Lempel-Ziv Coding

- Dictionary methods
- Ziv-Lempel 77
- The gzip variant of Ziv-Lempel 77
- Ziv-Lempel 78
- The LZW variant of Ziv-Lempel 78
- Asymptotic optimality of Ziv-Lempel
Dictionary methods

• Replace a substring in the file with a codeword that identifies the substring in a dictionary (or codebook).

• Static dictionary. One first builds a suitable dictionary, which will be used for all compression tasks. Examples: digram coding, where some of most frequently occurring pairs of letters are stored in the dictionary.

  Example: A reasonable small dictionary: 128 ASCII individual characters, followed by 128 pairs (properly selected out of the possible $2^{14}$ pairs) of ASCII symbols. In clear an ASCII character needs 7 bits. With the above dictionary, the favorable cases encodes by digrams (4 bits/character) while the unfavorable cases, coding a single character needs 8 bits/character instead of 7 bits/character).

• The dictionary may be enlarged, by adding longer words (phrases) to it (e.g. and, the). Unfortunately using a dictionary with long phrases will make it well adapted and efficient for a certain type of texts, but very inefficient for other texts (compare the dictionaries suitable for a mathematical textbook or for a collection of parliamentary speeches).
- **Semi-static dictionaries**: one can build a dictionary well suited for a text, and first send the dictionary as side information, and afterwards sending the text, encoded with the optimal dictionary. This has two drawbacks: (a) the overhead of side information may be very high, for short texts, and (b) we need to pass two times through the text (read two times a large file).

- **Adaptive dictionary** is elegant and simple. The dictionary is built on the fly (or it need not to be built at all, it exists only implicitly) using the text seen so far. Advantages: (a) there is only one pass through the text, (b) the dictionary is changing all the time, following the specificity of the recently seen text.

  **A substring of a text is replaced by a pointer to where it has occurred previously.**

- Almost all dictionary methods are variations of two methods, developed by Jacob Ziv and Abraham Lempel in 1977 and 1978, respectively. In both methods, the same principle is used: the dictionary is essentially all or part of the text seen before (prior to the current position), the codewords specify two types of information: a) pointers to previous positions and b) the length of the text to be copied from the past.

- The variants of Ziv-Lempel coding differ in how pointers are represented and in the limitations they impose on what is referred to by pointers.
A (cartoon-like) example of encoding with an adaptive dictionary is given in the image below. The decoder has to figure out what to put in each empty box, by following the arrow, and taking the amount of text suggested by the size of the box.

```
Pease porridge hot, Pease porridge cold, Pease porridge in a pot Nine days old. Some like it hot, Some like it cold, Some like it in a pot Nine days old.
```
Pease porridge hot,
Pease porridge cold,
Pease porridge in a pot
Nine days old.

Some like it hot,
Some like it cold,
Some like it in a pot
Nine days old.
The LZ77 family of adaptive dictionary coders

• The algorithm was devised such that decoding is fast and the memory requirements are low (the compression ratio was sacrificed in favor of low complexity).

• Any string of characters is first transformed into a strings of triplets, with the following significance
  – The first component of a triplet says how far back to look in the previous text to find the next phrase.
  – The second component records how long the phrase is. The first and second components form a pointer to a phrase in the past text.
  – The third component gives the character which will follow the next phrase. This is absolutely necessary if there is no phrase match in the past. It is included in every triplet for uniformity of decoding.

• We start with a decoding example. Suppose the encoded bitstream contains the triplets
  \[< 0, 0, a >, < 0, 0, b >, < 2, 1, a >, < 3, 2, b >, < 5, 3, b >, < 1, 10, a >\]

• When the triplet \(< 5, 3, b >\) is received, the previous decoded text is \(abaabab\). The pointer \(< 5, 3, >\) tells to copy the past phrase \(aab\) after \(abaabab\). The character \(< , , b >\) tells to append a \(b\) after \(abaababaab\)
• When the triplet \(<1, 10, a>\) is received, it tells to copy 10 characters starting with the last available character. This is a recursive reference, but fortunately it can be solved easily. We find that the 10 characters are in fact \(bbbbbbbbb\). Thus recursive references are similar to run-length coding (to be discussed in a later course).

