

# CS 240 – Data Structures and Data Management

## Module 10: Data Compression

Mark Petrick, Éric Schost

Based on lecture notes by many previous cs240 instructors

David R. Cheriton School of Computer Science, University of Waterloo

Spring 2024

# Outline

## 10 Data Compression

- Background
- Single-Character Encodings
- Huffman's Encoding Trie
- Run-Length Encoding
- Lempel-Ziv-Welch
- Combining Compression Schemes: bzip2
- Burrows-Wheeler Transform

# Outline

## 10 Data Compression

- Background
- Single-Character Encodings
- Huffman's Encoding Trie
- Run-Length Encoding
- Lempel-Ziv-Welch
- Combining Compression Schemes: bzip2
- Burrows-Wheeler Transform

# Data Compression Introduction

**The problem:** How to store and transmit data efficiently?

**Source text** The original data, string  $S$  of characters from the **source alphabet**  $\Sigma_S$

**Coded text** The encoded data, string  $C$  of characters from the **code alphabet**  $\Sigma_C$

**Encoding** An algorithm mapping source texts to coded texts

**Decoding** An algorithm mapping coded texts back to their original source text



- Source “text” can be any sort of data (not always text!)
- The code alphabet  $\Sigma_C$  is usually  $\{0, 1\}$ .
- We consider here only **lossless** compression: Can recover  $S$  from  $C$  without error.

# Judging Encoding Schemes

**Main objective:** For data compression, we want to minimize the size of the coded text. We will measure the **compression ratio**:

$$\frac{|C| \cdot \log |\Sigma_C|}{|S| \cdot \log |\Sigma_S|}$$

## Examples:

$73 = (73)_{10} \rightarrow (1001001)_2$  has compression ratio  $\frac{7 \cdot \log 2}{2 \cdot \log 10} \approx 1.05$

$127 = (127)_{10} \rightarrow (7F)_{16}$  has compression ratio  $\frac{2 \cdot \log 16}{3 \cdot \log 10} \approx 0.8$

# Judging Encoding Schemes

**Main objective:** For data compression, we want to minimize the size of the coded text. We will measure the **compression ratio**:

$$\frac{|C| \cdot \log |\Sigma_C|}{|S| \cdot \log |\Sigma_S|}$$

## Examples:

$73 = (73)_{10} \rightarrow (1001001)_2$  has compression ratio  $\frac{7 \cdot \log 2}{2 \cdot \log 10} \approx 1.05$

$127 = (127)_{10} \rightarrow (7F)_{16}$  has compression ratio  $\frac{2 \cdot \log 16}{3 \cdot \log 10} \approx 0.8$

We also consider the efficiency of the encoding/decoding algorithms.

- We always need time  $\Omega(|S| + |C|)$ , but sometimes need more

Other possible goals (not studied here  $\rightarrow$  CS489):

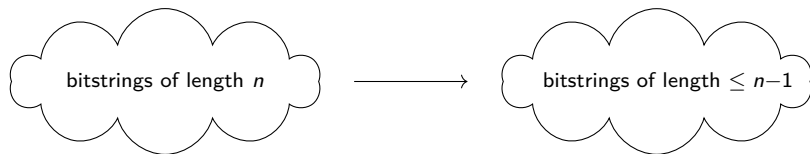
- Reliability (e.g. error-correcting codes)
- Security (e.g. encryption)

## Impossibility of compressing

**Observation:** No lossless encoding scheme can have compression ratio  $< 1$  for *all* input strings.

**Proof** (only for  $\Sigma_S = \Sigma_C = \{0, 1\}$ ): Fix one size  $n$ .

Assume for contradiction that all length- $n$  strings get shorter.

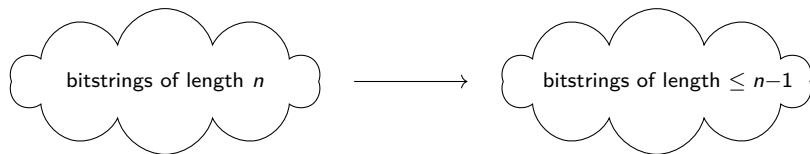


How big are these sets?

# Impossibility of compressing

**Observation:** No lossless encoding scheme can have compression ratio  $< 1$  for *all* input strings.

**Proof** (only for  $\Sigma_S = \Sigma_C = \{0, 1\}$ ): Fix one size  $n$ .  
Assume for contradiction that all length- $n$  strings get shorter.



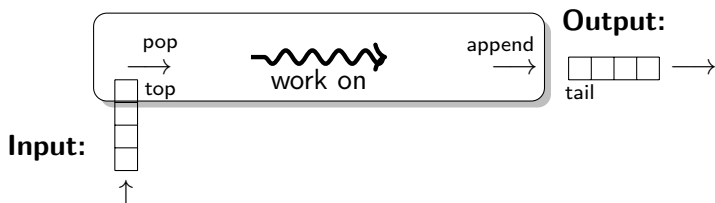
How big are these sets?

- So we cannot hope to prove good worst-case compression bounds.
- But real-life data can (usually) be compressed well due to patterns.



## Detour: Streams (Review)

The texts are often *huge* and will not fit onto the computer's memory. Therefore we usually use streams to store  $S$  and  $C$ .



- Input-stream ( $\sim$  `std::cin`): Read one character at a time. (We use a stack-like interface: *top/pop/is-empty*.)
- Output-stream ( $\sim$  `std::cout`): Write one character at a time. (We use queue-like interface: *append*.)
- Advantage: can start processing text while loading.
- Some algorithm will need to *reset* the input-stream.

# Outline

## 10 Data Compression

- Background
- **Single-Character Encodings**
- Huffman's Encoding Trie
- Run-Length Encoding
- Lempel-Ziv-Welch
- Combining Compression Schemes: bzip2
- Burrows-Wheeler Transform

## Character Encodings

A **character encoding** (or **single-character encoding**) maps each character in the source alphabet to a string in code alphabet.

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

**Example: ASCII**

NULL	...	!	...	0	...	A	...
0	...	21	...	30	...	65	...
0000000	...	0010101	...	0011110	...	1000001	...

- ASCII is a (length-7) **fixed-length code**: Each **codeword**  $E(c)$  has the same length.
- Encoding/Decoding is easy: just concatenate/decode the next 7 bits.

$APPLE \leftrightarrow (65, 80, 80, 76, 69) \leftrightarrow 1000001 \frown 1010000 \frown 1010000 \frown 1001100 \frown 1000101$

- Other (earlier) examples of fixed-length codes: Caesar shift, Baudot code, Murray code
- Fixed-length codes do not compress.

# Variable-Length Codes

## Better idea: Variable-length codes

- **Motivation:** Some letters in  $\Sigma_S$  occur more often than others.  
For example, consider the frequency of letters in typical English text:

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%		

# Variable-Length Codes

## Better idea: Variable-length codes

- **Motivation:** Some letters in  $\Sigma_S$  occur more often than others.  
For example, consider the frequency of letters in typical English text:

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%		

- **Idea:** Let's use shorter codes for more frequent characters.

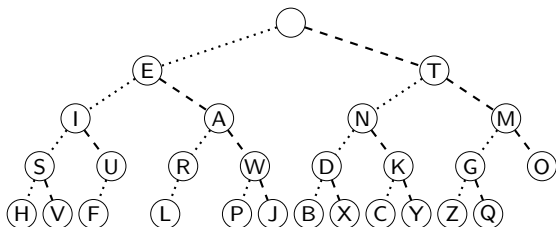
So as before, map source alphabet to codewords  $E : \Sigma_S \rightarrow \Sigma_C^*$

- But not all codewords must have the same length.
- This ought to make the code text shorter.

# Variable-Length Codes

## Example 1: Morse code.

A	• —	N	— •
B	— • • •	O	— — —
C	— • • •	P	• — — •
D	— • •	Q	— — • —
E	•	R	• — •
F	• • — •	S	• • •
G	— — •	T	—
H	• • • •	U	• • —
I	• •	V	• • • —
J	• — — —	W	— • —
K	— • •	X	— • • •
L	• • • •	Y	— • — —
M	— —	Z	— — • •



## Example 2: UTF-8 encoding of Unicode (~ 150,000 characters):

- Encodes any Unicode character using 1-4 bytes

# Encoding

Assume that we have some character encoding  $E : \Sigma_S \rightarrow \Sigma_C^*$ .

- Note that  $E$  is a dictionary with keys in  $\Sigma_S$ .

*singleChar::encoding*( $E, S, C$ )

$E$  : the encoding dictionary

$S$ : input-stream with characters in  $\Sigma_S$ ,  $C$ : output-stream of bits

1. **while**  $S$  is non-empty
2.      $w \leftarrow E.\text{search}(S.\text{pop}())$
3.     append each bit of  $w$  to  $C$

**Example:** Encode AN<sub>⊔</sub>ANT with the following code:

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

AN<sub>⊔</sub>ANT →

# Encoding

Assume that we have some character encoding  $E : \Sigma_S \rightarrow \Sigma_C^*$ .

- Note that  $E$  is a dictionary with keys in  $\Sigma_S$ .

*singleChar::encoding*( $E, S, C$ )

$E$  : the encoding dictionary

$S$ : input-stream with characters in  $\Sigma_S$ ,  $C$ : output-stream of bits

1. **while**  $S$  is non-empty
2.      $w \leftarrow E.\text{search}(S.\text{pop}())$
3.     append each bit of  $w$  to  $C$

**Example:** Encode AN<sub>␣</sub>ANT with the following code:

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

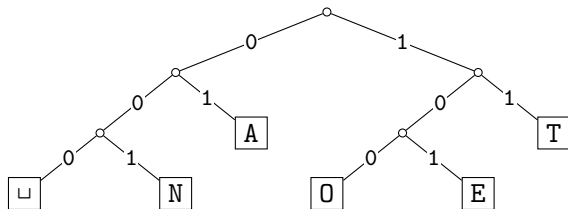
AN<sub>␣</sub>ANT  $\rightarrow$  010010000100111



# Decoding

The **decoding algorithm** must map  $\Sigma_C^*$  to  $\Sigma_S^*$ .

- The code must be lossless, i.e., *uniquely decodable*.
  - ▶ This is false for Morse code as described!
    - **— — — • — — —** decodes to WATT and ANO and WJ.  
(Morse code uses pause as end-sentinel to avoid ambiguity.)
- From now on only consider **prefix-free** codes  $E$ :  
no codeword is a prefix of another
- This corresponds to a *trie* with characters of  $\Sigma_S$  only at the leaves.



