CS 240 – Data Structures and Data Management

Module 10: Compression

T. Biedl   E. Schost   O. Veksler

Based on lecture notes by many previous cs240 instructors

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

Winter 2021

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 texts to coded texts
- Decoding
  - an algorithm mapping coded texts back to their original source text
- **Notes**
  - source "text" can be any sort of data (not always text)
  - usually the coded alphabet is just binary $\Sigma_C = \{0,1\}$
  - usually S and C are stored as *streams*
    - read/write only one character at a time
    - convenient for handling huge texts
      - can start processing text while it is still being loaded
    - input stream supports methods: $pop(), top(), isEmpty()$
    - output stream supports methods: $append(), isEmpty()$

Judging Encoding Schemes

- Can 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 course)
- Encoding schemes that try to minimize the size of the coded text perform *data compression*
- We will measure the *compression ratio*

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

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

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

  - 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 as binary string	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 → (65, 80, 80, 76, 69) → 01000001 1010000 1010000 1001100 1000101
- Standard in all computers and often our source alphabet
- Not well-suited for non-English text
  - ISO-8859 extends to 8 bits, handles most Western languages
- Other (earlier) fixed-length codes: Baudot code, Murray code
- To decode a fixed-lenth code (say codewords have k bits), we look up each k-bit pattern in a table

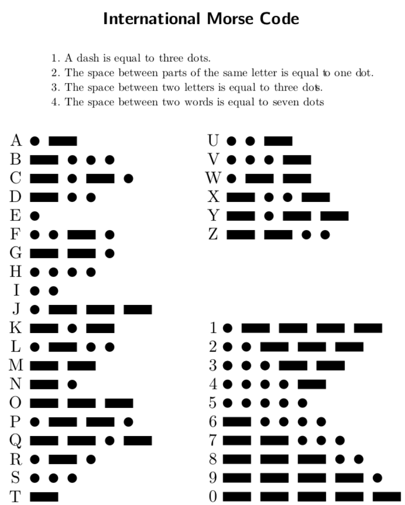
Variable-Length Codes

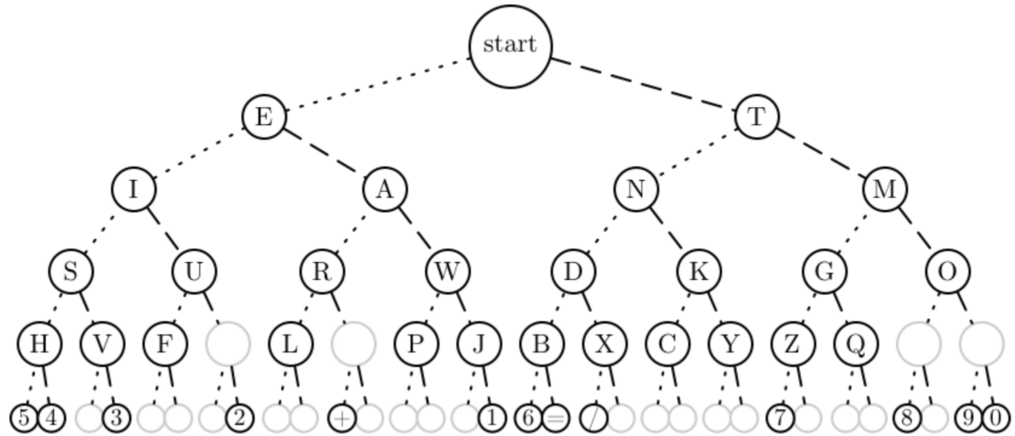
- **Overall goal:** Find an encoding that is short
- **Observation:** Some alphabet letters occur more often than others
  - idea: use shorter codes for more frequent characters
  - example: frequency of letters in typical English text

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





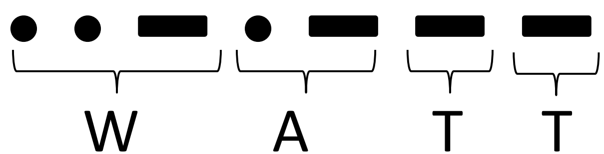
- Example 2: UTF-8 encoding of Unicode
  - there are more than 107,000 Unicode characters
  - uses 1-4 bytes to encode any Unicode character

Encoding

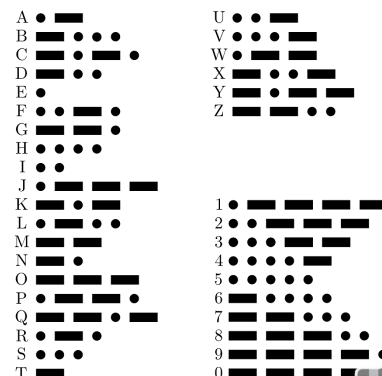
- Assume we have some character encoding $E: \Sigma_S \rightarrow \Sigma_C^*$
- E is a dictionary with keys in Σ_S
- Typically E would be stored as array indexed by Σ_S

```
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 ← E.search(S.pop())
        C.append (x)
```

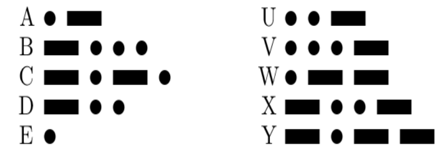
- Example: encode text "WATT" with Morse code



E



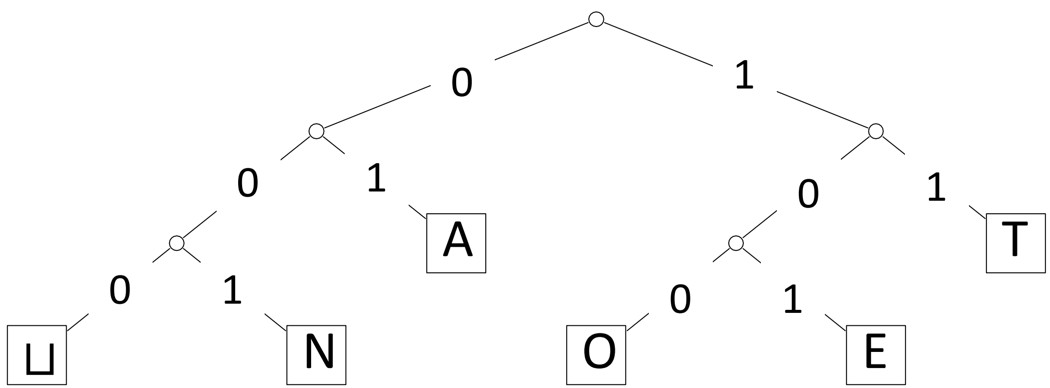
Decoding



- The decoding algorithm must map Σ_C^* to Σ_S
- The code must be *uniquely decodable*
  - false for Morse code as described

decodes to both WATT and EAJ

  - 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



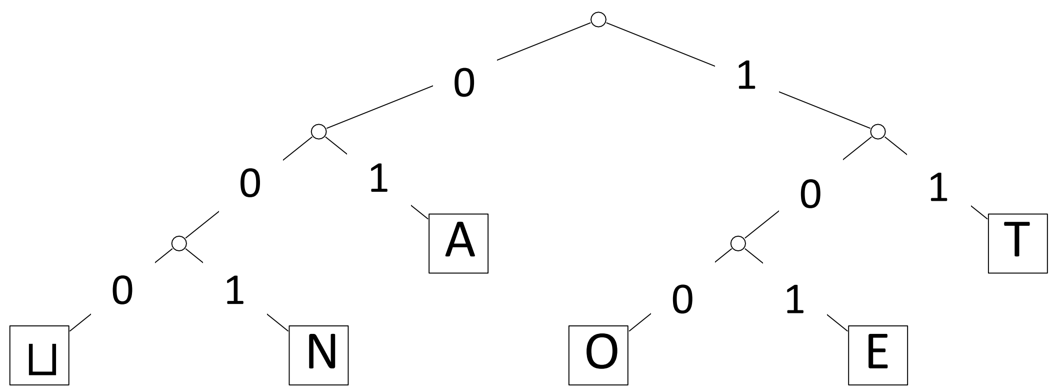
- Do not need symbol $, codewords are prefix-free by definition

Example: Prefix-free Encoding/Decoding

- Code as table

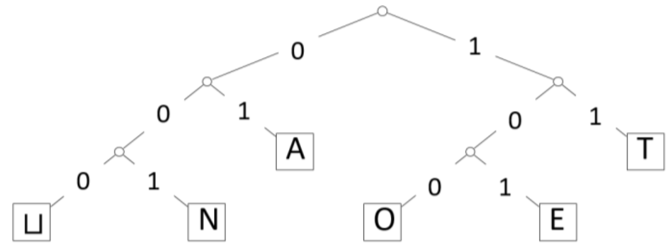
$c \in \Sigma_S$	⊔	A	E	N	O	T
$E(c)$	000	01	101	001	100	11

