5 Compression

27 October 2023

Sebastian Wild

COMP526 (Fall 2023)
University of Liverpool
version `2023-10-27 09:22`

Learning Outcomes

1. Understand the necessity for encodings and know *ASCII* and *UTF-8 character encodings*.
2. Understand (qualitatively) the *limits of compressibility*.
3. Know and understand the algorithms (encoding and decoding) for *Huffman codes*, *RLE*, *Elias codes*, *LZW*, *MTF*, and *BWT*, including their *properties* like running time complexity.
4. Select and *adapt* (slightly) a *compression* pipeline for specific type of data.

Unit 5: ***Compression***

Outline

5 Compression

5.1 Context

Overview

- Unit 4 & 8: How to *work* with strings
  - finding substrings
  - finding approximate matches $\rightsquigarrow$ Unit 8
  - finding repeated parts $\rightsquigarrow$ Unit 8
  - ...
  - assumed character array (random access)!
- Unit 5 & 6: How to *store/transmit* strings
  - computer memory: must be binary
  - how to compress strings (save space)
  - how to robustly transmit over noisy channels $\rightsquigarrow$ Unit 6

Clicker Question

What compression methods do you know?

→ sli.do/comp526

Terminology

- **source text:** string $S \in \Sigma_S^\star$ to be stored / transmitted
  Σ_S is some alphabet
- **coded text:** encoded data $C \in \Sigma_C^\star$ that is actually stored / transmitted
  usually use $\Sigma_C = \{0, 1\}$
- **encoding:** algorithm mapping source texts to coded texts $S \to C$
- **decoding:** algorithm mapping coded texts back to original source text $C \to S$

Terminology

- **source text:** string $S \in \Sigma_S^\star$ to be stored / transmitted
  Σ_S is some alphabet
- **coded text:** encoded data $C \in \Sigma_C^\star$ that is actually stored / transmitted
  usually use $\Sigma_C = \{0, 1\}$
- **encoding:** algorithm mapping source texts to coded texts
- **decoding:** algorithm mapping coded texts back to original source text
- **Lossy vs. Lossless**
  - **lossy compression** can only decode **approximately**; the exact source text S is lost
  - **lossless compression** always decodes S exactly
- For media files, lossy, logical compression is useful (e. g. JPEG, MPEG)
- We will concentrate on *lossless* compression algorithms.
  These techniques can be used for any application.

$S \to C \to S'$ S, S' 'similar'

What is a good encoding scheme?

- Depending on the application, goals can be
  - efficiency of encoding/decoding
  - resilience to errors/noise in transmission
  - security (encryption)
  - integrity (detect modifications made by third parties)
  - size

What is a good encoding scheme?

- Depending on the application, goals can be
  - efficiency of encoding/decoding
  - resilience to errors/noise in transmission
  - security (encryption)
  - integrity (detect modifications made by third parties)
  - size

size of a string?

$S \in \Sigma^n \rightsquigarrow n?$

$\Sigma_C = \Sigma^n \quad C = S$

- Focus in this unit: **size** of coded text

  Encoding schemes that (try to) minimize the size of coded texts perform *data compression*.

- We will measure the *compression ratio*: $\dfrac{|C| \cdot \lg |\Sigma_C|}{|S| \cdot \lg |\Sigma_S|} \overset{\Sigma_C = \{0,1\}}{=} \dfrac{|C|}{|S| \cdot \lg |\Sigma_S|}$

  < 1 means successful compression
  $= 1$ means no compression
  > 1 means "compression" made it bigger!? (yes, that happens ...)

Clicker Question

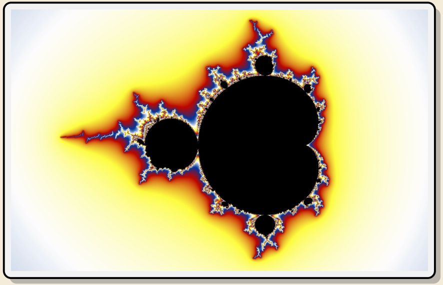
Do you know what uncomputable problems (halting problem, Post's correspondence problem, . . .) are?

- **A** Sure, I could explain what it is.
- **B** Heard that in a lecture, but don't quite remember
- **C** No, never heard of it

→ `sli.do/comp526`

Limits of algorithmic compression

Is this image compressible?

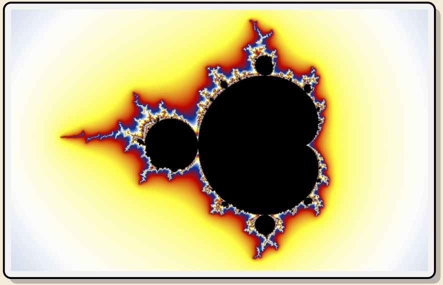


Limits of algorithmic compression

Is this image compressible?

