Source coding reduces redundancy and hence reduces bandwidth requirement.

For the source encoder to be efficient, we require the knowledge of the statistics of the source.
Consider a source with four possible symbols $x_1, x_2, x_3$ and $x_4$ having the probabilities of occurrence $P(x_1)=0.7$, $P(x_2)=0.2$, $P(x_3)=0.06$, $P(x_4)=0.04$. The symbol rate is 100 symbols/sec.

If fixed-length codeword is used, the symbols could be encoded as

- $x_1$ 00
- $x_2$ 01
- $x_3$ 10
- $x_4$ 11

The average codeword length $L_{av}$ is 2 and the data rate is 200bps.
The entropy of the source is

\[ H = -\sum_{i=1}^{4} P(x_i) \log_2 \left( P(x_i) \right) = 1.25 \text{ bits/symbol} \]

If variable-length codeword is used, the symbols could be encoded as

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Codeword</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x_1 ):</td>
<td>0</td>
</tr>
<tr>
<td>( x_2 ):</td>
<td>10</td>
</tr>
<tr>
<td>( x_3 ):</td>
<td>110</td>
</tr>
<tr>
<td>( x_4 ):</td>
<td>111</td>
</tr>
</tbody>
</table>

The average codeword length is

\[ L_{av} = \sum_{i=1}^{4} P(x_i)L_i = 0.7 \times 1 + 0.2 \times 2 + .. = 1.4 \text{ bits/symbol} \]

and the data rate is 140bps.
Shannon’s source coding theorem

Given a discrete memoryless source of entropy $H(X)$, the average codeword length $L_{av}$ for any distortionless source coding scheme is bounded as

$$L_{av} \geq H(X)$$

According to this theorem, the source entropy $H(X)$ represents a fundamental limit on the average number of bits per source symbol necessary to represent a discrete memoryless source in that it can be made as small as, but no smaller than the source entropy.
Source Efficiency

Source efficiency \( \eta = \frac{H(X)}{L_{av}} \)

\[ \eta \leq 1 \]

At the receiver,

Note: Symbol to Sequence or Sequence to Symbol is a one to one mapping
Fixed-length codewords

Suppose there are $M$ symbols and they are equally probable, the number of binary digits per symbol required for unique encoding is

$$L = \left\lfloor \log_2 M \right\rfloor$$

where $\left\lfloor x \right\rfloor$ denotes the smallest integer greater than $x$.

**Example:** Case I  
$M=4$,  
$L = \left\lfloor \log_2 4 \right\rfloor = 2$

Case II  
$M=5$,  
$L = \left\lfloor \log_2 5 \right\rfloor = 3$
Fixed-length codewords

As $H(X) = \log_2 M$ (symbols are equally probable), the source efficiencies are:

Case I \[ \eta = \frac{H(X)}{L} = \frac{2}{2} = 100\% \]

Case II \[ \eta = \frac{H(X)}{L} = \frac{2.3}{3} = 77\% \]
The efficiency of case II can be increased by encoding a sequence of $J$ ($J > 1$) symbols at a time.

Suppose the original source symbol are $x_1, x_2, x_3, x_4, x_5$

If $J=2$, the new symbols are

$$x_1 x_1, x_1 x_2, \ldots x_1 x_5, x_2 x_1, x_2 x_2 \ldots x_5 x_4, x_5 x_5$$ (25 symbols)

and then

$$\eta = \frac{H(X)}{L} = \log_2 25 / \left\lfloor \log_2 25 \right\rfloor = 4.6 / 5 = 93\%$$
Variable-length codewords

When the source symbols are not equally probable, a more efficient encoding method is to use variable-length codewords.

The probabilities of occurrence of the symbols are used to find the codewords -- called entropy coding.
Example

Discrete Memoryless Source (DMS)

Source symbols

\[ A, B, C, D \]

\[
\begin{align*}
P(A) &= 1/2 \\
P(B) &= 1/4 \\
P(C) &= 1/8 \\
P(D) &= 1/8
\end{align*}
\]

\[ H(X) = 1.75 \text{ bits/symbol} \]

The symbols can be coded in 3 different ways using variable-length code.
### Example

<table>
<thead>
<tr>
<th>Source Symbol</th>
<th>Probability of Occurrence</th>
<th>Code I</th>
<th>Code II</th>
<th>Code III</th>
</tr>
</thead>
<tbody>
<tr>
<td>(A)</td>
<td>0.5</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>(B)</td>
<td>0.25</td>
<td>00</td>
<td>10</td>
<td>01</td>
</tr>
<tr>
<td>(C)</td>
<td>0.125</td>
<td>01</td>
<td>110</td>
<td>011</td>
</tr>
<tr>
<td>(D)</td>
<td>0.125</td>
<td>10</td>
<td>111</td>
<td>111</td>
</tr>
<tr>
<td>(L_{av})</td>
<td>(H(X)=1.75)</td>
<td>1.5</td>
<td>1.75</td>
<td>1.75</td>
</tr>
</tbody>
</table>
Suppose we receive 001001……..  
Is this B,A,B,A,…… or B,D,C,………?

