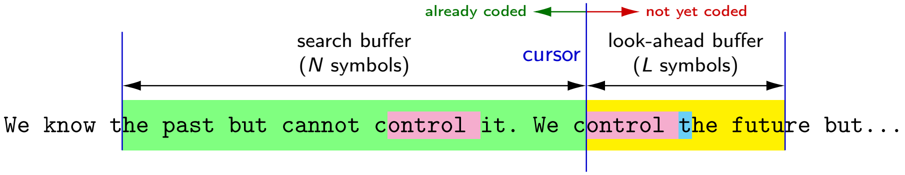
Dictionary-based Coding



Last Lecture: Predictive Lossless Coding

Predictive Lossless Coding

- Simple and effective way to exploit dependencies between neighboring symbols / samples
- Optimal predictor: Conditional mean (requires storage of large tables)

Affine and Linear Prediction

- Simple structure, low-complex implementation possible
- Optimal prediction parameters are given by solution of Yule-Walker equations
- Works very well for real signals (e.g., audio, images, ...)

Efficient Lossless Coding for Real-World Signals

- Affine/linear prediction (often: block-adaptive choice of prediction parameters)
- Entropy coding of prediction errors (e.g., arithmetic coding)
  - Using marginal pmf often already yields good results
  - Can be improved by using conditional pmfs (with simple conditions)

Dictionary-Based Coding

Coding of Text Files

- Very high amount of dependencies
- Affine prediction does not work (requires linear dependencies)
- Higher-order conditional coding should work well, but is way to complex (memory)

➡ Alternative: Do not code single characters, but words or phrases

Example: English Texts

- *Oxford English Dictionary* lists less than 230 000 words (including obsolete words)
- On average, a word contains about 6 characters

➡ Average codeword length per character would be limited by

$$\bar{\ell} < \frac{1}{6} \cdot \left\lceil \log_2 230\,000 \right\rceil \approx 3.0$$

➡ Including "phrases" would further increase coding efficiency

Lempel-Ziv Coding

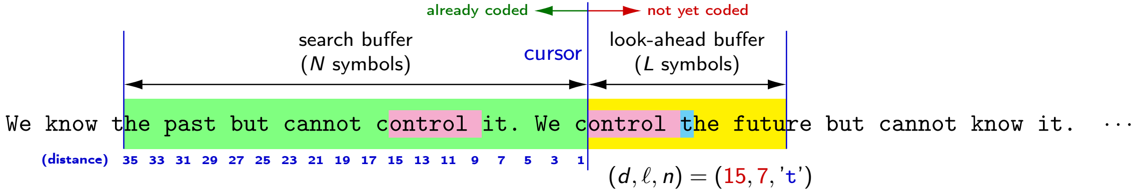
Universal Algorithms for Lossless Data Compression

- Based on the work of Abraham Lempel and Jacob Ziv
- Basic idea: Construct dictionary during encoding and decoding

Two Basic Variants

- **LZ77**: Based on [Ziv, Lempel, "A Universal algorithm for sequential data compresion", 1977]
  - ➡ Lempel-Ziv-Storer-Szymanski (LSZZ)
  - ➡ DEFLATE used in **ZIP**, **gzip**, **PNG**, **TIFF**, **PDF**, **OpenDocument**, ...
  - ➡ Lempel-Ziv-Markov Chain Algorithm (LZMA) used in **7zip**, **xv**, **lzip**
  - ➡ ...
- **LZ78**: Based on [Ziv, Lempel, "Compression of individual sequences via variable-rate coding", 1978]
  - ➡ Lempel-Ziv-Welch (LZW) used in **compress**, **GIF**, optional support in **PDF**, **TIFF**
  - ➡ ...

