# CSEP 521
# Applied Algorithms
Spring 2005

Statistical Lossless Data Compression

## Outline for Tonight

- Basic Concepts in Data Compression
- Entropy
- Prefix codes
- Huffman Coding
- Arithmetic Coding
- Run Length Coding (Golomb Code)

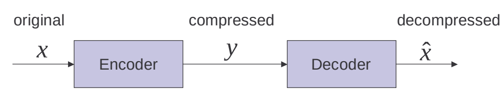


## Reading

- Huffman Coding: CLRS 385-392
- Other sources can be found:
  - Data Compression: The Complete Reference, 3rd Edition by David Salomon
  - Introduction to Data Compression by Khalid Sayood.



## Basic Data Compression Concepts




- Lossless compression $x = \hat{x}$
  - Also called entropy coding, reversible coding.
- Lossy compression $x \neq \hat{x}$
  - Also called irreversible coding.
- Compression ratio = $|x|/|y|$
  - $|x|$ is number of bits in $x$.



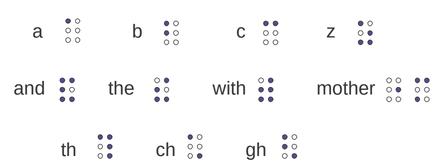
## Why Compress

- Conserve storage space
- Reduce time for transmission
  - Faster to encode, send, then decode than to send the original
- Progressive transmission
  - Some compression techniques allow us to send the most important bits first so we can get a low resolution version of some data before getting the high fidelity version
- Reduce computation
  - Use less data to achieve an approximate answer



## Braille

- System to read text by feeling raised dots on paper (or on electronic displays). Invented in 1820s by Louis Braille, a French blind man.

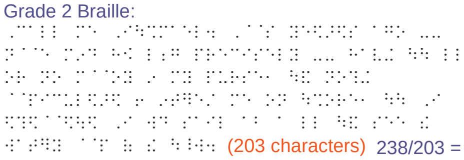


## Braille Example

Clear text:

Call me Ishmael. Some years ago -- never mind how long precisely -- having \\ little or no money in my purse, and nothing particular to interest me on shore, \\ I thought I would sail about a little and see the watery part of the world. (238 characters)

Grade 2 Braille:



(203 characters) 238/203 = 1.17



## Lossless Compression

- Data is not lost - the original is really needed.
  - text compression
  - compression of computer binary files
- Compression ratio typically no better than 4:1 for lossless compression on many kinds of files.
- Statistical Techniques
  - Huffman coding
  - Arithmetic coding
  - Golomb coding
- Dictionary techniques
  - LZW, LZ77
  - Sequitur
  - Burrows-Wheeler Method
- Standards - Morse code, Braille, Unix compress, gzip, zip, bzip, GIF, JBIG, Lossless JPEG



## Lossy Compression

- Data is lost, but not too much.
  - audio
  - video
  - still images, medical images, photographs
- Compression ratios of 10:1 often yield quite high fidelity results.
- Major techniques include
  - Vector Quantization
  - Wavelets
  - Block transforms
  - Standards - JPEG, JEPG2000, MPEG, H.264



## Why is Data Compression Possible

- Most data from nature has redundancy
  - There is more data than the actual information contained in the data.
  - Squeezing out the excess data amounts to compression.
  - However, unsqueezing is necessary to be able to figure out what the data means.
- Information theory is needed to understand the limits of compression and give clues on how to compress well.



## What is Information

- Analog data
  - Also called continuous data
  - Represented by real numbers (or complex numbers)
- Digital data
  - Finite set of symbols $\{a_1, a_2, \ldots, a_m\}$
  - All data represented as sequences (strings) in the symbol set.
  - Example: {a,b,c,d,r} abracadabra
  - Digital data can be an approximation to analog data



## Symbols

- Roman alphabet plus punctuation
- ASCII - 256 symbols
- Binary - {0,1}
  - 0 and 1 are called bits
  - All digital information can be represented efficiently in binary
  - {a,b,c,d} fixed length representation

| symbol | a | b | c | d |
|---|---|---|---|---|
| binary | 00 | 01 | 10 | 11 |

  - 2 bits per symbol

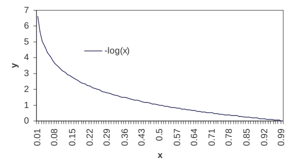
## Information Theory

- Developed by Shannon in the 1940's and 50's
- Attempts to explain the limits of communication using probability theory.
- Example: Suppose English text is being sent
  - It is much more likely to receive an "e" than a "z".
  - In some sense "z" has more information than "e".



