

7

Compression

20 April 2021

Sebastian Wild


Outline

7 Compression

- 7.1 Context
- 7.2 Character Encodings
- 7.3 Huffman Codes
- 7.4 Run-Length Encoding
- 7.5 Lempel-Ziv-Welch
- 7.6 Move-to-Front Transformation
- 7.7 Burrows-Wheeler Transform

7.1 Context

Overview

- ▶ Unit 4–6: How to *work* with strings
 - ▶ finding substrings
 - ▶ finding approximate matches
 - ▶ finding repeated parts
 - ▶ ...
- ▶ Unit 7–8: How to *store* strings
 - ▶ computer memory: must be binary
 - ▶ how to compress strings (save space)
 - ▶ how to robustly transmit over noisy channels  Unit 8

Terminology

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

What is a good encoding scheme?

- ▶ Depending on the application, goals can be
 - ▶ efficiency of encoding/decoding
 - ▶ resilience to errors/noise in transmission
 - ▶ security (encryption)
 - ▶ integrity (detect modifications made by third parties)
 - ▶ size

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

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

- ▶ We will measure the *compression ratio*:
$$\frac{|C| \cdot \lg |\Sigma_C|}{|S| \cdot \lg |\Sigma_S|} \stackrel{\Sigma_C=\{0,1\}}{=} \frac{|C|}{|S| \cdot \lg |\Sigma_S|}$$
 - < 1 means successful compression
 - = 1 means no compression
 - > 1 means “compression” made it bigger!? (yes, that happens ...)

Types of Data Compression

▶ Logical vs. Physical

- ▶ **Logical Compression** uses meaning of data
 - ↪ only applies to a certain domain, e. g., sound recordings
- ▶ **Physical Compression** only knows the (physical) **bits** in the data, not the meaning behind them

▶ 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 *physical*, *lossless* compression algorithms.
These techniques can be used for any application.

What makes data compressible?

- Physical, lossless compression methods mainly exploit two types of redundancies in source texts:

1. **uneven character frequencies**

some characters occur more often than others → Part I

2. **repetitive texts**

different parts in the text are (almost) identical → Part II



There is no such thing as a free lunch!

Not *everything* is compressible (→ tutorials)

~> focus on versatile methods that often work

Part I

Exploiting character frequencies

7.2 Character Encodings

Character encodings

- ▶ Simplest form of encoding: Encode each source character individually

↪ encoding function $E : \Sigma_S \rightarrow \Sigma_C^*$

- ▶ typically, $|\Sigma_S| \gg |\Sigma_C|$, so need several bits per character
- ▶ for $c \in \Sigma_S$, we call $E(c)$ the *codeword* of c
- ▶ fixed-length code: $|E(c)|$ is the same for all $c \in \Sigma_S$
- ▶ variable-length code: not all codewords of same length

Fixed-length codes

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

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

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

Fixed-length codes – Discussion



Encoding & Decoding as fast as it gets



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



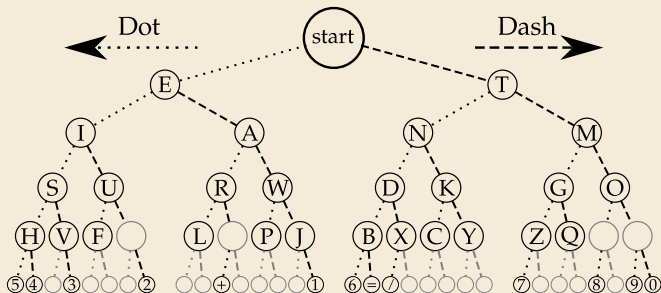
inflexible (how to support adding a new character?)

Variable-length codes

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

International Morse Code

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



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

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

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 characters start with 1–4 1s indicating the total number of bytes, followed by a 0 and 3–5 bits.

The remaining bytes each start with 10 followed by 6 bits.

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



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

Code tries

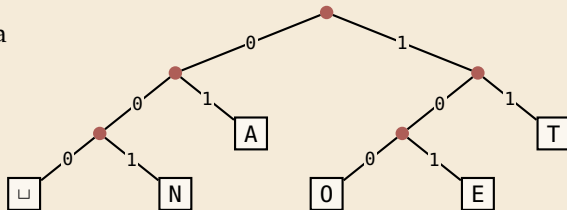
- ▶ From now on only consider prefix-free codes E :
 $E(c)$ is not a prefix of $E(c')$ for any $c, c' \in \Sigma_S$.

