7 Text Compression

24 November 2025

Prof. Dr. Sebastian Wild

CS566 (Wintersemester 2025–26)
Philipps-Universität Marburg

version `2025-11-25 13:57`

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 $\rightsquigarrow$ Unit 13
  - finding repeated parts $\rightsquigarrow$ Unit 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 $\rightsquigarrow$ 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
  decoding ↑ ↓ encoding 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

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 $S \to C \to S'$ $\quad S \approx S'$ for human perception
  - **lossless compression** always decodes S exactly $S \to C \to S$
- 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

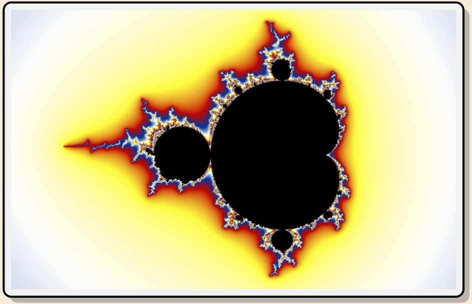
Do you know what uncomputable/undecidable problems (halting problem, Post's correspondence problem, . . .) are?

- **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?



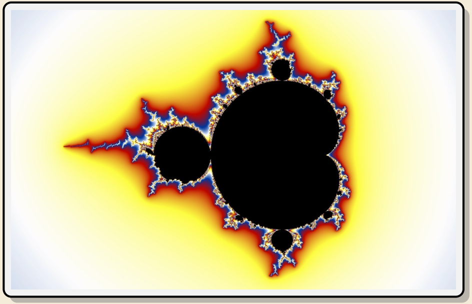
Limits of algorithmic compression

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

  ⇝ **can be generated by a very small program!**

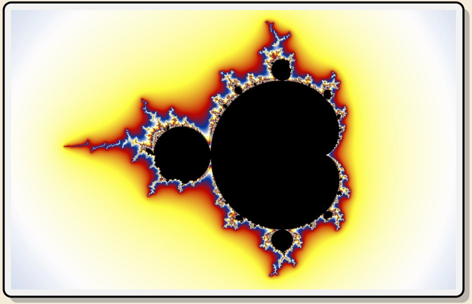


Limits of algorithmic compression

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***

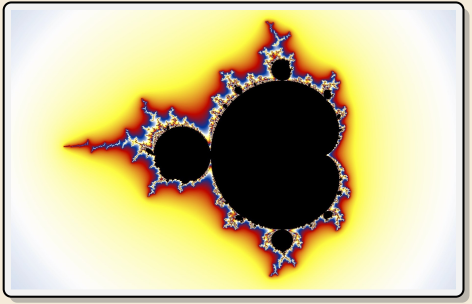
- C = *any program* that outputs S

  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

Digression: Uncomputability of Kolmogorov Complexity

- **Fact:** There are strings of arbitrarily large Kolmogorov complexity.
  - Otherwise only finitely many strings (deterministic programs!)

eval(" ... ")

Digression: Uncomputability of Kolmogorov Complexity

- **Fact:** There are strings of arbitrarily large Kolmogorov complexity.
  - Otherwise only finitely many strings (deterministic programs!)

Theorem 7.1

The Kolmogorov complexity is uncomputable. ◀

Digression: Uncomputability of Kolmogorov Complexity

- **Fact:** There are strings of arbitrarily large Kolmogorov complexity.
  - Otherwise only finitely many strings (deterministic programs!)

Theorem 7.1
The Kolmogorov complexity is uncomputable.

Proof:
length of shortest program for S
Assume otherwise, i. e., $K(S)$ computes Kolmogorov complexity of strings S.
$\rightsquigarrow$ K has some length $|K|$.

Digression: Uncomputability of Kolmogorov Complexity

- **Fact:** There are strings of arbitrarily large Kolmogorov complexity.
  - Otherwise only finitely many strings (deterministic programs!)

Theorem 7.1

The Kolmogorov complexity is uncomputable.

Proof:

Assume otherwise, i. e., $K(S)$ computes Kolmogorov complexity of strings S.

$\rightsquigarrow$ K has some length $|K|$.

Then the following program finds a string of large Kolmogorov complexity.

```
procedure findComplexString():
    for n := 1, 2, ...:
        for S ∈ Σ^n:
            if K(S) > |K| + 1000
                return S
```

Digression: Uncomputability of Kolmogorov Complexity

- **Fact:** There are strings of arbitrarily large Kolmogorov complexity.
  - Otherwise only finitely many strings (deterministic programs!)

Theorem 7.1

The Kolmogorov complexity is uncomputable.

Proof:

Assume otherwise, i. e., $K(S)$ computes Kolmogorov complexity of strings S.

$\rightsquigarrow$ K has some length $|K|$.

Then the following program finds a string of large Kolmogorov complexity.

```
procedure findComplexString():
    for n := 1, 2, ...:
        for S ∈ Σ^n:
            if K(S) > |K| + 1000
                return S
```

But findComplexString also outputs S and is smaller than $|K| + 1000$! ϟ ∎

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 → 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

7.2 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)

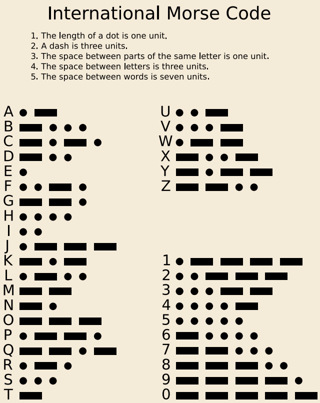
Fixed-length codes – Discussion

- 👍 Encoding & Decoding as fast as it gets

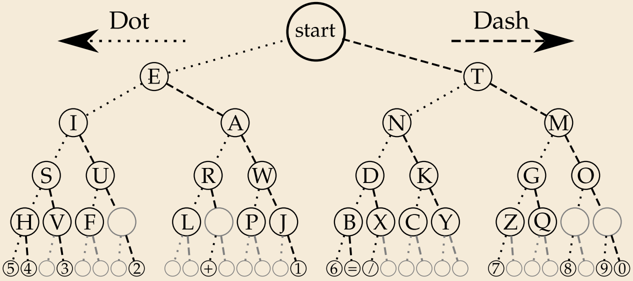
- 👎 Unless all characters equally likely, it wastes a lot of space
- 👎 inflexible (how to support adding a new character?)

