

ALGORITHMS\$EFFICIENT  
CIENTALGORITHMS\$EFFI  
**E F F I C I E N T** ALGORITHMS\$  
ENTALGORITHMS\$EFFICI  
FFICIENT**ALGORITHM**\$E  
FICIENTALGORITHMS\$EF  
GORITHMS\$EFFICIENTAL  
HMS\$EFFICIENTALGORIT

# 7

## Text Compression

24 November 2025

Prof. Dr. Sebastian Wild

# 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
- 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 6 & 13: How to *work* with strings
  - ▶ finding substrings
  - ▶ finding approximate matches ~ Unit 13
  - ▶ finding repeated parts ~ 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 ~ Unit 8

## Clicker Question



What compression methods do you know?



→ *[sli.do/cs566](https://sli.do/cs566)*

# Terminology

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

- ▶ **Lossy vs. Lossless**

- ▶ **lossy compression** can only decode **approximately**;  
the exact source text  $S$  is lost
- ▶ **lossless compression** always decodes  $S$  exactly

for human perception  
 $S \approx S'$

- ▶ 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*:
$$\frac{|C| \cdot \lg |\Sigma_C|}{|S| \cdot \lg |\Sigma_S|} \stackrel{\Sigma_C=\{0,1\}}{=} \frac{|C|}{|S| \cdot \lg |\Sigma_S|}$$
  - < 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?

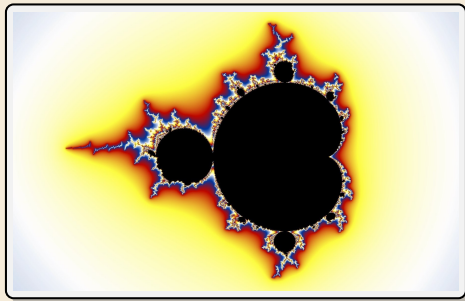
- A** Sure, I could explain what it is.
- B** Heard that in a lecture, but don't quite remember
- C** No, never heard of it



→ *[sli.do/cs566](https://sli.do/cs566)*

# Limits of algorithmic compression

*Is this image compressible?*

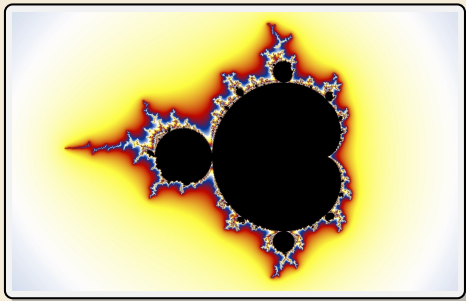


# Limits of algorithmic compression

*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!

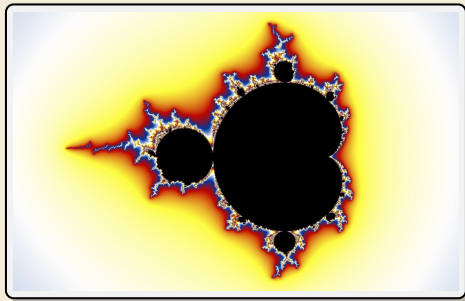


# Limits of algorithmic compression

*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

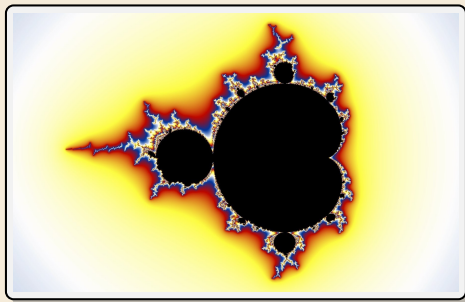
- ▶ Kolmogorov complexity = length of smallest such program

# 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:**

Assume otherwise, i. e.,  $K(S)$  computes Kolmogorov complexity of strings  $S$ . *length of shortest program for S*

↪  $K$  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  $S$ .

↪  $K$  has some length  $|K|$ .

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

---

```
1 procedure findComplexString():
2   for  $n := 1, 2, \dots$ :
3     for  $S \in \Sigma^n$ :
4       if  $K(S) > |K| + 1000$ 
5         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  $S$ .

↪  $K$  has some length  $|K|$ .

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

---

