Chapter 2: Coding for Discrete Sources

AAiT
Addis Ababa Institute of Technology
አዲስ አበባ ቴክኖሎጂ ኢንስቲትዩት
Addis Ababa University
አዲስ አበባ ዩኒቨርሲቲ

Graduate Program
School of Electrical and Computer Engineering

Coding for Discrete Sources

- We have seen that
  - Entropy H(X) of a source represents the average amount of information emitted by the source
  - Coding is the processes of representing the source output by a sequence of binary digits
- Knowledge of H(X) does not directly help us in the design of a coding algorithm
  - However, it provides a **measure of efficiency** of a source-encoding method by comparing the average number of binary digits per source letter to the entropy of the source

Coding for Discrete Sources …

- Encoding is simplified when the source is assumed to be ***discrete memoryless source*** (DMS)
  - I.e., symbols from the source are ***statistically independent*** and each symbol is encoded separately
- Few sources closely fit this idealized model
- We will see:
  1. Fixed-length vs. variable length encoding
  2. Blocks of symbols vs. symbol-by-symbol encoding
- It will be shown that, it is always efficient to encode ***blocks*** of symbols instead of each symbol separately

Coding for DMS

- A DMS produces an output letter or symbol every τ_s sec.
- Each letter is selected from an alphabet of symbols x_i, i=1,2,…..L, occurring with probabilities $p(x_i)$
- Entropy of the DMS

$$H(X) = -\sum_{i=1}^{L} p(x_i)\log_2 p(x_i) \leq \log_2 L$$

- Where equality holds when the symbols are *equally probable*
- The average number of source letters is H(X) and the ***source rate*** is defined as

$$\frac{H(X)}{\tau_s}$$

Coding for DMS - Fixed-Length Codewords

- Consider an encoding scheme where a unique set of R binary digits (codeword) is assigned to each symbol (letter)
  - R defines the code rate in bits/symbol
- If there are L symbols, the number of binary digits per source symbol required for unique encoding is given by

$$\mathbf{R} = \log_2 \mathbf{L}, \quad \text{When L is a power of 2}$$

OR

$$\mathbf{R} = \lfloor \log_2 \mathbf{L} \rfloor + 1, \quad \text{When } \mathbf{L} \text{ is not a power 2 and } \lfloor \mathbf{X} \rfloor \text{ denotes the largest integer less than } \mathbf{X}$$

- Since $H(X) \leq \log_2 L$, it follows that the code rate R bits/symbol is greater than average entropy H(X)
- *Thus H(X) is the lower bound of the rate R*

Coding for DMS - Fixed-Length Codewords …

- The efficiency of encoding is the ratio H(x)/R
- Note that
    1. If L is a power of 2 and the source letters are equally probable R = H(X) and the code is 100% efficient
    2. However, if L is not a power of 2 but the source letters are still equi-probable, R differs from H(X) by at most 1 bit per symbol
- When L is large, the efficiency can be high
- On the other hand, when L is small the encoding efficiency of fixed-length code can be increased by encoding a ***sequence of J letters*** at a time

Coding for DMS - Fixed-Length Codewords …

- Using a sequence of N binary digits, we can encode 2^N possible source symbols uniquely
- N must be selected such that:
  - $N \geq J \log_2 L$ or
  - $N = \lfloor J \log_2 L \rfloor + 1$ depending on whether L is a power of 2 or not
- The average number of bits per source symbol is $N/J = R$
- Hence, the inefficiency is reduced by approximately a factor of 1/J relative to the symbol-by-symbol encoding
- By making J sufficiently large the encoding efficiency measured by $JH(X)/N$ can be made as close to unity as desired

Coding for DMS - Fixed-Length Codewords …

- Such encoding does not introduce ***any distortion*** since the encoding of source symbols or blocks of symbols into codewords is unique
- Such encoding is referred to as ***noiseless***
- Suppose we reduce the code rate R by relaxing the condition that the encoding process be unique
  - This results in decoding failure

