Lempel-Ziv Compression Techniques

• Outline:
  – Classification of Lossless Compression techniques
  – Introduction to Lempel-Ziv Encoding: LZ77 & LZ78
  – LZ78 Encoding Algorithm
  – LZ78 Decoding Algorithm

[Thanks to Dr. Jauhar Ali]
Classification of Lossless Compression Techniques

- Lossless Compression techniques are classified into static, adaptive (or dynamic), and hybrid.

- Static coding requires two passes: one pass to compute probabilities (or frequencies) and determine the mapping, and a second pass to encode.

- **Examples of Static techniques:** Static Huffman Coding

- All of the adaptive methods are *one-pass* methods; only one scan of the message is required.

- **Examples of adaptive techniques:** LZ77, LZ78, LZW, and Adaptive Huffman Coding
  - Adaptive Huffman Coding: initial frequency counts cannot be made, so tree adapts as data arrives – basic idea is new data starts at top of tree and is “pushed down” as it becomes relatively less frequent
Introduction to Lempel-Ziv Encoding

• Data compression up until the late 1970's mainly directed towards creating better methodologies for Huffman coding.

• An innovative, radically different method was introduced in 1977 by Abraham Lempel and Jacob Ziv.

• This technique (called Lempel-Ziv) actually consists of two considerably different algorithms, LZ77 and LZ78.

• Due to patents, LZ77 and LZ78 led to many variants:

<table>
<thead>
<tr>
<th>LZ77 Variants</th>
<th>LZR</th>
<th>LZSS</th>
<th>LZB</th>
<th>LZH</th>
</tr>
</thead>
<tbody>
<tr>
<td>LZ78 Variants</td>
<td>LZW</td>
<td>LZC</td>
<td>LZT</td>
<td>LZMW</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>LZJ</th>
<th>LZFG</th>
</tr>
</thead>
</table>

• The **zip** and **unzip** use the LZH technique while UNIX's **compress** methods belong to the LZW and LZC classes.
LZ78 Compression Algorithm

LZ78 inserts one- or multi-character, non-overlapping, distinct patterns of the message to be encoded in a Dictionary.

The multi-character patterns are of the form: $C_0C_1 \ldots C_{n-1}C_n$. The prefix of a pattern consists of all the pattern characters except the last: $C_0C_1 \ldots C_{n-1}$

**LZ78 Output:**

<table>
<thead>
<tr>
<th>(0, char)</th>
<th>if one-character pattern is not in Dictionary.</th>
</tr>
</thead>
<tbody>
<tr>
<td>(DictionaryPrefixIndex, lastPatternCharacter)</td>
<td>if multi-character pattern is not in Dictionary.</td>
</tr>
<tr>
<td>(DictionaryPrefixIndex,  )</td>
<td>if the last input character or the last pattern is in the Dictionary.</td>
</tr>
</tbody>
</table>

Note: The dictionary is usually implemented as a hash table.
LZ78 Compression Algorithm (cont’d)

Dictionary ← empty ; Prefix ← empty ; DictionaryIndex ← 1;
while(characterStream is not empty)
{
    Char ← next character in characterStream;
    if(Prefix + Char exists in the Dictionary)
        Prefix ← Prefix + Char ;
    else
    {
        if(Prefix is empty)
            CodeWordForPrefix ← 0 ;
        else
            CodeWordForPrefix ← DictionaryIndex for Prefix ;
        Output: (CodeWordForPrefix, Char) ;
        insertInDictionary( ( DictionaryIndex , Prefix + Char) ) ;
        DictionaryIndex++ ;
        Prefix ← empty ;
    }
}
if(Prefix is not empty)
{
    CodeWordForPrefix ← DictionaryIndex for Prefix ;
    Output: (CodeWordForPrefix ,   ) ;
}
Example 3: LZ78 Compression

Encode (i.e., compress) the string $\text{AAAAAAAA}$ using the LZ78 algorithm.

1. $A$ is not in the Dictionary; insert it
2. $A$ is in the Dictionary
   - $AA$ is not in the Dictionary; insert it
3. $A$ is in the Dictionary.
   - $AA$ is in the Dictionary.
   - $AAA$ is not in the Dictionary; insert it.
4. $A$ is in the Dictionary.
   - $AA$ is in the Dictionary.
   - $AAA$ is in the Dictionary and it is the last pattern; output a pair containing its index: $(3, \ )$
Example 1: LZ78 Compression

Encode (i.e., compress) the string **ABBCBCABABCAABCAAB** using the LZ78 algorithm.

The compressed message is: \((0, A)(0, B)(2, C)(3, A)(2, A)(4, A)(6, B)\)

**Note:** The above is just a representation, the commas and parentheses are not transmitted; we will discuss the actual form of the compressed message later on in slide 12.
Example 1: LZ78 Compression (cont’d)

1. A is not in the Dictionary; insert it
2. B is not in the Dictionary; insert it
3. B is in the Dictionary.
   BC is not in the Dictionary; insert it.
4. B is in the Dictionary.
   BC is in the Dictionary.
   BCA is not in the Dictionary; insert it.
5. B is in the Dictionary.
   BA is not in the Dictionary; insert it.
   BC is in the Dictionary.
   BCA is in the Dictionary.
   BCAA is not in the Dictionary; insert it.
7. B is in the Dictionary.
   BC is in the Dictionary.
   BCA is in the Dictionary.
   BCAA is in the Dictionary.
   BCAAB is not in the Dictionary; insert it.
Example 2: LZ78 Compression

Encode (i.e., compress) the string **BABABAABRRRA** using the LZ78 algorithm.

