Arithmetic Coding

C.M. Liu
Perceptual Signal Processing Lab
College of Computer Science
National Chiao-Tung University

http://www.csie.nctu.edu.tw/~cmliu/Courses/Compression/

Office: EC538
(03)5731877
cmliu@cs.nctu.edu.tw

Introduction

- **Arithmetic coding**
  - Lossless data compression
  - Variable-length entropy coding.
  - Encodes the entire message into a single number, a fraction n where $(0.0 \leq n < 1.0)$.
    - As opposed to other entropy encoding techniques that separate the input message into its component symbols and replace each symbol with a code word,

Flashback: Extended Huffman Codes

- Consider the source:
  - $\mathcal{A} = \{a, b, c\}$, $P(a) = 0.8$, $P(b) = 0.02$, $P(c) = 0.18$
  - $H = 0.816$ bits/symbol
- Huffman code:

a	0
b	11
c	10

  - $l = 1.2$ bits/symbol
  - Redundancy = 0.384 b/sym (47%!)
- Q: Could we do better?

Flashback: Extended Huffman Codes (2)

- Idea
  - Consider encoding ***sequences of two letters*** as opposed to single letters

Letter	Probability	Code
aa	0.6400	0
ab	0.0160	10101
ac	0.1440	11
ba	0.0160	101000
bb	0.0004	10100101
bc	0.0036	1010011
ca	0.1440	100
cb	0.0036	10100100
cc	0.0324	1011

$l = 1.7228/2 = \mathbf{0.8614}$

Red. = **0.0045** bits/symbol

PSP LAB

Flashback: Extended Huffman Codes (3)

- The idea can be extended further
  - Consider all possible ***m^n*** sequences of length ***n*** (we did 3^2)
- In theory:
  - By considering more sequences we can improve the coding
- In reality:
  - The exponential growth of the alphabet makes this impractical
    - E.g., for length 3 ASCII seq.: $256^3 = 2^{24} = 16M$
    - Need to generate codes for ***all*** sequences of length ***m***
    - Most sequences would have zero frequency.
    - Decoder would be inefficient for *memory*, *speed*, and *probability perturbation.*
  - E.g.:
    - $\mathcal{A} = \{a, b, c\}$, P(a) = 0.95, P(b) = 0.02, P(c) = 0.03
    - ***H*** = 0.335 bits/symbol
    - $l_1 = 1.05$, $l_2 = 0.611$, …
    - Performance becomes acceptable at length $n = 8$.
    - But **|alphabet| = 3^8 = 6561**.

PSP LAB

Arithmetic Coding

- Brief history
  - Shannon mentioned the use of ***cdf***
  - Peter Elias (classmate of Huffman)— recursive algorithm
    - Not published until 1963
  - Jelinek 1968
  - Modern roots: Pasco/Rissanen 1976
- Basic idea
  - Generate a unique tag (code) for an entire sequence
  - Without generating codes for all the other possible sequences (as Huffman)
  - Tag is a number in [0,1)

Probability Notation Refresher

- Random variable:
  - A mapping between (sets of) outcomes of an experiment to real numbers.
- To replace symbols with numbers we use
  - $X(a_i) = i$, where $a_i \in \mathcal{A}$ ($\mathcal{A} = \{a_i\}$, $i = 1..n$)
- Given a probability model $\mathcal{P}$ for the source
  - Probability density function (***pdf***)

$$\boxed{P(X = i) = P(a_i)}$$

  - Cumulative density function (***cdf***)

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

PSP LAB

Generating a Tag

- Divide [0, 1) into *m* intervals:

$$[F_X(i-1), F_X(i)], i = 1..m, \quad F_X(0) = 0$$

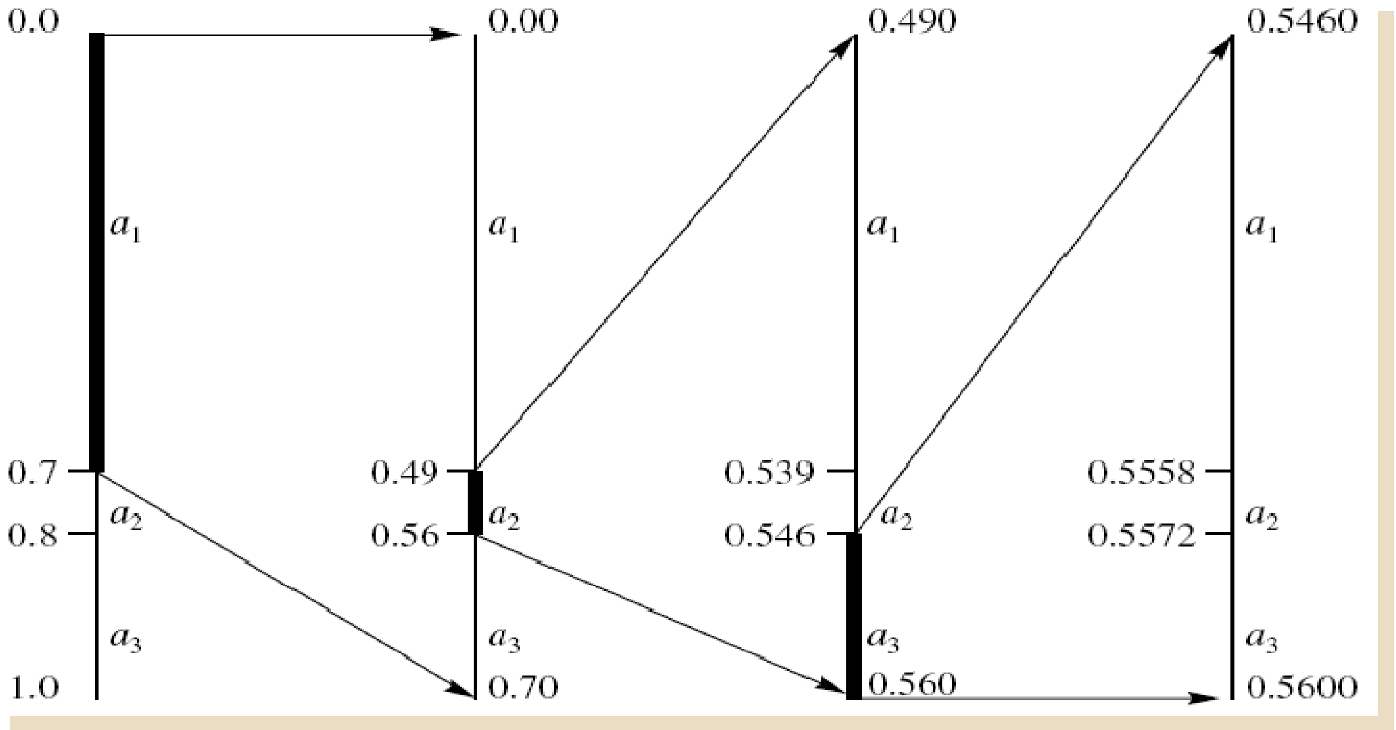
- We have a one-to-one mapping:
  - $a_k \Leftrightarrow$ **$[F_X(k\text{-}1), F_X(k)]$**, $k = 1..n$, $F_X(0) = 0$
  - ***Any*** real number in $[F_X(k\text{-}1), F_X(k)]$ can represent a_k
- Encoding a 2-letter sequence: $a_k\, a_j$
  - Pick $[F_X(k\text{-}1), F_X(k)]$ for a_k
  - Then split the interval into the same proportions and pick the j^{th} interval:

