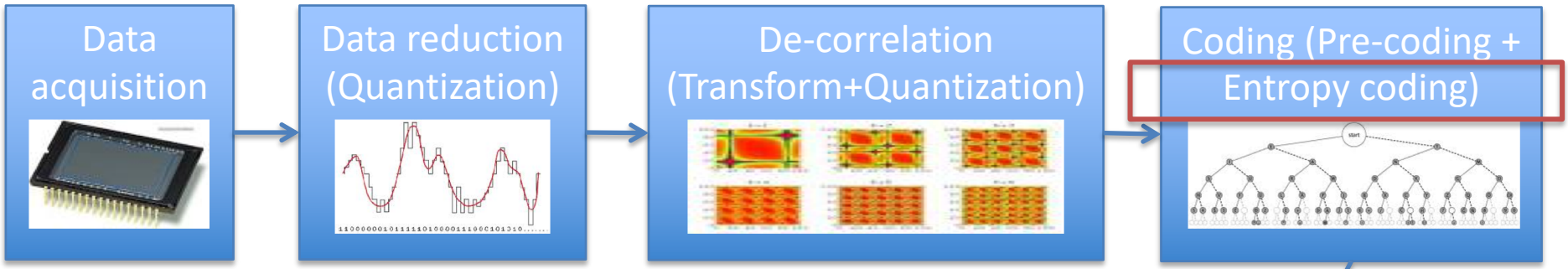
Image Data Compression

Entropy coding

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

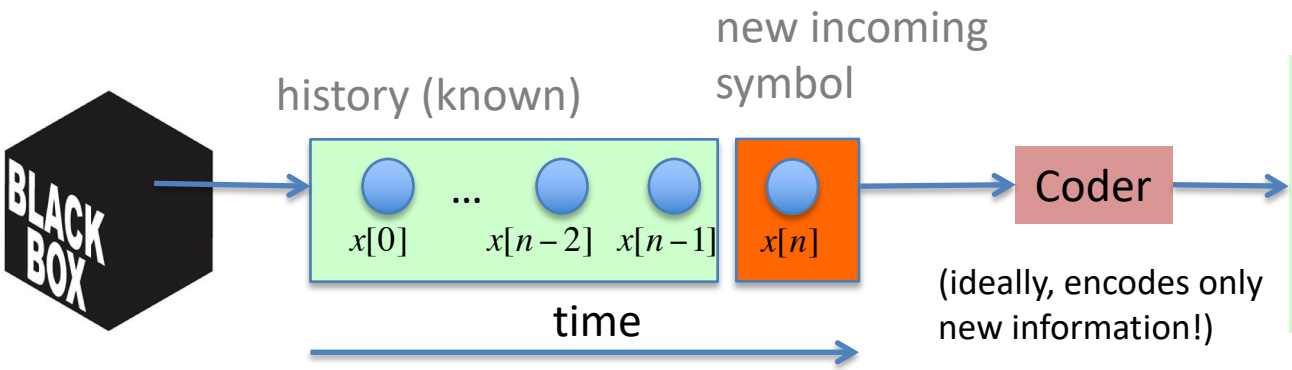
Compression:
Truncation of irrelevant signal parts, value quantization

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 (this lecture)
- 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

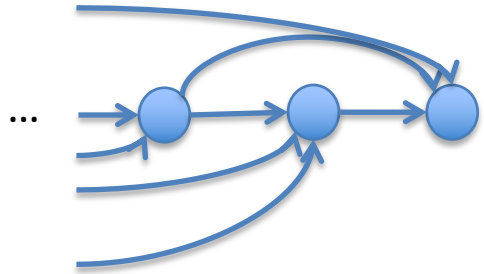
- 1-st order Markov process
- Efficient codes for arbitrary distributions: Huffman, Arithmetic, Golomb-Rice, …

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

Pre-Coding

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

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



Reminder: 1-st order model (entropy coding)

- Assume a "long enough" series of observations (history length), **N -> ∞**
- Input alphabet: $Z = \{s_i\},\ i = 1,\ldots,K,$ output alphabet: $\{0,1\}$ (binary digits, bits)
- Statistically independent (iid) symbols with known probabilities: $p_i^{(n)} = p_i$
- Average codeword length:

$$S_{src} = \langle l_i \rangle = \sum_{i=1}^{K} p_i \cdot l(s_i) \cdot [bits / symbol]$$

- **Source entropy**: average information content of a symbol:

$$H_{src} = \langle I(s_i) \rangle_Z = \sum_{i=1}^{K} p_i \cdot I(s_i) = \sum_{i=1}^{K} p_i \log_2 \frac{1}{p_i} \cdot [bits / symbol]$$

- **Shannon bound:**

$$H_{src} \le S_{src}$$

Goal: given a set of symbol probabilities $\mathbf{p_i}$, find code with $\mathbf{S_{src}}$ as close to $\mathbf{H_{src}}$ as possible

- Code redundancy:

$$\Delta R = S_{src} - H_{src} \ge 0$$

Types of entropy codes (theoretical classification)

All codes

Nonsingular codes

Definition: Every symbol s_i maps to a different codeword, $s_i \neq s_j \Rightarrow C(s_i) \neq C(s_i)$

Uniquely decodable codes

Definition: Any encoded string has only one possible source string producing it

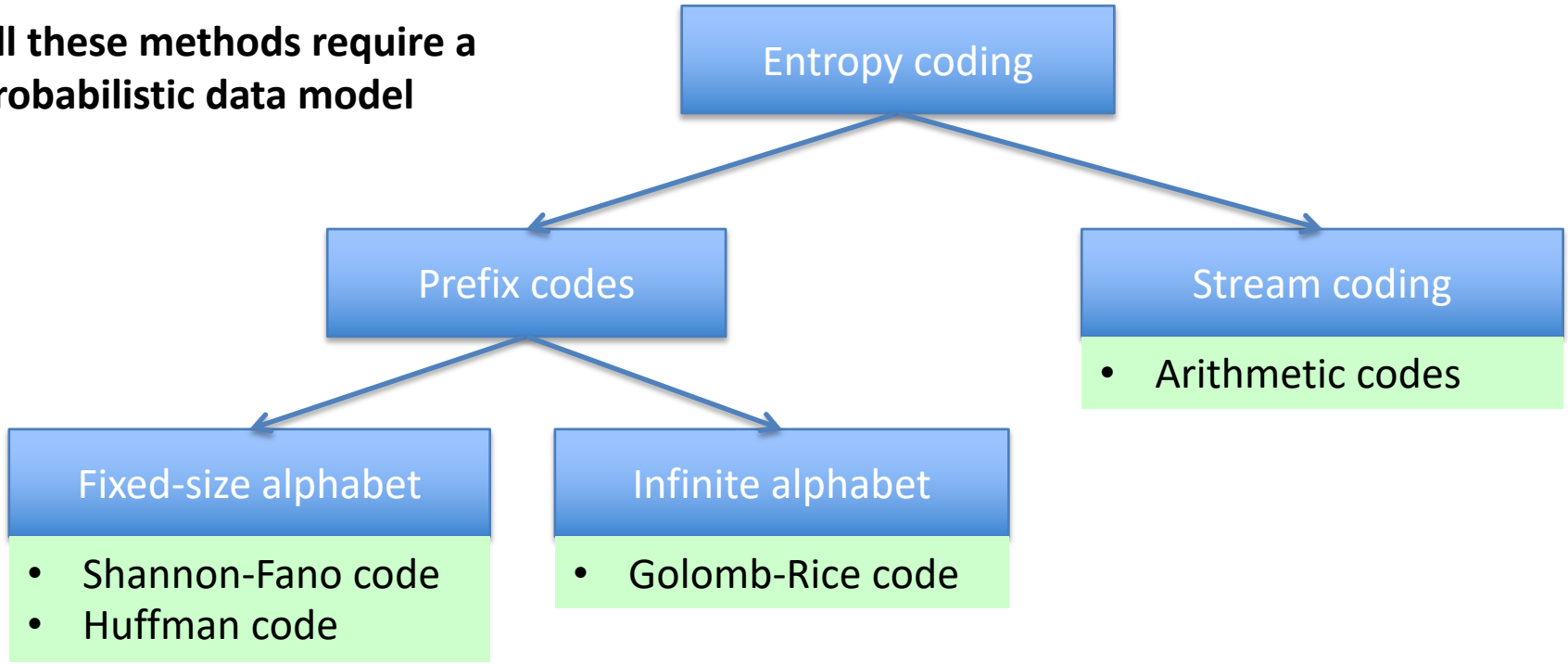
Instantaneous codes (= prefix codes = prefix-free codes)

