
# 7 Compression

*28 November 2022*

Sebastian Wild

COMP526 (Fall 2022)
University of Liverpool
*version* **2022-11-28 10:29**

# Learning Outcomes

1. Understand the necessity for encodings and know *ASCII* and *UTF-8 character encodings*.
2. Understand (qualitatively) the *limits of compressibility*.
3. Know and understand the algorithms (encoding and decoding) for *Huffman codes*, *RLE*, *Elias codes*, *LZW*, *MTF*, and *BWT*, including their *properties* like running time complexity.
4. Select and *adapt* (slightly) a *compression* pipeline for specific type of data.

**Unit 7: *Compression***

# Outline

## 7 Compression

# 7.1 Context

# Overview

- Unit 4–6: How to *work* with strings
  - finding substrings
  - finding approximate matches
  - finding repeated parts
  - ...
  - assumed character array (random access)!

- Unit 7–8: How to *store/transmit* strings
  - computer memory: must be binary
  - how to compress strings (save space)
  - how to robustly transmit over noisy channels $\rightsquigarrow$ Unit 8

# Clicker Question

What compression methods do you know?

→ *sli.do/comp526*

# Terminology

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

# Terminology

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

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

- 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 —> Unit 8
  - 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*: $\frac{|C| \cdot \lg |\Sigma_C|}{|S| \cdot \lg |\Sigma_S|} \overset{\Sigma_C = \{0,1\}}{=} \frac{|C|}{|S| \cdot \lg |\Sigma_S|}$

  $< 1$ means successful compression

  $= 1$ means no compression

  $> 1$ means "compression" made it bigger!? (yes, that happens . . . )

# Clicker Question

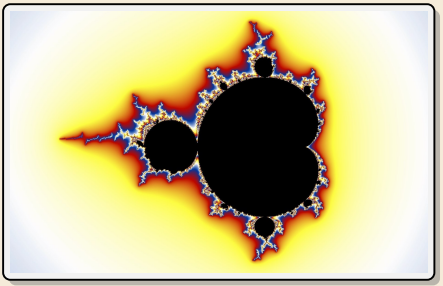
Do you know what uncomputable problems (halting problem, Post's correspondence problem, . . . ) are?

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

→ *sli.do/comp526*

# Limits of algorithmic compression

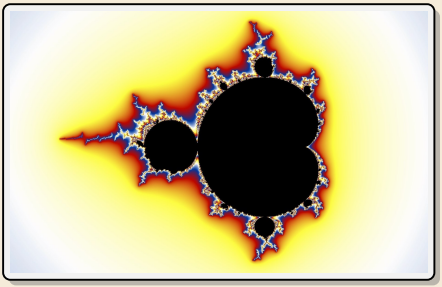
*Is this image compressible?*



# Limits of algorithmic compression

*Is this image compressible?*

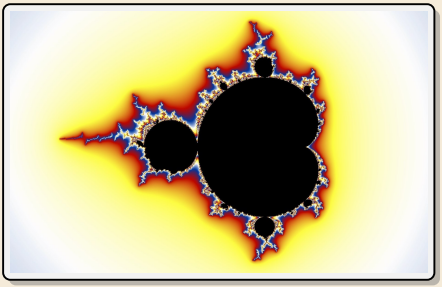
visualization of Mandelbrot set



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

# Limits of algorithmic compression

*Is this image compressible?*



visualization of Mandelbrot set

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

⇝ ***Kolmogorov complexity***

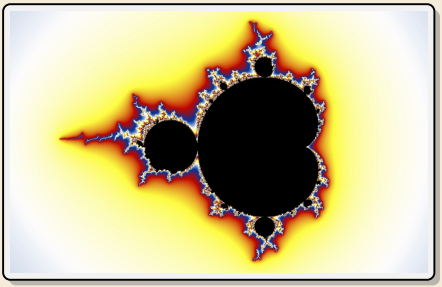
- $C$ = *any program* that outputs $S$

self-extracting archives!

- Kolmogorov complexity = length of smallest such program

# Limits of algorithmic compression

*Is this image compressible?*



visualization of Mandelbrot set

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

⇝ ***Kolmogorov complexity***

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

self-extracting archives!

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

# What makes data compressible?

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

  1. **uneven character frequencies**
     some characters occur more often than others → 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

$\rightsquigarrow$ encoding function $E : \Sigma_S \to \Sigma_C^\star$

  - typically, $|\Sigma_S| \gg |\Sigma_C|$, so need several bits per character
  - for $c \in \Sigma_S$, we call $E(c)$ the *codeword* of $c$

- **fixed-length code:** $|E(c)|$ is the same for all $c \in \Sigma_C$
- **variable-length code:** not all codewords of same length

# Fixed-length codes

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

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

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

# Fixed-length codes – Discussion

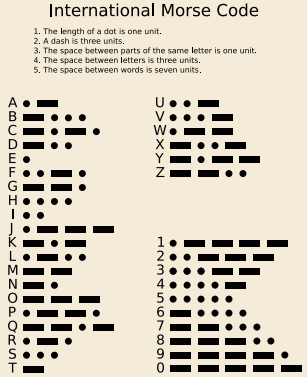

Encoding & Decoding as fast as it gets


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

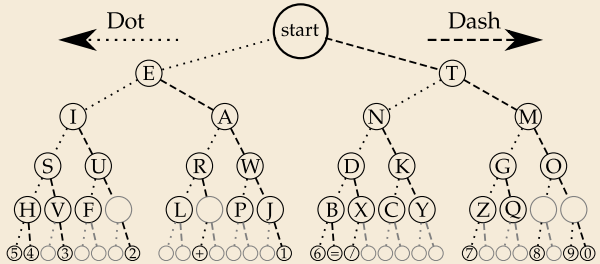

inflexible (how to support adding a new character?)

# Variable-length codes

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




https://commons.wikimedia.org/wiki/File:International_Morse_Code.svg




https://commons.wikimedia.org/wiki/File:Morse-code-tree.svg

# Clicker Question

How many characters are there in the alphabet of the coded text in Morse Code, i. e., what is $|\Sigma_C|$?

- **A** 1
- **B** 2
- **C** 3
- **D** 4
- **E** 26
- **F** 36
- **G** 256

→ *sli.do/comp526*

# Clicker Question

How many characters are there in the alphabet of the coded text in Morse Code, i. e., what is $|\Sigma_C|$?

- A ~~1~~
- B ~~2~~
- C 3 ✓
- D ~~4~~
- E ~~26~~
- F ~~36~~
- G ~~256~~

→ *sli.do/comp526*

# Variable-length codes – UTF-8

- Modern example: UTF-8 encoding of Unicode:

  default encoding for text-files, XML, HTML since 2009

  - Encodes any Unicode character (137 994 as of May 2019, and counting)
  - uses 1–4 bytes (codeword lengths: 8, 16, 24, or 32 bits)
  - Every ASCII character is encoded in 1 byte with leading bit `0`, followed by the 7 bits for ASCII
  - Non-ASCII charactters start with 1–4 `1`s indicating the total number of bytes, followed by a `0` and 3–5 bits.
    The remaining bytes each start with `10` followed by 6 bits.

| Char. number range (hexadecimal) | UTF-8 octet sequence (binary) |
|---|---|
| `0000 0000 – 0000 007F` | `0xxxxxxx` |
| `0000 0080 – 0000 07FF` | `110xxxxx 10xxxxxx` |
| `0000 0800 – 0000 FFFF` | `1110xxxx 10xxxxxx 10xxxxxx` |
| `0001 0000 – 0010 FFFF` | `11110xxx 10xxxxxx 10xxxxxx 10xxxxxx` |

For English text, most characters use only 8 bit,
but we can include any Unicode character, as well.

# Pitfall in variable-length codes

- Suppose we have the following code:

| $c$ | a | n | b | s |
|---|---|---|---|---|
| $E(c)$ | 0 | 10 | 110 | 100 |

- Happily encode text $S =$ `banana` with the coded text $C =$ `1100100100`

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 $S =$ `banana` with the coded text $C =$ `1100100100`
  b a n a n a

- $C =$ `1100100100` decodes **both** to `banana` and to `bass`: `1100100100`
  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 $S =$ banana with the coded text $C =$ 1100100100
  b a n a n a

- $C =$ 1100100100 decodes **both** to banana and to bass: 1100100100
  b a s s

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

but how should we have known?

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

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

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

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

# Code tries

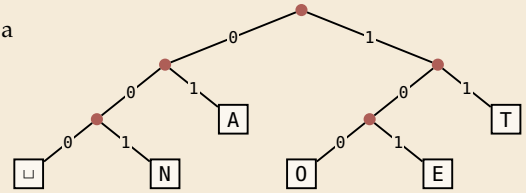
- From now on only consider prefix-free codes $E$:
  $E(c)$ is not a prefix of $E(c')$ for any $c, c' \in \Sigma_S$.

- **Example:**

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

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

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




- Encode `AN␣ANT`
- Decode `111000001010111`

# Code tries

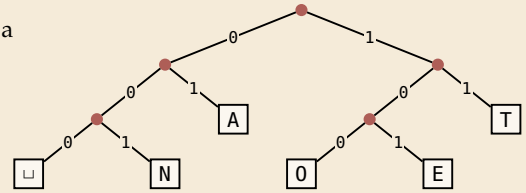
- From now on only consider prefix-free codes $E$:
  $E(c)$ is not a prefix of $E(c')$ for any $c, c' \in \Sigma_S$.

