COMPRESSION AND CODING I

Ole-Johan Skrede

26.04.2017

INF2310 - Digital Image Processing

Department of Informatics
The Faculty of Mathematics and Natural Sciences
University of Oslo

After original slides by Andreas Kleppe

TODAY'S LECTURE

- Three steps of compression

TODAY'S LECTURE

- Three steps of compression
- Redundancy

TODAY'S LECTURE

- Three steps of compression
- Redundancy
- Coding and entropy

today's lecture

- Three steps of compression
- Redundancy
- Coding and entropy
- Shannon-Fano coding

TODAY'S LECTURE

- Three steps of compression
- Redundancy
- Coding and entropy
- Shannon-Fano coding
- Huffman coding

TODAY'S LECTURE

- Three steps of compression
- Redundancy
- Coding and entropy
- Shannon-Fano coding
- Huffman coding
- Arithmetic coding

TODAY'S LECTURE

- Three steps of compression
- Redundancy
- Coding and entropy
- Shannon-Fano coding
- Huffman coding
- Arithmetic coding
- Sections from the compendium:
    - 18.1 *Hva er kompresjon*
    - 18.2 *Kompresjonsprosessen*
    - 18.3 *Melding, data, informasjon - og kapasitet*
    - 18.5 *Litt om informasjonsteori og sannsynlighet*
    - 18.6 *Naturlig binærkoding*
    - 18.7 *Koding med variabel lengde*
        - 18.7.1 *Shannon-Fano koding*
        - 18.7.2 *Huffman koding*
        - 18.7.4 *Aritmetisk koding*
    - Appendic B *Prefiks i titallsystemet og i det binære systemet*

MOTIVATION

- Compression is used to *reduce the number of bits* that is used to describe a signal (or a good approximation of the signal).

MOTIVATION

- Compression is used to *reduce the number of bits* that is used to describe a signal (or a good approximation of the signal).
- It has a number of applications in storage and transmission of data
    - Video
    - Remote analysis / metereology
    - Surveillance / remote control
    - Tele medicine / medical archives (PACS)
    - Mobile communication
    - MP3 music, DAB radio, digital camera etc.

MOTIVATION

- Compression is used to *reduce the number of bits* that is used to describe a signal (or a good approximation of the signal).
- It has a number of applications in storage and transmission of data
  - Video
  - Remote analysis / metereology
  - Surveillance / remote control
  - Tele medicine / medical archives (PACS)
  - Mobile communication
  - MP3 music, DAB radio, digital camera etc.
- *Time consumption is important*, but compression time and decompression time can vary.
  - Asymmetric compression: when one is more important than the other.
  - Symmetric compression: when both share the same importance.

INTRODUCTION

SI-UNITS AND PREFIXES

- We will use the symbol b to denote a *bit* and B to denote a *byte* ($= 8\,\mathrm{b}$).

SI-UNITS AND PREFIXES

- We will use the symbol b to denote a *bit* and B to denote a *byte* $(= 8\,\text{b})$.
- Transfer speed and bandwidth capacity is *always* given with SI-prefixes

1 kbps	=	10^3bps	=	1	kilo bit per second
1 Mbps	=	10^6bps	=	1	mega bit per second
1 Gbps	=	10^9bps	=	1	giga bit per second
1 Tbps	=	10^{12}bps	=	1	terra bit per second

- File size is *always* given with *binary* prefixes

1 KiB	=	2^{10}B	=	1 024 B	=	1	kibi byte
1 MiB	=	2^{20}B	=	1 048 576 B	=	1	mebi byte
1 GiB	=	2^{30}B	=	1 073 741 824 B	=	1	gibi byte
1 TiB	=	2^{40}B	=	1 099 511 627 776 B	=	1	tebi byte

SPACE AND TIME REQUIREMENTS

Capacity for some standards

- 3G: At least 200 kbps
- ADSL2+: Max 24 Mbps
- VDSL2: Max 100 Mbps

Capacity for some standards

- 3G: At least 200 kbps
- ADSL2+: Max 24 Mbps
- VDSL2: Max 100 Mbps

Example 1: Digital 8-bit RGB image: $8\,\text{bit} \times 512 \times 512 \times 3 = 6\,291\,456\,\text{bit}$

Example 2: X-ray image : $12\,\text{bit} \times 7112 \times 8636 = 737\,030\,784\,\text{bit}$

	64 kbps capacity	1 Mbps capacity
Example 1	ca 1 min. 38 s.	ca 6 s.
Example 2	ca 3 h. 12 min.	ca 12 min.

COMPRESSION AND DECOMPRESSION

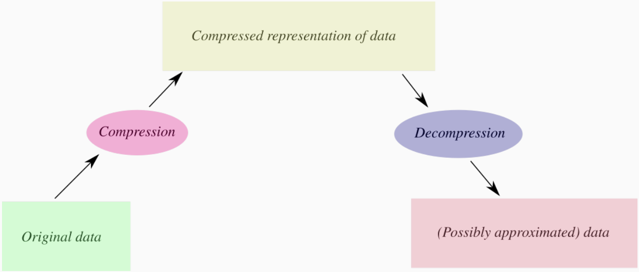


Figure 1: Compression and decompression pipeline

- We would like to compress our data, both to reduce storage and transmission load.
- In *compression*, we try to create a representation of the data which is smaller in size, while preserving vital information. That is, we throw away redundant information.
- The original data (or an approximated version) can be retrieved through *decompression*.

COMPRESSION

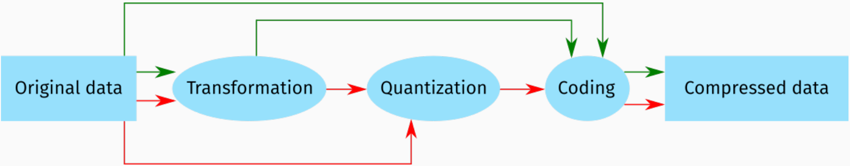


Figure 2: Three steps of compression

We can group compression in to three steps:

- **Transformation:** A more compact image representation.

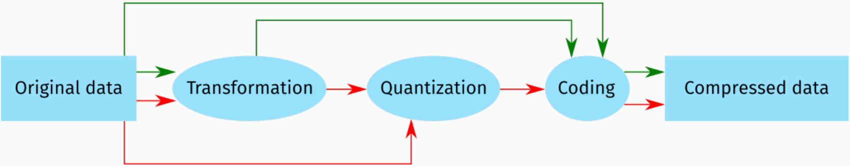


Figure 2: Three steps of compression

We can group compression in to three steps:

- **Transformation:** A more compact image representation.
- **Qunatization:** Representation approximation.

COMPRESSION

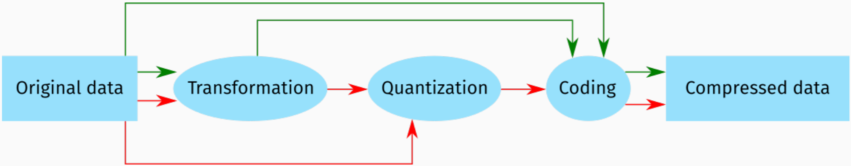


Figure 2: Three steps of compression

We can group compression in to three steps:

- **Transformation:** A more compact image representation.
- **Qunatization:** Representation approximation.
- **Coding:** Transformation from one set of symbols to another.
  - **Encoding:** Coding from an original format to some other format. E.g. encoding a digital image from raw numbers to JPEG.
  - **Decoding:** The reverse process, coding from some format to the original. E.g. decoding a JPEG image back to raw numbers.

LOSSY AND LOSSLESS COMPRESSION

Compression can either be *lossless* or *lossy*. There exists a number of methods for both types.

Lossless: We are able to perfectly reconstruct the original image.

Lossy: We can only reconstruct the original image to a certain degree (but not perfect).

THREE STEPS OF COMPRESSION

- Many compression methods are based on applying an image *transformation* in order to represent the image in a different way. Examples are the difference transform and the run-length transform.

THREE STEPS OF COMPRESSION

- Many compression methods are based on applying an image *transformation* in order to represent the image in a different way. Examples are the difference transform and the run-length transform.
- If we *quantize* the original (or transformed) image, this cannot be reversed, which implies a lossy compression.

THREE STEPS OF COMPRESSION

- Many compression methods are based on applying an image *transformation* in order to represent the image in a different way. Examples are the difference transform and the run-length transform.
- If we *quantize* the original (or transformed) image, this cannot be reversed, which implies a lossy compression.
- At the end, *encoding* is performed, which is some transformation to a binary representation. This is often based on normalized histograms.

- *Transforms* are allways *reversible*.
- *Quantizations* are *not reversible*.
- *Coding* is always *reversible*.

Signal: The signal that is to be stored or transmitted.

- A signal contains a certain amount of information.

Signal: The signal that is to be stored or transmitted.

- A signal contains a certain amount of information.

Information: A mathematical construct that quantify the amount of ”surprise” in a signal.

- An image with random pixel values contain more information than a monotone image.
- Object edges has typically high information content.

