CS/COE 1501

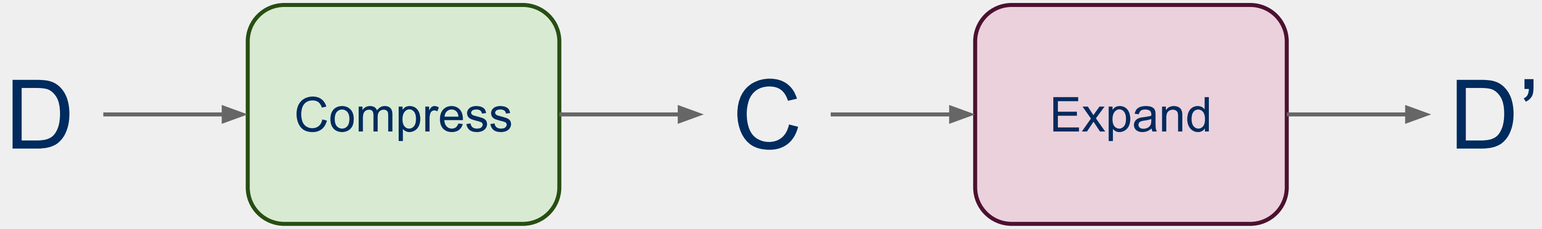
www.cs.pitt.edu/~lipschultz/cs1501/

Compression

What is compression?

- Represent the “same” data using less storage space
  - Can get more use out a disk of a given size
  - Can get more use out of memory
    - E.g., free up memory by compressing inactive sections
      - Faster than paging
      - Built in to OSX Mavericks and later
  - Can reduce the amount data transmitted
    - Faster file transfers
    - Cut power usage on mobile devices
- Two main approaches to compression...

Lossy Compression



- Information is permanently lost in the compression process
- Examples:
  - MP3, H264, JPEG
- With audio/video files this typically isn’t a huge problem as human users might not be able to perceive the difference

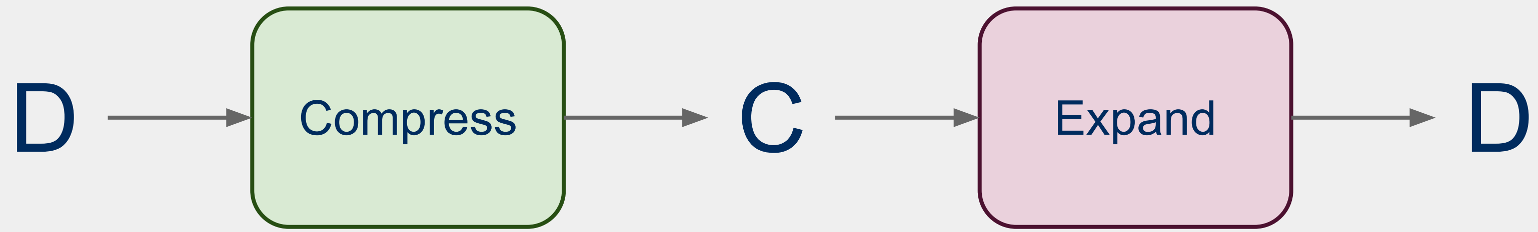
Lossy examples

- MP3
  - "Cuts out" portions of audio that are considered beyond what most people are capable of hearing
- JPEG

40K

28K

Lossless Compression



- Input can be recovered from compressed data exactly
- Examples:
  - zip files, FLAC

Huffman Compression

- Works on arbitrary bit strings, but pretty easily explained using characters
- Consider the ASCII character set
    - Essentially blocks of codes
        - In general, to represent R different characters in a block, you need lg R bits of storage per block
            - Consequently, n bit storage blocks can represent 2^n characters
        - Each 8 bit code block represents one of 256 possible characters in ASCII
        - Easy to encode/decode

Considerations for compressing ASCII

- What if we used *variable length* codewords instead of the constant 8? Could we store the same info in less space?
    - Different characters are represented using codes of different bit lengths
    - If all characters in the alphabet have the same usage frequency, we can't beat block storage
        - On a character by character basis…
    - What about different usage frequencies between characters?
        - In English, R, S, T, L, N, E are used much more than Q or X

