Ch. 4 Arithmetic Coding

<u>Motivation – What are Problems w/ Huffman</u>

1. Can be inefficient (i.e., large redundancy)
    - This can be "solved" through Block Huffman
    - But… # of codewords grows exponentially

    See Example 4.2.1: $H_1(S) = 0.335$ bits/symbol

    But using Huffman we get avg length = 1.05 bits/symbol

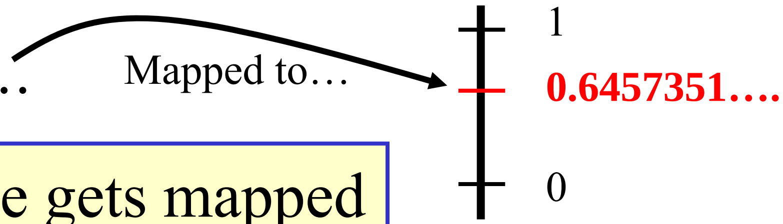
    Would need block size of 8 ➔ 6561-symbol alphabet to get close to H_1
2. High-Order Models (Non-IID) Hard to Address
    - Can be solved through Block Huffman
    - But # of codewords increases exponentially

➔ Underlying difficulty: Huffman requires keeping track of codewords <u>for *all* possible blocks</u>

➔ We need a way to assign a codeword to a <u>*particular* sequence</u> w/o having to generate codes for all possible sequences

Main Idea of Arithmetic Coding

Sequence: $S_1\ S_2\ S_3\ S_4 \ldots.$

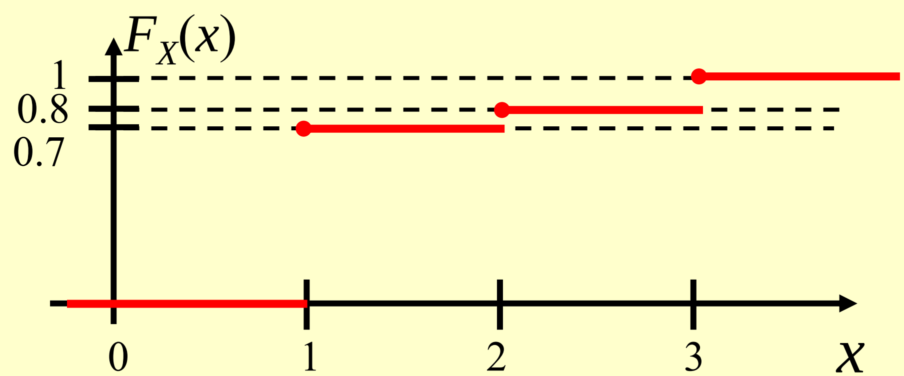


Each possible sequence gets mapped to a unique number in [0,1)

- The mapping depends on the prob. of the symbols
- You don't need to *a priori* determine all possible mappings
  - The Mapping is "built" as each new symbol arrives

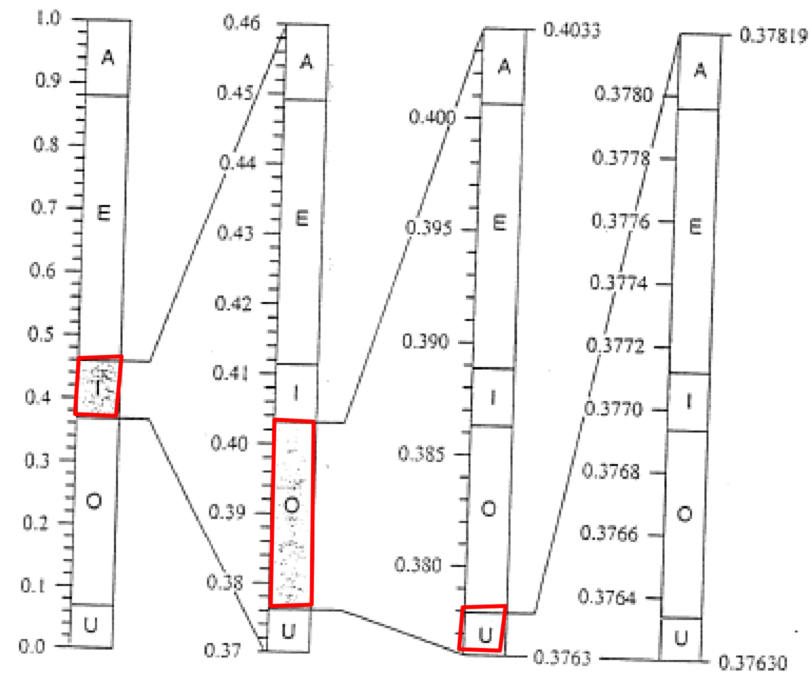
Recall **CDF of an RV**: symbols $\{a_1, a_2, a_3\}$ ➔ RV X w/ values: {1, 2, 3}

