7 Compression

21 March 2022

Sebastian Wild

COMP526 (Spring 2022)
University of Liverpool

version `2022-03-20 23:19`

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 7: ***Compression***

Outline

7 Compression

7.1 Context
7.2 Character Encodings
7.3 Huffman Codes
7.4 Entropy
7.5 Run-Length Encoding
7.6 Lempel-Ziv-Welch
7.7 Lempel-Ziv-Welch Decoding
7.8 Move-to-Front Transformation
7.9 Burrows-Wheeler Transform
7.10 Inverse BWT

7.1 Context

Overview

- Unit 4–6: How to *work* with strings
  - finding substrings
  - finding approximate matches
  - finding repeated parts
  - . . .
  - 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

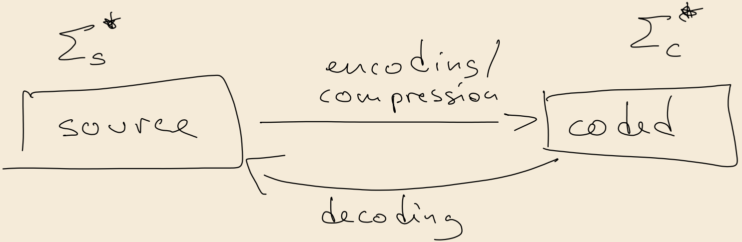
Clicker Question

What compression methods do you know?

`sli.do/comp526`

Terminology

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

- 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 coded text

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

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

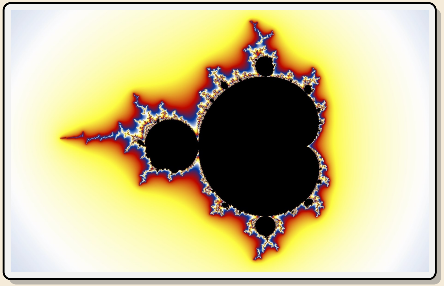
- We will measure the *compression ratio*: $\dfrac{|C| \cdot \lg |\Sigma_C|}{|S| \cdot \lg |\Sigma_S|} \overset{\Sigma_C=\{0,1\}}{=} \dfrac{|C|}{|S| \cdot \lg |\Sigma_S|}$

  size of source text

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

Limits of algorithmic compression

Is this image compressible?

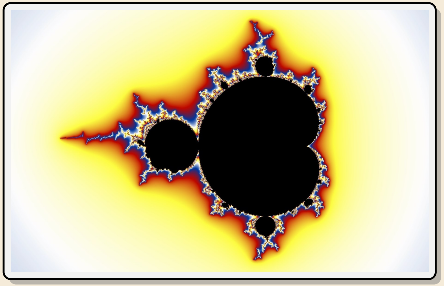


Limits of algorithmic compression

Is this image compressible?

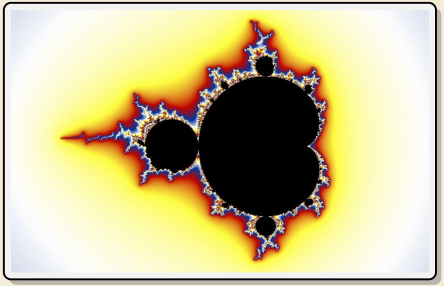
visualization of Mandelbrot set

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

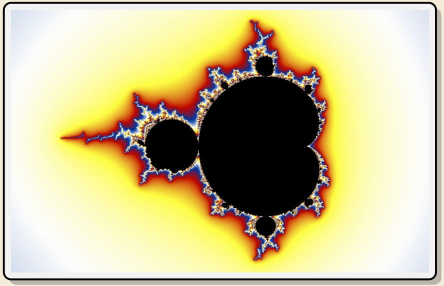
⇝ ***Kolmogorov complexity***

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

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

- C = *any program* that outputs S

  self-extracting archives!
- Kolmogorov complexity = length of smallest such program
- **Problem:** finding smallest such program is *uncomputable*.
- $\rightsquigarrow$ No optimal encoding algorithm is possible!
- $\rightsquigarrow$ must be inventive to get efficient methods

What makes data compressible?

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

  1. **uneven character frequencies**
     some characters occur more often than others $\longrightarrow$ Part I

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

What makes data compressible?

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

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

There is no such thing as a free lunch!

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

Part I

Exploiting character frequencies

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 $c \in \Sigma_C$
- **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)

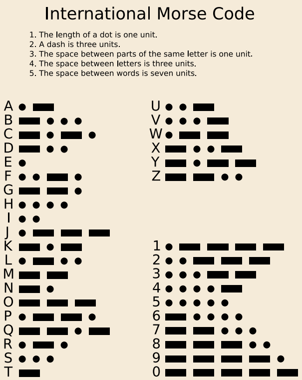
Fixed-length codes – Discussion

- 👍 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?)

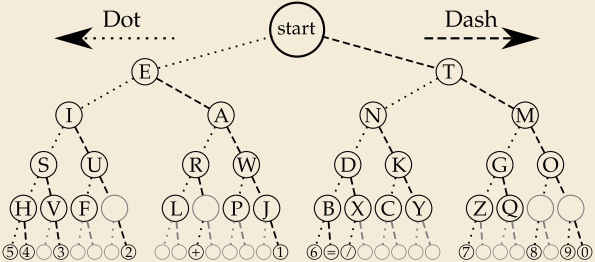
Variable-length codes

- 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/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 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 $S = \texttt{banana}$ 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 = $\underbrace{\texttt{110}}_{\texttt{b}}\underbrace{\texttt{0}}_{\texttt{a}}\underbrace{\texttt{10}}_{\texttt{n}}\underbrace{\texttt{0}}_{\texttt{a}}\underbrace{\texttt{10}}_{\texttt{n}}\underbrace{\texttt{0}}_{\texttt{a}}$

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

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?

100

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

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

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

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

Code tries

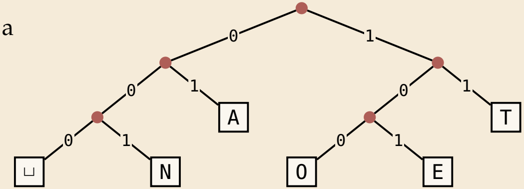
- From now on only consider prefix-free codes E:
  $E(c)$ is not a prefix of $E(c')$ for any $c, c' \in \Sigma_S$.

