4 Data Compression

- Goals:
  - save storage space
  - save time to transmit data
  - save bandwidth

- Fact: Most of all text is uncompressible
- Proof:
  - There are $(|\Sigma|^{m+1}-1)/(|\Sigma|-1)$ strings of length at most m
  - There are $|\Sigma|^n$ strings of length n
  - From these strings at most $(|\Sigma|^{m+1}-1)/(|\Sigma|-1)$ strings can be compressed
  - This is fraction of at most $|\Sigma|^{m-n+1}/(|\Sigma|-1)$
  - E.g. for $|\Sigma| = 256$ and m=n-10 this is 8.3×10^{-25} which implies that only 8.3×10^{-25} of all files of n bytes can be compressed to a string of length n-10

But: Most data we want to compress is not random, it has some regularities that can be exploited
  - letters have different frequencies (e.g., in General English Plain text From *Cryptographical Mathematics*, by Robert Edward Lewand):
    - e: 12%, t: 10%, a: 8%, i: 7%, n: 7%, ... , k: 0.4%, x: 0. 2%, j: 0. 2%, q: 0. 09%, z:0. 06%
  - Special characters like $,%,# occur even less frequent
  - Text underlies a lot of rules
    - Words are (usually) the same (collected in dictionaries)
    - Not all words can be used in combination
    - Sentences are structured (grammar)
    - Program codes use code words
    - Digitally encoded pictures have smooth areas, where colors change gradually
    - Patterns repeat

Issues to consider

- What data is compressed (binary, ASCII, pictures, music, video, ...)
- Should it be possible to uncompress compressed prefixes (i.e., on the fly decompression)?
- Is efficient compression important (e.g. chat) or only efficient decompression (e.g., DVD)
- Is it necessary to exactly reconstruct the data after compression (lossless compression)?
  - often not necessary, e.g. because small variations in images/video/audio can hardly be noticed, see JPEG, MPEG-2, MP3

Runlength encoding

- Very simple idea
- Replace multiple occurrences of a symbol by only one symbol, and a number specifying the number or replications

Example for binary strings:

Text: 000000000000000000000011111111111100000 (37 bit)

Encoded: 20 12 5 = 10100 01100 00101 (15 bit)

But:

Text: 00110000010 (11 bit)

Encoded: 2 2 5 1 1 = 00010 00010 00101 00001 00001 (25 bit)

-> Only efficient if runs are long, and alphabet is small (unlikely for text, but often useful for b/w images).

Huffman Code

- Variable-length character encoding: Assigns short codes to frequent characters, longer codes for rare characters
- Prefix-free: no code is prefix of another code

Example:

Symbol	a	b	c	d	e
Frequency	0.1	0.15	0.3	0.16	0.29
Code	000	001	10	01	11

- Optimal for symbol-by-symbol encoding with known input symbol frequencies
- Requires two passes through the document for encoding (or standard encoding is used based, e.g., on language)
- It is necessary to store and transmit coding table
- Can only encode fixed-length strings (usually characters)
- Efficient for large files with non-uniform distribution of symbols

Lempel-Ziv

- Family of compression techniques proposed by Lempel and Ziv in 1977/1978
- Lossless
- Creates a dictionary with variable-length entries, based on already encoded prefix
- Only one run through document necessary
- No need to send dictionary, as decoder can generate dictionary on the fly while decoding
- Used in, e.g., gzip

- Basic idea:
  - start with empty dictionary (or single characters)
  - break input file into pieces
  - for each piece:
    - if already in dictionary, ouput corresponding encoding
    - if not in dictionary, output uncompressed, add piece to dictionary

- Complexity of compression and decompression O(n)

More concrete LZW (Lempel-Ziv-Welch, 1984):

- Initialize the dictionary with all the symbols in the alphabet
- Start with empty string L
- REPEAT
  - Read next input symbol x
  - While Lx can be found in the dictionary:
    - L <- Lx
    - Read next symbol x
  - Add Lx to dictionary at the next dictionary position
  - Output dictionary entry for L
  - L=x
- UNTIL text is compressed

