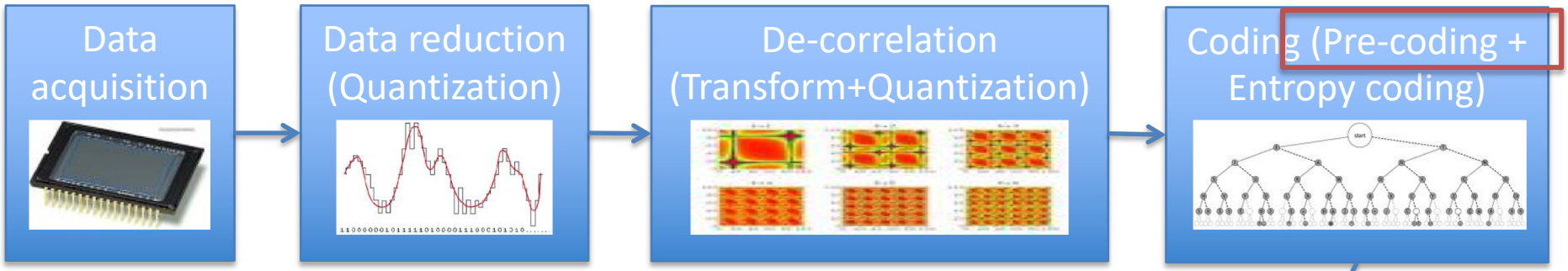
Image Data Compression

Pre-Coding techniques

Image signal path: overview



Output data:
Continuous multi-dimensional analog signal

Compression:
Physical sensor limitations, DAQ settings

Output data:
Correlated fully digital multi-dimensional symbol stream

Compression:
Spatio-temporal quantization, value quantization

Output data:
Linear stream of de-correlated symbols

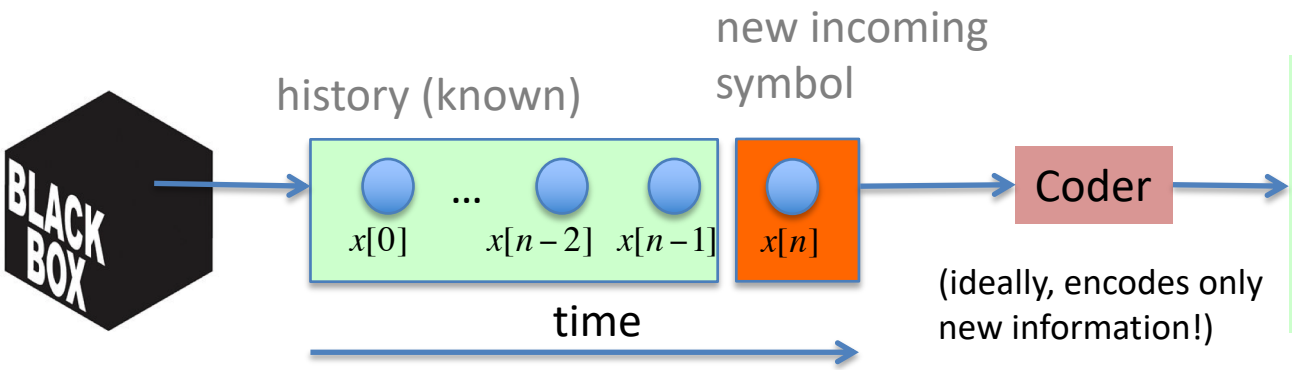
Compression:
Truncation of irrelevant signal parts, value quantization

Output data:
Linear stream of bits

Compression:
Reversible suppression of the redundant information

...

General problem of coding - review



- **Coding:** reversibly transforming the input stream of symbols to output stream of symbols
- **Goal:** reduce **redundancy** - information that can be recovered without being encoded
- Theoretical approach: represent input stream as a **stochastic process**, probabilistically modeled as a **Markov process** of some order
- **0-th order Markov model**: relatively simple and efficient (e.g. binary) codes
- Coding with a **1-st order Markov model (entropy coding)**: solved problem
- Coding with an **M-th order model**: becomes exponentially complex, not always useful (practical solution: various data-dependent pre-processing transforms, aka pre-coding)
- **General models**: cannot be optimally coded! (incomputable problem, AI and art)
- Key to efficient coding: <u>understanding your data</u>

Coding: overview

Coding finite strings

Entropy Coding

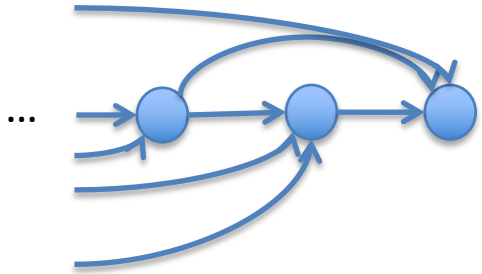
- 1-st order Markov process
- Efficient codes for arbitrary distributions: Huffman, Arithmetic, Golomb-Rice, …

$$p_i^{(n)} = p_i$$

Pre-Coding

- Higher- or undefined-order Markov process
- Practical ad hoc solutions: Run-Length coding, Phrase coding, Block-Sorting, …

$$p_i^{(n)} = p(x[n] = s_i \mid x[n-1],..,x[0])$$



Some [successful] pre-coding techniques

- **Run-Length coding** (text strings, sequences of numbers, …)

Replace a sequence of identical symbols with a new special symbol

- **Phrase Coding**

Replace arbitrary sub-strings with special symbols from another alphabet

- **Block-Sorting**

Increase correlations of adjacent symbols by re-arranging input symbols

- **Max-Value, Bit-Marking, Quadtree-Coding** etc. (2D data: images, matrices, …)