- **Example:**

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

Any prefix-free code corresponds to a ***(code) trie*** (trie of codewords) with characters of Σ_S at **leaves**.

no need for end-of-string symbols $ here
(already prefix-free!)



  - Encode `AN␣ANT` 01001000
  - Decode `1110000001010111`

    T O ␣

Code tries

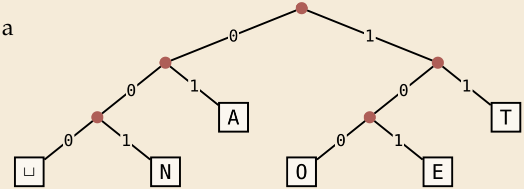
- From now on only consider prefix-free codes E:
  $E(c)$ is not a prefix of $E(c')$ for any $c, c' \in \Sigma_S$.

- **Example:**

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

Any prefix-free code corresponds to a ***(code) trie*** (trie of codewords) with characters of Σ_S at **leaves**.

no need for end-of-string symbols $ here (already prefix-free!)



- 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)

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 $\displaystyle\sum_{c \in \Sigma} w(c) \cdot |E(c)|$

(choose $w(c)$ = # occurrences of c in text

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 $\displaystyle\sum_{c \in \Sigma} w(c) \cdot |E(c)|$

- If we use $w(c) = \#\text{occurrences of } c \text{ in } S$,
  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.

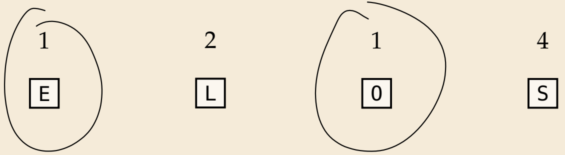
Huffman's algorithm:

1. Find two characters $\mathtt{a}$, $\mathtt{b}$ 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 $u = E(\boxed{\mathtt{ab}})$.

- 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

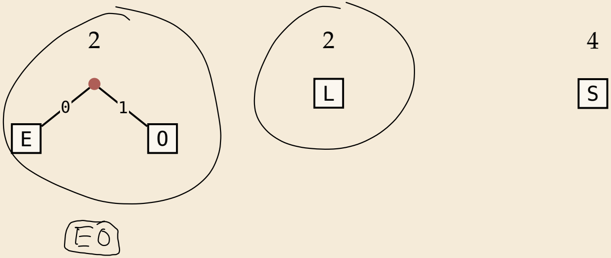
Huffman's algorithm – Example

- 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

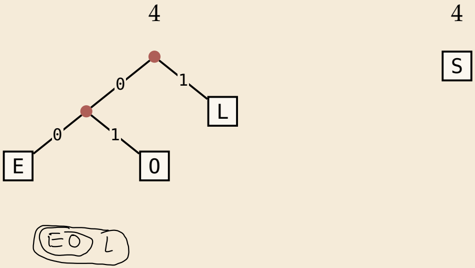
- Example text: $S = \texttt{LOSSLESS}$ $\rightsquigarrow$ $\Sigma_S = \{\texttt{E}, \texttt{L}, \texttt{O}, \texttt{S}\}$
- Character frequencies: $\texttt{E} : 1$, $\texttt{L} : 2$, $\texttt{O} : 1$, $\texttt{S} : 4$



Huffman's algorithm – Example

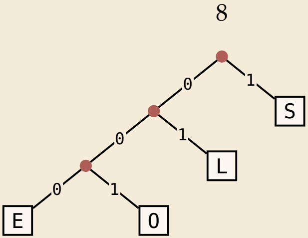
- Example text: S = LOSSLESS $\rightsquigarrow$ $\Sigma_S = \{$E, L, O, S$\}$

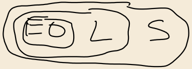
- Character frequencies: E : 1, L : 2, O : 1, S : 4



Huffman's algorithm – Example

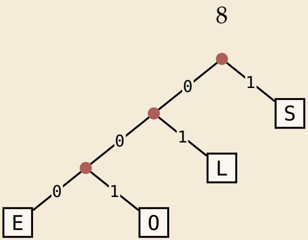
- Example text: $S = \texttt{LOSSLESS}$ $\rightsquigarrow$ $\Sigma_S = \{\texttt{E}, \texttt{L}, \texttt{O}, \texttt{S}\}$
- Character frequencies: $\texttt{E}: 1$, $\texttt{L}: 2$, $\texttt{O}: 1$, $\texttt{S}: 4$





Huffman's algorithm – Example

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

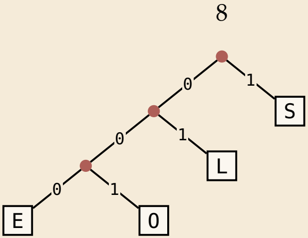


$\rightsquigarrow$ *Huffman tree* (code trie for Huffman code)

Huffman's algorithm – Example

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

c	E	L	O	S
$E(c)$	000	01	001	1



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

LOSSLESS $\longrightarrow$ 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 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.

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 7.1 (Optimality of Huffman's Algorithm)

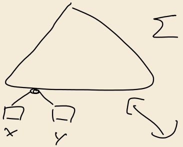
Given Σ and $w : \Sigma \to \mathbb{R}_{\geq 0}$, Huffman's Algorithm computes codewords $E : \Sigma \to \{0,1\}^\star$ with minimal expected codeword length $\ell(E) = \sum_{c\in\Sigma} w(c) \cdot |E(c)|$ among all prefix-free codes for Σ. ◀

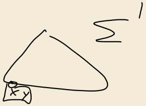
Huffman code – Optimality

Theorem 7.1 (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)
- any optimal code for $\Sigma' = \Sigma \setminus \{\text{a}, \text{b}\} \cup \{\boxed{\text{ab}}\}$ yields optimal code for Σ by replacing leaf $\boxed{\text{ab}}$ 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

$p_1 + p_2 + \cdots + p_n = 1$ $\qquad$ $p_i \in [0, 1]$

Definition 7.2 (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

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

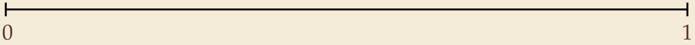
Entropy

Definition 7.2 (Entropy)

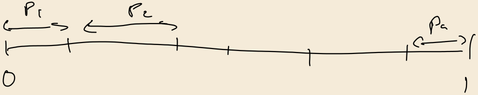
Given probabilities $p_1, \ldots, p_n$ (for outcomes $1, \ldots, n$ of a random variable), the *entropy* of the distribution is defined as

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

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



Entropy

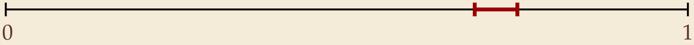


Definition 7.2 (Entropy)

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.



