Arithmetic Coding

National Chiao Tung University
Chun-Jen Tsai
10/09/2014

About Large Block Coding

- Huffman coding is inefficient if the probability model is biased (e.g. $P_{max} >> 0.5$). Although extended Huffman coding fixes this issue, it is expensive:
  - The codebook size increases exponentially w.r.t. alphabet set size
- Key idea:

  Can we assign codewords to a long sequences of symbols without generating codes for all possible sequences of the same length?

  Solution: Arithmetic Coding

Arithmetic Coding Background

- History
  - Shannon started using cumulative density function for codeword design
  - Original idea by Elias (Huffman's classmate) in early 1960s
  - First practical approach published in 1976, by Rissanen (IBM)
  - Made well-known by a paper in Communication of the ACM, by Witten et al. in 1987†
- Arithmetic coding addresses two issues in Huffman coding:
  - Integer codeword length problem
  - Adaptive probability model problem

† I.H. Witten, R.M. Neal, and J.G. Cleary, "Arithmetic coding for data compression," *Communication of the ACM*, 30, 6(June), 1987, pp. 520-540

Two-Steps of Coding Messages

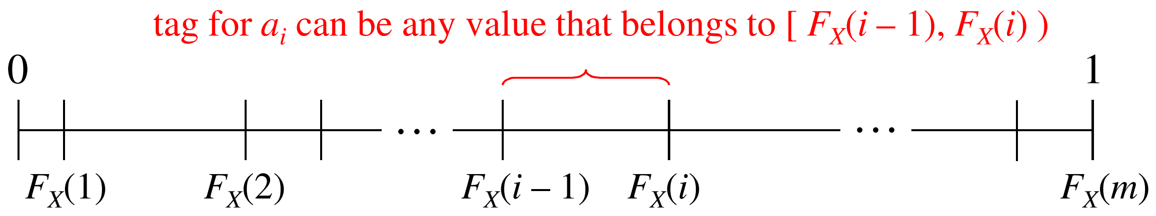
- To encode a long message into a single codeword without using a large codebook, we must
  - Step I: use a (hash) function to compute an ID (or tag) for the message. The function should be invertible
  - Step II: Given an ID (tag), assign a codeword for it using simple rules (e.g. maybe something similar to Golomb codes?), hence, there is no need to build a large codebook
- Arithmetic coding is an example of how these two steps can be achieved by using cumulative density function (CDF) as the hash function

CDF for Tag Generation

- Given a source alphabet $\mathcal{A} = \{a_1, a_2, \ldots, a_m\}$, a random variable $X(a_i) = i$, and a probability model $\mathcal{P}$: $P(X = i) = P(a_i)$. The CDF is defined as:

$$F_X(i) = \sum_{k=1}^{i} P(X = k).$$

- CDF divides $[0, 1)$ into disjoint subintervals:

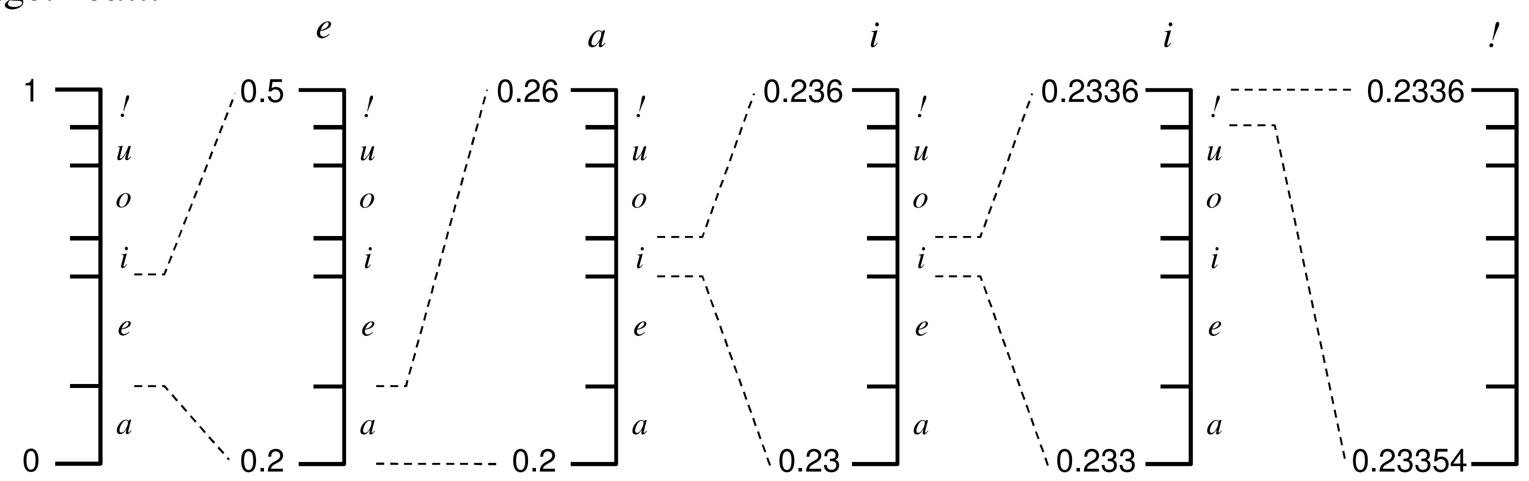


Example of Tag Generation

- In arithmetic coding, each symbol is mapped to an interval

Symbol	Probability	Interval
a	.2	[0, 0.2)
e	.3	[0.2, 0.5)
i	.1	[0.5, 0.6)
o	.2	[0.6, 0.8)
u	.1	[0.8, 0.9)
!	.1	[0.9, 1.0)

message: "*eaii!*"



Tag Selection for a Message (1/2)

- Since the intervals of messages are disjoint, we can pick any values from the interval as the tag
  - A popular choice is the lower limit of the interval
- Single symbol example: if the mid-point of the interval $[F_X(a_{i-1}), F_X(a_i))$ is used as the tag $T_X(a_i)$ of symbol a_i, then

$$T_X(a_i) = \sum_{k=1}^{i-1} P(X = k) + \frac{1}{2} P(X = i)$$

$$= F_X(i-1) + \frac{1}{2} P(X = i).$$

Note that: the function $T_X(a_i)$ is invertible.

Tag Selection for a Message (2/2)

- To generate a unique tag for a long message, we need an ordering on all message sequences
  - A logical choice of such ordering rule is the lexicographic ordering of the message
- With lexicographical ordering, for all messages of length m, we have

$$T_X^{(m)}(\mathbf{x}_i) = \sum_{\mathbf{y}<\mathbf{x}_i} P(\mathbf{y}) + \frac{1}{2} P(\mathbf{x}_i),$$

  where $\mathbf{y} < \mathbf{x}_i$ means $\mathbf{y}$ precedes $\mathbf{x}_i$ in the ordering of all messages.
- Bad news: need $P(\mathbf{y})$ for all $\mathbf{y} < \mathbf{x}_i$ to compute $T_X(\mathbf{x}_i)$!

Recursive Computation of Tags (1/3)

- Assume that we want to code the outcome of rolling a fair die for three times. Let's compute the upper and lower limits of the message "3-2-2."
  - For the first outcome "3," we have

$$l^{(1)} = F_X(2), \quad u^{(1)} = F_X(3).$$

  - For the second outcome "2," we have upper limit

$$\begin{aligned} F_X^{(2)}(32) &= [P(x_1 = 1) + P(x_1 = 2)] + P(\mathbf{x} = 31) + P(\mathbf{x} = 32) \\ &= F_X(2) + P(x_1 = 3)P(x_2 = 1) + P(x_1 = 3)P(x_2 = 2) \\ &= F_X(2) + P(x_1 = 3)F_X(2) = F_X(2) + [F_X(3) - F_X(2)]F_X(2). \end{aligned}$$

Thus, $u^{(2)} = l^{(1)} + (u^{(1)} - l^{(1)})F_X(2)$.

Similarly, the lower limit $F_X^{(2)}(31)$ is $l^{(2)} = l^{(1)} + (u^{(1)} - l^{(1)})F_X(1)$.