## First-order Information

- Suppose we are given symbols $\{a_1, a_2, \dots, a_m\}$.
- $P(a_i)$ = probability of symbol $a_i$ occurring in the absence of any other information.
  - $P(a_1) + P(a_2) + \dots + P(a_m) = 1$
- $inf(a_i) = \log_2(1/P(a_i))$ bits is the information of $a_i$ in bits.






## Example

- {a, b, c} with P(a) = 1/8, P(b) = 1/4, P(c) = 5/8
  - $inf(a) = \log_2(8) = 3$
  - $inf(b) = \log_2(4) = 2$
  - $inf(c) = \log_2(8/5) = .678$
- Receiving an "a" has more information than receiving a "b" or "c".



## First Order Entropy

- The first order entropy is defined for a probability distribution over symbols $\{a_1, a_2, \dots, a_m\}$.

$$H = \sum_{i=1}^{m} P(a_i)\log_2\left(\frac{1}{P(a_i)}\right)$$

- $H$ is the average number of bits required to code up a symbol, given all we know is the probability distribution of the symbols.
- $H$ is the Shannon lower bound on the average number of bits to code a symbol in this "source model".
- Stronger models of entropy include context.



## Entropy Examples

- {a, b, c} with a 1/8, b 1/4, c 5/8.
  - H = 1/8 *3 + 1/4 *2 + 5/8* .678 = 1.3 bits/symbol
- {a, b, c} with a 1/3, b 1/3, c 1/3. (worst case)
  - $H = 3^* (1/3)^*\log_2(3) = 1.6$ bits/symbol
- Note that a standard code takes 2 bits per symbol

| | | | |
|---|---|---|---|
| symbol | a | b | c |
| binary code | 00 | 01 | 10 |

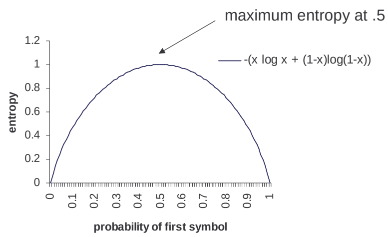


## An Extreme Case

- {a, b, c} with a 1, b 0, c 0
  - H = ?

## Entropy Curve

- Suppose we have two symbols with probabilities x and 1-x, respectively.

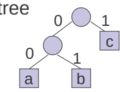





## A Simple Prefix Code

- {a, b, c} with a 1/8, b 1/4, c 5/8.
- A prefix code is defined by a binary tree
- Prefix code property
  - no output is a prefix of another

binary tree



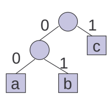

code

| input | output |
|---|---|
| a | 00 |
| b | 01 |
| c | 1 |

ccabccbccc
1 1 00 01 1 1 01 1 1 1



## Decoding a Prefix Code

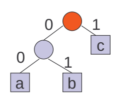



```
repeat
start at root of tree
   repeat
      if read bit = 1 then go right
      else go left
   until node is a leaf
   report leaf
until end of the code
```

11000111100



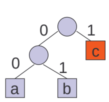
## Decoding a Prefix Code




11000111100



## Decoding a Prefix Code

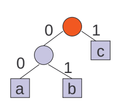



11000111100

c



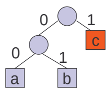
## Decoding a Prefix Code




11000111100

c

## Decoding a Prefix Code

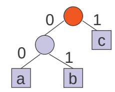



11000111100

cc



## Decoding a Prefix Code

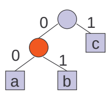



11000111100

cc



## Decoding a Prefix Code

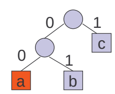



11000111100

cc



## Decoding a Prefix Code

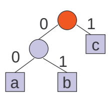



11000111100

cca



## Decoding a Prefix Code

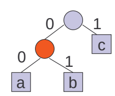



11000111100

cca



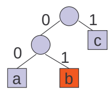
## Decoding a Prefix Code




11000111100

cca

## Decoding a Prefix Code

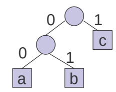



11000111100

ccab



## Decoding a Prefix Code

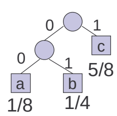



11000111100

ccabccca



## How Good is the Code




bit rate = (1/8)2 + (1/4)2 + (5/8)1 = 11/8 = 1.375 bps
Entropy = 1.3 bps
Standard code = 2 bps

(bps = bits per symbol)



## Design a Prefix Code 1

- abracadabra
- Design a prefix code for the 5 symbols {a,b,r,c,d} which compresses this string the most.



