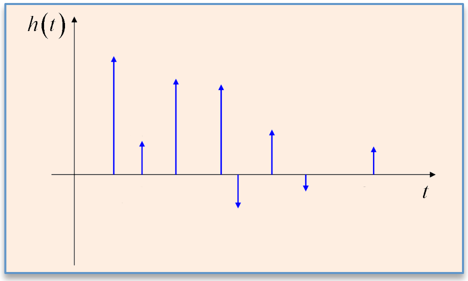
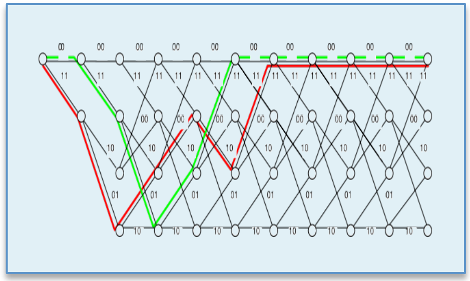
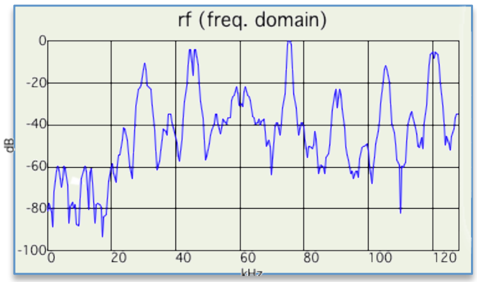
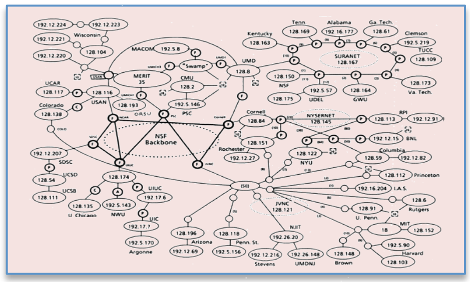








INTRODUCTION TO EECS II

DIGITAL COMMUNICATION SYSTEMS

6.02 Fall 2012
Lecture #2

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

Entropy and Coding

- The entropy H(S) of a source S at some time represents the uncertainty about the source output at that time, or the expected information in the emitted symbol.
- If the source emits repeatedly, choosing independently at each time from the same fixed distribution, we say the source generates independent and identically distributed (iid) symbols.
- With information being produced at this average rate of H(S) bits per emission, we need to transmit at least H(S) binary digits per emission on average (since the maximum information a binary digit can carry is one bit).

Bounds on Expected Code Length

- We limit ourselves to instantaneously decodable (i.e., prefix-free) codes --- these put the symbols at the leaves of a code tree.
- If L is the expected length of the code, the reasoning on the previous slide suggests that we need $H(S) \leq L$. The proof of this bound is not hard, see for example the very nice book by Luenberger, *Information Science*, 2006.
- Shannon showed how to construct codes satisfying $L \leq H(S)+1$ (see Luenberger for details), but did not have a construction for codes with minimal expected length.
- Huffman came up with such a construction.

