15-583:Algorithms in the Real World

Data Compression III

- Lempel-Ziv algorithms
- Burrows-Wheeler
- Introduction to Lossy Compression

Lempel-Ziv Algorithms

LZ77 (Sliding Window)

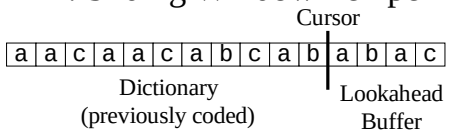
- Variants: LZSS (Lempel-Ziv-Storer-Szymanski)
- Applications: `gzip`, Squeeze, LHA, PKZIP, ZOO

LZ78 (Dictionary Based)

- Variants: LZW (Lempel-Ziv-Welch), LZC (Lempel-Ziv-Compress)
- Applications: `compress`, GIF, CCITT (modems), ARC, PAK

Traditionally LZ77 was better but slower, but the gzip version is almost as fast as any LZ78.

LZ77: Sliding Window Lempel-Ziv

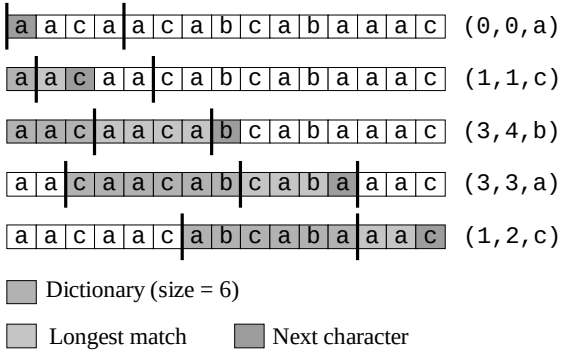


Dictionary and buffer "windows" are fixed length and slide with the cursor

On each step:

- Output (p,l,c)
  p = relative position of the longest match in the dictionary
  l = length of longest match
  c = next char in buffer beyond longest match
- Advance window by l + 1

LZ77: Example



LZ77 Decoding

Decoder keeps same dictionary window as encoder.

- For each message it looks it up in the dictionary and inserts a copy

What if l > p? (only part of the message is in the dictionary.)

- E.g. dict = `abcd`, codeword = `(2,9,e)`
- Simply copy starting at the cursor

```
for (i = 0; i < length; i++)
  out[cursor+i] = out[cursor-offset+i]
```

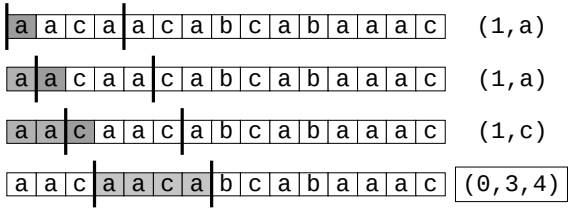
- Out = `abcdcdcdcdcdc`

LZ77 Optimizations used by `gzip`

LZSS: Output one of the following formats

`(0, position, length)` or `(1,char)`

Typically use the second format if length < 3.



Optimizations used by `gzip` (cont.)

- Huffman code the positions, lengths and chars
- Non greedy: possibly use shorter match so that next match is better
- Use hash table to store dictionary.
  - Hash is based on strings of length 3.
  - Find the longest match within the correct hash bucket.
  - Limit on length of search.
  - Store within bucket in order of position

Theory behind LZ77

Sliding Window LZ is Asymptotically Optimal [Wyner-Ziv,94]

Will compress long enough strings to the source entropy as the window size goes to infinity.

$$H_n = \sum_{X \in A^n} p(X) \log \frac{1}{p(X)}$$

$$H = \lim_{n \to \infty} H_n$$

Uses logarithmic code for position.

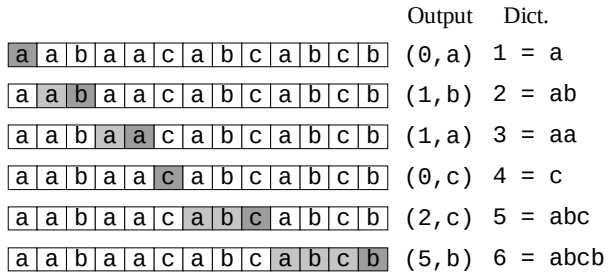
Problem: “long enough” is really **really** long.

LZ78: Dictionary Lempel-Ziv

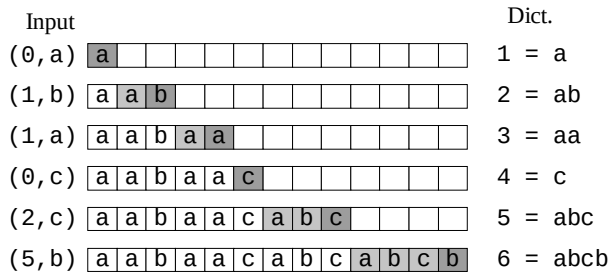
Basic algorithm:

- Keep dictionary of words with integer *id* for each entry (e.g. keep it as a trie).
- Coding loop
  - find the longest match *S* in the dictionary
  - Output the entry *id* of the match and the next character past the match from the input *(id,c)*
  - Add the string *Sc* to the dictionary
- Decoding keeps same dictionary and looks up *id*s

LZ78: Coding Example



LZ78: Decoding Example



LZW (Lempel-Ziv-Welch) [Welch84]

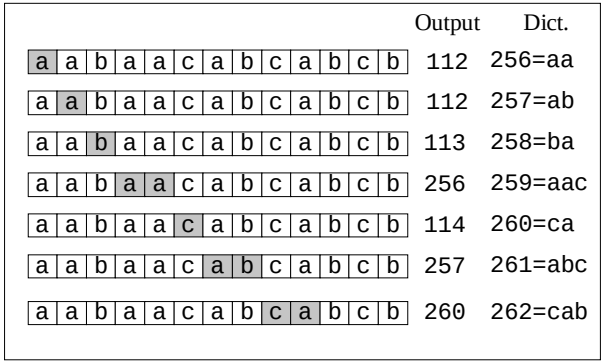
Don't send extra character *c*, but still add *Sc* to the dictionary.

