

# ECE253/CSE208 Introduction to Information Theory

## Lecture 8: Lossless Source Coding

Dr. Yu Zhang

ECE Department

University of California, Santa Cruz

- Chap 5 and 13 of *Elements of Information Theory (2nd Edition)* by Thomas Cover & Joy Thomas

# Data Compression

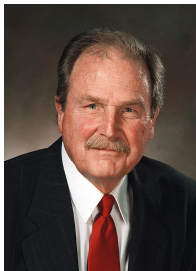
Source coding: Mapping from a sequence of symbols from an information source to a sequence of alphabet symbols (usually bits) such that the source symbols can be

- exactly recovered from the binary bits: **lossless source coding**.
- recovered within some distortion: **lossy source coding**.

We'll learn 6 lossless coding schemes:

1. Huffman coding
2. Shannon-Fano coding
3. Shannon-Fano-Elias coding
4. Shannon coding
5. Arithmetic coding
6. Lempel-Ziv coding





**Figure:** David A. Huffman (August 9, 1925 – October 7, 1999)

In 1951, David Huffman was a PhD student at MIT, where he developed the Huffman coding scheme as a term paper for Robert Fano's class. It outperforms Shannon-Fano coding. In 1967, David joined the faculty of UC Santa Cruz and helped found its CS Dept. He served as the chair from 1970–73 and retired in 1994.

Huffman coding is the work-horse of the compression industry: GZIP, PKZIP, JPEG, PNG, MPEG, etc. Newer arithmetic coding is often avoided because of patent issues.

# Huffman Coding Algorithm

- S1): Sort  $\{p_i\}$  in decreasing order, and treat each  $p_i$  as a leaf (external) node.
- S2): Combine two nodes with the smallest  $p_i$ 's to form a new (internal) node. Add the new node, whose probability is the sum of the probabilities of the two children, back to the list (we may choose to keep the decreasing order or not). Keep merging until the tree is complete (end up with the root node).
- S3): Assign 0/1 to each branch (e.g.,  $0 \mapsto$  left and  $1 \mapsto$  right). Read off the code strings *from the root to all leaf nodes*, which yields the Huffman code.

**Decoding procedure.** Starting with the first bit of the received data, the decoder uses successive bits to determine whether to go left or right in the Huffman tree (starting from the root node). When we reach a leaf node, a source symbol has been decoded.

## Huffman Coding (cont'd)

### Example

Consider 4 symbols with the probabilities:

$\{\Pr(A) = 0.5, \Pr(B) = 0.25, \Pr(C) = 0.125, \Pr(D) = 0.125\}$ .

Symbol	Code	Code length	Probability
A	0	1	0.5
B	10	2	0.25
C	110	3	0.125
D	111	3	0.125

Average code length:  $L = \sum_i p_i l_i = 1.75 = H(X)$ . This is the best we can do because of no redundancy at all. If we use a fixed length code, i.e., each symbol corresponds to a code of length 2, then the average code length is 2.

## Huffman Coding (cont'd)

### Example

Consider the following 8 symbols. We have  $L = 2.79 > 2.78 = H(X)$ .

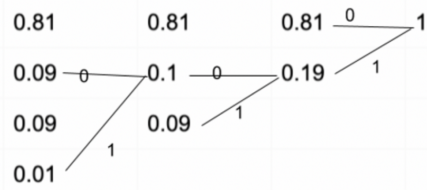
Symbol	Code	Code length	Probability
A	10	2	0.25
B	11	2	0.21
C	000	3	0.15
D	001	3	0.14
E	0100	4	0.0625
F	0101	4	0.0625
G	0110	4	0.0625
H	0111	4	0.0625

## Block Huffman Coding

### Example (Block Huffman coding reduces the average code length per symbol)

Consider two source symbols:  $\{\Pr(A) = 0.9, \Pr(B) = 0.1\}$ . Thus,  $L = 1$  and  $H = 0.47$  bits per symbol. Let's combine two symbols together given below:

Symbol	Code	Code length	Probability			
AA	0	1	0.81	0.81	0.81	0
AB	100	2	0.09	0.1	0.19	1
BA	11	3	0.09	0.09		
BB	101	3	0.01			



Average code length for two-symbol:

$$L = 1 \times 0.81 + 2 \times 0.09 + 3 \times 0.09 + 3 \times 0.01 = 1.29 \text{ bits}$$

Now average code length per symbol is  $1.29/2 = 0.645$  bits/symbol (closer to  $H$ ).