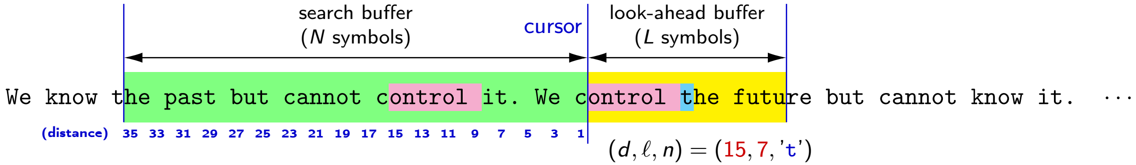
The Lempel-Ziv 1977 Algorithm (LZ77)



Basic Idea of the LZ77 Algorithm

- Dictionary of variable-length sequences is given by the preceding N symbols (sliding window)
  - ➡ Find longest possible match for the sequence at the start of the look-ahead buffer
- Message is coded as sequence of triples (d, ℓ, n):
  - ➡ d : distance of best match from next symbol to be coded
  - ➡ ℓ : length of matched phrase (match starts in search buffer but may reach into look-ahead buffer)
  - ➡ n : next symbol after matched sequence
- If no match is found, then $(1, 0, n)$ is coded (with n being the next symbol after the cursor)

Simplest Version: LZ77 Algorithm with Fixed-Length Coding



How Many Bits Do We Need ?

- **Distance d :** Can take values from $1 \dots N$ (*we could actually code $d-1$*)
  ➡ Require $n_d = \lceil \log_2 N \rceil$ bits
- **Length ℓ :** Can take values from $0 \dots L-1$ (*$\ell+1$ symbols must fit into look-ahead buffer*)
  ➡ Require $n_\ell = \lceil \log_2 L \rceil$ bits
- **Next symbol n :** Can be any symbol of the alphabet $\mathcal{A}$ with size $|\mathcal{A}|$
  ➡ Require $n_n = \lceil \log_2 |\mathcal{A}| \rceil$ bits (*in most applications : 8 bits*)

➡ The sizes of both the preview and the look-ahead buffer should be integer powers of two !

Toy Example: LZ77 Encoding

Message: `Miss␣Mississippi`

search buffer	look-ahead buffer	(d, ℓ, n)
	`Miss`	(1 , 0 , M)
`M`	`iss␣`	(1 , 0 , i)
`Mi`	`ss␣M`	(1 , 0 , s)
`Mis`	`s␣Mi`	(1 , 1 , ␣)
`Miss␣`	`Miss`	(5 , 3 , s)
`iss␣Miss`	`issi`	(3 , 3 , i)
`Mississi`	`ppi`	(1 , 0 , p)
`ississip`	`pi`	(1 , 1 , i)

original message:

- 16 characters (8 bits per symbols)
- ➡ 128 bits (16×8 bits)

LZ77 configuration:

- search buffer of $N = 8$ symbols
- look-ahead buffer of $L = 4$ symbols

coded representation (fixed-length):

- ➡ 8 triples (d, ℓ, n)
- 13 bits per triple ($3 + 2 + 8$ bits)
- ➡ 104 bits (19% bit savings)

Toy Example: LZ77 Decoding

Coded representation: (1, 0, M) (1, 0, i) (1, 0, s) (1, 1, ␣) (5, 3, s) (3, 3, i) (1, 0, p) (1, 1, i)

Decode message: Miss␣Mississippi

search buffer	(d , ℓ , n)	decoded phrase
	(1 , 0 , M)	M
M	(1 , 0 , i)	i
Mi	(1 , 0 , s)	s
Mis	(1 , 1 , ␣)	s␣
Miss␣	(5 , 3 , s)	Miss
iss␣Miss	(3 , 3 , i)	issi
Mississi	(1 , 0 , p)	p
ississip	(1 , 1 , i)	pi

Coding Efficiency and Complexity of LZ77

Coding Efficiency

- The LZ77 algorithm is asymptotically optimal (e.g., when using unary codes for d and ℓ)

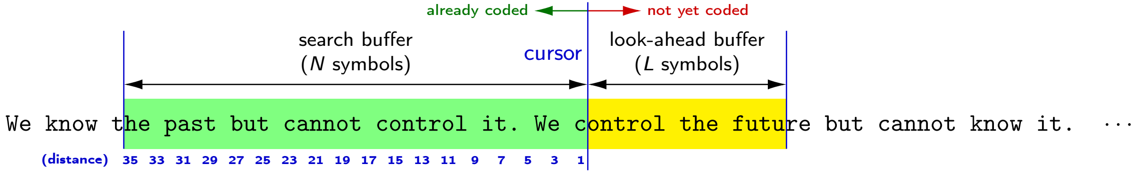
$$N \to \infty,\, L \to \infty \qquad \Longrightarrow \qquad \bar{\ell} \to \bar{H}$$