Definition: No codeword is a prefix of any other codeword

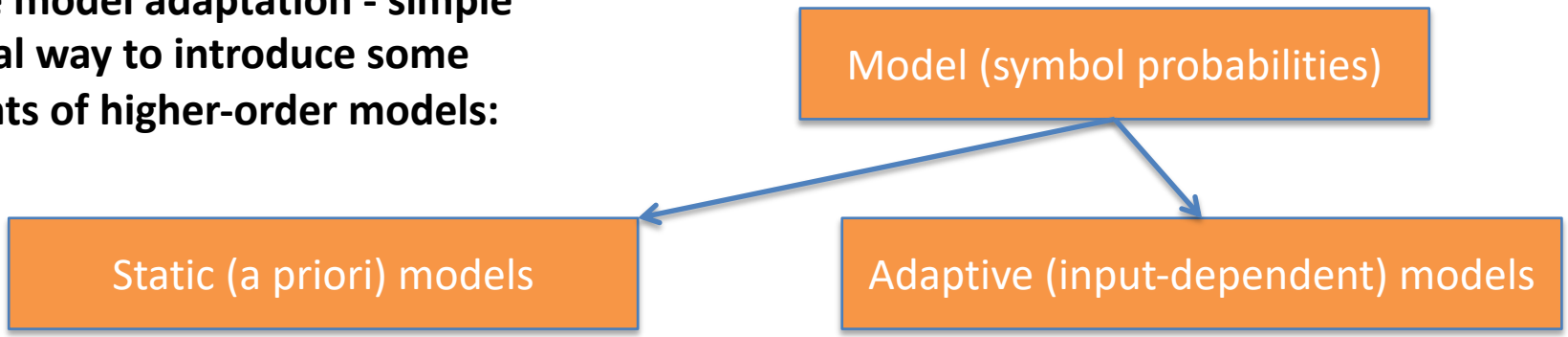
Kraft inequality (or **McMillan inequality**): uniquely decodable codes and instantaneous codes have exactly the same bounds on codeword length (i.e. are equally powerful)

Types of entropy codes (practical classification)

All these methods require a probabilistic data model



On-line model adaptation - simple practical way to introduce some elements of higher-order models:



Shannon-Fano code [Fano 1949] (finite alphabet, prefix)

1. Sort the alphabet symbols by their probability
2. Divide symbols into two groups with roughly equal sum of probabilities in each group
3. Assign "1" to the left group, "0" to the right
4. Repeat from step 2 recursively for each group that contains more than one symbol
5. Concatenate the bits assigned to each symbol into codewords

s_i	a	b	c	d	e	f	g
p_i	0.4	0.1	0.1	0.1	0.1	0.1	0.1

Shannon-Fano code [Fano 1949] (finite alphabet, prefix)

1. Sort the alphabet symbols by their probability
2. Divide symbols into two groups with roughly equal sum of probabilities in each group
3. Assign “1” to the left group, “0” to the right
4. Repeat from step 2 recursively for each group that contains more than one symbol
5. Concatenate the bits assigned to each symbol into codewords

s_i	a	b	c	d	e	f	g
p_i	**0.4**	**0.1**	**0.1**	**0.1**	**0.1**	**0.1**	**0.1**
	1		0				

Shannon-Fano code [Fano 1949] (finite alphabet, prefix)

1. Sort the alphabet symbols by their probability
2. Divide symbols into two groups with roughly equal sum of probabilities in each group
3. Assign "1" to the left group, "0" to the right
4. Repeat from step 2 recursively for each group that contains more than one symbol
5. Concatenate the bits assigned to each symbol into codewords

s_i	a	b	c	d	e	f	g
p_i	**0.4**	**0.1**	**0.1**	**0.1**	**0.1**	**0.1**	**0.1**
	1		0				
	1	0	1			0	

Shannon-Fano code [Fano 1949] (finite alphabet, prefix)

1. Sort the alphabet symbols by their probability
2. Divide symbols into two groups with roughly equal sum of probabilities in each group
3. Assign “1” to the left group, “0” to the right
4. Repeat from step 2 recursively for each group that contains more than one symbol
5. Concatenate the bits assigned to each symbol into codewords

s_i	a	b	c	d	e	f	g
p_i	**0.4**	**0.1**	**0.1**	**0.1**	**0.1**	**0.1**	**0.1**
	1		0				
	1	0	1			0	
			1		0	1	0
			1	0			

Shannon-Fano code [Fano 1949] (finite alphabet, prefix)

1. Sort the alphabet symbols by their probability
2. Divide symbols into two groups with roughly equal sum of probabilities in each group
3. Assign “1” to the left group, “0” to the right
4. Repeat from step 2 recursively for each group that contains more than one symbol
5. Concatenate the bits assigned to each symbol into codewords

s_i	a	b	c	d	e	f	g
p_i	**0.4**	**0.1**	**0.1**	**0.1**	**0.1**	**0.1**	**0.1**
	1		0				
	1	0	1			0	
			1		0	1	0
			1	0			
c_i	**11**	**10**	**0111**	**0110**	**010**	**001**	**000**

Shannon-Fano code [Fano 1949] (finite alphabet, prefix)

1. Sort the alphabet symbols by their probability
2. Divide symbols into two groups with roughly equal sum of probabilities in each group
3. Assign "1" to the left group, "0" to the right
4. Repeat from step 2 recursively for each group that contains more than one symbol
5. Concatenate the bits assigned to each symbol into codewords

s_i	a	b	c	d	e	f	g
p_i	**0.4**	**0.1**	**0.1**	**0.1**	**0.1**	**0.1**	**0.1**
	1		0				
	1	0	1			0	
			1		0	1	0
			1	0			
c_i	**11**	**10**	**0111**	**0110**	**010**	**001**	**000**

$$H_{src} = -0.4 \cdot \log_2 0.4 - 6 \cdot 0.1 \cdot \log_2 0.1 = 2.522 \text{[bits/symbol]}$$

$$S_{src} = 2 \cdot 0.4 + 2 \cdot 0.1 + 4 \cdot 0.1 + 4 \cdot 0.1 + 3 \cdot 0.1 + 3 \cdot 0.1 + 3 \cdot 0.1 = 2.7 \text{[bits/symbol]}$$

Shannon-Fano code [Fano 1949] (finite alphabet, prefix)

1. Sort the alphabet symbols by their probability
2. Divide symbols into two groups with roughly equal sum of probabilities in each group
3. Assign "1" to the left group, "0" to the right
4. Repeat from step 2 recursively for each group that contains more than one symbol
5. Concatenate the bits assigned to each symbol into codewords

s_i	a	b	c	d	e	f	g
p_i	**0.4**	**0.1**	**0.1**	**0.1**	**0.1**	**0.1**	**0.1**
	1		0				
	1	0	1		0		
			1		0	1	0
			1	0			
c_i	**11**	**10**	**0111**	**0110**	**010**	**001**	**000**

Pros: simple and fast!

Cons:

- where should we put the boundary in uncertain cases: "e|f" or "d|e"?

$$H_{src} = -0.4 \cdot \log_2 0.4 - 6 \cdot 0.1 \cdot \log_2 0.1 = 2.522 \text{[bits/symbol]}$$

$$S_{src} = 2 \cdot 0.4 + 2 \cdot 0.1 + 4 \cdot 0.1 + 4 \cdot 0.1 + 3 \cdot 0.1 + 3 \cdot 0.1 + 3 \cdot 0.1 = 2.7 \text{[bits/symbol]}$$

Shannon-Fano code [Fano 1949] (finite alphabet, prefix)

1. Sort the alphabet symbols by their probability
2. Divide symbols into two groups with roughly equal sum of probabilities in each group
3. Assign "1" to the left group, "0" to the right
4. Repeat from step 2 recursively for each group that contains more than one symbol
5. Concatenate the bits assigned to each symbol into codewords

s_i	a	b	c	d	e	f	g
p_i	**0.4**	**0.1**	**0.1**	**0.1**	**0.1**	**0.1**	**0.1**
	1		0				
	1	0	1		0		
			1		0	1	0
			1	0			
c_i	**11**	**10**	**0111**	**0110**	**010**	**001**	**000**

Pros: simple and fast!

