15-750: Algorithms in the Real World

Data Compression: Lecture 2
(Recap) Discrete or Blended

**Discrete**: each message is a fixed set of bits
- E.g., Huffman coding, Shannon-Fano coding

```plaintext
01001  11  0001  011
message:  1  2  3  4
```

**Blended**: bits can be “shared” among messages
- E.g., Arithmetic coding

```plaintext
010010111010
message:  1,2,3, and 4
```
(Recap) Arithmetic Coding

Allows “blending” of bits in a message sequence.

More expensive than Huffman coding

Used in many compression algorithms: E.g., JPEG/MPEG
Arithmetic Coding: message intervals

Assign each probability distribution to an interval range from 0 (inclusive) to 1 (exclusive).

e.g.

\[
\begin{align*}
&0.0 & \quad a = .2 \\
&0.2 & \quad b = .5 \\
&0.7 & \quad c = .3 \\
&1.0
\end{align*}
\]

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 reduce interval.
Arithmetic Coding: Decoding Example

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

The message is bbc.
Transformation Techniques

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

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**

<table>
<thead>
<tr>
<th>n</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>..001</td>
</tr>
<tr>
<td>2</td>
<td>..010</td>
</tr>
<tr>
<td>3</td>
<td>..011</td>
</tr>
<tr>
<td>4</td>
<td>..100</td>
</tr>
<tr>
<td>5</td>
<td>..101</td>
</tr>
<tr>
<td>6</td>
<td>..110</td>
</tr>
</tbody>
</table>

“Minimal” binary representation: Drop leading zeros

Q: What is the problem with minimal binary representation?

**Not a prefix code!**
Integer codes: Unary

<table>
<thead>
<tr>
<th>n</th>
<th>Binary</th>
<th>Unary</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>..001</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>..010</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>..011</td>
<td>110</td>
</tr>
<tr>
<td>4</td>
<td>..100</td>
<td>1110</td>
</tr>
<tr>
<td>5</td>
<td>..101</td>
<td>11110</td>
</tr>
<tr>
<td>6</td>
<td>..110</td>
<td>111110</td>
</tr>
</tbody>
</table>

n represented as n-1 ones 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

<table>
<thead>
<tr>
<th>n</th>
<th>Binary</th>
<th>Unary</th>
<th>Gamma</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>..001</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>..010</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>..011</td>
<td>110</td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td>..100</td>
<td>1110</td>
<td>110</td>
</tr>
<tr>
<td>5</td>
<td>..101</td>
<td>11110</td>
<td>110</td>
</tr>
<tr>
<td>6</td>
<td>..110</td>
<td>111110</td>
<td>110</td>
</tr>
</tbody>
</table>

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. \textit{abbbbaacccca} $\Rightarrow$ (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 $\Rightarrow$ use small number of bits overhead for these.

Used as a sub-step in many compression algorithms.
2. Move to Front 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 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:

\[
\begin{array}{ccc}
\text{NW} & \text{N} & \text{NE} \\
\text{W} & * &
\end{array}
\]

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 (wraps around)
- Reverse lexical sort each character by its full context. This is called the “block sorting transform”.
**Burrows Wheeler: Example**

To encode: \(d_1 e_2 c_3 o_4 d_5 e_6\)

(Numbered the characters to distinguish them.)

Context “wraps” around. Last char is most significant.

<table>
<thead>
<tr>
<th>Context</th>
<th>Char</th>
<th>Context</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>ecode_6</td>
<td>(d_1)</td>
<td>dedec_3</td>
<td>(o_4)</td>
</tr>
<tr>
<td>coded_1</td>
<td>(e_2)</td>
<td>coded_1</td>
<td>(e_2)</td>
</tr>
<tr>
<td>ode_2</td>
<td>(c_3)</td>
<td>decod_5</td>
<td>(e_6)</td>
</tr>
<tr>
<td>dedec_3</td>
<td>(o_4)</td>
<td>ode_2</td>
<td>(c_3)</td>
</tr>
<tr>
<td>edeco_4</td>
<td>(d_5)</td>
<td>edeco_4</td>
<td>(d_5)</td>
</tr>
<tr>
<td>decod_5</td>
<td>(e_6)</td>
<td>edeco_4</td>
<td>(d_5)</td>
</tr>
</tbody>
</table>