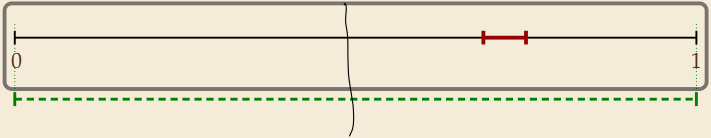
Entropy

Definition 7.2 (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.



left or right ?

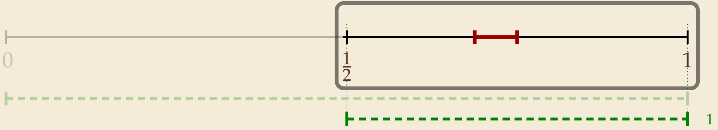
Entropy

Definition 7.2 (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.



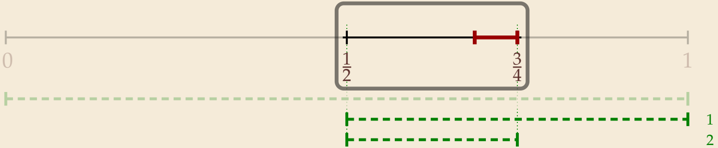
Entropy

Definition 7.2 (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

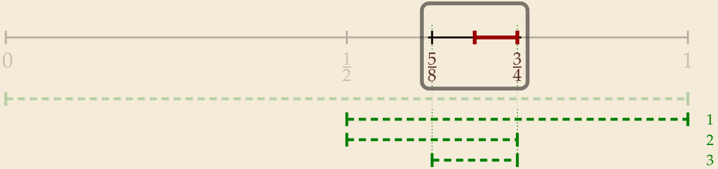
Definition 7.2 (Entropy)

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

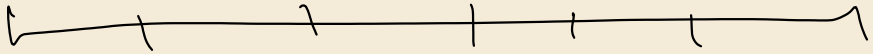
$$\mathcal{H}(p_1, \ldots, p_n) \;=\; -\sum_{i=1}^{n} p_i \lg p_i \;=\; \sum_{i=1}^{n} p_i \lg\left(\frac{1}{p_i}\right)$$

◀

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



Entropy

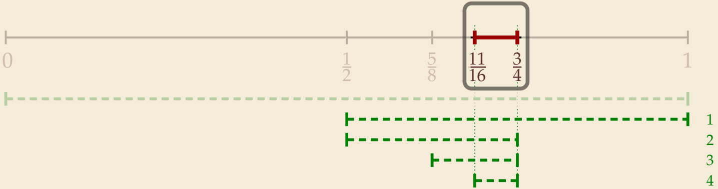


Definition 7.2 (Entropy)

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.



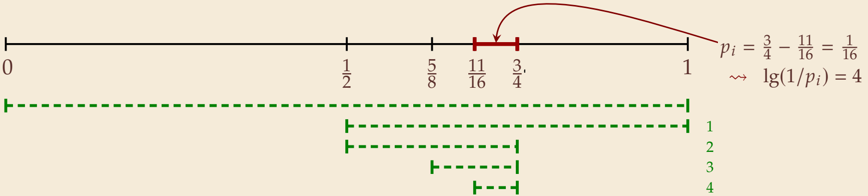
Entropy

Definition 7.2 (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.



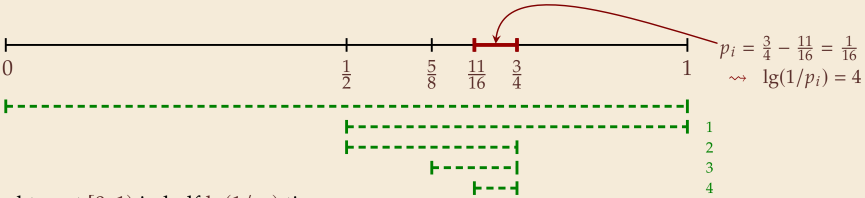
Entropy

Definition 7.2 (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 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 7.3 (Entropy bounds for Huffman codes)

For any $\Sigma = \{a_1, \ldots, a_\sigma\}$ and $w : \Sigma \to \mathbb{R}_{>0}$ and its Huffman code E, we have

$\boxed{\mathcal{H} \;\le\; \ell(E) \;\le\; \mathcal{H} + 1}$ where $\mathcal{H} = \mathcal{H}\left(\frac{w(a_1)}{W}, \ldots, \frac{w(a_\sigma)}{W}\right)$ and $W = w(a_1) + \cdots + w(a_\sigma)$.

$$\ell(E) = \sum_{i=1}^{\sigma} w(i) \cdot |E(a_i)|$$

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 7.3 (Entropy bounds for Huffman codes)

For any $\Sigma = \{a_1, \ldots, a_\sigma\}$ and $w : \Sigma \to \mathbb{R}_{>0}$ and its Huffman code E, we have

$$\boxed{\mathcal{H} \;\le\; \ell(E) \;\le\; \mathcal{H} + 1} \quad \text{where } \mathcal{H} = \mathcal{H}\left(\frac{w(a_1)}{W}, \ldots, \frac{w(a_\sigma)}{W}\right) \text{ and } W = w(a_1) + \cdots + w(a_\sigma).$$

Proof sketch:

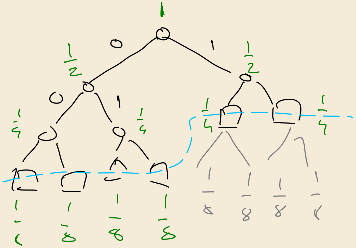
- $\ell(E) \ge \mathcal{H}$

  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

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



Entropy and Huffman codes [2]

Proof sketch (continued):

$\lceil x \rceil \leq x + 1$

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

  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 $q_1 \leq q_2 \leq \cdots \leq q_\sigma$

  $p_i < 1$

  $\lg\left(\frac{1}{p_i}\right) > \lg(1) = 0$

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

    $\lceil \lg(\frac{1}{p_i}) \rceil \geq 1$

  - 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 $q_2 + q_2 + \cdots + q_\sigma \leq 1$.