Block Huffman coding is asymptotically optimal as the block size goes to infinity.

## Dummy symbols in Huffman tree

Huffman coding is prefix-free, since a Huffman code can always be represented by a  $D$ -ary tree. If  $D = 2$ , we can always construct the Huffman tree for any number of source symbols  $N$ .

Codeword	$X$	Probability
1	1	0.25
2	2	0.25
01	3	0.2
02	4	0.1
000	5	0.1
001	6	0.1
002	Dummy	0.0

**Figure:** If  $D \geq 3$ , sometimes dummy symbols are needed to construct the  $D$ -ary tree.

- After each step of the coding algorithm (merging two nodes with smallest probabilities), we reduce the number of symbols by  $(D - 1)$ , and we are left with only one node at the end.
- Hence, we have  $k \times (D - 1) + 1 = N + E$  for some  $k \in \mathbb{Z}_+$ , where  $E$  is the number of dummy symbols added, and  $k$  is the number of merges.
- In other words, the number of dummy symbols satisfies  $N + E \equiv 1 \pmod{D - 1}$ .



# Optimality of Huffman Code

## Lemma (Properties of a particular optimal code)

*For any distribution, there exists an optimal prefix code (with minimum expected length) that satisfies the following properties:*

- *More likely symbols are assigned with shorter codes: if  $p_j > p_i \implies l_j \leq l_i$ .*
- *The two longest codewords have the same length. They correspond to the least likely symbols and differ only in the last bit.*
- *Not all optimal codes satisfy the second property, but we can find such an optimal code by rearranging.*

**Theorem (The average codelength of a Huffman code is no more than any uniquely decodable code)**

$$L_H \leq L_{UD}.$$

Huffman coding is optimal for symbol-by-symbol coding with a known input distribution.

## Optimality of Huffman Code (cont'd)

### Proof by induction:

Consider  $p_1 \geq p_2 \geq \dots \geq p_{m-1} \geq p_m$  and  $l_1 \leq l_2 \leq \dots \leq l_{m-1} = l_m$ .

- Assume the claim holds for  $C_{m-1}$ , which is the Huffman code for  $m-1$  symbols (trivial for  $m=2$ ).
- To construct  $C_m$ , we first merge  $p_m$  and  $p_{m-1}$ , which are *siblings* with the same length. The remaining steps construct a Huffman code  $C_{m-1}$  for a PMF with probabilities  $p_1, p_2, \dots, p_{m-2}, p_{m-1} + p_m$ . Hence,  
$$L(C_{m-1}) = \sum_{i=1}^{m-2} l_i p_i + (l_{m-1} - 1)(p_{m-1} + p_m) = L(C_m) - p_{m-1} - p_m.$$
- Let  $C'_m$  be an optimal code for the same distribution  $p$  and satisfy the lemma. Applying the same merging procedure, we have  $L(C'_{m-1}) = L(C'_m) - p_{m-1} - p_m$ .
- Since  $L(C_{m-1}) \leq L(C'_{m-1}) \implies L(C_m) \leq L(C'_m)$ .

## Redundancy of Huffman Code

**Redundancy.** The redundancy of a Huffman code of a PMF  $p$  satisfies:

$$0 \leq L_H - H(p) \leq 1.$$

The gap goes to 1 when  $H(p) \rightarrow 0$ .

**Q:** Typically how large is it?

Theorem (Gallager 1978')

$$L_H - H(p) \leq p_{\max} + c,$$

where  $p_{\max}$  is the probability of the most likely symbol, and  $c = \log(\frac{2}{e}) \times \log e \approx 0.086$ .