- Proof can be found in [Cover, Thomas, "Elements of Information Theory"]
- In practice: Require really large search buffer sizes N

Implementation Complexity

- **Decoder**: Very low complexity (just copying characters)
- **Encoder**: Highly depends on buffer size N and actual implementation
  - ➡ Use suitable data structures such as search trees, radix trees, hash tables
  - ➡ Not necessary to find the "best match" (note: shorter match can actually be more efficient)
  - ➡ There are very efficient implementations for rather large buffer sizes (e.g., $N = 32\,768$)

LZ77 Variant: The Lempel-Ziv-Storer-Szymanski Algorithm (LZSS)



Changes relative to LZ77 Algorithm

1. At first, code a single bit b to indicate whether a match is found
2. For matches, don't transmit the following symbol

➡ Message is coded as sequence of tuples $(b, \{d, \ell\} \mid n)$

- The indication bit b signals whether a match is found ($b = 1 \;\rightarrow\;$ match found)
- If $(b = 0)$, then code next symbol n as literal
- If $(b = 1)$, then code the match as distance-length pair $\{d, \ell\}$ (with $d \in [1, N]$ and $\ell \in [1, L]$)

Toy Example: LZSS Encoding

Message: `Miss␣Mississippi`

search buffer	look-ahead	$(\,b, \{d, \ell\} \mid n\,)$
	`Miss`	(0, M)
`M`	`iss␣`	(0, i)
`Mi`	`ss␣M`	(0, s)
`Mis`	`s␣Mi`	(1, 1, 1)
`Miss`	`␣Mis`	(0, ␣)
`Miss␣`	`Miss`	(1, 5, 4)
`iss␣Miss`	`issi`	(1, 3, 4)
`Mississi`	`ppi`	(0, p)
`ississip`	`pi`	(1, 1, 1)
`ssissipp`	`i`	(1, 3, 1)

original message:

- 16 characters (8 bits per symbols)
- ➜ 128 bits (16×8 bits)

LZSS configuration:

- search buffer of $N = 8$ symbols
- look-ahead buffer of $L = 4$ symbols

coded representation (fixed-length):

- ➜ 5 literals (5×9 bits)
- ➜ 5 matches (5×6 bits)
- ➜ 75 bits (41% bit savings)

Toy Example: LZSS Decoding

Coded representation: (0, M) (0, i) (0, s) (1, 1, 1) (0, ␣) (1, 5, 4) (1, 3, 4) (0, p) (1, 1, 1) (1, 3, 1)

Decode message: Miss␣Mississippi

search buffer	$(\, b,\ \{d,\ \ell\} \mid n\,)$	decoded phrase	
	(0, M)	M	
M	(0, i)	i	
Mi	(0, s)	s	
Mis	(1, 1, 1)	s	
Miss	(0, ␣)	␣	
Miss␣	(1, 5, 4)	Miss	
iss␣Miss	(1, 3, 4)	issi	(note: *copy symbol by symbol*)
Mississi	(0, p)	p	
ississip	(1, 1, 1)	p	
sissipp	(1, 3, 1)	i	

The DEFLATE Algorithm: Combining LZSS with Huffman Coding

The Concept of DEFLATE

- Pre-process message/file/symbol sequence using the **LZSS algorithm** (remove dependencies)
- Entropy coding of tuples $(b, \{d, \ell\} \mid n)$ using **Huffman coding**

Details of DEFLATE Format

- Input as interpreted as sequence of bytes (alphabet size of 256)
- LZSS configuration: Search buffer of $\mathbf{N = 32\,768}$, look-ahead buffer of $\mathbf{L = 258}$
- Input data are coded using **variable-length blocks** (for optimizing the Huffman coding)

3-bit block header (at start of each block)