Variable length encoding

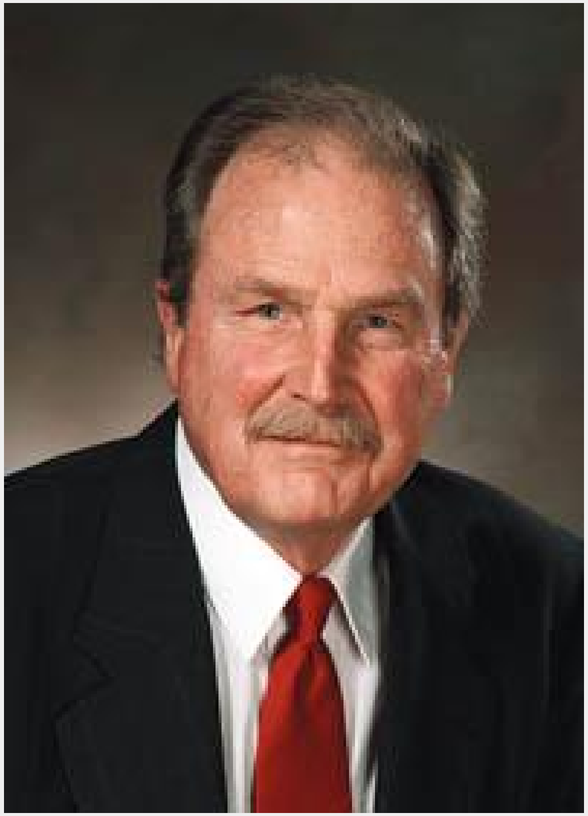
- Decoding was easy for block codes
  - Grab the next 8 bits in the bitstring
  - How can we decode a bitstring that is made of of variable length code words?
  - BAD example of variable length encoding:

1	A
00	T
01	K
001	U
100	R
101	C
10101	N

Variable length encoding for lossless compression

- Codes must be *prefix free*
    - No code can be a prefix of any other in the scheme
    - Using this, we can achieve compression by:
        - Using fewer bits to represent more common characters
        - Using longer codes to represent less common characters

How can we create these prefix-free codes?



Huffman encoding!

Generating Huffman codes

- Assume we have K characters that are used in the file to be compressed and each has a weight (its frequency of use)
- Create a forest, F, of K single-node trees, one for each character, with the single node storing that char’s weight
- while |F| > 1:
  - Select T1, T2 ∈ F that have the smallest weights in F
  - Create a new tree node N whose weight is the sum of T1 and T2’s weights
  - Add T1 and T2 as children (subtrees) of N
  - Remove T1 and T2 from F
  - Add the new tree rooted by N to F
- Build a tree for “ABRACADABRA!”

Implementation concerns

- To encode/decode, we'll need to read in characters and output codes/read in codes and output characters
  - …
  - Sounds like we'll need a symbol table!
    - What implementation would be best?
      - Same for encoding and decoding?
  - Note that this means we need access to the trie to expand a compressed file!

Further implementation concerns

- Need to efficiently be able to select lowest weight trees to merge when constructing the trie
  - Can accomplish this using a *priority queue*
- Need to be able to read/write bitstrings!
  - Unless we pick multiples of 8 bits for our codewords, we will need to read/write fractions of bytes for our codewords
    - We're not actually going to do I/O on fraction of bytes
    - We'll maintain a buffer of bytes and perform bit processing on this buffer
    - See BinaryStdIn.java and BinaryStdOut.java