Signal: The signal that is to be stored or transmitted.

- A signal contains a certain amount of information.

Information: A mathematical construct that quantify the amount of ”surprise” in a signal.

- An image with random pixel values contain more information than a monotone image.
- Object edges has typically high information content.

Data:

- A bit-sequence representing the signal.

- *Symbol* X: a unit component character.

- *Symbol* X: a unit component character.
- *Alphabet* $\mathcal{X}$: the collection of all used symbols.

- *Symbol* X: a unit component character.
- *Alphabet* $\mathcal{X}$: the collection of all used symbols.
- *Codeword* y: a compressed, finite-length sequence of symbols. The compressed signal is composed of codewords. We denote the range of y (collection of possible codewords) as $\mathcal{Y}$, that is, $y \in \mathcal{Y}$.

- *Symbol* X: a unit component character.
- *Alphabet* $\mathcal{X}$: the collection of all used symbols.
- *Codeword* y: a compressed, finite-length sequence of symbols. The compressed signal is composed of codewords. We denote the range of y (collection of possible codewords) as $\mathcal{Y}$, that is, $y \in \mathcal{Y}$.
- *Source code* (or *code book*) c: a mapping between a symbol x and its codeword $y = c(x)$.

$$
\begin{aligned}
c &: \mathcal{X} \to \mathcal{Y} \\
&: x \mapsto c(x)
\end{aligned}
$$

Consider the source code defined by the following table

$\mathcal{X}$:	a	b	g	d	r	z	e	t
$\mathcal{Y}$:	000	001	010	011	100	101	110	111

such that $c(\texttt{d}) = \texttt{011}$ etc. In this case

- The uncompressed symbols x are from the alphabet $\mathcal{X}$.
- The compressed codewords y are composed of symbols from the alphabet $S_y = \{\texttt{0}, \texttt{1}\}$.
- Each codeword is limited to 3 bits, and we can therefore only have 8 possible words and codewords.
- If each word x has the same probability of occurance, this is optimally coded.

EXAMPLE, NATURAL BINARY CODING

Now, the symbols x can be whatever they like, so we include this example, which is the same as the above one, except with a different input alphabeth.

$\mathcal{X}$:	`alpha`	`beta`	`gamma`	`delta`	`rho`	`zeta`	`eta`	`theta`
$\mathcal{Y}$:	`000`	`001`	`010`	`011`	`100`	`101`	`110`	`111`

such that $c(\texttt{delta}) = \texttt{011}$ etc. In this case

- The uncompressed symbols x are from the alphabet $\mathcal{X}$.
- The compressed codewords y are composed of symbols from the alphabet $S_y = \{\texttt{0}, \texttt{1}\}$.

REDUNDANCY

- We can use a different amount of data on the same signal.

REDUNDANCY

- We can use a different amount of data on the same signal.
- E.g. the signal `13`
    - ISO 8859-1 (Latin-1): 16 bits: 8 bits for `1` (at `0x31`) and 8 bits for `3` (at `0x33`).
    - 8-bit natural binary encoding: 8 bits: `00001101`
    - 4-bit natural binary encoding: 4 bits: `1101`

REDUNDANCY

- We can use a different amount of data on the same signal.
- E.g. the signal `13`
  - ISO 8859-1 (Latin-1): 16 bits: 8 bits for `1` (at `0x31`) and 8 bits for `3` (at `0x33`).
  - 8-bit natural binary encoding: 8 bits: `00001101`
  - 4-bit natural binary encoding: 4 bits: `1101`
- **Redundancy:** What can be removed from the data without loss of (relevant) information.
- In compression, we want to remove redundant bits.

DIFFERENT TYPES OF REDUNDANCY

- *Psychovisual redundancy*
    - Information that we cannot percieve.
    - Can be compressed by e.g. subsampling or by reducing the number of bits per pixel.

DIFFERENT TYPES OF REDUNDANCY

- *Psychovisual redundancy*
  - Information that we cannot percieve.
  - Can be compressed by e.g. subsampling or by reducing the number of bits per pixel.
- *Inter-pixel temporal redundancy*
  - Correlation between successive images in a sequence.
  - A sequence can be compressed by only storing some frames, and then only differences for the rest of the sequence.

DIFFERENT TYPES OF REDUNDANCY

- *Psychovisual redundancy*
    - Information that we cannot percieve.
    - Can be compressed by e.g. subsampling or by reducing the number of bits per pixel.
- *Inter-pixel temporal redundancy*
    - Correlation between successive images in a sequence.
    - A sequence can be compressed by only storing some frames, and then only differences for the rest of the sequence.
- *Inter-pixel spatial redundancy*
    - Correlation between neighbouring pixels within an image.
    - Can be compressed by e.g. run-length methods.

DIFFERENT TYPES OF REDUNDANCY

- *Psychovisual redundancy*
    - Information that we cannot percieve.
    - Can be compressed by e.g. subsampling or by reducing the number of bits per pixel.
- *Inter-pixel temporal redundancy*
    - Correlation between successive images in a sequence.
    - A sequence can be compressed by only storing some frames, and then only differences for the rest of the sequence.
- *Inter-pixel spatial redundancy*
    - Correlation between neighbouring pixels within an image.
    - Can be compressed by e.g. run-length methods.
- *Coding redundancy*
    - Information is not represented optimally by the symbols in the code.
    - This is often measured as the difference between average code length and some theoretical minimum code length.

COMPRESSION RATE AND REDUNDANCY

- The *compression rate* is defined as the ratio between the *uncompressed* size and *compressed* size

$$\textit{Compression rate} = \frac{\textit{Uncompressed size}}{\textit{Compressed size}}$$

or as the ratio between the *mean number of bits per symbol* in the compressed and uncompressed signal.

COMPRESSION RATE AND REDUNDANCY

- The *compression rate* is defined as the ratio between the *uncompressed* size and *compressed* size

$$\textit{Compression rate} = \frac{\textit{Uncompressed size}}{\textit{Compressed size}}$$

  or as the ratio between the *mean number of bits per symbol* in the compressed and uncompressed signal.

- *Space saving* is defined as the reduction in size relative to the uncompressed size, and is given as

$$\textit{Space savings} = 1 - \frac{\textit{Compressed size}}{\textit{Unompressed size}}$$

- The *compression rate* is defined as the ratio between the *uncompressed* size and *compressed* size

$$\textit{Compression rate} = \frac{\textit{Uncompressed size}}{\textit{Compressed size}}$$

or as the ratio between the *mean number of bits per symbol* in the compressed and uncompressed signal.

- *Space saving* is defined as the reduction in size relative to the uncompressed size, and is given as

$$\textit{Space savings} = 1 - \frac{\textit{Compressed size}}{\textit{Unompressed size}}$$

- Example: An 8-bit 512×512 image has an uncompressed size of 256 kiB, and a size of 64 kiB after compression.
  - *Compression rate*: 4
  - *Space saving*: 3/4

- It is convenient to represent words as random variables X with an appropriate probability mass function p_X that describe the probability of occuring in a signal.

- It is convenient to represent words as random variables X with an appropriate probability mass function p_X that describe the probability of occuring in a signal.
- With this we can construct a source code with
    - shorter codewords (few symbols) to words with high probability, and
    - longer coodewords to words with low probability.

- It is convenient to represent words as random variables X with an appropriate probability mass function p_X that describe the probability of occuring in a signal.
- With this we can construct a source code with
  - shorter codewords (few symbols) to words with high probability, and
  - longer coodewords to words with low probability.
- That is, we use a variable number of symbols to encode the words.

VARIABLE LENGTH CODING EXAMPLE: MORSE CODE

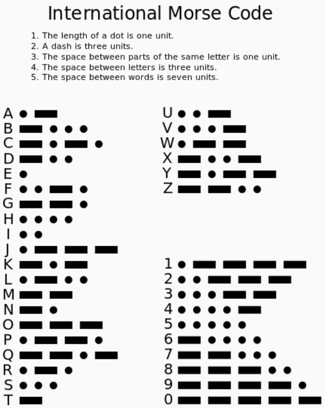


Figure 3: Morse code

- The morse code alphabet consist of four symbols: {a dot, a dash, a letter space, a word space}.
- Codeword length is approximately inversly proportional to the frequency of letters in the english language.

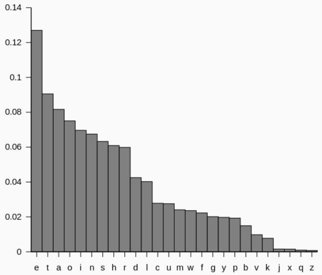


Figure 4: Relative letter frequency in the english language

EXPECTED CODE LENGTH

- The *expected length* L_c of a source code c for a random variable X with pmf. p_X is defined as

$$L_c = \sum_{x \in \mathcal{X}} p_X(x) l_c(x)$$

where $l_c(x)$ is the length of the codeword assigned to x in this source code.

