15-750:Algorithms in the Real World

Continue with Data Compression

(Recap) Relationship to Entropy

Theorem (lower bound): For any probability distribution p(S) with associated uniquely decodable code C,

$$H(S) \leq l_a(C)$$

Theorem (upper bound): For any probability distribution p(S) with associated <u>optimal</u> prefix code C,

$$l_a(C) \leq H(S) + 1$$

(Recap) Kraft McMillan Inequality

Theorem (Kraft-McMillan): For any uniquely decodable code C,

$$\sum_{c \in C} 2^{-l(c)} \leq 1$$

Conversely, for any set of lengths L such that $\sum_{l \in L} 2^{-l} \leq 1$

there is a prefix code C such that

$$l(c_i) = l_i (i = 1, \ldots, |L|)$$

Used Kraft McMillan for proving the upper bound theorem.

Another property of optimal codes

Theorem: If C is an optimal prefix code for the probabilities $\{p_1, \ldots, p_n\}$, then $p_i > p_j$ implies $l(c_i) \leq l(c_j)$

Proof: (by contradiction: **switching technique**)

Assume $l(c_i) > l(c_j)$ (for the sake of contradiction).

Consider switching codes c_i and c_j.

If l_a is the average length of the original code, the length of the new code is

$$
\begin{aligned}
l_a' &= l_a + p_j(l(c_i) - l(c_j)) + p_i(l(c_j) - l(c_i)) \\
&= l_a + (p_j - p_i)(l(c_i) - l(c_j)) \\
&< l_a
\end{aligned}
$$

This is a contradiction since l_a is not optimal

Huffman Codes

Invented by Huffman as a class assignment in 1950.

Used in many, if not most, compression algorithms

Properties:

- Generates optimal prefix codes
- Cheap to generate codes
- Cheap to encode and decode
- $l_a = H$ if probabilities are powers of 2

Huffman Codes

Huffman Algorithm:

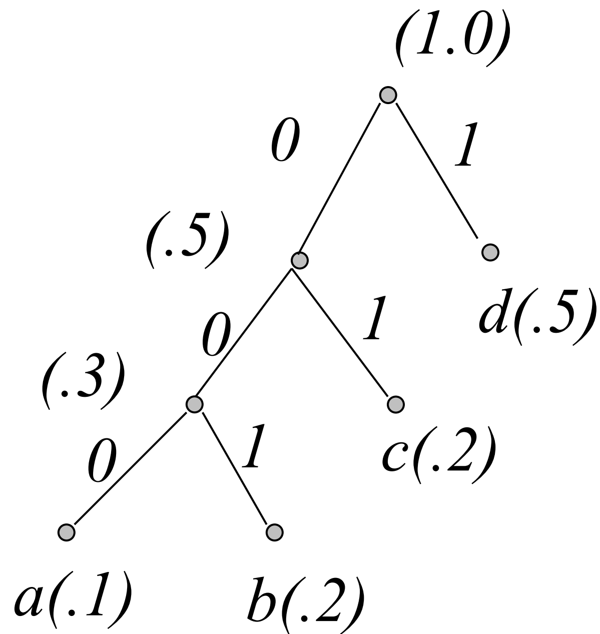
Start with a forest of trees each consisting of a single vertex corresponding to a message *s* and with weight *p(s)*

Repeat until one tree left:

- Select two trees with minimum weight roots p_1 and p_2
- Join into single tree by adding root with weight $p_1 + p_2$

Example

$p(a) = .1, \; p(b) = .2, \; p(c) = .2, \; p(d) = .5$

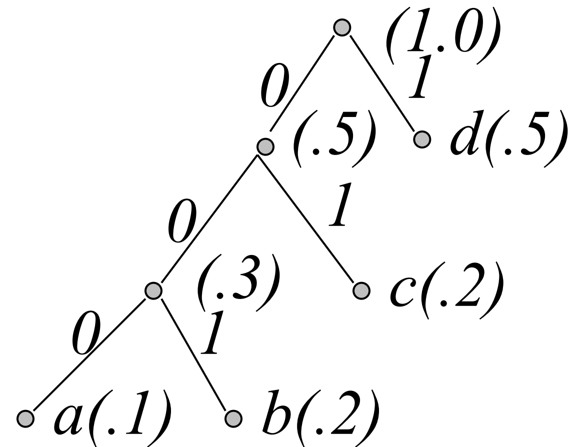


∘ a(.1) ∘ b(.2) ∘ c(.2) ∘ d(.5)