- The codewords need no end-sentinel \$ if codes are prefix-free.

# Decoding of Prefix-Free Codes

Any prefix-free code is uniquely decodable.

```
prefixFree::decoding(C, S, T)
```

C: input-stream with characters in  $\Sigma_C$ , S: output-stream, T : encoding-trie

1. **while** C is non-empty
2.      $z \leftarrow T.root$
3.     **while** z is not a leaf
4.         **if** C is empty or z has no child labelled  $C.top()$
5.             **return** "invalid encoding"
6.          $z \leftarrow$  child of z that is labelled with  $C.pop()$
7.     S.*append*(character stored at z)

Run-time:  $O(|C|)$ .

**Example:** Decode 111000001010111

## Decoding of Prefix-Free Codes

Any prefix-free code is uniquely decodable.

```
prefixFree::decoding(C, S, T)
```

C: input-stream with characters in  $\Sigma_C$ , S: output-stream, T : encoding-trie

1. **while** C is non-empty
2.      $z \leftarrow T.root$
3.     **while** z is not a leaf
4.         **if** C is empty or z has no child labelled  $C.top()$
5.             **return** “invalid encoding”
6.          $z \leftarrow$  child of z that is labelled with  $C.pop()$
7.     S.*append*(character stored at z)

Run-time:  $O(|C|)$ .

**Example:** Decode 111000001010111  $\rightarrow$  TO<sub>U</sub>EAT

## Encoding from the Trie

We already explained encoding if code is stored in table:

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

**Example:** Encode AN $\sqcup$ ANT  $\rightarrow$  010010000100111

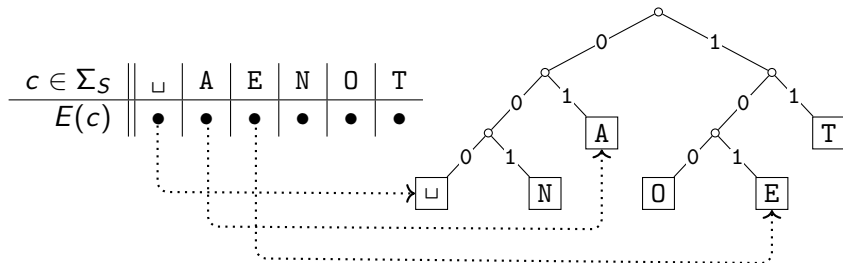
## Encoding from the Trie

We already explained encoding if code is stored in table:

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

**Example:** Encode  $AN\sqcup ANT \rightarrow 010010000100111$

- The table wastes space: Codewords may be quite long.
- Better idea: Store codewords via links to leaves in trie.



## Encoding from trie

*prefixFree::encoding*( $S, C, T$ )

$S$ : input-stream with characters in  $\Sigma_S$ ,  $C$ : output-stream,  $T$ : encoding trie

1.  $E \leftarrow$  array of trie-nodes indexed by  $\Sigma_S$
2. **for** all leaves  $\ell$  in  $T$  **do**  $E[\text{character at } \ell] \leftarrow \ell$
3. **while**  $S$  is non-empty
4.      $w \leftarrow$  empty bitstring
5.      $v \leftarrow E[S.\text{pop}()]$
6.     **while**  $v$  is not the root
7.          $w.\text{prepend}(\text{character from } v \text{ to its parent})$
8.     append each bit of  $w$  to  $C$

- Run-time:  $O(|T| + |C|)$
- We assume that all interior nodes of  $T$  have two children, otherwise encoding scheme can be improved (how?)
- Therefore  $|T| \leq 2|\Sigma_S| - 1$  and run-time is  $O(|\Sigma_S| + |C|)$

# Outline

## 10 Data Compression

- Background
- Single-Character Encodings
- **Huffman's Encoding Trie**
- Run-Length Encoding
- Lempel-Ziv-Welch
- Combining Compression Schemes: bzip2
- Burrows-Wheeler Transform

# Huffman's Algorithm: Building the best trie

How to determine the “best” trie (for a given source text  $S$ )?

**Idea:** Frequent characters should have short codewords.

Equivalently: Infrequent characters should be ‘far down’ in trie.



# Huffman's Algorithm: Building the best trie

How to determine the “best” trie (for a given source text  $S$ )?

**Idea:** Frequent characters should have short codewords.

Equivalently: Infrequent characters should be ‘far down’ in trie.

**Greedy-algorithm:** Always pair up most infrequent characters.

- 1 We store a set of encoding-tries.  
Initially each  $c \in \Sigma_S$  adds “ $\boxed{c}$ ” (height-0 trie holding  $c$ ).
- 2 Our tries have a *weight*: sum of frequencies of all letters in trie.  
(We assume character-frequencies are pre-computed.)
- 3 Find the two tries with the minimum weight and merge them.  
(Corresponds to adding one bit to the encoding of each character.)
- 4 Repeat Step 3 until there is only one trie left

How should we store the tries to make this efficient?

A min-ordered heap! Step 3 needs two *delete-mins* and one *insert*

## Example: Huffman tree construction

Example text: GREENENERGY,  $\Sigma_S = \{G, R, E, N, Y\}$

Character frequencies: G : 2, R : 2, E : 4, N : 2 Y : 1

List of tries:

2  
G

2  
R

4  
E

2  
N

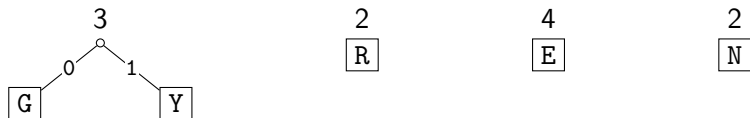
1  
Y

## Example: Huffman tree construction

Example text: GREENENERGY,  $\Sigma_S = \{G, R, E, N, Y\}$

Character frequencies: G : 2, R : 2, E : 4, N : 2 Y : 1

List of tries:

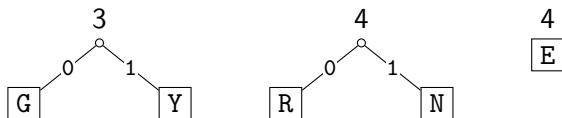


## Example: Huffman tree construction

Example text: GREENENERGY,  $\Sigma_S = \{G, R, E, N, Y\}$

Character frequencies: G : 2, R : 2, E : 4, N : 2, Y : 1

List of tries:

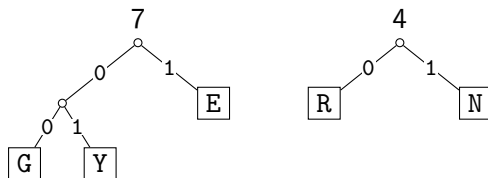


## Example: Huffman tree construction

Example text: GREENENERGY,  $\Sigma_S = \{G, R, E, N, Y\}$

Character frequencies: G : 2, R : 2, E : 4, N : 2, Y : 1

List of tries:

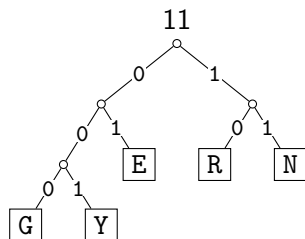


## Example: Huffman tree construction

Example text: GREENENERGY,  $\Sigma_S = \{G, R, E, N, Y\}$

Character frequencies: G : 2, R : 2, E : 4, N : 2 Y : 1

List of tries:



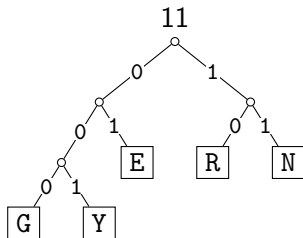
GREENENERGY →

## Example: Huffman tree construction

Example text: GREENENERGY,  $\Sigma_S = \{G, R, E, N, Y\}$

Character frequencies: G : 2, R : 2, E : 4, N : 2 Y : 1

List of tries:



GREENENERGY  $\rightarrow$  0000100101110111011000001

Compression ratio:  $\frac{25}{11 \cdot \log 5} \approx 97\%$

# Huffman's Algorithm: Pseudocode

*Huffman::encoding*( $S, C$ )

$S$ : input-stream with characters in  $\Sigma_S$ ,  $C$ : output-stream

1.  $f \leftarrow$  array indexed by  $\Sigma_S$ , initially all-0 // frequencies
2. **while**  $S$  is non-empty **do** increase  $f[S.pop()]$  by 1
3.  $Q \leftarrow$  min-oriented priority queue that stores tries // initialize PQ
4. **for** all  $c \in \Sigma_S$  with  $f[c] > 0$  **do**  $Q.insert(\boxed{c}, f[c])$
5. **while**  $Q.size() > 1$  **do** // build decoding trie
6.      $(T_1, f_1) \leftarrow Q.delete-min()$
7.      $(T_2, f_2) \leftarrow Q.delete-min()$
8.      $Q.insert(\text{trie with } T_1, T_2 \text{ as subtrees}, f_1+f_2)$
9.      $T \leftarrow Q.delete-min()$
10.  $C.append(\text{encoding trie } T)$  // send trie
11. Re-set input-stream  $S$
12. *prefixFree::encoding*( $S, C, T$ ) // actual encoding

This assumes that  $S$  has at least two distinct characters.



# Huffman Coding Discussion

**Can show:** The constructed trie is *optimal* in the sense that  $C$  is shortest (among prefix-free single-character encodings with  $\Sigma_C = \{0, 1\}$ ).

But:

- Constructed trie is *not unique* (unless we give tie-breaking rules).
- Decoder does not know the trie:
  - ▶ Either send decoding trie along (how to convert to bitstring?)
  - ▶ Or send character-frequencies and tie-breaking rules.
  - ▶ Either way, this adds to length of encoded text.
- *encoding* must pass through text twice (to compute frequencies and to encode). Cannot use a stream unless it can be re-set.

Run-time:

- Encoding:  $O(|\Sigma_S| \log |\Sigma_S| + |C|)$
- Decoding:  $O(|C|)$

Many variations (what to do with unused characters, estimate frequencies, adaptively change encoding, ....)

