CS 240 – Data Structures and Data Management

Module 10: Compression

T. Biedl É. Schost O. Veksler

Based on lecture notes by many previous cs240 instructors

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

Winter 2021

version 2021–03–30 19:22

Outline

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

Outline

1. 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

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

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

Usually S and C are stored as streams (read/write only one character at a time), which is convenient for handling huge texts.

TB changed recently: Discuss more what streams are, and use them in code wherever feasible. This affects nearly all pseudo-codes in this module.

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 *(main objective here)*

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 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

A **character encoding** (or more precisely **character-by-character encoding**) maps each character in the source alphabet to a string in coded alphabet.

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

For $c \in \Sigma_S$, we call $E(c)$ the **codeword** of c

Two possibilities:

- **Fixed-length code**: All codewords have the same length.
- **Variable-length code**: Codewords may have different lengths.

Fixed-length codes

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

char	null	start of heading	start of text	end of text	...	0	1	...	A	B	...	$\sim$	delete
code	0	1	2	3	...	48	49	...	65	66	...	126	127

- 7 bits to encode 128 possible characters:
  "control codes", spaces, letters, digits, punctuation

  $A{\cdot}P{\cdot}P{\cdot}L{\cdot}E \to (65, 80, 80, 76, 69) \to$ 1000001 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) examples: Caesar shift, Baudot code, Murray code

To decode a fixed-length code (say codewords have k bits), we look up each k-bit pattern in a table.

Variable-Length Codes

Overall goal: Find an encoding that is short.

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

TB changed recently: Moved this table to here (it was much later)

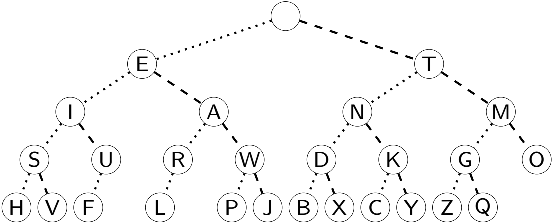
For example, the frequency of letters in typical English text is:

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%		

Variable-Length Codes

Example 1: Morse code.

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



Example 2: UTF-8 encoding of Unicode:

- Encodes any Unicode character (more than 107,000 characters) using 1-4 bytes

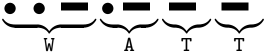
Encoding

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

- Note that E is a dictionary with keys in Σ_S.
- Typically E would be stored as array indexed by Σ_S.

```
charByChar::encoding($E$, $S$, $C$)
$E$ : the encoding dictionary
$S$: input-stream with characters in $\Sigma_S$, $C$: output-stream
    while $S$ is non-empty
        $x \leftarrow E$.search($S$.pop())
        $C$.append($x$)
```

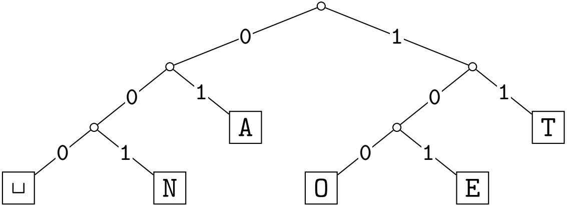
Example: encode text "WATT" with Morse code:



Decoding

The **decoding algorithm** must map Σ_C^* to Σ_S^*.

- The code must be *uniquely decodable*.
  - This is false for Morse code as described!
    • ▬ ▬ • ▬ ▬ ▬ decodes to WATT and ANO and WJ.
    (Morse code uses 'end of character' pause to avoid ambiguity.)
- From now on only consider **prefix-free** codes E:
  no codeword is a prefix of another
- This corresponds to a *trie* with characters of Σ_S only at the leaves.



- The codewords need no end-of-string symbol \$ if E is prefix-free.