- Code as trie



- Encode AN⊔ANT → 01 001 000 0100111
- Decode 111000001010111 → TO⊔EAT

Decoding of Prefix-Free Codes



- Any prefix-free code is uniquely decodable

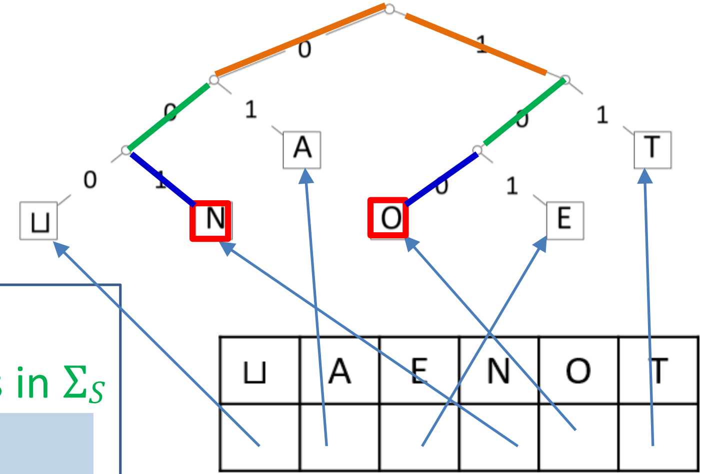
PrefixFree::decoding$(T,\ C,\ S)$
T: trie of a prefix-free code, C: input-stream with characters in Σ_C
S: output-stream

```
while C is non-empty
    r ← T.root
    while r is not a leaf
        if C is empty or has no child labelled C.top()
            return "invalid encoding"
        r ← child of r that is labelled with C.pop()
    S.append(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

$O(|T|)$

$L \leftarrow$ array of nodes in T indexed by Σ_S
for all leaves l in T
L[character at l] $\leftarrow l$

$O(|C|)$

while S is non-empty
$w \leftarrow$ empty string; $v \leftarrow L[S.pop()]$
while v is not the root
w.prepend (character from v to its parent)
$v \leftarrow$ parent(v)
// now w is the encoding of S
$C.append(w)$

$S = ON$
$i = 0$ (letter O)
$w = \Lambda$
$w = 0$
$w = 00$
$w = 100$
$C = 100$
$i = 1$ (letter N)
$w = \Lambda$
$w = 1$
$w = 01$
$w = 001$
$C = 100\ 001$

- Run-time: $O(|T| + |C|)$
  - $O(|\Sigma_S| + |C|)$ if T has no nodes with one child

Outline

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

Huffman's Algorithm: Building the Best Trie

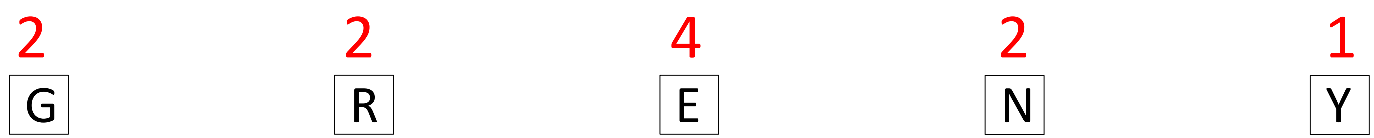
- For a given S the best trie can be constructed with Huffman tree algorithm
  - trie giving the shortest coded text C if alphabet is binary
    - $\Sigma_C = \{0,1\}$
  - tailored to frequencies in that particular S

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 (single node) trie
  - each trie has a frequency
  - initially, frequency is equal to its character frequency

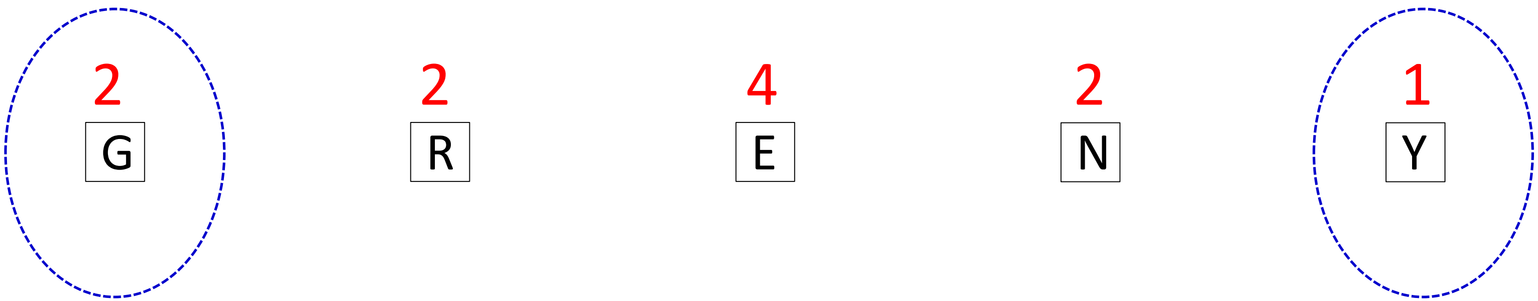


Example: Huffman Tree Construction

- 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

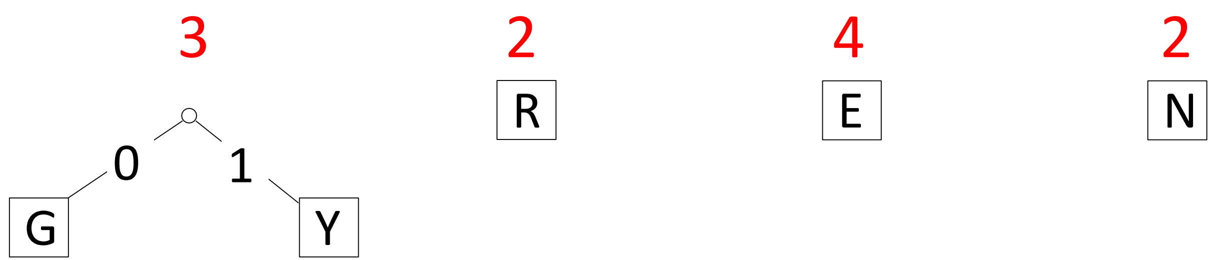


Example: Huffman Tree Construction

- 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

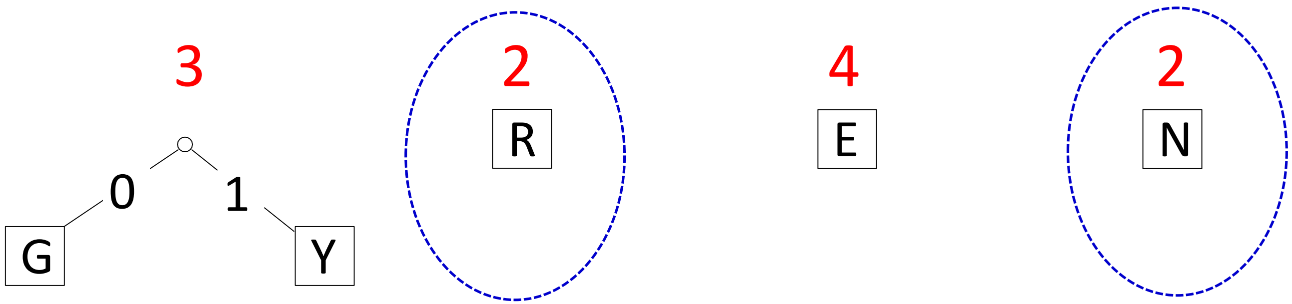


Example: Huffman Tree Construction

