ECE 515
Information Theory

Distortionless Source Coding 2

Huffman Coding

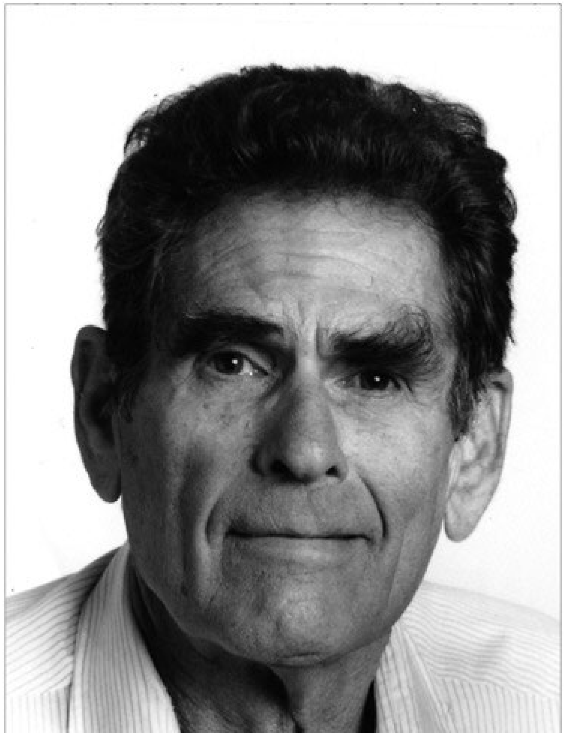
- The length of Huffman codewords has to be an integer number of symbols, while the self-information of the source symbols is almost always a non-integer.
- Thus the theoretical minimum message compression cannot always be achieved.
- For a binary source with $p(x_1) = 0.1$ and $p(x_2) = 0.9$
  - $H(X) = .469$ so the optimal average codeword length is .469 bits
  - Symbol x_1 should be encoded to $l_1 = -\log_2(0.1) = 3.32$ bits
  - Symbol x_2 should be encoded to $l_2 = -\log_2(0.9) = .152$ bits

Improving Huffman Coding

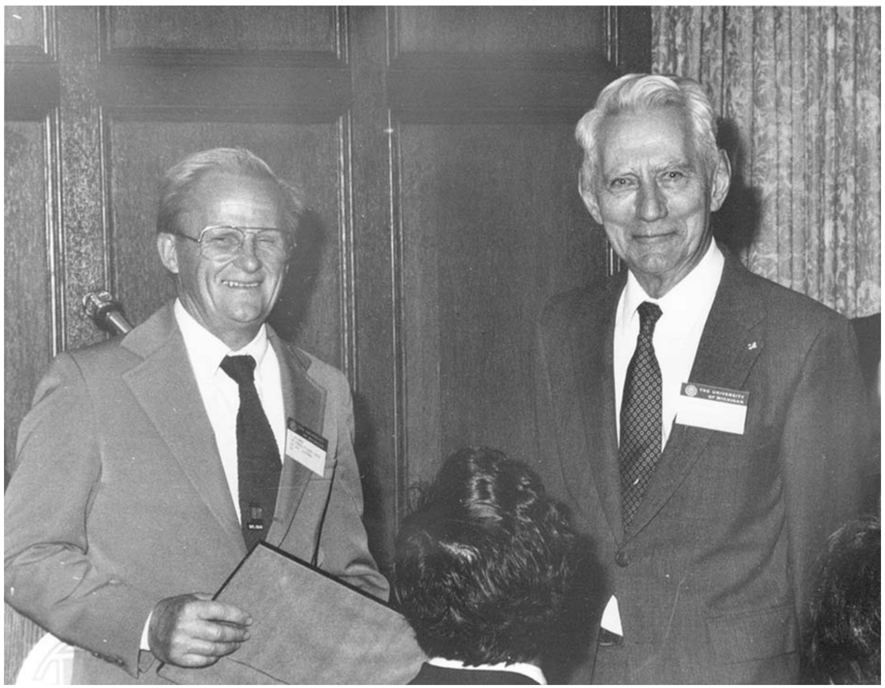
- One way to overcome the redundancy limitation is to encode blocks of several symbols.

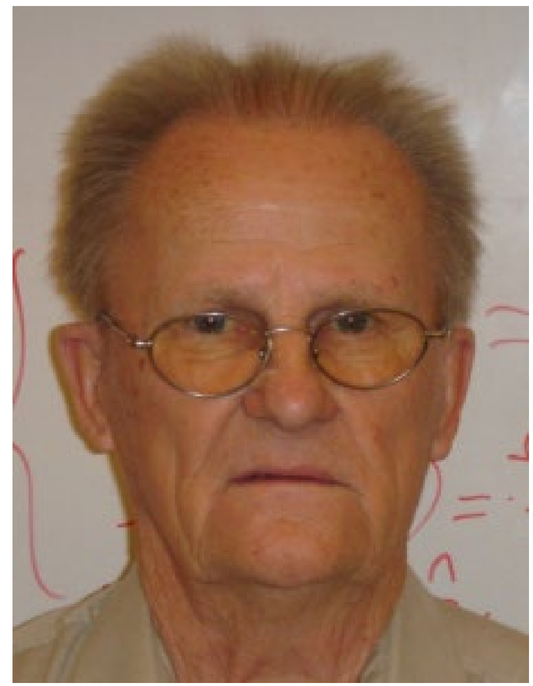
  In this way the per-symbol inefficiency is spread over an entire block.
  - $N = 1$: $\zeta = 46.9\%$ $N = 2$: $\zeta = 72.7\%$ $N = 3$: $\zeta = 80.0\%$
- However, using blocks is difficult to implement as there is a block for every possible combination of symbols, so the number of blocks (and thus codewords) increases exponentially with their length.
  - The probability of each block must be computed.