EXPECTED CODE LENGTH

- The *expected length* L_c of a source code c for a random variable X with pmf. p_X is defined as
$$L_c = \sum_{x \in \mathcal{X}} p_X(x) l_c(x)$$
where $l_c(x)$ is the length of the codeword assigned to x in this source code.
- Example: Let X be a random variable taking values in $\{1, 2, 3, 4\}$ with probabilities defined by p_X below.

EXPECTED CODE LENGTH

- The *expected length* L_c of a source code c for a random variable X with pmf. p_X is defined as

$$L_c = \sum_{x \in \mathcal{X}} p_X(x) l_c(x)$$

where $l_c(x)$ is the length of the codeword assigned to x in this source code.

- Example: Let X be a random variable taking values in $\{1, 2, 3, 4\}$ with probabilities defined by p_X below.
- Let us encode this with a variable length source code c_v, and a source code c_e with equal length codewords.

$$\begin{aligned} p_X(1) &= \tfrac{1}{2} & c_v(1) &= \text{0} & c_e(1) &= \text{00} \\ p_X(2) &= \tfrac{1}{4} & c_v(2) &= \text{10} & c_e(2) &= \text{01} \\ p_X(3) &= \tfrac{1}{8} & c_v(3) &= \text{110} & c_e(3) &= \text{10} \\ p_X(4) &= \tfrac{1}{8} & c_v(4) &= \text{111} & c_e(4) &= \text{11} \end{aligned}$$

EXPECTED CODE LENGTH

- The *expected length* L_c of a source code c for a random variable X with pmf. p_X is defined as

$$L_c = \sum_{x \in \mathcal{X}} p_X(x) l_c(x)$$

where $l_c(x)$ is the length of the codeword assigned to x in this source code.

- Example: Let X be a random variable taking values in $\{1, 2, 3, 4\}$ with probabilities defined by p_X below.
- Let us encode this with a variable length source code c_v, and a source code c_e with equal length codewords.

$$\begin{aligned} p_X(1) &= \tfrac{1}{2} & c_v(1) &= \text{0} & c_e(1) &= \text{00} \\ p_X(2) &= \tfrac{1}{4} & c_v(2) &= \text{10} & c_e(2) &= \text{01} \\ p_X(3) &= \tfrac{1}{8} & c_v(3) &= \text{110} & c_e(3) &= \text{10} \\ p_X(4) &= \tfrac{1}{8} & c_v(4) &= \text{111} & c_e(4) &= \text{11} \end{aligned}$$

- Expected length of the variable length coding: $L_{c_v} = 1.75$ bits.

- The *expected length* L_c of a source code c for a random variable X with pmf. p_X is defined as

$$L_c = \sum_{x \in \mathcal{X}} p_X(x) l_c(x)$$

where $l_c(x)$ is the length of the codeword assigned to x in this source code.

- Example: Let X be a random variable taking values in $\{1, 2, 3, 4\}$ with probabilities defined by p_X below.
- Let us encode this with a variable length source code c_v, and a source code c_e with equal length codewords.

$$\begin{aligned} p_X(1) &= \tfrac{1}{2} & c_v(1) &= \text{0} & c_e(1) &= \text{00} \\ p_X(2) &= \tfrac{1}{4} & c_v(2) &= \text{10} & c_e(2) &= \text{01} \\ p_X(3) &= \tfrac{1}{8} & c_v(3) &= \text{110} & c_e(3) &= \text{10} \\ p_X(4) &= \tfrac{1}{8} & c_v(4) &= \text{111} & c_e(4) &= \text{11} \end{aligned}$$

- Expected length of the variable length coding: $L_{c_v} = 1.75$ bits.
- Expected length of the equal length coding: $L_{c_e} = 2$ bits.

INFORMATION CONTENT

- We use *information content* (aka *self-information* and *surprisal*) to measure the level of information in an event X.

INFORMATION CONTENT

- We use *information content* (aka *self-information* and *surprisal*) to measure the level of information in an event X.
- The information content $I_X(x)$ (measured in *bits*) of an event x is defined as

$$I_X(x) = \log_2 \frac{1}{p_X(x)}.$$

- It can also be measured in *nats*

$$I_X(x) = \log_e \frac{1}{p_X(x)},$$

  that is, with the natural logarithm, or in *hartleys*

$$I_X(x) = \log_{10} \frac{1}{p_X(x)}.$$

- We use *information content* (aka *self-information* and *surprisal*) to measure the level of information in an event X.
- The information content $I_X(x)$ (measured in *bits*) of an event x is defined as

$$I_X(x) = \log_2 \frac{1}{p_X(x)}.$$

- It can also be measured in *nats*

$$I_X(x) = \log_e \frac{1}{p_X(x)},$$

that is, with the natural logarithm, or in *hartleys*

$$I_X(x) = \log_{10} \frac{1}{p_X(x)}.$$

- We can change base of the logarithm from b to k with

$$\log_b(x) = \frac{\log_k(x)}{\log_k(b)}.$$

- We use *information content* (aka *self-information* and *surprisal*) to measure the level of information in an event X.
- The information content $I_X(x)$ (measured in *bits*) of an event x is defined as

$$I_X(x) = \log_2 \frac{1}{p_X(x)}.$$

- It can also be measured in *nats*

$$I_X(x) = \log_e \frac{1}{p_X(x)},$$

  that is, with the natural logarithm, or in *hartleys*

$$I_X(x) = \log_{10} \frac{1}{p_X(x)}.$$

- We can change base of the logarithm from b to k with

$$\log_b(x) = \frac{\log_k(x)}{\log_k(b)}.$$

ENTROPY

- The *entropy* of a random variable X taking values $x \in \mathcal{X}$, and with pmf. p_X, is defined as

$$\begin{aligned} H(X) &= \sum_{x \in \mathcal{X}} p_X(x) I_X(x) \\ &= -\sum_{x \in \mathcal{X}} p_X(x) \log p_X(x), \end{aligned}$$

  and is thus the expected information content in X, measured in bits (unless another base for the logarithm is explicitly stated).

ENTROPY

- The *entropy* of a random variable X taking values $x \in \mathcal{X}$, and with pmf. p_X, is defined as

$$H(X) = \sum_{x \in \mathcal{X}} p_X(x) I_X(x) = -\sum_{x \in \mathcal{X}} p_X(x) \log p_X(x),$$

  and is thus the expected information content in X, measured in bits (unless another base for the logarithm is explicitly stated).
- We use the convention that $x \log(x) = 0$ when $x = 0$.

- The *entropy* of a random variable X taking values $x \in \mathcal{X}$, and with pmf. p_X, is defined as

$$H(X) = \sum_{x \in \mathcal{X}} p_X(x) I_X(x)$$
$$= -\sum_{x \in \mathcal{X}} p_X(x) \log p_X(x),$$

and is thus the expected information content in X, measured in bits (unless another base for the logarithm is explicitly stated).

- We use the convention that $x \log(x) = 0$ when $x = 0$.
- The entropy of a signal (a collection of events) gives a lower bound on how compact the sequence can be encoded (if every event is encoded separately).

- The *entropy* of a random variable X taking values $x \in \mathcal{X}$, and with pmf. p_X, is defined as
$$H(X) = \sum_{x \in \mathcal{X}} p_X(x) I_X(x)$$
$$= -\sum_{x \in \mathcal{X}} p_X(x) \log p_X(x),$$
and is thus the expected information content in X, measured in bits (unless another base for the logarithm is explicitly stated).
- We use the convention that $x \log(x) = 0$ when $x = 0$.
- The entropy of a signal (a collection of events) gives a lower bound on how compact the sequence can be encoded (if every event is encoded separately).
- The entropy of a fair coin toss is 1 bit (since $-2\frac{1}{2} \log_2 \frac{1}{2} = 1$)

- The *entropy* of a random variable X taking values $x \in \mathcal{X}$, and with pmf. p_X, is defined as
$$H(X) = \sum_{x \in \mathcal{X}} p_X(x) I_X(x)$$
$$= -\sum_{x \in \mathcal{X}} p_X(x) \log p_X(x),$$
and is thus the expected information content in X, measured in bits (unless another base for the logarithm is explicitly stated).
- We use the convention that $x \log(x) = 0$ when $x = 0$.
- The entropy of a signal (a collection of events) gives a lower bound on how compact the sequence can be encoded (if every event is encoded separately).
- The entropy of a fair coin toss is 1 bit (since $-2\frac{1}{2}\log_2 \frac{1}{2} = 1$)
- The entropy of a fair dice toss is ≈2.6 bit (since $-6\frac{1}{6}\log_2 \frac{1}{6} \approx 2.6$)

- **Maximal entropy:** every event has equal probability. For event that can take 2^b different values with equal probability $(1/2^b)$ the entropy is equal to the number of bits in the alphabeth, $H = b$.

- **Maximal entropy:** every event has equal probability. For event that can take 2^b different values with equal probability $(1/2^b)$ the entropy is equal to the number of bits in the alphabeth, $H = b$.
- **Minimal entropy:** only one event that occurs with probability 1. In this case the entropy is zero, $H = 0$.