```
1 procedure findComplexString():  
2   for  $n := 1, 2, \dots$ :  
3     for  $S \in \Sigma^n$ :  
4       if  $K(S) > |K| + 1000$   
5         return  $S$ 
```

---

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

# 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

# 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  $E : \Sigma_S \rightarrow \Sigma_C^*$

- ▶ 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  $c \in \Sigma_S$
- ▶ **variable-length code:** not all codewords of same length

# Fixed-length codes

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

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

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

## Fixed-length codes – Discussion



Encoding & Decoding as fast as it gets



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



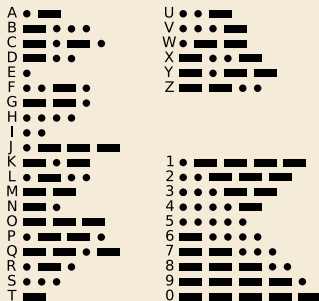
inflexible (how to support adding a new character?)

# Variable-length codes

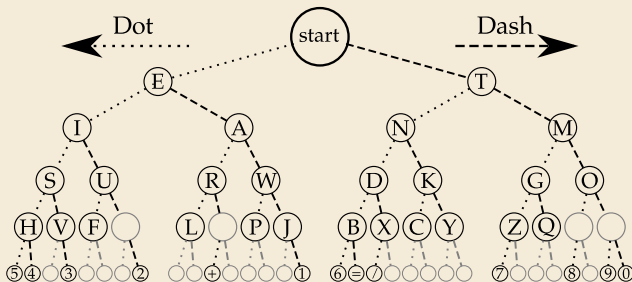
- ▶ to gain more flexibility, have to allow different lengths for codewords
- ▶ actually an old idea: **Morse Code**

## International Morse Code

1. The length of a dot is one unit,
2. A dash is three units,
3. The space between parts of the same letter is one unit,
4. The space between letters is three units,
5. The space between words is seven units,



[https://commons.wikimedia.org/wiki/File:International\\_Morse\\_Code.svg](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 1s 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  $S = \text{banana}$  with the coded text  $C = \underline{1100}\underline{100}\underline{100}$ 

b
a
n
a
n
a



## Pitfall in variable-length codes

- Suppose we have the following code:
 

$c$	a	n	b	s
$E(c)$	0	10	110	100
- Happily encode text  $S = \text{banana}$  with the coded text  $C = \underline{1100}\underline{100}\underline{100}$ 

b
a
n
a
n
a

⚡ C = 1100100100 decodes **both** to banana and to bass:  $\frac{1100100100}{\text{b a s s}}$

↗ 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  $S = \text{banana}$  with the coded text  $C = \underline{1100100100}$   

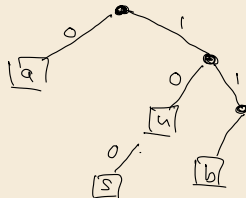
b
a
n
a
n
a

⚡  $C = 1100100100$  decodes **both** to banana and to bass:  $\underline{1100100100}$   

b
a
s
s

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

but how should we have known?



$S \hat{=} na$



$E(n) = 10$  is a (proper) **prefix** of  $E(s) = 100$

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

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

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

# Code tries

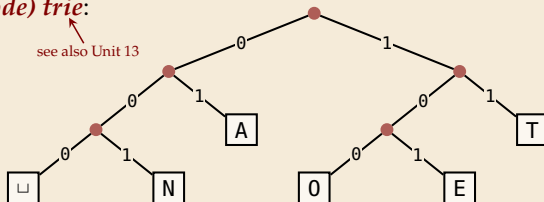
- From now on only consider prefix-free codes  $E$ :  
 $E(c)$  is not a proper prefix of  $E(c')$  for any  $c, c' \in \Sigma_S$ .

► **Example:**

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

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

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



- Example for using the code trie:

► Encode AN $\sqcup$ ANT      01 001 000

► Decode 111000001010111

T O  $\sqcup$

# Code tries

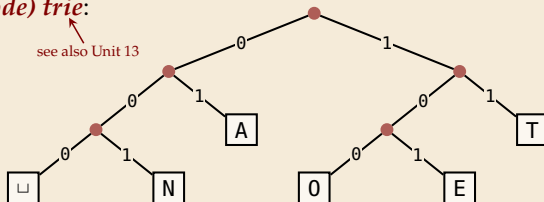
- ▶ From now on only consider prefix-free codes  $E$ :  
 $E(c)$  is not a proper prefix of  $E(c')$  for any  $c, c' \in \Sigma_S$ .

▶ **Example:**

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

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

- ▶ binary tree
- ▶ one **leaf** for each characters of  $\Sigma_S$
- ▶ path from root to leaf = codeword  
left child = 0; right child = 1



- ▶ Example for using the code trie:
  - ▶ Encode  $AN\sqcup ANT \rightarrow 010010000100111$
  - ▶ Decode  $111000001010111 \rightarrow T0\sqcup EAT$

# The Codeword Supermarket

0	00	000	0000	000000
			0001	000001
		001	0010	000100
			0011	000101
			0100	001000
			0101	001001
	01	010	0110	001100
			0111	001101
		011	1000	010000
			1001	010001
			1010	010100
			1011	010101
1	10	100	1100	011000
			1101	011001
		101	1110	011100
			1111	011101
	11	110	1000	011110
			1001	011111
		111	1010	100000
			1011	100001
			1010	100010
			1011	100011
		110	1010	100100
			1011	100101
			1010	100110
			1011	100111
		111	1100	101000
			1101	101001
			1110	101010
			1111	101011
		111	1110	101100
			1111	101101
			1110	101110
			1111	101111

total symbol codeword budget

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

# The Codeword Supermarket

0	00	000	0000	000000
			0001	000001
		001	0010	000100
			0011	000101
	01	010	0100	001000
			0101	001001
		011	0110	001100
			0111	001101
1	10	100	1000	010000
			1001	010001
		101	1010	010100
			1011	010101
	11	110	1100	011000
			1101	011001
		111	1110	011100
			1111	011101

total symbol codeword budget

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

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

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

# The Codeword Supermarket

0	00	000	0000	000000
			0001	000001
		001	0010	000100
			0011	000101
			0011	000110
	01	010	0100	001000
			0101	001001
		011	0110	001010
			0111	001011
			0111	001100
			0111	001101
1	10	100	1000	001110
			1001	001111
		101	1010	010000
			1011	010001
			1011	010010
			1011	010011
	11	110	1100	010100
			1101	010101
			1101	010110
		111	1110	010111
			1110	011000
			1111	011001
			1111	011010
			1111	011011
			1111	011100

total symbol codeword budget

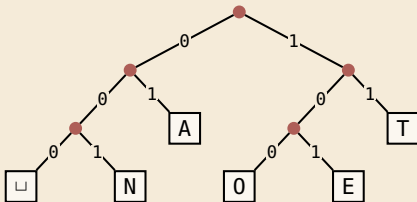
- Can “spend” at most budget of 1 across all codewords

- Codeword with  $\ell$  bits costs  $2^{-\ell}$

- Kraft-McMillan inequality:*  
any uniquely decodable code with codeword lengths  $\ell_1, \dots, \ell_\sigma$  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 → next)
  - ▶ **adaptive coding:** code depends on message and changes during encoding; implicitly stored withing the message (e. g., LZW → 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%	█			

~> Want shorter codes for more frequent characters!

# Huffman coding

e. g. frequencies / probabilities

- ▶ **Given:**  $\Sigma$  and weights  $w : \Sigma \rightarrow \mathbb{R}_{\geq 0}$
- ▶ **Goal:** prefix-free code  $E$  (= code trie) for  $\Sigma$  that minimizes coded text length  
i. e., a code trie minimizing  $\sum_{c \in \Sigma} w(c) \cdot |E(c)|$

# Huffman coding

e. g. frequencies / probabilities

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

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

- ▶ Let's abbreviate  $|S|_c = \text{\#occurrences of } c \text{ in } S$
- ▶ 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.,  $E(a) = u0$  and  $E(b) = u1$  for a bitstring  $u \in \{0, 1\}^*$  ( $u$  to be determined)
2. (Conceptually) replace  $a$  and  $b$  by a single character " $\boxed{ab}$ " with  $w(\boxed{ab}) = w(a) + w(b)$ .
3. Recursively apply Huffman's algorithm on the smaller alphabet. This in particular determines  $u = E(\boxed{ab})$ .

# 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.,  $E(a) = u0$  and  $E(b) = u1$  for a bitstring  $u \in \{0, 1\}^*$  ( $u$  to be determined)
  2. (Conceptually) replace  $a$  and  $b$  by a single character " $\boxed{ab}$ " with  $w(\boxed{ab}) = w(a) + w(b)$ .
  3. Recursively apply Huffman's algorithm on the smaller alphabet. This in particular determines  $u = E(\boxed{ab})$ .
- ▶ 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:  $S = \text{LOSSLESS}$        $\rightsquigarrow \Sigma_S = \{E, L, O, S\}$

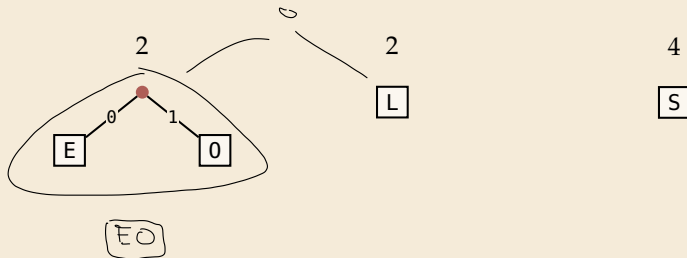
► Character frequencies:  $E : 1, \quad L : 2, \quad O : 1, \quad S : 4$

1	2	1	4
<div>E</div>	<div>L</div>	<div>O</div>	<div>S</div>

# Huffman's algorithm – Example

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

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

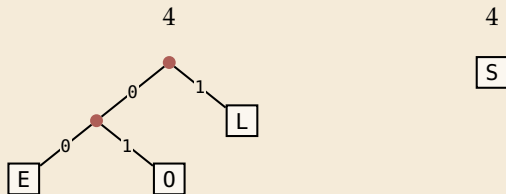




## Huffman's algorithm – Example

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

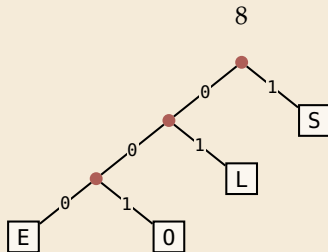
► Character frequencies:  $E : 1, \quad L : 2, \quad O : 1, \quad S : 4$



## Huffman's algorithm – Example

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

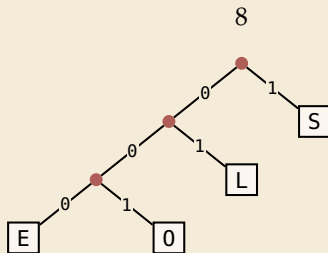
► Character frequencies:  $E : 1, \quad L : 2, \quad O : 1, \quad S : 4$



# Huffman's algorithm – Example

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

► Character frequencies:  $E : 1, \quad L : 2, \quad O : 1, \quad S : 4$

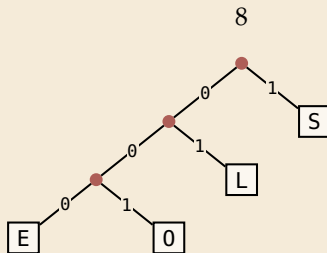


$\rightsquigarrow$  *Huffman tree* (code trie for Huffman code)

# Huffman's algorithm – Example

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

► Character frequencies:  $E : 1, \quad L : 2, \quad O : 1, \quad S : 4$



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

$\text{LOSSLESS} \rightarrow \underbrace{0100}_{L}1110100011$

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 CS566: 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.

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

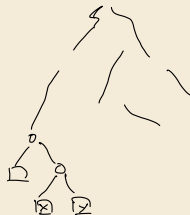
# Huffman code – Optimality

## Theorem 7.2 (Optimality of Huffman's Algorithm)

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

*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  $D$





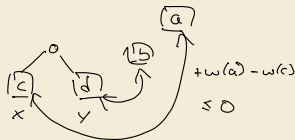
# Huffman code – Optimality

## Theorem 7.2 (Optimality of Huffman's Algorithm)

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

*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  $\ell$  smaller, so still optimal)



# Huffman code – Optimality

## Theorem 7.2 (Optimality of Huffman's Algorithm)

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

*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  $\ell$  smaller, so still optimal)
  - ▶ any optimal code for  $\Sigma' = \Sigma \setminus \{a, b\} \cup \{\boxed{ab}\}$  yields optimal code for  $\Sigma$  by replacing leaf  $\boxed{ab}$  by internal node with children  $a$  and  $b$ .
- $\rightsquigarrow$  recursive call yields optimal code for  $\Sigma'$  by inductive hypothesis, so Huffman's algorithm finds optimal code for  $\Sigma$ .



## 7.4 Entropy

# Entropy

## Definition 7.3 (Entropy)

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

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



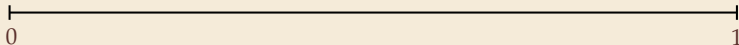
# Entropy

## Definition 7.3 (Entropy)

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

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

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



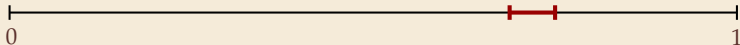
# Entropy

## Definition 7.3 (Entropy)

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

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

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



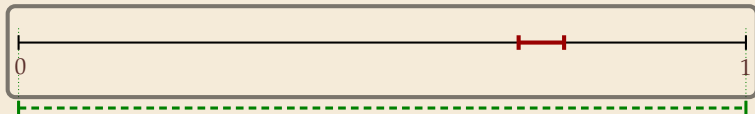
# Entropy

## Definition 7.3 (Entropy)

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

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

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



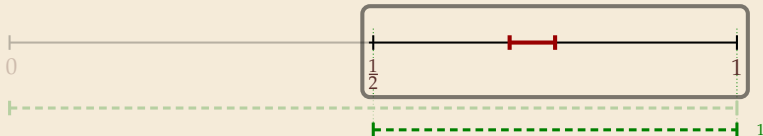
# Entropy

## Definition 7.3 (Entropy)

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

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

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





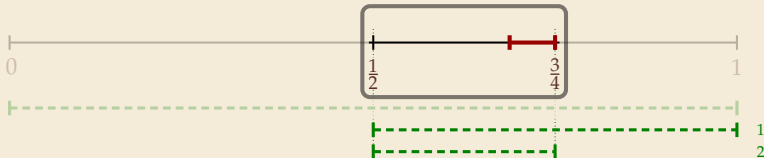
# Entropy

## Definition 7.3 (Entropy)

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

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

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



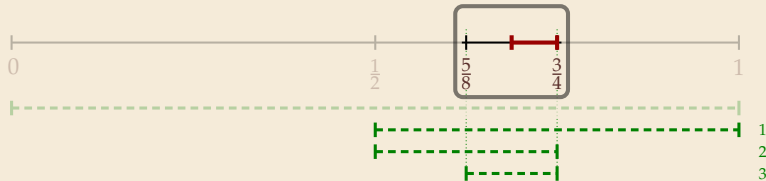
# Entropy

## Definition 7.3 (Entropy)

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

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

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



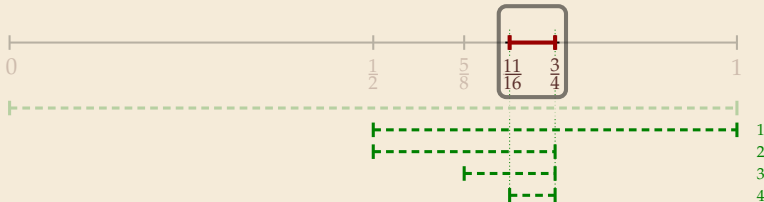
# Entropy

## Definition 7.3 (Entropy)

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

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

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



# Entropy

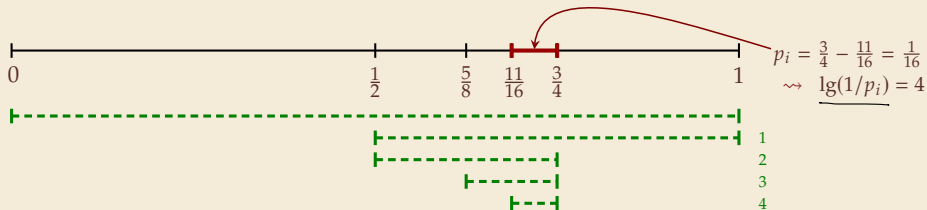
## Definition 7.3 (Entropy)

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

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

► entropy is a **measure of information** content of a distribution

► “20 Questions on  $[0, 1]$ ”: Land inside my interval by halving.



# Entropy

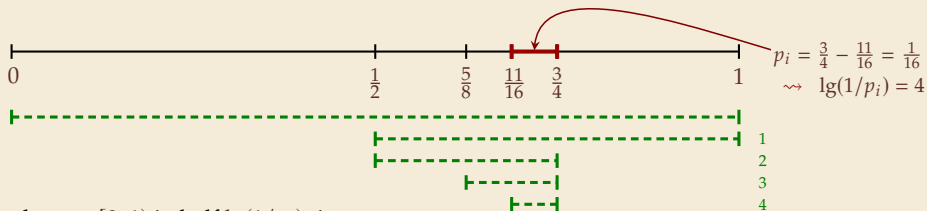
## Definition 7.3 (Entropy)

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

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

► 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

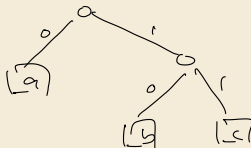
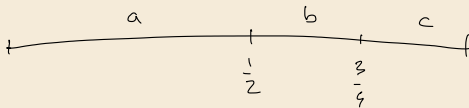
► more precisely: the expected number of bits (Yes/No questions) required to nail down the random value