1 bit	0	there are blocks that follow the current block
	1	this is the last block of the file / data stream
2 bits	00	uncompressed block (number of bytes in block is coded after block header, max. 65k)
	01	compressed block using pre-defined Huffman tables
	10	compressed block with transmitted Huffman tables **(most frequently used type)**
	11	reserved (forbidden)

The DEFLATE Format: Two Huffman Tables

Main Huffman table with 288 codewords

index n	**meaning** (additional codewords follow for $n = 257 \ldots 285$)	
0 – 255	literal with ASCII code being equal to n	
256	**end-of-block** (last symbol of a block)	
257 – 264	match with $\ell = (n - 254)$	
265 – 268	match with $\ell = 2 \cdot (n - 260) + 1 + x$	(1 extra bit for x)
269 – 272	match with $\ell = 4 \cdot (n - 265) + 3 + x$	(2 extra bits for x)
273 – 276	match with $\ell = 8 \cdot (n - 269) + 3 + x$	(3 extra bits for x)
277 – 280	match with $\ell = 16 \cdot (n - 273) + 3 + x$	(4 extra bits for x)
281 – 284	match with $\ell = 32 \cdot (n - 277) + 3 + x$	(5 extra bits for x)
285	match with $\ell = 258$	
286 – 287	reserved (forbidden codeword)	

Note 1: The values for x are coded using fixed-length codes.

Note 2: The match size must be in range $\ell = 3 \ldots 258$.

Huffman table for distance

n	**distance** d	**bits for** z
0 – 3	$d = 1 + n$	
4	$d = 5 + z$	1
5	$d = 7 + z$	1
6	$d = 9 + z$	2
7	$d = 13 + z$	2
8	$d = 17 + z$	4
⋮	⋮	⋮
26	$d = 8\,193 + z$	12
27	$d = 12\,289 + z$	12
28	$d = 16\,385 + z$	13
29	$d = 24\,577 + z$	13
30 – 31	reserved	

Note: The values for z are coded using fixed-length codes.

The DEFLATE Algorithm in Practice

Encoding and Decoding

- **Decoding**: Straightforward (follow format specification)
- **Encoding**: Can trade-off coding efficiency and complexity
  - ➡ Fixed pre-defined or dynamic Huffman tables
  - ➡ Determination of suitable block sizes
  - ➡ Simplified search for finding best matches

Applications

- One the most used algorithms in practice
  - ➡ Archive formats: Library **zlib**, **ZIP**, **gzip**, **PKZIP**, **Zopfli**, **CAB**
  - ➡ Lossless image coding: **PNG**, **TIFF**
  - ➡ Documents: **OpenDocument**, **PDF**
  - ➡ Cryptography: **Crypto++**
  - ➡ ...

LZ77 Variant: Lempel-Ziv-Markov Chain Algorithm (LZMA)

The Concept of LZMA

- Pre-process byte sequence using an **LZ77 variant** (similar to LZSS, but with special cases)
- Entropy coding of resulting bit sequence using a **range encoder** (adaptive binary arithmetic coding)

Improvements over DEFLATE

- Most important: **Context-based adaptive binary arithmetic coding** of bit sequences
- **Larger search buffer** of up to $\mathbf{N = 4\,294\,967\,296}$ (32 bit), look-ahead buffer of $\mathbf{L = 273}$
- **Special codes** for using same distances as for one of the last four matches

Applications of LZMA

- Next generation file compressors

➞ 7zip, xv, lzip, ZIPX

LZMA: Mapping of Byte Sequences to Bit Sequences

Code for single byte sequence (match or literal)

0	+ (byte)	Direct encoding of next byte (no match)
10	$+ \ell + d$	Conventional match (followed by codes for length ℓ and distance d)
1100		Match of length $\ell = 1$, distance d is equal to last used distance
1101	$+ \ell$	Match of length ℓ, distance d is equal to last used distance
1110	$+ \ell$	Match of length ℓ, distance d is equal to second last used distance
11110	$+ \ell$	Match of length ℓ, distance d is equal to third last used distance
11111	$+ \ell$	Match of length ℓ, distance d is equal to fourth last used distance

Code for length ℓ

0 + (3 bits)	Length in range $\ell = 2 \ldots 9$
10 + (3 bits)	Length in range $\ell = 10 \ldots 17$
11 + (8 bits)	Length in range $\ell = 18 \ldots 273$

