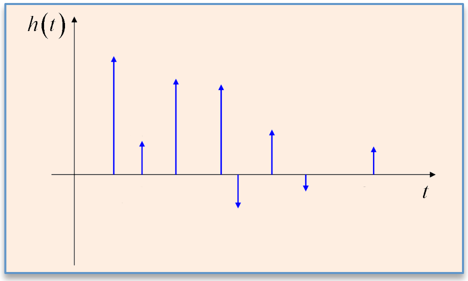
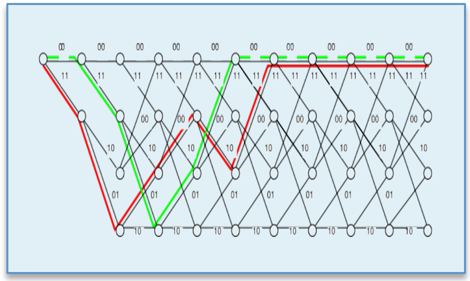
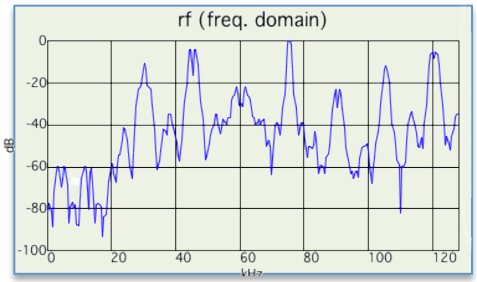
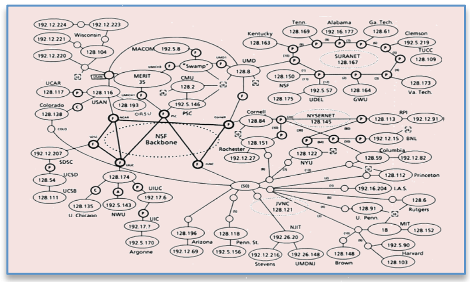
Sign up on Piazza please, ASAP! Only 2/3 of the class has done this so far.

**There's a lot of course business that gets transacted there,
and only there**









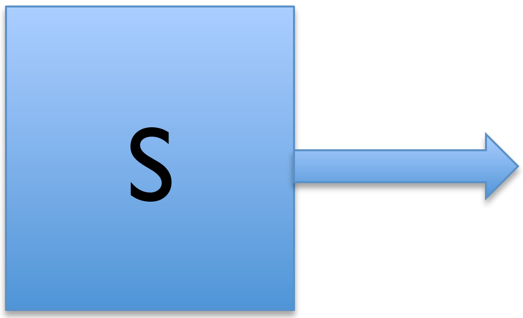
INTRODUCTION TO EECS II

DIGITAL COMMUNICATION SYSTEMS

6.02 Fall 2013
Lecture #2

- More on entropy, coding and Huffman codes
- Lempel-Ziv-Welch adaptive variable-length compression

A Probabilistic Message Source

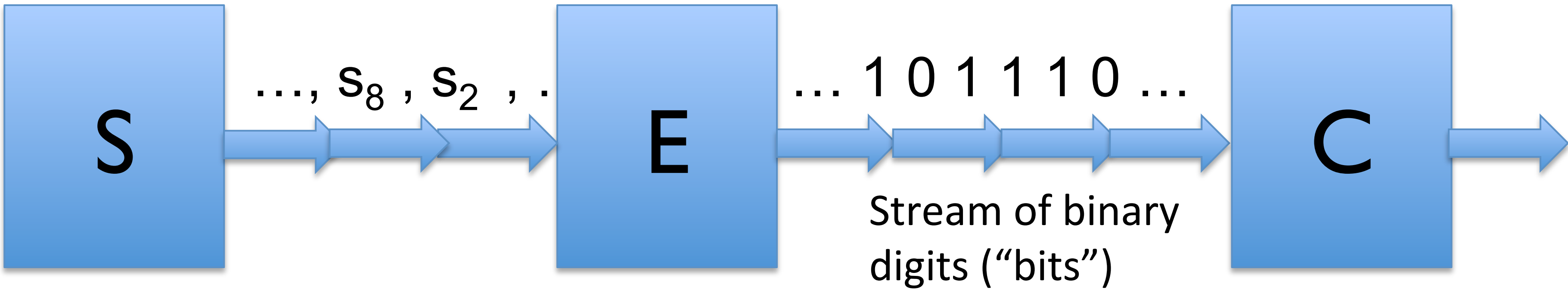


$\ldots, s_8, s_2, s_{11}, s_1, s_4, s_{13}, s_2, \ldots$

Shannon's simplest model of an information source:

Emits symbols sequentially in time, with the symbol at each time chosen independently of the choices at other times, but using the same probability distribution, i.e., an independent, identically distributed or i.i.d. symbol stream.

Source Coder for a Binary Channel



Source

Symbol s_i occurs with probability p_i

Entropy **H(S)** = average information (in bits) per symbol s_i

Binary Source Coder

Converts symbols s_i to codewords made up of sequences of binary digits 0/1 suited to transmission on a binary channel.

Binary Channel

Takes a 0 or 1 at the input, produces a 0 or 1 at the output.