# 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*!



$$|E(a)| = \lg\left(\frac{1}{p(a)}\right) \\ = 1 \frac{1}{2}$$

# 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  $p_1, \dots, p_\sigma$  for  $\Sigma = \{a_1, \dots, a_\sigma\}$ , the Huffman code  $E$  for  $\Sigma$  with weights  $p(a_i) = p_i$  satisfies  $\mathcal{H} \leq \ell(E) \leq \mathcal{H} + 1$  where  $\mathcal{H} = \mathcal{H}(p_1, \dots, p_\sigma)$ . ◀

# 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)

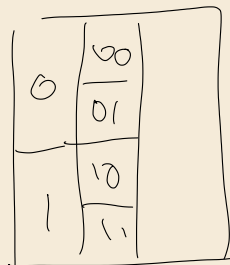
For any probabilities  $p_1, \dots, p_\sigma$  for  $\Sigma = \{a_1, \dots, a_\sigma\}$ , the Huffman code  $E$  for  $\Sigma$  with weights  $p(a_i) = p_i$  satisfies  $\mathcal{H} \leq \ell(E) \leq \mathcal{H} + 1$  where  $\mathcal{H} = \mathcal{H}(p_1, \dots, p_\sigma)$ .

*Proof sketch:*

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

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

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





# 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  $p_1, \dots, p_\sigma$  for  $\Sigma = \{a_1, \dots, a_\sigma\}$ , the Huffman code  $E$  for  $\Sigma$  with weights  $p(a_i) = p_i$  satisfies  $\mathcal{H} \leq \ell(E) \leq \mathcal{H} + 1$  where  $\mathcal{H} = \mathcal{H}(p_1, \dots, p_\sigma)$ . ◀

*Proof sketch:*

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

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

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

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

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

# 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  $p_1, \dots, p_\sigma$  for  $\Sigma = \{a_1, \dots, a_\sigma\}$ , the Huffman code  $E$  for  $\Sigma$  with weights  $p(a_i) = p_i$  satisfies  $\mathcal{H} \leq \ell(E) \leq \mathcal{H} + 1$  where  $\mathcal{H} = \mathcal{H}(p_1, \dots, p_\sigma)$ . ◀

*Proof sketch:*

- ▶  $\ell(E) \geq \mathcal{H}$

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

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

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

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

**Gibb's Inequality:**

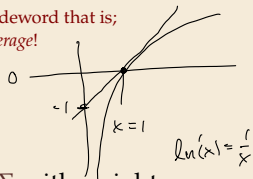
$\sum p_i = 1, \sum q_i \leq 1, 0 \leq p_i, q_i$

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

# 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  $p_1, \dots, p_\sigma$  for  $\Sigma = \{a_1, \dots, a_\sigma\}$ , the Huffman code  $E$  for  $\Sigma$  with weights  $p(a_i) = p_i$  satisfies  $\mathcal{H} \leq \ell(E) \leq \mathcal{H} + 1$  where  $\mathcal{H} = \mathcal{H}(p_1, \dots, p_\sigma)$ .

*Proof sketch:*

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

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

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

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

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

### Gibb's Inequality:

$\sum p_i = 1, \sum q_i \leq 1, 0 \leq p_i, q_i$

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

*Proof:*

Note: (\*)  $\ln(x) \leq x - 1$  ( $x \geq 0$ )  
(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  $p_1, \dots, p_\sigma$  for  $\Sigma = \{a_1, \dots, a_\sigma\}$ , the Huffman code  $E$  for  $\Sigma$  with weights  $p(a_i) = p_i$  satisfies  $\mathcal{H} \leq \ell(E) \leq \mathcal{H} + 1$  where  $\mathcal{H} = \mathcal{H}(p_1, \dots, p_\sigma)$ . ◀

*Proof sketch:*

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

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

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

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

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

### *Gibb's Inequality:*

$$\sum p_i = 1, \sum q_i \leq 1, 0 \leq p_i, q_i$$

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

*Proof:*

Note: (\*)  $\ln(x) \leq x - 1$  ( $x \geq 0$ )  
(by concavity of  $\ln$ )

$$\begin{aligned} & \sum p_i \ln\left(\frac{1}{p_i}\right) - \sum p_i \ln\left(\frac{1}{q_i}\right) \\ &= \sum p_i \ln\left(\frac{q_i}{p_i}\right) \stackrel{(*)}{\leq} \sum p_i \left(\frac{q_i}{p_i} - 1\right) \end{aligned}$$

# 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  $p_1, \dots, p_\sigma$  for  $\Sigma = \{a_1, \dots, a_\sigma\}$ , the Huffman code  $E$  for  $\Sigma$  with weights  $p(a_i) = p_i$  satisfies  $\mathcal{H} \leq \ell(E) \leq \mathcal{H} + 1$  where  $\mathcal{H} = \mathcal{H}(p_1, \dots, p_\sigma)$ .

*Proof sketch:*

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

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

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

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

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

### Gibb's Inequality:

$$\sum p_i = 1, \sum q_i \leq 1, 0 \leq p_i, q_i$$

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

*Proof:*

Note: (\*)  $\ln(x) \leq x - 1$  ( $x \geq 0$ )  
(by concavity of  $\ln$ )

$$\begin{aligned} & \sum p_i \ln\left(\frac{1}{p_i}\right) - \sum p_i \ln\left(\frac{1}{q_i}\right) \\ &= \sum p_i \ln\left(\frac{q_i}{p_i}\right) \stackrel{(*)}{\leq} \sum p_i \left(\frac{q_i}{p_i} - 1\right) \\ &= \sum q_i - \sum p_i \leq 0 \end{aligned}$$

## Entropy and Huffman codes [2]

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

►  $\ell(E) \leq \mathcal{H} + 1$       (2)  $\ell(E) \leq \ell(E')$

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

# 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  $\sum_{i=1}^{\sigma} p_i \lg\left(\frac{1}{q_i}\right) = \sum_{i=1}^{\sigma} p_i \lceil \lg(1/p_i) \rceil \leq \mathcal{H} + 1$ .

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

- If  $\sigma = 2$ ,  $E'$  uses a single bit each.

Here,  $q_i \leq 1/2$ , so  $\lg(1/q_i) \geq 1 = |E'(a_i)|$  ✓

# 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  $\sum_{i=1}^{\sigma} p_i \lg\left(\frac{1}{q_i}\right) = \sum_{i=1}^{\sigma} p_i \lceil \lg(1/p_i) \rceil \leq \mathcal{H} + 1$ .

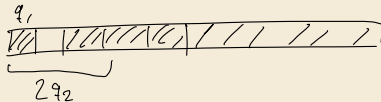
We construct a code  $E'$  for  $\Sigma$  with  $|E'(a_i)| \leq \lg(1/q_i)$  as follows;  
w.l.o.g. assume  $q_1 \leq q_2 \leq \dots \leq q_{\sigma}$

- 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  $q_1 + q_2 + \dots + q_{\sigma} \leq 1$ .





# 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  $\sum_{i=1}^{\sigma} p_i \lg\left(\frac{1}{q_i}\right) = \sum_{i=1}^{\sigma} p_i \lceil \lg(1/p_i) \rceil \leq \mathcal{H} + 1$ .

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

- 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  $q_1 + q_2 + \dots + q_{\sigma} \leq 1$ .

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

# Entropy and Huffman codes [2]

*Proof sketch (continued):*

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

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

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

- 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  $q_1 + q_2 + \dots + q_{\sigma} \leq 1$ .

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

By construction,  $|E'(a_1)| = |E'(a_2)| = |E'(\boxed{a_1 a_2})| + 1$ , 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  $\ell(E) \leq \ell(E') \leq \sum_{i=1}^{\sigma} p_i \lg\left(\frac{1}{q_i}\right) \leq \mathcal{H} + 1$ .

## Clicker Question

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



**A** always  $\leq$

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

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

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

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

*could be equal*



$\rightarrow$  [sli.do/cs566](https://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](https://sli.do/cs566)

# 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!)

# 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:  $p_i = \frac{|S|_{a_i}}{|S|} = \frac{\text{\#occurences of } a_i \text{ in string } S}{\text{length of } S}$

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

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

↪ Theorem 7.4 tells us rather precisely how well Huffman compresses:

$$\mathcal{H}_0(S) \cdot n \leq |C| \leq (\mathcal{H}_0(S) + 1)n$$

- ▶  $\mathcal{H}_0(S) = \mathcal{H}\left(\frac{|S|_{a_1}}{n}, \dots, \frac{|S|_{a_\sigma}}{n}\right) = \sum_{i=1}^{\sigma} \frac{n}{|S|_{a_i}} \log_2\left(\frac{|S|_{a_i}}{n}\right)$  is called the *empirical entropy* of  $S$  ↖ zero-th order empirical entropy

# Huffman coding – Discussion

- ▶ running time complexity:  $O(\sigma \log \sigma)$  to construct code
  - ▶ build PQ +  $\sigma \cdot (2 \text{ deleteMins and } 1 \text{ insert})$
  - ▶ 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 +  $\sigma \cdot (2 \text{ deleteMins and } 1 \text{ insert})$
  - ▶ 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 *cf exam*



have to store code alongside with coded text



# Part II

*Compressing repetitive texts*





## 7.5 Run-Length Encoding

# Run-Length encoding

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

[illegible]

same character repeated

- ▶ 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

[illegible]

same character repeated

- ▶ 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:  $S = \underbrace{00000}_{\text{run 1}} \underbrace{111}_{\text{run 2}} \underbrace{0000}_{\text{run 3}}$

# Run-Length encoding

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

[illegible]

same character repeated

- ▶ 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:  $S = \underbrace{00000}_{\text{0}} \underbrace{111}_{\text{1}} \underbrace{0000}_{\text{0}}$

↪ 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

[illegible]

same character repeated

- ▶ 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:  $S = \underbrace{00000}_{\text{0}} \underbrace{111}_{\text{1}} \underbrace{0000}_{\text{0}}$

⇒ 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 be arbitrarily long?

EOF



→ *[sli.do/cs566](https://sli.do/cs566)*

# Elias codes

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

## Elias codes

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

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


$7 \mapsto 00000001$      $3 \mapsto 0001$      $0 \mapsto 1$      $30 \mapsto 00000000000000000000000000000001$

## Elias codes

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

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

$7 \mapsto 00000001$      $3 \mapsto 0001$      $0 \mapsto 1$      $30 \mapsto 00000000000000000000000000000001$

 Much too long

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

## Elias codes

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

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

$7 \mapsto 00000001$      $3 \mapsto 0001$      $0 \mapsto 1$      $30 \mapsto 00000000000000000000000000000001$



👎 Much too long

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

- Refinement: *Elias gamma code*

- ▶ Store the **length**  $\ell$  of the binary representation in **unary**
- ▶ Followed by the binary digits themselves

# Elias codes

- ▶ Need a *prefix-free encoding* for  $\mathbb{N} = \{1, 2, 3, \dots\}$

- ▶ must allow arbitrarily large integers
- ▶ must know when to stop reading

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

$7 \mapsto 00000001$      $3 \mapsto 0001$      $0 \mapsto 1$      $30 \mapsto 00000000000000000000000000000001$



Much too long

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

- ▶ Refinement: **Elias gamma code**

- ▶ Store the **length**  $\ell$  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:

000110111011100110.



→ *[sli.do/cs566](https://sli.do/cs566)*

## Run-length encoding – Examples

► Encoding:

$$S = \textcolor{red}{1}111111001000000000000000000000001111111111$$
$$C = 1$$

► Decoding:

$C = 00001101001001010$

$$S =$$



## Run-length encoding – Examples

► Encoding:

$S = 111111100100000000000000000000001111111111$

 $k = 7$ 

$C = 100111$

► Decoding:

$$C = 00001101001001010$$
$$S =$$

## Run-length encoding – Examples

► Encoding:

$$S = 111111100100000000000000000000001111111111$$
 $k = 2$ 

$C = 100111010$

► Decoding:

$C = 00001101001001010$

$$S =$$

## Run-length encoding – Examples

► Encoding:

$S = 111111100100000000000000000000001111111111$

 $k = 1$ 

$C = 1001110101$

► Decoding:

$$C = 00001101001001010$$
$$S =$$

## Run-length encoding – Examples

► Encoding:

$$S = 1111111001000000000000000000001111111111$$
 $k = 20$ 

$C = 1001110101000010100$

► Decoding:

$C = 00001101001001010$

$$S =$$

## Run-length encoding – Examples

► Encoding:

$$S = 11111110010000000000000000000000000011111111111$$
 $k = 11$ 