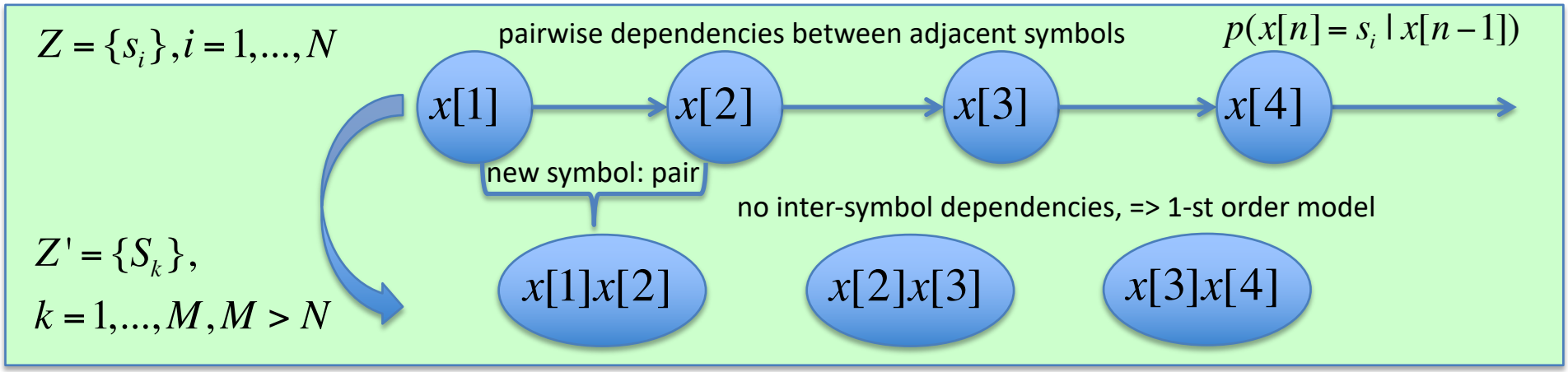
Use additional local information to simplify subsequent processing

- **(?)**

No definitive choice suitable for all situations; only “ad-hoc” solutions.
Efficiency evaluated empirically with some “standard” test datasets.

M-th order models: coding M-grams

Example: reducing a 2-nd order model to a 1-st order model with new alphabet (“digrams”)



Usefulness:

Original alphabet, Huffman code:

s_i	$p(s_i)$	code
a	0.3	0
b	0.7	1

$$H(Z) = 0.881[bits/symbol]$$
$$S(Z) = 1.0[bits/symbol]$$

Hard limit on prefix code length…

Di-gram alphabet (assume independence, Huffman coding):

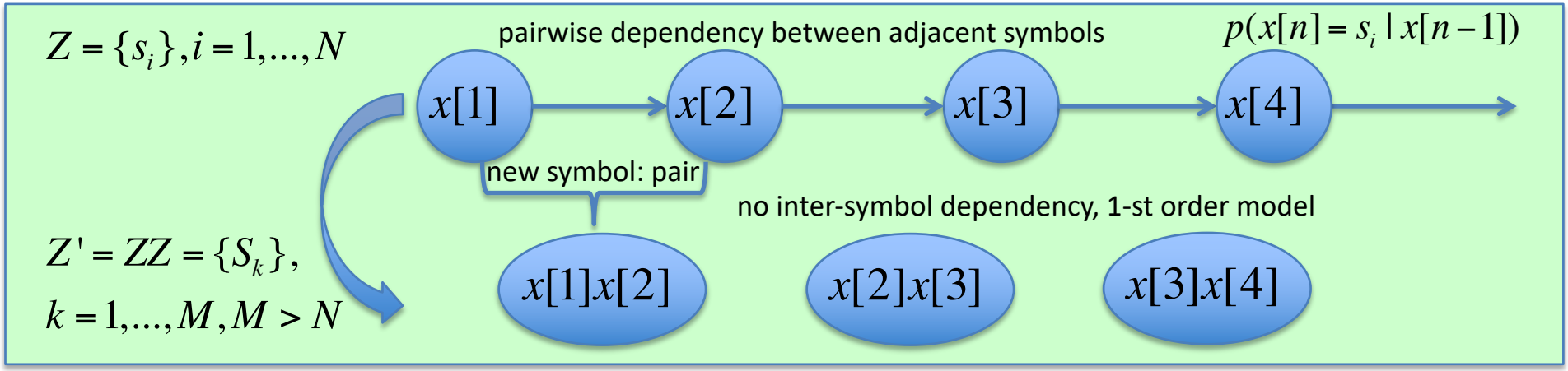
S_i	$p(S_i)$	code
aa	0.3 * 0.3 = 0.09	000
ab	0.3 * 0.7 = 0.21	001
ba	0.7 * 0.3 = 0.21	01
bb	0.7 * 0.7 = 0.49	1

Lower redundancy, length limit overridden!

$$H(Z') = 1.762[bits/double\_symbol] = 0.881[bits/symbol]$$
$$S(Z') = 1.81[bits/double\_symbol] = 0.905[bits/symbol]$$

Conditional and joint entropy

Recall example: reducing 2-nd order model to 1-st order model with new alphabet



Source entropy of the new alphabet Z' = ZZ (aka Joint Entropy):

$$H(ZZ) = -\sum_{k=1}^{M} p(S_k)\log_2(p(S_k))$$

$$= -\sum_{i=1}^{N}\sum_{j=1}^{N} p(s_i s_j)\log_2(p(s_i s_j))$$

$$= -\sum_{j=1}^{N} p(s_j)\log_2(p(s_j))\left(\sum_{i=1}^{N} p(s_i \mid s_j)\right) - \sum_{j=1}^{N} p(s_j)\left[\sum_{i=1}^{N} p(s_i \mid s_j)\log_2(p(s_i \mid s_j))\right]$$

$$= H_{src}(Z) + \sum_{j=1}^{N} p(s_j) H_{src}(Z \mid s_j) = H_{src}(Z) + H_{cond}(Z \mid Z)$$

$$p(s_i s_j) = p(s_i \mid s_j)p(s_j)$$

$$\log(ab) = \log(a) + \log(b)$$

$$\sum_{i=1}^{N} p(s_i \mid s_j) = 1$$

Conditional entropy

Conditional and joint entropy

$$H(ZZ) = H(Z) + H(Z \mid Z)$$

Joint entropy
Unit: [bits/digram]
Meaning: entropy of a source emitting di-grams (new alphabet of digrams)