Encoding and Decoding

Encoding: Start at leaf of Huffman tree and follow path to the root. Reverse order of bits and send.

$a=000, \ b=001, \ c=01, d=1$



Decoding: Start at root of Huffman tree and take branch for each bit received. When at leaf can output message and return to root.

Huffman codes are "optimal"

Theorem: The Huffman algorithm generates an optimal *prefix* code.

Proof outline:

Induction on the number of messages n.

Consider a message set S with $n+1$ messages

1. Can make it so that least probable messages of S are neighbors in the Huffman tree
2. Replace the two messages with one message with probability $p(m_1) + p(m_2)$ making S'
3. Show that if S' is optimal, then S is optimal
4. S' is optimal by induction

Problem with Huffman Coding

Consider a message with probability .999. The self information of this message is

$$-\log(.999) = .00144$$

If we were to send a 1000 such messages we might hope to use 1000*.0014 = 1.44 bits.

Using Huffman codes we require at least one bit per message, so we would require 1000 bits.

Need to “blend” bits among message symbols!

Discrete or Blended

Discrete: each message is a fixed set of bits

– E.g., Huffman coding, Shannon-Fano coding

01001	11	0001	011
1	2	3	4

message: 1 2 3 4

Blended: bits can be "shared" among messages

– E.g., Arithmetic coding

010010111010

message: 1,2,3, and 4

Arithmetic Coding: Introduction

Allows "blending" of bits in a message sequence.

Only requires 3 bits for the example

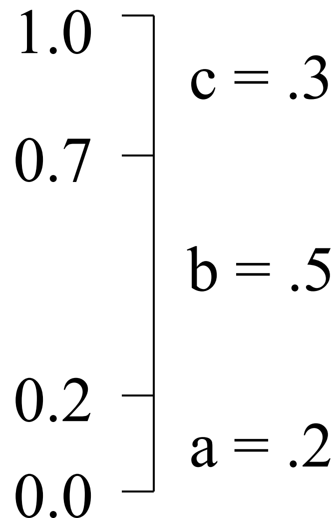
Can bound total bits required based on sum of self information:

$$l < 2 + \sum_{i=1}^{n} s_i$$

Used in many compression algorithms as building block

Arithmetic Coding: message intervals

Assign each message to an interval range from 0 (inclusive) to 1 (exclusive) based on the probabilities.