Peter Elias (1923 – 2001)



Jorma J. Rissanen (1932 – 2020)





Arithmetic Coding

- Arithmetic coding bypasses the idea of replacing a source symbol (or groups of symbols) with a specific codeword.
- Instead, a sequence of symbols is encoded to an interval in [0,1).
- Useful when dealing with sources with small alphabets, such as binary sources, and alphabets with highly skewed probabilities.

Arithmetic Coding Applications

- JPEG, MPEG-1, MPEG-2
  - Huffman and arithmetic coding
- JPEG2000, MPEG-4
  - Arithmetic coding only
- ZIP
  - prediction by partial matching (PPMd) algorithm
- H.263, H.264

Arithmetic Coding

- Lexicographic ordering
- Cumulative probabilities $P_j = \sum_{i=1}^{j-1} \mathrm{p}(u_i)$

$$
\begin{array}{lll}
u_1 & x_1 x_1 \ldots x_1 & P_1 \\
u_2 & x_1 x_1 \ldots x_2 & P_2 \\
\vdots & \vdots & \vdots \\
u_{K^N} & x_K x_K \ldots x_K & P_{K^N}
\end{array}
$$

- The interval P_j to P_{j+1} defines u_j

Example

- $K = 2$ $N = 3$ $p(x_1) = 0.1$ $p(x_2) = 0.9$

u_1	$x_1x_1x_1$	0
u_2	$x_1x_1x_2$	.001
u_3	$x_1x_2x_1$	.010
u_4	$x_1x_2x_2$	.019
u_5	$x_2x_1x_1$	.100
u_6	$x_2x_1x_2$	.109
u_7	$x_2x_2x_1$	.190
u_8	$x_2x_2x_2$	.271

$P_9 = 1$

Arithmetic Coding

- A sequence of source symbols is represented by an interval in [0,1).
- The probabilities of the source symbols are used to successively narrow the interval used to represent the sequence.
- As the interval becomes smaller, the number of bits needed to specify it grows.
- A high probability symbol narrows the interval less than a low probability symbol so that high probability symbols contribute fewer bits to the codeword.
- For a sequence *u* of *N* symbols, the codeword length should be approximately $l_u = \lceil -\log_2 \mathrm{p}(u) \rceil$ bits

Arithmetic Coding

- The output of an arithmetic encoder is a stream of bits.
- However we can think that there is a prefix 0, and the stream represents a fractional binary number between 0 and 1

$$01101010 \quad \rightarrow \quad 0.01101010$$

- In the examples, decimal numbers will be used for convenience.

Arithmetic Coding

- The initial intervals are based on the cumulative probabilities

$$P_k = \sum_{i=1}^{k-1} \mathrm{p}(x_i)$$

$P_1 = 0$ and $P_{K+1} = 1$

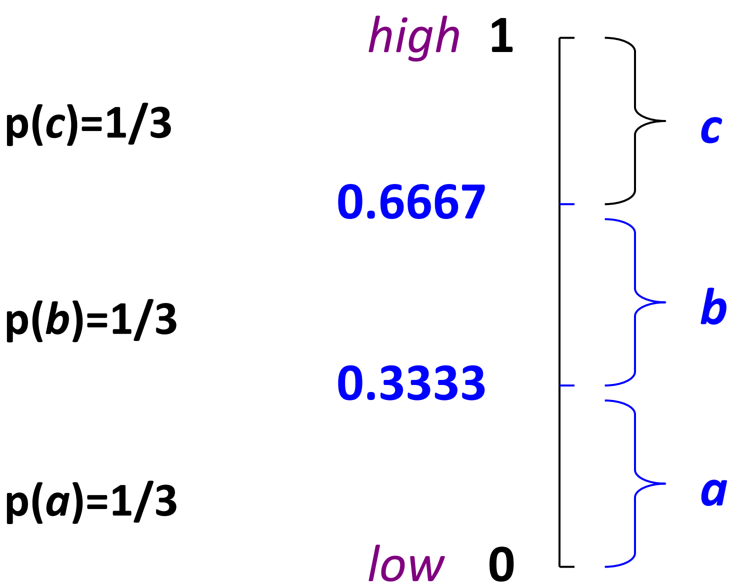
- Source symbol k is assigned the interval $[P_k, P_{k+1})$