▶ **Example:**

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

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

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



- ▶ Encode AN \sqcup ANT \rightarrow 010010000100111
- ▶ Decode 111000001010111 \rightarrow T \sqcup 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 → next)
 - ▶ **adaptive coding:** code depends on message and changes during encoding; implicitly stored withing the message (e. g., LZW → below)

7.3 Huffman Codes

Character frequencies

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

Typical English prose:

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

⇒ Want shorter codes for more frequent characters!

Huffman coding

e. g. frequencies / probabilities

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

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

- ▶ 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.

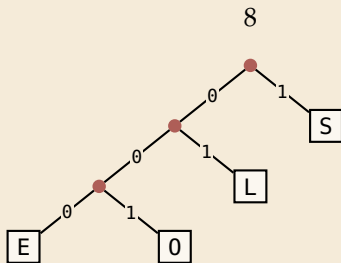
Huffman's algorithm:

1. Find two characters **a**, **b** with lowest weights.
 - ▶ We will encode them with the same prefix, plus one distinguishing bit, i.e., $E(a) = u0$ and $E(b) = u1$ for a bitstring $u \in \{0, 1\}^*$ (u to be determined)
 2. (Conceptually) replace **a** and **b** by a single character "**ab**" with $w(\text{ab}) = w(a) + w(b)$.
 3. Recursively apply Huffman's algorithm on the smaller alphabet. This in particular determines $u = E(\text{ab})$.
- ▶ efficient implementation using a (min-oriented) *priority queue*
 - ▶ start by inserting all characters with their weight as key
 - ▶ step 1 uses two deleteMin calls
 - ▶ step 2 inserts a new character with the sum of old weights as key

Huffman's algorithm – Example

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

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



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

LOSSLESS \rightarrow 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.

Huffman code – Optimality

Theorem 7.1 (Optimality of Huffman's Algorithm)

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

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

- ▶ Given any optimal prefix-free code E^* (as its code trie).
 - ▶ code trie $\rightsquigarrow \exists$ two sibling leaves x, y at largest depth D
 - ▶ swap characters in leaves to have two lowest-weight characters a, b in x, y (that can only make ℓ smaller, so still optimal)
 - ▶ any optimal code for $\Sigma' = \Sigma \setminus \{a, b\} \cup \{\boxed{ab}\}$ yields optimal code for Σ by replacing leaf \boxed{ab} by internal node with children a and b .
- \rightsquigarrow recursive call yields optimal code for Σ' by inductive hypothesis, so Huffman's algorithm finds optimal code for Σ . 

Entropy

Definition 7.2 (Entropy)

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

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

- ▶ entropy is a **measure of information** content of a distribution
 - ▶ more precisely: the expected number of bits (Yes/No questions) required to nail down the random value
- ↪ would ideally encode value i using $\lg(1/p_i)$ bits
that is not always possible; cannot use 1.5 bits ... but:

Theorem 7.3 (Entropy bounds for Huffman codes)

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

$$\mathcal{H}\left(\frac{w(a_1)}{W}, \dots, \frac{w(a_\sigma)}{W}\right) \leq \ell(E) \leq \mathcal{H}\left(\frac{w(a_1)}{W}, \dots, \frac{w(a_\sigma)}{W}\right) + 1$$

where $W = w(a_1) + \dots + w(a_\sigma)$.


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 coding – Discussion


- ▶ running time complexity: $O(\sigma \log \sigma)$ to construct code
 - ▶ build PQ + $\sigma \cdot (2 \text{ deleteMins and } 1 \text{ insert})$
 - ▶ can do $\Theta(\sigma)$ time when characters already sorted by weight
 - ▶ time for encoding: $O(n + |C|)$
- ▶ 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 compress this very much

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

7.4 Run-Length Encoding

Run-Length encoding

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

[illegible]

same character repeated

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

→ run-length encoding (RLE):

use runs as phrases: $S = \underbrace{00000}_{\text{run 1}} \underbrace{111}_{\text{run 2}} \underbrace{0000}_{\text{run 3}}$

⇒ 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!)