Consider $P(X=1) = 0.7$ $P(X=2) = 0.1$ $P(X=1) = 0.2$



Example: "Vowellish" (from Numerical Recipes Book)

Symbols	Probabilities	Optimal # Bits log2(1/Pi)
a	0.12	3.06
e	0.42	1.25
I	0.09	3.47
o	0.3	1.74
u	0.07	3.84



To send "iou": Send any # C such that

$$0.37630 \le C < 0.37819$$

Using Binary Fraction of

0.011000001 (9 bits)

9 bits for Arithmetic

vs

10 bits for Huffman

As each symbol is processed find new $\left\{ \begin{matrix} \text{upper limit} \\ \text{lower limit} \end{matrix} \right\}$ for interval

Math Result Needed to Program

Consider a sequence of RVs X = $(x_1, x_2, x_3, \ldots, x_n)$ corresponding to the sequence of symbols $(S_1, S_2, S_3, \ldots, S_n)$

Ex.

$$\text{Alphabet } = \{a_1, a_2, a_3\} \rightarrow \text{RV Values } \{1, 2, 3\}$$

$$(S_1\ S_2\ S_3\ S_4) = (a_2\ a_3\ a_3\ a_1) \rightarrow (x_1\ x_2\ x_3\ x_4) = (2\ 3\ 3\ 1)$$

Initial Values:

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

Interval Update:

$$l^{(n)} = \underbrace{l^{(n-1)} + \left[u^{(n-1)} - l^{(n-1)}\right]}_{\text{From Prev Interval}} \underbrace{F_X(x_n - 1)}_{\text{From Prob Model}}$$

$$u^{(n)} = \underbrace{l^{(n-1)} + \left[u^{(n-1)} - l^{(n-1)}\right]}_{\text{From Prev Interval}} \underbrace{F_X(x_n)}_{\text{From Prob Model}}$$

Checking Some Characteristics of Update

- What is the smallest $l^{(n)}$ can be?

$$l^{(n)} = l^{(n-1)} + \underbrace{\left[u^{(n-1)} - l^{(n-1)}\right]}_{>0} \underbrace{F_X(x_n - 1)}_{\geq 0} \Rightarrow \boxed{l^{(n)} \geq l^{(n-1)}}$$

- What is the largest $u^{(n)}$ can be?

$$u^{(n)} = l^{(n-1)} + \left[u^{(n-1)} - l^{(n-1)}\right] \underbrace{F_X(x_n)}_{\leq 1}$$

$$\leq l^{(n-1)} + \left[u^{(n-1)} - l^{(n-1)}\right] \Rightarrow \boxed{u^{(n)} \leq u^{(n-1)}}$$

These imply an important requirement for decoding:

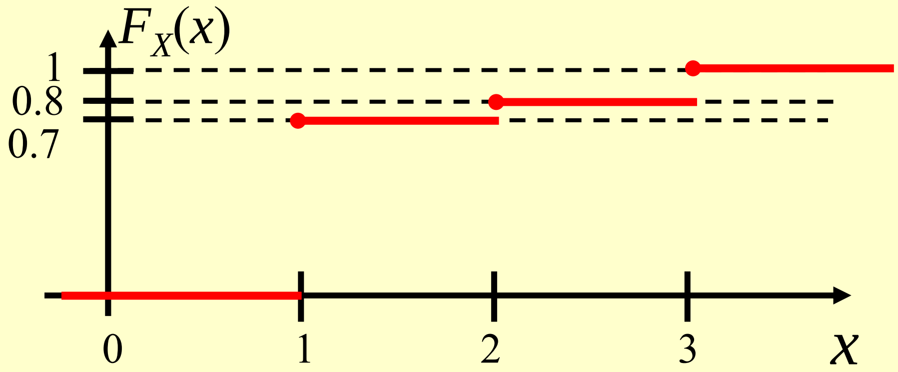
New Interval $\subseteq$ Old Interval

Example of Applying the Interval Update

Symbols $\{a_1, a_2, a_3\}$ ➔ RV X w/ values: $\{1, 2, 3\}$

Consider $P(X=1) = 0.7$ $P(X=2) = 0.1$ $P(X=1) = 0.2$

CDF for this alphabet/RV



Consider the sequence $(a_1\ a_3\ a_2) \rightarrow (1\ 3\ 2)$

To process the first symbol "1"