# Outline

## 10 Data Compression

- Background
- Single-Character Encodings
- Huffman's Encoding Trie
- **Run-Length Encoding**
- Lempel-Ziv-Welch
- Combining Compression Schemes: bzip2
- Burrows-Wheeler Transform

# Run-Length Encoding Idea

- Example of **multi-character encoding**: multiple source-text characters receive one code-word.
- **Requires**: Input must be a bitstring ( $\Sigma_S = \{0, 1\}$ )
- Decoding dictionary is uniquely defined and not explicitly stored.

**When to use**: if  $S$  has long runs:  $\underbrace{00000}_5 \underbrace{111}_3 \underbrace{0000000}_7$

## Encoding idea:

- Send the leftmost bit of  $S$  (either 0 or 1)
- Then send a sequence of integers indicating run lengths.
- We don't have to give the bit for runs since they alternate.

Example becomes: 0, 5, 3, 7

**Problem**: How do we 'send integers'? Base-10 representations plus commas? Can we do better?

## Detour: Encoding Positive Integers

Various ideas for encoding one integer  $k$ :

- **Base-2 encoding:**  $E_{b2}(k) :=$  the string  $w$  such that  $(w)_2 = k$

- **Bijjective base-2 encoding:**

$E_{bb2}(k) := E_{b2}(k+1)$  with the leading 1 removed

(Bijjective base-26 encoding used to enumerate spreadsheet-columns.)

$k$	1	2	3	4	5	6	7	8	...
$E_{b2}(k)$	1	10	11	100	101	110	111	1000	...
$E_{bb2}(k)$	0	1	00	01	10	11	000	001	...
	A	B	AA	AB	BA	BB	AAA	AAB	...

## Detour: Encoding Positive Integers

Various ideas for encoding one integer  $k$ :

- **Base-2 encoding:**  $E_{b2}(k) :=$  the string  $w$  such that  $(w)_2 = k$

- **Bijjective base-2 encoding:**

$$E_{bb2}(k) := E_{b2}(k+1) \text{ with the leading 1 removed}$$

(Bijjective base-26 encoding used to enumerate spreadsheet-columns.)

$k$	1	2	3	4	5	6	7	8	...
$E_{b2}(k)$	1	10	11	100	101	110	111	1000	...
$E_{bb2}(k)$	0	1	00	01	10	11	000	001	...
	A	B	AA	AB	BA	BB	AAA	AAB	...

For a sequence of integers, we need a separator-symbol (e.g. #):

- $5, 3, 7 \rightarrow 101\#11\#111$  has  $|C| \log |\Sigma_C| = 10 \log(3) \approx 15.8$
- $5, 3, 7 \rightarrow BA\#AA\#AAA$  has  $|C| \log |\Sigma_C| = 9 \log(3) \approx 14.22$

Can we develop codes that do not need a separator-symbol?

## Run-length encoding details

**Elias gamma code** (for an integer  $k$ ):

- To encode  $k$ , take binary representation  $E_{b_2}(k)$  of  $k$
- Add leading zeroes so that initial 1 is in the middle
- This means adding  $\lfloor \log k \rfloor$  leading zeroes

$k$	1	2	3	4	5	6	7	8	...
$E_{b_2}(k)$	1	10	11	100	101	110	111	1000	...
$\lfloor \log k \rfloor$	0	1	1	2	2	2	2	3	...
$E_\gamma(k)$	1	010	011	00100	00101	00110	00111	0001000	...

## Run-length encoding details

**Elias gamma code** (for an integer  $k$ ):

- To encode  $k$ , take binary representation  $E_{b_2}(k)$  of  $k$
- Add leading zeroes so that initial 1 is in the middle
- This means adding  $\lfloor \log k \rfloor$  leading zeroes

$k$	1	2	3	4	5	6	7	8	...
$E_{b_2}(k)$	1	10	11	100	101	110	111	1000	...
$\lfloor \log k \rfloor$	0	1	1	2	2	2	2	3	...
$E_\gamma(k)$	1	010	011	00100	00101	00110	00111	0001000	...

**Definition:** To do **run-length encoding**:

- 1 Write initial bit to output.
- 2 Determine the run-lengths  $k_1, k_2, \dots, k_d$
- 3 Write  $E_\gamma(k_1) \frown E_\gamma(k_2) \frown \dots \frown E_\gamma(k_d)$  to output

Steps 2-3 can be done simultaneously.

# RLE Encoding

*RLE::encoding*(*S*, *C*)

*S*: input-stream of bits, *C*: output-stream

1.  $b \leftarrow S.top()$ ;  $C.append(b)$
2. **while** *S* is non-empty **do**
3.      $k \leftarrow 1$                      // *k*: length of run
4.     **while** (*S* is non-empty and  $S.top() = b$ ) **do**  $k++$ ;  $S.pop()$
5.      $w \leftarrow$  empty string     // *w*: binary encoding of *k*
6.     **while**  $k > 1$
7.          $C.append(0)$      // write leading zeroes
8.          $w.prepend(k \bmod 2)$
9.          $k \leftarrow \lfloor k/2 \rfloor$
10.      $w.prepend(1)$
11.     append each bit of *w* to *C*
12.      $b \leftarrow 1 - b$



## RLE Example

Encoding:  $S = \underbrace{00000}_{5} \underbrace{111}_{3} \underbrace{0000000}_{7}$

run-lengths:

$C =$  has  $|C| \log |\Sigma_C| = \dots$   
(better than other schemes)

Decoding:  $C = 0\ 0\ 0\ 0\ 1\ 1\ 0\ 1\ 0\ 0\ 1\ 0\ 0\ 1\ 0\ 1\ 0:$

run-lengths:

$S =$

## RLE Decoding

**Claim:** A sequence of Elias gamma codes can be decoded uniquely.

- Each Elias gamma code has form  $0^\ell 1 *^\ell$  (where ‘\*’ means ‘some bit’)
- Can determine  $\ell$  by scanning until we encounter a 1.
- Convert this 1 plus the next  $\ell$  bits into the integer.

*RLE::decoding*( $C, S$ )

$C$ : input-stream of bits ,  $S$ : output-stream

1.  $b \leftarrow C.\text{pop}()$  //  $b$ : bit of the current run
2. **while**  $C$  is non-empty
3.      $\ell \leftarrow 0$  //  $\ell$ : number of leading zeroes
4.     **while**  $C.\text{pop}() = 0$  **do**  $\ell++$
5.      $k \leftarrow 1$  //  $k$ : integer that was encoded
6.     **for** ( $j \leftarrow 1$  to  $\ell$ ) **do**  $k \leftarrow k * 2 + C.\text{pop}()$
7.     **for** ( $j \leftarrow 1$  to  $k$ ) **do**  $S.\text{append}(b)$
8.      $b \leftarrow 1 - b$

If  $C.\text{pop}()$  is called when there are no bits left, then  $C$  was not valid input.

# RLE Discussion

- An all-0 string of length  $n$  would be compressed to  $0E_\gamma(n)$ , which has  $2\lfloor \log n \rfloor + 2 \in o(n)$  bits.
- Usually, we are not that lucky:
  - ▶ No compression until run-length  $k \geq 6$
  - ▶ *Expansion* when run-length  $k = 2$  or  $4$
- RLE was popular for fax-machines.
- RLE can be adapted to larger alphabet sizes (but then the encoding of each run must also store the character)  
Used in some image formats (e.g. TIFF)
- Method can be adapted to encode *only* runs of 0  
Used inside bzip2 (see later)

# Outline

## 10 Data Compression

- Background
- Single-Character Encodings
- Huffman's Encoding Trie
- Run-Length Encoding
- **Lempel-Ziv-Welch**
- Combining Compression Schemes: bzip2
- Burrows-Wheeler Transform

# Longer Patterns in Input

Huffman and RLE take advantage of frequent/repeated *single characters*.

**Observation:** Certain *substrings* are much more frequent than others.

- English text:  
Most frequent digraphs: TH, ER, ON, AN, RE, HE, IN, ED, ND, HA  
Most frequent trigrams: THE, AND, THA, ENT, ION, TIO, FOR, NDE
- HTML: “<a href”, “<img src”, “<br>”
- Video: repeated background between frames, shifted sub-image

**Ingredient 1** for Lempel-Ziv-Welch compression: take advantage of such substrings *without* needing to know beforehand what they are.

# Adaptive Dictionaries

ASCII, UTF-8, and RLE use *fixed* dictionaries.

In Huffman, the dictionary is not fixed, but it is *static*: the dictionary is the same for the entire encoding/decoding.

**Ingredient 2** for LZW: *adaptive encoding*:

- There is a fixed initial dictionary  $D_0$ . (Usually ASCII.)
- For  $i \geq 0$ ,  $D_i$  is used to determine the  $i$ th output character
- After writing the  $i$ th character to output, encoder updates  $D_i$  to  $D_{i+1}$

**Challenge:** Decoder must know (or reconstruct from coded text) how encoder changed the dictionary.

# LZW Overview

For now: convert  $S$  into a list of code-numbers.

- Start with dictionary  $D_0$  that stores ASCII (code-numbers  $0, \dots, 127$ ).
- Every step adds to dictionary a multi-character string, using code-numbers  $128, 129, \dots$
- Encoding alternates two steps:
  - 1 *Find longest string*  $w$  (among remaining characters) that is already in  $D_i$ . So all of  $w$  can be encoded with one number.
  - 2 *Add the substring that would have been useful* to dictionary: add  $wc$  where  $c$  is the character that follows  $w$  in  $S$ .

# LZW Overview

For now: convert  $S$  into a list of code-numbers.

- Start with dictionary  $D_0$  that stores ASCII (code-numbers  $0, \dots, 127$ ).
- Every step adds to dictionary a multi-character string, using code-numbers  $128, 129, \dots$
- Encoding alternates two steps:
  - 1 *Find longest string*  $w$  (among remaining characters) that is already in  $D_i$ . So all of  $w$  can be encoded with one number.
  - 2 *Add the substring that would have been useful* to dictionary: add  $wc$  where  $c$  is the character that follows  $w$  in  $S$ .

What data structure should the dictionary use?



# LZW Overview

For now: convert  $S$  into a list of code-numbers.

- Start with dictionary  $D_0$  that stores ASCII (code-numbers  $0, \dots, 127$ ).
- Every step adds to dictionary a multi-character string, using code-numbers  $128, 129, \dots$
- Encoding alternates two steps:
  - ① *Find longest string*  $w$  (among remaining characters) that is already in  $D_i$ . So all of  $w$  can be encoded with one number.
  - ② *Add the substring that would have been useful* to dictionary: add  $wc$  where  $c$  is the character that follows  $w$  in  $S$ .