Huffman Coding

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

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

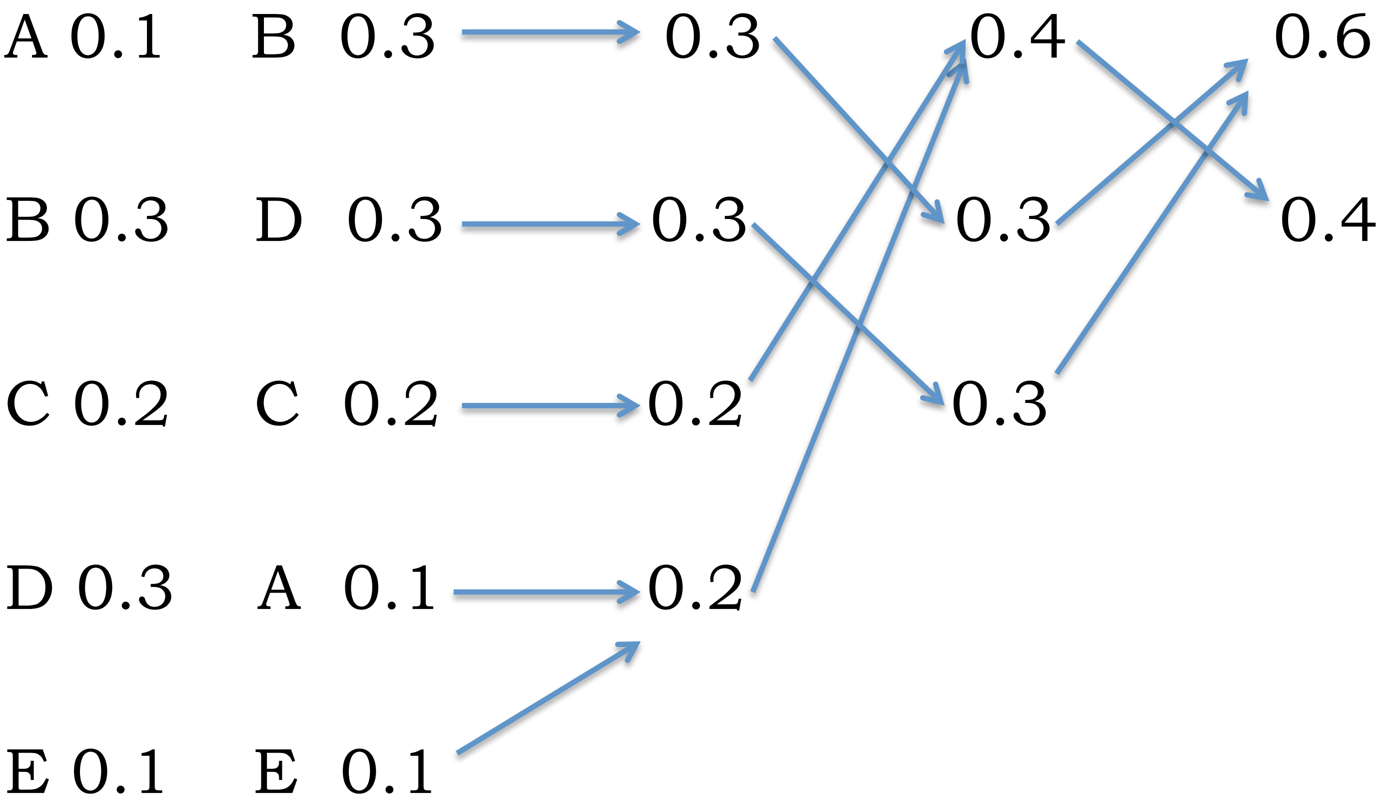
- Instead of coding the individual symbols of an iid source, we could code pairs $s_i s_j$, whose probabilities are $p_i p_j$. The entropy of this "super-source" is 2H(S) (because the two symbols are independently chosen), and the resulting Huffman code on N^2 "super-symbols" satisfies

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

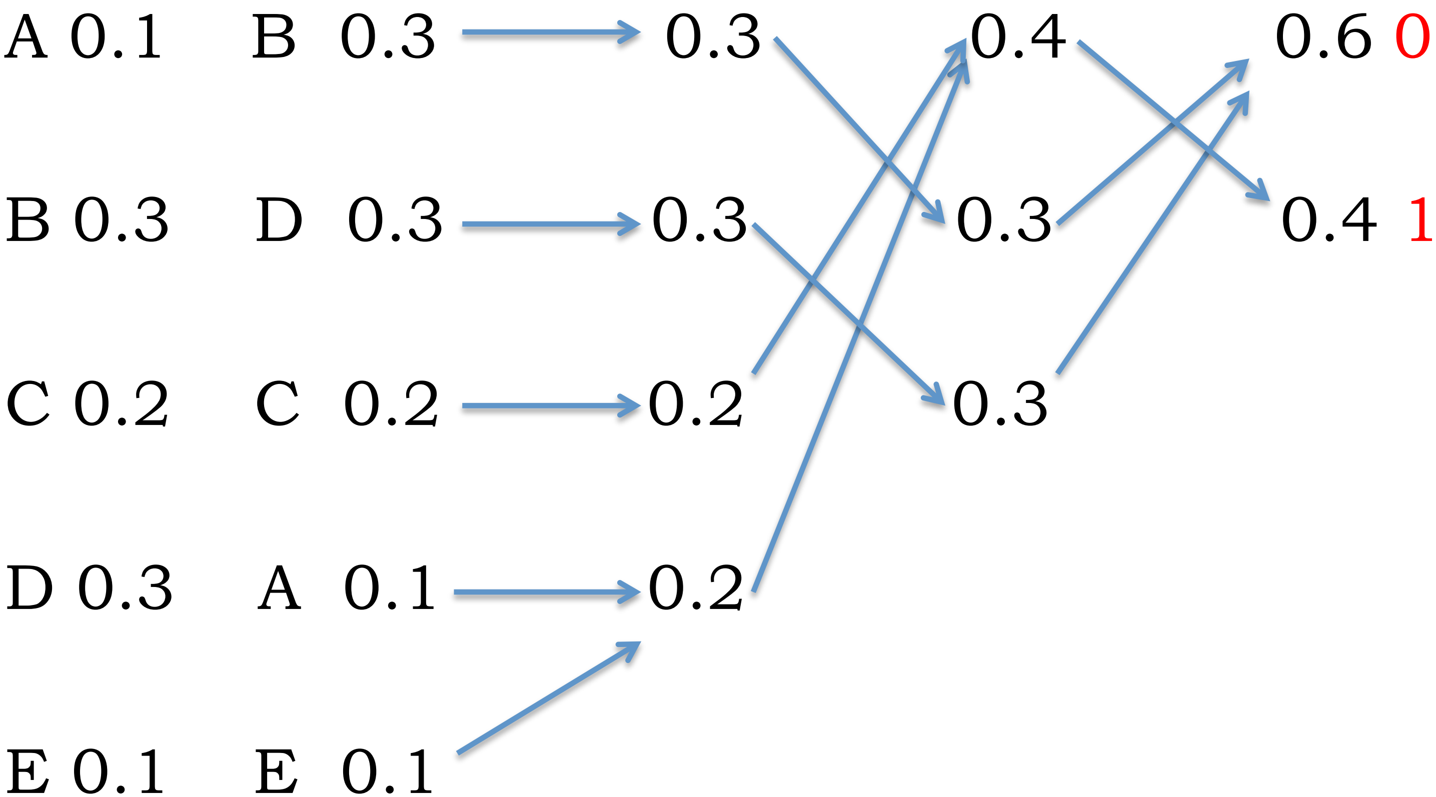
where L still denotes expected length per symbol codeword. So now $H(S) \leq L \leq H(S)+(1/2)$