• In LZ77 there are limitations on how far back a pointer can refer and the maximum size of the string referred to. Usually the window for search is limited to a few thousand characters. Example: with 13 bits one can address 8192 previous positions (several book pages). The length of the phrase is limited to about 16 characters. Longer pointers are expensive in bits, without a significant improvement of the compression. If the length of the phrase is 0 the position is not relevant.

• The decoder is very simple and fast, because each character decoded requires only a table look-up (the size of the array is usually smaller than the cache size). The decoding program is sometimes included with the data at very little cost, such that a compressed file can be downloaded from the network without any software. When executed, the program generates the original file.
Example of LZ77 compression

<table>
<thead>
<tr>
<th>encoder output</th>
<th>&lt; 0, 0, a &gt;</th>
<th>&lt; 0, 0, b &gt;</th>
<th>&lt; 2, 1, a &gt;</th>
<th>&lt; 3, 2, b &gt;</th>
<th>&lt; 5, 3, b &gt;</th>
<th>&lt; 1, 10, a &gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>decoder output</td>
<td>a</td>
<td>b</td>
<td>aa</td>
<td>bab</td>
<td>aabb</td>
<td>bbbbbbbbbba</td>
</tr>
</tbody>
</table>
Encoding procedure

Goal: Given the text S[1...N], and the window length W.

Produce a stream of triplets \(< f, \ell, c >\) position-length-next character (the binary codes for \(f, \ell, c\) are discussed latter).

1. Set \(p \leftarrow 1\). /* (S(p) is the next character to be encoded) */

2. While \((p \leq N)\) /* while we did not reach the end of the text */

2.1 Search for the longest match for \(S[p, p+1, \ldots]\) in \(S[p-W, \ldots, p-1]\). Denote \(m\) the position and \(\ell\) the length of the match \(S[m, m+1, \ldots, m+\ell-1] \equiv S[p, p+1, \ldots, p+\ell-1]\).

2.2 Write in the output stream the triplet \(<\text{position}, \text{length}, \text{next character}>\), i.e. \(<p-m, \ell, S[p+\ell]>\)

2.3 Set \(p \leftarrow p + \ell + 1\). /*Continue encoding from \(S[p + \ell + 1]\)*/
Decoding procedure

0 Given a stream of triplets \(<f, \ell, c>\) (the binary codes for \(f, \ell, c\) are discussed latter).

1 Set \(p \leftarrow 1\).  /* (\(S(p)\) is the next character to be decoded) */

2 While there are non-decoded triplets \(<f, \ell, c>\)  /* while we did not reach the end of the text */

   2.1 Read the triplet \(<f, \ell, c>\).

   2.2 Set \(S[p, p+1, \ldots, p+\ell-1] = S[p-f, p-f+1, \ldots, p-f+\ell-1]\).

   2.3 Set \(S[p+\ell] \leftarrow c\).

   2.4 Set \(p \leftarrow p+\ell+1\).  /*Continue encoding from \(S[p+\ell+1]\)*/
The gzip variant of LZ77

• Distributed by Gnu Free Software Foundation (author Gailly, 1993)

• Gzip uses a simple technique to speed up at the encoder the search for the best match in the past.
  – The next three characters are used as addresses in a look up table, which contains a linked list showing where the next three characters have occurred in the past. The length of the list is restricted in size, by a parameter selected by the user before starting the encoding. If there are long runs of the same characters, limiting the size of the list helps removing the unhelpful references in the list.
  – Recent occurrences are stored at the beginning of the list.

• Binary encoding of the triplets <position,length,next character>
  – In gzip the encoding is done slightly differently than ”classical” LZ77: instead of sending all the time the triplet <position,length,character>, gzip sends either a pair <length,position>, when a match is found, or it sends <character>, when no match was found in the past.
  – Therefore a previous match is represented by a pointer consisting in ”position” and ”length”. The ”position” is Huffman coded such that more frequent ”positions” (usually recent ones) are encoded using fewer bits than older ”positions”.
– The match length and the next character are encoded with a single Huffman code (more efficient than separately Huffman encoding the length and the character and adding an extra bit to signal that what follows is length or character).

