Stream Codes

Mário S. Alvim
(msalvim@dcc.ufmg.br)

Information Theory

DCC-UFMG
(2017/02)

Stream Codes - Introduction

- **Symbol codes** encode each symbol of the source individually, independently from the context in which they appear.
- **Stream codes** encode each symbol (or sequence of symbols) in a way that depends on the context in which the symbol (or sequence of symbols) appears.
- In this lecture we will discuss schemes of stream codes for data compression:
  - **Arithmetic coding**'s principle:

    "Know the source very well, and do an excellent job while compressing this particular source."
  - **Lempel-Ziv coding**'s principle:

    "Be universal, and do a reasonable job for any source."

Arithmetic Coding

Arithmetic coding - The guessing game

- Let's motivate the principle of arithmetic coding with a guessing-game example.

- **Example 1** (The guessing game.) Consider the English alphabet $\mathtt{A}, \mathtt{B}, \mathtt{C}, \ldots, \mathtt{Z}$ and the space symbol "–".

  The player repeatedly tries to guess the next character in a text file, the only feedback being whether or not the guess was a hit.

  We keep track of the number of guesses needed to correctly guess each character in the file.

T	H	E	R	E	-	I	S	-	N	O	-	R	E	V	E	R	S	E	-	O	N	-	A	-	M	O	T	O	R	C	Y	C	L	E	-
1	1	1	5	1	1	2	1	1	2	1	1	15	1	17	1	1	1	2	1	3	2	1	2	2	7	1	1	1	1	4	1	1	1	1	1

Arithmetic coding - The guessing game

- Example 1 (Continued)

  The string of numbers

  "1, 1, 1, 5, 1, 1, 2, 1, 1, 2, 1, 1, 15, 1, 17, 1, 1, 1, 2, 1, 3, 2, 1, 2, 2, 7, 1, 1, 1, 1, 4, 1, 1, 1, 1, 1"

  obtained is a representation of the original string with respect to the guessing strategy used by the player.

  The player encoded a string on the alphabet $\{\mathtt{A}, \mathtt{B}, \mathtt{C}, \ldots, \mathtt{Z}, -\}$ into a string on the alphabet $\{1, 2, 3, \ldots, 27\}$ marking when the right guess occurred for each character.

  The string of numbers is highly redundant (the frequency of numbers is far from uniform, so its entropy is low), and hence it can be efficiently compressed and decompressed.

  But how to map the string of numbers back into the original string in English?

Arithmetic coding - The guessing game

- Example 1 (Continued)

To decode the string of numbers

"1, 1, 1, 5, 1, 1, 2, 1, 1, 2, 1, 1, 15, 1, 17, 1, 1, 1, 2, 1, 3, 2, 1, 2, 2, 7, 1, 1, 1, 1, 4, 1, 1, 1, 1, 1":

1. Ask an identical twin of the guesser (one who thinks exactly like the other one and uses the same guessing strategy), and ask her to repeat the process of guessing a string character by character.
2. Listen to her guesses in order, and stop her when the guess matches the number you have in your number string: that will bring you back to the original English sentence!

Arithmetic coding - The guessing game

- Example 1 (Continued)

To get away with the need of twins, you can make the problem more formal by modeling the source as a strategy that pre-fixes guesses according to a context of size L letters.

Contexts of 3 letters	Guesses: 1	2	3	4	···	27
AAA	A	B	C	D	···	–
AAB	A	B	C	D	···	–
···	···	···	···	···	···	···
APP	L	–	R	A	···	Q
···	···	···	···	···	···	···
THE	–	R	Y	M	···	T
···	···	···	···	···	···	···
ZZZ	Z	Y	X	W	···	A

E.g., if the context is the trigram "THE", your first guess for the next symbol would be "–" (to complete the word "THE"), your second guess would be "R" (to form "THER"), etc.

◁

Arithmetic codes

- **Arithmetic codes** generalize the above example.
- Arithmetic codes need an explicit **probabilistic model** of the source, given by the conditional distribution

$$p(x_n = a_i \mid x_1, x_2, \ldots, x_{n-1}).$$

This model will be used both for compression and for decompression.