- 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

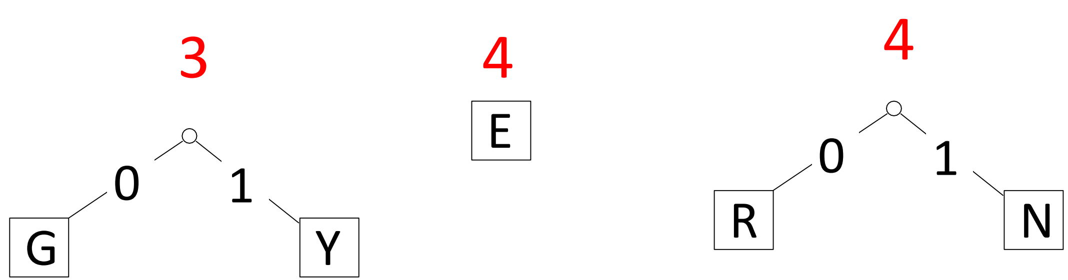


Example: Huffman Tree Construction

- 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

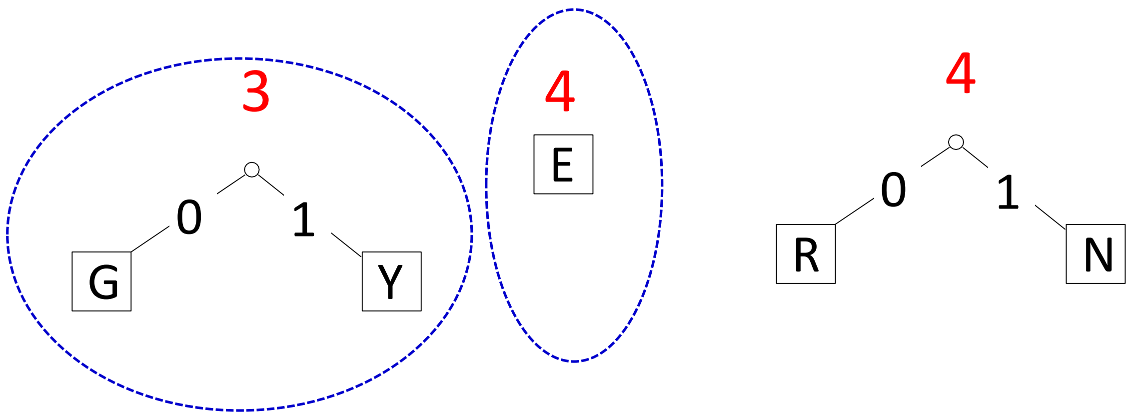


Example: Huffman Tree Construction

- 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

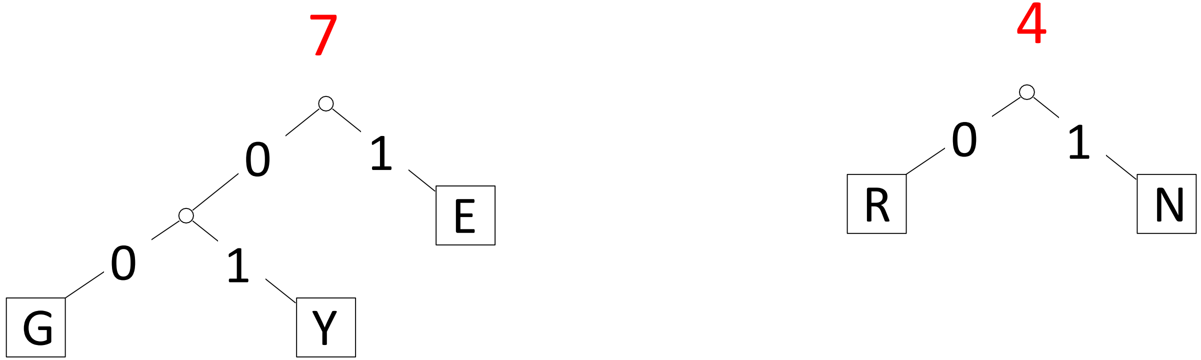


Example: Huffman Tree Construction

- 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

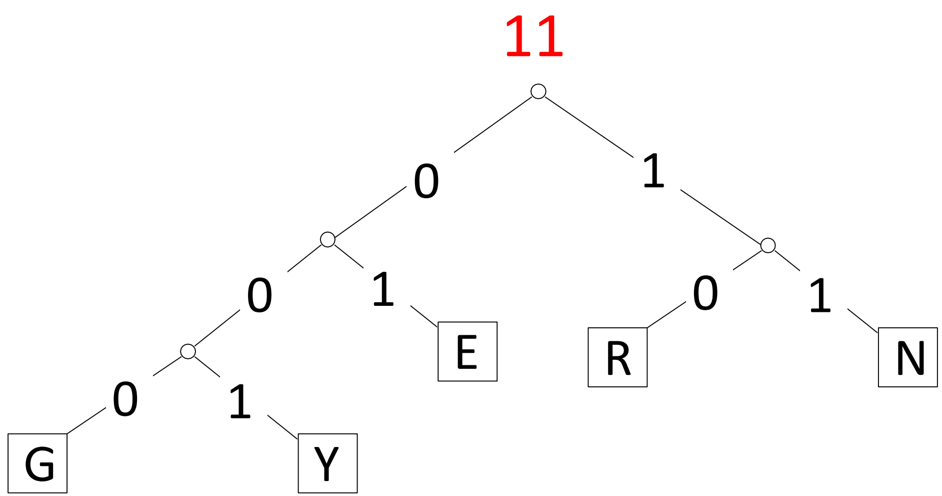


Example: Huffman Tree Construction

- 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

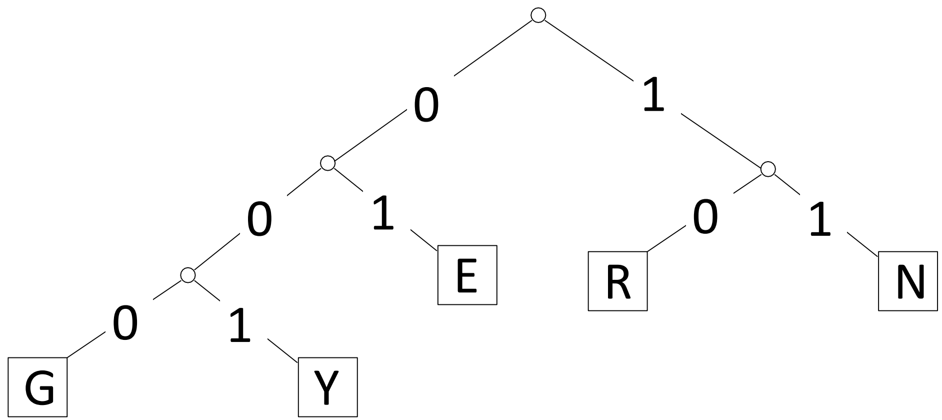


Example: Huffman Tree Construction

- 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

$$\frac{25}{11 \cdot \log 5} \approx 98\%$$

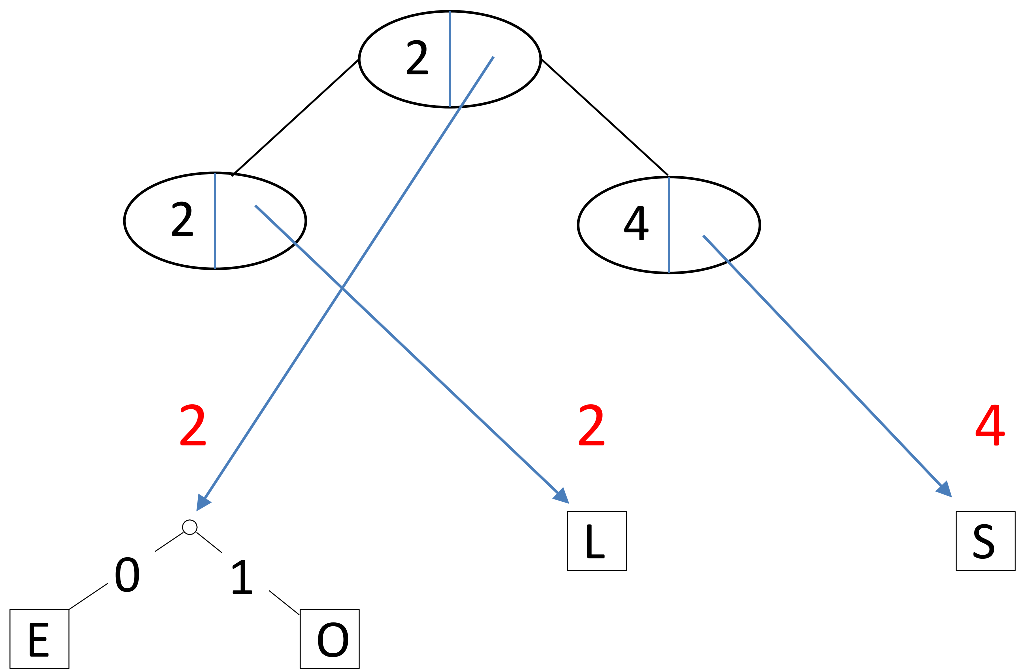
- These frequencies are not skewed enough to lead to good compression

Huffman Algorithm Summary

- 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
- Data structure for making this efficient?
  - min-ordered heap
    - step 4 is two *delete-min*
    - step 5 is *insert*

Heap Storing Tries during Huffman Tree Construction

- Efficient data structure to store tries
  - a min-ordered heap
  - (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 with characters in Σ_S ,  S: output-stream
    f ← array indexed by Σ_S , initialized to 0
    while S is non-empty do increase f[S.pop()] by 1 // get frequencies        O(n)
    Q ← min-oriented priority queue to store tries
    for all c ∈ Σ_S with f[c] > 0                                              O(|Σ_S| log |Σ_S|)
        Q.insert(single-node trie for c with weight f[c])
    while Q.size() > 1
        T_1 ← Q.deleteMin(), f_1 ← weight of T_1
        T_2 ← Q.deleteMin(), f_2 ← weight of T_2                               O(|Σ_S| log |Σ_S|)
        Q.insert(trie with T_1, T_2 as subtries and weight f_1 + f_2)
    T ← Q.deleteMin()  // trie for decoding
    reset input-stream S
    C ← PrefixFreeEncodingFromTrie(T, S) // perform actual encoding            O(|Σ_S| + |C|)
```