Example 1

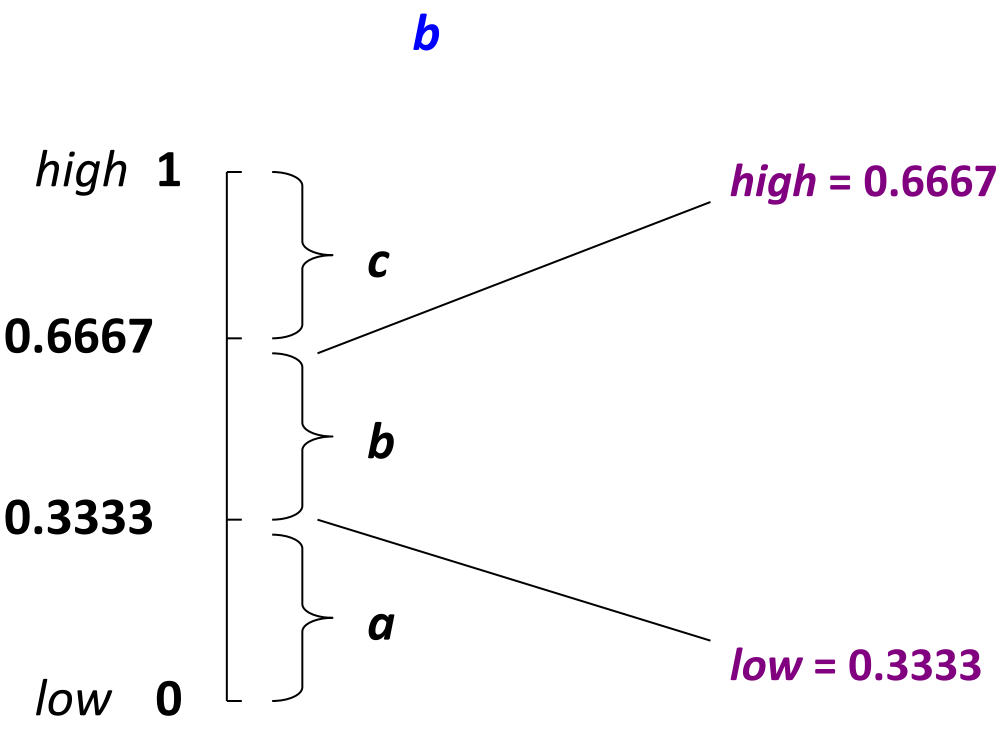
- Encode string *bccb* from the source X = {*a*,*b*,*c*}
- K=3
- p(*a*) = p(*b*) = p(*c*) = 1/3
- $P_1 = 0$ $P_2 = .3333$ $P_3 = .6667$ $P_4 = 1$
- The encoder maintains two numbers, *low* and *high*, which represent an interval [*low*,*high*) in [0,1)
- Initially *low* = 0 and *high* = 1

Example 1

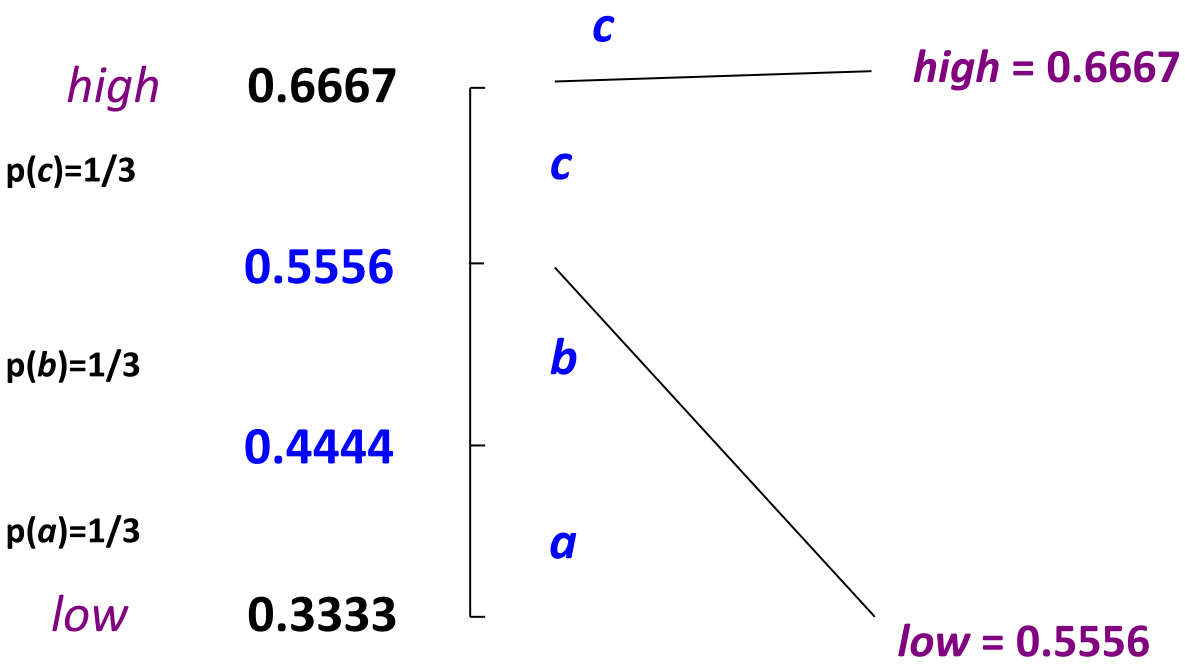
- The interval between *low* and *high* is divided among the symbols of the source alphabet according to their probabilities