- **Example:**

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

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

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




- Encode `AN␣ANT` → `010010000100111`
- Decode `111000001010111` → `TO␣EAT`

# Who decodes the decoder?

- Depending on the application, we have to **store/transmit** the **used code**!
- We distinguish:
  - **fixed coding:** code agreed upon in advance, not transmitted (e. g., Morse, UTF-8)
  - **static coding:** code depends on message, but stays same for entire message; it must be transmitted (e. g., Huffman codes $\rightarrow$ next)
  - **adaptive coding:** code depends on message and changes during encoding; implicitly stored withing the message (e. g., LZW $\rightarrow$ below)

# 7.3 Huffman Codes

# Character frequencies

- **Goal:** Find character encoding that produces short coded text
- Convention here: fix $\Sigma_C = \{0, 1\}$ (binary codes), abbreviate $\Sigma = \Sigma_S$,
- **Observation:** Some letters occur more often than others.

**Typical English prose:**

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

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

# Huffman coding

e. g. frequencies / probabilities

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

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

# Huffman coding

e. g. frequencies / probabilities

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

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

- If we use $w(c) = \#\text{occurrences of } c \text{ in } S$,
  this is the character encoding with smallest possible $|C|$

  $\rightsquigarrow$ best possible character-wise encoding

- Quite ambitious! *Is this efficiently possible?*

# Huffman's algorithm

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

**Huffman's algorithm:**

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

# Huffman's algorithm

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

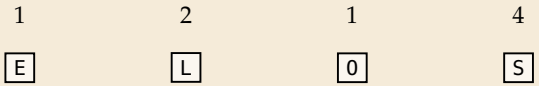
**Huffman's algorithm:**

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

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

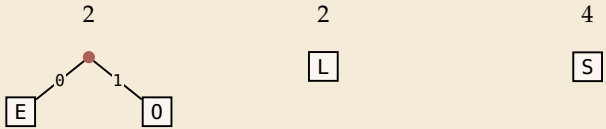
# Huffman's algorithm – Example

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



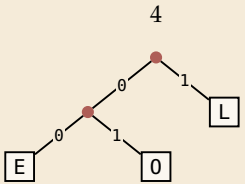
# Huffman's algorithm – Example

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



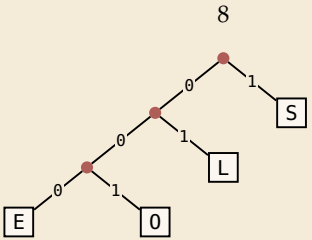
# Huffman's algorithm – Example

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



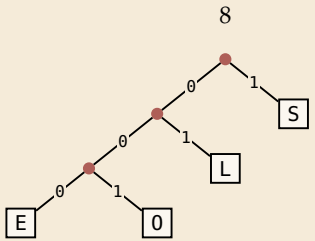

# Huffman's algorithm – Example

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



# Huffman's algorithm – Example

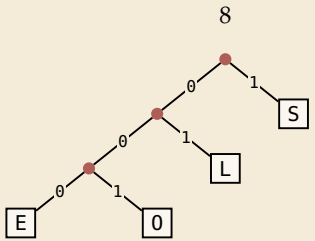
- Example text: $S = \texttt{LOSSLESS}$ $\rightsquigarrow$ $\Sigma_S = \{\texttt{E}, \texttt{L}, \texttt{O}, \texttt{S}\}$
- Character frequencies: $\texttt{E} : 1$, $\texttt{L} : 2$, $\texttt{O} : 1$, $\texttt{S} : 4$




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

# Huffman's algorithm – Example

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




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

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

# Huffman tree – tie breaking

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

- For COMP 526: always use the following rule:

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

# Encoding with Huffman code

- The overall encoding procedure is as follows:
  - Pass 1: Count character frequencies in $S$
  - Construct Huffman code $E$ (as above)
  - Store the Huffman code in $C$ (details omitted)
  - Pass 2: Encode each character in $S$ using $E$ and append result to $C$

- Decoding works as follows:
  - Decode the Huffman code $E$ from $C$. (details omitted)
  - Decode $S$ character by character from $C$ using the code trie.

- Note: Decoding is much simpler/faster!

# Huffman code – Optimality

**Theorem 7.1 (Optimality of Huffman's Algorithm)**
Given $\Sigma$ and $w : \Sigma \to \mathbb{R}_{\geq 0}$, Huffman's Algorithm computes codewords $E : \Sigma \to \{0,1\}^\star$ with minimal expected codeword length $\ell(E) = \sum_{c \in \Sigma} w(c) \cdot |E(c)|$ among all prefix-free codes for $\Sigma$. ◀

# Huffman code – Optimality

## Theorem 7.1 (Optimality of Huffman's Algorithm)

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

*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 $\Sigma' = \Sigma \setminus \{\texttt{a}, \texttt{b}\} \cup \{\boxed{\texttt{ab}}\}$ yields optimal code for $\Sigma$ by replacing leaf $\boxed{\texttt{ab}}$ 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 7.2 (Entropy)

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

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

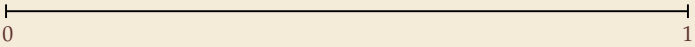
# Entropy

## Definition 7.2 (Entropy)

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

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

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



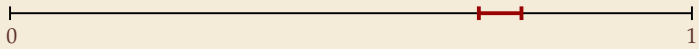
# Entropy

## Definition 7.2 (Entropy)

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

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

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



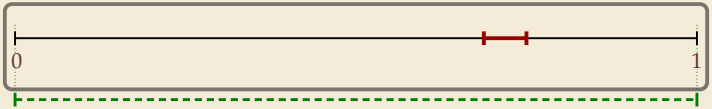
# Entropy

## Definition 7.2 (Entropy)

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

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

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



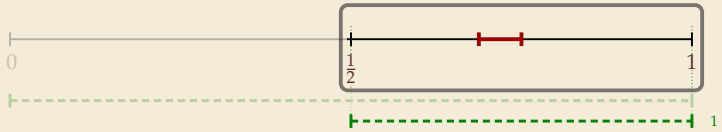
# Entropy

## Definition 7.2 (Entropy)

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

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

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



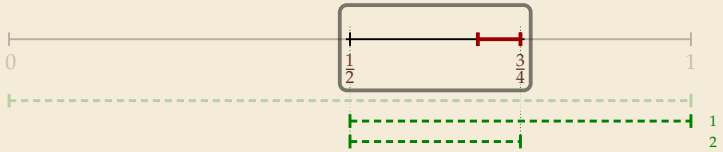
# Entropy

## Definition 7.2 (Entropy)

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

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

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



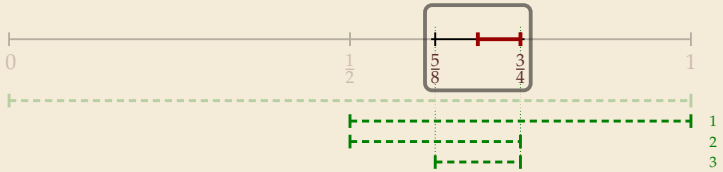
# Entropy

## Definition 7.2 (Entropy)

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

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

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



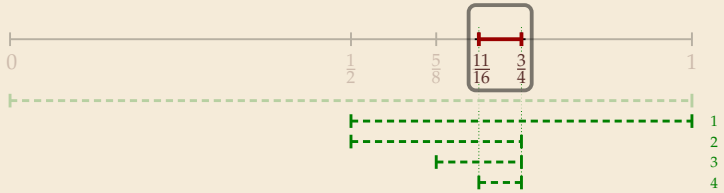
# Entropy

## Definition 7.2 (Entropy)

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

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

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



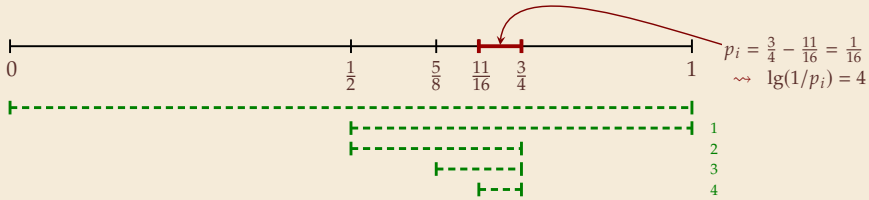
# Entropy

## Definition 7.2 (Entropy)

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

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

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



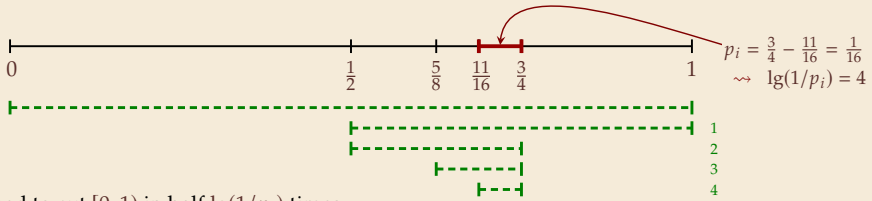
# Entropy

## Definition 7.2 (Entropy)

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

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

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




  - $\rightsquigarrow$ Need to cut $[0,1)$ in half $\lg(1/p_i)$ times
  - more precisely: the expected number of bits (Yes/No questions) required to nail down the random value

# Entropy and Huffman codes

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

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

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

# Entropy and Huffman codes

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

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

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

**Theorem 7.3 (Entropy bounds for Huffman codes)**

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

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

where $\mathcal{H} = \mathcal{H}\left(\frac{w(a_1)}{W}, \ldots, \frac{w(a_\sigma)}{W}\right)$ and $W = w(a_1) + \cdots + w(a_\sigma)$. ◀

# Entropy and Huffman codes

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

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

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

## Theorem 7.3 (Entropy bounds for Huffman codes)

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

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

*Proof sketch:*

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

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

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

  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 [2]

*Proof sketch (continued):*

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

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

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

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

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

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

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

# Clicker Question

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

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

→ *sli.do/comp526*

# Clicker Question

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

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

→ *sli.do/comp526*

# Huffman coding – Discussion

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

# Huffman coding – Discussion

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


optimal prefix-free character encoding


very fast decoding


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


have to store code alongside with coded text

# Part II

*Compressing repetitive texts*

# Beyond Character Encoding

- Many “natural” texts show repetitive redundancy

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

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

# Beyond Character Encoding

- Many “natural” texts show repetitive redundancy

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

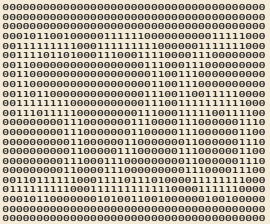
- character-by-character encoding will **not** capture such repetitions
  - ⇝ Huffman won’t compression this very much