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

Huffman Coding Evaluation

- Constructed trie is **not unique** (why?)
- So decoding trie must be transmitted along with the coded text C
- This may make encoding bigger than source text!
- Encoding must pass through strem twice
  - to compute frequencies and to encode
  - 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
  - C is shortest among all prefix-free character encodings with $\Sigma_C = \{0, 1\}$
  - proof omitted
- Many variations
  - give tie-breaking rules, estimate frequencies, adaptively change encoding, …

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 receive one codeword

$$S = \underbrace{\text{b}}_{01}\ \underbrace{\text{a}}_{1}\ \underbrace{\text{n}}_{11}\ \underbrace{\text{a}}_{1}\ \underbrace{\text{n}}_{11}\ \underbrace{\text{a}}_{1}$$

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

$$S = \underbrace{\text{b}}_{01}\ \underbrace{\text{a n a}}_{11}\ \underbrace{\text{n a}}_{101}$$

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
- **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
        - 000000000000010100000000000000110000000000000100

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	$\lfloor \log k \rfloor$	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

- Encoding

$$S = \mathbf{1}1111110010000000000000000000000011111111111$$

$$C = \mathbf{1}$$

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**

- Encoding

$$S = \mathbf{1111111}0010000000000000000000000011111111111$$

$$k = 7$$

$$C = 1\mathbf{00111}$$

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

- Encoding

$$S = \sout{1111111}\mathbf{00}10000000000000000000000011111111111$$

$$k = 2$$

$$C = 100111\ \mathbf{010}$$

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

- Encoding

$S =$ ~~111111100~~**1**0000000000000000000011111111111

$k = 1$

$C =$ 1001110101**1**

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
20	4	10100	**000010100**

- Encoding

$$S = \cancel{1111111001}\mathbf{00000000000000000000}11111111111$$

$$k = 20$$

$$C = 1001110101\ \mathbf{000010100}$$

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

  $S = \sout{111111100100000000000000000000}\mathbf{11111111111}$

  $k = 11$

  $C = 10011101010000010100\ \mathbf{0001011}$

- Compression ratio

  26/41 ≈ 63%

RLE Encoding

```
RLE::encoding(S, C)
S: input-stream of bits, C: output-stream
    b ← S.top()
    C.append(b)
    while S is non-empty do
        k ← 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 ←empty string
        while (k > 1)
            C.append(0) // 0 appended to output C directly
            K.prepend(k mod 2) // K is built from last digit forwards
            k ← ⌊k/2⌋
        K.prepend(1) // the very first digit of K was not computed
        C.append (K)
        b ← 1 − b
```

RLE Example: Decoding

- Recall that (# zeros+1) tells you the length of k in binary representation

- Decoding

$$C = 00001101001001010$$

$$b = 0$$

$$l = 4$$

$$k = 13$$

$$S = 0000000000000$$

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

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

- Decoding

$C = $ ~~00001101~~**00100**1010

$b = \mathbf{1}$

$l = \mathbf{3}$

$k = \mathbf{4}$

$S = 0000000000000\mathbf{1111}$

RLE Example: Decoding

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

- Decoding

$$C = \sout{0000110100100}1010$$

$$b = \mathbf{0}$$

$$l = \mathbf{1}$$

$$k = \mathbf{1}$$

$$S = 000000000000011110$$

RLE Example: Decoding

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

- Decoding

$$C = \sout{00001101001001}010$$

$$b = \mathbf{1}$$

$$l = \mathbf{2}$$

$$k = \mathbf{2}$$

$$S = 0000000000000111101\mathbf{1}$$

RLE Decoding

RLE-Decoding(C)

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

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

while C is non-empty

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

while $C.pop() = 0$

$l{+}{+}$

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

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

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

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

for $(j = 1$ to $k)$

$S.append(b)$

$b \leftarrow 1 - b$ // 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 is for $S = 000 \ldots 000$ 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
- Used in some image formats (e.g. TIFF)

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
- **Ingredient 1** for Lempel-Ziv-Welch compression
  - Encode characters and *frequent substrings*
    - no need to know which substrings are frequent
      - will discover frequent substring as we process text
      - will encode them as we read the text
    - dictionary constructed during encoding/decoding, no need to send it with encoding
      - how?

Adaptive Dictionaries

- ASCII, UTF-8, and RLE use *fixed* dictionaries
    - same dictionary for any text encoded
    - no need to pass dictionary to the dencoder
- In Huffman, the dictionary is not fixed, but it is *static*
    - each text has its own dictionary
    - the dictionary *does not change* for the entire encoding/decoding
    - need to pass dictionary to the decoder
- **Ingredient 2** for LZW: *adaptive dictionary*
    - start with some initial dictionary D_0
        - usually ASCII
    - at iteration $i \geq 0$, D_i is used to determine the ith output
    - after iteration i, update D_i to D_{i+1}
        - a new character combination is inserted
    - encoder and decoder must both know how dictionary changes
        - compute dictionary during encoding/decoding

LZW Overview

- Start with dictionary D_0 for Σ_S
  - usually $\Sigma_S = ASCII$
  - codes from 0 to 127
  - every step adds to dictionary multi-character string, using codenumbers 128, 129, …
- Iteration i of encoding, current dictionary D_i

  S = abbbc**bb**ad

  $D_i = \{$a:65, ab:140, bb:145, bbc:146$\}$

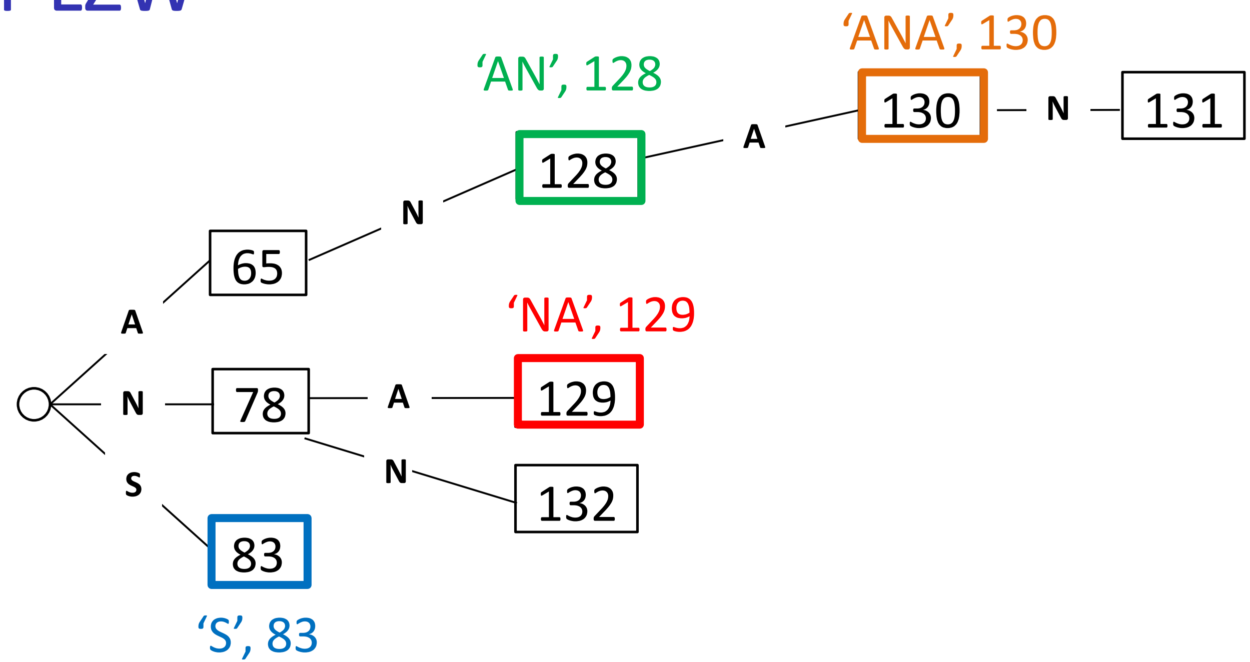
  find longest substring that starts at current pointer and already in dictionary

  encode ‘bb’

  $D_{i+1} = D_i$ .insert(‘bba’, next_available_code)

  (logic: ‘bba’ would have been useful at iteration i, so it may bey useful in the future)
- Store current dictionary D_i in a trie
- Output is a list of numbers (codewords)
  - each number is usually converted to bit-string with fixed-width encoding using 12 bits
    - this limits code numbers to 4096

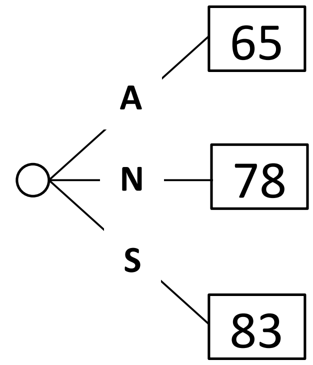
Tries for LZW



- Key-value pairs are (string,code)
- Trie stores KVP at all nodes (external and *internal*) except the root
    - works because a string is inserted only after all its prefixes are inserted
- We show code (value) at each node, because the key can be read off from the edges