$$\left[F_X(k-1)\frac{F_X(j-1)}{F_X(k)-F_X(k-1)}, F_X(k-1)\frac{F_X(j)}{F_X(k)-F_X(k-1)}\right]$$

PSP LAB

Tag Generation Example

- Consider encoding $a_1a_2a_3$:
  - $\mathcal{A} = \{a_1, a_2, a_3\}$, $\mathcal{P} = \{0.7, 0.1, 0.2)$
  - Mapping: $a_1 \Leftrightarrow 1$, $a_2 \Leftrightarrow 2$, $a_3 \Leftrightarrow 3$
  - ***cdf***: $F_X(1) = 0.7$, $F_X(2) = 0.8$, $F_X(3) = 1.0$



Mapping to Real Numbers

- $\mathcal{A} = \{a_1, a_2, \ldots, a_n\}$

$$\bar{T}_X(a_i) = \sum_{k=1}^{i-1} P(X=k) + \frac{1}{2}P(X=i) = F_X(i-1) + \frac{1}{2}P(X=i)$$

- Fair dice-throwing example: {1, 2, 3, 4, 5, 6}

$$P(X=k) = \frac{1}{6} \quad for\ k = 1..6$$

$$\bar{T}_X(2) = P(X=1) + \frac{1}{2}P(X=2) = 0.25$$

$$\bar{T}_X(5) = \sum_{k=1}^{4} P(X=k) + \frac{1}{2}P(X=5) = 0.75$$

Lexicographic Order

Outcome	Tag
1	$0.08\overline{33}$
3	$0.41\overline{66}$
4	$0.58\overline{33}$
6	$0.91\overline{66}$

- Lexicographic (dictionary) order of strings:

$$\bar{T}_X^{(m)}(x_i) = \sum_{\forall y : y \prec x_i} P(y) + \frac{1}{2} P(x_i)$$

  - where $y \prec x$ means 'y precedes *x* in alphabet ordering'
  - ***m*** is the length of the sequence

Lexicographic Order Example

- Consider ***two*** consecutive rolls of the die:
  - Outcomes = {11, 12, …, 16, 21, 22, …, 26, …, 61, 62, …, 66}

$$\bar{T}_X(13) = P(x=11) + P(x=12) + \frac{1}{2}P(x=13) = \frac{5}{72} = 0.69\bar{4}$$

Notes

To generate tag for 13, we did not have to generate any other tags

Problem: to generate a tag for a string of sequence, we need to know all the probabilities for sequences "less than" the sequence.

- Target: Try to have use only the probability of individual symbol.

PSP LAB

Interval Construction

- Observation
  - An interval containing a tag is disjoint from all other tag intervals
- Idea
  - Express lower/upper bounds for a sequence as a function of bounds for shorter sequences
- Recall fair-die example
  - Consider the sequence 3 2 2
  - Let $\mathbf{u}^{(m)}$, $\mathbf{l}^{(m)}$ be the upper/lower bound of length ***m***. Then,

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

    $\mathbf{u}^{(2)} = F_X^{(2)}(32)$, $\mathbf{l}^{(2)} = F_X^{(2)}(31)$

$$F_X^{(2)}(32) = P(x=11) + \ldots + P(x=16) + P(x=21) + \ldots + P(x=26) + P(x=31) + P(x=32)$$

