# 5 Compression

27 October 2023

Sebastian Wild

COMP526 (Fall 2023)
University of Liverpool
*version* `2023-11-02 14:41`

# 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
  $\Sigma_S$ 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
  $\Sigma_S$ 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

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

- 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.

# 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*: $\dfrac{|C| \cdot \lg|\Sigma_C|}{|S| \cdot \lg|\Sigma_S|} \stackrel{\Sigma_C=\{0,1\}}{=} \dfrac{|C|}{|S| \cdot \lg|\Sigma_S|}$

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

# Clicker Question

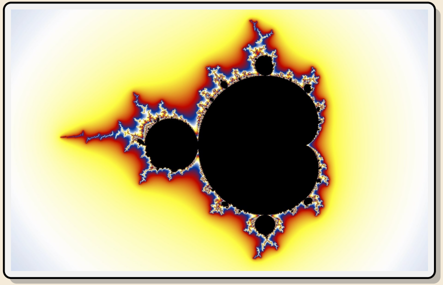
Do you know what uncomputable problems (halting problem, Post's correspondence problem, . . . ) are?

- **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?*

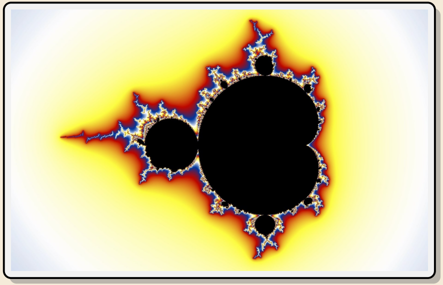


# Limits of algorithmic compression

*Is this image compressible?*

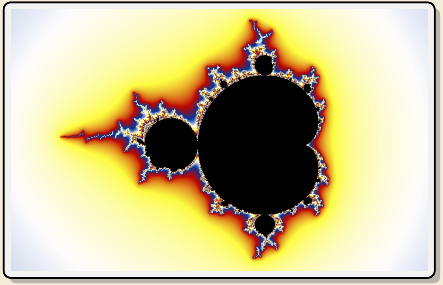
visualization of Mandelbrot set

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

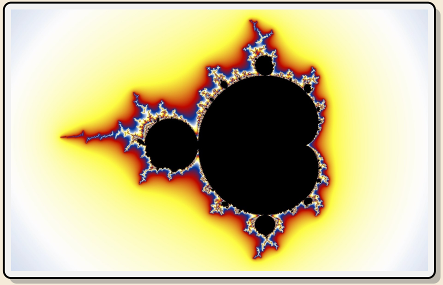
⇝ ***Kolmogorov complexity***

- $C$ = *any program* that outputs $S$

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

# 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 → Part I

  2. **repetitive texts**
     different parts in the text are (almost) identical → Part II

*There is no such thing as a free lunch!*

Not *everything* is compressible (→ tutorials)

⇝ 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

$\rightsquigarrow$ 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

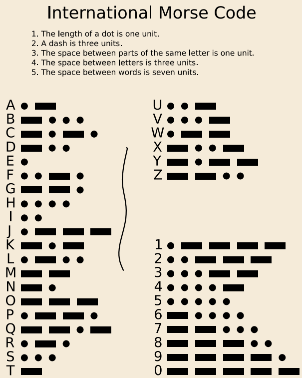
- 👍 Encoding & Decoding as fast as it gets
- 👎 Unless all characters equally likely, it wastes a lot of space
- 👎 inflexible (how to support adding a new character?)

# Variable-length codes

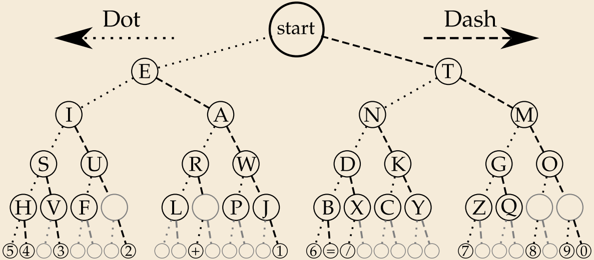
- to gain more flexibility, have to allow different lengths for codewords
- actually an old idea: **Morse Code**

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

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




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 $C = \underbrace{110}_{\texttt{b}}\underbrace{0}_{\texttt{a}}\underbrace{10}_{\texttt{n}}\underbrace{0}_{\texttt{a}}\underbrace{10}_{\texttt{n}}\underbrace{0}_{\texttt{a}}$

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

⇝ 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 $C = \underbrace{\texttt{110}}_{\texttt{b}}\underbrace{\texttt{0}}_{\texttt{a}}\underbrace{\texttt{10}}_{\texttt{n}}\underbrace{\texttt{0}}_{\texttt{a}}\underbrace{\texttt{10}}_{\texttt{n}}\underbrace{\texttt{0}}_{\texttt{a}}$

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

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

but how should we have known?