Source Coding Theorem I (Shannon, 1948)

- Let X be ensemble of letters from a DMS with entropy H(x)
- Consider blocks of symbols are encoded into codewords of length N from a binary alphabets
- For ε > 0 the probability of error p_e of a block decoding failure can be made arbitrarily small if

$$R = \frac{N}{J} \geq H(X) + \varepsilon \quad \textbf{and J is sufficient ly large}$$

- Conversely if

$$R \leq H(X) - \varepsilon$$

- Then p_e becomes arbitrarily close to one as J is made sufficiently large

Variable Length Codewords

- When source symbols are not equally probable, a more efficient coding method is to use variable length codewords
- Motivation for variable-length codes is the ability to achieve data compression by ***representing more probable symbols by shorter*** bit sequences (see Morse Code)

Variable Length Codewords …

- Variable-length source code C maps each source letter or symbol to a binary sequence C(x) with codeword length $l(x)$
- Codewords are transmitted as a continuous sequence of bits with no demarcation of codeword boundaries
- The decoder, once given the starting point, must determine the codeword boundaries
- The system requires buffers at the input and output sides of the synchronous channel and there are possibilities of buffer overflow

Variable Length Codewords …

- Thus unique decodability requires
  1. Initial synchronization and
  2. The condition that $C(x) \neq C(x')$ for each *x different from x'*
- For any source symbols $x_1, x_2, \ldots x_n$, the concatenation of codewords $C(x_1)\, C(x_2) \ldots C(x_n)$ differs from the concatenation of the codewords $C(x_1')\, C(x_2') \ldots C(x_n')$ for any other string $x_1', x_2' \ldots x_n'$
  - (Note that there are no commas in between the encoded bit sequences!)

Variable Length Codewords …

- Consider the alphabet X = {a,b,c} that may coded as C(a) = 0, C(b) = 1, and C(c) = 01
- This code is not uniquely decodable since the string 01 may be decoded as (a,b) or (c)
- Note that in the above code, the code for (a) is a **prefix** of the code for (c) and the code is said to be NOT prefix free
- Now consider the variable-length codes shown next for a four-symbol source

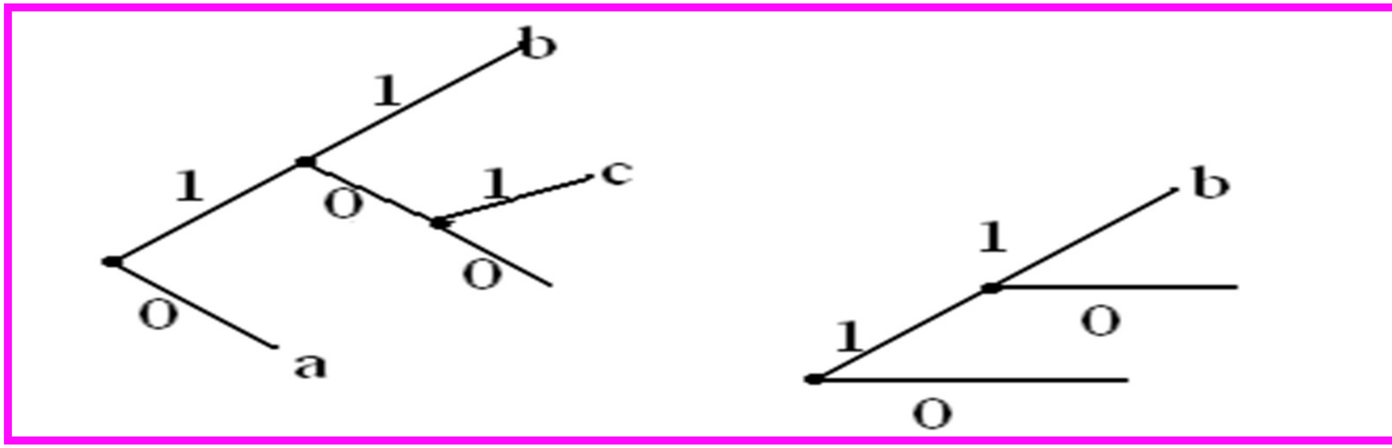
letter	$P(x_k)$	Code I	Code II	Code III
a_1	0.500	1	0	0
a_2	0.250	00	10	01
a_3	0.125	01	110	011
a_4	0.125	10	111	111

