

CS 240 – Data Structures and Data Management

Module 10: Compression

A. Hunt and O. Veksler

Based on lecture notes by many previous cs240 instructors

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

Winter 2023

Outline

- Compression

- Encoding Basics
- Huffman Codes
- Run-Length Encoding
- Lempel-Ziv-Welch
- bzip2
- Burrows-Wheeler Transform

Outline

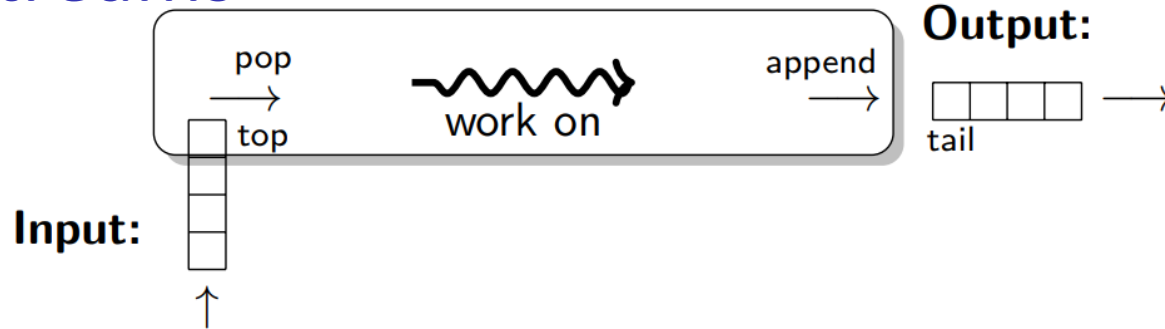
- **Compression**

- **Encoding Basics**
- Huffman Codes
- Run-Length Encoding
- Lempel-Ziv-Welch
- bzip2
- Burrows-Wheeler Transform

Data Storage and Transmission

- **The problem:** How to store and transmit data?
- Source text
 - the original data, string S of characters from the **source alphabet** Σ_S
- Coded text
 - the encoded data, string C of characters from the **coded alphabet** Σ_C
- Encoding
 - an algorithm mapping source text to coded text
- Decoding
 - an algorithm mapping coded text back to the original source text
- Source “text” can be any sort of data (not always text)
- Usually the coded alphabet is binary $\Sigma_C = \{0,1\}$

Detour: Streams



- Usually S and C are stored as *streams*
 - input stream
 - read one character at a time
 - *pop()*, *top()*
 - also supports *isEmpty()*
 - sometimes need *reset()* to start processing from the start
 - output stream
 - write one character at a time
 - *append()*
 - also supports *isEmpty()*
 - convenient for handling huge texts
 - can start processing text while it is still being loaded
 - avoids needing to hold the entire text in memory at once

Judging Encoding Schemes

- Measure time/space efficiency of encoding/decoding algorithms, as for any usual algorithm
- What other goals make sense?
 - reliability
 - error-correcting codes
 - security
 - encryption
 - **size** (our main objective in this module)
- Encoding schemes that try to minimize the size of the coded text perform ***data compression***
- We will measure the ***compression ratio*** $\frac{|C| \cdot \lceil \log |\Sigma_C| \rceil}{|S| \cdot \lceil \log |\Sigma_S| \rceil}$

Types of Data Compression

- **Logical vs. Physical**

- **Logical Compression**

- uses the meaning of the data
 - only applies to a certain domain (e.g. sound recordings)

- **Physical Compression**

- only know physical bits in data, not their meaning

- **Lossy vs. Lossless**

- **Lossy Compression**

- achieves better compression ratios
 - decoding is approximate
 - exact source text S is not recoverable

- **Lossless Compression**

- always decodes S exactly

- Lossy, logical compression is useful

- media files: JPEG, MPEG

- But we will concentrate on *physical, lossless* compression

- can be safely used for any application

Character Encodings

- **Definition:** **character encoding** E maps each *character* in the source alphabet to a *string* in coded alphabet

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

- for $c \in \Sigma_S$, $E(c)$ is called the **codeword** (or **code**) of c
- **Character encoding** sometimes is called **character-by-character** encoding
 - encode one character at a time
- Two possibilities
 - **Fixed-length code**: all codewords have the same length
 - **Variable-length code**: codewords may have different lengths

Fixed Length Codes

- Example: ASCII (American Standard Code for Information Interchange), 1963

char in Σ_S	null	start of heading	start of text	...	0	1	...	A	B	...	~	delete
code	0	1	2	...	48	49	...	65	66	...	126	127
code in binary	0000000	0000001	0000010		0110000	0110001		01000001	01000010		1111110	1111111

- 7 bits to encode 128 possible characters
 - control codes, spaces, letters, digits, punctuation
 - A P P L E \rightarrow (65, 80, 80, 76, 69) \rightarrow 01000001 1010000 1010000 1001100 1000101
- Standard in all computers and often our source alphabet
- Not well-suited for non-English text
 - many extensions to 8 bits or multi-byte encodings
 - nowadays all subsumed by Unicode
- Other (earlier) fixed-length codes: Baudot code, Murray code
- To decode a fixed-length code (say codewords have k bits), we look up each k -bit pattern in a table

Variable-Length Codes

- **Overall goal:** Find an encoding that is short
- **Observation:** Some alphabet letters occur more often than others
 - idea: use shorter codes for more frequent characters
 - example: 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

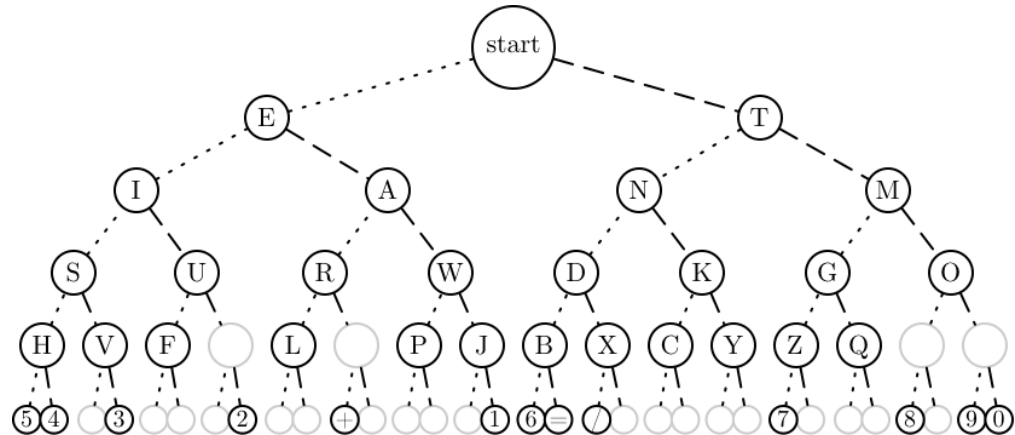
- Example 1: Morse code

International Morse Code

1. A dash is equal to three dots.
2. The space between parts of the same letter is equal to one dot.
3. The space between two letters is equal to three dots.
4. The space between two words is equal to seven dots.

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

1	• — • — • —
2	• • • — • —
3	• • • • — •
4	• • • • • —
5	• • • • • •
6	• — • • • •
7	• — • • • •
8	• — • • • •
9	• — • • • •
0	• — • • • •



- Example 2: UTF-8 encoding of Unicode
 - there are more than 107,000 Unicode characters
 - uses 1-4 bytes to encode any Unicode character

Encoding

- Assume we have some character encoding $E: \Sigma_S \rightarrow \Sigma_C^*$
- E is a dictionary with keys in Σ_S
- Typically E would be stored as array indexed by Σ_S

charByChar::Encoding(E, S, C)

E : encoding dictionary, S : input stream with characters in Σ_S

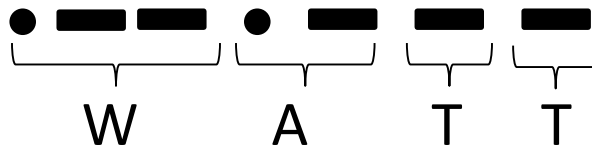
C : output stream

while S is non-empty

$x \leftarrow E.\text{search}(S.\text{pop}())$

$C.\text{append}(x)$

- Example: encode text “WATT” with Morse code



E

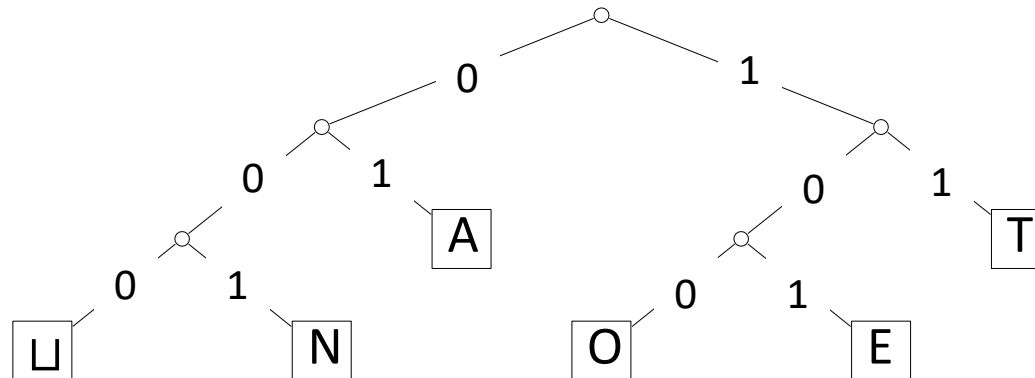
A	• —	U	• • —
B	• • — •	V	• • — •
C	• — • —	W	• — • —
D	• — • •	X	• • • —
E	•	Y	• • — •
F	• • — •	Z	• — • —
G	• — • —		
H	• • • •		
I	• •		
J	• — • —		
K	• • • •		
L	• • — •		
M	• — • —		
N	• — • •		
O	• — • —		
P	• — • •		
Q	• — • —		
R	• • — •		
S	• • • •		
T	• —		
		1	• — • — • —
		2	• • • — • —
		3	• • — • — •
		4	• • • — • —
		5	• • — • — •
		6	• • — • — •
		7	• • — • — •
		8	• • — • — •
		9	• • — • — •
		0	• — • — • —

Decoding

- The **decoding algorithm** must map Σ_C^* to Σ_S
- The code must be *uniquely decodable*
 - false for Morse code as described
 - $\bullet \bullet \blacksquare$ decodes to both U and EA
 - Morse code uses 'end of character' pause to avoid ambiguity
- From now on only consider **prefix-free** codes E
 - $E(c)$ is not a prefix of $E(c')$ for any $c, c' \in \Sigma_S$
- Store codes in a **trie** with characters of Σ_S at the leaves

A $\bullet \blacksquare$
 B $\blacksquare \bullet \bullet \bullet$
 C $\blacksquare \bullet \blacksquare \bullet$
 D $\blacksquare \bullet \bullet$
 E \bullet

U $\bullet \bullet \blacksquare$
 V $\bullet \bullet \bullet \blacksquare$
 W $\bullet \blacksquare \blacksquare$
 X $\blacksquare \bullet \bullet \blacksquare$
 Y $\blacksquare \bullet \blacksquare \blacksquare$



- Do not need symbol \$, codewords are prefix-free by definition

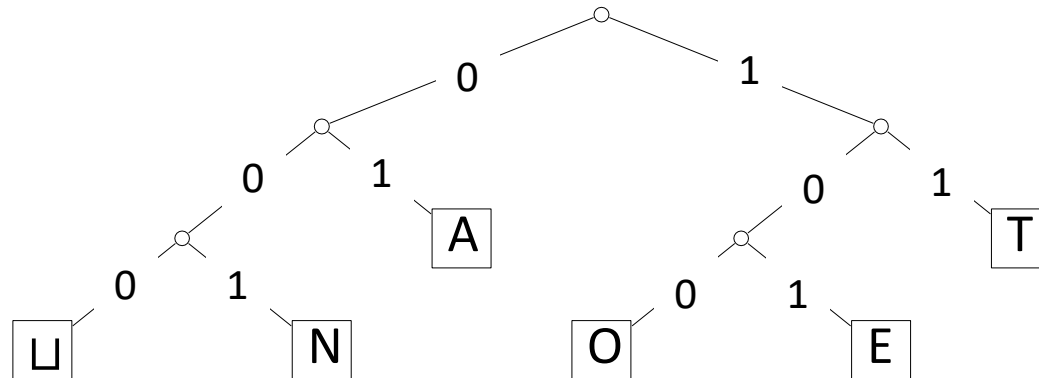
Example: Prefix-free Encoding/Decoding

- For encoding, use code stored in an array

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

- Encode **A**N**L****A**N**T** \rightarrow **01** **001** **000** **01****001****11**

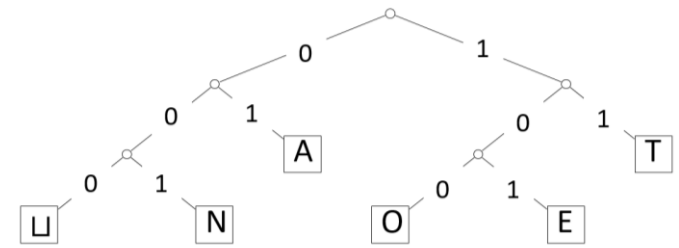
- For decoding, use code stored in a trie



- Decode **11****100****000****101****0111** \rightarrow **T****O****L****E****A****T**

Decoding of Prefix-Free Codes

- Any prefix-free code is uniquely decodable



$C = 11\ 100\ 000\ 101\ 01\ 11$

PrefixFree::decoding(T, C, S)