Recursive Computation of Tags (2/3)

- For the third outcome "2," we have

$$l^{(3)} = F_X^{(3)}(321), \quad u^{(3)} = F_X^{(3)}(322).$$

Using the same approach above, we have

$$F_X^{(3)}(321) = F_X^{(2)}(31) + [F_X^{(2)}(32) - F_X^{(2)}(31)]F_X(1).$$
$$F_X^{(3)}(322) = F_X^{(2)}(31) + [F_X^{(2)}(32) - F_X^{(2)}(31)]F_X(2).$$

Therefore,

$$l^{(3)} = l^{(2)} + (u^{(2)} - l^{(2)})F_X(1), \text{ and}$$
$$u^{(3)} = l^{(2)} + (u^{(2)} - l^{(2)})F_X(2).$$

Recursive Computation of Tags (3/3)

- In general, we can show that for any sequence $\mathbf{x} = (x_1 x_2 \ldots x_n)$,

$$l^{(n)} = l^{(n-1)} + (u^{(n-1)} - l^{(n-1)})F_X(x_n - 1)$$
$$u^{(n)} = l^{(n-1)} + (u^{(n-1)} - l^{(n-1)})F_X(x_n).$$

  If the mid-point is used as the tag, then

$$T_X(\mathbf{x}) = \frac{u^{(n)} + l^{(n)}}{2}.$$

- Note that we only need the CDF of the source alphabet to compute the tag of any long messages!

Deciphering The Tag

- The algorithm to deciphering the tag is quite straightforward:
  1. Initialize $l^{(0)} = 0$, $u^{(0)} = 1$.
  2. For each k, $k \geq 1$, find $t^* = (T_X(\mathbf{x}) - l^{(k-1)})/(u^{(k-1)} - l^{(k-1)})$.
  3. Find the value of x_k for which $F_X(x_k - 1) \leq t^* \leq F_X(x_k)$.
  4. Update $u^{(k)}$ and $l^{(k)}$.
  5. If there are more symbols, go to step 2.
- In practice, a special "end-of-sequence" symbol is used to signal the end of a sequence.

Example of Decoding Tag

- Given $\mathcal{A} = \{1, 2, 3\}$, $F_X(1) = 0.8$, $F_X(2) = 0.82$, $F_X(3) = 1$, $l^{(0)} = 0$, $u^{(0)} = 1$. If the tag is $T_X(\mathbf{x}) = 0.772352$, what is $\mathbf{x}$?

$t^* = (0.772352 - 0)/(1 - 0) = 0.772352$
$F_X(0) = 0 \le t^* \le 0.8 = F_X(1)$
$l^{(1)} = 0$, $u^{(1)} = 0.8$.

$\longrightarrow$ 1

Note:
$l^{(n)} = l^{(n-1)} + (u^{(n-1)} - l^{(n-1)})F_X(x_n - 1)$
$u^{(n)} = l^{(n-1)} + (u^{(n-1)} - l^{(n-1)})F_X(x_n)$

$t^* = (0.772352 - 0)/(0.8 - 0) = 0.96544$
$F_X(2) = 0.82 \le t^* \le 1 = F_X(3)$
$l^{(2)} = 0.656$, $u^{(2)} = 0.8$.

$\longrightarrow$ 13

$t^* = (0.772352 - 0.656)/(0.8 - 0.656) = 0.808$
$F_X(1) = 0.8 \le t^* \le 0.82 = F_X(2)$
$l^{(3)} = 0.7712$, $u^{(3)} = 0.77408$.

$\longrightarrow$ 132

$t^* = (0.772352 - 0.7712)/(0.77408 - 0.7712) = 0.4$
$F_X(1) = 0 \le t^* \le 0.8 = F_X(1)$

$\longrightarrow$ 1321

Binary Code for the Tag

- If the **mid-point** of an interval is used as the tag $T_X(x)$, a binary code for $T_X(x)$ is the binary representation of the number truncated to $l(x) = \lceil \log(1/P(x)) \rceil + 1$ bits.