Entropy and Coding

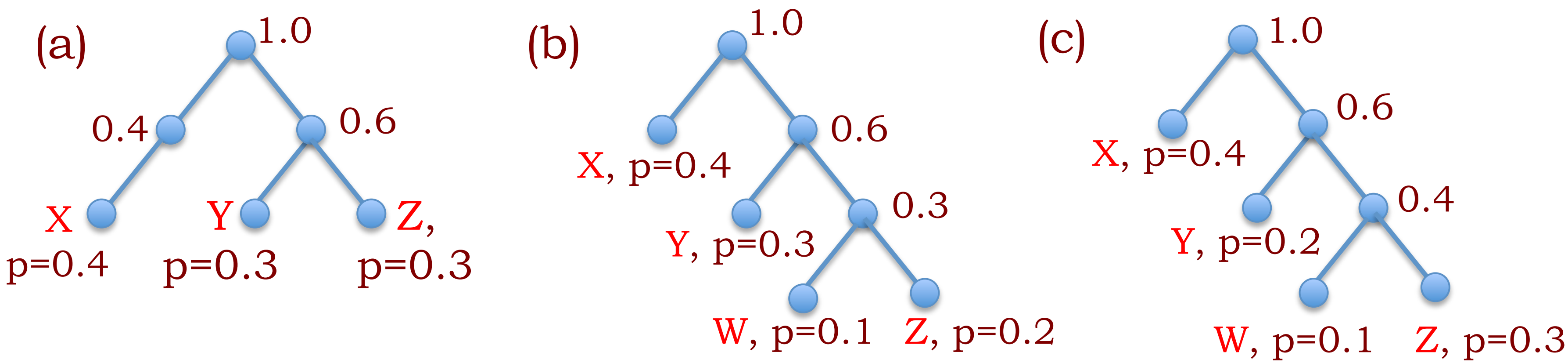
- The maximum average information a binary digit can carry is 1 bit.
- Hence, with source information being produced at an average rate of H(S) bits per emitted symbol, we need to transmit at least H(S) binary digits per emission on average. Thus the expected length of a binary code – i.e., the expected number of binary digits per symbol – must satisfy

$$L \geq H(S)$$

(Proof outlined later, after introducing the Kraft inequality)

Binary Code Trees

Key property: Symbols only at leaves, for instantaneous decoding

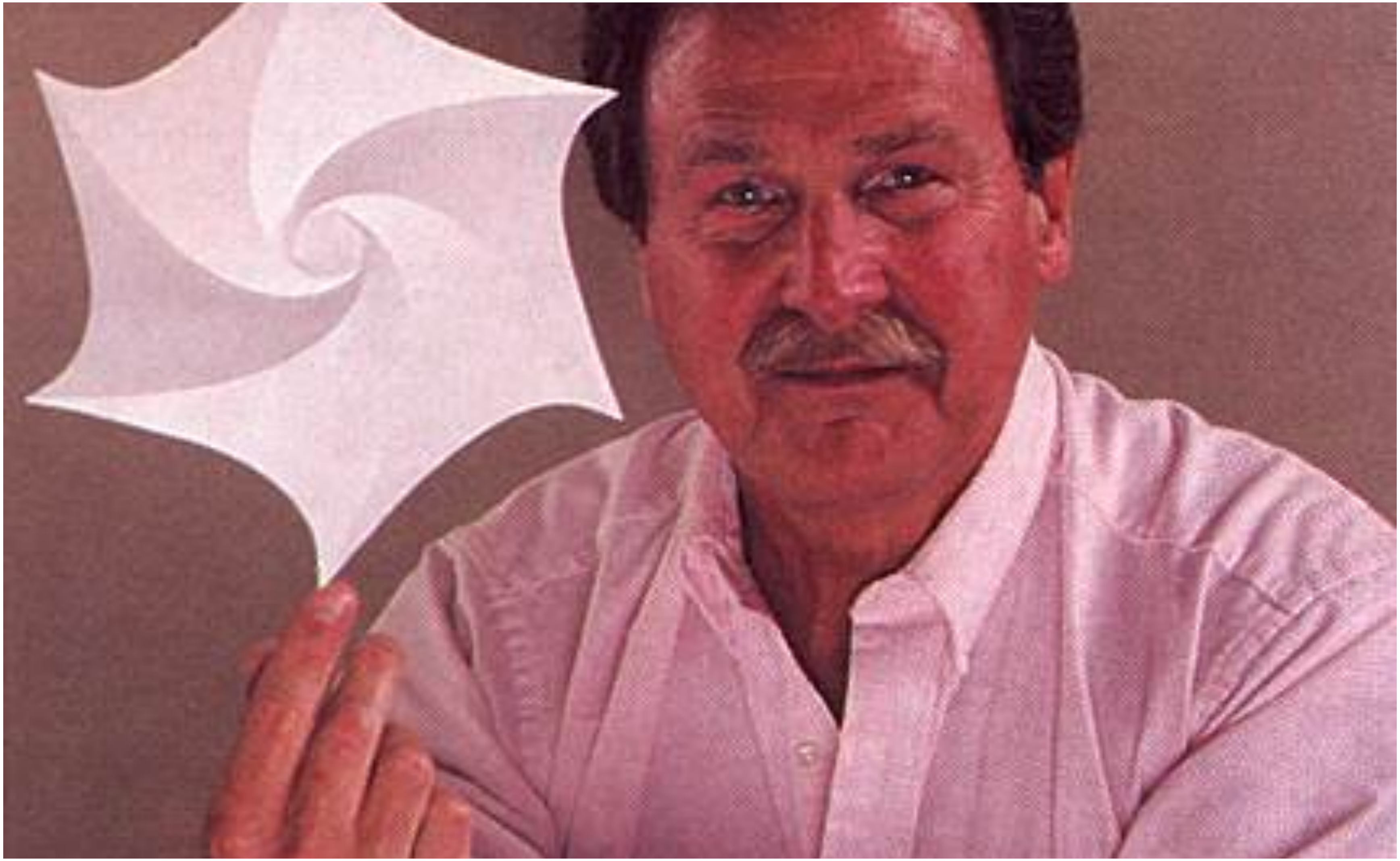


- Why is (a) inefficient (in terms of codeword lengths)? Not a full tree!
- Why is (c) non-optimal (in terms of expected codeword length)? Can be improved by swapping Y and Z!
- Is (b) optimal (i.e., minimum expected codeword length)? Yes!

