Reals in Binary

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

First Conversion

\[
\begin{align*}
L &:= 0; R := 1; i := 1 \\
\text{while } x > L \ast \\
& \quad \text{if } x < (L+R)/2 \text{ then } b_i := 0; R := (L+R)/2; \\
& \quad \quad i := i + 1 \\
& \quad \text{if } x > (L+R)/2 \text{ then } b_i := 1; L := (L+R)/2; \\
& \quad \quad i := i + 1 \\
& \end{while} \\
b_j := 0 \text{ for all } j > i \ast \\
\end{align*}
\]

* 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.

Binary Conversion with Scaling

\[
\begin{align*}
y &:= x; i := 0 \\
\text{while } y > 0 \ast \\
& \quad i := i + 1; \\
& \quad \text{if } y < 1/2 \text{ then } b_i := 0; y := 2y; \\
& \quad \quad \ldots b_1b_2 \ldots y/2^{i+1} = b_1 \ldots b_i \left( b_{i+1} + y/2^{i+1} \right) \\
& \quad \text{if } y > 1/2 \text{ then } b_i := 1; y := 2y - 1; \\
& \quad \quad \ldots b_1b_2 \ldots y/2^{i+1} = b_1 \ldots b_i + y/2^{i+1} \\
& \quad \text{end(while)} \\
b_j := 0 \text{ for all } j > i + 1 \ast \\
\end{align*}
\]

* Invariant: \( x = .b_1b_2 \ldots b_i \leq y/2^n \)

Proof of the Invariant

- Initially \( x = 0 + y/2^n \)
- Assume \( x = .b_1b_2 \ldots b_i + y/2^n \)
  - Case 1. \( y < 1/2 \): \( b_i = 0 \) and \( y' = 2y \)
    \[
    .b_1b_2 \ldots b_i \left( y/2^{i+1} = b_1 \ldots b_i \left( b_{i+1} + y/2^{i+1} \right) = .b_1b_2 \ldots b_i + y/2^n \right)
    = x
    \]
  - Case 2. \( y \geq 1/2 \): \( b_i = 1 \) and \( y' = 2y - 1 \)
    \[
    .b_1b_2 \ldots b_i \left( y/2^{i+1} = b_1 \ldots b_i \left( b_{i+1} + y/2^{i+1} \right) = .b_1b_2 \ldots b_i + 1/2^{i+1} + 2y/2^{i+1} - 1/2^{i+1} = .b_1b_2 \ldots b_i + y/2^n \right)
    = x
    \]
Example and Exercise

\[
x = \frac{1}{3} \quad x = \frac{17}{27}
\]
\[
y \quad i \quad b
\]
\[
\begin{array}{ccc}
1/3 & 1 & 0 \\
2/3 & 2 & 1 \\
1/3 & 3 & 0 \\
2/3 & 4 & 1 \\
\vdots & \vdots & \vdots \\
\end{array}
\]

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 \( t_1t_2t_3... \) is the code of \( x \). That is, \( \ldots t_1t_2t_3000... \) is in the interval \([L, R)\).
- It turns out that \( k = \log_2(1/(R-L)) \).

Example of Arithmetic Coding (1)

1. tag must be in the half open interval.
2. tag can be chosen to be \((L+R)/2\).
3. code is the significant bits of the tag.

Some Tags are Better than Others

Code Generation from Tag

- If binary tag is \( t_1t_2t_3... = (L+R)/2 \) in \([L, R)\) then we want to choose \( k \) to form the code \( t_1t_2t_3... \).
- Short code:
  - choose \( k \) to be as small as possible so that \( L \leq t_1t_2t_3...000... < R \).
- Guaranteed code:
  - choose \( k = \lceil \log_2(1/(R-L)) \rceil +1 \)
  - \( L \leq t_1b_1t_2b_2t_3b_3... < R \) for any bits \( b_1b_2b_3... \)
  - for fixed length strings provides a good prefix code.
  - example: \(.000000000...000100100...\), tag = \(.000001001...\)
  - Short code: 0
  - Guaranteed code: 000001