Decoding of Prefix-Free Codes

Any prefix-free code is uniquely decodable (why?)

```
prefixFree::decoding($T$, $C$, $S$)
$T$ : trie of a prefix-free code
$C$: input-stream with characters in $\Sigma_C$, $S$: output-stream
    while $C$ is non-empty
        $r \leftarrow T.root$
        while $r$ is not a leaf
            if $C$ is empty or $r$ has no child labelled $C.top()$
                return "invalid encoding"
            $r \leftarrow$ child of $r$ that is labelled with $C.pop()$
        $S.append$(character stored at $r$)
```

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

Encoding from the Trie

We can also encode directly from the trie.

```
prefixFree::encoding($T$, $S$, $C$)
$T$ : trie of a prefix-free code
$S$: input-stream with characters in $\Sigma_S$, $C$: output-stream
    $E \leftarrow$ array of nodes in $T$ indexed by $\Sigma_S$
    for all leaves $\ell$ in $T$
        $E$[character at $\ell$] $\leftarrow \ell$
    while $S$ is non-empty
        $w \leftarrow$ empty string
        $v \leftarrow E[S.pop()]$
        while $v$ is not the root
            $w$.prepend(character from $v$ to its parent)
        // Now $w$ is the encoding of character from $S$.
        $C$.append($w$)
```

Run-time: $O(|T| + |C|)$ $(= O(|\Sigma_S| + |C|)$ if T has no nodes with 1 child)

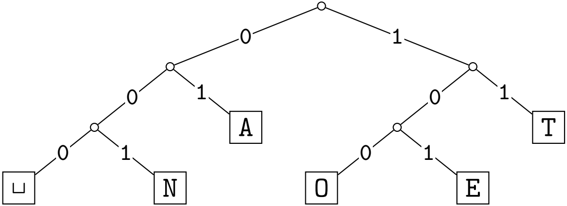
TB changed recently: small comment when $|T| \in O(|\Sigma_S|)$.

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` → `010010000100111`
- Decode `111000001010111` → `TO␣EAT`

Outline

1. 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 source text S, how to determine the "best" trie that minimizes the length of C?

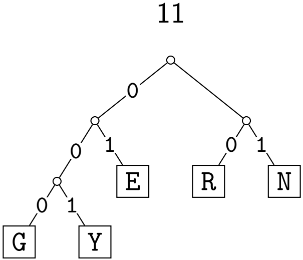
1. Determine frequency of each character $c \in \Sigma$ in S
2. For each $c \in \Sigma$, create "$\boxed{c}$" (height-0 trie holding c).
3. Our tries have a *weight*: sum of frequencies of all letters in trie. Initially, these are just the character frequencies.
4. Find the two tries with the minimum weight.
5. Merge these tries with new interior node; new weight is the sum. (Corresponds to adding one bit to the encoding of each character.)
6. Repeat last two steps until there is only one trie left

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

Example: Huffman tree construction

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

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



GREENENERGY → 000 10 01 01 11 01 11 01 10 000 001

Compression ratio: $\frac{25}{11 \cdot \log 5} \approx 97\%$
(These frequencies are not skewed enough to lead to good compression.)

Huffman's Algorithm: Pseudocode

```
Huffman::encoding($S$, $C$)
$S$: input-stream with characters in $\Sigma_S$, $C$: output-stream
    $f \leftarrow$ array indexed by $\Sigma_S$, initially all-0                    // frequencies
    while $S$ is non-empty do increase $f[S.pop()]$ by 1

    $Q \leftarrow$ min-oriented priority queue that stores tries      // initialize PQ
    for all $c \in \Sigma_S$ with $f[c] > 0$ do
        $Q$.insert(single-node trie for $c$ with weight $f[c]$)

    while $Q.size > 1$ do                                        // build decoding trie
        $T_1 \leftarrow Q$.deleteMin(), $f_1 \leftarrow$weight of $T_1$
        $T_2 \leftarrow Q$.deleteMin(), $f_2 \leftarrow$weight of $T_2$
        $Q$.insert(trie with $T_1$, $T_2$ as subtries and weight $f_1+f_2$)
    $T \leftarrow Q$.deleteMin

    $C$.append(encoding trie $T$)
    Re-set input-stream $S$                                      // actual encoding
    prefixFree::encoding($T, S, C$)
```

Huffman Coding Evaluation

- Note: constructed trie is *not unique* (why?)
  So decoding trie must be transmitted along with the coded text.
- This may make encoding bigger than source text!
- Encoding must pass through text twice (to compute frequencies and to encode). Cannot use a stream unless it can be re-set.