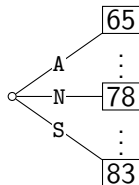
What data structure should the dictionary use?

- Need to match characters  $\rightarrow$  use a trie
- To find  $w$ : parse in trie until we reach node  $z$  with 'no such child' for next character  $c$
- To add to  $D_i$ : add child of  $z$  with link labelled  $c$

# LZW Example

Text:        A    N    A    N    A    S    A    N    N    A

Dictionary:



(Parts of dictionary not shown)

# LZW Example

128:AN

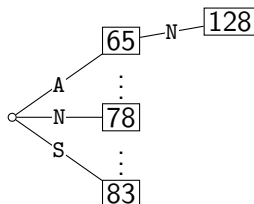
Text:

A N A N A S A N N A

Encoding:

65

Dictionary:



# LZW Example

128:AN 129:NA

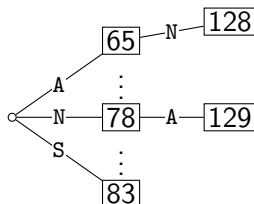
Text:

A N A N A S A N N A

Encoding:

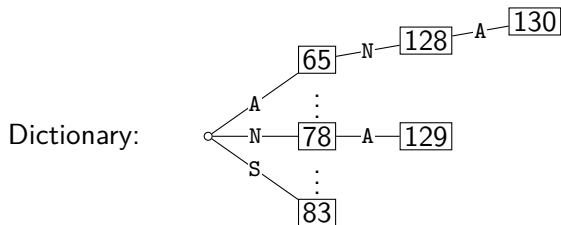
65 78

Dictionary:

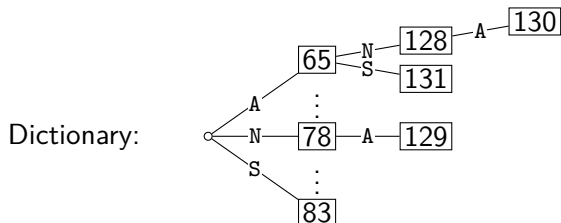
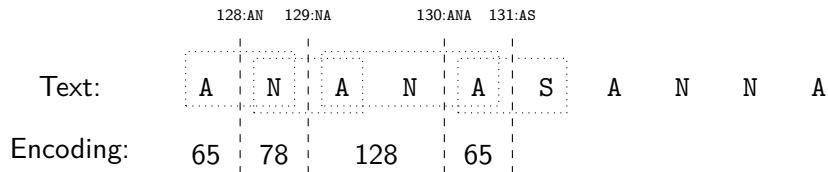


# LZW Example

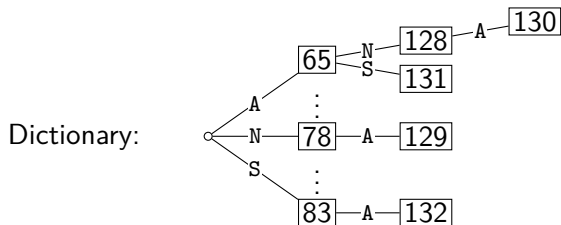
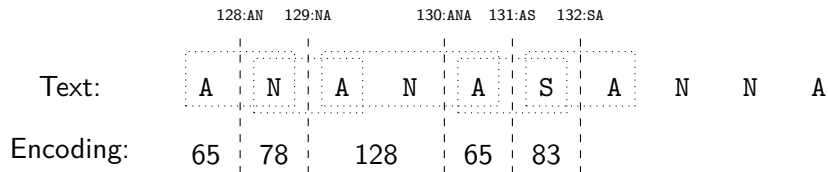
	128:AN	129:NA	130:ANA							
Text:	A	N	A	N	A	S	A	N	N	A
Encoding:	65	78	128							



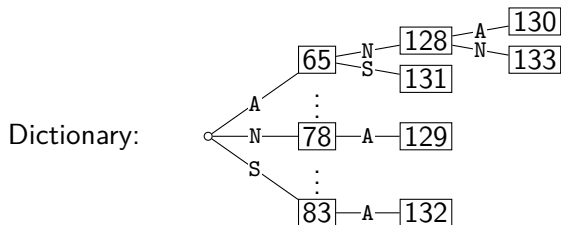
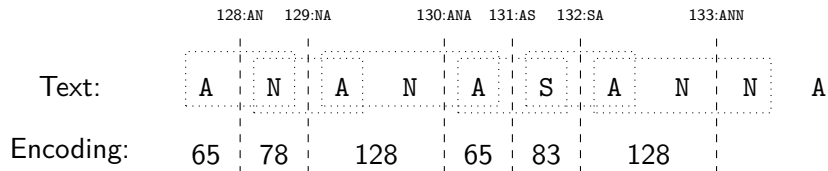
# LZW Example



# LZW Example



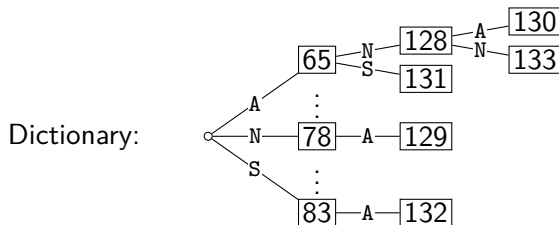
# LZW Example





# LZW Example

	128:AN	129:NA	130:ANA	131:AS	132:SA	133:ANN				
Text:	A	N	A	N	A	S	A	N	N	A
Encoding:	65	78	128	65	83	128			129	



## LZW encoding pseudocode

*LZW::encoding*( $S, C$ )

$S$ : input-stream of characters,  $C$ : output-stream of integers

1. Initialize dictionary  $D$  with ASCII in a trie
2.  $idx \leftarrow 128$
3. **while**  $S$  is non-empty **do**
4.      $z \leftarrow$  root of trie  $D$      // find longest string
5.     **while** ( $S$  is non-empty and  $z$  has a child  $c$  labelled  $S.top()$ )
6.          $z \leftarrow c$ ;  $S.pop()$
7.      $C.append$ (code-number stored at  $z$ )
8.     **if**  $S$  is non-empty     // add to dictionary
9.         create child of  $z$  labelled  $S.top()$  with code-number  $idx$
10.      $idx++$

Run-time:  $O(|S|)$ , assuming we can look up child in  $O(1)$  time.

# LZW decoding

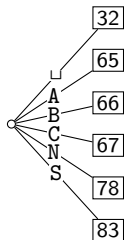
- Build dictionary while reading string by imitating encoder.
- We are one step behind.

67 65 78 32 66 129 133 83

- Example:

ASCII	
...	
32	␣
...	
...	
65	A
66	B
67	C
...	
78	N
...	
83	S
...	

What encoder did:



# LZW decoding

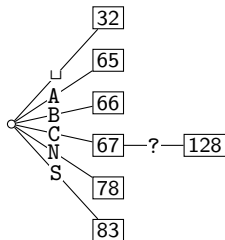
- Build dictionary while reading string by imitating encoder.
- We are one step behind.

67 65 78 32 66 129 133 83

- Example: C

ASCII	
...	
32	␣
...	
...	
65	A
66	B
67	C
...	
78	N
...	
83	S
...	

What encoder did:



# LZW decoding

- Build dictionary while reading string by imitating encoder.
- We are one step behind.

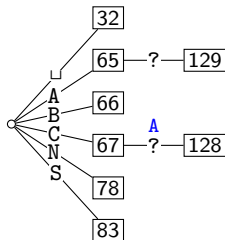
67 65 78 32 66 129 133 83

- Example: C A

ASCII	
...	
32	□
...	
...	
65	A
66	B
67	C
...	
78	N
...	
83	S
...	

What encoder did:

Deduced one step later:



# LZW decoding

- Build dictionary while reading string by imitating encoder.
- We are one step behind.

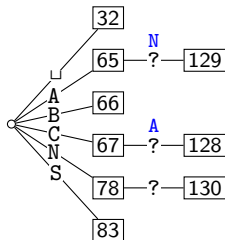
67 65 78 32 66 129 133 83

- Example: C A N

ASCII	
...	
32	□
...	
...	
65	A
66	B
67	C
...	
78	N
...	
83	S
...	

What encoder did:

Deduced one step later:



# LZW decoding

- Build dictionary while reading string by imitating encoder.
- We are one step behind.

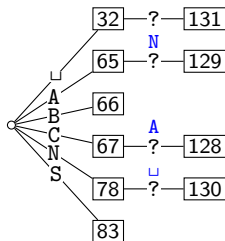
67 65 78 32 66 129 133 83

- Example: C A N □

ASCII	
...	
32	□
...	
...	
65	A
66	B
67	C
...	
78	N
...	
83	S
...	

What encoder did:

Deduced one step later:



# LZW decoding

- Build dictionary while reading string by imitating encoder.
- We are one step behind.

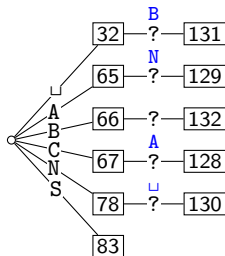
67 65 78 32 66 129 133 83

- Example: C A N □ B

ASCII	
...	
32	□
...	
...	
65	A
66	B
67	C
...	
78	N
...	
83	S
...	

What encoder did:

Deduced one step later:





# LZW decoding

- Build dictionary while reading string by imitating encoder.
- We are one step behind.

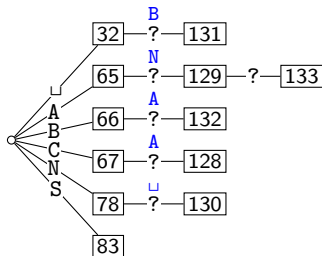
67 65 78 32 66 129 133 83

- Example: C A N □ B AN

ASCII	
...	
32	□
...	
...	
65	A
66	B
67	C
...	
78	N
...	
83	S
...	

What encoder did:

Deduced one step later:



# LZW decoding

- Build dictionary while reading string by imitating encoder.
- We are one step behind.

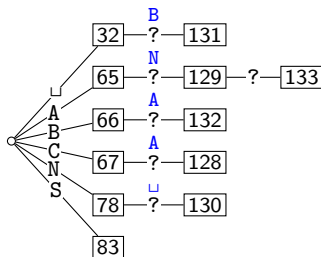
67 65 78 32 66 129 133 83

- Example: C A N  $\square$  B AN ???