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

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

⇝ Require a ***prefix-free*** code: $\boxed{\text{No codeword is a prefix of another.}}$

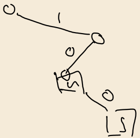
prefix-free $\Longrightarrow$ instantaneously decodable $\Longrightarrow$ uniquely decodable

# 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 $c, c' \in \Sigma_S$.

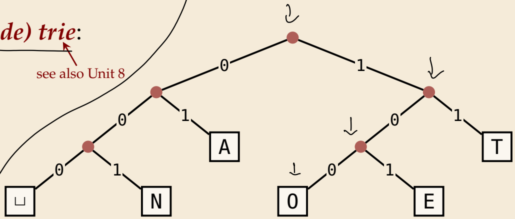
- **Example:**

| $c$ | 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 $\Sigma_S$
  - path from root to leave = codeword
    left child = 0; right child = 1




- Example for using the code trie:
  - Encode AN␣ANT  0100100001 00111
  - Decode 111000001010111  TO␣EAT

# Code tries

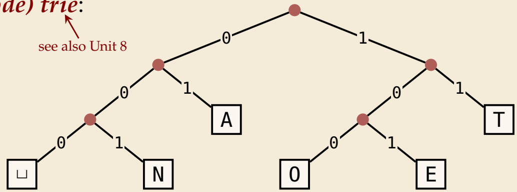
- From now on only consider prefix-free codes $E$:
  $E(c)$ is not a prefix of $E(c')$ for any $c, c' \in \Sigma_S$.

- **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 $\Sigma_S$
  - 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)

## 5.3 Huffman Codes

# Character frequencies

- **Goal:** Find character encoding that produces short coded text
- Convention here: fix $\underline{\Sigma_C = \{0,1\}}$ (binary codes), abbreviate $\Sigma = \Sigma_S$,

- **Observation:** Some letters occur more often than others.

**Typical English prose:**

| | | | | | |
|---|---|---|---|---|---|
| **e** | 12.70% | **d** | 4.25% | **p** | 1.93% |
| **t** | 9.06% | **l** | 4.03% | **b** | 1.49% |
| **a** | 8.17% | **c** | 2.78% | **v** | 0.98% |
| **o** | 7.51% | **u** | 2.76% | **k** | 0.77% |
| **i** | 6.97% | **m** | 2.41% | **j** | 0.15% |
| **n** | 6.75% | **w** | 2.36% | **x** | 0.15% |
| **s** | 6.33% | **f** | 2.23% | **q** | 0.10% |
| **h** | 6.09% | **g** | 2.02% | **z** | 0.07% |
| **r** | 5.99% | **y** | 1.97% | | |

$\rightsquigarrow$ Want shorter codes for more frequent characters!

# Huffman coding

e. g. frequencies / probabilities

- **Given:** $\Sigma$ and weights $w : \Sigma \to \mathbb{R}_{\geq 0}$
- **Goal:** prefix-free code $E$ (= code trie) for $\Sigma$ that minimizes coded text length

  i. e., a code trie minimizing $\displaystyle\sum_{c\in\Sigma} \underbrace{w(c)}_{\text{weight of } c} \cdot \underbrace{|E(c)|}_{\text{length of codeword for } c}$

# Huffman coding

- **Given:** $\Sigma$ and weights $w : \Sigma \to \mathbb{R}_{\geq 0}$ — e. g. frequencies / probabilities
- **Goal:** prefix-free code $E$ (= code trie) for $\Sigma$ that minimizes coded text length

  i. e., a code trie minimizing $\displaystyle\sum_{c \in \Sigma} w(c) \cdot |E(c)|$

- Let's abbreviate $|S|_c$ = #occurrences of $c$ in $S$
- If we use $w(c) = |S|_c$,
  this is the character encoding with smallest possible $|C|$

$\rightsquigarrow$ **best possible** ***character-wise*** **encoding**

prefix-free

- Quite ambitious! *Is this efficiently possible?*

# Huffman's algorithm

- Actually, yes! A greedy/myopic approach succeeds here.

