7 Text Compression

25 November 2024

Prof. Dr. Sebastian Wild

CS566 (Wintersemester 2024/25)
Philipps-Universität Marburg

version `2024-11-25 16:00`

Learning Outcomes

Unit 7: ***Text Compression***

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 a specific type of data.

Outline

7 Text Compression

7.1 Context

Overview

- Unit 6 & 13: How to *work* with strings
  - finding substrings
  - finding approximate matches ⇝ Unit ~~8~~ 13
  - finding repeated parts ⇝ Unit ~~8~~ 13
  - ...
  - assumed character array (random access)!
- Unit 7 & 8: How to *store/transmit* strings
  - computer memory: must be binary
  - how to compress strings (save space)
  - how to robustly transmit over noisy channels ⇝ Unit 8

Clicker Question

What compression methods do you know?

→ sli.do/cs566

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 $S \rightsquigarrow C$
- **decoding:** algorithm mapping coded texts back to original source text $C \rightsquigarrow S$

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.

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

- 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

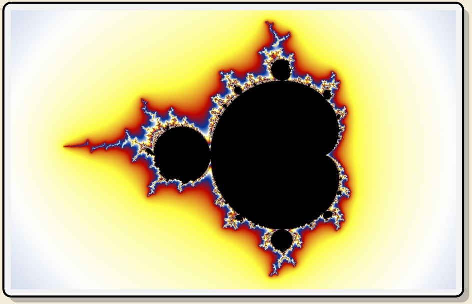
Do you know what uncomputable/undecidable problems (halting problem, Post's correspondence problem, . . .) are?

- 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/cs566

Limits of algorithmic compression

Is this image compressible?



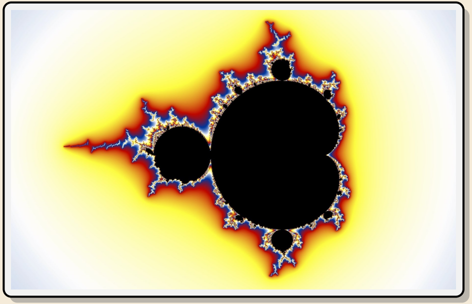
Limits of algorithmic compression

Is this image compressible?

visualization of Mandelbrot set

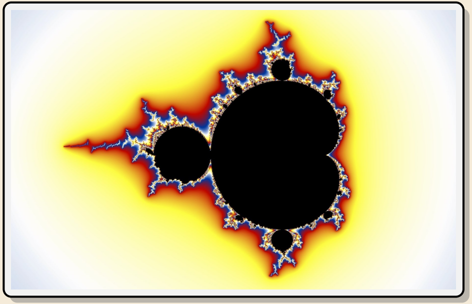
- Clearly a complex shape!
- Will not compress (too) well using, say, PNG.
- but:
  - completely defined by mathematical formula

  $\rightsquigarrow$ **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
  - ⇝ **can be generated by a very small program!**

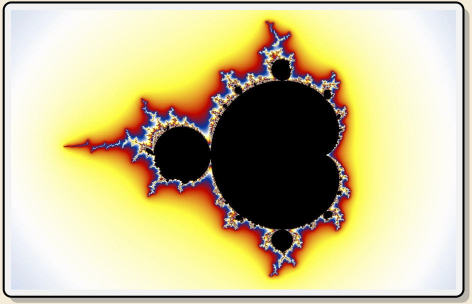
⇝ ***Kolmogorov complexity***

- C = *any program* that outputs S

  self-extracting archives!

  needs fixed machine model, but compilers transfer results
- 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!

  needs fixed machine model, but compilers transfer results
- 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 $\rightarrow$ Part I
  2. **repetitive texts**
     different parts in the text are (almost) identical $\rightarrow$ 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 $\rightarrow$ Part I

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

There is no such thing as a free lunch!

Not *everything* is compressible ($\rightarrow$ tutorials)

$\rightsquigarrow$ focus on versatile methods that often work

Part I

Exploiting character frequencies

7.2 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 $c \in \Sigma_{\cancel{C} S}$

- **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

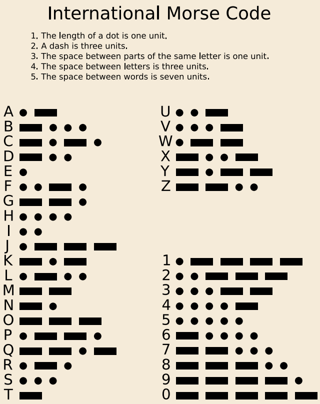
👍 Encoding & Decoding as fast as it gets

👎 Unless all characters equally likely, it wastes a lot of space

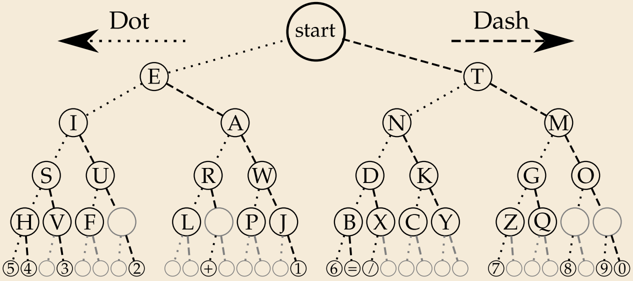
👎 inflexible (how to support adding a new character?)