Source entropy
Unit: [bits/symbol]
Meaning: entropy of a source emitting independent symbols (original alphabet)

Conditional entropy
Unit: [bits/symbol]
Meaning: average entropy of conditional (sub)sources (original alphabet, new source)

$$H(Z \mid Z) = -\sum_{j=1}^{N} p(s_j) \left[\sum_{i=1}^{N} p(s_i \mid s_j) \log_2 (p(s_i \mid s_j)) \right] = \sum_{j=1}^{N} p(s_j) H_{src}(Z \mid x[n-1] = s_j)$$

- Joint entropy is always larger than the source entropy. However, when translated to the same [bits/symbol] units, it is always smaller than the source entropy due to additional (inter-symbol) redundancy (ignored by 1-st order models).
- Conditional entropy measures information contained in inter-symbol correlations.
- Can be represented as an entropy of a source switching between N contexts as in adaptive entropy coding based on the previous symbol value.

Conditional and joint entropies: examples

1. Completely independent symbols, $p(s_i \mid s_j) = p(s_i)$ (i.e. s_i is independent of s_j):

$$H(Z \mid Z) = -\sum_{j=1}^{N} p(s_j)\left[\sum_{i=1}^{N} p(s_i)\log_2(p(s_i))\right] = -\sum_{i=1}^{N} p(s_i)\log_2(p(s_i)) = H(Z)$$

$$H(ZZ)/[bits/symbol] = \frac{1}{2}H(ZZ)/[bits/double\_symbol] = H(Z)/[bits/symbol]$$

2. Deterministically dependent symbols, $p(s_i \mid s_j) = \delta^i_{f(j)}$ (i.e. s_j is always followed by $s_{f(j)}$):

$$H(Z \mid Z) = -\sum_{j=1}^{N} p(s_j)\cdot 0 = 0$$

$$H(ZZ)/[bits/symbol] = \frac{1}{2}H(Z)/[bits/symbol]$$

Conditional and joint entropy

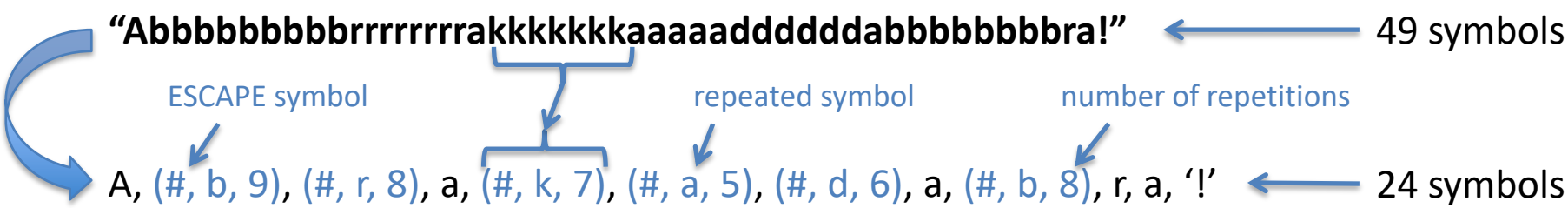
Boundaries on the digram joint entropy: $\frac{1}{2}H(Z) \leq H(ZZ) \leq H(Z)$

Generalization: M-symbol joint entropy: $\frac{1}{M}H(Z) \leq H(\underbrace{ZZ...ZZ}_{M}) \leq H(Z)$

- Practical recipe: apply a lossless transform of the input string to exploit higher-order correlations, followed by some entropy coding step.
- Using inter-symbol redundancy, we may reduce the redundancy beyond the entropy coding limitations!
- Maintaining true M-th order models quickly becomes expensive.
- All methods discussed further make very strong simplifying assumptions about possible types of inter-symbol dependencies.

- Assumed models may be **very** inefficient in wrong situations!

Run-length coding



- It makes sense to only encode sequences of length $n > 3$
- To slightly simplify coding of a sequence of length n, encode $(n - 4)$ instead of n
- When alphabet has no extra ESCAPE-symbol, use “aaaa” as ESCAPE (such sequences in the input string must all have been replaced, so they cannot appear at the output); efficiency for a few repetitions of “a” drops even further
- Example: numerical signals with **long sequences of zeros** (e.g. DCT coefficients in JPEG):

00001 000004 003 5 00002 0000000 07 …
41 54 23 05 42 70 17 …

Here: number of zeros is encoded with 3 bits (i.e. max value is 7). Otherwise, treat pair as a new symbol

