Reals in Binary

- Any real number $x$ in the interval $[0,1)$ can be represented in binary as $b_1b_2\ldots$ where $b_i$ is a bit.

![Binary representation](image)

First Conversion

```
L := 0; R := 1; i := 1
while x > L *
  if x < (L+R)/2 then b_i := 0 ; R := (L+R)/2;
  if x > (L+R)/2 then b_i := 1 ; L := (L+R)/2;
  i := i + 1
end{while}
```

- Invariant: $x$ is always in the interval $(L,R)$

Conversion using Scaling

- Always scale the interval to unit size, but $x$ must be changed as part of the scaling.

```
y := x; i := 0
while y > 0 *
  i := i + 1;
  if y < 1/2 then b_i := 0; y := 2y;
  if y > 1/2 then b_i := 1; y := 2y – 1;
end{while}
```

- Invariant: $x = .b_1b_2\ldots b_i + y/2$

Binary Conversion with Scaling

```
y := x; i := 0
while y > 0 *
  i := i + 1;
  if y < 1/2 then b_i := 0; y := 2y;
  if y > 1/2 then b_i := 1; y := 2y – 1;
end(while)
```

- Invariant: $x = .b_1b_2\ldots b_i + y/2$

Proof of the Invariant

- Initially $x = 0 + y/2^0$
- Assume $x = .b_1b_2\ldots b_i + y/2^i$
  - Case 1. $y < 1/2$: $b_{i+1} = 0$ and $y' = 2y$
    $b_i b_{i+1} \ldots b_{i+k} y'/2^{k+1} = b_i b_{i+1} \ldots b_i 2y 2^{-1}$
    $= x$
  - Case 2. $y \geq 1/2$: $b_{i+1} = 1$ and $y' = 2y – 1$
    $b_i b_{i+1} \ldots b_i y'/2^{k+1} = b_i b_{i+1} \ldots b_i 1 + (2y-1)/2$,
    $= x$
Example and Exercise

\[ x = \frac{1}{3} \quad x = \frac{17}{27} \]

<table>
<thead>
<tr>
<th></th>
<th>y</th>
<th>i</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>1/3</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2/3</td>
<td>2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1/3</td>
<td>3</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2/3</td>
<td>4</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

\[ \frac{17}{27} \quad y \quad i \quad b \]

17/27 1 1

Arithmetic Coding

Basic idea in arithmetic coding:
- represent each string \( x \) of length \( n \) by a unique interval \([L, R)\) in \([0,1)\).
- The width \( R-L \) of the interval \([L, R)\) represents the probability of \( x \) occurring.
- The interval \([L, R)\) can itself be represented by any number called a tag, within the half open interval.
- The \( k \) significant bits of the tag \( t_1 t_2 t_3 \ldots \) is the code of \( x \). That is, \( .t_1 t_2 t_3 \ldots 000 \) is in the interval \([L, R)\).
- It turns out that \( k \approx \log_2 \left( \frac{1}{R-L} \right) \).

Example of Arithmetic Coding (1)

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>ba</th>
<th>bba</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td>0</td>
<td>.001101000...</td>
</tr>
<tr>
<td>1/3</td>
<td>a</td>
<td></td>
<td></td>
<td>.101000010...</td>
</tr>
<tr>
<td>2/3</td>
<td>b</td>
<td></td>
<td></td>
<td>.001101000...</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>.101000010...</td>
</tr>
</tbody>
</table>

Using tag = \((L+R)/2\)

\[ \text{tag} = \frac{13}{27} = .011110110... \]

\[ \text{code} = 0111 \]

Alternative tag = \(14/27 = .100001001...\)

\[ \text{code} = 1 \]

