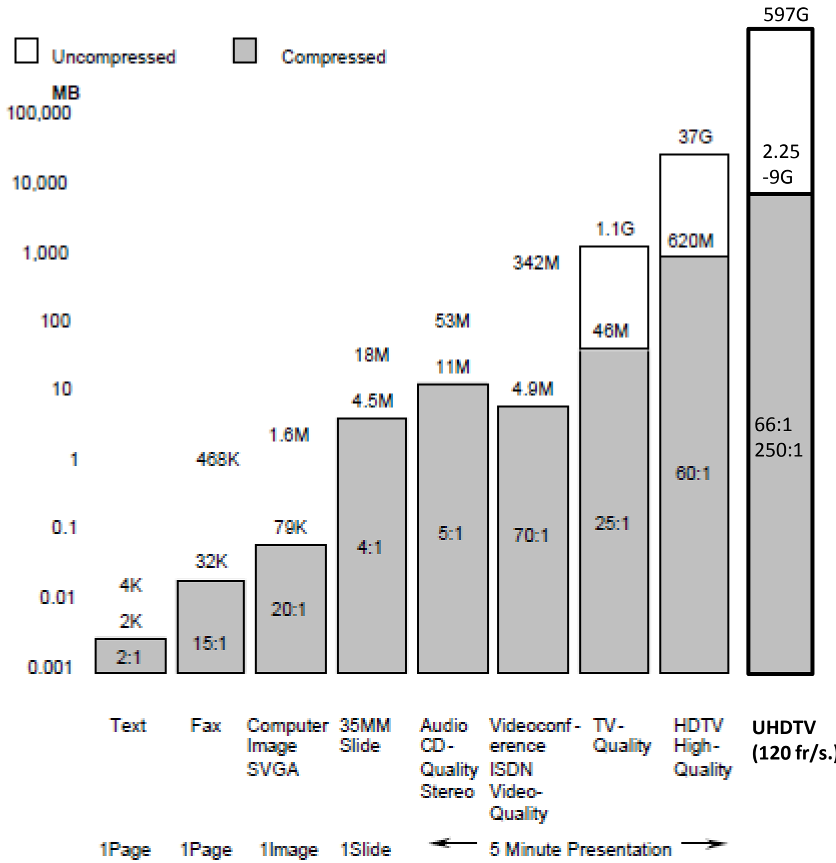
Compression

Prof. Ulug Bayazit

Outline

- Why compression?
- Classification
- Entropy and Information Theory
- Lossless compression techniques
  - Run Length Encoding
  - Variable Length Coding
  - Dictionary-Based Coding
- Lossy encoding

Why Compression



Complexity Example

- HDTV Quality Computer Display – very modest
  - 1920 x 1080 pixel image
  - 3 bytes per pixel (Red, Green, Blue)
  - 50 Frames per Second
- Bandwidth requirement
  - 311 MBps
  - USB 3.0 : 640MBps
- Storage requirement
  - CD-ROM would hold 2 seconds worth
  - 30 minutes would require 560 GB!
- Some form of compression required!

Widely Accepted Compression Methods

- JPEG/JPEG2000 for still images
- MPEG (1/2/4), H.264/AVC, H.265 (HEVC) for audio/video playback and VoD (retrieval mode applications)
- DVI (Digital Video Interactive) for still and continuous video applications (two modes of compression)
  - 1987 David Sarnoff Research Center, Princeton NJ
  - Presentation Level Video (PLV) - high quality compression, but very slow. Suitable for applications distributed on CD-ROMs
  - Real-time Video (RTV) - lower quality compression, but fast. Used in video conferencing applications.
  - ADPCM (Adaptive Differential PCM) for audio compression
- VP 7/8/9/10
  - Google video compression format
  - VP8 2008, VP9 2013 (Android 4.4 +), VP10 current

Compression Types

- Application purpose
  - Save storage space
  - Reduce communications capacity requirements.
- Data fidelity
  - Lossless
    - No information is lost.
    - Decompressed data is identical to the original uncompressed data.
    - Mainly for text files (e-mail), databases, binary object files, etc. where distortion is not permitted
  - Lossy
    - Approximation of the original data.
    - Better *compression ratio* than lossless compression.
    - Tradeoff between compression ratio and fidelity.
    - e.g., Audio, image, and video.

Compression Types

- *Entropy Coding*
  - Lossless encoding
  - Based on statistics of media data
  - Data taken as a simple digital sequence
  - Decompression process regenerates data completely
  - e.g., run-length coding, Huffman coding, Arithmetic coding
- *Source Coding*
  - Lossy encoding
  - Takes into account the semantics of the data
  - Degree of compression depends on data content
  - e.g., content prediction technique - DPCM, delta modulation , vector quantization
  - Alternatively some form of transform coding (Karhunen Loeve, discrete cosine, wavelet)
- Hybrid Coding (used by most multimedia systems)
  - Combine entropy coding with source encoding
  - e.g., JPEG, JPEG2000, Motion JPEG, H.261,H.263, DVI (RTV & PLV), MPEG-1, MPEG-2, MPEG-4 , H.264 AVC, H.265 (HEVC), VPx

Codes/Coding

- Mapping of letters from one alphabet to another
  - ASCII codes: computers do not process English language directly. Instead, English letters are first mapped into bits, then processed by the computer.
  - Morse codes
  - Decimal to binary conversion
- Process of mapping from letters in one alphabet to letters in another is called coding.

Codes/Coding

- Code is a simple mapping.
- Code $C : X \mapsto D^+$

$$
\begin{aligned}
&C : Code \\
&X : Input\ alphabet \\
&D : Output\ alphabet \\
&D^+ : Set\ of\ \ finite\ \ strings\ \ from\ D
\end{aligned}
$$

- Example

$$
C : \{a, b, c\} \mapsto \{0,1\}^+
$$

$$
\left.\begin{aligned}
C(a) &= 0 \\
C(b) &= 10 \\
C(c) &= 100
\end{aligned}\right\} codewords(code)
$$

Codes

- Extension: $C^+ : X^+ \mapsto D^+$
  - C^+ is formed by concatenating $C(x_i)\ \mathrm{i} = 1,\ldots,\mathrm{n}$
  - C(aabacb)=0010010010
  - Nonsingular: $x_1 \neq x_2 \Rightarrow C(x_1) \neq C(x_2)$
  - Uniquely decodable:
    - C^+ is nonsingular (given $C^+(x_1 \ldots x_n)$, $x_1 \ldots x_n$ is unambiguous)

Prefix codes

- Instantaneous or Prefix code:
  - No codeword is a prefix of another
- Prefix → uniquely decodable
- Examples:

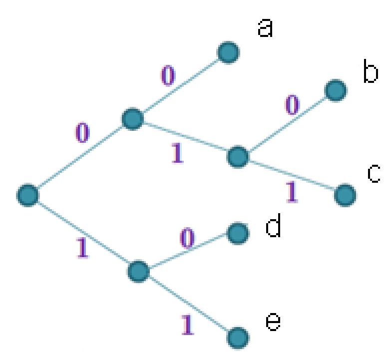
U: Uniquely decodable
P: Prefix
N:Neither

	C1	C2	C3	C4
W	0	00	0	0
X	11	01	10	01
Y	00	10	110	011
Z	01	11	1110	0111
	N	U	P	U

Code Tree (Variable length coding)

- Form a *M-ary tree* where $M = |D|$
  - M branches at each node
  - Each branch denotes a (output) symbol $d_i \in D$
  - Each leaf denotes a source symbol $x_i \in X$
  - $C(x_i)$: concatenation of symbols x_i along the path from the root to the leaf that represents x_i
  - Some leaves may be unused

C(a)=00
C(b)=010
C(c)=011
C(d)=10
C(e)=11



Entropy

- Motivation: Exploit statistical redundancies in the data to be compressed.
- The average information (uncertainty) per symbol in a file, called *entropy H,* is defined as

$$H = \sum_{i=1}^{K} p_i \log_2 \frac{1}{p_i}$$

where p_i is the probability of i’th distinct source symbol.

- Entropy is the lower bound for lossless compression, i.e., when the occurring probability of each source symbol is fixed, each source symbol should be represented with at least ***H bits on the average.***

Entropy

- The closer the compressed information in bits per symbol to entropy, the better the compression.
- For example, in an image with uniform distribution of gray-level intensity, i.e., $p_i = \frac{1}{256}$, the number of bits needed to code each gray level is 8 bits. The entropy of this image is 8.
  - No compression possible in this case!
- For an optimal code, the length of a codeword *i*, l_i, satisfies $\log_2(1/p_i) \le l_i \le \log_2(1/p_i) + 1$
- Hence $H(X) \le E[L] = \sum_i p_i l_i \le H(X) + 1$

Optimum entropy codes

- In compression, we are interested in constructing an *optimum code, i.e., one that gives the minimum average length for the encoded message.*
  - Easy if the probability of occurrence for all the symbols are the same, e.g., last example.
  - But what if symbols have different probability of occurrence
    - Huffman and Arithmetic Coding

Lossless Compression Techniques

- Run length encoding (RLE)
- Variable Length Coding (VLC)
  - Huffman encoding
  - Arithmetic encoding
- Dictionary-Based Coding
  - Lempel-Ziv-Welch (LZW) encoding

Run-Length Encoding

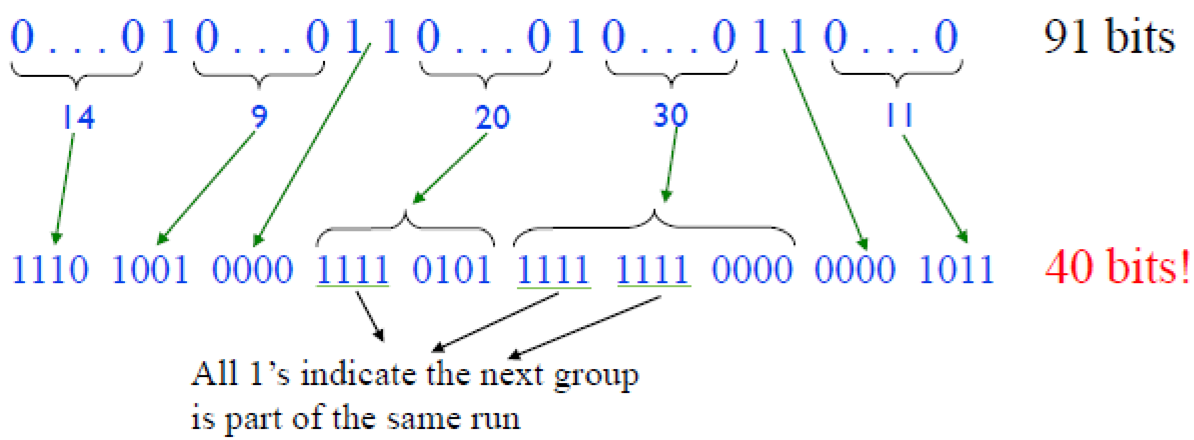
- Redundancy is exploited by not transmitting consecutive symbols that are equal.
- The value of long runs is coded once, along with the number of times the value is repeated (length of the run).
- Particularly useful for encoding black and white images where the input alphabet is {0,1}, e.g., fax.
- Methods
  - Binary RLE
  - Packbits

Binary RLE

- Transmit length of each run of 0's (or no transitions).
- Do not transmit runs of 1's (transitions).
- For a run length $n \times 2^k - 1 + b$, where k is the number of bits used to encode the run length
  - Transmit n x k 1's, followed by the value b encoded in k bits.
- Two consecutive 1's (transitions) are implicitly separated by a zero-length run.
- Appropriate for bi-level images, e.g., FAX.

Binary RLE

- Suppose 4 bits are used to represent the run length



- If symbol stream had started with 1 codestream would start with 0000.

Binary RLE

- Worst case behavior:
  - If each 1 (or transition) requires k-bits to encode, and we have a sequence of 1's (or transitions), k-1 of those bits are wasted. Bandwidth expanded by k times!
- Best case behavior:
  - A sequence of $2^k - 1$ consecutive 0 bits (no transitions) can be represented using k bits. Compression ratio is roughly $\left(2^k - 1\right)/k$
    - e.g., if k = 8, $\left(2^k - 1\right)$= 255
    - 255:8 = 32:1 compression (lossless).

