Module 9: Compression

CS 240 - Data Structures and Data Management

Reza Dorrigiv, Daniel Roche

School of Computer Science, University of Waterloo

Winter 2010

Data Storage and Transmission

The problem: How to store and transmit data?

Source text : The original data, string of characters S from the *source alphabet* Σ_S

Coded text : The encoded data, string of characters C 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

Note: Source "text" can be any sort of data (not always text!)

Usually the coded alphabet Σ_C is just binary: $\{0, 1\}$.

Judging Encoding Schemes

We can always measure efficiency of encoding/decoding algorithms.

What other goals might there be?

Judging Encoding Schemes

We can always measure efficiency of encoding/decoding algorithms.

What other goals might there be?

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

Judging Encoding Schemes

We can always measure efficiency of encoding/decoding algorithms.

What other goals might there be?

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

Encoding schemes that try to minimize $|C|$, the size of the coded text, perform *data compression*. We will measure the *compression ratio*:

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

Types of Data Compression

Logical vs. Physical

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

Lossy vs. Lossless

- **Lossy Compression** achieves better compression ratios, but the decoding is approximate; the exact source text S is not recoverable
- **Lossless Compression** always decodes S exactly

For media files, lossy, logical compression is useful (e.g. JPEG, MPEG)

We will concentrate on physical, lossless compression algorithms.
These techniques can safely be used for any application.

Character Encodings

Standard *character encodings* provide a matching from the source alphabet Σ_S (sometimes called a *charset*) to binary strings.

ASCII (American Standard Code for Information Interchange):

- Developed in 1963
- 7 bits to encode 128 possible characters:
  "control codes", spaces, letters, digits, punctuation
- Not well-suited for non-English text:
  ISO-8859 extends to 8 bits, handles most Western languages

Other (earlier) codes: Morse code, Baudot code

Decoding Dictionary

To decode ASCII, we look up each 7-bit pattern in a table.

More generally, we might have a *decoding dictionary D*
mapping $\Sigma_C^* \to \Sigma_S^*$:

- Must be **prefix-free** (why?)
- Might be used and stored explicitly (e.g. as a trie), or only implicitly
- Might be agreed in advance (*fixed*),
  stored alongside the message (*static*),
  or stored implicitly within the message (*adaptive*)

ASCII is called a *fixed-length code* because
every key string in D has the same length (7 bits)

Variable-Length Codes

Definition: Different key strings in D have different lengths

The UTF-8 encoding of Unicode provides a simple example:

- Encodes any Unicode character (more than 107,000 characters) using 1-4 bytes
- Every ASCII character is encoded in 1 byte with leading bit 0, followed by the 7 bits for ASCII
- Otherwise, the first byte starts with 1-4 1's indicating the total number of bytes, followed by a 0.
  The remaining 13–27 bits indicate which Unicode character.

Run-Length Encoding

RLE is a variable-length code with a fixed decoding dictionary,
but one which is not explicitly stored.
The source alphabet and coded alphabet are both binary: $\{0, 1\}$.

Observation: 0's and 1's in S may be repeated many times in a row
(called a "run").

S is encoded as the first bit of S (either 0 or 1),
followed by a sequence of integers indicating run lengths.
(We don't have to encode the value of each bit since it will alternate.)

Question: How to encode a run length k in binary?

Prefix-free Integer Encoding

The encoding of run-length k must be *prefix-free*,
because the decoder has to know when to stop reading k.

We will encode the binary length of k in **unary**,
followed by the actual value of k in **binary**.

The binary length of k is $len(k) = \lfloor \lg k \rfloor + 1$.
Since $k \geq 1$, we will encode $len(k) - 1$, which is at least 0.