- For example, $\mathcal{A} = \{ a_1, a_2, a_3, a_4 \}$ with probabilities $\{ 0.5, 0.25, 0.125, 0.125 \}$, a binary code for each symbol is as follows:

Symbol	F_X	$\overline{T}_X$	In Binary	$\lceil \log \frac{1}{P(x)} \rceil + 1$	Code
1	.500	.2500	.0100	2	01
2	.750	.6250	.1010	3	101
3	.875	.8125	.1101	4	1101
4	1.000	.9375	.1111	4	1111

- The binary code for a message is defined recursively!

Unique Decodability of the Code

- Note that the tag $T_X(\mathbf{x})$ uniquely specifies the interval $[F_X(\mathbf{x}-1), F_X(\mathbf{x}))$, if $\lfloor T_X(\mathbf{x}) \rfloor_{l(\mathbf{x})}$ is still in the interval, it is unique. Since $\lfloor T_X(\mathbf{x}) \rfloor_{l(\mathbf{x})} > F_X(\mathbf{x}-1)$ because $1/2^{l(x)} < P(x)/2 = T_X(\mathbf{x}) - F_X(\mathbf{x}-1)$, we know $\lfloor T_X(\mathbf{x}) \rfloor_{l(\mathbf{x})}$ is still in the interval.

- To show that the code is uniquely decodable, we can show that the code is a prefix code. This is true because $[\lfloor T_X(\mathbf{x}) \rfloor_{l(\mathbf{x})}, \lfloor T_X(\mathbf{x}) \rfloor_{l(\mathbf{x})} + (1/2^{l(\mathbf{x})})\ ) \subset [F_X(\mathbf{x}-1), F_X(\mathbf{x}))$. Therefore, any other code outside the interval $[F_X(\mathbf{x}-1), F_X(\mathbf{x}))$ will have a different $l(\mathbf{x})$-bit prefix.

Efficiency of Arithmetic Codes

- The average code length of a source $A^{(m)}$ is:

$$l_{A^{(m)}} = \sum P(\mathbf{x}) l(\mathbf{x}) = \sum P(\mathbf{x}) \left[\left\lceil \log \frac{1}{P(\mathbf{x})} \right\rceil + 1 \right]$$

$$< \sum P(\mathbf{x}) \left[\log \frac{1}{P(\mathbf{x})} + 1 + 1 \right] = -\sum P(\mathbf{x}) \log P(\mathbf{x}) + 2 \sum P(\mathbf{x})$$

$$= H(X^{(m)}) + 2.$$

Recall that for i.i.d. sources, $H(X^{(m)}) = mH(X)$. Thus,

$$H(X) \le l_A \le H(X) + \frac{2}{m}.$$

Arithmetic Coding Implementation

- Previous formulation for coding works, but we need real numbers with undetermined precision to work
  - Eventually $l^{(n)}$ and $u^{(n)}$ will be close enough to identify the message, but could take long iterations
  - To avoid recording long real numbers, we can sequentially outputs known digits, and rescale the interval as follows:

$$E_1: [0, 0.5) \rightarrow [0, 1); \quad E_1(x) = 2x$$
$$E_2: [0.5, 1) \rightarrow [0, 1); \quad E_2(x) = 2(x - 0.5).$$

- As interval narrows, we have one of three cases
  1. $[l^{(n)}, u^{(n)}] \subset [0, 0.5) \rightarrow$ output 0, then perform E_1 rescale
  2. $[l^{(n)}, u^{(n)}] \subset [0.5, 1) \rightarrow$ output 1, then perform E_2 rescale
  3. $l^{(n)} \in [0, 0.5), u^{(n)} \in [0.5, 1) \rightarrow$ output undetermined

Implementation Key Points

- Principle
  - Scale and shift simultaneously x, upper bound, and lower bound will gives us the same relative location of the tag.
- Encoder
  - Once we reach case 1 or 2, we can ignore the other half of $[0,1)$ by sending all the prefix bits so far to the decoder
  - Rescale tag interval to $[0, 1)$ by using $E_1(x)$ or $E_2(x)$.
- Decoder
  - Scale the tag interval in sync with the encoder

Tag Generation with Scaling (1/3)

- Consider $X(a_i) = i$, encode 1 3 2 1, given the model: Given $\mathcal{A} = \{1, 2, 3\}$, $F_X(1) = 0.8$, $F_X(2) = 0.82$, $F_X(3) = 1$, $l^{(0)} = 0$, $u^{(0)} = 1$.