Variable-length codes

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



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 $C = \underset{\texttt{b}}{\underline{\texttt{110}}}\underset{\texttt{a}}{\texttt{0}}\underset{\texttt{n}}{\underline{\texttt{10}}}\underset{\texttt{a}}{\texttt{0}}\underset{\texttt{n}}{\underline{\texttt{10}}}\underset{\texttt{a}}{\texttt{0}}$

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

↯ $C = \texttt{1100100100}$ 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?

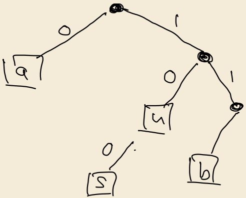
Pitfall in variable-length codes

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

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?

s ≟ na

$E(\texttt{n}) = 10$ 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: $\boxed{\text{No codeword is a prefix of another.}}$

prefix-free $\implies$ instantaneously decodable $\implies$ 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 $c, c' \in \Sigma_S$.

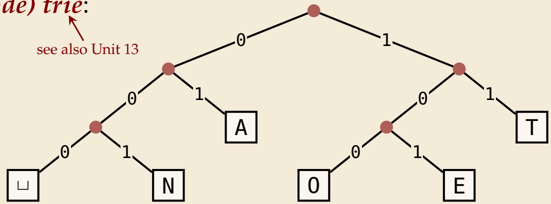
- **Example:**

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`   01 001 000
  - Decode `11100000010101111`
    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 $c, c' \in \Sigma_S$.

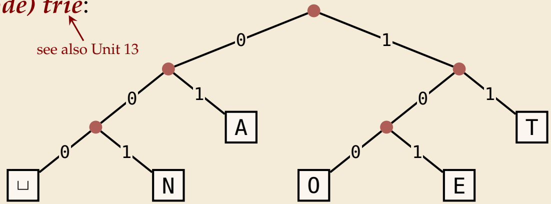
- **Example:**

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

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

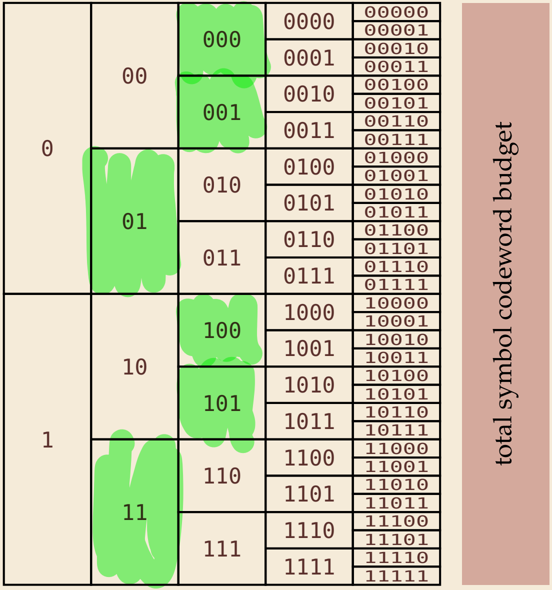
  see also Unit 13

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



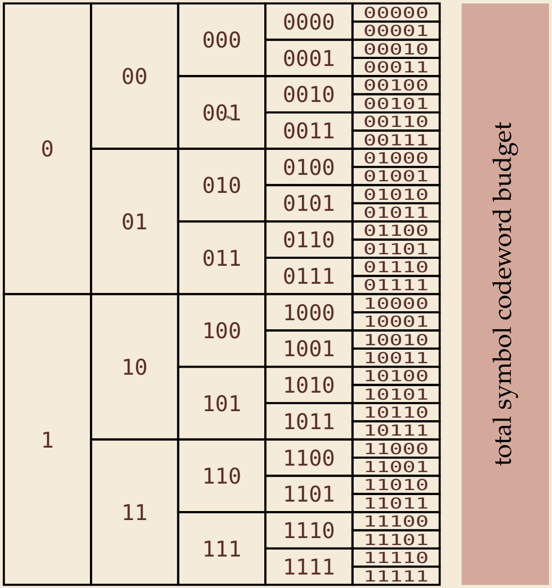
- Example for using the code trie:
  - Encode AN␣ANT $\longrightarrow$ 010010000100111
  - Decode 111000001010111 $\longrightarrow$ TO␣EAT

The Codeword Supermarket



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

The Codeword Supermarket

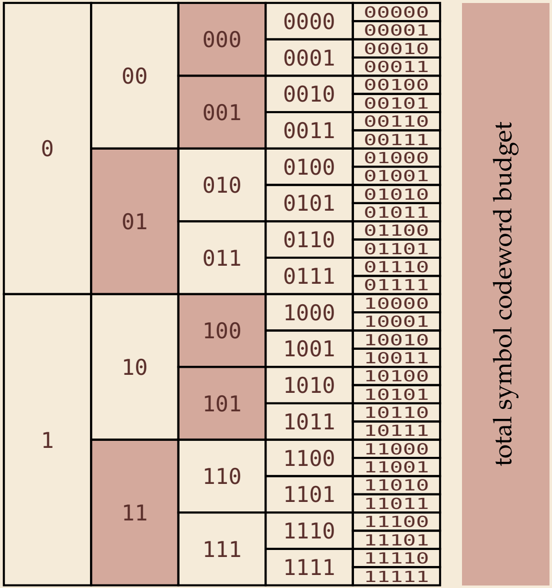


- Can "spend" at most budget of 1 across all codewords
  - Codeword with ℓ bits costs $2^{-\ell}$
- *Kraft-McMillan inequality:* any uniquely decodable code with codeword lengths $\ell_1, \ldots, \ell_\sigma$ satisfies

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

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

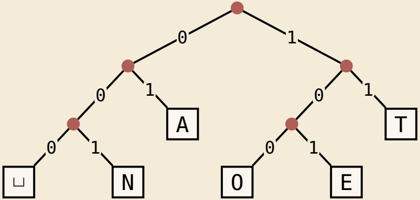
The Codeword Supermarket



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

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

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 `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 $u = E(\boxed{\texttt{ab}})$.

