Outline

- Run-Length Coding
- Golomb Coding
- Tunstall Coding
Lossless coding: Run-Length encoding (RLE)

- Redundancy is removed by not transmitting consecutive identical symbols (pixels or character values that are equal).

- The repeated value can be coded once, along with the number of times it repeats.

- Useful for coding black and white images e.g. fax.
Binary RLE

- Code the run length of 0’s using k bits. Transmit the code.
- Do not transmit runs of 1’s.
- Two consecutive 1’s are implicitly separated by a zero-length run of zero.

Example: suppose we use $k = 4$ bits to encode the run length (maximum run length of 15) for following bit patterns

\[0 \ldots 010 \ldots 0110 \ldots 010 \ldots 0110 \ldots 0\]

Length: 91 bits

All 1’s indicate the next group is part of the same run

\[1110 \quad 1001 \quad 0000 \quad 1111 \quad 0101 \quad 1111 \quad 1111 \quad 0000 \quad 0000 \quad 1011\]

Length: 40 bits!
RLE Performance

- Worst case behavior: transition occurs on each bit. Since we use $k$ bits to represent the transition, we waste $k-1$ bits.

- Best case behavior: no transition and use $k$ bits to represent run length then the compression ratio is $(2^k-1)/k$. 
Can you improve RLE coding?

- Why use fixed length coding for the length of a run?
Golomb Coding

- How to code a potential infinite number of symbols?
  - Code the number of consecutive heads in a sequence of coin tosses.
  - 110, 1111110, 11111110, ....
Golomb Coding

- 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: $m = 5$

```
  0 1 0 1
  0 1 2 0 1
  0 1 2 3 4
```
code for $r$
Example

- $n = qm + r$ is represented by:

\[
\underbrace{\overbrace{11 \ldots 10}^{q}}_{\hat{r}}
\]

- where $\hat{r}$ is the fixed prefix code for $r$

- Example (m=5):

<table>
<thead>
<tr>
<th>2</th>
<th>6</th>
<th>9</th>
<th>10</th>
<th>27</th>
</tr>
</thead>
<tbody>
<tr>
<td>010</td>
<td>1001</td>
<td>10111</td>
<td>11000</td>
<td>11111010</td>
</tr>
</tbody>
</table>
Another Way of Looking at Golomb Code (m=5)

<table>
<thead>
<tr>
<th>input</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>1</td>
</tr>
<tr>
<td>00001</td>
<td>0111</td>
</tr>
<tr>
<td>0001</td>
<td>0110</td>
</tr>
<tr>
<td>001</td>
<td>010</td>
</tr>
<tr>
<td>01</td>
<td>001</td>
</tr>
<tr>
<td>1</td>
<td>000</td>
</tr>
</tbody>
</table>
Run-Length Example, m = 5

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
  \[
  m = \left\lceil \frac{-1}{\log_2 p} \right\rceil
  \]
  is optimal.
- Example: \( p = 127/128 \).
  \[
  m = \left\lceil \frac{-1}{\log_2 (127/128)} \right\rceil = 89
  \]
Compression of Golomb Code

Average Bit Rate = \[
\frac{\text{Average output code length}}{\text{Average input code length}}
\]

- \( m = 4 \) as an example. With \( p \) as the probability of 0.

\[
\text{ABR} = \frac{p^4 + 3(1-p^4)}{4p^4 + 4p^3(1-p) + 3p^2(1-p) + 2p(1-p) + (1-p)}
\]

<table>
<thead>
<tr>
<th>Input probability</th>
<th>1</th>
<th>011</th>
<th>010</th>
<th>001</th>
<th>000</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0001</td>
<td>001</td>
<td>01</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>( p^4 )</td>
<td>( p^3(1-p) )</td>
<td>( p^2(1-p) )</td>
<td>( p(1-p) )</td>
<td>( 1-p )</td>
<td></td>
</tr>
</tbody>
</table>
GC Performance
Notes on GC

- Useful for binary compression when one symbol is much more likely than another.
  - binary images
  - fax documents
  - bit planes for wavelet image compression

- Need a parameter (the order)
  - training
  - adaptively learn the right parameter

- Variable-to-variable length code

- Last symbol needs to be a 1
  - coder always adds a 1
  - decoder always removes a 1
Tunstall Code

- Variable-to-fixed length code
- Example

<table>
<thead>
<tr>
<th>input</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>000</td>
</tr>
<tr>
<td>b</td>
<td>001</td>
</tr>
<tr>
<td>ca</td>
<td>010</td>
</tr>
<tr>
<td>cb</td>
<td>011</td>
</tr>
<tr>
<td>cca</td>
<td>100</td>
</tr>
<tr>
<td>ccb</td>
<td>101</td>
</tr>
<tr>
<td>ccc</td>
<td>110</td>
</tr>
</tbody>
</table>

```
a b cca cb ccc ...
000 001 110 011 110 ...
```
Tunstall Code Properties

- No input code is a prefix of another to assure unique *encodability*.
- Minimize the number of bits per symbol.
Prefix Code Properties

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>000</td>
</tr>
<tr>
<td>b</td>
<td>001</td>
</tr>
<tr>
<td>ca</td>
<td>010</td>
</tr>
<tr>
<td>cb</td>
<td>011</td>
</tr>
<tr>
<td>cca</td>
<td>100</td>
</tr>
<tr>
<td>ccb</td>
<td>101</td>
</tr>
<tr>
<td>ccc</td>
<td>110</td>
</tr>
</tbody>
</table>

Unused output code is 111.
Prefix Code Properties

- Consider the string “cc”. It does not have a code.
- Send the unused code and some fixed code for the cc.
- Generally, if there are $k$ internal nodes in the prefix tree then there is a need for $k-1$ fixed codes.
Designing Tunstall Code

- Suppose there are $m$ initial symbols.
- Choose a target output length $n$ where $2^n > m$.

1. Form a tree with a root and $m$ children with edges labeled with the symbols.
2. If the number of leaves is $> 2^n - m$ then halt.*
3. Find the leaf with highest probability and expand it to have $m$ children.** Go to 2.

* In the next step we will add $m-1$ more leaves.
** The probability is the product of the probabilities of the symbols on the root to leaf path.
Example

- $P(a) = 0.7$, $P(b) = 0.2$, $P(c) = 0.1$
- $n = 3$
Example

- $P(a) = 0.7$, $P(b) = 0.2$, $P(c) = 0.1$
- $n = 3$
Example

- $P(a) = 0.7$, $P(b) = 0.2$, $P(c) = 0.1$
- $n = 3$
Compression of Tunstall Code

- The length of the output code divided by the average length of the input code.
- Let \( p_i \) be the probability of input code \( i \) and \( r_i \) the length of input code \( i \) (\( 1 \leq i \leq s \)) and let \( n \) be the length of the output code.

\[
\text{Average bit rate} = \frac{n}{\sum_{i=1}^{s} p_i r_i}
\]
Average Bit Rate of Tunstall Code

\[ ABR = 3/\left[3 \left(0.343 + 0.098 + 0.049\right) + 2 \left(0.14 + 0.07\right) + 0.2 + 0.1\right] \]

\[ = 1.37 \text{ bits per symbol} \]

Entropy = 1.16 bits per symbol
Notes on Tunstall Code

- Variable-to-fixed length code

- Error resilient
  - A flipped bit will introduce just one error in the output.
  - Huffman is not error resilient. A single bit flip can destroy the code.