$C = 10011101010000101000001011$

► Decoding:

$C = 00001101001001010$

$$S =$$

## Run-length encoding – Examples

► Encoding:

$S = 11111111001000000000000000000000000011111111111$

$C = 10011101010000101000001011$

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

► Decoding:

$C = 00001101001001010$

$S =$

## Run-length encoding – Examples

► Encoding:

$S = 11111111001000000000000000000000000011111111111$

$C = 10011101010000101000001011$

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

► Decoding:

$C = 00001101001001010$

$S =$

## Run-length encoding – Examples

► Encoding:

$S = 11111111001000000000000000000000000011111111111$

$C = 10011101010000101000001011$

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

► Decoding:

$C = 00001101001001010$

$b = 0$

$S =$



## Run-length encoding – Examples

► Encoding:

$S = 111111110010000000000000000000000000000011111111111$

$C = 10011101010000101000001011$

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

► Decoding:

$C = 00001101001001010$

$b = 0$

$\ell = 3 + 1$

$S =$

## Run-length encoding – Examples

► Encoding:

$S = 11111111001000000000000000000000000011111111111$

$C = 10011101010000101000001011$

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

► Decoding:

$C = 00001101001001010$

$b = 0$

$\ell = 3 + 1$

$k = 13$

$S = 00000000000000$

## Run-length encoding – Examples

► Encoding:

$S = 11111111001000000000000000000000000011111111111$

$C = 10011101010000101000001011$

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

► Decoding:

$C = 00001101001001010$

$b = 1$

$\ell = 2 + 1$

$k =$

$S = 00000000000000$

## Run-length encoding – Examples

► Encoding:

$S = 111111100100000000000000000000001111111111$

$C = 10011101010000101000001011$

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

► Decoding:

$C = 00001101001001010$

$b = 1$

$\ell = 2 + 1$

$k = 4$

$S = 000000000000001111$

## Run-length encoding – Examples

► Encoding:

$S = 111111100100000000000000000000001111111111$

$C = 10011101010000101000001011$

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

► Decoding:

$C = 00001101001001010$

$b = 0$

$\ell = 0 + 1$

$k =$

$S = 000000000000001111$

## Run-length encoding – Examples

► Encoding:

$S = 111111100100000000000000000000001111111111$

$C = 10011101010000101000001011$

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

► Decoding:

$C = 00001101001001010$

$b = 0$

$\ell = 0 + 1$

$k = 1$

$S = 0000000000000011110$

# Run-length encoding – Examples

► Encoding:

$S = 111111100100000000000000000000001111111111$

$C = 10011101010000101000001011$

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

► Decoding:

$C = 00001101001001010$

$b = 1$

$\ell = 1 + 1$

$k =$

$S = 0000000000000011110$

## Run-length encoding – Examples

► Encoding:

$S = 111111100100000000000000000000001111111111$

$C = 10011101010000101000001011$

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

► Decoding:

$C = 000011010010010\mathbf{10}$

$b = 1$

$\ell = 1 + 1$

$k = 2$

$S = 0000000000000011110\mathbf{11}$



## 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

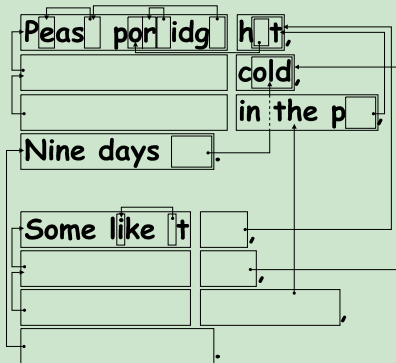
Peas porridge hot,  
cold,  
in the pot,  
Nine days .  
Some like it ,  
in the pot .

<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/cs566](https://sli.do/cs566)

# 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!
  - ~> each codeword refers to
    - ▶ either a single character in  $\Sigma_S$ ,
    - ▶ or a *substring* of  $S$  (that both encoder and decoder have seen before).

# 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!
  - ~> each codeword refers to
    - ▶ either a single character in  $\Sigma_S$ ,
    - ▶ or a *substring* of  $S$  (that both encoder and decoder have seen before).
  - ▶ Variants of Lempel-Ziv compression
    - ▶ "LZ77" Original version (sliding window, overlapping phrases)  
Derivatives: LZSS, LZFG, LZRW, LZW, DEFLATE, ... zip  
DEFLATE used in (pk)zip, gzip, PNG
    - ▶ "LZ78" Second version (whole-phrase references)  
Derivatives: LZW, LZMW, LZAP, LZY, ...  
LZW used in compress, GIF

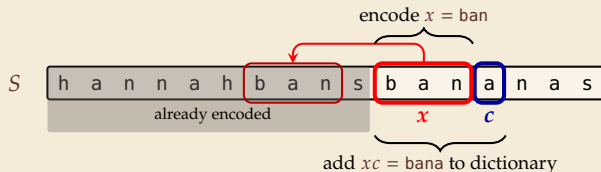


# Lempel-Ziv-Welch

- ▶ here: *Lempel-Ziv-Welch (LZW)* (arguably the “cleanest” variant of Lempel-Ziv)
- ▶ *variable-to-fixed encoding*
  - ▶ all codewords have  $k$  bits (typical:  $k = 12$ )  $\rightsquigarrow$  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

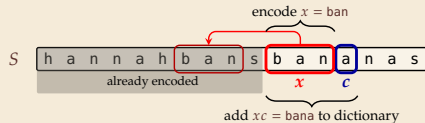
# LZW encoding – Example

Input: Y0!\_YOU!\_YOUR\_Y0Y0!

$\Sigma_S$  = ASCII character set (0–127)

C =

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	

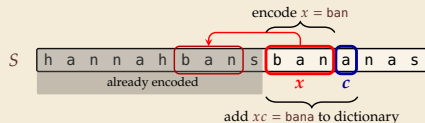
# LZW encoding – Example

Input: Y0!\_YOU!\_YOUR\_YOYO!

$\Sigma_S$  = 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

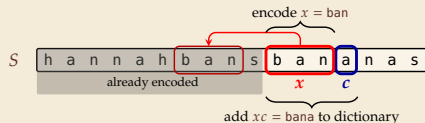
Input: **Y**0!\_YOU!\_YOUR\_YOYO!

Y

C = 89

$\Sigma_S$  = ASCII character set (0–127)

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	

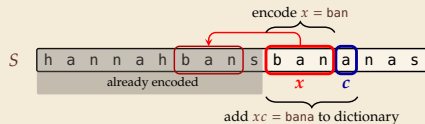
# LZW encoding – Example

Input: Y0!\_YOU!\_YOUR\_YOYO!

$\Sigma_S$  = 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	

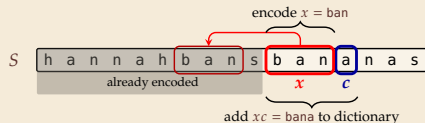
# LZW encoding – Example

Input: Y0!\_YOU!\_YOUR\_YOYO!

$\Sigma_S$  = 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	0!
130	
131	
132	
133	
134	
135	
136	
137	
138	
139	

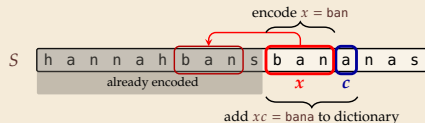
# LZW encoding – Example

Input: Y0!\_YOU!\_YOUR\_YOYO!

$\Sigma_S$  = ASCII character set (0–127)

Y    0    !  
C = 89   79   33

$D =$



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

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



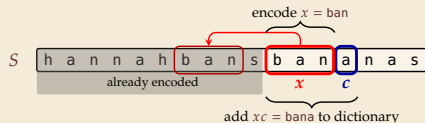
# LZW encoding – Example

Input: Y0!\_YOU!\_YOUR\_YOYO!

$\Sigma_S$  = ASCII character set (0–127)

Y    0    !  
C = 89   79   33

$D =$



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

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

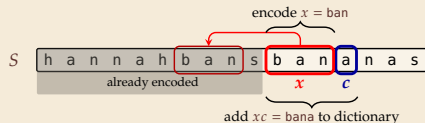
# LZW encoding – Example

Input: Y0!\_YOU!\_YOUR\_YOYO!

$\Sigma_S$  = ASCII character set (0–127)

Y    0    !    \_  
C = 89   79   33   32

$D =$



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

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

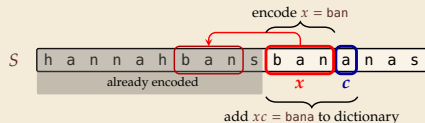
# LZW encoding – Example

Input: Y0!\_YOU!\_YOUR\_YOYO!

$\Sigma_S$  = ASCII character set (0–127)

Y    0    !    \_  
C = 89   79   33   32

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	

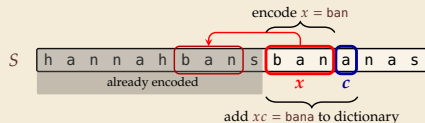
# LZW encoding – Example

Input: Y0!\_YOU!\_YOUR\_YOYO!

$\Sigma_S$  = 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	

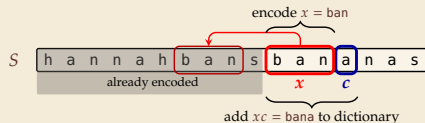
# LZW encoding – Example

Input: Y0!\_YOU!\_YOUR\_YOYO!

$\Sigma_S$  = 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	YOU
133	
134	
135	
136	
137	
138	
139	

# LZW encoding – Example

Input: Y0!\_YOU!\_YOUR\_YOY0!

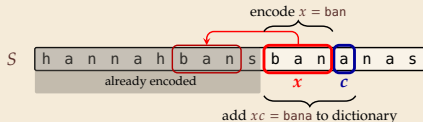
$\Sigma_S$  = ASCII character set (0–127)

Y    0    !    \_    Y0    U  
C = 89   79   33   32   128   85

$D =$

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

Code	String
128	Y0
129	0!
130	!_
131	_Y
132	YOU
133	
134	
135	
136	
137	
138	
139	



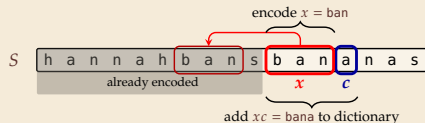
# LZW encoding – Example

Input: Y0!\_YOU!\_YOUR\_YOYO!

$\Sigma_S$  = ASCII character set (0–127)

Y	0	!	_	Y0	U
C = 89	79	33	32	128	85

$D =$



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

Code	String
128	Y0
129	0!
130	!_
131	_Y
132	YOU
133	U!
134	
135	
136	
137	
138	
139	

# LZW encoding – Example

Input: Y0!\_YOU!\_YOUR\_YOYO!

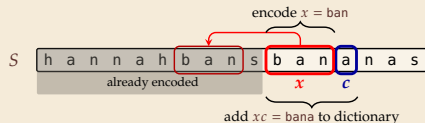
$\Sigma_S$  = ASCII character set (0–127)

Y    0    !    \_    Y0    U    !\_  
C = 89   79   33   32   128   85   130

D =

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

Code	String
128	Y0
129	0!
130	!_
131	_Y
132	YOU
133	U!
134	
135	
136	
137	
138	
139	





# LZW encoding – Example

Input: Y0!\_YOU!\_YOUR\_YOYO!

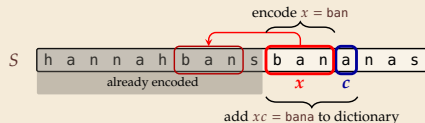
$\Sigma_S$  = ASCII character set (0–127)

Y    0    !    \_    Y0    U    !\_  
C = 89   79   33   32   128   85   130

D =

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

Code	String
128	Y0
129	0!
130	!_
131	_Y
132	YOU
133	U!
134	!_Y
135	
136	
137	
138	
139	



# LZW encoding – Example

Input: Y0!\_YOU!\_YOU\_R\_YOY0!

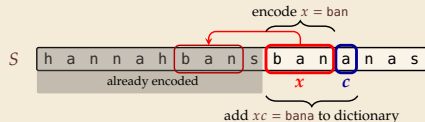
$\Sigma_S$  = ASCII character set (0–127)

Y	0	!	_	Y0	U	!_	YOU
C = 89	79	33	32	128	85	130	132

$D =$

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

Code	String
128	Y0
129	0!
130	!_
131	_Y
132	YOU
133	U!
134	!_Y
135	
136	
137	
138	
139	



# LZW encoding – Example

Input: Y0!\_YOU!\_YOU\_R\_Y0Y0!

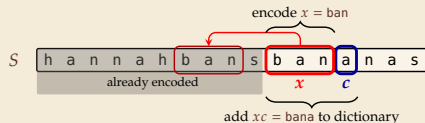
$\Sigma_S$  = ASCII character set (0–127)

Y    0    !    \_    Y0    U    !\_    YOU  
 $C =$  89   79   33   32   128   85   130   132

$D =$

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

Code	String
128	Y0
129	0!
130	!_
131	_Y
132	YOU
133	U!
134	!_Y
135	YOUR
136	
137	
138	
139	



# LZW encoding – Example

Input: Y0!\_YOU!\_YOUR\_Y0Y0!

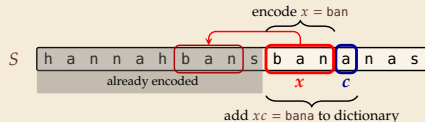
$\Sigma_S$  = ASCII character set (0–127)

Y    0    !    \_    Y0    U    !\_    YOU    R  
C = 89   79   33   32   128   85   130   132   82

D =

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

Code	String
128	Y0
129	0!
130	!_
131	_Y
132	YOU
133	U!
134	!_Y
135	YOUR
136	
137	
138	
139	



# LZW encoding – Example

Input: Y0!\_YOU!\_YOUR\_Y0Y0!

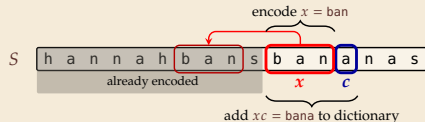
$\Sigma_S$  = ASCII character set (0–127)

Y    0    !    \_    Y0    U    !\_    YOU    R  
 $C =$  89   79   33   32   128   85   130   132   82

$D =$

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

Code	String
128	Y0
129	0!
130	!_
131	_Y
132	YOU
133	U!
134	!_Y
135	YOUR
136	<b>R_</b>
137	
138	
139	



# LZW encoding – Example

Input: Y0!\_YOU!\_YOUR\_Y0Y0!

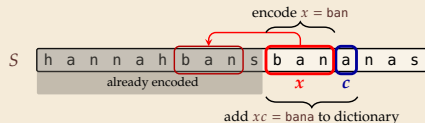
$\Sigma_S$  = ASCII character set (0–127)

Y	0	!	_	Y0	U	!_	YOU	R	_Y
C = 89	79	33	32	128	85	130	132	82	131

$D =$

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

Code	String
128	Y0
129	0!
130	!_
131	_Y
132	YOU
133	U!
134	!_Y
135	YOUR
136	R_
137	
138	
139	



# LZW encoding – Example

Input: Y0!\_YOU!\_YOUR\_Y0Y0!

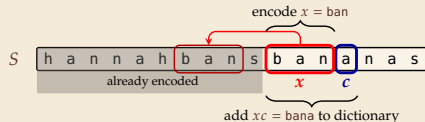
$\Sigma_S$  = ASCII character set (0–127)

Y	0	!	_	Y0	U	!_	YOU	R	_Y
C = 89	79	33	32	128	85	130	132	82	131

$D =$

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

Code	String
128	Y0
129	0!
130	!_
131	_Y
132	YOU
133	U!
134	!_Y
135	YOUR
136	R_
137	_Y0
138	
139	



# LZW encoding – Example

Input: Y0!\_YOU!\_YOUR\_YOYO!

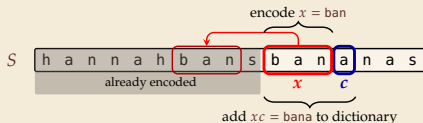
$\Sigma_S$  = ASCII character set (0–127)

Y	0	!	_	Y0	U	!_	YOU	R	_Y	0
C = 89	79	33	32	128	85	130	132	82	131	79

$D =$

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

Code	String
128	Y0
129	0!
130	!_
131	_Y
132	YOU
133	U!
134	!_Y
135	YOUR
136	R_
137	_Y0
138	
139	





# LZW encoding – Example

Input: Y0!\_YOU!\_YOUR\_YOY0!

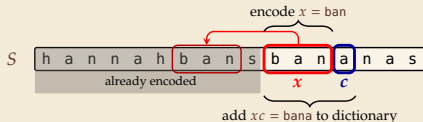
$\Sigma_S$  = ASCII character set (0–127)

Y    0    !    \_    Y0    U    !\_    YOU    R    \_Y    0  
 $C =$  89   79   33   32   128   85   130   132   82   131   79

$D =$

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

Code	String
128	Y0
129	0!
130	!_
131	_Y
132	YOU
133	U!
134	!_Y
135	YOUR
136	R_
137	_Y0
138	0Y
139	



# LZW encoding – Example

Input: Y0!\_YOU!\_YOUR\_Y0Y0!

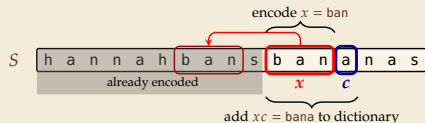
$\Sigma_S$  = ASCII character set (0–127)

Y	0	!	_	Y0	U	!_	YOU	R	_Y	0	Y0
C = 89	79	33	32	128	85	130	132	82	131	79	128

$D =$

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

Code	String
128	Y0
129	0!
130	!_
131	_Y
132	YOU
133	U!
134	!_Y
135	YOUR
136	R_
137	_Y0
138	0Y
139	



# LZW encoding – Example

Input: Y0!\_YOU!\_YOUR\_Y0Y0!

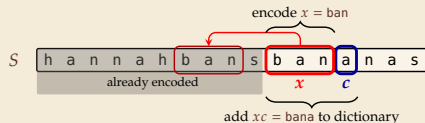
$\Sigma_S$  = ASCII character set (0–127)

Y 0 ! \_ Y0 U !\_ YOU R \_Y 0 Y0  
 $C = 89 \quad 79 \quad 33 \quad 32 \quad 128 \quad 85 \quad 130 \quad 132 \quad 82 \quad 131 \quad 79 \quad 128$

$D =$

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

Code	String
128	Y0
129	0!
130	!_
131	_Y
132	YOU
133	U!
134	!_Y
135	YOUR
136	R_
137	_Y0
138	0Y
139	Y0!



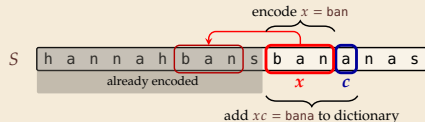
# LZW encoding – Example

Input: Y0!\_YOU!\_YOUR\_Y0Y0!

$\Sigma_S$  = ASCII character set (0–127)

Y    0    !    \_    Y0    U    !\_    YOU    R    \_Y    0    Y0    !  
 C = 89   79   33   32   128   85   130   132   82   131   79   128   33

D =



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

Code	String
128	Y0
129	0!
130	!_
131	_Y
132	YOU
133	U!
134	!_Y
135	YOUR
136	R_
137	_Y0
138	0Y
139	Y0!

# LZW encoding – Code

---

