Data compression continued…

Scribe volunteer?
Recap

Will use “message” in generic sense to mean the data to be compressed

**Lossless**: Input message = Output message

**Lossy**: Input message ≈ Output message
Recap: Model vs. Coder

To compress we need a bias on the probability of messages. The model determines this bias.

Diagram:

- Encoder
- Messages → Model → Probs. → Coder → Bits
Recap: Entropy

For a set of messages $S$ with probability $p(s)$, $s \in S$, the **self information** of $s$ is:

$$i(s) = \log \frac{1}{p(s)} = -\log p(s)$$

Measured in **bits if the log is base 2**.

**Entropy** is the weighted average of self information.

$$H(S) = \sum_{s \in S} p(s) \log \frac{1}{p(s)}$$
Recap: Assumptions and Definitions

**Message sequence:** a sequence of messages

Each message comes from a **message set** $S = \{s_1, \ldots, s_n\}$ with a **probability distribution** $p(s)$.

**Code $C(s)$:** A mapping from a message set to **codewords**, each of which is a string of bits
Recap: Uniquely Decodable Codes

A variable length code assigns a bit string (codeword) of variable length to every message value.

\[ a = 1, \ b = 01, \ c = 101, \ d = 011 \]

What if you get the sequence of bits 1011?

Is it \( \text{aba} \), \( \text{ca} \), or, \( \text{ad} \)?

A uniquely decodable code is a variable length code in which bit strings can always be uniquely decomposed into its codewords.
Recap: Prefix Codes

A **prefix code** is a variable length code in which no codeword is a prefix of another word.

e.g., \( a = 0, b = 110, c = 111, d = 10 \)

All prefix codes are uniquely decodable

Can be viewed as a binary tree with message values at the leaves and 0s or 1s on the edges

Codeword = values along the path from root to the leaf
Recap: Average Length

Let \( l(c) \) = length of the codeword \( c \) (a positive integer)

For a code \( C \) with associated probabilities \( p(c) \) the **average length** is defined as

\[
l_a(C) = \sum_{c \in C} p(c)l(c)
\]

We say that a prefix code \( C \) is **optimal** if for all prefix codes \( C' \), \( l_a(C) \leq l_a(C') \)
Recap: Relationship between Average Length and Entropy

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

\[
H(S) \leq l_a(C)
\]

(Shannon’s source coding theorem)

**Theorem (upper bound):** For any probability distribution \( p(S) \) with associated optimal prefix code \( C \),

\[
l_a(C) \leq H(S) + 1
\]
Recap: 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)
Recap: 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 \( p_1 + p_2 \)

Theorem: The Huffman algorithm generates an optimal prefix code.

Proof: (by induction)
Recap: 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 message we might hope to use

\[1000 \times .0014 = 1.44\] bits.

Using Huffman codes we require at least one bit per message, so we would require 1000 bits.
Recap: Discrete or Blended

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

```
01001 11 0001 011
```

message: 1 2 3 4

**Blended**: bits can be “shared” among messages
- Arithmetic coding

```
010010111010
```

message: 1,2,3, and 4
Arithmetic Coding: message intervals

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

e.g. a (0.2), b (0.5), c (0.3)

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)**
We call this the **sequence interval**
Arithmetic Coding: interval sizes

For a sequence of messages with message probabilities $p_i \ (i = 1..n)$

Size of intervals denoted by $s$:
$s_1 = p_1 \quad s_i = s_{i-1}p_i$

Each message narrows the interval by a factor of $p_i$.
Final interval size:
$s_n = \prod_{i=1}^{n} p_i$
Uniquely defining an interval

Q: Can sequence intervals overlap?

**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:

\[ a = .2 \]
\[ c = .3 \]
\[ b = .5 \]
\[ a = .2 \]
Arithmetic Coding: Decoding Example

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

- a = .2
- c = .3
- b = .5

0.0
0.2
0.49
0.7
1.0

0.2
0.3
0.49
0.55
0.7

a = .2
b = .5
c = .3
Arithmetic Coding: Decoding Example

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

The message is bbc.
Representing Fractions

Binary fractional representation:

\[
\begin{align*}
.75 & = .11 \\
1/3 & = .0101 \\
11/16 & = .1011
\end{align*}
\]

So how about just using the smallest binary fractional representation in the sequence interval.

\text{e.g.} \ [0,.33) = .01 \quad [.33,.66) = .1 \quad [.66,1) = .11

But what if you receive a 1?
Should we wait for another 1? Not a prefix code!
Representing an Interval

Key idea:
Can **view binary fractional numbers as intervals** by considering all completions. e.g.

\[
\begin{array}{ccc}
\text{min} & \text{max} & \text{interval} \\
0.11 & 0.110 & 0.111 & [0.75, 1.0) \\
0.101 & 0.1010 & 0.1011 & [0.625, 0.75)
\end{array}
\]

We will represent binary fractional codeword as an interval, called the **code interval**.
Q: When will code intervals overlap?
Code intervals overlap if one code is a prefix of the other.

**Lemma:** If a set of code intervals do not overlap then the corresponding codes form a prefix code.
Selecting the Code Interval

To find a prefix code find a binary fractional number whose code interval is fully contained in the sequence interval.