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

# 7.5 Run-Length Encoding

# Run-Length encoding

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

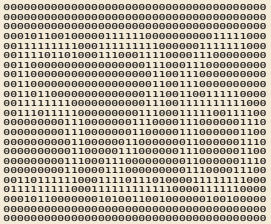


same character repeated

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

# Run-Length encoding

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



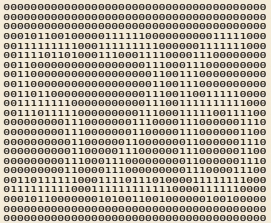
same character repeated

- 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: $S =$ `00000` `111` `0000`

# Run-Length encoding

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



same character repeated

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

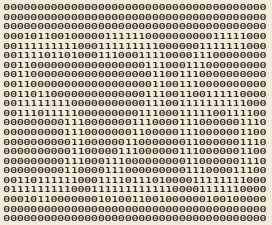
⇝ **run-length encoding (RLE)**:
use runs as phrases: $S =$ `00000` `111` `0000`

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

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

# Run-Length encoding

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



same character repeated

- 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: $S =$ `00000` `111` `0000`

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

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

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

# Clicker Question

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

→ *sli.do/comp526*

# Elias codes

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

# Elias codes

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

- But that's simple! Just use ***unary*** **encoding**!
  $7 \mapsto$ `00000001` $3 \mapsto$ `0001` $0 \mapsto$ `1` $30 \mapsto$ `000000000000000000000000000001`

# Elias codes

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

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

  $7 \mapsto$ `00000001` $3 \mapsto$ `0001` $0 \mapsto$ `1` $30 \mapsto$ `000000000000000000000000000001`

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

# Elias codes

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

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

  $7 \mapsto$ `00000001` $3 \mapsto$ `0001` $0 \mapsto$ `1` $30 \mapsto$ `000000000000000000000000000001`

  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 $\ell \geq 1$, so store $\ell - 1$ instead
    - binary representation always starts with `1` $\rightsquigarrow$ don't need terminating `1` in unary
  - $\rightsquigarrow$ Elias gamma code = $\ell - 1$ zeros, followed by binary representation

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

# Clicker Question

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

`000110111011100110.`

→ *sli.do/comp526*

# Run-length encoding – Examples

- Encoding:
  $S$ = 1111111001000000000000000000011111111111

  $C$ = 1

- Decoding:
  $C$ = 00001101001001010

  $S$ =

# Run-length encoding – Examples

- Encoding:
  $S =$ 1111111001000000000000000000011111111111
  $k = 7$
  $C =$ 100111

- Decoding:
  $C =$ 00001101001001010

  $S =$

# Run-length encoding – Examples

- Encoding:
  $S =$ `1111111001000000000000000000011111111111`
  $k = 2$
  $C =$ `100111010`

- Decoding:
  $C =$ `00001101001001010`

  $S =$

# Run-length encoding – Examples

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

- Decoding:
  $C$ = `00001101001001010`

  $S$ =

# Run-length encoding – Examples

- Encoding:
  $S$ = 111111100100000000000000000000011111111111
  $k$ = 20
  $C$ = 10011101010000010100

- Decoding:
  $C$ = 00001101001001010

  $S$ =

# Run-length encoding – Examples

- Encoding:
  $S$ = 11111110010000000000000000000011111111111
  $k$ = 11
  $C$ = 10011101010000101000001011

- Decoding:
  $C$ = 00001101001001010

  $S$ =

# Run-length encoding – Examples

- Encoding:
  $S$ = `11111110010000000000000000000011111111111`

  $C$ = `10011101010000101000001011`

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

- Decoding:
  $C$ = `00001101001001010`

  $S$ =

# Run-length encoding – Examples

- Encoding:
  $S$ = `11111110010000000000000000000011111111111`

  $C$ = `10011101010000101000001011`

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

- Decoding:
  $C$ = `00001101001001010`

  $S$ =

# Run-length encoding – Examples

- Encoding:
  $S =$ `11111110010000000000000000000011111111111`

  $C =$ `10011101010000101000001011`

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

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

  $S =$

# Run-length encoding – Examples

- Encoding:
  $S =$ `11111110010000000000000000000011111111111`

  $C =$ `10011101010000101000001011`

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

- Decoding:
  $C =$ `0000`1101001001010
  $b = 0$
  $\ell = 3 + 1$

  $S =$

# Run-length encoding – Examples

- Encoding:
  $S =$ `11111110010000000000000000000011111111111`

  $C =$ `10011101010000101000001011`

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

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

# Run-length encoding – Examples

- Encoding:
  $S =$ `11111110010000000000000000000011111111111`

  $C =$ `100111010100001010000001011`

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

- Decoding:
  $C =$ `00001101001001010`
  $b = 1$
  $\ell = 2 + 1$
  $k =$
  $S =$ `0000000000000`

# Run-length encoding – Examples

- Encoding:
  $S =$ `11111110010000000000000000000011111111111`

  $C =$ `100111010100001010000001011`

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

- Decoding:
  $C =$ `0000110100100 1010`
  $b = 1$
  $\ell = 2 + 1$
  $k = 4$
  $S =$ `00000000000001111`

# Run-length encoding – Examples

- Encoding:
  $S =$ `11111110010000000000000000000011111111111`

  $C =$ `10011101010000101000001011`

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

- Decoding:
  $C =$ `0000110100100101`0`10`
  $b = 0$
  $\ell = 0 + 1$
  $k =$
  $S =$ `00000000000001111`

# Run-length encoding – Examples

- Encoding:
  $S =$ `11111110010000000000000000000011111111111`

  $C =$ `10011101010000101000001011`

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

- Decoding:
  $C =$ `0000110100100101`0
  $b = 0$
  $\ell = 0 + 1$
  $k = 1$
  $S =$ `000000000000011110`

# Run-length encoding – Examples

- Encoding:
  $S$ = `11111110010000000000000000000011111111111`

  $C$ = `10011101010000101000001011`

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

- Decoding:
  $C$ = `0000110100100100010`
  $b = 1$
  $\ell = 1 + 1$
  $k =$
  $S$ = `000000000000011110`

# Run-length encoding – Examples

- Encoding:
  $S$ = `11111110010000000000000000000011111111111`

  $C$ = `10011101010000101000001011`

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

- Decoding:
  $C$ = `00001101001001010`
  $b = 1$
  $\ell = 1 + 1$
  $k = 2$
  $S$ = `00000000000001111011`

# Run-length encoding – Discussion

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

# Run-length encoding – Discussion

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


fairly simple and fast

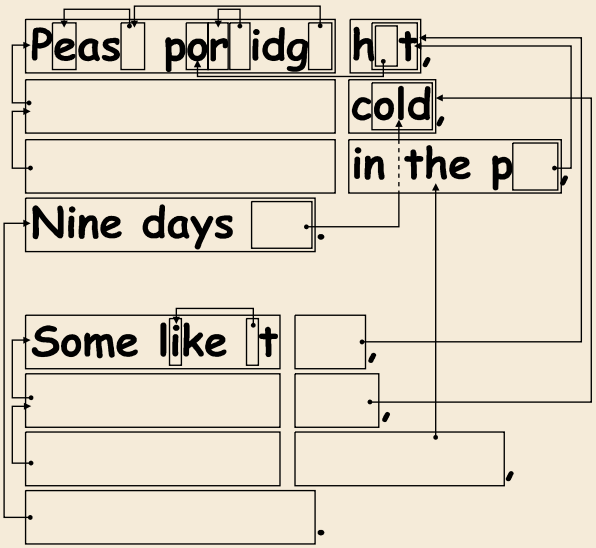

can compress $n$ bits to $\Theta(\log n)$!
for extreme case of constant number of runs


negligible compression for many common types of data

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

# 7.6 Lempel-Ziv-Welch

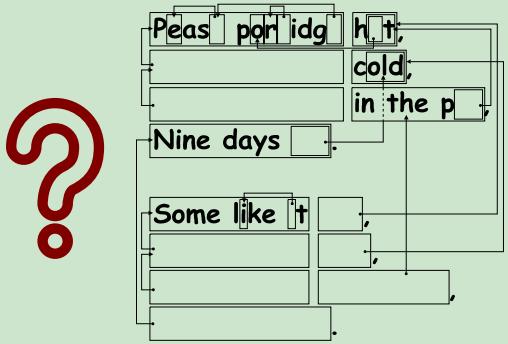
# Warmup




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

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

# Clicker Question




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

→ *sli.do/comp526*

# Lempel-Ziv Compression

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

# Lempel-Ziv Compression

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

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

# Lempel-Ziv Compression

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

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

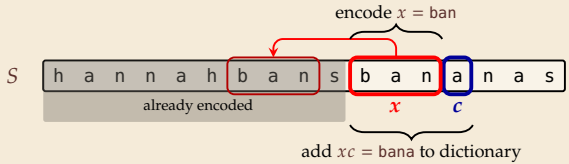
  - Variants of Lempel-Ziv compression
    - "LZ77" Original version ("sliding window")
      Derivatives: LZSS, LZFG, LZRW, LZP, DEFLATE, ...
      DEFLATE used in (`pk`)`zip`, `gzip`, `PNG`
    - "LZ78" Second (slightly improved) version
      Derivatives: LZW, LZMW, LZAP, LZY, ...
      LZW used in `compress`, `GIF`

