Lossless Source Coding

Lempel-Ziv Coding

Lempel-Ziv 1977 (LZ77)

The Lempel-Ziv Algorithms

- A family of data compression algorithms introduced in

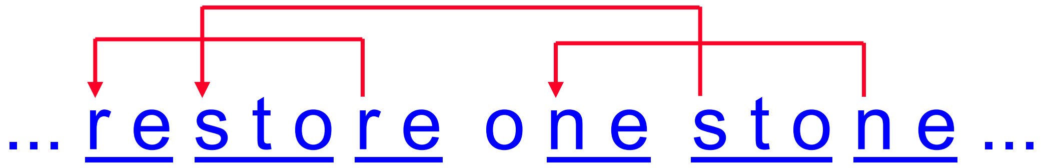
  [LZ77] J. Ziv and A. Lempel, “A universal algorithm for sequential data compression,” *IEEE Trans. Inform.Theory*, vol. IT-23, pp. 337–343, May 1977

  [LZ78] J. Ziv and A. Lempel, “Compression of individual sequences via variable rate coding,” *IEEE Trans. Inform. Theory*, vol. IT-24, pp. 530–536, Sept. 1978.

- Many desirable features, the conjunction of which was unprecedented at the time
  - *simple* and *elegant*
  - *universal* for *individual sequences* in the class of *finite-state encoders*
    - Arguably, every real-life computer is a finite-state automaton
  - processing is sequential, symbol by symbol, but compression ratio approaches entropy rate in the limit for *stationary ergodic sources*
  - *string matching* and *dictionaries*, no explicit probability model
  - very *practical*, with *fast and effective implementations* applicable to a wide range of data types and applications

Two Main Variants

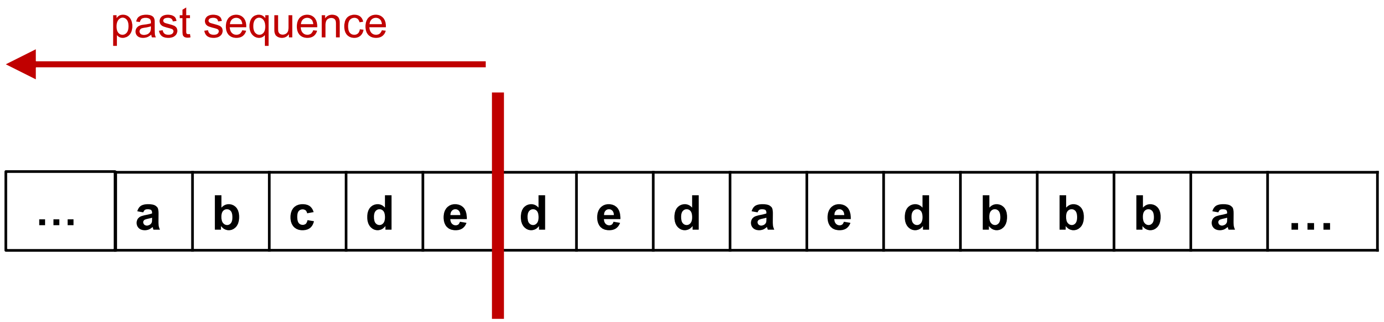
- [LZ77] and [LZ78] present different algorithms with common elements
  - The main mechanism in both schemes is *pattern matching*: find string patterns that have occurred in the past, and compress them by encoding a reference to the previous occurrence



... restore one stone ...

- Both schemes are in wide practical use
  - many variations exist on each of the major schemes
    - gzip, WinZIP, 7z, RAR, GIF, TIFF, PNG, ...
  - we give a brief description of LZ77 and its properties, and then focus in more detail on LZ78, which admits a simpler analysis with a stronger result

LZ77: Sliding Window Lempel-Ziv

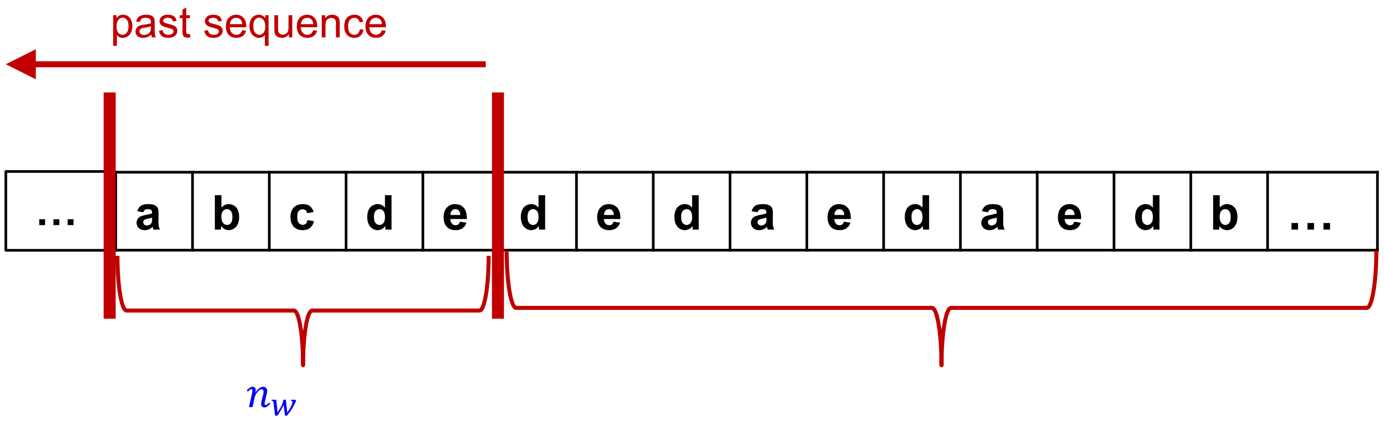


past sequence

… a b c d e d e d a e d b b b a …

- Sequence x_1^n over alphabet $A,\ |A| \geq 2$.
- Say we have already processed the sequence up to the indicated point
- Fix a *window size* $n_w \geq 1$

