

QI/EC Modeling and Entropy Pumps^{*}

Ratko V. Tomic

1stWorks Corporation[†]

^{*} The QI source code and tech. reports are available at: <http://www.1stworks.com/ref/qi.htm>

We use concepts developed in chapter 4 of TR05-0625 <http://www.1stworks.com/ref/TR/tr05-0625a.pdf>

[†] PO Box 1030, North Falmouth, MA 02556, USA, tel: (508) 541-6781, <http://www.1stWorks.com>

Table of Contents

Entropy Pumps for Bernoulli Source.....	3
<i>EP1: Single Cycle, 2 symbols/block.....</i>	<i>3</i>
ENCODING.....	4
DECODING.....	6
EP1 PERFORMANCE	6
<i>EP2: Two Cycles, 2 symbols/block.....</i>	<i>8</i>
BINARY DECOMPOSITION OF T.....	8
EP2 ENCODING	10
EP2 DECODING	11
EP2 PERFORMANCE	12
<i>EP3: Multicycle, 2 symbols/block.....</i>	<i>13</i>
EP3 ENCODING	18
EP3 COMPONENT LAYOUT.....	21
EP3 DECODING	23
<i>EP4: EP with block lengths $d > 2$ symbols.....</i>	<i>25</i>
EP4 ENCODING	25
EP4 DECODING	27
EP4 PERFORMANCE	28
MULTICYCLE EP4 VARIANTS.....	28
QI/EC MODELING OF MARKOV SOURCES	30
<i>Combinatorial Properties of Contexts.....</i>	<i>31</i>
<i>MS1: Modeling Markov Source of Known Order r.....</i>	<i>34</i>
ENCODING.....	34
DECODING.....	35
USE OF GENERAL PROBABILISTIC MODELS BY QI/EC	36
<i>GPM1: QI/EC GPM Modeler.....</i>	<i>36</i>
Appendix A: Postorder Tree Traversal (for EP3 Component Layout).....	38
Appendix B: Bit Fractions Removal via Mixed Radix Coding.....	39
<i>Block Bit Fractions Removal</i>	<i>41</i>
<i>Hierarchical Radix Coding (HRC).....</i>	<i>44</i>
2-LEVEL HRC	44
<i>k</i> -LEVEL HRC.....	45
MIXED RADIX HRC.....	46
Appendix C: Removing SWI Exponents from QI Tables.....	47
<i>RXL: Exponents Removal via Log Tables.....</i>	<i>47</i>
Gaps & Mantissa Excess in QI Binomial Tables ($n=1024$).....	51

Entropy Pumps for Bernoulli Source

The next sections present a detailed description of **Entropy Pump**^{*} (**EP**) construction for binary order 0 Markov source (Bernoulli source). The source is characterized by probabilities of symbols 1 and 0, denoted as $P(1) \equiv p$ and $P(0) \equiv q = 1-p$. Unless noted otherwise, we will use convention $p \leq q$, hence we have $0 \leq p \leq 1/2 \leq q \leq 1$. The input sequence being encoded is $\mathbf{S}[n] = a_1 a_2 a_3 \dots a_n$, where a_t are symbols 0 or 1. We denote count of ones in $\mathbf{S}[n]$ as k and count of zeros as $l = n - k$.

Shannon entropy $H(\mathbf{S})$ of the input sequence $\mathbf{S}[n]$ is given by:

$$H(\mathbf{S}) = n \cdot [p \cdot \log(1/p) + q \cdot \log(1/q)] \equiv n \cdot h(p) \quad (1)$$

where we denoted **entropy per symbol** as: $h(p) \equiv p \cdot \log(1/p) + q \cdot \log(1/q)$. Note that $H(\mathbf{S})$ does not include the cost of transmitting the $\mathbf{S}[n]$ parameters p and n . Therefore in our output size comparisons with $H(\mathbf{S})$ we do not include cost of transmission of p and n , even though the encoder and decoder need to know these parameters.

EP1: Single Cycle, 2 symbols/block

a) The **virtual dictionary** \mathbf{D} is defined as a set of all 2-symbol sequences:

$$\mathbf{D} = \{ 00, 01, 10, 11 \} \quad (2)$$

b) Dictionary \mathbf{D} is partitioned into **enumerative classes** (disjoint subsets of \mathbf{D} , each containing equiprobable sequences only) as follows[†]:

$$\mathbf{D} = \mathbf{E}_0 + \mathbf{E}_1 + \mathbf{E}_2 \quad (3)$$

where the enumerative classes \mathbf{E}_0 , \mathbf{E}_1 and \mathbf{E}_2 are defined as:

$$\mathbf{E}_0 = \{ 00 \} \quad (4a)$$

$$\mathbf{E}_1 = \{ 11 \} \quad (4b)$$

$$\mathbf{E}_2 = \{ 01, 10 \} \quad (4c)$$

The steps (a) and (b) are common to encoder and decoder.