ASCII	
...	
32	$\square$
...	
...	
65	A
66	B
67	C
...	
78	N
...	
83	S
...	

What encoder did:

Deduced one step later:



## LZW decoding pseudocode – incomplete

*LZW::decoding*( $C, S$ )

$C$ : input-stream of integers,  $S$ : output-stream

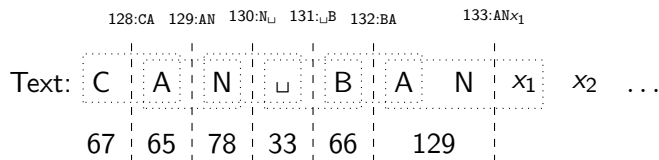
1.  $D \leftarrow$  dictionary that maps  $\{0, \dots, 127\}$  to ASCII
2.  $idx \leftarrow 128$
3.  $k \leftarrow C.pop()$ ;  $s \leftarrow D.search(k)$ ;  $S.append(s)$
4. **while** there are more codes in  $C$  **do**
5.      $s_{prev} \leftarrow s$ ;  $k \leftarrow C.pop()$
6.     **if**  $k < idx$  **do**  $s \leftarrow D.search(k)$
7.     **else** ???
- 8.
9.     append each character of  $s$  to  $S$
10.      $D.insert(idx, s_{prev} \frown s[0])$
11.      $idx++$

## LZW decoding: the catch

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

## LZW decoding: the catch

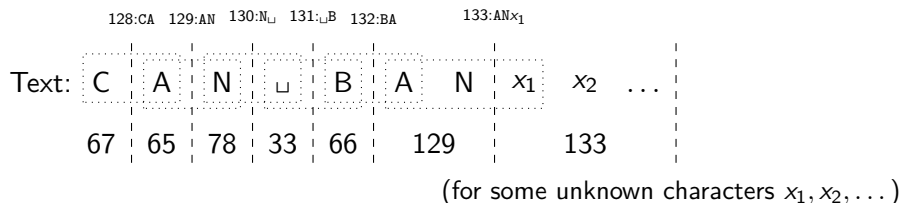
- In this example: Want to decode 133, but not yet in dictionary!
- What happened during the corresponding encoding?



(for some unknown characters  $x_1, x_2, \dots$ )

## LZW decoding: the catch

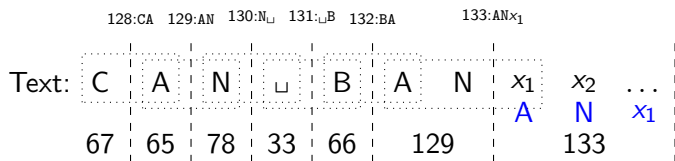
- In this example: Want to decode 133, but not yet in dictionary!
- What happened during the corresponding encoding?



- We know: 133 encodes AN $x_1$

## LZW decoding: the catch

- In this example: Want to decode 133, but not yet in dictionary!
- What happened during the corresponding encoding?

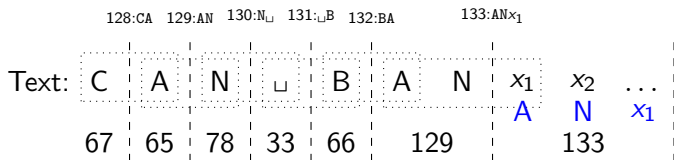


(for some unknown characters  $x_1, x_2, \dots$ )

- We know: 133 encodes AN $x_1$
- We know: Next step uses 133 = AN $x_1$

## LZW decoding: the catch

- In this example: Want to decode 133, but not yet in dictionary!
- What happened during the corresponding encoding?



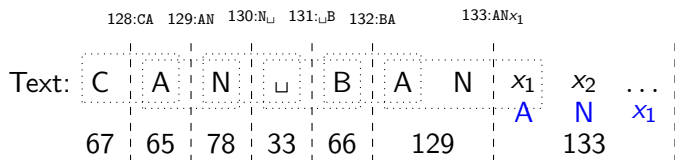
(for some unknown characters  $x_1, x_2, \dots$ )

- We know: 133 encodes AN $x_1$
- We know: Next step uses 133 = AN $x_1$
- So  $x_1 = A$  and 133 encodes ANA



## LZW decoding: the catch

- In this example: Want to decode 133, but not yet in dictionary!
- What happened during the corresponding encoding?



(for some unknown characters  $x_1, x_2, \dots$ )

- We know: 133 encodes AN $x_1$
- We know: Next step uses 133 = AN $x_1$
- So  $x_1 = A$  and 133 encodes ANA

Generally: If code number is about to be added to  $D$ , then it encodes

“previous string  $\cup$  first character of previous string”

# LZW decoding pseudocode

*LZW::decoding*( $C, S$ )

$C$ : input-stream of integers,  $S$ : output-stream

1.  $D \leftarrow$  dictionary that maps  $\{0, \dots, 127\}$  to ASCII
2.  $idx \leftarrow 128$
3.  $k \leftarrow C.pop()$ ;  $s \leftarrow D.search(k)$ ;  $S.append(s)$
4. **while** there are more codes in  $C$  **do**
5.      $s_{prev} \leftarrow s$ ;  $k \leftarrow C.pop()$
6.     **if**  $k < idx$  **do**  $s \leftarrow D.search(k)$
7.     **else if**  $k = idx$  **do**  $s \leftarrow s_{prev} \frown s_{prev}[0]$  // special situation
8.     **else return** FAIL // invalid encoding!
9.     append each character of  $s$  to  $S$
10.      $D.insert(idx, s_{prev} \frown s[0])$
11.      $idx++$

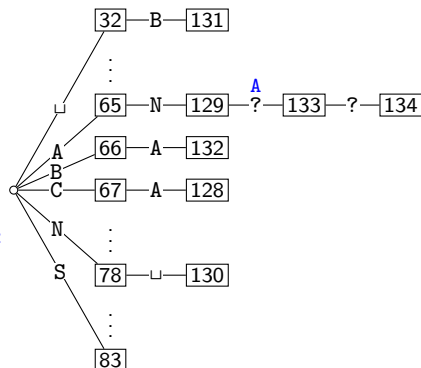
# LZW decoding example revisited

67 65 78 32 66 129 133  
C A N □ B AN ANA

ASCII	
...	
32	□
...	
...	
65	A
66	B
67	C
...	
78	N
...	
83	S
...	

What encoder did:

Deduced one step later:



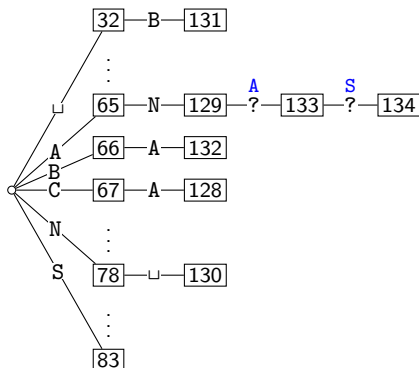
# LZW decoding example revisited

67 65 78 32 66 129 133 83  
C A N □ B AN ANA S

ASCII	
...	
32	□
...	
...	
65	A
66	B
67	C
...	
78	N
...	
83	S
...	

What encoder did:

Deduced one step later:



## LZW decoding - second example

98 97 114 128 114 97 131 134 129 101 110  
b a r ba r a

- No need to build a trie; store dictionary  $D$  as array.

$D$ (ASCII)	
...	
97	a
98	b
...	
101	e
...	
110	n
...	
114	r
...	

		Dictionary $D$ continued	
input	decodes to	Code #	String
98	b		
97	a	128	ba
114	r	129	ar
128	ba	130	rb
114	r	131	bar
97	a	132	ra
		133	
		134	
		135	
		136	
		137	

## LZW decoding - second example

98 97 114 128 114 97 131 134 129 101 110  
b a r ba r a **bar**

- No need to build a trie; store dictionary  $D$  as array.

$D$ (ASCII)	
...	
97	a
98	b
...	
101	e
...	
110	n
...	
114	r
...	

		Dictionary $D$ continued	
input	decodes to	Code #	String
98	b		
97	a	128	ba
114	r	129	ar
128	ba	130	rb
114	r	131	bar
97	a	132	ra
131	bar	133	ab
		134	
		135	
		136	
		137	

## LZW decoding - second example

98 97 114 128 114 97 131 134 129 101 110  
b a r ba r a bar barb

- No need to build a trie; store dictionary  $D$  as array.

$D$ (ASCII)	
...	
97	a
98	b
...	
101	e
...	
110	n
...	
114	r
...	

		Dictionary $D$ continued	
input	decodes to	Code #	String
98	b		
97	a	128	ba
114	r	129	ar
128	ba	130	rb
114	r	131	bar
97	a	132	ra
131	bar	133	ab
134	barb	134	barb
		135	
		136	
		137	

## LZW decoding - second example

98 97 114 128 114 97 131 134 129 101 110  
b a r ba r a bar barb ar

- No need to build a trie; store dictionary  $D$  as array.

$D$ (ASCII)	
...	
97	a
98	b
...	
101	e
...	
110	n
...	
114	r
...	

		Dictionary $D$ continued	
input	decodes to	Code #	String
98	b		
97	a	128	ba
114	r	129	ar
128	ba	130	rb
114	r	131	bar
97	a	132	ra
131	bar	133	ab
134	barb	134	barb
129	ar	135	barba
		136	
		137	



## LZW decoding - second example

98 97 114 128 114 97 131 134 129 101 110  
b a r ba r a bar barb ar e

- No need to build a trie; store dictionary  $D$  as array.

$D$ (ASCII)	
...	
97	a
98	b
...	
101	e
...	
110	n
...	
114	r
...	

		Dictionary $D$ continued	
input	decodes to	Code #	String
98	b		
97	a	128	ba
114	r	129	ar
128	ba	130	rb
114	r	131	bar
97	a	132	ra
131	bar	133	ab
134	barb	134	barb
129	ar	135	barba
101	e	136	are
		137	

## LZW decoding - second example

98 97 114 128 114 97 131 134 129 101 110  
b a r ba r a bar barb ar e n

- No need to build a trie; store dictionary  $D$  as array.

$D$ (ASCII)	
...	
97	a
98	b
...	
101	e
...	
110	n
...	
114	r
...	