- We can estimate the probability mass function with the normalized histogram.

ESTIMATING THE PROBABILITY MASS FUNCTION

- We can estimate the probability mass function with the normalized histogram.
- For a signal of length n with symbols taking values in the alphabeth $\{s_0, \ldots, s_{m-1}\}$, let n_i be the number of occurances of s_i in the signal, then the normalized histogram value for symbol s_i is
$$p_i = \frac{n_i}{n}.$$

ESTIMATING THE PROBABILITY MASS FUNCTION

- We can estimate the probability mass function with the normalized histogram.
- For a signal of length n with symbols taking values in the alphabeth $\{s_0, \ldots, s_{m-1}\}$, let n_i be the number of occurances of s_i in the signal, then the normalized histogram value for symbol s_i is

$$p_i = \frac{n_i}{n}.$$

- If one assume that the values in the signal are independent realizations of an underlying random variable, then p_i is an estimate on the probability that the variable is s_i.

In this case, we have an $M \times N$ image where each pixel is either 0 or 1. With no inter-pixel spatial redundancy, we must use MN bits to store the image, but the entropy is dependent on the distribution of values.

- *As many 0 as 1 in the image*: The information content is equal for each event, and the entropy is therefore 1.

$$H = -\frac{1}{2}\log_2\frac{1}{2} - \frac{1}{2}\log_2\frac{1}{2} = 1$$

In this case, we have an $M \times N$ image where each pixel is either 0 or 1. With no inter-pixel spatial redundancy, we must use MN bits to store the image, but the entropy is dependent on the distribution of values.

- *As many 0 as 1 in the image*: The information content is equal for each event, and the entropy is therefore 1.

$$H = -\frac{1}{2}\log_2\frac{1}{2} - \frac{1}{2}\log_2\frac{1}{2} = 1$$

- *Three times as many 1 as 0 in the image*: A value of 1 is less surprising than a value of 0. The entropy is then less than in the case above.

$$H = -\frac{1}{4}\log_2\frac{1}{4} - \frac{3}{4}\log_2\frac{3}{4} \approx 0.811$$

ENTROPY IN A BINARY IMAGE

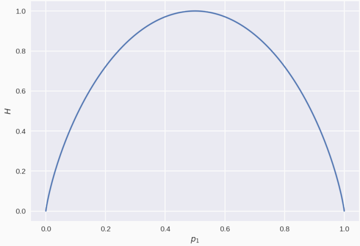


Figure 5: Entropy in a binary image

- When we store one by one pixel value, we need to use 1 bit per pixel, even if the entropy is close to 0.

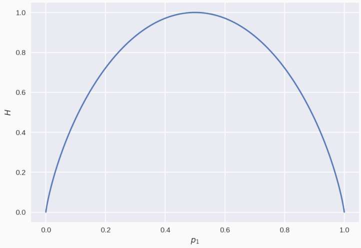


Figure 5: Entropy in a binary image

- When we store one by one pixel value, we need to use 1 bit per pixel, even if the entropy is close to 0.
- The coding redundancy is 0 for the case where $p_0 = p_1 = 0.5$.

We can separate all codes into the following subsets[1]

- Nonsingular codes
- Uniquely decodable codes
- Instantaneous (or prefix) codes

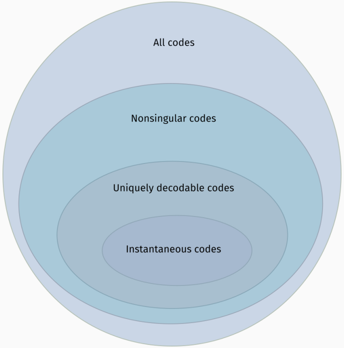


Figure 6: Code classes

[1] See e.g. *Elements of Information Theory* by T. M. Cover and J. A. Thomas for more

NONSINGULAR CODES

- A code is said to be *nonsingular* if every element in $\mathcal{X}$ has a unique codeword $y \in \mathcal{Y}$.

- A code is said to be *nonsingular* if every element in $\mathcal{X}$ has a unique codeword $y \in \mathcal{Y}$.
- That is, the source code c is nonsingular if, for every $x, x' \in \mathcal{X}$

$$x \neq x' \implies c(x) \neq c(x').$$

- We define the extension of a code c as the concatenation of codewords

$$c(x_1 x_2 \cdots x_n) = c(x_1)c(x_2)\cdots c(x_n)$$

- We define the extension of a code c as the concatenation of codewords

$$c(x_1 x_2 \cdots x_n) = c(x_1)c(x_2)\cdots c(x_n)$$

- Example: if $c(x_1) = ab$ and $c(x_2) = cd$, then $c(x_1 x_2) = abcd$.

- We define the extension of a code c as the concatenation of codewords

$$c(x_1 x_2 \cdots x_n) = c(x_1)c(x_2) \cdots c(x_n)$$

- Example: if $c(x_1) = ab$ and $c(x_2) = cd$, then $c(x_1 x_2) = abcd$.
- A code is said to be *uniquely decodable* if its extension is nonsingular.

- We define the extension of a code c as the concatenation of codewords

$$c(x_1 x_2 \cdots x_n) = c(x_1)c(x_2) \cdots c(x_n)$$

- Example: if $c(x_1) = ab$ and $c(x_2) = cd$, then $c(x_1 x_2) = abcd$.
- A code is said to be *uniquely decodable* if its extension is nonsingular.
- With this, every encoded string of symbols has *one and only one* sequence of source symbols.

- We define the extension of a code c as the concatenation of codewords

$$c(x_1 x_2 \cdots x_n) = c(x_1)c(x_2) \cdots c(x_n)$$

- Example: if $c(x_1) = ab$ and $c(x_2) = cd$, then $c(x_1 x_2) = abcd$.
- A code is said to be *uniquely decodable* if its extension is nonsingular.
- With this, every encoded string of symbols has *one and only one* sequence of source symbols.
- We may need to look at the entire encoded string to determine its individual codewords, and decode it.

- A code is said to be *instantaneous* or a *prefix code* if no codeword is a prefix of another codeword.

- A code is said to be *instantaneous* or a *prefix code* if no codeword is a prefix of another codeword.
- For this class of codes, it suffices to read a codeword in order to decode it.

INSTANTANEOUS CODE

- A code is said to be *instantaneous* or a *prefix code* if no codeword is a prefix of another codeword.
- For this class of codes, it suffices to read a codeword in order to decode it.
- That is, we do not need to process the entire string of codewords in order to decode it.

X	Singular	Nonsingular, but not uniquely decodable	Uniquely decodable, but not intantaneous	Instantaneous
a	0	0	10	0
b	0	010	00	10
c	0	01	11	110
d	0	10	110	111

- The code in column three is nonsingular, but we need a seperator between the codewords to be able to decode it. E.g. 00100110 can be decoded to
  - $abcd$ when separated as $0, 010, 01, 10$,
  - $aadcd$ when separated as $0, 0, 10, 01, 10$.

X	Singular	Nonsingular, but not uniquely decodable	Uniquely decodable, but not intantaneous	Instantaneous
a	0	0	10	0
b	0	010	00	10
c	0	01	11	110
d	0	10	110	111

- The code in column three is nonsingular, but we need a seperator between the codewords to be able to decode it. E.g. 00100110 can be decoded to
  - $abcd$ when separated as $0, 010, 01, 10$,
  - $aadcd$ when separated as $0, 0, 10, 01, 10$.
- In column four, the codeword for c is a prefix for the codeword associated with d, and therefore, the code is not instantaneous. We need to look at the whole string of codewords to decode it.

X	Singular	Nonsingular, but not uniquely decodable	Uniquely decodable, but not intantaneous	Instantaneous
a	0	0	10	0
b	0	010	00	10
c	0	01	11	110
d	0	10	110	111

- The code in column three is nonsingular, but we need a seperator between the codewords to be able to decode it. E.g. 00100110 can be decoded to
    - $abcd$ when separated as $0, 010, 01, 10$,
    - $aadcd$ when separated as $0, 0, 10, 01, 10$.
- In column four, the codeword for c is a prefix for the codeword associated with d, and therefore, the code is not instantaneous. We need to look at the whole string of codewords to decode it.
- The code in column five is instantaneous, and we can immediately decode a string of codewords.

OPTIMAL CODE LENGTHS

- For an alphabeth with n different symbols, the codeword lengths $l_1, \ldots, l_m$ from any instantaneous code must satisfy

$$\sum_{i=1}^{m} n^{-l_i} \leq 1.$$

- For an alphabeth with n different symbols, the codeword lengths $l_1, \ldots, l_m$ from any instantaneous code must satisfy

$$\sum_{i=1}^{m} n^{-l_i} \leq 1.$$

- If there exists a set of codelengths that satisfy this inequality, there exist an instantaneous code with these word lengths.

- For an alphabeth with n different symbols, the codeword lengths $l_1, \ldots, l_m$ from any instantaneous code must satisfy

$$\sum_{i=1}^{m} n^{-l_i} \leq 1.$$

- If there exists a set of codelengths that satisfy this inequality, there exist an instantaneous code with these word lengths.
- This result is called the *Kraft inequality theorem*.