- Extend to coding K at a time

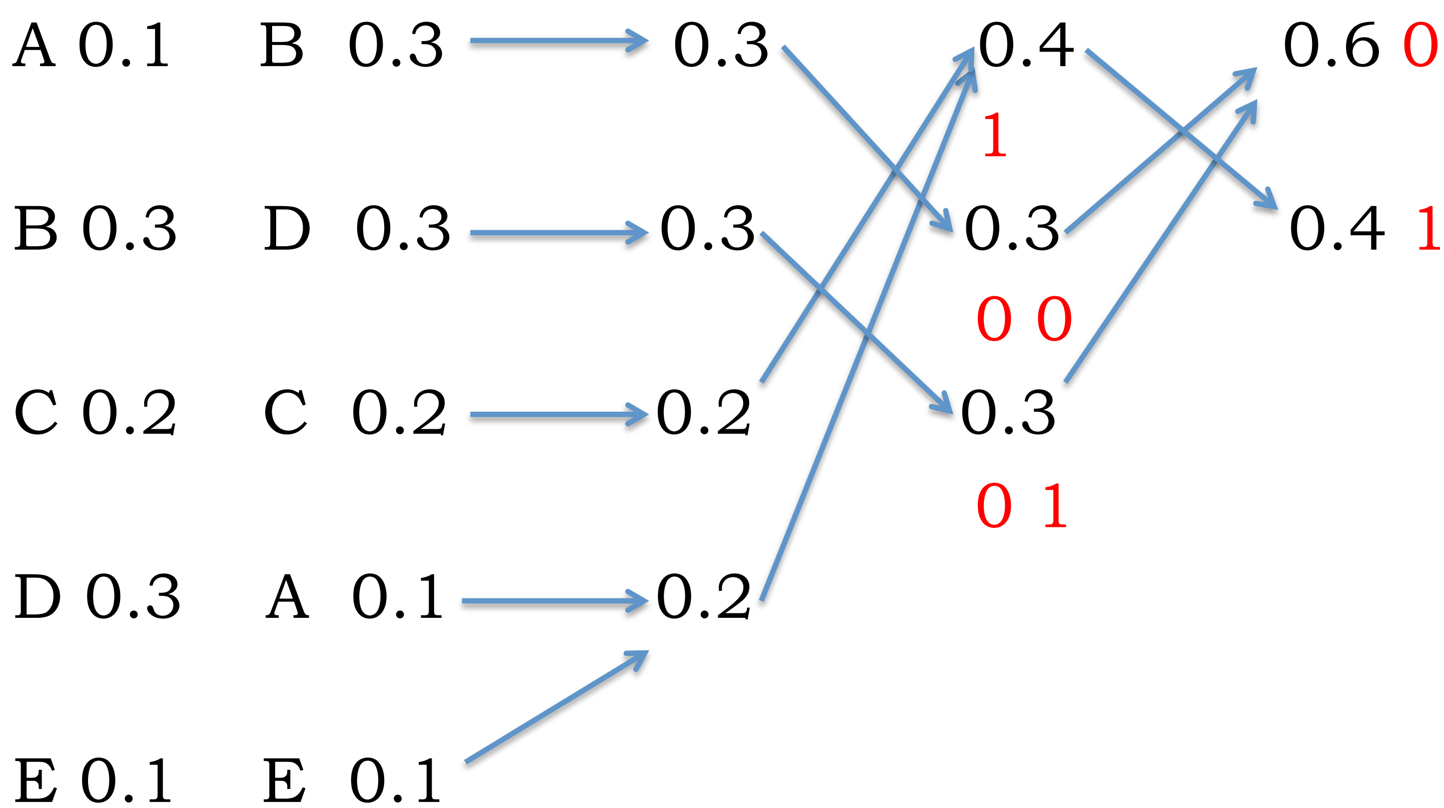
Reduction



Trace-back

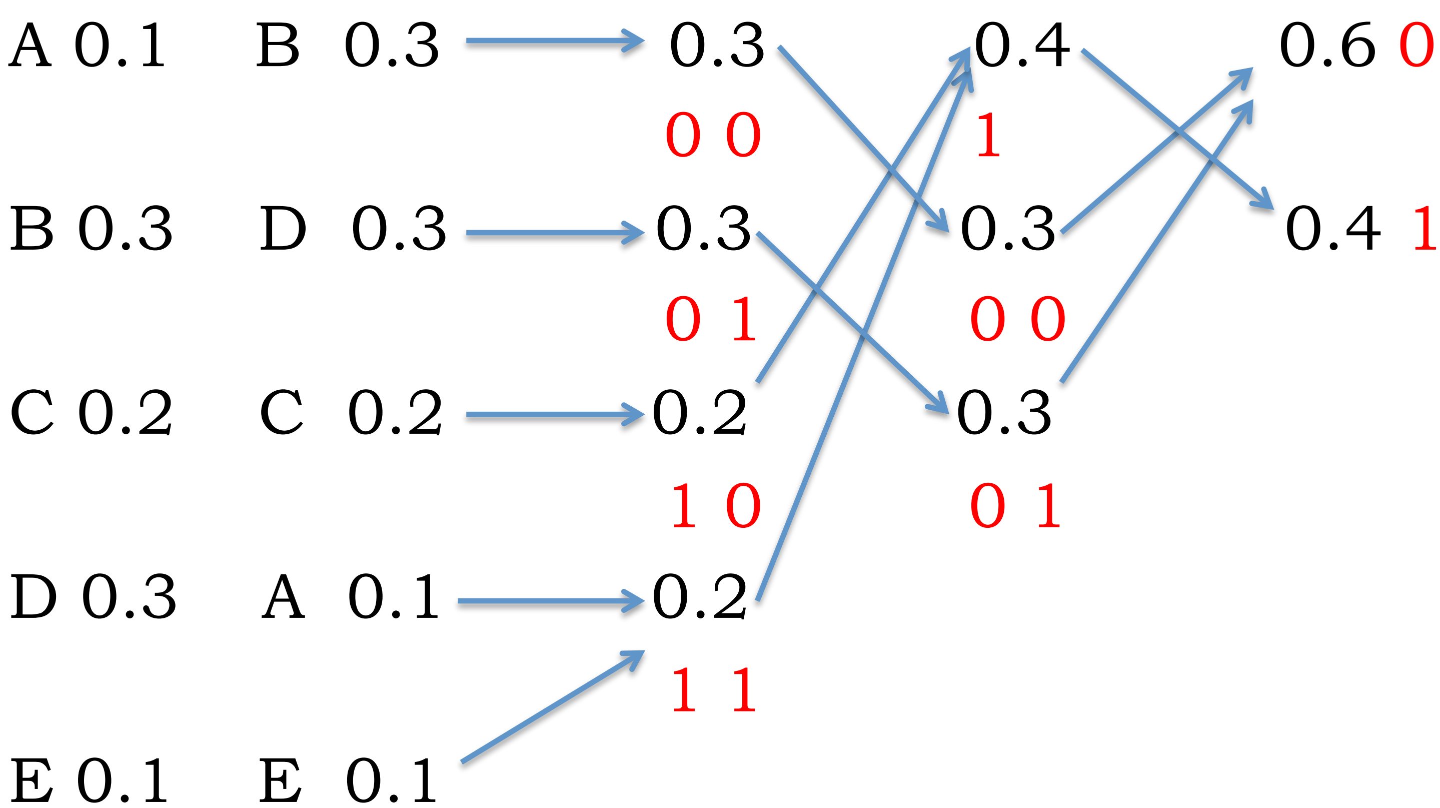

A 0.1
B 0.3
C 0.2
D 0.3
E 0.1
B 0.3
D 0.3
C 0.2
A 0.1
E 0.1
0.3
0.3
0.2
0.2
0.4
0.3
0.3
0.6 0
0.4 1

Trace-back

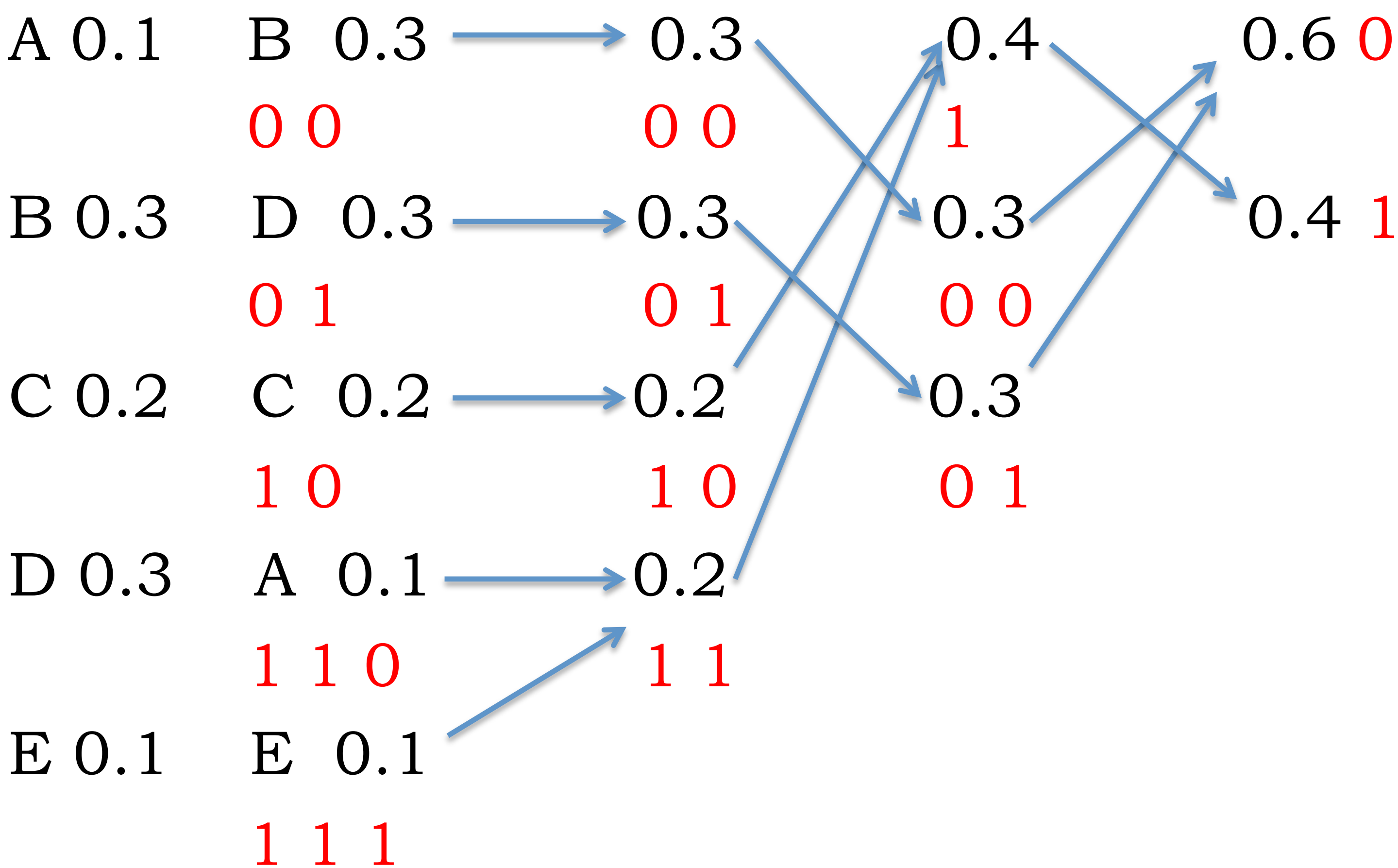


A 0.1
B 0.3
C 0.2
D 0.3
E 0.1
B 0.3
D 0.3
C 0.2
A 0.1
E 0.1
0.3
0.3
0.2
0.2
0.4
1
0.3
0 0
0.3
0 1
0.6 0
0.4 1

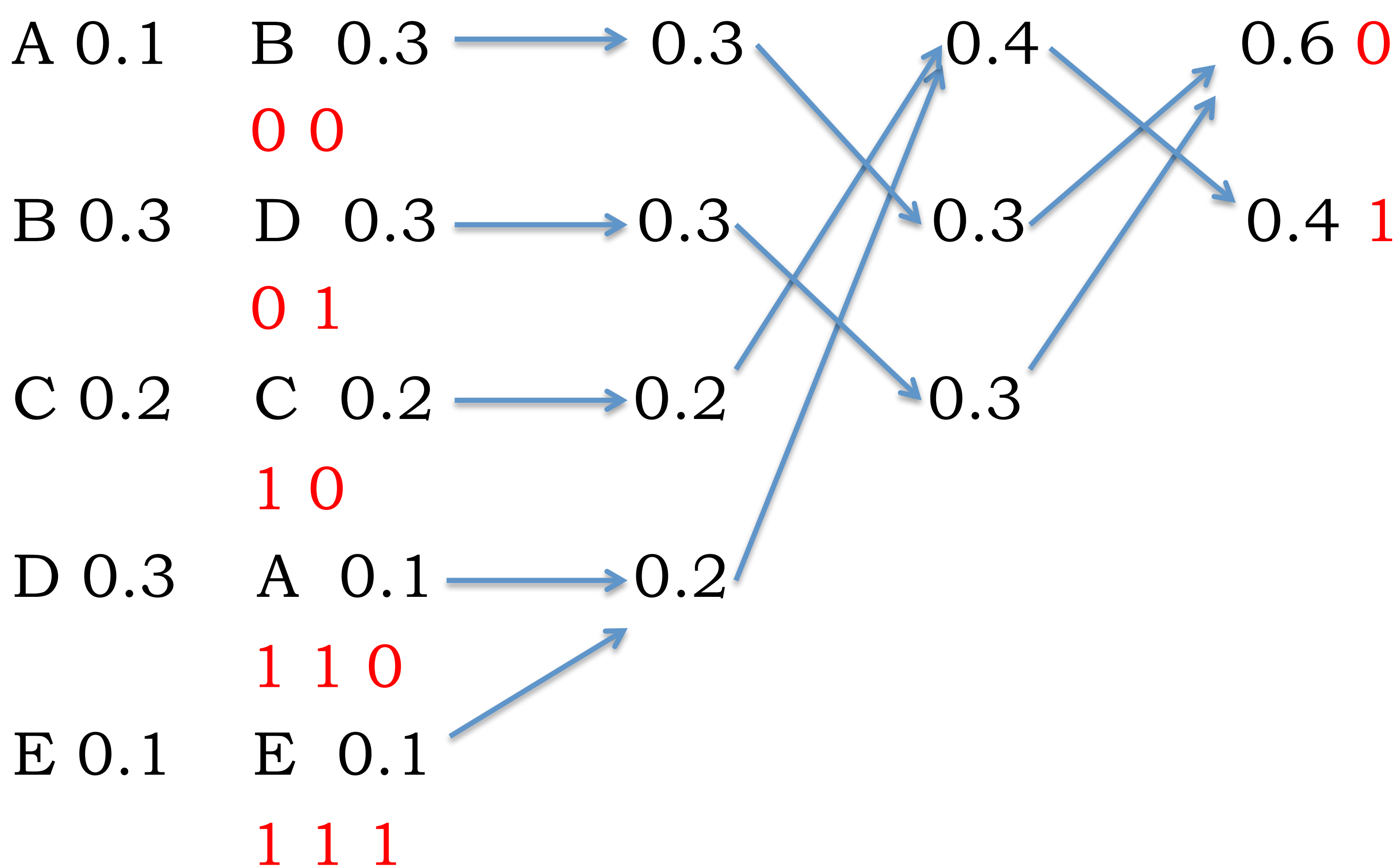
Trace-back


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

Trace-back


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

The Huffmann Code



Example from last lecture