$F_X(0)$

$$l^{(1)} = l^{(0)} + \left[u^{(0)} - l^{(0)}\right] F_X(1-1) = 0 + [1-0] \times 0 \quad = 0$$

$$u^{(1)} = l^{(0)} + \left[u^{(0)} - l^{(0)}\right] F_X(1) \quad = 0 + [1-0] \times 0.7 = 0.7$$

$F_X(1)$

To process the 2nd symbol "3"

$F_X(2)$

$$l^{(2)} = l^{(1)} + \left[u^{(1)} - l^{(1)}\right] F_X(3-1) = 0 + [0.7-0] \times 0.8 = 0.56$$

$$u^{(2)} = l^{(1)} + \left[u^{(1)} - l^{(1)}\right] F_X(3) \quad = 0 + [0.7-0] \times 1 \quad = 0.7$$

$F_X(3)$

To process the 3rd symbol "2"

$F_X(1)$

1 0 1 0 1 0 1 0 0 0

$$l^{(3)} = l^{(2)} + \left[u^{(2)} - l^{(2)}\right] F_X(2-1) = 0.56 + [0.7-0.56] \times 0.7 = 0.658$$

$$u^{(3)} = l^{(2)} + \left[u^{(2)} - l^{(2)}\right] F_X(2) \quad = 0.56 + [0.7-0.56] \times 0.8 = 0.672$$

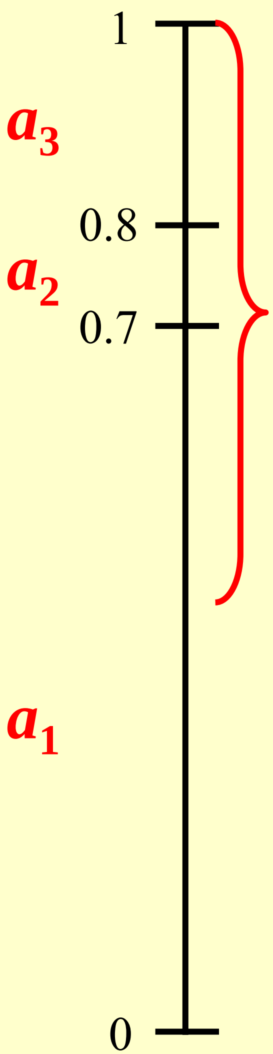
$F_X(2)$

So… send a number in the interval [0.658,0.672) Pick 0.6640625

$0.6640625_{10} = 0.1010101_2$ **Code = 1 0 1 0 1 0 1**

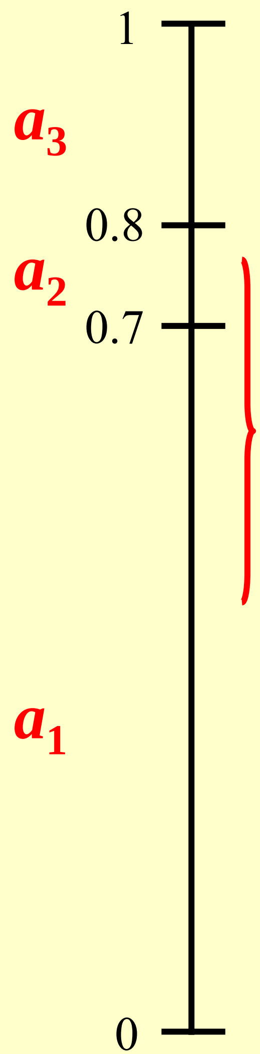
<u>**Decoding Received Code = 1 0 1 0 1 0 1**</u>

$$1 \rightarrow \begin{cases} 0.1111111\ldots = 1 \\ 0.1000000\ldots = 0.5 \end{cases}$$