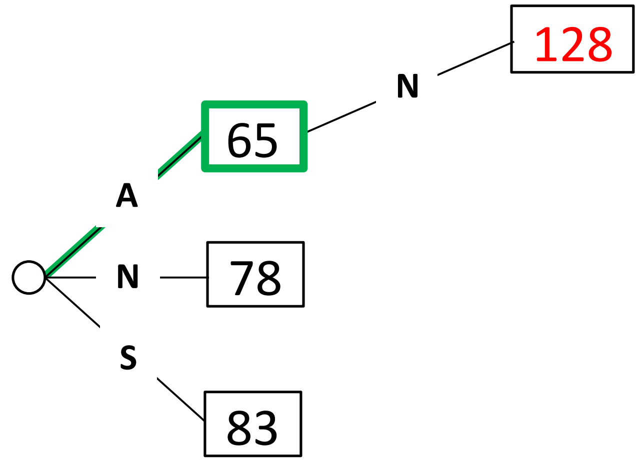
LZW Example

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



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

LZW Example



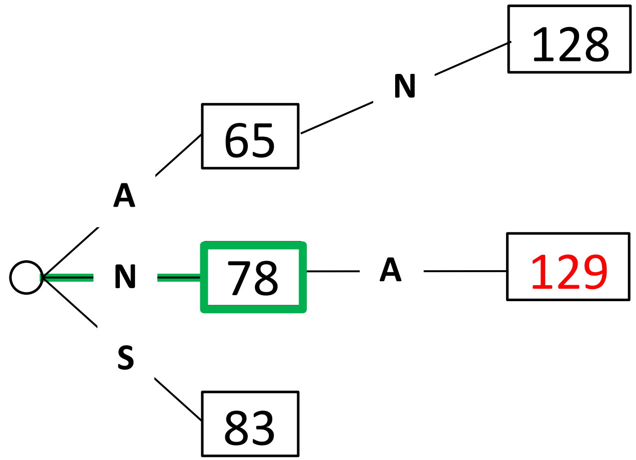
- Dictionary D
  - $idx =$ 129

add to dictionary

- 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

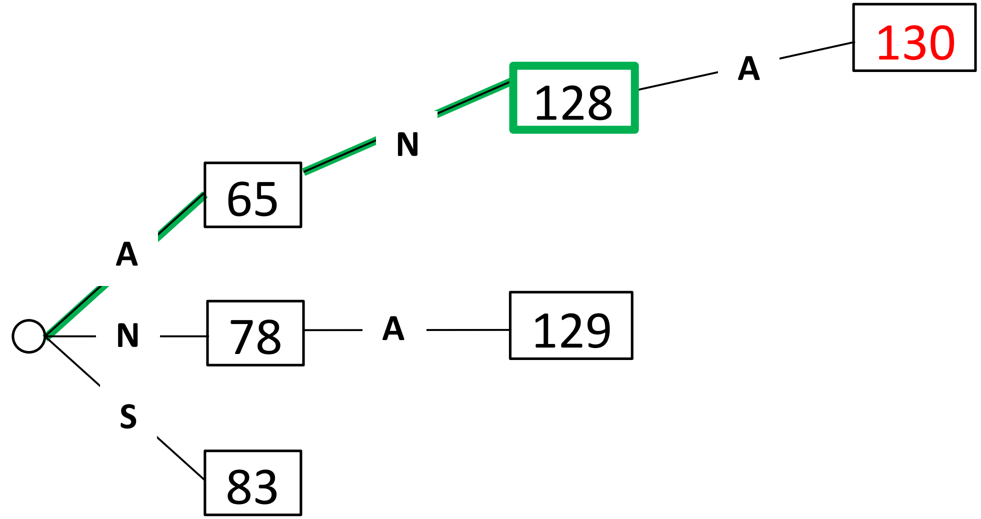


- Dictionary D
  - $idx = 130$

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

add to dictionary

LZW Example



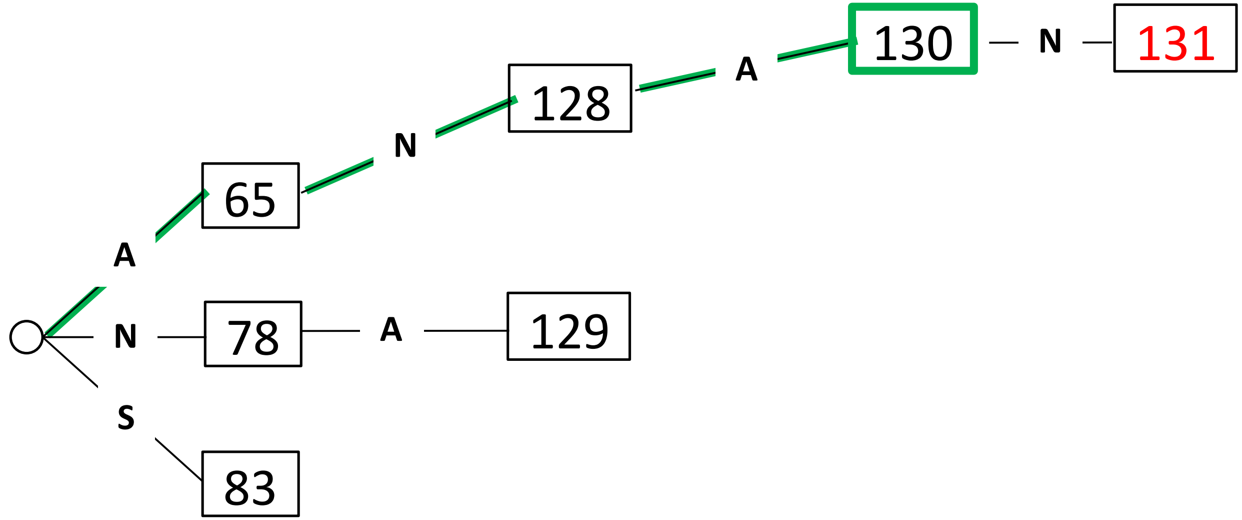
- Dictionary D
  - $idx = 131$

add to dictionary

- Text A | N | A N | A N A N N A
- Encoding 65 | 78 | 128 |

LZW Example

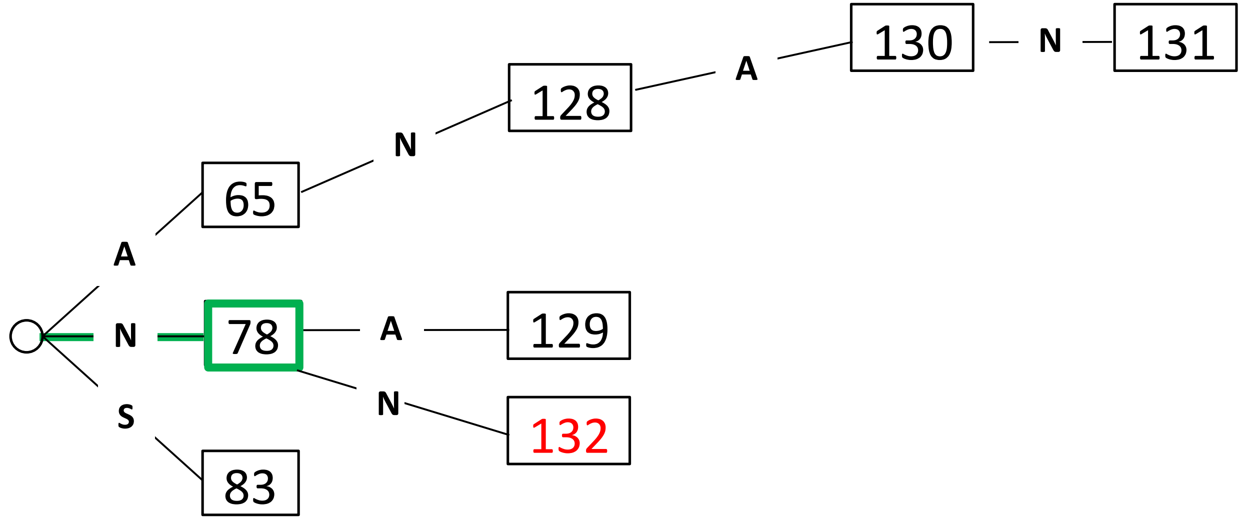
- Dictionary D
  - $idx = 132$



add to dictionary

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

LZW Example

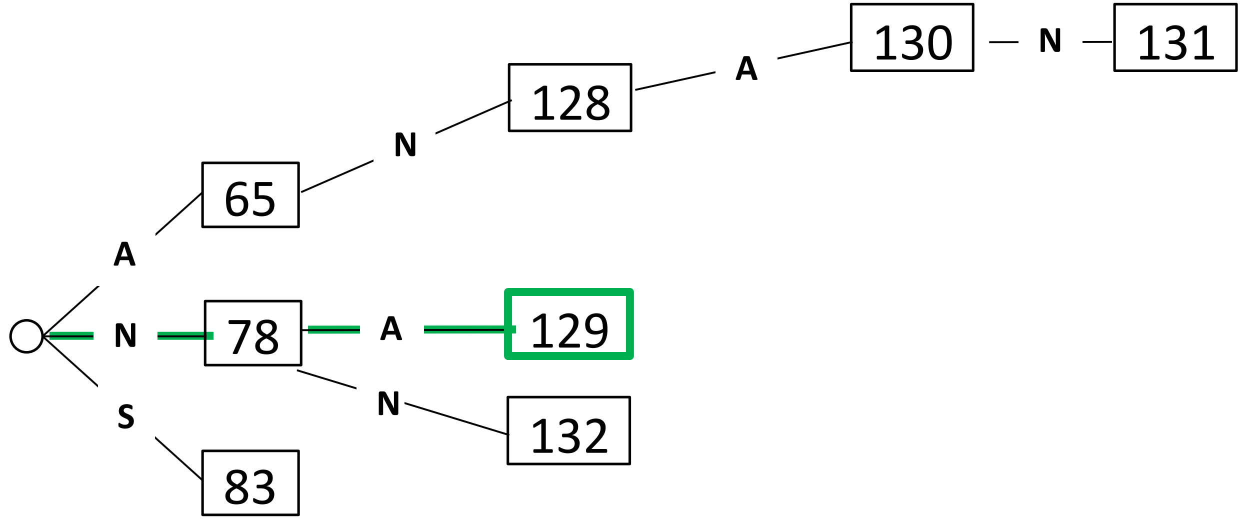


- Dictionary D
  - $idx = 133$

add to dictionary

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

LZW Example

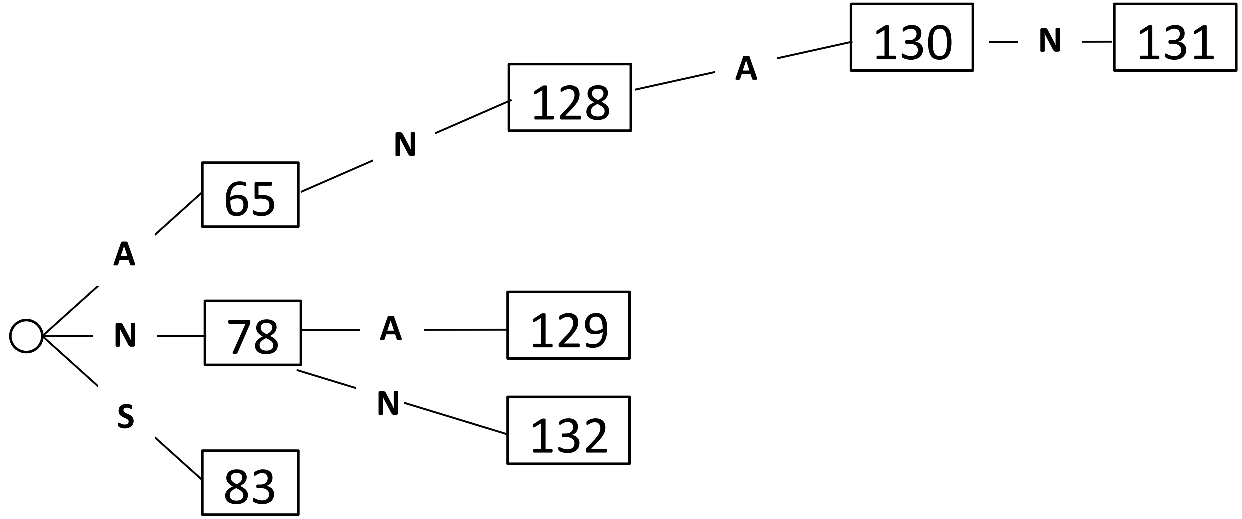


- Dictionary D
  - $idx = 133$

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 000001000001 000001001110 000010000000 00001000010 000001001110 000010000001
- Use fixed length (12 bits) per code
  - 12 bit binary string representation for each code
  - total of 2^{12} = 4096 codes available during encoding

LZW encoding pseudocode

```
LZW-encode(S)
S : input stream of characters, C: output-stream
    initialize dictionary D with ASCII in a trie
    idx ← 128
    while there is input in S do
        v ← root of trie D
        while S is non-empty and v has a child c labelled S.top()
            v ← c
            S.pop()
        C.append(codenumber stored at v)
        if S is non-empty
            create child of v labelled S.top() with code idx
            idx++
```

LZW encoding pseudocode

```
LZW-encode(S)
S : input stream of characters, C: output-stream
        initialize dictionary D with ASCII in a trie
        idx ← 128
        while there is input in S do
            v ← root of trie D
            while S is non-empty and v has a child c labelled S.top()   } trie search
                v ← c
                S.pop()
            C.append(codenumber stored at v)
            if S is non-empty                                              } new dictionary entry
                create child of v labelled S.top() with code idx
                idx ++
```

- 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 we can reconstruct the 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	A									

iter $i = 0$

$D =$

65	A
78	N
83	S

$idx = 128$

- First step: $s = D(65) =$ 'A'
- During encoding, added new string 'AN' to the dictionary at iteration $i = 0$
  - looked ahead at the text and saw 'N'
- During decoding, when read 65, cannot look ahead in the text
  - no new word added at iteration $i = 0$
  - but keep track of s_{prev} = string decoded at previous iteration
    - it is also the string encoder encoded at previous iteration

LZW Decoding Example

- Text: i=0 | A N | 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'
- First step: $s = D(78)$ = 'N'
- Now know that at iteration $i = 0$ of encoding, next character we peaked at was 'N'
- So can add string 'A' + 'N'='AN' to the dictionary

  ('A' ← s_{prev}, 'N' ← $s[0]$)

- s is string decoded at current iteration
- Starting at iteration $i = 1$ of decoding
  - add $s_{prev} + s[0]$ to dictionary

LZW Decoding Example Continued

		i=1									
▪ Text	A	N	A	N	A	N	A	N	N	A
▪ Encoding	65	78	128 (i=2)			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’
- First step: $s = D(128) =$ ‘AN’
- Next step: add to dictionary $s_{prev} + s[0]$

‘N’ + ’A’ = ‘NA’

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'
- First step: $s = D(130) = ???$
  - problem: code 130 is not in D
- Dictionary is exactly one step behind at decoding
- Current decoder iteration is $i = 3$
- Encoder added (s,130) to D at iteration $i = 2$