Some Tags are Better than Others

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>bab</th>
<th>ba</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td>.011010000...</td>
<td>.011010000...</td>
</tr>
<tr>
<td>1/3</td>
<td>a</td>
<td></td>
<td>.011010000...</td>
<td>.101000010...</td>
</tr>
<tr>
<td>2/3</td>
<td>b</td>
<td></td>
<td>.011010000...</td>
<td>.101000010...</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>.101000010...</td>
<td>.101000010...</td>
</tr>
</tbody>
</table>

\[ \text{tag} = \frac{15}{27} = .100011100... \]

\[ \text{code} = 0001 \]

\[ \text{tag} = \frac{9}{27} = .011010000... \]

\[ \text{code} = 0 \]

Code Generation from Tag

- If binary tag is \( t_1 t_2 t_3 \ldots = (L+R)/2 \) in \([L, R)\) then we want to choose \( k \) to form the code \( t_1 t_2 \ldots t_k \).
- Short code:
  - choose \( k \) to be as small as possible so that \( L < t_1 t_2 \ldots t_k 000 \ldots < R \).
- Guaranteed code:
  - choose \( k = \lceil \log_2 \left( \frac{1}{R-L} \right) \rceil + 1 \)
  - \( L \leq t_1 t_2 \ldots t_k b_1 b_2 b_3 \ldots < R \) for any bits \( b_1 b_2 b_3 \ldots \)
  - for fixed length strings provides a good prefix code.
  - example: \([.000000000..., .0000010010...], \text{tag} = .000001001...\]

Short code: 0

Guaranteed code: 00001

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>bab</th>
<th>ba</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>.000001001...</td>
</tr>
<tr>
<td>1/3</td>
<td>a</td>
<td></td>
<td></td>
<td>.000001001...</td>
</tr>
<tr>
<td>2/3</td>
<td>b</td>
<td></td>
<td></td>
<td>.000001001...</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>.000001001...</td>
</tr>
</tbody>
</table>

\[ .95 \text{bits/symbol} \]

\[ .92 \text{entropy lower bound} \]
Guaranteed Code Example
• \( P(a) = \frac{1}{3}, P(b) = \frac{2}{3} \).

\[ \begin{array}{c|c|c}
0 & a & b \\
--- & --- & ---
\hline
0/27 & 0/00001001 & 0/00001aa \\
0/27 & 0/00100100 & 0/0001aab \\
0/27 & 0/01000110 & 0/001 ab \\
0/27 & 0/10110111 & 0/0111bab \\
0/27 & 0/11011111 & 0/111 bab \\
0/27 & 1/01000010 & 1/011 bba \\
0/27 & 1/01110111 & 1/110 bbb \\
0/27 & 1/10100111 & 1/101 bbb \\
0/27 & 1/11011011 & 0/11 bab \\
0/27 & 1/11101111 & 1/111 bbb \\
0/27 & 1/11111111 & 1/1 bab \\
\end{array} \]

Choose code for tag = (L+R)/2.

Arithmetic Coding Algorithm
• \( P(a_1), P(a_2), \ldots, P(a_m) \)
• \( C(a_i) = P(a_1) + P(a_2) + \ldots + P(a_i) \)
• Encode \( x_1x_2\ldots x_n \)

\[ \text{Initialize } L = 0 \text{ and } R = 1; \]
\[ \text{for } i = 1 \text{ to } n \text{ do} \]
\[ \quad W := R - L; \]
\[ \quad L := L + W * C(x_i); \]
\[ \quad R := L + W * P(x_i); \]

Choose code for the tag.

Arithmetic Coding Example
• \( P(a) = \frac{1}{4}, P(b) = \frac{1}{2}, P(c) = \frac{1}{4} \)
• \( C(a) = 0, C(b) = \frac{1}{4}, C(c) = \frac{3}{4} \)
• abca

\[ \begin{array}{c|c|c|c}
\text{symbol} & W & L & R \\
--- & --- & --- & ---
\hline
a & 0 & 1/4 & 1/4 \\
b & 1/4 & 1/16 & 3/16 \\
c & 1/8 & 5/32 & 6/32 \\
a & 1/32 & 5/32 & 21/128 \\
\end{array} \]

\[ \text{tag} = \frac{(5/32 + 21/128)/2 = 41/256 = .001010010\ldots} \]
\[ \text{L = .001010000\ldots} \]
\[ \text{R = .001010100\ldots} \]