The dictionary is initialized with byte values being the first 256 entries (e.g. `a = 112`, ascii), otherwise there is no way to start it up.

The decoder is one step behind the coder since it does not know c

- Only an issue for strings of the form *SSc* where *S[0] = c*, and these are handled specially

LZW: Encoding Example

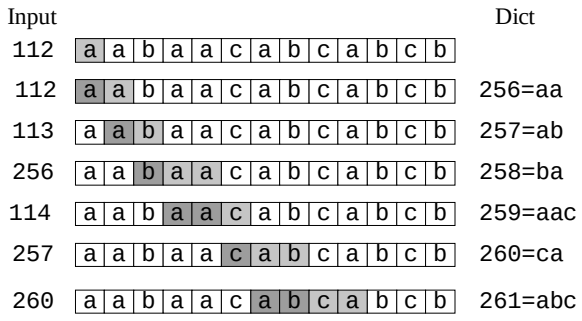


LZ78 and LZW issues

What happens when the dictionary gets too large?

- Throw the dictionary away when it reaches a certain size (used in GIF)
- Throw the dictionary away when it is no longer effective at compressing (used in unix `compress`)
- Throw the least-recently-used (LRU) entry away when it reaches a certain size (used in BTLZ, the British Telecom standard)

LZW: Decoding Example



Lempel-Ziv Algorithms Summary

Both LZ77 and LZ78 and their variants keep a "dictionary" of recent strings that have been seen.

The differences are:

- How the dictionary is stored
- How it is extended
- How it is indexed
- How elements are removed

Lempel-Ziv Algorithms Summary (II)

Adapt well to changes in the file (e.g. a Tar file with many file types within it).

Initial algorithms did not use probability coding and perform very poorly in terms of compression (e.g. 4.5 bits/char for English)

More modern versions (e.g. gzip) do use probability coding as "second pass" and compress much better.

Are becoming outdated, but ideas are used in many of the newer algorithms.

Burrows -Wheeler

Currently best "balanced" algorithm for text

Basic Idea:

- Sort the characters by their full context (typically done in blocks). This is called the **block sorting transform**.
- Use move-to-front encoding to encode the sorted characters.

The ingenious observation is that the decoder only needs the sorted characters and a pointer to the first character of the original sequence.

Burrows Wheeler: Example

Let's encode: $\mathbf{d_1e_2c_3o_4d_5e_6}$

We've numbered the characters to distinguish them.

Context "wraps" around.

Context	Char
$ecode_6$	d_1
$coded_1$	e_2
$odede_2$	c_3
$dedec_3$	o_4
$edeco_4$	d_5
$decod_5$	e_6

Sort Context →

Context	Output
$dedec_3$	o_4
$coded_1$	e_2
$decod_5$	e_6
$odede_2$	c_3
$ecode_6$	d_1 ⇐
$edeco_4$	d_5

Burrows-Wheeler (Continued)

Theorem: *After sorting, equal valued characters appear in the same order in the output as in the most significant position of the context.*

Proof: Since the chars have equal value in the most-sig-position of the context, they will be ordered by the rest of the context, *i.e.* the previous chars. This is also the order of the ouput since it is sorted by the previous characters.

Context	Output
$dedec_3$	o_4
$\mathbf{code}d_1$	e_2
$\mathbf{deco}d_5$	e_6
$odede_2$	c_3
$\mathbf{ecode}_6$	d_1
$\mathbf{edeco}_4$	d_5

Burrows-Wheeler Decode

Function BW_Decode(In, Start, n)
  S = MoveToFrontDecode(In,n)
  R = Rank(S)
  j = Start
  for *i=1* **to** *n* **do**
    Out[*i*] = S[*j*]
    j = *R*[*i*]

Rank gives position of each char in sorted order.

Decode Example

S	Sort(*S*)	Rank(*S*)		Out
o_4	c_3	6	e_6	d_1 ⇐
e_2	d_1	4	d_1	e_2
e_6	d_5	5	e_2	c_3
c_3	e_2	1	c_3	o_4
d_1	e_6	2 ⇐	o_4	d_5
d_5	o_4	3	d_5	e_6

ACB (Associate Coder of Buyanovsky)

Currently the second best compression for text
  (BOA, based on PPM, is marginally better)

Keep dictionary sorted by context
- Find best match for contex
- Find best match for contents
- Code
  - Distance between matches
  - Length of contents match

Has aspects of Burrows-Wheeler, residual coding, and LZ77

Context	Contents
dec	ode
d	ecode
decod	e
de	code
	decode
deco	de

Other Ideas?

More Ideas?