```
1 procedure LZWencode( $S[0..n]$ ):
2    $x := \varepsilon$  // previous phrase, initially empty
3    $C := \varepsilon$  // output, initially empty
4    $D :=$  dictionary, initialized with codes for  $c \in \Sigma_S$  // stored as trie ( $\rightsquigarrow$  Unit 13)
5    $k := |\Sigma_S|$  // next free codeword
6   for  $i := 0, \dots, n - 1$  do
7      $c := S[i]$ 
8     if  $D.\text{containsKey}(xc)$  then
9        $x := xc$ 
10    else
11       $C := C \cdot D.\text{get}(x)$  // append codeword for  $x$ 
12       $D.\text{put}(xc, k)$  // add  $xc$  to  $D$ , assigning next free codeword
13       $k := k + 1; x := c$ 
14  end for
15   $C := C \cdot D.\text{get}(x)$ 
16  return  $C$ 
```

$O(1)$  time

---

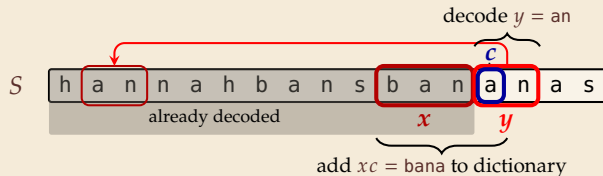
## 7.7 Lempel-Ziv-Welch Decoding

# LZW decoding

- Decoder has to replay the process of growing the dictionary!

⇒ **Decoding:**

after decoding a substring  $y$  of  $S$ , add  $xc$  to  $D$ ,  
where  $x$  is previously encoded/decoded substring of  $S$ ,  
and  $c = y[0]$  (first character of  $y$ )



⇒ Note: only start adding to  $D$  after *second* substring of  $S$  is decoded

# LZW decoding – Example

- ▶ Same idea: build dictionary while reading string.
- ▶ Example: 67 65 78 32 66 129 133

$D =$

Code #	String
...	
32	□
...	
...	
65	A
66	B
67	C
...	
78	N
...	
83	S
...	

input	decodes to	Code #	String (human)	String (computer)



# LZW decoding – Example

- ▶ Same idea: build dictionary while reading string.
- ▶ Example: 67 65 78 32 66 129 133

$D =$

Code #	String
...	
32	□
...	
...	
65	A
66	B
67	C
...	
78	N
...	
83	S
...	

input	decodes to	Code #	String (human)	String (computer)
67	C			

# LZW decoding – Example

- ▶ Same idea: build dictionary while reading string.
- ▶ Example: 67 65 78 32 66 129 133

$D =$

Code #	String
...	
32	□
...	
...	
65	A
66	B
67	C
...	
78	N
...	
83	S
...	

input	decodes to	Code #	String (human)	String (computer)
67	C			
65	A	128	CA	67, A

# LZW decoding – Example

- ▶ Same idea: build dictionary while reading string.
- ▶ Example: 67 65 78 32 66 129 133

$D =$

Code #	String
...	
32	□
...	
...	
65	A
66	B
67	C
...	
78	N
...	
83	S
...	

input	decodes to	Code #	String (human)	String (computer)
67	C			
65	A	128	CA	67, A
78	N	129	AN	65, N

# LZW decoding – Example

- ▶ Same idea: build dictionary while reading string.
- ▶ Example: 67 65 78 **32** 66 129 133

$D =$

Code #	String
...	
32	␣
...	
...	
65	A
66	B
67	C
...	
78	N
...	
83	S
...	

input	decodes to	Code #	String (human)	String (computer)
67	C			
65	A	128	CA	67, A
78	N	129	AN	65, N
<b>32</b>	<b>␣</b>	<b>130</b>	<b>N␣</b>	<b>78, ␣</b>

# LZW decoding – Example

- ▶ Same idea: build dictionary while reading string.
- ▶ Example: 67 65 78 32 **66** 129 133

$D =$

Code #	String
...	
32	␣
...	
...	
65	A
66	B
67	C
...	
78	N
...	
83	S
...	

input	decodes to	Code #	String (human)	String (computer)
67	C			
65	A	128	CA	67, A
78	N	129	AN	65, N
32	␣	130	N␣	78, ␣
<b>66</b>	<b>B</b>	<b>131</b>	<b>␣B</b>	<b>32, B</b>

# LZW decoding – Example

- ▶ Same idea: build dictionary while reading string.
- ▶ Example: 67 65 78 32 66 **129** 133

$D =$

Code #	String
...	
32	␣
...	
...	
65	A
66	B
67	C
...	
78	N
...	
83	S
...	

input	decodes to	Code #	String (human)	String (computer)
67	C			
65	A	128	CA	67, A
78	N	129	AN	65, N
32	␣	130	N␣	78, ␣
66	B	131	␣B	32, B
<b>129</b>	<b>AN</b>	<b>132</b>	<b>BA</b>	<b>66, A</b>

# LZW decoding – Example

- ▶ Same idea: build dictionary while reading string.
- ▶ Example: 67 65 78 32 66 129 **133**

$D =$

Code #	String
...	
32	␣
...	
...	
65	A
66	B
67	C
...	
78	N
...	
83	S
...	

input	decodes to	Code #	String (human)	String (computer)
67	C			
65	A	128	CA	67, A
78	N	129	AN	65, N
32	␣	130	N␣	78, ␣
66	B	131	␣B	32, B
129	AN	132	BA	66, A
<b>133</b>	<b>???</b>	<b>133</b>		

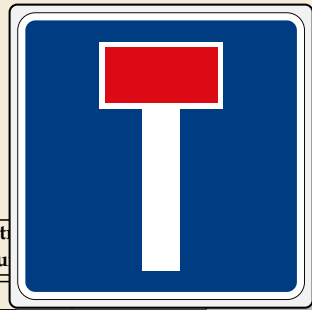
# LZW decoding – Example

- ▶ Same idea: build dictionary while reading string.
- ▶ Example: 67 65 78 32 66 129 133

$D =$

Code #	String
...	
32	␣
...	
...	
65	A
66	B
67	C
...	
78	N
...	
83	S
...	

input	decodes to	Code #	String (human)
67	C		
65	A	128	CA
78	N	129	AN
32	␣	130	N␣
66	B	131	␣B
129	AN	132	BA
133	???	133	





## LZW decoding – Bootstrapping

- ▶ example: Want to decode 133, but not yet in dictionary!



decoder is “one step behind” in creating dictionary

# LZW decoding – Bootstrapping

► example: Want to decode 133, but not yet in dictionary!



decoder is “one step behind” in creating dictionary

↪ problem occurs if *we want to use a code* that we are *just about to build*.

# LZW decoding – Bootstrapping

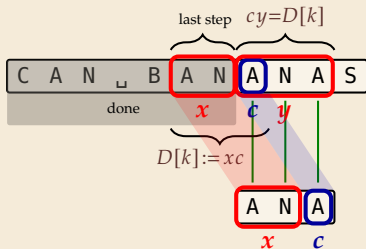
- ▶ example: Want to decode 133, but not yet in dictionary!



decoder is “one step behind” in creating dictionary

↪ problem occurs if *we want to use a code* that we are *just about to build*.

- ▶ But then we actually know what is going on!
  - ▶ Situation: decode using  $k$  in the step that will define  $k$ .
  - ▶ decoder knows last phrase  $x$ , needs phrase  $y = D[k] = xc$ .



1. en/decode  $x$ .

2. store  $D[k] := xc$

3. next phrase  $y$  equals  $D[k]$

↪  $D[k] = xc = \underline{x \cdot x[0]}$  (all known)

# LZW decoding – Code

---

