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
- **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**

**Cursor**

```
  a | a | c | a | c | a | b | c | a | b | a | c
```

**Dictionary** (previously coded)

```
  Buffer
```

**Lookahead**

```
  Cursor
```

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

**Repeat:**

Output $(p, l, c)$ where

- $p =$ position of the longest match that starts in the dictionary (relative to the cursor)
- $l =$ length of longest match
- $c =$ next char in buffer beyond longest match

Advance window by $l + 1$

---

**LZ77: Example**

```
  a | a | c | a | c | a | b | c | a | b | a | c
```

$(1, 0, a)$

```
  a | c | a | c | a | b | c | a | b | a | a | c
```

$(1, 1, c)$

```
  a | c | a | c | a | b | c | a | b | a | a | c
```

$(3, 4, b)$

```
  a | a | c | a | c | a | b | c | a | b | a | a | c
```

$(3, 3, a)$

```
  a | a | c | a | c | a | b | c | a | b | a | a | c
```

$(1, 2, c)$

**Dictionary (size = 6)**

**Longest match**

**Buffer (size = 4)**

**Next character**
**LZ77 Decoding**

Decoder keeps same dictionary window as encoder. For each message it looks it up in the dictionary and inserts a copy at the end of the string.

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

E.g. dict = abcd, codeword = (2, 9, e)

- Simply copy from left to right
  
  ```
  for (i = 0; i < length; i++)
      out[cursor+i] = out[cursor-offset+i]
  ```

- Out = abcdcdcdcdcdce

**LZ77 Optimizations used by gzip**

LZSS: Output one of the following two formats

- \((0, \text{position}, \text{length})\)
- \((1, \text{char})\)

Uses the second format if length < 3.

- \((0, 3, 4)\)
- \((1, a)\)
- \((1, a)\)
- \((1, c)\)
- \((0, 3, 4)\)

**Optimizations used by gzip (cont.)**

1. Huffman code the positions, lengths and chars
2. Non greedy: possibly use shorter match so that next match is better
3. Use a hash table to store the dictionary.
   - Hash keys are all strings of length 3 in the dictionary window.
   - Find the longest match within the correct hash bucket.
   - Puts a limit on the length of the search within a bucket.
   - Within each bucket store in order of position

**The Hash Table**

... 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 ...

... a a c a a c a b c a b a a a c ...

... a a c 9 c a b 13 a c a 13
... a a c 10 c a b 12 c a a 9
... a a c 7 a c a 8
Using Suffix Trees
Recall that at each stage, we output a pair \((p_i, l_i)\) where 
\(S[p_i .. p_i + l_i] = S[i .. i + l_i]\)
Find all pairs \((p_i, l_i)\) in linear time

Construct a suffix tree for \(S\)
Label each internal node with the minimum of labels of all
leaves below it - this is the first place in \(S\) where it
occurs. Call this label \(c_v\).
For every \(i\), search for the string \(S[i .. m]\) stopping just
before \(c_v, i\). This gives us \(l_i\) and \(p_i\).

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.
\[
H_n = \sum_{X \in \mathcal{A}^n} p(X) \log \frac{1}{p(X)} \\
H = \lim_{n \to \infty} H_n
\]
Uses logarithmic code (e.g. gamma) for the position.
Problem: “long enough” is really really long.

Lempel-Ziv Algorithms Summary
Adapts 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
performed poorly in terms of compression. More
modern versions (e.g. gzip) do use probability
coding as “second pass” and compress much better.
The algorithms are becoming outdated, but ideas are
used in many of the newer algorithms.

Burrows-Wheeler
Currently near best “balanced” algorithm for text:
(Archive Comparison Test, compressia is BW)
Breaks file into fixed-size blocks and encodes each
block separately.
For each block:
- Sort each character by its full context.
  This is called the block sorting transform.
- Use move-to-front transform 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: d₁e₂c₃o₄d₅e₆
We’ve numbered the characters to distinguish them.
Context “wraps” around. Last char is most significant.