Binary I/O

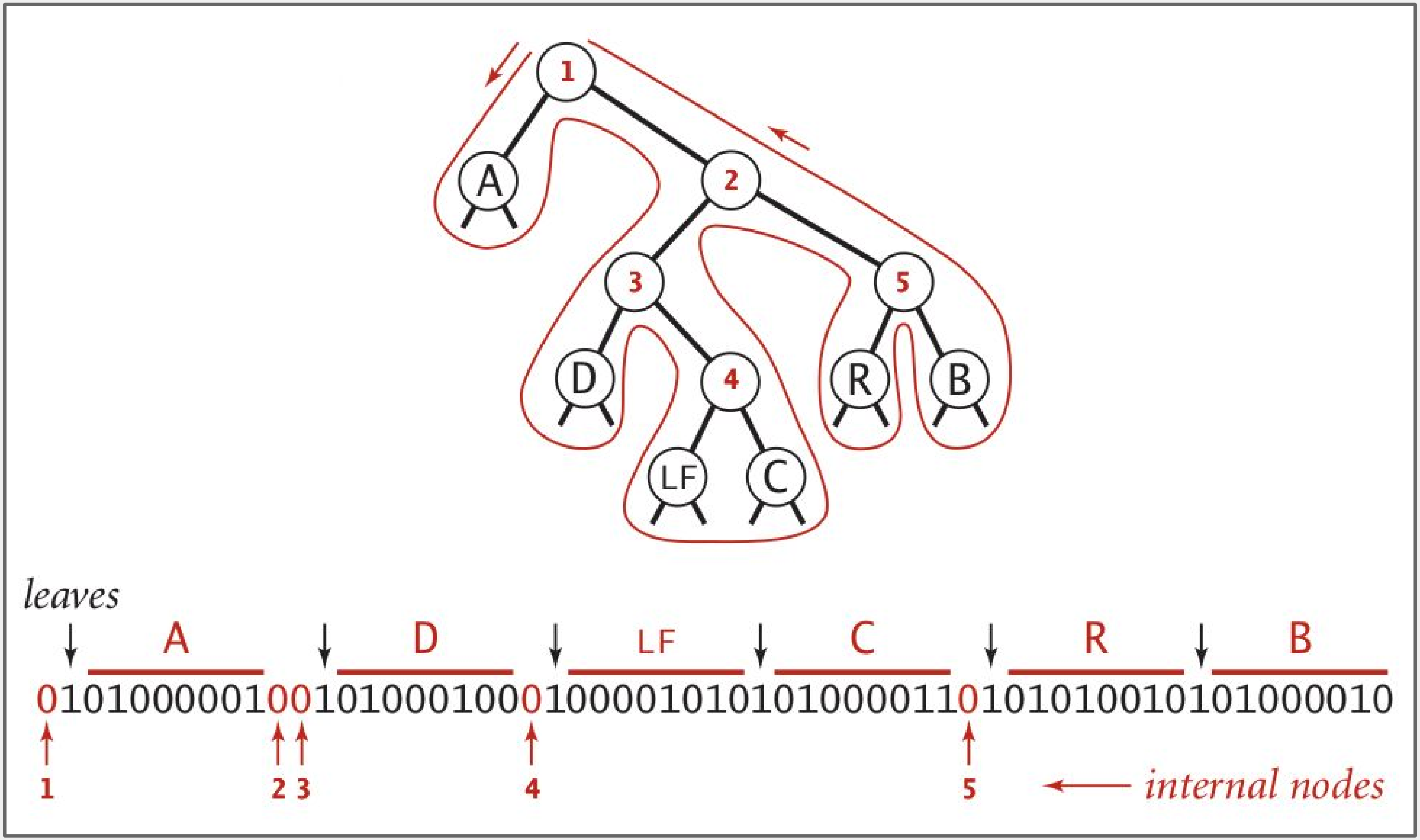
```
private static void writeBit(boolean bit) {
    // add bit to buffer
    buffer <<= 1;
    if (bit) buffer |= 1;
    // if buffer is full (8 bits), write out as a single byte
    N++;
    if (N == 8) clearBuffer();
}
```

```
writeBit(true);
writeBit(false);
writeBit(true);
writeBit(false);
writeBit(false);
writeBit(false);
writeBit(false);
writeBit(true);
```

buffer: `00000000`

N: `0`

Representing tries as bitstrings



Binary I/O

```
private static void writeTrie(Node x){
    if (x.isLeaf()) {
        BinaryStdOut.write(true);
        BinaryStdOut.write(x.ch);
        return;
    }
    BinaryStdOut.write(false);
    writeTrie(x.left);
    writeTrie(x.right);
}
```

```
private static Node readTrie() {
    if (BinaryStdIn.readBoolean())
        return new Node(BinaryStdIn.readChar(), 0, null, null);
    return new Node('\0', 0, readTrie(), readTrie());
}
```