$choice_i$	p_i	$log_2(1/p_i)$	$p_i * log_2(1/p_i)$	*Huffman encoding*	*Expected length*
"A"	1/3	1.58 bits	0.528 bits	10	0.667 bits
"B"	1/2	1 bit	0.5 bits	0	0.5 bits
"C"	1/12	3.58 bits	0.299 bits	110	0.25 bits
"D"	1/12	3.58 bits	0.299 bits	111	0.25 bits
			1.626 bits		1.667 bits

Entropy is 1.626 bits/symbol, expected length of Huffman encoding is 1.667 bits/symbol.

How do we do better?

16 Pairs: 1.646 bits/sym
64 Triples: 1.637 bits/sym
256 Quads: 1.633 bits/sym

Another way to think about Entropy and Coding

- Consider a source S emitting one of 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. This is an iid source --- the emitted symbols are independent and identically distributed
- 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".)
- A small detour to discuss the LLN

The Law of Large Numbers

- The expected or mean number of occurrences of symbol s_1 in K independent repetitions is Kp_1, where p_1 is the probability of getting s_1 in a single trial
- The standard deviation (std) around this mean is

$$\mathrm{sqrt}\{Kp_1(1-p_1)\}$$

- So the fractional one-std spread around around the mean is

$$\mathrm{sqrt}\{(1-p_1)/(Kp_1)\}$$

i.e., goes down as the square root of K.

- Hence for large K, the number of occurrences of s_1 is relatively tightly concentrated around the mean value of Kp_1

Application

- Symbol source = American electorate
  s_1="Obama", s_2="Romney", $p_2 = 1\text{-}p_1$

- Poll K people, and suppose M say "Obama".
  Then reasonable estimate of p_1 is M/K (i.e., we are expecting $M=Kp_1$). For this example, suppose estimate of p_1 is 0.55.
- The fractional one-std uncertainty in this estimate of p_1 is approximately sqrt{0.45*0.55/K} (note: we are looking at concentration around p_1, not Kp_1)
  For 1% uncertainty, we need to poll 2,475 people (not anywhere near 230 million!)