Code for distance d

- 6 bits for indicating "distance slot"
- followed by 0–30 of bits (depending on slot)

LZMA: Entropy Coding of Bit Sequence after LZ77 Variant

Entropy Coding of Bit Sequences

- Context-based Adaptive Binary Arithmetic Coding (called range encoder)
- Multiple adaptive binary probability models + bypass mode (probability 0.5)
- Sophisticated context modeling: Probability model for next bit is chosen based on ...
  - type of bit, value of preceding byte, preceding bits of current byte,
  - type of preceding byte sequences, ...

Binary Arithmetic Coding Engine

- 11 bits of precision for binary probability masses (only store p_0, since $p_1 = 2^{11} - p_0$)
- 32 bits of precision for interval width
- Probability models are updated according to

$$p_0 = \begin{cases} p_0 + \big((2^{11} - p_0) \gg 5\big) & : \quad \text{bit} = 0 \\ p_0 - (p_0 \gg 5) & : \quad \text{bit} = 1 \end{cases}$$

The Lempel-Ziv 1978 Algorithm (LZ78)

Main Difference to LZ77

- Dictionary is not restricted to preceding N symbols
- Dictionary is constructed during encoding and decoding

The LZ78 Algorithm

- Starts with an empty dictionary
- Next variable-length symbol sequence as coded by tuple $\{k, n\}$
  - k : Index for best match in dictionary (or "0" if no match is found)
  - n : Next symbol (similar to LZ77)
- After coding a tuple $\{k, n\}$, the represented phrase is added to the dictionary

Number of Bits for Dictionary Index

- Number of bits n_k for dictionary index depends in dictionary size

$$n_k = \left\lceil \log_2(1 + \text{dictionary size}) \right\rceil$$

- In practice: Dictionary is reset after it becomes too large

Toy Example: LZ78 Encoding

phrase	output	bits	dictionary
t	(0, t)	8	1 : t
h	(0, h)	9	2 : h
i	(0, i)	10	3 : i
n	(0, n)	10	4 : n
k	(0, k)	11	5 : k
in	(3, n)	11	6 : in
g	(0, g)	11	7 : g
␣	(0, ␣)	11	8 : ␣
th	(1, h)	12	9 : th
ing	(6, g)	12	10 : ing
s	(0, s)	12	11 : s
␣t	(8, t)	12	12 : ␣t
hr	(2, r)	12	13 : hr
o	(0, o)	12	14 : o
u	(0, u)	12	15 : u
gh	(7, h)	12	16 : gh

Message:

thinking␣things␣through

Result:

- Original message: 184 bits (23 bytes)
- Required 177 bits in total

Remember: Number of bits for dictionary index k

$$n_k = \left\lceil \log_2(1 + \text{dictionary size}) \right\rceil$$

Toy Example: LZ78 Decoding

input	phrase	dictionary
(0, t)	t	1 : t
(0, h)	h	2 : h
(0, i)	i	3 : i
(0, n)	n	4 : n
(0, k)	k	5 : k
(3, n)	in	6 : in
(0, g)	g	7 : g
(0, ␣)	␣	8 : ␣
(1, h)	th	9 : th
(6, g)	ing	10 : ing
(0, s)	s	11 : s
(8, t)	␣t	12 : ␣t
(2, r)	hr	13 : hr
(0, o)	o	14 : o
(0, u)	u	15 : u
(7, h)	gh	16 : gh

Decoded Message:

thinking␣things␣through

LZ78 Variant: The Lempel-Ziv-Welch Algorithm (LZW)

Main Difference to LZ78

- Dictionary is initialized with all strings of length one (i.e., all byte codes)
- Next symbol is not included in code

The LZW Algorithm

- Send code for dictionary entry that matches start of remaining sequence
- After sending a code, a new dictionary entry is added that consists of
  - the phrases that was just coded followed by
  - the next symbol in the message

Applications using the LZW Algorithm

- Unix file compression tool **compress**
- Image coding format **GIF**
- Optional compression mode in **PDF** and **TIFF**