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

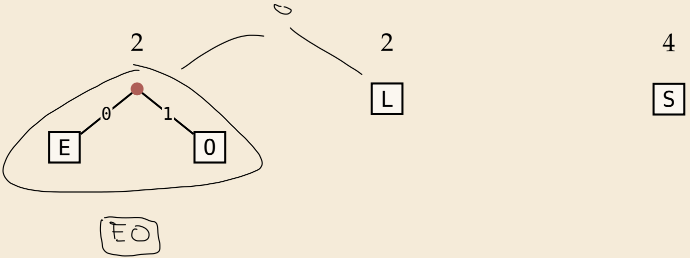
Huffman's algorithm – Example

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

1	2	1	4
$\boxed{\texttt{E}}$	$\boxed{\texttt{L}}$	$\boxed{\texttt{O}}$	$\boxed{\texttt{S}}$

Huffman's algorithm – Example

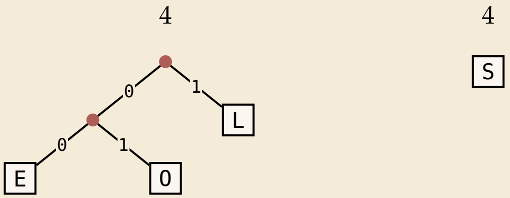
- Example text: $S =$ LOSSLESS $\rightsquigarrow$ $\Sigma_S = \{$E, L, O, S$\}$
- Character frequencies: E : 1, L : 2, O : 1, S : 4



Huffman's algorithm – Example

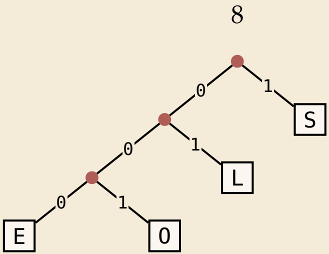
- Example text: $S =$ LOSSLESS $\rightsquigarrow$ $\Sigma_S = \{$E, L, O, S$\}$

- Character frequencies: E : 1, L : 2, O : 1, S : 4



Huffman's algorithm – Example

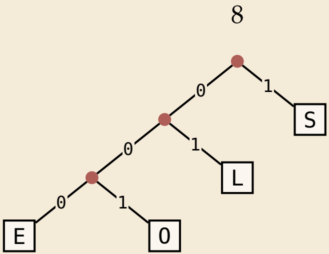
- Example text: S = `LOSSLESS` $\rightsquigarrow$ $\Sigma_S = \{$`E`, `L`, `O`, `S`$\}$
- Character frequencies: `E` : 1, `L` : 2, `O` : 1, `S` : 4



Huffman's algorithm – Example

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

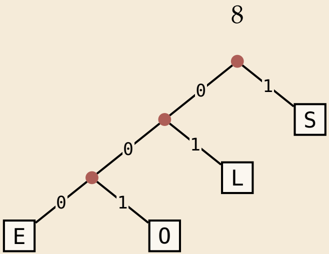
- Character frequencies: $\texttt{E} : 1$, $\texttt{L} : 2$, $\texttt{O} : 1$, $\texttt{S} : 4$



$\rightsquigarrow$ *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: $\texttt{E} : 1,\quad \texttt{L} : 2,\quad \texttt{O} : 1,\quad \texttt{S} : 4$



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

$\texttt{LOSSLESS} \rightarrow \underbrace{\texttt{01}}_{L}\underbrace{\texttt{001}}\texttt{110100011}$ 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 (tree containing the) smallest letter in alphabetical order.
>
> ***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!

Huffman code – Optimality

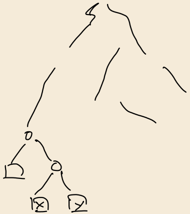
Theorem 7.2 (Optimality of Huffman's Algorithm)

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

Huffman code – Optimality

Theorem 7.2 (Optimality of Huffman's Algorithm)

Given Σ 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 Σ.



Proof sketch: by induction over $\sigma = |\Sigma|$

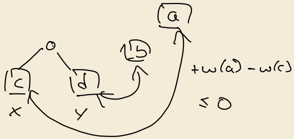
- Given any optimal prefix-free code E^* (as its code trie).
- code trie $\rightsquigarrow$ $\exists$ two sibling leaves x, y at largest depth D

Huffman code – Optimality

Theorem 7.2 (Optimality of Huffman's Algorithm)

Given Σ 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 Σ. ◀

Proof sketch: by induction over $\sigma = |\Sigma|$



- 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 ℓ smaller, so still optimal)

Huffman code – Optimality

Theorem 7.2 (Optimality of Huffman's Algorithm)

Given Σ and $w : \Sigma \to \mathbb{R}_{\ge 0}$, 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 Σ.

Proof sketch: by induction over $\sigma = |\Sigma|$

- 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 ℓ smaller, so still optimal)
- any optimal code for $\Sigma' = \Sigma \setminus \{\mathtt{a}, \mathtt{b}\} \cup \{\boxed{\mathtt{ab}}\}$ yields optimal code for Σ by replacing leaf $\boxed{\mathtt{ab}}$ by internal node with children a and b.

