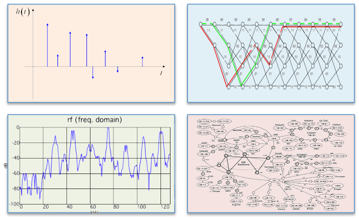


INTRODUCTION TO EECS II

DIGITAL COMMUNICATION SYSTEMS

6.02 Fall 2011
Lecture #2

- Properties and limitations of Huffman codes
- Adaptive variable-length codes: LZW

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?

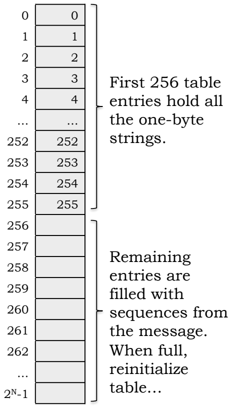
16 Pairs: 1.646 bits/sym
64 Triples: 1.637 bits/sym
256 Quads: 1.633 bits/sym

Huffman Codes - the final word?

- Given static symbol probabilities, the Huffman algorithm creates an optimal encoding when each symbol is encoded separately. (optimal ≡ no other encoding will have a shorter expected message length)
- Huffman codes have the biggest impact on average message length when some symbols are substantially more likely than other symbols.
- You can improve the results by adding encodings for symbol pairs, triples, quads, etc. But the number of possible encodings quickly becomes intractable.
- Symbol probabilities change message-to-message, or even within a single message.
- Can we do adaptive variable-length encoding?

Adaptive Variable-length Codes

- Algorithm first developed by Lempel and Ziv, later improved by Welch. Now commonly referred to as the "LZW Algorithm"
- As message is processed a "string table" is built which 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 based on information in the encoded stream – the table, while central to the encoding and decoding process, is never transmitted!



First 256 table entries hold all the one-byte strings.

Remaining entries are filled with sequences from the message. When full, reinitialize table…

LZW Encoding

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

output the code for STRING
```

1. Accumulate message bytes in S as long as S appears in table.
2. When S+b isn't in table: send code for S, add S+b to table.
3. Reinitialize S with b, 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

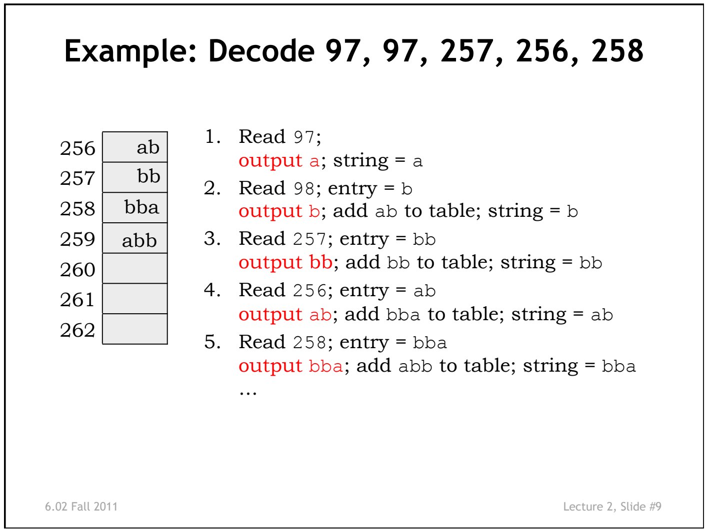
- The encoder algorithm is greedy – it's designed to find the longest possible match in the string table before it makes a transmission.
- The string table is filled with sequences actually found in the message stream. No encodings are wasted on sequences not actually found in the file.
- Note that in 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
output CODE
STRING = CODE

WHILE there are still codes to receive DO
    Read CODE
    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
```

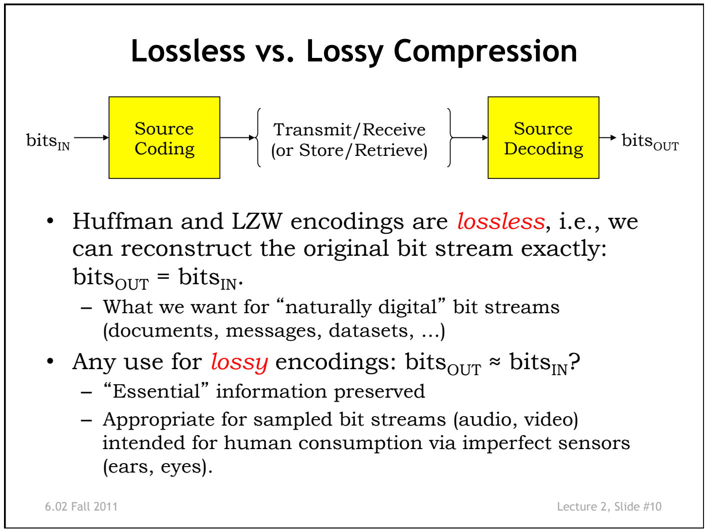
Easy: use table lookup to convert code to message string
Less easy: build table that's identical to that in encoder