LZ77: Sliding Window Lempel-Ziv

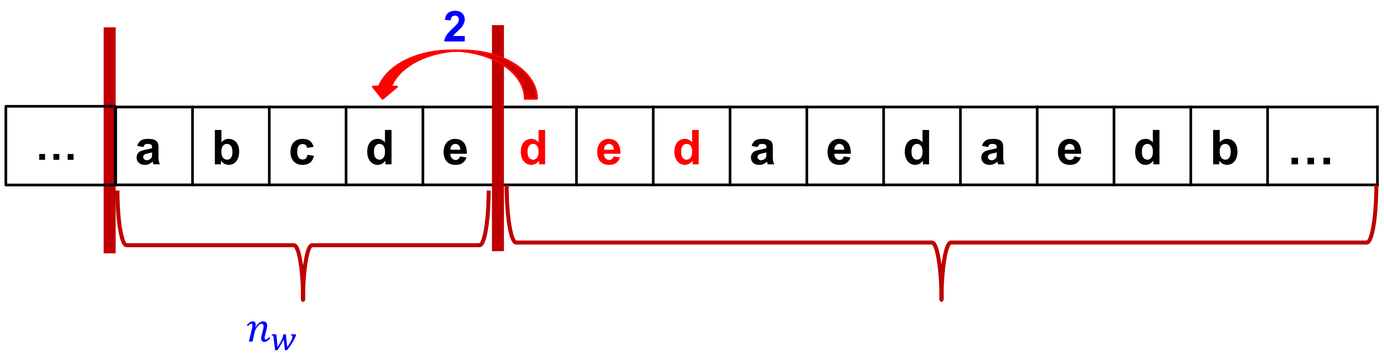


fixed size window (*dictionary*)

look-ahead buffer

- Next *phrase*:
  - find longest *match* to the look-ahead buffer, starting in the dictionary (but can go into the look-ahead buffer): length $L \geq 0$

LZ77: Sliding Window Lempel-Ziv



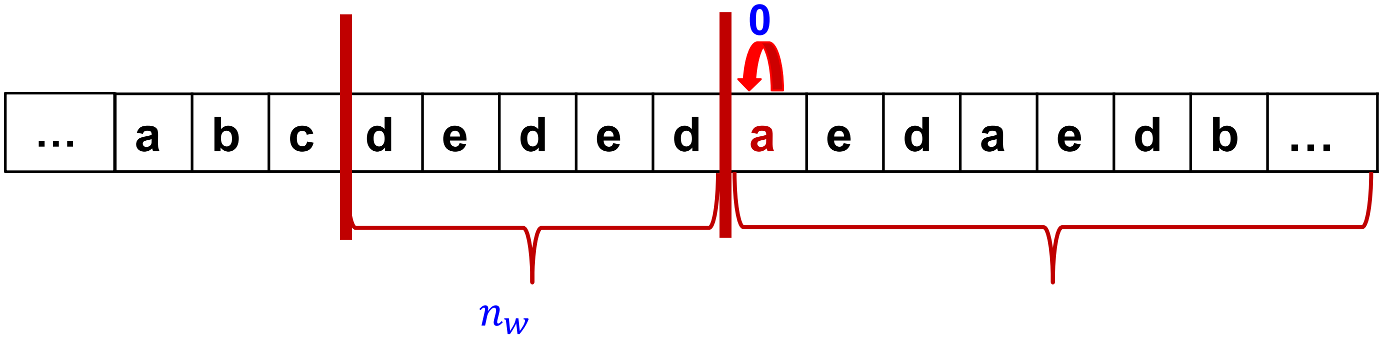
fixed size window (*dictionary*)

look-ahead buffer

- Next *phrase*:
  - find longest *match* to the look-ahead buffer, starting in the dictionary (but can go into the look-ahead buffer): length $L \geq 0$
  - represent phrase as (L, Δ)= (length, offset) if $L > 1$, or $(1, x_i)$ otherwise

    phrase 1 (ded) : $Y_1 = (3, 2)$

LZ77: Sliding Window Lempel-Ziv



fixed size window (*dictionary*)