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

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

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

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 ✓ $\leq$

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

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

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

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

$\sum_{i=1}^{\sigma} p_i \lg\left(\frac{1}{p_i}\right) \qquad p_1 = p_2 = \cdots = p_\sigma$

$= \lg(\sigma) \qquad \Downarrow \quad p_i = \frac{1}{\sigma}$

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

$< \lg(\sigma) - \cancel{1} + 1$

sli.do/comp526

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 continue

13:45

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
  - $\leadsto$ 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
  - $\rightsquigarrow$ Huffman won't compression this very much

$\rightsquigarrow$ 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

```
000000000000000000000000000000000000000
000000000000000000000000000000000000000
000000000000000000000000000000000000000
000101100100000111111000000000011111000
001111111110001111111110000001111111000
001111011010001110001111000011100000000
001100000000000000000111000111000000000
001100000000000000000011001110000000000
001100000000000000000011001110000000000
001101100000000000000111001100111110000
001111111100000000000111001111111111000
001110111110000000001110001111110011100
000000000111000000011100001110000001110
000000000111000000011000001110000001100
000000000011000000110000000110000001110
000000000011000001110000001110000001100
000000000111000111000000000110000001110
000000000110000111000000000111000011100
001101111110001111011101000011111111000
011111111100011111111111000011111110000
000101100000001010011001000000100100000
000000000000000000000000000000000000000
000000000000000000000000000000000000000
```

- 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

```
000000000000000000000000000000000000000000
000000000000000000000000000000000000000000
000000000000000000000000000000000000000000
000101100100000111111000000000000111111000
001111111111000111111111100000001111111000
001111011010001110001111000011100000000000
001100000000000000000111000111000000000000
001100000000000000000011001110000000000000
001100000000000000000011001110000000000000
001101100000000000000011100110011111100000
001111111110000000000011100111111111111000
001110111110000000000111000111110011111100
000000000111000000000111000001110000001110
000000000111000000001100000111000000001100
000000000011000000011000000001100000001110
000000000011000000111000000011100000001100
000000000111000111000000000011000000001110
000000000110000111000000000001110000011100
001101111110001111011101000011111111111000
011111111110001111111111111000011111110000
000101100000000101001100100000000100100000
000000000000000000000000000000000000000000
000000000000000000000000000000000000000000
```

- 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
0001011001000000111111000000000011111000
0011111111110001111111110000001111111000
0011110110100011100011110000011100000000
0011000000000000000000111000111000000000
0011000000000000000000011001110000000000
0011000000000000000000011001110000000000
0011011000000000000000111001100111110000
0011111111000000000000111001111111111000
0011101111100000000001110001111100111100
0000000001110000000011100001110000001110
0000000001110000000011000001110000001100
0000000000110000000110000000110000001110
0000000000110000001110000001110000001100
0000000001110001110000000000110000001110
0000000001100001110000000000111000011100
0011011111100011110111010000111111111000
0111111111000111111111111000011111100000
0001011000000010100110010000000100100000
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 each each 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
0001011001000000111111000000000011111000
0011111111100011111111110000001111111000
0011110110100011100011110000111000000000
0011000000000000000001110001110000000000
0011000000000000000000110011100000000000
0011000000000000000000110011100000000000
0011011000000000000001110011001111110000
0011111111100000000001110011111111111000
0011101111110000000001110001111100111100
0000000001110000000011100001110000001110
0000000001110000000011000001110000001100
0000000000110000000110000000110000001110
0000000000110000001110000000111000001100
0000000001110001110000000000110000001110
0000000001100001110000000000111000011100
0011011111110001111011101000011111111000
0111111111100011111111111100001111110000
0001011000000001010011001000000100100000
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 each each 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?

`sli.do/comp526`

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 $\ell \geq 1$, 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:

`000110111011100110.`

sli.do/comp526

Run-length encoding – Examples

- Encoding:

  S = 11111110010000000000000000000011111111111

  C = 1

- Decoding:

  C = 00001101001001010

  S =

Run-length encoding – Examples

- Encoding:

  S = 1111111001000000000000000000000011111111111

  $k = 7$

  C = 100111

- Decoding:

  C = 0000110100100101 0

  S =

Run-length encoding – Examples

- Encoding:

  $S = $ `11111110010000000000000000000011111111111`

  $k = 2$

  $C = $ `100111010`

- Decoding:

  $C = $ `00001101001001010`

  $S = $

Run-length encoding – Examples

- Encoding:
  $S =$ `11111110011000000000000000000011111111111`
  $k = 1$
  $C =$ `1001110101`

- Decoding:
  $C =$ `00001101001001010`

  $S =$

Run-length encoding – Examples

- Encoding:

  S = `1111111001000000000000000000000011111111111`

  $k = 20$

  C = `1001110101000010100`

- 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:
  $S =$ `11111110010000000000000000000011111111111`

  $C =$ `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`

  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:

  $C =$ `0000110100 1001010`

  $b = 0$

  $\ell = 3 + 1$

  $S =$

Run-length encoding – Examples

- Encoding:

  $S = \texttt{11111110010000000000000000000011111111111}$

  $C = \texttt{10011101010000101000001011}$

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

- Decoding:

  $C = \texttt{0000}\textcolor{red}{\texttt{1101}}\texttt{001001010}$

  $b = 0$

  $\ell = 3 + 1$

  $k = 13$

  $S = \textcolor{red}{\texttt{0000000000000}}$

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 = `0000110100100`$\color{red}{\texttt{1}}$`010`