The prefix-free encoding of the positive integer k is in two parts:

1. $\lfloor \lg k \rfloor$ copies of 0, followed by
2. The binary representation of k

Examples: $1 \rightarrow 1$, $3 \rightarrow 011$, $5 \rightarrow 00101$, $23 \rightarrow 000010111$

RLE Example

$S = 1$11111100100000000000000000000011111111111

$C = 1$

RLE Example

$S = $ 11111110010000000000000000000011111111111
$k = 7$
$C = $ 100111

RLE Example

$S = $ 1111111001000000000000000000011111111111
$k = 2$
$C = $ 100111010

RLE Example

$S = 1111111001000000000000000000011111111111$
$k = 1$
$C = 1001110101$

RLE Example

$S = 1111111001\textcolor{red}{0000000000000000000}11111111111$

$k = 20$

$C = 1001110101\textcolor{red}{000010100}$

RLE Example

$S = 1111111001000000000000000000000011111111111$
$k = 11$
$C = 1001110101000010100 0001011$

RLE Example

$S = 11111110010000000000000000000011111111111$

$C = 1001110101000001011$

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

RLE Properties

- Compression ratio could be smaller than 1%
- Usually, we are 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
- Used in some image formats (e.g. TIFF)

Character Frequency

Observation: Some letters in Σ occur more often than others.
So let's use shorter codes for more frequent characters.

For example, the frequency of letters in typical English text is:

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%		

Huffman Coding

- Source alphabet is arbitrary (say Σ), coded alphabet is $\{0, 1\}$
- We build a binary trie to store the decoding dictionary D
- Each character of Σ is a leaf of the trie

Example: $\Sigma = \{\text{AENOT}_\sqcup\}$
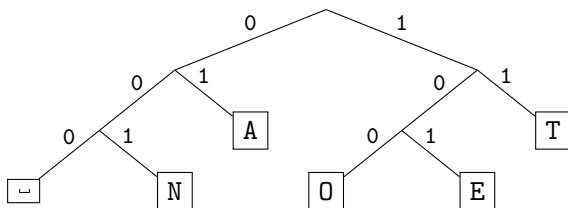
Huffman Encoding/Decoding Example
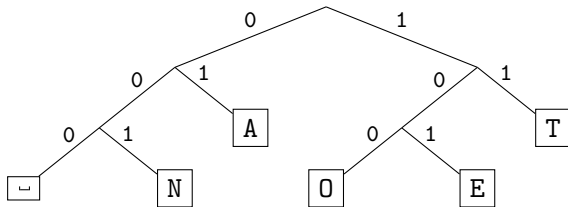
- Encode AN␣ANT
- Decode 111000001010111

Huffman Encoding/Decoding Example
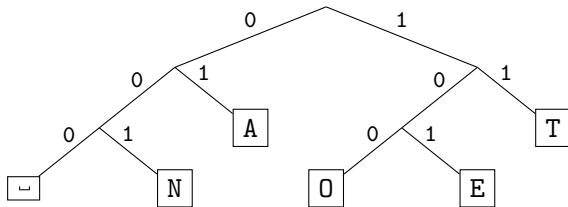
- Encode AN␣ANT → 010010000100111
- Decode 111000001010111 → TO␣EAT

Building the best trie

For a given source text S, how to determine the "best" trie which minimizes the length of C?

1. Determine the frequency of each character $c \in \Sigma$ in S
2. Make $|\Sigma|$ height-0 tries holding each character $c \in \Sigma$.
   Assign a "weight" to each trie: sum of frequencies of all letters in trie
   (initially, these are just the character frequencies)
3. Merge two tries with the least weights, new weight is their sum
   (corresponds to adding one bit to the encoding of each character)
4. Repeat Step 3 until there is only 1 trie left; this is D.

What data structure should we store the tries in to make this efficient?

Building the best trie

For a given source text S, how to determine the "best" trie which minimizes the length of C?

1. Determine the frequency of each character $c \in \Sigma$ in S
2. Make $|\Sigma|$ height-0 tries holding each character $c \in \Sigma$.
   Assign a "weight" to each trie: sum of frequencies of all letters in trie (initially, these are just the character frequencies)
3. Merge two tries with the least weights, new weight is their sum (corresponds to adding one bit to the encoding of each character)
4. Repeat Step 3 until there is only 1 trie left; this is D.