Cons:

- where should we put the boundary in uncertain cases: "e|f" or "d|e"?
- code length may not correspond well to the information content of the symbol

$$H_{src} = -0.4 \cdot \log_2 0.4 - 6 \cdot 0.1 \cdot \log_2 0.1 = 2.522 \text{[bits/symbol]}$$

$$S_{src} = 2 \cdot 0.4 + 2 \cdot 0.1 + 4 \cdot 0.1 + 4 \cdot 0.1 + 3 \cdot 0.1 + 3 \cdot 0.1 + 3 \cdot 0.1 = 2.7 \text{[bits/symbol]}$$

Huffman code [Huffman 1952] (finite alphabet, prefix)

Construct a binary tree:

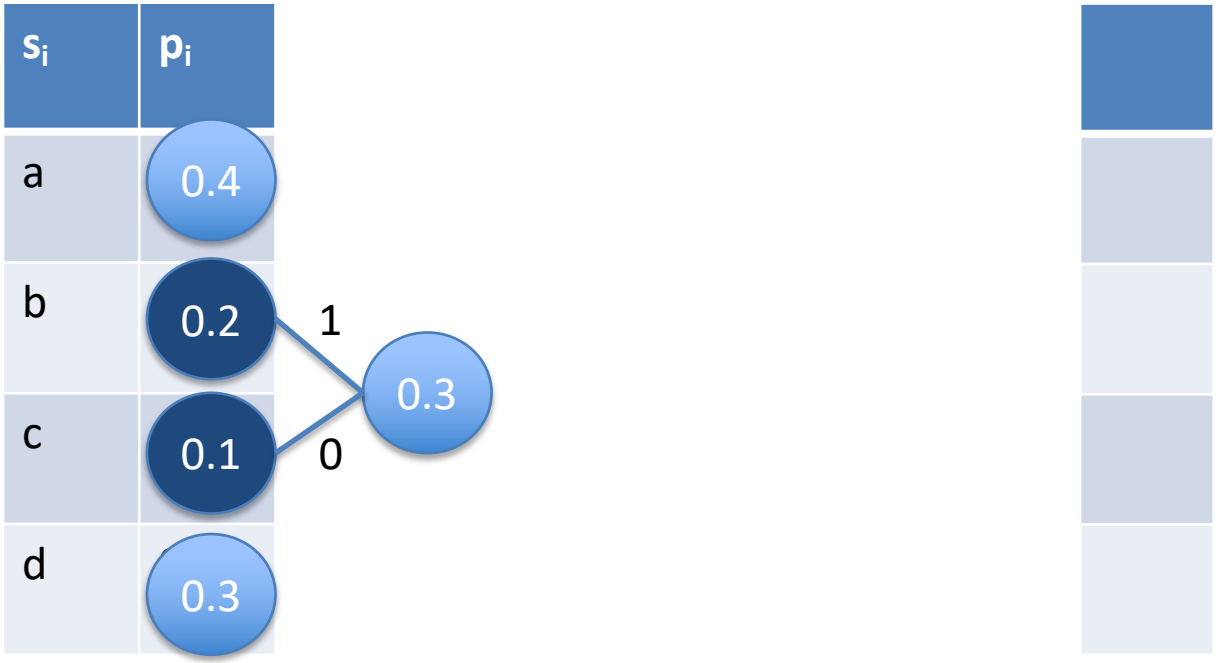
1. For each symbol, create active nodes with values equal to the symbol probabilities
2. Find two active nodes n_p and n_q with the smallest values p and q
3. Create a new active node n_{pq} with value (p + q), de-activate n_p and n_q, link them to n_{pq}
4. Denote the two added edges with “1” and “0”
5. Repeat from step 2 until only one node with value 1.0 remains
6. Read off the code for each symbol along the path from root to this symbol

s_i	p_i
a	0.4
b	0.2
c	0.1
d	0.3

Huffman code [Huffman 1952] (finite alphabet, prefix)

Construct a binary tree:

1. For each symbol, create active nodes with values equal to the symbol probabilities
2. Find two active nodes n_p and n_q with the smallest values p and q
3. Create a new active node n_{pq} with value (p + q), de-activate n_p and n_q, link them to n_{pq}
4. Denote the two added edges with “1” and “0”
5. Repeat from step 2 until only one node with value 1.0 remains
6. Read off the code for each symbol along the path from root to this symbol

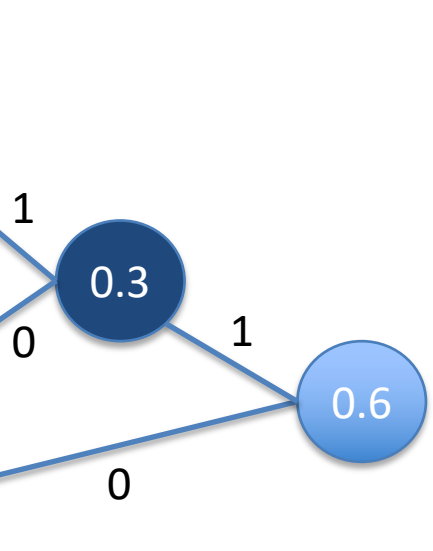


Huffman code [Huffman 1952] (finite alphabet, prefix)

Construct a binary tree:

1. For each symbol, create active nodes with values equal to the symbol probabilities
2. Find two active nodes n_p and n_q with the smallest values p and q
3. Create a new active node n_{pq} with value (p + q), de-activate n_p and n_q, link them to n_{pq}
4. Denote the two added edges with “1” and “0”
5. Repeat from step 2 until only one node with value 1.0 remains
6. Read off the code for each symbol along the path from root to this symbol

s_i	p_i
a	0.4
b	0.2
c	0.1
d	0.3

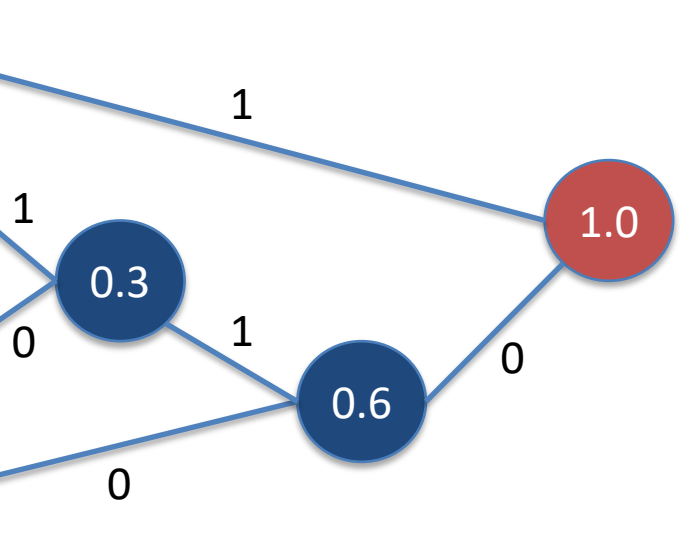


Huffman code [Huffman 1952] (finite alphabet, prefix)

Construct a binary tree:

1. For each symbol, create active nodes with values equal to the symbol probabilities
2. Find two active nodes n_p and n_q with the smallest values p and q
3. Create a new active node n_{pq} with value (p + q), de-activate n_p and n_q, link them to n_{pq}
4. Denote the two added edges with “1” and “0”
5. Repeat from step 2 until only one node with value 1.0 remains
6. Read off the code for each symbol along the path from root to this symbol

s_i	p_i
a	0.4
b	0.2
c	0.1
d	0.3

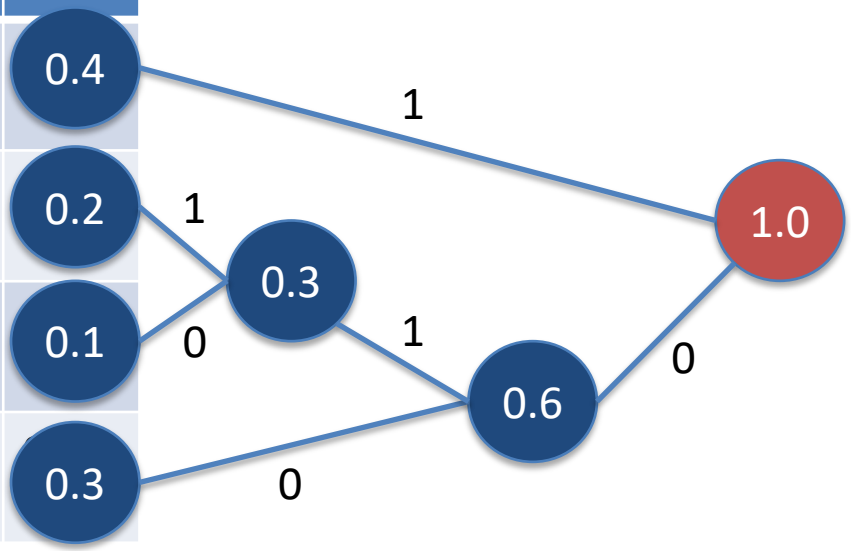


Huffman code [Huffman 1952] (finite alphabet, prefix)

Construct a binary tree:

1. For each symbol, create active nodes with values equal to the symbol probabilities
2. Find two active nodes n_p and n_q with the smallest values p and q
3. Create a new active node n_{pq} with value (p + q), de-activate n_p and n_q, link them to n_{pq}
4. Denote the two added edges with “1” and “0”
5. Repeat from step 2 until only one node with value 1.0 remains
6. Read off the code for each symbol along the path from root to this symbol

s_i	p_i	c_i
a	0.4	**1**
b	0.2	**011**
c	0.1	**010**
d	0.3	**00**

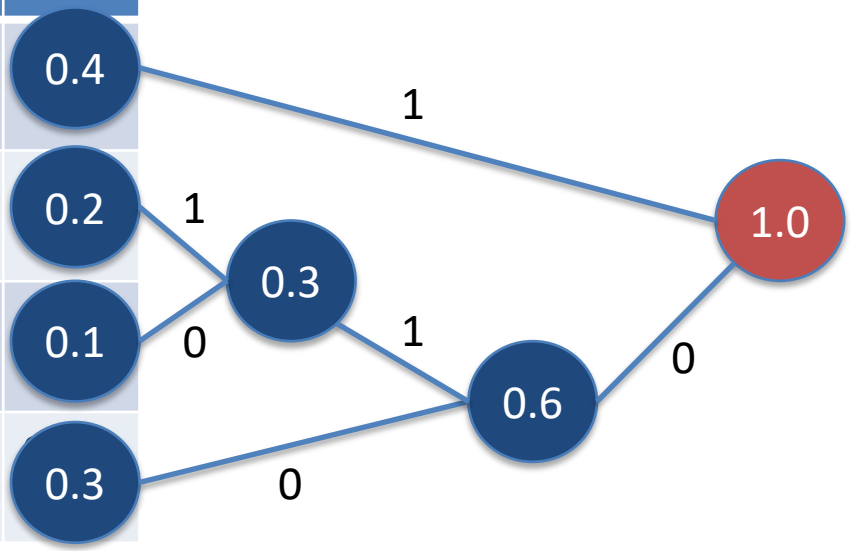


Huffman code [Huffman 1952] (finite alphabet, prefix)

Construct a binary tree:

1. For each symbol, create active nodes with values equal to the symbol probabilities
2. Find two active nodes n_p and n_q with the smallest values p and q
3. Create a new active node n_{pq} with value (p + q), de-activate n_p and n_q, link them to n_{pq}
4. Denote the two added edges with “1” and “0”
5. Repeat from step 2 until only one node with value 1.0 remains
6. Read off the code for each symbol along the path from root to this symbol

s_i	p_i	c_i
a	0.4	**1**
b	0.2	**011**
c	0.1	**010**
d	0.3	**00**



$$H_{src} = 1.846[\text{bits/symbol}]$$

$$S_{src} = 1.9[\text{bits/symbol}]$$

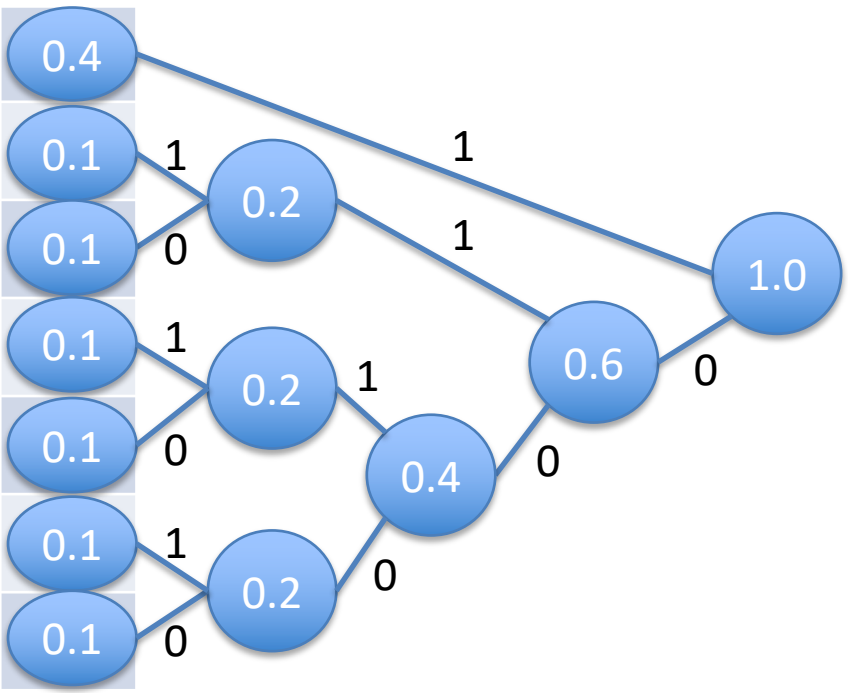
Not too bad: more probable symbols receive shorter codewords

Huffman code [Huffman 1952] (finite alphabet, prefix)

Construct a binary tree:

1. For each symbol, create active nodes with values equal to the symbol probabilities
2. Find two active nodes n_p and n_q with the smallest values p and q
3. Create a new active node n_{pq} with value (p + q), de-activate n_p and n_q, link them to n_{pq}
4. Denote the two added edges with “1” and “0”
5. Repeat from step 2 until only one node with value 1.0 remains
6. Read off the code for each symbol along the path from root to this symbol

s_i	p_i	c_i
a	0.4	**1**
b	0.1	**011**
c	0.1	**010**
d	0.1	**0011**
e	0.1	**0010**
f	0.1	**0001**
g	0.1	**0000**



$$H_{src} = 2.522[\text{bits/symbol}]$$

$$S_{src} = 2.6[\text{bits/symbol}]$$

(Shannon-Fano code: $\mathbf{S_{src} = 2.7}$)

Proven fact: Huffman code is optimal. For any uniquely decodable fixed-length code C:

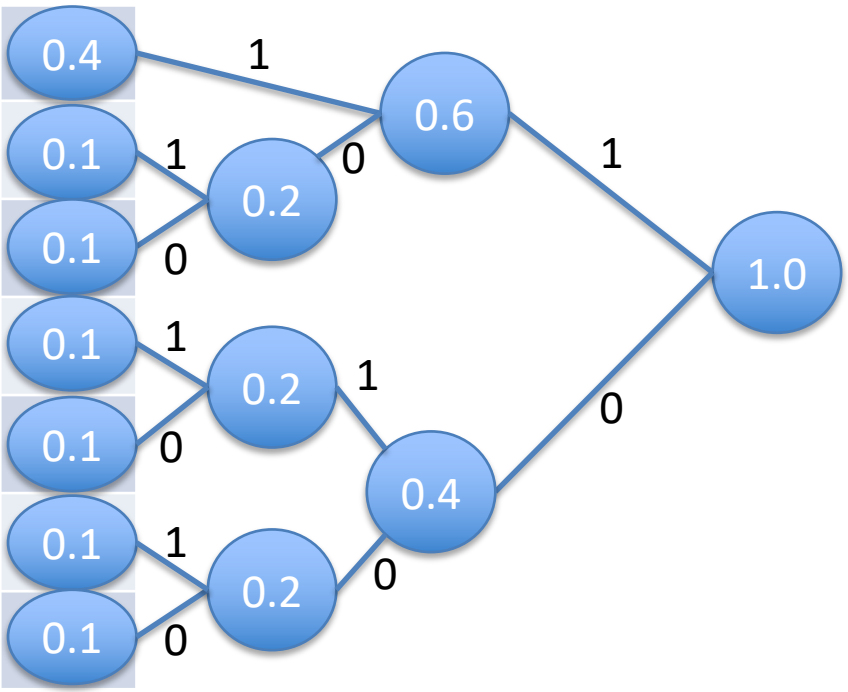
$$S_{src}^{\text{Huffman}} \le S_{src}^{\text{Code C}}$$

Huffman code [Huffman 1952] (finite alphabet, prefix)

Construct a binary tree:

1. For each symbol, create active nodes with values equal to the symbol probabilities
2. Find two active nodes n_p and n_q with the smallest values p and q
3. Create a new active node n_{pq} with value (p + q), de-activate n_p and n_q, link them to n_{pq}
4. Denote the two added edges with “1” and “0”
5. Repeat from step 2 until only one node with value 1.0 remains
6. Read off the code for each symbol along the path from root to this symbol

s_i	p_i	c_i
a	0.4	11
b	0.1	101
c	0.1	100
d	0.1	011
e	0.1	010
f	0.1	001
g	0.1	000



$$H_{src} = 2.522[\text{bits/symbol}]$$

$$S_{src} = 2.6[\text{bits/symbol}]$$

There may exist multiple Huffman codes for a given model, but all should have identical redundancy!