Run-length encoding – Examples

► Encoding:

$S = 11111111001000000000000000000000111111111111$

$C = 10011101010000101000001011$

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

► Decoding:

$C = 00001101001001010$

$b =$


$\ell =$


$k =$


$S = 0000000000000001111011$

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

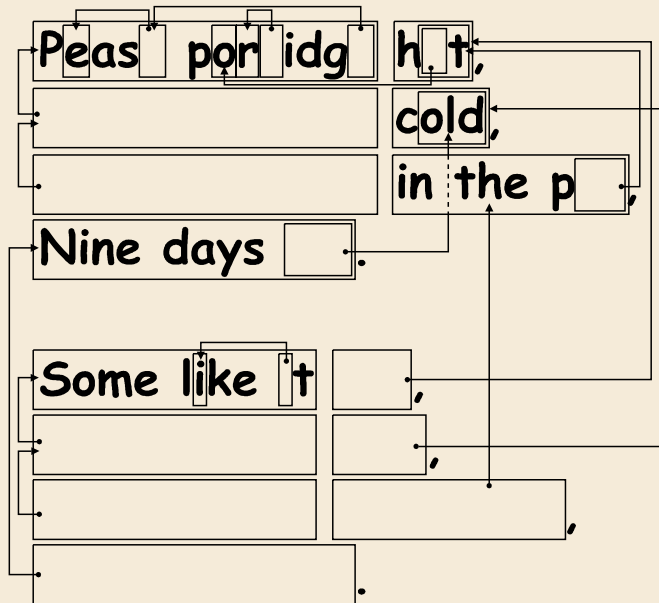
 can compress n bits to $\Theta(\log n)!$
for extreme case of constant number of runs

 negligible compression for many common types of data

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

7.5 Lempel-Ziv-Welch

Warmup



<https://classic.cs.unlunned.org/text-compression/>



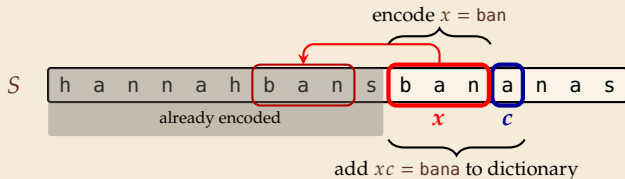
<https://www.flickr.com/photos/quintanaroo/2742726346>

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", "
"
- ▶ **Lempel-Ziv** stands for family of *adaptive* compression algorithms.
 - ▶ **Idea:** store repeated parts by reference!
 - ~> each codeword refers to
 - ▶ either a single character in Σ_S ,
 - ▶ or a *substring* of S (that both encoder and decoder have already seen).
 - ▶ Variants of Lempel-Ziv compression
 - ▶ "LZ77" Original version ("sliding window")
Derivatives: LZSS, LZFG, LZRW, LZW, DEFLATE, ...
DEFLATE used in (pk)zip, gzip, PNG
 - ▶ "LZ78" Second (slightly improved) version
Derivatives: LZW, LZMW, LZAP, LZJ, ...
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!
- ▶ maintain a **dictionary** D with 2^k entries \rightsquigarrow codewords = indices in dictionary
 - ▶ initially, first $|\Sigma_S|$ entries encode single characters (rest is empty)
 - ▶ **add** a new entry to D **after each step**:
 - ▶ **Encoding**: after encoding a substring x of S , add xc to D where c is the character that follows x in S .



\rightsquigarrow new codeword in D

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

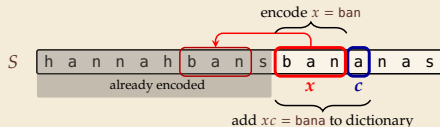
LZW encoding – Example

Input: Y0!_YOU!_YOUR_YOYO!

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

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

$D =$



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

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

LZW encoding – Code

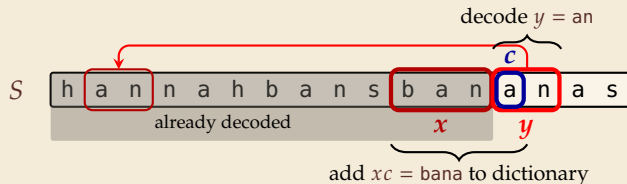
```
1 procedure LZWencode( $S[0..n]$ )
2    $x := \varepsilon$  // previous phrase, initially empty
3    $C := \varepsilon$  // output, initially empty
4    $D :=$  dictionary, initialized with codes for  $c \in \Sigma_S$  // stored as trie
5    $k := |\Sigma_S|$  // next free codeword
6   for  $i :=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

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)
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

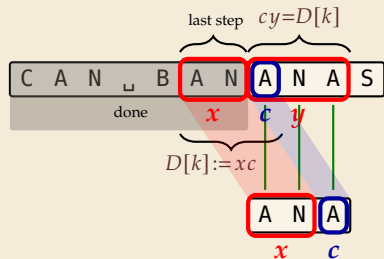
- ▶ example: Want to decode 133, but not yet in dictionary!



decoder is “one step behind” in creating dictionary

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

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



1. en/decode x .

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

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

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

LZW decoding – Code

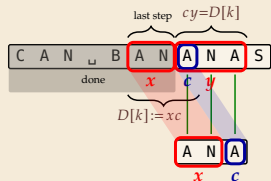
```
1 procedure LZWdecode( $C[0..m]$ )
2    $D :=$  dictionary  $[0..2^d) \rightarrow \Sigma_S^+$ , initialized with codes for  $c \in \Sigma_S$  // stored as array
3    $k := |\Sigma_S|$  // next unused codeword
4    $q := C[0]$  // first codeword
5    $y := D[q]$  // lookup meaning of  $q$  in  $D$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