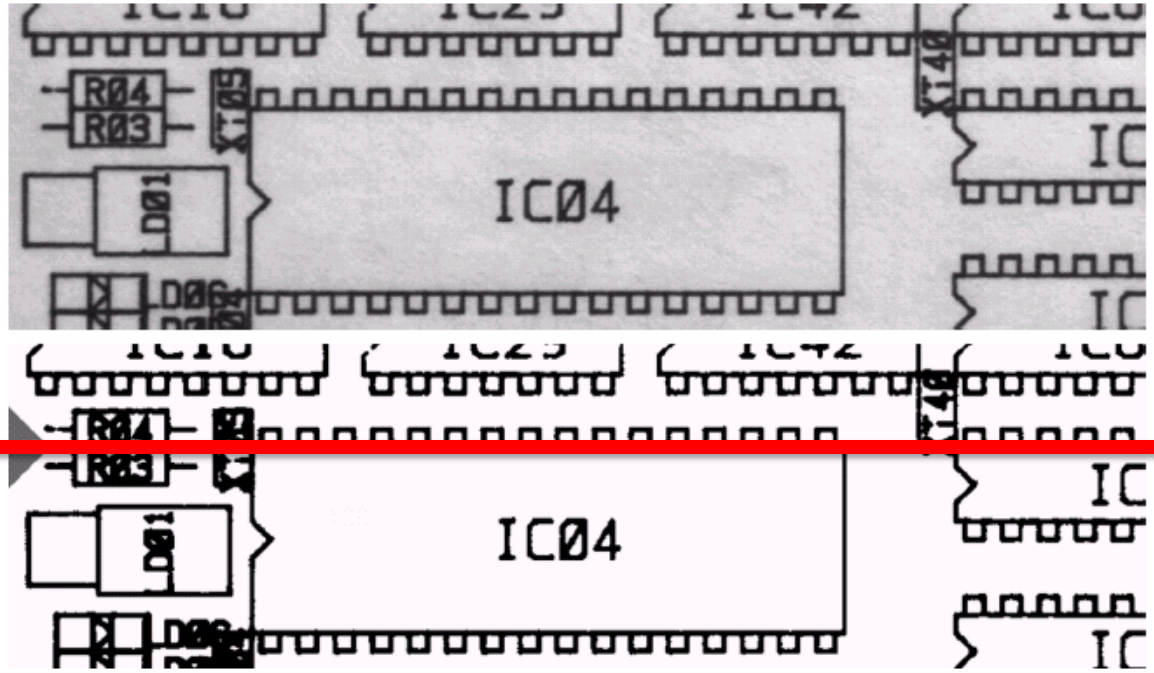
- **Binary signals:** only need to code the number of consecutive **1**-s or **0**-s:

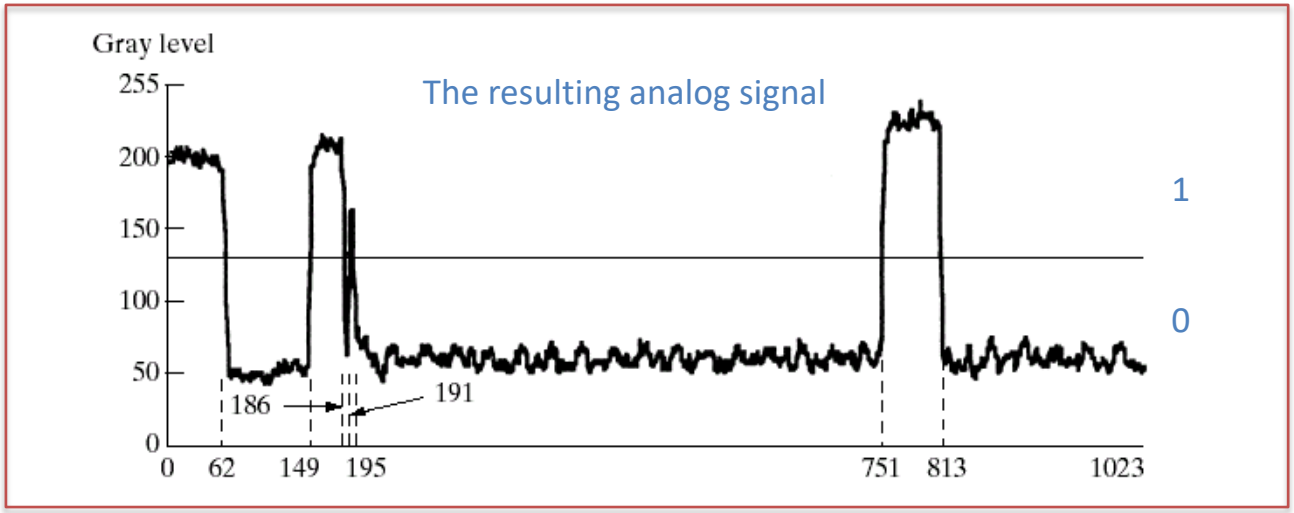
11 0000 11 00000 111111 00 1111111 00 …
02 4 2 5 6 2 7 2 …

Whether this leads to compression or size growth, depends on signal statistics

- Used to transmit black/white pixels in fax protocols (with Huffman-coded length values)

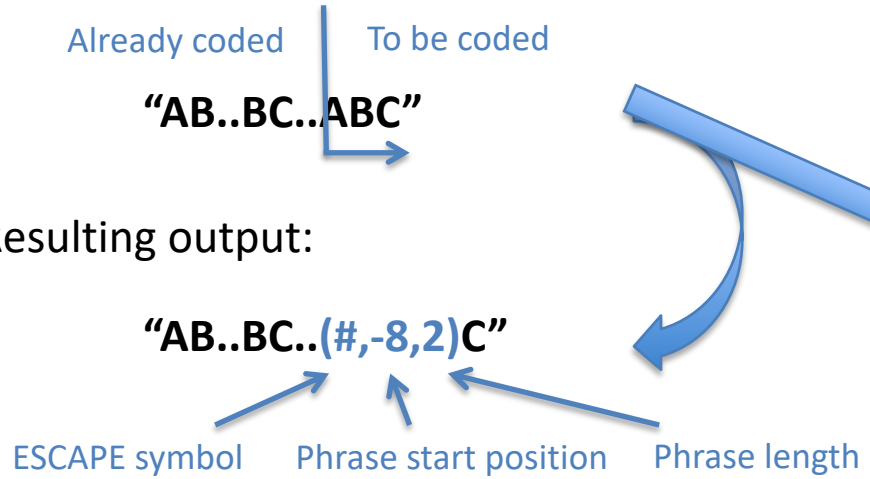
Run-length coding: application to BW-image encoding





Phrase coding (dictionary-based compression)

Extend RLE to previously encoded phrases:



Very simple and fast decoder, but a complex and slow encoder.

