Summary so far

**Model** generates probabilities, **Coder** uses them

**Probabilities** are related to **information**. The more you know, the less info a message will give.

More "skew" in probabilities gives lower **Entropy** $H$ and therefore better compression

**Context** can help "skew" probabilities (lower $H$)

Average length $l_o$ for **optimal prefix code** bound by

$H \leq l_o < H + 1$

**Huffman codes** are optimal prefix codes

**Arithmetic codes** allow "blending" among messages.

---

**Encoding: Model and Coder**

- **Model**
  - Static Part
  - Dynamic Part

- **Coder**
  - Codeword
    - $|w| = l_o(s) = -\log p(s)$

- **Message** $s \in S$

The **Static part** of the model is fixed.
The **Dynamic part** is based on previous messages.
The "optimality" of the code is relative to the probabilities.
If they are not accurate, the code is not going to be efficient.

---

**Decoding: Model and Decoder**

- **Model**
  - Static Part
  - Dynamic Part

- **Decoder**
  - Codeword

- **Message** $s \in S$

The probabilities $(p(s) \mid s \in S)$ generated by the model need to be the same as generated in the encoder.

**Note:** consecutive "messages" can be from a different message sets, and the probability distribution can change.
Codes with Dynamic Probabilities

Huffman codes:
Need to generate a new tree for new probabilities.
Small changes in probability, typically make small changes to the Huffman tree.
"Adaptive Huffman codes" update the tree without having to completely recalculate it.
Used frequently in practice

Arithmetic codes:
Need to recalculate the \( f(m) \) values based on current probabilities.
Can be done with a balanced tree.

Compression Outline

Introduction: Lossy vs. Lossless, Benchmarks, ...
Information Theory: Entropy, etc.
Probability Coding: Huffman + Arithmetic Coding
Applications of Probability Coding: PPM + others
- Transform coding: move to front, run-length, ...
- Context coding: fixed context, partial matching
Lempel-Ziv Algorithms: LZ77, gzip, compress, ...
Other Lossless Algorithms: Burrows-Wheeler
Lossy algorithms for images: JPEG, MPEG, ...
Compressing graphs and meshes: BBK

Applications of Probability Coding

How do we generate the probabilities?
Using character frequencies directly does not work very well (e.g. 4.5 bits/char for text).

Technique 1: transforming the data
- Run length coding (ITU Fax standard)
- Move-to-front coding (Used in Burrows-Wheeler)
- Residual coding (JPEG LS)

Technique 2: using conditional probabilities
- Fixed context (JBIG...almost)
- Partial matching (PPM)

Run Length Coding

Code by specifying message value followed by the number of repeated values:
e.g. abbbbaacccca => (a,1),(b,3),(a,2),(c,4),(a,1)
The characters and counts can be coded based on frequency.
This allows for small number of bits overhead for low counts such as 1.
Facsimile ITU T4 (Group 3)
Standard used by all home Fax Machines
ITU = International Telecommunications Standard
Run length encodes sequences of black+white pixels
Fixed Huffman Code for all documents. e.g.

<table>
<thead>
<tr>
<th>Run length</th>
<th>White</th>
<th>Black</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>000111</td>
<td>010</td>
</tr>
<tr>
<td>2</td>
<td>0111</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>00111</td>
<td>0000100</td>
</tr>
</tbody>
</table>
Since alternate black and white, no need for values.

Move to Front Coding
Transforms message sequence into sequence of integers, that can then be probability coded
Takes advantage of temporal locality
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 => output: 3, new order: [c,a,b,d,e,...]  
  a => output: 2, new order: [a,c,b,d,e,...]
Probability code the output.
The hope is that there is a bias for small numbers.

BZIP
Transform 1: (Burrows Wheeler) - covered later
  - 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
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.

Consider compressing a stock value over time.