$|\Sigma| = 2 \quad E(a_1) = 0 \quad E(a_2) = 1$

**Huffman's algorithm:**

1. Find two characters `a`, `b` with lowest weights.
   - We will encode them with the same prefix, plus one distinguishing bit, i. e., $E(\mathtt{a}) = u\mathtt{0}$ and $E(\mathtt{b}) = u\mathtt{1}$ for a bitstring $u \in \{0,1\}^\star$ ($u$ to be determined)
2. (Conceptually) replace `a` and `b` by a single character "$\boxed{\mathtt{ab}}$" with $w(\boxed{\mathtt{ab}}) = w(\mathtt{a}) + w(\mathtt{b})$.
3. Recursively apply Huffman's algorithm on the smaller alphabet. This in particular determines $u = E(\boxed{\mathtt{ab}})$.

# Huffman's algorithm

- Actually, yes! A greedy/myopic approach succeeds here.

**Huffman's algorithm:**

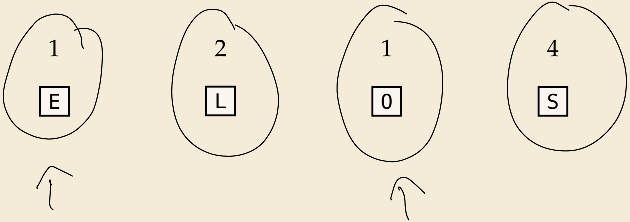
1. Find two characters `a`, `b` with lowest weights.
   - We will encode them with the same prefix, plus one distinguishing bit, i. e., $E(\texttt{a}) = u\texttt{0}$ and $E(\texttt{b}) = u\texttt{1}$ for a bitstring $u \in \{0,1\}^\star$ ($u$ to be determined)
2. (Conceptually) replace `a` and `b` by a single character "$\boxed{\texttt{ab}}$" with $w(\boxed{\texttt{ab}}) = w(\texttt{a}) + w(\texttt{b})$.
3. Recursively apply Huffman's algorithm on the smaller alphabet. This in particular determines $u = E(\boxed{\texttt{ab}})$.

- efficient implementation using a (min-oriented) ***priority queue***
  - start by inserting all characters with their weight as key
  - step 1 uses two `deleteMin` calls
  - step 2 inserts a new character with the sum of old weights as key

# Huffman's algorithm – Example

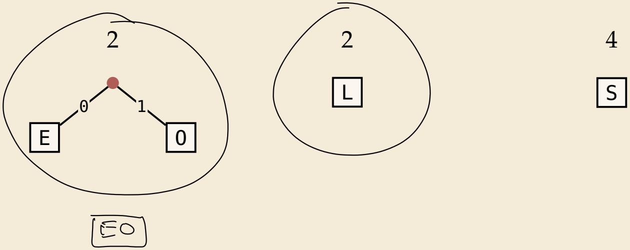
- Example text: $S = \texttt{LOSSLESS}$ $\rightsquigarrow$ $\Sigma_S = \{\texttt{E}, \texttt{L}, \texttt{O}, \texttt{S}\}$

- Character frequencies: $\texttt{E} : 1$, $\texttt{L} : 2$, $\texttt{O} : 1$, $\texttt{S} : 4$



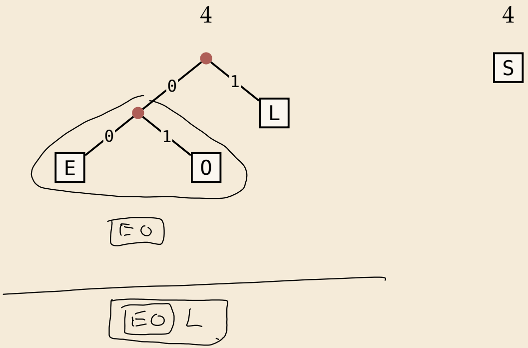
# Huffman's algorithm – Example

- Example text: $S = \texttt{LOSSLESS}$ $\rightsquigarrow$ $\Sigma_S = \{\texttt{E}, \texttt{L}, \texttt{O}, \texttt{S}\}$
- Character frequencies: $\texttt{E}: 1$, $\texttt{L}: 2$, $\texttt{O}: 1$, $\texttt{S}: 4$



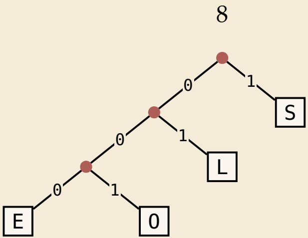
# Huffman's algorithm – Example

- Example text: $S$ = LOSSLESS $\rightsquigarrow$ $\Sigma_S = \{$E, L, O, S$\}$
- Character frequencies: E : 1, L : 2, O : 1, S : 4