Toy Example: LZW Encoding

phrase	next	output	dictionary
t	h	<116>	256: th
h	i	<104>	257: hi
i	n	<105>	258: in
n	k	<110>	259: nk
k	i	<107>	260: ki
in	g	<258>	261: ing
g	␣	<103>	262: g␣
␣	t	<32>	263: ␣t
th	i	<256>	264: thi
ing	s	<261>	265: ings
s	␣	<115>	266: s␣
␣t	h	<263>	267: ␣th
h	r	<104>	268: hr
r	o	<114>	269: ro
o	u	<111>	270: ou
u	g	<117>	271: ug
g	h	<103>	272: gh
h		<104>	273: h

Message:

thinking␣things␣through

Pre-initialized dictionary:

- All byte codes: <0> ... <255>

Result:

- Original message: 184 bits (23 bytes)
- Required 162 bits (18×9 bits)

Toy Example: LZW Decoding

input	output	dictionary	conjecture
<116>	t		256: t?
<104>	h	256: th	257: h?
<105>	i	257: hi	258: i?
<110>	n	258: in	259: n?
<107>	k	259: nk	260: k?
<258>	in	260: ki	261: in?
<103>	g	261: ing	262: g?
<32>	␣	262: g␣	263: ␣?
<256>	th	263: ␣t	264: th?
<261>	ing	264: thi	265: ing?
<115>	s	265: ings	266: s?
<263>	␣t	266: s␣	267: ␣t?
<104>	h	267: ␣th	268: h?
<114>	r	268: hr	269: r?
<111>	o	269: ro	270: o?
<117>	u	270: ou	271: u?
<103>	g	271: ug	272: g?
<104>	h	272: gh	273: h?

Message:

thinking␣things␣through

Pre-initialized dictionary:

- All byte codes: <0> ... <255>

LZW: The K-Omega-K Problem

Property of LZW Algorithm

- Decoder is one step behind encoder in constructing dictionary
- Encoder might send code for not yet completed dictionary entry

Example: Coding of sequence "...cXYZcXYZca..."

encoder

phrase	next	output	dictionary
			<300>: cXYZ
cXYZ	c	<300>	<400>: cXYZc
cXYZc	a	<400>	<401>: cXYZca

decoder

input	output	dictionary	conjecture
		<300>: cXYZ	
<300>	cXYZ		<400>: cXYZ?
<400>	cXYZ?	(cXYZ? must be cXYZc)	

How can the decoder correctly decode in such a case ?

- Incomplete dictionary entry is last added entry
- This entry is used only if the first symbol of new sequence is the last symbol of incomplete entry

➡ Last symbol must be equal to first symbol ! (in our example: "cXYZ?" = "cXYZc")

The Burrows-Wheeler Transform (BWT)

1. Create all rotations of the original message
2. Sort all rotations in lexicographical order
3. Output: Last column of the sorted block + index of original message (in sorted block)

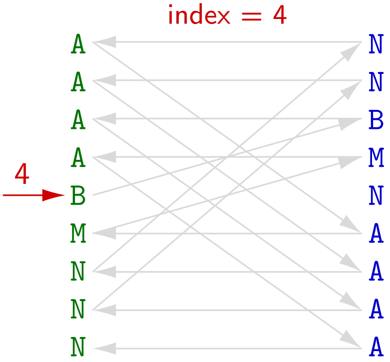
Example: Message "BANANAMAN"

rotations →	sorting →
BANANAMAN	AMANBANAN
ANANAMANB	ANAMANBAN
NANAMANBA	ANANAMANB
ANAMANBAN	ANBANANAM
NAMANBANA	BANANAMAN
AMANBANAN	MANBANANA
MANBANANA	NAMANBANA
ANBANANAM	NANAMANBA
NBANANAMA	NBANANAMA

last column → NNBMNAAAA

index → index = 4

BWT: The Inverse Transform (Can we reconstruct the original message?)



decoded message:

B A N A N A M A N

Given:

- Last column of sorted block "N N B M N A A A A"
- Index of original message in sorted block (4)