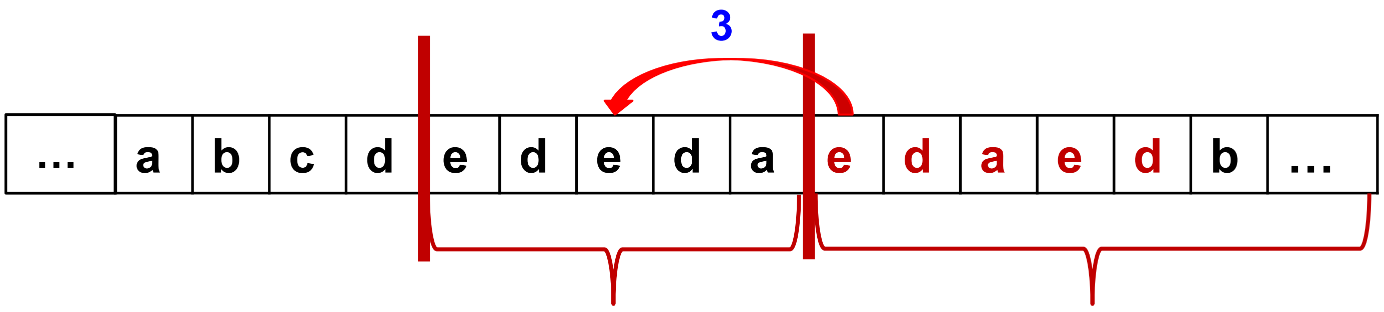
look-ahead buffer

- ☐ Next *phrase*:
  - find longest *match* to the look-ahead buffer, starting in the dictionary (but can go into the look-ahead buffer): length $L \geq 0$
  - represent phrase as (L, Δ)= (length, offset) if $L > 1$, or $(1, x_i)$ otherwise

    phrase 1 (ded) : $Y_1 = (3, 2)$

    phrase 2 (a) : $Y_2 = (1, \mathrm{a})$

LZ77: Sliding Window Lempel-Ziv



n_w

fixed size window (*dictionary*)

look-ahead buffer

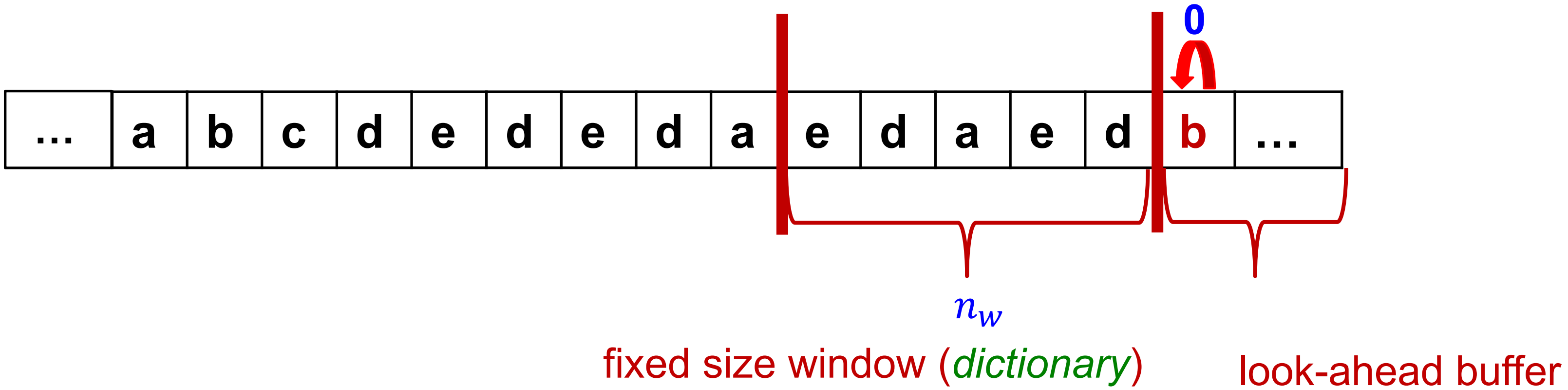
- Next *phrase*:
  - find longest *match* to the look-ahead buffer, starting in the dictionary (but can go into the look-ahead buffer): length $L \geq 0$
  - represent phrase as (L, Δ)= (length, offset) if $L > 1$, or $(1, x_i)$ otherwise

phrase 1 (ded) : $Y_1 = (3, 2)$

phrase 2 (a) : $Y_2 = (1, \mathrm{a})$

phrase 3 (edaedb) : $Y_3 = (5, 3)$

LZ77: Sliding Window Lempel-Ziv

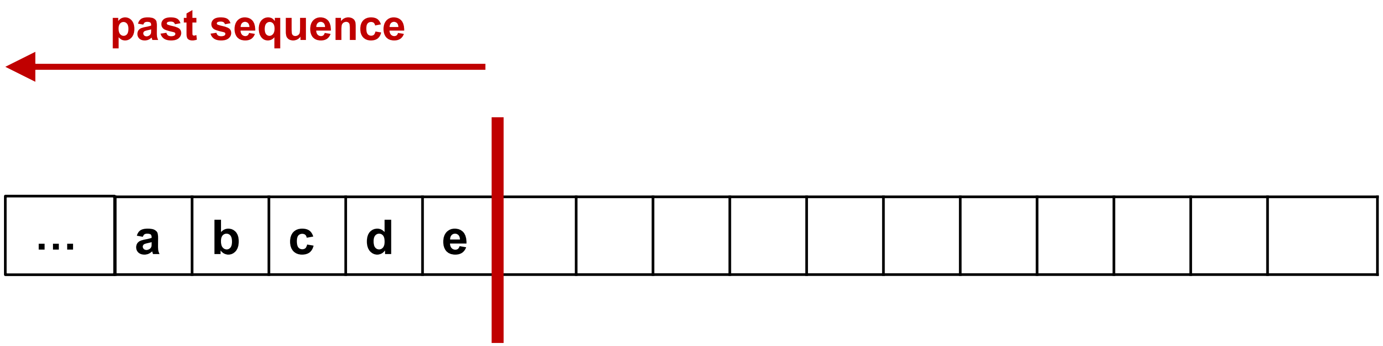


- Next *phrase*:
  - find longest *match* to the look-ahead buffer, starting in the dictionary (but can go into the look-ahead buffer): length $L \geq 0$
  - represent phrase as (L, Δ)= (length, offset) if $L > 1$, or $(1, x_i)$ otherwise