T : trie of a prefix-free code, C : input-stream with characters in Σ_C

S : output-stream

while C is non-empty // iterate over all codewords

$r \leftarrow T.\text{root}$

while r is not a leaf // read next codeword

if C is empty or has no child labelled $C.\text{top}()$

return “invalid encoding”

$r \leftarrow$ child of r that is labelled with $C.\text{pop}()$

$S.\text{append}(\text{character stored at } r)$

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

Encoding from the Trie

- Can encode directly from the trie T

PrefixFree::encoding(T, S, C)

T : prefix-free code trie, S : input-stream with characters in Σ_S

$E \leftarrow$ array of nodes in T indexed by Σ_S

for all leaves l in T

$E[\text{character at } l] \leftarrow l$

while S is non-empty

$w \leftarrow$ empty string; $v \leftarrow E[S.\text{pop}()]$

while v is not the root

$w.\text{prepend}$ (character from v to its parent)

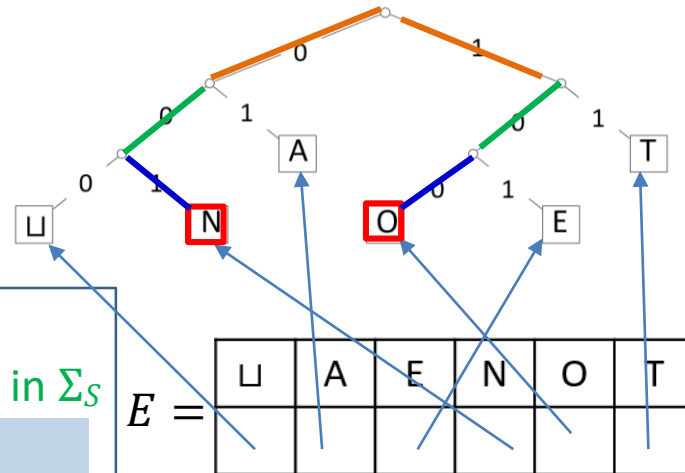
$v \leftarrow \text{parent}(v)$

// now w is the encoding of S

$C.\text{append}(w)$

- Run-time: $O(|T| + |C|)$

- have to visit all trie nodes, and insert leaves into E
- $O(|\Sigma_S| + |C|)$ if T has no nodes with one child
 - $\# \text{leaves} - 1 \geq \# \text{internal nodes}$



$S = ON$

$i = 0$ (letter O)

$w = \Lambda$

$w = 0$

$w = 00$

$w = 100$

$C = 100$

$i = 1$ (letter N)

$w = \Lambda$

$w = 1$

$w = 01$

$w = 001$

$C = 100\ 001$

Outline

- **Compression**

- Encoding Basics
- **Huffman Codes**
- Run-Length Encoding
- Lempel-Ziv-Welch
- bzip2
- Burrows-Wheeler Transform

Huffman's Algorithm: Building the Best Trie

- For a given S the best trie can be constructed with Huffman tree algorithm
 - trie giving the shortest coded text C if alphabet is binary
 - $\Sigma_C = \{0,1\}$
 - tailored to frequencies in that particular S

Example: Huffman Tree Construction

- Example text: *GREENENERGY*, $\Sigma_S = \{G, R, E, N, Y\}$
- Calculate character frequencies
 $G: 2, R: 2, E: 4, N: 2, Y: 1$
- Put each character into its own trie (single node, height 0)
 - each trie has a frequency
 - initially, frequency is equal to its character frequency

2
G

2
R

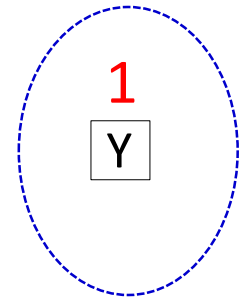
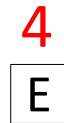
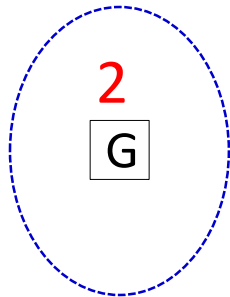
4
E

2
N

1
Y

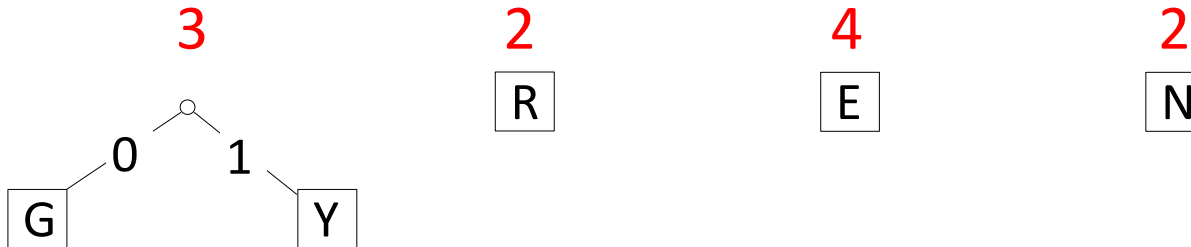
Example: Huffman Tree Construction

- Example text: *GREENENERGY*, $\Sigma_S = \{G, R, E, N, Y\}$
- Calculate character frequencies
 $G: 2, R: 2, E: 4, N: 2, Y: 1$
- Join two least frequent tries into a new trie
 - frequency of the new trie = sum of old trie frequencies



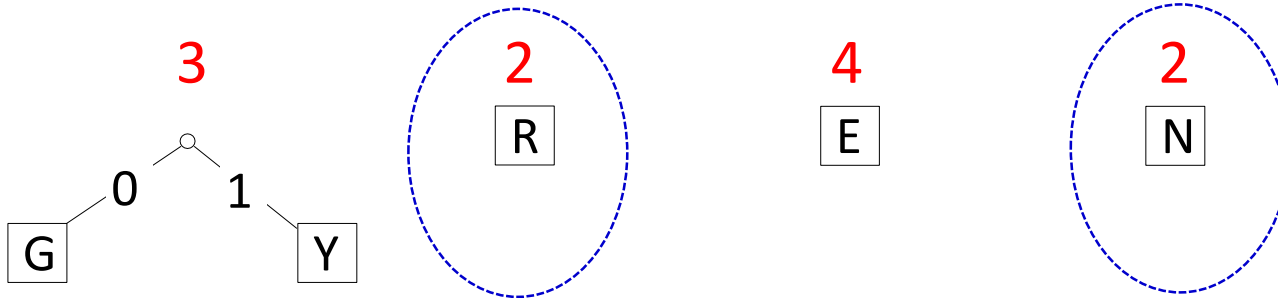
Example: Huffman Tree Construction

- Example text: *GREENENERGY*, $\Sigma_S = \{G, R, E, N, Y\}$
- Calculate character frequencies
 $G: 2, R: 2, E: 4, N: 2, Y: 1$
- Join two least frequent tries into a new trie
 - frequency of the new trie = sum of old trie frequencies



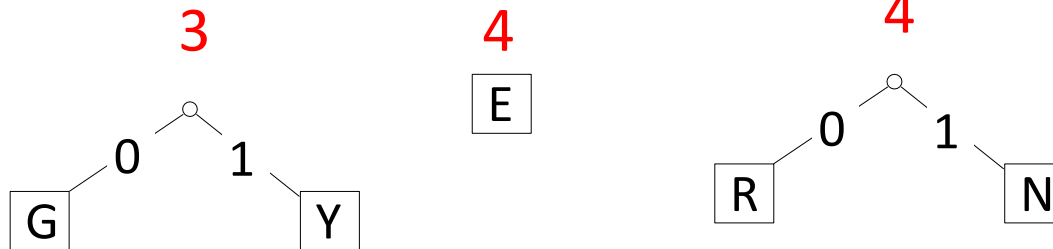
Example: Huffman Tree Construction

- Example text: *GREENENERGY*, $\Sigma_S = \{G, R, E, N, Y\}$
- Calculate character frequencies
 $G: 2, R: 2, E: 4, N: 2, Y: 1$
- Join two least frequent tries into a new trie
 - frequency of the new trie = sum of old trie frequencies



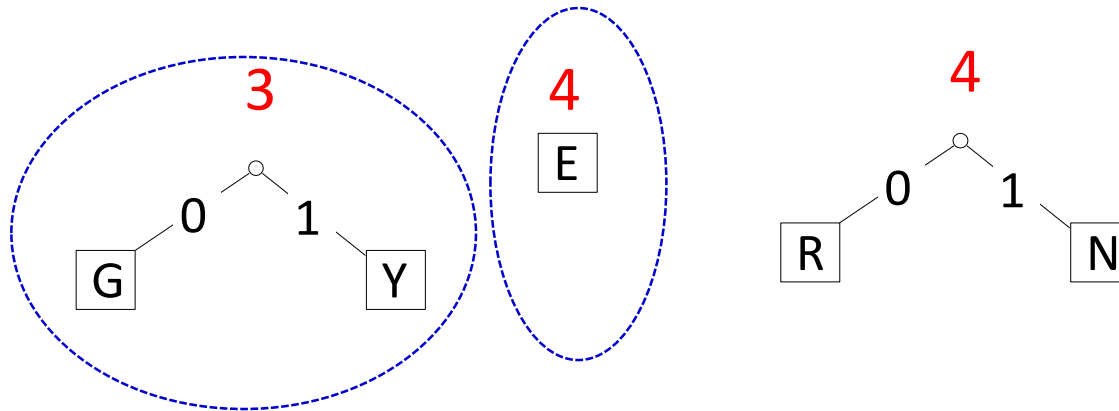
Example: Huffman Tree Construction

- Example text: *GREENENERGY*, $\Sigma_S = \{G, R, E, N, Y\}$
- Calculate character frequencies
 $G: 2, R: 2, E: 4, N: 2, Y: 1$
- Join two least frequent tries into a new trie
 - frequency of the new trie = sum of old trie frequencies



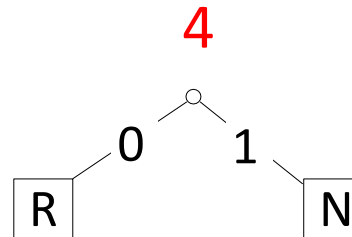
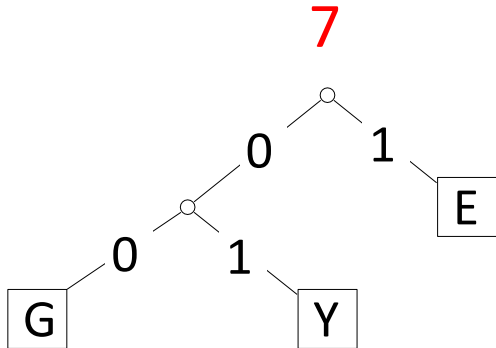
Example: Huffman Tree Construction

- Example text: *GREENENERGY*, $\Sigma_S = \{G, R, E, N, Y\}$
- Calculate character frequencies
 $G: 2, R: 2, E: 4, N: 2, Y: 1$
- Join two least frequent tries into a new trie
 - frequency of the new trie = sum of old trie frequencies



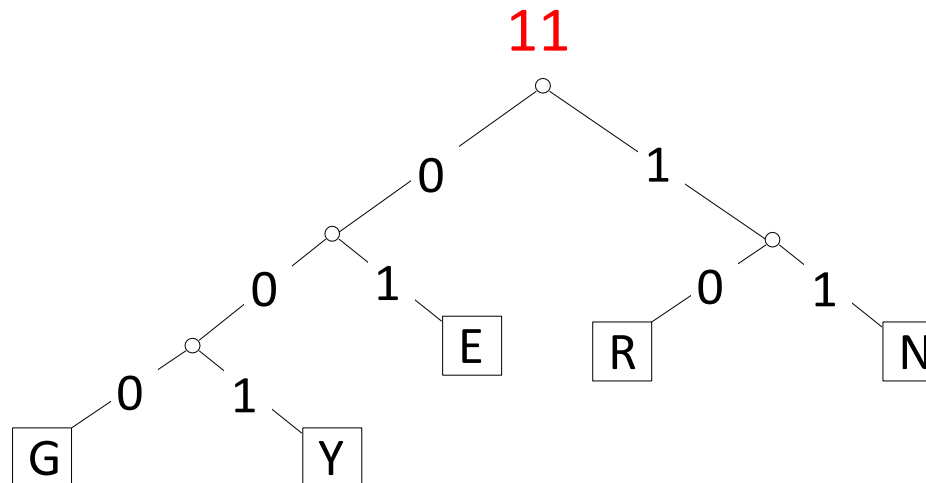
Example: Huffman Tree Construction

- Example text: *GREENENERGY*, $\Sigma_S = \{G, R, E, N, Y\}$
- Calculate character frequencies
 $G: 2, R: 2, E: 4, N: 2, Y: 1$
- Join two least frequent tries into a new trie
 - frequency of the new trie = sum of old trie frequencies



Example: Huffman Tree Construction

- Example text: *GREENENERGY*, $\Sigma_S = \{G, R, E, N, Y\}$
- Calculate character frequencies
 $G: 2, R: 2, E: 4, N: 2, Y: 1$
- Join two least frequent tries into a new trie
 - frequency of the new trie = sum of old trie frequencies