RLE - Packbits

- Used by Apple Macs.
- One of the standard forms of compression available in TIFF format (TIFF compression type 32773).
- TIFF = Tag Image File format.
- Each run is encoded into bytes as follows:

Header byte (value n)	Data following the header byte
0 to 127	(n+1) literal bytes of data (literal run)
-1 to -127	1 byte of data repeated 1-n times in the decompressed output (fill run)
-128	Skip and treat next byte as header

Packbits

- Example of Literal run

  <21>, M, a, r, y, , h, a, d, , a, , l, i, t, t, l, e, , l, a, m, b

- Example of Fill run

  C("AAAABBBBB")=<253>A<252>B

- Example of Fill run

  C("ABCCCCCCCCCDEFFFFGGG")=
  <0>A<0>B<248>C<0>D<0>E<253>F<254>G

Packbits Performance

- Worst case behavior
  - If entire image is transmitted without compression, the overhead is a byte (for the header byte) for each 128 bytes.
  - Alternating runs add more overhead
  - Poorly designed encoder could do worse.
- Best case behavior
  - If entire image is redundant, then 2 bytes are used for each 128 consecutive bytes. This provides a compression ratio of 64:1.
- Achieves a typical compression ratio of 2:1 to 5:1.

Huffman Coding

- Input values are mapped to output codewords of varying length, called *variable-length code (VLC):*
- Most probable inputs are coded with fewer bits.
- No code word can be prefix of another code word.

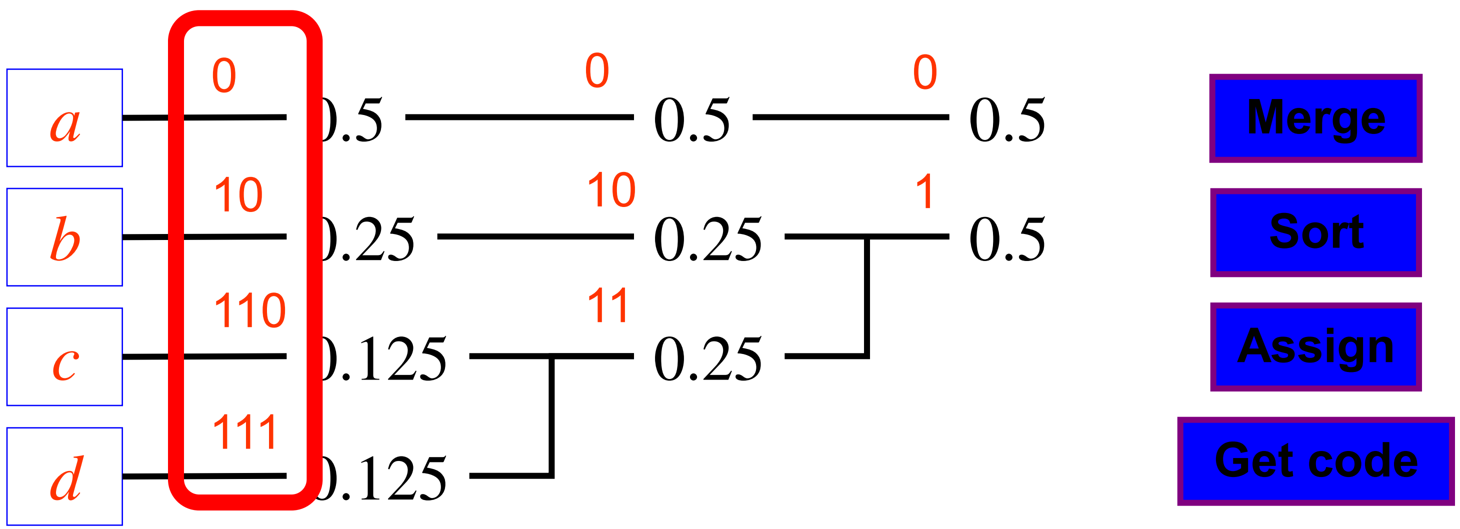
Huffman Code Construction

- Initialization: Put all nodes (values, characters, or symbols) in a sorted list.
- Repeat until the list has only one node:
  - Combine the two nodes having the lowest frequency (probability). Create a parent node assigning the sum of the children's probabilities and insert it in the list.
  - Assign code 0, 1 to the two branches, and delete children from the list.

Huffman Coding Example

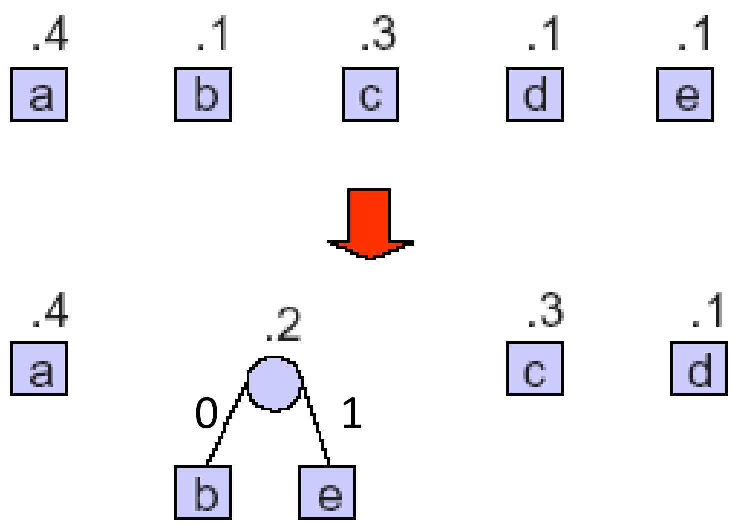
- Two-step algorithm:
  1. Iterate:
     - Merge the least probable symbols.
     - Sort.
  2. Assign bits.

$$P(a)=0.5, P(b)=0.25$$
$$P(c)=0.125, P(d)=0.125$$

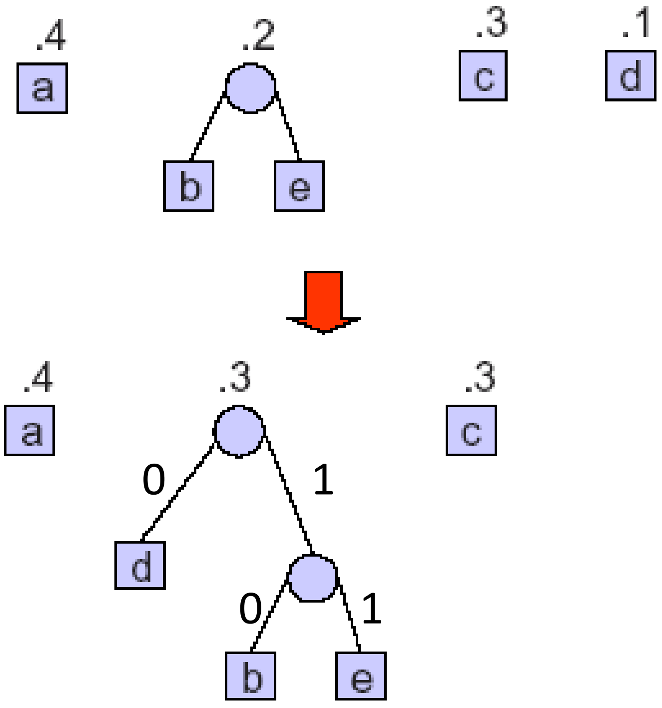


More Examples of Huffman Code

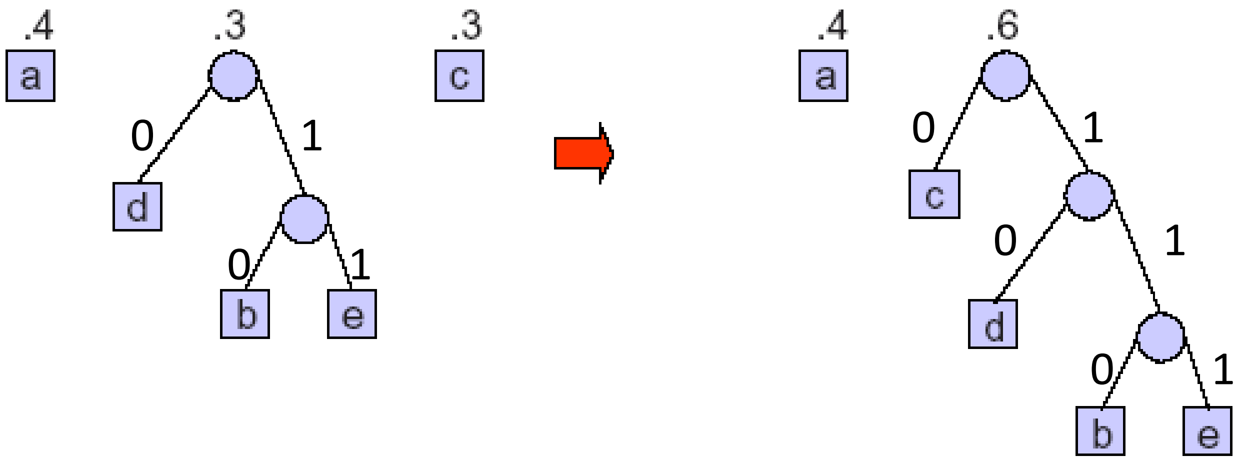
- P(a) =.4, P(b)=.1, P(c)=.3, P(d)=.1, P(e)=.1