Variable Length Codewords …

- Code I is a variable length code that has a basic defect
- Consider the sequence 001001
- This can be decoded as $a_2a_4a_3$ or $a_2a_1a_2a_1$
  - *It is not uniquely decodable*
- This might be decoded uniquely if we have more bits which will involve delay and render the code not instantaneously decodable

Variable Length Codewords …

- Code II is uniquely and instantaneously decodable
- 0 indicates end of a codeword for the first three codewords and no code is longer than three bits
- Not also that it satisfies the prefix-free condition; that is for a code $C_k = (x_1, x_2, \ldots . x_k)$ there is no other codeword $C_l(x) = (x_{1,} x_2, \ldots .. X_l)$ for $1 \leq l \leq k\text{-}1$
- Code III is neither uniquely decodable nor instantaneously decodable



a → 0 | a → 0
b → 11 | b → 11
c → 101 | c → 10

- Prefix-free condition ensures that each codeword corresponds to a leaf node, since any intermediate node represent a prefix of any leaf stemming from it
- Note the first code tree is not full since the string 100 does not represent a codeword
- This can be shortened without destroying the prefix-free property as in the second tree diagram, which is full

Variable Length Codewords …

- A prefix-free code can be decoded by simply reading a string or a sequence from left to right and following the corresponding path in the code tree until it reaches a leaf, which represents a codeword by the prefix free property
- Proceed after stripping off the first codeword
- Consider decoding the string or sequence 1010011 using the second code above: 10 → c;10 → c; 0 → a; 11 → b
- Thus the sequence is decoded into ccab and there cannot be any other set of letters into which the sequence can be decoded
- Further, note that the code can be decoded essentially without delay

Variable Length Codewords …

- Devise a systematic procedure for constructing uniquely decodable variable length codes that are efficient in the sense that the average number of bits per source symbol or letter, given as

$$\bar{\mathbf{R}} = \sum_{k=1}^{L} n_k \, p(a_k)$$

- is minimized
- Note that n_k is length of the codeword k

Kraft Inequality

- A necessary and sufficient condition for the existence of a binary code with codewords having lengths $n_1 \leq n_2 \leq n_3 \ldots\ldots \leq n_L$ that satisfy the prefix (free) condition is

$$\sum_{k=1}^{L} 2^{-n_k} \leq 1$$

- Alternatively, every prefix-free code with codeword lengths $n_1 \leq n_2 \leq n_3 \ldots\ldots \leq n_L$ satisfies the above inequality
- And conversely, if the above inequality is satisfied, then a prefix-free code with code lengths n_k exists

Kraft Inequality …

- In addition, every full prefix-free code satisfies the above condition with equality whereas every non-full prefix-free code satisfies it with strict inequality (*see proof in the text)*
- Note that the Kraft inequality tells us whether it is possible to construct a prefix-free code for a given source alphabet with a set of codeword length, n_k
- Example: A full prefix-free code for an alphabet size 3 with codeword lengths {1, 2, 2} exists, but there is no prefix-free code with codeword lengths {1, 1, 2} since this does not satisfy the Kraft inequality