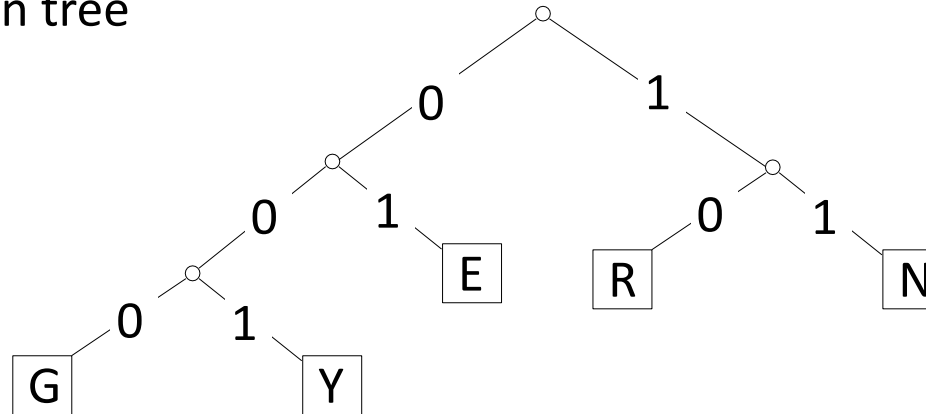


Example: Huffman Tree Construction

- Example text: *GREENENERGY*, $\Sigma_S = \{G, R, E, N, Y\}$
- Calculate character frequencies

$G: 2, R: 2, E: 4, N: 2, Y: 1$

- Final Huffman tree



- *GREENENERGY* → 000 10 01 01 11 01 11 01 10 000 001

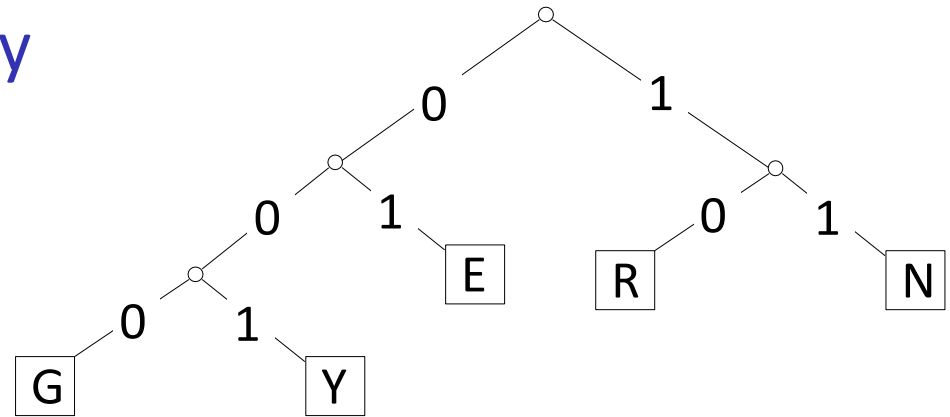
- Compression ratio

$$\frac{25}{11 \cdot \lceil \log 5 \rceil} \approx 76\%$$

- These frequencies are not skewed enough to lead to good compression

Huffman Algorithm Summary

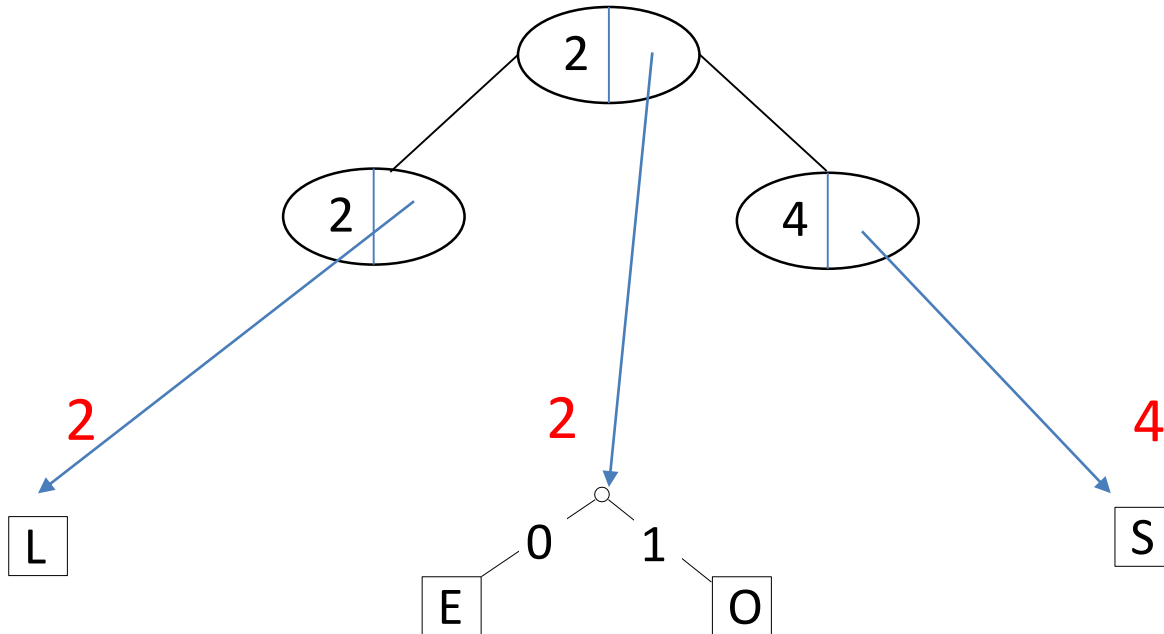
- Example text: *GREENENERGY*
- Character frequencies
 - $G:2, R:2, E:4, N:2, Y:1$



- For a given source S , to determine a trie that minimizes length of C
 - 1) determine frequency of each character $c \in \Sigma$ in S
 - 2) for each $c \in \Sigma$, create trie of height 0 holding only c
 - call it c -trie
 - 3) assign a weight to each trie
 - sum of frequencies of all letters in a trie
 - initially, these are just the character frequencies
 - 4) find the two tries with the minimum weight
 - 5) merge these tries with a new interior node
 - the new weight is the sum of merged tries weights
 - added one bit to the encoding of each character
 - 6) repeat Steps 4–5 until there is only 1 trie left
 - this is D , the final decoder
- Min-heap for efficient implementation: step 4 is two *delete-min* step 5 is *insert*

Heap Storing Tries during Huffman Tree Construction

- (key,value) = (trie weight, link to trie)
- step 4 is two *delete-mins*, step 5 is *insert*



Huffman's Algorithm Pseudocode

Huffman::encoding(S, C)

S : input-stream (length n) with characters in Σ_S , C : output-stream, initially empty

$f \leftarrow$ array indexed by Σ_S , initialized to 0

while S is non-empty **do increase** $f[S.\text{pop}()]$ by 1 // get frequencies $O(n)$

$Q \leftarrow$ min-oriented priority queue to store tries

for all $c \in \Sigma_S$ with $f[c] > 0$

$Q.\text{insert}(\text{single-node trie for } c \text{ with weight } f[c])$

while $Q.\text{size}() > 1$

$T_1 \leftarrow Q.\text{deleteMin}()$, $f_1 \leftarrow$ weight of T_1

$T_2 \leftarrow Q.\text{deleteMin}()$, $f_2 \leftarrow$ weight of T_2

$Q.\text{insert}(\text{trie with } T_1, T_2 \text{ as subtrees and weight } f_1 + f_2)$

$T \leftarrow Q.\text{deleteMin}()$ // trie for decoding

reset input-stream S // read all of S , need to read again for encoding

PrefixFree::encoding(T, S, C) // perform actual encoding $O(|\Sigma_S| + |C|)$

- Total time is $O(|\Sigma_S| \log |\Sigma_S| + |C|)$
 - $n < |C|$

Huffman Coding Discussion

- We require $|\Sigma_S| \geq 2$
- Constructed trie is **not unique**
 - so decoding trie must be transmitted along with the coded text
 - this may make encoding bigger than source text!
- Encoding must pass through stream twice
 1. to compute frequencies and to encode
 2. cannot use stream unless it can be reset

- Time to compute trie T and encode S

$$O(|\Sigma_S| \log |\Sigma_S| + |C|)$$

- Decoding run-time

$$O(|C|)$$

- The constructed trie is *optimal* in the sense that the coded text C is shortest among all prefix-free character encodings with $\Sigma_C = \{0, 1\}$
 - proof is in the course book
- Many variations
 - tie-breaking rules, estimate frequencies, adaptively change encoding, etc.

Outline

- **Compression**

- Encoding Basics
- Huffman Codes
- **Run-Length Encoding**
- Lempel-Ziv-Welch
- bzip2
- Burrows-Wheeler Transform

Single-Character vs Multi-Character Encoding

- **Single character encoding:** each source-text character receives one codeword

$S =$ b a n a n a
 └┘ └┘ └┘ └┘ └┘ └┘
 01 1 11 1 11 1

- **Multi-character encoding:** multiple source-text characters can receive one codeword

$S =$ b a n a n a
 └┘ └┐ └┐ └┘ └┐ └┘
 01 11 101

Run-Length Encoding

- RLE is an example of multi-character encoding
- Source alphabet and coded alphabet are both binary: $\Sigma = \{0, 1\}$
 - can be extended to non-binary alphabets
- Useful S has long runs of the same character: 00000 111 0000
- Dictionary is uniquely defined by *algorithm*
 - no need to store it explicitly

Run-Length Encoding

- **Encoding idea**

- give the first bit of S (either 0 or 1)
- then give a sequence of integers indicating run lengths
- do not have to give the bit for runs since they alternate

- Example 00000 111 0000

- becomes: 0 5 3 4

- Need to encode run length in binary, how?

- cannot use variable length binary encoding 10111100
 - do not know how to parse in individual run lengths
- fixed length binary encoding (say 16 bits) wastes space, bad compression
 - 0000000000000001010000000000000011000000000000100

Towards Prefix-free Encoding for Positive Integers

- To know where each number begins/ends, need to know number length
 - first say how many digits the number has
 - by printing as many zeros as the number of digits
 - then print the actual number

<i>number</i>		<i>encoding</i>
1	→	0 1
10	→	00 10
11	→	00 11
100	→	000 100
101	→	000 101

- The first zero is actually not necessary
 - $\# \text{ digits} = \# \text{ zeros} + 1$
 - get shorter encoding if remove the first zero

Prefix-free Encoding for Positive Integers

- Use *Elias gamma code* to encode k
 - $\lfloor \log k \rfloor$ copies of 0, followed by
 - binary representation of k (always starts with 1)

k	$\lfloor \log k \rfloor$	k in binary	encoding
1	0	1	1
2	1	10	010
3	1	11	011
4	2	100	00100
5	2	101	00101
6	2	110	00110

- Easy to decode
 - (number of zeros+1) tells you the length of k (in binary representation)
 - after zeros, read binary representation of k (it starts with 1)

RLE Example: Encoding

k	$\lceil \log k \rceil$	k in binary	encoding
1	0	1	1
2	1	10	010
3	1	11	011
4	2	100	00100
5	2	101	00101
6	2	110	00110
7	2	111	00111

- Encoding

$S =$ **1**111111001000000000000000000000001111111111

$C =$ **1**

RLE Example: Encoding

k	$\lceil \log k \rceil$	k in binary	encoding
1	0	1	1
2	1	10	010
3	1	11	011
4	2	100	00100
5	2	101	00101
6	2	110	00110
7	2	111	00111

- Encoding

$S =$ **111111**0010000000000000000000001111111111

$k = 7$

$C = 1$ **00111**

RLE Example: Encoding

k	$\lceil \log k \rceil$	k in binary	encoding
1	0	1	1
2	1	10	010
3	1	11	011
4	2	100	00100
5	2	101	00101
6	2	110	00110
7	2	111	00111

- Encoding

$S =$ ~~1111111~~**00**1000000000000000000000001111111111

$k = 2$

$C = 100111$ **010**

RLE Example: Encoding

k	$\lceil \log k \rceil$	k in binary	encoding
1	0	1	1
2	1	10	010
3	1	11	011
4	2	100	00100
5	2	101	00101
6	2	110	00110
7	2	111	00111

- Encoding

$S =$ ~~111111100~~**1**0000000000000000000000001111111111

$k = 1$

$C = 1001110101$ **1**

RLE Example: Encoding

k	$\lceil \log k \rceil$	k in binary	encoding
1	0	1	1
2	1	10	010
3	1	11	011
4	2	100	00100
5	2	101	00101
6	2	110	00110
7	2	111	00111
20	4	10100	000010100

- Encoding

$S =$ ~~1111111001~~**00000000000000000000**1111111111

$k = 20$

$C =$ 1001110101 **000010100**

RLE Example: Encoding

k	$\lfloor \log k \rfloor$	k in binary	encoding
1	0	1	1
2	1	10	010
3	1	11	011
4	2	100	00100
5	2	101	00101
6	2	110	00110
7	2	111	00111
11	3	1011	0001011

- Encoding

$S =$ ~~11111110010000000000000000000000~~1111111111

$$k = 11$$

$C = 1001110101000010100$ **0001011**

- Compression ratio

$$26/41 \approx 63\%$$

RLE Encoding

RLE::encoding(S, C)

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

$b \leftarrow S.top()$

$C.append(b)$

while S is non-empty **do**

$k \leftarrow 1$ // initialize run length

while (S is non-empty and $S.top() = b$) //compute run length

$k++$; $S.pop()$

// compute Elias gamma code K (binary string) for k

$K \leftarrow$ empty string

while ($k > 1$)

$C.append(0)$ // 0 appended to output C directly

$K.prepend(k \bmod 2)$ // K is built from last digit forwards

$k \leftarrow \lfloor k/2 \rfloor$

$K.prepend(1)$ // the very first digit of K was not computed

$C.append(K)$

$b \leftarrow 1 - b$

RLE Example: Decoding

- Recall that $(\# \text{ zeros} + 1)$ tells you the length of k in binary representation