# Lempel-Ziv-Welch

- here: ***Lempel-Ziv-Welch (LZW)*** (arguably the "cleanest" variant of Lempel-Ziv)
- ***variable-to-fixed* encoding**
  - all codewords have $k$ bits (typical: $k = 12$) $\rightsquigarrow$ 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**: `YO!␣YOU!␣YOUR␣YOYO!`

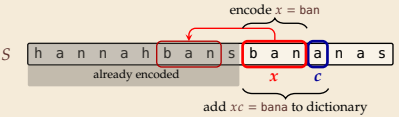
$\Sigma_S$ = ASCII character set (0–127)

$C$ =

$D$ =

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

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



# LZW encoding – Example

**Input**: YO!␣YOU!␣YOUR␣YOYO!

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

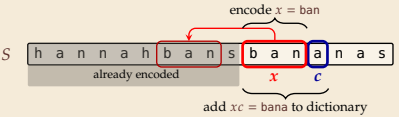
Y

$C = 89$

$D =$

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

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



# LZW encoding – Example

**Input**: YO!␣YOU!␣YOUR␣YOYO!

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

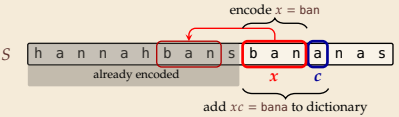
Y

$C$ = 89

$D$ =

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

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



# LZW encoding – Example

**Input**: Y0!␣YOU!␣YOUR␣YOYO!

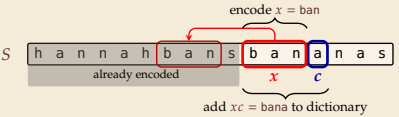
$\Sigma_S$ = ASCII character set (0–127)

| | 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!

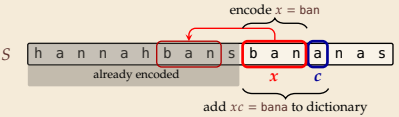
$\Sigma_S$ = ASCII character set (0–127)

| | 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**: `YO!␣YOU!␣YOUR␣YOYO!`

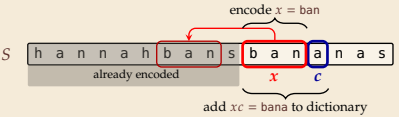
$\Sigma_S$ = ASCII character set (0–127)

| | | | |
|---|---|---|---|
| | Y | O | ! |
| $C$ = | 89 | 79 | 33 |

$D$ =

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

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



# LZW encoding – Example

**Input**: `YO!␣YOU!␣YOUR␣YOYO!`

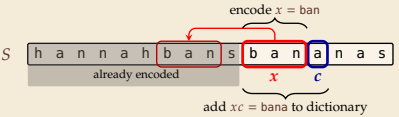
$\Sigma_S$ = ASCII character set (0–127)

| | Y | O | ! |
|---|---|---|---|
| $C$ = | 89 | 79 | 33 |

$D$ =

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

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



# LZW encoding – Example

**Input**: `YO!␣YOU!␣YOUR␣YOYO!`

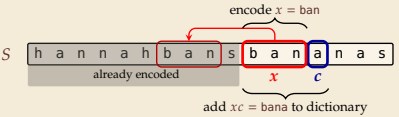
$\Sigma_S$ = ASCII character set (0–127)

| | Y | O | ! | ␣ |
|---|---|---|---|---|
| $C$ = | 89 | 79 | 33 | 32 |

$D$ =

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

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



# LZW encoding – Example

**Input**: `YO!␣YOU!␣YOUR␣YOYO!`

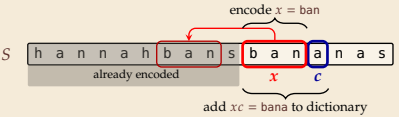
$\Sigma_S$ = ASCII character set (0–127)

| | Y | O | ! | ␣ |
|---|---|---|---|---|
| $C$ = | 89 | 79 | 33 | 32 |

$D$ =

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

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



# LZW encoding – Example

**Input**: YO!␣YOU!␣YOUR␣YOYO!

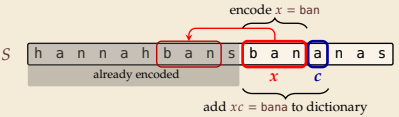
$\Sigma_S$ = ASCII character set (0–127)

| | Y | O | ! | ␣ | YO |
|---|---|---|---|---|---|
| $C$ = | 89 | 79 | 33 | 32 | 128 |

$D$ =

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

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



# LZW encoding – Example

**Input**: YO!␣YOU!␣YOUR␣YOYO!

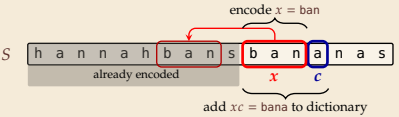
$\Sigma_S$ = ASCII character set (0–127)

| | Y | O | ! | ␣ | YO |
|---|---|---|---|---|---|
| $C$ = | 89 | 79 | 33 | 32 | 128 |

$D$ =

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

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



# LZW encoding – Example

**Input**: `YO!␣YOU!␣YOUR␣YOYO!`

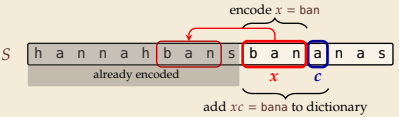
$\Sigma_S$ = ASCII character set (0–127)

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

$D$ =

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

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



# LZW encoding – Example

**Input**: YO!␣YOU!␣YOUR␣YOYO!

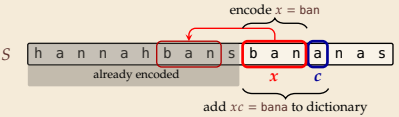
$\Sigma_S$ = ASCII character set (0–127)

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

$D$ =

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

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



# LZW encoding – Example

**Input**: `YO!␣YOU!␣YOUR␣YOYO!`

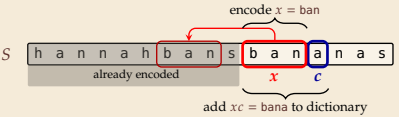
$\Sigma_S$ = ASCII character set (0–127)

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

$D$ =

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

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



# LZW encoding – Example

**Input**: `YO!␣YOU!␣YOUR␣YOYO!`

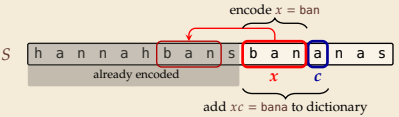
$\Sigma_S$ = ASCII character set (0–127)

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

$D$ =

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

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



# LZW encoding – Example

**Input**: YO!␣YOU!␣YOUR␣YOYO!

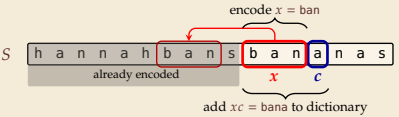
$\Sigma_S$ = ASCII character set (0–127)

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

$D$ =

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

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



# LZW encoding – Example

**Input**: YO!␣YOU!␣YOUR␣YOYO!

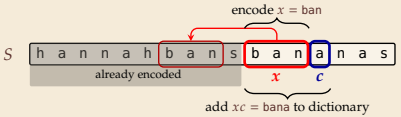
$\Sigma_S$ = ASCII character set (0–127)

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

$D$ =

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

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



# LZW encoding – Example

**Input**: `YO!␣YOU!␣YOUR␣YOYO!`

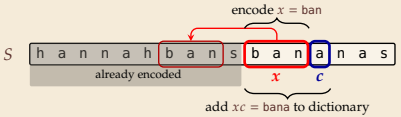
$\Sigma_S$ = ASCII character set (0–127)

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

$D$ =

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

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



# LZW encoding – Example

**Input**: YO!␣YOU!␣YOUR␣YOYO!

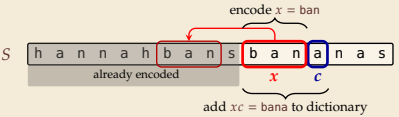
$\Sigma_S$ = ASCII character set (0–127)

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

$D$ =

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

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



# LZW encoding – Example

**Input**: YO!␣YOU!␣YOUR␣YOYO!

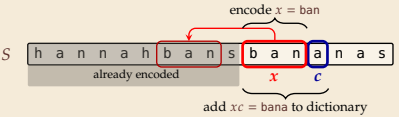
$\Sigma_S$ = ASCII character set (0–127)

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

$D$ =

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

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



# LZW encoding – Example

**Input**: YO!␣YOU!␣YOUR␣YOYO!

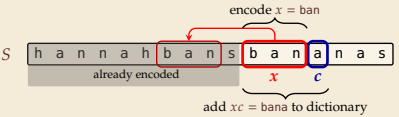
$\Sigma_S$ = ASCII character set (0–127)

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

$D$ =

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

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



# LZW encoding – Example

**Input**: YO!␣YOU!␣YOUR␣YOYO!

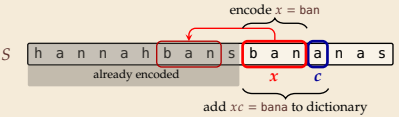
$\Sigma_S$ = ASCII character set (0–127)

| | Y | O | ! | ␣ | YO | U | !␣ | YOU | R | ␣Y | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $C$ = | 89 | 79 | 33 | 32 | 128 | 85 | 130 | 132 | 82 | 131 | 79 |

$D$ =

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

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



# LZW encoding – Example

**Input**: YO!␣YOU!␣YOUR␣YOYO!

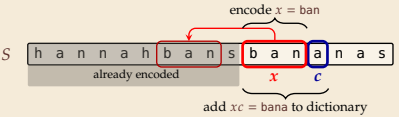
$\Sigma_S$ = ASCII character set (0–127)

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

$D$ =

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

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



# LZW encoding – Example

**Input**: YO!␣YOU!␣YOUR␣YOYO!

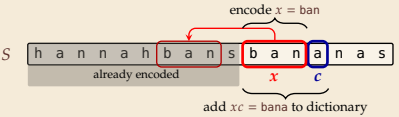
$\Sigma_S$ = ASCII character set (0–127)