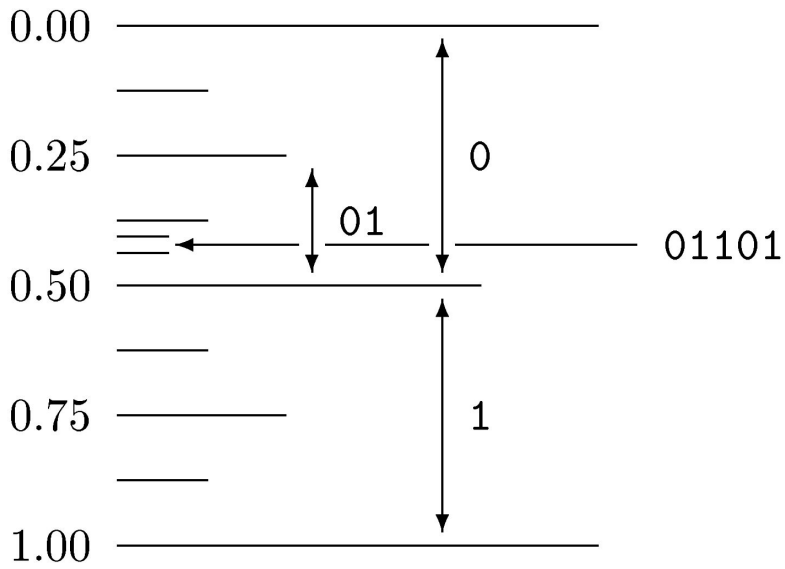
Arithmetic codes

- Before we model the source, note that we can define intervals within the real line $[0, 1)$ using binary notation:
  1. $[0.01, 0.10) = [1/4, 1/2) = [0.25, 0.50)$
- We use the convention that a binary string b represents the interval

$$[\ 0.b,\ \ 0.(b+1)\ \ ).$$

Examples:

1. 0 represents $[0.0, 0.1) = [0, 1/2)$
2. 010 represents $[0.010, 0.011) = [1/4, 3/8)$
3. 1101 represents $[0.1101, 0.1110) = [13/16, 14/16)$