Decoding procedure

1. Create first column of sorted block (by sorting)
2. First symbol is given at transmitted index
3. Next symbol is obtained by
   - a) Look for corresponding symbol in last column (i.e., same count of same letter)
   - b) Next symbol is at same position in first column (since following symbol is in first column)
4. Continue procedure until all letters are decoded

BWT: Why Is It Useful for Compression ?

```
A M A N B A N A N
A N A M A N B A N
A N A N A M A N B
A N B A N A N A M
B A N A N A M A N
M A N B A N A N A
N A M A N B A N A
N A N A M A N B A
N B A N A N A M A
```

Property of BTW (for large blocks)

- Symbols on left side of sorted block are *contexts* (symbols that follow last column in message)
- Block lines are sorted according to the contexts
- Likely that same symbol (last column) precedes same context (source with memory: conditional pmf with high peak)

➡ **Last column contains long sequences of identical symbols**

How to exploit this property ?

- In following processing steps
- Example: Move-to-front transform (MTF)

The Move-To-Front Transform (MTF)

MTF: Map Symbols Sequences to Sequence of Unsigned Integers

1. Replace next symbol with its alphabet index
2. Update alphabet $\mathcal{A}$ by moving symbol to the front

Example: Sequence "NNBMNAAAA" (result of BWT for "BANANAMAN")

N N B M N A A A A	13	$\mathcal{A} = \{$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$\}$
N **N** B M N A A A A	0	$\mathcal{A} = \{$N A B C D E F G H I J K L M O P Q R S T U V W X Y Z$\}$
N N **B** M N A A A A	2	$\mathcal{A} = \{$N A B C D E F G H I J K L M O P Q R S T U V W X Y Z$\}$
N N B **M** N A A A A	13	$\mathcal{A} = \{$B N A C D E F G H I J K L M O P Q R S T U V W X Y Z$\}$
N N B M **N** A A A A	2	$\mathcal{A} = \{$M B N A C D E F G H I J K L O P Q R S T U V W X Y Z$\}$
N N B M N **A** A A A	3	$\mathcal{A} = \{$N M B A C D E F G H I J K L O P Q R S T U V W X Y Z$\}$
N N B M N A **A** A A	0	$\mathcal{A} = \{$A N M B C D E F G H I J K L O P Q R S T U V W X Y Z$\}$
N N B M N A A **A** A	0	$\mathcal{A} = \{$A N M B C D E F G H I J K L O P Q R S T U V W X Y Z$\}$
N N B M N A A A **A**	0	$\mathcal{A} = \{$A N M B C D E F G H I J K L O P Q R S T U V W X Y Z$\}$

➡ **Effect**: Many small values for sequences with long repetitions (e.g., results of a BWT)

File Compression Utility BZIP2

Main Components for Compression

- Run-length encoding of input data (special V2V code)
- Block-wise **Burrows-Wheeler Transform** (BWT)
- **Move-To-Front Transform** (MTF) of BWT result
- Run-length encoding of MTF result
- Dynamic **Huffman coding**

Some more details

- Block size for BWT/MTF of up to 900 kBytes
- Smart coding of Huffman tables
- Up to 6 Huffman tables per block
- Adaptive selection between Huffman tables (every 50 symbols)

Universal File Compressors

Marginal Huffman Coding

➡ Very old Unix utility **pack**

Lempel-Ziv-Welch (LZW) Algorithm

➡ Old Unix utility **compress**

DEFLATE: Lempel-Ziv-Storer-Szymanski (LZSS) + Huffman Coding

➡ File compressors **ZIP**, **gzip**, **PKZIP**, **Zopfli**, **CAB**

Lempel-Ziv-Markov-Chain (LZMA) with binary arithmetic coding

➡ File compressors **7zip**, **xv**, **lzip**

Block Sorting: Burrows-Wheeler & Move-To-Front Transform

➡ File compressor **bzip2**

Lossless Audio Coding: Free Lossless Audio Codec (FLAC)

Basic Source Codec

1. Decompose audio file into variable-size blocks
   - ➞ Block sizes determines capability for adaptation to signal statistics
2. Inter-channel decorrelation (invertible)
   - For example: Stereo is coded as $\text{mid} = (\text{left} + \text{right})/2$
     $\text{side} = (\text{left} - \text{right})$
3. Linear prediction (4 types)
   - a) No prediction
   - b) Prediction by a constant value
   - c) Prediction using pre-defined linear predictor (order 1 to 4)
   - d) Prediction using adaptive linear predictor (up to order 32)
4. Entropy coding of prediction error samples
   - Rice coding with adaptive Rice parameter selection