- For an alphabeth with n different symbols, the codeword lengths $l_1, \ldots, l_m$ from any instantaneous code must satisfy

$$\sum_{i=1}^{m} n^{-l_i} \leq 1.$$

- If there exists a set of codelengths that satisfy this inequality, there exist an instantaneous code with these word lengths.
- This result is called the *Kraft inequality theorem*.

- The expected length of any instantaneous code over an alphabeth with n symbols for a random variable X is greater or equal to the entropy $H(X)$. That is

$$\sum_{i=1}^{m} l_i p_i \geq H(X)$$

with equality if and only if $n^{-l_i} = p_i$. Here, p_i is the value of the probability mass function at x_i: $p_i = Pr(X = x_i)$.

OPTIMAL CODE LENGTHS

- For an alphabeth with n different symbols, the codeword lengths $l_1, \ldots, l_m$ from any instantaneous code must satisfy

$$\sum_{i=1}^{m} n^{-l_i} \leq 1.$$

- If there exists a set of codelengths that satisfy this inequality, there exist an instantaneous code with these word lengths.
- This result is called the *Kraft inequality theorem*.

- The expected length of any instantaneous code over an alphabeth with n symbols for a random variable X is greater or equal to the entropy $H(X)$. That is

$$\sum_{i=1}^{m} l_i p_i \geq H(X)$$

with equality if and only if $n^{-l_i} = p_i$. Here, p_i is the value of the probability mass function at x_i: $p_i = Pr(X = x_i)$.
- An instantaneous code that achieves equality is thus optimal in terms of expected codeword lengths (something that we want to minimize).

- A simple method that produces an instantaneous code.

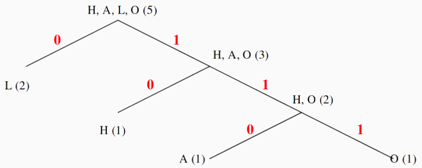
SHANNON-FANO CODING

- A simple method that produces an instantaneous code.
- The resulting is quite compact (but not optimal).

SHANNON-FANO CODING

- A simple method that produces an instantaneous code.
- The resulting is quite compact (but not optimal).
- Algorithm that produces a binary Shannon-Fano code (with alphabeth $\{0, 1\}$):
  1. Sort the symbols x_i of the signal that we want to code by probability of occurance.
  2. Split the symbols into two parts with approximately equal accumulated probability.
     - One group is assigned the symbol 0, and the other the symbol 1.
     - Do this step recursively (that is, do this step on every subgroup), until the group only contain one element.
  3. The result is a binary tree with the symbols that are to be encoded in the leaf nodes.
  4. Traverse the tree from root to the leaf nodes and record the sequence of symbols in order to produce the corresponding codeword.

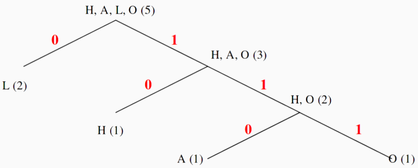
Two different encodings of the sequence "HALLO".



x	$p_X(x)$	$c(x)$	$l(x)$
L	$2/5$	0	1
H	$1/5$	10	2
A	$1/5$	110	3
O	$1/5$	111	3

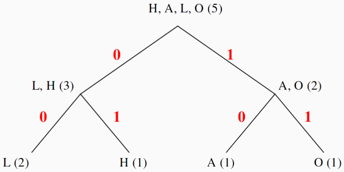
c(HALLO) = 1011000111, with length 10 bits.

Two different encodings of the sequence "HALLO".



x	$p_X(x)$	$c(x)$	$l(x)$
L	$2/5$	0	1
H	$1/5$	10	2
A	$1/5$	110	3
O	$1/5$	111	3

c(HALLO) = 1011000111, with length 10 bits.



x	$p_X(x)$	$c(x)$	$l(x)$
L	$2/5$	00	2
H	$1/5$	01	2
A	$1/5$	10	2
O	$1/5$	11	2

c(HALLO) = 0110000011, with length 10 bits.

- As we saw from the previous examples, the final code is dependent on how we split the groups.

- As we saw from the previous examples, the final code is dependent on how we split the groups.
- In this case, the two solutions are equally good, but this is not allways the case.

- As we saw from the previous examples, the final code is dependent on how we split the groups.
- In this case, the two solutions are equally good, but this is not allways the case.
- In general, the expected codeword length L is bounded as

$$H(X) \leq L \leq H(X) + 1.$$

That is, an upper bound for the coding redundancy of one bit.

- As we saw from the previous examples, the final code is dependent on how we split the groups.
- In this case, the two solutions are equally good, but this is not allways the case.
- In general, the expected codeword length L is bounded as

$$H(X) \leq L \leq H(X) + 1.$$

  That is, an upper bound for the coding redundancy of one bit.
- The expected codeword length is 2 in both example 1 and 2, and the entropy is about 1.92 bits.

HUFFMAN CODING

- An instantaneous coding algorithm.

- An instantaneous coding algorithm.
- *Optimal* in the sense that it achieves minimal coding redundancy.

HUFFMAN CODING

- An instantaneous coding algorithm.
- *Optimal* in the sense that it achieves minimal coding redundancy.
- Algorithm for encoding a sequence of n symbols with a binary Huffman code and alphabeth $\{0, 1\}$:
  1. Sort the symbols by decreasing probability.
  2. Merge the two least likely symbols to a group and give the group a probability equal to the sum of the probabilities of the members in the group. Sort the new sequence by decreasing probability.
  3. Repeat step 2. until there are only two groups left.
  4. Represent the merging as a binary tree, and assign 0 to the left branch and 1 to the right branch.
  5. Every symbol in the original sequence is now at a leaf node. Traverse the tree from the root to the corresponding leaf node, and append the symbols from the traversal to create the codeword.

The six most common letters in the english language, and their relative occurance frequency (normslized within this selection), is given in the table below. The resulting Huffman source code is given as c.

x	$p(x)$	$c(x)$
a	0.160	000
e	0.248	10
i	0.137	010
n	0.131	011
o	0.146	001
t	0.178	11

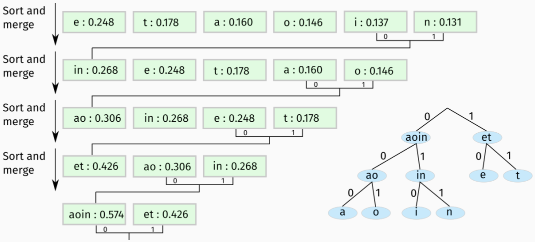


Figure 7: Huffman procedure example with resulting binary tree.

- The expected codeword length in the previous example is

$$
\begin{aligned}
L &= \sum_i l_i p_i \\
&= 3 \cdot 0.160 + 2 \cdot 0.248 + 3 \cdot 0.137 + 3 \cdot 0.131 + 3 \cdot 0.146 + 2 \cdot 0.178 \\
&= 2.574
\end{aligned}
$$

HUFFMAN CODING: EXAMPLE

- The expected codeword length in the previous example is

$$\begin{aligned} L &= \sum_i l_i p_i \\ &= 3 \cdot 0.160 + 2 \cdot 0.248 + 3 \cdot 0.137 + 3 \cdot 0.131 + 3 \cdot 0.146 + 2 \cdot 0.178 \\ &= 2.574 \end{aligned}$$

- And the entropy is

$$\begin{aligned} H &= -\sum_i p_i \log_2 p_i \\ &\approx 2.547 \end{aligned}$$

- Thus, the coding redundancy is $L - H \approx 0.027$.

- Shannon-Fano codes has a 1 bit upper limit for the coding redundancy.

[1] R.G. Gallagger, *Variations on a theme by Huffman*, IEEE Transactions on Information Theory, 24(6), 668-674, 1978.

- Shannon-Fano codes has a 1 bit upper limit for the coding redundancy.
- It can be shown[1] that Huffman codes can achieve an even tighter bound

$$L - H \leq p_{\max} + \log_2\left(\frac{2\log_2 e}{e}\right),$$

where $p_{\max}$ is the probability of the most frequent symbol.

[1] R.G. Gallagger, *Variations on a theme by Huffman*, IEEE Transactions on Information Theory, 24(6), 668-674, 1978.

- Shannon-Fano codes has a 1 bit upper limit for the coding redundancy.
- It can be shown[1] that Huffman codes can achieve an even tighter bound

$$L - H \leq p_{\max} + \log_2 \left(\frac{2 \log_2 e}{e} \right),$$

where $p_{\max}$ is the probability of the most frequent symbol.
- Thus, the coding redundancy increases with increasing $p_{\max}$.

[1] R.G. Gallagger, *Variations on a theme by Huffman*, IEEE Transactions on Information Theory, 24(6), 668-674, 1978.

- Both are instantaneous codes (no codeword is a prefix of another one).