Example 1

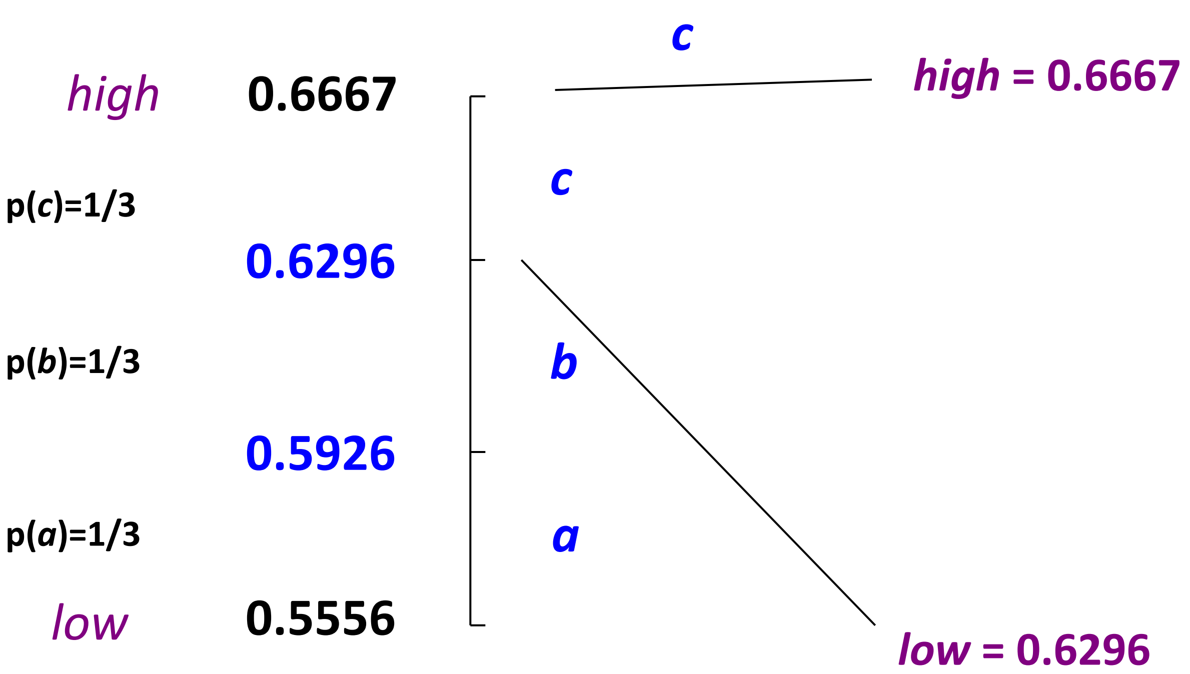


Example 1

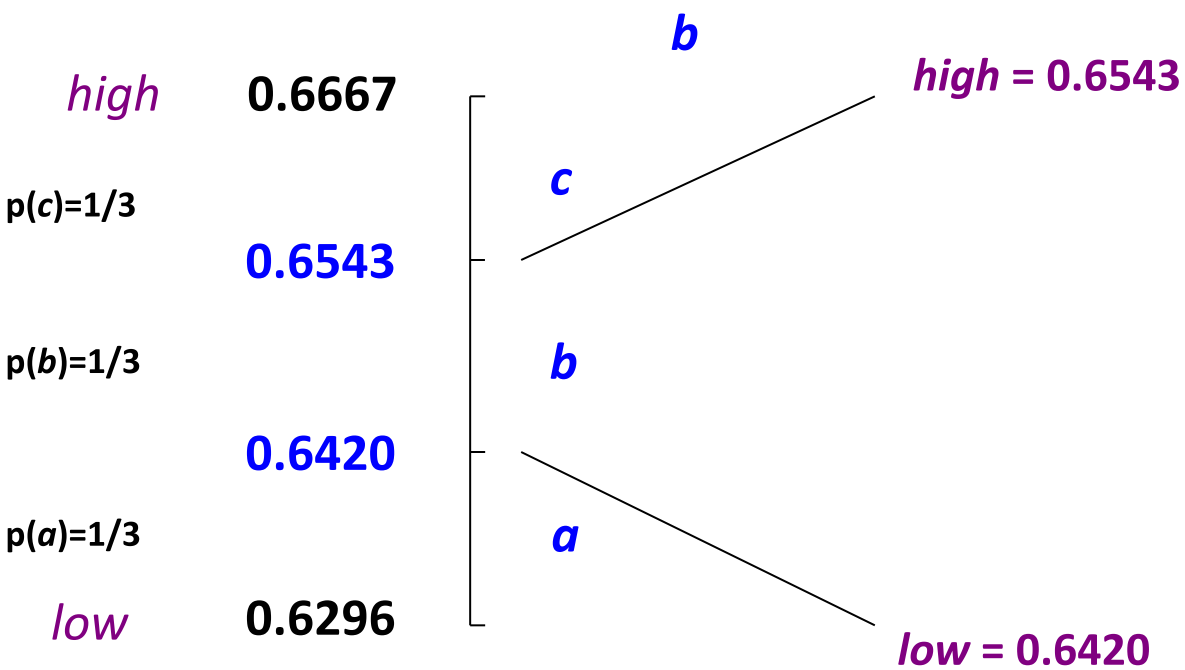


high 0.6667
p(*c*)=1/3
0.5556
p(*b*)=1/3
0.4444
p(*a*)=1/3
low 0.3333
c
c
b
a
high = 0.6667
low = 0.5556

Example 1


c
high
0.6667
high = 0.6667
c
p(c)=1/3
0.6296
p(b)=1/3
b
0.5926
p(a)=1/3
a
low
0.5556
low = 0.6296

Example 1


b
high 0.6667
high = 0.6543
c
p(c)=1/3
0.6543
b
p(b)=1/3
0.6420
a
p(a)=1/3
low 0.6296
low = 0.6420

Arithmetic Coding Algorithm

Set low to 0

Set high to 1

While there are still input symbols **Do**

    get next input symbol (x_i)

    range = high – low

    high = low + range × symbol_high_interval (P_{i+1})

    low = low + range × symbol_low_interval (P_i)

End While

output number between high and low

Example 2

- Source X with $K = 3$ symbols $\{x_1, x_2, x_3\}$
- $p(x_1) = 0.5$ $p(x_2) = 0.3$ $p(x_3) = 0.2$
  - $0 \le x_1 < 0.5$
  - $0.5 \le x_2 < 0.8$
  - $0.8 \le x_3 < 1$
  - $P_1 = 0, P_2 = .5, P_3 = .8, P_4 = 1$
- The encoder maintains two numbers, *low* and *high*, which represent an interval [*low*,*high*) in [0,1)
- Initially *low* = 0 and *high* = 1