Alternative encodings:

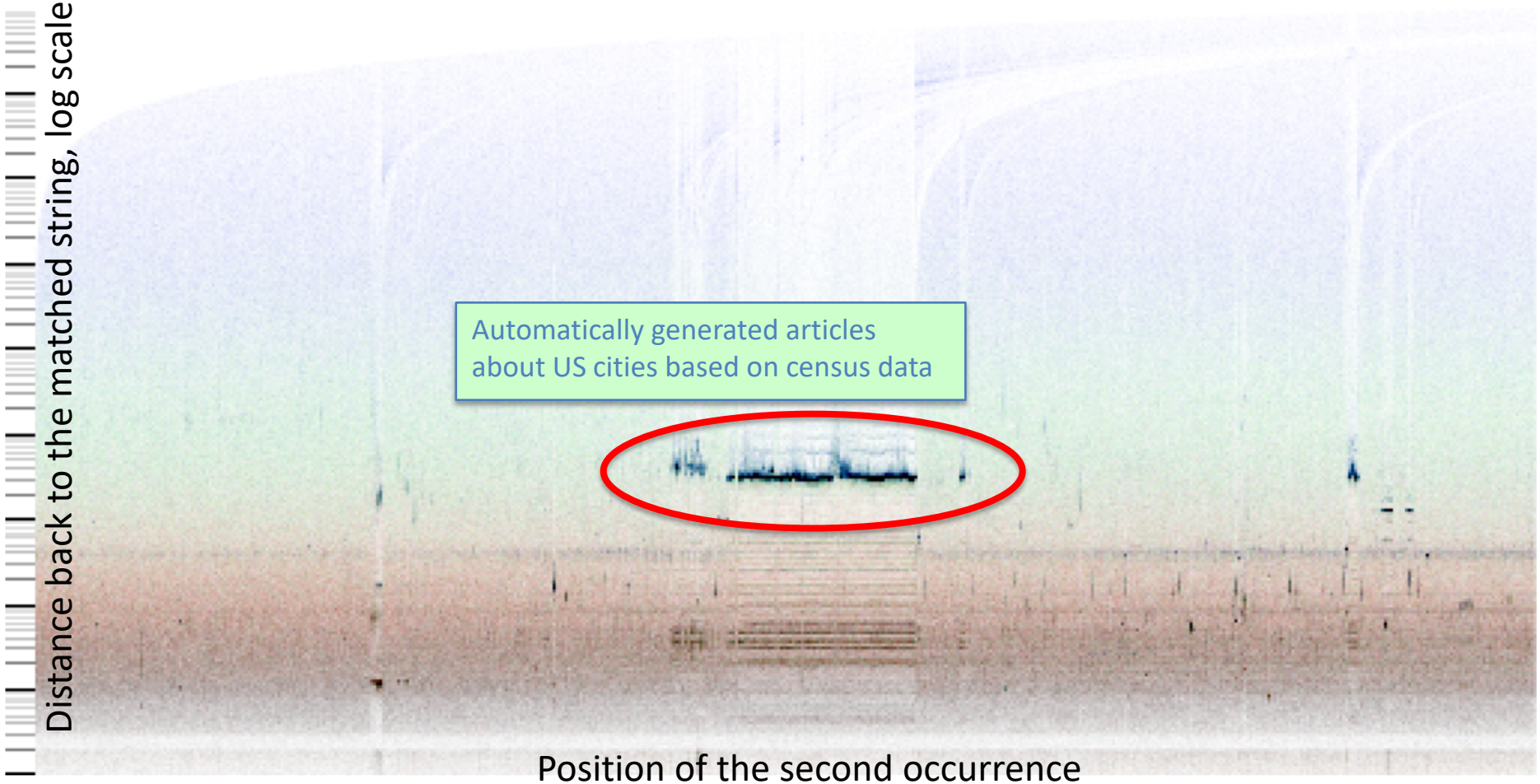
“AB..BC..A(#,-5,2)”

“AB..BC..ABC”