After 1st bit [0.5 1)…
Can’t Decode

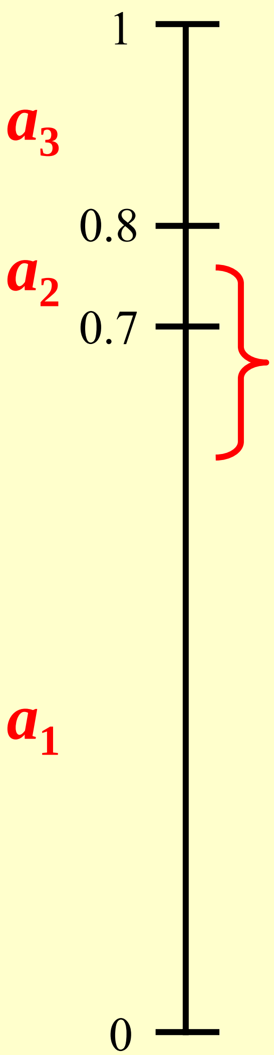
$$10 \rightarrow \begin{cases} 0.1011111\ldots = 0.75 \\ 0.1000000\ldots = 0.5 \end{cases}$$



After 2nd bit [0.5 0.75)…
Can’t Decode… But first symbol <u>can’t</u> be a_3

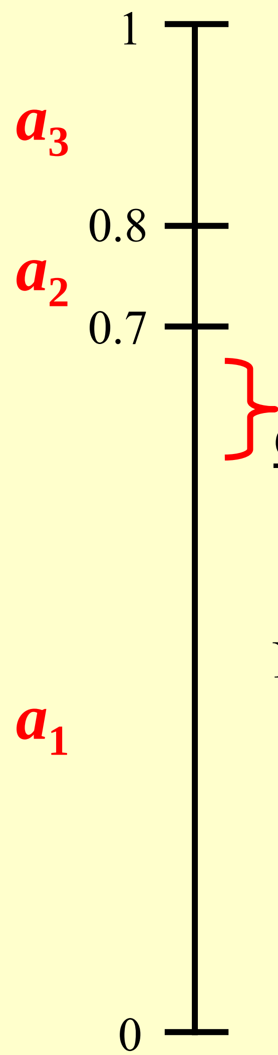
Decoding Continued: Received Code = 1 0 1 0 1 0 1

$$101 \rightarrow \begin{cases} 0.1011111... = 0.75 \\ 0.1010000... = 0.625 \end{cases}$$



[0.625 , 0.75)…
Can’t Decode

$$1010 \rightarrow \begin{cases} 0.1010111... = 0.6875 \\ 0.1010000... = 0.625 \end{cases}$$



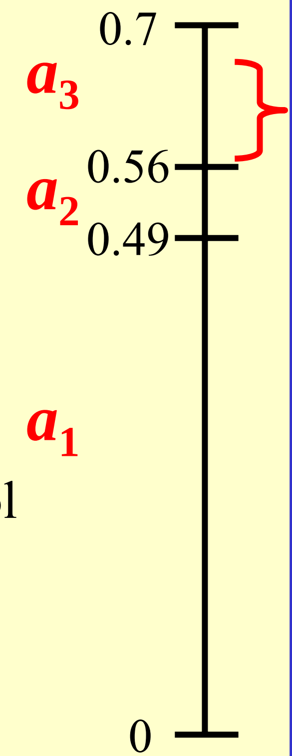
[0.625 , 0.6875)…
Can Decode 1st symbol!!!

It is a_1

Now slice up [0, 0.7)…

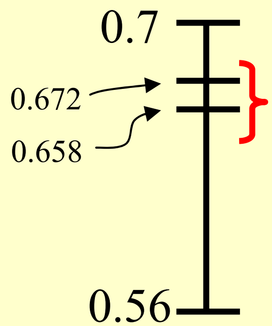
…can decode 2nd symbol

It is a_3



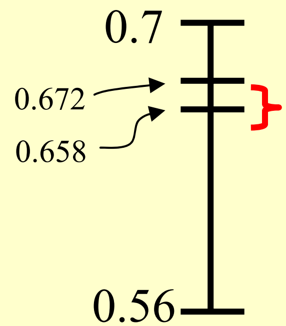
<u>Decoding Continued: Received Code = 1 0 1 0 1 0 1</u>

$$10101 \rightarrow \begin{cases} 0.1010111\ldots = 0.6875 \\ 0.1010100\ldots = 0.65625 \end{cases}$$