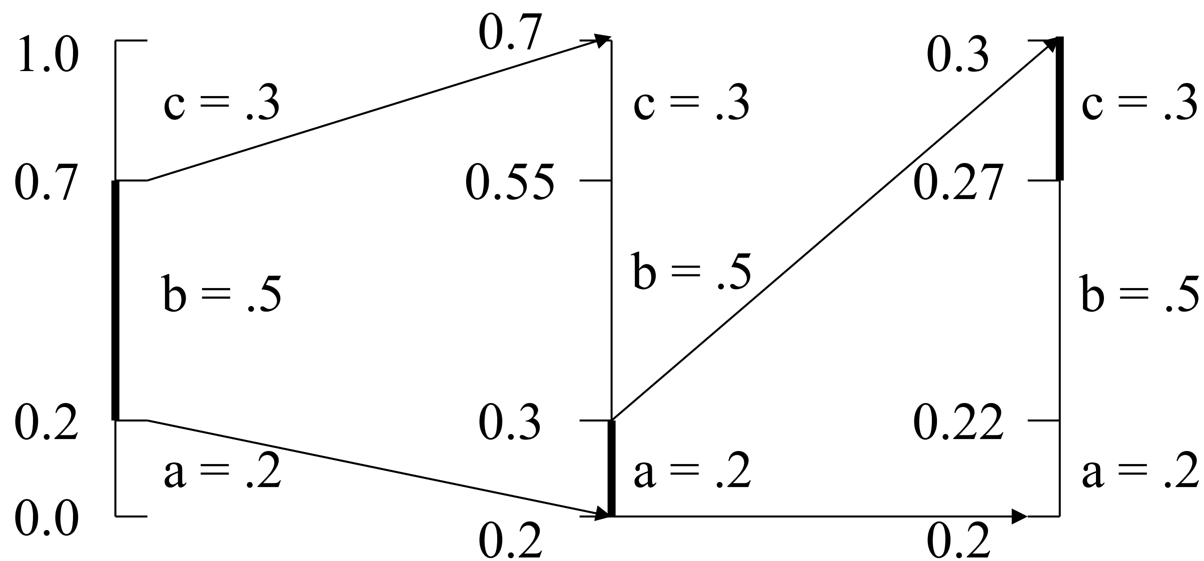


The interval for a particular message will be called the **message interval** (e.g for b the interval is [.2,.7))

Arithmetic Coding: Sequence intervals

Code a message sequence by composing intervals.

For example: **bac**



The final interval is **[.27,.3)**

Uniquely defining an interval

Important property: The sequence intervals for distinct message sequences of length *n* will never overlap

Therefore: specifying any number in the final interval uniquely determines the sequence.

Decoding is similar to encoding, but on each step need to determine what the message value is and then go backwards

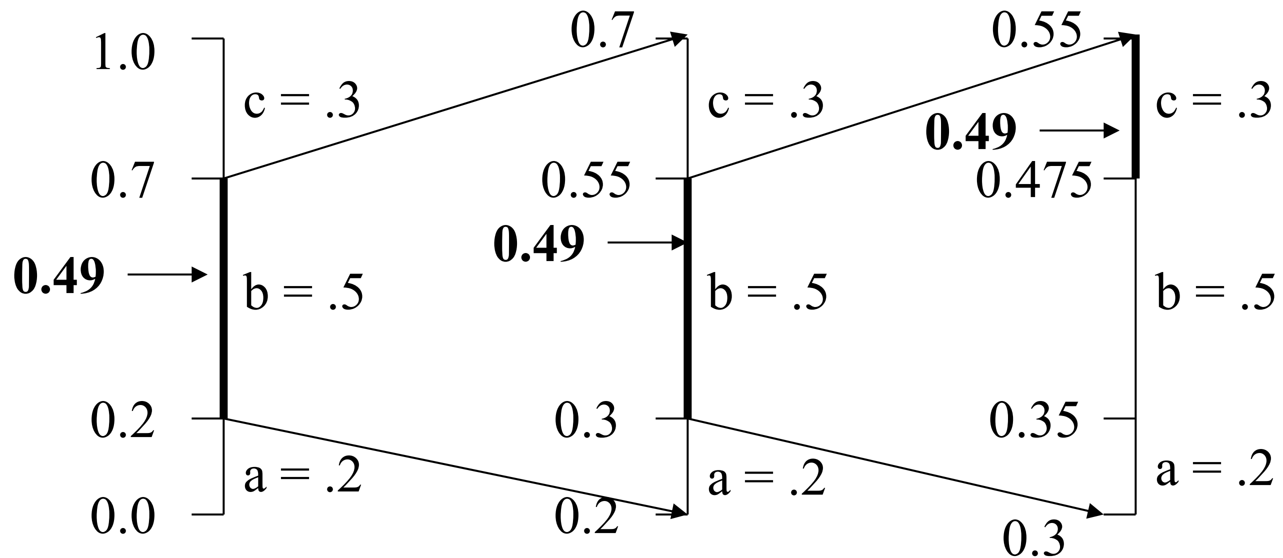
Decoding for Arithmetic Codes

Decoding is similar to encoding

On each step need to determine what the message value is and then go backwards

Arithmetic Coding: Decoding Example

Decoding the number .49, knowing the message is of length 3:



The message is **bbc.**

Arithmetic codes: takeaways

- Blending messages into a sequence helps achieve better compression
- Takes closer to the information theoretic lower bound

$$l < 2 + \sum_{i=1}^{n} s_i$$

- Arithmetic codes are more expensive than Huffman coding
  - Due to fractions involved
  - Integer implementations exist and are not too bad (converting all fractions to equivalent integer representations)

Transformation Techniques for Compression

1. Run length coding
2. Move-to-front coding
3. Residual coding
4. Burrows-Wheeler transform
5. Linear transform coding

Why transform?

- Help skew the probabilities
  - Why?
  - Recall higher the skew easier it is to compress
- In many algorithms message sequences are transformed into **integers with a skew towards small integers**
- We will take a detour to study codes for integers ...

Integer codes

- There are several “fixed” codes for encoding natural numbers
- With non-decreasing codeword lengths

Integer codes: binary

n	Binary
1	..001
2	..010
3	..011
4	..100
5	..101
6	..110

"Minimal" binary representation: Drop leading zeros

Q: What is the problem with minimal binary representation?

Not a prefix code!

Integer codes: Unary

n	Binary	Unary
1	..001	0
2	..010	10
3	..011	110
4	..100	1110
5	..101	11110
6	..110	111110

n represented as $(n-1)$ 1's and one 0
(0's and 1's can be interchanged)

Q: For what probability distribution unary codes are optimal prefix codes?

Integer codes: Gamma

n	Binary	Unary	Gamma
1	..001	0	0|
2	..010	10	10|0
3	..011	110	10|1
4	..100	1110	110|00
5	..101	11110	110|01
6	..110	111110	110|10

Many other fixed prefix codes:
Golomb, phased-binary, subexponential, ...

Back to **transforming data** for encoding…

Transformation Techniques

1. Run length coding
2. Move-to-front coding
3. Residual coding
4. Burrows-Wheeler transform
5. Linear transform coding

1. Run Length Coding

Code by specifying message value followed by the number of repeated values:

e.g. **abbbaacccca => (a,1),(b,3),(a,2),(c,4),(a,1)**

The characters and counts can be coded based on frequency (i.e., probability coding).

Typically low counts such as 1 and 2 are more common => use small number of bits overhead for these.

Used as a sub-step in many compression algorithms.

2. Move to Front (MTF) Coding

- Transforms message sequence into sequence of integers
- Then probability code

Start with values in a total order: e.g.: [a,b,c,d,…]

For each message

- output the position in the order
- move to the front of the order.

e.g.: **c a**

c => output: 3, new order: [c,a,b,d,e,…]
a => output: 2, new order: [a,c,b,d,e,…]

Probability code the output.

2. Move to Front (MTF) Coding

The hope is that there is a bias for small numbers.

Q: Why?

Temporal locality

Takes advantage of **temporal locality**

Used as a sub-step in many compression algorithms.

3. Residual Coding

Typically used for message values that represent some sort of amplitude:
e.g. gray-level in an image, or amplitude in audio.

Basic Idea:

- Guess next value based on current context.
- Output difference between guess and actual value.
- Use probability code on the output.

E.g.: Consider compressing a stock value over time.

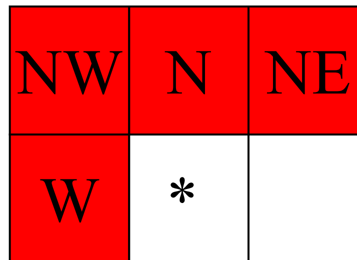
Residual coding is used in JPEG Lossless

Use of residual coding in JPEG-LS

JPEG Lossless

Codes in Raster Order.

Uses 4 pixels as context:



Tries to guess value of * based on W, NW, N and NE.

The residual between guessed and actual value is found and then coded using a Golomb-like code.