Arithmetic Coding Example 2

- $p(x_1) = 0.5$, $p(x_2) = 0.3$, $p(x_3) = 0.2$
- Symbol intervals: $0 \le x_1 < .5$ $.5 \le x_2 < .8$ $.8 \le x_3 < 1$
- $P_1 = 0$, $P_2 = .5$, $P_3 = .8$, $P_4 = 1$
- low = 0.0 high = 1.0
- Symbol sequence $x_1 x_2 x_3 x_2$

- Iteration 1

  x_1: range = 1.0 - 0.0 = 1.0

  high = 0.0 + 1.0 × 0.5 = 0.5

  low = 0.0 + 1.0 × 0.0 = 0.0
- Iteration 2

  x_2: range = 0.5 - 0.0 = 0.5

  high = 0.0 + 0.5 × 0.8 = 0.40

  low = 0.0 + 0.5 × 0.5 = 0.25
- Iteration 3

  x_3: range = 0.4 - 0.25 = 0.15

  high = 0.25 + 0.15 × 1.0 = 0.40

  low = 0.25 + 0.15 × 0.8 = 0.37

Arithmetic Coding Example 2

- Iteration 3

  x_3: range = 0.4 - 0.25 = 0.15

  high = 0.25 + 0.15 × 1.0 = 0.40

  low = 0.25 + 0.15 × 0.8 = 0.37
- Iteration 4

  x_2: range = 0.4 - 0.37 = 0.03

  high = 0.37 + 0.03 × 0.8 = 0.394

  low = 0.37 + 0.03 × 0.5 = 0.385

- $0.385 \leq x_1 x_2 x_3 x_2 < 0.394$

  0.385 = 0.0110001…

  0.394 = 0.0110010…

- The first 5 bits of the codeword are 01100

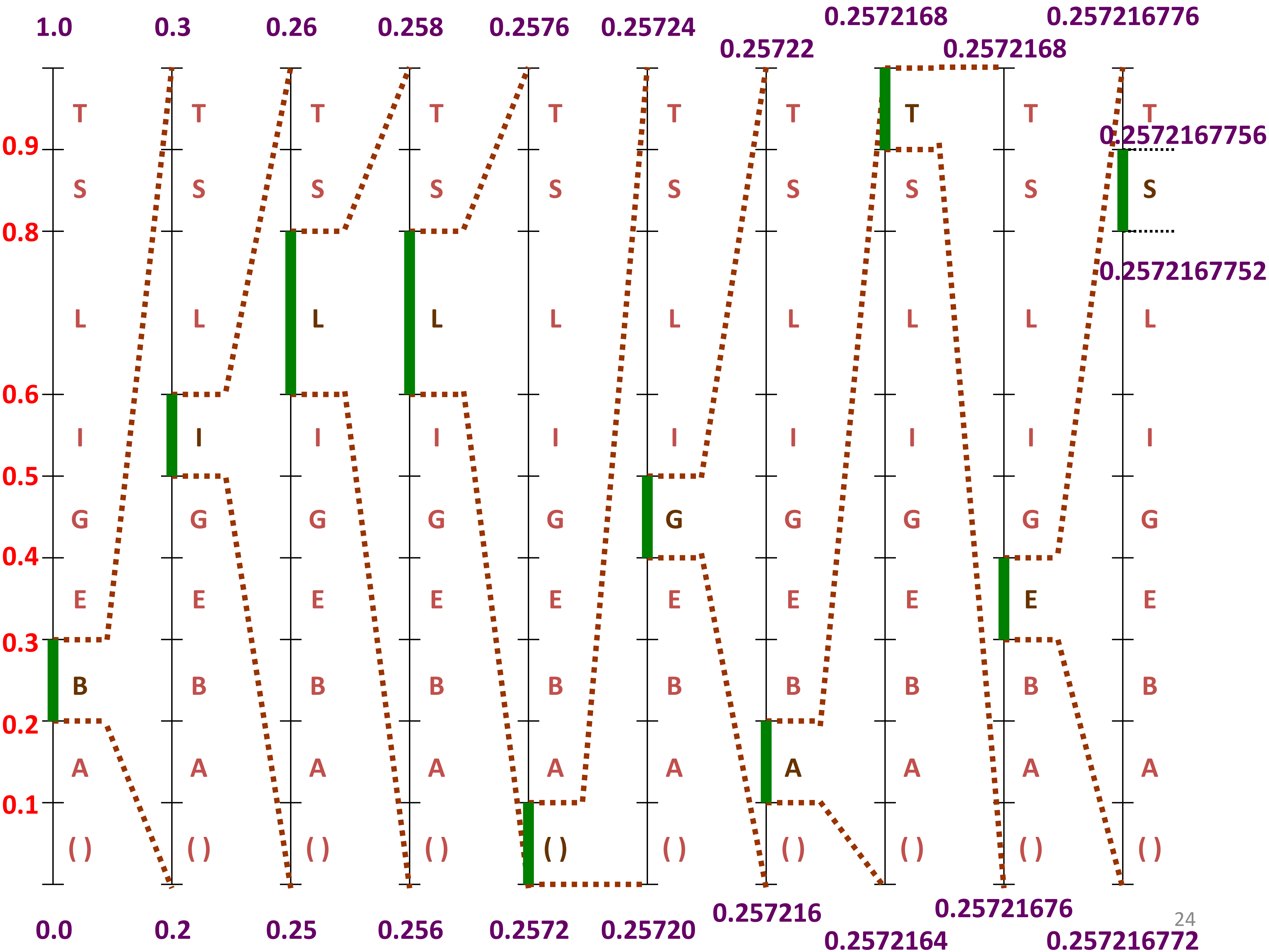
- If there are no additional symbols to be encoded the codeword is 011001