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

7.4 Entropy

Entropy

Definition 7.3 (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

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

◀

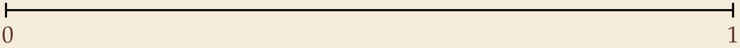
Entropy

Definition 7.3 (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.



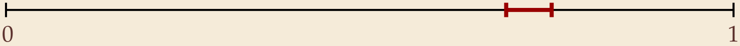
Entropy

Definition 7.3 (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.



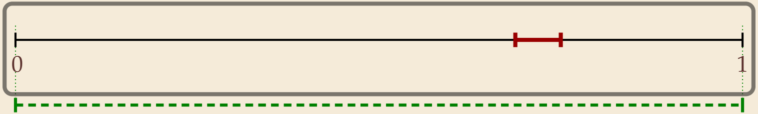
Entropy

Definition 7.3 (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.



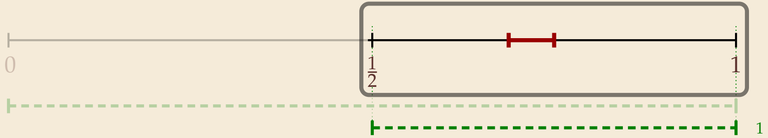
Entropy

Definition 7.3 (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.



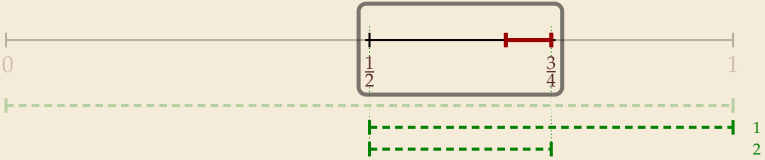
Entropy

Definition 7.3 (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.



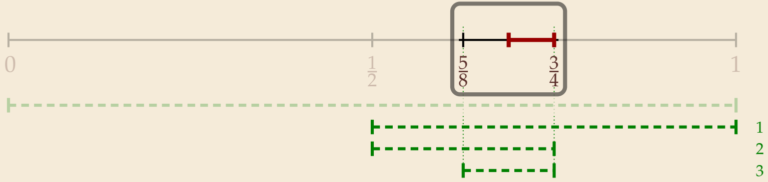
Entropy

Definition 7.3 (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.



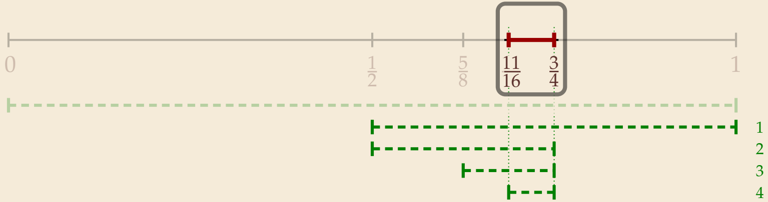
Entropy

Definition 7.3 (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.



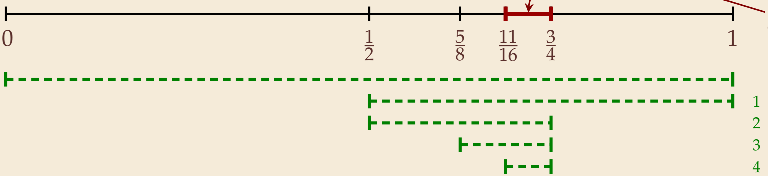
Entropy

Definition 7.3 (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.



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

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

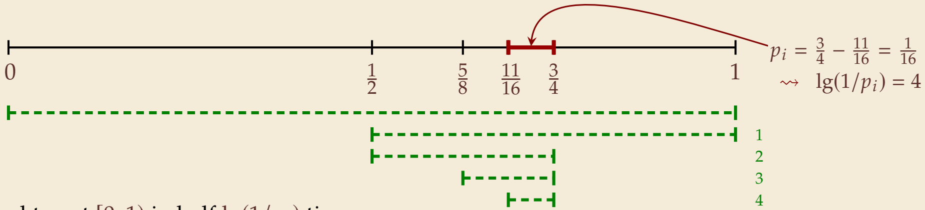
Entropy

Definition 7.3 (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.



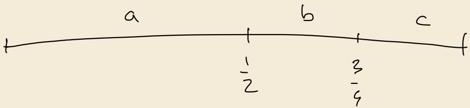
$\leadsto$ Need to cut $[0,1)$ in half $\lg(1/p_i)$ times

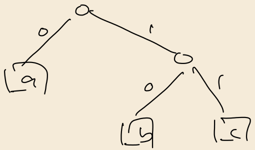
- more precisely: the expected number of bits (Yes/No questions) required to nail down the random value

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*!





$$|E(a)| = \lg\left(\frac{1}{P[a]}\right) = 1 \qquad P[a] = \tfrac{1}{2}$$

Entropy and Huffman codes

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

- would ideally encode value i using $\lg(1/p_i)$ bits
  not always possible; cannot use codeword of 1.5 bits ... but:

Theorem 7.4 (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 Σ with weights $p(a_i) = p_i$ satisfies $\boxed{\mathcal{H} \le \ell(E) \le \mathcal{H} + 1}$ where $\mathcal{H} = \mathcal{H}(p_1, \ldots, p_\sigma)$. ◀

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:

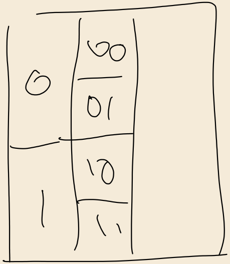
  not as length of single codeword that is; but can be possible *on average*!

Theorem 7.4 (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 Σ with weights $p(a_i) = p_i$ satisfies $\boxed{\mathcal{H} \le \ell(E) \le \mathcal{H} + 1}$ where $\mathcal{H} = \mathcal{H}(p_1, \ldots, p_\sigma)$. ◀

Proof sketch:

- $\ell(E) \ge \mathcal{H}$
  Prefix-free code E induces weights $q_i = 2^{-|E(a_i)|}$.
  By *Kraft's Inequality*, we have $q_1 + \cdots + q_\sigma \le 1$.