(Golomb codes are similar to Gamma codes)

4. Burrows –Wheeler Transform

Breaks file into fixed-size blocks and encodes each block separately.

For each block:

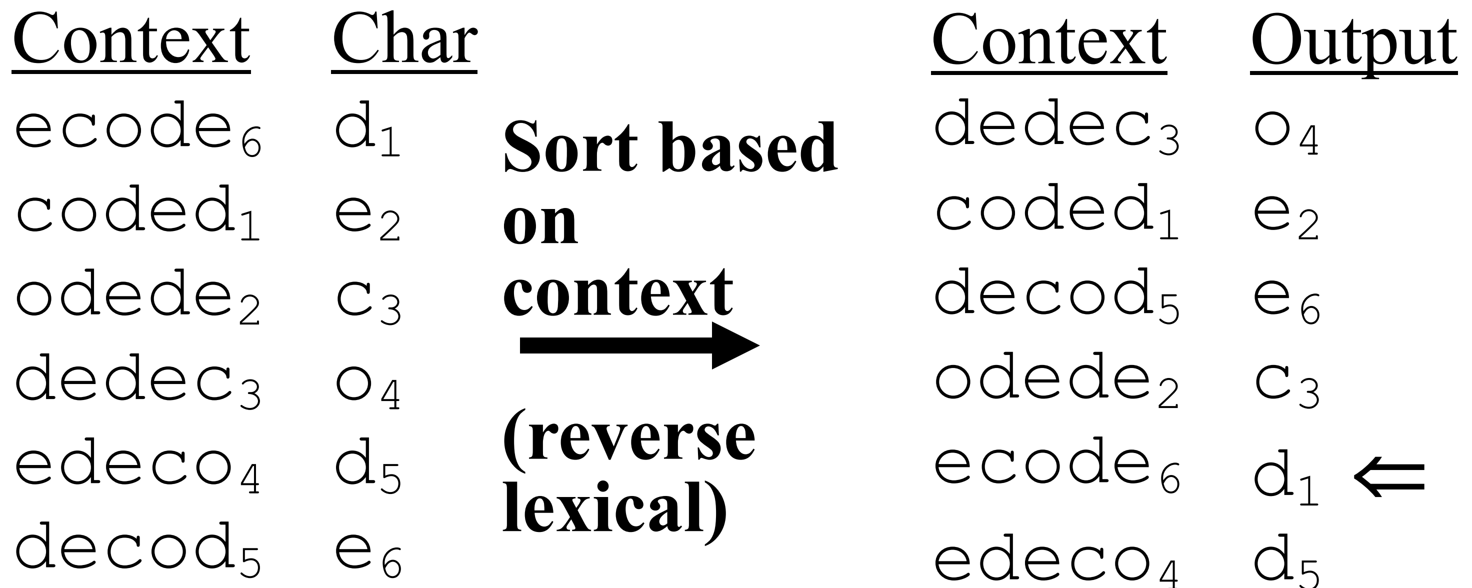
- Create full **context** for each character
- Context wraps around
- Reverse lexical sort each character by its full context. This is called the "block sorting transform".