## Design a Prefix Code 2

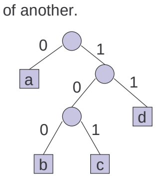
- Suppose we have n symbols each with probability 1/n. Design a prefix code with minimum average bit rate.
- Consider n = 2,3,4,5,6 first.



## Huffman Coding

- Huffman (1951)
- Uses frequencies of symbols in a string to build a variable rate prefix code.
  - Each symbol is mapped to a binary string.
  - More frequent symbols have shorter codes.
  - No code is a prefix of another.
- Example:

a 0
b 100
c 101
d 11



## Variable Rate Code Example

- Example: a 0, b 100, c 101, d 11
- Coding:
  - aabddcaa = 16 bits
  - 0 0 100 11 11 101 0 0= 14 bits
- Prefix code ensures unique decodability.
  - 00100111110100
  - a a b d d c a a



## Cost of a Huffman Tree

- Let $p_1, p_2, \ldots, p_m$ be the probabilities for the symbols $a_1, a_2, \ldots, a_m$, respectively.
- Define the cost of the Huffman tree T to be

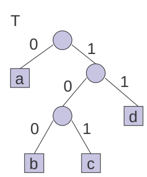
$$C(T) = \sum_{i=1}^{m} p_i r_i$$

where $r_i$ is the length of the path from the root to $a_i$.
- C(T) is the expected length of the code of a symbol coded by the tree T. C(T) is the average bit rate (ABR) of the code.



## Example of Cost

- Example: a 1/2, b 1/8, c 1/8, d 1/4




C(T) = 1 x 1/2 + 3 x 1/8 + 3 x 1/8 + 2 x 1/4 = 1.75
a b c d



## Huffman Tree

- Input: Probabilities $p_1, p_2, \ldots, p_m$ for symbols $a_1, a_2, \ldots, a_m$, respectively.
- Output: A tree that minimizes the average number of bits (bit rate) to code a symbol. That is, minimizes

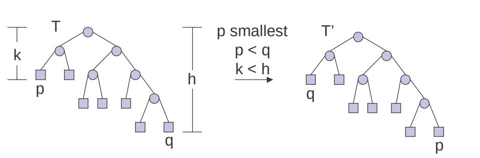
$$HC(T) = \sum_{i=1}^{m} p_i r_i \quad \text{bit rate}$$

where $r_i$ is the length of the path from the root to $a_i$. This is the Huffman tree or Huffman code



## Optimality Principle 1

- In a Huffman tree a lowest probability symbol has maximum distance from the root.
  - If not exchanging a lowest probability symbol with one at maximum distance will lower the cost.

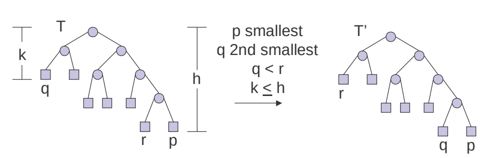



C(T') = C(T) + hp - hq + kq - kp = C(T) - (h-k)(q-p) < C(T)



## Optimality Principle 2

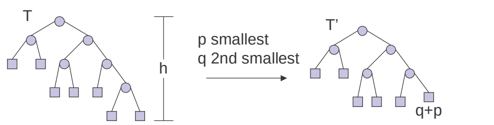
- The second lowest probability is a sibling of the the smallest in some Huffman tree.
  - If not, we can move it there not raising the cost.




C(T') = C(T) + hq - hr + kr - kq = C(T) - (h-k)(r-q) $\leq$ C(T)

## Optimality Principle 3

- Assuming we have a Huffman tree T whose two lowest probability symbols are siblings at maximum depth, they can be replaced by a new symbol whose probability is the sum of their probabilities.
  – The resulting tree is optimal for the new symbol set.

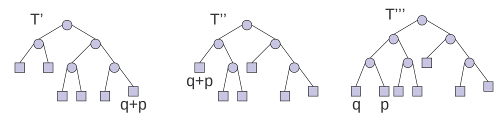



C(T') = C(T) + (h-1)(p+q) - hp -hq = C(T) - (p+q)



## Optimality Principle 3 (cont')

- If T' were not optimal then we could find a lower cost tree T''. This will lead to a lower cost tree T''' for the original alphabet.