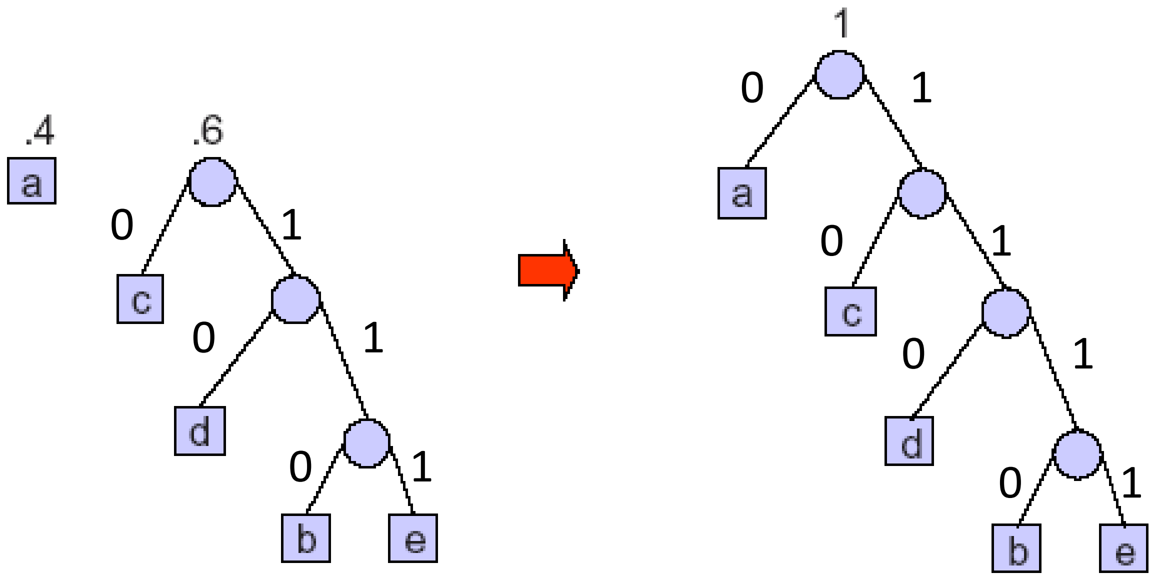
More Examples of Huffman Code



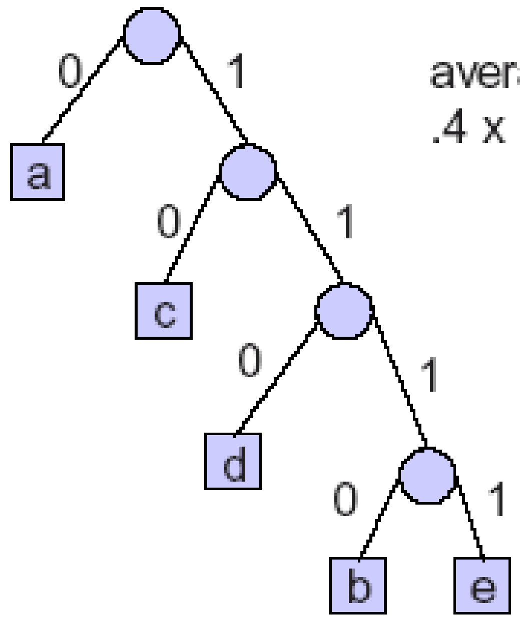
More Examples of Huffman Code



More Examples of Huffman Code



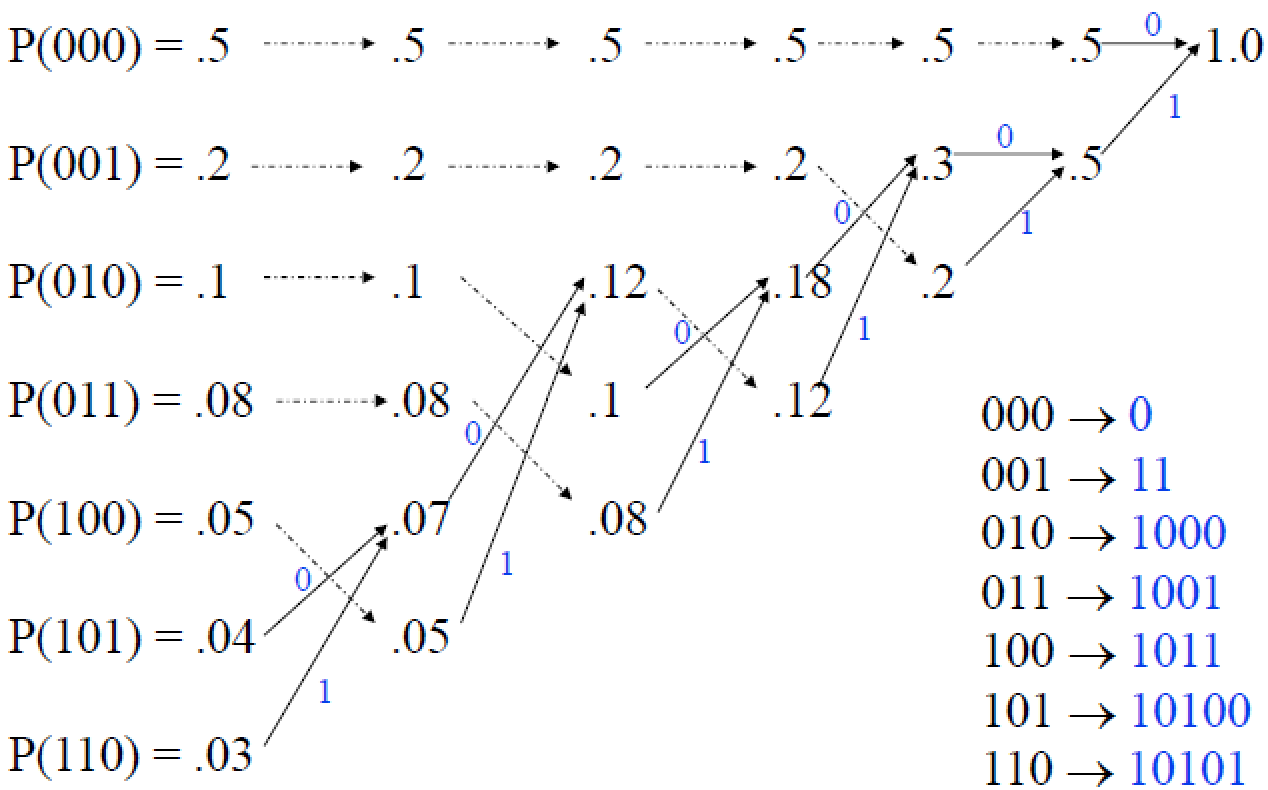
Average Huffman Code Length



average number of bits per symbol is $E[L] = \sum_{i=1}^{N} p_i l_i$
.4 x 1 + .1 x 4 + .3 x 2 + .1 x 3 + .1 x 4 = 2.1

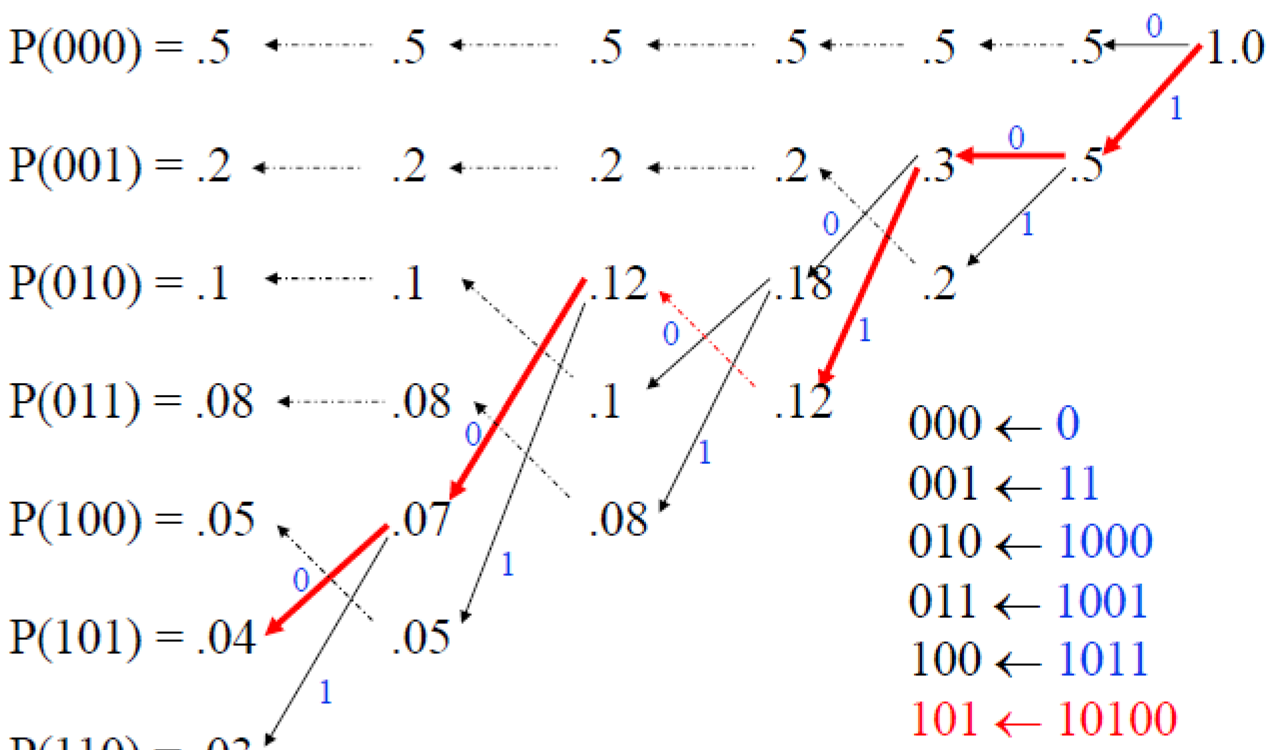
letter	codeword
a	0
b	1110
c	10
d	110
e	1111

Example of Huffman Coding



000 → 0
001 → 11
010 → 1000
011 → 1001
100 → 1011
101 → 10100
110 → 10101

Example of Huffman Decoding



000 ← 0
001 ← 11
010 ← 1000
011 ← 1001
100 ← 1011
101 ← 10100
110 ← 10101

Efficiency of Huffman Coding

- *E[L] :* Average number of bits *used to encode a symbol*
- Fixed length coding: each symbol coded with $\lceil \log_2(|X|) \rceil = 3$ bits
  - E[L]=3 x (0.5 +0.2 +0.1 +0.08 +0.05 +0.04 +0.03) = 3 bits/symbol
- Huffman coding:
  - E[L]=1 x 0.5 + 2 x 0.2 + 4 x 0.1 + 4 x 0.08 +4 x 0.05 + 5 x 0.04 + 5 x 0.03 = 2.17 bits/symbol
- Entropy H of the previous example:

  0.5 + 0.2 x2.32 + 0.1 x 3.21 + 0.08 x 3.64 + 0.05 x4.32 + 0.04 x 4.64 + 0.03 x 5.06 = 2.129 bits/symbol

- Efficiency of VLC code is 2.129/2.17=0.981. Very close to optimum!

- So Huffman code is better!

Adaptive Huffman Codes

- Fixed Huffman tree designed from training data
  - Do not have to transmit the Huffman tree because it is known to the decoder.
  - H.263 video coder
  - Weak when statistics used in design do not match statistics of coded data
- Semi-adaptive Huffman code
  - Must transmit the Huffman code or frequencies as well as the compressed input.
  - Requires two passes
- Adaptive Huffman code
  - One pass
  - Huffman tree changes on the fly as frequencies change

Adaptive Huffman Coding Algorithm

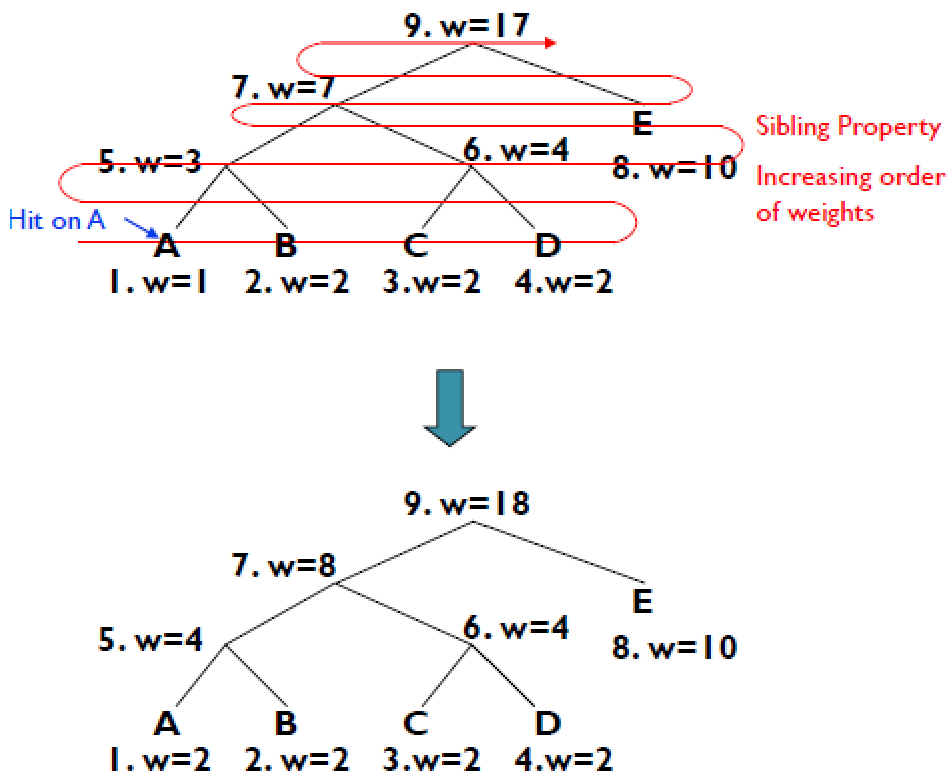
Encoder

```
initial_code();
  while ((c=getc(input))!=eof {
       encode (c,output);
       update_tree(c);
  }
```

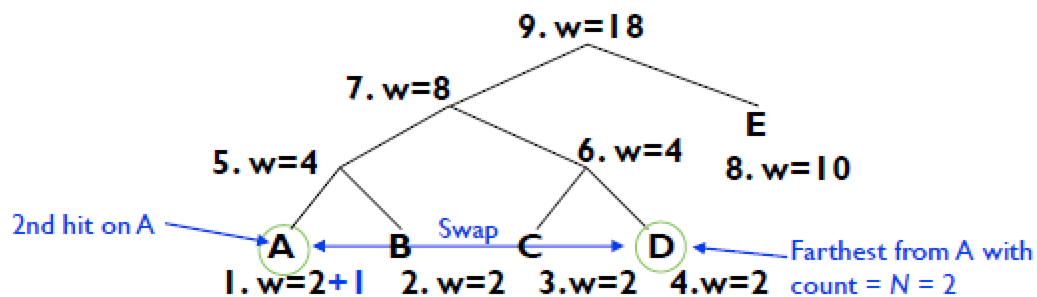
Decoder

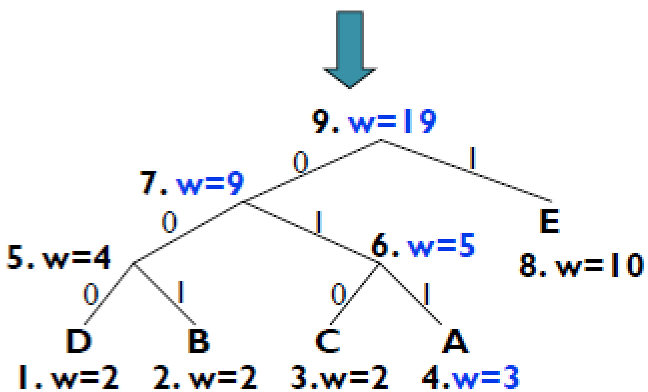
```
initial_code();
  while ((c=decode(input)) !=eof){
       putc (c,output);
       update_tree(c);
  }
```

- **`initial_code()`** assigns symbols with some initially agreed-upon codes without knowing the frequency counts for them.
  - e.g., ASCII
- **`update_tree()`** constructs an adaptive Huffman tree.
  - Huffman tree must maintain its sibling property
    - All nodes (internal and leaf) are arranged in the order of increasing counts (weights)=> from left to right, bottom to top.
  - When swap is necessary, the farthest nodes with count (weight) N is swapped with the nodes whose count (weight) has just been increased to N+1.