Source Coding Theorem

- Let X be ensemble of letters from a DMS with entropy H(x)
- It is possible to construct a code that satisfies the prefix condition and has an average length that satisfies the inequalities

$$H(X) \le \bar{R} \le H(X) + 1$$

- For lower bound consider codewords of length n_k, $1 \le k \le L$

$$H(X) - \bar{R} = \sum_k p_k \log \frac{1}{p_k} - \sum_k n_k p_k$$

$$= \sum_k p_k \log_2 \frac{2^{-n_k}}{p_k} \quad \text{and using } \ln x \le x - 1$$

$$H(X) - \bar{R} \le (\log_2 e) \sum_k p_k \left(\frac{2^{-n_k}}{p_k} - 1\right) \le (\log_2 e) \sum_k (2^{-n_k} - 1) \le 0$$

Huffman Coding Algorithm

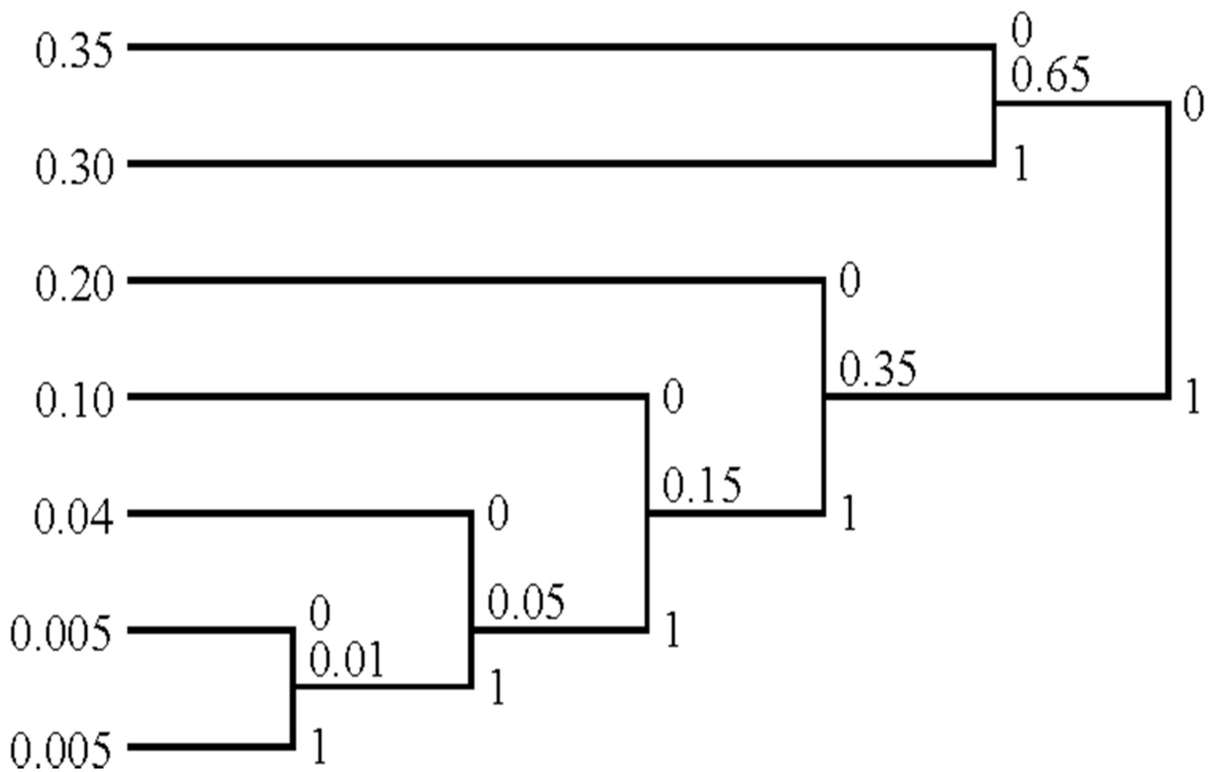
- The Huffman algorithm is a variable-length coding scheme based on the source letter probabilities p_k, k = 1, 2, ….., L
- The coding algorithm is optimum in the sense that the average number of binary digits required to represent the source letters is minimum
- This is subject to the constraint that they satisfy the prefix condition and the sequence of codewords are uniquely and instantaneously decodable