Code = 00101
Prefix code = 00101001

Arithmetic Coding Exercise
• \( P(a) = \frac{1}{4}, P(b) = \frac{1}{2}, P(c) = \frac{1}{4} \)
• \( C(a) = 0, C(b) = \frac{1}{4}, C(c) = \frac{3}{4} \)
• bbbb

\[ \begin{array}{c|c|c|c}
\text{symbol} & W & L & R \\
--- & --- & --- & ---
\hline
b & 0 & 1 & 1 \\
b & 1/4 & 1/16 & 3/16 \\
b & 1/16 & 1/256 & 5/256 \\
b & 1/256 & 1/4096 & 5/4096 \\
\end{array} \]

\[ \text{tag} = \frac{(5/32 + 21/128)/2 = 41/256 = .001010010\ldots} \]
\[ \text{L = .001010000\ldots} \]
\[ \text{R = .001010100\ldots} \]

Code = 00101
Prefix code = 00101001

Decoding (1)
• Assume the length is known to be 3.
• 0001 which converts to the tag .0001000...

Decoding (2)
• Assume the length is known to be 3.
• 0001 which converts to the tag .0001000...
Decoding (3)

- Assume the length is known to be 3.
- 0001 which converts to the tag .0001000...

Arithmetic Decoding Algorithm

- P(a₁), P(a₂), ..., P(aₙ)
- C(aᵢ) = P(a₁) + P(a₂) + ... + P(aᵢ-1)
- Decode b₁b₂...bₙ, number of symbols is n.

```
Initialize L := 0 and R := 1;
t := .b₁b₂...bₙ000...
for i = 1 to n do
    W := R - L;
    find j such that L + W * C(a_j) < t < L + W * (C(a_j) + P(a_j))
    output a_j;
    L := L + W * C(a_j);
    R := L + W * P(a_j);
```

Decoding Example

- P(a) = 1/4, P(b) = 1/2, P(c) = 1/4
- C(a) = 0, C(b) = 1/4, C(c) = 3/4
- 00101
  tag = .001000... = 5/32
  W  L  R  output
  0  1  0  a
  1/4  1/16  3/16  b
  1/8  5/32  6/32  c
  1/32  5/32  21/128  a

Decoding Issues

- There are at least two ways for the decoder to know when to stop decoding.
  1. Transmit the length of the string
  2. Transmit a unique end of string symbol

Practical Arithmetic Coding

- Scaling:
  - By scaling we can keep L and R in a reasonable range of values so that W = R - L does not underflow.
  - The code can be produced progressively, not at the end.
  - Complicates decoding some.
- Integer arithmetic coding avoids floating point altogether.

More Issues

- Context
- Adaptive
- Comparison with Huffman coding
Scaling

- By scaling we can keep L and R in a reasonable range of values so that \( W = R - L \) does not underflow.
- The code can be produced progressively, not at the end.
- Complicates decoding some.

Scaling during Encoding

**Lower half**
- If \([L, R)\) is contained in \([0, .5)\) then
  - \( L := 2L; R := 2R \)
  - output 0, followed by \( C \) 1's
  - \( C := 0 \)

**Upper half**
- If \([L, R)\) is contained in \([.5, 1)\) then
  - \( L := 2L - 1; R := 2R - 1 \)
  - output 1, followed by \( C \) 0's
  - \( C := 0 \)

**Middle half**
- If \([L, R)\) is contained in \([.25, .75)\) then
  - \( L := 2L - .5; R := 2R -.5 \)
  - \( C := C + 1 \)

Example

- **baa**

  - \( L = 1/3; R = 3/3 \)
  - \( C = 0 \)

  - Scale middle half

  - \( L = 3/9; R = 5/9 \)
  - \( C = 0 \)