phrase 1 (ded) : $Y_1 = (3, 2)$
phrase 2 (a) : $Y_2 = (1, \mathrm{a})$
phrase 3 (edaedb) : $Y_3 = (5, 3)$
phrase 4 (b) : $Y_4 = (1, \mathrm{b})$

LZ77: Decoding

- Given $Y_1, Y_2, Y_3, \ldots$, we can reconstruct x_1^n
- Say we have already decoded the sequence up to the indicated point



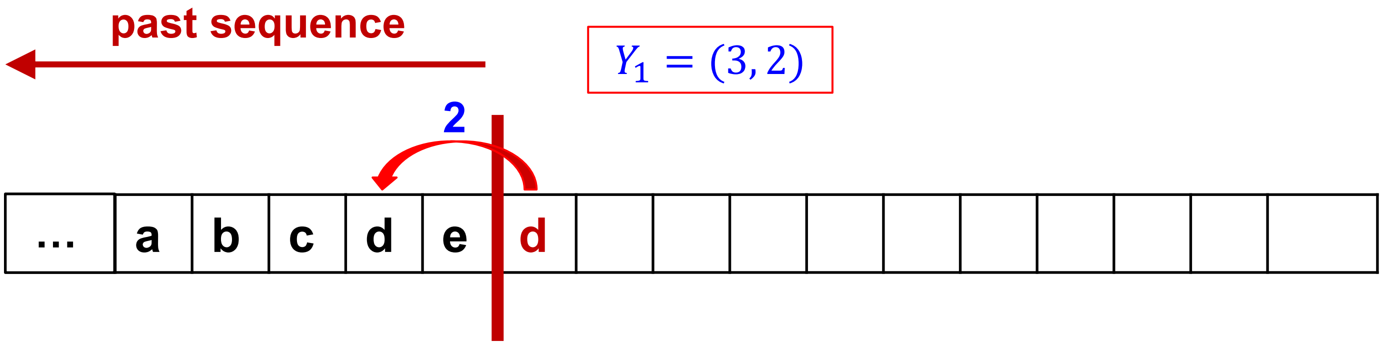
past sequence

… a b c d e

Encoder sent Y_1= (3, 2), Y_2 = (1, a), Y_3 = (5, 3), Y_4 = (1, b), …

- $Y_j = (L, \Delta)$= (length, offset) if $L > 1$, or $(1, x_i)$ otherwise

LZ77: Decoding

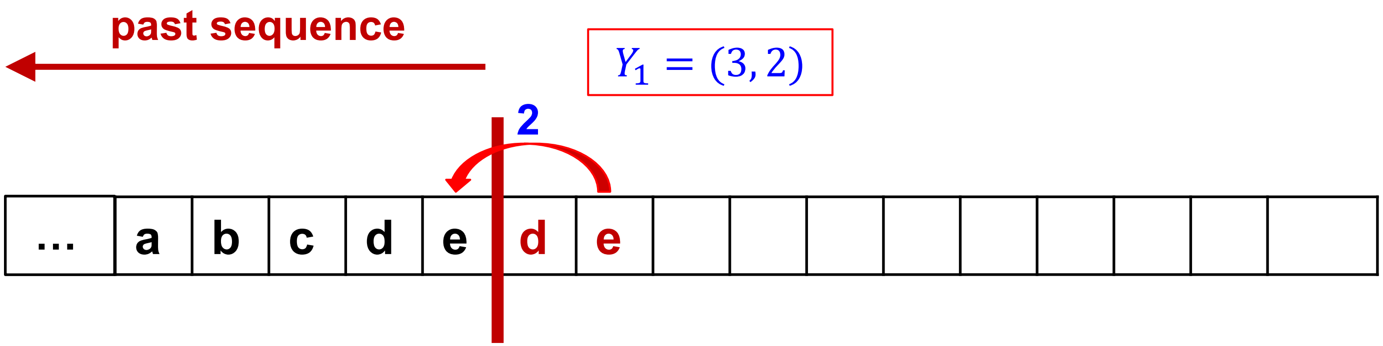


$Y_1 = (3, 2)$

Encoder sent $Y_1 = (3, 2)$, $Y_2 = (1, a)$, $Y_3 = (5, 3)$, $Y_4 = (1, b)$, …

