CS 240 – Data Structures and Data Management

Module 10: Compression

Mark Petrick   Olga Veksler

Based on lecture notes by many previous cs240 instructors

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

Winter 2020

References: Goodrich & Tamassia 10.3

version 2020-04-01 04:54

Outline

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

Outline

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: $\{0, 1\}$.

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 *(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

$$A{\cdot}P{\cdot}P{\cdot}L{\cdot}E \to (65, 80, 80, 76, 69) \to 1000001\ 1010000\ 1010000\ 1001100\ 1000101$$

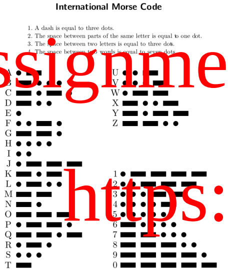
- Standard in *all* computers and often our source alphabet.
- Not well-suited for non-English text: ISO-8859 extends to 8 bits, handles most Western languages

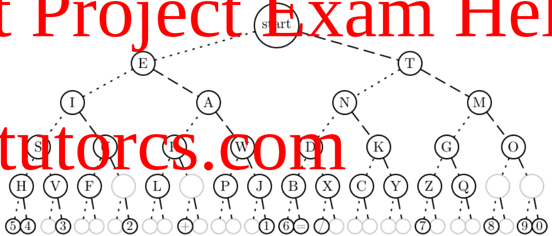
Other (earlier) examples: Caesar shift, Baudot code, Murray code

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

Variable-Length Codes

Example 1: Morse code.





Pictures taken from http://apfelmus.nfshost.com/articles/fun-with-morse-code.html

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.

Encoding(E, $S[0..n-1]$)
E : the encoding dictionary, S: text with characters in Σ_S

```
initialize empty string C
for i = 0 ... n − 1
    x ← E.search(S[i])
    C.append(x)
return C
```

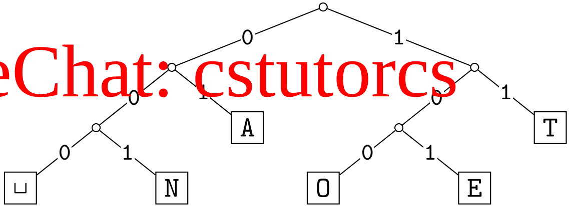

Example: encode text "`WATT`" with Morse code:

$$\underbrace{\bullet\ \bullet\ \text{—}}_{\texttt{W}}\ \underbrace{\bullet\ \text{—}}_{\texttt{A}}\ \underbrace{\text{—}}_{\texttt{T}}\ \underbrace{\text{—}}_{\texttt{T}}$$

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?)

```
PrefixFreeDecoding(T, C[0..n−1])
T: the trie of a prefix-free code, C: text with characters in Σ_C
    initialize empty string S
    i ← 0
    while i < n
        r ← T.root
        while r is not a leaf
            if i = n return "invalid encoding"
            c ← child of r that is labelled with C[i]
            i ← i + 1
            r ← c
        S.append(character stored at r)
    return S
```

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

Encoding from the Trie

We can also encode directly from the trie.

PrefixFreeEncodingFromTrie$(T, S[0..n-1])$
T: the trie of a prefix-free code, S: text with characters in Σ_S

$L \leftarrow$ array of nodes in T indexed by Σ_S
for all leaves ℓ in T
  $L[\text{character at } \ell] \leftarrow \ell$
initialize empty string C
for $i = 0$ to $n-1$
  $w \leftarrow$ empty string; $v \leftarrow L[S[i]]$
  while v is not the root
    w.*prepend*(character from v to its parent)
  // Now w is the encoding of $S[i]$.
  C.*append*(w)
return C

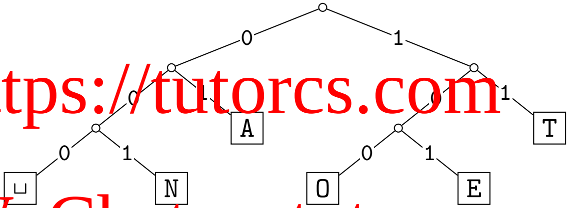
Run-time: $O(|T| + |C|) = O(|\Sigma_S| + |C|)$.