LZW decoding – Example continued

► Example: 67 65 78 32 66 129 133 83

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 $D[k] := \mathbf{x}\mathbf{c}$
3. next phrase y equals $D[k]$
 $\rightsquigarrow D[k] = \mathbf{x}\mathbf{c} = \mathbf{x} \cdot \mathbf{x}[\mathbf{0}]$ (all known)

LZW – Discussion

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

- 👍 fast encoding & decoding
- 👍 works in streaming model (no random access, no backtrack on input needed)
- 👍 significant compression for many types of data
- 👎 captures only local repetitions (with bounded dictionary)

Compression summary

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

Part III

Text Transforms

Text transformations

- ▶ compression is effective if we have one of 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
- ▶ Enter: **text transformations**
 - ▶ invertible functions of text
 - ▶ do not by themselves reduce the space usage
 - ▶ but help compressors “see” existing redundancy
 - \rightsquigarrow use as pre-/postprocessing in compression pipeline

7.6 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
- ▶ Here: use such a list for storing source alphabet Σ_S
 - ▶ to encode c , access it in list
 - ▶ encode c using its (old) position in list
 - ▶ then apply MTF to the list
 - ↪ codewords are integers, i. e., $\Sigma_C = [0..\sigma)$
- ↪ clusters of few characters ↪ many small numbers

MTF – Code

► Transform (encode):

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

► Inverse transform (decode):

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

► Important: encoding and decoding produce same accesses to list

MTF – Example

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
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

$S = \text{INEFFICIENCIES}$

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

- ▶ 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)

~> prime use as part of longer pipeline

- ▶ two simple ideas for encoding codewords:
 - ▶ Elias gamma code ~> 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 (→ next)

7.7 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(!)
- ▶ Encoding algorithm needs **all** of S (no streaming possible).
 - ↪ BWT is a *block compression method*.
- ▶ BWT followed by MTF, RLE, and Huffman is the algorithm used by the bzip2 program. achieves best compression on English text of any algorithm we have seen:

4047392 bible.txt

1191071 bible.txt.gz

888604 bible.txt.7z

845635 bible.txt.bz2

BWT transform

- *cyclic shift* of a string:

$T = \text{time_flies_quickly_}$

$\text{flies_quickly_time_}$

- add *end-of-word character* \$ to S
(as in Unit 6)

~> can recover original string



~> cyclic shift



- 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

$S = \text{alf_eats_alfalfa\$}$

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

$B = \text{asff\$f_e_lllaaata}$

```
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