  $b = 0$

  $\ell = 0 + 1$

  $k =$

  S = `00000000000001111`

Run-length encoding – Examples

- Encoding:

  $S =$ `11111110010000000000000000000001111111111`

  $C =$ `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:

  $S =$ `11111110010000000000000000000011111111111`

  $C =$ `10011101010000101000001011`

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

- Decoding:

  $C =$ `00001101001001` `0` `10`

  $b = 1$

  $\ell = 1 + 1$

  $k =$

  $S =$ `000000000000011110`

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 = `000000000000011110` `11`

Run-length encoding – Discussion

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

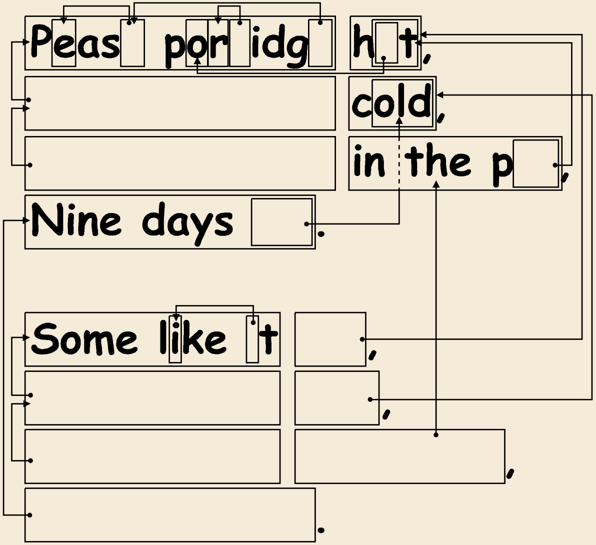
Run-length encoding – Discussion

- 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

7.6 Lempel-Ziv-Welch

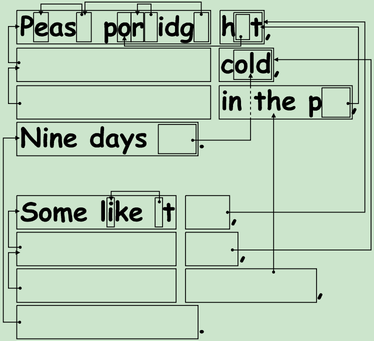
Warmup



https://classic.csunplugged.org/text-compression/

https://www.flickr.com/photos/quintanaroo/2742726346

Clicker Question



What is the **second-to-last line** of the poem to the left?

sli.do/comp526

Lempel-Ziv Compression

- Huffman and RLE mostly take advantage of frequent or repeated *single characters*.
- **Observation**: Certain *substrings* are much more frequent than others.
  - in English text: the, be, to, of, and, a, in, that, have, I
  - in HTML: "`<a href`", "`<img src`", "`<br/>`"

Lempel-Ziv Compression

- Huffman and RLE mostly take advantage of frequent or repeated *single characters*.
- **Observation**: Certain *substrings* are much more frequent than others.
  - in English text: the, be, to, of, and, a, in, that, have, I
  - in HTML: "`<a href`", "`<img src`", "`<br/>`"
- **Lempel-Ziv** stands for family of *adaptive* compression algorithms.
  - **Idea:** store repeated parts by reference!

  $\rightsquigarrow$ each codeword refers to
    - either a single character in Σ_S,
    - or a *substring* of S (that both encoder and decoder have already seen).

Lempel-Ziv Compression

- Huffman and RLE mostly take advantage of frequent or repeated *single characters*.
- **Observation**: Certain *substrings* are much more frequent than others.
  - in English text: the, be, to, of, and, a, in, that, have, I
  - in HTML: "`<a href`", "`<img src`", "`<br/>`"
- **Lempel-Ziv** stands for family of *adaptive* compression algorithms.
  - **Idea:** store repeated parts by reference!

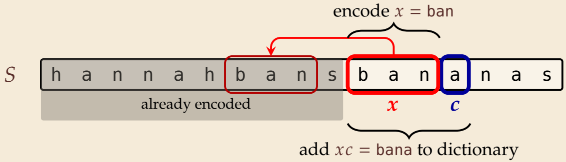
  $\rightsquigarrow$ each codeword refers to
    - either a single character in Σ_S,
    - or a *substring* of S (that both encoder and decoder have already seen).
  - Variants of Lempel-Ziv compression
    - "LZ77" Original version ("sliding window")
      Derivatives: LZSS, LZFG, LZRW, LZP, DEFLATE, . . .
      DEFLATE used in `(pk)zip`, `gzip`, `PNG`
    - "LZ78" Second (slightly improved) version
      Derivatives: LZW, LZMW, LZAP, LZY, . . .
      LZW used in `compress`, `GIF`

Lempel-Ziv-Welch

- here: ***Lempel-Ziv-Welch (LZW)*** (arguably the "cleanest" variant of Lempel-Ziv)
- ***variable-to-fixed* encoding**
  - all codewords have k bits (typical: $k = 12$) $\leadsto$ fixed-length
  - but they represent a variable portion of the source text!

Lempel-Ziv-Welch

- here: ***Lempel-Ziv-Welch (LZW)*** (arguably the "cleanest" variant of Lempel-Ziv)
- ***variable-to-fixed* encoding**
  - all codewords have k bits (typical: $k = 12$) $\rightsquigarrow$ fixed-length
  - but they represent a variable portion of the source text!
- maintain a **dictionary** D with 2^k entries $\rightsquigarrow$ codewords = indices in dictionary
  - initially, first $|\Sigma_S|$ entries encode single characters (rest is empty)
  - **add** a new entry to D **after each step**:
  - **Encoding:** after encoding a substring x of S,
    add xc to D where c is the character that follows x in S.



$\rightsquigarrow$ new codeword in D

- D actually stores codewords for x and c, not the expanded string

LZW encoding – Example

Input: `YO!␣YOU!␣YOUR␣YOYO!`

Σ_S = ASCII character set (0–127)

$C =$

$D =$

Code	String
...	
32	␣
33	!
...	
79	O
...	
82	R
...	
85	U
...	
89	Y
...	