# Huffman's algorithm – Example

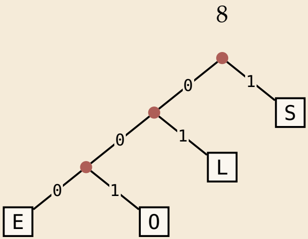
- Example text: $S$ = LOSSLESS $\rightsquigarrow$ $\Sigma_S = \{$E, L, O, S$\}$
- Character frequencies: E : 1, L : 2, O : 1, S : 4




E(L) = 01

# Huffman's algorithm – Example

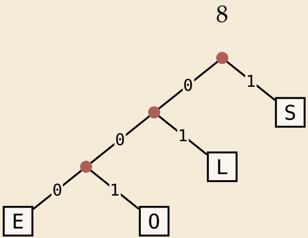
- Example text: $S$ = LOSSLESS $\rightsquigarrow$ $\Sigma_S = \{$E, L, O, S$\}$
- Character frequencies: E : 1, L : 2, O : 1, S : 4




$\rightsquigarrow$ *Huffman tree* (code trie for Huffman code)

# Huffman's algorithm – Example

- Example text: $S$ = LOSSLESS ⇝ $\Sigma_S = \{\texttt{E}, \texttt{L}, \texttt{O}, \texttt{S}\}$
- Character frequencies: E : 1, L : 2, O : 1, S : 4




⇝ *Huffman tree* (code trie for Huffman code)

LOSSLESS → 01001110100011

compression ratio: $\frac{14}{8 \cdot \log 4} = \frac{14}{16} \approx 88\%$

(but: must store code)

# Huffman tree – tie breaking

- The above procedure is ambiguous:
  - which characters to choose when weights are equal?
  - which subtree goes left, which goes right?

- For COMP 526: always use the following rule:

1. To break ties when selecting the two characters,
   first use the smallest letter according to the alphabetical order,
   or the tree containing the smallest alphabetical letter.
2. When combining two trees of different values,
   place the lower-valued tree on the left (corresponding to a `0`-bit).
3. When combining trees of equal value,
   place the one containing the smallest letter to the left.

$\rightsquigarrow$ practice in tutorials

# Encoding with Huffman code

- The overall encoding procedure is as follows:
  - **Pass 1:** Count character frequencies in $S$
  - Construct Huffman code $E$ (as above)
  - Store the Huffman code in $C$ (details omitted) canonicalize
  - **Pass 2:** Encode each character in $S$ using $E$ and append result to $C$
- Decoding works as follows:
  - Decode the Huffman code $E$ from $C$. (details omitted)
  - Decode $S$ character by character from $C$ using the code trie.
- Note: Decoding is much simpler/faster!

# Huffman code – Optimality

### Theorem 5.1 (Optimality of Huffman's Algorithm)

Given $\Sigma$ and $w : \Sigma \to \mathbb{R}_{\geq 0}$, Huffman's Algorithm computes codewords $E : \Sigma \to \{0,1\}^\star$ with minimal expected codeword length $\ell(E) = \sum_{c \in \Sigma} w(c) \cdot |E(c)|$ among all prefix-free codes for $\Sigma$. ◀

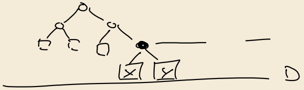
# Huffman code – Optimality

## Theorem 5.1 (Optimality of Huffman's Algorithm)

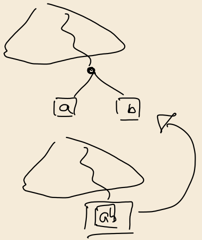
Given $\Sigma$ and $w : \Sigma \to \mathbb{R}_{\geq 0}$, Huffman's Algorithm computes codewords $E : \Sigma \to \{0,1\}^\star$ with minimal expected codeword length $\ell(E) = \sum_{c \in \Sigma} w(c) \cdot |E(c)|$ among all prefix-free codes for $\Sigma$.

IB: $\sigma \leq 2$ only one possible $\ell(E)$ ✓

*Proof sketch:* by induction over $\sigma = |\Sigma|$   IS: $\sigma \geq 3$




- Given any optimal prefix-free code $E^*$ (as its code trie).
- code trie $\rightsquigarrow$ $\exists$ two sibling leaves $x, y$ at largest depth $D$
- swap characters in leaves to have two lowest-weight characters a, b in $x, y$ (that can only make $\ell$ smaller, so still optimal)
- any optimal code for $\Sigma' = \Sigma \setminus \{\text{a}, \text{b}\} \cup \{\boxed{\text{ab}}\}$ yields optimal code for $\Sigma$ by replacing leaf $\boxed{\text{ab}}$ by internal node with children a and b.