Redundancy can reach zero iff

$$p_i = 2^{-n}, \quad n = 1,2,3,\ldots$$

(dyadic probability distribution)

Golomb code [Golomb 1966] (infinite alphabet, prefix)

Example: a reporter plays in a casino, should send a message to the newspaper, communicating the value of *r* - number of zeros appearing in a row. The probability of having a zero in each game is *q*, and the mobile operator charges per each transferred bit.

Alphabet: potentially unbounded! Probability of some value *r*:

$$p(r) = q^r (1 - q)$$

Golomb code [Golomb 1966] (infinite alphabet, prefix)

Example: a reporter plays in a casino, should send a message to the newspaper, communicating the value of *r* - number of zeros appearing in a row. The probability of having a zero in each game is *q*, and the mobile operator charges per each transferred bit.

Alphabet: potentially unbounded! Probability of some value *r*: $p(r) = q^r(1-q)$

Consider auxiliary number $m = \ln(1/2)/\ln(q)$
Then $q^m = 1/2$, or $p(r+m) = 0.5p(r)$
=> may code *m* values of *r* with codes of length *L*, the following *m* values – with length *L + 1*, etc.

Rice codes [Rice 1979]: if $m = 2^k$, then need exactly *k* bits to identify each code within a sequence of *m* codewords
(simple coding/decoding, but limited application)

	q = 1/2, m = 1			
r	p(r)	code		
0	1/2	0		
1	1/4	10		
2	1/8	110		
3	1/16	1110		
4	1/32	11110		
5	1/64	111110		
6	1/128	1111110		
7	1/256	11111110		

Golomb code [Golomb 1966] (infinite alphabet, prefix)

Example: a reporter plays in a casino, should send a message to the newspaper, communicating the value of *r* - number of zeros appearing in a row. The probability of having a zero in each game is *q*, and the mobile operator charges per each transferred bit.

Alphabet: potentially unbounded! Probability of some value *r*: $p(r) = q^r(1-q)$

Consider auxiliary number $m = \ln(1/2)/\ln(q)$
Then $q^m = 1/2$, or $p(r+m) = 0.5p(r)$
=> may code *m* values of *r* with codes of length *L*, the following *m* values – with length *L + 1*, etc.

Rice codes [Rice 1979]: if $m = 2^k$, then need exactly *k* bits to identify each code within a sequence of *m* codewords
(simple coding/decoding, but limited application)

Chunk number *c*: encoded with unary code (sequence of *c* 1s, ending with 0)
Position within chunk: simple binary code

	q = 1/2, m = 1		q = 0.841, m = 4	
r	p(r)	code	p(r)	code
0	1/2	0	0.159	0 0 0
1	1/4	10	0.134	0 0 1
2	1/8	110	0.113	0 1 0
3	1/16	1110	0.095	0 1 1
4	1/32	11110	0.080	1 0 0 0
5	1/64	111110	0.067	1 0 0 1
6	1/128	1111110	0.056	1 0 1 0
7	1/256	11111110	0.047	1 0 1 1

Simple unary coding of the sequence number

Binary coding of m values

Golomb code [Golomb 1966] (infinite alphabet, prefix)

More formal and general recipe: given input value r and parameter m, divide with remainder: $r = q \times m + d$, such that $d < m$, encode q with unary code, and d with truncated binary code. Finally, concatenate results.

- Choice of m: can be adaptive (based on real data), or computed for the known probability distribution (as on the previous slide).
- Rice code coincides with the Huffman code when the latter is computed for an infinite alphabet.
- **Widely used in real codecs!**

	q = 0.794, m = 3	
r	p(r)	code
0	0.206	0 0
1	0.164	0 1 0
2	0.130	0 1 1
3	0.103	1 0 0
4	0.082	1 0 1 0
5	0.065	1 0 1 1
6	0.052	1 1 0 0
7	0.041	1 1 0 1 0

chunk length: $m = 3$

unary-coded sequence number

truncated binary-coded position in the chunk

Arithmetic coding: inspiration from the science fiction

Problem: how do you transport a library in a space ship? (assume that no flash drives exist)

Elegant solution [Stanislaw Lem]:

- Digitize all books, represent the result as a long binary string: *s* = "0001101011100011…"
- Pick number *x* between 0 and 1 s.t. its fractional part is *s*, i.e. *x = 0,0001101011100011…*
- Manufacture a stick with the length that exactly equals *x* meters (you can call it a USB-stick)
- Carry the stick with you. If want to read a book – just measure the stick length ☺

(Yeah, yeah. Physics unfortunately places limits on the achievable measurement accuracy…)

Arithmetic coding: stream coding, finite alphabet

- AC: one of the most practically used coding methods (e.g. in the JPEG standard)
- Can produce near-optimal output for any set of symbols and probabilities

Issues with the Huffman coding (and other codeword-based methods):

- Each codeword length is an integer, i.e. $\mathbf{l(s_i) \geq 1}$ (hence the average length $\mathbf{S_{src} \geq 1}$)

Example:

- Alphabet ***Z = {a, b, c}***, all probabilities equal: ***p = {1/3, 1/3, 1/3}***
- Huffman code: ***{"00","01","10"}*** – codeword ***"11"*** is wasted!
- Encode string ***"abbcab"*** – prefix codes need at least ***12*** bits

One way to improve:

- Alphabet *{aa, ab, ac, ba,…}*,
- More symbols, smaller individual probabilities, reduced redundancy.
- Further: *Z = {aaa, aab, …}*?

Alternative:

- Consider ***"abbcab"*** as the fractional part in a ternary number (i.e. base 3): $\mathbf{0.011201_3}$
- Convert this number to binary base: $\mathbf{0.011201_3 = 0.0010110001_2}$
- Now need only ***10*** bits to encode its fractional part!
- Note the new philosophy: represent a stream of symbols with a stream of bits

Some [relatively minor] issues with this method:

- One actually needs a way to indicate where the stream ends (e.g. extra symbol "data end")
- As described above, only a 0-th order model will be coded efficiently
- Need high-precision arithmetics for longer strings

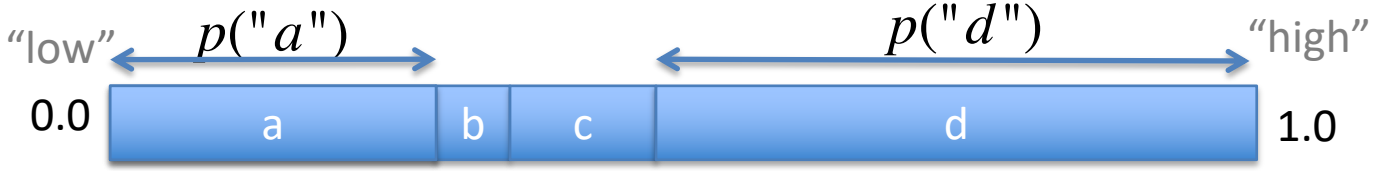
Arithmetic coding with a 1-st order model

1. Assume alphabet: ***{a,b,c,d}***, incoming string: ***"adbc"***, symbol probabilities p_i
2. **Start:** denote the unit interval ***[0, 1]*** on the real axis, assume it as the current interval
3. Divide the current interval into pieces of length proportional (equal) to the symbol probabilities
4. Choose the sub-interval according to the actual incoming symbol, assume it as the current interval
5. **Continue to step 3**

6. **Result:**
   choose any point in the last current interval, convert to binary and output

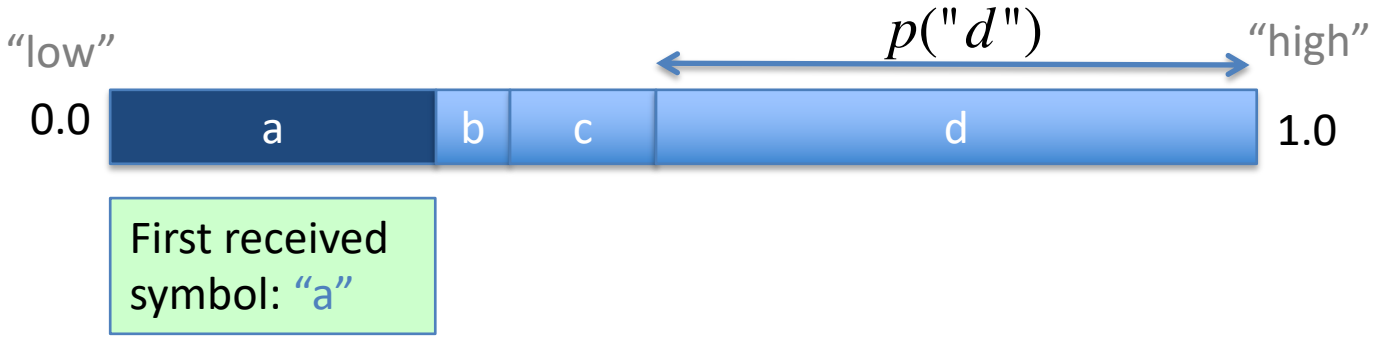
Arithmetic coding with a 1-st order model