Arithmetic Coding Example 3

Suppose that we want to encode the message

BILL GATES

Character	Probability	Interval
SPACE	1/10	$0.00 \le x_1 < 0.10$
A	1/10	$0.10 \le x_2 < 0.20$
B	1/10	$0.20 \le x_3 < 0.30$
E	1/10	$0.30 \le x_4 < 0.40$
G	1/10	$0.40 \le x_5 < 0.50$
I	1/10	$0.50 \le x_6 < 0.60$
L	2/10	$0.60 \le x_7 < 0.80$
S	1/10	$0.80 \le x_8 < 0.90$
T	1/10	$0.90 \le x_9 < 1.00$


1.0
0.3
0.26
0.258
0.2576
0.25724
0.25722
0.2572168
0.2572168
0.257216776
0.2572167756
0.2572167752
0.9
0.8
0.6
0.5
0.4
0.3
0.2
0.1
T
S
L
I
G
E
B
A
( )
0.0
0.2
0.25
0.256
0.2572
0.25720
0.257216
0.2572164
0.25721676
0.257216772

Arithmetic Coding Example 3

New Symbol	Low	High
	0.0	1.0
B	0.2	0.3
I	0.25	0.26
L	0.256	0.258
L	0.2572	0.2576
SPACE	0.25720	0.25724
G	0.257216	0.257220
A	0.2572164	0.2572168
T	0.25721676	0.2572168
E	0.257216772	0.257216776
S	0.2572167752	0.2572167756

Binary Codeword

- 0.2572167752 in binary is
  0.0100000111011000111101010110010 1…
- 0.2572167756 in binary is
  0.0100000111011000111101010110011 1…
- The codeword is then
  0100000111011000111101010110011
- 31 bits long

Decoding Algorithm

get encoded number (codeword)

Do

find the symbol whose interval contains the encoded number (x_i)

output the symbol

subtract symbol_low_interval (P_i) from the encoded number

divide by the probability of the output symbol (p(x_i))

Until termination

Decoding BILL GATES

Encoded Number	Output Symbol	Low	High	Probability
0.2572167752	B	0.2	0.3	0.1
0.572167752	I	0.5	0.6	0.1
0.72167752	L	0.6	0.8	0.2
0.6083876	L	0.6	0.8	0.2
0.041938	SPACE	0.0	0.1	0.1
0.41938	G	0.4	0.5	0.1
0.1938	A	0.2	0.3	0.1
0.938	T	0.9	1.0	0.1
0.38	E	0.3	0.4	0.1
0.8	S	0.8	0.9	0.1
0.0				

Finite Precision

Symbol	Probability (fraction)	Interval (8-bit precision) fraction	Interval (8-bit precision) binary	Interval boundaries in binary
a	1/3	[0,85/256)	[0.00000000, 0.01010101)	00000000 01010100
b	1/3	[85/256,171/256)	[0.01010101, 0.10101011)	01010101 10101010
c	1/3	[171/256,1)	[0.10101011, 1.00000000)	10101011 11111111

Renormalization

Symbol	Probability (fraction)	Interval boundaries	Digits that can be output	Boundaries after renormalization
a	1/3	**0**0000000 **0**1010100	0	0000000**0** 1010100**1**
b	1/3	01010101 10101010	none	01010101 10101010
c	1/3	**1**0101011 **1**1111111	1	0101011**0** 1111111**1**

Termination Symbol

Symbol	Probability (fraction)	Interval (8-bit precision) fraction	Interval (8-bit precision) binary	Interval boundaries in binary
a	1/3	[0,85/256)	[0.00000000, 0.01010101)	00000000 01010100
b	1/3	[85/256,170/256)	[0.01010101, 0.10101011)	01010101 10101001
c	1/3	[170/256,255/256)	[0.10101011, 0.11111111)	10101010 11111110
term	1/256	[255/256,1)	[0.11111111, 1.00000000)	11111111

Huffman vs Arithmetic Codes

- $K = 4$ $X = \{a,b,c,d\}$
- $p(a) = .5$, $p(b) = .25$, $p(c) = .125$, $p(d) = .125$
- Huffman code

a	0
b	10
c	110
d	111

Huffman vs Arithmetic Codes

- X = {*a*,*b*,*c*,*d*}
- p(*a*) = .5, p(*b*) = .25, p(*c*) =.125, p(*d*) = .125
- $P_1 = 0$, $P_2 = .5$, $P_3 = .75$, $P_4 = .875$, $P_5 = 1$
- Arithmetic code intervals

a	[0, .5)
b	[.5, .75)
c	[.75, .875)
d	[.875, 1)

Huffman vs Arithmetic Codes

- encode *abcdc*
- Huffman codewords
  - 010110111110 12 bits
- Arithmetic code
  - low = $.010110111110_2$
  - high = $.010110111111_2$
  - codeword 010110111110 12 bits
  - $p(u) = (.5)(.25)(.125)^3 = 2^{-12}$
  - $l_u = \lceil -\log_2 p(u) \rceil = 12$ bits

Huffman vs Arithmetic Codes

- X = {*a*,*b*,*c*,*d*}
- p(*a*) = .7, p(*b*) = .12, p(*c*) =.10, p(*d*) = .08
- Huffman code

a	0
b	10
c	110
d	111

Huffman vs Arithmetic Codes

- X = {*a*,*b*,*c*,*d*}
- p(*a*) = .7, p(*b*) = .12, p(*c*) =.10, p(*d*) = .08
- $P_1 = 0$, $P_2 = .7$, $P_3 = .82$, $P_4 = .92$, $P_5 = 1$
- Arithmetic code intervals

a	[0, .7)
b	[.7, .82)
c	[.82, .92)
d	[.92, 1)