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

- The constructed trie is *optimal* in the sense that coded text is shortest (among all prefix-free character-encodings with $\Sigma_C = \{0, 1\}$).
  We will not go through the proof.
- Many variations (give tie-breaking rules, estimate frequencies, adaptively change encoding, ...)

Outline

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

Run-Length Encoding

- Variable-length code
- Example of **multi-character encoding**: multiple source-text characters receive one code-word.
- The source alphabet and coded alphabet are both binary: $\{0,1\}$.
- Decoding dictionary is uniquely defined and not explicitly stored.

When to use: if S has long runs: $\underbrace{00000}\underbrace{111}\underbrace{0000}$

Encoding idea:

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

Example becomes: $0, 5, 3, 4$

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

Prefix-free Encoding for Positive Integers

Use **Elias gamma coding** 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
⋮	⋮	⋮	⋮

RLE Encoding

```
$RLE::encoding(S, C)$
$S$: input-stream of bits, $C$: output-stream
    $b \leftarrow S.top()$; $C.append(b)$
    while $S$ is non-empty do
        $k \leftarrow 1$                                   // length of run
        while ($S$ is non-empty and $S.top() = b$) do
            $k$++; $S.pop()$

        // compute and append Elias gamma code
        $K \leftarrow$ empty string
        while $k > 1$
            $C.append(0)$
            $K.prepend(k \bmod 2)$
            $k \leftarrow \lfloor k/2 \rfloor$
        $K.prepend(1)$                       // $K$ is binary encoding of $k$
        $C.append(K)$

        $b \leftarrow 1 - b$
```

TB changed recently: The code was incorrect; a single-character run at the end

RLE Decoding

```
*RLE::decoding*$(C, S)$
$C$: input-stream of bits , $S$: output-stream
    $b \leftarrow C.pop()$          // bit-value for the current run
    **while** $C$ is non-empty
        $\ell \leftarrow 0$          // length of base-2 number $-1$
        **while** $C.pop() = 0$ **do** $\ell{+}{+}$
        $k \leftarrow 1$          // base-2 number converted
        **for** $(j \leftarrow 1$ to $\ell)$ **do** $k \leftarrow k * 2 + C.pop()$
        **for** $(j \leftarrow 1$ to $k)$ **do** $S.append(b)$
        $b \leftarrow 1 - b$
```

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

RLE Example

Encoding:
$S =$ 11111110010000000000000000000011111111111

Decoding:
$C =$ 00001101001001010
$S =$ 00000000000001111011

RLE Properties

- An all-0 string of length n would be compressed to $2\lfloor \log n \rfloor + 2 \in o(n)$ bits.
- Usually, we are not that lucky:
  - No compression until run-length $k \geq 6$
  - *Expansion* when run-length $k = 2$ or 4
- Used in some image formats (e.g. TIFF)
- Method can be adapted to larger alphabet sizes (but then the encoding of each run must also store the character)
- Method can be adapted to encode *only* runs of 0 (we will need this soon)

Outline

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

Longer Patterns in Input

TB changed recently: Moved LZW back in its place.

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

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

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

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

Adaptive Dictionaries

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

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

Ingredient 2 for LZW: *adaptive encoding*:

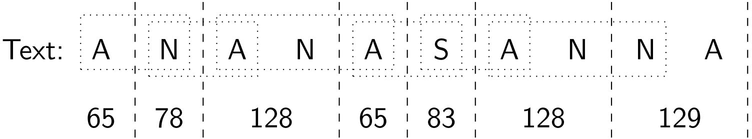
- There is a fixed initial dictionary D_0. (Usually ASCII.)
- For $i \geq 0$, D_i is used to determine the ith output character
- After writing the ith character to output, both encoder and decoder update D_i to D_{i+1}

Encoder and decoder must both know how the dictionary changes.