C(T''') = C(T'') + p + q < C(T') + p + q = C(T) which is a contradiction



## Recursive Huffman Tree Algorithm

1. If there is just one symbol, a tree with one node is optimal. Otherwise
2. Find the two lowest probability symbols with probabilities p and q respectively.
3. Replace these with a new symbol with probability p + q.
4. Solve the problem recursively for new symbols.
5. Replace the leaf with the new symbol with an internal node with two children with the old symbols.



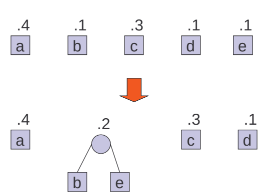
## Iterative Huffman Tree Algorithm

```
form a node for each symbol a_i with weight p_i;
insert the nodes in a min priority queue ordered by probability;
while the priority queue has more than one element do
   min1 := delete-min;
   min2 := delete-min;
   create a new node n;
   n.weight := min1.weight + min2.weight;
   n.left := min1;
   n.right := min2;
   insert(n)
return the last node in the priority queue.
```



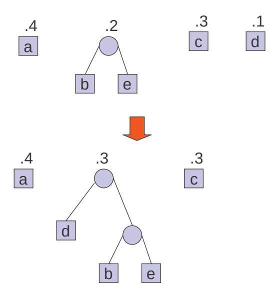
## Example of Huffman Tree Algorithm (1)

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

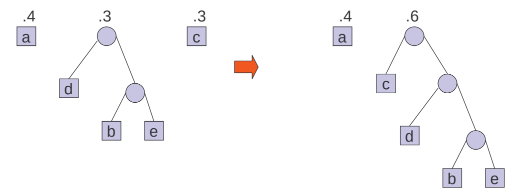





## Example of Huffman Tree Algorithm (2)

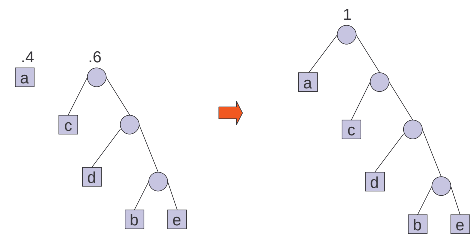


## Example of Huffman Tree Algorithm (3)



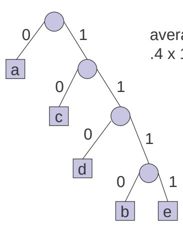



## Example of Huffman Tree Algorithm (4)






## Huffman Code




average number of bits per symbol is
.4 x 1 + .1 x 4 + .3 x 2 + .1 x 3 + .1 x 4 = 2.1

| | |
|---|---|
| a | 0 |
| b | 1110 |
| c | 10 |
| d | 110 |
| e | 1111 |



## Optimal Huffman Code vs. Entropy

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

Entropy

H = -(.4 x $\log_2$(.4) + .1 x $\log_2$(.1) + .3 x $\log_2$(.3)
+ .1 x $\log_2$(.1) + .1 x $\log_2$(.1))
= 2.05 bits per symbol

Huffman Code

HC = .4 x 1 + .1 x 4 + .3 x 2 + .1 x 3 + .1 x 4
= 2.1 bits per symbol
pretty good!



## In Class Exercise

- P(a) = 1/2, P(b) = 1/4, P(c) = 1/8, P(d) = 1/16, P(e) = 1/16
- Compute the Huffman tree and its bit rate.
- Compute the Entropy
- Compare
- Hint: For the tree change probabilities to be integers: a:8, b:4, c:2, d:1, e:1. Normalize at the end.

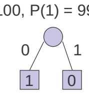


## Quality of the Huffman Code

- The Huffman code is within one bit of the entropy lower bound.

$$H \leq HC \leq H + 1$$

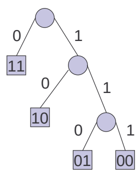
- Huffman code does not work well with a two symbol alphabet.
  – Example: P(0) = 1/100, P(1) = 99/100
  – HC = 1 bits/symbol




  – H = -((1/100)*$\log_2$(1/100) + (99/100)$\log_2$(99/100))
  = .08 bits/symbol
- If probabilities are powers of two then HC = H.

## Extending the Alphabet

- Assuming independence P(ab) = P(a)P(b), so we can lump symbols together.
- Example: P(0) = 1/100, P(1) = 99/100
  – P(00) = 1/10000, P(01) = P(10) = 99/10000, P(11) = 9801/10000.