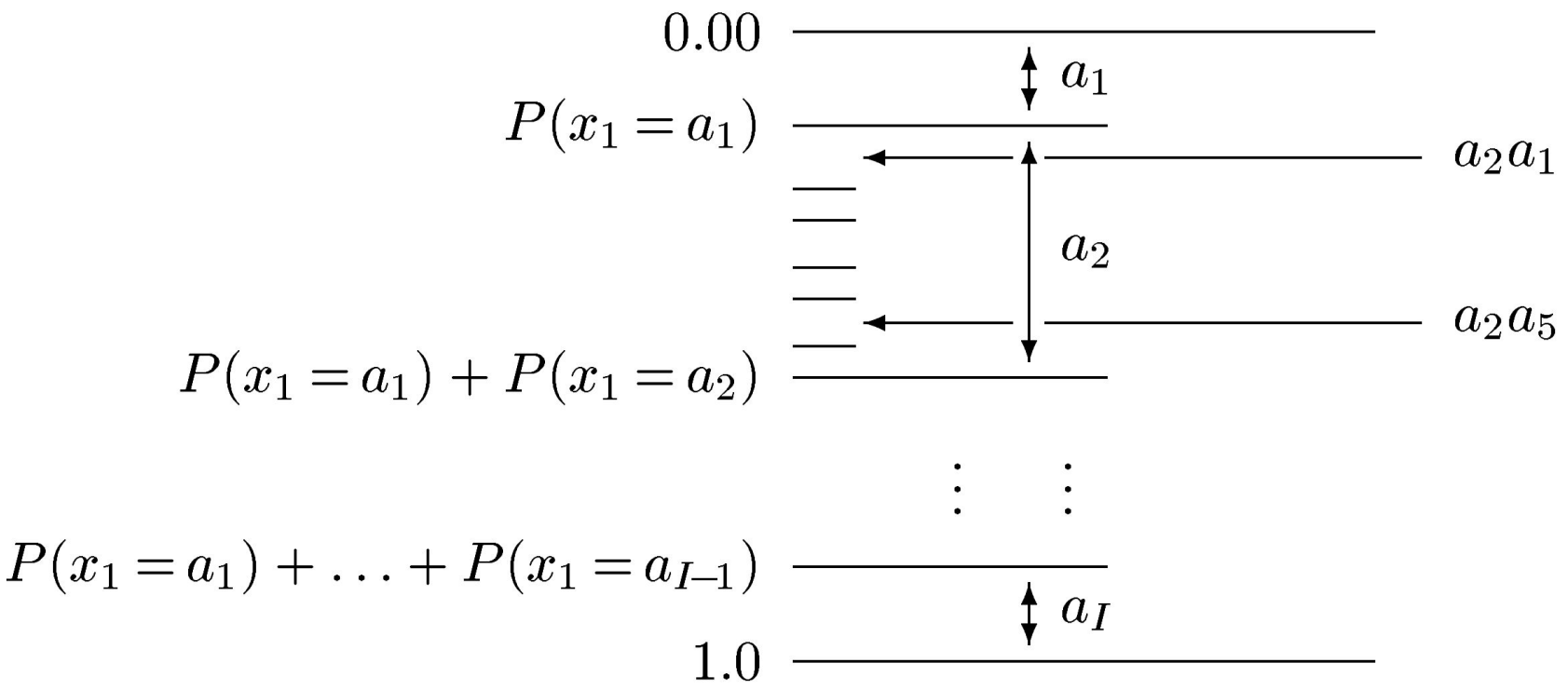


Arithmetic codes

- We can represent our probabilistic model of the source

$$p(x_n = a_i \mid x_1, x_2, \ldots, x_{n-1})$$

as a sequence of real intervals in $[0, 1)$.



Arithmetic codes

- To encode a string $x_1, x_2, \ldots, x_N$, we locate the interval corresponding to $x_1, x_2, \ldots, x_N$, and send a binary string whose intervals lies within that interval.

  This encode is performed on the fly, as the next example demonstrates.

Arithmetic codes

- Example 2 (Compressing the tosses of a bent coin.) A bent coin is tossed a number of times.

  At each time the outcome is either a or b, or an end of file symbol $\square$.

  We can think of this experiment as a source X producing a string on the alphabet $\mathcal{A}_X = \{\mathtt{a}, \mathtt{b}, \square\}$.

  Let us encode the source string

  $$\text{“}\mathtt{bba}\square\text{”}.$$

Arithmetic codes

- Example 2 (Continued)

The first thing is to have probabilistic model of the source.

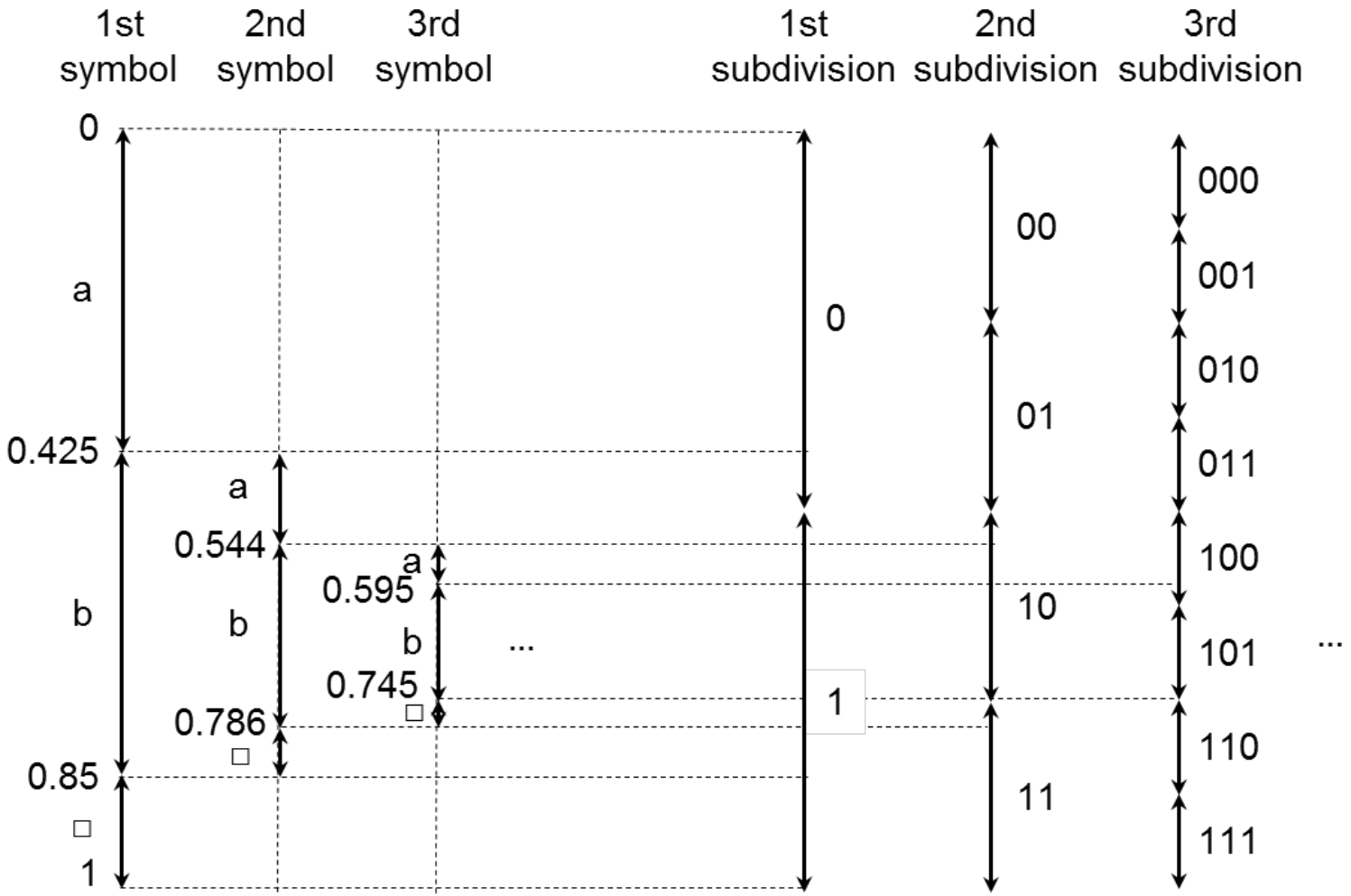
Using Laplace's method (Section 3.2), we can come up with the following model.