PSP LAB

Interval Construction (2)

$$F_X^{(2)}(32) = [P(x=11)+\ldots+P(x=16)] + [P(x=21)+\ldots+P(x=26)] + P(x=31)+P(x=32)$$

$$\sum_{i=1}^{6} P(x=ki) = \sum_{i=1}^{6} P(x_1=k, x_2=i) = P(x_1=k)\sum_{i=1}^{6} P(x_2=i) = P(x_1=k), \quad where\, x = x_1x_2$$

$$F_X^{(2)}(32) = P(x=1)+P(x=2)+P(x=31)+P(x=32) = F_X(2)+P(x=31)+P(x=32)$$

$$P(x=31)+P(x=32) = P(x_1=3)\big(P(x_2=1)+P(x_2=2)\big) = P(x_1=3)F_X(2)$$

$$P(x_1=3) = F_X(3) - F_X(2)$$

$$F_X^{(2)}(32) = F_X(2) + \big(F_X(3)-F_X(2)\big)F_X(2)$$

$$u^{(2)} = l^{(1)} + \big(u^{(1)} - l^{(1)}\big)F_X(2)$$

PSP LAB

Interval Construction (3)

$$F_X^{(2)}(32) = F_X(2) + \left(F_X(3) - F_X(2)\right)F_X(2)$$

$$u^{(2)} = l^{(1)} + \left(u^{(1)} - l^{(1)}\right)F_X(2)$$

$$F_X^{(2)}(31) = F_X(2) + \left(F_X(3) - F_X(2)\right)F_X(1)$$

$$l^{(2)} = l^{(1)} + \left(u^{(1)} - l^{(1)}\right)F_X(1)$$

$$u^{(3)} = F_X^{(2)}(322), \quad l^{(3)} = F_X^{(2)}(321)$$

$$F_X^{(3)}(322) = F_X(31) + \left(F_X(32) - F_X(31)\right)F_X(2)$$

$$F_X^{(3)}(321) = F_X(31) + \left(F_X(32) - F_X(31)\right)F_X(1)$$

$$u^{(3)} = l^{(2)} + \left(u^{(2)} - l^{(2)}\right)F_X(2)$$

$$l^{(3)} = l^{(2)} + \left(u^{(2)} - l^{(2)}\right)F_X(1)$$

Generating a Tag

- In general, for any sequence $\mathbf{x} = x_1 x_2 \ldots x_n$

$$u^{(k)} = l^{(k-1)} + \left(u^{(k-1)} - l^{(k-1)}\right) F_X(x_k)$$
$$l^{(k)} = l^{(k-1)} + \left(u^{(k-1)} - l^{(k-1)}\right) F_X(x_k - 1)$$

$$\bar{T}_X(x) = \frac{u^{(n)} + l^{(n)}}{2}$$

Tag Generation Example

- Consider random variable $X(a_i) = i$
  - Encode sequence 1 3 2 1, given the following

$$F_X(k)=0, k \le 0, \quad F_X(1)=0.8, \quad F_X(2)=0.82, \quad F_X(3)=1, \quad F_X(k)=1, k>3$$

$$l^{(0)}=0, \quad u^{(0)}=1$$

1
$$l^{(1)}=l^{(0)}+\left(u^{(0)}-l^{(0)}\right)F_X(0)=0$$
$$u^{(1)}=l^{(0)}+\left(u^{(0)}-l^{(0)}\right)F_X(1)=0.8$$

13
$$l^{(2)}=l^{(1)}+\left(u^{(1)}-l^{(1)}\right)F_X(2)=0.656$$
$$u^{(2)}=l^{(1)}+\left(u^{(1)}-l^{(1)}\right)F_X(3)=0.8$$

132
$$l^{(3)}=l^{(2)}+\left(u^{(2)}-l^{(2)}\right)F_X(1)=0.7712$$
$$u^{(3)}=l^{(2)}+\left(u^{(2)}-l^{(2)}\right)F_X(2)=0.77408$$

1321
$$l^{(4)}=l^{(1)}+\left(u^{(3)}-l^{(3)}\right)F_X(2)=0.7712$$
$$u^{(4)}=l^{(3)}+\left(u^{(3)}-l^{(3)}\right)F_X(1)=0.773504$$

$$\boxed{\bar{T}_X(1321)=\frac{u^{(4)}+l^{(4)}}{2}=0.772352}$$

PSP LAB

Decoding a Tag

- Algorithm
  - Initialize $l^{(0)} = 0$, $u^{(0)} = 1$.

1. For each i, $i = 1..n$
   - Compute t^{*}=(tag-$l^{(k-1)}$)/(u$^{(k-1)}$-$l^{(k-1)}$).
2. Find the x_k: $F_X(x_k\text{-}1) \leq t^{*} \leq F_X(x_k)$.
3. Update $u^{(n)}$, $l^{(n)}$
4. If done--exit, otherwise goto 1.

Decoding Example

- Algorithm
  - Initialize $l^{(0)} = 0$, $u^{(0)} = 1$.
  1. Compute $t^*=(\text{tag}-l^{(k-1)})/(u^{(k-1)}-l^{(k-1)})$.
  2. Find the x_k: $F_X(x_k-1) \le t^* \le F_X(x_k)$.
  3. Update $u^{(k)}$, $l^{(k)}$
  4. If done--exit, otherwise goto 1.

$$\bar{T}_X(1321) = 0.772352$$

$$F_X(k) = 0, k \le 0, \quad F_X(1) = 0.8,$$
$$F_X(2) = 0.82, \quad F_X(3) = 1, \quad F_X(k) = 1, k > 3$$

$$u^{(k)} = l^{(k-1)} + \left(u^{(k-1)} - l^{(k-1)}\right)F_X(x_k)$$
$$l^{(k)} = l^{(k-1)} + \left(u^{(k-1)} - l^{(k-1)}\right)F_X(x_k - 1)$$

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

$\Rightarrow$ $\underline{1}$

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

$\Rightarrow$ 1$\underline{3}$

$$t^* = \frac{0.772352 - 0.656}{0.8 - 0.656} = 0.808$$
$$F_X(1) = 0.8 \le t^* \le 0.82 = F_X(2)$$
$$l^{(3)} = l^{(2)} + \left(u^{(2)} - l^{(2)}\right)F_X(1) = 0.7712$$
$$u^{(3)} = l^{(2)} + \left(u^{(2)} - l^{(2)}\right)F_X(2) = 0.77408$$