Code	String
128	
129	
130	
131	
132	
133	
134	
135	
136	
137	
138	
139	

encode x = ban

S h a n n a h b a n s b a n a n a s

already encoded

x c

add xc = bana to dictionary

LZW encoding – Example

Input: Y0!␣Y0U!␣Y0UR␣Y0Y0!

Σ_S = ASCII character set (0–127)

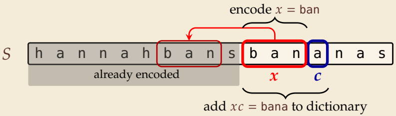
Y

C = 89

D =

Code	String
...	
32	␣
33	!
...	
79	0
...	
82	R
...	
85	U
...	
89	Y
...	

Code	String
128	
129	
130	
131	
132	
133	
134	
135	
136	
137	
138	
139	



LZW encoding – Example

Input: YO!␣YOU!␣YOUR␣YOYO!

Σ_S = ASCII character set (0–127)

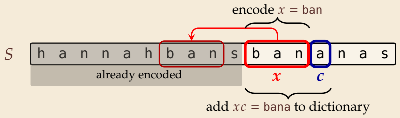
Y

C = 89

D =

Code	String
...	
32	␣
33	!
...	
79	O
...	
82	R
...	
85	U
...	
89	Y
...	

Code	String
128	YO
129	
130	
131	
132	
133	
134	
135	
136	
137	
138	
139	



LZW encoding – Example

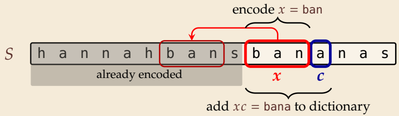
Input: Y0!␣YOU!␣YOUR␣YOYO! Σ_S = ASCII character set (0–127)

	Y	0
$C =$	89	79

$D =$

Code	String
...	
32	␣
33	!
...	
79	0
...	
82	R
...	
85	U
...	
89	Y
...	

Code	String
128	Y0
129	
130	
131	
132	
133	
134	
135	
136	
137	
138	
139	



LZW encoding – Example

Input: YO!␣YOU!␣YOUR␣YOYO!

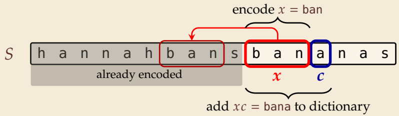
Σ_S = ASCII character set (0–127)

	Y	O
$C =$	89	79

$D =$

Code	String
. . .	
32	␣
33	!
. . .	
79	O
. . .	
82	R
. . .	
85	U
. . .	
89	Y
. . .	

Code	String
128	YO
129	O!
130	
131	
132	
133	
134	
135	
136	
137	
138	
139	



LZW encoding – Example

Input: `YO!␣YOU!␣YOUR␣YOYO!`

Σ_S = ASCII character set (0–127)

	Y	O	!
$C =$	89	79	33

$D =$

Code	String
...	
32	␣
33	!
...	
79	O
...	
82	R
...	
85	U
...	
89	Y
...	

Code	String
128	YO
129	O!
130	
131	
132	
133	
134	
135	
136	
137	
138	
139	

encode x = `ban`

S `h a n n a h b a n s b a n a n a s`

already encoded

x c

add xc = `bana` to dictionary

LZW encoding – Example

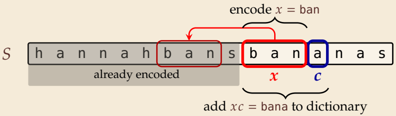
Input: `YO!␣YOU!␣YOUR␣YOYO!` $\qquad \Sigma_S$ = ASCII character set (0–127)

	Y	O	!
$C =$	89	79	33

$D =$

Code	String
...	
32	␣
33	!
...	
79	O
...	
82	R
...	
85	U
...	
89	Y
...	

Code	String
128	YO
129	O!
130	!␣
131	
132	
133	
134	
135	
136	
137	
138	
139	



LZW encoding – Example

Input: `YO!␣YOU!␣YOUR␣YOYO!`

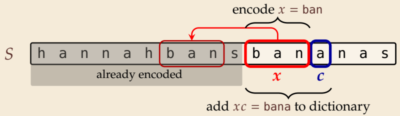
Σ_S = ASCII character set (0–127)

	Y	O	!	␣
$C =$	89	79	33	32

$D =$

Code	String
. . .	
32	␣
33	!
. . .	
79	O
. . .	
82	R
. . .	
85	U
. . .	
89	Y
. . .	

Code	String
128	YO
129	O!
130	!␣
131	
132	
133	
134	
135	
136	
137	
138	
139	



LZW encoding – Example

Input: `YO!␣YOU!␣YOUR␣YOYO!`

Σ_S = ASCII character set (0–127)

	Y	O	!	␣
$C =$	89	79	33	32

$D =$

Code	String
...	
32	␣
33	!
...	
79	O
...	
82	R
...	
85	U
...	
89	Y
...	

Code	String
128	YO
129	O!
130	!␣
131	␣Y
132	
133	
134	
135	
136	
137	
138	
139	

encode x = ban

S h a n n a h b a n s b a n a n a s

already encoded x c

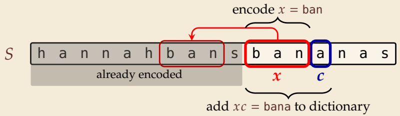
add xc = bana to dictionary

LZW encoding – Example

Input: `Y0!␣Y0U!␣Y0UR␣Y0Y0!`

Σ_S = ASCII character set (0–127)

	Y	0	!	␣	Y0
$C =$	89	79	33	32	128



$D =$

Code	String
. . .	
32	␣
33	!
. . .	
79	0
. . .	
82	R
. . .	
85	U
. . .	
89	Y
. . .	

Code	String
128	Y0
129	0!
130	!␣
131	␣Y
132	
133	
134	
135	
136	
137	
138	
139	

LZW encoding – Example

Input: `YO!␣YOU!␣YOUR␣YOYO!` Σ_S = ASCII character set (0–127)

	Y	O	!	␣	YO
$C =$	89	79	33	32	128

$D =$

Code	String
...	
32	␣
33	!
...	
79	O
...	
82	R
...	
85	U
...	
89	Y
...	