Huffman Coding Algorithm …

1. Order the symbols in decreasing order of probabilities
2. Add two lowest probabilities
3. Reorder probabilities
4. Break ties in any way you want
5. Assign 0 to top branch and 1 to bottom branch (or vice versa)
6. Continue until we have only one probability equal to 1

Huffman Coding Algorithm …

- Example 1: Given a DMS with seven source letters x_1, x_2, …..x_7 with probabilities 0.35, 0.30, 0.20, 0.10,0.04,0.005, 0.005, respectively
- Order the symbols in decreasing order of probabilities



Huffman Coding Algorithm …

Letter	Prob.	I(x)	Code
x_1	0.35	1.5146	00
x_2	0.30	1.7370	01
x_3	0.20	2.3219	10
x_4	0.10	3.3219	110
x_5	0.04	4.6439	1110
x_6	0.005	7.6439	11110
x_7	0.005	7.6439	11111

$H(X) = \sum p(x_i)I(x_i) = 2.11$bits/sym $\bar{R} = \sum p(x_k)\, n_k = 2.21$bits/sym

Efficiency $$\eta = \frac{H(X)}{\bar{R}} = \frac{2.11}{2.21} \times 100\% = 95\%$$

Huffman Coding Algorithm …

- The above code is not necessarily unique
- We can devise an alternative code as shown in the following for the same source as above

  X_1______0

  X_2______10

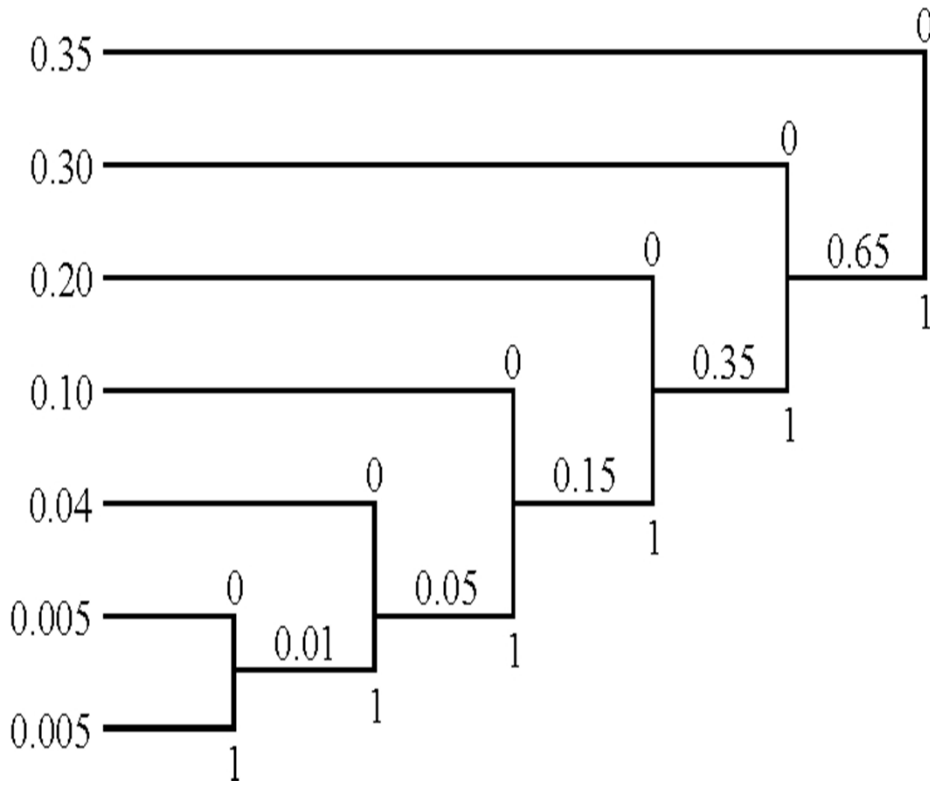
  X_3______110

  X_4______1110

  X_5______11110

  X_6______111110

  X_7______111111



- The average codeword length is the same as above (show?)