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

$D$ =

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

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



# LZW encoding – Example

**Input**: YO!␣YOU!␣YOUR␣YOYO!

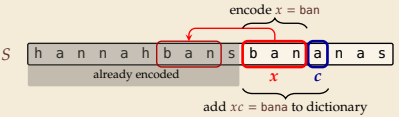
$\Sigma_S$ = ASCII character set (0–127)

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

$D$ =

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

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

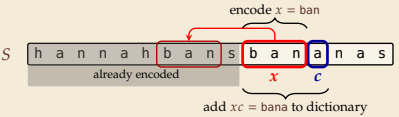


# LZW encoding – Example

**Input**: `YO!␣YOU!␣YOUR␣YOYO!`

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

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




$D$ =

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

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

# LZW encoding – Code

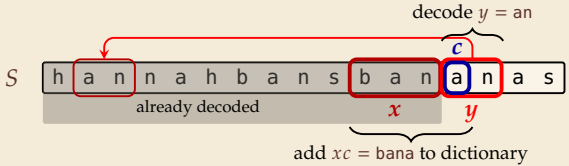
```
1  procedure LZWencode(S[0..n))
2      x := ε // previous phrase, initially empty
3      C := ε // output, initially empty
4      D := dictionary, initialized with codes for c ∈ Σ_S // stored as trie
5      k := |Σ_S| // next free codeword
6      for i := 0, ..., n − 1 do
7          c := S[i]
8 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

# 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 |
| 32 | ␣ | 130 | N␣ | 78, ␣ |
| | | | | |
| | | | | |
| | | | | |

# 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 |
| | | | | |
| | | | | |

# 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 |
| | | | | |

# LZW decoding – Example

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

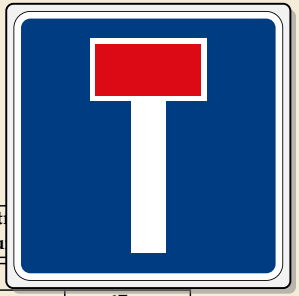
$D =$

| 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 |
| 133 | ??? | 133 | | |

# 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 # | St (hu | |
|---|---|---|---|---|
| 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 | ??? | 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

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

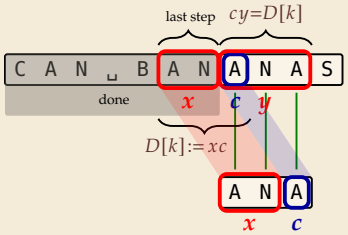
# LZW decoding – Bootstrapping

- 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 $D[k] := \boldsymbol{xc}$
3. next phrase $y$ equals $D[k]$
   ⇝ $D[k] = \boldsymbol{xc} = \boldsymbol{x} \cdot \boldsymbol{x[0]}$ (all known)

# LZW decoding – Code

```
1  procedure LZWdecode(C[0..m))
2      D := dictionary [0..2^d) → Σ_S^+, initialized with codes for c ∈ Σ_S // stored as array
3      k := |Σ_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, ..., m − 1 do
8          x := y // remember last decoded phrase
9          q := C[j] // next codeword
10         if q == k then
11             y := x · x[0] // bootstrap case
12         else
13             y := D[q]
14         S := S · y // append decoded phrase
15         D[k] := x · y[0] // store new phrase
16         k := k + 1
17     end for
18     return S
```

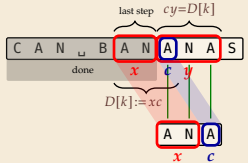
# LZW decoding – Example continued

- 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)

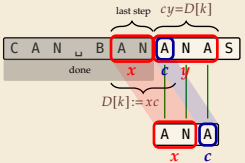
# LZW decoding – Example continued

- 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 $S = \mathtt{a}^n$, a run of $n$ copies of as?

- **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/comp526*

# Clicker Question

How many phrases will LZW create on $S = \mathtt{a}^n$, a run of $n$ copies of as?

- **A** ~~$n$~~
- **B** ~~$n/2$~~
- **C** ~~$n/4$~~
- **D** ~~$\Theta(n/\log n)$~~
- **E** $\Theta(\sqrt{n})$ ✓
- **F** ~~$\Theta(\log n)$~~
- **G** ~~$\Theta(\log \log n)$~~
- **H** ~~2~~
- **I** ~~1~~

→ *sli.do/comp526*

# 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)

# 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 encopding | 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 Transforms*

# Text transformations

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

# Text transformations

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

  ⇝ use as pre-/postprocessing in 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 ones cheaper

# 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)
  - $\rightsquigarrow$ list "learns" probabilities of access to objects makes access to frequently requested ones 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
  - $\rightsquigarrow$ codewords are integers, i. e., $\Sigma_C = [0..\sigma)$

# 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)
  - $\rightsquigarrow$ list "learns" probabilities of access to objects makes access to frequently requested ones 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
  - $\rightsquigarrow$ codewords are integers, i. e., $\Sigma_C = [0..\sigma)$

$\rightsquigarrow$ clusters of few characters $\rightsquigarrow$ many small numbers

# Clicker Question

Assume a MTF list currently contains the items `X Y Z A B C`, and we now access `A`. What is the list content after the MTF rule has been applied?

→ *sli.do/comp526*

# MTF – Code

- **Transform (encode):**

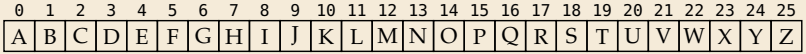
```
1  procedure MTF–encode(S[0..n))
2      L := list containing Σ_S (sorted order)
3      C := ε
4      for i := 0, ..., n − 1 do
5          c := S[i]
6          p := position of c in L
7          C := C · p
8          Move c to front of L
9      end for
10     return C
```

- **Inverse transform (decode):**

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

- Important: encoding and decoding produce same accesses to list

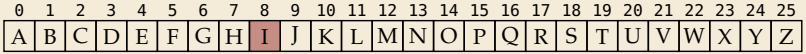
# MTF – Example




$S =$ INEFFICIENCIES

$C =$

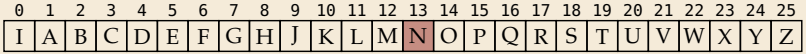
# MTF – Example




$S =$ INEFFICIENCIES

$C = 8$

# MTF – Example




$S =$ INEFFICIENCIES

$C =$ 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 |

$S =$ INEFFICIENCIES

$C =$ 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 |

$S =$ INEFFICIENCIES

$C =$ 8 13 6 7

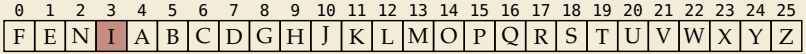
# MTF – Example

| 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 |

$S =$ INEFFICIENCIES

$C =$ 8 13 6 7 0

# MTF – Example




$S =$ INEFFICIENCIES

$C = 8\,13\,6\,7\,0\,3$

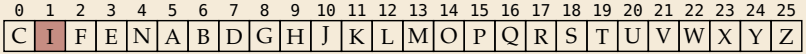
# MTF – Example

| 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 |

$S =$ INEFFICIENCIES

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

# MTF – Example




$S =$ INEFFICIENCIES

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

# 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 |

$S =$ `INEFFICIENCIES`

$C = 8\ 13\ 6\ 7\ 0\ 3\ 6\ 1\ 3\ 4\ 3\ 3\ 3\ 18$

- 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)

$\rightsquigarrow$ prime use as part of longer pipeline

- two simple ideas for encoding codewords:
  - Elias gamma code $\rightsquigarrow$ smaller numbers gets shorter codewords
    works well for text with small "local effective" alphabet
  - Huffman code (better compression, but need 2 passes)

- but: most effective after BWT ($\rightarrow$ next)

# 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 with MTF(!)

# 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 with MTF(!)

- Encoding algorithm needs **all** of $S$ (no streaming possible).
  - $\rightsquigarrow$ 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 with MTF(!)

- Encoding algorithm needs **all** of $S$ (no streaming possible).
  - $\rightsquigarrow$ 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
1191071 bible.txt.gz
 888604 bible.txt.7z
 845635 bible.txt.bz2
```

# BWT transform

- *cyclic shift* of a string:

$T$ = `time␣flies␣quickly␣`

`flies␣quickly␣time␣`


⇝ cyclic shift

# BWT transform

- *cyclic shift* of a string:
- add *end-of-word character* `$` to $S$ (as in Unit 6)

⇝ can recover original string

$T$ = `time␣flies␣quickly␣`


⇝ cyclic shift

`flies␣quickly␣time␣`

# BWT transform

- *cyclic shift* of a string:
- add *end-of-word character* `$` to $S$ (as in Unit 6)

⇝ can recover original string

$T$ = `time␣flies␣quickly␣`


⇝ cyclic shift

`flies␣quickly␣time␣`


- The Burrows-Wheeler Transform proceeds in three steps:
  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 transform – Example

$S$ = `alf␣eats␣alfalfa$`

1. Write all cyclic shifts

```
alf␣eats␣alfalfa$
lf␣eats␣alfalfa$a
f␣eats␣alfalfa$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

# BWT transform – Example

$S$ = `alf␣eats␣alfalfa$`

1. Write all cyclic shifts
2. Sort cyclic shifts

