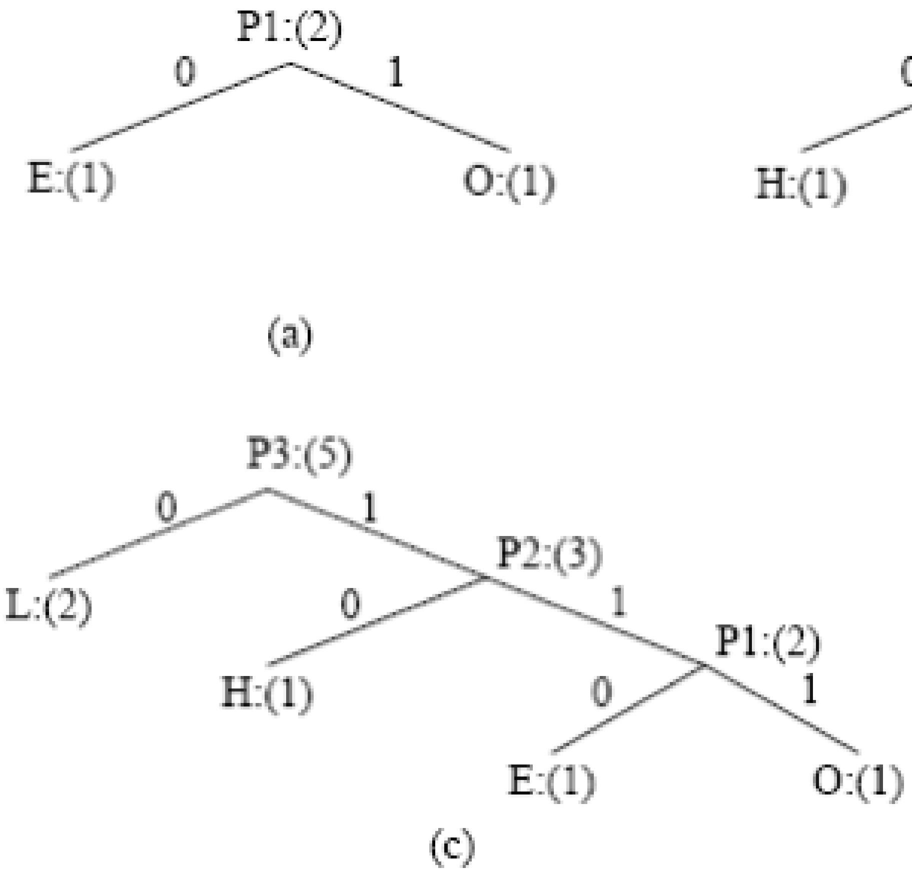
Another Result of Performing Shannon-Fano on HELLO

Symbol	Count	$\log_2 \frac{1}{p_i}$	Code	# of bits used
L	2	1.32	00	4
H	1	2.32	01	2
E	1	2.32	10	2
O	1	2.32	11	2
TOTAL number of bits:				10

Huffman Coding

- Huffman Coding Algorithm – a bottom-up approach
  1. Initialization: Put all symbols on a list sorted according to their frequency counts.
  2. Repeat until the list has only one symbol left:
     i. From the list pick two symbols with the lowest frequency counts. Form a Huffman sub-tree that has these two symbols as child nodes and create a parent node.
     ii. Assign the sum of the children's frequency counts to the parent and insert it into the list such that the order is maintained.
     iii. Delete the children from the list.
  3. Assign a codeword for each leaf based on the path from the root.

Coding Tree for "HELLO" using the Huffman Algorithm



Code-words:

L:	0
H:	10
E:	110
O:	111

Properties of Huffman Coding

- **Unique** Prefix Property: No Huffman code is a prefix of any other Huffman code - precludes any ambiguity in decoding.
- **Optimality**: *minimum redundancy code* - proved *optimal* for a given data model (i.e., a given, accurate, probability distribution):
  - The two least frequent symbols will have the same length for their Huffman codes, differing only at the last bit.
  - Symbols that occur more frequently will have shorter Huffman codes than symbols that occur less frequently.
  - The average code length for an information source *S* is strictly less than $\eta + 1$. We have:

$$\eta \le len < \eta + 1$$

Outline

- Basics of Information Theory
- Run-Length Coding
- Variable-Length Coding
- Arithmetic Coding

Arithmetic Coding

- Arithmetic coding is a widely used entropy coder, such as in JPEG, etc.
- It has good compression ratio (better than Huffman coding), approaching to the lower bound of entropy coding. Only problem is it's computational cost due to large symbol tables.
- Why it is better than Huffman coding?
  - Huffman coding use an integer number (k) of bits for each symbol, and k is never less than 1.

Idea of Arithmetic Coding

- To have a probability line, between 0 – 1, and assign to every symbol a range in this line based on its probability.
- The higher the probability, the higher range which assigns to it.
- Once we have defined the ranges and the probability line, start to encode symbols.
- Every symbol defines where the output floating point number lands within the range.

An example

- Assume we have the following token symbol stream BACA
  - A occurs with **probability 0.5**
  - B and C occur with **probabilities 0.25**.

