Module 10 Notes

Encoding Basics

Data Storage and Transmission

- Any sort of data: DNA, Text, ...

Reliability

- Very important for: hard drives, SSD, signal transmission, email, etc

Error Correcting Codes

- Store and send some extra info to check if a packet of data arrived okay (or not).
  Example: transmit 7 bit characters: 0110100, 1101100, 0101110.
  Add an 8th parity bit: 1 if char contains odd number of 1s; 0 if even number of 1s.
  $\Rightarrow$ 0110100**1**, 1101100**0**, 0101110**0**

Security/Privacy - large research area (CS 458)

- Public Key Cryptography
- We sent information over public lines (internet) but don't want other to be able to read it.

Runtime of encoding/decoding may not be as important (compared with previous topics).

- Transmission time/cost may be much more expensive
- eg. online video is often very highly compressed
- Sometime we may have the all the data at the start and want to send all at once - can encode the whole thing at once and then decode whole thing at the other end.
- Other times maybe we are streaming, send/encode only part and decode as received.

Compression Ratio

- Typically we are more concerned with size.
- Careful: note the ceilings - round up to whole number of bits
- Often coded language is binary so $\Sigma_C = \{0, 1\}$ where $\log_2 |\Sigma_C| = 1$
- k-bits can represent 2^k different patterns/items $\Rightarrow$ need $\lceil \log n \rceil$ bits to represent n different items

- ASCII - 128 characters ⇒ only 7 bits + 1 parity bit
  Extended ASCII (second 128 are platform specific, not standard) - 256 chars ⇒ 8 bits
  Unicode 16 or 32 bits, etc

Logical - Sound recordings, image compression, etc
Often with domain specific knowledge we can make good lossy compression algorithms where we can't even notice the loss of information. For example, removing frequencies from audio that the human ear can't hear, etc.
⇒ We'll do Physical and Lossless
Think text - it is not okay if you drop a few words or letters in your essay

Character Encodings
Example: $\Sigma_S = \{a, b, \ldots, z, A, B, \ldots, Z, 0, 1, \ldots, 9, +, -, !, \ldots\}$
$\Sigma_C = \{0, 1\}$
Σ^* - Kleene Star - 0 or more items concatenated from the set Σ

How do you decode ASCII?

- Loop up every block of 7 (or 8) bits in a table or dictionary

How do you decode UTF-8?

- look at first bit: if 0, decode as ASCII; if 1, count consecutive 1s until reach 0 to determine number of bytes, ...
- variable length codes
- use less bits to store frequently occurring characters

Why Prefix-free codes?

Lets build an Oracle! You can ask the Oracle any Yes/No question and it will answer correctly but it only responds by encoding the answer. E: 1010
S: 11
O: 1011
Y: 01
N: 0110
Note: all codes are distinct.

Will I pass CS 240?
Answer: 01101011

- Remember there are no breaks or delimiters to separate where one block ends.
- If codes are not prefix-free we may not have a unique decoding.

- Need prefix-free codes; i.e. no code is a prefix of any other code

Huffman Coding

- Build a binary trie. Don't explicitly store labels.
  Left branch: 0
  Right branch: 1
- Trie stores letters from Σ_S so trie operations have runtime $O(\log |\Sigma_S|)$ rather than n.
- Items with lowest frequency are chosen first so they will be pushed further down in the trie $\Rightarrow$ longer codes; high frequencies added last at top of tree $\Rightarrow$ shorter codes.
- How to break ties - same frequency, what goes left or right? Doesn't matter for Huffman $\Rightarrow$ trie is not unique - many equally valid tries.
  Note: we will often specify a convention so there is only 1 trie.
- Output is a sequence of bits, no delimiters or end-of-word markers - wasted space anyway! If you are getting a stream of bits, how would you recognize a delimiting character?
- In our examples, dictionary is built from the input string so must be passed along with the encoded string.
- Decoding is faster than encoding since you are given the decoding trie.
- Typical English test reduced by up to 50% of original.
- Huffman optimal if encoding 1 char at a time and frequencies are known and independent.
- Dictionary created is static - don't know it before but once we create it, it won't change. Variable length codes.

However, some character combinations are frequent; how often does a 'u' follow a 'q'? Maybe better to have a code for "qu" ...

Run-Length Encoding (RLE)

- Used to encode runs of characters.
  0000 is a run of 4 zeroes
  111111 is a run of 6 ones
- Think about a black and white image.
- A binary string is a sequence of runs that alternate between 0s and 1s or vice versa.
- Encoding: Need 1 bit to say which is the first run (0s or 1s), then only need the length of each run (since we know they alternate between 0s and 1s).