$\Rightarrow$ 13$\underline{2}$

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

$\Rightarrow$ 132$\underline{1}$

Implementation

- So far
  - We have encoding/decoding algorithms that work
  - They assume real numbers (infinite precision)
    - Eventually $l^{(n)}$ and $u^{(n)}$ will converge
  - We want to be able to encode a string incrementally
- Observation: as interval narrows, either
  1. $[l^{(n)}, u^{(n)}] \subset [0, 0.5)$, or
  2. $[l^{(n)}, u^{(n)}] \subset [0.5, 1)$, or
  3. $l^{(n)} \in [0, 0.5)$, $u^{(n)} \in [0.5, 1)$.
  - Our plan: focus on 1. & 2. now, deal with 3. later

Implementation (2)

- Principle
  - Scale and shift simultaneously x, upper bound, and lower bound will bring the same relative location.

$$u^{(k)} = l^{(k-1)} + \left(u^{(k-1)} - l^{(k-1)}\right)F_X(x_k)$$
$$l^{(k)} = l^{(k-1)} + \left(u^{(k-1)} - l^{(k-1)}\right)F_X(x_k - 1)$$

- Encoder
  - Once we reach 1. or 2., we can ignore the other half of [0,1)
  - We can also indicate to decoder which half the tag is confined to:
    - Send 0/1 bit to indicate lower/upper
  - Rescale tag interval to [0, 1):
    - E_1: [0, 0.5) => [0, 1); $E_1(x) = 2x$
    - E_2: [0.5, 1) => [0, 1); $E_2(x) = 2(x-0.5)$
  - Note: we lost the most significant bit during rescaling
  - Not a problem--we already sent it out the door
- Decoder
  - Follow the 0/1 bits and rescale accordingly
    - Stays in sync with encoder

Tag Generation Example w/ Scaling

- Consider random variable $X(a_i) = i$
  - Encode sequence 1 3 2 1, given the following

$$F_X(k)=0, k\le 0,\quad F_X(1)=0.8,\quad F_X(2)=0.82,\quad F_X(3)=1,\quad F_X(k)=1, k>3$$

$$l^{(0)}=0,\quad u^{(0)}=1$$

Input: $\underline{1}321$

$$l^{(1)}=l^{(0)}+\left(u^{(0)}-l^{(0)}\right)F_X(0)=0$$
$$u^{(1)}=l^{(0)}+\left(u^{(0)}-l^{(0)}\right)F_X(1)=0.8$$

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

Output:

Input: $-\underline{3}21$

$$l^{(2)}=l^{(1)}+\left(u^{(1)}-l^{(1)}\right)F_X(2)=0.656$$
$$u^{(2)}=l^{(1)}+\left(u^{(1)}-l^{(1)}\right)F_X(3)=0.8$$

$$[0.656,0.8]\subset[0.5,1)$$
$$l^{(2)}=2\times(0.656-0.5)=0.312$$
$$u^{(2)}=2\times(0.8-0.5)=0.6$$

Output: $\underline{1}$

PSP LAB

Tag Generation Example w/ Scaling (2)

$$l^{(2)} = 0.312, \quad u^{(2)} = 0.6$$

Input： $--\underline{2}1$

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

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

Output： $1\underline{1}$

Input： $---1$

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

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

Output： $11\underline{0}$

Input： $---1$

$$l^{(3)} = 2 \times 0.0848 = 0.1696$$

$$u^{(3)} = 2 \times 0.09632 = 0.19264$$

Output： $110\underline{0}$

Input： $---1$

$$l^{(3)} = 2 \times 0.1696 = 0.3392$$

$$u^{(3)} = 2 \times 0.19264 = 0.38528$$

Output： $1100\underline{0}$

Tag Generation Example w/ Scaling (3)

Input： ---1

$$l^{(3)} = 2 \times 0.3392 = 0.6784$$

$$u^{(3)} = 2 \times 0.38528 = 0.77056$$

Output： 11000$\underline{1}$

Input： ---1

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

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

Output： 110001

Input： ---$\underline{1}$

$$l^{(4)} = 0.3568 + (0.54112 - 0.3568) F_X(0) = 0.3568$$

$$u^{(4)} = 0.3568 + (0.54112 - 0.3568) F_X(1) = 0.504256$$

Output： 110001

- EOT:
  - Any (convenient) number in $[l^{(n)}, u^{(n)}]$
  - We pick $0.5_{10} = 0.1_2$

Output： 11000$\underline{1}$10⋯

❖ Note: $0.1100011 = 2^{-1}+2^{-2}+2^{-6}+2^{-7} = 0.7734375 \in [0.7712, 0.77408]$

PSP LAB

Incremental Tag Decoding

- We want incremental decoding
  - E.g. network transmissions
- Issues
  - How to start?
  - How to continue?
  - How to end?
- Continuation:
  - Once we have unambiguous start, just mimic encoder

Tag Decoding Example

- Assume word length is set to 6
- Input： $\underline{110001}100000$
  - $0.110001_2 = 0.765625_{10}$
  - First bit is 1

Output： $\underline{1}$

Input： $\underline{110001}100000$

$$t^* = (0.765625 - 0)/(0.8 - 0) = 0.9570$$
$$F_X(2) = 0.82 \le t^* \le 1 = F_X(3)$$

Output： $1\underline{3}$

$$l^{(2)} = l^{(1)} + \left(u^{(1)} - l^{(1)}\right)F_X(2) = 0.656$$
$$u^{(2)} = l^{(1)} + \left(u^{(1)} - l^{(1)}\right)F_X(3) = 0.8$$

Input：$-\underline{100011}00000$ (shift left)

$$l^{(2)} = 2 \times (0.656 - 0.5) = 0.312$$
$$u^{(2)} = 2 \times (0.8 - 0.5) = 0.6$$