Furthermore, if  $p_{\max} \geq 0.5 \implies L_H - H(p) \leq 2 - H(p_{\max}) - p_{\max} \leq p_{\max}$ .

## Shannon-Fano (SF) Coding Algorithm

- S1): Sort the source symbols according to probability, with the most frequent symbols on the left.
- S2): Divide the list into two parts, such that the sums of probabilities in both parts are as equal as possible<sup>1</sup> (always keep the order of the symbols, NO shuffle).
- S3): Code the left part as 0, while the right part as 1. Keep doing until each symbol becomes a leaf code.
- S4): Read off the code strings from the root to all leaf nodes, which yields the SF codes.

---

<sup>1</sup>This may introduce ambiguity; see an example in

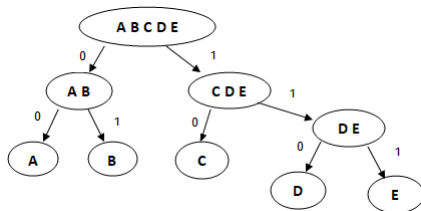
<https://stackoverflow.com/questions/71399572/is-shannon-fano-coding-ambiguous>

## Shannon-Fano (SF) Coding Algorithm (cont'd)

### Example

Consider the following source:

Symbol	A	B	C	D	E
Count	15	7	6	6	5
Probability	0.385	0.179	0.154	0.154	0.128
Huffman code	1	000	001	010	011
SF code	00	01	10	110	111



## Shannon-Fano (SF) Coding Algorithm (cont'd)

For the average code length per symbol, we get

$$L_{\text{SF}} = \frac{2 \times (15 + 7 + 6) + 3 \times (6 + 5)}{39} = 2.28 \text{ bits/sym}$$

$$L_{\text{H}} = \frac{1 \times 15 + 3 \times (7 + 6 + 6 + 5)}{39} = 2.23 \text{ bits/sym}$$

$$H(X) = 2.185 \text{ bits/sym}$$

$$H(X) < L_{\text{H}} < L_{\text{SF}}$$

**Code efficiency.** It is shown<sup>2</sup> that the expected length of Fano's method is bounded above as  $L_{\text{SF}} \leq H(X) + 1 - p_{\min}$ .

**Insight.** In terms of constructing the binary tree, Shannon-Fano coding is a *top-down* approach while Huffman coding is *bottom-up*.

---

<sup>2</sup>S. Krajči et al, "Performance analysis of Fano coding," 2015 IEEE ISIT.

## Shannon-Fano-Elias (SFE) Coding

Instead of using the PMF of the source, the SFE coding (in the class of interval coding) uses the CDF:  $F(x) = \sum_{y \leq x} p(y) = \sum_{y < x} p(y) + p(x)$ .

Define a modified CDF:  $\bar{F}(x) := \sum_{y < x} p(y) + \frac{1}{2}p(x)$ ,  $\forall x \in \mathcal{X}$ .

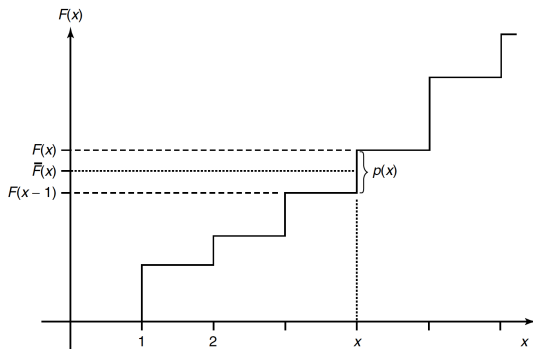


Figure: The modified CDF for SFE coding.

## SFE Coding Algorithm

Assume all source symbols have been sorted in a certain sense (e.g., lexicographic), for each  $x \in \mathcal{X}$ ,

- S1): Let  $Z(x)$  be the binary representation of  $\overline{F}(x)$  (using a decimal-to-binary converter).
- S2): Choose the code length  $l(x) = \lceil \log \frac{1}{p(x)} \rceil + 1$ .
- S3): The encode string  $C(x)$  is the first  $l(x)$  most significant bits (MSB) after the decimal point of  $Z(x)$ .