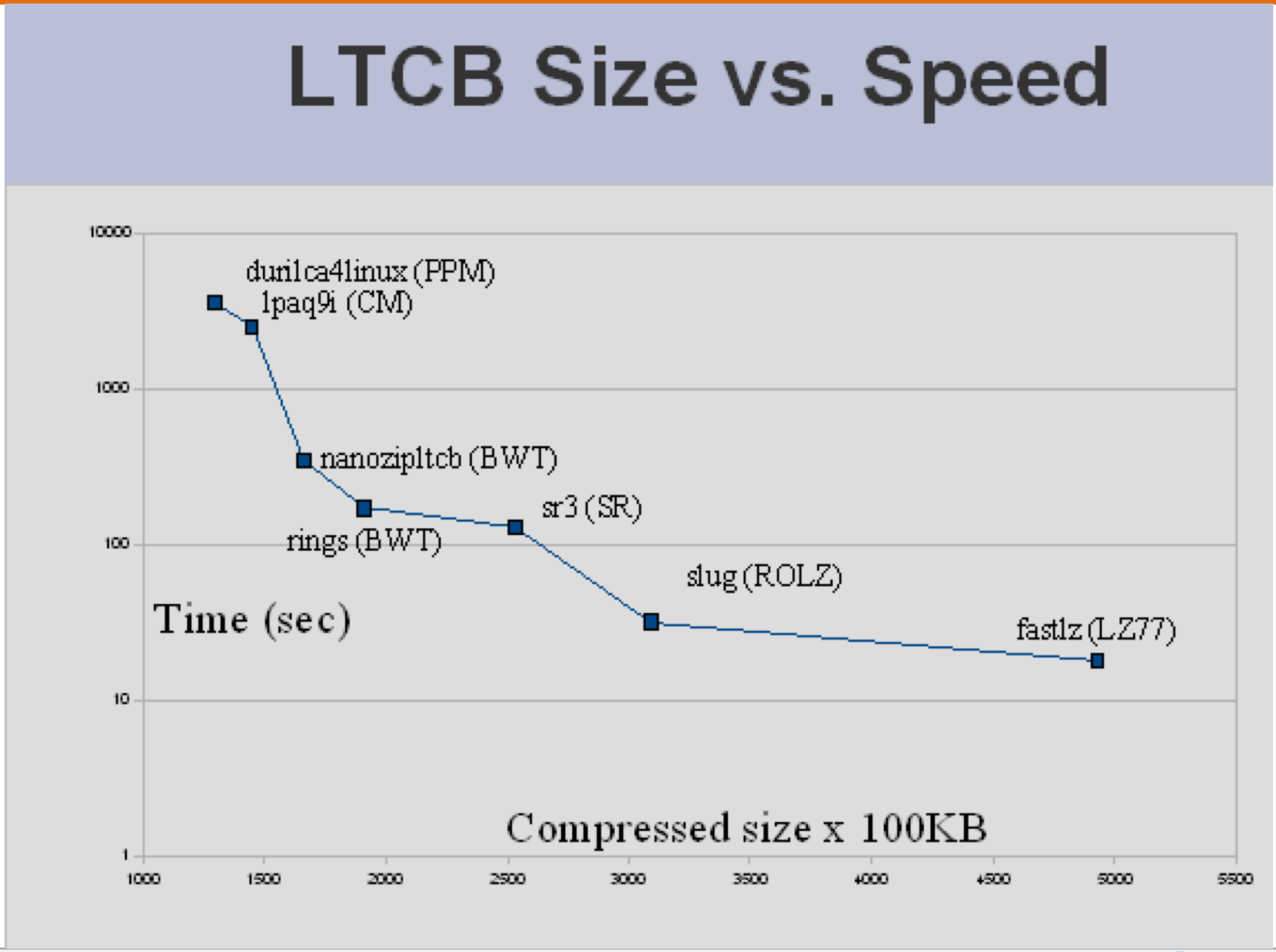
- [Lempel, Ziv 1977]: original publication (**LZ77**), later: **LZ78** – two families of algorithms
- [Welch, 1984]: **LZW** (from **LZ77**), in Unix “compress” utility: dictionary-based
- **DEFLATE** (from **LZ78** family): Unix “**gzip**” utility.
- 2004: IEEE named **LZ** data compression algorithm one of the Web-enabling milestones
- Same decoding, different encoding methods: still very active field of research!
- Current “holy grail” of research: efficient parallel encoding (parallel indexing of very large datasets, stochastic search for matching sub-strings, fast hashing strategies, …)

Large Text Compression Benchmark: “repetitiveness” of text

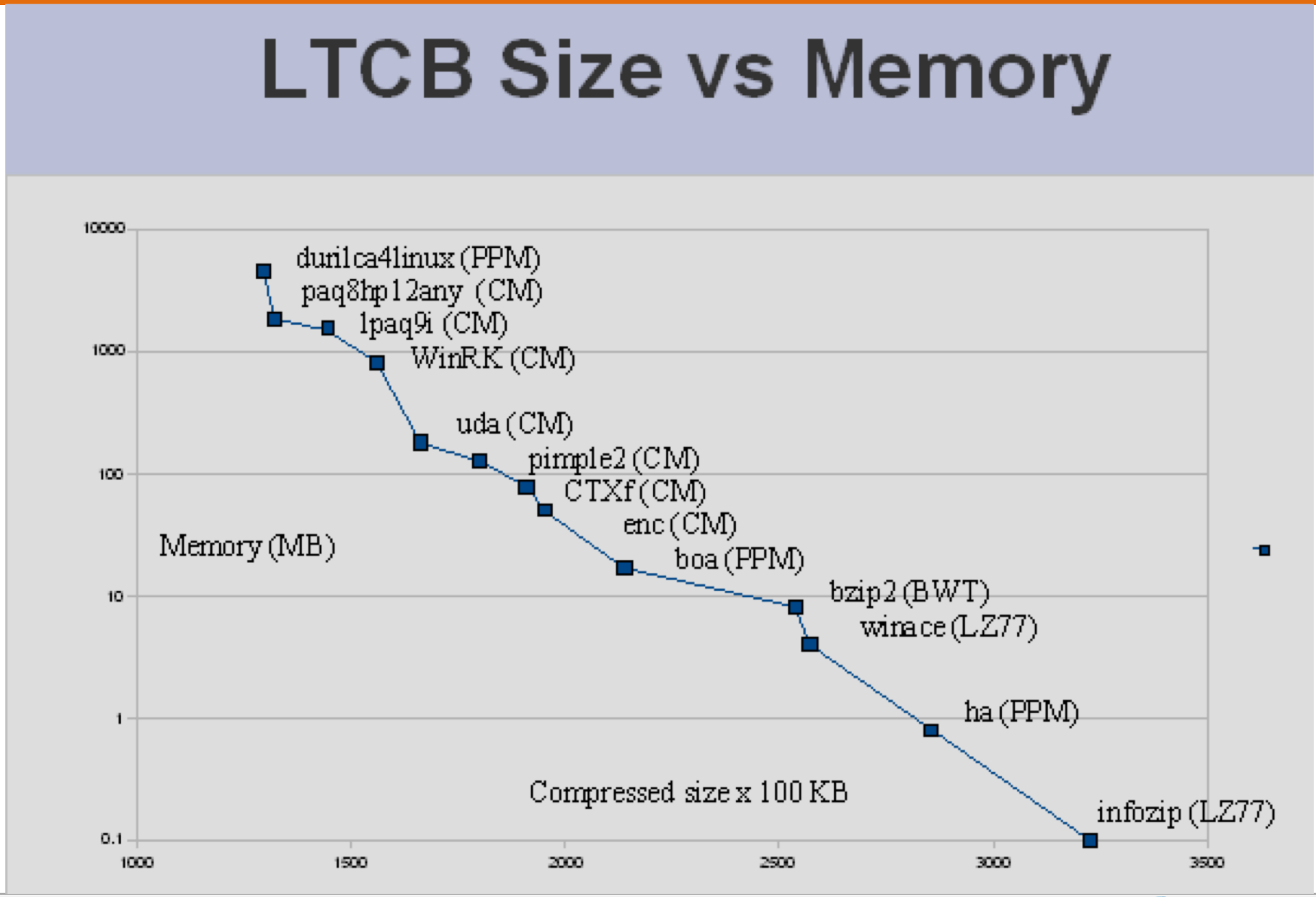
- Dump of Wikipedia text from Mar 3rd, 2006, truncated to 10^9 bytes
- Each point is a string match between the subsequent occurrences of a string
- Color: length of match: black for 1 byte, red for 2, green for 4, blue for 8 bytes