HUFFMAN AND SHANNON-FANO CODING, GENERAL REMARKS

- Both are instantaneous codes (no codeword is a prefix of another one).
- Huffman codes are optimal, Shannon-Fano codes are not. Optimal in the sense that they achieve minimal expected codeword length (and when we encode one symbol at the time).

HUFFMAN AND SHANNON-FANO CODING, GENERAL REMARKS

- Both are instantaneous codes (no codeword is a prefix of another one).
- Huffman codes are optimal, Shannon-Fano codes are not. Optimal in the sense that they achieve minimal expected codeword length (and when we encode one symbol at the time).
- There exist several optimal codes (we can for instance just interchange every symbol in an optimal code to create a new one).

HUFFMAN AND SHANNON-FANO CODING, GENERAL REMARKS

- Both are instantaneous codes (no codeword is a prefix of another one).
- Huffman codes are optimal, Shannon-Fano codes are not. Optimal in the sense that they achieve minimal expected codeword length (and when we encode one symbol at the time).
- There exist several optimal codes (we can for instance just interchange every symbol in an optimal code to create a new one).
- Codewords for frequent symbols are shorter than codewords for rare symbols.

HUFFMAN AND SHANNON-FANO CODING, GENERAL REMARKS

- Both are instantaneous codes (no codeword is a prefix of another one).
- Huffman codes are optimal, Shannon-Fano codes are not. Optimal in the sense that they achieve minimal expected codeword length (and when we encode one symbol at the time).
- There exist several optimal codes (we can for instance just interchange every symbol in an optimal code to create a new one).
- Codewords for frequent symbols are shorter than codewords for rare symbols.
- The two least likely symbols have equal codeword length. And differ only in the last bit.

HUFFMAN AND SHANNON-FANO CODING, GENERAL REMARKS

- Both are instantaneous codes (no codeword is a prefix of another one).
- Huffman codes are optimal, Shannon-Fano codes are not. Optimal in the sense that they achieve minimal expected codeword length (and when we encode one symbol at the time).
- There exist several optimal codes (we can for instance just interchange every symbol in an optimal code to create a new one).
- Codewords for frequent symbols are shorter than codewords for rare symbols.
- The two least likely symbols have equal codeword length. And differ only in the last bit.
- Note that the source code also needs to be transmitted with the code in order to be able to decode it. The source code for a b-bit image contains up to $n = 2^b$ codewords, where the longes codeword can be up to $n - 1$ bits.

- For an optimal code, the expected codeword length L must be equal to the entropy

$$\sum_x p(x)l(x) = -\sum_x p(x)\log_2 p(x)$$

- That is, $l(x) = \log_2(1/p(x))$, which is the information content of the event x.

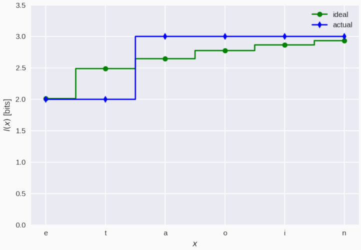


Figure 8: Ideal and actual codeword length from example in fig. 7

- The ideal codeword length is

$$l(x) = \log_2(1/p(x))$$

WHEN DOES HUFFMAN CODING NOT GIVE ANY CODING REDUNDANCY

- The ideal codeword length is

$$l(x) = \log_2(1/p(x))$$

- Since we only deal with integer codeword lengths, this is only possible when

$$p(x) = \frac{1}{2^k}$$

  for some integer k.

WHEN DOES HUFFMAN CODING NOT GIVE ANY CODING REDUNDANCY

- The ideal codeword length is

$$l(x) = \log_2(1/p(x))$$

- Since we only deal with integer codeword lengths, this is only possible when

$$p(x) = \frac{1}{2^k}$$

for some integer k.

- Example

x	x_1	x_2	x_3	x_4	x_5	x_6
$p(x)$	$\frac{1}{2}$	$\frac{1}{4}$	$\frac{1}{8}$	$\frac{1}{16}$	$\frac{1}{32}$	$\frac{1}{64}$
$c(x)$	0	10	110	1110	11110	11111

- In this example $L = H = 1.9375$, that is, no coding redundancy.

- Lossless compression method.

ARITHMETIC CODING

- Lossless compression method.
- Variable code length, codes more probable symbols more compactly.

ARITHMETIC CODING

- Lossless compression method.
- Variable code length, codes more probable symbols more compactly.
- Contrary to Shannon-Fano coding and Huffman coding, which codes symbol by symbol, arithmetic coding encodes the entire signal to one number $d \in [0, 1)$.

ARITHMETIC CODING

- Lossless compression method.
- Variable code length, codes more probable symbols more compactly.
- Contrary to Shannon-Fano coding and Huffman coding, which codes symbol by symbol, arithmetic coding encodes the entire signal to one number $d \in [0, 1)$.
- Same expected codeword length as Huffman code.

ARITHMETIC CODING

- Lossless compression method.
- Variable code length, codes more probable symbols more compactly.
- Contrary to Shannon-Fano coding and Huffman coding, which codes symbol by symbol, arithmetic coding encodes the entire signal to one number $d \in [0, 1)$.
- Same expected codeword length as Huffman code.
- Can achieve shorter codewords for the entire sequence than Huffman code. This is because one is not limited to integer codewords for each symbol.

- The signal is a string of symbols x, where the symbols are taken from some alphabeth $\mathcal{X}$.

- The signal is a string of symbols x, where the symbols are taken from some alphabeth $\mathcal{X}$.
- As usual, we model the symbols as realizations of a discrete random variable X with an associated probability mass function p_X.

- The signal is a string of symbols x, where the symbols are taken from some alphabeth $\mathcal{X}$.
- As usual, we model the symbols as realizations of a discrete random variable X with an associated probability mass function p_X.
- For each symbol in the signal, we use the pmf. to associate a unique interval with the part of the signal we have processed so far.

- The signal is a string of symbols x, where the symbols are taken from some alphabeth $\mathcal{X}$.
- As usual, we model the symbols as realizations of a discrete random variable X with an associated probability mass function p_X.
- For each symbol in the signal, we use the pmf. to associate a unique interval with the part of the signal we have processed so far.
- At the end, when the whole signal is processed, we are left with a decimal interval which is unique to the string of symbols that is our signal.

ARITHMETIC CODING: ENCODING OVERVIEW

- The signal is a string of symbols x, where the symbols are taken from some alphabeth $\mathcal{X}$.
- As usual, we model the symbols as realizations of a discrete random variable X with an associated probability mass function p_X.
- For each symbol in the signal, we use the pmf. to associate a unique interval with the part of the signal we have processed so far.
- At the end, when the whole signal is processed, we are left with a decimal interval which is unique to the string of symbols that is our signal.
- We then find the number within this decimal with the shortest binary representation, and use this as the encoded signal.

1. Initialize *current interval* to $[0.0, 1.0)$

1. Initialize *current interval* to $[0.0, 1.0)$
2. Create a list of interval edges $q = [0, p_X(x_1), p_X(x_1) + p_X(x_2), \ldots, \sum_{i=1}^{k} p_X(x_k), \ldots]$

1. Initialize *current interval* to $[0.0, 1.0)$
2. Create a list of interval edges $q = [0, p_X(x_1), p_X(x_1) + p_X(x_2), \ldots, \sum_{i=1}^{k} p_X(x_k), \ldots]$
3. Initialize the interval $[c_{\min}, c_{\max}) = [0, 1)]$.

1. Initialize *current interval* to $[0.0, 1.0)$
2. Create a list of interval edges $q = [0, p_X(x_1), p_X(x_1) + p_X(x_2), \ldots, \sum_{i=1}^{k} p_X(x_k), \ldots]$
3. Initialize the interval $[c_{\min}, c_{\max}) = [0, 1)]$.
4. For each symbol x_i in the signal (from left to right):
    4.1 Create a new set of interval edges: $q_{new} \leftarrow c_{\min} + (c_{\min} - c_{\max}) \cdot q$
    4.2 Let the new *current interval* $[c_{\min}, c_{\max})$ be the interval in q_{new} that x_i correspond to.

1. Initialize *current interval* to $[0.0, 1.0)$
2. Create a list of interval edges $q = [0, p_X(x_1), p_X(x_1) + p_X(x_2), \ldots, \sum_{i=1}^{k} p_X(x_k), \ldots]$
3. Initialize the interval $[c_{\min}, c_{\max}) = [0, 1)]$.
4. For each symbol x_i in the signal (from left to right):
   4.1 Create a new set of interval edges: $q_{new} \leftarrow c_{\min} + (c_{\min} - c_{\max}) \cdot q$
   4.2 Let the new *current interval* $[c_{\min}, c_{\max})$ be the interval in q_{new} that x_i correspond to.
5. Find a decimal number within the final interval with the shortes binary sequence.

