COMP261 Lecture 20

Marcus Frean

Data Compression 2: Lempel-Ziv Coding

Victoria
UNIVERSITY OF WELLINGTON
Te Whare Wānanga
o te Ūpoko o te Ika a Māui
CAPITAL CITY UNIVERSITY

some string-searching addenda…

- Knuth-Morris-Pratt visualization (also Huffman, many others): https://people.ok.ubc.ca/ylucet/DS/Algorithms.html

- If you're interested in Boyer-Moore: https://dwnusbaum.github.io/boyer-moore-demo/

- The "Moore" in Boyer-Moore has a nice interactive demos of both Knuth-Morris-Pratt and Boyer-Moore algorithms: http://www.cs.utexas.edu/users/moore/best-ideas/string-searching/

some Huffman coding addenda…

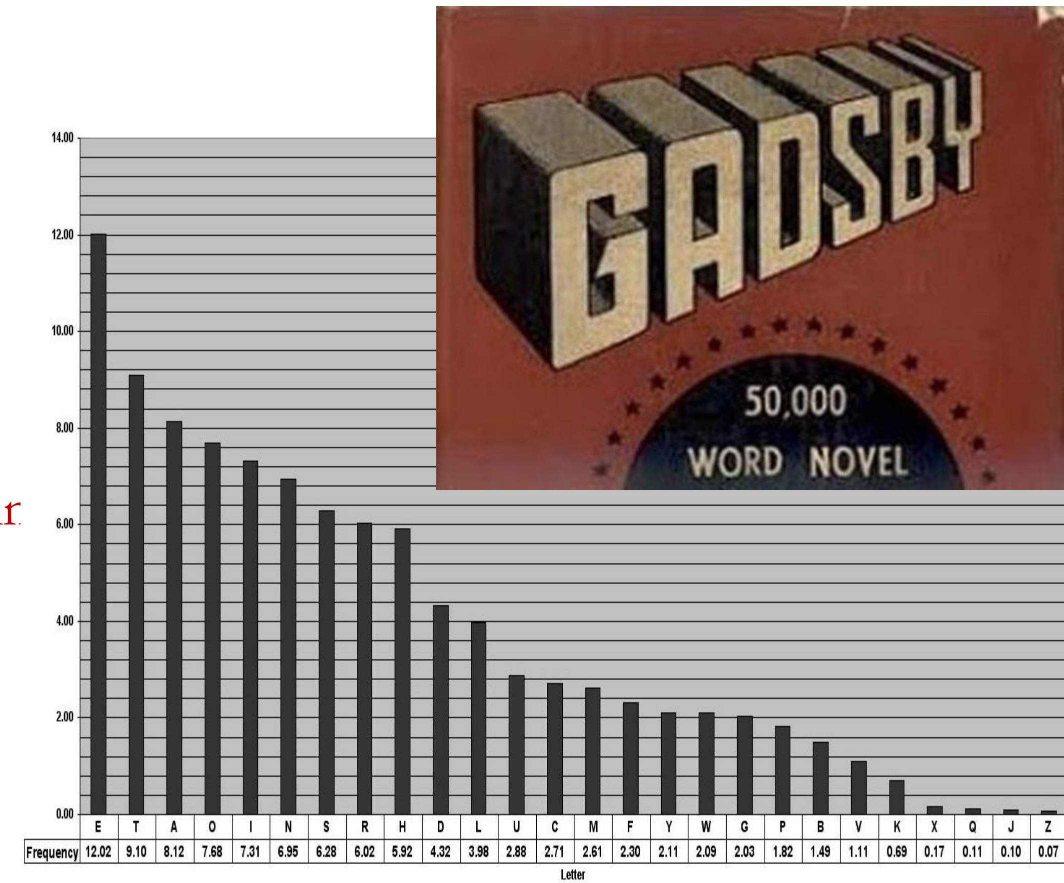
- http://www.csfieldguide.org.nz/en/interactives/huffman-tree/index.html
- https://people.ok.ubc.ca/ylucet/DS/Huffman.html 

for fun: each verse has [a-z] except e (& check out “lipogram, Gadsby”)

Bold Nassan quits his caravan,
A hazy mountain grot to scan;
Climbs jaggy rocks to find his way,
Doth tax his sight, but far doth stray.