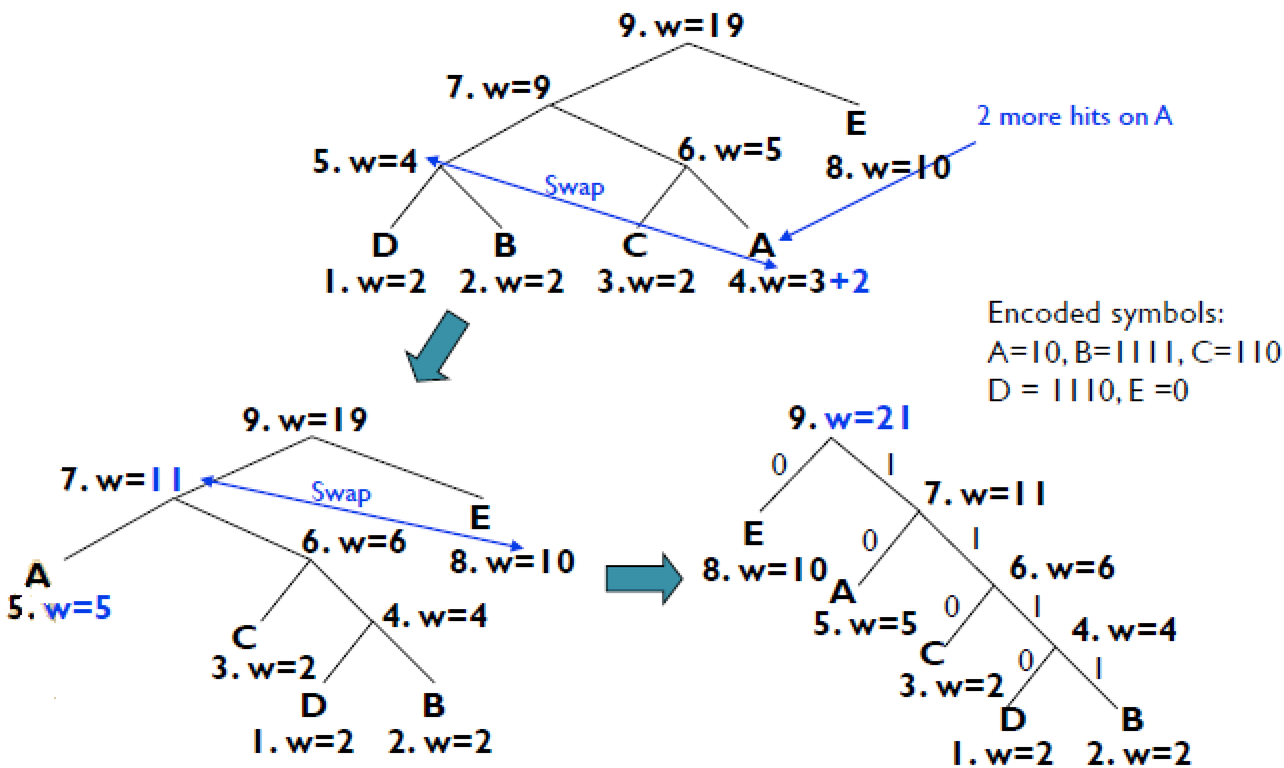


9. w=17
7. w=7
E
Sibling Property
5. w=3
6. w=4
8. w=10
Increasing order
of weights
Hit on A
A
B
C
D
1. w=1
2. w=2
3.w=2
4.w=2
9. w=18
7. w=8
E
5. w=4
6. w=4
8. w=10
A
B
C
D
1. w=2
2. w=2
3.w=2
4.w=2





Encoded symbols:
A=011, B=001, C=010
D = 000, E =1



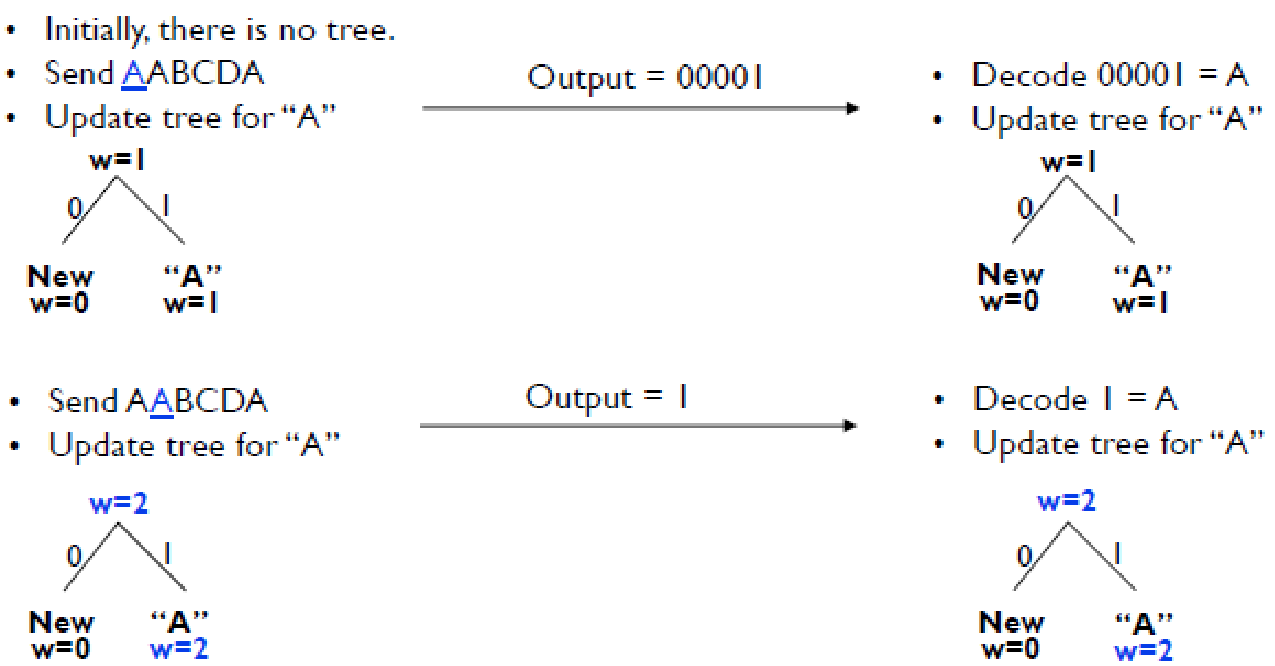
9. w=19
7. w=9
E
8. w=10
2 more hits on A
5. w=4
6. w=5
Swap
D
1. w=2
B
2. w=2
C
3.w=2
A
4.w=3+2
Encoded symbols:
A=10, B=1111, C=110
D = 1110, E =0
9. w=19
7. w=11
Swap
E
8. w=10
6. w=6
A
5. w=5
C
3. w=2
4. w=4
D
1. w=2
B
2. w=2
9. w=21
0
1
E
8. w=10
7. w=11
0
1
A
5. w=5
6. w=6
0
1
C
3. w=2
4. w=4
0
1
D
1. w=2
B
2. w=2

Adaptive Huffman Coding - Initialization

- Initially there is no tree.
- Symbols sent for the first time use fixed code:
  - e.g., ASCII
- Any new node spawned from “New”.
- Except for the first symbol, any symbol sent afterwards for the first time is preceded by a special symbol “New”.
  - Count for “New” is always 0.
  - Code for symbol “New” depends on the location of the node “New” in the tree.
- Code for symbol sent for the first time is
  - code for “New” + fixed code for symbol
- All other codes are based on the Huffman tree.

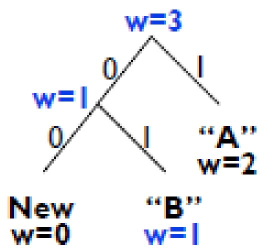
Initialization Example

- Suppose we are sending AABCDAD…
  - Fixed code: A = 00001, B = 00010, C = 00011, D = 00100, …
  - “New” = 0



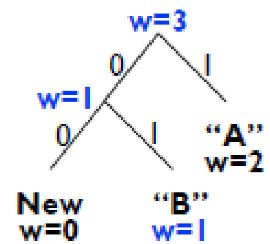
Initialization Example

- Send AABCDA
- Update tree for "B"

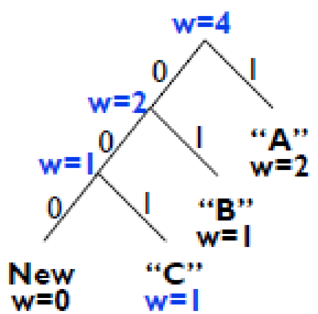


Output = 0 00010

- Decode 0 00010 = B
- Update tree for "B"

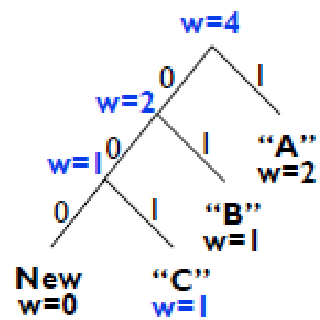


- Send AABCDA
- Update tree for "C"

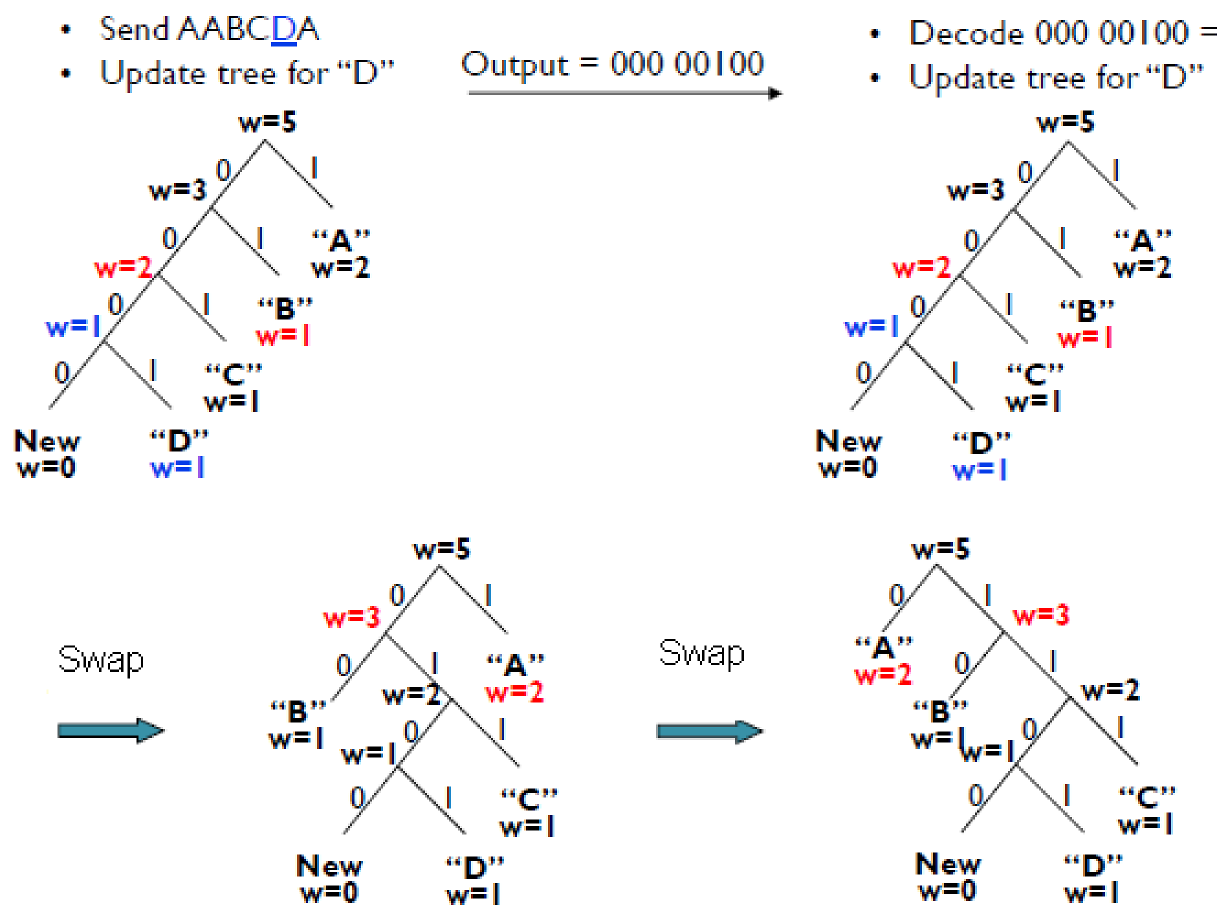


Output = 00 00011

- Decode 00 00011 = C
- Update tree for "C"