1. Initialize *current interval* to $[0.0, 1.0)$
2. Create a list of interval edges $q = [0, p_X(x_1), p_X(x_1) + p_X(x_2), \ldots, \sum_{i=1}^{k} p_X(x_k), \ldots]$
3. Initialize the interval $[c_{\min}, c_{\max}) = [0, 1)]$.
4. For each symbol x_i in the signal (from left to right):
   4.1 Create a new set of interval edges: $q_{new} \leftarrow c_{\min} + (c_{\min} - c_{\max}) \cdot q$
   4.2 Let the new *current interval* $[c_{\min}, c_{\max})$ be the interval in q_{new} that x_i correspond to.
5. Find a decimal number within the final interval with the shortes binary sequence.
6. The encoded signal is this shortest binary sequence.

- Suppose we have an alphabeth $\{a_1, a_2, a_3, a_4\}$ with associated pmf. $p_X = [0.2, 0.2, 0.4, 0.2]$.

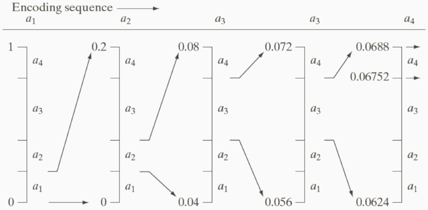
ARITHMETIC ENCODING: EXAMPLE

- Suppose we have an alphabeth $\{a_1, a_2, a_3, a_4\}$ with associated pmf. $p_X = [0.2, 0.2, 0.4, 0.2]$.
- We want to encode the sequence $a_1 a_2 a_3 a_3 a_4$.

- Suppose we have an alphabeth $\{a_1, a_2, a_3, a_4\}$ with associated pmf. $p_X = [0.2, 0.2, 0.4, 0.2]$.
- We want to encode the sequence $a_1 a_2 a_3 a_3 a_4$.

Step-by-step solution, current interval is initialized to $[0, 1)$.

Symbol	Interval	Sequence	Interval
a_1	$[0.0, 0.2)$	a_1	$[0.0, 0.2)$
a_2	$[0.2, 0.4)$	$a_1 a_2$	$[0.04, 0.08)$
a_3	$[0.4, 0.8)$	$a_1 a_2 a_3$	$[0.056, 0.072)$
a_3	$[0.4, 0.8)$	$a_1 a_2 a_3 a_3$	$[0.0624, 0.0688)$
a_4	$[0.8, 1.0)$	$a_1 a_2 a_3 a_3 a_4$	$[0.06752, 0.0688)$



- We do not store/transmit the signal as a decimal number, but as a binary sequence.

DECIMAL NUMBERS AS BINARY SEQUENCE

- We do not store/transmit the signal as a decimal number, but as a binary sequence.
- How to represent some interval with the shortest possible binary sequence, that is, with the least number of bits?

DECIMAL NUMBERS AS BINARY SEQUENCE

- We do not store/transmit the signal as a decimal number, but as a binary sequence.
- How to represent some interval with the shortest possible binary sequence, that is, with the least number of bits?
- First, we need to know how to represent a decimal number in binary.
  - Any decimal number $d \in [0, 1)$ can be written as a power series

    $$d = \sum_{n=1}^{\infty} b_n \left(\frac{1}{2}\right)^n = b_1 \frac{1}{2^1} + b_2 \frac{1}{2^2} + b_3 \frac{1}{2^3} + \cdots$$

    where the weights are either 0 or 1 ($b_n \in \{0, 1\}, \quad n \in \{1, 2, 3, \ldots\}$).
  - The resulting binary sequence $b_1 b_2 b_3 \cdots$ is then the binary representation of d.
  - We use a subscript to indicate what system we are in d_{10} for decimal, and $0.d_2$ for binary.
  - For instance: $0.703125_{10} = 0.101101_2$ since

    $$0.703125 = 1\frac{1}{2^1} + 0\frac{1}{2^2} + 1\frac{1}{2^3} + 1\frac{1}{2^4} + 0\frac{1}{2^5} + 1\frac{1}{2^6}$$

Algorithm 1 Binary representation of decimal number $d \in [0, 1)$

```
procedure BINARY(d)
    r ← d                          ▷ Reminder
    b ← ⌊2r⌋                       ▷ Binary weight
    s ← [b]                        ▷ Binary sequence
    while r > 0 do
        r ← 2r − b
        b ← ⌊2r⌋
        s ← s + [b]                ▷ Append b to s
    end while
    return s
end procedure
```

- Define a reminder $r_k = \sum_{n=k}^{\infty} b_n \left(\frac{1}{2}\right)^n$.
- Notice that $d = r_1$.
- Notice also that $2r_1 = b_1 + r_2$.
- If the integer part of $2r_1$ is 0, b_1 must be 0.
- Also, if the integer part of $2r_1$ is 1, b_1 must be 1.
- This is also the case for the rest of the reminders: $b_k = \lfloor 2r_k \rfloor$.
- And we can find the next reminder $r_{k+1} = 2r_k - b_k$.
- Terminate the search when the reminder is zero.

- We know how to represent a decimal number with a binary sequence.

SHORTEST BINARY REPRESENTATION WITHIN A DECIMAL INTERVAL

- We know how to represent a decimal number with a binary sequence.
- We now need to find the shortest binary sequence within a decimal interval $[d_{\min}, d_{\max})$.

SHORTEST BINARY REPRESENTATION WITHIN A DECIMAL INTERVAL

- We know how to represent a decimal number with a binary sequence.
- We now need to find the shortest binary sequence within a decimal interval $[d_{\min}, d_{\max})$.
- Imagine that we have a current interval $[c_{\min}, c_{\max})$.

SHORTEST BINARY REPRESENTATION WITHIN A DECIMAL INTERVAL

- We know how to represent a decimal number with a binary sequence.
- We now need to find the shortest binary sequence within a decimal interval $[d_{\min}, d_{\max})$.
- Imagine that we have a current interval $[c_{\min}, c_{\max})$.
- We want to fit this interval inside the decimal interval.

SHORTEST BINARY REPRESENTATION WITHIN A DECIMAL INTERVAL

- We know how to represent a decimal number with a binary sequence.
- We now need to find the shortest binary sequence within a decimal interval $[d_{\min}, d_{\max})$.
- Imagine that we have a current interval $[c_{\min}, c_{\max})$.
- We want to fit this interval inside the decimal interval.
- We do this stepwise; at each step $k = 1, 2, \ldots$ we either increase $c_{\min}$ or decrease $c_{\max}$.

SHORTEST BINARY REPRESENTATION WITHIN A DECIMAL INTERVAL

- We know how to represent a decimal number with a binary sequence.
- We now need to find the shortest binary sequence within a decimal interval $[d_{\min}, d_{\max})$.
- Imagine that we have a current interval $[c_{\min}, c_{\max})$.
- We want to fit this interval inside the decimal interval.
- We do this stepwise; at each step $k = 1, 2, \ldots$ we either increase $c_{\min}$ or decrease $c_{\max}$.
- We increase $c_{\min}$ by adding $\frac{1}{2^k}$.

SHORTEST BINARY REPRESENTATION WITHIN A DECIMAL INTERVAL

- We know how to represent a decimal number with a binary sequence.
- We now need to find the shortest binary sequence within a decimal interval $[d_{\text{min}}, d_{\text{max}})$.
- Imagine that we have a current interval $[c_{\text{min}}, c_{\text{max}})$.
- We want to fit this interval inside the decimal interval.
- We do this stepwise; at each step $k = 1, 2, \ldots$ we either increase c_{min} or decrease c_{max}.
- We increase c_{min} by adding $\frac{1}{2^k}$.
- We decrease c_{max} by subtracting $\frac{1}{2^k}$.

SHORTEST BINARY REPRESENTATION WITHIN A DECIMAL INTERVAL

- We know how to represent a decimal number with a binary sequence.
- We now need to find the shortest binary sequence within a decimal interval $[d_{\mathrm{min}}, d_{\mathrm{max}})$.
- Imagine that we have a current interval $[c_{\mathrm{min}}, c_{\mathrm{max}})$.
- We want to fit this interval inside the decimal interval.
- We do this stepwise; at each step $k = 1, 2, \ldots$ we either increase c_{min} or decrease c_{max}.
- We increase c_{min} by adding $\frac{1}{2^k}$.
- We decrease c_{max} by subtracting $\frac{1}{2^k}$.
- If we need to increase c_{min}, we add 1 to the binary sequence.

SHORTEST BINARY REPRESENTATION WITHIN A DECIMAL INTERVAL

- We know how to represent a decimal number with a binary sequence.
- We now need to find the shortest binary sequence within a decimal interval $[d_{\text{min}}, d_{\text{max}})$.
- Imagine that we have a current interval $[c_{\text{min}}, c_{\text{max}})$.
- We want to fit this interval inside the decimal interval.
- We do this stepwise; at each step $k = 1, 2, \ldots$ we either increase c_{min} or decrease c_{max}.
- We increase c_{min} by adding $\frac{1}{2^k}$.
- We decrease c_{max} by subtracting $\frac{1}{2^k}$.
- If we need to increase c_{min}, we add 1 to the binary sequence.
- If we need to decrease c_{max}, we add 0 to the binary sequence.