Entropy and Huffman codes

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

- would ideally encode value i using $\lg(1/p_i)$ bits
  not always possible; cannot use codeword of 1.5 bits . . . but:

Theorem 7.4 (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 Σ with weights $p(a_i) = p_i$ satisfies $\boxed{\mathcal{H} \le \ell(E) \le \mathcal{H} + 1}$ where $\mathcal{H} = \mathcal{H}(p_1, \ldots, p_\sigma)$. ◀

Proof sketch:

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

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

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 7.4 (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 Σ with weights $p(a_i) = p_i$ satisfies $\boxed{\mathcal{H} \le \ell(E) \le \mathcal{H} + 1}$ where $\mathcal{H} = \mathcal{H}(p_1, \ldots, p_\sigma)$.

Proof sketch:

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

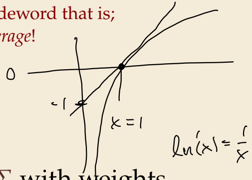
$$\mathcal{H} \;=\; \sum_{i=1}^{\sigma} p_i \lg\left(\frac{1}{p_i}\right) \;\le\; \sum_{i=1}^{\sigma} p_i \lg\left(\frac{1}{q_i}\right) \;=\; \ell(E).$$

Gibb's Inequality:
$\sum p_i = 1,\ \sum q_i \le 1,\ 0 \le p_i, q_i$
$\rightsquigarrow\ \sum p_i \ln\left(\frac{1}{p_i}\right) \le \sum p_i \ln\left(\frac{1}{q_i}\right)$

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 7.4 (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 Σ with weights $p(a_i) = p_i$ satisfies $\boxed{\mathcal{H} \le \ell(E) \le \mathcal{H} + 1}$ where $\mathcal{H} = \mathcal{H}(p_1, \ldots, p_\sigma)$. ◀

Proof sketch:

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

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

Gibb's Inequality:
$\sum p_i = 1,\ \sum q_i \le 1,\ 0 \le p_i, q_i$

$$\rightsquigarrow \sum p_i \ln\left(\frac{1}{p_i}\right) \le \sum p_i \ln\left(\frac{1}{q_i}\right)$$

Proof:
Note: $(*)\ \ln(x) \le x - 1 \quad (x \ge 0)$
(by concavity of ln)

$$\sum p_i \ln\left(\frac{1}{p_i}\right) - \sum p_i \ln\left(\frac{1}{q_i}\right)$$

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 7.4 (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 Σ with weights $p(a_i) = p_i$ satisfies $\boxed{\mathcal{H} \le \ell(E) \le \mathcal{H} + 1}$ where $\mathcal{H} = \mathcal{H}(p_1, \ldots, p_\sigma)$. ◀

Proof sketch:

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

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

Gibb's Inequality:
$\sum p_i = 1,\ \sum q_i \le 1,\ 0 \le p_i, q_i$

$$\rightsquigarrow \sum p_i \ln\left(\frac{1}{p_i}\right) \le \sum p_i \ln\left(\frac{1}{q_i}\right)$$

Proof:
Note: $(*)\ \ln(x) \le x - 1 \quad (x \ge 0)$
(by concavity of ln)

$$\sum p_i \ln\left(\frac{1}{p_i}\right) - \sum p_i \ln\left(\frac{1}{q_i}\right)$$

$$= \sum p_i \ln\left(\frac{q_i}{p_i}\right) \underset{(*)}{\le} \sum p_i\left(\frac{q_i}{p_i} - 1\right)$$

Entropy and Huffman codes

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

- would ideally encode value i using $\lg(1/p_i)$ bits
  not always possible; cannot use codeword of 1.5 bits ... but:

Theorem 7.4 (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 Σ with weights $p(a_i) = p_i$ satisfies $\boxed{\mathcal{H} \;\le\; \ell(E) \;\le\; \mathcal{H} + 1}$ where $\mathcal{H} = \mathcal{H}(p_1, \ldots, p_\sigma)$.

Proof sketch:

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

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

Gibb's Inequality:
$\sum p_i = 1,\ \sum q_i \le 1,\ 0 \le p_i, q_i$

$$\rightsquigarrow\ \sum p_i \ln\left(\frac{1}{p_i}\right) \;\le\; \sum p_i \ln\left(\frac{1}{q_i}\right)$$

Proof:
Note: $(*)\ \ln(x) \le x - 1 \quad (x \ge 0)$
(by concavity of ln)

$$\sum p_i \ln\left(\frac{1}{p_i}\right) - \sum p_i \ln\left(\frac{1}{q_i}\right)$$

$$= \sum p_i \ln\left(\frac{q_i}{p_i}\right) \underset{(*)}{\le} \sum p_i \left(\frac{q_i}{p_i} - 1\right)$$

$$= \sum q_i - \sum p_i \;\le\; 0$$

Entropy and Huffman codes [2]

Proof sketch (continued): Strategy: (1) Construct prefix-free code w/ $\ell(E') \leq \mathcal{H} + 1$

(2) $\ell(E) \leq \ell(E')$

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

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

Entropy and Huffman codes [2]

Proof sketch (continued):

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

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

  We construct a code E' for Σ 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 $\lg(1/q_i) \ge 1 = |E'(a_i)|$ ✓

Entropy and Huffman codes [2]

Proof sketch (continued):

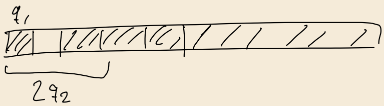
- $\ell(E) \leq \mathcal{H} + 1$

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

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

  - If $\sigma = 2$, E' uses a single bit each.
    Here, $q_i \leq 1/2$, so $\lg(1/q_i) \geq 1 = |E'(a_i)|$ ✓

  - If $\sigma \geq 3$, 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
    $q_2 + q_2 + \cdots + q_\sigma \leq 1$.