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:



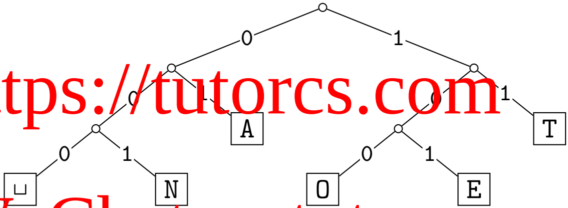
- Encode `AN␣ANT`
- Decode 111000001010111

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` $\rightarrow$ 010010000100111
- Decode 111000001010111 $\rightarrow$ `TO␣EAT`

Outline

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

Character Frequency

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.

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'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?

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?
A min-ordered heap! Step 4 is two *delete-min*s, Step 5 is *insert*

Example: Huffman tree construction

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

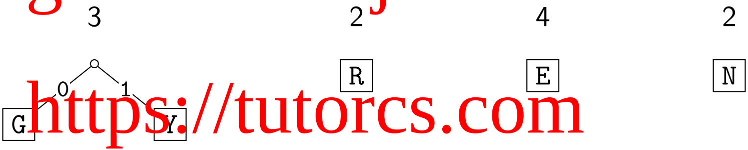
Character frequencies: G : 2, R : 2, E : 4, N : 2 Y : 1

2	2	4	2	1
G	R	E	N	Y

Example: Huffman tree construction

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

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



Example: Huffman tree construction

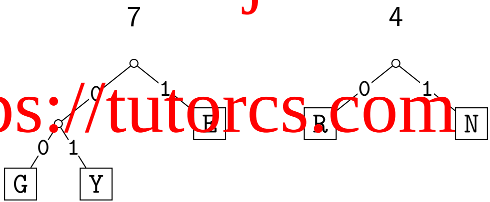
Example text: GREENENERGY, $\Sigma_S = \{G, R, E, N, Y\}$

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

Example: Huffman tree construction

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

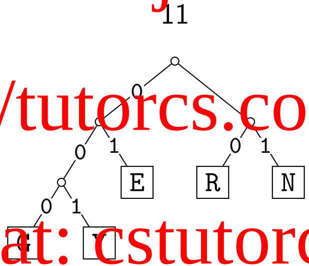
Character frequencies: G : 2, R : 2, E : 4, N : 2 Y : 1



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

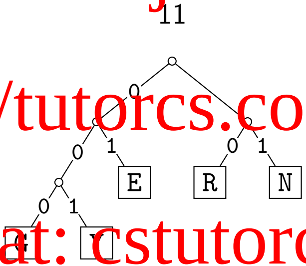


GREENENERGY $\rightarrow$

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 $\rightarrow$ 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($S[0..n-1]$)

S: text over some alphabet Σ_S

$f \leftarrow$ array indexed by Σ_S, initially all-0 // frequencies

for $i = 0$ to $n-1$ **do** increase $f[S[i]]$ 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*

$C \leftarrow$ *PrefixFreeEncodingFromTrie*(T, S)

return C and T

Huffman Coding Evaluation

- Note: 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 text twice (to compute frequencies and to encode)
- Encoding run-time: $O(|\Sigma_S| \log |\Sigma_S| + |C|)$
- Decoding run-time: $O(|C|)$
- The constructed trie is *optimal* in the sense that C is shortest (among all prefix-free character-encodings with $\Sigma_C = \{0, 1\}$).
  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
   - bzip2
   - Burrows-Wheeler Transform
   - Lempel-Ziv-Welch

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
$\vdots$	$\vdots$	$\vdots$	$\vdots$

RLE Encoding

RLE-Encoding$(S[0...n-1])$
S: bitstring

initialize output string $C \leftarrow S[0]$
$i \leftarrow 0$ // index of parsing S
while $i < n$ **do**
  $k \leftarrow 1$ // length of run
  while $(i + k < n$ and $S[i+k] = S[i])$ **do** $k{+}{+}$
  $i \leftarrow i + k$
  // 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)
return C