Variable-length codes

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



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/cs566

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/cs566

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 (154 998 as of Nov 2024, 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 characters 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 $C = \underbrace{\texttt{110}}_{\texttt{b}}\underbrace{\texttt{0}}_{\texttt{a}}\underbrace{\texttt{10}}_{\texttt{n}}\underbrace{\texttt{0}}_{\texttt{a}}\underbrace{\texttt{10}}_{\texttt{n}}\underbrace{\texttt{0}}_{\texttt{a}}$

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

ϟ 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?

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

⚡ 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?

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

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

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

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

Code tries

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

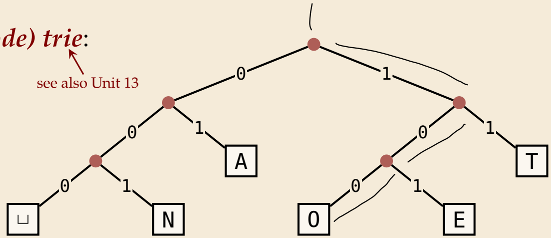
- **Example:**

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 13

- 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`
  - Decode `111000001010111`

T O

Code tries

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

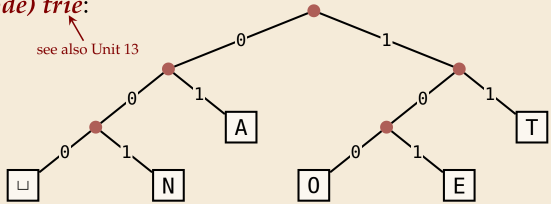
- **Example:**

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

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

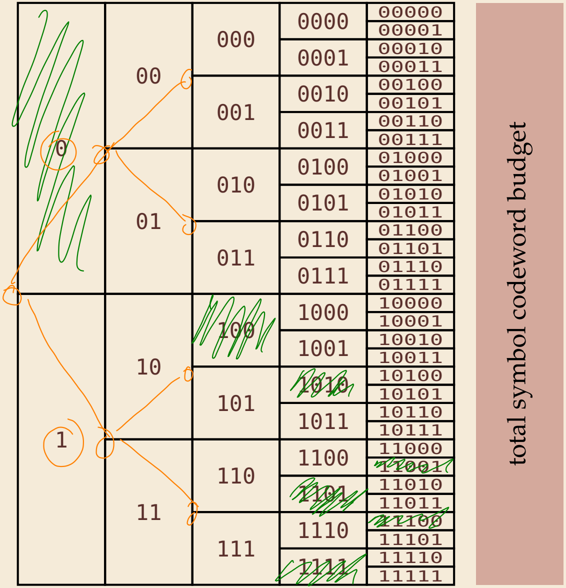
see also Unit 13

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

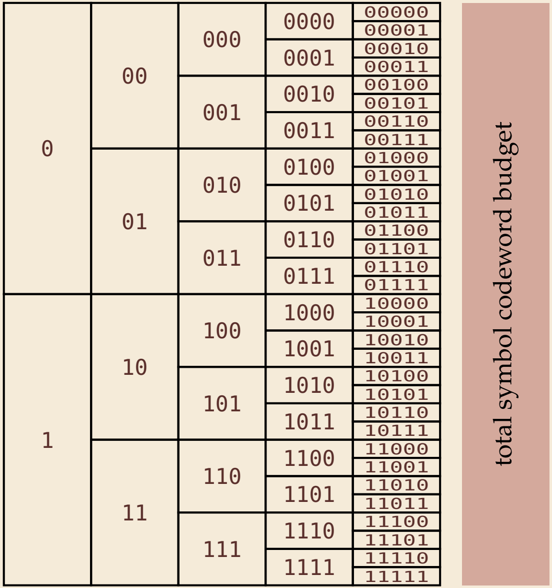


- Example for using the code trie:
  - Encode `AN␣ANT` ⟶ `010010000100111`
  - Decode `111000001010111` ⟶ `TO␣EAT`

The Codeword Supermarket



The Codeword Supermarket

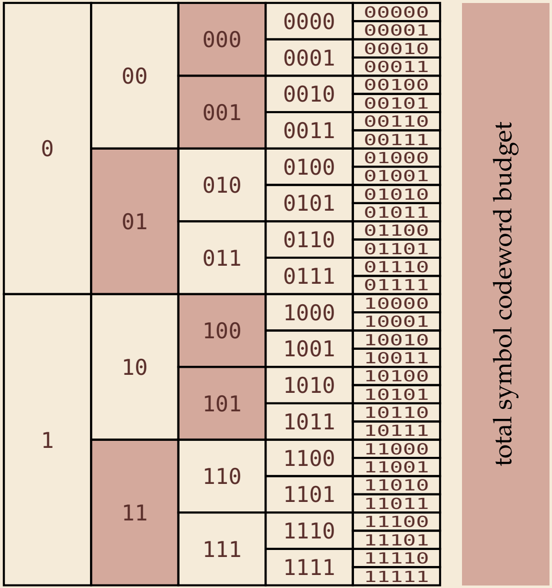


- Can "spend" at most budget of 1 across all codewords
  - Codeword with ℓ bits costs $2^{-\ell}$
- *Kraft-McMillan inequality:* any uniquely decodable code with codeword lengths $\ell_1, \ldots, \ell_\sigma$ satisfies