Huffman pseudocode

- Encoding approach:
  - Read input
  - Compute frequencies
  - Build trie/codeword table
  - Write out trie as a bitstring to compressed file
  - Write out character count of input
  - Use table to write out the codeword for each input character
- Decoding approach:
  - Read trie
  - Read character count
  - Use trie to decode bitstring of compressed file

How do we determine character frequencies?

- Option 1: Preprocess the file to be compressed
    - Upside: Ensure that Huffman's algorithm will produce the best output for the given file
    - Downsides:
        - Requires two passes over the input, one to analyze frequencies/build the trie/build the code lookup table, and another to compress the file
        - Trie must be stored with the compressed file, reducing the quality of the compression
            - This especially hurts small files
            - Generally, large files are more amenable to Huffman compression
                - Just because a file is large, however, does not mean that it will compress well!

How do we determine character frequencies?

- Option 2: Use a static trie
  - Analyze multiple sample files, build a single tree that will be used for all compressions/expansions
  - Saves on trie storage overhead…
  - But in general not a very good approach
    - Different character frequency characteristics of different files means that a code set/trie that works well for one file could work very poorly for another
      - Could even cause an increase in file size after “compression”!

How do we determine character frequencies?

- Option 3: Adaptive Huffman coding
    - Single pass over the data to construct the codes and compress a file with no background knowledge of the source distribution
    - Not going to really focus on adaptive Huffman in the class, just pointing out that it exists...

Ok, so how good is Huffman compression

- ASCII requires 8*m bits to store m characters
- For a file containing c different characters
  - Given Huffman codes $\{h_0, h_1, h_2, \ldots, h_{(c-1)}\}$
  - And frequencies $\{f_0, f_1, f_2, \ldots, f_{(c-1)}\}$
  - Sum from 0 to c-1: $|h_i| * f_i$
- Total storage depends on the differences in frequencies
  - The bigger the differences, the better the potential for compression
- Huffman is optimal for character-by-character prefix-free encodings
  - Proof in Propositions T and U of Section 5.5 of the text

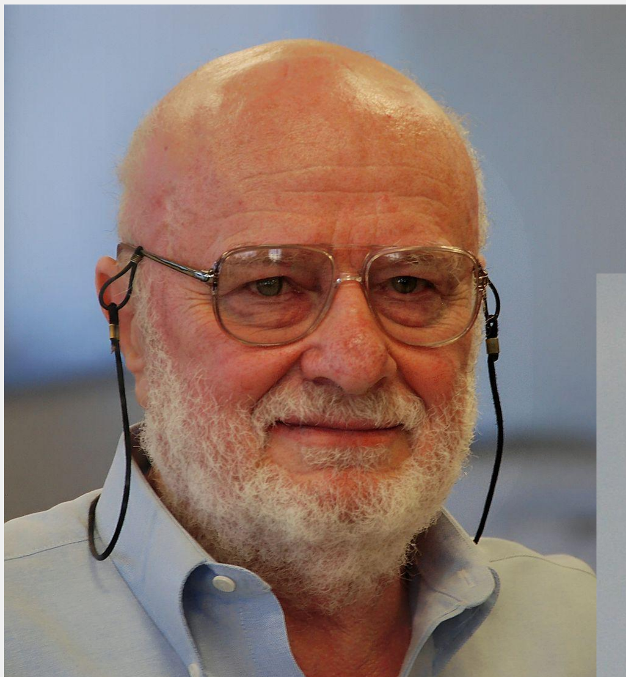
That seems like a bit of a caveat...