<table>
<thead>
<tr>
<th>Context</th>
<th>Char</th>
</tr>
</thead>
<tbody>
<tr>
<td>ecode₆</td>
<td>d₁</td>
</tr>
<tr>
<td>coded₁</td>
<td>e₂</td>
</tr>
<tr>
<td>odede₂</td>
<td>c₃</td>
</tr>
<tr>
<td>dedec₃</td>
<td>o₄</td>
</tr>
<tr>
<td>edeco₄</td>
<td>d₅</td>
</tr>
<tr>
<td>decod₅</td>
<td>e₆</td>
</tr>
</tbody>
</table>

Context Char  Sorted Context
ecode₆ d₁  dedec₃ o₄
coded₁ e₂  coded₁ e₂
dodec₂ c₃  decod₅ e₆
dedec₃ o₄  edeco₄ d₅
edeco₄ d₅  ecode₆ d₁

denced₃ o₄  code e₂
coded₁ e₂  deco d₅
dedec₃ o₄  edeco₄ d₅

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 sketch: Since the chars have equal value in the most-significant-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 output since it is sorted by the previous characters.

<table>
<thead>
<tr>
<th>Context</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>c₃</td>
<td>o₄</td>
</tr>
<tr>
<td>d₁</td>
<td>e₂</td>
</tr>
<tr>
<td>d₅</td>
<td>e₆</td>
</tr>
<tr>
<td>e₂</td>
<td>c₃</td>
</tr>
<tr>
<td>e₆</td>
<td>d₁</td>
</tr>
<tr>
<td>o₄</td>
<td>d₅</td>
</tr>
</tbody>
</table>

Burrows-Wheeler: Decoding

Consider dropping all but the last character of the context.
- What follows the underlined a?
- What follows the underlined b?
- What is the whole string?

Answer: b, a, abacab

<table>
<thead>
<tr>
<th>Context</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>a c</td>
<td></td>
</tr>
<tr>
<td>a b</td>
<td></td>
</tr>
<tr>
<td>b a</td>
<td></td>
</tr>
<tr>
<td>b a</td>
<td></td>
</tr>
</tbody>
</table>

Burrows-Wheeler: Decoding

What about now?

Answer: cabbaa

Can also use the “rank”. The “rank” is the position of a character if it were sorted using a stable sort.

<table>
<thead>
<tr>
<th>Context</th>
<th>Output Rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>a c</td>
<td>6</td>
</tr>
<tr>
<td>a a</td>
<td>1</td>
</tr>
<tr>
<td>a b</td>
<td>4</td>
</tr>
<tr>
<td>b b</td>
<td>5</td>
</tr>
<tr>
<td>b a</td>
<td>2</td>
</tr>
<tr>
<td>c a</td>
<td>3</td>
</tr>
</tbody>
</table>
**Burrows-Wheeler Decode**

Function \( BW\_Decode(In, \text{Start}, n) \)

\[ S = \text{MoveToFrontDecode}(In, n) \]

\[ R = \text{Rank}(S) \]

\[ j = \text{Start} \]

for \( i := 1 \) to \( n \) do

\[ \text{Out}[i] = S[j] \]

\[ j = R[j] \]

Rank gives position of each char in sorted order.

**Decode Example**

<table>
<thead>
<tr>
<th>( S )</th>
<th>( \text{Rank}(S) )</th>
<th>( \text{Out} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( o_4 )</td>
<td>( 6 )</td>
<td>( e_6 ) (&lt;) ( d_1 )</td>
</tr>
<tr>
<td>( e_2 )</td>
<td>( 4 )</td>
<td>( d_1 ) (&lt;) ( e_2 )</td>
</tr>
<tr>
<td>( e_6 )</td>
<td>( 5 )</td>
<td>( e_2 ) (&lt;) ( c_3 )</td>
</tr>
<tr>
<td>( c_3 )</td>
<td>( 1 )</td>
<td>( c_3 ) (&lt;) ( o_4 )</td>
</tr>
<tr>
<td>( d_1 )</td>
<td>( 2 )</td>
<td>( o_4 ) (&lt;) ( d_5 )</td>
</tr>
<tr>
<td>( d_5 )</td>
<td>( 3 )</td>
<td>( d_5 ) (&lt;) ( e_6 )</td>
</tr>
</tbody>
</table>

**BW Transform**

Need to sort based on all prefixes:

Naïve : \( O(m \times n \log n) \)

How can we sort much faster?