## SFE Coding Algorithm (cont'd)

### Example

Consider  $X = \{A, B, C, D\}$  with  $p_i = \{\frac{1}{3}, \frac{1}{4}, \frac{1}{6}, \frac{1}{4}\}$

- $\bar{F}(A) = \frac{1}{2}P(A) = \frac{1}{6} = (0.1666)_{10} \Rightarrow Z(A) = (0.00101010)_2$   
 $l(A) = \lceil \log \frac{1}{p(A)} \rceil + 1 = 3 \Rightarrow A \rightarrow \mathbf{001}$
- $\bar{F}(B) = P(A) + \frac{1}{2}P(B) = \frac{1}{3} + \frac{1}{8} = (0.458333)_{10} \Rightarrow Z(B) = (0.011101010)_2$   
 $l(B) = \lceil \log \frac{1}{p(B)} \rceil + 1 = 3 \Rightarrow B \rightarrow \mathbf{011}$
- $\bar{F}(C) = (0.6666)_{10} \Rightarrow Z(C) = (0.10101010)_2 \Rightarrow C \rightarrow 1010$
- $\bar{F}(D) = (0.875)_{10} \Rightarrow Z(D) = (0.111)_2 \Rightarrow D \rightarrow 111$

**SFE code efficiency.** Since  $a \leq \lceil a \rceil < a + 1$ , we shall pay extra 1 to 2 bits more than  $H(X)$  per source symbol:

$$H(X) + 1 \leq L_{SFE} = \sum_{x \in \mathcal{X}} p(x) \left( \left\lceil \log \frac{1}{p(x)} \right\rceil + 1 \right) < H(X) + 2.$$

## Decimal to Binary Conversion

$$\begin{array}{ll} 0.188 \times 2 = 0.376 & \text{carry} = 0 \\ 0.376 \times 2 = 0.752 & \text{carry} = 0 \\ 0.752 \times 2 = 1.504 & \text{carry} = 1 \\ 0.504 \times 2 = 1.008 & \text{carry} = 1 \\ 0.008 \times 2 = 0.016 & \text{carry} = 0 \end{array}$$

MSB ↓

Answer = .00110 (for five significant digits)

$$\begin{array}{r} 2 \overline{) 29} \\ 2 \overline{) 14} \\ 2 \overline{) 7} \\ 2 \overline{) 3} \\ 2 \overline{) 1} \\ 0 \end{array}$$

Remainders

1	LSB
0	
1	
1	
1	MSB

Read the remainders from the bottom up

29 decimal = 11101 binary

**Figure:** Decimal-to-Binary conversion: i) Take the fractional part and multiply it by 2. Note the digit to the left of the decimal point and repeat the process until reaching the desired precision or the fractional part becomes 0. ii) Take the integer part and divide it by 2. Note the remainder and repeat the process until the quotient is 0, then write the remainders in reverse order.

Special numbers can be done by observation, e.g.

$$(0.75)_{10} = 2^{-1} + 2^{-2} = (0.11)_2, \quad \left(\frac{1}{3}\right)_{10} = \sum_{i=1}^{\infty} 2^{-2i} = (0.01010101...)_{2} \triangleq (0.\overline{01})_2$$

## SFE Coding Algorithm (cont'd)

### Lemma

*SFE code is uniquely decodable: With codeword  $\lfloor \overline{F}(x) \rfloor_{l(x)}$  we can uniquely identify the symbol  $x$ .*