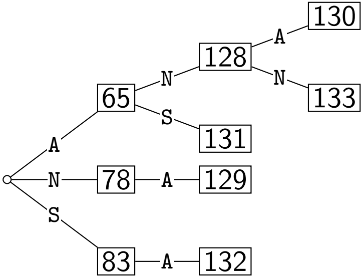
LZW Overview

- Start with dictionary D_0 for $|\Sigma_S|$.
  Usually $\Sigma_S = ASCII$, then this uses codenumbers $0, \ldots, 127$.
- Every step adds to dictionary a multi-character string, using codenumbers $128, 129, \ldots$.
- Encoding:
  - Store current dictionary D_i as a trie.
  - Parse trie to find longest prefix w already in D_i.
    So all of w can be encoded with one number.
  - Add to dictionary the *substring that would have been useful*:
    add wK where K is the character that follows w in S.
  - This creates one child in trie at the leaf where we stopped.
- Output is a list of numbers. This is usually converted to bit-string with fixed-width encoding using 12 bits.
  - This limits the codenumbers to 4096.

LZW Example



Dictionary:



Final output: $\underbrace{000001000001}_{65}\underbrace{000001001110}_{78}\underbrace{000010000000}_{128}\underbrace{000001000001}_{65}\underbrace{000001010011}_{83}\underbrace{000010000000}_{128}\underbrace{000010000001}_{129}$

LZW encoding pseudocode

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

LZW decoding

TB changed recently: I liked how Eric in his video built up the trie and tried to emulate that here.

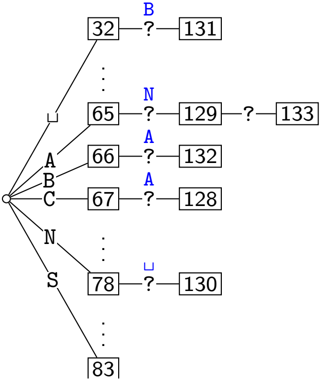
- Build dictionary while reading string by imitating encoder.
- We are one step behind.
- Example:

67	65	78	32	66	129	133
C	A	N	␣	**B**	**AN**	**???**

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

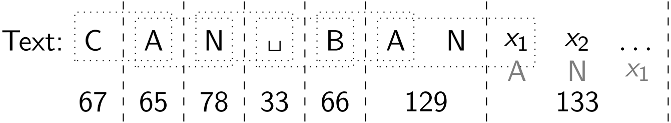
What encoder did:

Deduced one step later:



LZW decoding: the catch

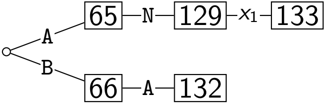
- In this example: Want to decode 133, but incomplete in dictionary!
- What happened during the corresponding encoding?



Text:	C	A	N	␣	B	A N	x_1 x_2 ...
							A N x_1
	67	65	78	33	66	129	133

Dictionary

(parts omitted):



- We know: 133 encodes ANx_1 (for unknown x_1)
- We know: Next step uses 133 = ANx_1
- So x_1 =A and 133 encodes ANA

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

"previous string + first character of previous string"

LZW decoding pseudocode

```
LZW::decoding($C, S$)
$C$: input-stream of integers, $S$: output-stream
    $D \leftarrow$ dictionary that maps $\{0, \ldots, 127\}$ to ASCII
    $idx \leftarrow 128$

    $code \leftarrow C.pop()$; $s \leftarrow D(code)$; $S.append(s)$
    while there are more codes in $C$ do
        $s_{prev} \leftarrow s$; $code \leftarrow C.pop()$

        if $code < idx$
            $s \leftarrow D(code)$
        else if $code = idx$ // special situation!
            $s \leftarrow s_{prev} + s_{prev}[0]$
        else FAIL                // Encoding was invalid

        $S.append(s)$
        $D.insert(idx, s_{prev} + s[0])$
        $idx{+}{+}$
```

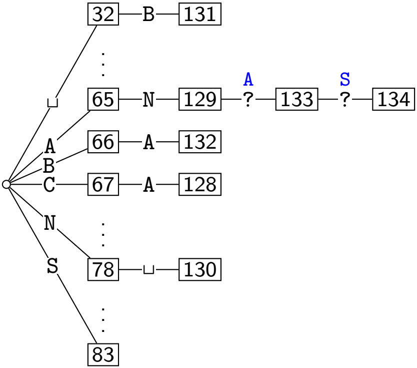
LZW decoding example revisited

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

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

What encoder did:

Deduced one step later:



LZW decoding - second example

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

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

ASCII	
...	
97	a
98	b
...	
101	e
...	
110	n
...	
114	r
...	

