# LZ77 – sliding window

## Algorithm description

LZ77 is a lossless data compression algorithm developed by A. Lempel and J. Ziv in the year 1977. It uses a sliding window usually split into 2 buffers: a search buffer and look-ahead buffer. The algorithm tries to compress data by replacing current occurrence with previously seen data

Algorithm steps:

- Find the longest match from search buffer inside the look-ahead buffer.
- This prefix is encoded as triplet (i, j, X) – called token
  - i is the distance of the beginning of the found prefix from the end of the search buffer. For your implementation It will always start from the right side of the search buffer. The value of the first index is 0

  *** In some examples on the internet you might find that some people are starting with index from as 1 and starting to search inside SB from the left side of the SB.
  - j is the length of the found prefix
  - X is the first character after the prefix in look-ahead buffer.

| | Search buffer | | | | | | | | | Look-ahead buffer | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| …a | c | C | a | b | r | a | c | a | d | a | b | r | a | r | r | a | r | r | a | c | … |

| 6 | 5 | 4 | 3 | 2 | 1 | 0 | | | | | | | output |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | a | b | r | a | c | ada….. | 0,0,a |
| | | | | | | a | b | r | a | c | a | dab… | 0,0,b |
| | | | | | a | b | r | a | c | a | d | abr… | 0,0,r |
| | | | | a | b | r | a | c | a | d | a | bra… | 2,1,c |
| | | a | b | r | a | c | a | d | a | b | r | ad | 1,1,d |
| a | b | r | a | c | a | d | a | b | r | a | d | | 6,4,d |
| a | d | a | b | r | a | d | | | | | | | |
| search buffer | | | | | | | look-ahead buffer | | | | | | |

Interesting situations where we can code something that is not inside search buffer but inside look-ahead buffer:

| 6 | 5 | 4 | 3 | 2 | 1 | 0 | | | | | | | | output |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | a | b | a | b | a | b | c | 0,0,a |
| | | | | | | a | b | a | b | a | b | c | | 0,0,b |
| | | | | | a | b | a | b | a | b | c | | | **1,4,c** |
| search buffer | | | | | | | look-ahead buffer | | | | | | | |

## Implementation requirements for **Coder**:

- UI shall contain the following option
  - Load button – it will offer the user to load a file and will display the full file path(including name) for the selected file
  - Choose the number of bits for the offset – [3-16]
  - Choose the number of bits for the length – [2-7]
  - Encoded button – when press it will create another file with the following filename format:
    - File.ext.o[offset_number_selected_from_ui]l[length_number_selected_from_UI].lz77
  - A checkbox to display on UI the emitted tokens
- The encoded file will contain the following"
  - On the first 4 bits you will write the offset value selected from UI
  - On the next 3 bits you will write the length value selected from UI
  - You will than read from the input file and emit tokens in the encoded file:
    - Offset – you will write the value on number of bits for the offset selected from UI
    - Length – you will write the value on number of bits for the length selected from UI
    - Symbol – you will write the value on 8 bits
  - At the end you'll write additionally 7 bits to fill the output buffer for the BitWriter

## Implementation requirements for **Decoder**:

- UI shall contain the following option
  - Load button – it will offer the user to load a file and will display the full file path (including name) for the selected file. The file must have the lz77 extensions.
  - Decode button – when press it will create another file with the following filename format:
    - File.ext.o[offset_number_selected_from_ui]l[length_number_selected_from_UI].lz77.ext
- The decoder shall read the following from the input encoded file
  - 4 bits – the value of the offset selected for the encoded file
  - 3 bits – the value of the length selected for the encoded file
  - You will than read from the encoded input file tokens:
    - Offset
    - Length
    - Symbol

## Suggestions:

1. **Encoding end of file**

The algorithm always must generate tokens. Thus the problem of encoding end of file appears. In case the last character is part of the token (as the char not matching the sub part) than we are in the happy situation. In case the last byte is part of a sub-string found inside the SB than we need to define a solution. One possible solution is to ignore the last character and treat it like an unknown character thus being able to generate a token. For the following example. We only have 2 remaining characters inside the LAB buffer. Those 2 characters are found inside SB, but we can't encode them as Position, Length, Symbol - because we do not have any characters left to encode – entire “ab” is found inside SB. One possible option is to treat the last symbol as an unseen symbol. Thus the token for the last part will be **6 , 1, ‘B’**

| | Search buffer | | | | | | | | | Look-ahead buffer | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| … | | | a | b | r | A | c | a | d | a | B |

2. Encoding in advance (look inside the future)

Let’s say the SB + LAB look like in the table below:

| Search buffer | | | | | Look-ahead buffer | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | a | B | c | d | a | b | c | d | a | b | c | d | a | b | c | d | a | b | c | d | e |

If we treat the SB+LAB as one buffer and imagine that with each symbol the window slides, than the above sample can be encoded with one token: **3, 16,’E’** . This is possible because what was already coded at the starting of the LAB buffer it can later be decoded

3. Buffer implementation

Although there are multiple possible solution how to implement the SB+LAB - one possible solution to implement is to use 2 “infinite” arrays/lists. One will be called SB and will be empty initially, and the other one will be called LAB and will initially contain the entire file to be coded. Each time we emit tokens, we take the traversed symbols from LAB and add them to SB. We can see clearly that this option consumes a lot of memory. Another possible solution is to go with only one single buffer/array/list with size ~ of SB + LAB. We shall maintain an index from where the LAB starts. At starting of the algorithm we just read as many characters in order to fill in the LAB part from that buffer. After encoding the first symbol, we shift the buffer to the left. In this way we add a symbol into the SB part and remove it from the LAB part of the buffer. We than read another symbol from the file and added to the right of the buffer. To make more generic the solution proposed - each time we encode n characters, we shift the buffer with n position to the left, and read n characters from the original file and add them to the right of the buffer.