HC = 1.03 bits/symbol (2 bit symbol)
= .515 bits/bit

Still not that close to H = .08 bits/bit



## Quality of Extended Alphabet

- Suppose we extend the alphabet to symbols of length k then

$$H \le HC \le H + 1/k$$

- Pros and Cons of Extending the alphabet
  + Better compression
  - $2^k$ symbols
  - padding needed to make the length of the input divisible by k



## Context Modeling

- Data does not usually come from a 1st order statistical source.
  – English text: "u" almost always follows "q"
  – Images: a pixel next to a blue pixel is likely to be blue
- Practical coding: Divide the data by contexts and code the data in each context as its own 1st order source.



## Huffman Codes with Context

- Suppose we add a one symbol context. That is in compressing a string $x_1x_2...x_n$ we want to take into account $x_{k-1}$ when encoding $x_k$.
  – New model, so entropy based on just independent probabilities of the symbols doesn't hold. The new entropy model (2nd order entropy) has for each symbol a probability for each other symbol following it.
  – Example: {a,b,c}

next

| prev | | a | b | c |
|---|---|---|---|---|
| | a | .4 | .2 | .4 |
| | b | .1 | .9 | 0 |
| | c | .1 | .1 | .8 |

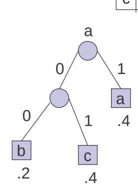


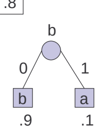
## Multiple Codes

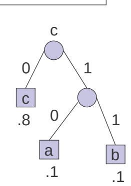
next

| prev | | a | b | c |
|---|---|---|---|---|
| | a | .4 | .2 | .4 |
| | b | .1 | .9 | 0 |
| | c | .1 | .1 | .8 |

| Code for first symbol |
|---|
| a 00 |
| b 01 |
| c 10 |




a b b a c c

<u>00</u> <u>00</u> <u>0</u> <u>1</u> <u>01</u> <u>0</u>



## Complexity of Huffman Code Design

- Time to design Huffman Code is O(n log n) where n is the number of symbols.
  – Each step consists of a constant number of priority queue operations (2 deletemin's and 1 insert)

## Approaches to Huffman Codes

1. Frequencies computed for each input
   – Must transmit the Huffman code or frequencies as well as the compressed input
   – Requires two passes
2. 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
3. Adaptive Huffman code
   – One pass
   – Huffman tree changes as frequencies change



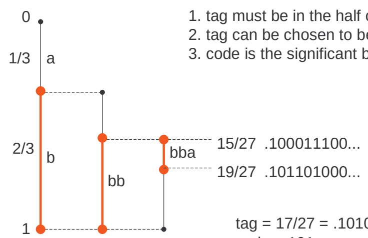
## Arithmetic Coding

- Basic idea in arithmetic coding:
  – represent each string x of length n by a unique interval [L,R) in [0,1).
  – The width R-L of the interval [L,R) represents the probability of x occurring.
  – The interval [L,R) can itself be represented by any number, called a tag, within the half open interval.
  – 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).
    - It turns out that $k \approx \log_2(1/(R\text{-}L))$.



## Example of Arithmetic Coding (1)

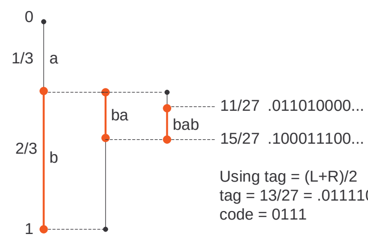
1. tag must be in the half open interval.
2. tag can be chosen to be (L+R)/2.
3. code is the significant bits of the tag.




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



## Some Tags are Better than Others



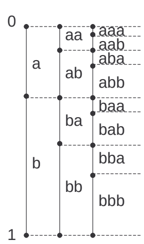

Using tag = (L+R)/2
tag = 13/27 = .011110110...
code = 0111

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



## Example of Codes

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




| | |
|---|---|
| 0/27 | .000000000... |
| 1/27 | .000010010... |
| 3/27 | .000111000... |
| 5/27 | .001011110... |
| 9/27 | .010101010... |
| 11/27 | .011010000... |
| 15/27 | .100011100... |
| 19/27 | .101101000... |
| 27/27 | .111111111... |