```
1 procedure LZWdecode( $C[0..m]$ ):  
2    $D := \text{dictionary } [0..2^d) \rightarrow \Sigma_S^+$ , initialized with codes for  $c \in \Sigma_S$  // stored as array  
3    $k := |\Sigma_S|$  // next unused codeword  
4    $q := C[0]$  // first codeword  
5    $y := D[q]$  // lookup meaning of  $q$  in  $D$   
6    $S := y$  // output, initially first phrase  
7   for  $j := 1, \dots, m - 1$  do  
8      $x := y$  // remember last decoded phrase  
9      $q := C[j]$  // next codeword  
10    if  $q == k$  then  
11       $y := x \cdot x[0]$  // bootstrap case  
12    else  
13       $y := D[q]$   
14       $S := S \cdot y$  // append decoded phrase  
15       $D[k] := x \cdot y[0]$  // store new phrase  
16       $k := k + 1$   
17  end for  
18  return  $S$ 
```

---

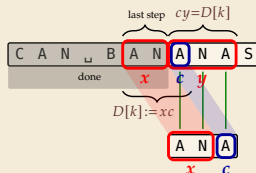
# LZW decoding – Example continued

► Example: 67 65 78 32 66 129 133 83

$D =$

Code #	String
...	
32	␣
...	
...	
65	A
66	B
67	C
...	
78	N
...	
83	S
...	

input	decodes to	Code #	String (human)	String (computer)
67	C			
65	A	128	CA	67, A
78	N	129	AN	65, N
32	␣	130	N␣	78, ␣
66	B	131	␣B	32, B
129	AN	132	BA	66, A
133	ANA	133	ANA	129, A



1. en/decode  $x$ .

2. store  $D[k] := xc$

3. next phrase  $y$  equals  $D[k]$

$\rightsquigarrow D[k] = xc = x \cdot x[0]$  (all known)

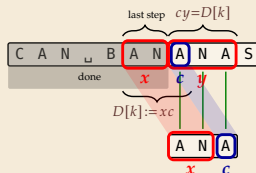
# LZW decoding – Example continued

► Example: 67 65 78 32 66 129 133 83

$D =$

Code #	String
...	
32	␣
...	
...	
65	A
66	B
67	C
...	
78	N
...	
83	S
...	

input	decodes to	Code #	String (human)	String (computer)
67	C			
65	A	128	CA	67, A
78	N	129	AN	65, N
32	␣	130	N␣	78, ␣
66	B	131	␣B	32, B
129	AN	132	BA	66, A
133	ANA	133	ANA	129, A
83	S	134	ANAS	133, S



1. en/decode  $x$ .

2. store  $D[k] := xc$

3. next phrase  $y$  equals  $D[k]$

$\rightsquigarrow D[k] = xc = x \cdot x[0]$  (all known)

## Clicker Question

How many phrases will LZW create on  $S = a^n$ , a run of  $n$  copies of  $a$ s?



**A**  $\sim n$

**B**  $\sim n/2$

**C**  $\sim n/4$

**D**  $\Theta(n/\log n)$

**E**  $\Theta(\sqrt{n})$

**F**  $\Theta(\log n)$

**G**  $\Theta(\log \log n)$

**H** 2

**I** 1



→ [sli.do/cs566](https://sli.do/cs566)

$\underbrace{a a a a a a}_{1+2+3+4+5} a a a a a = \sum_{i=1}^m i \cdot i! \cdot n$

**F**  ~~$\Theta(\log n)$~~

**G**  ~~$\Theta(\log \log n)$~~

**H** 2

1



→ [sli.do/cs566](https://sli.do/cs566)



# LZW – Discussion

- ▶ As presented, LZW uses coded alphabet  $\Sigma_C = [0..2^d)$ .
  - ↪ use another encoding for code numbers  $\mapsto$  binary, e. g., Huffman
- ▶ need a rule when dictionary is full; different options:
  - ▶ increment  $d$  ↪ longer codewords
  - ▶ “flush” dictionary and start from scratch ↪ limits extra space usage
  - ▶ often: reserve a codeword to trigger flush at any time
- ▶ encoding and decoding both run in linear time (assuming  $|\Sigma_S|$  constant)

# LZW – Discussion

- ▶ As presented, LZW uses coded alphabet  $\Sigma_C = [0..2^d)$ .
  - $\rightsquigarrow$  use another encoding for code numbers  $\mapsto$  binary, e. g., Huffman
- ▶ need a rule when dictionary is full; different options:
  - ▶ increment  $d \rightsquigarrow$  longer codewords
  - ▶ “flush” dictionary and start from scratch  $\rightsquigarrow$  limits extra space usage
  - ▶ often: reserve a codeword to trigger flush at any time
- ▶ encoding and decoding both run in linear time (assuming  $|\Sigma_S|$  constant)



fast encoding & decoding



works in streaming model (no random access, no backtrack on input needed)



significant compression for many types of data



captures only local repetitions (with bounded dictionary)

# Compression summary

Huffman codes	Run-length encoding	Lempel-Ziv-Welch
fixed-to-variable	variable-to-variable	variable-to-fixed
2-pass	1-pass	1-pass
must send dictionary	can be worse than ASCII	can be worse than ASCII
60% compression on English text	bad on text	45% compression on English text
optimal binary character encoding	good on long runs (e.g., pictures)	good on English text
rarely used directly	rarely used directly	frequently used
part of pkzip, JPEG, MP3	fax machines, old picture-formats	GIF, part of PDF, Unix compress

# Part III

## *Text Transformations*

# Text transformations

- ▶ compression is effective if we have one the following:
  - ▶ long runs  $\rightsquigarrow$  RLE
  - ▶ frequently used characters  $\rightsquigarrow$  Huffman
  - ▶ many (locally) repeated substrings  $\rightsquigarrow$  LZW

# Text transformations

- ▶ compression is effective if we have one the following:
  - ▶ long runs  $\rightsquigarrow$  RLE
  - ▶ frequently used characters  $\rightsquigarrow$  Huffman
  - ▶ many (locally) repeated substrings  $\rightsquigarrow$  LZW
- ▶ but methods can be frustratingly “blind” to other “obvious” redundancies
  - ▶ LZW: repetition too distant ⚡ dictionary already flushed
  - ▶ Huffman: changing probabilities (local clusters) ⚡ averaged out globally
  - ▶ RLE: run of alternating pairs of characters ⚡ not a run

# Text transformations

- ▶ compression is effective if we have one the following:
  - ▶ long runs  $\rightsquigarrow$  RLE
  - ▶ frequently used characters  $\rightsquigarrow$  Huffman
  - ▶ many (locally) repeated substrings  $\rightsquigarrow$  LZW
- ▶ but methods can be frustratingly “blind” to other “obvious” redundancies
  - ▶ LZW: repetition too distant ⚡ dictionary already flushed
  - ▶ Huffman: changing probabilities (local clusters) ⚡ averaged out globally
  - ▶ RLE: run of alternating pairs of characters ⚡ not a run
- ▶ Enter: **text transformations**
  - ▶ invertible functions of text
  - ▶ do not by themselves reduce the space usage
  - ▶ but help compressors “see” existing redundancy
  - $\rightsquigarrow$  use as pre-/postprocessing in a compression pipeline

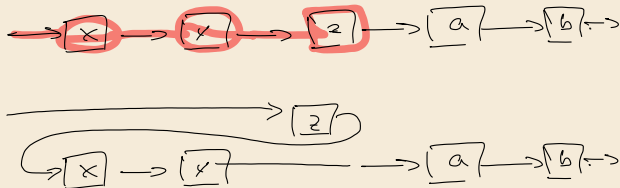
## 7.8 Move-to-Front Transformation



# Move to Front

- ▶ **Move to Front (MTF)** is a heuristic for *self-adjusting linked lists*
  - ▶ unsorted linked list of objects
  - ▶ whenever an element is accessed, it is moved to the front of the list (leaving the relative order of other elements unchanged)
- ~> list “learns” probabilities of access to objects
  - makes access to frequently requested objects cheaper

search z



# Move to Front

- ▶ *Move to Front (MTF)* is a heuristic for *self-adjusting linked lists*
  - ▶ unsorted linked list of objects
  - ▶ whenever an element is accessed, it is moved to the front of the list (leaving the relative order of other elements unchanged)
  - ~> list “learns” probabilities of access to objects  
makes access to frequently requested objects cheaper
- ▶ Here: use such a list for storing *source alphabet*  $\Sigma_S$ 
  - ▶ to encode  $c$ , access it in list
  - ▶ encode  $c$  using its (old) **position in list**
  - ▶ then apply MTF to the list
  - ~> codewords are integers, i. e.,  $\Sigma_C = \underline{[0..\sigma)}$

# Move to Front

- ▶ *Move to Front (MTF)* is a heuristic for *self-adjusting linked lists*
  - ▶ unsorted linked list of objects
  - ▶ whenever an element is accessed, it is moved to the front of the list (leaving the relative order of other elements unchanged)
  - ↪ list “learns” probabilities of access to objects
    - makes access to frequently requested objects cheaper
- ▶ Here: use such a list for storing *source alphabet*  $\Sigma_S$ 
  - ▶ to encode  $c$ , access it in list
  - ▶ encode  $c$  using its (old) **position in list**
  - ▶ then apply MTF to the list
  - ↪ codewords are integers, i. e.,  $\Sigma_C = [0..\sigma)$
- ↪ clusters of few characters    ↪ many small numbers

# MTF – Code

## ► Transform (encode):

---