Input: <u>1</u>321
$l^{(1)} = l^{(0)} + (u^{(0)} - l^{(0)})F_X(0) = 0$
$u^{(1)} = l^{(0)} + (u^{(0)} - l^{(0)})F_X(1) = 0.8$
Output:

$[l^{(1)}, u^{(1)}) \not\subset [0, 0.5)$
$[l^{(1)}, u^{(1)}) \not\subset [0.5, 1)$
$\rightarrow$ get next symbol

Input: *<u>3</u>21
$l^{(2)} = 0.656$, $u^{(2)} = 0.8$
$[l^{(2)}, u^{(2)}) \subset [0.5, 1) \rightarrow$ Output: <u>1</u>

E_2 rescale:
$l^{(2)} = 2\times(0.656 - 0.5) = 0.312$
$u^{(2)} = 2\times(0.8 - 0.5) = 0.6$
Output: 1

Tag Generation with Scaling (2/3)

Input: **$\underline{2}$1

$l^{(3)} = l^{(2)} + (u^{(2)} - l^{(2)})F_X(1) = 0.5424$

$u^{(3)} = l^{(2)} + (u^{(2)} - l^{(2)})F_X(2) = 0.54816$

$[l^{(3)}, u^{(3)}) \subset [0.5, 1) \rightarrow$ Output: 1$\underline{1}$

E_2 rescale:

$l^{(3)} = 2\times(0.5424 - 0.5) = 0.0848$

$u^{(3)} = 2\times(0.54816 - 0.5) = 0.09632$

$[l^{(3)}, u^{(3)}) \subset [0, 0.5) \rightarrow$ Output: 11$\underline{0}$

E_1 rescale:

$l^{(3)} = 2\times0.0848 = 0.1696$

$u^{(3)} = 2\times0.09632 = 0.19264$

$[l^{(3)}, u^{(3)}) \subset [0, 0.5) \rightarrow$ Output: 110$\underline{0}$

E_1 rescale:

$l^{(3)} = 2\times0.1696 = 0.3392$

$u^{(3)} = 2\times0.19264 = 0.38528$

$[l^{(3)}, u^{(3)}) \subset [0, 0.5) \rightarrow$ Output: 1100$\underline{0}$

E_1 rescale:

$l^{(3)} = 2\times0.3392 = 0.6784$

$u^{(3)} = 2\times0.38528 = 0.77056$

$[l^{(3)}, u^{(3)}) \subset [0.5, 1) \rightarrow$ Output: 11000$\underline{1}$

E_2 rescale:

$l^{(3)} = 2\times(0.6784 - 0.5) = 0.3568$

$u^{(3)} = 2\times(0.77056 - 0.5) = 0.54112$

Output: 110001

Tag Generation with Scaling (3/3)

- The final symbol '1' in the input sequence results in:

Input: ***$\underline{1}$
$l^{(4)} = l^{(3)} + (u^{(3)} - l^{(3)})F_X(0) = 0.3568$
$u^{(4)} = l^{(3)} + (u^{(3)} - l^{(3)})F_X(1) = 0.504256$
Output: 110001

- End-of-sequence symbol can be a pre-defined value in $[l^{(n)}, u^{(n)})$. If we pick 0.5_{10} as EOS†, the final output of the sequence is $11000\underline{110\ldots0}$.
- Note that $0.110001 = 2^{-1} + 2^{-2} + 2^{-6}$

$$= 0.765625.$$

† The number of bits for the EOS symbol shall be the same as the decoder word-length.

Tag Decoding Example (1/2)

- Assume word length is set to 6, the input sequence is 110001100000.