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

- To save space, store string as code of prefix + one character.
- Can still look up s in $O(|s|)$ time.

Lempel-Ziv-Welch discussion

- Encoding: $O(|S|)$ time, uses a trie of encoded substrings to store the dictionary
- Decoding: $O(|S|)$ time, uses an array indexed by code numbers to store the dictionary.
- Encoding and decoding need to go through the string only *once* and do not need to see the whole string
  $\Rightarrow$ can do compression while streaming the text
- Compresses quite well ($\approx 45\%$ on English text).

Brief history:

LZ77 Original version ("sliding window")
Derivatives: LZSS, LZFG, LZRW, LZP, DEFLATE, ...
DEFLATE used in (`pk`)`zip`, `gzip`, `PNG`

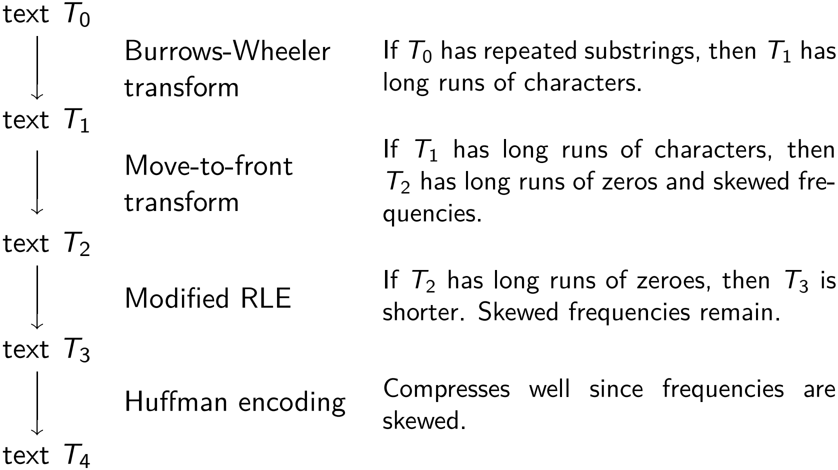
LZ78 Second (slightly improved) version
Derivatives: LZW, LZMW, LZAP, LZY, ...
LZW used in `compress`, `GIF` (patent issues!)

Outline

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

bzip2 overview

To achieve even better compression, bzip2 uses *text transform*: Change input into a different text that is not necessarily shorter, but that has other desirable qualities.



text T_0

Burrows-Wheeler transform

If T_0 has repeated substrings, then T_1 has long runs of characters.

text T_1

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

Modified RLE

If T_2 has long runs of zeroes, then T_3 is shorter. Skewed frequencies remain.

text T_3

Huffman encoding

Compresses well since frequencies are skewed.

text T_4

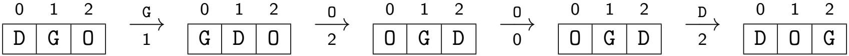
Move-to-Front transform

Recall the MTF heuristic for self-organizing search:

- Dictionary L is stored as an unsorted array or linked list
- After an element is accessed, move it to the front of the dictionary

How can we use this idea for transforming a text with repeat characters?

- Encode each character of source text S by its index in L.
- After each encoding, update L with Move-To-Front heuristic.
- **Example**: $S =$ `GOOD` becomes $C = 1, 2, 0, 2$



Observe: A character in S repeats k times $\Leftrightarrow$ C has run of $k-1$ zeroes

Observe: C contains lots of small numbers and few big ones.

C has the same length as S, but better properties.