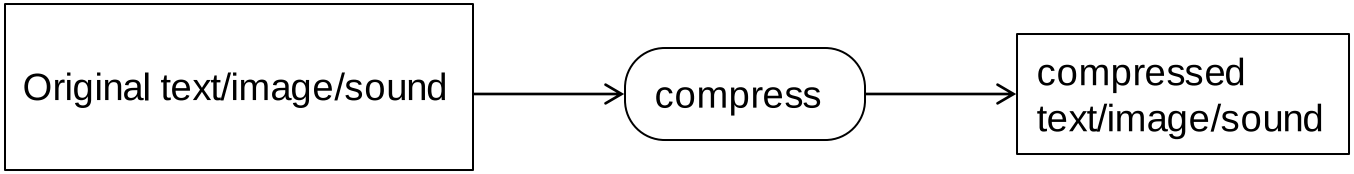
Not work of man, nor sport of child
Finds Nassan on this mazy wild;
Lax grow his joints, limbs toil in vain—
Poor wight! why didst thou quit that plain

Vainly for succour Nassan calls;
Know, Zillah, that thy Nassan falls;
But prowling wolf and fox may joy
To quarry on thy Arab boy.



Data/Text Compression

- Reducing the memory required to store some information.



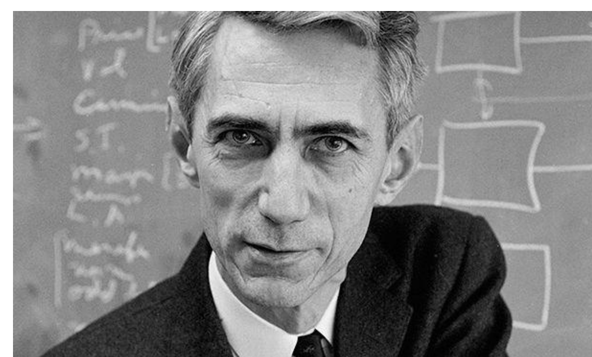
- Huffman coding minimised the number of bits <u>for each symbol</u>.

- Perhaps we could do better by looking at <u>sequences of symbols</u>?

Shannon, and information theory

- Claude Shannon
- "…kept a box on his desk called the "Ultimate Machine". Otherwise featureless, the box possessed a single switch on its side. When the switch was flipped, the lid of the box opened and a mechanical hand reached out, flipped off the switch, then retracted back inside the box."

"This duality can be pursued further and is related to a duality between past and future and the notions of control and knowledge. **Thus we may have knowledge of the past but cannot control it; we may control the future but have no knowledge of it.**"



predictable → shorter codes

- Shannon's source coding theorem: the *optimal* code length for a symbol is $-\log_2 P$, where P is the probability of the input symbol. The average of this, over the whole alphabet, is called the *entropy, H*.
  - If P was "flat" (all letters equally likely), *H*=4.75 bits, for English
  - With the actual P, it drops a bit to 4.2 bits/char. You can try it here.
  - but that ignores the fact it's a *sequence*.

Fr xmpl, y cn prbbly gss wht ths sntnc sys, vn wth ll f th vwls mssng. Tht ndcts tht th nfrmtn cntnt cn b xtrctd frm th rmnng smbls.

Aoccdrnig to rscheearch at Cmabirgde Uinervtisy, it deosn't mttaer in waht oredr the ltteers in a wrod are, the olny iprmoetnt tihng is taht the frist and lsat ltteer be at the rghit pclae. The rset can be a ttoal mses and you can sitll raed it wouthit porbelm. Tihs is bcuseae the huamn mnid deos not raed ervey lteter by istlef, but the wrod as a wlohe. Amzanig huh?

predictable → shorter codes

- Digrams and trigrams? In English the most common are:

Digrams	Trigrams
EN	ENT
RE	ION
ER	AND
NT	ING
TH	IVE
ON	TIO
IN	FOR
TR	OUR
AN	THI
OR	ONE

- Entropy if you use trigrams drops to about 2.6 bits/char
- So what's the entropy if you go to *n*-grams, and let *n* get "big"?
  - it ends up somewhere between 0.6 and 1.3 bits/char !!

the "Shannon limit"

Run Length Encoding

- If data contains lots of runs of repeated symbols, it may be efficient to store as (count, symbol) pairs.

- E.g. #1:
  could use two bytes for each character:
  1 byte for the count (up to 256), and 1 byte for the character
  aaabbaaaaaapaa → 3a2b6a1p2a

  Poor for: abcabcabbcabcabaabcabcccababcabc…

- E.g. #2:
  could use 6 bits to store black and white image data:
  5 bits for the count, and 1 bit to say what is repeated
  1111111100000011111111111111 → 010001 001100 011011

  Poor for: 01011101110010101010101010101….

Lempel-Ziv

- Lossless compression.
- LZ77 = simple compression, using repeated patterns
  - basis for many later, more sophisticated compression schemes.

- Key idea:
  - If you find a repeated pattern, replace later ones by *a link to the first*:

`a contrived text containing riveting contrast`

a contrived text*[15,5]* ain*[2,2]* g *[22,4]* t*[9,4][35,5]* ast

(Note: This ignores patterns of length 1 – they are included later.)