Huffman Coding Algorithm …

- Note that the assignment of 0 to the upper branch and 1 to the lower branch is arbitrary and by reversing this we obtain an equally efficient code that satisfies the prefix condition
- The above procedure always results in a prefix free variable length code that satisfy the bounds on the average length codeword R
- However, as discussed earlier, an efficient procedure is to encode J letters or symbols at a time

Huffman Coding Algorithm …

- As an illustration consider the following example

- Example: Let the output of a DMS consist of x_1, x_2 and x_3 with probabilities 0.45, 0.35, 0.2, respectively
- Entropy of the source

$$H(X) = -\sum_{K=1}^{3} p(x_k)\log_2 p(x_k) = 1.51 \textbf{ bits/symbol}$$

- If these are encoded individually
  - Using Huffman encoding procedure: x_1--1, x_2--00,and x_3--01 with an average codeword length of 1.55 and an efficiency of 97.7%

Huffman Coding Algorithm …

- If pairs of symbols are encoded using the Huffman algorithm, one possible variable length code can be as given next
- $2H(x) = 3.036$; $R_2 = 3.0675$ and the efficiency $\eta = (3.036/3.0675)\ 100\%$ $= 99\%$

Letter pairs	Prob.	I(X)	Code
$x_1 x_1$	0.2025	2.312	10
$x_1 x_2$	0.1575	2.676	001
$x_2 x_1$	0.1575	2.676	010
$x_2 x_2$	0.1225	3.039	011
$x_1 x_3$	0.090	3.486	111
$x_3 x_1$	0.090	3.486	0000
$x_2 x_3$	0.07	3.850	0001
$x_3 x_2$	0.07	3.850	1100
$x_3 x_3$	0.04	4.660	1101

Huffman Coding Algorithm …

- Example: Consider a discrete memoryless source X which has six symbols x_1, x_2, $x_{3,}$ $x_{4,}$ x_5 and x_6 with probabilities 0.45, 0.20, 0.12, 0.10, 0.09 and 0.04, respectively.
  1. Construct the Huffman code for X.
  2. Calculate the efficiency of the code.

Huffman Coding Algorithm …

- Example: A discrete memoryless source X has four symbols x_1, x_2, x_3 and x_4 with probabilities 0.4, 0.25, 0.19 and 0.16, respectively.
  1. Construct the Huffman code.
  2. Calculate the efficiency of the code.
  3. If pair of symbols are encoded using the Huffman algorithm, what is the efficiency of the new code? Compare the result with the one in part (2).

Lempel-Ziv Algorithm

- Huffman coding gives minimum average code length and satisfy the prefix condition
- To design Huffman coding, we need to know the probabilities of occurrence of all the source letter
  - In practice, the statistic of a source output are often unknown
  - Huffman coding methods in generally impractical for many sources
- Lempel-Ziv source coding algorithm is designed to be independent of the source statistics

Lempel-Ziv Algorithm – Operation

1. The sequence at the output of the discrete source is parsed into ***variable-length*** blocks, called *phrases*
2. A ***new phrase*** is introduced every time a block of letters from the source differs from some previous phrase in the ***last letter***
   - I.e., new phrase will be one of the minimum length that has not appeared before

- Example 1: Consider the binary sequence

  10101101001001110101000011001110101100011011

- Parsing the sequence results in the following phrases

  1,0,10,11,01,00,100,111,010,1000,011,001,110,101, 10001,1011

Lempel-Ziv Algorithm – Operation

3. The phrases are listed in a ***dictionary***, which stores the ***location*** of the existing phrases
   - Dictionary locations are numbered consecutively beginning with 1
4. In encoding a new phrase
   - Specify the location of the existing phrase in the dictionary &
   - Append the new letter
   - Location 0000 is used to encode a phrase that has ***not appeared previously***