Code	String
128	YO
129	O!
130	!␣
131	␣Y
132	YOU
133	
134	
135	
136	
137	
138	
139	

encode x = ban

S `h a n n a h b a n s b a n a n a s`

already encoded

x c

add xc = bana to dictionary

LZW encoding – Example

Input: `YO!␣YOU!␣YOUR␣YOYO!`

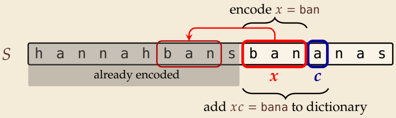
Σ_S = ASCII character set (0–127)

	Y	O	!	␣	YO	U
$C =$	89	79	33	32	128	85

$D =$

Code	String
...	
32	␣
33	!
...	
79	O
...	
82	R
...	
85	U
...	
89	Y
...	

Code	String
128	YO
129	O!
130	!␣
131	␣Y
132	YOU
133	
134	
135	
136	
137	
138	
139	



LZW encoding – Example

Input: `YO!␣YOU!␣YOUR␣YOYO!`

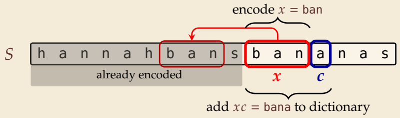
Σ_S = ASCII character set (0–127)

	Y	O	!	␣	YO	U
$C =$	89	79	33	32	128	85

$D =$

Code	String
...	
32	␣
33	!
...	
79	O
...	
82	R
...	
85	U
...	
89	Y
...	

Code	String
128	YO
129	O!
130	!␣
131	␣Y
132	YOU
133	U!
134	
135	
136	
137	
138	
139	



LZW encoding – Example

Input: `YO!␣YOU!␣YOUR␣YOYO!`

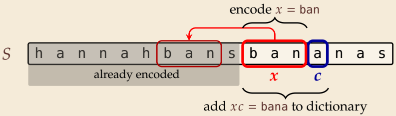
Σ_S = ASCII character set (0–127)

	Y	O	!	␣	YO	U	!␣
$C =$	89	79	33	32	128	85	130

$D =$

Code	String
...	
32	␣
33	!
...	
79	O
...	
82	R
...	
85	U
...	
89	Y
...	

Code	String
128	YO
129	O!
130	!␣
131	␣Y
132	YOU
133	U!
134	
135	
136	
137	
138	
139	



LZW encoding – Example

Input: `YO!␣YOU!␣YOUR␣YOYO!`

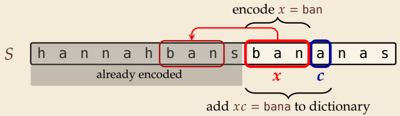
Σ_S = ASCII character set (0–127)

	Y	O	!	␣	YO	U	!␣
$C =$	89	79	33	32	128	85	130

$D =$

Code	String
...	
32	␣
33	!
...	
79	O
...	
82	R
...	
85	U
...	
89	Y
...	

Code	String
128	YO
129	O!
130	!␣
131	␣Y
132	YOU
133	U!
134	!␣Y
135	
136	
137	
138	
139	



LZW encoding – Example

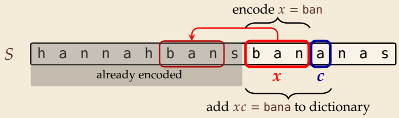
Input: `YO!␣YOU!␣YOUR␣YOYO!` Σ_S = ASCII character set (0–127)

	Y	O	!	␣	YO	U	!␣	YOU
C =	89	79	33	32	128	85	130	132

D =

Code	String
...	
32	␣
33	!
...	
79	O
...	
82	R
...	
85	U
...	
89	Y
...	

Code	String
128	YO
129	O!
130	!␣
131	␣Y
132	YOU
133	U!
134	!␣Y
135	
136	
137	
138	
139	



LZW encoding – Example

Input: `YO!␣YOU!␣YOUR␣YOYO!`

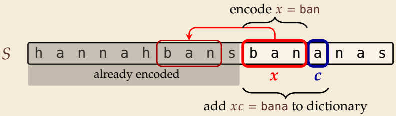
Σ_S = ASCII character set (0–127)

	Y	O	!	␣	YO	U	!␣	YOU
$C =$	89	79	33	32	128	85	130	132

$D =$

Code	String
...	
32	␣
33	!
...	
79	O
...	
82	R
...	
85	U
...	
89	Y
...	

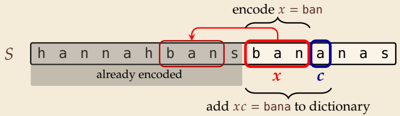
Code	String
128	YO
129	O!
130	!␣
131	␣Y
132	YOU
133	U!
134	!␣Y
135	YOUR
136	
137	
138	
139	



LZW encoding – Example

Input: `YO!␣YOU!␣YOUR␣YOYO!` $\qquad \Sigma_S$ = ASCII character set (0–127)

	Y	O	!	␣	YO	U	!␣	YOU	R
C =	89	79	33	32	128	85	130	132	82



D =

Code	String
...	
32	␣
33	!
...	
79	O
...	
82	R
...	
85	U
...	
89	Y
...	

Code	String
128	YO
129	O!
130	!␣
131	␣Y
132	YOU
133	U!
134	!␣Y
135	YOUR
136	
137	
138	
139	

LZW encoding – Example

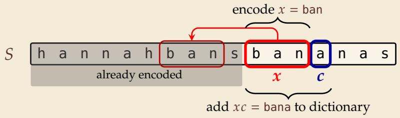
Input: `YO!␣YOU!␣YOUR␣YOYO!` $\quad\quad \Sigma_S$ = ASCII character set (0–127)

	Y	O	!	␣	YO	U	!␣	YOU	R
$C =$	89	79	33	32	128	85	130	132	82

$D =$

Code	String
...	
32	␣
33	!
...	
79	O
...	
82	R
...	
85	U
...	
89	Y
...	