^{*} We use notation and terms defined in TR2, ref [1] p. 27.

[†] The operator '+' between set operands denotes the set union of the operands.

ENCODING

c) We *segment the input* $S[n]$ into $m = \lfloor n/2 \rfloor$ two-symbol blocks* as: $\mathbf{B} = \mathbf{B}_1 \mathbf{B}_2 \dots \mathbf{B}_m$ and generate the corresponding sequence of *class tags*: $\mathbf{T}[m] \equiv \mathbf{T} = T_1 T_2 \dots T_m$, where each T_j is a symbol 0, 1 or 2 identifying the enumerative class \mathbf{E}_c ($c=0,1,2$) to which the block \mathbf{B}_j belongs. Hence \mathbf{T} is a sequence in ternary alphabet ($\alpha=3$) defined via:

$$\mathbf{B}_j \in \mathbf{E}_c \Rightarrow T_j = c \quad \text{where: } c=0,1 \text{ or } 2 \text{ and } j=1,2,\dots,m \quad (5)$$

The probabilities p_c of occurrence of tags T_j in \mathbf{T} with values $c=0,1$ or 2 are:

$$p_0 \equiv P(T_j=0) = q^2 \quad \text{and} \quad q_0 \equiv 1-p_0 \quad (6a)$$

$$p_1 \equiv P(T_j=1) = p^2 \quad \text{and} \quad q_1 \equiv 1-p_1 \quad (6b)$$

$$p_2 \equiv P(T_j=2) = 2pq \quad \text{and} \quad q_2 \equiv 1-p_2 \quad (6c)$$

We denote the counts of tags T_j in \mathbf{T} with values $c=0,1$ or 2 as:

$$m_0 \equiv \text{Count_of}(T_j=0), \quad \underline{m}_0 \equiv m - m_0 \quad (7a)$$

$$m_1 \equiv \text{Count_of}(T_j=1), \quad \underline{m}_1 \equiv m - m_1 \quad (7b)$$

$$m_2 \equiv \text{Count_of}(T_j=2), \quad \underline{m}_2 \equiv m - m_2 \quad (7c)$$

Hence, the counts m_c satisfy constraints:

$$m_0 + m_1 + m_2 = m \equiv \lfloor n/2 \rfloor \quad (8a)$$

$$0 \leq m_c \leq m \quad \text{for } c=0,1,2 \quad (8b)$$

From eqs. (6)-(8) it follows that the probability for a given m_c to have a value μ (where μ satisfies: $0 \leq \mu \leq m$) is given via binomial distribution:

$$P(m_c = \mu) = p_c^\mu q_c^{m-\mu} \binom{m}{\mu} \quad (9)$$

d) We *compute the index* of each block \mathbf{B}_j within its enumerative class.

Since classes \mathbf{E}_0 and \mathbf{E}_1 contain only a single element each, all enumerative indices of blocks within these classes are 0, thus these indices are *not included into the output*.

The class \mathbf{E}_2 contains 2 equiprobable elements, hence its index requires 1 bit per block from \mathbf{E}_2 . By our *colex ranking* convention (cf. [2] p. 14) we assign index values 0 and 1 to the \mathbf{E}_2 elements 10 and 01 respectively i.e. the block index is the last bit of the block. We denote the sequence of m_2 index values (bits) for all m_2 blocks $\mathbf{B}_j \in \mathbf{E}_2$ as $\mathbf{Z} \equiv \mathbf{Z}[m_2]$.

* For odd sequence lengths n we need to send the last symbol separately from the blocks \mathbf{B}_j via a convention agreed to in advance by encoder and decoder.

e) **Encoder output** contains the following components (note that $\mathbf{S}[n]$ parameters p and n are assumed to be known to encoder and decoder):

e1) Optimally coded sequence of class tags $\mathbf{T} = T_1 T_2 \dots T_m$

$\mathbf{T}[m] \equiv \mathbf{T}$ is a sequence of m symbols in ternary alphabet ($\alpha=3$) with known symbol probabilities (given via eqs. (6a)-(6c)) and known symbol count $m = \lfloor n/2 \rfloor$.

An optimal multi-alphabet entropy coder can be used to encode \mathbf{T} , such as arithmetic coder or QI (e.g. via binary decomposition or via direct multinomial enumeration, cf. [2] pp. 31-39). The average output length for \mathbf{T} from an optimal coder is $L(\mathbf{T}) = m \cdot \sum_c p_c \log(1/p_c)$, which after substitution of p_c values given in eqs. (6), yields $L(\mathbf{T})$ in terms of p and q as:

$$\begin{aligned} L(\mathbf{T}) &= m [p^2 \log(1/p^2) + q^2 \log(1/q^2) + 2pq \log(1/2pq)] = \\ &= 2m [p \log(1/p) + q \log(1/q)] - 2mpq = \\ &= H(\mathbf{S}) - 2mpq - (n\&1) \cdot h(p) \end{aligned} \quad (10)$$

where the factor $(n\&1)$ results from identity $2m + (n\&1) = n$.

e2) Enumerative indices for blocks from \mathbf{E}_2 (as computed in step (d)).