  - encoder adds $s_{prev} + s[0]$
  - s_{prev}= 'AN'

130	AN + $s[0]$

s

$s[0] = s_{prev}[0] =$'A'

$s =$'ANA'

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
  - AN + A = ANA
- Continue the example
- 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} = '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 Pseudocode

```
LZW::decoding(C, S)
C : input-stream of integers, S: output-stream
    D ← dictionary that maps {0, . . . , 127} to ASCII
    idx ← 128 // next available code
    code ← C.pop()
    s ← D(code)
    S.append(s)
    while there are more codes in C do
        s_prev ← s
        code ← C.pop()
        if code < idx
            s ← D(code) //code in D, look up string s
        if code = idx  // code not in D yet, reconstruct string
            s ← s_prev + s_prev [0]
        else Fail
        S.append(s)
        D.insert(idx , s_prev + s[0])
        idx ++
```

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

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	means 'AN'
129	78, A	means 'NA'
130	128, A	means 'ANA'
131	130, N	means 'ANAA'

Lempel-Ziv Family

- Lempel-Ziv is a family of *adaptive* compression algorithms
    - LZ77 Original version (“sliding window”)
        - Derivatives: LZSS, LZFG, LZRW, LZP, DEFLATE, . . .
            - DEFLATE used in (pk)zip, gzip, PNG
    - LZ78 Second (slightly improved) version
        - Derivatives LZW, LZMW, LZAP, LZY, . . .
        - LZW used in compress, GIF
            - patent issues

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

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

$T_0 =$ a l f e a t s a l f a l f a

Burrows-Wheeler transform

if T_0 has repeated longer substrings, then T_1 has long runs of characters

$T_1 =$ a f f s $ e f l l l a a a t a

Move-to-front transform

if T_1 has long runs of characters, then T_2 has long runs of zeros and skewed frequencies

text T_2 = 1 3 0 5 3 4 3 5 0 0 6 0 0 6

Modified RLE

if T_2 has long runs of zeroes, then T_3 is shorter and skewed frequencies remain

text T_3

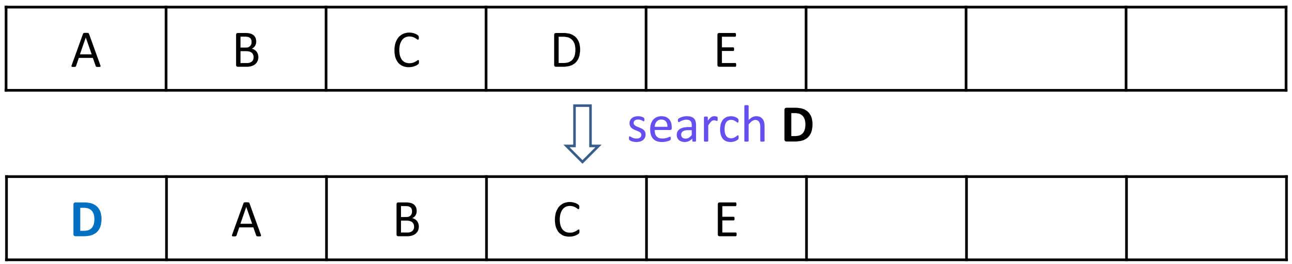
Huffman encoding

compresses well since frequencies are skewed

text T_4

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

MTF Encoding Example

- Source alphabet Σ_S with size $|\Sigma_S| = m$
- Store alphabet in an array

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 us encoding dictionary D
  - single character encoding E
- Code of any character = index of array where character stored in dictionary D
  - $E(\text{'}B\text{'}) = 1$
  - $E(\text{'}H\text{'}) = 7$
- Coded alphabet is $\Sigma_C = \{0,1 \ldots, m - 1\}$
- 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 = **M**ISSISSIPPI

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 = **M**ISSISSIPPI

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 = ~~M~~ISSISSIPPI

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 = ~~MI~~**S**SISSIPPI

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 = ~~MISS~~**S**ISSIPPI

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 = ~~MISS~~**I**SSIPPI

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~~**S**SIPPI

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 = ~~MISSIS~~**S**IPPI

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 = ~~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

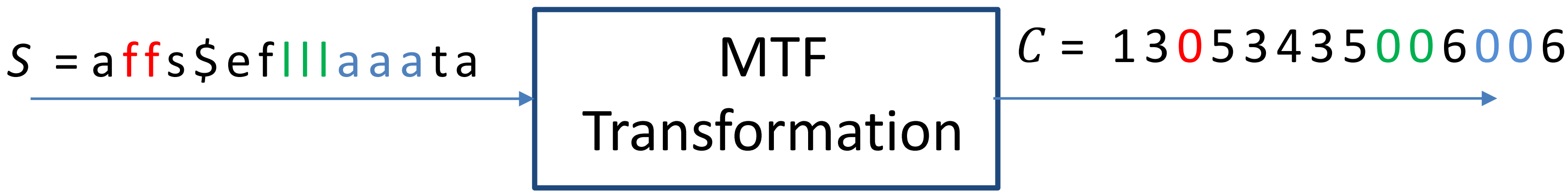
MTF Decoding Example

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**
  $c \leftarrow S.pop()$
  $i \leftarrow$ 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
  $S.append(L[i])$
  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 $\Sigma_C = \{0, 1, \ldots, m-1\}$
  - 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 a coded text with the same letters but in different order
    - source and coded alphabets are the same
  - if original text had frequently occurring substrings, then transformed text should have many runs of the same character
    - *more suitable for MTF transformation*