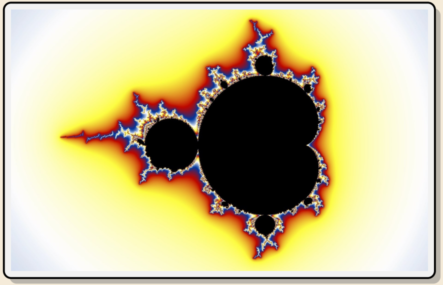
visualization of Mandelbrot set

- Clearly a complex shape!
- Will not compress (too) well using, say, PNG.
- but:
  - completely defined by mathematical formula
  - $\rightsquigarrow$ **can be generated by a very small program!**



Limits of algorithmic compression

Is this image compressible?



visualization of Mandelbrot set

- Clearly a complex shape!
- Will not compress (too) well using, say, PNG.
- but:
  - completely defined by mathematical formula
  - $\rightsquigarrow$ **can be generated by a very small program!**

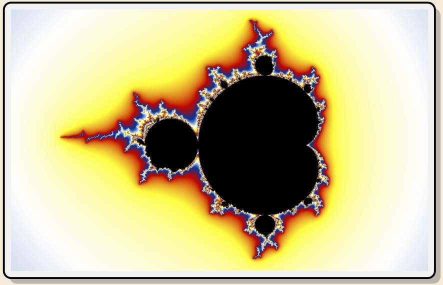
$\rightsquigarrow$ ***Kolmogorov complexity***

- C = *any program* that outputs S

  self-extracting archives!
- Kolmogorov complexity = length of smallest such program

Limits of algorithmic compression

Is this image compressible?



visualization of Mandelbrot set

- Clearly a complex shape!
- Will not compress (too) well using, say, PNG.
- but:
  - completely defined by mathematical formula
  - ⇝ **can be generated by a very small program!**

⇝ ***Kolmogorov complexity***

- C = *any program* that outputs S

  self-extracting archives!
- Kolmogorov complexity = length of smallest such program
- **Problem:** finding smallest such program is *uncomputable*.

⇝ No optimal encoding algorithm is possible!

⇝ must be inventive to get efficient methods

What makes data compressible?

- Lossless compression methods mainly exploit two types of redundancies in source texts:

  1. **uneven character frequencies**
     some characters occur more often than others $\longrightarrow$ Part I
  2. **repetitive texts**
     different parts in the text are (almost) identical $\longrightarrow$ Part II

What makes data compressible?

- Lossless compression methods mainly exploit two types of redundancies in source texts:

  1. **uneven character frequencies**
     some characters occur more often than others $\longrightarrow$ Part I
  2. **repetitive texts**
     different parts in the text are (almost) identical $\longrightarrow$ Part II

There is no such thing as a free lunch!

Not *everything* is compressible ($\longrightarrow$ tutorials)
$\rightsquigarrow$ focus on versatile methods that often work

Part I

Exploiting character frequencies

5.2 Character Encodings

Character encodings

- Simplest form of encoding: Encode each source character individually

⇝ encoding function $E : \Sigma_S \to \Sigma_C^\star$

  - typically, $|\Sigma_S| \gg |\Sigma_C|$, so need several bits per character
  - for $c \in \Sigma_S$, we call $E(c)$ the *codeword* of c

- **fixed-length code:** $|E(c)|$ is the same for all $c \in \Sigma_C$
- **variable-length code:** not all codewords of same length

Fixed-length codes

- fixed-length codes are the simplest type of character encodings
- Example: **ASCII** (American Standard Code for Information Interchange, 1963)

0000000 NUL	0010000 DLE	0100000	0110000 0	1000000 @	1010000 P	1100000 ‘	1110000 p
0000001 SOH	0010001 DC1	0100001 !	0110001 1	1000001 A	1010001 Q	1100001 a	1110001 q
0000010 STX	0010010 DC2	0100010 "	0110010 2	1000010 B	1010010 R	1100010 b	1110010 r
0000011 ETX	0010011 DC3	0100011 #	0110011 3	1000011 C	1010011 S	1100011 c	1110011 s
0000100 EOT	0010100 DC4	0100100 $	0110100 4	1000100 D	1010100 T	1100100 d	1110100 t
0000101 ENQ	0010101 NAK	0100101 %	0110101 5	1000101 E	1010101 U	1100101 e	1110101 u
0000110 ACK	0010110 SYN	0100110 &	0110110 6	1000110 F	1010110 V	1100110 f	1110110 v
0000111 BEL	0010111 ETB	0100111 ’	0110111 7	1000111 G	1010111 W	1100111 g	1110111 w
0001000 BS	0011000 CAN	0101000 (	0111000 8	1001000 H	1011000 X	1101000 h	1111000 x
0001001 HT	0011001 EM	0101001)	0111001 9	1001001 I	1011001 Y	1101001 i	1111001 y
0001010 LF	0011010 SUB	0101010 *	0111010 :	1001010 J	1011010 Z	1101010 j	1111010 z
0001011 VT	0011011 ESC	0101011 +	0111011 ;	1001011 K	1011011 [	1101011 k	1111011 {
0001100 FF	0011100 FS	0101100 ,	0111100 <	1001100 L	1011100 \	1101100 l	1111100 |
0001101 CR	0011101 GS	0101101 -	0111101 =	1001101 M	1011101]	1101101 m	1111101 }
0001110 SO	0011110 RS	0101110 .	0111110 >	1001110 N	1011110 ^	1101110 n	1111110 ~
0001111 SI	0011111 US	0101111 /	0111111 ?	1001111 O	1011111 _	1101111 o	1111111 DEL