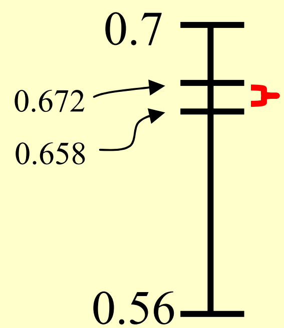
[0.65625 , 0.6875)…
Can't Decode

$$101010 \rightarrow \begin{cases} 0.10101011\ldots = 0.671875 \\ 0.10101000\ldots = 0.65625 \end{cases}$$



[0.65625, 0.671875)…
Can't Decode

$$1010101 \rightarrow \begin{cases} 0.1010101111\ldots = 0.671875 \\ 0.1010101000\ldots = 0.6640625 \end{cases}$$



[0.6640625, 0.671875)
<u>Can</u> Decode 3rd symbol!!!

It is a_2

In practice there are ways to handle termination issues!

Main Results on Uniqueness & Efficiency

1. A "binary tag" lying between $l^{(n)}$ and $u^{(n)}$ can be found
2. The tag can be truncated to a finite # of bits
3. The truncated tag still lies between $l^{(n)}$ and $u^{(n)}$
4. The truncated tag is Unique & Decodable
5. For IID sequence of length *m:*

$$\boxed{H(S) \leq \bar{l}_{Arith} < H(S) + \boxed{\frac{2}{m}}}$$

Compared to Huffman:

$$\boxed{H(S) \leq \bar{l}_{Huff} < H(S) + \boxed{\frac{1}{m}}}$$

Hey! AC is worse than Huffman??!! So why consider AC???!!!
Remember, this is for coding the entire length of *m* symbols…
You'd need 2^m codewords in Huffman… which is impractical!
But for AC is VERY practical!!!

For Huffman must be kept small but for AC it can be VERY large!

How AC Overcomes Huffman's Problems

1. **Efficiency**: Huffman can only achieve close to $H(S)$ by using large block codes… which means you need a pre-designed codebook of exponentially growing size
   - AC enables coding large blocks w/o having to know codewords *a priori*
   - w/ AC you just generate the code for the entire given sequence
     - No *a priori* codebook is needed
2. **Higher-Order Models**: Huffman can use Cond. Prob. Models… ***but*** you need to build an *a priori* codebook for each context… which means a large codebook
   - Context coding via conditional probabilities is easy in AC
   - For each context you have a prob model for the symbols
     - Next "slicing of the interval" is done using prob model for the currently observed context… no need to generate all the *a priori* codewords!

Ex.: 1st Order Cond. Prob Models for AC

Suppose you have three symbols and you have a 1st order conditional probability model for the source emitting these symbols…

For the first symbol in the sequence you have a std Prob Model

For subsequent symbols in the sequence you have 3 context models

$$\begin{aligned} P(a_1) &= 0.2 \\ P(a_2) &= 0.4 \\ P(a_3) &= 0.4 \\ \hline \sum_i P(a_i) &= 1 \end{aligned}$$

$$\begin{aligned} P(a_1 \mid a_1) &= 0.1 \\ P(a_2 \mid a_1) &= 0.5 \\ P(a_3 \mid a_1) &= 0.4 \\ \hline \sum_i P(a_i \mid a_1) &= 1 \end{aligned}$$

$$\begin{aligned} P(a_1 \mid a_2) &= 0.95 \\ P(a_2 \mid a_2) &= 0.01 \\ P(a_3 \mid a_2) &= 0.04 \\ \hline \sum_i P(a_i \mid a_2) &= 1 \end{aligned}$$

$$\begin{aligned} P(a_1 \mid a_3) &= 0.45 \\ P(a_2 \mid a_3) &= 0.45 \\ P(a_3 \mid a_3) &= 0.1 \\ \hline \sum_i P(a_i \mid a_3) &= 1 \end{aligned}$$