- Decoding

$C = 00001101001001010$

$b = 0$

$l = 4$

$k = 13$

$S = 000000000000000$

k	$\lfloor \log k \rfloor$	k in binary	encoding
1	0	1	1
2	1	10	010
3	1	11	011
4	2	100	00100
5	2	101	00101
6	2	110	00110
13	3	1101	0001101

RLE Example: Decoding

- Decoding

$C = 00001101001001010$

$b = 1$

$l = 3$

$k = 4$

$S = 000000000000001111$

k	$\lceil \log k \rceil$	k in binary	encoding
1	0	1	1
2	1	10	010
3	1	11	011
4	2	100	00100
5	2	101	00101
6	2	110	00110
13	3	1101	0001101

RLE Example: Decoding

- Decoding

$C = 00001101001001010$

$b = 0$

$l = 1$

$k = 1$

$S = 0000000000000011110$

k	$\lceil \log k \rceil$	k in binary	encoding
1	0	1	1
2	1	10	010
3	1	11	011
4	2	100	00100
5	2	101	00101
6	2	110	00110
13	3	1101	0001101

RLE Example: Decoding

- Decoding

$C =$ ~~00001101001001~~**010**

$b =$ **1**

$l =$ **2**

$k =$ **2**

$S =$ 0000000000000011110**11**

k	$\lceil \log k \rceil$	k in binary	encoding
1	0	1	1
2	1	10	010
3	1	11	011
4	2	100	00100
5	2	101	00101
6	2	110	00110
13	3	1101	0001101

RLE Decoding

RLE-Decoding(C, S)

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

$b \leftarrow C.\text{pop}()$ // bit-value for the first run

while C is non-empty

$l \leftarrow 0$ // length of base-2 number - 1

while $C.\text{pop}() = 0$

$l++$

$k \leftarrow 1$ // base-2 number converted

for ($j = 1$ to l) // translate k from binary string to integer

$k \leftarrow k * 2 + C.\text{pop}()$

// if C runs out of bits then encoding was invalid

for ($j = 1$ to k)

$S.\text{append}(b)$

$b \leftarrow 1 - b$ // alternate bit-value

RLE Properties

- Variable length encoding
- Dictionary is uniquely defined by an algorithm
 - no need to explicitly store or send dictionary
- Best compression (for most n) is for $S = 000 \dots 000$ of length n
 - compressed to $2\lfloor \log n \rfloor + 2 \in o(n)$ bits
 - 1 for the initial bit
 - $\lfloor \log n \rfloor$ zeros to encode the length of binary representation of integer n
 - $\lfloor \log n \rfloor + 1$ digits that represent n itself in binary
- Usually not that lucky
 - no compression until run-length $k \geq 6$
 - **expansion** when run-length $k = 2$ or 4
- Method can be adapted to larger alphabet sizes
 - but then the encoding for each run must also store the character
- Method can be adapted to encode only runs of 0
 - we will need this soon

Outline

- **Compression**

- Encoding Basics
- Huffman Codes
- Run-Length Encoding
- **Lempel-Ziv-Welch**
- bzip2
- Burrows-Wheeler Transform

Longer Patterns in Input

- Huffman and RLE take advantage of frequent or repeated *single characters*
- **Observation**: certain *substrings* are much more frequent than others
- Examples
 - English text
 - most frequent digraphs: TH, ER, ON, AN, RE, HE, IN, ED, ND, HA
 - most frequent trigraphs: THE, AND, THA, ENT, ION, TIO, FOR, NDE
 - HTML
 - “<a href”, “<img src”, “
”
 - Video
 - repeated background between frames, shifted sub-image
- Could find the most frequent substrings of length up to k and store them in a dictionary (in addition to characters, i.e. strings of length 1)

	null	start of heading	start of text	...	A	...	delete	er	in	...	ed	the
code	0	1	2	...	65	...	127	128	129	...		255
code in binary	00000000	00000001	00000010		001000001		01111111	11000001	11000010	...	11111110	11111111

- however, each text has its own set of most frequently occurring substrings

Lempel-Ziv-Welch Compression

- **Ingredient 1** for Lempel-Ziv-Welch compression
 - encode characters and frequent substrings
 - discover and encode frequent substring as we process text
 - no need to know frequent substrings beforehand


Adaptive Dictionaries

- ASCII and RLE use *fixed* dictionaries
 - same dictionary for any text encoded
 - no need to pass dictionary to the decoder
- Huffman's dictionary is not *fixed* but it is *static*
 - dictionary is not fixed: each text has its own dictionary
 - dictionary is *static*: dictionary *does not change* for entire encoding/decoding
 - need to pass dictionary to the decoder
- **Ingredient 2** for LZW: *adaptive dictionary*
 - dictionary constructed during encoding/decoding
 - start with some initial dictionary D_0
 - usually ASCII
 - at iteration $i \geq 0$, D_i is used to determine the i th output
 - after iteration i , update D_i to D_{i+1}
 - a new character combination is inserted
 - encoder and decoder both know the algorithm for how dictionary changes
 - no need to send dictionary with the encoding, like with RLE

LZW Encoding: Main Idea

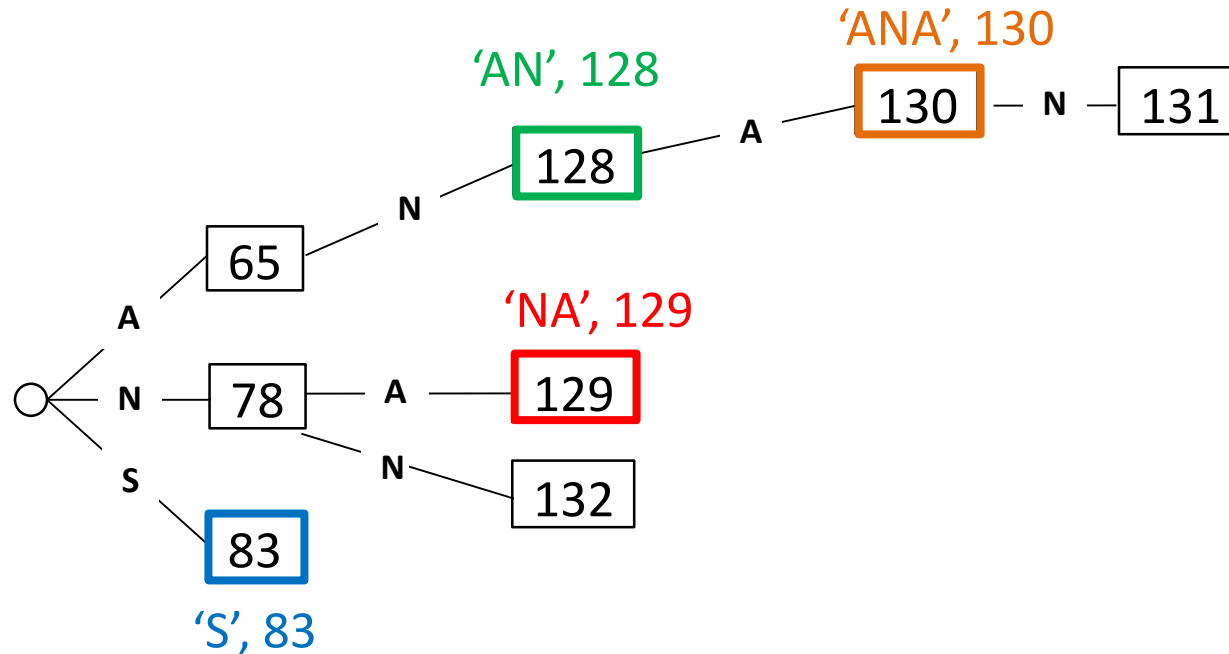
- Iteration i of encoding
- Current $D_i = \{a:65, b: 66, ab:140, bb:145, bbc:146\}$

S = abbbcbad



- find longest substring that starts at current pointer and is in the dictionary
- encode 'bb' with 145
- $D_{i+1} = D_i.\text{insert}(\text{'bba'}, \text{next_available_codenumber})$
- logic: 'bba' would have been useful at iteration i , so it is likely useful in the future
- meaning of 'codenumber', 'codeword' and 'code' is the same

Tries for LZW Encoding

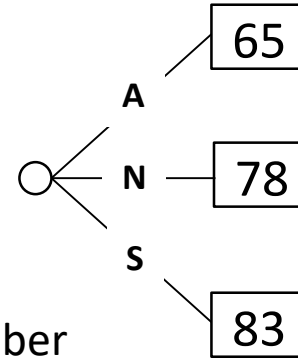


- Trie stores codenumbers at all nodes (external and *internal*) except the root
 - works because a string is inserted only after all its prefixes are inserted
- Read the string corresponding to each codenumber from the edges

LZW Example

- Start dictionary D

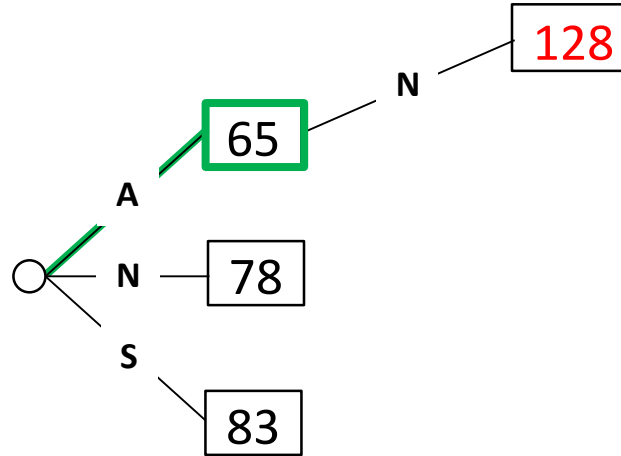
- ASCII characters
- codes from 0 to 127
- next inserted code will be 128
- variable idx keeps track of next available codenumber
- initialize $idx = 128$



- Text A N A N A N A N N A

LZW Example

- Dictionary D
 - $idx = 129$



- Text

A	N
---	---

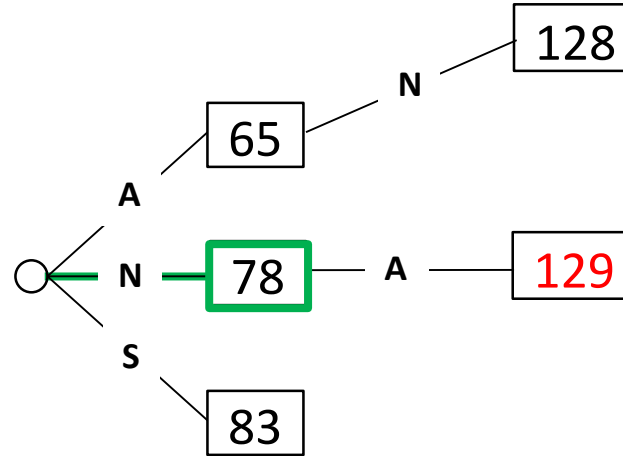
 A N A N A N N A
- Encoding

65

- Add to dictionary “string just encoded” + “first character of next string to be encoded”
- Inserting new item is $O(1)$ since we stopped at the right node in the trie when we searched for ‘A’