RLE Decoding


```
RLE-Decoding(C)
C: stream of bits
    initialize output string S
    b ← C.pop()              // bit-value for the current run
    repeat
        ℓ ← 0                // length of base-2 number −1
        while C.pop() = 0 do ℓ++
        k ← 1                // base-2 number converted
        for (j ← 1 to ℓ) do k ← k ∗ 2 + C.pop()
        for (j ← 1 to k) do S.append(b)
        b ← 1 − b
    until C has no more bits left
    return S
```

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

RLE Example

Encoding:

$S = 11111110010000000000000000000000011111111111$

$C = 1$

Decoding:

$C = 00001101001001010$

$S =$

RLE Example

Encoding:

$S = 1111111001000000000000000000000011111111111$

$k =$ [illegible]

$C = 100111$

Decoding:

$C = 00001101001001010$

$S =$

RLE Example

Encoding:

$S =$ 11111110010000000000000000000011111111111

$k = 2$

$C =$ 100111010

Decoding:

$C =$ 00001101001001010

$S =$

RLE Example

Encoding:

$S = 1111111001000000000000000000011111111111$

$k = 1$

$C = 1001110101$

Decoding:

$C = 00001101001001010$

$S =$

RLE Example

Encoding:

$S = 1111111001\textcolor{red}{00000000000000000000}11111111111$

$k = 20$

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

Decoding:

$C = 00001101001001010$

$S =$

RLE Example

Encoding:

$S = 1111111001000000000000000000011111111111$

$k = 11$

$C = 10011101010000101000001011$

Decoding:

$C = 00001101001001010$

$S =$

RLE Example

Encoding:

$S =$ 11111110010000000000000000000011111111111

$C =$ 10011101010000101000001011

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

Decoding:

$C =$ 00001101001001010

$S =$

RLE Example

Encoding:

$S = 11111110010000000000000000000011111111111$

$C = 10011101010000101000001011$

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

Decoding:

$C = 00001101001001010$

$S =$

RLE Example

Encoding:

$S = 11111110010000000000000000000011111111111$

$C = 10011101010000101000001011$

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

Decoding:

$C = 00001101001001010$

$b = 0$

$S =$

RLE Example

Encoding:

$S = 11111110010000000000000000000011111111111$

$C = 10011101010000101000001011$

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

Decoding:

$C = 00001101001001010$

$b = 0$

$\ell = 3$

$S =$

RLE Example

Encoding:

$S = 11111110010000000000000000000011111111111$

$C = 10011101010000101000001011$

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

Decoding:

$C = 0000110100100\,1010$

$b = 0$

$\ell = 3$

$k = 13$

$S = 0000000000000$

RLE Example

Encoding:

$S = 11111110010000000000000000000011111111111$

$C = 10011101010000101000001011$

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

Decoding:

$C = 00001101{\color{red}00}1001010$

$b = 1$

$\ell = 2$

$k =$

$S = 0000000000000$

RLE Example

Encoding:

$S = 11111110010000000000000000000011111111111$

$C = 10011101010000101000001011$

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

Decoding:

$C = 00001101001001010$

$b = 1$

$\ell = 2$

$k = 4$

$S = 0000000000001111$

RLE Example

Encoding:

$S = 11111110010000000000000000000011111111111$

$C = 10011101010000101000001011$

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

Decoding:

$C = 00001101001001010$

$b = 0$

$\ell = 0$

$k =$

$S = 0000000000001111$

RLE Example

Encoding:

$S = 11111110010000000000000000000011111111111$

$C = 10011101010000101000001011$

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

Decoding:

$C = 00001101001001010$

$b = 0$

$\ell = 0$

$k = 1$

$S = 00000000000001110$

RLE Example

Encoding:

$S = 11111110010000000000000000000011111111111$

$C = 10011101010000101000001011$

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

Decoding:

$C = 00001101001001010$

$b = 1$

$\ell = 1$

$k =$

$S = 00000000000000011110$

RLE Example

Encoding:

$S = 11111110010000000000000000000011111111111$

$C = 10011101010000101000001011$

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

Decoding:

$C = 00001101001001010$

$b = 1$

$\ell = 1$

$k = 2$

$S = 000000000000011110\textcolor{red}{11}$

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
   - bzip2
   - Burrows-Wheeler Transform
   - Lempel-Ziv-Welch

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		
$\downarrow$	Burrows-Wheeler transform	If T_0 has repeated substrings, then T_1 has long runs of characters.
text T_1		
$\downarrow$	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		
$\downarrow$	Modified RLE	If T_2 has long runs of zeroes, then T_3 is shorter. Skewed frequencies remain.
text T_3		
$\downarrow$	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?