- 7 bit per character
- just enough for English letters and a few symbols (plus control characters)

Fixed-length codes – Discussion

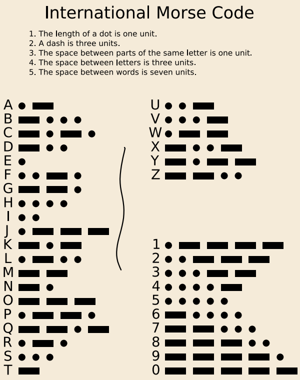
- 👍 Encoding & Decoding as fast as it gets
- 👎 Unless all characters equally likely, it wastes a lot of space
- 👎 inflexible (how to support adding a new character?)

Variable-length codes

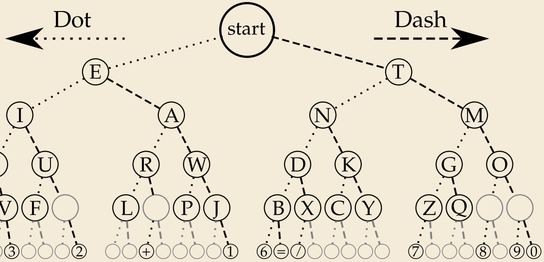
- to gain more flexibility, have to allow different lengths for codewords
- actually an old idea: **Morse Code**

$$\Sigma_S = \{A, \dots, Z, 0, \dots, 9\}$$

$$\Sigma_C = \{\bullet, \text{—}, \text{pause}\}$$



https://commons.wikimedia.org/wiki/File:International_Morse_Code.svg



https://commons.wikimedia.org/wiki/File:Morse-code-tree.svg

Clicker Question

How many characters are there in the alphabet of the coded text in Morse Code, i. e., what is $|\Sigma_C|$?

A 1

B 2

C 3

D 4

E 26

F 36

G 256

→ sli.do/comp526

Clicker Question

How many characters are there in the alphabet of the coded text in Morse Code, i. e., what is $|\Sigma_C|$?

A ~~1~~

B ~~2~~

C 3 ✓

D ~~4~~

E ~~26~~

F ~~36~~

G ~~256~~

→ sli.do/comp526

Variable-length codes – UTF-8

- Modern example: UTF-8 encoding of Unicode:

  default encoding for text-files, XML, HTML since 2009

  - Encodes any Unicode character (~~137 994 as~~ 15 000 of May 2019, and counting)
  - uses 1–4 bytes (codeword lengths: 8, 16, 24, or 32 bits)
  - Every ASCII character is encoded in 1 byte with leading bit `0`, followed by the 7 bits for ASCII
  - Non-ASCII charactters start with 1–4 `1`s indicating the total number of bytes, followed by a `0` and 3–5 bits.
    The remaining bytes each start with `10` followed by 6 bits.

Char. number range (hexadecimal)	UTF-8 octet sequence (binary)
`0000 0000 – 0000 007F`	`0xxxxxxx`
`0000 0080 – 0000 07FF`	`110xxxxx 10xxxxxx`
`0000 0800 – 0000 FFFF`	`1110xxxx 10xxxxxx 10xxxxxx`
`0001 0000 – 0010 FFFF`	`11110xxx 10xxxxxx 10xxxxxx 10xxxxxx`

For English text, most characters use only 8 bit, but we can include any Unicode character, as well.