This is the sequence of indices $\mathbf{Z}[m_2]$ of m_2 bits with *equiprobable* symbols 0 and 1, hence it requires *no further encoding*. The average length of this sequence $L(\mathbf{Z})$ obtained by averaging m_2 values via Binomial distribution (eq. (9), $c = 2$) is:

$$L(\mathbf{Z}) = \sum_{\mu=0}^m \mu p_2^\mu q_2^{m-\mu} \binom{m}{\mu} = mp_2 = 2mpq \quad (11)$$

e3) Residual bit* for odd n

For odd values of n there is a residual single bit rb which does not belong to any block \mathbf{B}_j constructed in step (c). The cost $L(1)$ of transmitting rb , encoded by an optimal entropy coder (such as arithmetic coder or QI) is:

$$L(1) = (n\&1) \cdot h(p) \quad (12)$$

where the factor $(n\&1)$ uses bitwise ‘AND’ operator and has values 1 for odd n , 0 for even n . Of course, if rb were the sole item being transmitted in whole number of bits, the cost $L(1)$ would be rounded to the next whole bit, yielding :

$$L(1) = (n\&1) \quad (13)$$

* Although this cost is insignificant for one or two cycle pump, in the full multicycle pump its contribution, due to the $O(n)$ growth in the number of components, grows as $O(n)$. Hence its optimal coding is important.

Since we are sending multiple items, this component-wise whole bit rounding is not necessary (e.g. we can use a convention to encode this bit at the end of the component e1), hence we can use the optimal value given in eq. (12). For the multi-cycle variants of EP, the sequence of such residual bits can be coded optimally as a separate component.

The total EP encoded output size $L(\mathbf{S})$ for the sequence $\mathbf{S}[n]$ is the sum of sizes of the three output components e1-e3, which results in:

$$L(\mathbf{S}) = L(\mathbf{T}) + L(\mathbf{Z}) + L(1) = H(\mathbf{S}) \quad (14)$$

Hence the EP output size $L(\mathbf{S})$ is equal to the entropy $H(\mathbf{S})$. The EP optimality was already shown to hold generally, for any source and any proper (exact) enumerative classes (cf. [1] p. 27). The result (14) is thus a special case of the general EP optimality.

DECODING

Decoder receives the encoded components e1-e3 and, by convention of eq. (1), it ‘knows’ the $\mathbf{S}[n]$ parameters p and n .

ex1) Sequence of class tags $\mathbf{T} = T_1 T_2 \dots T_m$ is decoded from the component e1 (using the matching decoder to the entropy encoder from step (e1) and parameters p and n). The residual single bit is also decoded if n is odd. The length of sequence \mathbf{Z} is obtained as the count of symbols $T_j = 2$.

ex2) Sequence of blocks $\mathbf{B} = \mathbf{B}_1 \mathbf{B}_2 \dots \mathbf{B}_m$ is reconstructed from \mathbf{T} by setting block \mathbf{B}_j to 00 for $T_j = 0$ and to 11 for $T_j = 1$. For $T_j=2$, a bit b is read directly from the received component e2 and the \mathbf{B}_j is set to: $\underline{b} b$, where \underline{b} is a binary complement of b .

ex3) For odd n , the residual bit obtained in (ex1) is appended to the block sequence \mathbf{B} .

EP1 PERFORMANCE

Although the Single Cycle EP still requires a full scale regular entropy coder for coding in step (e1), the principal difference is that the **EP outputs $2mpq$ bits** (cf. eqs. (10), (11)) of the full entropy $H(\mathbf{S})$ at a **very low computational cost**. Namely, the EP1 steps (c) and (d) amount to reading 2 bits at a time as $\mathbf{B}_j=b_1b_2$ from $\mathbf{S}[n]$ (for $j=1\dots m$), then in the case $b_1 \neq b_2$ the bit b_2 is simply stored at the current end of the output array \mathbf{Z} and symbol $T_j=2$ is passed to the regular coder, and for $b_1=b_2$, symbol $T_j=b_1$ is passed to the regular coder.