$\rightsquigarrow$ recursive call yields optimal code for $\Sigma'$ by inductive hypothesis, so Huffman's algorithm finds optimal code for $\Sigma$.

## 5.4 Entropy

# Entropy

## Definition 5.2 (Entropy)

Given probabilities $p_1, \ldots, p_n$ (for outcomes $1, \ldots, n$ of a random variable), the *entropy* of the distribution is defined as

$p_i \in [0,1]$

$$\mathcal{H}(p_1, \ldots, p_n) = -\sum_{i=1}^{n} p_i \lg p_i = \sum_{i=1}^{n} p_i \lg\left(\frac{1}{p_i}\right)$$

$$0 \le \mathcal{H}(p_1, \ldots, p_n) \le \lg n$$

maximal if $p_i = \frac{1}{n}$

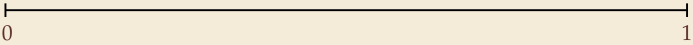
# Entropy

## Definition 5.2 (Entropy)

Given probabilities $p_1, \ldots, p_n$ (for outcomes $1, \ldots, n$ of a random variable), the *entropy* of the distribution is defined as

$$\mathcal{H}(p_1, \ldots, p_n) \;=\; -\sum_{i=1}^{n} p_i \lg p_i \;=\; \sum_{i=1}^{n} p_i \lg\left(\frac{1}{p_i}\right)$$

- entropy is a **measure** of **information** content of a distribution
  - *"20 Questions on $[0,1)$":* Land inside my interval by halving.



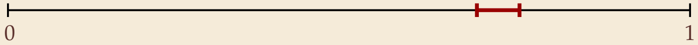
# Entropy

## Definition 5.2 (Entropy)

Given probabilities $p_1, \ldots, p_n$ (for outcomes $1, \ldots, n$ of a random variable), the *entropy* of the distribution is defined as

$$\mathcal{H}(p_1, \ldots, p_n) \;=\; -\sum_{i=1}^{n} p_i \lg p_i \;=\; \sum_{i=1}^{n} p_i \lg\left(\frac{1}{p_i}\right)$$

- entropy is a **measure** of **information** content of a distribution
  - *"20 Questions on $[0,1)$":* Land inside my interval by halving.



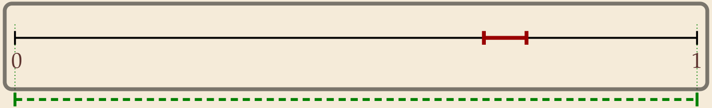
# Entropy

## Definition 5.2 (Entropy)

Given probabilities $p_1, \ldots, p_n$ (for outcomes $1, \ldots, n$ of a random variable), the *entropy* of the distribution is defined as

$$\mathcal{H}(p_1, \ldots, p_n) \;=\; -\sum_{i=1}^{n} p_i \lg p_i \;=\; \sum_{i=1}^{n} p_i \lg\left(\frac{1}{p_i}\right)$$

- entropy is a **measure** of **information** content of a distribution
  - *"20 Questions on* $[0,1)$*":* Land inside my interval by halving.



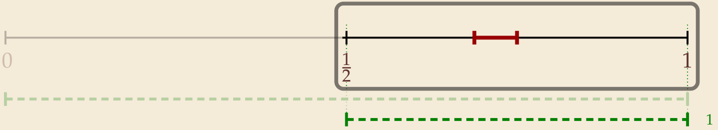
# Entropy

## Definition 5.2 (Entropy)

Given probabilities $p_1, \ldots, p_n$ (for outcomes $1, \ldots, n$ of a random variable), the *entropy* of the distribution is defined as

$$\mathcal{H}(p_1, \ldots, p_n) \;=\; -\sum_{i=1}^{n} p_i \lg p_i \;=\; \sum_{i=1}^{n} p_i \lg\left(\frac{1}{p_i}\right)$$

- entropy is a **measure** of **information** content of a distribution
  - *"20 Questions on $[0, 1)$":* Land inside my interval by halving.



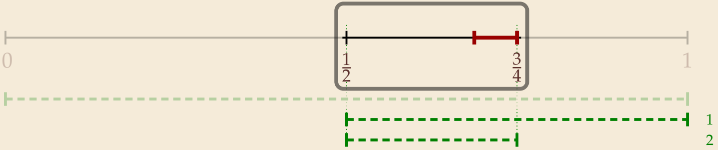
# Entropy

## Definition 5.2 (Entropy)

Given probabilities $p_1, \ldots, p_n$ (for outcomes $1, \ldots, n$ of a random variable), the *entropy* of the distribution is defined as

$$\mathcal{H}(p_1, \ldots, p_n) \;=\; -\sum_{i=1}^{n} p_i \lg p_i \;=\; \sum_{i=1}^{n} p_i \lg\left(\frac{1}{p_i}\right)$$

- entropy is a **measure** of **information** content of a distribution
  - *"20 Questions on* $[0,1)$*":* Land inside my interval by halving.