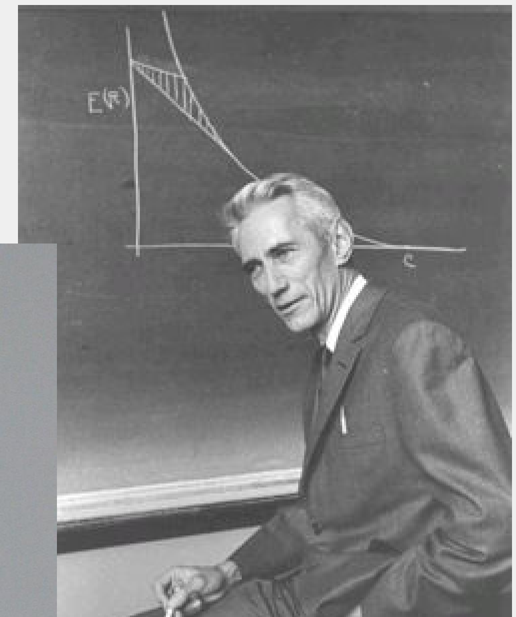
- Where does Huffman fall short?
  - What about repeated patterns of multiple characters?
    - Consider a file containing:
      - 1000 A's
      - 1000 B's
      - …
      - 1000 of every ASCII character
    - Will this compress at all with Huffman encoding?
      - Nope!
    - But it seems like it should be compressible...

Run length encoding

- Could represent the previously mentioned string as:
  - 1000A1000B1000C, etc.
    - Assuming we use 10 bits to represent the number of repeats, and 8 bits to represent the character…
      - 4608 bits needed to store run length encoded file
      - vs. 2048000 bits for input file
      - Huge savings!
- Note that this incredible compression performance is based on a very specific scenario…
  - Run length encoding is not generally effective for most files, as they often lack long runs of repeated characters

What else can we do to compress files?





Patterns are compressible, need a general approach

- Huffman used variable-length codewords to represent fixed-length portions of the input…
    - Let’s try another approach that uses fixed-length codewords to represent variable-length portions of the input
- Idea: the more characters can be represented in a single codeword, the better the compression
    - Consider “the”: 24 bits in ASCII
    - Representing “the” with a single 12 bit codeword cuts the used space in half
        - Similarly, representing longer strings with a 12 bit codeword would mean even better savings!

How do we know that “the” will be in our file?

- Need to avoid the same problems as the use of a static trie for Huffman encoding…
- So use an adaptive algorithm and build up our patterns and codewords as we go through the file

LZW compression

- Initialize codebook to all single characters
  - e.g., character maps to its ASCII value
- While !EOF:
  - Match longest prefix in codebook
  - Output codeword
  - Take this longest prefix, add the next character in the file, and add the result to the dictionary with a new codeword

LZW compression example

- Compress, using 12 bit codewords:
  - TOBEORNOTTOBEORTOBEORNOT

Cur	Output	Add
T	84	TO:256
O	79	OB:257
B	66	BE:258
E	69	EO:259
O	79	OR:260
R	82	RN:261
N	78	NO:262
O	79	OT:263
T	84	TT:264
TO	256	TOB:265
BE	258	BEO:266
OR	260	ORT:267
TOB	265	TOBE:268
EO	259	EOR:269
RN	261	RNO:270
OT	263	--

LZW expansion

- Initialize codebook to all single characters
    - e.g., ASCII value maps to its character
- While !EOF:
    - Read next codeword from file
    - Lookup corresponding pattern in the codebook
    - Output that pattern
    - Add the previous pattern + the first character of the current pattern to the codebook

Note:

- Different from compression
- No codebook addition after first pattern output

LZW expansion example

Won't be represented quite like this in the file...

84, 79, 66, 69, 79, 82, 78, 79, 84, 256, 258, 260, 265, 259, 261, 263

Cur	Output	Add
84	T	--
79	O	256:TO
66	B	257:OB
69	E	258:BE
79	O	259:EO
82	R	260:OR
78	N	261:RN
79	O	262:NO
84	T	263:OT
256	TO	264:TT
258	BE	265:TOB
260	OR	266:BEO
265	TOB	267:ORT
259	EO	268:TOBE
261	RN	269:EOR
263	OT	270:RNO

How does this work out?