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**: $S = \texttt{GOOD}$ becomes $C = 1, 2, 0, 2$

$$\begin{array}{ccc} 0 & 1 & 2 \\ \hline \texttt{D} & \texttt{G} & \texttt{O} \end{array} \xrightarrow[1]{\texttt{G}} \begin{array}{ccc} 0 & 1 & 2 \\ \hline \texttt{G} & \texttt{D} & \texttt{O} \end{array} \xrightarrow[2]{\texttt{O}} \begin{array}{ccc} 0 & 1 & 2 \\ \hline \texttt{O} & \texttt{G} & \texttt{D} \end{array} \xrightarrow[0]{\texttt{O}} \begin{array}{ccc} 0 & 1 & 2 \\ \hline \texttt{O} & \texttt{G} & \texttt{D} \end{array} \xrightarrow[2]{\texttt{D}} \begin{array}{ccc} 0 & 1 & 2 \\ \hline \texttt{D} & \texttt{O} & \texttt{G} \end{array}$$

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**: $S = \texttt{GOOD}$ becomes $C = 1, 2, 0, 2$

$$\begin{array}{ccc} 0 & 1 & 2 \\ \hline \texttt{D} & \texttt{G} & \texttt{O} \end{array} \xrightarrow[1]{\texttt{G}} \begin{array}{ccc} 0 & 1 & 2 \\ \hline \texttt{G} & \texttt{D} & \texttt{O} \end{array} \xrightarrow[2]{\texttt{O}} \begin{array}{ccc} 0 & 1 & 2 \\ \hline \texttt{O} & \texttt{G} & \texttt{D} \end{array} \xrightarrow[0]{\texttt{O}} \begin{array}{ccc} 0 & 1 & 2 \\ \hline \texttt{O} & \texttt{G} & \texttt{D} \end{array} \xrightarrow[2]{\texttt{D}} \begin{array}{ccc} 0 & 1 & 2 \\ \hline \texttt{D} & \texttt{O} & \texttt{G} \end{array}$$

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-encode(S)

$L \leftarrow$ array with Σ_S in some pre-agreed, fixed order (usually ASCII)
while S has more characters **do**
    $c \leftarrow$ next character of S
    output index i such that $L[i] = c$
    for $j = i - 1$ down to 0
        swap $L[j]$ and $L[j+1]$

Decoding works in *exactly* the same way:

MTF-decode(C)

$L \leftarrow$ array with Σ_S in some pre-agreed, fixed order (usually ASCII)
while C has more characters **do**
    $i \leftarrow$ next integer from C
    output $L[i]$
    for $j = i - 1$ down to 0
        swap $L[j]$ and $L[j+1]$

Outline

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

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

```
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 Encoding 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 Encoding 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 Encoding 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
```


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

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


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

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


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

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*).


```
 $,3.........a,0
 a,0.........r,1
 a,6.........d,2
 a,7.........$,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

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


```
 $,3..........a,0
 a,0..........r,1
 a,6..........d,2
 a,7..........$,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

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 = a

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

BWT Decoding

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 = ab

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

BWT Decoding

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 = abr

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

BWT Decoding

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 = `abracadabra$`

$,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$(C[0..n-1])$
C : string of characters over alphabet Σ_S

    $A \leftarrow$ 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 // where is the \$-char?
        if $C[j] = \$$ **break**
    $S \leftarrow$ empty string
    repeat
        $j \leftarrow$ index stored in $A[j]$
        $S.$*append*$(C[j])$
    until $C[j] = \$$
    return S

BWT Overview

Encoding cost: $O(n(n + |\Sigma_S|))$ (using MSD or LSD radix sort) and often better

Encoding is theoretically possible in $O(n)$ time, assuming $|\Sigma_S| = O(1)$:

- Sorting cyclic shifts of S is equivalent to sorting the suffixes of $S \cdot S$ that have length $> n$
- This can be done by traversing the suffix tree of $S \cdot S$

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.