What data structure should we store the tries in to make this efficient?
A min-ordered heap! Step 3 is two *delete-min*s and one *insert*

Building trie example

Example text: LOSSLESS

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

1	2	1	4
E	L	O	S

Building trie example
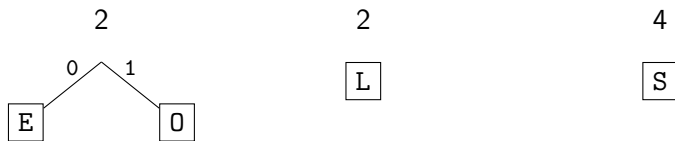
Example text: LOSSLESS

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

Building trie example

Example text: `LOSSLESS`

Character frequencies: `E : 1,    L : 2,    O : 1,    S : 4`
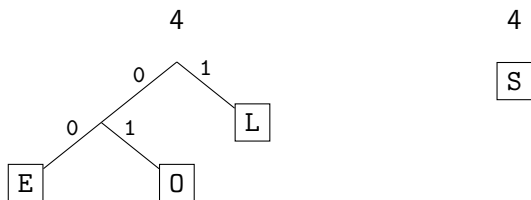
Building trie example

Example text: LOSSLESS

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

LOSSLESS $\rightarrow$

Building trie example

Example text: LOSSLESS

Character frequencies: E : 1,   L : 2,   O : 1,   S : 4
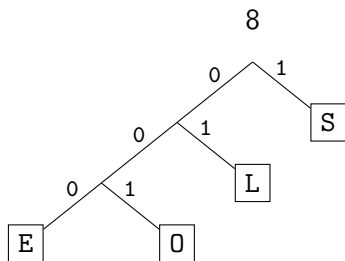
LOSSLESS $\rightarrow$ 01001110100011

Compression ratio: $14/16 \approx 88\%$

Huffman Coding Summary

- Encoder must do lots of work:
  1. Build decoding trie (one pass through S, cost is $O(|S| + |\Sigma| \log |\Sigma|)$)
  2. Construct *encoding* dictionary from the trie mapping $\Sigma \to \{0,1\}^*$
  3. Encode $S \to C$ (second pass through S)
- Note: constructed trie is **not necessarily unique** (why?)
- Decoding trie must be transmitted along with the coded text C
- Decoding is faster; this is an *asymmetric* scheme.
- The constructed trie is an *optimal* one that will give the shortest C (we will not go through the proof)
- Huffman is the best we can do for encoding one character at a time.

Adaptive Dictionaries

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

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

Properties of *adaptive encoding*:

- There is an initial dictionary D_0. Usually this is fixed.
- For $i \geq 0$, D_i is used to determine the i'th output character
- After writing the i'th character to output, both encoder and decoder update D_i to D_{i+1}

Note that both encoder and decoder must have the same information. Usually encoding and decoding algorithms will have the same cost.

Move-to-Front

Recall the MTF heuristic for self-organizing search:

- Dictionary is stored as an unsorted linked list
- After an element is accessed, move it to the front of the list.

How can we use this idea for compression?

Move-to-Front

Recall the MTF heuristic for self-organizing search:

- Dictionary is stored as an unsorted linked list
- After an element is accessed, move it to the front of the list.

How can we use this idea for compression?

Take advantage of *locality* in the data.
If we see a character now, we'll probably see it again soon.

Specifics: MTF is an *adaptive* compression algorithm.
If the source alphabet is Σ_S with size $|\Sigma_S| = m$,
then the coded alphabet will be $\Sigma_C = \{0, 1, \ldots, m-1\}$.

Move-to-Front Encoding/Decoding

MTF-encode(S)
1. $L \leftarrow$ linked list with Σ_S in some pre-agreed, fixed order
2.   **while** S has more characters **do**
3. $c \leftarrow$ next character of S
4.   **output** index i such that $L[i] = c$
5.   Move c to position $L[0]$

Decoding works in *exactly* the same way:

MTF-decode(C)
1. $L \leftarrow$ linked list with Σ_S in some pre-agreed, fixed order
2.   **while** C has more characters **do**
3. $i \leftarrow$ next integer from C
4.   **output** $L[i]$
5.   Move $L[i]$ to position $L[0]$

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

$$S = \texttt{INEFFICIENCIES}$$

$$C =$$

MTF Example

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

$C = $ 8

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

$S =$ INEFFICIENCIES

$C =$ 8 13

MTF Example

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

$$S = \text{INEFFICIENCIES}$$

$$C = \ 8\ 13\ 6$$

MTF Example

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

$S = $ INEFFICIENCIES

$C = $ 8 13 6 7

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

$$S = \text{INEFFICIENCIES}$$

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

MTF Example

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

$$S = \texttt{INEFFICIENCIES}$$

$$C = \text{ 8 13 6 7 0 3}$$

MTF Example

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

$$S = \texttt{INEFFICIENCIES}$$

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

MTF Example

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

$$S = \text{INEFFICIENCIES}$$

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

MTF Example

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

$$S = \texttt{INEFFICIENCIES}$$

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

- What does a run in S encode to in C?
- What does a run in C mean about the source S?