		Dictionary $D$ continued	
input	decodes to	Code #	String
98	b		
97	a	128	ba
114	r	129	ar
128	ba	130	rb
114	r	131	bar
97	a	132	ra
131	bar	133	ab
134	barb	134	barb
129	ar	135	barba
101	e	136	are
110	n	137	en

## Lempel-Ziv-Welch decoding details

- Dictionary  $D$  maps consecutive integers to words. Use an array!
- Run-time:  $O(|S|)$ .
  - ▶  $\Theta(|s|)$  each round to append  $s$  to output.
  - ▶ Everything else takes no longer
- Dictionary wastes space: words may get long
  - ▶ To save space, store string as code of prefix + one character.
  - ▶ Can still look up  $s$  in  $O(|s|)$  time.
  - ▶ So run-time remains  $O(|S|)$

*LZW::dictionary-lookup*( $D, k$ )

1.  $s \leftarrow$  empty word
2. **while**  $k > 127$  **do**  $(k, c) \leftarrow D[k], s.\textit{prepend}(c)$
3.  $s.\textit{prepend}(D[k])$

# LZW decoding - second example revisited

98 97 114 128 114 97 131 134 129 101 110  
b a r ba r a bar barb ar e n

<i>D</i> (ASCII)	
...	
97	a
98	b
...	
101	e
...	
110	n
...	
114	r
...	

Dictionary <i>D</i> continued				
input	decodes to	Code #	String (human)	String (computer)
98	b			
97	a	128	ba	98, a
114	r	129	ar	97, r
128	ba	130	rb	114, b
114	r	131	bar	128, r
97	a	132	ra	114, a
131	bar	133	ab	97, b
134	barb	134	barb	131, b
129	ar	135	barba	134, a
101	e	136	are	129, e
110	n	137	en	101, n

## Lempel-Ziv-Welch discussion

- Encoding and decoding take  $O(|S|)$  time (assuming constant alphabet)
- Encoding and decoding need to go through the string only *once*  
 $\Rightarrow$  can do compression while streaming the text
- So far, we encoded with numbers. How to convert to bitstring?  
Use length-12 **fixed-length encoding**, e.g.  $129 = 000010000001$ .  
(Stop adding to  $D$  once we are at code-number  $2^{12}-1 = 4095$ .)
- Compresses quite well ( $\approx 45\%$  on English text).

### Brief history:

**LZ77** Original version (“sliding window”)

Derivatives: LZSS, LZFG, LZRW, LZIP, DEFLATE, ...  
DEFLATE used in (pk)zip, gzip, PNG

**LZ78** Second (slightly improved) version

Derivatives: LZMW, LZAP, LZJ, ...

LZW used in compress, GIF (patent issues!)

# Outline

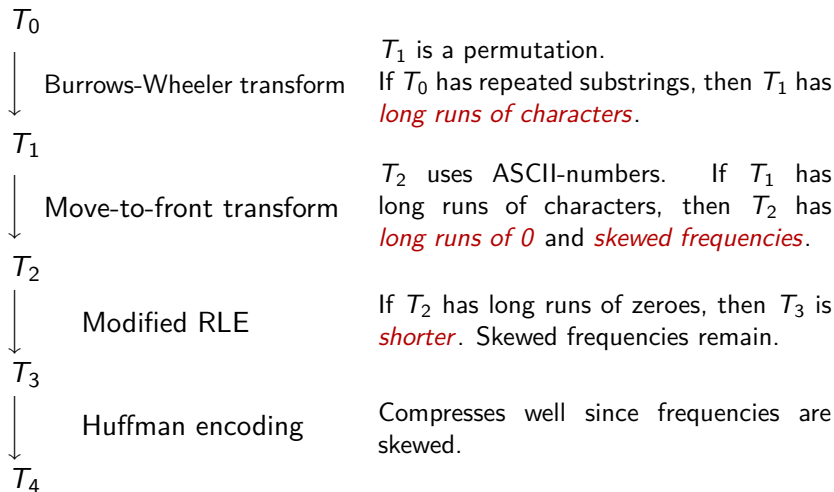
## 10 Data Compression

- Background
- Single-Character Encodings
- Huffman's Encoding Trie
- Run-Length Encoding
- Lempel-Ziv-Welch
- **Combining Compression Schemes: bzip2**
- Burrows-Wheeler Transform

## bzip2 overview

**Idea:** Combine multiple compression schemes!

**Example:** bzip2. Key ingredient is to use *text transforms*: Change input into a different text that compresses better.



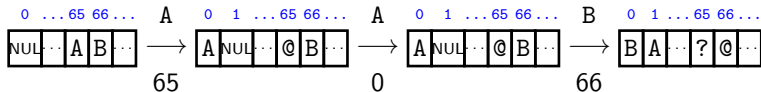




## bzip2 — the easy steps

- Move-to-front transform:

- ▶ Dictionary  $D : \{0, \dots, 127\} \rightarrow \text{ASCII}$  is unordered array with MTF.
- ▶ Character  $c$  gets mapped to index  $i$  with  $D[i] = c$



- ▶ A character in  $S$  repeats  $k$  times  $\Leftrightarrow C$  has run of  $k-1$  zeroes
- ▶ We would expect lots of small numbers in the output.

- Modified run-length encoding:

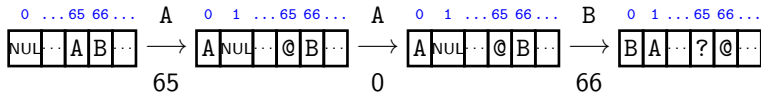
- ▶ Input is a list of 'characters' in  $\{0, \dots, 127\}$
- ▶ Encode *only* the runs of 0s (other numbers unchanged)
- ▶ Encode run-length  $k$  with bijective base-2 encoding (shorter), using two new characters  $A, B$ .

'65' '0' '0' '0' '67' '0' '0' '72' becomes '65' A B '67' B '72'

## bzip2 — the easy steps

- Move-to-front transform:

- ▶ Dictionary  $D : \{0, \dots, 127\} \rightarrow ASCII$  is unordered array with MTF.
- ▶ Character  $c$  gets mapped to index  $i$  with  $D[i] = c$



- ▶ A character in  $S$  repeats  $k$  times  $\Leftrightarrow C$  has run of  $k-1$  zeroes
- ▶ We would expect lots of small numbers in the output.

- Modified run-length encoding:

- ▶ Input is a list of 'characters' in  $\{0, \dots, 127\}$
- ▶ Encode *only* the runs of 0s (other numbers unchanged)
- ▶ Encode run-length  $k$  with bijective base-2 encoding (shorter), using two new characters  $A, B$ .

'65' '0' '0' '0' '67' '0' '0' '72' becomes '65' A B '67' B '72'

- Huffman encoding: exactly as seen before.

# Outline

## 10 Data Compression

- Background
- Single-Character Encodings
- Huffman's Encoding Trie
- Run-Length Encoding
- Lempel-Ziv-Welch
- Combining Compression Schemes: bzip2
- Burrows-Wheeler Transform

# Burrows-Wheeler Transform

**Given:** Source text  $S$  as an *array* with end-sentinel.

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16  
S: 

a	l	f	␣	e	a	t	s	␣	a	l	f	a	l	f	a	\$
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	----

**Step 1:** Write down cyclic shifts.

- **$i$ th cyclic shift:** move  $i$  characters from front to back.
- We treat (exceptionally) end-sentinel \$ like any other character

start-index

0	a	l	f	␣	e	a	t	s	␣	a	l	f	a	l	f	a	\$
1	l	f	␣	e	a	t	s	␣	a	l	f	a	\$	a			
2	f	␣	e	a	t	s	␣	a	l	f	a	\$	a	l			
3	␣	e	a	t	s	␣	a	l	f	a	\$	a	l	f			
4	e	a	t	s	␣	a	l	f	a	\$	a	l	f	␣			
5	a	t	s	␣	a	l	f	a	\$	a	l	f	␣	e			
6	t	s	␣	a	l	f	a	\$	a	l	f	␣	e	a			
7	s	␣	a	l	f	a	\$	a	l	f	␣	e	a	t			
8	␣	a	l	f	a	\$	a	l	f	␣	e	a	t	s			
9	a	l	f	a	\$	a	l	f	␣	e	a	t	s	␣			
10	l	f	a	\$	a	l	f	␣	e	a	t	s	␣	a			
11	f	a	\$	a	l	f	␣	e	a	t	s	␣	a	l			
12	a	l	f	a	\$	a	l	f	␣	e	a	t	s	␣	a		
13	l	f	a	\$	a	l	f	␣	e	a	t	s	␣	a	l		
14	f	a	\$	a	l	f	␣	e	a	t	s	␣	a	l	f		
15	a	\$	a	l	f	␣	e	a	t	s	␣	a	l	f	a		
16	\$	a	l	f	␣	e	a	t	s	␣	a	l	f	a			

# Burrows-Wheeler Transform

**Given:** Source text  $S$  as an *array* with end-sentinel.

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16  
S: 

a	l	f	␣	e	a	t	s	␣	a	l	f	a	l	f	a	\$
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	----

**Step 1:** Write down cyclic shifts.

- **$i$ th cyclic shift:** move  $i$  characters from front to back.
- We treat (exceptionally) end-sentinel \$ like any other character

(We do not need to write cyclic shifts explicitly; they can be read via start-index from  $S$ .)

start-index	
0	alf␣eats␣alfalfa\$
1	lf␣eats␣alfalfa\$a
2	f␣eats␣alfalfa\$a␣
3	␣eats␣alfalfa\$a␣lf
4	eats␣alfalfa\$a␣lf␣
5	ats␣alfalfa\$a␣lf␣e
6	ts␣alfalfa\$a␣lf␣ea
7	s␣alfalfa\$a␣lf␣eat
8	␣alfalfa\$a␣lf␣eats
9	alfalfa\$a␣lf␣eats␣
10	lfa\$alf␣eats␣al
11	fa\$alf␣eats␣al
12	alfa\$alf␣eats␣alf
13	lfa\$alf␣eats␣alfa
14	fa\$alf␣eats␣alfal
15	a\$alf␣eats␣alfalf
16	\$alf␣eats␣alfalfa

# Burrows-Wheeler Transform

**Given:** Source text  $S$  as an *array* with end-sentinel.

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16  
S: 

a	l	f	␣	e	a	t	s	␣	a	l	f	a	l	f	a	\$
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	----

**Step 1:** Write down cyclic shifts.

- **$i$ th cyclic shift:** move  $i$  characters from front to back.
- We treat (exceptionally) end-sentinel  $\$$  like any other character

(We do not need to write cyclic shifts explicitly; they can be read via start-index from  $S$ .)