– The Huffman codes are generated semi-statically: blocks of up to 64Kbytes from the input file are processed at a time. The canonical Huffman codes are generated for the pointers and raw characters, and a code table is placed at the beginning of the compressed form of the block. The program does not need to read twice the file (64 Kbytes can be kept in memory).

• With its fast list search method and compact Huffman representation of pointers and characters on Huffman codes, gzip is faster and compresses better than other Ziv-Lempel methods. However, faster versions exist, but their compression ratio is smaller.
The LZ78 family of adaptive dictionary coders

- In LZ77 pointers can refer to any substring in the window of previous text. This may be inefficient, since the same substring may appear many times in the window, and we spare multiple codewords for the same substring.

- In LZ78 only some substrings can be referenced, but now there is no window restriction in the previous text.

- The encoded stream consists of pairs \(<\text{index, character}>\), where \(<\text{index, >}\) points in a table to the longest substring matching the current one, and \(<\text{character}>\) is the character following the matched substring.

- Example

  - We want to encode the string \(abaababaa\)
  - The encoder goes along the string and creates a table where it dynamically adds new entries. When encoding a new part of the string, the encoder searches the existing table to find a match for the new part, and if there are many such matches, it selects the longest one. Then it will add to the encoded stream the address in the table of the longest match. Additionally, he will add to the bitstream the code for the next character.
– When starting to encode the string *abaababaa*, the table is empty, so there is no match in it, and the encoder adds to the output bitstream the pair <0, a> (”0” for no match found in the table, and ”a” for the next character). After this, the encoder adds to the dictionary an entry for the string ”a”, which will have address 1.

– Continuing to encode the rest of the string, *baababaa*, the table has the single entry ”a”, so no match is found in the table. The encoder adds to the output bitstream the pair <0, b> (”0” for no match found in the table, and ”b” for the next character). After this, the encoder adds to the dictionary an entry for the string ”b”, which will have address 2.

– Continuing to encode the rest of the string, *aababaa*, we can find in the table the entry ”a”, which is the longest match now. The encoder adds to the output bitstream the pair <1, a> (”0” for match found in the first entry of the table, and ”a” for the next character). After this, the encoder adds to the dictionary an entry for the string ”aa”, which will have address 3.
How the decoder works.

<table>
<thead>
<tr>
<th>encoder output</th>
<th>&lt; 0, a &gt;</th>
<th>&lt; 0, b &gt;</th>
<th>&lt; 1, a &gt;</th>
<th>&lt; 2, a &gt;</th>
<th>&lt; 4, a &gt;</th>
<th>&lt; 4, b &gt;</th>
<th>&lt; 2, b &gt;</th>
<th>&lt; 7, b &gt;</th>
<th>&lt; 8, b &gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>decoder output</td>
<td>a,</td>
<td>b,</td>
<td>aa,</td>
<td>ba,</td>
<td>baa,</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Table entries</td>
<td>a</td>
<td>b</td>
<td>aa</td>
<td>ba</td>
<td>baa</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Table addresses</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In the example the encoded stream is < 0, a > < 0, b > < 1, a > < 2, a > < 4, a > < 4, b > < 2, b > < 7, b > < 8, b >. After decoding the first 5 pairs we have found the original text a, b, aa, ba, baa. When processing the sixth pair, < 4, b >, which represents the phrase 4 (i.e. ba) and the following character is b, therefore we complete the decoded string to a, b, aa, ba, baa, bab and bab is added to the dictionary as the phrase 6. The rest of the decoded string is a run of b’s.

The separation of the input string in substrings (the separation by commas in the previous example) is called a parsing strategy. The parsing strategy can be implemented in a trie structure. The characters of each phrase specify the path from the root to the node labelled by the index of the phrase.
TRIE DATA FOR LZ78 CODING

0

1     2
a      b

3     4
a      a

5     6
a      b
• the data structure in LZ78 grows without any bounds, so the growth must be stopped to avoid the use of too much memory. At the stopping moment the trie can be removed and re-initialized. Or it can be partly rebuilt using a few hundred of the recent bytes.