LZW Example

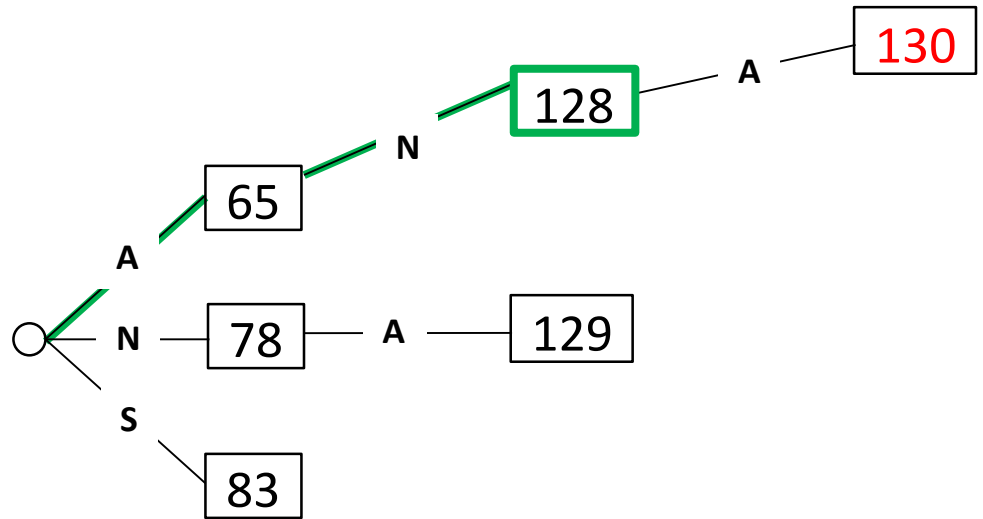
- Dictionary D
 - $idx = 130$



- Text A N A N A N A N N A
- Encoding 65 78

LZW Example

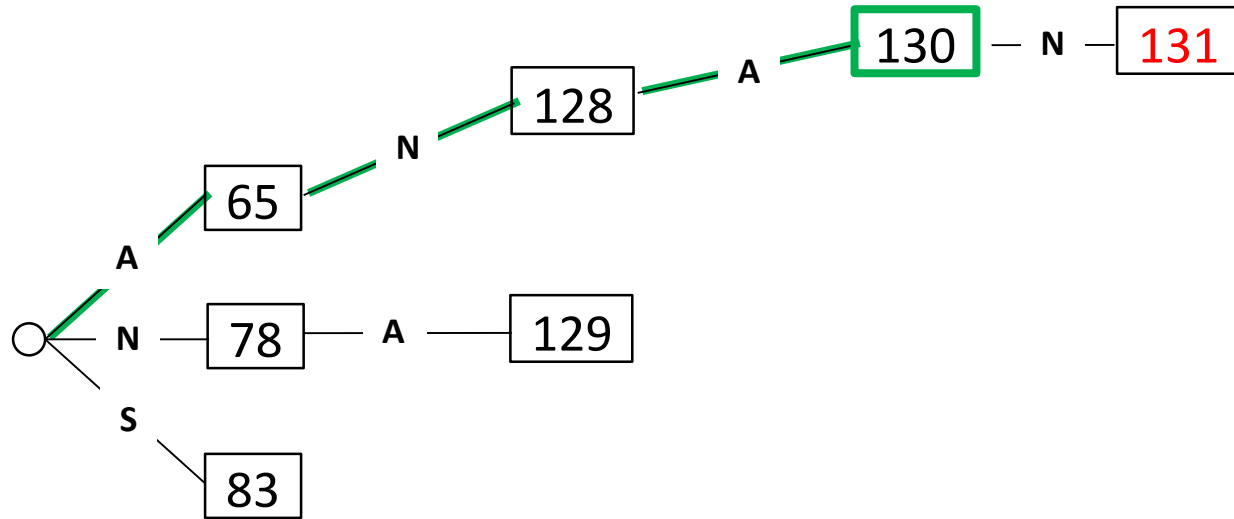
- Dictionary D
 - $idx = 131$



	add to dictionary									
Text	A	N	A	N	A	N	A	N	N	A
Encoding	65	78	128							

LZW Example

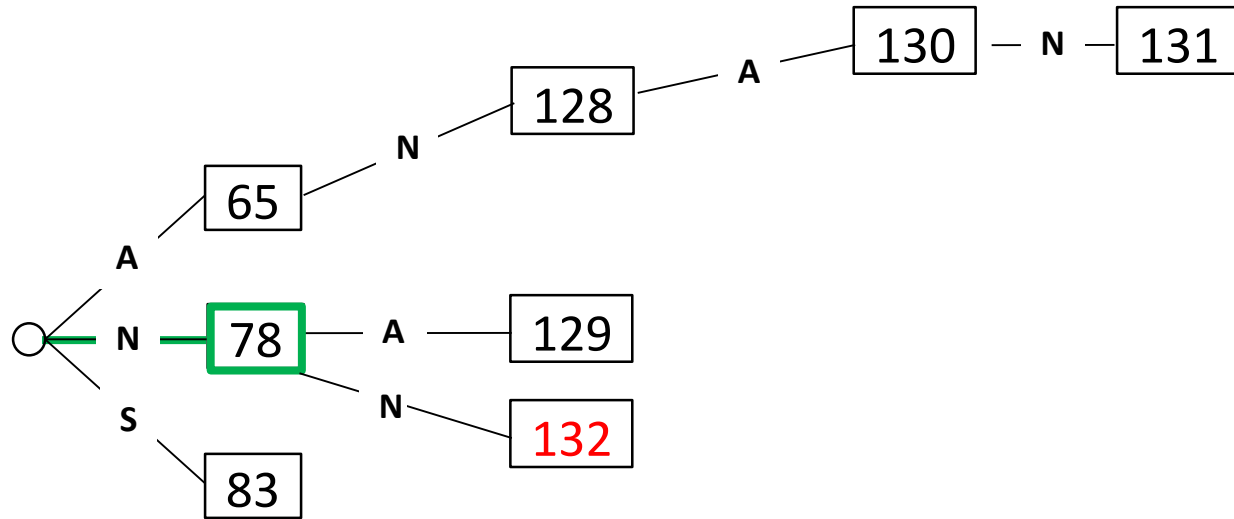
- Dictionary D
 - $idx = 132$



Text	A	N	A	N	A	N	A	N	A
Encoding	65	78	128		130				

LZW Example

- Dictionary D
 - $idx = 133$



Text	A	N	A	N	A	N	A	N	A
Encoding	65	78	128		130		78		

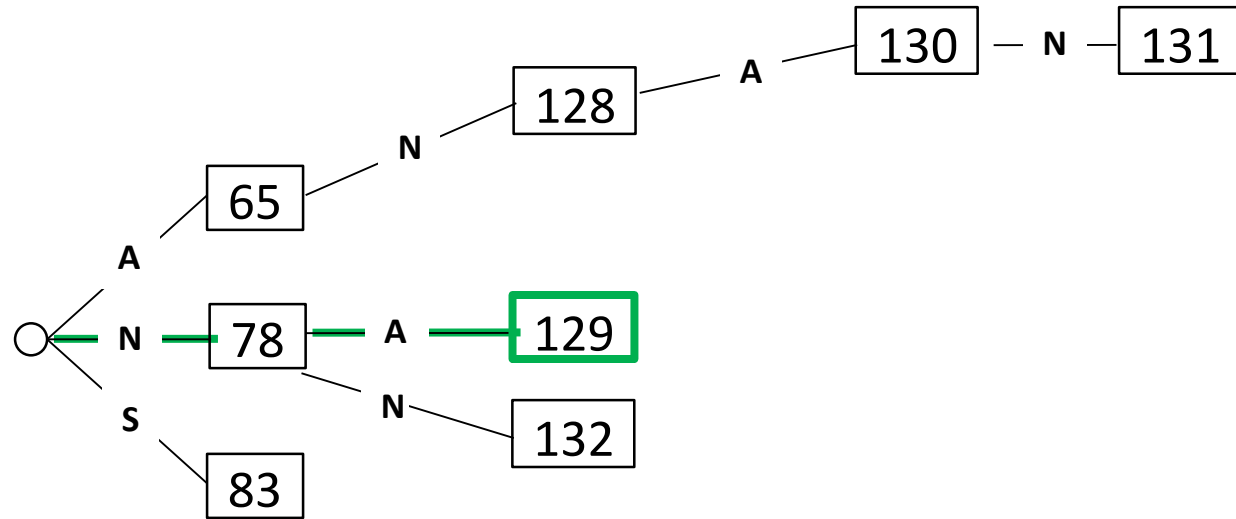
add to dictionary

N N

78

LZW Example

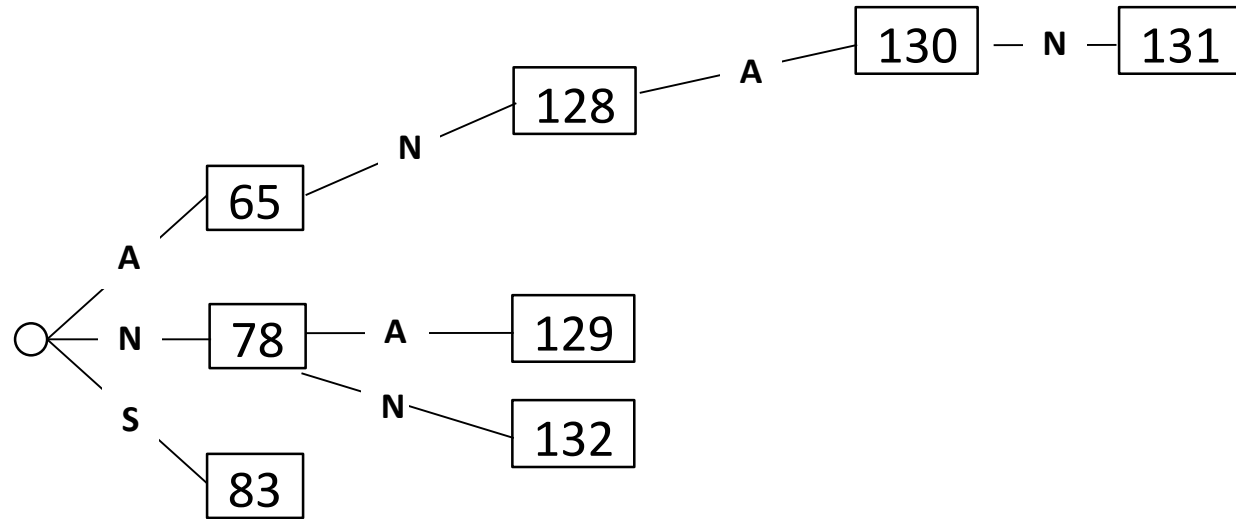
- Dictionary D
 - $idx = 133$



Text	A	N	A	N	A	N	A	N	N	A
Encoding	65	78	128		130		78	129		

LZW Example

- Dictionary D
 - $idx = 133$



- Text A N A N A N A N N A
- Encoding 65 78 128 130 78 129
- Final output 00001000001 00001001110 0001000000 0001000010 00001001110 00010000001
- Use fixed length (12 bits) per codenumber
 - 12 bit binary string representation for each code
 - total of $2^{12} = 4096$ codesnumbers available during encoding
 - if you run out of codenumbers, stop inserting new elements in the dictionary

LZW encoding pseudocode

LZW::encoding(S, C)

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

initialize dictionary D with ASCII in a trie

$idx \leftarrow 128$

while S is not empty **do**

$v \leftarrow$ root of trie D

while S is non-empty and v has a child c labelled $S.top()$

$v \leftarrow c$

$S.pop()$

trie
search

$C.append$ (codenumber stored at v)

if S is non-empty

 create child of v labelled $S.top()$ with code idx

$idx++$

new
dictionary
entry

- Running time is $O(|S|)$

LZW Encoder vs Decoder

- For decoding, need a dictionary
- Construct a dictionary during decoding, but one step behind
 - at iteration i of decoding can reconstruct substring which encoder inserted into dictionary at iteration $i - 1$
 - delay is due to not having access to the original text

LZW Decoding Example

- Given encoding to decode back to the source text

65 78 128 130 78 129

- Build dictionary adaptively, while decoding
- Decoding starts with the same initial dictionary as encoding
 - use array instead of trie, need D that allows efficient search by code
- We will show the original text during decoding in this example, but just for reference
 - do not need original text to decode

initial D

65	A
78	N
83	S

$idx = 128$

LZW Decoding Example

$i=0$

- Text A N A N A N A N N A
- Encoding 65 78 128 130 78 129
- Decoding
iter $i = 0$ A

$D =$

65	A
78	N
83	S

$idx = 128$

- First step: $s = D(65) = A$
- Encoding iteration $i = 0$
 - looked ahead in the text, saw N, and added AN to the dictionary
- Decoding iteration $i = 0$
 - know text starts with A, but cannot look ahead as the text is not available
 - no new word added at iteration $i = 0$
 - keep track of s_{prev} = string decoded at previous iteration
 - s_{prev} is also string encoder encoded at previous iteration

LZW Decoding Example

Text

$i=0$
A N A N A N N A

Encoding

65 78 128 130 78 129

Decoding

$i=1$
A N

iter $i = 1$

$D =$

65	A
78	N
83	S
128	AN

$idx = 129$

▪ $s_{prev} = A$

▪ string encoded/decoded at previous iteration

▪ First step: $s = D(78) = N$

▪ The first letter of s is exactly what the encoder looked ahead at during previous iteration!

▪ And we know which string encoder encoded at the previous iteration, we stored it in s_{prev}

▪ So at previous iteration, encoder added to the dictionary $s_{prev} + s[0]$

A N

▪ Starting at iteration $i = 1$ of decoding

▪ add $s_{prev} + s[0]$ to dictionary

▪ encoder added this string at previous iteration

LZW Decoding Example Continued

Text	A	N	A	N	A	N	A	N	N	A
Encoding	65	78	128		130		78		129	
Decoding	A	N	AN							

iter $i = 2$

$D =$

65	A
78	N
83	S
128	AN
129	NA

$idx = 130$

- $s_{prev} = N$
 - string encoded/decoded at previous iteration
- First step: $s = D(128) = AN$
- Next step: add to dictionary $s_{prev} + s[0]$

$$N + A = NA$$
 - this is the string encoder added to the dictionary at the previous iteration

LZW Decoding Example

Text	A	N	A	N	A	N	A	N	N	A
Encoding	65	78	128		130			78	129	
Decoding	A	N	AN		$s = ???$					

iter $i = 3$

$D =$

65	A
78	N
83	S
128	AN
129	NA

$idx = 130$

- $s_{prev} = AN$
 - string encoded/decoded at previous iteration
- First step: $s = D(130) = ???$ (code 130 is not in D)
- Dictionary is exactly one step behind at decoding
- Encoder added ($s, 130$) to D at previous iteration
- What did the encoder add at the previous iteration?
- We have derived a rule for it, encoder added

$$\begin{aligned}
 & \overset{\text{known}}{s_{prev}} + \overset{\text{unknown}}{s[0]} = \overset{\text{unknown}}{s} \\
 & \text{AN} + s[0] = s \\
 & \quad \quad \quad \leftarrow \text{red arrow} \rightarrow \\
 & s[0] = s_{prev}[0] = A \\
 & \quad \quad \quad \text{ANA} = s
 \end{aligned}$$

LZW Decoding Example

	$i=2$									
Text	A	N	A	N	A	N	A	N	N	A
Encoding	65	78	128		130		78		129	
Decoding	A	N	AN		ANA					

iter $i = 3$

$D =$

65	A
78	N
83	S
128	AN
129	NA
130	ANA

$idx = 131$

- General rule: if code C is not in D
 - $s = s_{prev} + s_{prev}[0]$
- in our example, $s_{prev} = AN$
 - $s = AN + A = ANA$
- Now that we recovered s , continue as usual
- Add to dictionary $s_{prev} + s[0]$