Lossless Image Coding: Portable Network Graphics (PNG)

Basic Source Codec

1. Separate Coding of Individual Color Planes

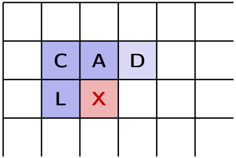
2. Prediction of Image Samples
   - ➡ Predictor is selected per image row
   - ➡ Five predictors are pre-defined (no adaptive prediction coefficients)

0	none	direct coding of image samples
1	left	prediction using left sample
2	above	prediction using above sample
3	average	prediction using rounded average of left and above sample
4	Paeth	non-linear prediction using left, above, and corner sample (**most often use**)

3. Entropy Coding of Prediction Error Samples
   - DEFLATE algorithm:
   - ➡ Lempel-Ziv-Storer-Szymanski (LZSS) algorithm for dependency removal
   - ➡ Huffman coding of LZSS output (adaptive Huffman tables)

Lossless Image Coding: JPEG-LS (Joint Photographic Experts Group)

Basic Source Codec



1. First prediction stage: LOCO Predictor

$$\hat{X} = \begin{cases} \min(L, A) & : \; C \geq \max(L, A) \\ \max(L, A) & : \; C \leq \min(L, A) \\ L + A - C & : \; \text{otherwise} \end{cases}$$

2. Second order prediction using conditional mean $\mathrm{E}\{\, x \,|\, g_1, g_2, g_3 \,\}$
   - Given by clipped gradients (365 contexts after merging contexts with positive and negative signs)

$$\begin{aligned} g_1 &= \max(-4, \min(4, D - A)) \\ g_2 &= \max(-4, \min(4, A - C)) \\ g_3 &= \max(-4, \min(4, C - L)) \end{aligned}$$

3. Entropy Coding of Prediction Error Samples
   - Rice codes
   - Optional: Run-length coding (for uniform areas)

Comparison: Universal vs Specialized Compressors

compression		text  compression factor	images  compression factor	audio  compression factor
gzip	(DEFLATE)	2.60	1.20	1.09
lzip	(LZMA)	3.53	1.41	1.17
bzip2	(BWT+MTF)	3.55	1.39	1.15
PNG	(prediction)		1.62	
FLAC	(prediction)			1.82

➡ **Specialized Compressors achieve Higher Coding Efficiency**

Summary of Lecture

Dictionary-based Coding

- Lempel-Ziv 1977 and 1978 algorithms (LZ77, LZ78): Basis for many universal compressors
- Lempel-Ziv-Storer-Szymanski (LZSS): Variant of LZ77
- Lempel-Ziv-Welch (LZW): Variant of LZ78
- DEFLATE: Combining LZSS with Huffman Coding
- Lempel-Ziv-Markov Chain Algorithm (LZMA): LZ78 Variant with Binary Arithmetic Coding

Lossless Coding using Block Sorting

- Burrows-Wheeler Transform (BWT)
- Move-To-Front Transform (MFT)

Lossless Compression Applications

- Universal File Compression: compress, gzip, bezip2, lzip
- Lossless Audio Coding: FLAC
- Lossless Image Coding: PNG, JPEG-LS

Exercise: Lossless Image Compression Challenge (Part II)

Improve your codec for lossless coding of 8-bit color images

- Try different things discussed in lectures and exercises

The following might be worth trying

- Prediction
  - Simple prediction using left sample
  - Fixed non-linear predictor like LOCO or Paeth predictor
  - Line- or block-adaptive selection of predictor (e.g., between horizontal, vertical, ...)
- Entropy Coding of Prediction Errors
  - Simple Rice codes (may be with adaptive Rice parameter)
  - Arithmetic coding with adaptive marginal pmf
  - Arithmetic coding with conditional pmf (very simple conditions)

Measure and provide the compressed file sizes for the Kodak test set!