• Encoding with LZ78 may be faster than with LZ77, but decoding is slower, since we have to rebuild the dictionaries (tables) at decoding time.
The LZW variant of LZ78

- LZW is more popular than Ziv-Lempel coding, it is the basis of Unix compress program.
- LZW encodes only phrase numbers and does not have explicit characters in the encoded stream. This is possible by initializing the list of phrases to include all characters, say the entries 0 to 128, such that ”a” has address 97 and ”b” has address 98.
- a new phrase is built from an existing one by appending the first character of the next phrase to it.

<table>
<thead>
<tr>
<th>encoder input</th>
<th>a</th>
<th>b</th>
<th>a</th>
<th>ab</th>
<th>ab</th>
<th>ba</th>
<th>aba</th>
<th>abaa</th>
</tr>
</thead>
<tbody>
<tr>
<td>encoder output</td>
<td>97</td>
<td>98</td>
<td>97</td>
<td>128</td>
<td>128</td>
<td>129</td>
<td>131</td>
<td>134</td>
</tr>
<tr>
<td>New entry added</td>
<td>ab</td>
<td>ba</td>
<td>aa</td>
<td>aba</td>
<td>abb</td>
<td>baa</td>
<td>abaa</td>
<td></td>
</tr>
<tr>
<td>address of new entry</td>
<td>128</td>
<td>129</td>
<td>130</td>
<td>131</td>
<td>132</td>
<td>133</td>
<td>134</td>
<td></td>
</tr>
</tbody>
</table>

- in the example the encoder output is formed of the indexes in the dictionary 97, 98, 97, 128, 128, 129, 131, 134. Decoding 97, 98, 97, 128, 128 we find the original text a, b, a, ab, ab and construct the new entries in the dictionary 128, 129, 130, 131. We explain in detail the decoding starting from the next received index, 129.
- First we read from the encoded stream the entry 129 to be ba, which can be appended to
the decoded string, $a, b, a, ab, ab, ba$. At this moment the new phrase to be added to the dictionary is phrase $132 = abb$.

• Then we read from the encoded stream the entry 131. This is found to be $aba$ and added to the decoded string, $a, b, a, ab, ab, ba, aba$. We also add to the dictionary the new phrase $133 = baa$.

• the lag in the construction of the dictionary creates a problem when the encoder references a phrase index which is not yet available to the decoder. This is the case when 134 is received in the encoded stream: there is no 134 index in the dictionary yet. However, we know that 134 should start with $aba$ and contains an extra character. Therefore, we add to the decoded string $a, b, a, ab, ab, ba, aba, aba$?. Now we are able to say what is the phrase 134, namely $abaa$, and after this we can substitute $?$ by $a$.

• There are several variants of LZW. Unix compress is using an increasing number of bits for the indices: fewer when there are fewer entries (other variants are using for the same file the maximum number of bits necessary to encode all parsed substrings of the file).

• When a specified number of phrases are exceeded (full dictionary) the adaptation is stopped. The compression performance is monitored, and if it deteriorates significantly, the dictionary is rebuilt from scratch.
Encoding procedure for LZW

Given the text $S[1\ldots N]$

1. Set $p \leftarrow 1$. /* ($S(p)$ is the next character to be encoded) */

2. For each character $d \in \{0, \ldots, q - 1\}$ in the alphabet do /* initial dictionary */
   
   Set $D[d] \leftarrow$ character $d$.

3. Set $d \leftarrow q - 1$ /* $d$ points to the last entry in the dictionary */

4. While there is still text remaining to be coded do

   4.1 Search for the longest match for $S[p, p + 1, \ldots]$ in $D$. Suppose the match occurs at entry $c$, with length $\ell$.

   4.2 Output the code of $c$

   4.3 Set $d \leftarrow d + 1$. /* Add an entry to the dictionary*/

   4.4 Set $p \leftarrow p + \ell$.

   4.5 Set $D[d] \leftarrow D[c] + +S[p]$ /* Add an entry to the dictionary by concatenation*/
Decoding procedure for LZW

1. Set \( p \leftarrow 1 \). /* (\( S(p) \) is the next character to be decoded) */

2. For each character \( d \in \{0, \ldots, q - 1\} \) in the alphabet do /* initial dictionary */
   
   Set \( D[d] \leftarrow \) character \( d \).