Pareto frontier: LTCB, as of 2008



Pareto frontier: LTCB, as of 2008



Block-sorting (Burrows-Wheeler Transform)

- [Burrows, Wheeler 1994]: used in **bzip2**, image codecs, …
- Novel application: DNA sequencing (i.e. reconstruction of DNA from many pieces)
- Reversible re-arrangement (permutation) of the input string
- In general, produces long streaks of common symbols that are easier to compress than the original string (if the original has certain long-distance correlations)

Encoding:

Input string	All string rotations	Lexicographically ordered (from left)
^BANANA|	^BANANA| |^BANANA A|^BANAN NA|^BANA ANA|^BAN NANA|^BA ANANA|^B BANANA|^	ANANA|^B ANA|^BAN A|^BANAN BANANA|^ NANA|^BA NA|^BANA ^BANANA| |^BANANA

Per se, no size reduction

But: BWT output may be better suitable for e.g. run-length compression

Additional EOF symbol

Output: **BNN^AA|A**

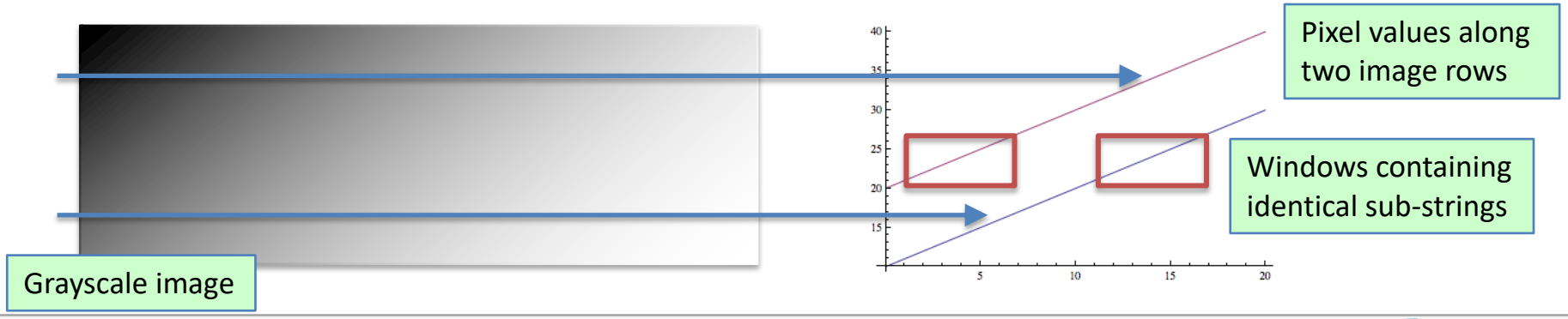
Block-sorting (Burrows-Wheeler Transform)

Decoding: repeated Add (A) and Sort (S) steps

A1	S1	A2	S2	A3	S3	…	A8	S8
B	A	BA	AN	BAN	ANA		BANANA|^	ANANA|^B
N	A	NA	AN	NAN	ANA		NANA|^BA	ANA|^BAN
N	A	NA	A|	NA|	A|^		NA|^BANA	A|^BANAN
^	B	^B	BA	^BA	BAN		^BANANA|	BANANA|^
A	N	AN	NA	ANA	NAN		ANANA|^B	NANA|^BA
A	N	AN	NA	ANA	NA|		ANA|^BAN	NA|^BANA
|	^	|^	^B	|^B	^BA		|^BANANA	^BANANA|
A	|	A|	|^	A|^	|^B		A|^BANAN	|^BANANA

Output: the entry ending with EOF

- There exist linear-time algorithms, which output symbols sequentially
- Well-suited for image compression, encodes similar sequences of gray values:



Bit-marking

- Model: one symbol appears very often, but not in continuous streaks (bad for RLE)

Input signal: **"0 0 7 0 1 0 0 6 0 4 3 0 5 0 0 2 0 2 5 3"** ← 20 * 3 = 60 bits

Output:

Bit mask: **0 0 1 0 1 0 0 1 0 1 1 0 1 0 0 1 0 1 1 1**

Signal: **7 1 6 4 3 5 2 2 5 3** ← 20 + 10 * 3 = 50 bits

- If the alphabet size is **K**, and the probability of a frequent symbol (e.g., **0**) is **p**:

Original average codeword length (assume binary coding):

$$\lceil \log_2 K \rceil > 1 + (1-p) \cdot \lceil \log_2 K \rceil,$$

$$\Rightarrow 1 / \lceil \log_2 K \rceil < p$$

Average codeword length for bit marking: need 1 bit in mask + either 0 or the original length per symbol

- Example: with 256 signal values, need **p > 0.125** for bit-marking to be useful
- Binary symbols cannot be compressed with bit-marking
- Used in several video encoding standards (applied to frame blocks rather than individual pixels, mark blocks carrying no significant information)

Quadtree coding: bit-marking of zero regions in 2D matrices

0	0	0	0	1	2	3	4
0	0	1	2	2	3	4	5
0	1	2	3	3	4	5	6
0	2	3	4	4	5	6	7
0	3	4	5	0	0	0	0
0	4	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Original image values