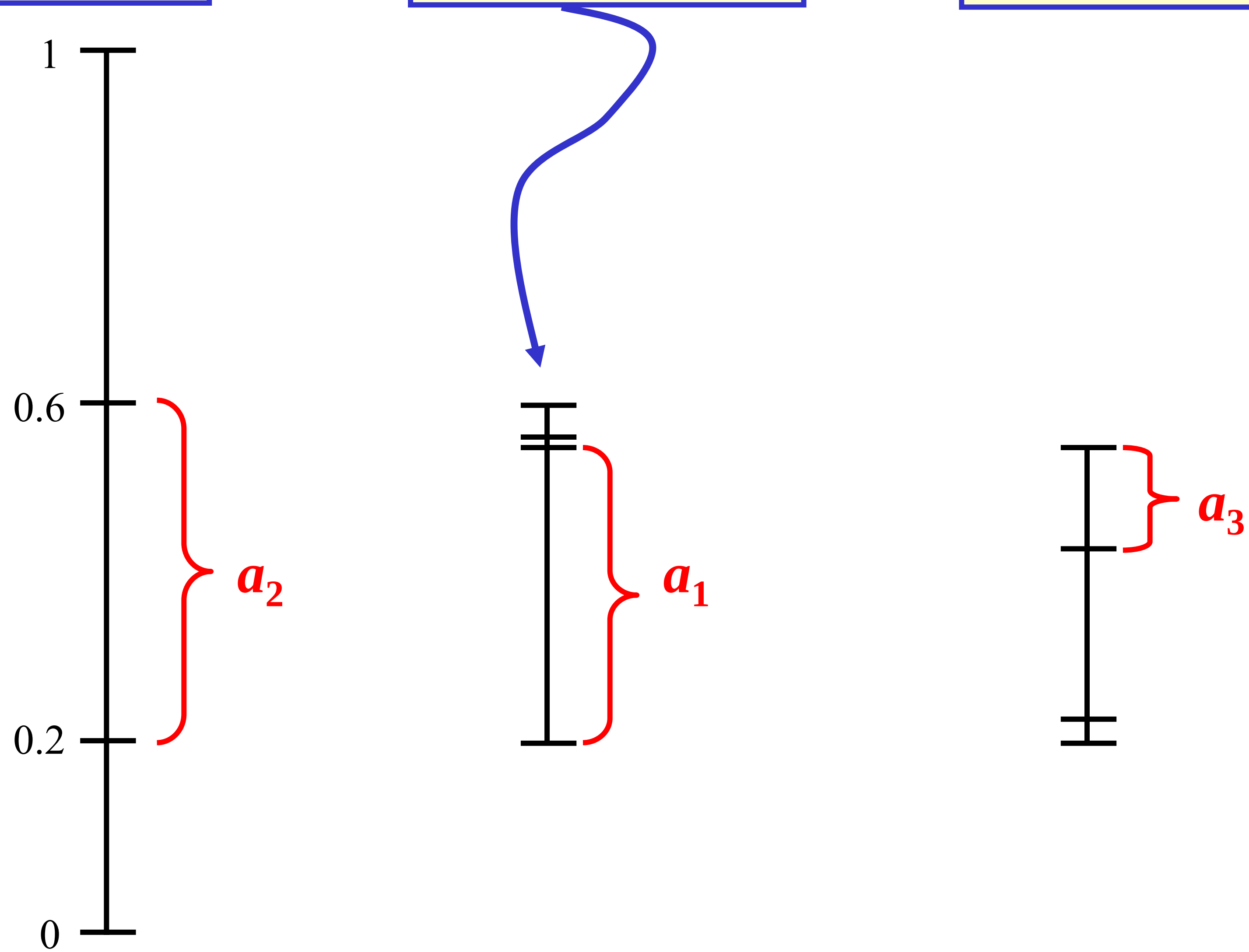
Now let's see how these are used to code the sequence $a_2\ a_1\ a_3$

Note: Decoder needs to know these models

$$P(a_1) = 0.2$$
$$P(a_2) = 0.4$$
$$P(a_3) = 0.4$$

$$P(a_1 \mid a_2) = 0.95$$
$$P(a_2 \mid a_2) = 0.01$$
$$P(a_3 \mid a_2) = 0.04$$

$$P(a_1 \mid a_1) = 0.1$$
$$P(a_2 \mid a_1) = 0.5$$
$$P(a_3 \mid a_1) = 0.4$$



Ex.: Similar for Adaptive Prob. Models

- Start with some *a priori* "prototype" prob model (could be cond.)
- Do coding with that for awhile as you observe the actual frequencies of occurrence of the symbols
  - Use these observations to update the probability models to better model the ***ACTUAL*** source you have!!
- Can continue to adapt these models as more symbols are observed
  - Enables tracking probabilities of source with changing probabilities
- Note: Because the decoder starts with the same prototype model and sees the same symbols the coder uses to adapt… it can automatically synchronize adaptation of its models to the coder!
  - As long as there are no transmission errors!!!