3. Set \( d \leftarrow q - 1 \) /* \( d \) points to the last entry in the dictionary */

4. For each code \( c \) in the input do

   4.1 If \( d \neq (q - 1) \) then /* first time is an exception */
       
       Set last character of \( D[d] \leftarrow \) first character of \( D[c] \).

   4.2 Output \( D[c] \).

   4.3 Set \( d \leftarrow d + 1 \). /* Add an entry to the dictionary */

   4.4 Set \( D[d] \leftarrow D[c] + + ? \) /* Add an entry to the dictionary by concatenation, but the last character is currently unknown */
**Statistical analysis of a simplified Ziv-Lempel**

Algorithm for the universal data compression system: The binary source sequence is sequentially parsed into strings that have not appeared so far.

Let $c(n)$ be the number of phrases in the parsing of the input $n$-sequence. We need $\log c(n)$ bits to describe the location of the prefix to the phrase and 1 bit to describe the last bit.

The above two pass algorithm may be changed to a one pass algorithm, which allocates fewer bits for coding the prefix location.

The modifications do not change the asymptotic behaviour.

- Parse the source string into segments.
- Collect a dictionary of segments.
- Add to the dictionary a segment one symbol longer than the longest match so far found.
- Coding: Transmit the index of the matching segment in the dictionary plus the terminal bit;

Example: 010100010 → 0|1|01|00|010|
<table>
<thead>
<tr>
<th>Index in dictionary</th>
<th>Segment</th>
<th>Transmitted message</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>(0,0) → 00</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>(0,1) → 01</td>
</tr>
<tr>
<td>3</td>
<td>01</td>
<td>(1,1) → 11</td>
</tr>
<tr>
<td>4</td>
<td>00</td>
<td>(1,0) → 010</td>
</tr>
<tr>
<td>5</td>
<td>010</td>
<td>(3,0) → 0110</td>
</tr>
</tbody>
</table>

- Length of the code for increasing sizes of segment indices is
  \[
  L = \sum_{j=1}^{\text{Number of segments}} \log_2(j) + \text{Number of segments}
  \]

- If we assign the worst case length to all segment indices, and if the number of segments is \(c(n)\) with \(n\) the total length of the input string, the length is
  \[
  l = c(n)(1 + \log c(n))
  \]

  and the average length per input symbol is
  \[
  l = \frac{c(n)(1 + \log c(n))}{n}
  \]
• Definition A parsing of a binary string $x_1x_2\ldots x_n$ is a division of the string into phrases, separated by commas. A distinct parsing is a parsing such that no two phrases are identical.

• Lemma (Lempel and Ziv) The number of phrases $c(n)$ in a distinct parsing of a binary sequence $x_1x_2\ldots x_n$ satisfies

$$c(n) \leq \frac{n}{(1 - \varepsilon_n) \log n}$$

where $\varepsilon = \min(1, \frac{\log(\log n) + 4}{\log n})$.

• Theorem

Let $\{X_n\}$ be a stationary ergodic process with entropy rate $H(X)$ and let $c(n)$ be the number of distinct phrases in a distinct parsing of a sample of length $n$ from this process. Then

$$\lim \sup_{n \to \infty} \frac{c(n) \log c(n)}{n} \leq H(X)$$

with probability 1.
• Theorem

Let \( \{X_n\} \) be a stationary ergodic process with entropy rate \( H(X) \). Let \( l(X_1, X_2, \ldots, X_n) \) be the Lempel-Ziv codeword length associated with \( X_1, X_2, \ldots, X_n \). Then

\[
\limsup_{n \to \infty} \frac{1}{n} l(X_1, X_2, \ldots, X_n) \leq H(X)
\]

with probability 1.

Proof We know that \( l(X_1, X_2, \ldots, X_n) = c(n)(1 + \log c(n)) \). By Lemma Lempel-Ziv \( \frac{c(n)}{n} \to 0 \) and thus

\[
\limsup_{n \to \infty} \frac{1}{n} l(X_1, X_2, \ldots, X_n) = \limsup_{n \to \infty} \left( \frac{c(n) \log c(n)}{n} + \frac{c(n)}{n} \right) \leq H(X)
\]