```
alf␣eats␣alfalfa$          $alf␣eats␣alfalfa
lf␣eats␣alfalfa$a          ␣alfalfa$alf␣eats
f␣eats␣alfalfa$al          ␣eats␣alfalfa$alf
␣eats␣alfalfa$alf          a$alf␣eats␣alfalf
eats␣alfalfa$alf␣          alf␣eats␣alfalfa$
ats␣alfalfa$alf␣e          alfa$alf␣eats␣alf
ts␣alfalfa$alf␣ea   ⇝      alfalfa$alf␣eats␣
s␣alfalfa$alf␣eat   sort   ats␣alfalfa$alf␣e
␣alfalfa$alf␣eats          eats␣alfalfa$alf␣
alfalfa$alf␣eats␣          f␣eats␣alfalfa$al
lfalfa$alf␣eats␣a          fa$alf␣eats␣alfal
falfa$alf␣eats␣al          falfa$alf␣eats␣al
alfa$alf␣eats␣alf          lf␣eats␣alfalfa$a
lfa$alf␣eats␣alfa          lfa$alf␣eats␣alfa
fa$alf␣eats␣alfal          lfalfa$alf␣eats␣a
a$alf␣eats␣alfalf          s␣alfalfa$alf␣eat
$alf␣eats␣alfalfa          ts␣alfalfa$alf␣ea
```

# BWT transform – Example

$S$ = `alf␣eats␣alfalfa$`

1. Write all cyclic shifts
2. Sort cyclic shifts
3. Extract last column

$B$ = `asff$f␣e␣lllaaata`

# Clicker Question

What is the relation between suffix array $L[0..n]$ and BWT $B[0..n]$ of a string $T[0..n)\$$?

- **A** $L$ can be very easily computed from $B$ and $T$
- **B** $B$ can be very easily computed from $L$ and $T$
- **C** Both A and B
- **D** Neither A nor B

→ *sli.do/comp526*

# Clicker Question

What is the relation between suffix array $L[0..n]$ and BWT $B[0..n]$ of a string $T[0..n)\$$?

- **A** ~~$L$ can be very easily computed from $B$ and $T$~~
- **B** $B$ can be very easily computed from $L$ and $T$ ✓
- **C** ~~Both A and B~~
- **D** ~~Neither A nor B~~

→ *sli.do/comp526*

# BWT – Implementation & Properties

**Compute BWT efficiently:**

- cyclic shifts $S$ $\widehat{=}$ suffixes of $S$
- BWT is essentially suffix sorting!
  - $B[i] = S[L[i]-1]$ ($L$ = suffix array!)
    (if $L[i] = 0$, $B[i] = \$$)
  - $\rightsquigarrow$ Can compute $B$ in $O(n)$ time

| | $r$ | $\downarrow$ | $L[r]$ |
|---|---|---|---|
| alf␣eats␣alfalfa$ | 0 | $alf␣eats␣alfalfa | 16 |
| lf␣eats␣alfalfa$a | 1 | ␣alfalfa$alf␣eats | 8 |
| f␣eats␣alfalfa$al | 2 | ␣eats␣alfalfa$alf | 3 |
| ␣eats␣alfalfa$alf | 3 | a$alf␣eats␣alfalf | 15 |
| eats␣alfalfa$alf␣ | 4 | alf␣eats␣alfalfa$ | 0 |
| ats␣alfalfa$alf␣e | 5 | alfa$alf␣eats␣alf | 12 |
| ts␣alfalfa$alf␣ea | 6 | alfalfa$alf␣eats␣ | 9 |
| s␣alfalfa$alf␣eat | 7 | ats␣alfalfa$alf␣e | 5 |
| ␣alfalfa$alf␣eats | 8 | eats␣alfalfa$alf␣ | 4 |
| alfalfa$alf␣eats␣ | 9 | f␣eats␣alfalfa$al | 2 |
| lfalfa$alf␣eats␣a | 10 | fa$alf␣eats␣alfal | 14 |
| falfa$alf␣eats␣al | 11 | falfa$alf␣eats␣al | 11 |
| alfa$alf␣eats␣alf | 12 | lf␣eats␣alfalfa$a | 1 |
| lfa$alf␣eats␣alfa | 13 | lfa$alf␣eats␣alfa | 13 |
| fa$alf␣eats␣alfal | 14 | lfalfa$alf␣eats␣a | 10 |
| a$alf␣eats␣alfalf | 15 | s␣alfalfa$alf␣eat | 7 |
| $alf␣eats␣alfalfa | 16 | ts␣alfalfa$alf␣ea | 6 |

# BWT – Implementation & Properties

**Compute BWT efficiently:**

- cyclic shifts $S$ $\widehat{=}$ suffixes of $S$
- BWT is essentially suffix sorting!
  - $B[i] = S[L[i] - 1]$ ($L$ = suffix array!)
    (if $L[i] = 0$, $B[i] =$ \$)
  - $\rightsquigarrow$ Can compute $B$ in $O(n)$ time

**Why does BWT help?**

- sorting groups characters *by what follows*
  - Example: `lf` always preceded by `a`

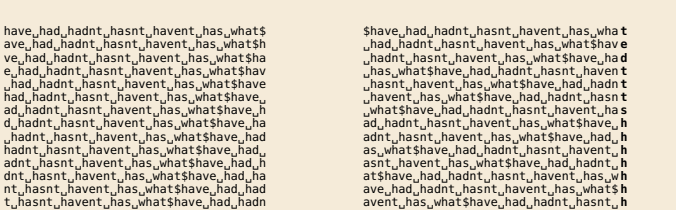
$\rightsquigarrow$ $B$ has local clusters of characters

- that makes MTF effective

- repeated substring in $S$ $\rightsquigarrow$ *runs* of characters in $B$
  - picked up by RLE

| | $r$ | | $\downarrow L[r]$ |
|---|---|---|---|
| alf␣eats␣alfalfa$ | 0 | $alf␣eats␣alfalfa | 16 |
| lf␣eats␣alfalfa$a | 1 | ␣alfalfa$alf␣eats | 8 |
| f␣eats␣alfalfa$al | 2 | ␣eats␣alfalfa$alf | 3 |
| ␣eats␣alfalfa$alf | 3 | a$alf␣eats␣alfalf | 15 |
| eats␣alfalfa$alf␣ | 4 | alf␣eats␣alfalfa$ | 0 |
| ats␣alfalfa$alf␣e | 5 | alfa$alf␣eats␣alf | 12 |
| ts␣alfalfa$alf␣ea | 6 | alfalfa$alf␣eats␣ | 9 |
| s␣alfalfa$alf␣eat | 7 | ats␣alfalfa$alf␣e | 5 |
| ␣alfalfa$alf␣eats | 8 | eats␣alfalfa$alf␣ | 4 |
| alfalfa$alf␣eats␣ | 9 | f␣eats␣alfalfa$al | 2 |
| lfalfa$alf␣eats␣a | 10 | fa$alf␣eats␣alfal | 14 |
| falfa$alf␣eats␣al | 11 | falfa$alf␣eats␣al | 11 |
| alfa$alf␣eats␣alf | 12 | lf␣eats␣alfalfa$a | 1 |
| lfa$alf␣eats␣alfa | 13 | lfa$alf␣eats␣alfa | 13 |
| fa$alf␣eats␣alfal | 14 | lfalfa$alf␣eats␣a | 10 |
| a$alf␣eats␣alfalf | 15 | s␣alfalfa$alf␣eat | 7 |
| $alf␣eats␣alfalfa | 16 | ts␣alfalfa$alf␣ea | 6 |

# Bigger Example




$T =$ h a v e ␣ h a d ␣ h a d n t ␣ h a s n t ␣ h a v e n t ␣ h a s ␣ w h a t $

$B =$ t e d t t t s h h h h h h h a a v v ␣ ␣ ␣ ␣ ␣ w $ ␣ e d s a a a n n n a a ␣

$MTF(B) =$ 8 5 5 2 **0 0** 8 7 **0 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 $T$ = `have␣had␣hadnt␣hasnt␣havent␣has␣what$`.
The BWT is $B$ = `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 `a`s are preceded by `h`

**F** `h` is the 4th character in the alphabet

→ *sli.do/comp526*

# Clicker Question

Consider $T$ = have␣had␣hadnt␣hasnt␣havent␣has␣what$.
The BWT is $B$ = 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~~

→ *sli.do/comp526*

# 7.10 Inverse BWT

# Inverse BWT

- Great, can compute BWT efficiently and it helps compression. *But how can we decode it?*

not even obvious that it is at all invertible!

# Inverse BWT

- Great, can compute BWT efficiently and it helps compression. *But how can we decode it?*

not even obvious that it is at all invertible!

- **"Magic" solution:**
  1. Create array $D[0..n]$ of pairs: $D[r] = (B[r], r)$.
  2. Sort $D$ *stably* with respect to *first entry*.
  3. Use $D$ as linked list with (char, next entry)

# Inverse BWT

- Great, can compute BWT efficiently and it helps compression. *But how can we decode it?*

not even obvious that it is at all invertible!

- **"Magic" solution:**
  1. Create array $D[0..n]$ of pairs: $D[r] = (B[r], r)$.
  2. Sort $D$ *stably* with respect to *first entry*.
  3. Use $D$ as linked list with (char, next entry)

**Example:**

$B =$ `ard$rcaaaabb`

$S =$

$D$

| | |
|---|---|
| 0 | (a, 0) |
| 1 | (r, 1) |
| 2 | (d, 2) |
| 3 | ($, 3) |
| 4 | (r, 4) |
| 5 | (c, 5) |
| 6 | (a, 6) |
| 7 | (a, 7) |
| 8 | (a, 8) |
| 9 | (a, 9) |
| 10 | (b, 10) |
| 11 | (b, 11) |

# Inverse BWT

- Great, can compute BWT efficiently and it helps compression. *But how can we decode it?*

not even obvious that it is at all invertible!

- **"Magic" solution:**
  1. Create array $D[0..n]$ of pairs: $D[r] = (B[r], r)$.
  2. Sort $D$ *stably* with respect to *first entry*.
  3. Use $D$ as linked list with (char, next entry)

**Example:**

$B$ = `ard$rcaaaabb`

$S$ =

| | $D$ | sorted $D$ | |
|---|---|---|---|
| | | char | next |
| 0 | (a, 0) | ($, 3) | |
| 1 | (r, 1) | (a, 0) | |
| 2 | (d, 2) | (a, 6) | |
| 3 | ($, 3) | (a, 7) | |
| 4 | (r, 4) | (a, 8) | |
| 5 | (c, 5) | (a, 9) | |