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

.95 bits/symbol
.92 entropy lower bound

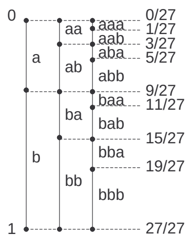


## Code Generation from Tag

- If binary tag is $.t_1t_2t_3... = (L+R)/2$ in [L,R) then we want to choose k to form the code $t_1t_2...t_k$.
- Short code:
  – choose k to be as small as possible so that $L \leq .t_1t_2...t_k000... < R$.
- Guaranteed code:
  – choose $k = \lceil \log_2 (1/(R-L)) \rceil + 1$
  – $L \leq .t_1t_2...t_kb_1b_2b_3... < R$ for any bits $b_1b_2b_3...$
  – for fixed length strings provides a good prefix code.
  – example: [.000000000..., .000010010...), tag = .000001001...
    Short code: 0
    Guaranteed code: 000001

## Guaranteed Code Example

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




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



## Arithmetic Coding Algorithm

- $P(a_1), P(a_2), \ldots, P(a_m)$
- $C(a_i) = P(a_1) + P(a_2) + \ldots + P(a_{i-1})$
- Encode $x_1x_2...x_n$

```
Initialize L := 0 and R:= 1;
for i = 1 to n do
   W := R - L;
   L := L + W * C(x_i);
   R := L + W * P(x_i);
t := (L+R)/2;
choose code for the tag
```



## Arithmetic Coding Example

- P(a) = 1/4, P(b) = 1/2, P(c) = 1/4
- C(a) = 0, C(b) = 1/4, C(c) = 3/4
- abca

W := R - L;
L := L + W C(x);
R := L + W P(x)

| 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



## Arithmetic Coding Exercise

- P(a) = 1/4, P(b) = 1/2, P(c) = 1/4
- C(a) = 0, C(b) = 1/4, C(c) = 3/4
- bbbb

W := R - L;
L := L + W C(x);
R := L + W P(x)

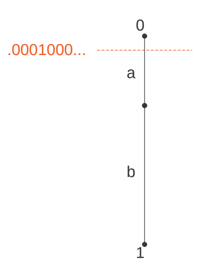
| symbol | W | L | R |
|---|---|---|---|
| | | 0 | 1 |
| b | 1 | | |
| b | | | |
| b | | | |
| b | | | |

tag =
L =
R =
code =
prefix code =



## Decoding (1)

- Assume the length is known to be 3.
- 0001 which converts to the tag .0001000...

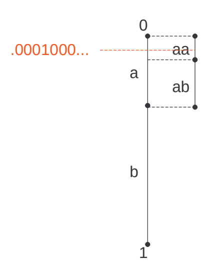



output a



## Decoding (2)

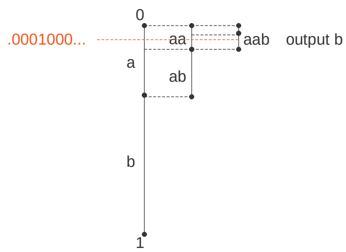
- Assume the length is known to be 3.
- 0001 which converts to the tag .0001000...




output a

## Decoding (3)

- Assume the length is known to be 3.
- 0001 which converts to the tag .0001000...






## Arithmetic Decoding Algorithm

- $P(a_1), P(a_2), \ldots, P(a_m)$
- $C(a_i) = P(a_1) + P(a_2) + \ldots + P(a_{i-1})$
- Decode $b_1b_2 \ldots b_k$, number of symbols is n.

Initialize L := 0 and R := 1;
t := $.b_1b_2 \ldots b_k000\ldots$
for i = 1 to n do
  W := R - L;
  find j such that $L + W * C(a_j) \leq t < L + W * (C(a_j)+P(a_j))$
  output $a_j$;
  L := L + W * $C(a_j)$;
  R := L + W * $P(a_j)$;



## Decoding Example and Exercise

- P(a) = 1/4, P(b) = 1/2, P(c) = 1/4
- C(a) = 0, C(b) = 1/4, C(c) = 3/4
- 00101 and n = 4

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 at least two ways for the decoder to know when to stop decoding.
  1. Transmit the length of the string
  2. Transmit a unique end of string symbol



## 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.
- Context:
  - Different contexts can be handled easily
- Adaptivity:
  - Coding can be done adaptively, learning the distribution of symbols dynamically
- Integer arithmetic coding avoids floating point altogether.