Useful observation for computation of expected codeword length L:

L = sum of probabilities at all internal nodes (including the root)

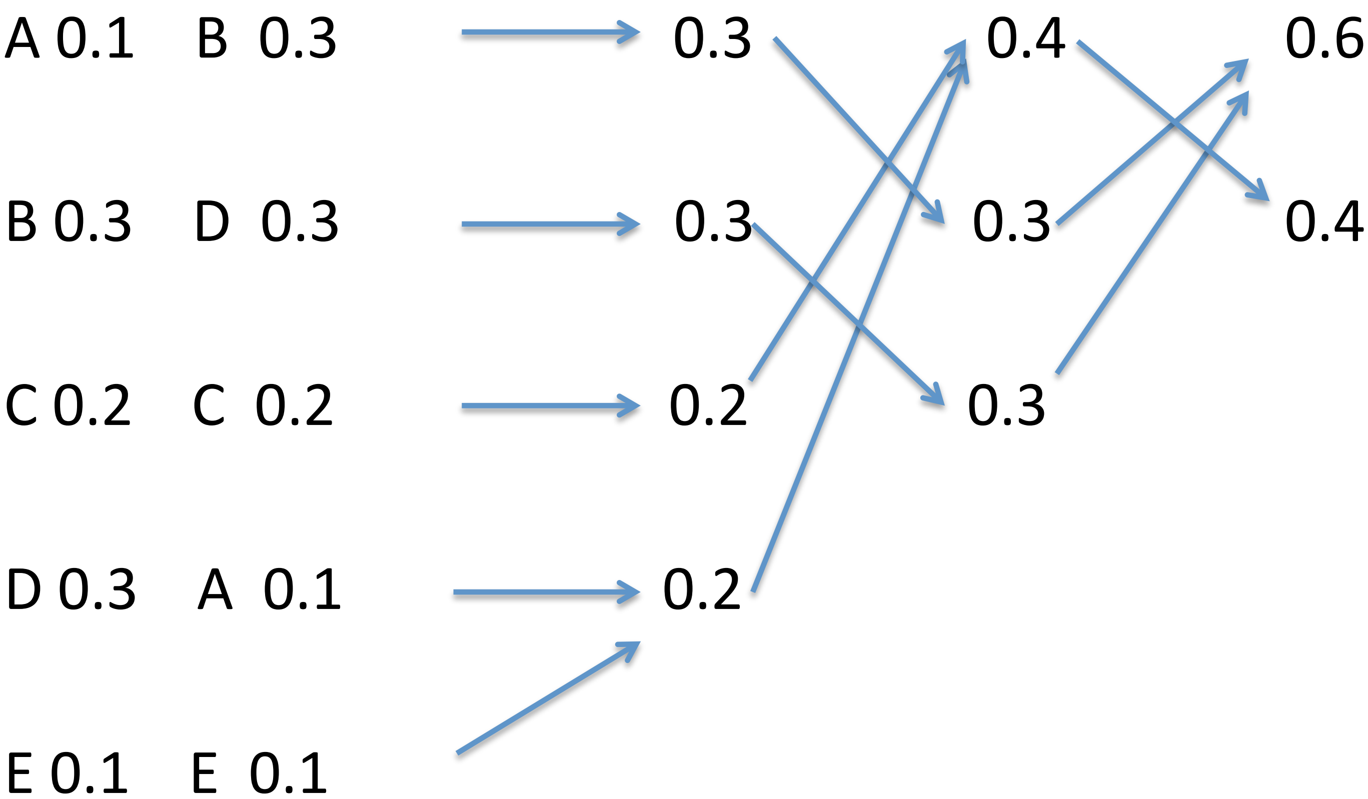
David Huffman



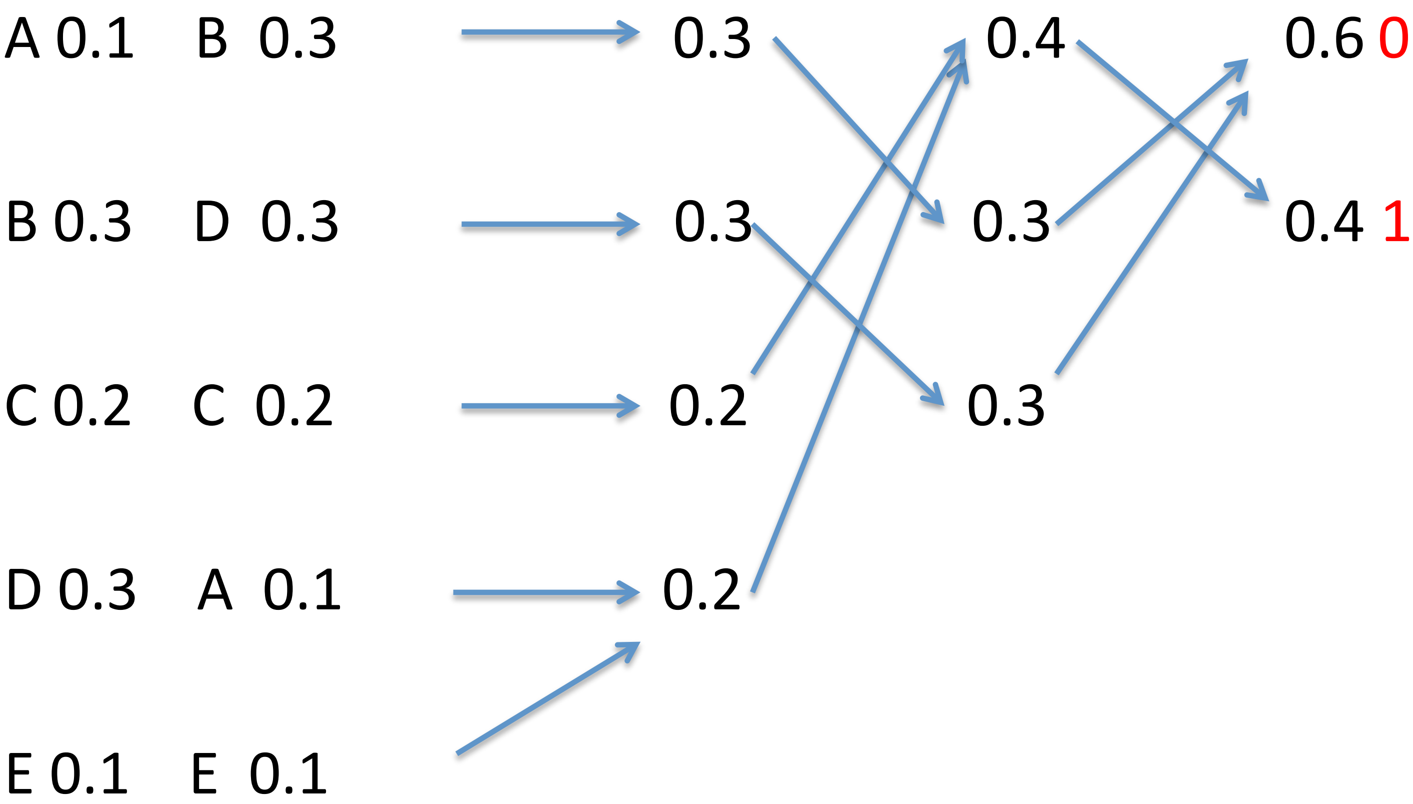
http://www.huffmancoding.com/my-uncle/scientific-american