Burrows Wheeler: Example

To encode: $d_1e_2c_3o_4d_5e_6$

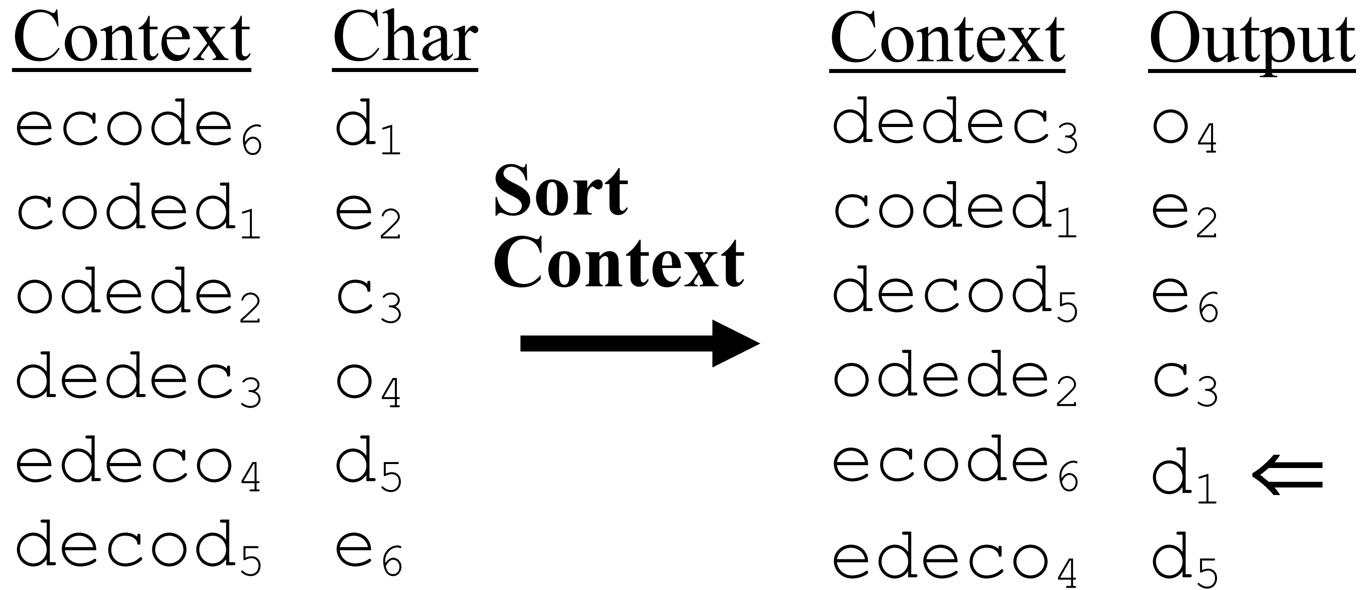
(Numbered the characters to distinguish them.)

Context "wraps" around. Last char is most significant.



Q: Why is the output more easier to compress?

Burrows Wheeler: Example



Gets similar characters together
(because we are ordering by context)

Why not just sort the original block of characters?

Can we invert BW Transform?

Output

o_4

e_2

e_6

c_3

d_1 ⇐

d_5

Can we invert BW Transform?

Suppose we are given the context… then?

Context	Output
c_3	o_4
d_1	e_2
d_5	e_6
e_2	c_3
e_6	d_1 ⇐
o_4	d_5

How can we get the last column of the context column from the output column?

Sort!

Any problem? Equal valued chars

Burrows-Wheeler (Continued)

Theorem: After sorting, equal valued characters appear in the same order in the output column as in the last column of the sorted context.

Proof sketch:

The chars with equal value in the most-significant-position (i.e., last column) of the context will be ordered by the rest of the context, i.e. the previous chars.

This is also the order of the output since it is sorted by the previous characters.

Context	Output
$\texttt{dedec}_3$	$\texttt{o}_4$
code$\texttt{d}_1$	$\texttt{e}_2$
deco$\texttt{d}_5$	$\texttt{e}_6$
$\texttt{odede}_2$	$\texttt{c}_3$
ecode$_6$	$\texttt{d}_1$
edeco$_4$	$\texttt{d}_5$

Burrows-Wheeler: Decoding

- What follows the underlined **a** ?
- What follows the underlined **b**?
- What is the whole string?

Context	Output	
a	c	
a	**b**	
a	b	
b	a	
b	**a**	⇐
c	a	

Answer: b, a, abacab

BZIP

Transform 1: (Burrows Wheeler)

- **input** : character string (block)
- **output** : reordered character string

Transform 2: (move to front)

- **input** : character string
- **output** : MTF numbering

Transform 3: (run length)

- **input** : MTF numbering
- **output** : sequence of run lengths

Probabilities: (on run lengths)

Dynamic based on counts for each block.

Coding: Originally arithmetic, but changed to Huffman in bzip2 due to patent concerns