Input tag: 110001100000
Output: 1

$t^* = (0.765625 - 0)/(0.8 - 0) = 0.9579$
$F_X(2) = 0.82 \le t^* \le 1 = F_X(3)$
Output: 13
$l^{(2)} = 0 + (0.8 - 0)\times F_X(2) = 0.656,$
$u^{(2)} = 0 + (0.8 - 0)\times F_X(3) = 0.8$

E_2 rescale:
$l^{(2)} = 2\times(0.656 - 0.5) = 0.312$
$u^{(2)} = 2\times(0.8 - 0.5) = 0.6$
Update tag: *10001100000

Input tag: *10001100000
$t^* = (0.546875 - 0.312)/(0.6 - 0.312) = 0.8155$
$F_X(1) = 0.8 \le t^* \le 0.82 = F_X(2)$
Output: 132
$l^{(3)} = 0.5424,\ u^{(3)} = 0.54816$

E_2 rescale:
$l^{(3)} = 2\times(0.5424 - 0.5) = 0.0848$
$u^{(3)} = 2\times(0.54816 - 0.5) = 0.09632$
Update tag: **0001100000

Tag Decoding Example (2/2)

E_1 rescale:
$l^{(3)} = 2\times0.0848 = 0.1696$
$u^{(3)} = 2\times0.09632 = 0.19264$
Update tag: ***$\underline{001}\underline{100000}$

E_1 rescale:
$l^{(3)} = 2\times0.1696 = 0.3392$
$u^{(3)} = 2\times0.19264 = 0.38528$
Update tag: ****$\underline{01}\underline{100000}$

E_1 rescale:
$l^{(3)} = 2\times0.3392 = 0.6784$
$u^{(3)} = 2\times0.38528 = 0.77056$
Update tag: *****$\underline{1}\underline{100000}$

E_2 rescale:
$l^{(3)} = 2\times(0.6784 - 0.5) = 0.3568$
$u^{(3)} = 2\times(0.77056 - 0.5) = 0.54112$
Update tag: ******$\underline{100000}$

Now, since the final pattern 100000 is the EOS symbol, we do not have anymore input bits.

The final digit is 1 because the final interval is in
$F_X(0) = 0 \leq l^{(3)} \leq \mathrm{u}^{(3)} \leq 0.8 = F_X(1)$
Output: 132$\underline{1}$

Rescaling in Case 3

- If the limits of the interval contains 0.5, i.e., $l^{(n)} \in [0.25, 0.5)$, $u^{(n)} \in [0.5, 0.75)$, we can perform rescaling by E_3: $[0.25, 0.75) \to [0, 1)$; $E_3(x) = 2(x - 0.25)$.

- If we decide to perform E_3 rescaling, what output do we produce for an E_3 rescale operation?
  - Recall that, for E_1, 0 is sent, and for E_2, 1 is sent
  - For E_3, it depends on the non-E_3 rescale operation after it. That is, we can keep count of consecutive E_3 rescales and issue the same number of zeros/ones after the first encounter of E_2/E_1 rescale operation.
    For example, $E_3E_3E_3E_2 \to 1\underbrace{000}$.

Only used to properly rescale the intervals at the decoder!

Integer Implementation

- Assume that the interval limits are represented using integer word length of n, thus

$$[0.0,\ 1.0) \to [\overbrace{00\ldots0}^{n\text{ times}},\ \overbrace{11\ldots1}^{n\text{ times}}), \text{ and } 0.5 \to 1\overbrace{0\ldots0}^{n-1\text{ times}}.$$

- Furthermore, if symbol j occurs n_j times in a total of n_{total} symbols, then the CDF can be estimated by $F_X(k) = CC(k) / n_{total}$, where $CC(k)$ is the cumulative count defined by $$CC(k) = \sum_{i=1}^{k} n_i.$$

  Thus, interval limits are:

$$l^{(n)} = l^{(n-1)} + \left\lfloor (u^{(n-1)} - l^{(n-1)} + 1) \times CC(x_n - 1) / n_{total} \right\rfloor$$
$$u^{(n)} = l^{(n-1)} + \left\lfloor (u^{(n-1)} - l^{(n-1)} + 1) \times CC(x_n) / n_{total} \right\rfloor - 1.$$

Encoder (Integer Implementation)

Initialize l and u.
Get symbol.