Entropy and Huffman codes [2]

Proof sketch (continued):

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

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

  We construct a code E' for Σ 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 $\left|E'(\boxed{a_1 a_2})\right| \le \lg\left(\frac{1}{2q_2}\right) = \lg\left(\frac{1}{q_2}\right) - 1.$

Entropy and Huffman codes [2]

Proof sketch (continued):

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

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

  We construct a code E' for Σ 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 $\displaystyle \ell(E) \le \ell(E') \le \sum_{i=1}^{\sigma} p_i \lg\left(\frac{1}{q_i}\right) \le \mathcal{H} + 1.$

Clicker Question

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

A always $\leq$

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

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

could be equal

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

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

→ sli.do/cs566

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/cs566`

Empirical Entropy

- Theorem 7.4 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 7.4 works for *any* character *probabilities* $p_1, \dots, p_\sigma$

  ... 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 $\Sigma = \{a_1, \dots, a_\sigma\}$,
  length of Huffman-coded text is

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

$\rightsquigarrow$ Theorem 7.4 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

- $\mathcal{H}_0(S) \;=\; \mathcal{H}\left(\dfrac{|S|_{a_1}}{n}, \dots, \dfrac{|S|_{a_\sigma}}{n}\right) \;=\; \displaystyle\sum_{i=1}^{\sigma} \frac{n}{|S|_{a_i}} \log_2\left(\frac{|S|_{a_i}}{n}\right)$ 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 [illegible]

👎 have to store code alongside with coded text

Part II

Compressing repetitive texts

Beyond Character Encoding

- Many "natural" texts show repetitive redundancy

```
All work and no play makes Jack a dull boy.  All work and no play makes Jack a dull
boy.  All work and no play makes Jack a dull boy.  All work and no play makes Jack
a dull boy.  All work and no play makes Jack a dull boy.  All work and no play makes
Jack a dull boy.  All work and no play makes Jack a dull boy.  All work and no play
makes Jack a dull boy.  All work and no play makes Jack a dull boy.  All work and no
play makes Jack a dull boy.  All work and no play makes Jack a dull boy.  All work and
no play makes Jack a dull boy.  All work and no play makes Jack a dull boy.  All work
and no play makes Jack a dull boy.  All work and no play makes Jack a dull boy.  All
work and no play makes Jack a dull boy.  All work and no play makes Jack a dull boy.
```

- character-by-character encoding will **not** capture such repetitions
  - ⇝ Huffman won't compression this very much

Beyond Character Encoding

- Many "natural" texts show repetitive redundancy

```
All work and no play makes Jack a dull boy.  All work and no play makes Jack a dull
boy.  All work and no play makes Jack a dull boy.  All work and no play makes Jack
a dull boy.  All work and no play makes Jack a dull boy.  All work and no play makes
Jack a dull boy.  All work and no play makes Jack a dull boy.  All work and no play
makes Jack a dull boy.  All work and no play makes Jack a dull boy.  All work and no
play makes Jack a dull boy.  All work and no play makes Jack a dull boy.  All work and
no play makes Jack a dull boy.  All work and no play makes Jack a dull boy.  All work
and no play makes Jack a dull boy.  All work and no play makes Jack a dull boy.  All
work and no play makes Jack a dull boy.  All work and no play makes Jack a dull boy.
```

- character-by-character encoding will **not** capture such repetitions
  - ⇝ Huffman won't compression this very much

⇝ Have to encode whole ***phrases*** of S by a single codeword

7.5 Run-Length Encoding

Run-Length encoding

- simplest form of repetition: ***runs*** of characters

  same character repeated

```
00000000000000000000000000000000000000000
00000000000000000000000000000000000000000
00000000000000000000000000000000000000000
00010110010000000001111100000000011111000
00111111111000000111111100000001111111000
00111101101000001110000000000011100000000
00110000000000011100000000000111000000000
00110000000000111000000000001110000000000
00110000000000111000000000001110000000000
00110110000000110011111000001100111110000
00111111110000111111111100001111111111000
00111011111000111100111110001111100111100
00000000011100111000000111001110000001110
00000000011100111000000110001110000001100
00000000001100011000000111000110000001110
00000000001100111000000110001110000001100
00000000011100011000000111000110000001110
00000000011000011100001110000111000011100
00110111111000001111111000000011111111000
01111111110000000111111000000001111110000
00010110000000000010010000000000100100000
00000000000000000000000000000000000000000
00000000000000000000000000000000000000000
```

- here: only consider $\Sigma_S = \{0,1\}$ (work on a binary representation)
  - can be extended for larger alphabets

Run-Length encoding

- simplest form of repetition: ***runs*** of characters

  same character repeated

```
00000000000000000000000000000000000000000
00000000000000000000000000000000000000000
00000000000000000000000000000000000000000
00010110010000000001111100000000011111000
00111111111000000111111100000001111111000
00111101101000001110000000000011100000000
00110000000000011100000000000111000000000
00110000000000111000000000001110000000000
00110000000000111000000000001110000000000
00110110000000110011111000001100111110000
00111111110000111111111100001111111111000
00111011111000111110011110001111100111100
00000000011100111000000111001110000001110
00000000011100111000000110001110000001100
00000000001100011000000111000110000001110
00000000001100111000000110001110000001100
00000000011100011000000111000110000001110
00000000011000011100001110000111000011100
00110111111000001111111100000001111111000
01111111111000000011111100000001111110000
00010110000000000010010000000000010010000
00000000000000000000000000000000000000000
00000000000000000000000000000000000000000
```

- here: only consider $\Sigma_S = \{0, 1\}$ (work on a binary representation)
  - can be extended for larger alphabets