Input： $-\underline{100011}00000$ (**0.546875**)

$$t^* = (0.546875 - 0.312)/(0.6 - 0.312) = 0.8155$$
$$F_X(1) = 0.8 \le t^* \le 0.82 = F_X(2)$$

Output： $13\underline{2}$

$$l^{(3)} = l^{(2)} + \left(u^{(2)} - l^{(2)}\right)F_X(1) = 0.5424$$
$$u^{(3)} = l^{(2)} + \left(u^{(2)} - l^{(2)}\right)F_X(2) = 0.54816$$

Input： $--\underline{000110}0000$ (shift left)

$$l^{(3)} = 2 \times (0.5424 - 0.5) = 0.0848$$
$$u^{(3)} = 2 \times (0.54816 - 0.5) = 0.09632$$

Input： $---\underline{001100}000$ (shift left)

PSP LAB

Tag Decoding Example (2)

$$l^{(3)} = 2 \times 0.0848 = 0.1696$$
$$u^{(3)} = 2 \times 0.09632 = 0.19264$$

Input： ----$\underline{011000}$00 (shift left)

$$l^{(3)} = 2 \times 0.1696 = 0.3392$$
$$u^{(3)} = 2 \times 0.19264 = 0.38528$$

Input： -----$\underline{110000}$0 (shift left)

$$l^{(3)} = 2 \times 0.3392 = 0.6784$$
$$u^{(3)} = 2 \times 0.38528 = 0.77056$$

Input： -----$\underline{100000}$ (shift left)

Input： -----$\underline{100000}$

$$l^{(3)} = 2 \times (0.6784 - 0.5) = 0.3568$$
$$u^{(3)} = 2 \times (0.77056 - 0.5) = 0.54112$$

$$t^* = (0.5 - 0.3568)/(0.54112 - 0.3568) = 0.7769$$
$$F_X(0) = 0 \le t^* \le 0.8 = F_X(1)$$

Output： 132$\underline{1}$

Q.E.D.

Managing E_3

- Recall the three cases
  1. $[l^{(n)}, u^{(n)}] \subset [0, 0.5)$: E_1: $[0, 0.5) \Rightarrow [0, 1)$; $E_1(x) = 2x$
  2. $[l^{(n)}, u^{(n)}] \subset [0.5, 1)$: E_2: $[0.5, 1) \Rightarrow [0, 1)$; $E_2(x) = 2(x-0.5)$
  3. **$l^{(n)} \in [0, 0.5)$, $u^{(n)} \in [0.5, 1)$:** $E_3(x) = ???$
- E_3 rescaling
  - E_3: $[0.25, 0.75) \Rightarrow [0, 1)$; $E_3(x) = 2(x-0.25)$
- Encoding
  - $E_1 = 0$, $E_2 = 1$, $E_3 = ???$
  - Note that:
    - $E_3 \ldots E_3 E_1 = E_1 E_2 \ldots E_2$
    - $E_3 \ldots E_3 E_2 = E_2 E_1 \ldots E_1$
  - **Rule:** keep count of consecutive E_3 and issue that number of zeroes/ones after the next encounter of E_2/E_1.

PSP LAB

Integer Implementation

- Assume word length of ***m***. Then

$$[0,1) \rightarrow \overbrace{00\ldots0}^{m\text{ times}}\ldots\overbrace{11\ldots1}^{m\text{ times}}$$

$$0.5 = 1\overbrace{0\ldots0}^{m-1\text{ times}}$$

- n_j = frequency of ***j*** in sequence of length **TotalCount**
- Cumulative Count **CC**

$$CC(k) = \sum_{i=1}^{k} n_i$$

$$F_X(k) = CC(k)/TC$$

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

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

PSP LAB

Integer Implementation (2)

- MSB(x) = Most Significant Bit of *x*
- LSB(*x*) = Least Significant Bit of *x*
- SB(x, i) = i^{th} Significant Bit of *x*
  - MSB(*x*) = SB(*x*, 1); LSB(*x*) = SB(*x*, *m*)
- E3(*l*, *u*) = (SB(*l*, 2) == 1 && SB(*u*, 2) == 0)

PSP LAB

Integer Encoder


```
l=00…0, u=11…1, e3_count=0
repeat
  x=get_symbol
  l=l+ ⌊(u-l+1)×CC(x-1)/TC⌋         // lower bound update
  u=l+ ⌊(u-l+1)×CC(x)/TC⌋-1        // upper bound update
  while(MSB(u)==MSB(l) OR E3(u,l)) // MSB(u)=MSB(l)=0 ➔ E1 rescaling
    if(MSB(u)==MSB(l))             // MSB(u)=MSB(l)=1 ➔ E2 rescaling
      send(MSB(u))
      l = (l<<1)+0                 // shift left, set LSB to 0
      u = (u<<1)+1                 // shift left, set LSB to 1
      while(e3_count>0)
        send(!MSB(u))              // encode accumulated E3 rescalings
        e3_count--
      endwhile
    endif
    if(E3(u,l))                    // perform E3 rescaling & remember
      l = (l<<1)+0
      u = (u<<1)+1
      complement MSB(u) and MSB(l)
      e3_count++
    endif
  endwhile
until done
```

Integer Encoding Example

- Sequence 1 3 2 1
- Count {1, 2, 3} = {40, 1, 9}
- Total count TC = 50
- Cumulative count
  - CC {0, 1, 2, 3} = {0, 40, 41, 50}
- Recall that interval endpoint should never overlap
  - m = ?
  - smallest [l(n),u(n)] = 1/4 of entire range 0..TC;