Sequence Interval

<table>
<thead>
<tr>
<th>Code Interval (.101)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.61 - 0.625</td>
</tr>
<tr>
<td>0.625</td>
</tr>
<tr>
<td>0.75</td>
</tr>
<tr>
<td>0.79</td>
</tr>
</tbody>
</table>

[0,.33) = ?
[.33,.66) = ?
[.66,1) = ?
Selecting a Code Interval

Recall accumulated probabilities:
E.g.: a (0.2), b (0.5), c (0.3)

Represent **message probabilities** with $p(j)$:

$$p(1) = 0.2, \quad p(2) = 0.5, \quad p(3) = 0.3$$

Accumulated probabilities $f(i)$:

$$f(i) = \sum_{j=1}^{i-1} p(j)$$

Accumulated probabilities:

- $f(1) = .0$
- $f(2) = .2$
- $f(3) = .7$
Selecting the Code Interval

Bottom of interval denoted by

\[ l + \frac{s}{2} \]

Can use the fraction \( l + \frac{s}{2} \) truncated to bits

\[
\left\lfloor -\log(s/2) \right\rfloor = 1 + \left\lfloor -\log s \right\rfloor
\]

Note: Smaller \( s \) \( \Rightarrow \) higher number of bits (higher precision)
Selecting a code interval: example

E.g. for \([0, .33)\), \(l = 0, s = .33\)

\[
\begin{align*}
  l + \frac{s}{2} &= .165 = .0010... \\
  \text{truncated to} &\quad 1 + \left\lceil -\log s \right\rceil = 1 + \left\lceil -\log(.33) \right\rceil = 3 \quad \text{bits is .001}
\end{align*}
\]
Warning

Three types of interval:

- **message interval**: interval for a single message
- **sequence interval**: composition of message intervals
- **code interval**: interval for a specific code used to represent a sequence interval
RealArith Encoding and Decoding

RealArithEncode:
Determine \( l \) and \( s \) using original recurrences
Code using \( l + \frac{s}{2} \) truncated to \( 1+\lceil -\log s \rceil \) bits

RealArithDecode:
Read bits as needed so code interval falls within a message interval, and then narrow sequence interval.
Repeat until \( n \) messages have been decoded.
\( (n \) is either predetermined or sent as a header.)
RealArith: Decoding Example

Decoding the number 0.10000, knowing the message is of length 3:

\[0.10000 = [0.5, 0.5156)\]

Code interval of:
\[0.1 = [0.5, 1)\] not within a message interval
(\text{read more bits})
\[0.10 = [0.5, 0.75)\] not within a message interval
(\text{read more bits})
\[0.100 = [0.5, 0.625) \Rightarrow b\]
Decoding the number 0.10000, knowing the message is of length 3:

\[0.10000 = [0.5, 0.5156)\]

Code interval of:
- \(0.1 = [0.5, 1)\)
- \(0.10 = [0.5, 0.75)\)
- \(0.100 = [0.5, 0.625) \Rightarrow b\)
- \(0.1000 = [0.5, 0.5625)\) not within a message interval (read more bits)
- \(0.10000 = [0.5, 0.5156) \Rightarrow b\)
### Bound on Length

**Theorem:** For \( n \) messages with self information \( \{i(s_1), \ldots, i(s_n)\} \) RealArithEncode will generate at most \(<\text{board}>\) bits.

**Proof:** Ideas?

\[
1 + \left\lceil -\log s \right\rceil = 1 + \left\lceil -\log \left( \prod_{i=1}^{n} p_i \right) \right\rceil \\
= 1 + \left\lceil \sum_{i=1}^{n} -\log p_i \right\rceil \\
\]

\(<\text{board}>\)
Integer Arithmetic Coding

Problem with RealArithCode is that operations on arbitrary precision real numbers is expensive.

Integer version (approximation to RealArith):

Key Ideas:
• Using counts instead of probabilities
• Keep integers in range \([0..R)\) where \(R=2^k\) (some power of 2)
• Use rounding to generate integer sequence interval
• Whenever sequence interval falls into top, bottom or middle half, expand the interval by factor of 2

This integer Algorithm is an approximation of the real algorithm.

(Detailed example in the notes.)
Exploiting context when compressing

The “optimality” of the code is relative to the probabilities.
If probabilities are not accurate, the code is not going to be efficient

Model can be **static or dynamic to varying degrees:**
- Static over all message sequences (predetermine (hardcoded) frequencies)
- Static over a single message sequence (execute one pass to determine prob. and then encode)
- Dynamic over the message sequence (prob. updated during encoding)
The **Static part** of the model is fixed
The **Dynamic part** is based on previous messages
Decoding: Model and Decoder

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.
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)
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? $1/2^i$
Integer codes: Gamma

Invented by Peter Elias #
“n” represented as a pair of “length” and “offset”

**Offset:** integer in binary, with the leading bit “1” removed
E.g.: 15
1111 -> 111

**Length:** (length of the offset + 1) in unary
E.g.: For above example
Length = 4 in unary = 1110
Gamma code for 15 = 1110 | 111

"Universal codeword sets and representations of the integers”, IEEE Transactions on Information Theory, March 1975
Integer codes: Gamma

“n” represented as a pair of “length” and “offset”

**Offset**: integer in binary, with the leading bit “1” removed

**Length**: (length of the offset + 1) in unary

E.g.: 15
Gamma code for 15 = 1110 | 111

Q: How to decode a Gamma code?
Read until hit a 0 => gives the length to read further

Q: How are Gamma codes fixing the issue with minimal binary?
**Integer codes: Gamma**

**Offset:** integer in binary, with the leading bit “1” removed

**Length:** (length of the offset + 1) in unary

Q: What is the length of the Gamma code?

<board>

- Always odd
- Just twice over the size of minimum binary
- **Within factor 3 of optimal for any probability distribution**
  - Hence called “universal”
# 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…
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. abbbbaaccccca => (a,1),(b,3),(a,2),(c,4),(a,1)

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

Q: Why?

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.
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.
Move to Front Coding

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

Q: Why?
Temporal locality

Takes advantage of **temporal locality**

Use of Splay tree data structure:
Encode the path and then move ("splay") it to the root

Used as a sub-step in many compression algorithms.