- Both compression and expansion construct the same codebook!
  - Compression stores character string → codeword
  - Expansion stores codeword → character string
  - They contain the same pairs in the same order
    - Hence, the codebook doesn't need to be stored with the compressed file, saving space

Just one tiny little issue to sort out...

- Expansion can sometimes be a step ahead of compression…
  - If, during compression, the (pattern, codeword) that was just added to the dictionary is immediately used in the next step, the decompression algorithm will not yet know the codeword.
  - This is easily detected and dealt with, however

LZW corner case example

- Compress, using 12 bit codewords: AAAAAA

Cur	Output	Add
A	65	AA:256
AA	256	AAA:257
AAA	257	--

- Expansion:

Cur	Output	Add
65	A	--
256	AA	AA:256
257	AAA	AAA:257

LZW implementation concerns: codebook

- How to represent/store during:
    - Compression
    - Expansion
- Considerations:
    - What operations are needed?
    - How many of these operations are going to be performed?
- Discuss

Further implementation issues: codeword size

- How long should codewords be?
    - Use fewer bits:
        - Gives better compression earlier on
        - But, leaves fewer codewords available, which will hamper compression later on
    - Use more bits:
        - Delays actual compression until longer patterns are found due to large codeword size
        - More codewords available means that greater compression gains can be made later on in the process

Variable width codewords

- This sounds eerily like variable length codewords…
  - Exactly what we set out to avoid!
- Here, we're talking about a different technique
- Example:
  - Start out using 9 bit codewords
  - When codeword 512 is inserted into the codebook, switch to outputting/grabbing 10 bit codewords
  - When codeword 1024 is inserted into the codebook, switch to outputting/grabbing 11 bit codewords…
  - Etc.

Even further implementation issues: codebook size

- What happens when we run out of codewords?
  - Only 2^n possible codewords for n bit codes
  - Even using variable width codewords, they can't grow arbitrarily large…
- Two primary options:
  - Stop adding new keywords, use the codebook as it stands
    - Maintains long already established patterns
    - But if the file changes, it will not be compressed as effectively
  - Throw out the codebook and start over from single characters
    - Allows new patterns to be compressed
    - Until new patterns are built up, though, compression will be minimal

The showdown you've all been waiting for...

HUFFMAN vs LZW

- In general, LZW will give better compression
    - Also better for compressing archived directories of files
        - Why?
            - Very long patterns can be built up, leading to better compression
            - Different files don't "hurt" each other as they did in Huffman
                - Remember our thoughts on using static tries?

So lossless compression apps use LZW?

- Well, gifs can use it
  - And pdfs
- Most dedicated compression applications use other algorithms:
  - DEFLATE (combination of LZ77 and Huffman)
    - Used by PKZIP and gzip
  - Burrows-Wheeler transforms
    - Used by bzip2
  - LZMA
    - Used by 7-zip
  - brotli
    - Introduced by Google in Sept. 2015
    - Based around a " … combination of a modern variant of the LZ77 algorithm, Huffman coding[,] and 2nd order context modeling … "

DEFLATE et al achieve even better general compression?

- How much can they compress a file?
- Better question:
  - How much can a file be compressed by any algorithm?
- No algorithm can compress every bitstream
  - Assume we have such an algorithm
  - We can use to compress its own output!
  - And we could keep compressing its output until our compressed file is 0 bits!
    - Clearly this can't work
- Proofs in Proposition S of Section 5.5 of the text

Can we reason about how much a file can be compressed?

- Yes! Using Shannon Entropy

Information theory in a single slide...

- Founded by Claude Shannon in his paper “A Mathematical Theory of Communication”
- *Entropy* is a key measure in information theory
    - Slightly different from thermodynamic entropy
    - A measure of the unpredictability of information content
    - By losslessly compressing data, we represent the same information in less space
    - Hence, 8 bits of uncompressed text has less entropy than 8 bits of compressed data

Entropy applied to language:

- Translating a language into binary, the entropy is the average number of bits required to store a letter of the language
- Entropy of a message * length of message = amount of information contained in that message
- On average, a lossless compression scheme cannot compress a message to have more than 1 bit of information per bit of compressed message
- Uncompressed, English has between 0.6 and 1.3 bits of entropy per character of the message