```
$alf_eats_alfalfa
_alfalfa$alf_eats
_eats_alfalfa$alf
a$alf_eats_alfalf
alf_eats_alfalfa$
alfa$alf_eats_alf
alfalfa$alf_eats_
ats_alfalfa$alf_e
eats_alfalfa$alf_
f_eats_alfalfa$al
fa$alf_eats_alfal
falfa$alf_eats_al
lf_eats_alfalfa$a
lfa$alf_eats_alfa
lfalfa$alf_eats_a
s_alfalfa$alf_eat
ts_alfalfa$alf_ea
```

BWT
↓

BWT – Implementation & Properties

Compute BWT efficiently:

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

Why does BWT help?

- ▶ sorting groups characters *by what follows*
 - ▶ Example: `lf` always preceded by `a`
- ↪ B has local clusters of characters
 - ▶ that makes MTF effective
- ▶ repeated substring in S ↪ 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\$alf	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\$a	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

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:

$B = \text{ard\$rcaaaabb}$

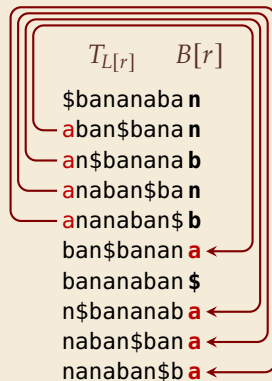
$S = \text{abracadabra\$}$

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)
 - ↪ $O(n)$ with counting sort
- ▶ *but why does this work!?*
- ▶ decode char by char
 - ▶ can find unique \$ ↪ 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
 - ↪ then we can walk through and decode
- ▶ for (i): first column = characters of B in sorted order ✓
- ▶ for (ii): relative order of same character same!
 - ▶ i th a in first column = i th a in BWT
 - ↪ 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
 - ↪ decoding much simpler & faster (but same Θ -class)

👎 typically slower than other methods

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

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

Bigger Example

have had hadnt hasnt havent has what\$
ave had hadnt hasnt havent has what\$h
ve had hadnt hasnt havent has what\$ha
e had hadnt hasnt havent has what\$hav
had hadnt hasnt havent has what\$have
had hadnt hasnt havent has what\$have
ad hadnt hasnt havent has what\$have h
d hadnt hasnt havent has what\$have ha
hadnt hasnt havent has what\$have had
hadnt hasnt havent has what\$have had
adnt hasnt havent has what\$have had h
dnt hasnt havent has what\$have had ha
nt hasnt havent has what\$have had had
t hasnt havent has what\$have had hadnt
hasnt havent has what\$have had hadnt
hasnt havent has what\$have had hadnt
asnt havent has what\$have had hadnt h
snt havent has what\$have had hadnt ha
nt havent has what\$have had hadnt has
t havent has what\$have had hadnt hasn
havent has what\$have had hadnt hasnt
havent has what\$have had hadnt hasnt
avent has what\$have had hadnt hasnt h
vent has what\$have had hadnt hasnt ha
ent has what\$have had hadnt hasnt hav
nt has what\$have had hadnt hasnt have
t has what\$have had hadnt hasnt haven
has what\$have had hadnt hasnt havent
has what\$have had hadnt hasnt havent
as what\$have had hadnt hasnt havent h
s what\$have had hadnt hasnt havent ha
what\$have had hadnt hasnt havent has
what\$have had hadnt hasnt havent has
hat\$have had hadnt hasnt havent has w
at\$have had hadnt hasnt havent has wh
t\$have had hadnt hasnt havent has wha
\$have had hadnt hasnt havent has what

\$have had hadnt hasnt havent has what
had hadnt hasnt havent has what\$have
hadnt hasnt havent has what\$have had
has what\$have had hadnt hasnt havent
hasnt havent has what\$have had hadnt
havent has what\$have had hadnt hasnt
what\$have had hadnt hasnt havent has
ad hadnt hasnt havent has what\$have h
adnt hasnt havent has what\$have had h
as what\$have had hadnt hasnt havent h
asnt havent has what\$have had hadnt h
at\$have had hadnt hasnt havent has wh
ave had hadnt hasnt havent has what\$h
avent has what\$have had hadnt hasnt h
d hadnt hasnt havent has what\$have ha
dnt hasnt havent has what\$have had ha
e had hadnt hasnt havent has what\$hav
ent has what\$have had hadnt hasnt hav
had hadnt hasnt havent has what\$have
hadnt hasnt havent has what\$have had
has what\$have had hadnt hasnt havent
hasnt havent has what\$have had hadnt
hat\$have had hadnt hasnt havent has w
have had hadnt hasnt havent has what\$
havent has what\$have had hadnt hasnt
nt has what\$have had hadnt hasnt have
nt hasnt havent has what\$have had had
nt havent has what\$have had hadnt has
s what\$have had hadnt hasnt havent ha
snt havent has what\$have had hadnt ha
t\$have had hadnt hasnt havent has wha
t has what\$have had hadnt hasnt haven
t hasnt havent has what\$have had hadnt
t havent has what\$have had hadnt hasn
ve had hadnt hasnt havent has what\$ha
vent has what\$have had hadnt hasnt ha
what\$have had hadnt hasnt havent has

→ $B = \text{tedtttshhhhhhhaavv_w\$_edsaaannnaa_}$

MTF yields 8,5,5,2,0,0,8,7,0,0,0,0,0,7,0,9,0,8,0,0,0,10,9,2,9,9,8,7,0,0,10,0,0,1,0,5

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
 - Augments dictionary with repeated substrings
- MTF** Adaptive, transforms to smaller integers
 - should be followed by variable-width integer encoding
- BWT** Block compression method, should be followed by MTF