Outline

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

Longer Patterns in Input

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

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

- English text:
  Most frequent digraphs: `TH, ER, ON, AN, RE, HE, IN, ED, ND, HA`
  Most frequent 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.

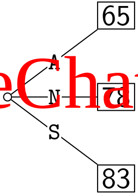
LZW Overview

- 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

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

Dictionary:

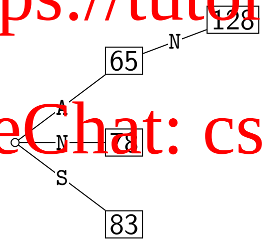


LZW Example

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

65

Dictionary:

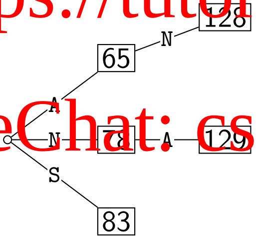


LZW Example

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

65 78

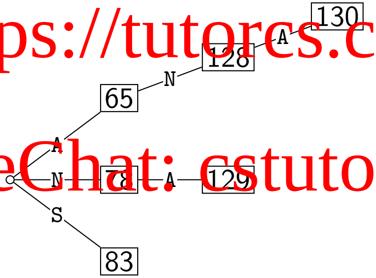
Dictionary:



LZW Example

Text:	A	N	A	N	A	S	A	N	N	A
	65	78	128							

Dictionary:

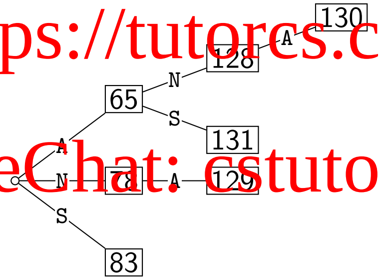


LZW Example

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

65 78 128 65

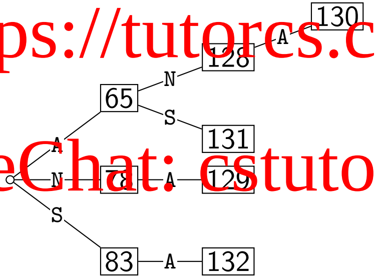
Dictionary:



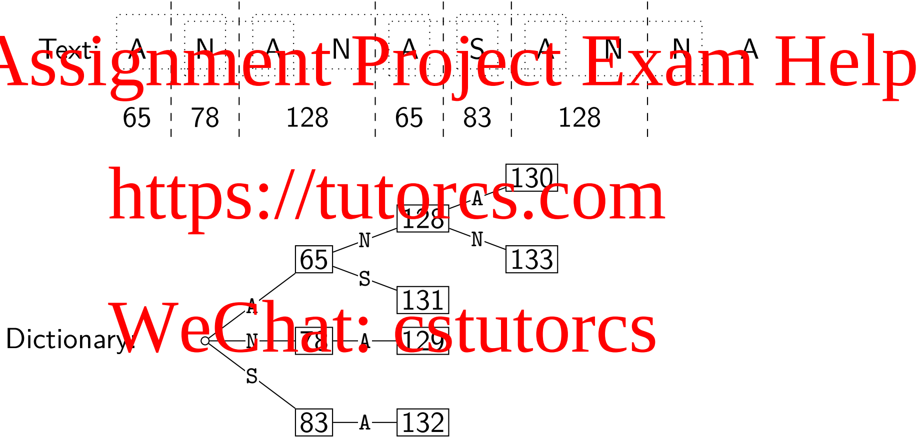
LZW Example

Text:	A	N	A N	A	S	A N N A
	65	78	128	65	83	

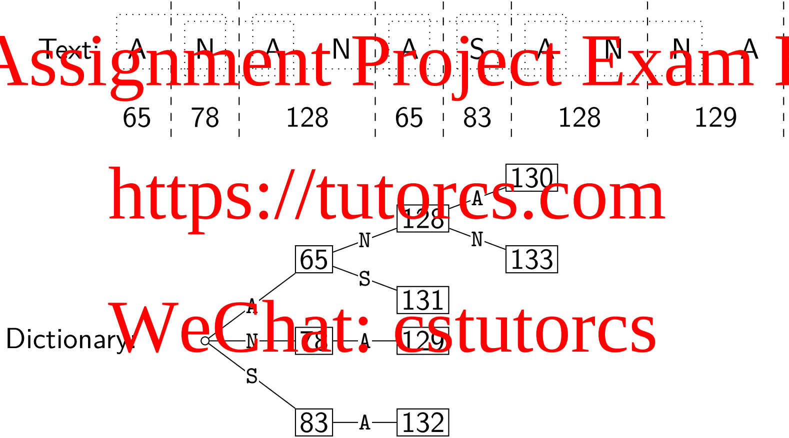
Dictionary:



LZW Example


Text: A N A N A S A N N A
65 78 128 65 83 128
Dictionary:
65
78
83
128
129
130
131
132
133
A
N
S

LZW Example

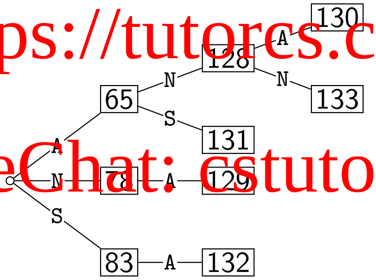


Text: A N A N A S A N N A
65 78 128 65 83 128 129
Dictionary:
65 A
78 N
83 S
128 N
129 A
130 A
131 S
132 A
133 N

LZW Example

Text:	A	N	A N	A	S	A N	N A
	65	78	128	65	83	128	129

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-encode(S)
S : stream of characters

```
1.   Initialize dictionary D with ASCII in a trie
2.   idx ← 128
3.   while there is input in S do
4.       v ← root of trie D
5.       K ← S.peek()
6.       while (v has a child c labelled K)
7.           v ← c; S.pop()
8.           if there is no more input in S break    (goto 10)
9.           K ← S.peek()
10.      output codenumber stored at v
11.      if there is more input in S
12.          create child of v labelled K with codenumber idx
13.          idx++
```

LZW decoding

- Same idea: build dictionary while reading string.
- Dictionary maps numbers to strings.
  To save space, store string as code of prefix + one character.
- Example:

$D =$

Code #	String
...	
32	␣
...	
...	
65	A
66	B
67	C
...	
78	N
...	
83	S
...	

input	decodes to	Code #	String (human)	String (computer)

LZW decoding

- Same idea: build dictionary while reading string.
- Dictionary maps numbers to strings.
  To save space, store string as code of prefix + one character.
- Example: 67

$D =$

Code #	String
...	
32	␣
...	
...	
65	A
66	B
67	C
...	
78	N
...	
83	S
...	

Code # input	decodes to	Code #	String (human)	String (computer)
67	C			

LZW decoding

- Same idea: build dictionary while reading string.
- Dictionary maps numbers to strings.
  To save space, store string as code of prefix + one character.
- Example: 67 65

$D =$

Code #	String
...	
32	␣
...	
...	
65	A
66	B
67	C
...	
78	N
...	
83	S
...	

input	decodes to	Code #	String (human)	String (computer)
67	C			
65	A	128	CA	67, A

LZW decoding

- Same idea: build dictionary while reading string.
- Dictionary maps numbers to strings.
  To save space, store string as code of prefix + one character.
- Example: 67 65 78

$D =$

Code #	String
...	
32	␣
...	
...	
65	A
66	B
67	C
...	
78	N
...	
83	S
...	

input	decodes to	Code #	String (human)	String (computer)
67	C			
65	A	128	CA	67, A
78	N	129	AN	65, N

LZW decoding

- Same idea: build dictionary while reading string.
- Dictionary maps numbers to strings.
  To save space, store string as code of prefix + one character.
- Example: 67 65 78 32

$D =$

Code #	String
...	
32	␣
...	
...	
65	A
66	B
67	C
...	
78	N
...	
83	S
...	

input	decodes to	Code #	String (human)	String (computer)
67	C			
65	A	128	CA	67, A
78	N	129	AN	65, N
32	␣	130	N␣	78, ␣