Q: Why is the output more easier to compress?
### Burrows Wheeler: Example

<table>
<thead>
<tr>
<th>Context</th>
<th>Char</th>
<th>Context</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>ecode₆</td>
<td>d₁</td>
<td>dedec₃</td>
<td>o₄</td>
</tr>
<tr>
<td>coded₁</td>
<td>e₂</td>
<td>coded₁</td>
<td>e₂</td>
</tr>
<tr>
<td>odeđe₂</td>
<td>c₃</td>
<td>decod₅</td>
<td>e₆</td>
</tr>
<tr>
<td>dedec₃</td>
<td>o₄</td>
<td>odeđe₂</td>
<td>c₃</td>
</tr>
<tr>
<td>edeco₄</td>
<td>d₅</td>
<td>ecode₆</td>
<td>d₁</td>
</tr>
<tr>
<td>decod₅</td>
<td>e₆</td>
<td>edeco₄</td>
<td>d₅</td>
</tr>
</tbody>
</table>

Sort Context

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

Why not just sort?
Can we invert BW Transform?

Output

\[ \begin{align*}
&o_4 \\
e_2 \\
e_6 \\
c_3 \\
d_1 \leftarrow \\
d_5
\end{align*} \]
Can we invert BW Transform?

<table>
<thead>
<tr>
<th>Context</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>(c_3)</td>
<td>(o_4)</td>
</tr>
<tr>
<td>(d_1)</td>
<td>(e_2)</td>
</tr>
<tr>
<td>(d_5)</td>
<td>(e_6)</td>
</tr>
<tr>
<td>(e_2)</td>
<td>(c_3)</td>
</tr>
<tr>
<td>(e_6)</td>
<td>(d_1)</td>
</tr>
<tr>
<td>(o_4)</td>
<td>(d_5)</td>
</tr>
</tbody>
</table>

Suppose we are given the context… then?

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

Sort!

Any problem? Equal valued chars
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.
Burrows-Wheeler: Decoding

– What follows the underlined a?
– What follows the underlined b?
– What is the whole string?

<table>
<thead>
<tr>
<th>Context</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>c</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>b</td>
<td>a</td>
</tr>
<tr>
<td>b</td>
<td>a</td>
</tr>
</tbody>
</table>

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
5. Linear Transform Coding

Goal: Transform the data into a form that is easily compressible (through **lossless** or **lossy** compression)

Select a set of linear basis functions $\phi_i$ that span the space
- sin, cos, spherical harmonics, wavelets, …
Linear Transform Coding (continued)

Coefficients:

$$\Theta_i = \sum_j x_j \phi_i(j) = \sum_j x_j a_{ij}$$

$\Theta_i = \text{ } i^{th}$ resulting coefficient

$x_j = \text{ } j^{th}$ input value

$a_{ij} = \text{ } ij^{th}$ transform coefficient = $\phi_i(j)$

In matrix notation:  $\Theta = Ax$

$x = A^{-1} \Theta$

Where A is an $n \times n$ “transform” matrix, and each row defines a basis function
Example: Cosine Transform

\[ \Theta_i = \sum_j x_j \phi_i(j) \]
Other Transforms

Polynomial:

1

x

x^2

Wavelet (Haar):
How to Pick a Transform

Goals:

– Decorrelate the data
– Low coefficients for many terms
– Basis functions that can be ignored from the perception point-of-view
15-750: Algorithms in the Real World

Quantization (lossy)
Scalar Quantization

Quantize regions of values into a single value
E.g. Drop least significant bit
    (Can be used to reduce # of bits for a pixel)

Q: Why is this lossy?
Many-to-one mapping

Two types
    – Uniform: Mapping is linear
    – Non-uniform: Mapping is non-linear
Scalar Quantization

Q: Why use non-uniform?
Error metric might be non-uniform.
E.g. Human eye sensitivity to specific color regions

Can formalize the mapping problem as an optimization problem
Vector Quantization

Mapping a multi-dimensional space into a smaller set of messages

In

Generate Vector

Codebook

Find closest code vector

Index

Encode

Out

Generate Output

Index

Codebook

Decode
Vector Quantization (VQ)

What do we use as vectors?
- Color (Red, Green, Blue)
  - Can be used, for example to reduce 24bits/pixel to 8bits/pixel
  - Used in some monitors to reduce data rate from the CPU (colormaps)
- K consecutive samples in audio
- Block of K pixels in an image

How do we decide on a codebook
- Typically done with clustering

VQ most effective when the variables along the dimensions of the space are correlated
Observations:

1. Highly correlated: Concentration of representative points

2. Higher density is more common regions.
Case Study: JPEG

A nice example since it uses many techniques:

- Transform coding (Cosine transform)
- Scalar quantization
- Difference coding
- Run-length coding
- Huffman or arithmetic coding