## Scaling

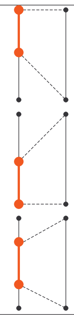
- Scaling:
  - By scaling we can keep L and R in a reasonable range of values so that W = R – L does not underflow.
  - The code can be produced progressively, not at the end.
  - Complicates decoding some.

## Scaling Principle

Lower half
If [L,R) is contained in [0,.5) then
L := 2L; R := 2R
output 0, followed by C 1's
C := 0.

Upper half
If [L,R) is contained in [.5,1) then
L := 2L –1, R := 2R - 1
output 1, followed by C 0's
C := 0

Middle Half
If [L,R) is contained in [.25,.75) then
L := 2L –.5, R := 2R -.5
C := C + 1.





## Example

- baa

C = 0

L = 1/3 R = 3/3

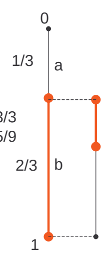



## Example

- baa

C = 0

L = 1/3 R = 3/3
L = 3/9 R = 5/9




Scale middle half

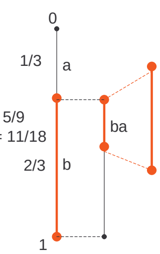


## Example

- baa

C = 1

L = 3/9 R = 5/9
L = 3/18 R = 11/18



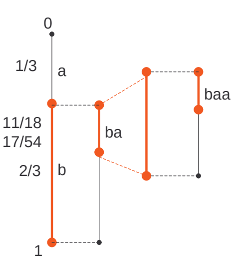



## Example

- baa

C = 1

L = 3/18 R = 11/18
L = 9/54 R = 17/54




Scale lower half

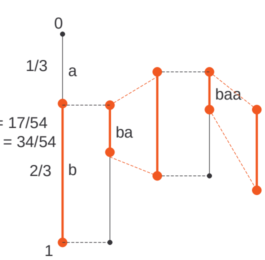


## Example

- baa 01

C = 0

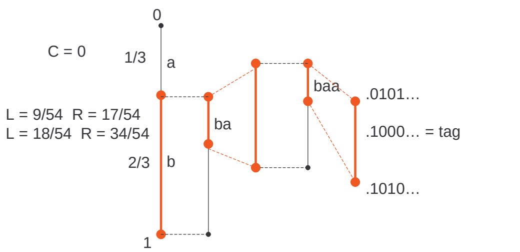
L = 9/54 R = 17/54
L = 18/54 R = 34/54



## Example

- baa 011

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






## Decoding with Scaling

- Use the same scaling algorithm as the encoder
  - There is no need to keep track of C because we know the complete tag.
  - Each scaling step will consume a symbol of the tag
    - Lower half: $0x \rightarrow x$ ($10 \times .0x = .x$ in binary)
    - Upper half: $1x \rightarrow x$ ($10 \times .1x - 1 = .x$)
    - Middle half: $10x \rightarrow 1x$ or $01x \rightarrow 0x$ ($10 \times .10x - .1 = .1x$ or $10 \times .01x - .1 = .0x$)



## Integer Implementation

- m bit integers
  - Represent 0 with 000…0 (m times)
  - Represent 1 with 111…1 (m times)
- Probabilities represented by frequencies
  - $n_i$ is the number of times that symbol $a_i$ occurs
  - $C_i = n_1 + n_2 + \ldots + n_{i-1}$
  - $N = n_1 + n_2 + \ldots + n_m$

$$W := R - L + 1$$
$$L' := L + \left\lfloor \frac{W \cdot C_i}{N} \right\rfloor$$
$$R := L + \left\lfloor \frac{W \cdot C_{i+1}}{N} \right\rfloor - 1$$
$$L := L'$$

Coding the i-th symbol using integer calculations. Must use scaling!



## Context

- Consider 1 symbol context.
- Example: 3 contexts.

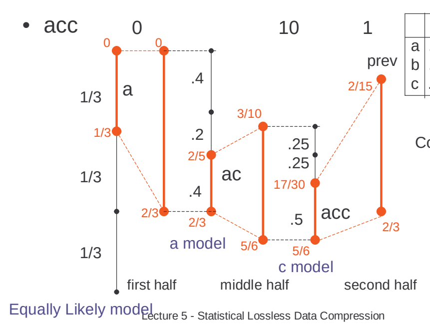
next

| prev | a | b | c |
|---|---|---|---|
| a | .4 | .2 | .4 |
| b | .1 | .8 | .1 |
| c | .25 | .25 | .5 |



## Example with Context and Scaling

- acc




next

| prev | a | b | c |
|---|---|---|---|
| a | .4 | .2 | .4 |
| b | .1 | .8 | .1 |
| c | .25 | .25 | .5 |

Code = 0101



## Arithmetic Coding with Context

- Maintain the probabilities for each context.
- For the first symbol use the equal probability model
- For each successive symbol use the model for the previous symbol.