LZW decoding

- Same idea: build dictionary while reading string.
- Dictionary maps numbers to strings.
  To save space, store string as code of prefix + one character.
- Example: 67 65 78 32 66

$D =$

Code #	String
. . .	
32	␣
. . .	
. . .	
65	A
66	B
67	C
. . .	
78	N
. . .	
83	S
. . .	

input	decodes to	Code #	String (human)	String (computer)
67	C			
65	A	128	CA	67, A
78	N	129	AN	65, N
32	␣	130	N␣	78, ␣
66	B	131	␣B	32, B

LZW decoding

- Same idea: build dictionary while reading string.
- Dictionary maps numbers to strings.
  To save space, store string as code of prefix + one character.
- Example: 67 65 78 32 66 129

$D =$

Code #	String
...	
32	␣
...	
...	
65	A
66	B
67	C
...	
78	N
...	
83	S
...	

input	decodes to	Code #	String (human)	String (computer)
67	C			
65	A	128	CA	67, A
78	N	129	AN	65, N
32	␣	130	N␣	78, ␣
66	B	131	␣B	32, B
129	AN	132	BA	66, A

LZW decoding

- Same idea: build dictionary while reading string.
- Dictionary maps numbers to strings.
  To save space, store string as code of prefix + one character.
- Example: 67 65 78 32 66 129 133

$D =$

Code #	String
...	
32	␣
...	
...	
65	A
66	B
67	C
...	
78	N
...	
83	S
...	

input	decodes to	Code #	String (human)	String (computer)
67	C			
65	A	128	CA	67, A
78	N	129	AN	65, N
32	␣	130	N␣	78, ␣
66	B	131	␣B	32, B
129	AN	132	BA	66, A
133	???	133		

LZW decoding: the catch

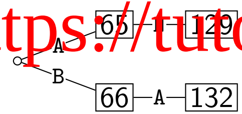
- In this example: Want to decode 133, but not yet in dictionary!
- What happened during the corresponding encoding?

LZW decoding: the catch

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

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

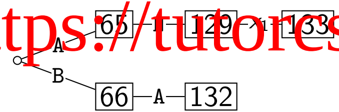
Dictionary
(parts omitted):



LZW decoding: the catch

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

Dictionary
(parts omitted):



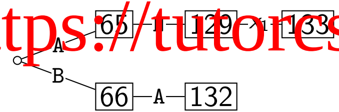
- We know: 133 encodes ANx_1 (for unknown x_1)

LZW decoding: the catch

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

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

Dictionary
(parts omitted):

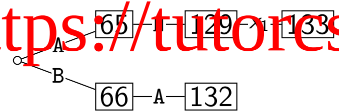


- We know: 133 encodes $\texttt{AN}x_1$ (for unknown x_1)
- We know: Next step uses $133 = \texttt{AN}x_1$

LZW decoding: the catch

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

Dictionary
(parts omitted):



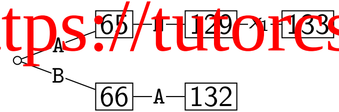
- We know: 133 encodes ANx_1 (for unknown x_1)
- We know: Next step uses 133 = ANx_1
- So x_1 =A and 133 encodes ANA

LZW decoding: the catch

- In this example: Want to decode 133, but not yet 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-decode(C)
C: stream of integers
    D ← dictionary that maps {0, ..., 127} to ASCII
    idx ← 128
    S ← empty string
    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)
        else if code = idx // special situation!
            s ← s_prev + s_prev[0]
        else FAIL          // Encoding was invalid
        S.append(s)
        D.insert(idx, s_prev + s[0])
        idx++
    return S
```

LZW decoding example revisited

- Example: 67 65 78 32 66 129 133 83

$D =$

Code #	String
...	
32	␣
...	
...	
65	A
66	B
67	C
...	
78	N
...	
83	S
...	

input	decodes to	Code #	String (human)	String (computer)
67	C			
65	A	128	CA	67, A
78	N	129	AN	65, N
32	␣	130	N␣	78, ␣
66	B	131	␣B	32, B
129	AN	132	BA	66, A
133	ANA	133	ANA	129, A
83	S	134	ANAS	133, S

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!)

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