S = a l f e a t s a l f a l f a → **Burrows-Wheeler Transform** → C = a f f s $ e f l l l a a a t a

- Required: the source text S ends with *end-of-word character* $
  - $ occurs nowhere else in S
- Based on *cyclic* shifts for a string

  - a *cyclic shift* of string X of length n is the concatenation of $X[i+1 \ldots n-1]$ and $X[0 \ldots i]$, for $0 \le i < n$
  - example

	string	**a cyclic shift**
	abcde	cdeab

BWT Algorithm and Example

S = `alfeatsalfalfa$`

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

```
alfeatsalfalfa$
lfeatsalfalfa$a
featsalfalfa$al
eatsalfalfa$alf
atsalfalfa$alfe
tsalfalfa$alfea
salfalfa$alfeat
alfalfa$alfeats
lfalfa$alfeatsa
falfa$alfeatsal
alfa$alfeatsalf
lfa$alfeatsalfa
fa$alfeatsalfal
a$alfeatsalfalf
$alfeatsalfalfa
```

BWT Algorithm and Example

S = alfeatsalfalfa$

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

```
alfeatsalfalfa$
lfeatsalfalfa$a
featsalfalfa$al
eatsalfalfa$alf
atsalfalfa$alfe
tsalfalfa$alfea
salfalfa$alfeat
alfalfa$alfeats
lfalfa$alfeatsa
falfa$alfeatsal
alfa$alfeatsalf
lfa$alfeatsalfa
fa$alfeatsalfal
a$alfeatsalfalf
$alfeatsalfalfa
```

BWT Algorithm and Example

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

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

```
alfeatsalfalfa$
lfeatsalfalfa$a
featsalfalfa$al
eatsalfalfa$alf
atsalfalfa$alfe
tsalfalfa$alfea
salfalfa$alfeat
alfalfa$alfeats
lfalfa$alfeatsa
falfa$alfeatsal
alfa$alfeatsalf
lfa$alfeatsalfa
fa$alfeatsalfal
a$alfeatsalfalf
$alfeatsalfalfa
```

BWT Algorithm and Example

S = alfeatsalfalfa$

- 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
  - sort groups 'lf' lines together, and they all end with '**a**'

sorted shifts array

$alfeatsalfalfa
a$alfeatsalfalf
alfa$alfeatsalf
alfalfa$alfeats
alfeatsalfalfa$
atsalfalfa$alfe
eatsalfalfa$alf
fa$alfeatsalfal
falfa$alfeatsal
featsalfalfa$al
lfa$alfeatsalf**a**
lfalfa$alfeats**a**
lfeatsalfalfa$**a**
salfalfa$alfeat
tsalfalfa$alfea

BWT Algorithm and Example

S = 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
  - sort groups 'lf' lines together, and they all end with 'a'
  - could happen that another pattern will interfere
    - 'hlfd' broken into 'h' and 'lfd'
  - the longer is repeated pattern, the less chance of interference

sorted shifts array

```
$alfeatsalfalfa
a$alfeatsalfalf
alfa$alfeatsalf
alfalfa$alfeats
alfeatsalfalfa$
atsalfalfa$alfe
eatsalfalfa$alf
fa$alfeatsalfal
falfa$alfeatsal
featsalfalfa$al
lfa$alfeatsalfa
lfalfa$alfeatsa
lfd    ………    h
lfeatsalfalfa$a
salfalfa$alfeat
tsalfalfa$alfea
```

BWT Algorithm and Example

S = `alfeatsalfalfa$`

- 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 = **`affs$eflllaaata`**

sorted shifts array

```
$alfeatsalfalfa
a$alfeatsalfalf
alfa$alfeatsalf
alfalfa$alfeats
alfeatsalfalfa$
atsalfalfa$alfe
eatsalfalfa$alf
fa$alfeatsalfal
falfa$alfeatsal
featsalfalfa$al
lfa$alfeatsalfa
lfalfa$alfeatsa
lfeatsalfalfa$a
salfalfa$alfeat
tsalfalfa$alfea
```

BWT Fast Encoding: Efficient Sorting

S =alfeatsalfalfa$

i	cyclic shift
0	alfeatsalfalfa$
1	lfeatsalfalfa$a
2	featsalfalfa$al
3	eatsalfalfa$alf
4	atsalfalfa$alfe
5	tsalfalfa$alfea
6	salfalfa$alfeat
7	alfalfa$alfeats
8	lfalfa$alfeatsa
9	falfa$alfeatsal
10	alfa$alfeatsalf
11	lfa$alfeatsalfa
12	fa$alfeatsalfal
13	a$alfeatsalfalf
14	$alfeatsalfalfa

- Can 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

alfalfa$alfeats

lfa$alfeatsalfa

BWT Fast Encoding: Efficient Sorting

S = alfeatsalfalfa$

i	cyclic shift
0	alfeatsalfalfa$
1	lfeatsalfalfa$a
2	featsalfalfa$al
3	eatsalfalfa$alf
4	atsalfalfa$alfe
5	tsalfalfa$alfea
6	salfalfa$alfeat
7	alfalfa$alfeats
8	lfalfa$alfeatsa
9	falfa$alfeatsal
10	alfa$alfeatsalf
11	lfa$alfeatsalfa
12	fa$alfeatsalfal
13	a$alfeatsalfalf
14	$alfeatsalfalfa

- Can 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

lfa$alfeatsalfa

salfalfa$alfeat

BWT Fast Encoding: Efficient Sorting

S = alfeatsalfalfa$

i	cyclic shift
0	alfeatsalfalfa$
1	lfeatsalfalfa$a
2	featsalfalfa$al
3	eatsalfalfa$alf
4	atsalfalfa$alfe
5	tsalfalfa$alfea
6	salfalfa$alfeat
7	alfalfa$alfeats
8	lfalfa$alfeatsa
9	falfa$alfeatsal
10	alfa$alfeatsalf
11	lfa$alfeatsalfa
12	fa$alfeatsalfal
13	a$alfeatsalfalf
14	$alfeatsalfalfa

- Can 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

lfa$alfeatsalfa

lfalfa$alfeatsa

BWT Fast Encoding: Efficient Sorting

S =alfeatsalfalfa$

i	cyclic shift
0	**alfeatsalfalfa$**
1	**lfeatsalfalfa$**a
2	**featsalfalfa$**al
3	**eatsalfalfa$**alf
4	**atsalfalfa$**alfe
5	**tsalfalfa$**alfea
6	**salfalfa$**alfeat
7	**alfalfa$**alfeats
8	**lfalfa$**alfeatsa
9	**falfa$**alfeatsal
10	**alfa$**alfeatsalf
11	**lfa$**alfeatsalfa
12	**fa$**alfeatsalfal
13	**a$**alfeatsalfalf
14	**$**alfeatsalfalfa

- Can refer to a cyclic shift by the start index in the text, no need to write it out explicitely
- 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 = alfeatsalfalfa$

i	cyclic shift
0	**alfeatsalfalfa$**
1	**lfeatsalfalfa$**a
2	**featsalfalfa$**al
3	**eatsalfalfa$**alf
4	**atsalfalfa$**alfe
5	**tsalfalfa$**alfea
6	**salfalfa$**alfeat
7	**alfalfa$**alfeats
8	**lfalfa$**alfeatsa
9	**falfa$**alfeatsal
10	**alfa$**alfeatsalf
11	**lfa$**alfeatsalfa
12	**fa$**alfeatsalfal
13	**a$**alfeatsalfalf
14	**$**alfeatsalfalfa

j	$A^s[j]$	sorted cyclic shifts
0	14	**$**alfeatsalfalfa
1	13	**a$**alfeatsalfalf
2	10	**alfa$**alfeatsalf
3	7	**alfalfa$**alfeats
4	0	**alfeatsalfalfa$**
5	4	**atsalfalfa$**alfe
6	3	**eatsalfalfa$**alf
7	12	**fa$**alfeatsalfal
8	9	**falfa$**alfeatsal
9	2	**featsalfalfa$**al
10	11	**lfa$**alfeatsalfa
11	8	**lfalfa$**alfeatsa
12	1	**lfeatsalfalfa$**a
13	6	**salfalfa$**alfeat
14	5	**tsalfalfa$**alfea

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 $S[5]$

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

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

j	$A^s[j]$	
0	14	$alfeatsalfalfa
1	13	a$alfeatsalfalf
2	10	alfa$alfeatsalf
3	7	alfalfa$alfeats
4	0	alfeatsalfalfa$
5	4	atsalfalfa$alfe
6	3	eatsalfalfa$alf
7	12	fa$alfeatsalfal
8	9	falfa$alfeatsal
9	2	featsalfalfa$al
10	11	lfa$alfeatsalfa
11	8	lfalfa$alfeatsa
12	1	lfeatsalfalfa$a
13	6	salfalfa$alfeat
14	5	tsalfalfa$alfea

BWT Decoding

$C =$ `affs$eflllaaata`

- Unsorted array of cyclic shifts
  - the first column is the original S

unsorted shifts array

```
alfeatsalfalfa$
lfeatsalfalfa$a
featsalfalfa$al
eatsalfalfa$alf
atsalfalfa$alfe
tsalfalfa$alfea
salfalfa$alfeat
alfalfa$alfeats
lfalfa$alfeatsa
falfa$alfeatsal
alfa$alfeatsalf
lfa$alfeatsalfa
fa$alfeatsalfal
a$alfeatsalfalf
$alfeatsalfalfa
```