Context (sequence thus far)	Probability of next symbol		
	$P(\mathtt{a}) = 0.425$	$P(\mathtt{b}) = 0.425$	$P(\square) = 0.15$
b	$P(\mathtt{a} \mid \mathtt{b}) = 0.28$	$P(\mathtt{b} \mid \mathtt{b}) = 0.57$	$P(\square \mid \mathtt{b}) = 0.15$
bb	$P(\mathtt{a} \mid \mathtt{bb}) = 0.21$	$P(\mathtt{b} \mid \mathtt{bb}) = 0.64$	$P(\square \mid \mathtt{bb}) = 0.15$
bbb	$P(\mathtt{a} \mid \mathtt{bbb}) = 0.17$	$P(\mathtt{b} \mid \mathtt{bbb}) = 0.68$	$P(\square \mid \mathtt{bbb}) = 0.15$
bbba	$P(\mathtt{a} \mid \mathtt{bbba}) = 0.28$	$P(\mathtt{b} \mid \mathtt{bbba}) = 0.57$	$P(\square \mid \mathtt{bbba}) = 0.15$

Arithmetic codes

- Example 2 (Continued)

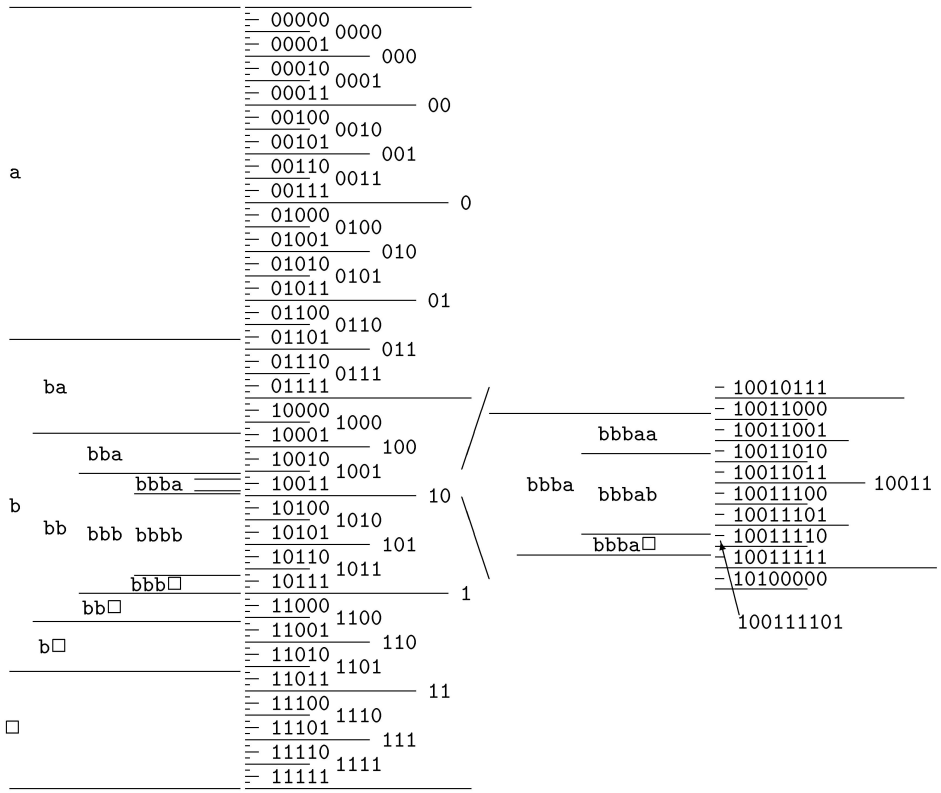
The model creates the following arithmetic code for string "bba□".