  => to maintain unique representation we need range of at least 4x50 = 200

  => minimum ***m*** = 8 ($2^8 = 256$)

```
l=00…0, u=11…1, e3_count=0
repeat
  x=get_symbol
  l=l+ ⌊(u-l+1)×CC(x-1)/TC⌋
  u=l+ ⌊(u-l+1)×CC(x)/TC⌋-1
  while(MSB(u)==MSB(l) OR E3(u,l))
    if(MSB(u)==MSB(l))
      send(MSB(u))
      l = (l<<1)+0
      u = (u<<1)+1
      while(e3_count>0)
        send(!MSB(u))
        e3_count--
      endwhile
    endif
    if(E3(u,l))
      l = (l<<1)+0
      u = (u<<1)+1
      complement MSB(u) and MSB(l)
      e3_count++
    endif
  endwhile
until done
```

Integer Encoding Example (2)

$l^{(0)} = 0 = (00000000)_2$

$u^{(0)} = 255 = (11111111)_2$

Input： <u>1</u>321

$l^{(1)} = 0 + \lfloor 256 \times 0/50 \rfloor = 0 = (00000000)_2$

$u^{(1)} = 0 + \lfloor 256 \times 40/50 \rfloor - 1 = 203 = (11001011)_2$

$MSB(l) \neq MSB(u),\ E_3 = \texttt{false}$

Output：

Input： -<u>3</u>21

$l^{(2)} = 0 + \lfloor 204 \times 41/50 \rfloor = 167 = (10100111)_2$

$u^{(2)} = 0 + \lfloor 204 \times 50/50 \rfloor - 1 = 203 = (11001011)_2$

$MSB(l) = MSB(u) = 1$

Output： <u>1</u>

```
l=00…0, u=11…1, e3_count=0
repeat
  x=get_symbol
  l=l+ ⌊(u-l+1)×CC(x-1)/TC⌋
  u=l+ ⌊(u-l+1)×CC(x)/TC⌋-1
  while(MSB(u)==MSB(l) OR E3(u,l))
    if(MSB(u)==MSB(l))
      send(MSB(u))
      l = (l<<1)+0
      u = (u<<1)+1
      while(e3_count>0)
        send(!MSB(u))
        e3_count--
      endwhile
    endif
    if(E3(u,l))
      l = (l<<1)+0
      u = (u<<1)+1
      complement MSB(u) and MSB(l)
      e3_count++
    endif
  endwhile
until done
```

Integer Encoding Example (3)

$$l^{(2)} = (10101011)_2 << 1 + 0 = (01001110)_2 = 78$$

$$u^{(2)} = (11001011)_2 << 1 + 1 = (10010111)_2 = 151$$

$$E_3 = \textbf{true}$$

$$l^{(2)} = \left((01001110)_2 << 1 + 0\right) \textbf{ xor } (10000000) = 28$$

$$u^{(2)} = \left((10010111)_2 << 1 + 1\right) \textbf{ xor } (10000000)_2 = 175$$

e3_count = <u>1</u>

Input： --<u>2</u>1

$$l^{(3)} = 28 + \lfloor 148 \times 40/50 \rfloor = 146 = (10010010)_2$$

$$u^{(3)} = 28 + \lfloor 148 \times 41/50 \rfloor - 1 = 148 = (10010100)_2$$

$$MSB(l) = MSB(u) = 1, \quad \textbf{e3\_count} = 1$$

Output： 1<u>10</u>

```
l=00…0, u=11…1, e3_count=0
repeat
  x=get_symbol
  l=l+ ⌊(u-l+1)×CC(x-1)/TC⌋
  u=l+ ⌊(u-l+1)×CC(x)/TC⌋-1
  while(MSB(u)==MSB(l) OR E3(u,l))
    if(MSB(u)==MSB(l))
      send(MSB(u))
      l = (l<<1)+0
      u = (u<<1)+1
      while(e3_count>0)
        send(!MSB(u))
        e3_count--
      endwhile
    endif
    if(E3(u,l))
      l = (l<<1)+0
      u = (u<<1)+1
      complement MSB(u) and MSB(l)
      e3_count++
    endif
  endwhile
until done
```

Integer Encoding Example (4)

Input： ---1

$$l^{(3)} = (10010010)_2 << 1 = (00100100)_2 = 36$$

$$u^{(3)} = (10010100)_2 << 1 + 1 = (00101001)_2 = 41$$

$$MSB(l) = MSB(u) = 0$$

Output： 110<u>0</u>

Input： ---1

$$l^{(3)} = (00100100)_2 << 1 = (01001000)_2 = 72$$

$$u^{(3)} = (00101001)_2 << 1 + 1 = (01010011)_2 = 83$$

$$MSB(l) = MSB(u) = 0$$

Output： 1100<u>0</u>

Input： ---1

$$l^{(3)} = (01001000)_2 << 1 = (10010000)_2 = 144$$

$$u^{(3)} = (01010011)_2 << 1 + 1 = (10100111)_2 = 167$$

$$MSB(l) = MSB(u) = 1$$

Output： 11000<u>1</u>

```
l=00…0, u=11…1, e3_count=0
repeat
  x=get_symbol
  l=l+ ⌊(u-l+1)×CC(x-1)/TC⌋
  u=l+ ⌊(u-l+1)×CC(x)/TC⌋-1
  while(MSB(u)==MSB(l) OR E3(u,l))
    if(MSB(u)==MSB(l))
      send(MSB(u))
      l = (l<<1)+0
      u = (u<<1)+1
      while(e3_count>0)
        send(!MSB(u))
        e3_count--
      endwhile
    endif
    if(E3(u,l))
      l = (l<<1)+0
      u = (u<<1)+1
      complement MSB(u) and MSB(l)
      e3_count++
    endif
  endwhile
until done
```

Integer Encoding Example (5)

