Data Compression

- important for storage and exchange of data (although less so due to increased storage capabilities/capacity)

Basics

- all data is ultimately represented in binary
- bitstreams and bytestreams
- read a bitstream through a compression algorithm, use the compressed version (i.e. store, transfer, etc), expand back

Reading and writing binary data

- for compatibility, need byte size and ordering compatibility
- reading from stdin may not be byte aligned
- need to be able to read and write binary dumps
- useful to have hex to ASCII table

Limitations

- properties:
  - no algorithm can compress every bitstream
- 1 in 2^500 chance of being able to compress by half a random 1000-bit stream
- undecidability:
  - for random bits, not likely to find a lossless compression algorithm to compree
  - writing as ASCII and reading can achieve a .3 precent compression ratio
- 4 general methods based on:
  - small alphabets
  - long sequences of identical bits/characters
  - frequently used characters
  - long reused bit/character sequences

Genomics

- contain only 4 characters in the alphabet so each can be encoded with just 2 bits per character
- store each as 2 bit repr (i.e. 8 bit to 2 bit), write number of encoded bits to file (32 bits)
- easily reversed

Run-length encoding

- all bit strings are composed of alternating strings of extended repeated bits

- how many bits are used to store counts
- how to handle repeated strings that are longer than the max count?
- how to handle repeated strings that are shorter than number of bits needed to store length?
- general choices:
    - counts between 0 and 255, encoded with 8 bits
    - runs less than 256 are handled with runs of 0 length
    - encode short runs, even though doing so might lengthen output
- bitmaps:
    - used to represent images and documents
    - table of 0s and 1s
- compress:
    - read a bit
    - if different from last
        - write current count
        - reset count
    - if same
        - if count is max
            - write count
            - write 0 count
            - reset count
    - increment count

Huffman compression

- use less bits to encode frequently used characters and more bits for less frequently used characters
- either need a delimiter or ensure no character code is the prefix of another (prefix-free)
- the general method for finding the optimal prefix-free code is Huffman coding (1952)
- Huffman coding:
    - view bitstream as a bystestream
    - use a prefix-free code for the characters as follow:
        - build an encoding trie
        - write the trie (as a bitstream) for use in expansion
        - use the trie to encode the bytestream as a bitstream

    - for expansion:
        - read the trie (encoded at the beginning)
        - use the trie to decode the bitstream
    - general compression process:
        - read input
        - tabulate frequency of occurrence of each char value in the input
        - build the Huffman encoding trie corresponding to those frequencies
        - build the corresponding codeword table to associate a bitstring with each char
        - write the trie, encoded as a bitstring
        - write the count of characters at the input, encoded as a bitstring
        - use the codeword table to write the codeword for each input character
    - general expansion process:
        - read the trie (encoded at the beginning of a bitstream)
        - read the count of characters to be decoded
        - use the trie to decode the bitstream

LZW compression (Lempel-Ziv-Welch)

- read input as stream of 7 bit characters and write output as 8-bit bytes
- associates string keys with fixed length codeword values
- initialize symbol table with 128 possible single character string keys with 8-bit codewords
- compression:
    - find the longest string, s, that is a prefix of the unscanned input
    - write the 8-bit value associated with s
    - scan one character past s in the input
    - associate the next codeword with s + c (c appended to s) in the symbol table, where c is the next character in the input
- expansion:
    - write the current string val
    - read a codeword x from the input
    - set s to the value associated with x in the symbol table
    - associate the next unassigned codeword value to val + c in the symbol table, where c is the first character of s
    - set the current val to s
- use a trie (TST) for the symbol table

- trie representation:
  - left = 0
  - right = 1
  - stores char
  - store freq
- use the search location of the char to build the prefix-free code for the char
- use a min priority queue to build the trie
- properties:
  - for any prefix-free code, the length of the encoded bitstring is equal to the weighted external path length of the corresponding trie
  - given a set of r symbols and frequencies, the Huffman algorithm builds an optimal prefix-free code

Lempel-Ziv Code

- many different variations of Lempel-Ziv
  - Lempel–Ziv–Markov chain algorithm
  - Lempel–Ziv–Oberhumer
  - Lempel–Ziv–Stac
  - Lempel–Ziv–Storer–Szymanski
  - Lempel–Ziv–Welch
  - LZ77
  - LZ78
  - LZW
  - LZS
  - LZRW1
  - LZRW1-A
  - LZRW2
  - LZRW3
  - LZRW3-A
  - LZRW4
  - LZRW5
  - LZ4
  - LZJB
  - LZWL
  - LZX
  - liblzg/LZG

- LZ78
  - construct dictionary on the fly from input string
  - as new substrings are encountered when moving through the string, add it to the dictionary with an id (a number, in binary)
  - modify mapped substrings in relationship to previously mapped substrings
  - encode string into binary based on above ids

Additional

- https://en.wikipedia.org/wiki/Category:Lossless_compression_algorithms
- https://en.wikipedia.org/wiki/Category:Lossy_compression_algorithms