Pitfall in variable-length codes

- Suppose we have the following code:

c	a	n	b	s
$E(c)$	0	10	110	100

- Happily encode text S = `banana` with the coded text $C = \underset{\texttt{b}}{\underline{\texttt{110}}}\underset{\texttt{a}}{\texttt{0}}\underset{\texttt{n}}{\underline{\texttt{10}}}\underset{\texttt{a}}{\texttt{0}}\underset{\texttt{n}}{\underline{\texttt{10}}}\underset{\texttt{a}}{\texttt{0}}$

Pitfall in variable-length codes

- Suppose we have the following code:

c	a	n	b	s
$E(c)$	0	10	110	100

- Happily encode text S = banana with the coded text $C = \underbrace{\mathtt{110}}_{\mathtt{b}}\underbrace{\mathtt{0}}_{\mathtt{a}}\underbrace{\mathtt{10}}_{\mathtt{n}}\underbrace{\mathtt{0}}_{\mathtt{a}}\underbrace{\mathtt{10}}_{\mathtt{n}}\underbrace{\mathtt{0}}_{\mathtt{a}}$

ϟ C = 1100100100 decodes **both** to banana and to bass: $\underbrace{\mathtt{110}}_{\mathtt{b}}\underbrace{\mathtt{0}}_{\mathtt{a}}\underbrace{\mathtt{100}}_{\mathtt{s}}\underbrace{\mathtt{100}}_{\mathtt{s}}$

$\rightsquigarrow$ not a valid code . . . (cannot tolerate ambiguity)

but how should we have known?

Pitfall in variable-length codes

▶ Suppose we have the following code:

c	a	n	b	s
$E(c)$	0	10	110	100

▶ Happily encode text S = banana with the coded text $C = \underbrace{110}_{\text{b}}\underbrace{0}_{\text{a}}\underbrace{10}_{\text{n}}\underbrace{0}_{\text{a}}\underbrace{10}_{\text{n}}\underbrace{0}_{\text{a}}$

ϟ C = 1100100100 decodes **both** to banana and to bass: $\underbrace{110}_{\text{b}}\underbrace{0}_{\text{a}}\underbrace{100}_{\text{s}}\underbrace{100}_{\text{s}}$

⇝ not a valid code . . . (cannot tolerate ambiguity)

but how should we have known?

$E(\texttt{n})$ = 10 is a (proper) **prefix** of $E(\texttt{s})$ = 100

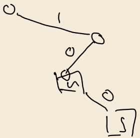
⇝ Leaves decoder wondering whether to stop after reading 10 or continue!

⇝ Require a ***prefix-free*** code: No codeword is a prefix of another.

prefix-free $\Longrightarrow$ instantaneously decodable $\Longrightarrow$ uniquely decodable

Code tries

from before
$n = 10$
$s = 100$



- From now on only consider prefix-free codes E: $E(c)$ is not a prefix of $E(c')$ for any $c, c' \in \Sigma_S$.

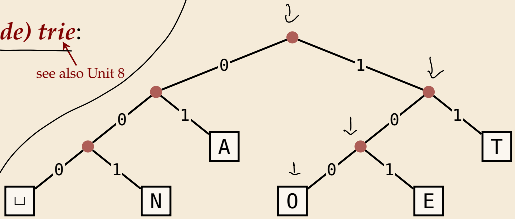
- **Example:**

c	A	E	N	O	T	␣
$E(c)$	01	101	001	100	11	000

Any prefix-free code corresponds to a ***(code) trie***:

see also Unit 8

- binary tree
- one **leaf** for each characters of Σ_S
- path from root to leave = codeword
  left child = 0; right child = 1



- Example for using the code trie:
  - Encode AN␣ANT   01001 000 01 00111
  - Decode 111000001010111   TO ␣ EAT

Code tries

- From now on only consider prefix-free codes E:
  $E(c)$ is not a prefix of $E(c')$ for any $c, c' \in \Sigma_S$.

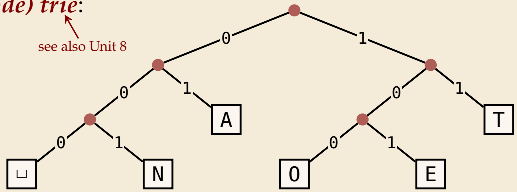
- **Example:**

c	A	E	N	O	T	␣
$E(c)$	01	101	001	100	11	000

  Any prefix-free code corresponds to a ***(code) trie***:

  see also Unit 8

  - binary tree
  - one **leaf** for each characters of Σ_S
  - path from root to leave = codeword
    left child = 0; right child = 1



- Example for using the code trie:
  - Encode AN␣ANT $\longrightarrow$ 010010000100111
  - Decode 111000001010111 $\longrightarrow$ TO␣EAT

Who decodes the decoder?

- Depending on the application, we have to **store/transmit** the **used code**!
- We distinguish:
  - **fixed coding:** code agreed upon in advance, not transmitted (e. g., Morse, UTF-8)
  - **static coding:** code depends on message, but stays same for entire message; it must be transmitted (e. g., Huffman codes $\rightarrow$ next)
  - **adaptive coding:** code depends on message and changes during encoding; implicitly stored withing the message (e. g., LZW $\rightarrow$ below)