The compressed message is: **(0,B)(0,A)(1,A)(2,B)(0,R)(5,R)(2, )**
Example 2: LZ78 Compression (cont’d)

1. \textbf{B} is not in the Dictionary; insert it
2. \textbf{A} is not in the Dictionary; insert it
3. \textbf{B} is in the Dictionary.
   - \textbf{BA} is not in the Dictionary; insert it.
4. \textbf{A} is in the Dictionary.
   - \textbf{AB} is not in the Dictionary; insert it.
5. \textbf{R} is not in the Dictionary; insert it.
6. \textbf{R} is in the Dictionary.
   - \textbf{RR} is not in the Dictionary; insert it.
7. \textbf{A} is in the Dictionary and it is the last input character; output a pair containing its index: (2, )
**LZ78 Compression: Number of bits transmitted**

- Example: Uncompressed String: `ABBCBCABABCAABCAAB`
  
  Number of bits = Total number of characters * 8
  
  \[= 18 \times 8\]
  
  \[= 144 \text{ bits}\]

- Suppose the codewords are indexed starting from 1:
  
  Compressed string (codewords): `(0, A) (0, B) (2, C) (3, A) (2, A) (4, A) (6, B)`

<table>
<thead>
<tr>
<th>Codeword index</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
</table>

- Each code word consists of an integer and a character:
  
  - The character is represented by 8 bits.
  
  - The number of bits \(n\) required to represent the integer part of the codeword with index \(i\) is given by:

    \[
    n = \begin{cases} 
    1 & \text{if } i = 1 \\
    \left\lfloor \log_2 i \right\rfloor & \text{if } i > 1 
    \end{cases}
    \]

- Alternatively, number of bits required to represent the integer part of the codeword with index \(i\) is the number of significant bits required to represent the integer \(i - 1\).
LZ78 Compression: Number of bits transmitted (cont’d)

<table>
<thead>
<tr>
<th>index</th>
<th>index - 1</th>
<th>bits</th>
<th>Number of significant bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>11</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>100</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>101</td>
<td>4</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>110</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>7</td>
<td>111</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>8</td>
<td>1000</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>9</td>
<td>1001</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>10</td>
<td>1010</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>11</td>
<td>1011</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>12</td>
<td>1100</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>13</td>
<td>1101</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>14</td>
<td>1110</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>15</td>
<td>1111</td>
<td></td>
</tr>
</tbody>
</table>

Codeword (0, A) (0, B) (2, C) (3, A) (2, A) (4, A) (6, B)

index 1 2 3 4 5 6 7
Bits: (1 + 8) + (1 + 8) + (2 + 8) + (2 + 8) + (3 + 8) + (3 + 8) + (3 + 8) = 71 bits

The actual compressed message is: 0A0B10C11A010A100A110B

where each character is replaced by its binary 8-bit ASCII code.
LZ78 Decompression Algorithm

Dictionary ← empty ; DictionaryIndex ← 1 ;
while(there are more (CodeWord, Char) pairs in codestream){
    CodeWord ← next CodeWord in codestream ;
    Char ← character corresponding to CodeWord ;
    if(CodeWord = = 0)
        String ← empty ;
    else
        String ← string at index CodeWord in Dictionary ;
    Output: String + Char ;
    insertInDictionary( (DictionaryIndex , String + Char) ) ;
    DictionaryIndex++;
}

Summary:

- **input**: (CW, character) pairs
- **output**:
  - if(CW == 0)
    - output: currentCharacter
  - else
    - output: stringAtIndex CW + currentCharacter
- **Insert**: current output in dictionary
Example 1: LZ78 Decompression

Decode (i.e., decompress) the sequence \((0, A) (0, B) (2, C) (3, A) (2, A) (4, A) (6, B)\)

The decompressed message is: ABBCBCABABABCAABCAAB
### Example 2: LZ78 Decompression

Decode (i.e., decompress) the sequence \((0, B) (0, A) (1, A) (2, B) (0, R) (5, R) (2, \ )\)

<table>
<thead>
<tr>
<th>output</th>
<th>index</th>
<th>string</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>1</td>
<td>B</td>
</tr>
<tr>
<td>A</td>
<td>2</td>
<td>A</td>
</tr>
<tr>
<td>BA</td>
<td>3</td>
<td>BA</td>
</tr>
<tr>
<td>AB</td>
<td>4</td>
<td>AB</td>
</tr>
<tr>
<td>R</td>
<td>5</td>
<td>R</td>
</tr>
<tr>
<td>RR</td>
<td>6</td>
<td>RR</td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The decompressed message is: **BABAABRRRA**
Example 3: LZ78 Decompression

Decode (i.e., decompress) the sequence \((0, A) (1, A) (2, A) (3, \ )\)

<table>
<thead>
<tr>
<th>output</th>
<th>index</th>
<th>string</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>AA</td>
<td>2</td>
<td>AA</td>
</tr>
<tr>
<td>AAA</td>
<td>3</td>
<td>AAA</td>
</tr>
<tr>
<td>AAAA</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The decompressed message is: AAAAAA
LZW: Lempel-Ziv-Welch

Improvement of LZ78 that uses an initial (standard) predefined dictionary (e.g., 26 characters)

Dictionary can grow (up to a predetermined size)

Dictionary can be specialized to different data types (e.g., text, images, etc.)