LZW Decoding Example

Text	A	N	A	N	A	N	A	N	N	A
Encoding	65	78	128		130		78	129		
Decoding	A	N	AN		ANA		N			

iter $i = 4$

$D =$

65	A
78	N
83	S
128	AN
129	NA
130	ANA
131	ANAN

$idx = 132$

- $s_{prev} = \text{ANA}$
- If code C is not in D

$$s = s_{prev} + s_{prev}[0]$$
- Add to dictionary $s_{prev} + s[0]$

LZW Decoding Example

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

iter $i = 5$

$D =$

65	A
78	N
83	S
128	AN
129	NA
130	ANA
131	ANAN

$idx = 132$

- $s_{prev} = N$
- If code C is not in D

$$s = s_{prev} + s_{prev}[0]$$
- Add to dictionary $s_{prev} + s[0]$

LZW decoding

- To save space, store new codes using its prefix code + one character
 - for each codeword, can find corresponding string s in $O(|s|)$ time

$D =$

65	A
78	N
83	S
128	AN
129	NA
130	ANA
131	ANAN


wasteful storage

65	A
78	N
83	S
128	65, N
129	78, A
130	128, A
131	130, N

ANAN

65, NAN

128, AN



LZW decoding, Another Example

Encoding: 98 97 114 128 114 97 131 134 129 101 135

$D =$

next
available
code

code	string (human)	string (implementation)
97	A	
98	B	
101	E	
110	N	
114	R	
128		

LZW decoding, Another Example

Encoding: 98 97 114 128 114 97 131 134 129 101 135

Decoding: B

$D =$

code	string (human)	string (implementation)
97	A	
98	B	
101	E	
110	N	
114	R	
128		

$s = B$

nothing added to dictionary at iteration 0

LZW decoding, Another Example

Encoding: 98 97 114 128 114 97 131 134 129 101 135

Decoding: B A

$D =$

code	string (human)	string (implementation)
97	A	
98	B	
101	E	
110	N	
114	R	
128	BA	98, A

$s_{prev} = B, code_{prev} = 98$

$s = A$

add to dictionary $s_{prev} + s[0] = BA$

LZW decoding, Another Example

Encoding: 98 97 114 128 114 97 131 134 129 101 135

Decoding: B A R

$D =$

code	string (human)	string (implementation)
97	A	
98	B	
101	E	
110	N	
114	R	
128	BA	98, A
129	AR	97, R

$s_{prev} = A, code_{prev} = 97$

$s = R$

add to dictionary $s_{prev} + s[0] = AR$

LZW decoding, Another Example

Encoding: 98 97 114 128 114 97 131 134 129 101 135

Decoding: B A R BA

$D =$

code	string (human)	string (implementation)
97	A	
98	B	
101	E	
110	N	
114	R	
128	BA	98, A
129	AR	97, R
130	RB	114, B

$s_{prev} = R, code_{prev} = 114$

$s = BA$

add to dictionary $s_{prev} + s[0] = RB$

LZW decoding, Another Example

Encoding: 98 97 114 128 114 97 131 134 129 101 135

Decoding: B A R BA R

$D =$

code	string (human)	string (implementation)
97	A	
98	B	
101	E	
110	N	
114	R	
128	BA	98, A
129	AR	97, R
130	RB	114, B
131	BAR	128, R

$s_{prev} = BA, code_{prev} = 128$

$s = R$

add to dictionary $s_{prev} + s[0] = BAR$

LZW decoding, Another Example

Encoding: 98 97 114 128 114 97 131 134 129 101 135
Decoding: B A R BA R A

$D =$

code	string (human)	string (implementation)
97	A	
98	B	
101	E	
110	N	
114	R	
128	BA	98, A
129	AR	97, R
130	RB	114, B
131	BAR	128, R
132	RA	114, A

$s_{prev} = R, code_{prev} = 114$
 $s = A$

add to dictionary $s_{prev} + s[0] = RA$

LZW decoding, Another Example

Encoding: 98 97 114 128 114 97 131 134 129 101 135

Decoding: B A R BA R A BAR

$D =$

code	string (human)	string (implementation)
97	A	
98	B	
101	E	
110	N	
114	R	
128	BA	98, A
129	AR	97,R
130	RB	114,B
131	BAR	128,R
132	RA	114,A
133	AB	97, B

$s_{prev} = A, code_{prev} = 97$
 $s = BAR$

add to dictionary $s_{prev} + s[0] = AB$

LZW decoding, Another Example

Encoding: 98 97 114 128 114 97 131 134 129 101 135

Decoding: B A R BA R A BAR BARB

$D =$

code	string (human)	string (implementation)
97	A	
98	B	
101	E	
110	N	
114	R	
128	BA	98, A
129	AR	97, R
130	RB	114, B
131	BAR	128, R
132	RA	114, A
133	AB	97, B
134	BARB	131, B

$$s_{prev} = \text{BAR}, code_{prev} = 131$$
$$s = ?$$

if code is not in dictionary

$$s = s_{prev} + s_{prev}[0]$$

$$s = \text{BAR} + \text{B} = \text{BARB}$$

add to dictionary $s_{prev} + s[0] = \text{BARB}$

LZW decoding, Another Example

Encoding: 98 97 114 128 114 97 131 134 129 101 135

Decoding: B A R BA R A BAR BARB AR

$D =$

code	string (human)	string (implementation)
97	A	
98	B	
101	E	
110	N	
114	R	
128	BA	98, A
129	AR	97, R
130	RB	114, B
131	BAR	128, R
132	RA	114, A
133	AB	97, B
134	BARB	131, B
135	BARBA	134, A

$s_{prev} = \text{BARB}, code_{prev} = 134$

$s = \text{AR}$

add to dictionary $s_{prev} + s[0] = \text{BARBA}$

LZW decoding, Another Example

Encoding: 98 97 114 128 114 97 131 134 129 101 135

Decoding: B A R BA R A BAR BARB AR E

$D =$

code	string (human)	string (implementation)
97	A	
98	B	
101	E	
110	N	
114	R	
128	BA	98, A
129	AR	97, R
130	RB	114, B
131	BAR	128, R
132	RA	114, A
133	AB	97, B
134	BARB	131, B
135	BARBA	134, A
136	ARE	129, E

$s_{prev} = \text{AR}, code_{prev} = 129$

$s = \text{E}$

add to dictionary $s_{prev} + s[0] = \text{ARE}$

LZW decoding, Another Example

Encoding: 98 97 114 128 114 97 131 134 129 101 135

Decoding: B A R BA R A BAR BARB AR E BARBA

$D =$

code	string (human)	string (implementation)
97	A	
98	B	
101	E	
110	N	
114	R	
128	BA	98, A
129	AR	97,R
130	RB	114,B
131	BAR	128,R
132	RA	114,A
133	AB	97,B
134	BARB	131,B
135	BARBA	134,A
136	ARE	129,E

$$s_{prev} = E$$
$$s = \text{BARBA}$$

LZW Decoding Pseudocode

LZW::decoding(C, S)

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

$D \leftarrow$ dictionary that maps $\{0, \dots, 127\}$ to ASCII

$idx \leftarrow 128$ // next available code

$code \leftarrow C.pop()$

$s \leftarrow D(code)$

$S.append(s)$

while there are more codes in C **do**

$S_{prev} \leftarrow S$

$code \leftarrow C.pop()$

if $code < idx$

$s \leftarrow D(code)$ //code in D , look up string s

if $code = idx$ // code not in D yet, reconstruct string

$S \leftarrow S_{prev} + S_{prev}[0]$

else **Fail** // invalid encoding

$S.append(s)$

$D.insert(idx, S_{prev} + s[0])$

$idx ++$

- Running time is $O(|S|)$

LZW Summary

- Encoding is $O(|S|)$ time, uses a trie of encoded substrings to store the dictionary
- Decoding is $O(|S|)$ time, uses an array indexed by code numbers to store the dictionary
- Encoding and decoding need to go through the string only one time and do not need to see the whole string
 - can do compression while streaming the text
- Works badly if no repeated substrings
 - dictionary gets bigger, but no new useful substrings inserted
- In practice, compression rate is around 45% on English text

Lempel-Ziv Family

- Lempel-Ziv is a family of *adaptive* compression algorithms
 - **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
 - patent issues

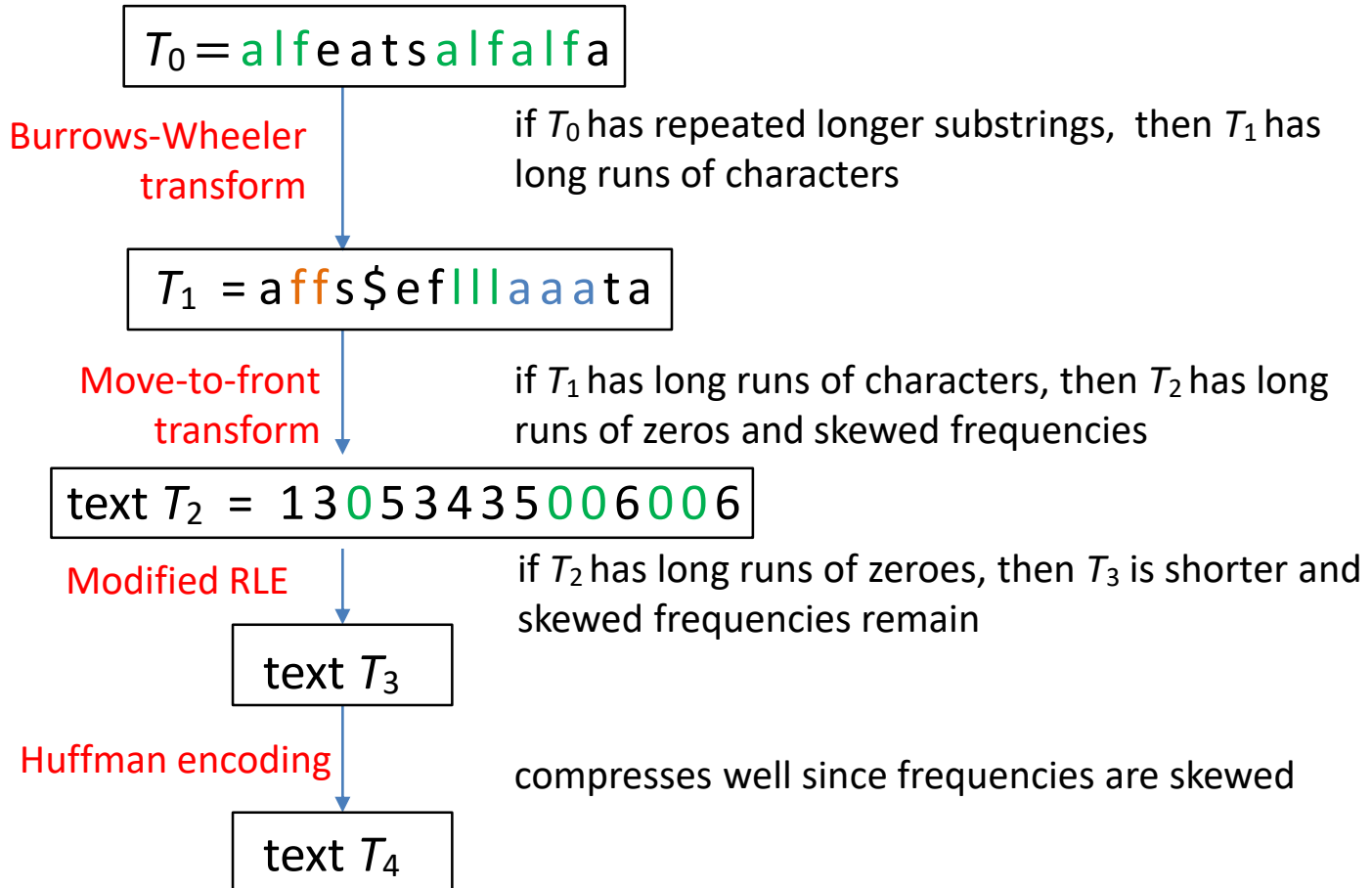
Outline

- **Compression**

- Encoding Basics
- Huffman Codes
- Run-Length Encoding
- Lempel-Ziv-Welch
- **bzip2**
- Burrows-Wheeler Transform

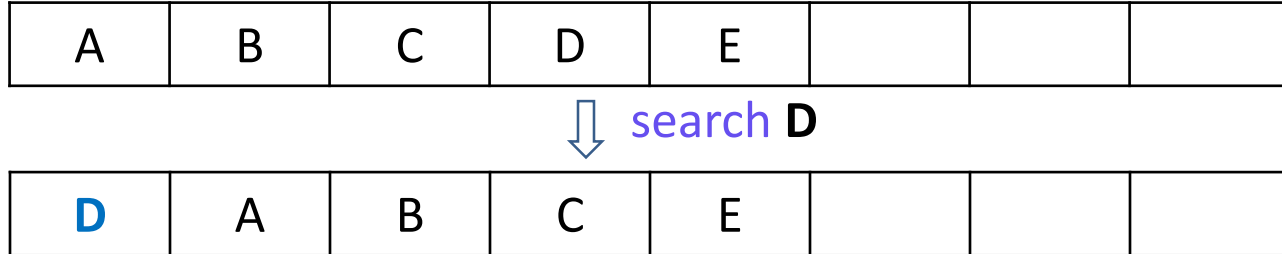
Overview of bzip2

- **Text transform** changes input text into a *different text*
 - not necessarily shorter
 - but has properties likely to lead to better compression at a later stage
- To achieve better compression, bzip2 uses the following text transforms



Move-to-Front transform

- Recall the MTF heuristic
 - after an element is accessed, move it to array front



- Use this idea for **MTF** (move to front) text transformation
 - transformed text is likely to have text with repeated zeros and skewed frequencies