Lempel-Ziv

How can we distinguish pointers from ordinary characters?

Store text as triples:

- `[offset,length,symbol]` where symbol is just the next symbol.
- so if there's no repetition to reference: just `[0,0,symbol]`

To limit size of `offset` and `length`, we:

- limit the size of the window to left of current position in which we look for a match, and
- limit the distance ahead we look in the input for a match.

Lempel-Ziv Example

```
a_contrived_text_containing_riveting_contrasting …
```

```
[0,0,a] [0,0,_] [0,0,c] [0,0,o] [0,0,n] [0,0,t]
[0,0,r] [0,0,i] [0,0,v] [0,0,e] [0,0,d]
      → → → a _ c o n t r i v e d
```

no repeats, so all at just [0,0,symbol] so far

```
[10,1,t]   → _ t
[4,1,x]    → e x
[3,1,_]    → t _
[15,4,a]   → cont a
[15,1,n]   → i n
[2,2,g]    → in g
[11,1,r]   → _ r
[22,3,t]   → ive t
[9,4,c]    → ing_ c
[35,4,a]   → ontr a
```

notice that including matches of length 1 changes the encoding

This takes 69 bytes to store 48 characters - assuming that offset, length and character each take one byte.
Would improve with longer text.

Lempel-Ziv 77

- **skljsadf lkjhwep oury d dmsmesjkh fjdhfjdfjdpppdjkhf sdjkh fjdhfjds fjksdh kjjjfiuiwe dsd fdsf sdsa**

- Outputs a string of tuples:
  - [offset, length, nextCharacter]   or [0,0,character]
- Moves a cursor through the text one character at a time
  - cursor points at the next character to be encoded.
- Drags a "sliding window" behind the cursor.
  - searches for matches only in this sliding window
- Expands a lookahead buffer from the cursor
  - this is the string it tries to match in the sliding window.
- Searches for a match for the longest possible lookahead
  - stops expanding when there isn't a match
- Insert triple of match point, length, and next character

Lempel-Ziv 77 – high level

```
cursor ← 0;  windowSize ← 100  // some suitable size
while cursor < text.length-1:
     look for longest prefix of text[cursor .. text.length-1]
              in text[max(cursor-windowSize,0) .. cursor-1]
     if found, add [offset,length,text[cursor+length]] to output
     else add [0, 0, text[cursor]] to output
     advance cursor by length+1
```

We can use various approaches to find that longest-matching-substring:

- Brute force: Look for longest match at each position in window
- KMP, or Boyer Moore…

Lempel-Ziv 77 – coding, a first attempt

```
cursor ← 0
windowSize ← 100 // some suitable size
while cursor < text.size
    length ← 0
    prevMatch ← 0
    loop
        match ← stringMatch( text[cursor.. cursor+length],
          text[(cursor<windowSize)?0:cursor-windowSize .. cursor-1])
        if match succeeded then:
              prevMatch ← match
            length ← length + 1
        else:
            output( [a value for prevMatch, length, text[cursor+length ]])
            cursor ← cursor + length + 1
            break
```

However:
(cursor – windowSize) should never point before 0,

and (cursor + length) mustn't go past end of text

- This looks for an occurrence of text[cursor..cursor+length] in text[start..cursor-1], for increasing values of length, until none is found, then outputs a triple.
- This is pretty wasteful – we know there is no match before prevMatch, so there's no point looking there again! Probably better starting from prevMatch?
- Or (maybe) find longest match starting at each position in window and record longest?

Decompression

a_contrived_text_containing_riveting_contrasting_t

➔

[0,0,a][0,0,_][0,0,c][0,0,o][0,0,n][0,0,t][0,0,r][0,0,i][0,0,v][0,0,e][0,0,d][10,1,t]
[4,1,x][3,1,_][15,4,a][15,1,n][2,2,g][11,1,r][22,3,t][9,4,c][35,4,a][0,0,s][12,5,t]

- so we can just decode each tuple in turn:

```
cursor ← 0
for each tuple
   if [0, 0, ch ]  :   output[cursor++] ← ch
   elif [offset, length, ch ]  :
      for j = 0 to length-1
         output [cursor++]  ←  output[cursor-offset ]
      output[cursor++]  ←  ch
```

Lempel Ziv – note that…

- Encoding is expensive, decoding is cheap
- Many improvements/variants have been proposed
  - See Wikipedia and other online summaries
- e.g.: could use two types of output value:
  - (offset, length) pair for repeated sequence,
  - character for non-repeat
  - How can we distinguish them?
- Can be used in conjunction with Huffman coding.