# Entropy

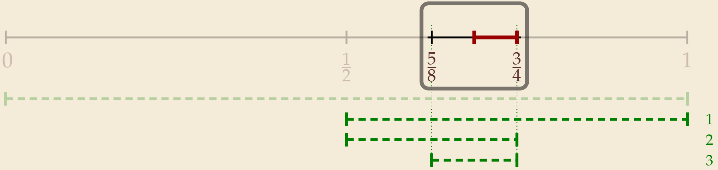
## Definition 5.2 (Entropy)

Given probabilities $p_1, \ldots, p_n$ (for outcomes $1, \ldots, n$ of a random variable), the *entropy* of the distribution is defined as

$$\mathcal{H}(p_1, \ldots, p_n) \quad = \quad -\sum_{i=1}^{n} p_i \lg p_i \quad = \quad \sum_{i=1}^{n} p_i \lg\left(\frac{1}{p_i}\right)$$

◀

- entropy is a **measure** of **information** content of a distribution
  - *"20 Questions on* $[0,1)$*":* Land inside my interval by halving.



# Entropy

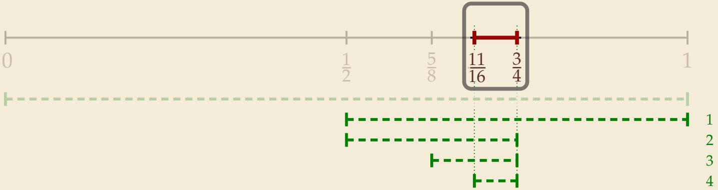
## Definition 5.2 (Entropy)

Given probabilities $p_1, \ldots, p_n$ (for outcomes $1, \ldots, n$ of a random variable), the *entropy* of the distribution is defined as

$$\mathcal{H}(p_1, \ldots, p_n) \;=\; -\sum_{i=1}^{n} p_i \lg p_i \;=\; \sum_{i=1}^{n} p_i \lg\left(\frac{1}{p_i}\right)$$

◀

- entropy is a **measure** of **information** content of a distribution
  - *"20 Questions on* $[0,1)$*":* Land inside my interval by halving.



# Entropy

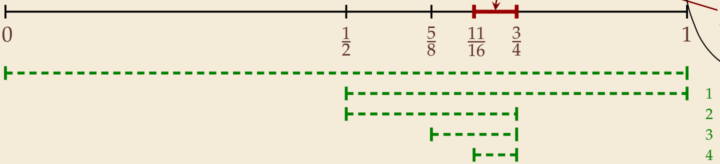
## Definition 5.2 (Entropy)

Given probabilities $p_1, \ldots, p_n$ (for outcomes $1, \ldots, n$ of a random variable), the *entropy* of the distribution is defined as

$\lg = \log_2$

$$\mathcal{H}(p_1, \ldots, p_n) \;=\; -\sum_{i=1}^{n} p_i \lg p_i \;=\; \sum_{i=1}^{n} p_i \lg\left(\frac{1}{p_i}\right)$$

- entropy is a **measure** of **information** content of a distribution
  - *"20 Questions on $[0,1)$":* Land inside my interval by halving.




$p_i = \frac{3}{4} - \frac{11}{16} = \frac{1}{16}$

$\rightsquigarrow \lg(1/p_i) = 4$

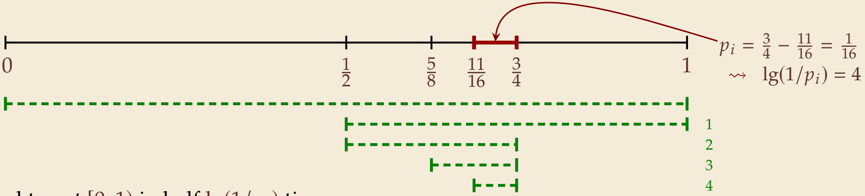
# Entropy

## Definition 5.2 (Entropy)

Given probabilities $p_1, \ldots, p_n$ (for outcomes $1, \ldots, n$ of a random variable), the *entropy* of the distribution is defined as

$$\mathcal{H}(p_1, \ldots, p_n) \;=\; -\sum_{i=1}^{n} p_i \lg p_i \;=\; \sum_{i=1}^{n} p_i \lg\left(\frac{1}{p_i}\right)$$

- entropy is a **measure** of **information** content of a distribution
  - *"20 Questions on* $[0,1)$*":* Land inside my interval by halving.