Huffman vs Arithmetic Codes

- encode *aaab*
- Huffman codewords
  - 00010 5 bits
- Arithmetic code
  - low = $.00111101\ldots_2$
  - high = $.01001000\ldots_2$
  - codeword 01 2 bits
  - p(*u*) = $(.7)^3(.12)$ = .04116
  - $l_u = \lceil -\log_2 \mathrm{p}(u) \rceil = \lceil 4.60 \rceil = 5$ bits

Huffman vs Arithmetic Codes

- encode *abcdaaa*
- Huffman codewords
  - 010110111000   12 bits
- Arithmetic code
  - low = $.1001000100001101\ldots_2$
  - high = $.1001000100011100\ldots_2$
  - codeword 100100010001 12 bits
  - $\mathrm{p}(u) = (.7)^3(.12)(.10)(.08) = .0002305$
  - $l_u = \lceil -\log_2 \mathrm{p}(u) \rceil = \lceil 12.08 \rceil = 13$ bits

Huffman vs Arithmetic Codes

- Huffman code L(C) = 1.480 bits
- H(X) = 1.351 bits
- Redundancy = L(C) – H(X) = .129 bit
- Arithmetic code will achieve the theoretical performance H(X)
- For a file of size $N = 10^6$ symbols
  - Arithmetic code $N \times H(X) = 1.351 \times 10^6$ bits
  - Huffman code $N \times L(C) = 1.480 \times 10^6$ bits
  - Difference 1.29×10^5 bits

Robustness of Huffman Codes
and
Universal Source Coding

Robustness of Huffman Codes

$$p_k = \mathrm{p}(x_k) \qquad \text{(actual)}$$

$$q_k = p_k + \varepsilon_k \qquad \text{(estimated)}$$

$$\sum_{k=1}^{K} p_k = 1 \qquad \sum_{k=1}^{K} q_k = 1$$

$$\therefore \sum_{k=1}^{K} \varepsilon_k = 0$$

Robustness of Huffman Codes

$$\mathrm{L}(\mathrm{C}) = \sum_{k=1}^{K} p_k l_k \qquad \mathrm{L}(\hat{\mathrm{C}}) = \sum_{k=1}^{K} p_k \hat{l}_k$$

$$\Delta \mathrm{L} = \mathrm{L}(\hat{\mathrm{C}}) - \mathrm{L}(\mathrm{C}) = \sum_{k=1}^{K} p_k \hat{l}_k - \sum_{k=1}^{K} p_k l_k$$

$$= \sum_{k=1}^{K} p_k \left(\hat{l}_k - l_k \right)$$

Upper and Lower Bounds

- p(X) actual distribution $\mathrm{L}(\mathrm{C}) = \sum_{k=1}^{K} \mathrm{p}(x_k) l_k$
- q(X) estimated distribution $\mathrm{L}(\hat{\mathrm{C}}) = \sum_{k=1}^{K} \mathrm{p}(x_k) \hat{l}_k$

$$\frac{\mathrm{H}(\mathrm{p}(\mathrm{X}))}{\log_b J} \leq \mathrm{L}(\mathrm{C}) < \frac{\mathrm{H}(\mathrm{p}(\mathrm{X}))}{\log_b J} + 1$$

$$\frac{\mathrm{H}(\mathrm{p}(\mathrm{X})) + \mathrm{D}(\mathrm{p}(\mathrm{X}) \,||\, \mathrm{q}(\mathrm{X}))}{\log_b J} \leq \mathrm{L}(\hat{\mathrm{C}}) < \frac{\mathrm{H}(\mathrm{p}(\mathrm{X})) + \mathrm{D}(\mathrm{p}(\mathrm{X}) \,||\, \mathrm{q}(\mathrm{X}))}{\log_b J} + 1$$

Upper and Lower Bounds

$$\frac{H(p(X)) + D(p(X)||q(X))}{\log_b J} \leq L(\hat{C}) < \frac{H(p(X)) + D(p(X)||q(X))}{\log_b J} + 1$$

$$\frac{H(p,q)}{\log_b J} \leq L(\hat{C}) < \frac{H(p,q)}{\log_b J} + 1$$

$$\text{if } b = j \quad H(p,q) \leq L(\hat{C}) < H(p,q) + 1$$

Gadsby by Ernest Vincent Wright

If youth, throughout all history, had had a champion to stand up for it; to show a doubting world that a child can think; and, possibly, do it practically; you wouldn't constantly run across folks today who claim that "a child don't know anything." A child's brain starts functioning at birth; and has, amongst its many infant convolutions, thousands of dormant atoms, into which God has put a mystic possibility for noticing an adult's act, and figuring out its purport.

Up to about its primary school days a child thinks, naturally, only of play. But many a form of play contains disciplinary factors. "You can't do this," or "that puts you out," shows a child that it must think, practically or fail. Now, if, throughout childhood, a brain has no opposition, it is plain that it will attain a position of "status quo," as with our ordinary animals. Man knows not why a cow, dog or lion was not born with a brain on a par with ours; why such animals cannot add, subtract, or obtain from books and schooling, that paramount position which Man holds today.

Lossless Compression Techniques

1 Model and code

The source is modelled as a random variable. The probabilities (statistics) are given or acquired.

2 Dictionary-based

There is no explicit model and no explicit statistics gathering. Instead, a codebook (or dictionary) is used to map sourcewords into codewords.