MTF Compression Ratio

So far, MTF does not provide any compression *on its own* (why?)

We need to encode the integer sequence.

Two possible approaches:

MTF Compression Ratio

So far, MTF does not provide any compression *on its own* (why?)

We need to encode the integer sequence.

Two possible approaches:

- Prefix-free integer encoding (like in RLE)
- Huffman coding

Longer Patterns in Input

RLE, Huffman, and MTF mostly 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`", "`<br>`"
- Video: repeated background between frames, shifted sub-image

Lempel-Ziv

Lempel-Ziv is a family of *adaptive* compression algorithms.

Main Idea: Each character in the coded text C either
refers to a single character in Σ_S, or a *substring* of S
that both encoder and decoder have already seen.

Variants:

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

Lempel-Ziv

Lempel-Ziv is a family of *adaptive* compression algorithms.

Main Idea: Each character in the coded text C either refers to a single character in Σ_S, or a *substring* of S that both encoder and decoder have already seen.

Variants:

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 Overview

- Fixed-width encoding using k bits (e.g. $k = 12$).
  Store decoding dictionary with 2^k entries.
- First $|\Sigma_S|$ entries are for single characters,
  remaining entries involve multiple characters
- Upon encoding/decoding a substring x of S, add xc to D,
  where c is the character following x in S

LZW Example

Input: YO!␣YOU!␣YOUR␣YOYO! Σ_S = ASCII character set (0–127)

$C =$

$D =$

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

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

LZW Example

Input: YO!␣YOU!␣YOUR␣YOYO!　　　　Σ_S = ASCII character set (0–127)

$$C = \frac{\text{Y}}{89}$$

$D =$

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

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

LZW Example

Input: YO!␣YOU!␣YOUR␣YOYO! Σ_S = ASCII character set (0–127)

$$C = \begin{array}{cc} \text{Y} & \text{O} \\ 89 & 79 \end{array}$$

$D =$

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

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

LZW Example

Input: YO!␣YOU!␣YOUR␣YOYO! Σ_S = ASCII character set (0–127)

$$C = \begin{array}{ccc} Y & O & ! \\ 89 & 79 & 33 \end{array}$$

$D =$

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

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

LZW Example

Input: YO!␣YOU!␣YOUR␣YOYO! Σ_S = ASCII character set (0–127)

$$
\begin{array}{cccc}
 & Y & O & ! & ␣ \\
C = & 89 & 79 & 33 & 32
\end{array}
$$

$D =$

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

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

LZW Example

Input: YO!␣YOU!␣YOUR␣YOYO!　　　　Σ_S = ASCII character set (0–127)

$$
\begin{array}{cccccc}
& Y & O & ! & ␣ & YO \\
C = & 89 & 79 & 33 & 32 & 128
\end{array}
$$

$D =$

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

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

LZW Example

Input: YO!␣YOU!␣YOUR␣YOYO! Σ_S = ASCII character set (0–127)

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

$D =$

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

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

Burrows-Wheeler Transform

The Burrows-Wheeler Transform is a sophisticated compression technique

- Transforms source text to a coded text with the same letters, just in a different order
- *The coded text will be more easily compressible with MTF*
- Compression algorithm does not make just a few "passes" over S. BWT is a *block* compression method.
- (As we will see) decoding is more efficient than encoding, so BWT is an *asymmetric* scheme.

BWT (follwed by MTF, RLE, and Huffman)
is the algorithm used by the `bzip2` program.

It achieves the best compression of any algorithm we have seen
(at least on English text).

BWT Encoding

A *cyclic shift* of a string X of length n is the concatentation of $X[i+1..n-1]$ and $X[0..i]$, for $0 \leq i < n$.

For Burrows-Wheeler, we assume the source text S ends with a special *end-of-word character* $\$$ that occurs nowhere else in S.