$\rightsquigarrow$ Need to cut $[0,1)$ in half $\lg(1/p_i)$ times

- more precisely: the expected number of bits (Yes/No questions) required to nail down the random value

# Entropy and Huffman codes

- would ideally encode value $i$ using $\lg(1/p_i)$ bits

  not always possible; cannot use codeword of $1.5$ bits . . .

not as length of single codeword that is;
but can be possible *on average*!

# Entropy and Huffman codes

- would ideally encode value $i$ using $\lg(1/p_i)$ bits

  not as length of single codeword that is;
  but can be possible *on average*!

  not always possible; cannot use codeword of $1.5$ bits . . . but:

### Theorem 5.3 (Entropy bounds for Huffman codes)

For any probabilities $p_1, \ldots, p_\sigma$ for $\Sigma = \{a_1, \ldots, a_\sigma\}$, the Huffman code $E$ for $\Sigma$ with weights $p(a_i) = p_i$ satisfies $\boxed{\mathcal{H} \le \ell(E) \le \mathcal{H} + 1}$ where $\underline{\mathcal{H} = \mathcal{H}(p_1, \ldots, p_\sigma)}$. ◀

# Entropy and Huffman codes

- would ideally encode value $i$ using $\lg(1/p_i)$ bits

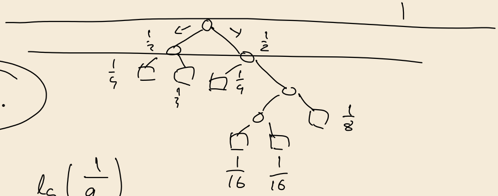
  not always possible; cannot use codeword of $1.5$ bits ... but:

not as length of single codeword that is; but can be possible *on average*!

## Theorem 5.3 (Entropy bounds for Huffman codes)

For any probabilities $p_1, \ldots, p_\sigma$ for $\Sigma = \{a_1, \ldots, a_\sigma\}$, the Huffman code $E$ for $\Sigma$ with weights $p(a_i) = p_i$ satisfies $\boxed{\mathcal{H} \underset{q}{\le} \ell(E) \le \mathcal{H} + 1}$ where $\mathcal{H} = \mathcal{H}(p_1, \ldots, p_\sigma)$.

Kraft's




*Proof sketch:*

- $\ell(E) \ge \mathcal{H}$

  Any prefix-free code $E$ induces weights $q_i = 2^{-|E(a_i)|}$.

  By *Kraft's Inequality*, we have $q_1 + \cdots + q_\sigma \le 1$.

  Hence we can apply *Gibb's Inequality* to get

$$\underbrace{\mathcal{H} = \sum_{i=1}^{\sigma} p_i \lg\left(\frac{1}{p_i}\right) \overset{\checkmark}{\le} \sum_{i=1}^{\sigma} p_i \lg\left(\frac{1}{q_i}\right)} = \ell(E).$$

$$\lg\left(\frac{1}{q_i}\right)$$

$$= \lg\left(\frac{1}{2^{-|E(a_i)|}}\right) = \lg\left(2^{|E(a_i)|}\right)$$

$$= |E(a_i)|$$

# Entropy and Huffman codes [2]

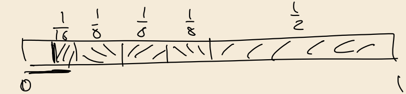
*Proof sketch (continued):*

$\approx p_i$ but next smaller $2^{-k}$

- $\ell(E) \le \mathcal{H} + 1$

  Set $q_i = 2^{-\lceil \lg(1/p_i) \rceil}$. We have $\sum_{i=1}^{\sigma} p_i \lg\left(\frac{1}{q_i}\right) = \sum_{i=1}^{\sigma} p_i \underbrace{\lceil \lg(1/p_i) \rceil}_{\le \lg(1/p_i)+1} \le \mathcal{H} + 1.$

  We construct a code $E'$ for $\Sigma$ with $|E'(a_i)| \le \lg(1/q_i)$ as follows;
  w.l.o.g. assume $q_1 \le q_2 \le \cdots \le q_\sigma$

  - If $\sigma = 2$, $E'$ uses a single bit each.
    Here, $q_i \le 1/2$, so $\lg(1/q_i) \ge 1 = |E'(a_i)|$ ✓
  - If $\sigma \ge 3$, we merge $a_1$ and $a_2$ to $\boxed{a_1 a_2}$, assign it weight $2q_2$ and recurse.
    If $q_1 = q_2$, this is like Huffman; otherwise, $q_1$ is a unique smallest value and $q_2 + q_2 + \cdots + q_\sigma \le 1$.