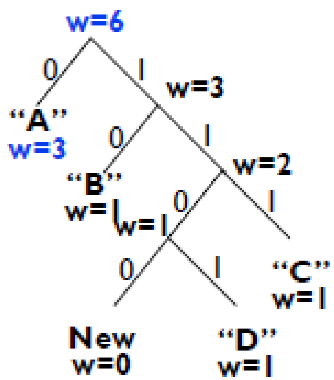


Initialization Example



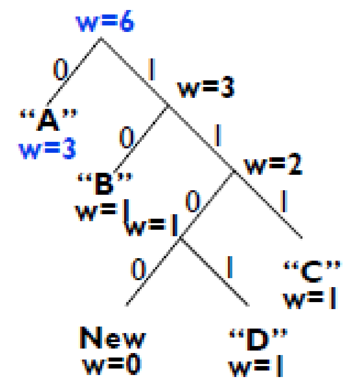
Initialization Example

- Send AABCDA
- Update tree for "A"



Output = 0

- Decode 0 = A
- Update tree for "A"



Final transmitted code

A A B C D A

00001 1 000010 0000011 00000100 0

Pros and cons of Huffman Coding

- Pros
  - Optimality, i.e., shortest possible symbol code
- Cons
  - $E[L]<H(X)+1$: for small alphabets ($H(X)$) too large redundancy
  - Poor compression for skewed probability
  - Employing block coding helps, i.e., consider combining X_N of N symbols in a block into a single symbol.
    - For N iid symbols $E[L_N]<H_N(X_N)+1$ => For a single symbol $E[L]<H(X)+1/N$
  - However,
    - must re-compute the entire table of block symbols if the probability of a single symbol changes.
      - For *N symbols, a table of $|X|N$ must be pre-calculated to build the tree*
    - Symbol decoding must wait until an entire block is received

Block Huffman Code

$$A = \{a_{1,} a_2, \ldots a_m\}, A^n = \{\underbrace{a_1 a_1 \ldots a_1}_{\text{n times}}, a_1 a_1 \ldots a_2, \ldots, a_m a_m \ldots a_m\}$$

m^n symbols in the A^{n} alphabet

$$H(\underline{X}) \le \bar{l} < H(\underline{X}) + 1$$

$$H_n(\underline{X}) \le \bar{l} < H_n(\underline{X}) + 1/n$$

$$H(X) \le \bar{l} < H(X) + 1/n \quad \text{for i.i.d. sources}$$

$\bar{l}$: Average length of Huffman Code

$H(S)$: Entropy of the source

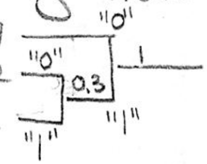
Block Huffman Coding Example

Letter	Prob
a_1	0.7
a_2	0.2
a_3	0.1

$\Big\}\ H(X) = 1.15678$ bits

Huffman coding of individual source symbols

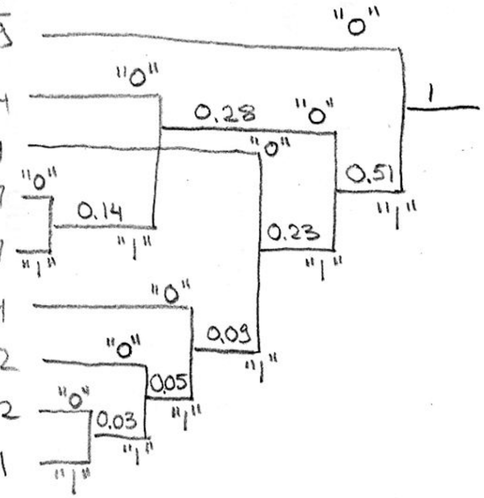
a_1	0.7
a_2	0.2
a_3	0.1



$$\bar{\ell} = E[\ell] = (0.7)(1) + (0.2)(2) + (0.1)(2) = 1.3\ \frac{\text{bits}}{\text{sample}}$$

Huffman coding of block of source symbols

Block	Prob
$a_1\, a_1$	0.49
$a_1\, a_2$	0.14
$a_2\, a_1$	0.14
$a_1\, a_3$	0.07
$a_3\, a_1$	0.07
$a_2\, a_2$	0.04
$a_2\, a_3$	0.02
$a_3\, a_2$	0.02
$a_3\, a_3$	0.01



$$\bar{\ell} = E[\ell] = (0.49)(1) + (0.14)(3) + (0.14)(3) + (0.07)(4)$$
$$+ (0.07)(4) + (0.04)(4) + (0.02)(5)$$
$$+ (0.02)(6) + (0.01)(6)$$
$$= 2.33\ \frac{\text{bits}}{\text{2 source samples}} = 1.165\ \frac{\text{bits}}{\text{sample}}$$

$$H(\underline{X}) = 2.31356\ \text{bits} \qquad = 1.15678\ \frac{\text{bits}}{\text{sample}}$$

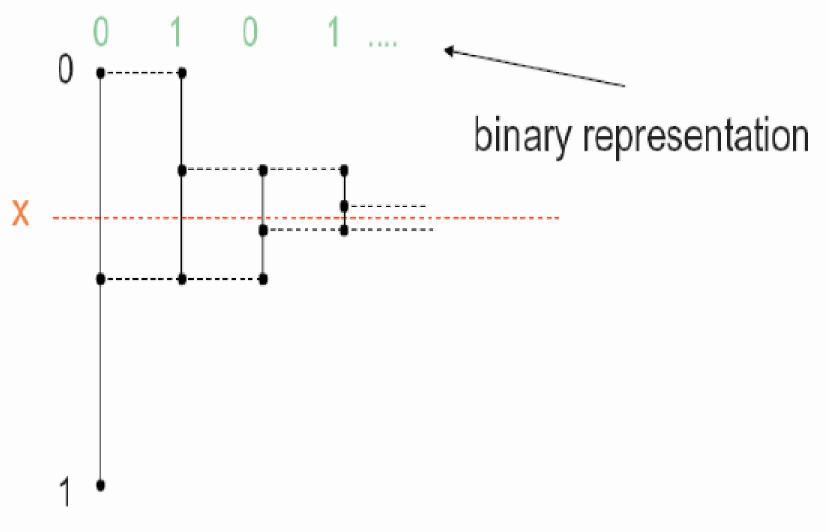
Arithmetic coding

- Need to encode long strings without worrying about code size!

Representation of Real Number in Binary

- Any real number x in the interval [0,1) can be represented in binary as

  $.b_1 b_2 \ldots$ where b_i is a bit



Real-to-Binary Conversion Algorithm

```
L := 0; R :=1; i := 1
while x > L *
    if x < (L+R)/2 then b_i := 0 ; R := (L+R)/2;
    if x ≥ (L+R)/2 then b_i := 1 ; L := (L+R)/2;
    i := i + 1
end{while}
b_j := 0 for all j ≥ i
```

* Invariant: x is always in the interval [L,R)

- The last step results in x to be approximated by L

Arithmetic Coding

- Developed by IBM (too bad, they didn't patent it)
  - Majority of current compression schemes use it
- Good:
  - No need for pre-calculated tables of big size
  - Easy to adapt to change in probabilities of symbols
  - Single symbol can be decoded immediately without waiting for the entire block of symbols to be received.
- Each symbol is coded by considering prior data
  - Relies on the fact that coding efficiency can be improved when symbols are combined.
  - Yields a single code word for each string of characters.
  - Each symbol is a portion of a real number between 0 and 1.

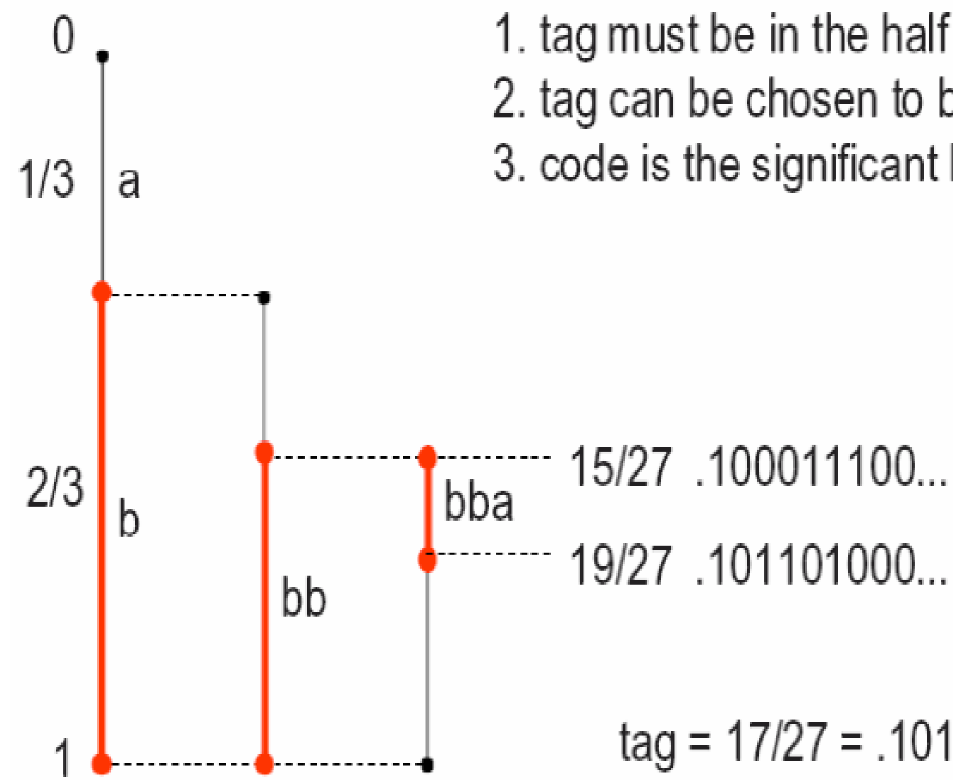
Idea of Arithmetic Coding

- Basic idea (Shannon-Fano-Elias):
  - Confine each string x of length n to a unique interval [L,R) in [0,1).
  - The width R-L of the interval [L,R) corresponds to the probability of x occurring in this interval
  - The string can be mapped to and approximated by any number, called a tag, within the half open interval [L,R) .
  - The k significant bits of the tag $.t_1t_2t_3...$ is the code of x. That is,

    . $.t_1t_2t_3...t_k000...$ is in the interval [L,R).