Algorithm in term paper for MIT graduate class, 1951

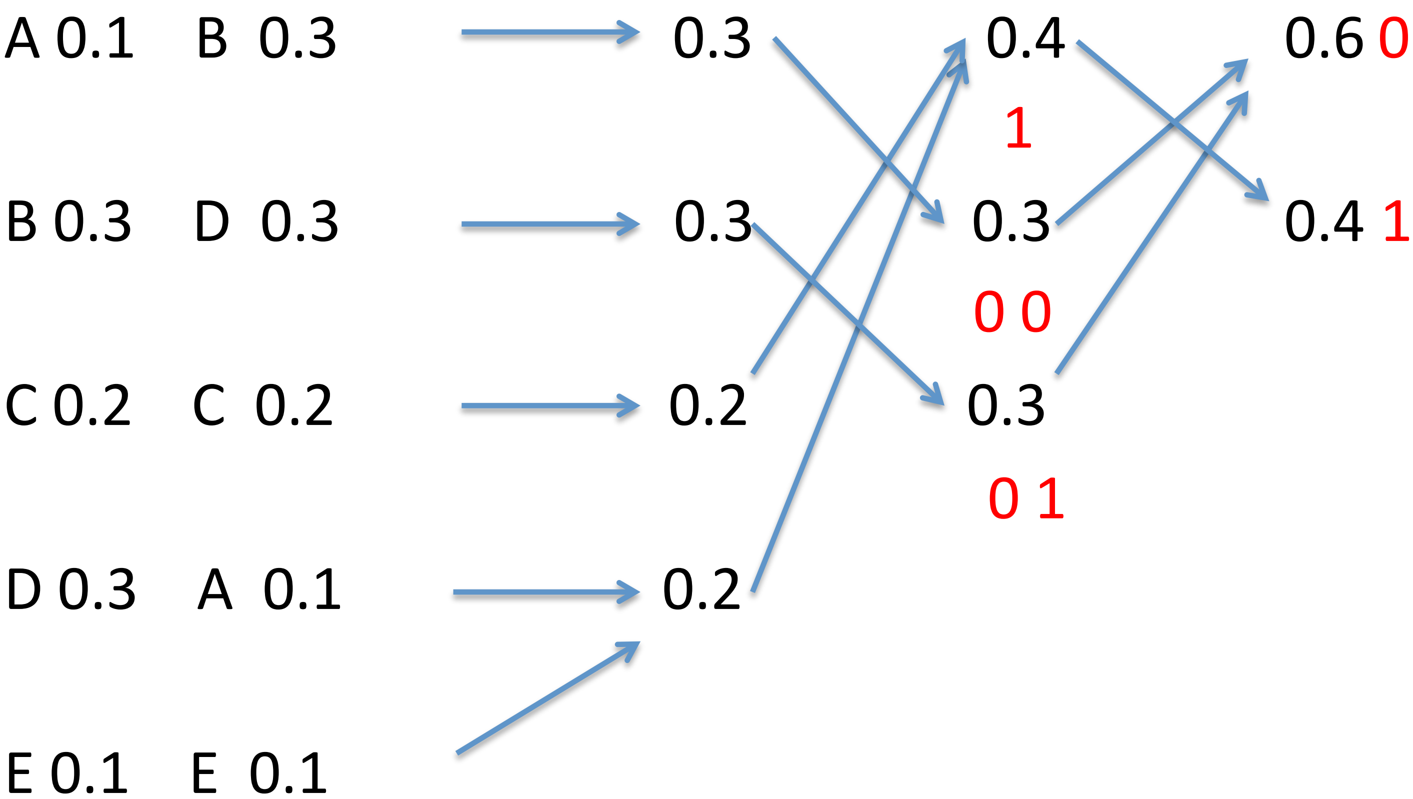
Huffman Reduction



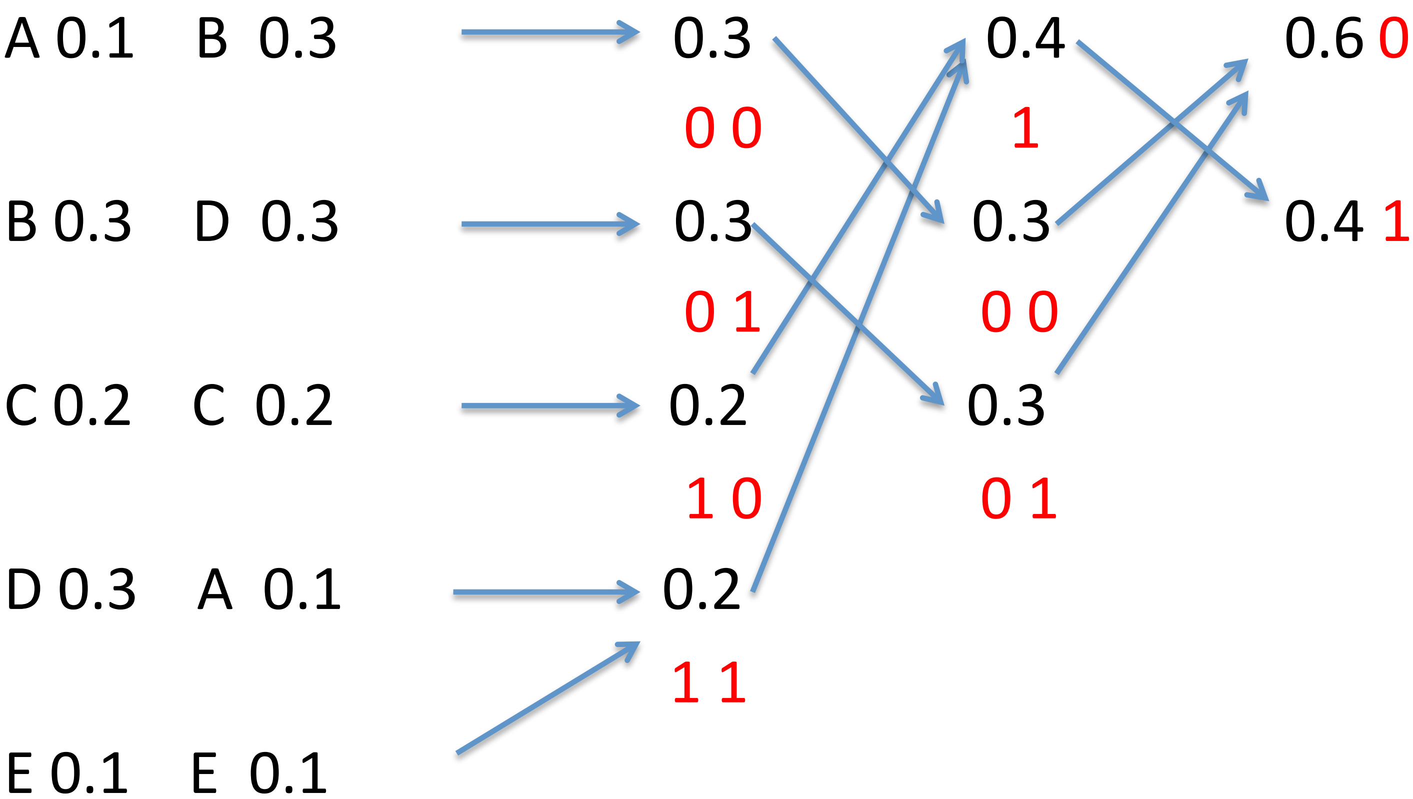
Huffman Trace-back



Huffman Trace-back



Huffman Trace-back



Huffman Trace-back

A 0.1	B 0.3	0.3	0.4	0.6 0
	0 0	0 0	1	
B 0.3	D 0.3	0.3	0.3	0.4 1
	0 1	0 1	0 0	
C 0.2	C 0.2	0.2	0.3	
	1 0	1 0	0 1	
D 0.3	A 0.1	0.2		
	1 1 0	1 1		
E 0.1	E 0.1			
	1 1 1			

$\log_2 5 = 2.32$

$L = 2.2$

$H = 2.17$

Kraft Identity for Full Binary Trees

- MIT EE Masters thesis, 1949 (two years before Huffman's insight)
- For a full binary code tree (i.e., two child nodes for every non-leaf node) with codeword lengths $\{l_i\}$,

$$\sum_i \frac{1}{2^{l_i}} = 1$$

- "Physics" proof: Start with a "mass" of 1 at the root. Move it down the tree, sending exactly half the mass down each branch. A mass of $\frac{1}{2^{l_i}}$ ends up at the i-th leaf, which is at distance l_i from the root. The masses must add up to 1.

(If the tree is not full, then mass can't be split at some stage, so exceeds what it should be at next level, and = changes to ≤, to yield Kraft's inequality)

Conversely, …

- … for a set of lengths $\{l_i\}$ satisfying the Kraft inequality, there is a binary code tree with codewords of precisely these lengths.
- Proof by construction: Imagine a complete binary tree of depth $l_{\max}$, with each node from the root downwards bifurcating into two descendants till this depth is reached.