Code	String
128	YO
129	O!
130	!␣
131	␣Y
132	YOU
133	U!
134	!␣Y
135	YOUR
136	R␣
137	
138	
139	



LZW encoding – Example

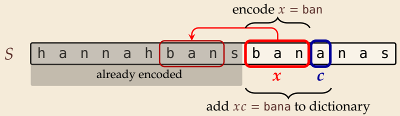
Input: `YO!␣YOU!␣YOUR␣YOYO!` Σ_S = ASCII character set (0–127)

	Y	O	!	␣	YO	U	!␣	YOU	R	␣Y
$C =$	89	79	33	32	128	85	130	132	82	131

$D =$

Code	String
...	
32	␣
33	!
...	
79	O
...	
82	R
...	
85	U
...	
89	Y
...	

Code	String
128	YO
129	O!
130	!␣
131	␣Y
132	YOU
133	U!
134	!␣Y
135	YOUR
136	R␣
137	
138	
139	



LZW encoding – Example

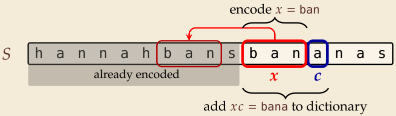
Input: `YO!␣YOU!␣YOUR␣YOYO!` $\qquad \Sigma_S$ = ASCII character set (0–127)

	Y	O	!	␣	YO	U	!␣	YOU	R	␣Y
$C =$	89	79	33	32	128	85	130	132	82	131

$D =$

Code	String
...	
32	␣
33	!
...	
79	O
...	
82	R
...	
85	U
...	
89	Y
...	

Code	String
128	YO
129	O!
130	!␣
131	␣Y
132	YOU
133	U!
134	!␣Y
135	YOUR
136	R␣
137	␣YO
138	
139	



LZW encoding – Example

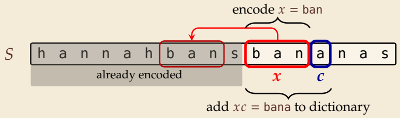
Input: `YO!␣YOU!␣YOUR␣YOYO!` Σ_S = ASCII character set (0–127)

	Y	O	!	␣	YO	U	!␣	YOU	R	␣Y	O
$C =$	89	79	33	32	128	85	130	132	82	131	79

$D =$

Code	String
...	
32	␣
33	!
...	
79	O
...	
82	R
...	
85	U
...	
89	Y
...	

Code	String
128	YO
129	O!
130	!␣
131	␣Y
132	YOU
133	U!
134	!␣Y
135	YOUR
136	R␣
137	␣YO
138	
139	



LZW encoding – Example

Input: `YO!␣YOU!␣YOUR␣YOYO!` $\quad \Sigma_S$ = ASCII character set (0–127)

	Y	O	!	␣	YO	U	!␣	YOU	R	␣Y	O
$C =$	89	79	33	32	128	85	130	132	82	131	79

$D =$

Code	String
...	
32	␣
33	!
...	
79	O
...	
82	R
...	
85	U
...	
89	Y
...	

Code	String
128	YO
129	O!
130	!␣
131	␣Y
132	YOU
133	U!
134	!␣Y
135	YOUR
136	R␣
137	␣YO
138	OY
139	

encode x = ban

S `h a n n a h b a n s b a n a n a s`

already encoded $\quad x \quad c$

add xc = bana to dictionary

LZW encoding – Example

Input: `YO!␣YOU!␣YOUR␣YOYO!` $\qquad \Sigma_S$ = ASCII character set (0–127)

	Y	O	!	␣	YO	U	!␣	YOU	R	␣Y	O	YO
$C =$	89	79	33	32	128	85	130	132	82	131	79	128

$D =$

Code	String
...	
32	␣
33	!
...	
79	O
...	
82	R
...	
85	U
...	
89	Y
...	

Code	String
128	YO
129	O!
130	!␣
131	␣Y
132	YOU
133	U!
134	!␣Y
135	YOUR
136	R␣
137	␣YO
138	OY
139	

encode x = ban

S `h a n n a h b a n s b a n a n a s`

already encoded $\quad x \quad c$

add xc = bana to dictionary

LZW encoding – Example

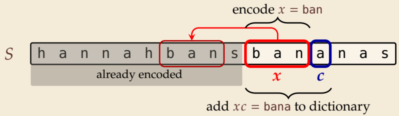
Input: `YO!␣YOU!␣YOUR␣YOYO!` $\qquad \Sigma_S =$ ASCII character set (0–127)

	Y	O	!	␣	YO	U	!␣	YOU	R	␣Y	O	YO
$C =$	89	79	33	32	128	85	130	132	82	131	79	128

$D =$

Code	String
...	
32	␣
33	!
...	
79	O
...	
82	R
...	
85	U
...	
89	Y
...	

Code	String
128	YO
129	O!
130	!␣
131	␣Y
132	YOU
133	U!
134	!␣Y
135	YOUR
136	R␣
137	␣YO
138	OY
139	YO!



LZW encoding – Example

Input: `YO!␣YOU!␣YOUR␣YOYO!` $\quad \Sigma_S$ = ASCII character set (0–127)

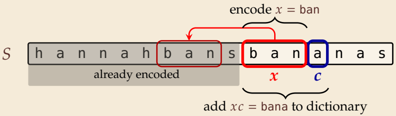
	Y	O	!	␣	YO	U	!␣	YOU	R	␣Y	O	YO	!
$C =$	89	79	33	32	128	85	130	132	82	131	79	128	33

store dictionary as trie

$D =$

Code	String
...	
32	␣
33	!
...	
79	O
...	
82	R
...	
85	U
...	
89	Y
...	

Code	String
128	YO
129	O!
130	!␣
131	␣Y
132	YOU
133	U!
134	!␣Y
135	YOUR
136	R␣
137	␣YO
138	OY
139	YO!



LZW encoding – Code

```
procedure LZWencode(S[0..n))
    x := ε // previous phrase, initially empty
    C := ε // output, initially empty
    D := dictionary, initialized with codes for c ∈ Σ_S // stored as trie
    k := |Σ_S| // next free codeword
    for i := 0, . . . , n − 1 do
        c := S[i]
        if D.containsKey(xc) then
            x := xc
        else
            C := C · D.get(x) // append codeword for x
            D.put(xc, k) // add xc to D, assigning next free codeword
            k := k + 1;  x := c
    end for
    C := C · D.get(x)
    return C
```