**Proof:** The codeword length is  $l(x) = \left\lceil \log \frac{1}{p(x)} \right\rceil + 1$ , and the modified CDF is  $\overline{F}(x) = \sum_{y < x} p(y) + \frac{1}{2}p(x)$ . Let  $\lfloor \overline{F}(x) \rfloor_{l(x)}$  be the truncation of  $\overline{F}$  to  $l(x)$  of MSB after the decimal point. Hence, due to truncation we get

$$\begin{aligned}\overline{F}(x) - \lfloor \overline{F}(x) \rfloor_{l(x)} &< 2^{-l(x)} \leq 2^{(-1+\log p(x))} \\ &= \frac{p(x)}{2} = \overline{F}(x) - F(x-1)\end{aligned}$$

$$\Rightarrow F(x-1) < \overline{F}(x) - 2^{-l(x)} < \lfloor \overline{F}(x) \rfloor_{l(x)} \leq \overline{F}(x).$$

Hence, the codeword  $\lfloor \overline{F}(x) \rfloor_{l(x)}$  lies within the jump height at  $x$ .

## SFE Coding Algorithm (cont'd)

### Theorem

*SFE code is prefix-free.*

**Proof:** Let  $[\overline{F}(x)]_{l(x)} = 0.z_1z_2 \dots z_l$ . Suppose it is a prefix of another code, then

$$z' = 0.z_1z_2 \dots z_lz_{l+1} \dots z_n \in [0.z_1z_2 \dots z_l, 0.z_1z_2 \dots z_l + 2^{-l}) \subset [\overline{F}(x), F(x)).$$

By the aforementioned Lemma, we know that there is no such codeword. In other words, the intervals  $[0.z_1z_2 \dots z_l, 0.z_1z_2 \dots z_l + 2^{-l})$  corresponding to different codewords are disjoint, and hence the code is prefix-free.

**Code interval.** View binary fractional numbers as intervals by considering *all completions*; e.g.,

	min	max	code interval
.11	.11 $\overline{0}$	.11 $\overline{1}$	[.75, 1.0)
.101	.101 $\overline{0}$	.101 $\overline{1}$	[0.625, .75)

### Lemma

*If a set of code intervals are disjoint, then the corresponding codes are prefix-free.*

## Shannon Coding

The Shannon code is defined in a similar way as the SFE code with the exception of

- The symbols are ordered by decreasing probability  $p_1 \geq p_2 \geq \dots \geq p_m$ .
- The codeword of  $i$ -th symbol is formed by the  $\lceil \log \frac{1}{p_i} \rceil$  bits of  $\tilde{F}_i = \sum_{k=1}^{i-1} p_k$ .
- Alternatively (from most-to-least probable symbols), picking each codeword to be the lexicographically first alphabets of the correct length that maintains the prefix-free property.<sup>3</sup>

$i$	$p_i$	$l_i$	$\sum_{n=0}^{i-1} p_n$	Previous value in binary	Codeword for $a_i$
1	0.36	2	0.0	0.0000	00
2	0.18	3	0.36	0.0101...	010
3	0.18	3	0.54	0.1000...	100
4	0.12	4	0.72	0.1011...	1011
5	0.09	4	0.84	0.1101...	1101
6	0.07	4	0.93	0.1110...	1110

Figure: An example of Shannon coding.

<sup>3</sup>[https://en.wikipedia.org/wiki/Shannon%E2%80%93Fano\\_coding](https://en.wikipedia.org/wiki/Shannon%E2%80%93Fano_coding)

## Competitive Optimality

We consider the Shannon code and compare its performance with any other uniquely decodable code.

### Theorem (Competitive optimality)

*Let  $l(x)$  be the codeword length associated with the Shannon code. Let  $l'(x)$  be the codeword length associated with any other uniquely decodable code. Then*

$$\Pr(l(X) \geq l'(X) + c) \leq \frac{1}{2^{c-1}}.$$

**Proof:**

$$\begin{aligned}\Pr(l(X) > l'(X) + c) &\leq \Pr(-\log_2 p(X) + 1 \geq l'(X) + c) \\&= \Pr(p(X) \leq 2^{-l'(X)-c+1}) \\&= \sum_x p(x) \times \mathbb{1}(p(x) \leq 2^{-l'(x)-c+1}) \\&\leq \sum_x 2^{-l'(x)-c+1} \times \mathbb{1}(p(x) \leq 2^{-l'(x)-c+1}) \\&\leq 2^{1-c} \times \sum_{x \in \mathcal{X}} 2^{-l'(x)} \leq \frac{1}{2^{c-1}} \quad [\text{by McMillan's inequality}]\end{aligned}$$