1. Assume alphabet: ***{a,b,c,d}***, incoming string: ***"adbc"***, symbol probabilities p_i
2. **Start:** denote the unit interval ***[0, 1]*** on the real axis, assume it as the current interval
3. Divide the current interval into pieces of length proportional (equal) to the symbol probabilities
4. Choose the sub-interval according to the actual incoming symbol, assume it as the current interval
5. **Continue to step 3**
6. **Result:** choose any point in the last current interval, convert to binary and output



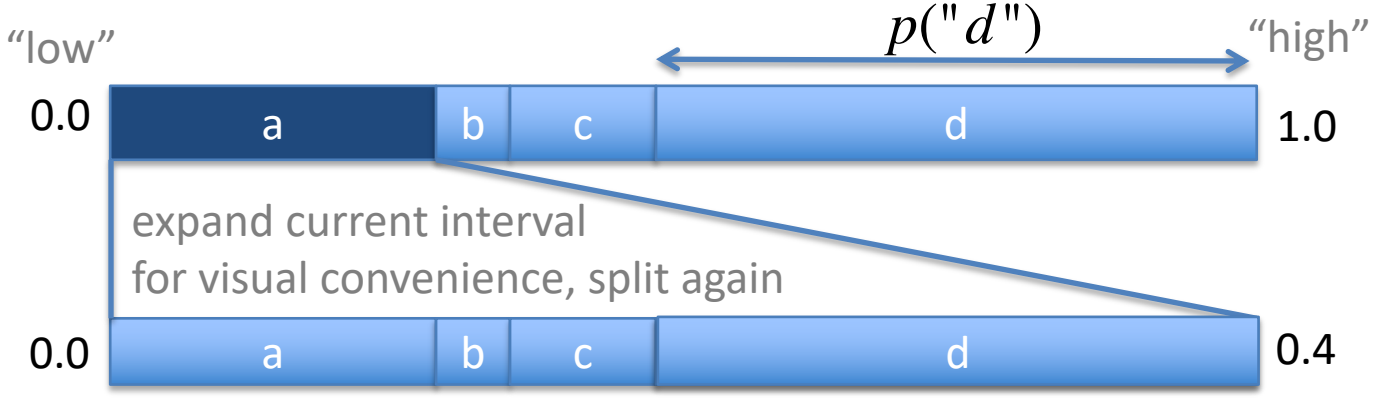
Arithmetic coding with a 1-st order model

1. Assume alphabet: ***{a,b,c,d}***, incoming string: ***"adbc"***, symbol probabilities p_i
2. **Start:** denote the unit interval ***[0, 1]*** on the real axis, assume it as the current interval
3. Divide the current interval into pieces of length proportional (equal) to the symbol probabilities
4. Choose the sub-interval according to the actual incoming symbol, assume it as the current interval
5. **Continue to step 3**
6. **Result:** choose any point in the last current interval, convert to binary and output



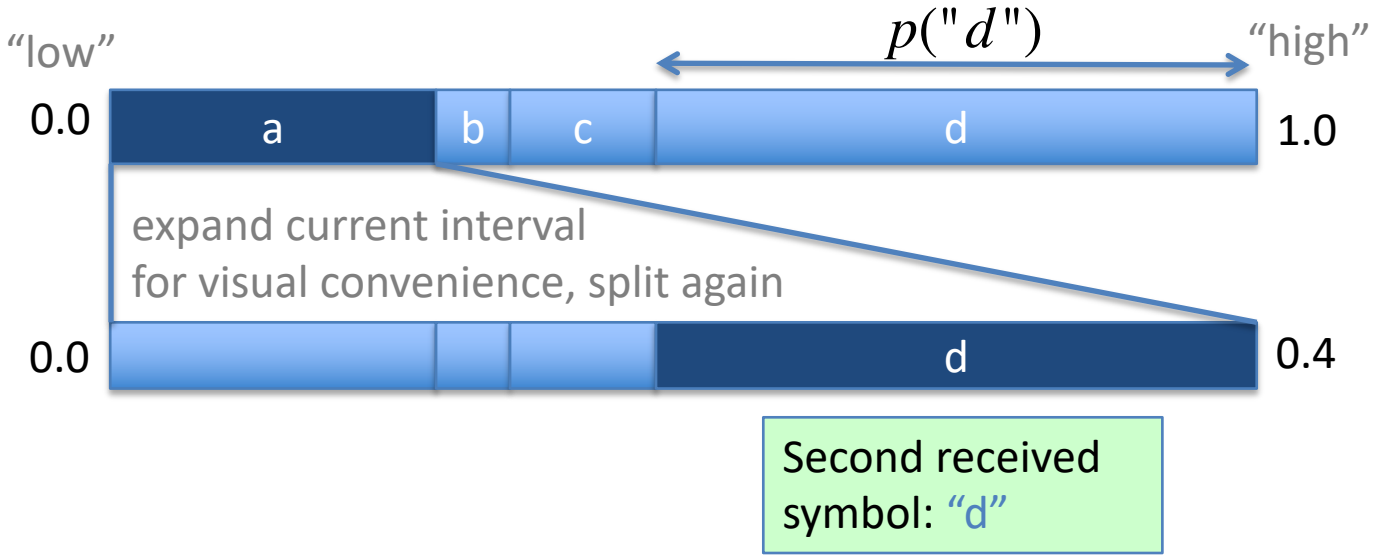
Arithmetic coding with a 1-st order model

1. Assume alphabet: ***{a,b,c,d}***, incoming string: ***"adbc"***, symbol probabilities p_i
2. **Start:** denote the unit interval ***[0, 1]*** on the real axis, assume it as the current interval
3. Divide the current interval into pieces of length proportional (equal) to the symbol probabilities
4. Choose the sub-interval according to the actual incoming symbol, assume it as the current interval
5. **Continue to step 3**
6. **Result:** choose any point in the last current interval, convert to binary and output



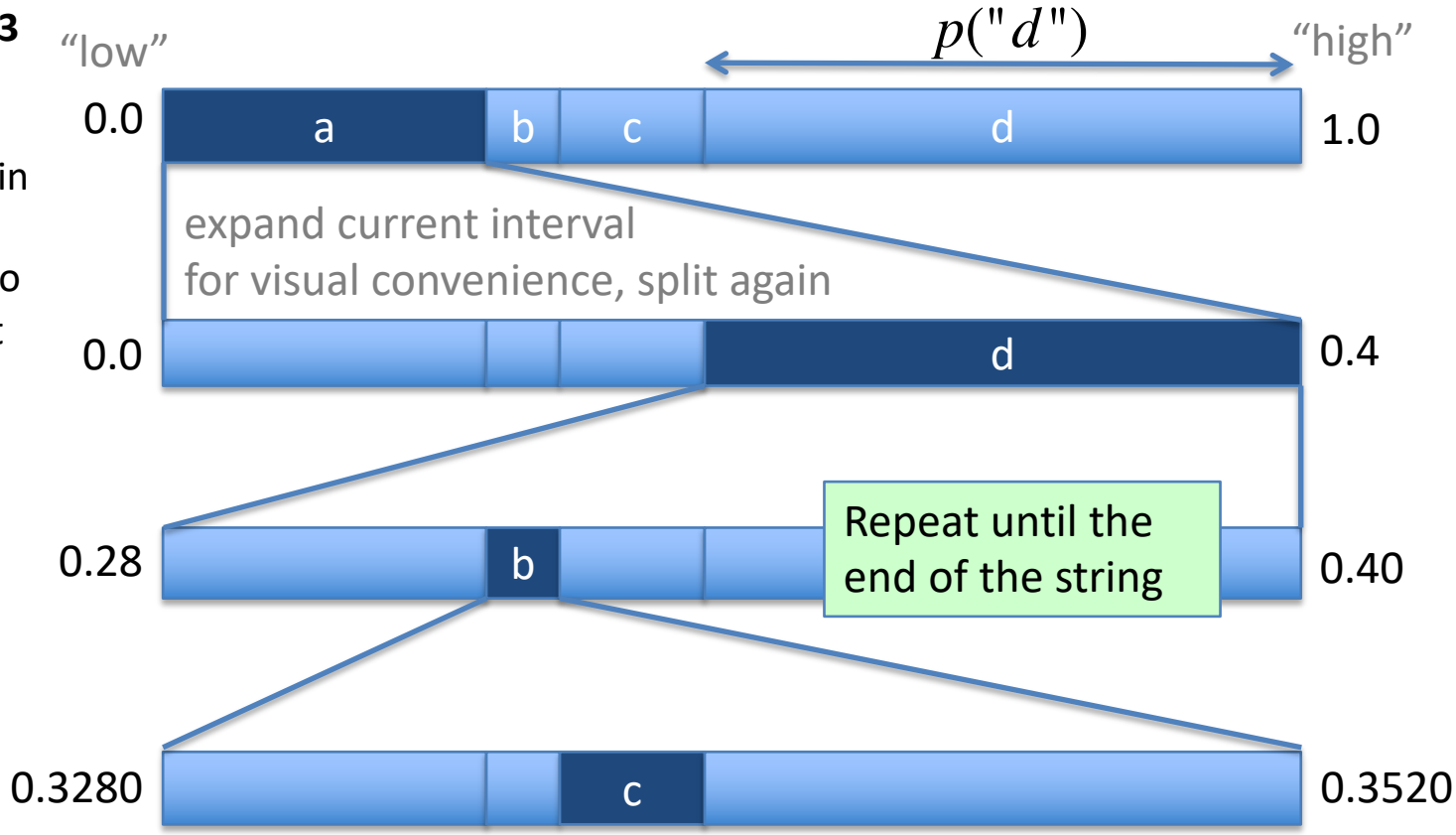
Arithmetic coding with a 1-st order model