SHORTEST BINARY REPRESENTATION WITHIN A DECIMAL INTERVAL

- We know how to represent a decimal number with a binary sequence.
- We now need to find the shortest binary sequence within a decimal interval $[d_{\text{min}}, d_{\text{max}})$.
- Imagine that we have a current interval $[c_{\text{min}}, c_{\text{max}})$.
- We want to fit this interval inside the decimal interval.
- We do this stepwise; at each step $k = 1, 2, \ldots$ we either increase c_{min} or decrease c_{max}.
- We increase c_{min} by adding $\frac{1}{2^k}$.
- We decrease c_{max} by subtracting $\frac{1}{2^k}$.
- If we need to increase c_{min}, we add 1 to the binary sequence.
- If we need to decrease c_{max}, we add 0 to the binary sequence.
- Allways try to add a 1, and add a 0 if this is not possible.

SHORTEST BINARY REPRESENTATION WITHIN A DECIMAL INTERVAL

- We know how to represent a decimal number with a binary sequence.
- We now need to find the shortest binary sequence within a decimal interval $[d_{\min}, d_{\max})$.
- Imagine that we have a current interval $[c_{\min}, c_{\max})$.
- We want to fit this interval inside the decimal interval.
- We do this stepwise; at each step $k = 1, 2, \ldots$ we either increase $c_{\min}$ or decrease $c_{\max}$.
- We increase $c_{\min}$ by adding $\frac{1}{2^k}$.
- We decrease $c_{\max}$ by subtracting $\frac{1}{2^k}$.
- If we need to increase $c_{\min}$, we add 1 to the binary sequence.
- If we need to decrease $c_{\max}$, we add 0 to the binary sequence.
- Allways try to add a 1, and add a 0 if this is not possible.
- If neither $c_{\min}$ or $c_{\max}$ is changed at the current step, terminate the search.

Algorithm 2 Find shortest binary representation in decimal interval

```
procedure ShortestBinSequence((d_min, d_max))
    (c_min, c_max) ← (0.0, 1.0)                         ▷ Initialize current interval
    k ← 1                                               ▷ Step counter
    s ← []                                              ▷ Binary sequence
    while True do
        if (c_min < d_min) ∧ (c_min + 1/2^k < d_max) then
            c_min ← c_min + 1/2^k
            s ← s + [1]                                 ▷ Append 1 to s
        else
            if (c_max > d_max) ∧ (c_max − 1/2^k > d_min) then
                c_min ← c_min + 1/2^k
                s ← s + [0]                             ▷ Append 0 to s
            else
                Exit loop
            end if
        end if
    end while
    return s
end procedure
```

- Given an encoded signal $b_1 b_2 b_3 \cdots b_k$, we first find the decimal representation

$$d = \sum_{n=1}^{k} b_n \left(\frac{1}{2} \right)^n$$

- Given an encoded signal $b_1 b_2 b_3 \cdots b_k$, we first find the decimal representation

$$d = \sum_{n=1}^{k} b_n \left(\frac{1}{2}\right)^n$$

- Similar to what we did in the encoding, we define a list of interval edges based on the pmf $q = [0, p_X(x_1), p_X(x_1) + p_X(x_2), \ldots, \sum_{i=1}^{k} p_X(x_k), \ldots]$

- Given an encoded signal $b_1 b_2 b_3 \cdots b_k$, we first find the decimal representation

$$d = \sum_{n=1}^{k} b_n \left(\frac{1}{2}\right)^n$$

- Similar to what we did in the encoding, we define a list of interval edges based on the pmf $q = [0, p_X(x_1), p_X(x_1) + p_X(x_2), \ldots, \sum_{i=1}^{k} p_X(x_k), \ldots]$
  1. See what interval the decimal number lies in, set this as the current interval.
  2. Decode the symbol corresponding to this interval (this is found via the alphabeth and q).
  3. Scale the q to lie within the current interval.
  4. Do step 1 to 3 until termination.
- Termination:
  - Define a **eod** symbol (*end of data*), and stop when this is decoded. Note that this will also need an associated probability in the model.
  - Or, only decode a predefined number of symbols.

- Alphabeth: $\{a, b, c\}$. $p_X = [0.6, 0.2, 0.2]$. $q = [0.0, 0.6, 0.8, 1.0]$

DECODING: EXAMPLE

- Alphabeth: $\{a, b, c\}$. $p_X = [0.6, 0.2, 0.2]$. $q = [0.0, 0.6, 0.8, 1.0]$
- Signal to decode: 10001

- Alphabeth: $\{a, b, c\}$. $p_X = [0.6, 0.2, 0.2]$. $q = [0.0, 0.6, 0.8, 1.0]$
- Signal to decode: 10001
- First, we find that $0.10001_2 = 0.53125_{10}$

- Alphabeth: $\{a, b, c\}$. $p_X = [0.6, 0.2, 0.2]$. $q = [0.0, 0.6, 0.8, 1.0]$
- Signal to decode: 10001
- First, we find that $0.10001_2 = 0.53125_{10}$
- Then we continue decoding symbol for symbol until termination:

$[c_{\min}, c_{\max})$	q_{new}	Symbol	Sequence
$[0.0, 1.0)$	$[0.0, 0.6, 0.8, 1.0)$	a	a
$[0.0, 0.6)$	$[0.0, 0.36, 0.48, 0.6)$	c	ac
$[0.48, 0.6)$	$[0.48, 0.552, 0.576, 0.6)$	a	aca
$[0.48, 0.552)$	$[0.48, 0.5232, 0.5376, 0.552)$	b	$acab$
$[0.5232, 0.5376)$	$[0.5232, 0.53184, 0.53472, 0.5376)$	a	$acaba$

- The size of decimal intervals can be very small, and require high floating point precision:
  - English alphabeth and letter frequency from wikipedia[1].
  - Encoding the signal: `helloworld`
  - Final interval in encoding: $[0.35040662146355034, 0.35040662146372126)$.
  - Encoded to the binary sequence: `0101100110110100001111111001001101111010000`

[1] https://en.wikipedia.org/wiki/Letter_frequency

- The size of decimal intervals can be very small, and require high floating point precision:
  - English alphabeth and letter frequency from wikipedia[1].
  - Encoding the signal: `helloworld`
  - Final interval in encoding: $[0.35040662146355034, 0.35040662146372126)$.
  - Encoded to the binary sequence: `0101100110110100001111111001001101111010000`
- One solution can be to store/transmit the most significan bit as soon as it is known, and then double the size of the current interval.

[1] `https://en.wikipedia.org/wiki/Letter_frequency`

- The size of decimal intervals can be very small, and require high floating point precision:
    - English alphabeth and letter frequency from wikipedia[1].
    - Encoding the signal: `helloworld`
    - Final interval in encoding: $[0.35040662146355034, 0.35040662146372126)$.
    - Encoded to the binary sequence: `0101100110110100001111111001001101111101 0000`
- One solution can be to store/transmit the most significan bit as soon as it is known, and then double the size of the current interval.
- Many solutions exist, but they are often computationally expensive and behind patents.

[1] `https://en.wikipedia.org/wiki/Letter_frequency`

ARITHMETIC CODING: DIFFERENT MODELS

- Static histogram-based models are not optimal.

ARITHMETIC CODING: DIFFERENT MODELS

- Static histogram-based models are not optimal.
- There exists adaptive models where the probability mass function is adapted to the stream of symbols.

ARITHMETIC CODING: DIFFERENT MODELS

- Static histogram-based models are not optimal.
- There exists adaptive models where the probability mass function is adapted to the stream of symbols.
- We know *a priori* that certain symbols are more likely given what has been processed.

- Static histogram-based models are not optimal.
- There exists adaptive models where the probability mass function is adapted to the stream of symbols.
- We know *a priori* that certain symbols are more likely given what has been processed.
- As an example is the letter `u` more likely if we just have processed a `q` (or `Q`) in the english language.

ARITHMETIC CODING: DIFFERENT MODELS

- Static histogram-based models are not optimal.
- There exists adaptive models where the probability mass function is adapted to the stream of symbols.
- We know *a priori* that certain symbols are more likely given what has been processed.
- As an example is the letter `u` more likely if we just have processed a `q` (or `Q`) in the english language.
- No matter what, the transmitter and receiver needs to have the same model.

SUMMARY

- We compress data to reduce number of bits needed to represent the data.

SUMMARY

- We compress data to reduce number of bits needed to represent the data.
- We do this by removing or reducing redundancy
    - Psychovisual-, inter-pixel temporal-, inter-pixel spatial-, coding-redundancy.
    - The compression is lossy when we remove irrelevant information from the data.

SUMMARY

- We compress data to reduce number of bits needed to represent the data.
- We do this by removing or reducing redundancy
  - Psychovisual-, inter-pixel temporal-, inter-pixel spatial-, coding-redundancy.
  - The compression is lossy when we remove irrelevant information from the data.
- Compression consist of three parts
  - Transform
  - Quantization. Leads to lossy compression.
  - Coding, examples: Huffman, Shannon-Fano, Arithmetic.

Questions?