$$\sum_{i=1}^{\sigma} 2^{-\ell_i} \le 1$$

and for any such lengths there is a prefix-free code

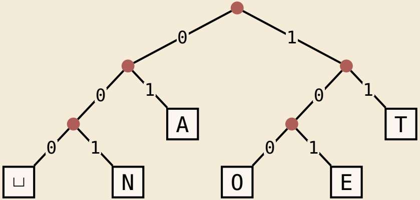
The Codeword Supermarket



- Can "spend" at most budget of 1 across all codewords
  - Codeword with ℓ bits costs $2^{-\ell}$
- *Kraft-McMillan inequality:* any uniquely decodable code with codeword lengths $\ell_1, \ldots, \ell_\sigma$ satisfies

$$\sum_{i=1}^{\sigma} 2^{-\ell_i} \leq 1$$

and for any such lengths there is a prefix-free code



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)

7.3 Huffman Codes

Character frequencies

- **Goal:** Find character encoding that produces short coded text
- Convention here: fix $\Sigma_C = \{0, 1\}$ (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:** Σ and weights $w : \Sigma \rightarrow \mathbb{R}_{\geq 0}$
- **Goal:** prefix-free code E (= code trie) for Σ that minimizes coded text length

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

Huffman coding

e. g. frequencies / probabilities

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

- Quite ambitious! *Is this efficiently possible?*

Huffman's algorithm

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

Huffman's algorithm:

1. Find two characters $\mathtt{a}$, $\mathtt{b}$ 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 $\mathtt{a}$ and $\mathtt{b}$ 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.

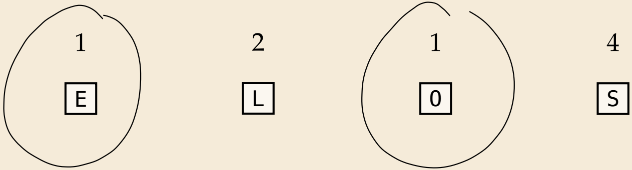
Huffman's algorithm:

1. Find two characters a, b with lowest weights.
   - We will encode them with the same prefix, plus one distinguishing bit, i. e., $E(\mathtt{a}) = u\mathtt{0}$ 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 $u = E(\boxed{\mathtt{ab}})$.

- 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

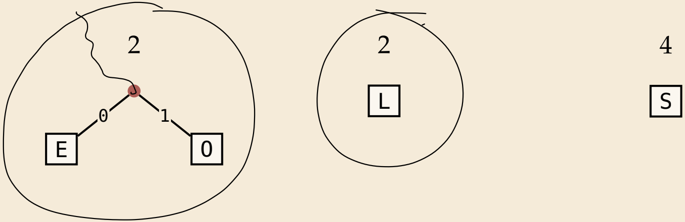
Huffman's algorithm – Example

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



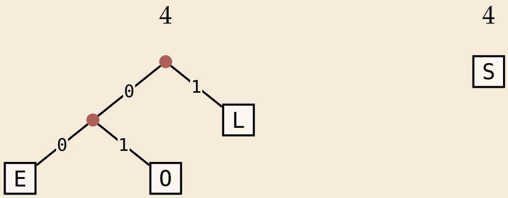
Huffman's algorithm – Example

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



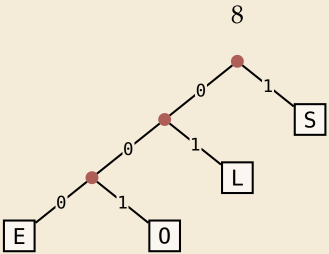
Huffman's algorithm – Example

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



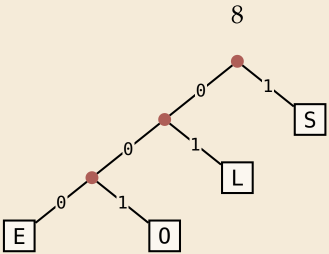
Huffman's algorithm – Example

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



Huffman's algorithm – Example

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

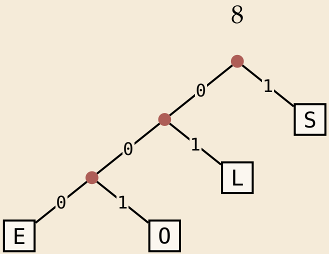


$\rightsquigarrow$ *Huffman tree* (code trie for Huffman code)

Huffman's algorithm – Example

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

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



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

$\texttt{LOSSLESS} \rightarrow \texttt{01001110100011}$ compression ratio: $\frac{14}{8 \cdot \log 4} = \frac{14}{16} \approx 88\%$

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 CS 566: 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)
  - **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!