## Adaptation

- Simple solution – Equally Probable Model.
  - Initially all symbols have frequency 1.
  - After symbol x is coded, increment its frequency by 1
  - Use the new model for coding the next 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



## Zero Frequency Problem

- How do we weight symbols that have not occurred yet.
  - Equal weights? Not so good with many symbols
  - Escape symbol, but what should its weight be?
  - When a new symbol is encountered send the <esc>, followed by the symbol in the equally probable model. (Both encoded arithmetically.)

| | | a | a | b | a | a | c |
|---|---|---|---|---|---|---|---|
| a | 0 | 1 | 2 | 2 | 3 | 4 | 4 |
| b | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| c | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| d | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| <esc> | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

After aabaac is encoded
The probability model is
a 4/7 b 1/7
c 1/7 d 0
<esc> 1/7



## Arithmetic vs. Huffman

- Both compress very well. For m symbol grouping.
  - Huffman is within 1/m of entropy.
  - Arithmetic is within 2/m of entropy.
- Symbols
  - Huffman needs a reasonably large set of symbols
  - Arithmetic works fine on binary symbols
- Context
  - Huffman needs a tree for every context.
  - Arithmetic needs a small table of frequencies for every context.
- Adaptation
  - Huffman has an elaborate adaptive algorithm
  - Arithmetic has a simple adaptive mechanism.
- Bottom Line – Arithmetic is more flexible than Huffman.



## Run-Length Coding

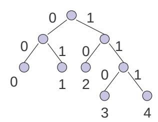
- Lots of 0's and not too many 1's.
  - Fax of letters
  - Graphics
- Simple run-length code
  - Input
    00000010000000001000000000010001001.....
  - Symbols
    6 9 10 3 2 ...
  - Code the bits as a sequence of integers
  - Problem: How long should the integers be?



## Golomb Code of Order m
## Variable Length Code for Integers

- Let n = qm + r where $0 \leq r < m$.
  - Divide m into n to get the quotient q and remainder r.
- Code for n has two parts:
  1. q is coded in unary
  2. r is coded as a fixed prefix code

Example: m = 5




code for r



## Example

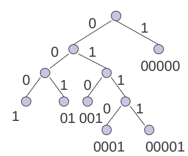
- n = qm + r is represented by:

$$\overbrace{11\cdots1}^{q}0\hat{r}$$

  - where $\hat{r}$ is the fixed prefix code for r
- Example (m = 5):

| 2 | 6 | 9 | 10 | 27 |
|---|---|---|---|---|
| 010 | 1001 | 10111 | 11000 | 11111010 |

## Alternative Explanation
## Golomb Code of order 5




| input | output |
|---|---|
| 00000 | 1 |
| 00001 | 0111 |
| 0001 | 0110 |
| 001 | 010 |
| 01 | 001 |
| 1 | 000 |

Variable length to variable length code.



## Run Length Example: m = 5

00000010000000001000000000010001001.....
1
00000010000000001000000000010001001.....
001
00000010000000001000000000010001001.....
1
00000010000000001000000000010001001.....
0111

In this example we coded 17 bit in only 9 bits.



## Choosing m

- Suppose that 0 has the probability p and 1 has probability 1-p.
- The probability of $0^n1$ is $p^n(1-p)$. The Golomb code of order $$m = \left\lceil -1/\log_2 p \right\rceil$$ is optimal.
- Example: p = 127/128.

$$m = \left\lceil -1/\log_2 (127/128) \right\rceil = 89$$



## Golomb Coding Exercise

- Construct the Golomb Code of order 9. Show it as a prefix code (a binary tree).



## PPM

- Prediction with Partial Matching
  - Cleary and Witten (1984)
- State of the art arithmetic coder
  - Arbitrary order context
  - The context chosen is one that does a good prediction given the past
  - Adaptive
- Example
  - Context "the" does not predict the next symbol "a" well. Move to the context "he" which does.



## Summary

- Statistical codes are very common as parts of image, video, music, and speech coder.
- Arithmetic and Huffman are most popular.
- Special statistical codes like Golomb codes are used in some situations.