Example

- which witch wished this wish? (29x8=232 bits)

x	Lx	Output	Dictionary
w	w		
h	wh	w	257
i	hi	h	258
c	ic	i	259
h	ch	c	260
_	h_	h	261
w	_w	_	262
i	wi	w	263
t	it	i	264
c	tc	t	265
h	ch		
_	ch_	260	266
w	_w		
i	_wi	262	267

x	Lx	Output	Dictionary
s	is	i	268
h	sh	s	269
e	he	h	270
d	ed	e	271
_	d_	d	272
t	_t	_	273
h	th	t	274
i	hi		
s	his	258	275
_	s_	s	276
w	_w		
i	_wi		
s	_wis	267	277
h	sh	269	

Output: which_wit<260><262>ished_t<258>s<267><269> (22x9=198 bits)

Decoding

- L1=D(first symbol)
- output D(x)
- WHILE text not fully decoded DO
  - read next symbol x
  - L2=D(x)
  - output L2
  - IF (L1L2[0]) does not exist in the dictionary
    - add L1L2[0] as next available entry in dictionary
  - L1 <- L2
- DONE

Example

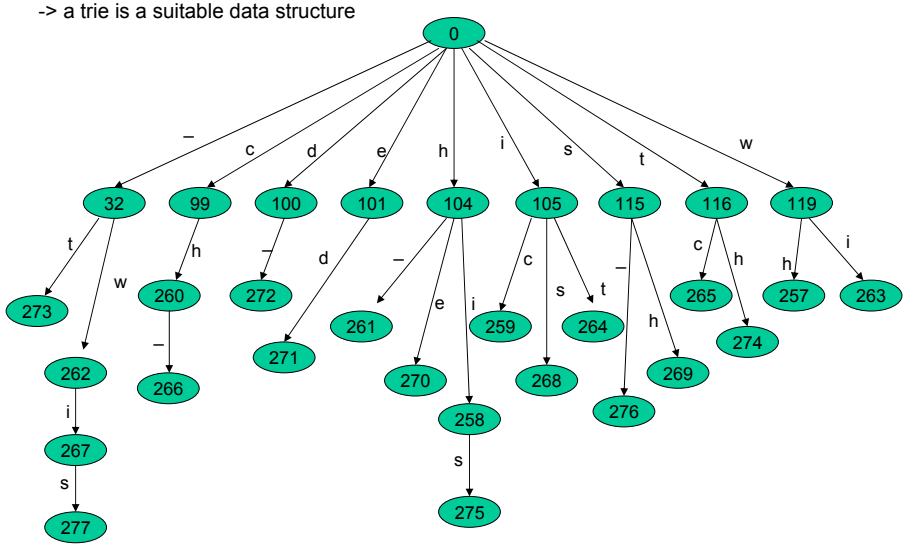
which_wit<260><262>
ished_t<258>s<267><269>

L1	x	L2=D(x)	Output	Dictionary
w			w	
w	h	h	h	wh=257
h	i	i	i	hi=258
i	c	c	c	ic=259
c	h	h	h	ch=260
h	_	_	_	h_=261
_	w	w	w	_w=262
w	i	i	i	wi=263
i	t	t	t	it=264
t	260	ch	ch	tc=265
ch	262	_w	_w	ch_=266
_w	i	i	i	_wi=267
i	s	s	s	is=268
s	h	h	h	sh=269
h	e	e	e	he=270
e	d	d	d	ed=271
d	_	_	_	d_=272
_	t	t	t	_t=273
t	258	hi	hi	th=274
hi	s	s	s	his=275
s	267	_wi	_wi	s_=276
_wi	269	sh	sh	_wis=277

How to store the dictionary

- For every string in the dictionary, also all prefixes are stored

-> a trie is a suitable data structure



Further Comments

- Trie is very broad (a node can have |∑| children)
- Efficient implementation necessary, e.g., with hashing functions
- There is a whole family of algorithms related to the ideas of Lempel and Ziv

Summary

- Only structured data can be compressed
- Main classes of lossless compression:
  1. Statistical methods (e.g., Huffman)
     good for large data with known characteristics
  2. Dictionary methods (e.g., LZW)
     good for arbitrary data
- Several types of data can be compressed much better by allowing (small) losses in decoded data