JPEG-LS

JPEG Lossless (not to be confused with lossless JPEG)

Codes in Raster Order. Uses 4 pixels as context:

```
NW N NE
W *
```

Tries to guess value of * based on W, NW, N and NE.
Works in two stages

JPEG LS: Stage 1

Uses the following equation:

\[ P = \begin{cases} 
\min(N, W) & \text{if } NW \geq \max(N, W) \\
\max(N, W) & \text{if } NW < \min(N, W) \\
N + W - NW & \text{otherwise}
\end{cases} \]

Averages neighbors and captures edges. e.g.

\[
\begin{array}{ccc}
40 & 3 & * \\
40 & 3 & \\
40 & 3 & 30 \\
30 & 40 & 3 \\
3 & 3 & * \\
\end{array}
\]

JPEG LS: Stage 2

Uses 3 gradients: W-NW, NW-N, N-NE
Classifies each into one of 9 categories.
This gives \(9^3=729\) contexts, of which only 365 are needed because of symmetry.
Each context has a bias term that is used to adjust the previous prediction
After correction, the residual between guessed and actual value is found and coded using a Golomb-like code. (Golomb codes are similar to Gamma codes)
JPEG LS

Transform: (residual)
- input: gray-level image (8 bits/pixel)
- output: difference from guess at each pixel

Probabilities: (on the differences)
Static probabilities based on golomb code --- something like \( p(n) = \frac{c}{n^2} \).

Coding: Golomb code

Using Conditional Probabilities: PPM

Use previous \( k \) characters as the context.
- Makes use of conditional probabilities

Base probabilities on counts:
  e.g. if seen th 12 times followed by e 7 times, then the conditional probability \( p(e|th) = \frac{7}{12} \).

Need to keep \( k \) small so that dictionary does not get too large (typically less than 8).
Note that 8-gram Entropy of English is about 2.3bits/char while PPM does as well as 1.7bits/char

PPM: Partial Matching

Problem: What do we do if we have not seen the context followed by the character before?
  - Cannot code 0 probabilities!

The key idea of PPM is to reduce context size if previous match has not been seen.
  - If character has not been seen before with current context of size 3, try context of size 2, and then context of size 1, and then no context

Keep statistics for each context size \( < k \)

PPM: Changing between context

How do we tell the decoder to use a smaller context?
Send an escape message. Each escape tells the decoder to reduce the size of the context by 1.
The escape can be viewed as special character, but needs to be assigned a probability.
  - Different variants of PPM use different heuristics for the probability.
## PPM: Example Contexts

<table>
<thead>
<tr>
<th>Context</th>
<th>Counts</th>
</tr>
</thead>
<tbody>
<tr>
<td>Empty</td>
<td>A = 4</td>
</tr>
<tr>
<td></td>
<td>B = 2</td>
</tr>
<tr>
<td></td>
<td>C = 5</td>
</tr>
<tr>
<td></td>
<td>$ = 3</td>
</tr>
<tr>
<td></td>
<td>A</td>
</tr>
<tr>
<td></td>
<td>B</td>
</tr>
<tr>
<td></td>
<td>C</td>
</tr>
<tr>
<td></td>
<td>$</td>
</tr>
<tr>
<td>AC</td>
<td>B = 1</td>
</tr>
<tr>
<td>$ = 1</td>
<td>C = 2</td>
</tr>
<tr>
<td>A = 2</td>
<td>$ = 2</td>
</tr>
<tr>
<td>C = 1</td>
<td>$</td>
</tr>
<tr>
<td>A = 1</td>
<td>$</td>
</tr>
<tr>
<td>B = 2</td>
<td>C</td>
</tr>
<tr>
<td>C = 2</td>
<td>$</td>
</tr>
<tr>
<td>$ = 3</td>
<td>CB</td>
</tr>
<tr>
<td>CA</td>
<td>A</td>
</tr>
<tr>
<td>$ = 1</td>
<td>$</td>
</tr>
<tr>
<td>C = 1</td>
<td>$</td>
</tr>
<tr>
<td>CC</td>
<td>A</td>
</tr>
<tr>
<td>A = 1</td>
<td>B</td>
</tr>
<tr>
<td>B = 1</td>
<td>$</td>
</tr>
<tr>
<td>$ = 2</td>
<td></td>
</tr>
</tbody>
</table>

String = ACCBACCACBA

k = 2

---

## PPM: Other important optimizations

If context has not been seen before, automatically escape (no need for an escape symbol since decoder knows previous contexts)

Can exclude certain possibilities when switching down a context. This can save 20% in final length!

It is critical to use arithmetic codes since the probabilities are small.