Therefore, the gain achieved by the single cycle EP is in offloading of the:

$$f(p) = 2mpq/H[\mathbf{S}] = pq/h(p) - O(1/n) \quad (15)$$

fraction of coding/decoding work from the regular modeler+coder and replacing it by a much simpler and quicker coding work (steps (c) & (d)). We call the fraction $f(p)$ the single cycle **pump efficiency** at a given probability p . Table 1 below illustrates single cycle pump efficiencies $f(p)$ for different values of p (figures are in percentages). The maximum efficiency f is 25% (achieved for $p = 50\%$) and the minimum efficiency $f = 0$ is reached in the limit $p \rightarrow 0$.

Note that in the vicinity of both extreme points ($p = 0.5$ or 0) the coding can be simplified further. In the *high entropy limit*, when p is “close enough” to 0.5 (meaning $0.5 - 1/\sqrt{n} \leq p \leq 0.5$) the sequence $\mathbf{S}[n]$ is incompressible, hence we can output it without encoding. In the *low entropy limit* ($p \rightarrow 0$), where the pump efficiency drops to 0 , regular QI coder becomes computationally much more efficient, making the additional use of EP for performance gain unnecessary. Fast run-length coders also become practical in this limit (albeit at a lower compression ratios than the asymptotically optimal coders, such as QI and AC).

Single Cycle Pump Efficiencies

p %	$f(p)$ %	p %	$f(p)$ %	p %	$f(p)$ %	p %	$f(p)$ %
0.5	10.955	13.0	20.289	25.5	23.193	38.0	24.592
1.0	12.254	13.5	20.451	26.0	23.272	38.5	24.626
1.5	13.150	14.0	20.608	26.5	23.349	39.0	24.658
2.0	13.857	14.5	20.760	27.0	23.423	39.5	24.689
2.5	14.452	15.0	20.907	27.5	23.496	40.0	24.718
3.0	14.970	15.5	21.050	28.0	23.567	40.5	24.746
3.5	15.431	16.0	21.188	28.5	23.635	41.0	24.772
4.0	15.849	16.5	21.323	29.0	23.702	41.5	24.797
4.5	16.231	17.0	21.453	29.5	23.766	42.0	24.820
5.0	16.585	17.5	21.580	30.0	23.829	42.5	24.842
5.5	16.915	18.0	21.703	30.5	23.889	43.0	24.863
6.0	17.224	18.5	21.823	31.0	23.948	43.5	24.882
6.5	17.515	19.0	21.940	31.5	24.005	44.0	24.899
7.0	17.791	19.5	22.053	32.0	24.061	44.5	24.915
7.5	18.052	20.0	22.163	32.5	24.114	45.0	24.930
8.0	18.300	20.5	22.270	33.0	24.166	45.5	24.943
8.5	18.537	21.0	22.374	33.5	24.216	46.0	24.955
9.0	18.764	21.5	22.475	34.0	24.264	46.5	24.966
9.5	18.981	22.0	22.574	34.5	24.311	47.0	24.975
10.0	19.190	22.5	22.670	35.0	24.356	47.5	24.983
10.5	19.390	23.0	22.763	35.5	24.399	48.0	24.989
11.0	19.583	23.5	22.854	36.0	24.441	48.5	24.994
11.5	19.769	24.0	22.942	36.5	24.481	49.0	24.997
12.0	19.949	24.5	23.028	37.0	24.519	49.5	24.999
12.5	20.122	25.0	23.112	37.5	24.556	50.0	25.000

Table 1.

EP2: Two Cycles, 2 symbols/block

In this method instead of encoding the class tag sequence $\mathbf{T} = T_1 T_2 \dots T_m$ in step (EP1.e1) via general multi-alphabet coder, we perform *binary decomposition* of ternary sequence \mathbf{T} (cf. [2] pp. 35-39) into two binary sequences, \mathbf{X} and \mathbf{Y} , then apply method EP1 twice to encode \mathbf{X} and \mathbf{Y} optimally, obtaining thus the optimal encoding for \mathbf{T} , hence the optimal encoding for $\mathbf{S}[n]$.

BINARY DECOMPOSITION OF \mathbf{T}

e1.b) To perform binary decomposition ([2] pp. 35-39) of sequence \mathbf{T} , we first assign prefix codes F_j to the tags T_j (for $j=1..m$) based on the tag values as follows:

$$T_j = 0 \Rightarrow F_j = 00 \quad (16a)$$

$$T_j = 1 \Rightarrow F_j = 01 \quad (16b)$$

$$T_j = 2 \Rightarrow F_j = 1 \quad (16c)$$

The mappings (16) map sequence \mathbf{T} into a sequence of prefix codes $\mathbf{F} = F_1 F_2 \dots F_m$. We define below binary sequences \mathbf{X} and \mathbf{Y} and deduce some of their properties:

e1.bx) Binary sequence $\mathbf{X}[n_x] = X_1 X_2 \dots X_{n_x}$, where $n_x = m = \lfloor n/2 \rfloor$, is constructed by setting bit X_j to the *first bit* of F_j , for $j=1..m$. The probabilities of ones p_x and zeros q_x in $\mathbf{X}[n_x]$ are computed in terms of p and q (via eqs. (6)) as:

$$p_x = 2pq \quad \text{and} \quad q_x = p^2 + q^2 \quad (17)$$

Since by our convention $p \leq q$, eq. (17) implies:

$$p \leq p_x \leq q_x \leq q \quad (18)$$

$$p_x/q_x = 2pq / (p^2 + q^2) \geq p/q \quad (19)$$

i.e. the sequence $\mathbf{X}[n_x]$ is *denser* (it has higher entropy density) than the input sequence $\mathbf{S}[n]$.

