### 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.

\[
\begin{array}{c}
0.1 \ 0 \ 1 \ \ldots \\
\hline
x
\end{array}
\]

**binary representation**

### First Conversion

\[
L := 0; R := 1; i := 1
\]

while \( x > L \star 
\]

if \( x < (L+R)/2 \) then \( b_i := 0 ; U := (L+R)/2 ; i := i + 1 \)

if \( x > (L+R)/2 \) then \( b_i := 1 ; L := (L+R)/2 \); \( i := i + 1 \)

end {while}

\( b_j := 0 \) for all \( j \geq i + 1 \)

* 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.

\[
\begin{array}{c}
0.1 \ 0 \ 1 \ \ldots \\
\hline
x
\end{array}
\]

### Binary Conversion with Scaling

\[
y := x; i := 0
\]

while \( y > 0 \star 
\]

if \( y < 1/2 \) then \( b_i := 0 ; y := 2y \)

if \( y \geq 1/2 \) then \( b_i := 1 ; y := 2y - 1 \)

end {while}

\( b_j := 0 \) for all \( j \geq i + 1 \)

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

### Proof of the Invariant

- Initially \( x = 0 + y/2^0 \)
- Assume \( x = b_0 b_1 \ldots b_i + y/2^i \)
  - Case 1. \( y < 1/2 \): \( b_{i+1} = 0 \) and \( y' = 2y \)
    \[
    b_0 b_1 \ldots b_i + y/2^{i+1} = b_0 b_1 \ldots b_i + 2y/2^{i+1}
    \]
    \( = x \)
  - Case 2. \( y \geq 1/2 \): \( b_{i+1} = 1 \) and \( y' = 2y - 1 \)
    \[
    b_0 b_1 \ldots b_i + y/2^{i+1} = b_0 b_1 \ldots b_i + (2y-1)/2^{i+1}
    \]
    \( = x \)
Example and Exercise

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

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

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.

Example of Codes

- \( P(a) = \frac{1}{3}, P(b) = \frac{2}{3} \)
- Tag = \((L+R)/2\)
- Code

Some Tags are Better than Others

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.

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 \( L \leq t_1 t_2 \ldots t_k 000 \ldots < R \).
- Guaranteed code:
  - Choose \( k = \lceil \log_2 (1/(R-L)) \rceil + 1 \)
  - \( L \leq t_1 t_2 t_3 \ldots t_k b_1 b_2 b_3 \ldots < R \) for any bits \( b_1 b_2 b_3 \ldots \)
  - For fixed length strings provides a good prefix code.
- Example: \([.0000000000, .0000010010..) \), tag = .000001001...
- Guaranteed code: 0

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 \( t_1 t_2 \ldots t_k \) is the code of \( x \). That is, \( t_1 t_2 \ldots t_k 000 \ldots \) is in the interval \([L,R)\).
Guaranteed Code Example

* P(a) = 1/3, P(b) = 2/3.

| a | 1/3 | 0/97 |
| b | 2/3 | 0/97 |

| a a |
| a b |
| b a |
| b b |

| a a a |
| a a b |
| a b a |
| a b b |
| b a a |
| b a b |
| b b a |
| b b b |

Arithmetic Coding Algorithm

* P(a), P(b), ..., P(a_n)
* C(a_i) = P(a_1) + P(a_2) + ... + P(a_i)
* Encode x_1x_2...x_n

Arithmetic Coding Example

* P(a) = 1/4, P(b) = 1/2, P(c) = 1/4
* C(a) = 0, C(b) = 1/4, C(c) = 3/4

<table>
<thead>
<tr>
<th>symbol</th>
<th>W</th>
<th>L</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0</td>
<td>1</td>
<td>1/4</td>
</tr>
<tr>
<td>b</td>
<td>1/4</td>
<td>1/16</td>
<td>3/16</td>
</tr>
<tr>
<td>c</td>
<td>1/8</td>
<td>5/32</td>
<td>6/32</td>
</tr>
<tr>
<td>a</td>
<td>1/32</td>
<td>5/32</td>
<td>21/128</td>
</tr>
</tbody>
</table>

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

- \( P(a_1), P(a_2), \ldots, P(a_m) \)
- \( C(a) = P(a_1) + P(a_2) + \ldots + P(a_i) \)
- Decode \( b_1b_2\ldots b_m \), number of symbols is \( n \).

1. Initialize \( L := 0 \) and \( R := 1 \);
2. \( t := b_1b_2\ldots b_n000... \)
3. For \( i = 1 \) to \( n \) do
   - \( W := R - L \)
   - Find \( j \) such that \( L + W \times C(a_j) < t < L + W \times (C(a_j) + P(a_j)) \)
   - Output \( a_j \);
   - \( L := L + W \times C(a_j) \);
   - \( R := L + W \times P(a_j) \);

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

<table>
<thead>
<tr>
<th>Tag</th>
<th>W</th>
<th>L</th>
<th>R</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>.00101000</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>a</td>
</tr>
<tr>
<td>1/4</td>
<td>1/16</td>
<td>1/16</td>
<td>b</td>
<td></td>
</tr>
<tr>
<td>1/8</td>
<td>5/32</td>
<td>6/32</td>
<td>c</td>
<td></td>
</tr>
<tr>
<td>1/32</td>
<td>5/32</td>
<td>21/128</td>
<td>a</td>
<td></td>
</tr>
</tbody>
</table>

Decoding Issues

- There are 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