$$l \leftarrow l + \left\lfloor \frac{(u - l + 1) \times Cum\_Count(x-1)}{Total\_Count} \right\rfloor$$
$$u \leftarrow l + \left\lfloor \frac{(u - l + 1) \times Cum\_Count(x)}{Total\_Count} \right\rfloor - 1$$

```
while (MSB of u and l are both equal to b or E3 condition holds)
if (MSB of u and l are both equal to b)
{
      send b
      shift l to the left by 1 bit and shift 0 into LSB
      shift u to the left by 1 bit and shift 1 into LSB
      while(Scale3 > 0)
      {
            send complement of b
            decrement Scale3
      }
}

if (E3 condition holds)
{
      shift l to the left by 1 bit and shift 0 into LSB
      shift u to the left by 1 bit and shift 1 into LSB
      complement (new) MSB of l and u
      increment Scale3
}
```

number of digits for E_3 scaling operations

Decoder (Integer Implementation)

Initialize l and u.
Read the first m bits of the received bitstream into tag t.
$k = 0$

while $\left(\left\lfloor \dfrac{(t - l + 1) \times Total\_Count - 1}{u - l + 1} \right\rfloor \geqslant Cum\_Count(k) \right)$

$k \leftarrow k + 1$
Decode symbol x.

$$l \leftarrow l + \left\lfloor \frac{(u - l + 1) \times Cum\_Count(x - 1)}{Total\_Count} \right\rfloor$$

$$u \leftarrow l + \left\lfloor \frac{(u - l + 1) \times Cum\_Count(x)}{Total\_Count} \right\rfloor - 1$$

while (MSB of u and l are both equal to b or E_3 condition holds)
if (MSB of u and l are both equal to b)
{
    shift l to the left by 1 bit and shift 0 into LSB
    shift u to the left by 1 bit and shift 1 into LSB
    shift t to the left by 1 bit and read next bit from received bitstream into LSB
}

if (E_3 condition holds)
{
    shift l to the left by 1 bit and shift 0 into LSB
    shift u to the left by 1 bit and shift 1 into LSB
    shift t to the left by 1 bit and read next bit from received bitstream into LSB
    complement (new) MSB of l, u, and t
}

Binary Arithmetic Coders

- Most arithmetic coders used today are binary coders, i.e., the alphabet = $\{0, 1\}$
- For non-binary data sources, you must apply a "binarization" process to turn the messages into binary messages before coding
- Because there are only two letters in the alphabet, the probability model consists of a single number.
  - Easier to adopt context-sensitive probability models
  - Easier to adopt "quantized" probabilities for simplification of calculations

Arithmetic vs. Huffman Coding

- Average code length of m symbol sequence:
  - Arithmetic code: $H(X) \le l_A < H(X) + 2/m$
  - Extended Huffman code: $H(X) \le l_H < H(X) + 1/m$
- Both codes have same asymptotic behavior
- Extended Huffman coding requires large codebook for m^n extended symbols while AC does not
- In general,
  - Small alphabet sets favor Huffman coding
  - Skewed distributions favor arithmetic coding
- Arithmetic coding can adapt to input statistics easily

Adaptive Arithmetic Coding

- In arithmetic coding, since coding of each new incoming symbol is based on a probability table, we can update the table easily as long as the transmitter and receiver stays in sync
- Adaptive arithmetic coding:
  - Initially, all symbols are assigned a fixed initial probability (e.g. occurrence count is set to 1)
  - After a symbol is encoded, update symbol probability (i.e. occurrence count) in both transmitter and receiver
  - Note that the occurrence count may overflow, we have to rescale the count before this happens. For example:

$$c = \lceil c/2 \rceil.$$

Applications: Image Compression

- Compression of pixel values directly

Image Name	Bits/Pixel	Total Size (bytes)	Compression Ratio (arithmetic)	Compression Ratio (Huffman)
Sena	6.52	53,431	1.23	1.16
Sensin	7.12	58,306	1.12	1.27
Earth	4.67	38,248	1.71	1.67
Omaha	6.84	56,061	1.17	1.14

- Compression of pixel differences

Image Name	Bits/Pixel	Total Size (bytes)	Compression Ratio (arithmetic)	Compression Ratio (Huffman)
Sena	3.89	31,847	2.06	2.08
Sensin	4.56	37,387	1.75	1.73
Earth	3.92	32,137	2.04	2.04
Omaha	6.27	51,393	1.28	1.26