Start at the level of the tree corresponding to length $l_{\min}$, pick any set of nodes at this level to be the leaves corresponding to the codewords of this (minimum) length; remove their descendants.

Now pick any set of remaining nodes (the Kraft inequality guarantees there will be remaining nodes) at level $l_{\min}+1$ to be the codewords of this length; remove their descendants. And so on.

Proofs of Code Properties

- Formal proof of $L \geq H$ uses the easily demonstrated fact that for probability distributions $\{p_i\}$, $\{q_i\}$,

$$-\sum_i p_i \log q_i \geq -\sum_i p_i \log p_i ,$$

with equality if and only if $p_i = q_i$ for all i. Pick $q_i = \boxed{\frac{1}{2^{l_i}}}$.
Note that Kraft's identity for full binary code trees guarantees that this is a valid probability distribution.

- Equality, $L = H$, when $p_i = \frac{1}{2^{l_i}}$, i.e., when $l_i = \log(1/p_i)$, i.e., when length of each codeword = information in that codeword

Proofs, continued

- Proof that there exists a sensible code with $L < H+1$ (strict inequality):

By constructing a code, not necessarily optimal, that satisfies this.

Choose $l_i \geq \log(1/p_i) > l_i - 1$, i.e., the i-th codeword length is the "ceiling" (smallest integer above or equal to) of the information in the i-th symbol. This choice satisfies Kraft's inequality, so an associated code can be constructed. Multiplying the above inequalities by p_i and summing, we get

$L \geq H > L-1$, hence $L < H+1$ for this code, and therefore for Huffman too.

Huffman Coding of Symbol Blocks

- Given the symbol probabilities, Huffman finds an instantaneously decodable code of minimal expected length L, and satisfying

$$H(S) \leq L \leq H(S)+1$$

- Instead of coding the individual symbols of an iid source, we could code pairs $s_i s_j$, whose probabilities are $p_i p_j$. The entropy of this "super-source" is 2H(S) (because the two symbols are independently chosen), and the resulting Huffman code on N^2 "super-symbols" satisfies

$$2H(S) \leq 2L \leq 2H(S)+1$$

where L still denotes expected length per symbol codeword. So now $H(S) \leq L \leq H(S)+(1/2)$

- Extend to coding K at a time

Another (optional) way to think about Entropy and Coding: Typical Sequences

- Consider an iid source S emitting one of the symbols $s_1, s_2, \ldots, s_N$ at each time, with probabilities $p_1, p_2, \ldots, p_N$ respectively, independently of symbols emitted at other times
- In a very long string of K emissions, we expect to typically get $Kp_1, Kp_2, \ldots, Kp_N$ instances of the symbols $s_1, s_2, \ldots, s_N$ respectively. (This is a very simplified statement of the "law of large numbers".)
- Also, all ways of getting these are equally likely

So …

- The probability of any one such typical string is

  p_1^(Kp_1).p_2^(Kp_2)… p_N^(Kp_N)   (^ = exponentiation)

  so the number of such strings is approximately

  p_1^($-Kp_1$).p_2^($-Kp_2$)… p_N^($-Kp_N$).

Taking the $\log_2$ of this number, we get KH(S).

- So the number of such typical sequences is around $2^{KH(S)}$. It takes KH(S) binary digits to count this many sequences, so around H(S) binary digits per symbol to code the typical sequences.
- Inefficient coding of the non-typical sequences does not hurt expected length much

Some Limitations of Huffman Coding

- Symbol probabilities
  - may not be known
  - may change with time
- Source
  - may not generate iid symbols, e.g., English text.

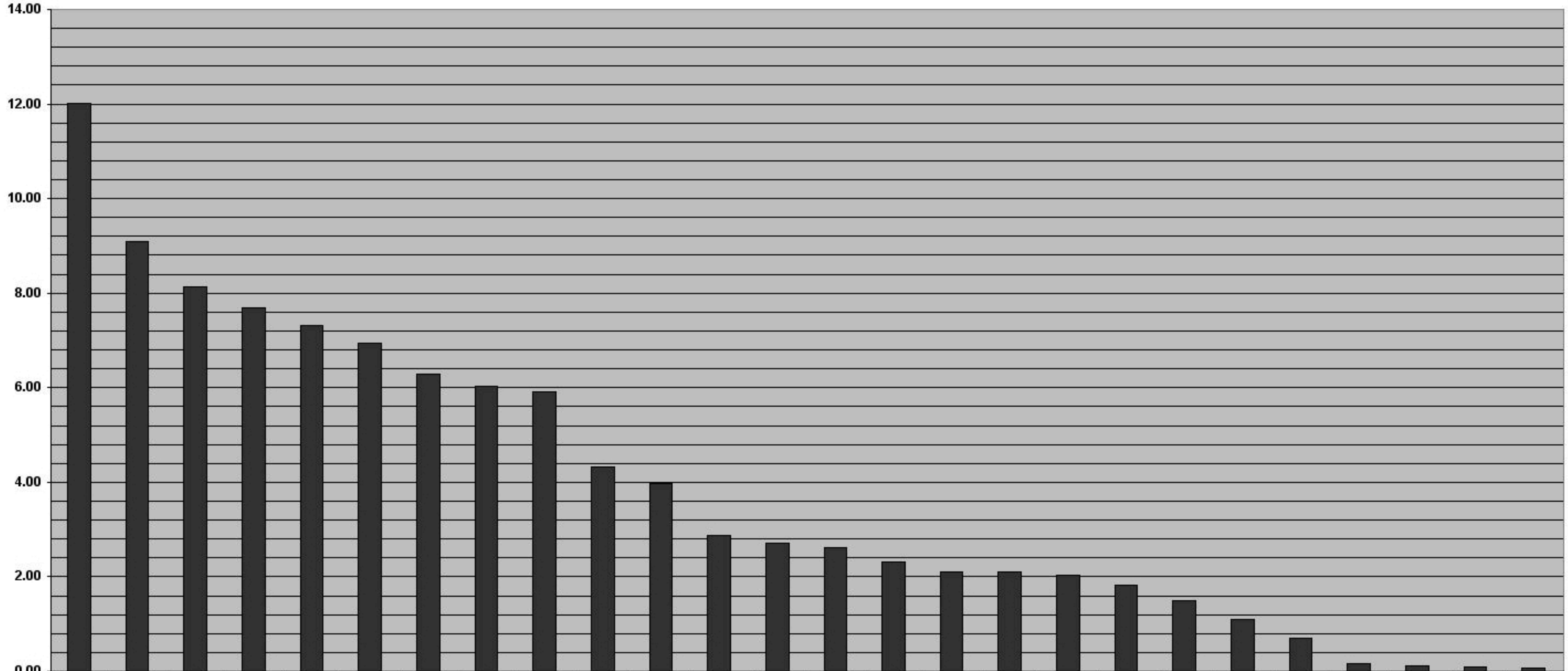
  Could still code symbol by symbol, but this won't be efficient at exploiting the redundancy in the text.