## Competitive Optimality (cont'd)

**Shannon code efficiency.** Since  $l_i = \lceil \log \frac{1}{p_i} \rceil \implies$

$$\log \frac{1}{p_i} \leq l_i \leq 1 + \log \frac{1}{p_i} \implies H(X) \leq L_S \leq H(X) + 1.$$

### Theorem

*For a dyadic PMF  $p(x)$ , let  $l(x) = \log \frac{1}{p(x)}$  be the codeword lengths of the binary Shannon code. Let  $l'(x)$  be the lengths of any other uniquely decodable binary code. Then*

$$\Pr(l(X) \leq l'(X)) \geq \Pr(l(X) > l'(X)) ,$$

*with equality iff  $l'(x) = l(x)$  for all  $x$ . The code length  $l(x) = \log \frac{1}{p(x)}$  is uniquely competitively optimal.*

To this end, we have:  $H(X) \leq L_H \leq L_{SF} \leq L_S \leq H(X) + 1 \leq L_{SFE} < H(X) + 2$

## Comparing Different Source Coding

- Shannon code takes a probability-matching approach: the code length of a symbol matches the probability of the symbol.
- Fano code uses equal partitioning based on probability: minimize the difference of the probabilities of the two partitioned sets.
- Huffman code uses sorting and merging of symbol sets based on probability: merge the two sets with the least probabilities.



## Arithmetic Coding (AC)

The SFE coding is the precursor to the arithmetic coding. Instead of using the symbol-by-symbol coding approach, arithmetic coding encodes the entire message (a sequence of symbols) into a single number  $q \in [0, 1)$ . The key idea is to represent any finite-length sequence of symbols by a sub-interval of the unit interval.

### Problem with Huffman coding.

- Consider two symbols with the probabilities:  
 $\{\Pr(A) = 0.9999, \Pr(B) = 0.0001\} \Rightarrow H = 1.47 \times 10^{-3}$  bits.
- If we were to send “A” 1000 times, we might hope to use  $1000 \times H = 1.47$  bits.
- Using Huffman codes at least one bit per symbol is needed, so we would require 1000 bits.
- We could alleviate this loss by using the block Huffman coding with a codebook of size  $|\mathcal{X}|^n$  (need to enumerate all sequences of length  $n$ ), which increases exponentially with the block length  $n$ , *not practical!*
- For certain high-speed applications, the space complexity and speed of Huffman coding is a bottleneck.

## Basic ideas.

- The entire message is represented by an interval in  $[0, 1)$
- The interval is successively divided into sub-intervals proportional to symbol probabilities
- Final interval size = product of symbol probabilities
- Final code = any value from the interval
- Intervals for distinct message sequences are disjoint
- Decoding computes the same sequence of intervals

## AC Algorithm (cont'd)

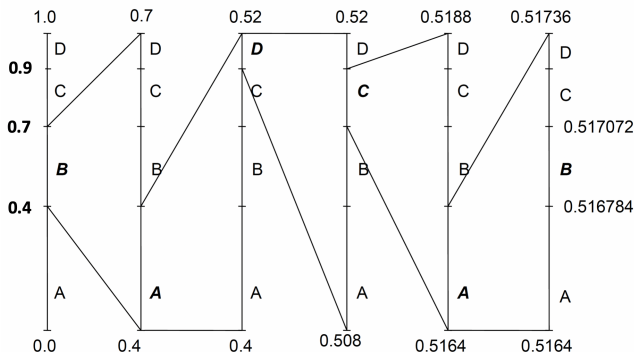


Figure: Sequence "BADCAB" maps to any number  $q \in [0.516784, 0.517072]$ .

- Encoding with rescaled intervals: each symbol narrows the interval by a factor of  $p_i$ .
- Final *sequence interval* size:  $s_n = \prod_{i=1}^n p_i$ .
- Specifying any number in the final interval uniquely determines the sequence.