Arithmetic codes

- Example 2 (Continued)

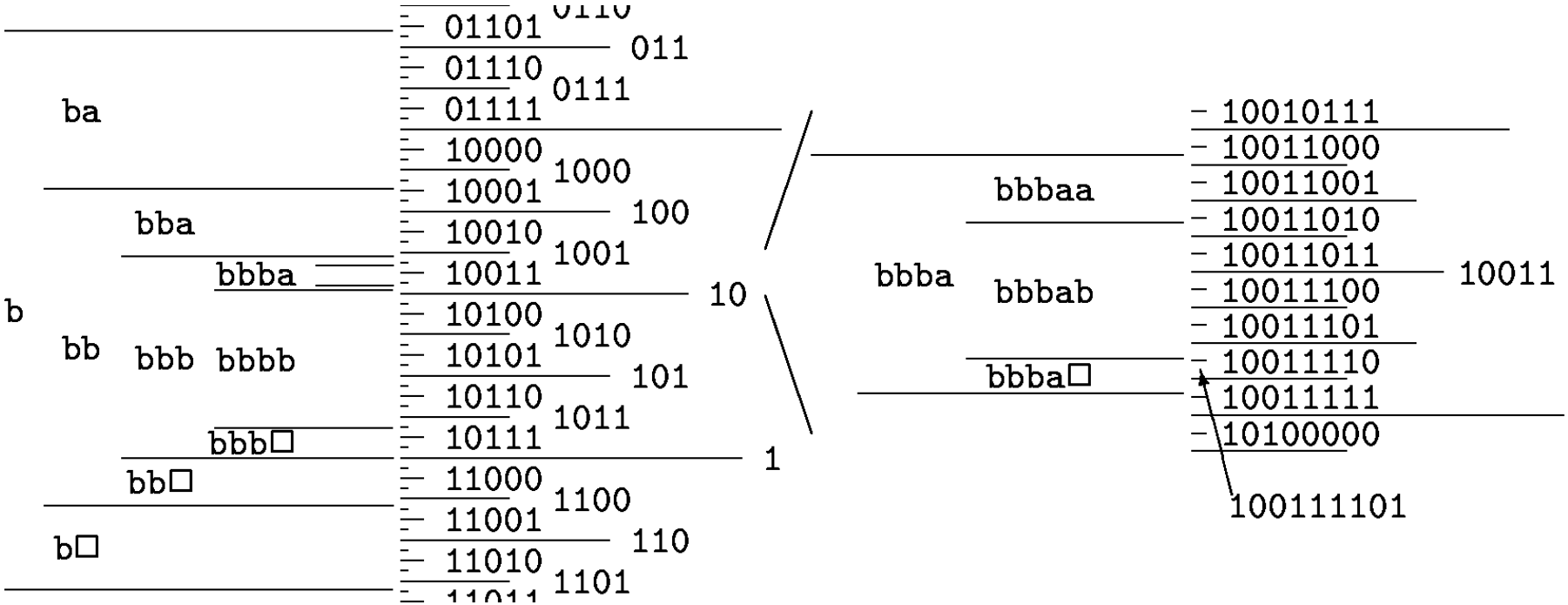
Continuing the expansion we get the following encoding.



Arithmetic codes

- Example 2 (Continued)

Zooming in we get



And the encoding for "bbba□" is 100111101.

Arithmetic codes - Application for text input

- An interesting application of Arithmetic codes is text production.
- Compression is, intuitively, a mapping from long texts to short sequences of bits.

  Efficient text production can be see as the opposite direction: a mapping from a short sequence of gestures to a long text.
- The Dasher [http://www.inference.phy.cam.ac.uk/dasher/] uses Arithmetic codes to convert up and down movements (which can be seen as bits 0 and 1) into long text productions.