```
1 procedure MTF–encode( $S[0..n]$ ):  
2    $L :=$  list containing  $\Sigma_S$  (sorted order)  
3    $C := \varepsilon$   
4   for  $i := 0, \dots, n - 1$  do  
5      $c := S[i]$   
6      $p :=$  position of  $c$  in  $L$   
7      $C := C \cdot p$   
8     Move  $c$  to front of  $L$   
9   end for  
10  return  $C$ 
```

---

# MTF – Code

## ► Transform (encode):

---

```
1 procedure MTF–encode( $S[0..n]$ ):  
2    $L :=$  list containing  $\Sigma_S$  (sorted order)  
3    $C := \varepsilon$   
4   for  $i := 0, \dots, n - 1$  do  
5      $c := S[i]$   
6      $p :=$  position of  $c$  in  $L$   
7      $C := C \cdot p$   
8     Move  $c$  to front of  $L$   
9   end for  
10  return  $C$ 
```

---

## ► Inverse transform (decode):

---

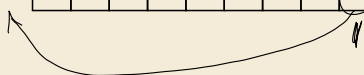
```
1 procedure MTF–decode( $C[0..m]$ ):  
2    $L :=$  list containing  $\Sigma_S$  (sorted order)  
3    $S := \varepsilon$   
4   for  $j := 0, \dots, m - 1$  do  
5      $p := C[j]$   
6      $c :=$  character at position  $p$  in  $L$   
7      $S := S \cdot c$   
8     Move  $c$  to front of  $L$   
9   end for  
10  return  $S$ 
```

---

► Important: encoding and decoding produce same accesses to list

## MTF – Example

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z



$S =$  I N E F F I C I E N C I E S

$C =$   $\emptyset$

## MTF – Example

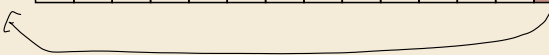
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z

$S =$  I N E F F I C I E N C I E S

$C =$  8

## MTF – Example

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
I	A	B	C	D	E	F	G	H	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z



$S =$  I N E F F I C I E N C I E S

$C =$  8 13



## MTF – Example

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
N	I	A	B	C	D	E	F	G	H	J	K	L	M	O	P	Q	R	S	T	U	V	W	X	Y	Z

$S =$  I N E F F I C I E N C I E S

$C =$  8 13 6

## MTF – Example

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
E	N	I	A	B	C	D	F	G	H	J	K	L	M	O	P	Q	R	S	T	U	V	W	X	Y	Z

$S =$  I N E **F** F I C I E N C I E S

$C =$  8 13 6 **7**

## MTF – Example

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
F	E	N	I	A	B	C	D	G	H	J	K	L	M	O	P	Q	R	S	T	U	V	W	X	Y	Z

$S =$  I N E F F I C I E N C I E S

$C =$  8 13 6 7 0

## MTF – Example

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
F	E	N	I	A	B	C	D	G	H	J	K	L	M	O	P	Q	R	S	T	U	V	W	X	Y	Z

$S =$  I N E F F I C I E N C I E S

$C =$  8 13 6 7 0 3

## MTF – Example

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
I	F	E	N	A	B	C	D	G	H	J	K	L	M	O	P	Q	R	S	T	U	V	W	X	Y	Z

$S =$  I N E F F I C I E N C I E S

$C =$  8 13 6 7 0 3 6

## MTF – Example

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
C	I	F	E	N	A	B	D	G	H	J	K	L	M	O	P	Q	R	S	T	U	V	W	X	Y	Z

$S =$  I N E F F I C I E N C I E S

$C =$  8 13 6 7 0 3 6 1

## MTF – Example

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
S	E	I	C	N	F	A	B	D	G	H	J	K	L	M	O	P	Q	R	T	U	V	W	X	Y	Z

$S =$  I N E F F I C I E N C I E S  
 $C =$  8 13 6 7 0 3 6 1 3 4 3 3 3 18

- What does a run in  $S$  encode to in  $C$ ?
- What does a run in  $C$  mean about the source  $S$ ?

# MTF – Discussion

- ▶ MTF itself does not compress text (if we store codewords with fixed length)

~> used as part of longer pipeline



- ▶ Effect (informal):

*MTF converts locally low empirical entropy to globally low empirical entropy(!)*

~> makes Huffman coding much more effective!

- ▶ cheaper option: Elias gamma code

~> smaller numbers gets shorter codewords

works well for text with small “local effective” alphabet

👎 many natural texts do not have locally low empirical entropy

👍 but we can often make it so ... stay tuned (→ BWT)



## **7.9 Burrows-Wheeler Transform**

# Burrows-Wheeler Transform

- ▶ Burrows-Wheeler Transform (BWT) is a sophisticated text-transformation technique.
  - ▶ coded text has same letters as source, just in a different order
  - ▶ But: coded text is (typically) more compressible (local char frequencies)

# Burrows-Wheeler Transform

- ▶ Burrows-Wheeler Transform (BWT) is a sophisticated text-transformation technique.
  - ▶ coded text has same letters as source, just in a different order
  - ▶ But: coded text is (typically) more compressible (local char frequencies)
- ▶ Encoding algorithm needs **all** of  $S$  (no streaming possible).
  - ↪ BWT is a *block compression method*.

# Burrows-Wheeler Transform

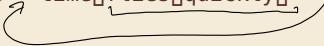
- ▶ Burrows-Wheeler Transform (BWT) is a sophisticated text-transformation technique.
  - ▶ coded text has same letters as source, just in a different order
  - ▶ But: coded text is (typically) more compressible (local char frequencies)
- ▶ Encoding algorithm needs **all** of  $S$  (no streaming possible).
  - ↪ BWT is a *block compression method*.
- ▶ BWT followed by MTF, RLE, and Huffman is the algorithm used by the bzip2 program. achieves best compression on English text of any algorithm we have seen:

4047392	bible.txt	# original	
1191071	bible.txt.gz	# gzip	(0.2s)
888604	bible.txt.7z	# 7z	(2s)
845635	bible.txt.bz2	# bzip2	(0.3s)
632634	bible.txt.paq8l	# paq8l -8	(6min!)

# BWT – Definitions

► *cyclic shift* of a string:

$T = \text{time\_flies\_quickly\_}$



$\text{flies\_quickly\_time\_}$

↪ cyclic shift



# BWT – Definitions

- *cyclic shift* of a string:

$T = \text{time\_flies\_quickly\_}$

$\text{flies\_quickly\_time\_}$

- add *end-of-word character* \$ to  $S$   
(always assumed in this section!)

⇒ can **recover**  
original string



⇒ cyclic shift



# BWT – Definitions

- *cyclic shift* of a string:

$T = \text{time\_flies\_quickly\_}$

$\text{flies\_quickly\_time\_}$

- add *end-of-word character* \$ to  $S$   
(always assumed in this section!)

~> can **recover**  
original string



~> cyclic shift



- The Burrows-Wheeler Transform proceeds in three steps:

0. Append end-of-word character \$ to  $S$ .

1. Place *all cyclic shifts* of  $S$  in a list  $L$

2. Sort the strings in  $L$  lexicographically

3.  $B$  is the *list of trailing characters* (last column, top-down) of each string in  $L$

## BWT – Example

$S = \text{alf\_eats\_alfalfa}$

1. Take all cyclic shifts of  $S$

alf\_eats\_alfalfa  
lf\_eats\_alfalfa\$  
f\_eats\_alfalfas  
\_eats\_alfalfasalf  
eats\_alfalfasalf\_  
ats\_alfalfasalf\_e  
ts\_alfalfasalf\_ea  
s\_alfalfasalf\_eat  
\_alfalfasalf\_eats  
alfalfasalf\_eats\_  
lfalfasalf\_eats\_a  
falfasalf\_eats\_al  
alfasalf\_eats\_alf  
lfasalf\_eats\_alfa  
fasalf\_eats\_alfal  
asalf\_eats\_alfalf  
\$alf\_eats\_alfalfa

sort



# BWT – Example

$S = \text{alf\_eats\_alfalfa\$}$

1. Take all cyclic shifts of  $S$
2. Sort cyclic shifts

alf\_eats\_alfalfa\$  
lf\_eats\_alfalfa\$  
f\_eats\_alfalfa\$  
\_eats\_alfalfa\$  
eats\_alfalfa\$  
ats\_alfalfa\$  
ts\_alfalfa\$  
s\_alfalfa\$  
\_alfalfa\$  
alfalfa\$  
lfalfa\$  
falfa\$  
alfa\$  
lfa\$  
fa\$  
a\$  
\$alf\_eats\_alfalfa

sort

\$alf\_eats\_alfalfa  
\_alfalfa\$alf\_eats  
\_eats\_alfalfa\$alf  
a\$alf\_eats\_alfalf  
alf\_eats\_alfalfa\$  
alfalfa\$alf\_eats\_  
ats\_alfalfa\$alf\_e  
eats\_alfalfa\$alf\_  
f\_eats\_alfalfa\$al  
fa\$alf\_eats\_alfal  
falfa\$alf\_eats\_al  
lf\_eats\_alfalfa\$  
lfa\$alf\_eats\_alfa  
lfalfa\$alf\_eats\_a  
s\_alfalfa\$alf\_eat  
ts\_alfalfa\$alf\_ea

# BWT – Example

$S = \text{alf\_eats\_alfalfa\$}$

1. Take all cyclic shifts of  $S$
2. Sort cyclic shifts
3. Extract last column

$B = \text{asff\$f\_e\_lllaaata}$



↑  
each column contains letters  
S[0], S[1], ..., S[u]

↑  
here too!

# BWT – Example

$S = \text{alf\_eats\_alfalfa\$}$

1. Take all cyclic shifts of  $S$
2. Sort cyclic shifts
3. Extract last column

$B = \text{asff\$f\_e\_lllaaata}$

alf\_eats\_alfalfa\$  
lf\_eats\_alfalfa\$  
f\_eats\_alfalfa\$al  
\_eats\_alfalfa\$alf  
eats\_alfalfa\$alf\_  
ats\_alfalfa\$alf\_e  
ts\_alfalfa\$alf\_ea  
s\_alfalfa\$alf\_eat  
\_alfalfa\$alf\_eats  
alfalfa\$alf\_eats\_  
lfalfa\$alf\_eats\_a  
falfa\$alf\_eats\_al  
alfa\$alf\_eats\_alf  
lfa\$alf\_eats\_alfa  
fa\$alf\_eats\_alfal  
a\$alf\_eats\_alfalf  
\$alf\_eats\_alfalfa

sort

\$alf\_eats\_alfalf**a**  
\_alfalfa\$alf\_eat**s**  
\_eats\_alfalfa\$alf**f**  
a\$alf\_eats\_alfalf**f**  
alf\_eats\_alfalfa**\$**  
alfalfa\$alf\_eats\_**alf**  
ats\_alfalfa\$alf\_**e**  
eats\_alfalfa\$alf\_**t**  
f\_eats\_alfalfa\$a**l**  
fa\$alf\_eats\_alfal**l**  
falfa\$alf\_eats\_**alf**  
lf\_eats\_alfalfa**\$a**  
lfa\$alf\_eats\_alf**a**  
lfalfa\$alf\_eats\_**a**  
s\_alfalfa\$alf\_eat**t**  
ts\_alfalfa\$alf\_ea**a**

BWT  
↓

- ▶ BWT can be computed in  $O(n)$  time!
  - ▶ totally non-obvious from definition (naive sorting could take  $\Omega(n^2)$  time in worst case!)
  - ▶ will use one of the most sophisticated algorithms we cover  $\rightsquigarrow$  Unit 13!

# BWT – Properties

## Why does BWT help for compression?

- ▶ sorting *groups* characters *by what follows*
  - ▶ Example: `lf` always preceded by `a`
  - ▶ more generally: BWT can be partitioned into letters following a given context

↪ repeated substring in  $S$  ↪ *runs* in  $B$

- ▶ Example: alf ↪ run of `as`
- ▶ picked up by RLE

(formally: low higher-order empirical entropy)