Example of Codes

\[
P(a) = \frac{1}{3}, P(b) = \frac{2}{3}.
\]

\[
\begin{array}{lll}
tag = \frac{(L+R)/2}{2} & \text{code}
\end{array}
\]

\[
\begin{array}{cccccc}
0.07 & 0000000000 & 0 & \text{aa} \\
0.37 & 0000010000 & 000010001 & \text{a}a \\
0.77 & 0001001110 & 001111010 & \text{aaa} \\
a & 0.97 & 0011101000 & 001011100 & \text{aab} \\
ab & 0.92 & 0100101110 & 010111100 & \text{abb} \\
bb & 0.15 & 0111001110 & 011101110 & \text{abb} \\
b & 0.32 & 1000101100 & 101000100 & \text{bba} \\
b & 0.82 & 1001010000 & 110110100 & \text{bba} \\
0 & 0.27 & 1111111111 & 000000000 & \text{bba} \\
1 & 0.73 & 1111111111 & 000000000 & \text{bba}
\end{array}
\]

95 bits/symbol .92 entropy lower bound
Guaranteed Code Example

- \( P(a) = \frac{1}{3}, P(b) = \frac{2}{3} \)

<table>
<thead>
<tr>
<th>Symbol</th>
<th>W</th>
<th>L</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0.27</td>
<td>0.27</td>
<td>0.27</td>
</tr>
<tr>
<td>b</td>
<td>0.0000101001...</td>
<td>0</td>
<td>0.001</td>
</tr>
<tr>
<td>ab</td>
<td>000101100...</td>
<td>0.001</td>
<td>0.001</td>
</tr>
<tr>
<td>abb</td>
<td>010000101...</td>
<td>0.01</td>
<td>0.001</td>
</tr>
<tr>
<td>bba</td>
<td>010111110...</td>
<td>0.0101</td>
<td>0.111</td>
</tr>
<tr>
<td>bab</td>
<td>011110111...</td>
<td>0.0111</td>
<td>0.0111</td>
</tr>
<tr>
<td>bbb</td>
<td>11.010110100...</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>bbbb</td>
<td>27/27</td>
<td>27/27</td>
<td>27/27</td>
</tr>
</tbody>
</table>

Short code
- \( a \rightarrow .0000000... \)
- \( b \rightarrow .0001001... \)
- \( ab \rightarrow .0010000... \)
- \( abb \rightarrow .0100001... \)
- \( bba \rightarrow .0101111... \)
- \( bab \rightarrow .0111101... \)
- \( bbb \rightarrow .11.0101100... \)
- \( bbbb \rightarrow .27/27 \)

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 \( x_1 x_2 \ldots x_n \)

**Initialize** \( L := 0 \) and \( R := 1 \):
for \( i = 1 \) to \( n \) do:
- \( W := R - L \)
- \( L := L + W \times C(x_i) \)
- \( R := L + W \times P(x_i) \)
- \( t := (L + R) / 2 \)
- Choose code for the tag

Arithmetic Coding Example

- \( P(a) = \frac{1}{4}, P(b) = \frac{1}{2}, P(c) = \frac{1}{4} \)
- \( C(a) = 0, C(b) = \frac{1}{4}, C(c) = \frac{3}{4} \)
- \( abca \)

symbol | W | L | R |
--------|---|---|---|
| a      | 0 | 1 | 1/4 |
| b      | 1/4 | 1/16 | 3/16 |
| c      | 1/8 | 5/32 | 6/32 |
| ab     | 1/32 | 5/32 | 21/128 |

tag = (5/32 + 21/128)/2 = 41/256 = .001010010...
L = .001010000...
R = .001010100...
code = 00101
prefix code = 00101001

Arithmetic Coding Exercise

- \( P(a) = \frac{1}{4}, P(b) = \frac{1}{2}, P(c) = \frac{1}{4} \)
- \( C(a) = 0, C(b) = \frac{1}{4}, C(c) = \frac{3}{4} \)
- \( bbbb \)

symbol | W | L | R |
--------|---|---|---|
| b      | 1 | 0 | 1 |
| b      | 1/4 | 1/32 | 3/16 |
| c      | 1/8 | 5/32 | 6/32 |
| ab     | 1/32 | 5/32 | 21/128 |

tag = L = R = code = prefix code =

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

• \( P(a_1), P(a_2), \ldots, P(a_n) \)
• \( C(a_i) = P(a_1) + P(a_2) + \ldots + P(a_{i-1}) \)
• Decode \( b_1b_2b_3 \ldots b_n \), number of symbols is \( n \).

Initialize \( L := 0 \) and \( R := 1 \);

\[ t := .b_1b_2b_3 \ldots b_k 000 \ldots \]

for \( i = 1 \) to \( n \) do

\[ W := R - L; \]

find \( j \) such that \( L + W \cdot C(a_j) < t < L + W \cdot (C(a_j) + P(a_j)) \)

output \( a_j \);

\[ L := L + W \cdot C(a_j); \]

\[ R := L + W \cdot P(a_j); \]

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

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