An example (cont.)

- Start by assigning each symbol to the probability range 0 –1. Sort symbols with the highest probability first

Symbol	Range
A	[0.0, 0.5)
B	[0.5, 0.75)
C	[0.75,1.0)

- The first symbol in the example stream is B. Thus we know that the code will be in the range 0.5 to 0.74999 . . ..

An example (cont.)

- We need to narrow down the range to obtain a code of the input stream.
- **The arithmetic coding iteration:** Subdivide the range for the first token given the probabilities of the second token then the third, etc.

For all the symbols:

range = high - low;

high = low + range * high_range of the symbol being coded;

low = low + range * low_range of the symbol being coded;

where:

range keeps track of where the next range should be.

high and low specify the output number.

Initially high = 1.0, low = 0.0

An example (cont.)

Symbol	Range
A	[0.0, 0.5)
B	[0.5, 0.75)
C	[0.75,1.0)

- The range of first symbol B is [0.5, 0.75), we know low=0.5, high=0.75 and range=high-low=0.25.
- The next symbol being coded is A. Thus high_range=0.5 and low_range=0.0. Now:

  high=0.5+0.25*0.5=0.625

  low=0.5+0.25*0.0=0.5
- After the iteration for the second symbol A, we have the range of stream BA is [0.5, 0.625).

An example (cont.)

Symbol	Range
A	[0.0, 0.5)
B	[0.5, 0.75)
C	[0.75,1.0)

- The range stream BA is [0.5, 0.625). we know low=0.5, high=0.625 and range=0.125.
- The next symbol being coded is C. Thus high_range=1.0 and low_range=0.75. Now:

  high=0.5+0.125*1.0=0.625

  low=0.5+0.125*0.75=0.59375
- After the iteration for the third symbol C, we have the range of stream BAC is [0.59375, 0.625).

An example (cont.)

Symbol	Range
A	[0.0, 0.5)
B	[0.5, 0.75)
C	[0.75,1.0)

- The range stream BAC is [0.59375, 0.625). we know low=0.59375, high=0.625 and range=0.03125.
- The next symbol being coded is A. Thus high_range=0.5 and low_range=0.0. Now:

  high=0.59375+0.03125*0.5=0.609375

  low=0.59375+0.03125*0.0=0.59375
- After the iteration for the last symbol A, we have the range of stream BACA is [0.59375, 0.609375).

Arithmetic Coding: Encoder

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

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

   output a code so that low <= code < high;
END
```

Note that usually we put a terminator symbol “$” at the end of the stream.

Binary codeword generation

- Now the output code for stream BACA is any number in the range [0.59375, 0.609375).
- However, there is still one problem left – we need to convert the decimal number into binary format.
- Binary fractions:
  - 0.1 binary $=1*1/2^1= 0.5$ decimal
  - 0.01 binary $= 0*1/2^1 +1*1/2^2= 0.25$ decimal
  - 0.101binary $= 1*1/2^1 +0*1/2^2 +1*1/2^3= 0.625$ decimal

Generating Binary Codeword

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

         k = k + 1;
       }
END
```

"code" is a binary codeword;
value(code) means getting the decimal value of binary "code".
The above algorithm will ensure that the shortest binary codeword is found.

The binary codeword of BACA

- The range of BACA is [0.59375, 0.609375).
- If we assign 1 to the first binary fraction bit, i.e. 0.1 binary, its decimal value(code) = value(0.1) = 0.5 decimal < low = 0.59375. Then we go to the next loop.
- If we assign 1 to the second binary fraction bit, i.e. 0.11 binary, its decimal value(code) = value(0.11) = 0.75 > high = 0.609375. Then we let the second bit be 0.
- Since value(0.10) = 0.5 < low = 0.59375, we continue.
- If we assign 1 to the third binary fraction bit, i.e. 0.101 binary, its decimal value(code) = value(0.101) = 0.625 > high = 0.609375. Then we let the third bit be 0.

The binary codeword of BACA

- Since value(0.100) = 0.5 < low = 0.59375, we continue.
- If we assign 1 to the forth binary fraction bit, i.e. 0.1001 binary, its decimal value(code) = value(0.1001) = 0.5625 < low = 0.59375. We continue.
- If we assign 1 to the fifth binary fraction bit, i.e. 0.10011 binary, its decimal value(code) = value(0.10011) = 0.59375 < high but = low.
- Then we stop and get the binary codeword 0.10011.
- Exercise
  - Code BACA using Huffman coding and compare its codeword with arithmetic coding.

Decoding

```
BEGIN
   get binary code and convert to
       decimal value = value(code);
   Do
       { find a symbol s so that
              Range_low(s) <= value < Range_high(s);
         output s;
         low = Rang_low(s);
         high = Range_high(s);
         range = high - low;
         value = [value - low] / range;
       }
   Until symbol s is a terminator
END
```

The decoding process is just the opposite of the encoding.

References

- Ze-Nian Li, M. S. Drew, *Fundamentals of Multimedia*, Prentice Hall Inc., 2004. Chapter 7.