| 6 | (a, 6) | (b, 10) | |
| 7 | (a, 7) | (b, 11) | |
| 8 | (a, 8) | (c, 5) | |
| 9 | (a, 9) | (d, 2) | |
| 10 | (b, 10) | (r, 1) | |
| 11 | (b, 11) | (r, 4) | |

# Inverse BWT

- Great, can compute BWT efficiently and it helps compression. *But how can we decode it?*

not even obvious that it is at all invertible!

- **"Magic" solution:**
  1. Create array $D[0..n]$ of pairs: $D[r] = (B[r], r)$.
  2. Sort $D$ *stably* with respect to *first entry*.
  3. Use $D$ as linked list with (char, next entry)

**Example:**

$B$ = `ard$rcaaaabb`

$S$ = `a`

| | $D$ | sorted $D$ | |
|---|---|---|---|
| | | char | next |
| 0 | (a, 0) | ($, | 3) |
| 1 | (r, 1) | (a, | 0) |
| 2 | (d, 2) | (a, | 6) |
| 3 | ($, 3) | (a, | 7) |
| 4 | (r, 4) | (a, | 8) |
| 5 | (c, 5) | (a, | 9) |
| 6 | (a, 6) | (b, | 10) |
| 7 | (a, 7) | (b, | 11) |
| 8 | (a, 8) | (c, | 5) |
| 9 | (a, 9) | (d, | 2) |
| 10 | (b, 10) | (r, | 1) |
| 11 | (b, 11) | (r, | 4) |

# Inverse BWT

- Great, can compute BWT efficiently and it helps compression. *But how can we decode it?*

not even obvious that it is at all invertible!

- **"Magic" solution:**
  1. Create array $D[0..n]$ of pairs: $D[r] = (B[r], r)$.
  2. Sort $D$ *stably* with respect to *first entry*.
  3. Use $D$ as linked list with (char, next entry)

**Example:**

$B$ = `ard$rcaaaabb`

$S$ = `ab`

| | $D$ |
|---|---|
| 0 | (a, 0) |
| 1 | (r, 1) |
| 2 | (d, 2) |
| 3 | ($, 3) |
| 4 | (r, 4) |
| 5 | (c, 5) |
| 6 | (a, 6) |
| 7 | (a, 7) |
| 8 | (a, 8) |
| 9 | (a, 9) |
| 10 | (b, 10) |
| 11 | (b, 11) |

| | sorted $D$ |
|---|---|
| | char next |
| 0 | ($, 3) |
| 1 | (a, 0) |
| 2 | (a, 6) |
| 3 | (a, 7) |
| 4 | (a, 8) |
| 5 | (a, 9) |
| 6 | (b, 10) |
| 7 | (b, 11) |
| 8 | (c, 5) |
| 9 | (d, 2) |
| 10 | (r, 1) |
| 11 | (r, 4) |

# Inverse BWT

- Great, can compute BWT efficiently and it helps compression. *But how can we decode it?*

not even obvious that it is at all invertible!

- **"Magic" solution:**
  1. Create array $D[0..n]$ of pairs: $D[r] = (B[r], r)$.
  2. Sort $D$ *stably* with respect to *first entry*.
  3. Use $D$ as linked list with (char, next entry)

**Example:**

$B$ = `ard$rcaaaabb`

$S$ = `abr`

| | $D$ |
|---|---|
| 0 | (a, 0) |
| 1 | (r, 1) |
| 2 | (d, 2) |
| 3 | ($, 3) |
| 4 | (r, 4) |
| 5 | (c, 5) |
| 6 | (a, 6) |
| 7 | (a, 7) |
| 8 | (a, 8) |
| 9 | (a, 9) |
| 10 | (b, 10) |
| 11 | (b, 11) |

| | sorted $D$ | |
|---|---|---|
| | char | next |
| 0 | ($, | 3) |
| 1 | (a, | 0) |
| 2 | (a, | 6) |
| 3 | (a, | 7) |
| 4 | (a, | 8) |
| 5 | (a, | 9) |
| 6 | (b, | 10) |
| 7 | (b, | 11) |
| 8 | (c, | 5) |
| 9 | (d, | 2) |
| 10 | (r, | 1) |
| 11 | (r, | 4) |

# Inverse BWT

- Great, can compute BWT efficiently and it helps compression. *But how can we decode it?*

not even obvious that it is at all invertible!

- **"Magic" solution:**
  1. Create array $D[0..n]$ of pairs: $D[r] = (B[r], r)$.
  2. Sort $D$ *stably* with respect to *first entry*.
  3. Use $D$ as linked list with (char, next entry)

**Example:**

$B$ = `ard$rcaaaabb`

$S$ = `abra`

| | $D$ |
|---|---|
| 0 | (a, 0) |
| 1 | (r, 1) |
| 2 | (d, 2) |
| 3 | ($, 3) |
| 4 | (r, 4) |
| 5 | (c, 5) |
| 6 | (a, 6) |
| 7 | (a, 7) |
| 8 | (a, 8) |
| 9 | (a, 9) |
| 10 | (b, 10) |
| 11 | (b, 11) |

| | sorted $D$ | |
|---|---|---|
| | char | next |
| 0 | ($, | 3) |
| 1 | (a, | 0) |
| 2 | (a, | 6) |
| 3 | (a, | 7) |
| 4 | (a, | 8) |
| 5 | (a, | 9) |
| 6 | (b, | 10) |
| 7 | (b, | 11) |
| 8 | (c, | 5) |
| 9 | (d, | 2) |
| 10 | (r, | 1) |
| 11 | (r, | 4) |

# Inverse BWT

- Great, can compute BWT efficiently and it helps compression. *But how can we decode it?*

not even obvious that it is at all invertible!

- **"Magic" solution:**
  1. Create array $D[0..n]$ of pairs: $D[r] = (B[r], r)$.
  2. Sort $D$ *stably* with respect to *first entry*.
  3. Use $D$ as linked list with (char, next entry)

**Example:**

$B$ = `ard$rcaaaabb`

$S$ = `abrac`

| | $D$ |
|---|---|
| 0 | (a, 0) |
| 1 | (r, 1) |
| 2 | (d, 2) |
| 3 | ($, 3) |
| 4 | (r, 4) |
| 5 | (c, 5) |
| 6 | (a, 6) |
| 7 | (a, 7) |
| 8 | (a, 8) |
| 9 | (a, 9) |
| 10 | (b, 10) |
| 11 | (b, 11) |

| | sorted $D$ | |
|---|---|---|
| | char | next |
| 0 | ($, | 3) |
| 1 | (a, | 0) |
| 2 | (a, | 6) |
| 3 | (a, | 7) |
| 4 | (a, | 8) |
| 5 | (a, | 9) |
| 6 | (b, | 10) |
| 7 | (b, | 11) |
| 8 | (c, | 5) |
| 9 | (d, | 2) |
| 10 | (r, | 1) |
| 11 | (r, | 4) |

# Inverse BWT

- Great, can compute BWT efficiently and it helps compression. *But how can we decode it?*

not even obvious that it is at all invertible!

- **"Magic" solution:**
  1. Create array $D[0..n]$ of pairs: $D[r] = (B[r], r)$.
  2. Sort $D$ *stably* with respect to *first entry*.
  3. Use $D$ as linked list with (char, next entry)

**Example:**

$B$ = `ard$rcaaaabb`

$S$ = `abraca`

| | $D$ |
|---|---|
| 0 | (a, 0) |
| 1 | (r, 1) |
| 2 | (d, 2) |
| 3 | ($, 3) |
| 4 | (r, 4) |
| 5 | (c, 5) |
| 6 | (a, 6) |
| 7 | (a, 7) |
| 8 | (a, 8) |
| 9 | (a, 9) |
| 10 | (b, 10) |
| 11 | (b, 11) |

| | sorted $D$ | |
|---|---|---|
| | char | next |
| 0 | ($, | 3) |
| 1 | (a, | 0) |
| 2 | (a, | 6) |
| 3 | (a, | 7) |
| 4 | (a, | 8) |
| 5 | (a, | 9) |
| 6 | (b, | 10) |
| 7 | (b, | 11) |
| 8 | (c, | 5) |
| 9 | (d, | 2) |
| 10 | (r, | 1) |
| 11 | (r, | 4) |

# Inverse BWT

- Great, can compute BWT efficiently and it helps compression. *But how can we decode it?*

not even obvious that it is at all invertible!

- **"Magic" solution:**
  1. Create array $D[0..n]$ of pairs: $D[r] = (B[r], r)$.
  2. Sort $D$ *stably* with respect to *first entry*.
  3. Use $D$ as linked list with (char, next entry)

**Example:**

$B$ = `ard$rcaaaabb`

$S$ = `abracad`

| | $D$ |
|---|---|
| 0 | (a, 0) |
| 1 | (r, 1) |
| 2 | (d, 2) |
| 3 | ($, 3) |
| 4 | (r, 4) |
| 5 | (c, 5) |
| 6 | (a, 6) |
| 7 | (a, 7) |
| 8 | (a, 8) |
| 9 | (a, 9) |
| 10 | (b, 10) |
| 11 | (b, 11) |

| | sorted $D$ | |
|---|---|---|
| | char | next |
| 0 | ($, | 3) |
| 1 | (a, | 0) |
| 2 | (a, | 6) |
| 3 | (a, | 7) |
| 4 | (a, | 8) |
| 5 | (a, | 9) |
| 6 | (b, | 10) |
| 7 | (b, | 11) |
| 8 | (c, | 5) |
| 9 | (d, | 2) |
| 10 | (r, | 1) |
| 11 | (r, | 4) |

# Inverse BWT

- Great, can compute BWT efficiently and it helps compression. *But how can we decode it?*

not even obvious that it is at all invertible!

- **"Magic" solution:**
  1. Create array $D[0..n]$ of pairs: $D[r] = (B[r], r)$.
  2. Sort $D$ *stably* with respect to *first entry*.
  3. Use $D$ as linked list with (char, next entry)

**Example:**

$B$ = `ard$rcaaaabb`

$S$ = `abracada`

| | $D$ |
|---|---|
| 0 | (a, 0) |
| 1 | (r, 1) |
| 2 | (d, 2) |
| 3 | ($, 3) |
| 4 | (r, 4) |
| 5 | (c, 5) |
| 6 | (a, 6) |
| 7 | (a, 7) |
| 8 | (a, 8) |
| 9 | (a, 9) |
| 10 | (b, 10) |
| 11 | (b, 11) |