e1.by) Binary sequence $\mathbf{Y}[n_y] = Y_1 Y_2 \dots Y_{n_y}$ is constructed by setting bit Y_i to the *second bit* of the i -th occurrence of a two bit prefix code (00 or 01) in \mathbf{F} . The number of symbols n_y in $\mathbf{Y}[n_y]$ and its average \underline{n}_y are (using eqs. (7), (9)):

$$n_y = m_0 + m_1 = m - m_2 = \underline{m}_2 \quad (20)$$

$$\underline{n}_y = m \cdot q_x = m \cdot (p^2 + q^2) \quad (21)$$

i.e. the length n_y of $\mathbf{Y}[n_y]$ is the same as the number of zeros in $\mathbf{X}[n_x]$. The probabilities of ones p_y and zeros q_y in $\mathbf{Y}[n_y]$ are computed in terms of p and q (via eqs. (6)) as:

$$p_y = p^2 / (p^2 + q^2) \quad \text{and} \quad q_y = q^2 / (p^2 + q^2) \quad (22)$$

From $p \leq q$ and eq. (22) we deduce relations between the four probabilities:

$$p_y \leq p \leq q \leq q_y \quad (23)$$

$$p_y/q_y = (p/q)^2 \leq p/q \quad (24)$$

i.e. the sequence $\mathbf{Y}[n_y]$ is *sparser* (it has lower entropy density) than the input sequence $\mathbf{S}[n]$. Further, eq. (24) shows that the density of $\mathbf{S}[n]$ drops quadratically in the component $\mathbf{Y}[n_y]$, hence the drop is very fast (exponential function $(p/q)^{2\lambda}$, where λ is the depth of the decomposition chain $\mathbf{Y}_\lambda \rightarrow \mathbf{Y}_{\lambda+1}$).

Table 2 below shows the average ratios of ones to zeros for the arrays $\mathbf{S}[n]$, $\mathbf{X}[n_x]$ and $\mathbf{Y}[n_y]$ for different values of the probability p of ones in $\mathbf{S}[n]$.

Ratios of ones to zeros for arrays $\mathbf{S}[n]$, $\mathbf{X}[n_x]$ and $\mathbf{Y}[n_y]$

p %	p/q	p_x/q_x	p_y/q_y	p %	p/q	p_x/q_x	p_y/q_y
1.0	0.0101	0.0202	0.0001	26.0	0.3514	0.6255	0.1234
2.0	0.0204	0.0408	0.0004	27.0	0.3699	0.6507	0.1368
3.0	0.0309	0.0618	0.0010	28.0	0.3889	0.6756	0.1512
4.0	0.0417	0.0832	0.0017	29.0	0.4085	0.7001	0.1668
5.0	0.0526	0.1050	0.0028	30.0	0.4286	0.7241	0.1837
6.0	0.0638	0.1271	0.0041	31.0	0.4493	0.7476	0.2018
7.0	0.0753	0.1497	0.0057	32.0	0.4706	0.7705	0.2215
8.0	0.0870	0.1726	0.0076	33.0	0.4925	0.7928	0.2426
9.0	0.0989	0.1959	0.0098	34.0	0.5152	0.8142	0.2654
10.0	0.1111	0.2195	0.0123	35.0	0.5385	0.8349	0.2899
11.0	0.1236	0.2435	0.0153	36.0	0.5625	0.8546	0.3164
12.0	0.1364	0.2677	0.0186	37.0	0.5873	0.8734	0.3449
13.0	0.1494	0.2923	0.0223	38.0	0.6129	0.8911	0.3757
14.0	0.1628	0.3172	0.0265	39.0	0.6393	0.9077	0.4088
15.0	0.1765	0.3423	0.0311	40.0	0.6667	0.9231	0.4444
16.0	0.1905	0.3676	0.0363	41.0	0.6949	0.9372	0.4829
17.0	0.2048	0.3931	0.0420	42.0	0.7241	0.9501	0.5244
18.0	0.2195	0.4188	0.0482	43.0	0.7544	0.9616	0.5691
19.0	0.2346	0.4447	0.0550	44.0	0.7857	0.9716	0.6173
20.0	0.2500	0.4706	0.0625	45.0	0.8182	0.9802	0.6694
21.0	0.2658	0.4966	0.0707	46.0	0.8519	0.9873	0.7257
22.0	0.2821	0.5225	0.0796	47.0	0.8868	0.9928	0.7864
23.0	0.2987	0.5485	0.0892	48.0	0.9231	0.9968	0.8521
24.0	0.3158	0.5743	0.0997	49.0	0.9608	0.9992	0.9231
25.0	0.3333	0.6000	0.1111	50.0	1.0000	1.0000	1.0000

Table 2.