Model and Code

- Huffman code
- Tunstall code
- Fano code
- Shannon code
- Arithmetic code

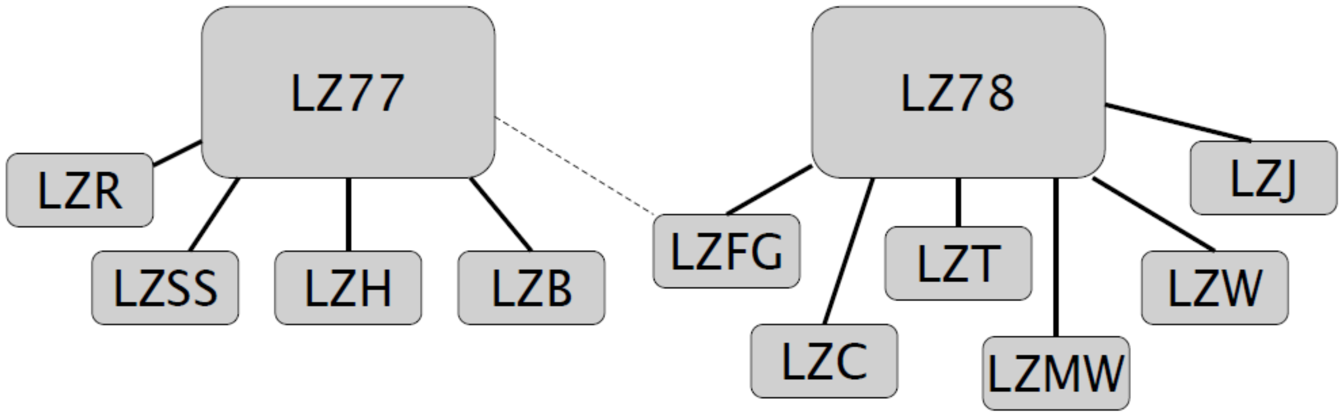
Dictionary-based Techniques

- Lempel-Ziv
  - LZ77 – sliding window
  - LZ78 – explicit dictionary
- Adaptive Huffman coding
- Due to patents, LZ77 and LZ78 have many variants

LZ77 Variants	LZR	LZSS	DEFLATE	LZH		
LZ78 Variants	LZW	LZC	LZT	LZMW	LZJ	LZFG

- Zip methods use LZH and LZR among other techniques
- UNIX compress uses LZC (a variant of LZW)

Lempel-Ziv Coding



Applications:
- zip
- gzip
- Stacker
- ...

Applications:
- GIF
- V.42
- compress
- ...

Lempel-Ziv Coding

- Source symbol sequences are replaced by codewords that are dynamically determined.
- The code table is encoded into the compressed data so it can be reconstructed during decoding.

Lempel-Ziv Example

Let X be a source of information for which we do not know the distribution $\mathbf{p}$. Suppose that we want to *source encode* the following sequence S generated by the source X:

$$S = 001000101110000011011010111101\ldots$$

$$S = \underbrace{00}_{S_3=00}\ 1000101110000011011010111101\ldots$$

$$S = 00\ \underbrace{10}_{S_4=10}\ 00101110000011011010111101\ldots$$

$$S = 0010\ \underbrace{001}_{S_5=001}\ 01110000011011010111101\ldots$$

$$S = 0010001\ \underbrace{01}_{S_6=01}\ 110000011011010111101\ldots$$

$$S = 001000101\ \underbrace{11}_{S_7=11}\ 0000011011010111101\ldots$$

$$S = 00100010111\ \underbrace{000}_{S_8=000}\ 0011011010111101\ldots$$

$$S = 00100010111000\ \underbrace{0011}_{S_9=0011}\ 011010111101\ldots$$

$$S = 001000101110000011\ \underbrace{011}_{S_{10}=011}\ 010111101\ldots$$

$$S = 001000101110000011011\ \underbrace{010}_{S_{11}=010}\ 111101\ldots$$

$$S = 001000101110000011011010\ \underbrace{111}_{S_{12}=111}\ 101\ldots$$

$$S = 001000101110000011011010111\ \underbrace{101}_{S_{13}=101}\ \ldots$$

Table 2.4: Example of a Lempel-Ziv code.

position	subsequence S_n		numerical representation	binary codeword
1	S_1	0		
2	S_2	1		
3	S_3	00	1 1	001 0
4	S_4	10	2 1	010 0
5	S_5	001	3 2	011 1
6	S_6	01	1 2	001 1
7	S_7	11	2 2	010 1
8	S_8	000	3 1	011 0
9	S_9	0011	5 2	101 1
10	S_{10}	011	6 2	110 1
11	S_{11}	010	6 1	110 0
12	S_{12}	111	7 2	111 1
13	S_{13}	101	4 2	100 1

Lempel-Ziv Codeword

$$S_C = 0010\ 0100\ 0111\ 0011\ 0101\ 0110\ 1011\ 1101\ 1100\ 1111\ 1001$$

Compression Comparison

Compression as a percentage of the original file size

File Type	UNIX Compact Adaptive Huffman	UNIX Compress Lempel-Ziv-Welch
ASCII File	66%	44%
Speech File	65%	64%
Image File	94%	88%

Compression Comparison

Compressed to (percentage):	Lempel-Ziv (unix gzip)	Huffman (unix pack)
html (25k) *Token based ascii file*	**20%**	65%
pdf (690k) *Binary file*	**75%**	95%
ABCD (1.5k) *Random ascii file*	33%	**28.2%**
ABCD(500k) *Random ascii file*	29%	**28.1%**

ABCD – $\{p_A = 0.5, p_B = 0.25, p_C = 0.125, p_D = 0.125\}$

Lempel-Ziv is asymptotically optimal