$\rightsquigarrow$ **run-length encoding (RLE)**:
use runs as phrases: $S = \underbrace{\texttt{00000}}\ \underbrace{\texttt{111}}\ \underbrace{\texttt{0000}}$

Run-Length encoding

- simplest form of repetition: ***runs*** of characters

  same character repeated

```
0000000000000000000000000000000000000000
0000000000000000000000000000000000000000
0000000000000000000000000000000000000000
0001011001000000000111110000000001111000
0011111111100000011111110000000111111000
0011110110100000111000000000011100000000
0011000000000001110000000000111000000000
0011000000000011100000000001110000000000
0011000000000011100000000001110000000000
0011011000000011001111100001100111110000
0011111111100001111111110000111111111000
0011101111100011111001111100111110011100
0000000001110011100000011100111000001110
0000000001110011100000011000111000001100
0000000000110001100000011100011000001110
0000000000110011100000011000111000001100
0000000001110001100000011100011000001110
0000000001100001110000111000011100011100
0011011111100000111111100000011111111000
0111111111100000011111100000001111110000
0001011000000000000100100000000010010000
0000000000000000000000000000000000000000
0000000000000000000000000000000000000000
```

- here: only consider $\Sigma_S = \{0,1\}$ (work on a binary representation)
  - can be extended for larger alphabets

⇝ **run-length encoding (RLE):**
use runs as phrases: $S = \underbrace{\texttt{00000}}\ \underbrace{\texttt{111}}\ \underbrace{\texttt{0000}}$

⇝ We have to store
  - the first bit of S (either `0` or `1`)
  - the length of each subsequent run
  - Note: don't have to store bit for later runs since they must alternate.

- Example becomes: $0, 5, 3, 4$

Run-Length encoding

- simplest form of repetition: ***runs*** of characters
  same character repeated

```
0000000000000000000000000000000000000000
0000000000000000000000000000000000000000
0000000000000000000000000000000000000000
0001011001000000001111100000000011111000
0011111111100000011111100000001111111000
0011110110100000111000000000011100000000
0011000000000001110000000000111000000000
0011000000000011100000000001110000000000
0011000000000011100000000001110000000000
0011011000000110011111000001100111110000
0011111111100001111111100001111111111000
0011101111100011111001111000111110011100
0000000001110011100000011100111000001110
0000000001110011100000011000111000001100
0000000000110001100000011100011000001110
0000000000110011100000011000111000001100
0000000001110001100000011100011000001110
0000000001100001110000111000011100011100
0011011111000001111111100000011111111000
0111111111000000111111000000001111110000
0001011000000000010010000000000100100000
0000000000000000000000000000000000000000
0000000000000000000000000000000000000000
```

- here: only consider $\Sigma_S = \{0, 1\}$ (work on a binary representation)
  - can be extended for larger alphabets

⇝ **run-length encoding (RLE)**:
use runs as phrases: $S = \underbrace{\texttt{00000}}\ \underbrace{\texttt{111}}\ \underbrace{\texttt{0000}}$

⇝ We have to store
  - the first bit of S (either `0` or `1`)
  - the length of each subsequent run
  - Note: don't have to store bit for later runs since they must alternate.

- Example becomes: $0, 5, 3, 4$

- **Question**: How to encode a run length k in binary? (k can be arbitrarily large!)

Clicker Question

How would you encode a string that can we arbitrarily long?

$\boxed{\text{EOF}}$

→ sli.do/cs566

Elias codes

- Need a *prefix-free encoding* for $\mathbb{N} = \{1, 2, 3, \ldots, \}$
  - must allow arbitrarily large integers
  - must know when to stop reading

Elias codes

- Need a *prefix-free encoding* for $\mathbb{N} = \{1, 2, 3, \ldots, \}$
  - must allow arbitrarily large integers
  - must know when to stop reading
- But that's simple! Just use ***unary*** **encoding**!
  $7 \mapsto$ `00000001` $3 \mapsto$ `0001` $0 \mapsto$ `1` $30 \mapsto$ `0000000000000000000000000000001`

Elias codes

- Need a *prefix-free encoding* for $\mathbb{N} = \{1, 2, 3, \ldots, \}$
  - must allow arbitrarily large integers
  - must know when to stop reading

- But that's simple! Just use ***unary*** **encoding**!

  $7 \mapsto$ `00000001` $3 \mapsto$ `0001` $0 \mapsto$ `1` $30 \mapsto$ `0000000000000000000000000000001`

  Much too long
  - (wasn't the whole point of RLE to get rid of long runs??)

Elias codes

- Need a *prefix-free encoding* for $\mathbb{N} = \{1, 2, 3, \ldots, \}$
  - must allow arbitrarily large integers
  - must know when to stop reading
- But that's simple! Just use ***unary*** **encoding**!
  $7 \mapsto$ `00000001` $3 \mapsto$ `0001` $0 \mapsto$ `1` $30 \mapsto$ `0000000000000000000000000000001`

  Much too long
  - (wasn't the whole point of RLE to get rid of long runs??)
- Refinement: ***Elias gamma code***
  - Store the **length** ℓ of the binary representation in **unary**
  - Followed by the binary digits themselves

Elias codes

- Need a *prefix-free encoding* for $\mathbb{N} = \{1, 2, 3, \ldots, \}$
  - must allow arbitrarily large integers
  - must know when to stop reading
- But that's simple! Just use ***unary* encoding**!

  $7 \mapsto$ `00000001` $3 \mapsto$ `0001` $0 \mapsto$ `1` $30 \mapsto$ `0000000000000000000000000000001`

  Much too long
  - (wasn't the whole point of RLE to get rid of long runs??)
- Refinement: ***Elias gamma code***
  - Store the **length** ℓ of the binary representation in **unary**
  - Followed by the binary digits themselves
  - little tricks:
    - always have $\ell \geq 1$, so store $\ell - 1$ instead
    - binary representation always starts with `1` $\rightsquigarrow$ don't need terminating `1` in unary

$\rightsquigarrow$ Elias gamma code = $\ell - 1$ zeros, followed by binary representation

Examples: $1 \mapsto$ `1`, $3 \mapsto$ `011`, $5 \mapsto$ `00101`, $30 \mapsto$ `000011110`

Clicker Question

Decode the **first** number in Elias gamma code (at the beginning) of the following bitstream:

`00011011101110011 0`.