Input： ---1

$$l^{(3)} = (10010000)_2 << 1 = (00100000)_2 = 32$$

$$u^{(3)} = (10100111)_2 << 1 + 1 = (01001111)_2 = 79$$

$$MSB(l) = MSB(u) = 0$$

Output： 110001<u>0</u>

Input： ---1

$$l^{(3)} = (00100000)_2 << 1 = (01000000)_2 = 64$$

$$u^{(3)} = (01001111)_2 << 1 + 1 = (10011111)_2 = 159$$

$$MSB(l) \neq MSB(u), E_3 = \mathbf{true}$$

$$l^{(3)} = \left((01000000)_2 << 1 + 0\right) \mathbf{xor} \left(10000000\right) = 0$$

$$u^{(3)} = \left((10011111)_2 << 1 + 1\right) \mathbf{xor} \left(10000000\right)_2 = 191$$

e3_count = <u>1</u>

```
l=00…0, u=11…1, e3_count=0
repeat
  x=get_symbol
  l=l+ ⌊(u-l+1)×CC(x-1)/TC⌋
  u=l+ ⌊(u-l+1)×CC(x)/TC⌋-1
  while(MSB(u)==MSB(l) OR E3(u,l))
    if(MSB(u)==MSB(l))
      send(MSB(u))
      l = (l<<1)+0
      u = (u<<1)+1
      while(e3_count>0)
        send(!MSB(u))
        e3_count--
      endwhile
    endif
    if(E3(u,l))
      l = (l<<1)+0
      u = (u<<1)+1
      complement MSB(u) and MSB(l)
      e3_count++
    endif
  endwhile
until done
```

Integer Encoding Example (5)

Input： ---<u>1</u>

$$l^{(4)} = 0 + \lfloor 192 \times 0/50 \rfloor = 0 = (00000000)_2$$

$$u^{(4)} = 0 + \lfloor 192 \times 40/50 \rfloor - 1 = 152 = (10011000)_2$$

$$MSB(l) \neq MSB(u), E_3 = \texttt{false}$$

Output： 1100010

- Termination
  - Generally, send $l^{(4)}$: $(00000000)_2$
  - However e3_count = 1, so
  - we send '1' after first '0' from $l^{(4)}$:

Final output： 1100010<u>010000000</u>

```
l=00…0, u=11…1, e3_count=0
repeat
  x=get_symbol
  l=l+ ⌊(u-l+1)×CC(x-1)/TC⌋
  u=l+ ⌊(u-l+1)×CC(x)/TC⌋-1
  while(MSB(u)==MSB(l) OR E3(u,l))
    if(MSB(u)==MSB(l))
      send(MSB(u))
      l = (l<<1)+0
      u = (u<<1)+1
      while(e3_count>0)
        send(!MSB(u))
        e3_count--
      endwhile
    endif
    if(E3(u,l))
      l = (l<<1)+0
      u = (u<<1)+1
      complement MSB(u) and MSB(l)
      e3_count++
    endif
  endwhile
until done
```

Integer Decoder


```
Initialize l, u, t                         // t = first m bits
repeat
  k=0
  while( ⌊((u-l+1)×TC-1)/(u-l+1)⌋ ≥ CC(k)
    k++
  x = decode_symbol(k)
  l=l+ ⌊(u-l+1)×CC(x-1)/TC⌋
  u=l+ ⌊(u-l+1)×CC(x)/TC⌋-1
  while(MSB(u)==MSB(l) OR E3(u,l))
    if(MSB(u)==MSB(l))                     // Perform E1/E2 rescaling of l,u,t
      l = (l<<1)+0
      u = (u<<1)+1
      t = (u<<1)+next_bit
    endif
    if(E3(u,l))                            // Perform E3 rescaling of l,u,t
      l = (l<<1)+0
      u = (u<<1)+1
      t = (u<<1)+next_bit
      complement MSB(u), MSB(l), MSB(t)
    endif
  endwhile
until done
```

Integer Decoding Example

Input： 11000100 10000000

$$l = 0 = (00000000)_2$$
$$u = 255 = (11111111)_2$$
$$t = 196 = (11000100)_2$$

$$t^* = 0 + \lfloor (197 \times 50)/256 \rfloor - 1 = 38$$
$$\Rightarrow k = 1 \quad \Rightarrow x = 1$$

Output： 1

$$l = 0 + \lfloor 256 \times 0/50 \rfloor = 0 = (00000000)_2$$
$$u = 0 + \lfloor 256 \times 40/50 \rfloor - 1 = 203 = (11001011)_2$$
$$MSB(l) \neq MSB(u),\ E_3 = \texttt{false}$$

$$t^* = 0 + \lfloor (197 \times 50)/203 \rfloor - 1 = 48$$
$$\Rightarrow k = 3 \quad \Rightarrow x = 3$$

Output： 13

```
Initialize l, u, t
repeat
  k=0
  while( ⌊(u-l+1)×CC(x-1)/TC⌋ ≥ CC(k))
    k++
  x = decode_symbol(k)
  l=l+ ⌊(u-l+1)×CC(x-1)/TC⌋
  u=l+ ⌊(u-l+1)×CC(x)/TC⌋-1
  while(MSB(u)==MSB(l) OR E3(u,l))
    if(MSB(u)==MSB(l))
      l = (l<<1)+0
      u = (u<<1)+1
      t = (u<<1)+next_bit
    endif
    if(E3(u,l))
      l = (l<<1)+0
      u = (u<<1)+1
      t = (u<<1)+next_bit
      complement MSB(u), MSB(l), MSB(t)
    endif
  endwhile
until done
```

Integer Decoding Example (2)

Input： 1$\underline{10001001}$0000000

$$t = (10001001)_2$$
$$l = 0 + \lfloor 204 \times 41/50 \rfloor = 167 = (10100111)_2$$
$$u = 0 + \lfloor 204 \times 50/50 \rfloor - 1 = 203 = (11001011)_2$$
$$MSB(l) = MSB(u)$$