BWT Decoding

$C =$ `affs$eflllaaata`

- Given C, last column of sorted shifts array
- Can reconstruct the first column of sorted shifts array by sorting
  - first column has exactly the same characters as the last column
  - *and* they must be sorted

sorted shifts array

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

BWT Decoding

$C =$ `affs$eflllaaata`

- Given C, last column of sorted shifts array
- Can reconstruct the first column of sorted shifts array by sorting
  - first column has exactly the same characters as the last column
  - *and* they must be sorted
- Also need row number for decoding

sorted shifts array

```
. . . . . . . . . . a , 0
. . . . . . . . . . f , 1
. . . . . . . . . . f , 2
. . . . . . . . . . s , 3
. . . . . . . . . . $ , 4
. . . . . . . . . . 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 Decoding

$C =$ `affs$eflllaaata`

sorted shifts array

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

- Now have the first and the last columns of sorted shifts array
  - use stable sort
  - equal letters stay in the same order

BWT Decoding

$C =$ `affs$eflllaaata`

- Now have the first and the last columns of sorted shifts array
- Key for decoding is figuring out where in the sorted shifts array are the unsorted rows 0, 1, ...
- Where is row 0 of *unsorted* shifts array?
  - must end with '$'

sorted shifts array

```
$ , 4 . . . . . . . . a , 0
a , 0 . . . . . . . . f , 1
a , 1 0 . . . . . . . f , 2
a , 1 1 . . . . . . . s , 3
a , 1 2 . . . . . . . $ , 4     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
s , 3 . . . . . . . . t , 1 3
t , 1 3 . . . . . . . a , 1 4
```

BWT Decoding

$C =$ affs$eflllaaata

$S =$ **a**

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

sorted shifts array

$, 4 . . . . a , 0		
a , 0 . . . . f , 1		
a , 1 0 . . . . f , 2		
a , 1 1 . . . . s , 3		
a , 1 2 . . . . $, 4	**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		
s , 3 . . . . t , 1 3		
t , 1 3 . . . . a , 1 4		

BWT Decoding

$C =$ `affs$eflllaaata`

- In the unsorted shifts array, any row ends with the first letter of previous row
    - unsorted row 1 ends with the same letter that unsorted row 0 begins with

unsorted shifts array

```
alfeatsalfalfa$
lfeatsalfalfa$a
featsalfalfa$al
eatsalfalfa$alf
atsalfalfa$alfe
tsalfalfa$alfea
salfalfa$alfeat
alfalfa$alfeats
lfalfa$alfeatsa
falfa$alfeatsal
alfa$alfeatsalf
lfa$alfeatsalfa
fa$alfeatsalfal
a$alfeatsalfalf
$alfeatsalfalfa
```

BWT Decoding

$C =$ `affs$eflllaaata`

$S =$ **a**

- Row = 0 of unsorted shifts starts with **a**
- Therefore
    - string S starts with a
- Where is row = 1 of the unsorted shifts array?
    - row = 1 of unsorted shifts array ends 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	**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		
s , 3 . . . . t , 1 3		
t , 1 3 . . . . a , 1 4		

BWT Decoding

$C =$ affs$eflllaaata

$S =$ **a**

- Row = 0 of unsorted shifts starts with **a**
- Therefore
    - string S starts with a
- Where is row = 1 of the unsorted shifts array?
    - row = 1 of unsorted shifts array ends with a
- Multiple rows end with a, which one is row 1 of unsorted shifts?

sorted shifts array

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

BWT Algorithm and Example

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

- Consider all patterns in sorted array that start with '**a**'

sorted shifts array

```
$alfeatsalfalfa
a$alfeatsalfalf
alfa$alfeatsalf
alfalfa$alfeats
alfeatsalfalfa$
atsalfalfa$alfe
eatsalfalfa$alf
fa$alfeatsalfal
falfa$alfeatsal
featsalfalfa$al
lfa$alfeatsalfa
lfalfa$alfeatsa
lfeatsalfalfa$a
salfalfa$alfeat
tsalfalfa$alfea
```

BWT Algorithm and Example

S = `alfeatsalfalfa$`

- Consider all patterns in sorted array that start with ‘**a**’

```
a$alfeatsalfalf
alfa$alfeatsalf
alfalfa$alfeats
alfeatsalfalfa$
atsalfalfa$alfe
```

- Take their cyclic shifts (by one letter)

```
$alfeatsalfalfa
lfa$alfeatsalfa
lfalfa$alfeatsa
lfeatsalfalfa$a
tsalfalfa$alfea
```

- Find them in sorted array of cyclic shifts

sorted shifts array

```
$alfeatsalfalfa
a$alfeatsalfalf
alfa$alfeatsalf
alfalfa$alfeats
alfeatsalfalfa$
atsalfalfa$alfe
eatsalfalfa$alf
fa$alfeatsalfal
falfa$alfeatsal
featsalfalfa$al
lfa$alfeatsalfa
lfalfa$alfeatsa
lfeatsalfalfa$a
salfalfa$alfeat
tsalfalfa$alfea
```

- They have ‘**a**’ at the end, and are the only rows that have ‘**a**’ at the end
- They appear in the same relative order as before cyclic shift
    - for patterns with same first letter, cyclic shift by one letter does not change relative sorting order

BWT Algorithm and Example

S = alfeatsalfalfa$

- Consider all patterns in sorted array that start with '**a**'

```
a$alfeatsalfalf
alfa$alfeatsalf
alfalfa$alfeats
alfeatsalfalfa$
atsalfalfa$alfe
```

- Take their cyclic shifts (by one letter)

```
$alfeatsalfalfa
lfa$alfeatsalfa
lfalfa$alfeatsa
lfeatsalfalfa$a
tsalfalfa$alfea
```

sorted shifts array

	$alfeatsalfalfa	0	
0	a$alfeatsalfalf		
10	alfa$alfeatsalf		
11	alfalfa$alfeats		
12	**alfeatsalfalfa$**		row 0
14	atsalfalfa$alfe		
	eatsalfalfa$alf		
	fa$alfeatsalfal		
	falfa$alfeatsal		
	featsalfalfa$al		
	lfa$alfeatsalfa	10	
	lfalfa$alfeatsa	11	
	lfeatsalfalfa$a	12	
	salfalfa$alfeat		
	tsalfalfa$alfea	14	

- Unsorted row 1 is a cyclic shift by 1 letter of unsorted row 0
  - unsorted row 0 is #4 among all rows starting with 'a'
  - unsorted row 1 is #4 among all rows ending with 'a'

BWT Algorithm and Example

S = `alfeatsalfalfa$`

- Consider all patterns in sorted array that start with ‘**a**’

  `a$alfeatsalfalf`
  `alfa$alfeatsalf`
  `alfalfa$alfeats`
  `alfeatsalfalfa$`
  `atsalfalfa$alfe`

- Take their cyclic shifts (by one letter)

  `$alfeatsalfalfa`
  `lfa$alfeatsalfa`
  `lfalfa$alfeatsa`
  `lfeatsalfalfa$a`
  `tsalfalfa$alfea`

sorted shifts array

	`$alfeatsalfalfa`	0	
0	`a$alfeatsalfalf`		
10	`alfa$alfeatsalf`		
11	`alfalfa$alfeats`		
12	**`alfeatsalfalfa$`**		row 0
14	`atsalfalfa$alfe`		
	`eatsalfalfa$alf`		
	`fa$alfeatsalfal`		
	`falfa$alfeatsal`		
	`featsalfalfa$al`		
	`lfa$alfeatsalfa`	10	
	`lfalfa$alfeatsa`	11	
	`lfeatsalfalfa$a`	12	
	`salfalfa$alfeat`		
	`tsalfalfa$alfea`	14	

- Unsorted row 1 is a cyclic shift by 1 letter of unsorted row 0
  - unsorted row 0 is #4 among all rows starting with ‘a’
  - unsorted row 1 is #4 among all rows ending with ‘a’

BWT Decoding

$C =$ affs$eflllaaata

$S =$ a l

- Multiple rows end with a, which one is row 1 of unsorted shifts?
- Unsorted row = 1 is located in row 12 of the sorted shifts
  - $S[1] = l$

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

$C =$ `affs$eflllaaata`

$S =$ `al`f

- Unsorted row $= 2$ is located in row 9 of the sorted shifts
  - $S[2] = f$

sorted shifts array

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

BWT Decoding

$C =$ `affs$eflllaaata`

$S =$ `alf`e

- Unsorted row $= 3$ is located in row 6 of the sorted shifts
  - $S[3]$ = 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		
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

$C =$ affs$eflllaaata

$S =$ alfe*a*

- Unsorted row $= 4$ is located in row 5 of the sorted shifts
  - $S[4]$ = *a*

sorted shifts array

$,4	. . . .	a,0	
a,0	. . . .	f,1	
a,10	. . . .	f,2	
a,11	. . . .	s,3	
a,12	. . . .	$,4	**row 0**
a,14	. . . .	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,10	
l,8	. . . .	a,11	
l,9	. . . .	a,12	**row 1**
s,3	. . . .	t,13	
t,13	. . . .	a,14	

BWT Decoding

S = alfeatsalfalfa$

sorted shifts array

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

BWT Decoding

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

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

```
    A ← array of size n     // leftmost column
    for i = 0 to n − 1
        A[i] ← (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(character stored in A[j])
        j ← index stored in A[j]
    until we have appended $
```

BWT Overview

- **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