1. Assume alphabet: ***{a,b,c,d}***, incoming string: ***"adbc"***, symbol probabilities p_i
2. **Start:** denote the unit interval ***[0, 1]*** on the real axis, assume it as the current interval
3. Divide the current interval into pieces of length proportional (equal) to the symbol probabilities
4. Choose the sub-interval according to the actual incoming symbol, assume it as the current interval
5. **Continue to step 3**
6. **Result:** choose any point in the last current interval, convert to binary and output



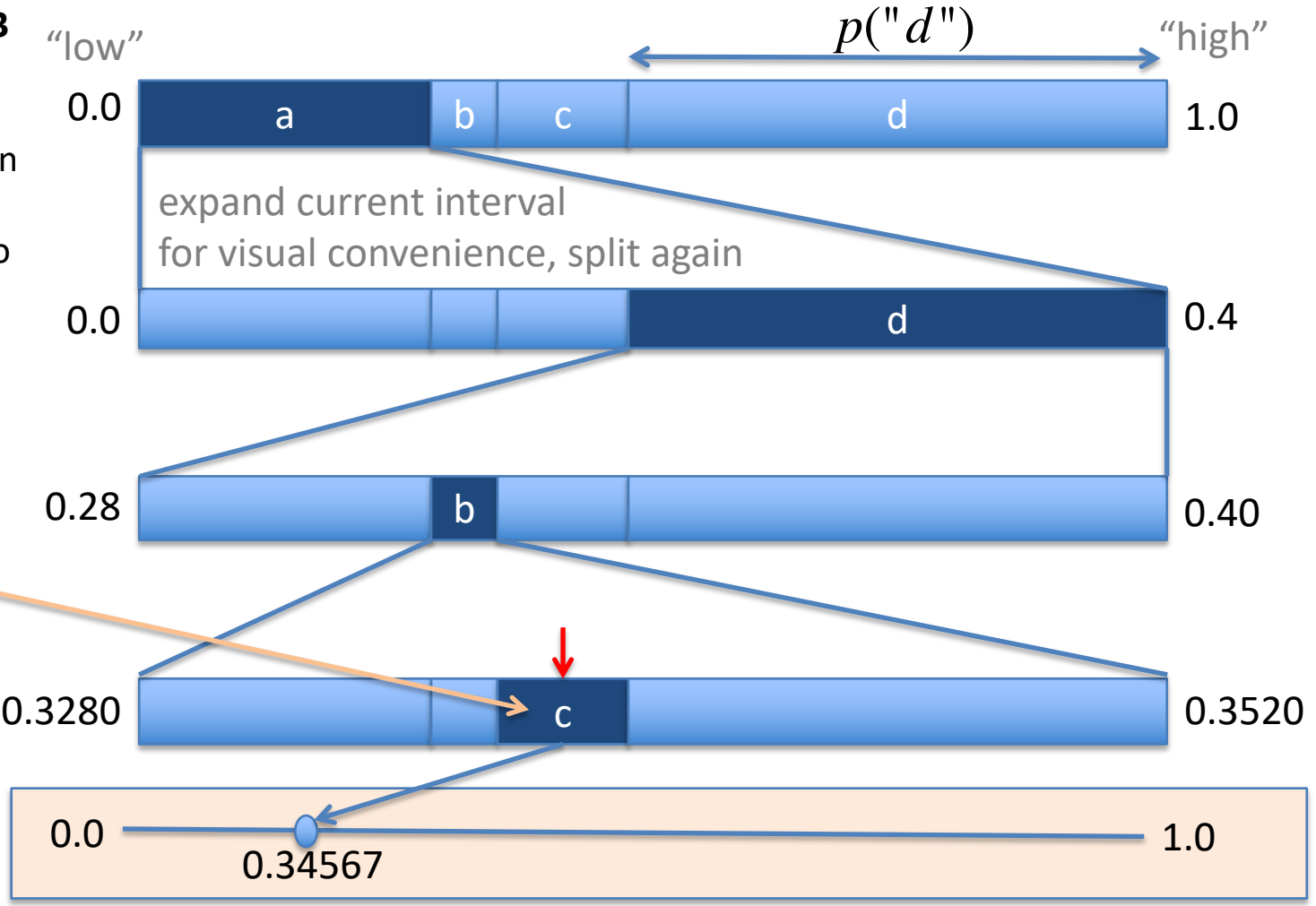
Arithmetic coding with a 1-st order model

1. Assume alphabet: ***{a,b,c,d}***, incoming string: ***"adbc"***, symbol probabilities p_i
2. **Start:** denote the unit interval ***[0, 1]*** on the real axis, assume it as the current interval
3. Divide the current interval into pieces of length proportional (equal) to the symbol probabilities
4. Choose the sub-interval according to the actual incoming symbol, assume it as the current interval
5. **Continue to step 3**
6. **Result:** choose any point in the last current interval, convert to binary and output



Arithmetic coding with a 1-st order model

1. Assume alphabet: ***{a,b,c,d}***, incoming string: ***"adbc"***, symbol probabilities p_i
2. **Start:** denote the unit interval ***[0, 1]*** on the real axis, assume it as the current interval
3. Divide the current interval into pieces of length proportional (equal) to the symbol probabilities
4. Choose the sub-interval according to the actual incoming symbol, assume it as the current interval
5. **Continue to step 3**
6. **Result:** choose any point in the last current interval, convert to binary and output



Arithmetic coding with a 1-st order model

Formal description of the intuitive algorithm:

1. Given alphabet and model: $Z = \{s_0, s_1, \ldots, s_{K-1}\}$
2. Initialize cumulative probabilities: $c_0 = 0, c_{i+1} = c_i + p_i$
3. Set initial interval boundaries: $low = 0.0,\ high = 1.0$
4. Receive next symbol s_j, determine its index *j*
5. Update interval boundaries:
$$range \leftarrow high - low$$
$$high \leftarrow low + range \cdot (c_{j+1} / c_K)$$
$$low \leftarrow low + range \cdot (c_j / c_K)$$
6. Repeat from step 3 until the end of input
7. Final value is any point x, s.t. $low \le x < high$

Example:

$$s_i = \{a, b, c, d\}$$
$$p_i = \{0.4, 0.2, 0.1, 0.3\}$$
$$c_i = \{0, 0.4, 0.6, 0.7, 1.0\}$$

Decoder:
the same sequence, but interval index *j* (i.e. decoded symbol) at step 4 is chosen according to the given value *x*

Some issues with this algorithm (as described above):

- Need to encode the entire signal before the transmission
- Need arithmetics with arbitrary precision

Practical arithmetic coding [Witten, Neal, Cleary 1987]

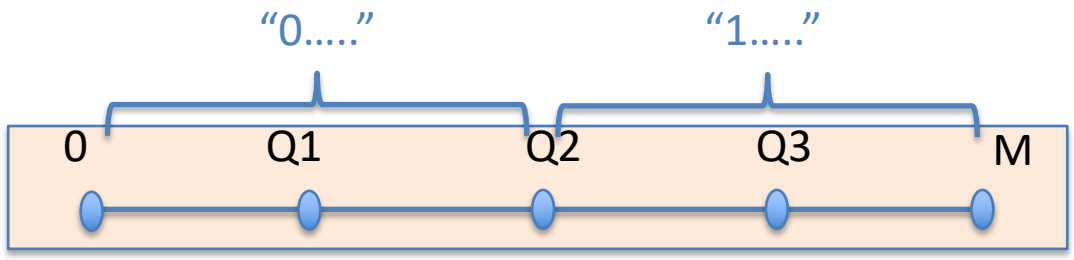
Integer-only solution: use integer high and low, output higher bits when they coincide