## AC Algorithm (cont'd)

**Recursive formulas.** Consider a message sequence of length  $n$  to be encoded by the arithmetic coding. For  $i = 1, 2, \dots, n$ , let  $\ell_i$  and  $s_i$  denote the **low end** and the **size** of the interval after we read off the  $i$ -th symbol in the message sequence. The final interval  $[\ell_n, \ell_n + s_n)$  can be determined by the following recursions:

$$\boxed{\ell_i = \ell_{i-1} + s_{i-1}f_{k(i)}, \quad s_i = s_{i-1}p_i,} \quad \text{with } \ell_0 = 0, s_0 = 1.$$

- $p_i$  is the probability of the  $i$ -th symbol in the sequence, which corresponds to the  $k(i)$ -th symbol that has been sorted in the original source set.
- $f_{k(i)} = \sum_{j=1}^{k(i)-1} \tilde{p}_j$  is the modified CDF of the  $k(i)$ -th symbol, which sums up the probabilities ( $\tilde{p}_j$ 's) of all its preceding symbols in the source set.

## AC Algorithm (cont'd)

For the above example, the message sequence to be encoded is “BAD CAB”. The sorted source symbols and associated quantities are summarized in the following table:

Sorted symbols	A	B	C	D
Probability	0.4	0.3	0.2	0.1
Modified CDF	0	0.4	0.7	0.9

Use the recursions, we have:

$$\ell_1 = \ell_0 + s_0 f_B = 0 + 1 \times 0.4 = 0.4, \quad s_1 = s_0 p_1 = 1 \times 0.3 = 0.3$$

$$\ell_2 = \ell_1 + s_1 f_A = 0.4 + 0.3 \times 0 = 0.4, \quad s_2 = s_1 p_2 = 0.3 \times 0.4 = 0.12$$

$$\ell_3 = \ell_2 + s_2 f_D = 0.4 + 0.12 \times 0.9 = 0.508, \quad s_3 = s_2 p_3 = 0.12 \times 0.1 = 0.012$$

$$\ell_4 = \ell_3 + s_3 f_C = 0.508 + 0.012 \times 0.7 = 0.5164, \quad s_4 = s_3 p_4 = 0.012 \times 0.2 = 0.0024$$

# RealArith Encoding & Decoding

- RealArithEncoder:
  1. Determine the low end  $\ell_n$  and the size  $s_n$  of the final interval by using the recursions.
  2. Code using the mid-point  $m_n = \ell_n + \frac{s_n}{2}$  truncated to  $\lceil \log \frac{1}{s_n} \rceil + 1$  bits.
- RealArithDecoder:
  1. Read bits as needed so code interval falls within a message interval.
  2. Repeat until the entire message sequence of length  $n$  has been decoded.

## Theorem

Let  $s_n$  denote the final probability interval of the RealArithEncoder. The binary representation of  $m_n = \ell_n + \frac{s_n}{2}$  ( $\text{mid} = \frac{\text{upper} + \text{lower}}{2}$ ) truncated to  $l(x) = 1 + \lceil \log \frac{1}{s_n} \rceil$  bits is a uniquely decodable code for message  $x$  among prefix-free messages.

## RealArith Encoding & Decoding (cont'd)

### Theorem (Bound of AC length)

For a message sequence of length  $n$  with self information  $\left\{ \text{SI}_i = \log \frac{1}{p_i} \right\}_{i=1}^n$ , *RealArithEncode* generates at most  $2 + \sum_{i=1}^n \text{SI}_i$  bits.

**Proof:**  $1 + \lceil \log \frac{1}{s_n} \rceil = 1 + \lceil \log (\prod_i p_i)^{-1} \rceil = 1 + \lceil \sum_i \log \frac{1}{p_i} \rceil < 2 + \sum_{i=1}^n \text{SI}_i$ .