- $Y_j = (L, \Delta) =$ (length, offset) if $L > 1$, or $(1, x_i)$ otherwise

LZ77: Decoding

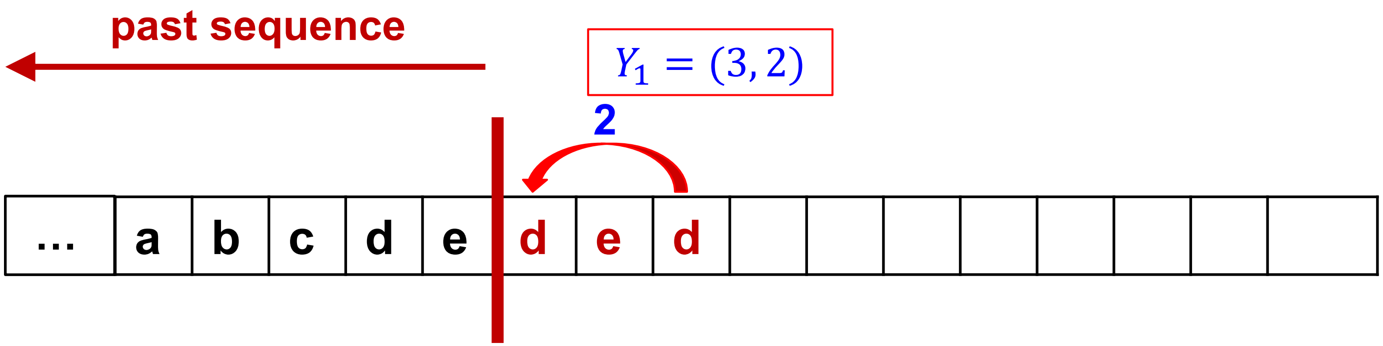


$Y_1 = (3, 2)$

Encoder sent $Y_1 = (3, 2)$, $Y_2 = (1, \text{a})$, $Y_3 = (5, 3)$, $Y_4 = (1, \text{b})$, …

- $Y_j = (L, \Delta) =$ (length, offset) if $L > 1$, or $(1, x_i)$ otherwise

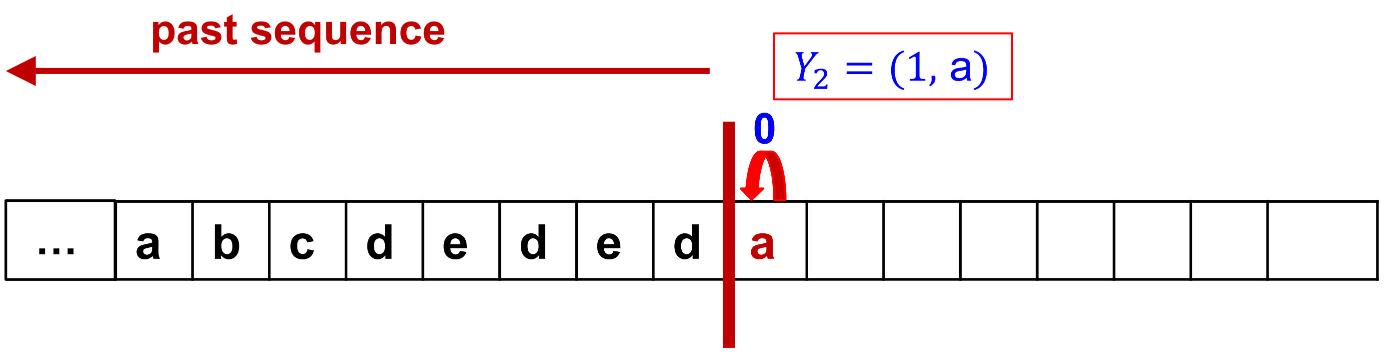
LZ77: Decoding



Encoder sent Y_1= (3, 2), Y_2 = (1, a), Y_3 = (5, 3), Y_4 = (1, b), …

- $Y_j = (L, \Delta)$= (length, offset) if $L > 1$, or $(1, x_i)$ otherwise

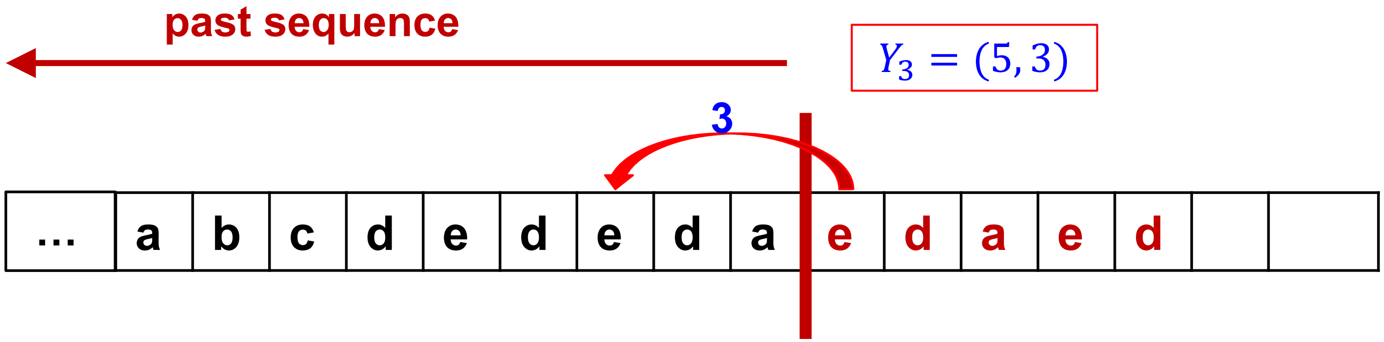
LZ77: Decoding



Encoder sent $Y_1 = (3, 2)$, $Y_2 = (1, \text{a})$, $Y_3 = (5, 3)$, $Y_4 = (1, \text{b})$, …