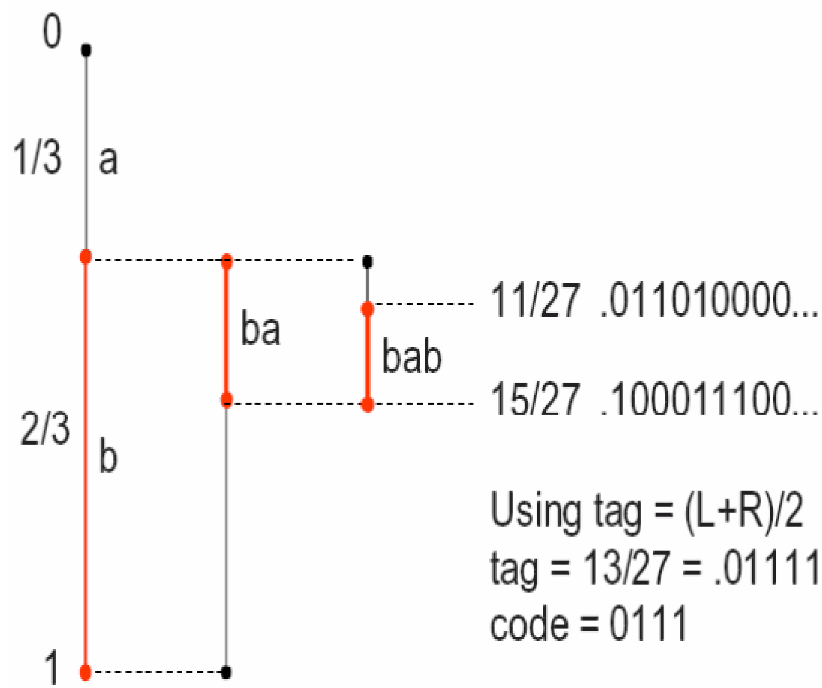
Example of Arithmetic Coding

1. tag must be in the half open interval.
2. tag can be chosen to be (L+R)/2.
3. code is the significant bits of the tag.



tag = 17/27 = .101000010...
code = 101

Some Tags are better than others



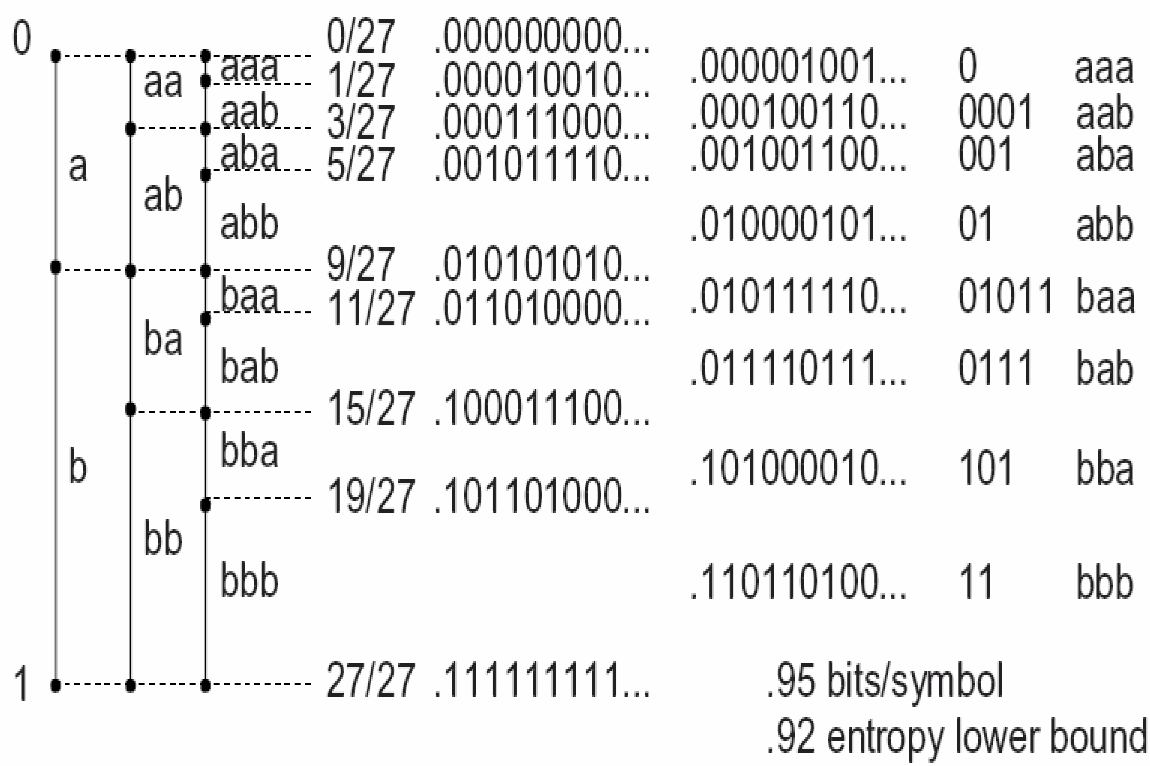
Using tag = (L+R)/2
tag = 13/27 = .011110110...
code = 0111

Alternative tag = 14/37 = .100001001...
code = 1

Examples

- $P(a) = 1/3$, $P(b) = 2/3$.

tag = (L+R)/2 code



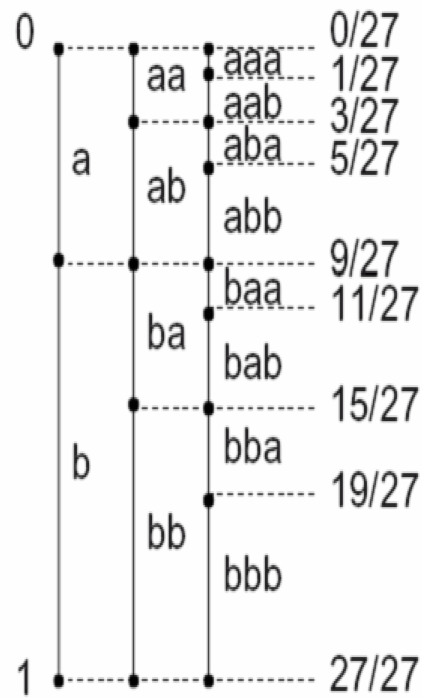
Code Generation from Tags

- If binary tag is $.t_1t_2... = (L+R)/2$ in [L,R) then we want to choose k to form the code $t_1t_2...t_k$
- Short code
  - Choose k as small as possible so that $L \le .t_1t_2...t_k000... < R$
- Guaranteed code
  - Choose $k = \left\lceil \log_2\left(\frac{1}{R-L}\right) \right\rceil + 1$
  - $L \le .t_1t_2...t_kb_1b_2b_3... < R$ for any bits $b_1b_2b_3$
  - Example: [.000000000..., .000010010...), tag = .000001001...

Short code: 0 Guaranteed code: 000001 (k=6)

Guaranteed code example

- P(a)=1/3, P(b)=2/3
-



tag = (L+R)/2	short code	Prefix code	
.000001001...	0	0000	aaa
.000100110...	0001	0001	aab
.001001100...	001	001	aba
.010000101...	01	0100	abb
.010111110...	01011	01011	baa
.011110111...	0111	0111	bab
.101000010...	101	101	bba
.110110100...	11	11	bbb

Algorithm pseudocode

- Cumulative probabilities $C(x_i) = \sum_{l=1}^{i} P(x_i)$
- Initialize L=0, R=1
- For k=1 to K do
  - W=R-L
  - Read next symbol as x_i
  - Update $L = L + W * C(x_{i-1})$

    $R = L + W * C(x_i)$
- Tag: T=(L+R)/2
- Code: Truncate T to $k = \left\lceil \log_2 (\frac{1}{R-L}) \right\rceil + 1$ bits

Example

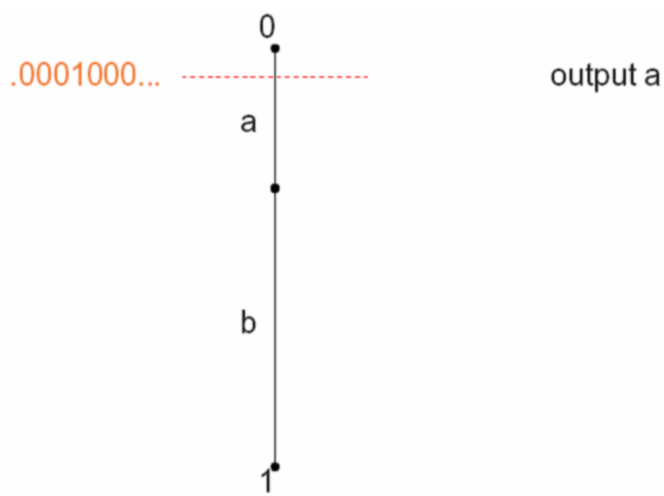
- P(a)=0.25, P(b)=0.5,P(c)=0.25
- C(a)=0.25,C(b)=0.75,C(c)=1
- Source sequence ''abca''

symbol	W	L	R
		0	1
a	1	0	1/4
b	1/4	1/16	3/16
c	1/8	5/32	6/32
a	1/32	5/32	21/128

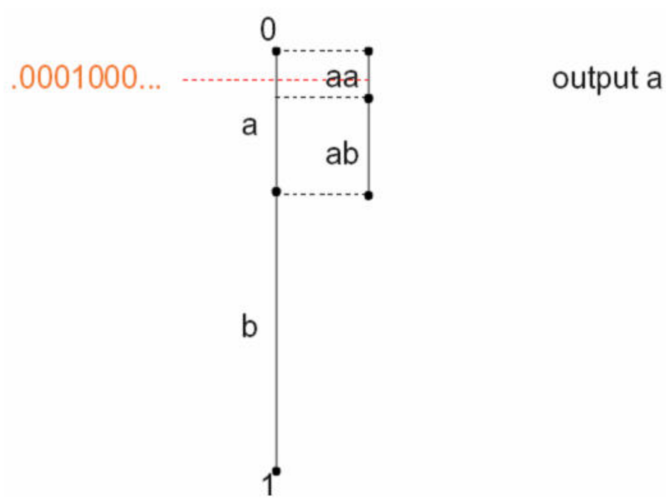
tag = (5/32 + 21/128)/2 = 41/256 = .001010010...
L = .001010000...
R = .001010100...
code = 00101
prefix code = 00101001

Decoding

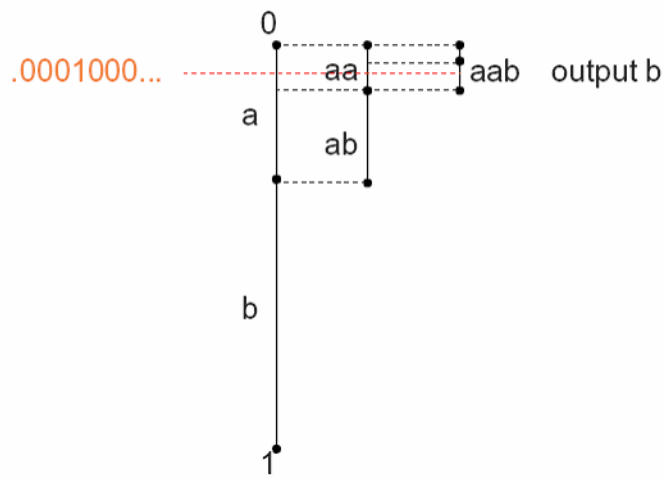
- Assume the length of the source sequence is known to be 3
- Code received: 0001 => Tag to be decoded: .0001000...



Decoding (cont.)



Decoding (cont.)