Example: Decode 97, 97, 257, 256, 258

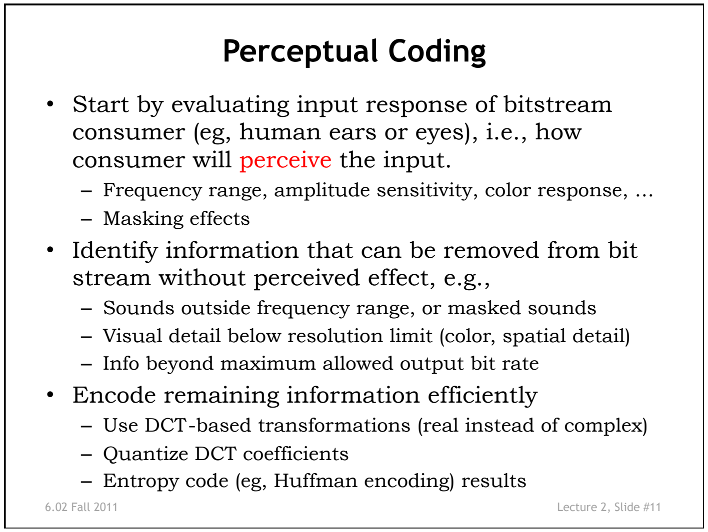
256	ab
257	bb
258	bba
259	abb
260	
261	
262	

1. Read `97`;
   output `a`; string = `a`
2. Read `98`; entry = `b`
   output `b`; add `ab` to table; string = `b`
3. Read `257`; entry = `bb`
   output bb; add `bb` to table; string = `bb`
4. Read `256`; entry = `ab`
   output `ab`; add `bba` to table; string = `ab`
5. Read `258`; entry = `bba`
   output `bba`; add `abb` to table; string = `bba`
   …



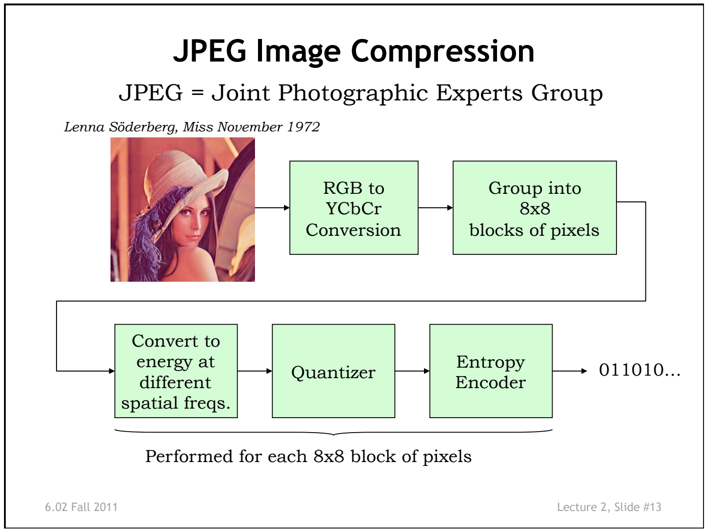
Lossless vs. Lossy Compression

- Huffman and LZW encodings are *lossless*, i.e., we can reconstruct the original bit stream exactly: bits$_{OUT}$ = bits$_{IN}$.
  - What we want for “naturally digital” bit streams (documents, messages, datasets, …)
- Any use for *lossy* encodings: bits$_{OUT}$ ≈ bits$_{IN}$?
  - “Essential” information preserved
  - Appropriate for sampled bit streams (audio, video) intended for human consumption via imperfect sensors (ears, eyes).



Perceptual Coding

- Start by evaluating input response of bitstream consumer (eg, human ears or eyes), i.e., how consumer will perceive the input.
  - Frequency range, amplitude sensitivity, color response, …
  - Masking effects
- Identify information that can be removed from bit stream without perceived effect, e.g.,
  - Sounds outside frequency range, or masked sounds
  - Visual detail below resolution limit (color, spatial detail)
  - Info beyond maximum allowed output bit rate
- Encode remaining information efficiently
  - Use DCT-based transformations (real instead of complex)
  - Quantize DCT coefficients
  - Entropy code (eg, Huffman encoding) results



JPEG Image Compression

JPEG = Joint Photographic Experts Group

Lenna Söderberg, Miss November 1972

Performed for each 8x8 block of pixels