The Burrows-Wheeler Transform proceeds in three steps:

1. Place all *cyclic shifts* of S in a list L
2. Sort the strings in L lexicographically
3. C is the list of trailing characters of each string in L

BWT Example

$S = $ `alf␣eats␣alfalfa$`

1 Write all cyclic shifts

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

BWT Example

$S = $ `alf⎵eats⎵alfalfa$`

1. Write all cyclic shifts
2. Sort cyclic shifts

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

BWT Example

$S = $ `alf␣eats␣alfalfa$`

1. Write all cyclic shifts
2. Sort cyclic shifts
3. Extract last characters from sorted shifts

$C = $ `asff$f␣e␣lllaaata`

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

BWT Decoding

Idea: Given C, We can generate the *first column* of the array by sorting. This tells us which character comes after each character in S.

Decoding Algorithm:

View the coded text C as an array of characters.

1. Make array of A of tuples $(C[i], i)$
2. Sort A by the characters, record integers in array N
   (Note: $C[N[i]]$ follows $C[i]$ in S, for all $0 \le i < n$)
3. Set j to index of \$ in C and S to empty string
4. Set $j \leftarrow N[j]$ and append $C[j]$ to S
5. Repeat Step 4 until $C[j] = \$$

BWT Decoding Example

$C = $ `ard$rcaaaabb`

$S = $

$$
\begin{array}{cc}
 & A \\
\left[
\begin{array}{ll}
\texttt{a,} & 0 \\
\texttt{r,} & 1 \\
\texttt{d,} & 2 \\
\texttt{\$,} & 3 \\
\texttt{r,} & 4 \\
\texttt{c,} & 5 \\
\texttt{a,} & 6 \\
\texttt{a,} & 7 \\
\texttt{a,} & 8 \\
\texttt{a,} & 9 \\
\texttt{b,} & 10 \\
\texttt{b,} & 11
\end{array}
\right]
\end{array}
$$

BWT Decoding Example

$C = $ `ard$rcaaaabb`

$S =$

$$
\begin{array}{cc}
A & sort(A) \\
\begin{bmatrix}
\text{a,} & 0 \\
\text{r,} & 1 \\
\text{d,} & 2 \\
\text{\$,} & 3 \\
\text{r,} & 4 \\
\text{c,} & 5 \\
\text{a,} & 6 \\
\text{a,} & 7 \\
\text{a,} & 8 \\
\text{a,} & 9 \\
\text{b,} & 10 \\
\text{b,} & 11
\end{bmatrix}
&
\begin{bmatrix}
\text{\$,} & 3 \\
\text{a,} & 0 \\
\text{a,} & 6 \\
\text{a,} & 7 \\
\text{a,} & 8 \\
\text{a,} & 9 \\
\text{b,} & 10 \\
\text{b,} & 11 \\
\text{c,} & 5 \\
\text{d,} & 2 \\
\text{r,} & 1 \\
\text{r,} & 4
\end{bmatrix}
\end{array}
$$

with $\longrightarrow$ between the two matrices.

BWT Decoding Example

$C = \texttt{ard\$rcaaaabb}$

$S =$

$$
\begin{array}{ccc}
A & sort(A) & N \\
\begin{bmatrix}
\texttt{a,} & 0 \\
\texttt{r,} & 1 \\
\texttt{d,} & 2 \\
\texttt{\$,} & 3 \\
\texttt{r,} & 4 \\
\texttt{c,} & 5 \\
\texttt{a,} & 6 \\
\texttt{a,} & 7 \\
\texttt{a,} & 8 \\
\texttt{a,} & 9 \\
\texttt{b,} & 10 \\
\texttt{b,} & 11
\end{bmatrix}
\longrightarrow
\begin{bmatrix}
\texttt{\$,} & 3 \\
\texttt{a,} & 0 \\
\texttt{a,} & 6 \\
\texttt{a,} & 7 \\
\texttt{a,} & 8 \\
\texttt{a,} & 9 \\
\texttt{b,} & 10 \\
\texttt{b,} & 11 \\
\texttt{c,} & 5 \\
\texttt{d,} & 2 \\
\texttt{r,} & 1 \\
\texttt{r,} & 4
\end{bmatrix}
\longrightarrow
\begin{bmatrix}
3 \\
0 \\
6 \\
7 \\
8 \\
9 \\
10 \\
11 \\
5 \\
2 \\
1 \\
4
\end{bmatrix}
\end{array}
$$