EP2 ENCODING

The encoding proceeds as in EP1, except that the output component EP1.e1 is replaced by the two components: $\mathbf{X}[n_x]$ and $\mathbf{Y}[n_y]$ encoded using method EP1. Each of these components will contain 3 sub-components (from the application of EP1), resulting in the total of $2+1+1=4$ components (or $2+3+3=8$ sub-components). In order to avoid bit fraction loss on the boundaries between the components, the component size should not be rounded to the next whole bit. If the entropy coder used in the step (EP1.e1) is AC, the components should be coded into a single stream (with different probabilities). If QI is used, the bit fractions between components should be removed using mixed radix codes (cf. [2] pp. 46-52 and note N4 in [3] p. 9).

e1.X) Encoding of $\mathbf{X}[n_x]$ via EP1

Since the number of symbol $n_x = m = \lfloor n/2 \rfloor$ and the probabilities p_x and q_x for $\mathbf{X}[n_x]$ are known to decoder (via eq. (17) and the known p and q), we don't need to transmit n_x, p_x and q_x . Hence the encoded output length $L(\mathbf{X})$ produced by EP1(\mathbf{X}) will be the entropy $H(\mathbf{X}) = m h(p_x)$.

e1.Y) Encoding of $\mathbf{Y}[n_y]$ via EP1

The number of symbols n_y in $\mathbf{Y}[n_y]$ is the same as the count of zeros in $\mathbf{X}[n_x]$, therefore we don't need to transmit n_y separately. Further, from eqs. (22), p_y and q_y are computed from the known probabilities p and q , hence p_y and q_y need not be transmitted either. Consequently, the encoding EP1(\mathbf{Y}) will result in the encoded length $L(\mathbf{Y}) = H(\mathbf{Y})$.

The combined output size of components e1.X and e1.Y is then:

$$L(\mathbf{X}) + L(\mathbf{Y}) = H(\mathbf{X}) + H(\mathbf{Y}) = H(\mathbf{T}) \quad (25)$$

where the last identity in (25) follows from the optimality of our binary decomposition (cf. multinomial factorization into products of binomials [2] pp. 32-34).

EP2 DECODING

The decoder receives 4 components, e1.X, e1.Y, e2 and e3. The component e1.X, which contains 3 sub-components, is decoded by computing p_x and q_x from known p and q using eq. (17), then applying the EP1 decoder. The decoded array $\mathbf{X}[n_x]$ is then used to compute n_y (the expanded length of array $\mathbf{Y}[n_y]$) by setting n_y equal the count of zeros in $\mathbf{X}[n_x]$. The probabilities p_y and q_y are computed using eq. (22). With the known length n_y and the probabilities p_y and q_y , the EP1 decoder can decode the encoded sequence $\mathbf{Y}[n_y]$.

With the decoded sequences $\mathbf{X}[n_x]$ and $\mathbf{Y}[n_y]$, EP2 decoder reconstructs the tag sequence \mathbf{T} by reversing the mappings (16). Specifically, decoder sets the bit index i for $\mathbf{Y}[n_y]$, $i = 0$, then it loops through the bits X_j from $\mathbf{X}[n_x]$, for $j=1..n_x$. If the current X_j is 1 then T_j is set to 2, otherwise T_j is set to the bit value of Y_i and the index i is incremented.

The remaining steps of the EP2 method decode the components e2 and e3 as in EP1, then reconstruct array $\mathbf{S}[n]$ from the decoded components e1-e3 as in EP1 decoder.

EP2 PERFORMANCE

The principal difference between EP1 and EP2 is that EP2 algorithm is slightly more complex to implement in return for faster performance. Namely EP2 pumps out inexpensively additional entropy from the components $\mathbf{X}[n_x]$ and $\mathbf{Y}[n_y]$, which EP1 had to encode as ternary sequence $\mathbf{T}[m]$ via regular entropy coder. Using the pump efficiency formula, eq. (15), to arrays $\mathbf{X}[n_x]$ and $\mathbf{Y}[n_y]$, in addition to $\mathbf{Z}[m_2]$ from the step (EP2.e2), we obtain the average pump efficiency $f_2(p)$ for EP2 as:

$$\begin{aligned} f_2(p) &= (2mpq + 2 \lfloor n_x/2 \rfloor p_x q_x + 2 \lfloor n_y/2 \rfloor p_y q_y) / H(\mathbf{S}) = (1 + q_x + pq/2q_x) pq/h(p) = \\ &= (1 + q_x + pq/2q_x) \cdot f_1(p) \geq 1.75 f_1(p) \end{aligned} \quad (26)$$

where $f_1(p)$ is the pump efficiency of EP1 from eq. (15). Table 3 shows $f_2(p)$, $f_1(p)$ and their ratios for different values of p . The ratios f_2/f_1 range between 1.75 and 2.