1. Prepare cumulative symbol counts, e.g.: H = {0, 4, 6, 7, 10};
2. Representation width: B bits, largest number: $M = 2^B - 1$
3. Auxiliary constants: $Q_1 = M / 4 + 1$, $Q_2 = 2Q_1$, $Q_3 = 3Q_1$
4. Initialize: low = 0; high = M; n = 0;
5. Receive input symbol s_j
6. Update interval boundaries:

```
range = high – low + 1;
high = low + range * (H_{j+1} / H_K) - 1;
low = low + range * (H_j / H_K);
```

7. Output identical higher bits in high and low:
8. Repeat from step 5 until the end of input
9. Finalize:

```
send((low < Q1) ? ('0' + n x '1') : ('1' + n x '0'));
```



We require that *high* > *low*, i.e. *range* > Q_1, and $H_K < Q_1$, => may need to re-scale *high* and *low*:

```
while (true) {
  if (high < Q2) {
    send('0' + n x '1');
    n = 0;
  } elseif (low ≥ Q2) {
    send('1' + n x '0');
    n = 0;
    low -= Q2;
    high -= Q2;
  } elseif ((low ≥ Q1) and (high < Q3)) {
    n++;
    low -= Q1;
    high -= Q1;
  } else {
    break;
  }
  low *= 2;
  high = 2 * high + 1;
}
```

Practical Arithmetic Coding [Witten, Neal, Cleary 1987]

Decoding:

1. Same initialization as in the encoder (H, K, M, B, Q_1, Q_2, Q_3, high, low)
2. Read B bits from the input stream, interpret as a binary number pos
3. Update interval boundaries and find encoded symbol:

```
range = high – low + 1;
j = K – 1;
while ((pos – low + 1) * H_K – 1/range < H_j) {
  j--;
}
high = low + range * (H_{j+1} / H_K) - 1;
low = low + range * (H_j / H_K);
```

4. Re-scale boundaries and read off the next bit(s): →
5. Repeat from step 3 until the end of input ☺

```
while (true) {
  if (low ≥ Q_2) {
    low -= Q_2;
    high -= Q_2;
    pos -= Q_2;
  } elseif ((low ≥ Q_1) and (high < Q_3)) {
    low -= Q_1;
    high -= Q_1;
    pos -= Q_1;
  } else {
    break;
  }
  low *= 2;
  high = 2 * high + 1;
  pos = 2 * pos + get_next_bit();
}
```

From static to adaptive models

Trivial adaptation of prefix codes:

- First pass: scan input, count symbols, create the model
- Send the model to the decoder
- Second pass: actually encode the input string
- Non-stationary source: re-create model when needed

Context adaptation / context modeling:

- Prepare in advance several static tables based on a priori information, share with decoder
- During coding, count the actual symbol frequencies
- Select the table providing maximum compression
- Send the selected table number to the decoder
- Golomb-Rice code context: need to only send m
- **In theory, already not quite a 1-st order model!**

[Possible] context quantization of pixel values:

- Context chosen based on neighbor values (or deltas)
- Similar contexts grouped together using vector quantization techniques
- Used in JPEG-LS

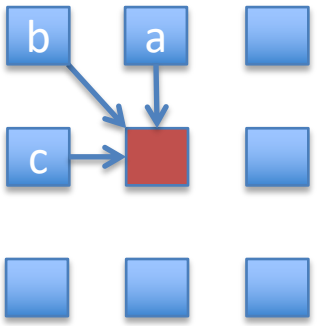
Model:

$$p_i = p(s_i)$$

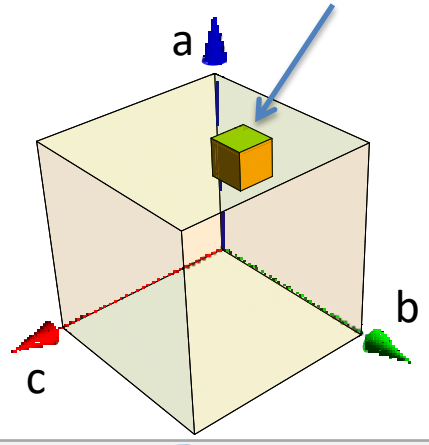
Context

Weather	Whole year	Summer	Winter
Sun	0.5	0.5	0.5
Clouds	0.25	0.25	0.25
Rain	0.125	0.25	0.0
Snow	0.125	0.0	0.25

Image pixel values



Context selection



From static to adaptive models

Adaptation via sorting:

- No a priori knowledge of real symbol frequencies
- Assume some viable frequencies, produce codewords, order by their lengths
- Assign codewords to symbols according to some rule
- After each encoded symbol, re-sort the symbols by accumulated frequencies
- Perform an identical operation synchronously in the decoder

s_i	#	Code
a	0	0
b	0	10
c	0	110
d	0	1110

"b"

s_i	#	Code
b	1	0
a	0	10
c	0	110
d	0	1110

"c"

s_i	#	Code
b	1	0
c	1	10
a	0	110
d	0	1110

"c"

s_i	#	Code
c	2	0
b	1	10
a	0	110
d	0	1110

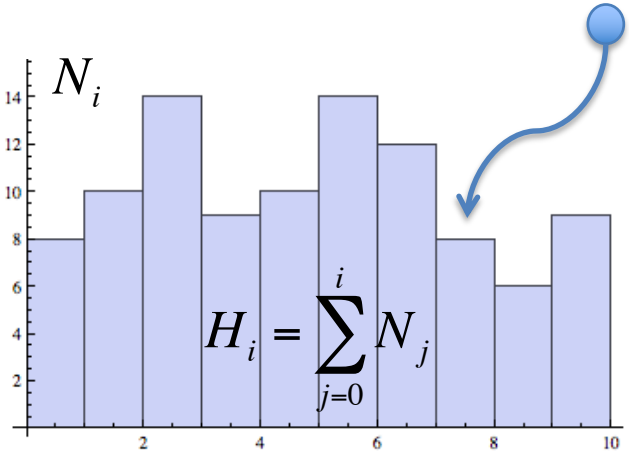
...

Adaptation of arithmetic coding: extremely convenient!

- Accumulate absolute frequencies H_i from the signal
- As H_K reaches some maximum value, divide all H_i by 2

AC adaptation + context switching:

- Increase H_i for selected symbols (depending on the context) by more than a unit (in order to faster adapt to arbitrary stationary sources)

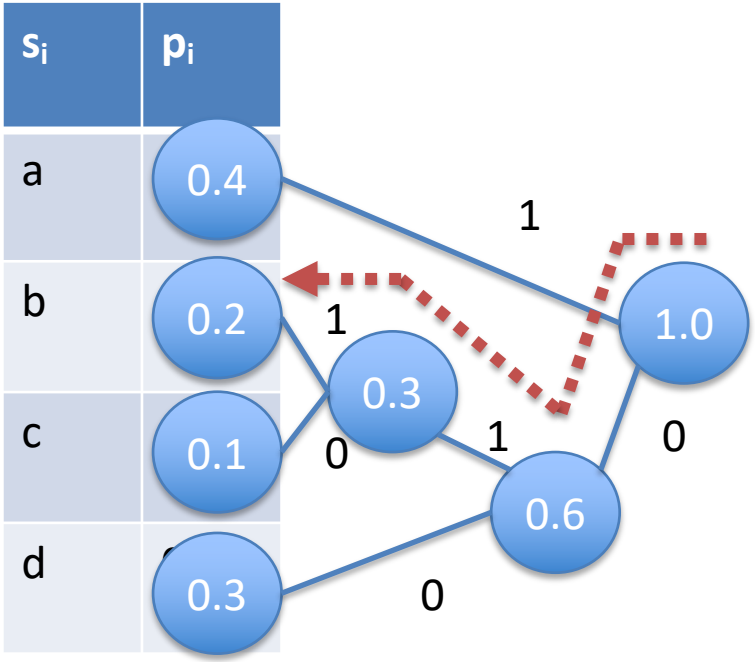


Some (useful?) extensions of the coding theory: sampling

You have a fair coin. How do you generate a stream of iid symbols with a given distribution?

- Generate a Huffman code tree according to the given symbol probabilities
- Start from trunk, toss the coin, follow the respective branch, repeat.
- Expected number of tosses equals the mean codeword length: $S_{src} \leq H_{src} + 1$

Events to be selected with the given probabilities



c_i
1
011
010
00

Sequences of toss outcomes leading to the respective symbols

Resulting iid sequence of samples: *a, b, a, d, c, d, a, d, a, b, …*