BWT Decoding Example

$C = \text{ard}\textcolor{red}{\$}\text{rcaaaabb}$

$S =$

$$
\begin{array}{cccc}
A & sort(A) & N & \\
\begin{bmatrix}
\text{a,} & 0 \\
\text{r,} & 1 \\
\text{d,} & 2 \\
\text{\$,} & 3 \\
\text{r,} & 4 \\
\text{c,} & 5 \\
\text{a,} & 6 \\
\text{a,} & 7 \\
\text{a,} & 8 \\
\text{a,} & 9 \\
\text{b,} & 10 \\
\text{b,} & 11
\end{bmatrix}
\longrightarrow
\begin{bmatrix}
\text{\$,} & 3 \\
\text{a,} & 0 \\
\text{a,} & 6 \\
\text{a,} & 7 \\
\text{a,} & 8 \\
\text{a,} & 9 \\
\text{b,} & 10 \\
\text{b,} & 11 \\
\text{c,} & 5 \\
\text{d,} & 2 \\
\text{r,} & 1 \\
\text{r,} & 4
\end{bmatrix}
\longrightarrow
\begin{bmatrix}
3 \\
0 \\
6 \\
7 \\
8 \\
9 \\
10 \\
11 \\
5 \\
2 \\
1 \\
4
\end{bmatrix}
& j = 3
\end{array}
$$

BWT Decoding Example

$C = $ `ard$rcaaaabb`

$S = $ `a`

$$
\begin{array}{ccc}
A & sort(A) & N \\
\begin{bmatrix}
\text{a,} & 0 \\
\text{r,} & 1 \\
\text{d,} & 2 \\
\text{\$,} & 3 \\
\text{r,} & 4 \\
\text{c,} & 5 \\
\text{a,} & 6 \\
\text{a,} & 7 \\
\text{a,} & 8 \\
\text{a,} & 9 \\
\text{b,} & 10 \\
\text{b,} & 11
\end{bmatrix}
&
\longrightarrow
\begin{bmatrix}
\text{\$,} & 3 \\
\text{a,} & 0 \\
\text{a,} & 6 \\
\text{a,} & 7 \\
\text{a,} & 8 \\
\text{a,} & 9 \\
\text{b,} & 10 \\
\text{b,} & 11 \\
\text{c,} & 5 \\
\text{d,} & 2 \\
\text{r,} & 1 \\
\text{r,} & 4
\end{bmatrix}
\longrightarrow
&
\begin{bmatrix}
3 \\
0 \\
6 \\
7 \\
8 \\
9 \\
10 \\
11 \\
5 \\
2 \\
1 \\
4
\end{bmatrix}
\end{array}
\qquad j = 7
$$

BWT Decoding Example

$C = \texttt{ard\$rcaaaab}\textcolor{red}{\texttt{b}}$

$S = \texttt{a}\textcolor{red}{\texttt{b}}$

$$
\begin{array}{ccc}
A & sort(A) & N \\
\begin{bmatrix}
\texttt{a,} & 0 \\
\texttt{r,} & 1 \\
\texttt{d,} & 2 \\
\texttt{\$,} & 3 \\
\texttt{r,} & 4 \\
\texttt{c,} & 5 \\
\texttt{a,} & 6 \\
\texttt{a,} & 7 \\
\texttt{a,} & 8 \\
\texttt{a,} & 9 \\
\texttt{b,} & 10 \\
\texttt{b,} & 11
\end{bmatrix}
&
\begin{bmatrix}
\texttt{\$,} & 3 \\
\texttt{a,} & 0 \\
\texttt{a,} & 6 \\
\texttt{a,} & 7 \\
\texttt{a,} & 8 \\
\texttt{a,} & 9 \\
\texttt{b,} & 10 \\
\texttt{b,} & 11 \\
\texttt{c,} & 5 \\
\texttt{d,} & 2 \\
\texttt{r,} & 1 \\
\texttt{r,} & 4
\end{bmatrix}
&
\begin{bmatrix}
3 \\
0 \\
6 \\
7 \\
8 \\
9 \\
10 \\
11 \\
5 \\
2 \\
1 \\
4
\end{bmatrix}
\end{array}
\qquad j = 11
$$

$\longrightarrow$ (between A and $sort(A)$)

$\longrightarrow$ (between $sort(A)$ and N)