- Can't simply give a sequence of numbers - without delimiters, how do we know where one ends and the next begins?
- Elias Gamma Coding to encode k: $\lfloor \log k \rfloor$ zeroes followed by the binary representation of k (which always starts with 1)
  eg: 11 $\Rightarrow \lfloor \log 13 \rfloor = 3$ zeroes followed by 11 in binary: 1101 $\Rightarrow$ 0001101
- Decode, read and count the number 0s before we reach a 1, suppose we counted ℓ zeroes. This tells us how many bits we need to read after the 1. Read and interpret the 1 and the following ℓ bits as a binary number. Repeat for the next run-length.
- Example: 0000110100100 $\Rightarrow$ First bit is a 0 so the first block will be zeroes. Next, there are 3 zeroes before the one so read 3 bits after the 1 $\Rightarrow$ 1101 which is 13. Next, there are 2 zeroes so read 2 bits after the 1 $\Rightarrow$ 100 which is 4.
  Decode string: 0000000000001111
- 2 and 4 both have expansion:
  Source: 11 (run-length of 2) $\Rightarrow$ 010 (source is 2 bits, coded is 3) Source: 1111 (run-length of 4) $\Rightarrow$ 00100 (source is 4 bits, coded is 5)
- Used in the real world: TIFF images

Lempel-Ziv-Welch (LZW)

- Idea: create codes for longer than 1 character patterns that occur within the data. Build the dictionary as you go.
  Remember: anything the encoder does, the decoder must also be able to undo.
- Start with a fixed dictionary; e.g. D_0 = ASCII - industry standard that everyone uses. Adaptive (non-static) dictionary.
  Assigns fixed-length codes to common sequences of more than 1 ASCII characters - 12 bit codes is typical ($2^{12} = 4096$ entries in the dictionary.
  Note: Encoding in strictly ASCII assigns 8 bits to each char, LZW would then assign 12 - you only get savings when you frequently use codes for multiple characters.
- LZ - many variations
- Patterns added to the dictionary are 1 character longer than a code that has been used and is a pattern found in the source; e.g may add "th", then later "the", "them", etc.
- Decoder will start with ASCII and then rebuild.
- Advantages: 1 pass - input/output could be files or a stream can be encoded/decoded on the fly.

For later text we will hopefully have better codes to use; i.e. compress ¿1 char with one 12-bit codes.
Typically better than Huffman.

- Disadvantage: early text of input will not have much compression and will generally be longer than simply using ASCII codes - don't put patterns of ¿1 char in dictionary until later.

Decoder

- Decoder is "one step behind".
  Encoder added to D with the next character, but decoder doesn't know the "next character" until it decodes the next block.

- Suppose at time k we want to use the code we are about to insert.
  Let s_{prev} be string encoded/decoded in the previous step.
  Let s_{curr} be string encoded/decoded in the current step.
  In previous step, encoder added $s_{prev} + c$ to dictionary with code k.
  In current step, we want to use code k, so $s_{curr} = s_{prev} + c$.
  Tricky Part: BUT, c was the first character of what came after $s_{prev} \Rightarrow c = s_{curr}[0] = s_{prev}[0]$.

Decoding: any dictionary implementation will do.
Encoding: need to find longest prefix of S in D, so use a trie - insert and lookup proportional to the length of string read from input.

What if the Dictionary fills up?

- Stop adding - simple and fast but bad if data changes structure (old patterns aren't used as much and no codes for new patterns).

- Clear and start over with only ASCII - temporary poor compression again.

- Discard less frequently used codes - maybe expensive to store counts of usage, then find min values, etc.

- Increase k to more bits - complicated but possible, new codes longer than old codes.

Remember: Decoder must do exactly the same thing as the Encoder.

MTF

- Note: Encoder and Decoder are very similar $\Rightarrow$ they must update the dictionary in the same way.

- What does a run of the same letter in S encode to in C?
  A run of 0s in C.

- What does a run in C mean about the source S? A repeated string; e.g. INEFFICIENCIES $\Rightarrow$ 8 13 6 7 0 3 6 1 3 4 **3 3 3** 18
- Also, expect frequently used characters to be at the front so will have smaller integers.

Burrows-Wheeler Transform

Goal: permute the input so it is more compressable but still reversible.

Example: Input: "the quick brown fox jumped over the lazy dog"
Sorted: "<8 spaces>abcddeeefghhijklmnoooopqrrttuuvwxyz"
Very compressible!
BUT: How to decode? Transformation is not reversible!
BWT will permute input, but is reversible.
Repeated substrings will cause runs of each letter in the substring (the are grouped together in the ordered cyclic shifts).