MTF Encoding Example

- Source alphabet Σ_S with size $|\Sigma_S| = m$
- Put alphabet in array L , initially in sorted order, but allow L to get unsorted

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

- This gives encoding dictionary L
 - single character encoding E
- Code of any character = index of array where character stored in dictionary L
 - $E('B') = 1$
 - $E('H') = 7$
- After each encoding, update L with Move-To-Front heuristic
- Coded alphabet is $\Sigma_C = \{0, 1, \dots, m - 1\}$
- Change dictionary D dynamically (like LZW)
 - unlike LZW
 - no new items added to dictionary
 - codeword for one or more letters can change at each iteration

MTF Encoding Example

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

$S =$ MISSISSIPPI

$C =$

MTF Encoding Example

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

$S = \text{MISSISSIPPI}$

$C = 12$

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

$S = \text{MISSISSIPPI}$

$C = 12 \ 9$

MTF Encoding Example

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

$S = \text{MISSISSIPPI}$

$C = 12\ 9\ 18$

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

$S = \text{MISSIPPI}$

$C = 12\ 9\ 18\ 0$

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

S = MISSISSIPPI

C = 12 9 18 0 **1**

MTF Encoding Example

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

S = ~~MISSI~~SSIPPI

C = 12 9 18 0 1 1

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

S = MISSISSIPPI

C = 12 9 18 0 1 1 0

MTF Encoding Example

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

$S = \text{MISSISSIPPI}$

$C = 12\ 9\ 18\ 0\ 1\ 1\ 0\ 1\ 16\ 0\ 1$

- What does a run in C mean about the source S ?
 - zeros tell us about consecutive character runs

MTF Decoding Example

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

$S =$

$C = 12\ 9\ 18\ 0\ 1\ 1\ 0\ 1\ 16\ 0\ 1$

- Decoding is similar
- Start with the same dictionary D as encoding
- Apply the same MTF transformation at each iteration
 - dictionary D undergoes exactly the transformations when decoding
 - no delays, identical dictionary at encoding and decoding iteration i
 - can always decode original letter

MTF Decoding Example

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

$S = M$

$C = 12\ 9\ 18\ 0\ 1\ 1\ 0\ 1\ 16\ 0\ 1$

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

$S = M$ I

$C = 12$ 9 18 0 1 1 0 1 16 0 1

MTF Decoding Example

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

$S = M \ I \ S$

$C = 12 \ 9 \ 18 \ 0 \ 1 \ 1 \ 0 \ 1 \ 16 \ 0 \ 1$

Move-to-Front Transform: Properties



- If a character in S repeats k times, then C has a run of $k - 1$ zeros
- C contains a lot of small numbers and a few big ones
- C has the same length as S , but better properties for encoding

Move-to-Front Encoding/Decoding Pseudocode

MTF::encoding(S, C)

$L \leftarrow$ array with Σ_S in some pre-agreed, fixed order (i.e. ASCII)

while S is non-empty **do**

$c \leftarrow S.\text{pop}()$

$i \leftarrow$ index such that $L[i] = c$

for $j = i - 1$ down to 0

 swap $L[j]$ and $L[j + 1]$

MTF::decoding(C, S)

$L \leftarrow$ array with Σ_S in some pre-agreed, fixed order (i.e. ASCII)

while C is non-empty **do**

$i \leftarrow$ next integer of C

$S.\text{append}(L[i])$

for $j = i - 1$ down to 0

 swap $L[j]$ and $L[j + 1]$

Move-to-Front Transform Summary

■ MTF text transform

- source alphabet is Σ_S with size $|\Sigma_S| = m$
- store alphabet in an array
 - code of any character = index of array where character stored
 - coded alphabet is $\Sigma_C = \{0, 1, \dots, m - 1\}$
- Dictionary is adaptive
 - nothing new is added, but meaning of codewords are changed
- MTF is an *adaptive* text-transform algorithm
 - it does not compress input
 - the output has the same length as input
 - but output has better properties for compression

Outline

- **Compression**

- Encoding Basics
- Huffman Codes
- Run-Length Encoding
- Lempel-Ziv-Welch
- bzip2
- **Burrows-Wheeler Transform**

Burrows-Wheeler Transform

- Transformation (not compression) algorithm
 - transforms source text to coded text with same letters but in different order
 - source and coded alphabets are the same
 - if original text had frequently occurring substrings, transformed text should have many runs of the same character
 - more suitable for MTF transformation*



- Required: the source text S ends with *end-of-word character* $\$$
 - $\$$ occurs nowhere else in S
- Based on *cyclic* shifts for a string
 - example
 - string
 a b c d e
 - cyclic shift
 c d e a b
- Formal definition
 - a *cyclic shift* of string X of length n is the concatenation of $X[i + 1 \dots n - 1]$ and $X[0 \dots i]$, for $0 \leq i < n$



BWT Algorithm and Example

S = a l f e a t s a l f a l f a \$

- Write all consecutive cyclic shifts
 - forms *an array of shifts*
 - last letter in any row is the first letter of the previous row

```
a l f e a t s a l f a l f a $  
l f e a t s a l f a l f a $ a  
f e a t s a l f a l f a $ a l  
e a t s a l f a l f a $ a l f  
a t s a l f a l f a $ a l f e  
t s a l f a l f a $ a l f e a  
s a l f a l f a $ a l f e a t  
a l f a l f a $ a l f e a t s  
l f a l f a $ a l f e a t s a  
f a l f a $ a l f e a t s a l  
a l f a $ a l f e a t s a l f  
l f a $ a l f e a t s a l f a  
f a $ a l f e a t s a l f a l  
a $ a l f e a t s a l f a l f  
$ a l f e a t s a l f a l f a
```

BWT Algorithm and Example

$S = \text{alfeatsalfalfa\$}$

- Array of cyclic shifts
 - the first column is the original S

```
a l f e a t s a l f a l f a $
l f e a t s a l f a l f a $ a
f e a t s a l f a l f a $ a l
e a t s a l f a l f a $ a l f
a t s a l f a l f a $ a l f e
t s a l f a l f a $ a l f e a
s a l f a l f a $ a l f e a t
a l f a l f a $ a l f e a t s
l f a l f a $ a l f e a t s a
f a l f a $ a l f e a t s a l
a l f a $ a l f e a t s a l f
l f a $ a l f e a t s a l f a
f a $ a l f e a t s a l f a l
a $ a l f e a t s a l f a l f
$ a l f e a t s a l f a l f a
```

BWT Algorithm and Example

$S = \text{a|lfeatsa|lfa|lfa\$}$

- Array of cyclic shifts
- S has **alf** repeated 3 times
 - 3 different shifts start with **lf** and end with **a**

```
alf eatsalfalfa$  
lfeatsalfalfa$a  
featsalfalfa$alf  
eatsalfalfa$alf  
atsalfalfa$alf  
tsalfalfa$alf  
salfalfa$alf  
alfalfa$alf  
lfalfa$alf  
falfa$alf  
alfa$alf  
lfa$alf  
fa$alf  
a$alf  
$alf
```

BWT Algorithm and Example

$S = \text{a|lfeatsa|lfa|lfa\$}$

- Array of cyclic shifts
- Sort (lexographically) cyclic shifts
 - strict sorting order due to \$
- First column (of course) has many consecutive character runs
- But also the last column has many consecutive character runs
 - 3 different shifts start with **lf** and end with **a**
 - sort groups **lf** lines together, and they all end with **a**

sorted shifts array

```
$alfeatsalalfa  
a$alfeatsalalfa  
alfa$alfeatsalf  
alfalfa$alfeats  
alfeatsalalfa$  
atsalalfa$alf  
eatsalalfa$alf  
fa$alfeatsalfal  
falfa$alfeatsalf  
featsalfalfa$alf  
lfa$alfeatsalfa  
lfalfa$alfeatsa  
lfeatsalfalfa$a  
salfalfa$alf  
tsalfalfa$alf
```

BWT Algorithm and Example

$S = \text{a|l f e a t s a|l f a|l f a \$}$

- Array of cyclic shifts
- Sort (lexographically) cyclic shifts
 - strict sorting order due to '\$'
- First column (of course) has many consecutive character runs
- But also the last column has many consecutive character runs
 - 3 different shifts start with **lf** and end with **a**
 - sort groups **lf** lines together, and they all end with **a**
 - could happen that another pattern will interfere
 - **hlfd** broken into **h** and **lfd**
 - chance of interference is small

sorted shifts array

```

$ a l f e a t s a l f a l f a
a $ a l f e a t s a l f a l f
a l f a $ a l f e a t s a l f
a l f a l f a $ a l f e a t s
a l f e a t s a l f a l f a $
a t s a l f a l f a $ a l f e
e a t s a l f a l f a $ a l f
f a $ a l f e a t s a l f a l
f a l f a $ a l f e a t s a l
f e a t s a l f a l f a $ a l
l f a $ a l f e a t s a l f a
l f a l f a $ a l f e a t s a
l f d ... .. h
l f e a t s a l f a l f a $ a
s a l f a l f a $ a l f e a t
t s a l f a l f a $ a l f e a
    
```

BWT Algorithm and Example

$S = \text{a l f e a t s a l f a l f a \$}$

- Sorted array of cyclic shifts
- First column is useless for encoding
 - cannot decode it
- Last column can be decoded
- BWT Encoding
 - last characters from sorted shifts
 - i.e. the last column

$C = \text{a f f s \$ e f l l l a a t a}$

sorted shifts array

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

BWT Fast Encoding: Efficient Sorting

$S = \text{a l f e a t s a l f a l f a \$}$

i	cyclic shift
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	a t s a l f a l f a \$ a l f e
5	t s a l f a l f a \$ a l f e a
6	s a l f a l f a \$ a l f e a t
7	a l f a l f a \$ a l f e a t s
8	l f a l f a \$ a l f e a t s a
9	f a l f a \$ a l f e a t s a l
10	a l f a \$ a l f e a t s a l f
11	l f a \$ a l f e a t s a l f a
12	f a \$ a l f e a t s a l f a l
13	a \$ a l f e a t s a l f a l f
14	\$ a l f e a t s a l f a l f a

- Refer to a cyclic shift by the start index in the text, no need to write it out explicitly
- For sorting, letters after \$ do not matter

a l f a l f a \$ a l f e a t s
<
l f a \$ a l f e a t s a l f a

BWT Fast Encoding: Efficient Sorting

$S = \text{a l f e a t s a l f a l f a \$}$

i	cyclic shift
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	a t s a l f a l f a \$ a l f e
5	t s a l f a l f a \$ a l f e a
6	s a l f a l f a \$ a l f e a t
7	a l f a l f a \$ a l f e a t s
8	l f a l f a \$ a l f e a t s a
9	f a l f a \$ a l f e a t s a l
10	a l f a \$ a l f e a t s a l f
11	l f a \$ a l f e a t s a l f a
12	f a \$ a l f e a t s a l f a l
13	a \$ a l f e a t s a l f a l f
14	\$ a l f e a t s a l f a l f a

- Refer to a cyclic shift by **the start index in the text**, no need to write it out explicitly
- For sorting, letters after \$ do not matter

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

<

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

BWT Fast Encoding: Efficient Sorting

$S = \text{a l f e a t s a l f a l f a \$}$

i	cyclic shift
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	a t s a l f a l f a \$ a l f e
5	t s a l f a l f a \$ a l f e a
6	s a l f a l f a \$ a l f e a t
7	a l f a l f a \$ a l f e a t s
8	l f a l f a \$ a l f e a t s a
9	f a l f a \$ a l f e a t s a l
10	a l f a \$ a l f e a t s a l f
11	l f a \$ a l f e a t s a l f a
12	f a \$ a l f e a t s a l f a l
13	a \$ a l f e a t s a l f a l f
14	\$ a l f e a t s a l f a l f a

- Refer to a cyclic shift by the start index in the text, no need to write it out explicitly
- For sorting, letters after \$ do not matter
- This is the same as sorting suffixes of S
- We already know how to do it
 - exactly as for suffix arrays, with MSD-Radix-Sort
 - $O(n \log n)$ running time

BWT Fast Encoding: Efficient Sorting

$S = \text{a l f e a t s a l f a l f a \$}$

i	cyclic shift
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	a t s a l f a l f a \$ a l f e
5	t s a l f a l f a \$ a l f e a
6	s a l f a l f a \$ a l f e a t
7	a l f a l f a \$ a l f e a t s
8	l f a l f a \$ a l f e a t s a
9	f a l f a \$ a l f e a t s a l
10	a l f a \$ a l f e a t s a l f
11	l f a \$ a l f e a t s a l f a
12	f a \$ a l f e a t s a l f a l
13	a \$ a l f e a t s a l f a l f
14	\$ a l f e a t s a l f a l f a

j	$A^S[j]$	sorted cyclic shifts
0	14	\$ a l f e a t s a l f a l f a
1	13	a \$ a l f e a t s a l f a l f
2	10	a l f a \$ a l f e a t s a l f
3	7	a l f a l f a \$ a l f e a t s
4	0	a l f e a t s a l f a l f a \$
5	4	a t s a l f a l f a \$ a l f e
6	3	e a t s a l f a l f a \$ a l f
7	12	f a \$ a l f e a t s a l f a l
8	9	f a l f a \$ a l f e a t s a l
9	2	f e a t s a l f a l f a \$ a l
10	11	l f a \$ a l f e a t s a l f a
11	8	l f a l f a \$ a l f e a t s a
12	1	l f e a t s a l f a l f a \$ a
13	6	s a l f a l f a \$ a l f e a t
14	5	t s a l f a l f a \$ a l f e a