  Assuming 27 symbols (lower-case letters and space) for English text, could use a fixed-length binary code with 5 binary digits (counts up to $2^5 = 32$).

  Could do better with a variable-length code because even assuming equiprobable symbols,

$$H = \log_2 27 = 4.755 \text{ bits/symbol}$$

What is the Entropy of English?



	E	T	A	O	I	N	S	R	H	D	L	U	C	M	F	Y	W	G	P	B	V	K	X	Q	J	Z
Frequency	12.02	9.10	8.12	7.68	7.31	6.95	6.28	6.02	5.92	4.32	3.98	2.88	2.71	2.61	2.30	2.11	2.09	2.03	1.82	1.49	1.11	0.69	0.17	0.11	0.10	0.07

Taking account of actual individual symbol probabilities, but not using context, entropy = 4.177 bits per symbol

http://www.math.cornell.edu/~mec/2003-2004/cryptography/subs/frequencies.html

What exactly is it we want to determine?

- Average per-symbol entropy over long sequences, or the entropy rate:

$$\underline{H} = \lim_{K \to \infty} H(S_1, S_2, S_3, \ldots, S_K)/K$$

or the related (under certain conditions, identical)

$$\underline{H'} = \lim_{K \to \infty} H(S_K \mid S_{K-1}, S_{K-2}, \ldots, S_1)/K$$

where S_j denotes the symbol in position j in the text.

English Text has Lots of Context

- Write down the next letter (or next 3 letters!) in the snippet

*Nothing can be said to be certain, except death and ta*_

But x has a very low occurrence probability

(0.0017) in English words

  – Letters are not independently generated!

- Shannon (1951) and others have found that the entropy of English text is a lot lower than 4.177
  – Shannon estimated 0.6-1.3 bits/letter using human experiments
  – More recent estimates: 1-1.5 bits/letter

Lempel-Ziv-Welch (1977,'78,'84)

- Universal lossless compression of sequential (streaming) data by adaptive variable-length coding
- Widely used, sometimes in combination with Huffman (gif, tiff, png, pdf, zip, gzip, …)
- Patents have expired --- much confusion and distress over the years around these and related patents
- Ziv was also (like Huffman and Kraft) an MIT graduate student in the "golden years" of information theory, early 1950's
- Theoretical performance: Under appropriate assumptions on the source, asymptotically attains the lower bound $\underline{H}$ on compression performance

We'll learn LZW by doing

Compress the following message:

abcabcabcabcabcabcabc

assuming the dictionary contains a, b, c to begin with.

(You need to go some distance out on this to encounter the special case discussed later.)