- In the previous example 1,0,10, 11,01,00,100, 111,010,1000, 011, 001,110, 101,10001, 1011

Dictionary for Lempel-Ziv algorithm

	Dictionary location	Dictionary contents	Code word
1	0001	1	00001
2	0010	0	00000
3	0011	10	00010
4	0100	11	00011
5	0101	01	00101
6	0110	00	00100
7	0111	100	00110
8	1000	111	01001
9	1001	010	01010
10	1010	1000	01110
11	1011	011	01011
12	1100	001	01101
13	1101	110	01000
14	1110	101	00111
15	1111	10001	10101
16		1011	11101

Lempel-Ziv Algorithm - Operation …

- Example 1: Consider the binary sequence

  10101101001001110101000011001110101100011011

- Parsing the sequence results in the following phrases

  1,0,10,11,01,00,100,111,010,1000,011,001,110,101, 10001,1011

Lempel-Ziv Algorithm - Operation …

5. Codewords are determined by listing the dictionary location (in binary form) of the previous phrase that matches the new phrase in all but the last location
6. The new output letter is appended to the dictionary location of the previous phrase
7. The location 0000 is used to encode a phrase that has ***not appeared previously***
8. The source decoder constructs an ***identical copy of the dictionary*** and decodes the received sequence in step with the transmitted data sequence

Lempel-Ziv Algorithm - Operation …

- Example 1: Consider the binary sequence
  10101101001001110101000011001110101100011011
- Parsing the sequence results in the following phrases
  1,0,10,11,01,00,100,111,010,1000,011,001,110,101, 10001,1011
- Dictionary locations are numbered consecutively
  - Beginning with 1 and counting up, in this case to 16, which is the number of phrases in the sequence

Dictionary for Lempel-Ziv algorithm

	Dictionary location	Dictionary contents	Code word
1	0001	1	00001
2	0010	0	00000
3	0011	10	00010
4	0100	11	00011
5	0101	01	00101
6	0110	00	00100
7	0111	100	00110
8	1000	111	01001
9	1001	010	01010
10	1010	1000	01110
11	1011	011	01011
12	1100	001	01101
13	1101	110	01000
14	1110	101	00111
15	1111	10001	10101
16		1011	11101

Lempel-Ziv Algorithm …

- Lempel-Ziv Algorithm does not work well for short string
  - In the example, 44 source bits are encoded into 16 code words of 5 bits each, resulting in 80 coded bits
  - Hence, the algorithm provided no data compression at all
  - However, the inefficiency is due to the fact that the sequence we have considered is very short
- No matter how large the table is, it will eventually overflow
  - To solve the overflow problem, the source encoder and decoder must use an identical procedure to remove phrases from the dictionaries that are not useful and substitute new phrases in their place
- Often used in practice compress and uncompress utility
  - ZIP
  - "compress" and "uncompress" utilities in UNIX@ OS

Lempel-Ziv Algorithm …

- Example 2: Consider the binary sequence:

001101100011010101001001001101000001010010110010110

- Parsing the sequence as the following phrases:

0,01,1,011,00,0110,10,101,001,0010,01101,000,00101,001011,0010110

- Since we have 16 strings, we will need 4 bits

Lempel-Ziv Algorithm …

String	Position Number of this string	Dictionary Location	Prefix	Position Number of Prefix	Coded String
0	1	0001	emty	0000	00000
01	2	0010	0	0001	00011
1	3	0011	emty	0000	00001
011	4	0100	01	0010	00101
00	5	0101	0	0001	00010
0110	6	0110	011	0100	01000
10	7	0111	1	0011	00110
101	8	1000	10	0111	01111
001	9	1001	00	0101	01011
0010	10	1010	101	1000	10000
01101	11	1011			
000	12	1100			
00101	13	1101			
001011	14	1110			
0010110	15	1111			