BWT Fast Encoding: Efficient Sorting

- Can read BWT encoding from suffix array in $O(n)$ time

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

$A^s =$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
	14	13	10	7	0	4	3	12	9	2	11	8	1	6	5



cyclic shift starts at $S[14]$

we need the last letter of that cyclic shift, it is at $S[13]$


a

BWT Fast Encoding: Efficient Sorting

- Can read BWT encoding from suffix array in $O(n)$ time

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

$A^s =$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
	14	13	10	7	0	4	3	12	9	2	11	8	1	6	5



cyclic shift starts at $S[13]$

we need the last letter of that cyclic shift, it is at $S[12]$


a f

BWT Fast Encoding: Efficient Sorting

- Can read BWT encoding from suffix array in $O(n)$ time

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

$A^s =$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
	14	13	10	7	0	4	3	12	9	2	11	8	1	6	5



cyclic shift starts at $S[10]$

we need the last letter of that cyclic shift, it is at $S[9]$


a f f

BWT Fast Encoding: Efficient Sorting

- Can read BWT encoding from suffix array in $O(n)$ time

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

$A^s =$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
	14	13	10	7	0	4	3	12	9	2	11	8	1	6	5



cyclic shift starts at $S[5]$

we need the last letter of that cyclic shift, it is at $S[4]$

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

BWT Fast Encoding: Efficient Sorting

- Can read BWT encoding from suffix array in $O(n)$

$$S =$$

0	1	2	3	4	5	6	7	8	9
a	l	f	e	a	t	s	a	l	f

$$A^s =$$

0	1	2	3	4	5	6	7	8	9
14	13	10	7	0	4	3	12	9	2

cyclic shift starts at $S[5]$

we need the last letter of that cyclic shift, it is a

j	$A^s[j]$	
0	14	\$ a l f e a t s a l f a
1	13	a \$ a l f e a t s a l f
2	10	a l f a \$ a l f e a t s
3	7	a l f a l f a \$ a l f e a t s
4	0	a l f e a t s a l f a l f a \$
5	4	a t s a l f a l f a \$ a l f e
6	3	e a t s a l f a l f a \$ a l f
7	12	f a \$ a l f e a t s a l f a l
8	9	f a l f a \$ a l f e a t s a l
9	2	f e a t s a l f a l f a \$ a l
10	11	l f a \$ a l f e a t s a l f a
11	8	l f a l f a \$ a l f e a t s a
12	1	l f e a t s a l f a l f a \$ a
13	6	s a l f a l f a \$ a l f e a t
14	5	t s a l f a l f a \$ a l f e a

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

BWT Decoding

$C = \text{affs\$eflll1aaata}$

- In the unsorted shifts array, the first column is the original S
- Of course, we do not have the unsorted shifts array at decoding
- But knowing the first letter of each row in the unsorted shifts array is enough for decoding

unsorted shifts array

```
a l f e a t s a l f a l f a $  
l f e a t s a l f a l f a $ a  
f e a t s a l f a l f a $ a l  
e a t s a l f a l f a $ a l f  
a t s a l f a l f a $ a l f e  
t s a l f a l f a $ a l f e a  
s a l f a l f a $ a l f e a t  
a l f a l f a $ a l f e a t s  
l f a l f a $ a l f e a t s a  
f a l f a $ a l f e a t s a l  
a l f a $ a l f e a t s a l f  
l f a $ a l f e a t s a l f a  
f a $ a l f e a t s a l f a l  
a $ a l f e a t s a l f a l f  
$ a l f e a t s a l f a l f a
```


BWT Decoding

$$C = \text{affs}\$eflllaata$$

- Now have the first and last columns of **sorted** shifts array
- Need to figure out the first column of **unsorted** shifts array

unsorted shifts array

1st letter → a l f e a t s a l f a l f a \$

2nd letter → l f e a t s a l f a l f a \$ a

3rd letter → f e a t s a l f a l f a \$ a l

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

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

... ..

- Need to figure out where in **sorted** shifts array are the rows $0, 1, \dots, n - 1$ of the **unsorted** shifts array

sorted shifts array

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

BWT Decoding

$C = \text{affs\$eflll1aaata}$

$S = \text{a}$

- Row 0 of unsorted shifts starts with **a**
- Therefore string S starts with **a**
- Where is row 1 of unsorted shifts array?

unsorted shifts array

alfeatsalfalfa\$

lfeatsalfalfa\$a**a**

featsalfalfa\$a**l**

eatsalfalfa\$alf

.....

- In the unsorted shifts array, any row ends with the first letter of previous row
 - row 1 ends with the first letter of row 0
 - with **a** in our example

sorted shifts array

\$ a

a f

a f

a s

a \$ row 0

a e

e f

f l

f l

f l

l a

l a

l a

s t

t a

BWT Algorithm and Example

- Multiple rows end with **a**, which one is row 1 of unsorted shifts?
 - row 1 is a cyclic shift by one of row 0
- Rows that end with **a** are cyclic shifts by one of rows that start with **a**

sorted shifts array

```
$alfeatsalalfa  
a$alfeatsalalf  
a1fa$alfeatsalf  
a1falfa$alfeats  
a1featsalalfa$  
a1tsalalfa$alf  
eatsalalfa$alf  
fa$alfeatsalfal  
falfa$alfeatsal  
featsalfalfa$al  
lfa$alfeatsalf  
lfalfa$alfeatsa  
lfeatsalfalfa$a  
salalfa$alfeats  
tsalalfa$alf
```

BWT Algorithm and Example

- Multiple rows end with **a**, which one is row 1 of unsorted shifts?
 - row 1 is a cyclic shift by one of row 0
- Rows that end with **a** are cyclic shifts by one of rows that start with **a**
- Rows that start with **a** appear in exactly the same order as rows that end with **a**
 - for both group of patterns, sorting does not depend on **a**, and all other letters are the same between these two groups

a\$alfeatsalfalf	\$alfeatsalfalf	a
a1fa\$alfeatsalf	lfa\$alfeatsalf	a
a1falfa\$alfeats	lfalfalfa\$alfeats	a
a1featsalfalfalfa\$	lfeatsalfalfalfa\$	a
a1tsalfalfalfa\$alfe	tsalfalfalfa\$alfe	a

row 0 of unsorted shifts is #4 among all rows starting with a

its cyclic shift by one, which is row 1 of unsorted shifts is also #4 among all rows ending with a

sorted shifts array

	\$alfeatsalfalf	a 1
1	a\$alfeatsalfalf	
2	a1fa\$alfeatsalf	
3	a1falfa\$alfeats	
4	a1featsalfalfalfa\$	row 0
	a1tsalfalfalfa\$alfe	
	eatsalfalfalfa\$alf	
	fa\$alfeatsalfalf	
	alfalfa\$alfeatsalf	
	featsalfalfalfa\$alf	
	lfa\$alfeatsalf	a 2
	lfalfalfa\$alfeats	a 3
	lfeatsalfalfalfa\$	a 4
	salfalfalfa\$alf	row 1
	tsalfalfalfa\$alfe	a

- But direct 'counting' takes $O(n)$ to find row 1

BWT Algorithm and Example

- Form KVP=(letter, row number) in the last column, and sort KVPs using stable sort
 - bucket sort
 - $O(n + |\Sigma_S|)$

sorted shifts array

.	a	,	0	
.	f	,	1	
.	f	,	2	
.	s	,	3	
.	\$,	4	row 0
.	e	,	5	
.	f	,	6	
.	l	,	7	
.	l	,	8	
.	l	,	9	
.	a	,	1 0	
.	a	,	1 1	
.	a	,	1 2	
.	t	,	1 3	
.	a	,	1 4	

BWT Algorithm and Example

- Form KVP=(letter, row number) in the last column, and sort KVPs using stable sort
 - bucket sort
- Equal letters stay in the same relative order because we used stable sort
- Each letter in the first column 'remembers' which row (before sorting) it came from
- Now the row number can be directly 'read' in constant time!**

the same **KVP**
(a,row in last column) { #4 among all rows starting with a
#4 among all rows starting with a

sorted shifts array

\$,	4	a	,	0	
a	,	0	f	,	1	
a	,	1	0	f	,	2	
a	,	1	1	s	,	3	
a	,	1	2	\$,	4	row 0
a	,	1	4	e	,	5	
e	,	5	f	,	6	
f	,	1	l	,	7	
f	,	2	l	,	8	
f	,	6	l	,	9	
l	,	7	a	,	1	0
l	,	8	a	,	1	1
l	,	9	a	,	1	2 row 1
s	,	3	t	,	1	3
t	,	1	3	a	,	1	4

BWT Decoding Continued

$$C = \text{affs}\$eflllaata$$
$$S = a \mathbf{1}$$

- Multiple rows end with **a**, which one is row 1 of unsorted shifts?

sorted shifts array

\$, 4	a , 0
a , 0	f , 1
a , 1 0	f , 2
a , 1 1	s , 3
a , 1 2	\$, 4
a , 1 4	e , 5
e , 5	f , 6
f , 1	l , 7
f , 2	l , 8
f , 6	l , 9
l , 7	a , 1 0
l , 8	a , 1 1
1 , 9	a , 1 2
s , 3	t , 1 3
t , 1 3	a , 1 4

row 0

row 1

$$S[1] = \mathbf{1} \leftarrow$$

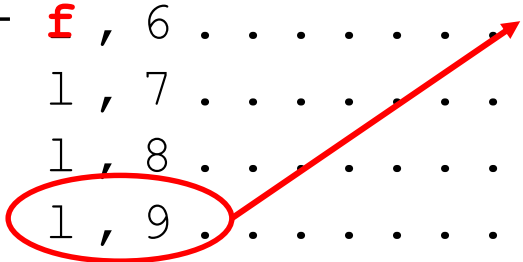
BWT Decoding Continued

$C = \text{affs\$eflll1aaata}$

$S = \text{al}\mathbf{f}$

sorted shifts array

	\$,	4	a	,	0		
	a	,	0	f	,	1		
	a	,	1	0	f	,	2		
	a	,	1	1	s	,	3		
	a	,	1	2	\$,	4	row 0	
	a	,	1	4	e	,	5		
	e	,	5	f	,	6		
	f	,	1	l	,	7		
	f	,	2	l	,	8		
$S[2] = \mathbf{f} \leftarrow$	\mathbf{f}	,	6	l	,	9	row 2	
	l	,	7	a	,	1	0	
	l	,	8	a	,	1	1	
	l	,	9	a	,	1	2	row 1
	s	,	3	t	,	1	3	
	t	,	1	3	a	,	1	4	



BWT Decoding Continued

$C = \text{affs\$eflll1aaata}$

$S = \text{alf}$ **e**

sorted shifts array

	\$, 4 a , 0	
	a , 0 f , 1	
	a , 1 0 f , 2	
	a , 1 1 s , 3	
	a , 1 2 \$, 4	row 0
	a , 1 4 e , 5	
$S[3] = \text{e} \leftarrow$	e , 5 f , 6	row 3
	f , 1 l , 7	
	f , 2 l , 8	
	f , 6 l , 9	row 2
	l , 7 a , 1 0	
	l , 8 a , 1 1	
	l , 9 a , 1 2	row 1
	s , 3 t , 1 3	
	t , 1 3 a , 1 4	

BWT Decoding Continued

$C = \text{affs\$eflll1aaata}$

$S = \text{alfe}$ **a**

sorted shifts array

	\$, 4 a , 0	
	a , 0 f , 1	
	a , 1 0 f , 2	
	a , 1 1 s , 3	
	a , 1 2 \$, 4	row 0
$S[4] = \text{a} \leftarrow$	a , 1 4 e , 5	row 4
	e , 5 f , 6	row 3
	f , 1 l , 7	
	f , 2 l , 8	
	f , 6 l , 9	row 2
	l , 7 a , 1 0	
	l , 8 a , 1 1	
	l , 9 a , 1 2	row 1
	s , 3 t , 1 3	
	t , 1 3 a , 1 4	

BWT Decoding Continued

$C =$ affs\$eflll1aaata

$S =$ alfeatsalfalfa\$

sorted shifts array

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

BWT Decoding Pseudocode

BWT::decoding($C[0 \dots n - 1], S$)

C : string of characters over alphabet Σ_C , S : output stream

$A \leftarrow$ array of size n // leftmost column

for $i = 0$ to $n - 1$

$A[i] \leftarrow (C[i], i)$ // store character and index

stably sort A by character

for $j = 0$ to n // find \$

if $C[j] = \$$ **break**

repeat

$S.append(\text{character stored in } A[j])$

$j \leftarrow$ index stored in $A[j]$

until we have appended \$

BWT Summary

- **Encoding cost**

- $O(n \log n)$ with special sorting algorithm
 - in practice MSD sort is good enough but worst case is $\Theta(n^2)$
- read encoding from the suffix array

- **Decoding cost**

- faster than encoding
- $O(n + |\Sigma_S|)$
- Encoding and decoding both use $O(n)$ space
- They need all of the text (no streaming possible)
 - can use on blocks of text (block compression method)
- BWT tends to be slower than other methods
- But combined with MTF, RLE and Huffman leads to better compression

Compression Summary

Huffman	Run-length encoding	Lempel-Ziv-Welch	Bzip2 (uses Burrows-Wheeler)
variable-length	variable-length	fixed-length	multi-step
single-character	multi-character	multi-character	multi-step
2-pass	1-pass	1-pass	not streamable
60% compression on English text	bad on text	45% compression on English text	70% compression on English text
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, old picture-formats	GIF, some variants of PDF, Unix compress	bzip2 and variants