→ *sli.do/cs566*

Run-length encoding – Examples

- Encoding:
  S = 11111110010000000000000000000011111111111

  C = 1

- Decoding:
  C = 00001101001001010

  S =

Run-length encoding – Examples

- Encoding:

  S = `11111110010000000000000000000011111111111`

  $k = 7$

  C = `100111`

- Decoding:

  C = `00001101001001010`

  S =

Run-length encoding – Examples

- Encoding:

  S = `11111110010000000000000000000011111111111`

  $k = 2$

  C = `100111010`

- Decoding:

  C = `00001101001001010`

  S =

Run-length encoding – Examples

- Encoding:
  $S =$ `11111110010000000000000000000011111111111`
  $k = 1$
  $C =$ `1001110101`

- Decoding:
  $C =$ `00001101001001010`

  $S =$

Run-length encoding – Examples

- Encoding:
  $S =$ `1111111001`**`00000000000000000000`**`11111111111`
  $k = 20$
  $C =$ `1001110101`**`000010100`**

- Decoding:
  $C =$ `00001101001001010`

  $S =$

Run-length encoding – Examples

- Encoding:
  $S =$ `111111100100000000000000000000`11111111111
  $k = 11$
  $C =$ `1001110101000010100`0001011

- Decoding:
  $C =$ `00001101001001010`

  $S =$

Run-length encoding – Examples

- Encoding:
  S = `11111110010000000000000000000001111111111`

  C = `10011101010000101000001011`

  Compression ratio: $26/41 \approx 63\%$

- Decoding:
  C = `00001101001001010`

  S =

Run-length encoding – Examples

- Encoding:

  S = `111111100100000000000000000000011111111111`

  C = `10011101010000101000001011`

  Compression ratio: $26/41 \approx 63\%$

- Decoding:

  C = `00001101001001010`

  S =

Run-length encoding – Examples

- Encoding:
  S = `11111110010000000000000000000011111111111`

  C = `10011101010000101000001011`

  Compression ratio: $26/41 \approx 63\%$

- Decoding:
  C = `00001101001001010`
  $b = 0$

  S =

Run-length encoding – Examples

- Encoding:

  S = `11111110010000000000000000000011111111111`

  C = `10011101010000101000001011`

  Compression ratio: $26/41 \approx 63\%$

- Decoding:

  C = `0000110100100101` `0`

  $b = 0$

  $\ell = 3 + 1$

  S =

Run-length encoding – Examples

- Encoding:
  $S =$ `11111110010000000000000000000000001111111111`

  $C =$ `10011101010000101000001011`

  Compression ratio: $26/41 \approx 63\%$

- Decoding:
  $C =$ `00001101001001010`
  $b = 0$
  $\ell = 3 + 1$
  $k = 13$
  $S =$ `0000000000000`

Run-length encoding – Examples

- Encoding:
  S = `11111110010000000000000000000011111111111`

  C = `10011101010000101000001011`

  Compression ratio: $26/41 \approx 63\%$

- Decoding:
  C = `0000110100 1001010`
  $b = 1$
  $\ell = 2 + 1$
  $k =$
  S = `0000000000000`

Run-length encoding – Examples

- Encoding:
  $S =$ `11111110010000000000000000000011111111111`

  $C =$ `10011101010000101000001011`

  Compression ratio: $26/41 \approx 63\%$

- Decoding:
  $C =$ `00001101001001010`
  $b = 1$
  $\ell = 2 + 1$
  $k = 4$
  $S =$ `00000000000001111`

Run-length encoding – Examples

- Encoding:
  S = `11111110010000000000000000000011111111111`

  C = `10011101010000101000001011`

  Compression ratio: $26/41 \approx 63\%$

- Decoding:
  C = `00001101001001010`
  $b = 0$
  $\ell = 0 + 1$
  $k =$
  S = `00000000000001111`

Run-length encoding – Examples

- Encoding:

  S = `11111110010000000000000000000011111111111`

  C = `10011101010000101000001011`

  Compression ratio: $26/41 \approx 63\%$

- Decoding:

  C = `00001101001001010`

  $b = 0$

  $\ell = 0 + 1$

  $k = 1$

  S = `00000000000001111 0`

Run-length encoding – Examples

- Encoding:
  $S = \texttt{11111110010000000000000000000011111111111}$

  $C = \texttt{10011101010000101000001011}$

  Compression ratio: $26/41 \approx 63\%$

- Decoding:
  $C = \texttt{00001101001001010}$
  $b = 1$
  $\ell = 1 + 1$
  $k =$
  $S = \texttt{00000000000011110}$

Run-length encoding – Examples

- Encoding:

  S = `11111110010000000000000000000011111111111`

  C = `10011101010000101000001011`

  Compression ratio: $26/41 \approx 63\%$

- Decoding:

  C = `000011010010010`**`10`**

  $b = 1$

  $\ell = 1 + 1$

  $k = 2$

  S = `00000000000001111011`**`11`**

Run-length encoding – Discussion

- extensions to larger alphabets possible (must store next character then)
- used in some image formats (e. g. TIFF)

Run-length encoding – Discussion

- extensions to larger alphabets possible (must store next character then)
- used in some image formats (e. g. TIFF)

👍 fairly simple and fast

👍 can compress n bits to $\Theta(\log n)$!
for extreme case of constant number of runs

👎 negligible compression for many common types of data

- No compression until run lengths $k \geq 6$
- **expansion** for run length $k = 2$ or 6