| | sorted $D$ | |
|---|---|---|
| | char | next |
| 0 | ($, | 3) |
| 1 | (a, | 0) |
| 2 | (a, | 6) |
| 3 | (a, | 7) |
| 4 | (a, | 8) |
| 5 | (a, | 9) |
| 6 | (b, | 10) |
| 7 | (b, | 11) |
| 8 | (c, | 5) |
| 9 | (d, | 2) |
| 10 | (r, | 1) |
| 11 | (r, | 4) |

# Inverse BWT

- Great, can compute BWT efficiently and it helps compression. *But how can we decode it?*

not even obvious that it is at all invertible!

- **"Magic" solution:**
  1. Create array $D[0..n]$ of pairs: $D[r] = (B[r], r)$.
  2. Sort $D$ *stably* with respect to *first entry*.
  3. Use $D$ as linked list with (char, next entry)

**Example:**

$B$ = `ard$rcaaaabb`

$S$ = `abracadab`

| | $D$ | sorted $D$ | |
|---|---|---|---|
| | | char | next |
| 0 | (a, 0) | ($, 3) | |
| 1 | (r, 1) | (a, 0) | |
| 2 | (d, 2) | (a, 6) | |
| 3 | ($, 3) | (a, 7) | |
| 4 | (r, 4) | (a, 8) | |
| 5 | (c, 5) | (a, 9) | |
| 6 | (a, 6) | (b, 10) | |
| 7 | (a, 7) | (b, 11) | |
| 8 | (a, 8) | (c, 5) | |
| 9 | (a, 9) | (d, 2) | |
| 10 | (b, 10) | (r, 1) | |
| 11 | (b, 11) | (r, 4) | |

# Inverse BWT

- Great, can compute BWT efficiently and it helps compression. *But how can we decode it?*

not even obvious that it is at all invertible!

- **"Magic" solution:**
  1. Create array $D[0..n]$ of pairs: $D[r] = (B[r], r)$.
  2. Sort $D$ *stably* with respect to *first entry*.
  3. Use $D$ as linked list with (char, next entry)

**Example:**

$B$ = `ard$rcaaaabb`

$S$ = `abracadabr`

| | $D$ | sorted $D$ | |
|---|---|---|---|
| | | char | next |
| 0 | (a, 0) | ($, 3) | |
| 1 | (r, 1) | (a, 0) | |
| 2 | (d, 2) | (a, 6) | |
| 3 | ($, 3) | (a, 7) | |
| 4 | (r, 4) | (a, 8) | |
| 5 | (c, 5) | (a, 9) | |
| 6 | (a, 6) | (b, 10) | |
| 7 | (a, 7) | (b, 11) | |
| 8 | (a, 8) | (c, 5) | |
| 9 | (a, 9) | (d, 2) | |
| 10 | (b, 10) | (r, 1) | |
| 11 | (b, 11) | (r, 4) | |

# Inverse BWT

- Great, can compute BWT efficiently and it helps compression. *But how can we decode it?*

not even obvious that it is at all invertible!

- **"Magic" solution:**
  1. Create array $D[0..n]$ of pairs: $D[r] = (B[r], r)$.
  2. Sort $D$ *stably* with respect to *first entry*.
  3. Use $D$ as linked list with (char, next entry)

**Example:**

$B$ = `ard$rcaaaabb`

$S$ = `abracadabra`

| | $D$ |
|---|---|
| 0 | (a, 0) |
| 1 | (r, 1) |
| 2 | (d, 2) |
| 3 | ($, 3) |
| 4 | (r, 4) |
| 5 | (c, 5) |
| 6 | (a, 6) |
| 7 | (a, 7) |
| 8 | (a, 8) |
| 9 | (a, 9) |
| 10 | (b, 10) |
| 11 | (b, 11) |

| | sorted $D$ | |
|---|---|---|
| | char | next |
| 0 | ($, | 3) |
| 1 | (a, | 0) |
| 2 | (a, | 6) |
| 3 | (a, | 7) |
| 4 | (a, | 8) |
| 5 | (a, | 9) |
| 6 | (b, | 10) |
| 7 | (b, | 11) |
| 8 | (c, | 5) |
| 9 | (d, | 2) |
| 10 | (r, | 1) |
| 11 | (r, | 4) |

# Inverse BWT

- Great, can compute BWT efficiently and it helps compression. *But how can we decode it?*

not even obvious that it is at all invertible!

- **"Magic" solution:**
  1. Create array $D[0..n]$ of pairs: $D[r] = (B[r], r)$.
  2. Sort $D$ *stably* with respect to *first entry*.
  3. Use $D$ as linked list with (char, next entry)

**Example:**

$B$ = `ard$rcaaaabb`

$S$ = `abracadabra$`

| | $D$ | sorted $D$ | | |
|---|---|---|---|---|
| | | | char | next |
| 0 | (a, 0) | 0 | ($, 3) | |
| 1 | (r, 1) | 1 | (a, 0) | |
| 2 | (d, 2) | 2 | (a, 6) | |
| 3 | ($, 3) | 3 | (a, 7) | |
| 4 | (r, 4) | 4 | (a, 8) | |
| 5 | (c, 5) | 5 | (a, 9) | |
| 6 | (a, 6) | 6 | (b, 10) | |
| 7 | (a, 7) | 7 | (b, 11) | |
| 8 | (a, 8) | 8 | (c, 5) | |
| 9 | (a, 9) | 9 | (d, 2) | |
| 10 | (b, 10) | 10 | (r, 1) | |
| 11 | (b, 11) | 11 | (r, 4) | |

# Inverse BWT – The magic revealed

- Inverse BWT very easy to compute:
  - only sort individual characters in $B$ (not suffixes)
  - $\rightsquigarrow$ $O(n)$ with counting sort
- *but why does this work!?*

# Inverse BWT – The magic revealed

- Inverse BWT very easy to compute:
  - only sort individual characters in $B$ (not suffixes)
  - $\rightsquigarrow$ $O(n)$ with counting sort
- *but why does this work!?*
- decode char by char
  - can find unique \$ $\rightsquigarrow$ starting row
- to get next char, we need
  - (i) char in *first* column of *current row*
  - (ii) find row with that char's copy in BWT
  - $\rightsquigarrow$ then we can walk through and decode

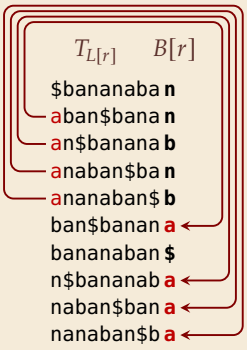
# Inverse BWT – The magic revealed

- Inverse BWT very easy to compute:
  - only sort individual characters in $B$ (not suffixes)
  - $\rightsquigarrow$ $O(n)$ with counting sort
- *but why does this work!?*
- decode char by char
  - can find unique \$ $\rightsquigarrow$ starting row
- to get next char, we need
  - (i) char in *first* column of *current row*
  - (ii) find row with that char's copy in BWT
  - $\rightsquigarrow$ then we can walk through and decode
- for (i): first column = characters of $B$ in sorted order ✓

# Inverse BWT – The magic revealed

- Inverse BWT very easy to compute:
  - only sort individual characters in $B$ (not suffixes)
  - $\rightsquigarrow$ $O(n)$ with counting sort
- *but why does this work!?*
- decode char by char
  - can find unique `$` $\rightsquigarrow$ starting row
- to get next char, we need
  - (i) char in *first* column of *current row*
  - (ii) find row with that char's copy in BWT
  - $\rightsquigarrow$ then we can walk through and decode
- for (i): first column = characters of $B$ in sorted order ✓
- for (ii): relative order of same character stays same:
  $i$th `a` in first column = $i$th `a` in BWT
  - $\rightsquigarrow$ stably sorting $(B[r], r)$ by first entry enough ✓

| $r$ | $L[r]$ |
|---|---|
| 0 | 9 |
| 1 | 5 |
| 2 | 7 |
| 3 | 3 |
| 4 | 1 |
| 5 | 6 |
| 6 | 0 |
| 7 | 8 |
| 8 | 4 |
| 9 | 2 |



# BWT – Discussion

- Running time: $\Theta(n)$
  - **encoding** uses suffix sorting
  - decoding only needs counting sort
  - $\rightsquigarrow$ decoding much simpler & faster (but same $\Theta$-class)

# BWT – Discussion

- Running time: $\Theta(n)$
  - **encoding** uses suffix sorting
  - decoding only needs counting sort
  - $\rightsquigarrow$ decoding much simpler & faster (but same $\Theta$-class)

👎 typically slower than other methods

👎 need access to entire text (or apply to blocks independently)

👍 BWT-MTF-RLE-Huffman (bzip2) pipeline tends to have best compression

# Summary of Compression Methods

| | |
|---|---|
| Huffman | Variable-width, single-character (optimal in this case) |
| RLE | Variable-width, multiple-character encoding |
| LZW | Adaptive, fixed-width, multiple-character encoding<br>Augments dictionary with repeated substrings |
| MTF | Adaptive, transforms to smaller integers<br>should be followed by variable-width integer encoding |
| BWT | Block compression method, should be followed by MTF |