Arithmetic Decoding Algorithm

- Decoding binary sequence $b_1,\ldots,b_m$
- Let $t = .b_1,\ldots,\ b_m$
- Initialize L=0, R=1
- For k=1 to K do
  - W=R-L
  - Find j such that $L+W*C(x_{j-1}) \le t < L+W*C(x_j)$
  - Output x_j
  - Update $L = L+W*C(x_{j-1})$

    $R = L+W*C(x_j)$

Decoding example

- P(a)=0.25, P(b)=0.5,P(c)=0.25
- C(a)=0.25,C(b)=0.75,C(c)=1
- Code received: 00101

tag = .00101000... = 5/32

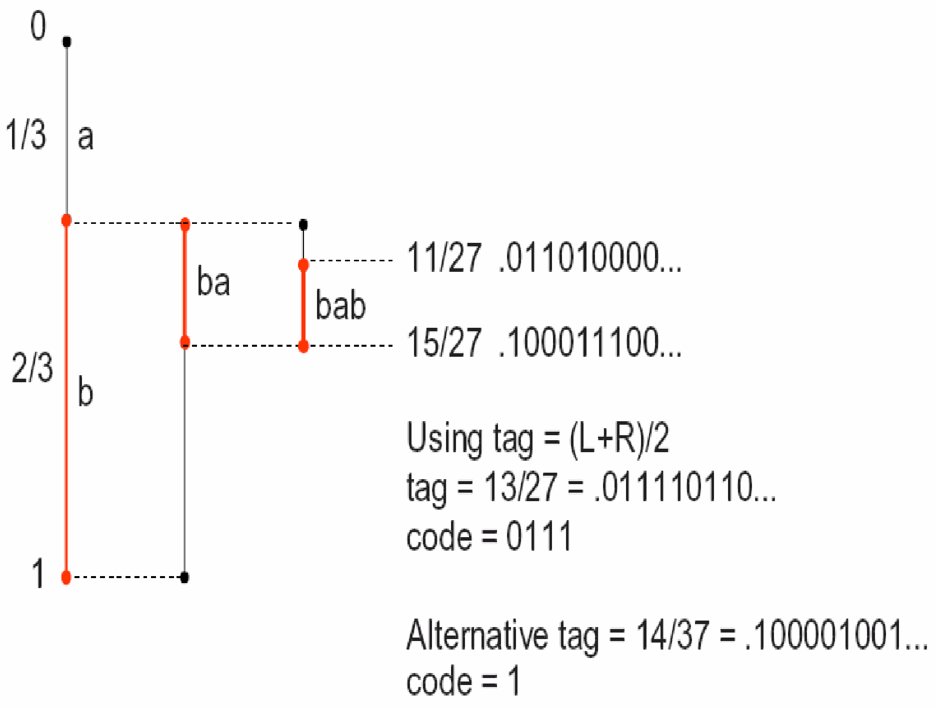
W	L	R	output
	0	1	
1	0	1/4	a
1/4	1/16	3/16	b
1/8	5/32	6/32	c
1/32	5/32	21/128	a

Decoding issues

- There are two ways for the decoder to know when to stop decoding.
  - Transmit the length of the string
  - Transmit a unique end of string symbol

Issues with Arithmetic Coding

- The intervals are getting smaller as the sequence of symbols is getting longer.
- Arithmetics (computations) on very small numbers results in underflow!
- Need to rescale at every step!

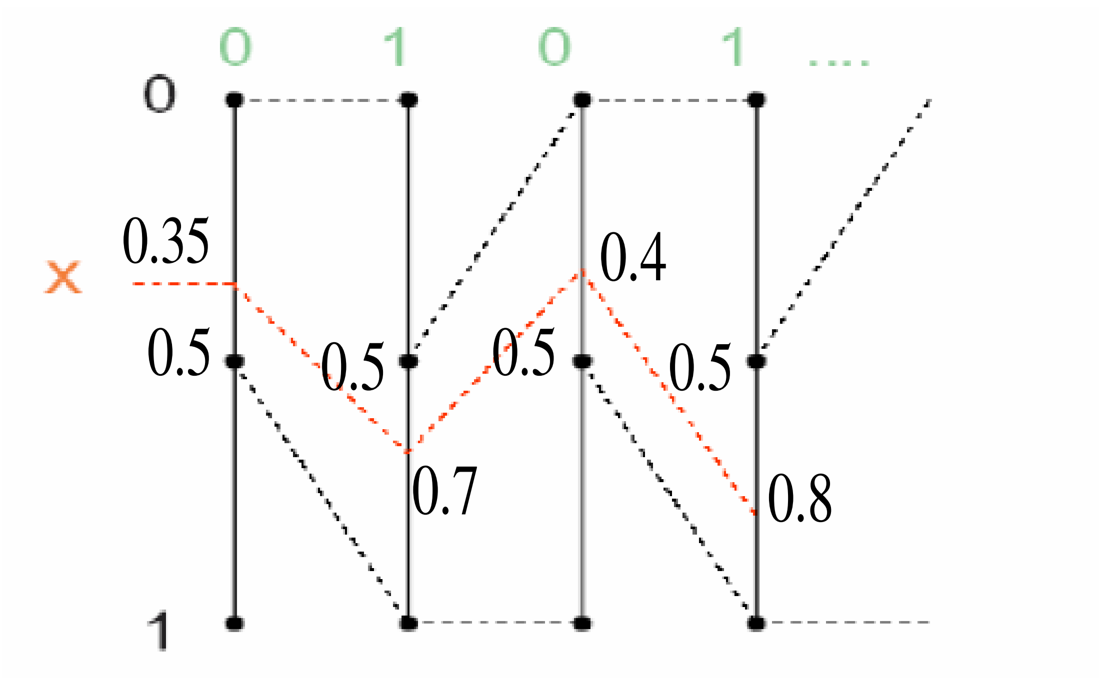


Practical Arithmetic Coding

- Scaling:
  - By scaling we can keep L and R in a reasonable range of values so that W = R - L does not underflow (need for too much precision avoided).
  - The code can be produced progressively, not at the end.
  - Complicates decoding slightly.
- Integer arithmetic coding avoids floating point altogether

Representation of Real Number in Binary

- Always scale the interval to unit size, but x must be changed as part of scaling



Code generation algorithm based on scaling

```
y := x; i := 0
while y > 0 *
    i := i + 1;
    if y < 1/2 then b_i := 0; y := 2y;
    if y ≥ 1/2 then b_i := 1; y := 2y – 1;
end{while}
b_j := 0 for all j ≥ i + 1
```

* Invariant: $x = .b_1 b_2 \ldots b_i + y/2^i$

Proof of Invariant

- Initially $x = 0 + y/2^0$
- Assume $x = .b_1b_2 \ldots b_i + y/2^i$
  - Case 1. $y < 1/2$. $b_{i+1} = 0$ and $y' = 2y$

$$
\begin{aligned}
.b_1b_2 \ldots b_i b_{i+1} + y'/2^{i+1} &= .b_1b_2 \ldots b_i 0 + 2y/2^{i+1} \\
&= .b_1b_2 \ldots b_i + y/2^i \\
&= x
\end{aligned}
$$

  - Case 2. $y \geq 1/2$. $b_{i+1} = 1$ and $y' = 2y - 1$

$$
\begin{aligned}
.b_1b_2 \ldots b_i b_{i+1} + y'/2^{i+1} &= .b_1b_2 \ldots b_i 1 + (2y-1)/2^{i+1} \\
&= .b_1b_2 \ldots b_i + 1/2^{i+1} + 2y/2^{i+1} - 1/2^{i+1} \\
&= .b_1b_2 \ldots b_i + y/2^i \\
&= x
\end{aligned}
$$

Numeric example

- x=1/3

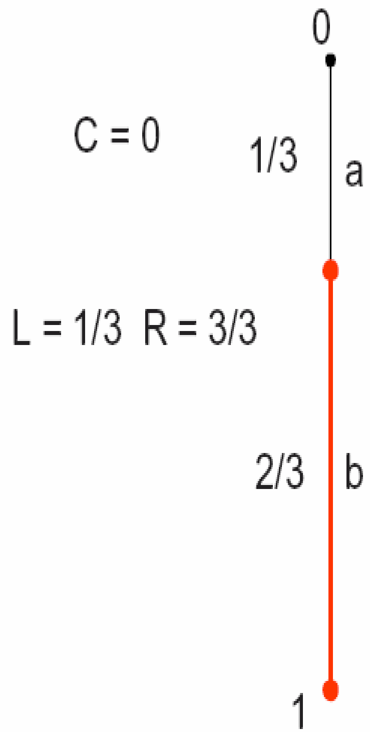
y	i	b_i
$1/3$	1	0
$2/3$	2	1
$1/3$	3	0
$2/3$	4	1
...	...	...

Scaling algorithm for arithmetic coding

- [L,R) (not a point!) is found in one of three intervals after scaling
  - Lower half: $[L,R) \in [0,.5)$
    - update step E_1 : L=2L, R=2R
    - Code generation: $0\,\underbrace{1...1}_{C \text{ of them}}$
    - C=0
  - Upper half: $[L,R) \in [.5,1)$
    - update step E_2 : L=2L-1, R=2R-1
    - Code generation: $1\,\underbrace{0...0}_{C \text{ of them}}$
    - C=0
  - Middle half: $[L,R) \in [.25,.75)$
    - update step E_3 : L=2L-0.5, R=2R-0.5
    - C=C+1
  - Notice: $\underbrace{E_3 ... E_3}_{C \text{ of them}} E_i \equiv E_i \underbrace{E_{1-i} ... E_{1-i}}_{C \text{ of them}}$

Example

- <u>b</u>aa



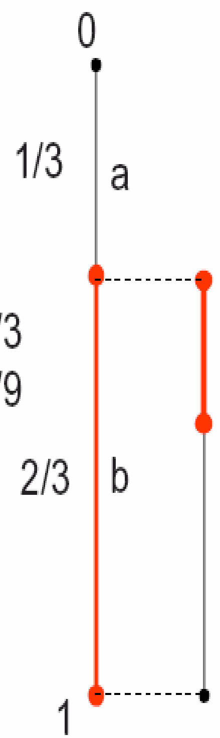
Example (cont.)

- b<u>a</u>a

C = 0

L = 1/3 R = 3/3

L = 3/9 R = 5/9



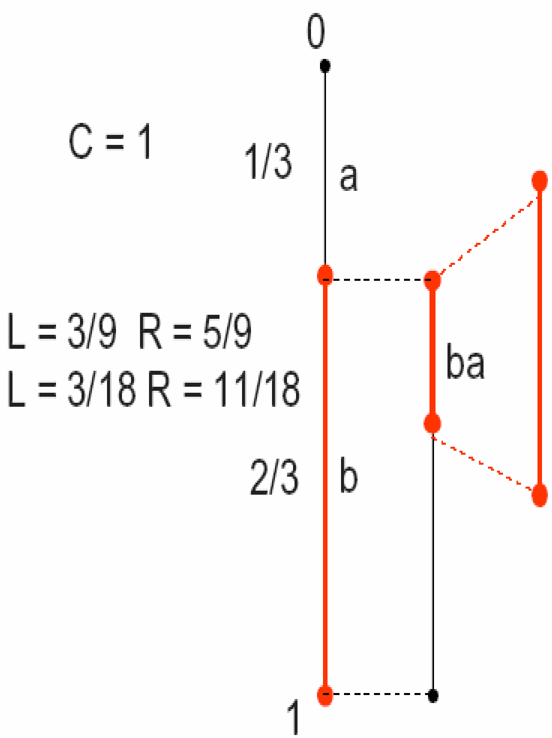
$$[3/9, 5/9) \in [0.25, 0.75)$$

$$\Downarrow$$

Scale middle half

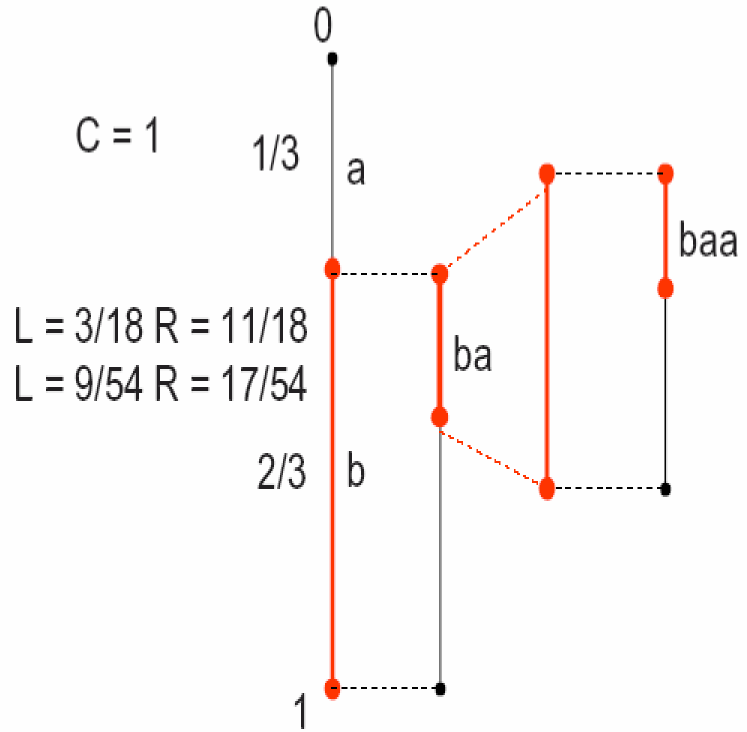
Example (cont.)