↪ If  $S$  allows predicting symbols from context,  
 $B$  has locally low entropy of characters.

- ▶ that makes MTF effective!

	$r$		$L[r]$
alf_eats_alfalfa\$	0	\$alf_eats_alfalfa <b>a</b>	16
lf_eats_alfalfa\$a	1	_alfalfa\$alf_eat <b>s</b>	8
f_eats_alfalfa\$al	2	_eats_alfalfa\$al <b>f</b>	3
_eats_alfalfa\$alf	3	a\$alf_eats_alfal <b>f</b>	15
eats_alfalfa\$alf_	4	alf_eats_alfalfa <b>\$</b>	0
ats_alfalfa\$alf_e	5	alfa\$alf_eats_al <b>f</b>	12
ts_alfalfa\$alf_ea	6	alfalfa\$alf_eats_ <b>_</b>	9
s_alfalfa\$alf_eat	7	ats_alfalfa\$alf_ <b>e</b>	5
_alfalfa\$alf_eats	8	eats_alfalfa\$alf_ <b>_</b>	4
alfalfa\$alf_eats_	9	f_eats_alfalfa\$al <b>l</b>	2
lfalfa\$alf_eats_a	10	fa\$alf_eats_alfal <b>l</b>	14
falfa\$alf_eats_al	11	falfa\$alf_eats_al <b>l</b>	11
alfa\$alf_eats_alf	12	lf_eats_alfalfa\$ <b>a</b>	1
lfa\$alf_eats_alfa	13	lfa\$alf_eats_alf <b>a</b>	13
fa\$alf_eats_alfal	14	lfalfa\$alf_eats_ <b>a</b>	10
a\$alf_eats_alfalf	15	s_alfalfa\$alf_eat <b>t</b>	7
\$alf_eats_alfalfa	16	ts_alfalfa\$alf_ea	6

# A Bigger Example

have\_had\_hadnt\_hasnt\_havent\_has\_what\$  
 ave\_had\_hadnt\_hasnt\_havent\_has\_what\$h  
 ve\_had\_hadnt\_hasnt\_havent\_has\_what\$ha  
 e\_had\_hadnt\_hasnt\_havent\_has\_what\$hav  
 \_had\_hadnt\_hasnt\_havent\_has\_what\$have  
 \_had\_hadnt\_hasnt\_havent\_has\_what\$have\_  
 ad\_hadnt\_hasnt\_havent\_has\_what\$have\_h  
 d\_hadnt\_hasnt\_havent\_has\_what\$have\_ha  
 \_hadnt\_hasnt\_havent\_has\_what\$have\_had  
 hadnt\_hasnt\_havent\_has\_what\$have\_had\_  
 adnt\_hasnt\_havent\_has\_what\$have\_had\_h  
 dnt\_hasnt\_havent\_has\_what\$have\_had\_ha  
 nt\_hasnt\_havent\_has\_what\$have\_had\_had  
 t\_hasnt\_havent\_has\_what\$have\_had\_hadn  
 \_hasnt\_havent\_has\_what\$have\_had\_hadnt  
 hasnt\_havent\_has\_what\$have\_had\_hadnt\_  
 asnt\_havent\_has\_what\$have\_had\_hadnt\_h  
 snt\_havent\_has\_what\$have\_had\_hadnt\_ha  
 nt\_havent\_has\_what\$have\_had\_hadnt\_has  
 t\_havent\_has\_what\$have\_had\_hadnt\_hasn  
 \_havent\_has\_what\$have\_had\_hadnt\_hasnt  
 havent\_has\_what\$have\_had\_hadnt\_hasnt\_  
 avent\_has\_what\$have\_had\_hadnt\_hasnt\_h  
 vent\_has\_what\$have\_had\_hadnt\_hasnt\_ha  
 ent\_has\_what\$have\_had\_hadnt\_hasnt\_hav  
 nt\_has\_what\$have\_had\_hadnt\_hasnt\_have  
 t\_has\_what\$have\_had\_hadnt\_hasnt\_haven  
 \_has\_what\$have\_had\_hadnt\_hasnt\_havent  
 has\_what\$have\_had\_hadnt\_hasnt\_havent\_  
 as\_what\$have\_had\_hadnt\_hasnt\_havent\_h  
 s\_what\$have\_had\_hadnt\_hasnt\_havent\_ha  
 \_what\$have\_had\_hadnt\_hasnt\_havent\_has  
 what\$have\_had\_hadnt\_hasnt\_havent\_has\_  
 hat\$have\_had\_hadnt\_hasnt\_havent\_has\_w  
 at\$have\_had\_hadnt\_hasnt\_havent\_has\_wh  
 t\$have\_had\_hadnt\_hasnt\_havent\_has\_wha  
 \$have\_had\_hadnt\_hasnt\_havent\_has\_what

\$have\_had\_hadnt\_hasnt\_havent\_has\_what  
 \_had\_hadnt\_hasnt\_havent\_has\_what\$have  
 \_hadnt\_hasnt\_havent\_has\_what\$have\_had  
 \_has\_what\$have\_had\_hadnt\_hasnt\_havent  
 \_hasnt\_havent\_has\_what\$have\_had\_hadnt  
 \_havent\_has\_what\$have\_had\_hadnt\_hasnt  
 \_what\$have\_had\_hadnt\_hasnt\_havent\_ha  
 s\_ad\_hadnt\_hasnt\_havent\_has\_what\$have\_h  
 adnt\_hasnt\_havent\_has\_what\$have\_had\_h  
 as\_what\$have\_had\_hadnt\_hasnt\_havent\_h  
 asnt\_havent\_has\_what\$have\_had\_hadnt\_h  
 at\$have\_had\_hadnt\_hasnt\_havent\_has\_w  
 ave\_had\_hadnt\_hasnt\_havent\_has\_what\$  
 avent\_has\_what\$have\_had\_hadnt\_hasnt\_h  
 d\_hadnt\_hasnt\_havent\_has\_what\$have\_ha  
 dnt\_hasnt\_havent\_has\_what\$have\_had\_ha  
 e\_had\_hadnt\_hasnt\_havent\_has\_what\$ha  
 v\_ent\_has\_what\$have\_had\_hadnt\_hasnt\_ha  
 v\_had\_hadnt\_hasnt\_havent\_has\_what\$have\_  
 \_hadnt\_hasnt\_havent\_has\_what\$have\_had\_  
 has\_what\$have\_had\_hadnt\_hasnt\_havent\_  
 hasnt\_havent\_has\_what\$have\_had\_hadnt\_  
 hat\$have\_had\_hadnt\_hasnt\_havent\_has\_w  
 have\_had\_hadnt\_hasnt\_havent\_has\_what\$  
 havent\_has\_what\$have\_had\_hadnt\_hasnt\_  
 nt\_has\_what\$have\_had\_hadnt\_hasnt\_hav  
 nt\_hasnt\_havent\_has\_what\$have\_had\_had  
 nt\_havent\_has\_what\$have\_had\_hadnt\_ha  
 s\_s\_what\$have\_had\_hadnt\_hasnt\_havent\_h  
 asnt\_havent\_has\_what\$have\_had\_hadnt\_h  
 a\_t\$have\_had\_hadnt\_hasnt\_havent\_has\_w  
 h\_a\_t\_has\_what\$have\_had\_hadnt\_hasnt\_hav  
 n\_t\_hasnt\_havent\_has\_what\$have\_had\_had  
 nt\_havent\_has\_what\$have\_had\_hadnt\_ha  
 s\_n\_t\_havent\_has\_what\$have\_had\_hadnt\_h  
 a\_n\_v\_e\_had\_hadnt\_hasnt\_havent\_has\_what\$  
 h\_a\_v\_e\_had\_hadnt\_hasnt\_havent\_ha  
 \_what\$have\_had\_hadnt\_hasnt\_ha  
 what\$have\_had\_hadnt\_hasnt\_havent\_has\_

$T =$  have\_had\_hadnt\_hasnt\_havent\_has\_what\$

$B =$  tedttts h h h h h h h a a v v \_ \_ \_ \_ w \$ \_ \_ eds a a a n n n a a \_

$MTF(B) =$  8 5 5 2 0 0 8 7 0 0 0 0 0 0 7 0 9 0 8 0 0 0 10 9 2 9 9 8 7 0 0 10 0 0 1 0 5

## A Bigger Example

For  $T$  some English text,  
 $MTF(B)$  has typically  
around 50% zeroes!

[illegible]

have\_had\_hadnt\_hasnt\_havent\_has\_what  
 \_had\_hadnt\_hasnt\_havent\_has\_what\$have  
 \_hadnt\_hasnt\_havent\_has\_what\$have\_had  
 \_has\_what\$have\_had\_hadnt\_hasnt\_havent  
 \_hasnt\_havent\_has\_what\$have\_had\_hadnt  
 \_havent\_has\_what\$have\_had\_hadnt\_hasnt  
 \_what\$have\_had\_hadnt\_hasnt\_havent\_ha  
 \_ad\_hadnt\_hasnt\_havent\_has\_what\$have\_h  
 \_adnt\_hasnt\_havent\_has\_what\$have\_had\_h  
 \_as\_what\$have\_had\_hadnt\_hasnt\_havent\_h  
 \_asnt\_havent\_has\_what\$have\_had\_hadnt\_h  
 \_at\$have\_had\_hadnt\_hasnt\_havent\_has\_w  
 \_ave\_had\_hadnt\_hasnt\_havent\_has\_what\$  
 \_avent\_has\_what\$have\_had\_hadnt\_hasnt\_h  
 \_d\_hadnt\_hasnt\_havent\_has\_what\$have\_h  
 \_adnt\_hasnt\_havent\_has\_what\$have\_had\_h  
 \_e\_had\_hadnt\_hasnt\_havent\_has\_what\$ha  
 \_vent\_has\_what\$have\_had\_hadnt\_hasnt\_ha  
 \_vad\_hadnt\_hasnt\_havent\_has\_what\$have  
 \_hadnt\_hasnt\_havent\_has\_what\$have\_had  
 \_has\_what\$have\_had\_hadnt\_hasnt\_havent  
 \_hasnt\_havent\_has\_what\$have\_had\_hadnt  
 \_hat\$have\_had\_hadnt\_hasnt\_havent\_has\_w  
 \_have\_had\_hadnt\_hasnt\_havent\_has\_what\$  
 \_havent\_has\_what\$have\_had\_hadnt\_hasnt  
 \_nt\_has\_what\$have\_had\_hadnt\_hasnt\_have  
 \_nt\_hasnt\_havent\_has\_what\$have\_had\_ha  
 \_dnt\_havent\_has\_what\$have\_had\_hadnt\_ha  
 \_s\_s\_what\$have\_had\_hadnt\_hasnt\_havent\_ha  
 \_snt\_havent\_has\_what\$have\_had\_hadnt\_ha  
 \_t\$have\_had\_hadnt\_hasnt\_havent\_has\_w  
 \_t\_has\_what\$have\_had\_hadnt\_hasnt\_have  
 \_nt\_hasnt\_havent\_has\_what\$have\_had\_had  
 \_nt\_havent\_has\_what\$have\_had\_hadnt\_ha  
 \_ve\_had\_hadnt\_hasnt\_havent\_has\_what\$ha  
 \_vent\_has\_what\$have\_had\_hadnt\_hasnt\_ha  
 \_what\$have\_had\_hadnt\_hasnt\_havent\_ha

$T =$  have, had, hadn't, hasn't, haven't, has, what,

$B = \text{ t e d t t t s h h h h h h a a v v }_{\text{uuuu}} \text{ w \$ }_{\text{uu}} \text{ e d s a a a n n n a a }_{\text{uu}}$

$$MTF(B) = 8\ 5\ 5\ 2\ 0\ 0\ 8\ 7\ 0\ 0\ 0\ 0\ 0\ 0\ 7\ 0\ 9\ 0\ 8\ 0\ 0\ 0\ 10\ 9\ 2\ 9\ 9\ 8\ 7\ 0\ 0\ 10\ 0\ 0\ 1\ 0\ 5$$

## Clicker Question

Consider  $T = \text{have\_had\_hadnt\_hasnt\_havent\_has\_what\$}$ .  
The BWT is  $B = \text{tedtttshhhhhhhaavv\_w\$\_edsaaannnaa\_}$ .  
How can we explain the long run of hs in  $B$ ?



- A** h is the most frequent character
- B** h always appears at the beginning of a word
- C** almost all words start with h
- D** h is always followed by a
- E** all as are preceded by h
- F** h is the 4th character in the alphabet



→ *sl.i.do/cs566*

## Clicker Question

Consider  $T = \text{have\_had\_hadnt\_hasnt\_havent\_has\_what\$}$ .  
The BWT is  $B = \text{tedtttshhhhhhhaavv\_w\$\_edsaaannnaa\_}$ .  
How can we explain the long run of  $h$ s in  $B$ ?



- ☐ ~~A  $h$  is the most frequent character~~
- ☐ ~~B  $h$  always appears at the beginning of a word~~
- ☐ ~~C almost all words start with  $h$~~
- ☐ ~~D  $h$  is always followed by  $a$~~
- ☒ E all  $as$  are preceded by  $h$  ✓
- ☐ ~~F  $h$  is the 4th character in the alphabet~~



→ [sli.do/cs566](https://sli.do/cs566)



# Run-length BWT Compression

- ▶ amazingly, just run-length compressing the BWT is already powerful!
- ▶  $r$  = number of runs in BWT

## Example:

$S = \text{alf\_eats\_alfalfa\$}$

$B = \text{asff\$f\_e\_lll\_\underline{aa}ata}$

$RL(B) = \begin{bmatrix} a \\ 1 \end{bmatrix} \begin{bmatrix} s \\ 1 \end{bmatrix} \begin{bmatrix} f \\ 2 \end{bmatrix} \begin{bmatrix} \$ \\ 1 \end{bmatrix} \begin{bmatrix} f \\ 1 \end{bmatrix} \begin{bmatrix} \_ \\ 1 \end{bmatrix} \begin{bmatrix} e \\ 1 \end{bmatrix} \begin{bmatrix} \_ \\ 1 \end{bmatrix} \begin{bmatrix} l \\ 3 \end{bmatrix} \begin{bmatrix} a \\ 3 \end{bmatrix} \begin{bmatrix} t \\ 1 \end{bmatrix} \begin{bmatrix} a \\ 1 \end{bmatrix}$

$\rightsquigarrow r = |RL(B)| = 12; \quad n = 17$

## Larger Example:

$S = \text{have\_had\_hadnt\_hasnt\_havent\_has\_what\$}$

$\rightsquigarrow r = 19; \quad n = 36$

- ▶ Indeed:  $r = O(z \log^2(n))$ ,  $z$  number of LZ77 phrases proven in 2019 (!)