- $Y_j = (L, \Delta)$= (length, offset) if $L > 1$, or $(1, x_i)$ otherwise

LZ77: Decoding



past sequence

$Y_3 = (5,3)$

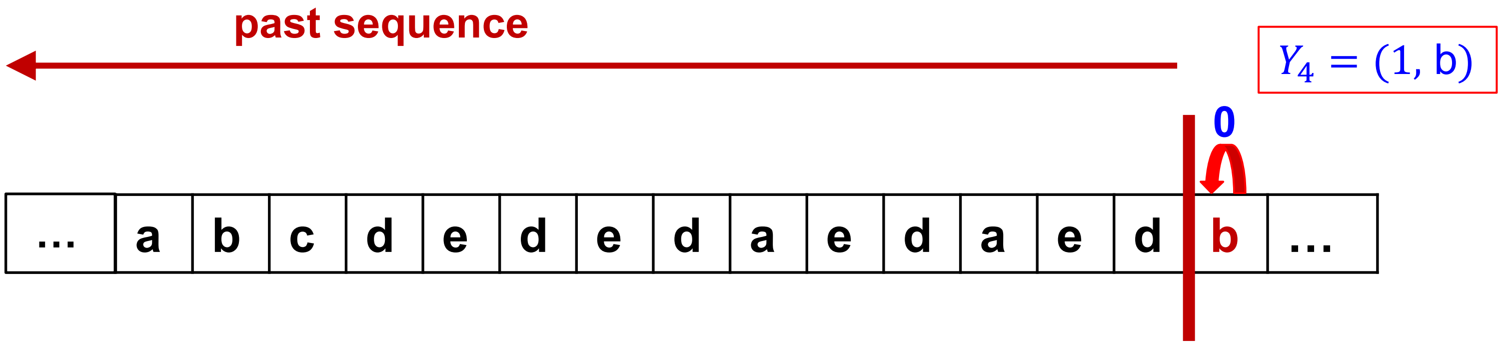
3

… a b c d e d e d a e d a e d

Encoder sent Y_1= (3, 2), Y_2 = (1, a), Y_3 = (5, 3), Y_4 = (1, b), …

- $Y_j = (L, \Delta)$= (length, offset) if $L > 1$, or $(1, x_i)$ otherwise

LZ77: Decoding



Encoder sent $Y_1 = (3, 2)$, $Y_2 = (1, \text{a})$, $Y_3 = (5, 3)$, $Y_4 = (1, \text{b})$, …

- $Y_j = (L, \Delta)$ = (length, offset) if $L > 1$, or $(1, x_i)$ otherwise

LZ77: Binary Encoding of Phrases

- ☐ Phrase $Y_j = (L, \Delta)$ with $L > 1$, or $Y_j = (1, x_i)$
  - ● Δ : $\lceil \log n_w \rceil$ bits (log in base 2)
  - ● L : use prefix-free, variable length code for nonnegative integers

    Example: let $\ell = \lceil \log(L+1) \rceil$, $\ell' = \lceil \log(\ell+1) \rceil$

    represent L as $$0^{\ell'-1}1 \cdot \underbrace{\mathrm{binary}(\ell)}_{\ell} \cdot \underbrace{\mathrm{binary}(L)}_{\ell'}$$

    total length for $L \approx \log L + 2\log\log L$
  - ● x_i : $\lceil \log|A| \rceil$ bits
- ☐ Appropriate conventions are needed for the first n_w symbols
- ☐ Let $\mathcal{L}_{n_w}(x_1^n)$ = *total length* (in bits) of representations of $Y_1, Y_2, Y_3, \ldots$
- ☐ *Compression ratio*: $R_{n_w,n}(x^n) = \frac{1}{n}\mathcal{L}_{n_w}(x^n)$ (bits/symbol)

Optimality of LZ77

- ☐ Let $X_1^\infty \sim P$ be a *stationary ergodic* process over A.
- ☐ Recall

  n-th order entropy rate: $H_n(X_1^n) = -\frac{1}{n}\sum_{x^n \in A^n} P(x_1^n)\log P(x_1^n)$

  entropy rate: $H(X_1^\infty) = \lim_{n\to\infty} H_n(X_1^n)$ (in bits/symbol, limit exists)

  LZ77 average compression ratio: $\bar{R}_{n_w,n} = E_P\left[R_{n_w,n}(X_1^n)\right]$

- ☐ Theorem

$$\lim_{n_w\to\infty}\lim_{n\to\infty}\bar{R}_{n_w,n} = H$$

  - *Optimal* due to Shannon's lower bound
  - *Universal:* achieves optimal compression ratio without any prior knowledge of P
  - Proof : A. D. Wyner and J. Ziv, "The sliding-window Lempel-Ziv algorithm is asymptotically optimal," *Proc. IEEE*, vol. 82, pp. 872--877, June 1994.
  - Original LZ77 paper did not show optimality in a stochastic sense

gzip: An application of LZ77 (+Huffman)