**Observe:** Every column contains a permutation of  $S$ .

start-index	
0	a l f ␣ e a t s ␣ a l f a l f a \$
1	l f ␣ e a t s ␣ a l f a l f a \$ a
2	f ␣ e a t s ␣ a l f a l f a \$ a l
3	␣ e a t s ␣ a l f a l f a \$ a l f
4	e a t s ␣ a l f a l f a \$ a l f ␣
5	a t s ␣ a l f a l f a \$ a l f ␣ e
6	t s ␣ a l f a l f a \$ a l f ␣ e a
7	s ␣ a l f a l f a \$ a l f ␣ e a t
8	␣ a l f a l f a \$ a l f ␣ e a t s
9	a l f a l f a \$ a l f ␣ e a t s ␣
10	l f a l f a \$ a l f ␣ e a t s ␣ a
11	f a l f a \$ a l f ␣ e a t s ␣ a l
12	a l f a \$ a l f ␣ e a t s ␣ a l f
13	l f a \$ a l f ␣ e a t s ␣ a l f a
14	f a \$ a l f ␣ e a t s ␣ a l f a l
15	a \$ a l f ␣ e a t s ␣ a l f a l f
16	\$ a l f ␣ e a t s ␣ a l f a l f a

# Burrows-Wheeler Transform

**Given:** Source text  $S$  as an *array* with end-sentinel.

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16  
S: [a|l|f|\_|e|a|t|s|\_|a|l|f|a|l|f|a|\$]

**Step 1:** Write down cyclic shifts.

**Step 2:** Sort lexicographically.

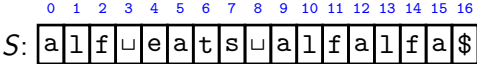
- Use MSD-radix-sort.
- $\Theta(n)$  strings of length  $\Theta(n)$   
 $\Rightarrow \Theta(n^2)$  worst-case time
- But usually much faster.

start-index

16	\$alf_eats_alfalfa
8	_alfalfa\$alf_eats
3	_eats_alfalfa\$alf
15	a\$alf_eats_alfalf
0	alf_eats_alfalfa\$
12	alf\$aalf_eats_alf
9	alfalfa\$alf_eats_
5	ats_alfalfa\$alf_e
4	eats_alfalfa\$alf_
2	f_eats_alfalfa\$alf
14	fa\$alf_eats_alfalf
11	falfa\$alf_eats_alf
1	lf_eats_alfalfa\$a
13	lfa\$alf_eats_alf
10	lfa\$alf_eats_alf
7	s_alfalfa\$alf_eat
6	ts_alfalfa\$alf_ea

# Burrows-Wheeler Transform

**Given:** Source text  $S$  as an *array* with end-sentinel.



**Step 1:** Write down cyclic shifts.

**Step 2:** Sort lexicographically.

- Use MSD-radix-sort.
- $\Theta(n)$  strings of length  $\Theta(n)$   
 $\Rightarrow \Theta(n^2)$  worst-case time
- But usually much faster.

**Observe:** Every column continues to contain a permutation of  $S$ .

start-index	
16	\$alf␣eats␣alfalfa
8	␣alfalfa\$alf␣eats
3	␣eats␣alfalfa\$alf
15	a\$alf␣eats␣alfalf
0	alf␣eats␣alfalfa\$
12	alfalfa\$alf␣eats␣alf
9	alfalfa\$alf␣eats␣
5	ats␣alfalfa\$alf␣e
4	eats␣alfalfa\$alf␣
2	f␣eats␣alfalfa\$alf
14	fa\$alf␣eats␣alfalf
11	falfalfa\$alf␣eats␣alf
1	lf␣eats␣alfalfalfa\$
13	lfa\$alf␣eats␣alfalfa
10	lfalfalfa\$alf␣eats␣alf
7	s␣alfalfalfa\$alf␣eat
6	ts␣alfalfalfa\$alf␣ea



# Burrows-Wheeler Transform

**Given:** Source text  $S$  as an *array* with end-sentinel.

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16  
S: [a|l|f|\_|e|a|t|s|\_|a|l|f|a|l|f|a|\$]

**Step 1:** Write down cyclic shifts.

**Step 2:** Sort lexicographically.

**Step 3:** Extract rightmost column.

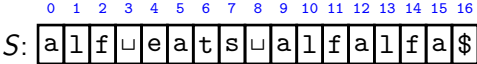
The **Burrows-Wheeler transform** consists of the last characters of the cyclic shifts of  $S$  after sorting them lexicographically.

start-index

16	\$alf_eats_alfalf	a
8	_alfalfa\$alf_eat	s
3	_eats_alfalfa\$alf	f
15	a\$alf_eats_alfalf	f
0	alf_eats_alfalfa\$	
12	alfalfa\$alf_eats_alf	f
9	alfalfa\$alf_eats_	
5	ats_alfalfa\$alf_e	e
4	eats_alfalfa\$alf_	
2	f_eats_alfalfa\$alf	
14	fa\$alf_eats_alfalf	
11	falfalfa\$alf_eats_alf	
1	lf_eats_alfalfa\$alf	
13	lfa\$alf_eats_alfalf	
10	lfalfalfa\$alf_eats_alf	
7	s_alfalfa\$alf_eat	t
6	ts_alfalfa\$alf_ea	

# Burrows-Wheeler Transform

**Given:** Source text  $S$  as an *array* with end-sentinel.



- Step 1:** Write down cyclic shifts.
- Step 2:** Sort lexicographically.
- Step 3:** Extract rightmost column.

The **Burrows-Wheeler transform** consists of the last characters of the cyclic shifts of  $S$  after sorting them lexicographically.

- Observe:**  $C$  is a permutation of  $S$ .
- Observe:** Substring **alf** occurs three times in  $S$  and causes runs **lll** and **aaa** in  $C$  (why?)

start-index	
16	\$alf␣eats␣alfalf <b>a</b>
8	␣alfalfa\$alf␣eat <b>s</b>
3	␣eats␣alfalfa\$alf
15	a\$alf␣eats␣alfalf <b>f</b>
0	alf␣eats␣alfalf <b>\$</b>
12	alfalfa\$alf␣eats␣alf
9	alfalfa\$alf␣eats␣
5	ats␣alfalfalfa\$alf␣ <b>e</b>
4	eats␣alfalfalfa\$alf␣
2	f␣eats␣alfalfalfa\$alf
14	fa\$alf␣eats␣alfalf
11	falfalfa\$alf␣eats␣alf
1	lf␣eats␣alfalfalf <b>\$a</b>
13	lfa\$alf␣eats␣alfalf <b>a</b>
10	lfalfalfa\$alf␣eats␣alf <b>a</b>
7	s␣alfalfalfa\$alf␣eat
6	ts␣alfalfalfa\$alf␣ <b>e</b> a

# Fast Burrows-Wheeler Encoding

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

$i$	$i$ th cyclic shift	$A^s$	corresponding suffix
0	16 \$alf_eats_alfalfa	0	16 \$alf_eats_alfalfa
1	8 _alfalfa\$alf_eats	1	8 _alfalfa\$alf_eats
2	3 _eats_alfalfa\$alf	2	3 _eats_alfalfa\$alf
3	15 a\$alf_eats_alfalf	3	15 a\$alf_eats_alfalf
4	0 alf_eats_alfalfa\$	4	0 alf_eats_alfalfa\$
5	12 alfa\$alf_eats_alf	5	12 alfa\$alf_eats_alf
6	9 alfalfa\$alf_eats_	6	9 alfalfa\$alf_eats_
7	5 ats_alfalfa\$alf_e	7	5 ats_alfalfa\$alf_e
8	4 eats_alfalfa\$alf_	8	4 eats_alfalfa\$alf_
9	2 f_eats_alfalfa\$alf	9	2 f_eats_alfalfa\$alf
10	14 fa\$alf_eats_alfalf	10	14 fa\$alf_eats_alfalf
11	11 falfa\$alf_eats_alf	11	11 falfa\$alf_eats_alf
12	1 lf_eats_alfalfa\$a	12	1 lf_eats_alfalfa\$a
13	13 lfa\$alf_eats_alfalf	13	13 lfa\$alf_eats_alfalf
14	10 lfalfa\$alf_eats_alf	14	10 lfalfa\$alf_eats_alf
15	7 s_alfalfa\$alf_eat	15	7 s_alfalfa\$alf_eat
16	6 ts_alfalfa\$alf_ea	16	6 ts_alfalfa\$alf_ea

- **Observe:** Same sorting permutation for cyclic shifts and suffixes.

# Fast Burrows-Wheeler Encoding

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

	$i$	$i$ th cyclic shift		$A^s$	corresponding suffix
0	16	\$alf_eats_alfalfa	0	16	\$alf_eats_alfalfa
1	8	_alfalfa\$alf_eats	1	8	_alfalfa\$alf_eats
2	3	_eats_alfalfa\$alf	2	3	_eats_alfalfa\$alf
3	15	a\$alf_eats_alfalf	3	15	a\$alf_eats_alfalf
4	0	alf_eats_alfalfa\$	4	0	alf_eats_alfalfa\$
5	12	alfa\$alf_eats_alf	5	12	alfa\$alf_eats_alf
6	9	alfalfa\$alf_eats_	6	9	alfalfa\$alf_eats_
7	5	ats_alfalfa\$alf_e	7	5	ats_alfalfa\$alf_e
8	4	eats_alfalfa\$alf_	8	4	eats_alfalfa\$alf_
9	2	f_eats_alfalfa\$alf	9	2	f_eats_alfalfa\$alf
10	14	fa\$alf_eats_alfalf	10	14	fa\$alf_eats_alfalf
11	11	falfa\$alf_eats_alf	11	11	falfa\$alf_eats_alf
12	1	lf_eats_alfalfa\$a	12	1	lf_eats_alfalfa\$a
13	13	lfa\$alf_eats_alfalf	13	13	lfa\$alf_eats_alfalf
14	10	lfalfa\$alf_eats_alf	14	10	lfalfa\$alf_eats_alf
15	7	s_alfalfa\$alf_eat	15	7	s_alfalfa\$alf_eat
16	6	ts_alfalfa\$alf_ea	16	6	ts_alfalfa\$alf_ea

- **Observe:** Same sorting permutation for cyclic shifts and suffixes.
- That's the suffix array  $A^s$ ! Can compute this in  $O(n \log n)$  time.
- Read BWT encoding from it:  $C[i] = \begin{cases} S[A^s[i]-1] & \text{if } A^s[i] > 0 \\ \$ & \text{otherwise} \end{cases}$

# BWT Decoding

**Given:** String  $C = \text{ard\$rcaaaaabb}$  (obtained from a BWT encoding).

We can reconstruct the *first* and *last column* of the matrix of cyclic shifts.