Example

- **baa**

  - \( L = 3/9; R = 5/9 \)
  - \( L = 3/18; R = 11/18 \)
  - \( C = 1 \)

  - Scale lower half

  - \( L = 3/18; R = 11/18 \)
  - \( L = 9/54; R = 17/54 \)
Example

\[
\begin{align*}
\text{baa 01} \\
C = 0 \\
L = 9/54 \quad R = 17/54 \\
L = 18/54 \quad R = 34/54 \\
\end{align*}
\]

In end \(L < \frac{1}{2} < R\), choose tag to be 1/2

\[
\begin{align*}
\text{baa 011} \\
C = 0 \\
L = 9/54 \quad R = 17/54 \\
L = 18/54 \quad R = 34/54 \\
\end{align*}
\]

Exercise

Model: \(a: 1/4; b: 3/4\)

Encode: bba

Decoding

The decoder behaves just like the encoder except that \(C\) does not need to be maintained.

Instead, the input stream is consumed during scaling.

Scaling during Decoding

- **Lower half**
  - If \([L,R)\) is contained in \([0,.5)\) then
    - \(L := 2L\) \(R := 2R\)
    - consume 0 from the encoded stream
- **Upper half**
  - If \([L,R)\) is contained in \([.5,1)\) then
    - \(L := 2L - 1, R := 2R - 1\)
    - consume 1 from the encoded stream
- **Middle half**
  - If \([L,R)\) is contained in \([.25,.75)\) then
    - \(L := 2L - .5, R := 2R - .5\)
    - Replace 01 with 0 on stream
    - Replace 10 with 1 on stream

Scaling Math for the Tag

- **Lower Half**
  - \(.0b, b_2, \ldots \times 10 = .b, b_2\)
- **Upper Half**
  - \(.1b, b_2, \ldots \times 10 - 1 = .b_1b_2\)
- **Middle Half**
  - \(.01b, b_2, \ldots \times 10 - .1 = .0b, b_2\)
  - \(.10b, b_2, \ldots \times 10 - .1 = .1b, b_2\)
Exercise
Model: \(a: \frac{1}{4}; \ b: \frac{3}{4}\)
Decode: 001 to 3 symbols

Integer Implementation
• \(m\) bit integers
  – Represent 0 with \(000...0\) (\(m\) times)
  – Represent 1 with \(111...1\) (\(m\) times)
• Probabilities represented by frequencies
  – \(n_i\) is the number of times that symbol \(a_i\) occurs
  – \(C_i = n_1 + n_2 + ... + n_{i-1}\)
  – \(N = n_1 + n_2 + ... + n_m\)

\[
\begin{align*}
W &= R - L + 1 \\
L_i' &= L_i + \frac{W C_i}{N} \\
R_i &= L_i + \frac{W C_i}{N} - 1
\end{align*}
\]
Coding the \(i\)th symbol using integer calculations.
Must use scaling!

Context
• Consider 1 symbol context.
• Example: 3 contexts.

\[
\begin{array}{ccc}
\text{prev} & \text{next} & \text{a} \quad \text{b} \quad \text{c} \\
\hline
a & .4 & .2 \quad .4 \ \\
b & .1 & .8 \quad .1 \ \\
c & .25 & .25 \quad .5 \ \\
\end{array}
\]

Example with Scaling
\[
\begin{array}{ccc}
\text{prev} & \text{next} & \text{a} \quad \text{b} \quad \text{c} \\
\hline
0 & 10 & 1 \ \\
\hline
0 & .4 & .2 \quad .3 \ \\
1/3 & .25 & .25 \ \\
1/3 & .25 & .25 \ \\
3/10 & 17/30 & .5 \ \\
\hline
\text{Code} = 0101
\end{array}
\]

Arithmetic Coding with Context
• Maintain the probabilities for each context.
• For the first symbol use the equal probability model
• For each successive symbol use the model for the previous symbol