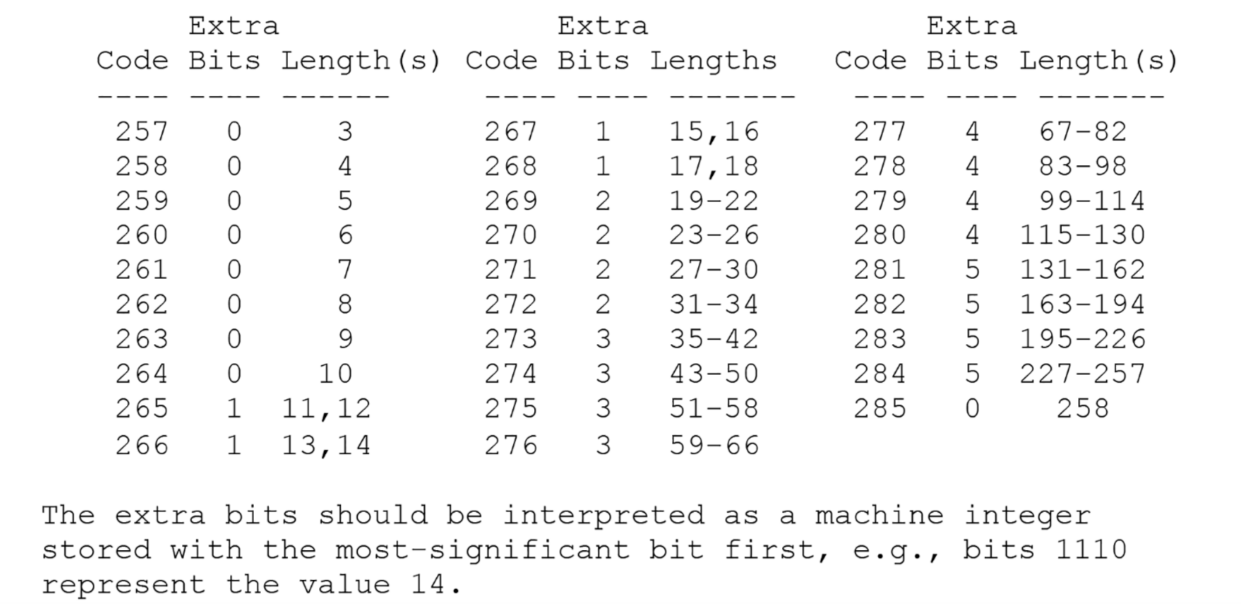
- A popular lossless compression program available in most computing platforms (Windows, Linux, MacOS, etc.)
- Used for general purpose file compression
- The main compression algorithm in *gzip* is called *deflate*, a variant of LZ77 (+Huffman)
  - blocks of data can also be stored uncompressed
  - *deflate* also at the core of *zip*, *PKzip*, *Winzip, PNG*, and others
- Main elements of *deflate*:
  - a block of data is encoded as a sequence of *tokens*
  - each token is encoded with a prefix-free (Huffman) code, and can represent
    - a *literal byte* (0 .. 255)
    - a *length* in a *<length, offset>* pair (3 .. 258) *minimal match length is 3*
    - an *offset* in a *<length, offset>* pair (1 .. 2^{15})
  - Two alphabets, and two Huffman codes are used
    - one for literals and match lengths (merged into one alphabet)
    - one for offsets
  - Huffman codes can be *fixed* (pre-defined defaults) or *dynamic* (described in the encoded stream)