Back to another way to think about Entropy and Coding

- In a very long string of K emissions, we expect to typically get Kp_1, Kp_2, …, Kp_N instances of the symbols s_1, s_2, …, s_N respectively, and all ways of getting these are equally likely
- The probability of any one such typical string is

$$p_1\text{^}(Kp_1).p_2\text{^}(Kp_2)\ldots\ p_N\text{^}(Kp_N)$$

so the number of such strings is approximately $p_1\text{^}(-Kp_1).p_2\text{^}(-Kp_2)\ldots\ p_N\text{^}(-Kp_N)$. Taking the $\log_2$ of this number, we get KH(S).

- So the number of such typical sequences is $2^{KH(S)}$. It takes KH(S) binary digits to count this many sequences, so an average of H(S) binary digits per symbol to code the typical sequences.

Some limitations

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

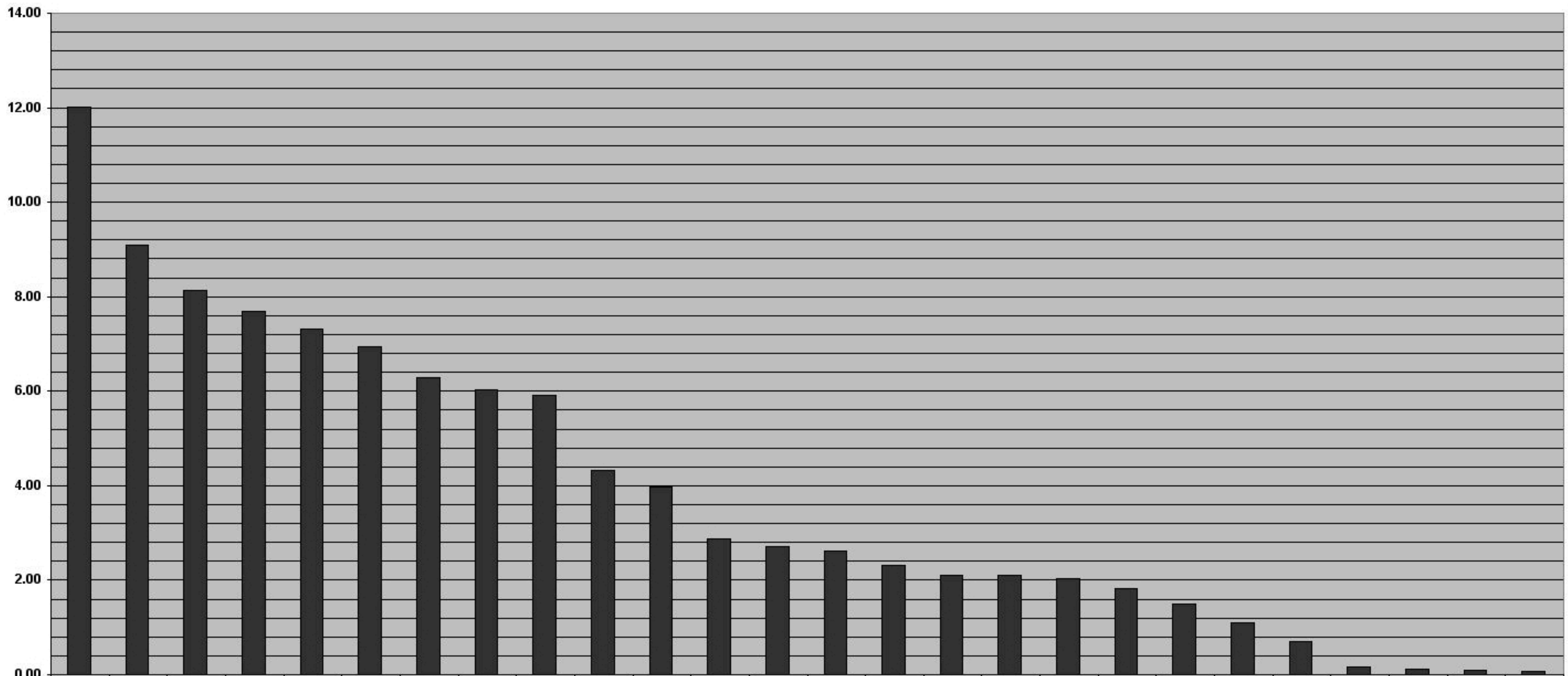
Could still code symbol by symbol, but this won't be efficient at exploiting the redundancy in the text.