Adaptation
• Simple solution – Equally Probable Model.
  – Initially all symbols have frequency 1.
  – After symbol \(x\) is coded, increment its frequency by 1
  – Use the new model for coding the next symbol
• Example in alphabet \(a,b,c,d\)

\[
\begin{array}{cccc}
a & b & a & c \\
1 & 2 & 3 & 4 \ 5 \ 5 \\
b & 1 & 1 & 2 \ 2 \ 2 \\
c & 1 & 1 & 1 \ 1 \ 2 \\
d & 1 & 1 & 1 \ 1 \ 1 \\
\end{array}
\]
After aabaac is encoded
The probability model is
\[
\begin{array}{cccc}
a & b & a & c \\
5/10 & 2/10 & \ \\
5/6 & 2/5 & 17/30 \ & .5 \\
\hline
2/3 & 2/15 & \ & \ \\
\end{array}
\]
Zero Frequency Problem

- How do we weight symbols that have not occurred yet.
  - Equal weights? Not so good with many symbols
  - Escape symbol, but what should its weight be?
  - When a new symbol is encountered send the <esc>, followed by the symbol in the equally probable model. (Both encoded arithmetically.)

\[
\begin{array}{c|cccc}
  a & b & a & a & c \\
  a & 0 & 1 & 2 & 2 & 3 & 4 & 4 \\
  b & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
  c & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\
  d & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
  \text{<esc>} & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
\end{array}
\]

After aabaac is encoded

The probability model is

\[
\begin{array}{c|c|c}
  a & 4/7 & b & 1/7 \\
  c & 1/7 & d & 0 \\
  \text{<esc>} & 1/7 \\
\end{array}
\]

PPM

- Prediction with Partial Matching
  - Cleary and Witten (1984)
- State of the art arithmetic coder
  - Arbitrary order context
  - Adaptive

Needs good data structures to be efficient.

PPM Example

\[
\begin{array}{c|c|c|c|c|c|c|c}
  0-order context & a & b & r & c & d & \text{<esc>} \\
  a & 3 & 1 & 1 & 1 & 1 & 1 \\
  b & 1 & 1 & 1 & 1 & 1 & 1 \\
  r & 1 & 1 & 1 & 1 & 1 & 1 \\
  c & 1 & 1 & 1 & 1 & 1 & 1 \\
  d & 1 & 1 & 1 & 1 & 1 & 1 \\
  \text{<esc>} & 1 & 1 & 1 & 1 & 1 & 1 \\
\end{array}
\]

1st-order context

\[
\begin{array}{c|c|c|c|c|c|c|c}
  a & b & a & r & c & d & \text{<esc>} \\
  a & 1 & 1 & 1 & 1 & 1 & 1 \\
  b & 1 & 1 & 1 & 1 & 1 & 1 \\
  r & 1 & 1 & 1 & 1 & 1 & 1 \\
  c & 1 & 1 & 1 & 1 & 1 & 1 \\
  d & 1 & 1 & 1 & 1 & 1 & 1 \\
  \text{<esc>} & 1 & 1 & 1 & 1 & 1 & 1 \\
\end{array}
\]

2nd-order context

\[
\begin{array}{c|c|c|c|c|c|c|c}
  a & b & a & a & r & c & d & \text{<esc>} \\
  a & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
  b & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
  a & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
  r & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
  c & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
  d & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
  \text{<esc>} & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
\end{array}
\]

Output

1-order <esc>

0-order <esc>

(-1)-order d.

Update tables
Arithmetic vs. Huffman

- Both compress very well. For m symbol grouping.
  - Huffman is within 1/m of entropy.
  - Arithmetic is within 2/m of entropy.
- Context
  - Huffman needs a tree for every context.
  - Arithmetic needs a small table of frequencies for every context.
- Adaptation
  - Huffman has an elaborate adaptive algorithm
  - Arithmetic has a simple adaptive mechanism.
- Bottom Line — Arithmetic is more flexible than Huffman.