8 x 8 x 3 = 192 bits

Mask values for 2x2 blocks

0	1	1	1
1	1	1	1
1	1	0	0
0	0	0	0

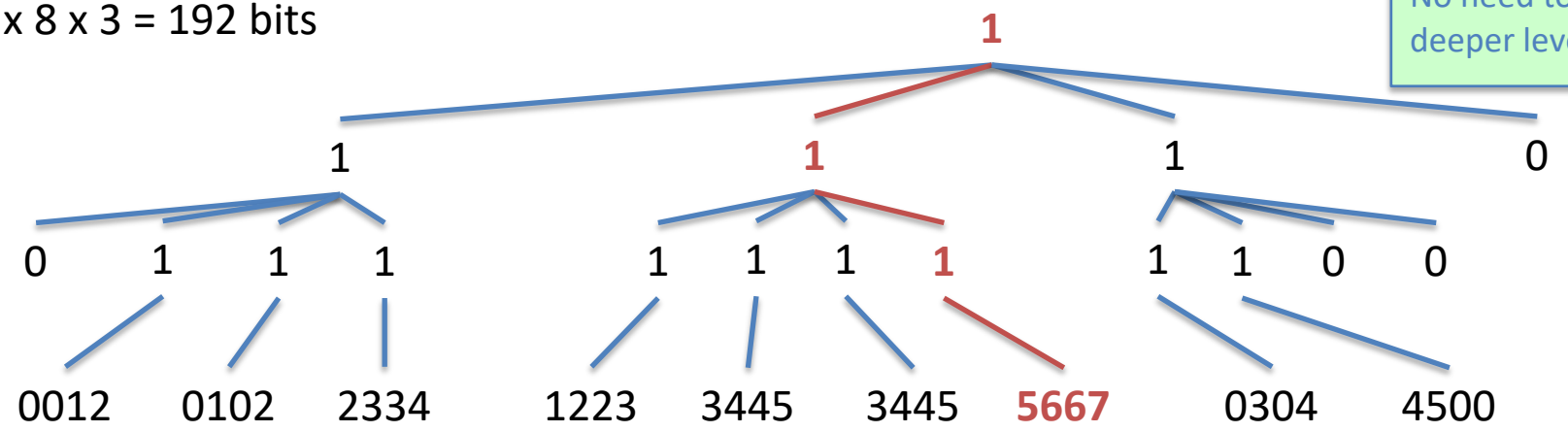
Mask values for 4x4 blocks

1	1
1	0

1

Mask for 8x8 block

No need to keep deeper levels!



(1 + 4 + 12) + 9 x 4 x 3 = 17 + 125 = 142 bits

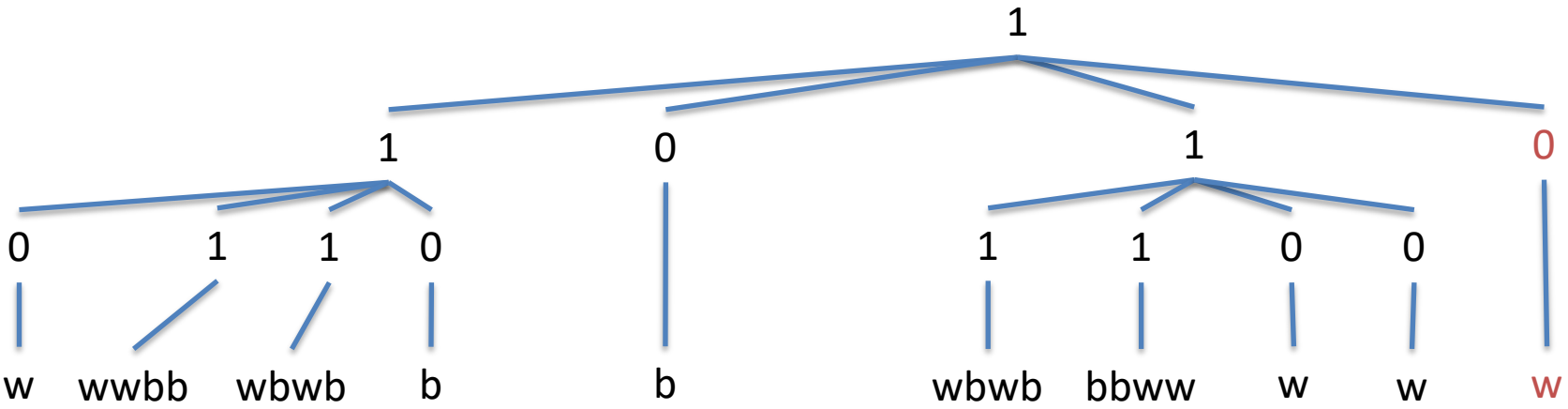
Quadtree coding: marking of uniform areas in binary images

w	w	w	w	b	b	b	b
w	w	b	b	b	b	b	b
w	b	b	b	b	b	b	b
w	b	b	b	b	b	b	b
w	b	b	b	w	w	w	w
w	b	w	w	w	w	w	w
w	w	w	w	w	w	w	w
w	w	w	w	w	w	w	w

0	1	0	0
1	0	0	0
1	1	0	0
0	0	0	0

1	0
1	0

1



QTC+quantization: “nearly-homogeneous” areas in photos

6	7	7	3	0	0	1	1
7	7	5	3	2	1	2	1
6	2	2	2	1	1	2	1
4	2	3	2	1	2	2	2
4	5	6	4	2	2	2	2
7	6	7	5	3	3	3	2
7	6	6	4	3	4	3	2
6	6	5	5	4	4	3	2

7	7	7	3	1	1	1	1
7	7	5	3	1	1	1	1
6	2	2	2	1	1	1	1
4	2	2	2	1	1	1	1
4	5	6	4	3	3	3	3
7	6	7	5	3	3	3	3
6	6	5	5	3	3	3	3
6	6	5	5	3	3	3	3

Criterion of homogeneity: if **(max – min) < 3**,
replace with a “representative value” (e.g. mean):