∴ Code I is not uniquely decodable.

Sometimes this ambiguity can be resolved by waiting for additional bits. But then the decoding is not instantaneous and not desirable in practice.
Code II

A:0  B:10  C:110  D:111

It is uniquely decodable and instantaneous.

Initial state

Code tree for code II
Code III

\[ \begin{align*}
A &: 0 \\
B &: 01 \\
C &: 011 \\
D &: 111
\end{align*} \]

It is uniquely decodable but not instantaneously decodable.

Code tree for code III
Instantaneously decodable

To achieve instantaneously decodable, no codeword in the code is a prefix of another codeword.

For a given code word \( C_k \) of length \( k \), \( C_k = [b_1 \ b_2 \ b_3 \ \cdots \ b_k] \)

\[ C_l = [b_1 \ b_2 \ b_3 \ \cdots \ b_l] \] is a prefix of \( C_k \quad l < k \)

**Example:** prefix of 011 are 01 and 0

**Example:** For Code III, \( A:0, \ B:01, \ C:011, \ D:111 \). Therefore, codewords for \( A \) and \( B \) are prefix of the codeword for \( C \).
Prefix coding

A prefix code is defined as a code in which no codeword is the prefix of any other codeword.

**Example:** Code II

\[
\begin{align*}
A &: 0 \\
B &: 10 \\
C &: 110 \\
D &: 111
\end{align*}
\]

A prefix code is always *uniquely decodable* and *instantaneous*.
Kraft inequality

A prefix code always satisfy the Kraft inequality

\[ \sum_{i=1}^{M} 2^{-l_i} \leq 1 \]

where \( l_i \) are the lengths of the codewords and \( M \) is the number of symbols.

Note: Kraft inequality does not tell us that a source code is a prefix code. Rather, it is merely a condition on the codeword lengths of the code and not on the code words themselves.
Example

**Code I**  
A:1  B:00  C:01  D:10

LHS of the Kraft inequality becomes

\[ \sum_{i=1}^{M} 2^{-l_i} = 2^{-1} + 2^{-2} + 2^{-2} + 2^{-2} = 1.25 \geq 1 \]

Therefore, it cannot be a prefix code.

**Code II**  
A:0  B:10  C:110  D:111

**Code III**  
A:0  B:01  C:011  D:111

LHS of the Kraft inequality is

\[ \sum_{i=1}^{M} 2^{-l_i} = 1 \]

Both codes II and III satisfy the Kraft inequality but only code II is a prefix code.
Prefix code

Given a discrete memoryless source of entropy $H(X)$, a prefix code can be constructed with an average codeword length $L_{av}$ which is bounded as follows:

$$H(X) \leq L_{av} \leq H(X) + 1$$

The left-hand bound is satisfied with equality (i.e. $H(X)=L_{av}$) when

$$P(x_i) = 2^{-l_i}$$

i.e. $l_i = -\log_2 P(x_i)$
Example

Consider a source with three possible symbol $x_1$, $x_2$ and $x_3$ having the probabilities of occurrence $P(x_1)=0.5$, $P(x_2)=0.25$, $P(x_3)=0.25$.

A prefix code for this source is $x_1:0 \quad x_2:10 \quad x_3:11$

This prefix code satisfies $l_i = -\log_2 P(x_i)$

$H(X) = - \sum_{i=1}^{3} P(x_i) \log_2 P(x_i) = 0.5 + 0.5 + 0.5 = 1.5$ bits/symbol

$L_{av} = \sum_{i=1}^{3} P(x_i)l_i = 0.5 + 0.5 + 0.5 = 1.5$ bits/symbol
Prefix code

However, the condition $l_i = -\log_2 P(x_i)$ can only be satisfied in some special situations. (such as the previous example)

This problem can be solved by using extended code (i.e. encoding a sequence of $n$ symbols at a time.)

Example:
For a 2-symbol source with $x_1$ and $x_2$
The symbol of its extended source with $n=2$ are
$x_1 \ x_1 \ , \ x_1 \ x_2 \ , \ x_2 \ x_1 \ and \ x_2 \ x_2$
Entropy of extended source

Suppose $P(x_1) = 0.9, P(x_2) = 0.1$

$$H(X) = -\sum_{i=1}^{2} P(x_i) \log_2 P(x_i) = 0.47 \text{ bits/symbol}$$

The symbol probabilities of the extended source are $P(x_1x_1) = 0.81, P(x_1x_2) = 0.09, P(x_2x_1) = 0.09, P(x_2x_2) = 0.01$

and the entropy of the extended code $H(X^2)$ is

$$H(X^2) = -\sum_{i=1}^{2} \sum_{j=1}^{2} P(x_ix_j) \log_2 P(x_ix_j) = 0.94 = 2H(X)$$

In general,

$$H(X^n) = nH(X)$$
Prefix code

Using the previous result,

\[ H(X) \leq L_{av} \leq H(X) + 1 \]

\[ \Rightarrow H(X^n) \leq L_{av}^n \leq H(X^n) + 1 \]

\[ \Rightarrow nH(X) \leq L_{av}^n \leq nH(X) + 1 \]

\[ \Rightarrow H(X) \leq L_{av}^n / n \leq H(X) + 1 / n \]

As \( n \) approaches infinity, we have

\[ \lim_{n \to \infty} \frac{1}{n} L_{av}^n = H(X) \]

Therefore, the average codeword length of an extended prefix code can be make as small as the entropy of the source. However, the price we have to pay for decreasing the average code-word length is increased decoding complexity.
Huffman Coding

Huffman codes is an important class of prefix codes.