Comparison of EP1 and EP2 Pump Efficiencies

p %	f_1 %	f_2 %	f_2/f_1	p %	f_1 %	f_2 %	f_2/f_1
1.0	12.254	24.326	1.98525	26.0	23.272	41.228	1.77157
2.0	13.857	27.313	1.97100	27.0	23.423	41.424	1.76848
3.0	14.970	29.300	1.95725	28.0	23.567	41.611	1.76570
4.0	15.849	30.810	1.94400	29.0	23.702	41.791	1.76323
5.0	16.585	32.030	1.93124	30.0	23.829	41.963	1.76103
6.0	17.224	33.053	1.91899	31.0	23.948	42.128	1.75911
7.0	17.791	33.931	1.90722	32.0	24.061	42.285	1.75743
8.0	18.300	34.696	1.89595	33.0	24.166	42.435	1.75599
9.0	18.764	35.374	1.88517	34.0	24.264	42.578	1.75476
10.0	19.190	35.979	1.87488	35.0	24.356	42.713	1.75372
11.0	19.583	36.524	1.86507	36.0	24.441	42.841	1.75285
12.0	19.949	37.019	1.85574	37.0	24.519	42.962	1.75214
13.0	20.289	37.472	1.84688	38.0	24.592	43.074	1.75157
14.0	20.608	37.888	1.83849	39.0	24.658	43.179	1.75112
15.0	20.907	38.272	1.83057	40.0	24.718	43.276	1.75077
16.0	21.188	38.629	1.82310	41.0	24.772	43.364	1.75051
17.0	21.453	38.961	1.81609	42.0	24.820	43.443	1.75032
18.0	21.703	39.273	1.80951	43.0	24.863	43.514	1.75019
19.0	21.940	39.565	1.80337	44.0	24.899	43.576	1.75010
20.0	22.163	39.841	1.79765	45.0	24.930	43.629	1.75005
21.0	22.374	40.102	1.79234	46.0	24.955	43.672	1.75002
22.0	22.574	40.349	1.78743	47.0	24.975	43.706	1.75001
23.0	22.763	40.585	1.78292	48.0	24.989	43.731	1.75000
24.0	22.942	40.809	1.77878	49.0	24.997	43.745	1.75000
25.0	23.112	41.023	1.77500	50.0	25.000	43.750	1.75000

Table 3.

EP3: Multicycle, 2 symbols/block

In order to extend the single recursive step of EP2 to the general multicycle EP3, we represent graphically the processing steps and data flows of methods EP1 (Figs. 1,2) and EP2 (Fig. 3). The *processing blocks* EC:3 and EC:2 denote ternary and binary entropy coders while EP1:3 and EP1:2 denote the EP1 variants using EC:3 and EC:2 as entropy coders in step (EP1.e1). To streamline the diagrams, we omit depictions of the component e3 (*residual bit* for odd n), with understanding that e3 is passed to EC after EC has encoded e1 component (*without flushing* the EC state in between e1 and e3 encodings).