Move-to-Front Encoding/Decoding

```
MTF::encoding($S, C$)
    $L \leftarrow$ array with $\Sigma_S$ in some pre-agreed, fixed order (usually ASCII)
    while $S$ is not empty do
        $c \leftarrow S.pop()$
        $i \leftarrow$ index such that $L[i] = c$
        $C.append(i)$
        for $j = i - 1$ down to 0
            swap $L[j]$ and $L[j + 1]$
```

Decoding works in *exactly* the same way:

```
MTF::decoding($C, S$)
    $L \leftarrow$ array with $\Sigma_S$ in some pre-agreed, fixed order (usually ASCII)
    while $C$ is not empty do
        $i \leftarrow$ next integer from $C$
        $S.append(L[i])$
        for $j = i - 1$ down to 0
            swap $L[j]$ and $L[j + 1]$
```

Outline

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

Burrows-Wheeler Transform

Idea:

- *Permute* the source text S: the coded text C has the exact same letters (and the same length), but in a different order.
- **Goal:** If S has repeated substrings, then C should have long runs of characters.
- We need to choose the permutation carefully, so that we can *decode* correctly.

Details:

- Assume that the source text S ends with end-of-word character \$ that occurs nowhere else in S.
- A **cyclic shift** of S is the concatenation of $S[i+1..n-1]$ and $S[0..i]$, for $0 \leq i < n$.
- The encoded text C consists of the last characters of the cyclic shifts of S after sorting them.

BWT Encoding Example

$S =$ `alf␣eats␣alfalfa$`

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

C =

```
$alf␣eats␣alfalfa
␣alfalfa$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
```

Observe: Substring `alf` occurs three times and causes runs `lll` and `aaa` in C (why?)

Fast Burrows-Wheeler Encoding

TB changed recently: This slide is new since regular now has suffix arrays

$S =$ `alf␣eats␣alfalfa$`

	i	ith cyclic shift
0	16	$alf␣eats␣alfalf**a**
1	9	␣alfalfa$alf␣eat**s**
2	4	␣eats␣alfalfa$al**f**
3	16	a$alf␣eats␣alfal**f**
4	1	alf␣eats␣alfalfa**$**
5	13	alfa$alf␣eats␣al**f**
6	10	alfalfa$alf␣eats**␣**
7	6	ats␣alfalfa$alf␣**e**
8	5	eats␣alfalfa$alf**␣**
9	3	f␣eats␣alfalfa$a**l**
10	15	fa$alf␣eats␣alfa**l**
11	12	falfa$alf␣eats␣a**l**
12	2	lf␣eats␣alfalfa$**a**
13	14	lfa$alf␣eats␣alf**a**
14	11	lfalfa$alf␣eats␣**a**
15	8	s␣alfalfa$alf␣ea**t**
16	7	ts␣alfalfa$alf␣e**a**

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

- Need: sorting permutation of cyclic shifts.
- Observe: This is the same as the sorting permutation of the suffixes.
- That's the suffix array! Can compute this in $O(n \log n)$ time.
- Can read BWT encoding from suffix array in linear time.

BWT Decoding

Idea: Given C, we can reconstruct the *first* and *last column* of the array of cyclic shifts by sorting.

C = `ard$rcaaaabb`

1. **Last column:** C
2. **First column:** C **sorted**
3. **Disambiguate by row-index**
   Can argue: Repeated characters are in the same order in the first and the last column (the sort was *stable*).
4. **Starting from $, recover** S

S =

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

BWT Decoding

```
$BWT::decoding(C[0..n-1], S)$
$C$ : string of characters over alphabet $\Sigma_S$, $S$: output-stream
    $A \leftarrow$ array of size $n$        // leftmost column
    for $i = 0$ to $n-1$
        $A[i] \leftarrow (C[i], i)$        // store character and index
    Stably sort $A$ by character
    for $j = 0$ to $n-1$                   // where is the $-char?
        if $C[j] = \$$ break
    repeat
        $S.append$(character stored in $A[j]$)
        $j \leftarrow$ index stored in $A[j]$
    until we have appended $
```

TB changed recently: second for-loop had error (it went to n)

BWT Overview

Encoding cost: $O(n \log n)$

- Read encoding from the suffix array.
- In practice MSD radix sort is good enough (but worst-case $\Theta(n^2)$).

Decoding cost: $O(n + |\Sigma_S|)$ (faster than encoding)

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

They need *all* of the text (no streaming possible). BWT is a **block compression method**.

BWT tends to be slower than other methods, but (combined with MTF, modified RLE and Huffman) gives 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, must send dictionary	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, compress	bzip2 and variants