1. Arrange the source symbols in descending order of probability

**Example**: Source symbols $A, B, C, D, E$ and probabilities are $P(A) = 0.4$, $P(B) = 0.2$, $P(C) = 0.2$, $P(D) = 0.1$, $P(E) = 0.1$

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0.4</td>
</tr>
<tr>
<td>B</td>
<td>0.2</td>
</tr>
<tr>
<td>C</td>
<td>0.2</td>
</tr>
<tr>
<td>D</td>
<td>0.1</td>
</tr>
<tr>
<td>E</td>
<td>0.1</td>
</tr>
</tbody>
</table>
2. Create a new source with one less symbol by combining (adding) the two symbols having the lowest probability.

3. Repeat step 1 and 2 until a single-symbol source is achieved.
4. Associate '1' and a '0' with each pair of probabilities so combined.

5. Encode each original source symbol into binary sequence generated by the various combinations, with the first combination as the least significant digit in the sequence.

Therefore, we have $A:1$, $B:01$, $C:000$, $D:0010$ and $E:0011$.
Tree Diagram

Initial state

\[
H(X) = -\sum_{i=1}^{m} P_i \log_2 (P_i)
\]

\[
= -0.4 \log_2 0.4 - 2[0.2 \log_2 0.2] - 2[0.1 \log_2 0.1]
\]

\[
= 2.12 \text{ bits/symbol}
\]

\[
L_{av} = 0.4(1)+0.2(2)+0.2(3)+0.1(4)+0.1(4) = 2.2 \text{ bit/symbol}
\]

\[
\eta = H(X)/L_{av} =0.964
\]
Run-length Coding

A drawback of the Huffman code is that it requires knowledge of a probabilistic model of the source.

There are some coding methods which do not require the exact probabilistic model.

Run length coding technique is particularly suitable for binary sources where one of the symbols (the '1' say) occurs very much less often then others, so that there are long runs of successive '0's (e.g. a scanned and digitized line drawing). In this case, it is more efficient to encode by counting the number of the consecutive '0's between '1's.
Example

If a binary source generates a sequence of 25 digits,
000001,0000000,0001,1,0000001
This sequence is encoded into a 15-digit sequence
101,111,011,000,110.
Encoding in the Lempel-Ziv algorithm is accomplished by parsing the source data stream into segments that are the shortest subsequences not encountered previously.

**Example:** 000101  110010  100101…

Assumed that the binary symbols 0 and 1 are already stored in that order in the code book, We can write

- Subsequences stored: 0, 1
- Data to be parsed: 000101  110010  100101…
Example

The encoding process begins at the left. With symbols 0 and 1 already stored, the shortest subsequence of the data stream encountered for the first time and not seem before is 00, so we write

Subsequences stored: 0, 1, 00
Data to be parsed: 0101 110010 100101…

Similarly, we have

Subsequences stored: 0, 1, 00, 01
Data to be parsed: 01 110010 100101…
Example

and then

Subsequences stored: 0, 1, 00, 01, 011
Data to be parsed: 10010 100101…

The process is continued until the given data stream has been completely parsed. Finally, we have

0, 1, 00, 01, 011, 10, 010, 100, 101
## Example

<table>
<thead>
<tr>
<th>Numerical Position</th>
<th>Subsequences</th>
<th>Numerical Representation</th>
<th>Binary encoded blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>11</td>
<td>0010</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>12</td>
<td>0011</td>
</tr>
<tr>
<td>3</td>
<td>00</td>
<td>42</td>
<td>1001</td>
</tr>
<tr>
<td>4</td>
<td>01</td>
<td>21</td>
<td>0100</td>
</tr>
<tr>
<td>5</td>
<td>011</td>
<td>41</td>
<td>1000</td>
</tr>
<tr>
<td>6</td>
<td>10</td>
<td>61</td>
<td>1100</td>
</tr>
<tr>
<td>7</td>
<td>010</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>101</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Lempel-Ziv coding

Lempel-Ziv algorithm uses fixed-length codes to represent a variable number of source symbol; this feature makes the Lempel-Ziv code suitable for synchronous transmission.

In practice, fixed block of 12 bits long are used, which implies a code book of 4096 entries.
Example

Source: English text

Huffman coding  compaction ratio 43%
Lempel-Ziv algorithm  compaction ratio 55%

Therefore, Lempel-Ziv algorithm is a better choice for compressing English text.

The reason for is that Huffman coding does not take advantage of the intercharacter redundancies of the language.