**Optimality (expected length of an  $n$ -symbol message).**

$$\begin{aligned} L^{(n)} &= \sum_{x^n \in \mathcal{X}^n} p(x^n) l(x^n) = \sum p(x^n) \left( \lceil \log \frac{1}{p(x^n)} \rceil + 1 \right) \\ &\leq \sum p(x^n) \left( \log \frac{1}{p(x^n)} + 2 \right) = H(X^n) + 2 \implies \end{aligned}$$

Bits per symbol:  $\frac{H(X^n)}{n} \leq L_{AC} \leq \frac{H(X^n)}{n} + \frac{2}{n} \Rightarrow L_{AC} \rightarrow H(\mathcal{X})$  as  $n \rightarrow \infty$ .

*Note:* Although this is not necessarily optimal for any fixed block length (Huffman code has a lower average codeword length), the procedure is incremental and can be used for any block length.

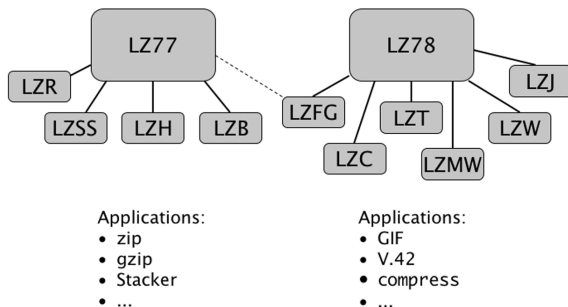
# Universal Source Coding

- Huffman coding and arithmetic coding give at least as good, and often better compression than any universal codes.
- Universal codes are useful when Huffman coding is not applicable; e.g., when one does not know the exact probability of each message.
- Universal codes are also useful when Huffman coding is inconvenient. For example, when the transmitter but not the receiver knows the probabilities of the messages, Huffman coding requires an overhead of transmitting those probabilities to the receiver. Using a universal code does not have that overhead.



## Lempel-Ziv Coding

There are many variations of Lempel Ziv around. We will only concentrate on one of the simplest to explain and analyze the idea: **parse the sequence into distinct phrases**.



**Figure:** LZ77 and LZ78 (aka LZ1 and LZ2) are the two lossless data compression algorithms published in papers by Abraham Lempel and Jacob Ziv in 1977 and 1978. Figure source: “The Lempel Ziv Algorithm” by Christina Zeeh, Jan 2013.

## Lempel-Ziv Coding (cont'd)

Consider the source sequence as *AABABBBABAABABBBABBABB*.

- Start with the shortest phrase that we haven't seen before. This will always be a single letter, in this case A:

*A|ABABBBABAABABBBABBABB*

- Now take the next phrase we haven't seen. So we take AB:

*A|AB|ABBBABAABABBBABBABB*

- The next phrase we haven't seen is ABB. Continuing, we get B after that:

*A|AB|ABB|B|ABAABABBBABBABB*

- The rest of the string parses into:

*A|AB|ABB|B|ABA|ABAB|BB|ABBA|BB*

It's fine to have a repeated phrase at the end since we've run out of letters.

## Lempel-Ziv Coding (cont'd)

Index	1	2	3	4	5	6	7	8	9
Phrase	A	AB	ABB	B	ABA	ABAB	BB	ABBA	BB
Encoding	$\emptyset$ A	1B	2B	$\emptyset$ B	2A	5B	4B	3A	7
Codeword	0	11	101	001	0100	1011	1001	0110	0111

Table: LZ coding scheme.

- Each phrase is broken up into a reference (in blue) to a previous phrase, as well as a letter (in red) that is always an innovation. To get the final LZ code, we convert both parts into binary codes with a mapping  $A \rightarrow 0$  and  $B \rightarrow 1$ .
- Start with the  $2^{k-1} + 1$  dictionary element (i.e., index = 2, 3, 5, 9, ... for  $k = 1, 2, 3, 4, \dots$ ), the reference part uses  $k$  bits. So the number of bits used for this part keeps increasing. This ensures that the decoding algorithm knows where to divide.

*Thank You!*

Email: <zhangy@ucsc.edu>

Homepage: <https://people.ucsc.edu/~yzhan419/>