Input： 11$\underline{00010010}$000000

$$t = (00010010)_2$$
$$l = (10100111)_2 << 1 = (01001110)_2$$
$$u = (11001011)_2 << 1 + 1 = (10010111)_2$$
$$MSB(l) \neq MSB(u),\ E_3 = \texttt{true}$$

Input： 110$\underline{00100100}$00000

$$t = (00010010)_2\ \texttt{xor}\ (10000000) = (10010010)_2 = 146$$
$$l = ((01001110)_2 << 1)\ \texttt{xor}\ (10000000) = (00011100)_2 = 28$$
$$u = ((10010111)_2 << 1 + 1)\ \texttt{xor}\ (10000000) = (10111001)_2 = 175$$
$$MSB(l) \neq MSB(u),\ E_3 = \texttt{false}$$

```
Initialize l, u, t
repeat
  k=0
  while( ⌊(u-l+1)×CC(x-1)/TC⌋ ≥ CC(k))
    k++
  x = decode_symbol(k)
  l=l+ ⌊(u-l+1)×CC(x-1)/TC⌋
  u=l+ ⌊(u-l+1)×CC(x)/TC⌋-1
  while(MSB(u)==MSB(l) OR E3(u,l))
    if(MSB(u)==MSB(l))
      l = (l<<1)+0
      u = (u<<1)+1
      t = (u<<1)+next_bit
    endif
    if(E3(u,l))
      l = (l<<1)+0
      u = (u<<1)+1
      t = (u<<1)+next_bit
      complement MSB(u), MSB(l), MSB(t)
    endif
  endwhile
until done
```

Integer Decoding Example (3)

$$t^* = \lfloor ((146-28+1)\times 50-1)/(175-28+1) \rfloor = 40$$
$$\Rightarrow k = 2 \quad \Rightarrow x = 2$$

Output： 13<u>2</u>

$$l = 28 + \lfloor 148 \times 40/50 \rfloor = 146 = (10010010)_2$$
$$u = 28 + \lfloor 148 \times 41/50 \rfloor - 1 = 148 = (10010100)_2$$
$$MSB(l) = MSB(u)$$

- Completion
  - Five more rounds of bit shifting (do it as an exercise)
  - Eventually a '1' should be decoded
- Termination
  - After the advertised number of symbols have been decoded, or
  - EOT is reached

```
Initialize l, u, t
repeat
  k=0
  while( ⌊(u-l+1)×CC(x-1)/TC⌋ ≥ CC(k))
    k++
  x = decode_symbol(k)
  l=l+ ⌊(u-l+1)×CC(x-1)/TC⌋
  u=l+ ⌊(u-l+1)×CC(x)/TC⌋-1
  while(MSB(u)==MSB(l) OR E3(u,l))
    if(MSB(u)==MSB(l))
      l = (l<<1)+0
      u = (u<<1)+1
      t = (u<<1)+next_bit
    endif
    if(E3(u,l))
      l = (l<<1)+0
      u = (u<<1)+1
      t = (u<<1)+next_bit
      complement MSB(u), MSB(l), MSB(t)
    endif
  endwhile
until done
```

Arithmetic vs. Huffman

- m = sequence length
- Average code length:
  - Arithmetic

$$H(X) \le l_A \le H(x) + 2/m$$

  - Extended Huffman (groups of m symbols)

$$H(X) \le l_H \le H(x) + 1/m$$

- Observations
  - Both show the same asymptotic behavior
    - Slight edge for Huffman
  - Extended Huffman requires potentially enormous amounts of storage & code generation m^n
  - Arithmetic does not

➔ It is practical to assume large ***m*** for arithmetic but not Huffman

➔ We can expect arithmetic to do better (except when prob. are powers of 2)

PSP LAB

Arithmetic vs. Huffman (2)

- Gains are a function of the size & distribution of the alphabet
  - Small alphabet (generally) favors Huffman
  - Skewed distributions favor arithmetic
- It is easier to adapt arithmetic to use multiple codes
- It is easier to adapt arithmetic to changing input statistics
  - No tree to rearrange
  - We can separate modeling & coding

Adaptive Arithmetic Coding

- So far:
  - We start with a known probability model
- Reality
  - That information is usually unavailable or too time-consuming to obtain
  - Need a solution with no initial knowledge
- Adaptive solution
  - Start with all counts set to 1
    - Or some other known (to the decoder) model
  - After a symbol is encoded, update count
  - Decoder will be able to stay in sync
  - Quirks
    - No total count => cannot pick *m* based on that
    - Given a word length of *m*, we can accommodate max count of 2m-2
    - To prevent count overflow, rescale all counts
      - This has the added benefit of reducing the importance of older data

Applications: Image Compression

Adaptive arithmetic
pixel *values*

Image	Bits/pixel	Ratio Arithmetic	Ratio Huffman
Sena	6.52	1.23	1.16
Sensin	7.12	1.12	1.27
Earth	4.67	1.71	1.67
Omaha	6.84	1.17	1.14

Adaptive arithmetic
pixel *differences*

Image	Bits/pixel	Ratio Arithmetic	Ratio Huffman
Sena	3.89	2.06	2.08
Sensin	4.56	1.75	1.73
Earth	3.92	2.04	2.04
Omaha	6.27	1.28	1.26

Summary

- Introduced arithmetic coding
  - Direct coding of sequences (not a concatenation of codes)
  - Provably uniquely decodable
  - Asymptotically approaches entropy bound
  - More efficient for skewed distributions than Huffman
  - Only necessary codes are generated
  - Somewhat more complicated to implement
  - Easy adaptive implementation
  - Allows clean separation of model and coding

Problems & Extra

- Homeworks (Sayood 3rd, pp.114-115)
  - 4, 5, 6, 7, 8