- b<u>a</u>a



Example (cont.)

- baa

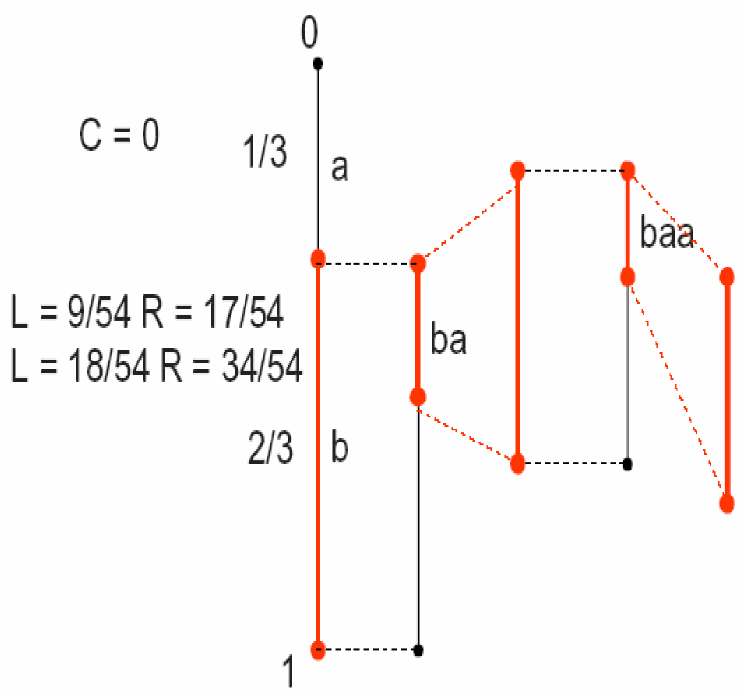


$$[9/54, 17/54) \in [0, 0.5)$$
$$\Downarrow$$

Scale lower half

Example (cont.)

- baa 01

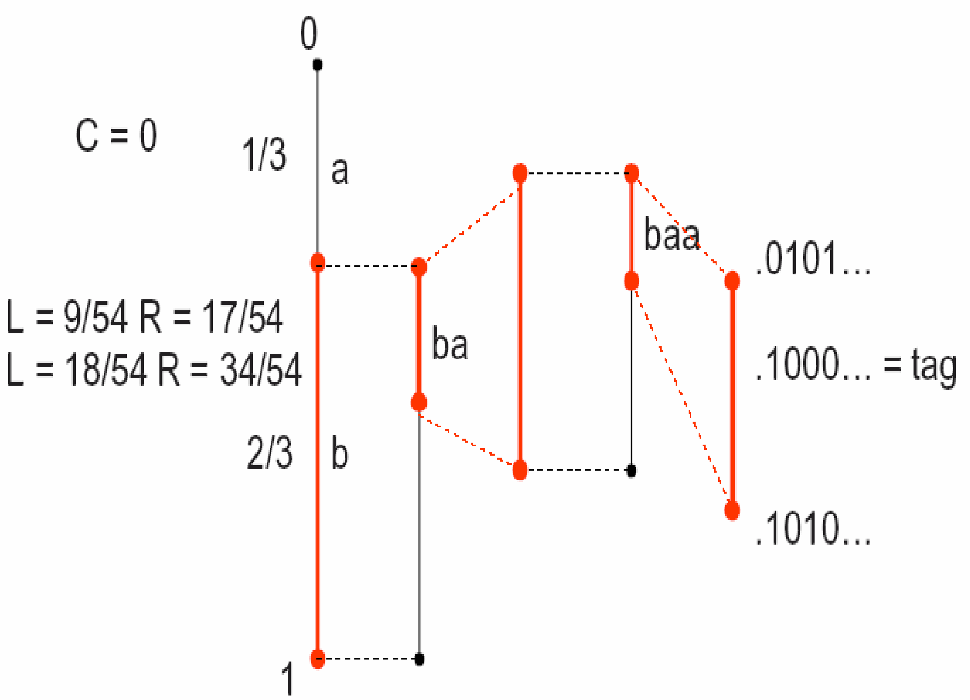


Example (cont.)

- baa 011

Based on tag, not critical

In end L < ½ < R, choose tag to be 1/2



Arithmetic coding with adaptive probability models

- Maintain the probabilities for each context.
  - Estimate probabilities from frequencies of occurences
- For the first symbol use the equal probability model.
- For each successive symbol use the model for the previous symbol.
- Example in alphabet {a,b,c,d}

		a	a	b	a	a	c
a	1	2	3	3	4	5	5
b	1	1	1	2	2	2	2
c	1	1	1	1	1	1	2
d	1	1	1	1	1	1	1

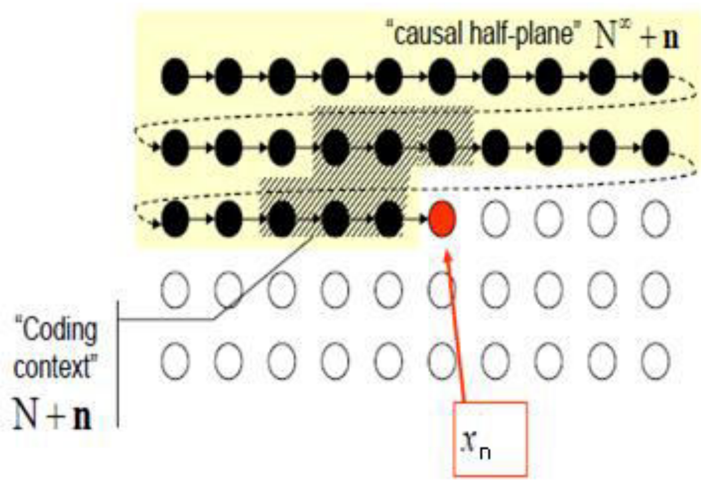
After aabaac is encoded
The probability model is
a 5/10 b 2/10
c 2/10 d 1/10

Comparison of Arithmetic and Huffman Coding

- Both compress very well. For m symbol blocks.
  - Huffman is within 1/m of entropy. $H_m(\underline{X}) \le E[L] < H_m(\underline{X}) + 1/m$
  - Arithmetic is within 2/m of entropy. $H_m(\underline{X}) \le E[L] < H_m(\underline{X}) + 2/m$
- Complexity
  - (Block) Huffman coding requires exponential growth of table with no. of symbols due to modelling joint pmf
  - Arithmetic Coding applies first order pmf/cdf to code each symbol
- Adaptivity
  - Huffman has an elaborate adaptive algorithm (update tree)
  - Arithmetic has a simple adaptive mechanism (update probability model)
- Bottom Line – Arithmetic is more flexible than Huffman.
  - Used in MPEG standards (CABAC: Context Adaptive Binary Arithmetic Coding)

Context Adaptive Arithmetic Coding

- Achieves $H(X_n \mid X_{N+n}) \leq H(X_n)$
  - Neigborhood information reduces uncertainty
  - N+n:neighborhood of n'th sample



Dictionary Coding

- LZ77
- LZ78
- LZW
- Applications

Encoder codes the index

index	pattern
1	a
2	b
3	ab
…	
n	abc

Encoder $\xrightarrow{\text{Indices}}$ Decoder

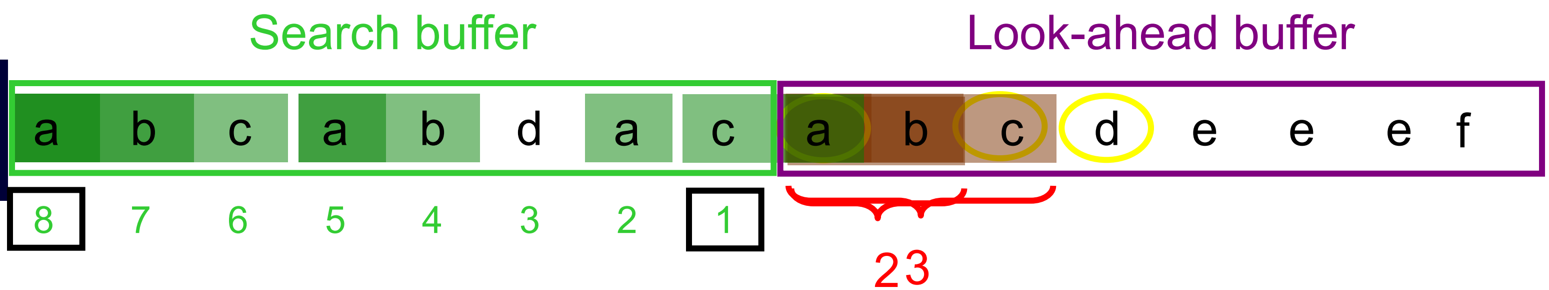
Both encoder and decoder are assumed to have the same dictionary (table)

Ziv-Lempel Coding (ZL or LZ)

- Named after J. Ziv and A. Lempel (1977).

- Adaptive dictionary technique.
  - Store previously coded symbols in a buffer.
  - Search for the current sequence of symbols to code.
  - If found, transmit buffer offset, length and next.

LZ77



Output triplet <offset, length, next>

Transmitted to decoder: 8 3 d 0 0 e 1 2 f

If the size of the search buffer is N and the size of the alphabet is M we need

$$\lceil \log(N+1) \rceil + \lceil \log(N+1) \rceil + \lceil \log M \rceil$$

bits to code a triplet.

Variation: Use a VLC to code the triplets!

PKZip, Zip, Lharc, PNG, gzip, ARJ

Drawback with LZ77

- Repetetive patterns with a period longer than the search buffer size are not found.
- If the search buffer size is 4, the sequence
  a b c d e a b c d e a b c d e a b c d e …
  will be expanded, not compressed.

LZ78

- Store patterns in a dictionary
- Transmit a tuple <dictionary index, next>

LZ78

a b c a b a b c

Output tuple <dictionary index, next>

Transmitted to decoder: 0 a 0 b 0 c 1 b 4 c

Decoded: a b c a b a b c

Dictionary:

1	a
2	b
3	c
4	a b
5	a b c

Strategy needed for limiting dictionary size!

LZW

- Modification to LZ78 by Terry Welch, 1984.
- Applications: GIF, v42bis
- Patented by UniSys Corp.
- Transmit only the dictionary index.
- The alphabet is stored in the dictionary in advance.

LZW

[redacted] a b c a b a b c

Output: dictionary index

Transmitted:

1	2	3	5	5

Decoded:

a	b	c	a b	a b

Encoder dictionary:

1	a	6	bc
2	b	7	ca
3	c	8	aba
4	d	9	abc
5	a b		

Decoder dictionary:

1	a	6	bc
2	b	7	ca
3	c	8	aba
4	d		
5	a b		

GIF

- CompuServe Graphics Interchange Format (1987, 89).
- Features:
  - Designed for up/downloading images via PSTN.
  - 1-, 4-, or 8-bit *colour palettes*.
  - Interlace for *progressive decoding* (four passes, starts with every 8th row).
  - *Transparent colour* for non-rectangular images.
  - Supports multiple images in one file (”animated GIFs”).

GIF: Method

- Compression by LZW.
- Dictionary size 2^{b+1} 8-bit symbols
  - b is the number of bits in the palette.
- Dictionary size doubled if filled (max 4096).
- Works well on computer generated images.

GIF: Problems

- Unsuitable for natural images (photos):
  - Maximum 256 colors () bad quality).
  - Repetetive patterns uncommon () bad compression).
- LZW patented by UniSys Corp.
- Alternative: PNG

PNG: Portable Network Graphics

- Designed to replace GIF.
- Some features:
  - Indexed *or* true-colour images (· 16 bits per plane).
  - Alpha channel.
  - Gamma information.
  - Error detection.
- No support for multiple images in one file.
  - Use MNG for that.
- Method:
  - Compression by LZ77 using a 32KB search buffer.
  - The LZ77 triplets are Huffman coded.
- More information: www.w3.org/TR/REC-png.html