Encoding of literals/match lengths

Codes 0 .. 255: literal bytes

Code 256: end of block

Codes 257.. 285: match lengths

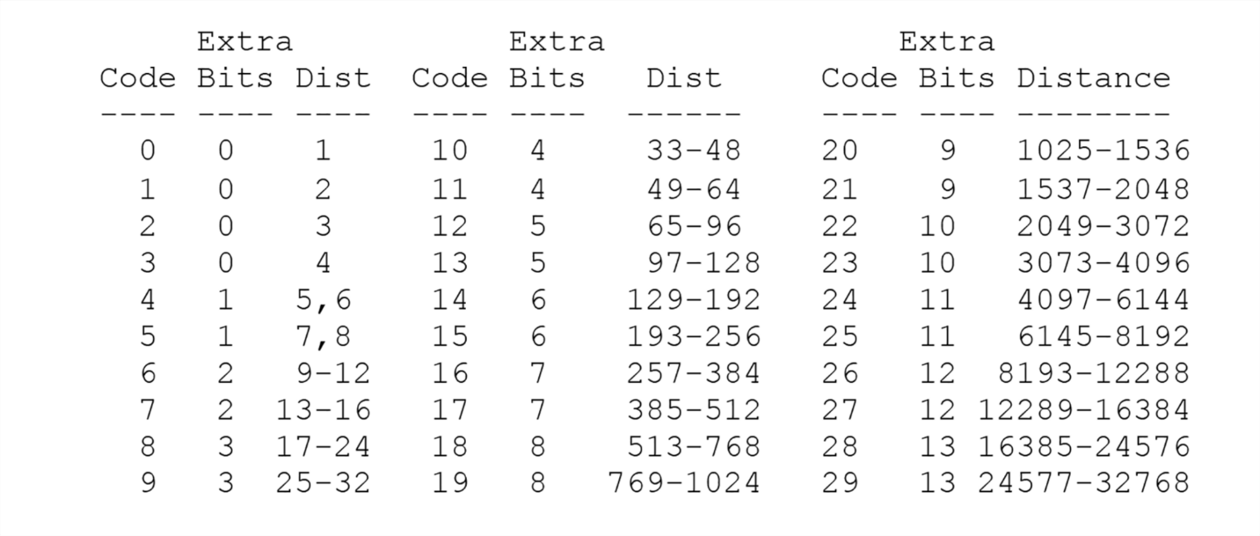


Code	Extra Bits	Length(s)	Code	Extra Bits	Lengths	Code	Extra Bits	Length(s)
257	0	3	267	1	15,16	277	4	67-82
258	0	4	268	1	17,18	278	4	83-98
259	0	5	269	2	19-22	279	4	99-114
260	0	6	270	2	23-26	280	4	115-130
261	0	7	271	2	27-30	281	5	131-162
262	0	8	272	2	31-34	282	5	163-194
263	0	9	273	3	35-42	283	5	195-226
264	0	10	274	3	43-50	284	5	227-257
265	1	11,12	275	3	51-58	285	0	258
266	1	13,14	276	3	59-66			

The extra bits should be interpreted as a machine integer stored with the most-significant bit first, e.g., bits 1110 represent the value 14.

A Huffman code over the alphabet $\{0,1,\ldots,285\}$ is used for these codes + an appropriate number of extra bits

Encoding of offsets

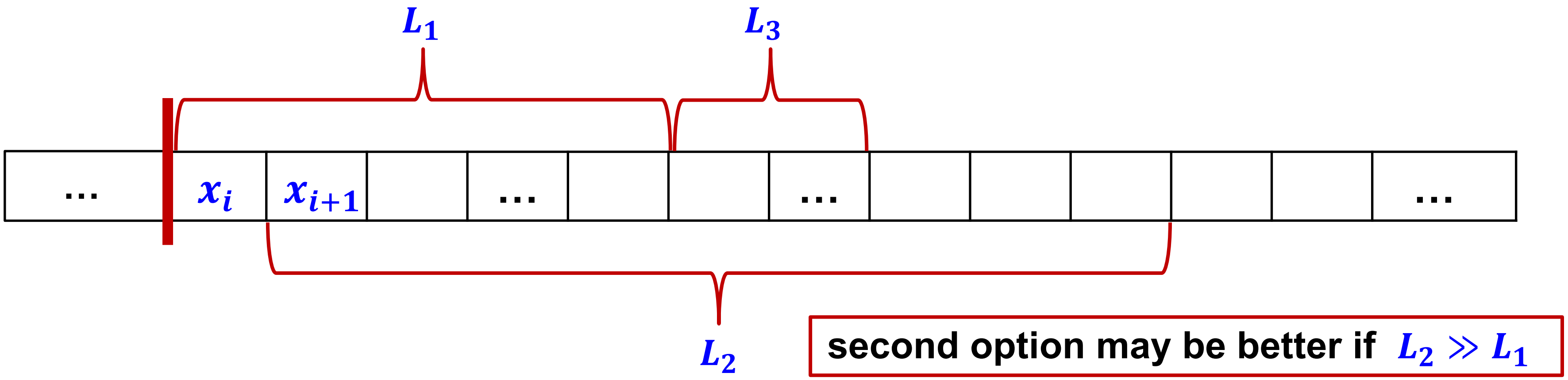


Code	Extra Bits	Dist	Code	Extra Bits	Dist	Code	Extra Bits	Distance
0	0	1	10	4	33-48	20	9	1025-1536
1	0	2	11	4	49-64	21	9	1537-2048
2	0	3	12	5	65-96	22	10	2049-3072
3	0	4	13	5	97-128	23	10	3073-4096
4	1	5,6	14	6	129-192	24	11	4097-6144
5	1	7,8	15	6	193-256	25	11	6145-8192
6	2	9-12	16	7	257-384	26	12	8193-12288
7	2	13-16	17	7	385-512	27	12	12289-16384
8	3	17-24	18	8	513-768	28	13	16385-24576
9	3	25-32	19	8	769-1024	29	13	24577-32768

A Huffman code over the alphabet $\{0,1,\ldots,29\}$ is used for these codes + an appropriate number of extra bits.

Encoding algorithm

- The description so far specifies how the encoded stream is *interpreted*, not how it is *generated*
- There are many ways to generate *gzip*-compliant streams
  - in case of multiple matches, prefer the closest one (smaller offsets will tend to have shorter Huffman codes)
  - matches described need not be *maximal* : *decoder will not complain!*
  - *lazy matching* :
    - find longest match from current position i, then check for longest match from position $i+1$
    - Choose the most economical encoding: describe match starting at position i, or describe x_i as literal + match starting at position $i+1$



second option may be better if $L_2 \gg L_1$

Some comparisons

- ☐ Input file: Don Quijote de la Mancha, HTML
  file size: 2,261,865 bytes

Compressor	Output bytes	bits/symbol
Huffman	1,284,323	4.54
vanilla LZ77	1,310,561	4.63
gzip -1	994,295	3.52
gzip -9	816,909	2.89