- ① Last column:  $C$  .....a
- ② First column:  $C$  sorted .....r
  - ▶ It was a permutation of  $S$ . .....d
  - ▶ It was in sorted order. ....\$
  - ▶  $C$  was also a permutation of  $S$ . .....r.....c
- .....a
- .....a
- .....a
- .....a
- .....b
- .....b

# BWT Decoding

**Given:** String  $C = \text{ard}\$ \text{rcaaaabb}$  (obtained from a BWT encoding).

We can reconstruct the *first* and *last column* of the matrix of cyclic shifts.

- 1 Last column:  $C$
- 2 First column:  $C$  sorted
  - ▶ It was a permutation of  $S$ .
  - ▶ It was in sorted order.
  - ▶  $C$  was also a permutation of  $S$ .
- 3 What was the first character of  $S$ ?

```
$.....a
a.....r
a.....d
a.....$
a.....r
a.....c
b.....a
b.....a
c.....a
d.....a
r.....b
r.....b
```

# BWT Decoding

**Given:** String  $C = \text{ard\$rcaaaabb}$  (obtained from a BWT encoding).

We can reconstruct the *first* and *last column* of the matrix of cyclic shifts.

- |   |                 |
|---|-----------------|
| ① Last column: $C$  | $\$. . . . . a$ |
| ② First column: $C$ sorted  | $a. . . . . r$  |
| ▶ It was a permutation of $S$ .   | $a. . . . . d$  |
| ▶ It was in sorted order.   | $a. . . . . \$$ |
| ▶ $C$ was also a permutation of $S$ .   | $a. . . . . r$  |
|   | $a. . . . . c$  |
| ③ What was the first character of $S$ ?   | $b. . . . . a$  |
|   | $b. . . . . a$  |
| ④ Presume you know which a of $C$ this is:<br>What is the next character of $S$ ? | $c. . . . . a$  |
|   | $d. . . . . a$  |
|   | $r. . . . . b$  |
|   | $r. . . . . b$  |

**Question:** Which character on the left corresponds to which character on the right?

# BWT Decoding - Disambiguation

**Idea:** Attach row-index at each character of  $C$ .

- Two cyclic shifts end in  $b$ , call them  $w_{10}(b, 10)$  and  $w_{11}(b, 11)$
- *Without* knowing  $w_{10}, w_{11}$ , we know  $w_{10}(b, 10) <_{\text{lex}} w_{11}(b, 11)$  (why?)

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



# BWT Decoding - Disambiguation

**Idea:** Attach row-index at each character of  $C$ .

- Two cyclic shifts end in  $b$ , call them  $w_{10}(b, 10)$  and  $w_{11}(b, 11)$
- *Without* knowing  $w_{10}, w_{11}$ , we know  $w_{10}(b, 10) <_{\text{lex}} w_{11}(b, 11)$  (why?)
- Therefore  $w_{10} <_{\text{lex}} w_{11}$
- Two cyclic shifts start with  $b$ :  
 $(b, 10)w_{10}$  and  $(b, 11)w_{11}$ .

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

# BWT Decoding - Disambiguation

**Idea:** Attach row-index at each character of  $C$ .

- Two cyclic shifts end in  $b$ , call them  $w_{10}(b, 10)$  and  $w_{11}(b, 11)$
- *Without* knowing  $w_{10}, w_{11}$ , we know  $w_{10}(b, 10) <_{\text{lex}} w_{11}(b, 11)$  (why?)
- Therefore  $w_{10} <_{\text{lex}} w_{11}$
- Two cyclic shifts start with  $b$ :  
 $(b, 10)w_{10}$  and  $(b, 11)w_{11}$ .
- We know  $(b, 10)w_{10} <_{\text{lex}} (b, 11)w_{11}$ .

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

# BWT Decoding - Disambiguation

**Idea:** Attach row-index at each character of  $C$ .

- Two cyclic shifts end in  $b$ , call them  $w_{10}(b, 10)$  and  $w_{11}(b, 11)$
- *Without* knowing  $w_{10}, w_{11}$ , we know  $w_{10}(b, 10) <_{\text{lex}} w_{11}(b, 11)$  (why?)
- Therefore  $w_{10} <_{\text{lex}} w_{11}$
- Two cyclic shifts start with  $b$ :  $(b, 10)w_{10}$  and  $(b, 11)w_{11}$ .
- We know  $(b, 10)w_{10} <_{\text{lex}} (b, 11)w_{11}$ .
- Therefore  $(b, 10)$  comes *before*  $(b, 11)$  in first column.

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

**Result:** Equal characters are in the same order in the first and last column.

# BWT Decoding Continued

**Given:** String  $C = \text{ard\$rcaaaabb}$  (obtained from a BWT encoding).

- 1 Last column:  $C$ 
  - ▶ Disambiguate by row-index.
- 2 First column  $A$ :  $C$  sorted *stably*

( $\$,3$ ) . . . . . (a,0)

(a,0) . . . . . (r,1)

(a,6) . . . . . (d,2)

(a,7) . . . . . ( $\$,3$ )

(a,8) . . . . . (r,4)

(a,9) . . . . . (c,5)

(b,10) . . . . . (a,6)

(b,11) . . . . . (a,7)

(c,5) . . . . . (a,8)

(d,2) . . . . . (a,9)

(r,1) . . . . . (b,10)

(r,4) . . . . . (b,11)

## BWT Decoding Continued

**Given:** String  $C = \text{ard}\$rcaaaabb$  (obtained from a BWT encoding).

- 1 Last column:  $C$ 
  - ▶ Disambiguate by row-index.
- 2 First column  $A$ :  $C$  sorted *stably*
- 3 Find index  $j$  of end-sentinel  $\$$  in  $C$
- 4 Starting from  $A[j]$ , recover  $S$  by looking up next character and next index.

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

# BWT Decoding Continued

**Given:** String  $C = \text{ard}\$rcaaaabb$  (obtained from a BWT encoding).

- 1 Last column:  $C$ 
  - ▶ Disambiguate by row-index.
- 2 First column  $A$ :  $C$  sorted *stably*
- 3 Find index  $j$  of end-sentinel  $\$$  in  $C$
- 4 Starting from  $A[j]$ , recover  $S$  by looking up next character and next index.

$S = a$

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

# BWT Decoding Continued

**Given:** String  $C = \text{ard}\$rcaaaabb$  (obtained from a BWT encoding).

- 1 Last column:  $C$ 
  - ▶ Disambiguate by row-index.
- 2 First column  $A$ :  $C$  sorted *stably*
- 3 Find index  $j$  of end-sentinel  $\$$  in  $C$
- 4 Starting from  $A[j]$ , recover  $S$  by looking up next character and next index.

$S = ab$

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

# BWT Decoding Continued

**Given:** String  $C = \text{ard}\$rcaaaabb$  (obtained from a BWT encoding).

- 1 Last column:  $C$ 
  - ▶ Disambiguate by row-index.
- 2 First column  $A$ :  $C$  sorted *stably*
- 3 Find index  $j$  of end-sentinel  $\$$  in  $C$
- 4 Starting from  $A[j]$ , recover  $S$  by looking up next character and next index.

$S = \text{abr}$

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



# BWT Decoding Continued

**Given:** String  $C = \text{ard}\$rcaaaabb$  (obtained from a BWT encoding).

- 1 Last column:  $C$ 
  - ▶ Disambiguate by row-index.
- 2 First column  $A$ :  $C$  sorted *stably*
- 3 Find index  $j$  of end-sentinel  $\$$  in  $C$
- 4 Starting from  $A[j]$ , recover  $S$  by looking up next character and next index.

$S = \text{abracadabra}\$$

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

# BWT Decoding

*BWT::decoding*(C, S)

C : string of characters, one of which (not necessarily last one) is \$

S: output-stream

1. Initialize array A // leftmost column
2. **for** all indices  $i$  of C
3.      $A[i] \leftarrow (C[i], i)$  // store character and index
4. Stably sort A by first aspect
5. **for** all indices  $j$  of C // where is the \$-char?
6.     if  $C[j] = \$$  **break**
7. **do** // extract source text
8.     S.*append*(character stored in  $A[j]$ )
9.      $j \leftarrow$  index stored in  $A[j]$
10. **while** appended character is not \$

What sorting-algorithm would you use?

## BWT and bzip2 Discussion

**BWT encoding cost:**  $O(n \log n)$

- Read encoding from the suffix array.

**BWT decoding cost:**  $O(n + |\Sigma_S|)$  (faster than encoding)

Encoding and decoding both use  $O(n)$  space.

They need *all* of the text (no streaming possible). BWT (hence bzip2) is a **block compression method** that compresses one block at a time.

**bzip2 encoding cost:**  $O(n(\log n + |\Sigma|))$  with a big constant.

**bzip2 decoding cost:**  $O(n|\Sigma|)$  with a big constant.

bzip2 tends to be slower than other methods, but gives better compression.

## Compression summary

<b>Huffman</b>	<b>Run-length encoding</b>	<b>Lempel-Ziv-Welch</b>	<b>bzip2</b> (uses Burrows-Wheeler)
variable-length	variable-length	fixed-length	multi-step
single-character	multi-character	multi-character	multi-step
2-pass, must send dictionary	1-pass	1-pass	not streamable
optimal 01-prefix-code	good on long runs (e.g., pictures)	good on English text	better on English text
requires uneven frequencies	requires runs	requires repeated substrings	requires repeated substrings
rarely used directly	rarely used directly	frequently used	used but slow
part of pkzip, JPEG, MP3	fax machines, TIFF	GIF, some variants of PDF, compress	bzip2 and variants