BWT Decoding Example

$C = \texttt{ard\$rcaaaabb}$
$S = \texttt{abr}$

$$
\begin{array}{ccc}
A & sort(A) & N \\[4pt]
\begin{bmatrix}
\texttt{a,} & 0 \\
\texttt{r,} & 1 \\
\texttt{d,} & 2 \\
\texttt{\$,} & 3 \\
\texttt{r,} & 4 \\
\texttt{c,} & 5 \\
\texttt{a,} & 6 \\
\texttt{a,} & 7 \\
\texttt{a,} & 8 \\
\texttt{a,} & 9 \\
\texttt{b,} & 10 \\
\texttt{b,} & 11
\end{bmatrix}
\longrightarrow
\begin{bmatrix}
\texttt{\$,} & 3 \\
\texttt{a,} & 0 \\
\texttt{a,} & 6 \\
\texttt{a,} & 7 \\
\texttt{a,} & 8 \\
\texttt{a,} & 9 \\
\texttt{b,} & 10 \\
\texttt{b,} & 11 \\
\texttt{c,} & 5 \\
\texttt{d,} & 2 \\
\texttt{r,} & 1 \\
\texttt{r,} & 4
\end{bmatrix}
\longrightarrow
\begin{bmatrix}
3 \\
0 \\
6 \\
7 \\
8 \\
9 \\
10 \\
11 \\
5 \\
2 \\
1 \\
4
\end{bmatrix}
& \quad j = 4
\end{array}
$$

BWT Decoding Example

$C = \texttt{ard\$rcaaaabb}$

$S = \texttt{abracadabra\$}$

$$
\begin{array}{ccc}
A & \textit{sort}(A) & N \\[4pt]
\begin{bmatrix}
\texttt{a,} & 0 \\
\texttt{r,} & 1 \\
\texttt{d,} & 2 \\
\texttt{\$,} & 3 \\
\texttt{r,} & 4 \\
\texttt{c,} & 5 \\
\texttt{a,} & 6 \\
\texttt{a,} & 7 \\
\texttt{a,} & 8 \\
\texttt{a,} & 9 \\
\texttt{b,} & 10 \\
\texttt{b,} & 11
\end{bmatrix}
\longrightarrow
\begin{bmatrix}
\texttt{\$,} & 3 \\
\texttt{a,} & 0 \\
\texttt{a,} & 6 \\
\texttt{a,} & 7 \\
\texttt{a,} & 8 \\
\texttt{a,} & 9 \\
\texttt{b,} & 10 \\
\texttt{b,} & 11 \\
\texttt{c,} & 5 \\
\texttt{d,} & 2 \\
\texttt{r,} & 1 \\
\texttt{r,} & 4
\end{bmatrix}
\longrightarrow
\begin{bmatrix}
3 \\
0 \\
6 \\
7 \\
8 \\
9 \\
10 \\
11 \\
5 \\
2 \\
1 \\
4
\end{bmatrix}
\end{array}
$$
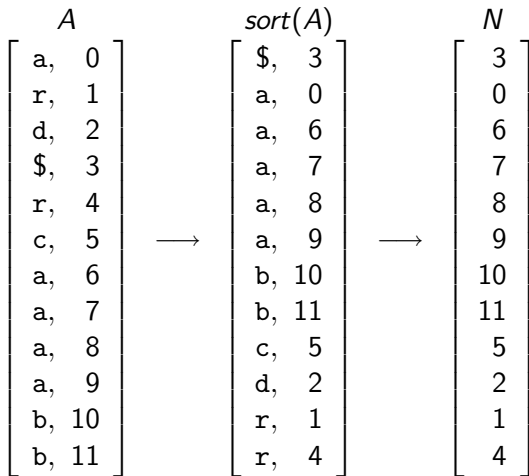
BWT Overview

Encoding cost: $O(n^2)$ (using radix sort)

- Sorting cyclic shifts is equivalent to sorting suffixes
- This can be done by traversing suffix trie
- Possible in $O(n)$ time

Decoding cost: $O(n)$ (faster than encoding)

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

Tends to be slower than other methods but give better compression.

Summary of Compression Algorithms

RLE Variable-width, multiple-character encoding

Huffman Variable-width, single-character (optimal in this case)

MTF Adaptive, transforms to smaller integers
Must be followed by variable-width integer encoding

LZW Adaptive, fixed-width, multiple-character encoding
Augments dictionary with repeated substrings

BWT Block compression method, must be followed by MTF