    By the inductive hypothesis, we have $\left|E'(\boxed{a_1 a_2})\right| \le \lg\left(\frac{1}{2q_2}\right) = \lg\left(\frac{1}{q_2}\right) - 1.$

    By construction, $|E'(a_1)| = |E'(a_2)| = \left|E'(\boxed{a_1 a_2})\right| + 1$, so $|E'(a_1)| \le \lg(\frac{1}{q_1})$ and $|E'(a_2)| \le \lg(\frac{1}{q_2})$.

By optimality of $E$, we have $\ell(E) \le \ell(E') \le \sum_{i=1}^{\sigma} p_i \lg\left(\frac{1}{q_i}\right) \le \mathcal{H} + 1.$

''

$\sum p_i |E(a_i)|$

# Clicker Question

When does Huffman coding yield more efficient compression than a fixed-length character encoding?

**A** always

**B** when $\mathcal{H} \approx \lg(\sigma)$

**C** when $\mathcal{H} < \lg(\sigma)$

**D** when $\mathcal{H} < \lg(\sigma) - 1$

**E** when $\mathcal{H} \approx 1$

→ `sli.do/comp526`

# Clicker Question

When does Huffman coding yield more efficient compression than a fixed-length character encoding?

- **A** always
- **B** ~~when $\mathcal{H} \approx \lg(\sigma)$~~
- **C** ~~when $\mathcal{H} < \lg(\sigma)$~~
- **D** when $\mathcal{H} < \lg(\sigma) - 1$
- **E** ~~when $\mathcal{H} \approx 1$~~

→ sli.do/comp526

# Empirical Entropy

- Theorem 5.3 works for *any* character *probabilities* $p_1, \ldots, p_\sigma$

  $\ldots$ but we only have a string $S$! (nothing random about it!)

# Empirical Entropy

- Theorem 5.3 works for *any* character *probabilities* $p_1, \ldots, p_\sigma$

  ... but we only have a string $S$! (nothing random about it!)

use relative frequencies: $p_i = \dfrac{|S|_{a_i}}{|S|} = \dfrac{\#\text{occurences of } a_i \text{ in string } S}{\text{length of } S}$

- Recall: For $S[0..n)$ over $\Sigma = \{a_1, \ldots, a_\sigma\}$,
  length of Huffman-coded text is

$$|C| \;=\; \sum_{i=1}^{\sigma} |S|_{a_i} \cdot |E(a_i)| \;=\; n \sum_{i=1}^{\sigma} \overbrace{\boxed{\frac{|S|_{a_i}}{n}}}^{=p_i} \cdot |E(a_i)| \;=\; n\ell(E)$$

$\leadsto$ Theorem 5.3 tells us rather precisely how well Huffman compresses:
$\mathcal{H}_0(S) \cdot n \;\le\; |C| \;\le\; (\mathcal{H}_0(S) + 1)n$

zero-th order empirical entropy

- $\mathcal{H}_0(S) \;=\; \mathcal{H}\left(\dfrac{|S|_{a_1}}{n}, \ldots, \dfrac{|S|_{a_\sigma}}{n}\right) \;=\; \displaystyle\sum_{i=1}^{\sigma} \frac{n}{|S|_{a_i}} \log_2\left(\frac{|S|_{a_i}}{n}\right)$ is called the *empirical entropy* of $S$

# Huffman coding – Discussion

- running time complexity: $O(\sigma \log \sigma)$ to construct code
  - build PQ + $\sigma \cdot$ (2 deleteMins and 1 insert)
  - can do $\Theta(\sigma)$ time when characters already sorted by weight
  - time for encoding text (after Huffman code done): $O(n + |C|)$
- many variations in use (tie-breaking rules, estimated frequencies, adaptive encoding, . . . )

# Huffman coding – Discussion

- running time complexity: $O(\sigma \log \sigma)$ to construct code
  - build PQ + $\sigma \cdot$ (2 deleteMins and 1 insert)
  - can do $\Theta(\sigma)$ time when characters already sorted by weight
  - time for encoding text (after Huffman code done): $O(n + |C|)$
- many variations in use (tie-breaking rules, estimated frequencies, adaptive encoding, ...)

👍 optimal prefix-free character encoding

👍 very fast decoding

👎 needs 2 passes over source text for encoding
  - one-pass variants possible, but more complicated

👎 have to store code alongside with coded text