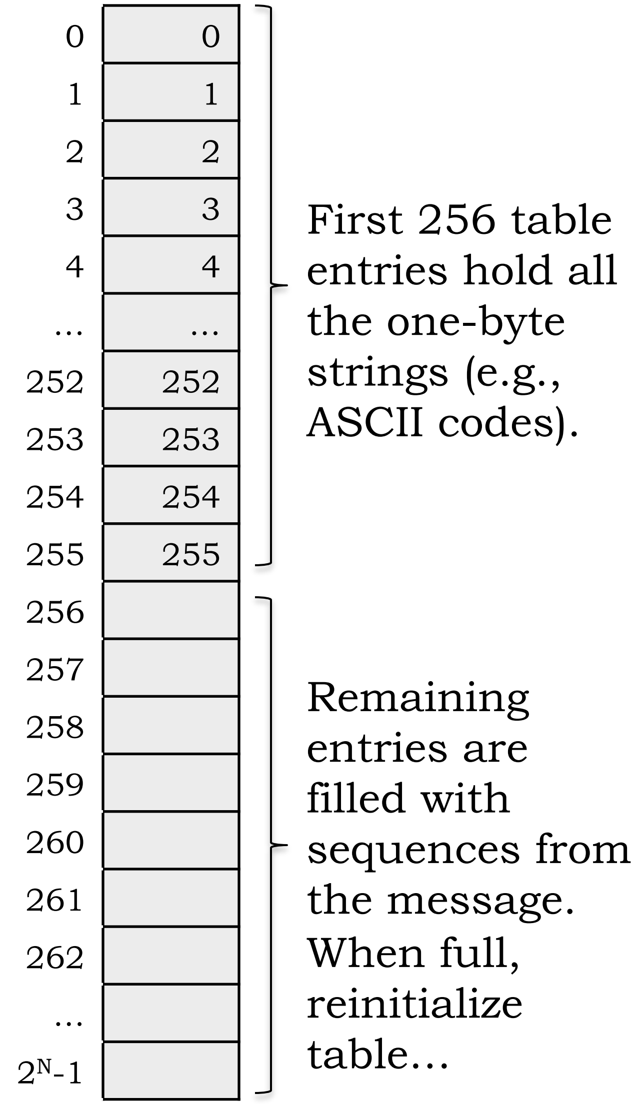
Characteristics of LZW

"Universal lossless compression of sequential (streaming) data by adaptive variable-length coding"

- Universal: doesn't need to know source statistics in advance. Learns source characteristics in the course of building a dictionary for sequential strings of symbols encountered in the source text
- Compresses streaming text to sequence of dictionary addresses --- these are the codewords sent to the receiver
- Variable length source strings assigned to fixed length dictionary addresses (codes)
- Starting from an agreed core dictionary of symbols, receiver builds up a dictionary that mirrors the sender's, with a one-step delay, and uses this to exactly recover the source text (losslessly)
- Regular resetting of the dictionary when it gets too big allows adaptation to changing source characteristics

LZW: An Adaptive Variable-length Code

- Algorithm first developed by Ziv and Lempel (LZ88, LZ78), later improved by Welch.
- As message is processed, encoder builds a "string table" that maps symbol sequences to an N-bit fixed-length code. Table size = 2^N
- Transmit table indices, usually shorter than the corresponding string → compression!
- Note: String table can be reconstructed by the decoder using information in the encoded stream – the table, while central to the encoding and decoding process, *is never transmitted*!



LZW Encoding

```
STRING = get input symbol
WHILE there are still input symbols DO
    SYMBOL = get input symbol
    IF STRING + SYMBOL is in the STRINGTABLE THEN
        STRING = STRING + SYMBOL
    ELSE
        output the code for STRING
        add STRING + SYMBOL to STRINGTABLE
        STRING = SYMBOL
    END
END

output the code for STRING
```

S=string, c=symbol (character) of text

1. If S+c is in table, set S=S+c and read in next c.
2. When S+c isn't in table: send code for S, add S+c to table.
3. Reinitialize S with c, back to step 1.

From http://marknelson.us/1989/10/01/lzw-data-compression/

Example: Encode "abbbabbbab···"

256	ab
257	bb
258	bba
259	abb
260	bbab
261	
262	

1. Read a; string = a
2. Read b; ab not in table
   output 97, add ab to table, string = b
3. Read b; bb not in table
   output 98, add bb to table, string = b
4. Read b; bb in table, string = bb
5. Read a; bba not in table
   output 257, add bba to table, string = a
6. Read b, ab in table, string = ab
7. Read b, abb not in table
   output 256, add abb to table, string = b
8. Read b, bb in table, string = bb
9. Read a, bba in table, string = bba
10. Read b, bbab not in table
    output 258, add bbab to table, string = b

Encoder Notes

- The encoder algorithm is greedy – it's designed to find the longest possible match in the string table before it makes a transmission.
- The string table is filled with sequences actually found in the message stream. No encodings are wasted on sequences not actually found in the input data.
- Note that in this example the amount of compression increases as the encoding progresses, i.e., more input bytes are consumed between transmissions.
- Eventually the table will fill and then be reinitialized, recycling the N-bit codes for new sequences. So the encoder will eventually adapt to changes in the probabilities of the symbols or symbol sequences.

LZW Decoding

```
Read CODE
STRING = TABLE[CODE] // translation table
output STRING
WHILE there are still codes to receive DO
    Read CODE from encoder
    IF CODE is not in the translation table THEN
        ENTRY = STRING + STRING[0]
    ELSE
        ENTRY = get translation of CODE
    END
    output ENTRY
    add STRING+ENTRY[0] to the translation table
    STRING = ENTRY
END
```

(Ignoring special case in IF):

1. Translate received code to output the corresponding table entry E=e+R (e is first symbol of entry, R is rest)
2. Enter S+e in table.
3. Reinitialize S with E, back to step 1.

A special case: cScSc

- Suppose the string being examined at the source is cSc, where c is a specific character or symbol, S is an arbitrary (perhaps null) but specific string (i.e., all c and S here denote the same fixed symbol, resp. string).
- Suppose cS is in the source and receiver tables already, and cSc is new, then the algorithm outputs the address of cS, enters cSc in its table, and holds the symbol c in its string, anticipating the following input text.
- The receiver does what it needs to, and then holds the string cS in anticipation of the next transmission. All good.
- But if the next portion of input text is Scx, the new string at the source is cScx ---not in the table, so the algorithm outputs the address of cSc and makes a new entry for cScx.
- The receiver does not yet have cSc in its table, because it's one step behind! However, it has the string cS, and can deduce that the latest table entry at the source must have its last symbol equal to its first. So it enters cSc in its table, and then decodes the most recently received address.

A couple of concluding thoughts

- LZW is a good example of compression or communication schemes that "transmit the model" (with auxiliary information to run the model), rather than "transmit the data"

- There's a whole world of lossy compression!

Sign up on Piazza please, ASAP! Only 2/3 of the class has done this so far.

There's a lot of course business that gets transacted there, and only there