Lempel-Ziv Coding

Lempel-Ziv coding

- The **Lempel-Ziv codings** are widely used for compression (e.g., by the `gzip` command).
- It does not need an explicit model for the source, it learns on the fly.

  This means that the method is general, but not always optimal.

Lempel-Ziv coding

- The basic Lempel-Ziv conding works as follows:

  1. Read the string to be compressed from left to right, and parse it in a sequence of substrings that have not yet appeared.

     This parsing will create a **dictionary** for the string, representing all substrings that have occurred in the string.

     Note that during the parsing, a substring is added to the dictionary only if it is exactly one bit longer than some substring already in the dictionary.

     In other words, every substring of size n added to the dictionary has a prefix of size $n - 1$ already in the dictionary.

  2. Once the string has been parsed and the dictionary has been created, encode the string by replacing each of its parsed substrings as a pointer to its prefix in the dictionary plus the extra bit needed to complete its suffix.

- If the string is highly redundant, the dictionary will tend to be small, and most substrings don't need to be repeated, so you can just point to their representation in the dictionary, saving up a lot of bits in the encoding.

Lempel-Ziv coding

- Example 3 Let us use the Lempel-Ziv code to encode 1011010100010.

  First we parse the string, obtaining the sequence

  $$\lambda, 1, 0, 11, 01, 010, 00, 10,$$

  where each substring is a word in our dictionary, and λ represents the **empty word** (i.e., a word composed of no symbols).

  Then we index every word in the dictionary, and rewrite every substring in the original string as a pair of a pointer to its prefix in the dictionary and the extra bit for its suffix.

source substrings	λ	1	0	11	01	010	00	10
$s(n)$	0	1	2	3	4	5	6	7
$s(n)$ using 1 bit	0	1	–	–	–	–	–	–
$s(n)$ using 2 bits	00	01	10	11	–	–	–	–
$s(n)$ using 3 bits	000	001	010	011	100	101	110	111
(pointer, bit)	–	$(,1)$	$(0,0)$	$(01,1)$	$(10,1)$	$(100,0)$	$(010,0)$	$(001,0)$

Lempel-Ziv coding

- Example 3 (Continued)

  Our encoding is the concatenation of the pairs (pointer, bit), so we have as the final encoding the string 100011101100001000010.

  Note that in this example the original string wasn't shortened because it is not redundant.

  If the source string were redundant and long enough, we would achieve much better compression.

  ◁

Compression algorithms on the market

Compression algorithms in the market

- Some tools that use arithmetic coding:
  1. DjVu
  2. JBIG
  3. PPM
- Tools that use Lempel-Ziv coding:
  1. `gzip`
  2. `compress`

Summary on Data Compression

Summary on compression

- **Fixed-length (lossy) block codes** (Chapter 4):
  - Mappings from a fixed number of source symbols to a fixed-length binary message.
  - Only a tiny fraction of source strings are given a codeword, but the ones that are not given a codeword are not common strings anyway, so there is little loss.
  - These codes were useful to identify entropy as a measure of compressibility, but they are not of much practical use.

Summary on compression

- **Symbol codes** (Chapter 5):
  - Employ a variable-length code for each symbol in the source alphabet. Codelengths are integers determined by the probabilities of the symbols.
  - **Huffman's algorithm** produces an optimal symbol code.

    The code is uniquely decodable and instantaneous, and the expected number of bits used per symbol of the font is in between $H(X)$ and $H(X + 1)$.
  - If the distribution used for compression differs from the real distribution of the source, the extra number of bits needed for compression will be given by the relative entropy (a.k.a. Kullback-Leibler divergence) between the two distributions.

Summary on compression

- **Stream codes** (Chapter 6):
  - These codes are not constrained to emit at least one bit for every symbol read from the source stream.

    Large numbers of source symbols may be coded into a smaller number of bits.
  - **Arithmetic codes** combine a probabilistic model with an encoding algorithm that identifies each string with a sub-interval [0,1) of size equal to the probability of that string under that model.

    These codes are almost-optimal, achieving almost $H(X)$ bits per symbol in compression.

    Their philosophy is that good compression depends good data modeling.
  - **Lempel-Ziv codes** are adaptive and memorize strings that have already occurred in the input file.

    Their philosophy is that we do not have any model for the input file to be compressed, and we should do a reasonable job whatever the distribution is.