Assuming 27 symbols (lower-case letters and space), 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

In fact, 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 expts.
  – More recent estimates: 1-1.5 bits/letter

What exactly is it we want to determine?

- Average per-symbol entropy over long sequences:

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

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

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) 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

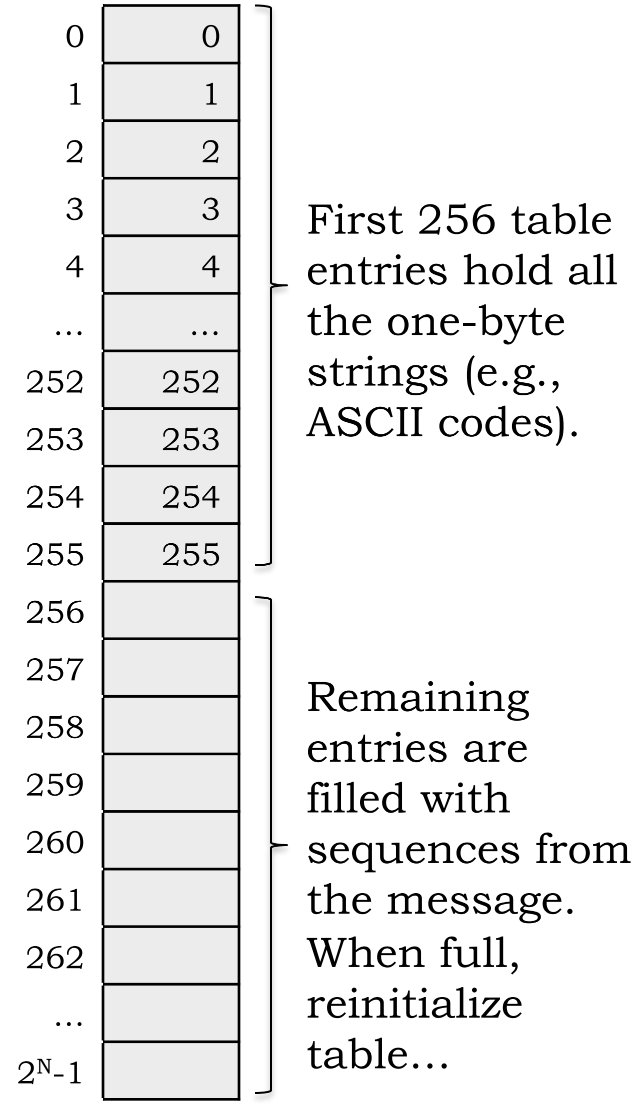
Characteristics of LZW

"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 (lossless)
- 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*!



Try out LZW on

abcabcabcabcabcabcabcabc

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

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

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.

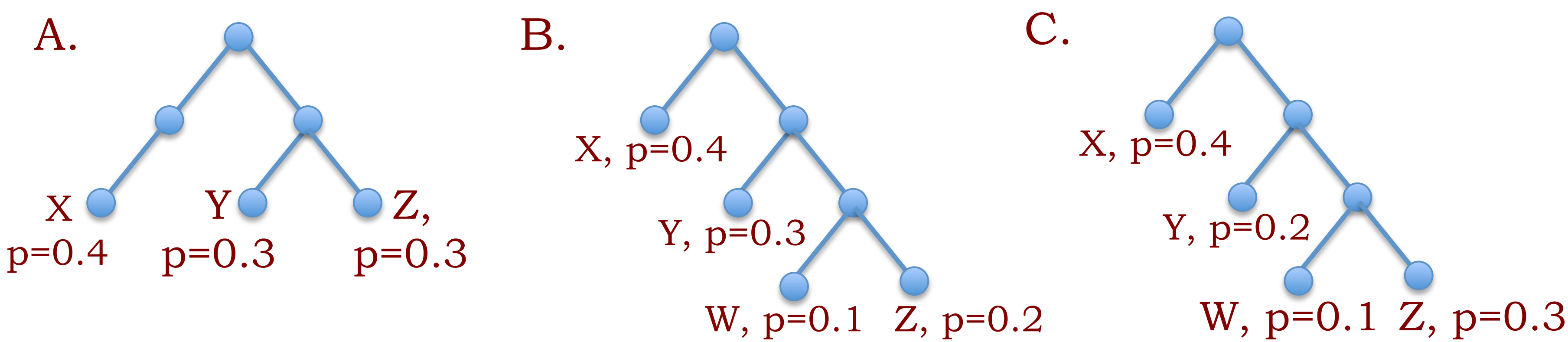
A couple of concluding thoughts

- 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! (Perhaps we'll say a little later in the course.)

Pop Quiz

Which of these (A, B, C) is a valid Huffman code tree?



What is the *expected length* of the code in tree C above?

Sign up on Piazza please, ASAP! Only ¾ of the class has done this so far.

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