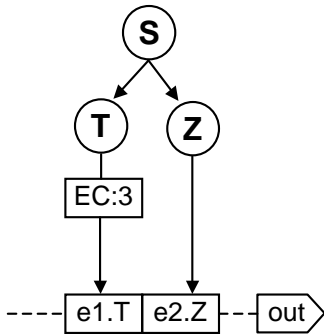


Fig. 1: EP1:3

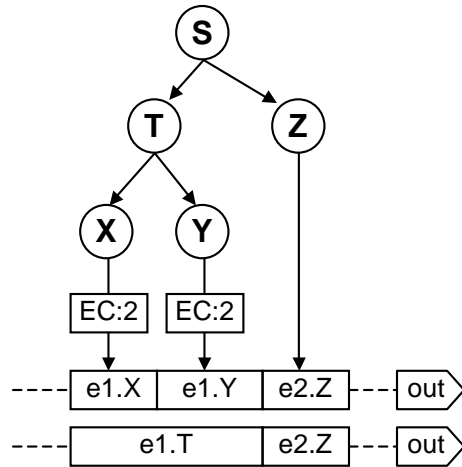


Fig. 2: EP1:2

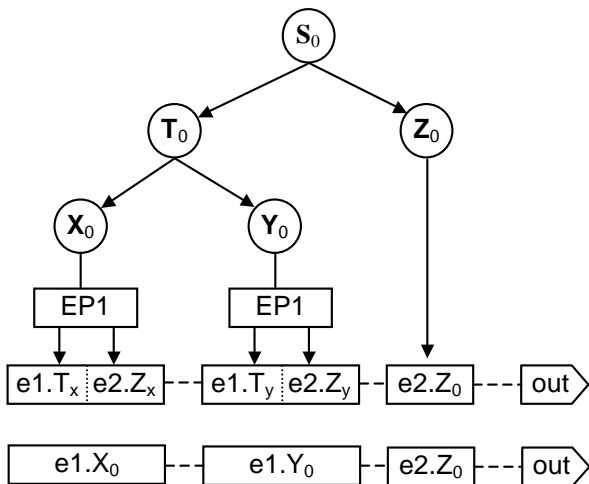


Fig. 3: EP2

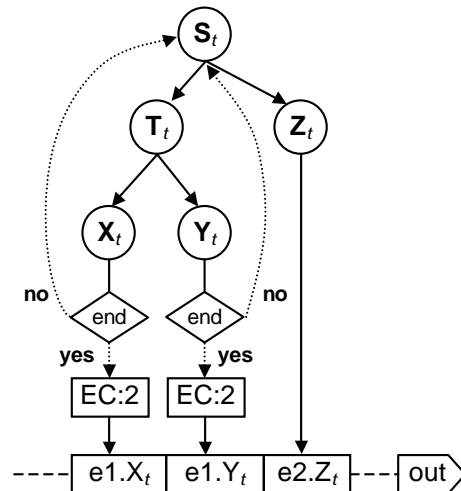


Fig. 4: EP3

The layouts of the serialized outputs (output streams) are depicted in the rows containing the **|out>** block. In Figs. 2 and 3 we also show alternative (higher level) depictions of the output streams as the lower **|out>** rows.

In Fig. 3 depicting EP2, we use component subscripts 0, x and y to distinguish the three EP input sequences \mathbf{S}_0 , \mathbf{S}_x and \mathbf{S}_y (as well as their respective outputs), where $\mathbf{S}_0 \equiv \mathbf{S}$ (the original input sequence), $\mathbf{S}_x \equiv \mathbf{X}_0$ and $\mathbf{S}_y \equiv \mathbf{Y}_0$.

Fig. 4 represents the *general recursive EP method* (for 2-symbol blocks), EP3, depicted as an extension of the EP1:2 method (Fig. 2). The essential difference between EP1:2 and EP3 is that EP3 introduces *termination decision test* (denoted as **<end>**) on the current data, right before passing the data to the entropy coders **EC:2**, which yields an answer “yes” or “no”. The two data paths taken after an **<end>** test are labeled as:

- “yes” path – end recursion and pass current data to the **EC:2**
- “no” path – continue recursion: process the current data as the new input \mathbf{S}_t .

A simple type of **<end>** test is to check the current *depth of recursion** λ against some *maximum depth* λ_{\max} and return “yes” if $\lambda \geq \lambda_{\max}$ and “no” otherwise. The special cases of $\lambda_{\max} = 0$ and $\lambda_{\max} = 1$ are equivalent to the EP1:2 and EP2 methods. While simple to implement, this type of **<end>** test is insensitive to the data being processed and setting λ_{\max} to a too large value may hinder performance due to the processing overhead for a large number of small components. A better decision whether to end recursion should in some form compare the savings (in the amount of work) from outputting the entropy via the inexpensive components \mathbf{Z} with the processing costs of managing multiple components of decreasing sizes (as the recursion depth grows). Another element to take into account is the cost of computing the **<end>** test itself i.e. this computation should be simple. The following three part **<end>** test, with its “yes” criteria (c1)-(c3) provides a reasonable practical balance with 3-parameter flexibility to adapt to various input data and processing environments:

- c1)** if ($p' > p_{\max}$) \rightarrow **<end>** = **yes**, else do (c2)
- c2)** if ($n' \leq n_{\min}$) \rightarrow **<end>** = **yes**, else do (c3)
- c3)** if ($p' \leq p_{\min}$) \rightarrow **<end>** = **yes**, else **<end>** = **no**.

where n' and p' are size and probability of ones (the less frequent symbol) for the data presented to the test and n_{\min} , p_{\min} and p_{\max} are algorithm adjustment parameters, (adjusted to a given computing environment and redundancy requirements). The individual tests (c1), (c2) and (c3) are performed in succession: the next one is evaluated only if the previous one returns ‘no’ (this is a short-circuited logical OR operator, such as **||** operator in C/C++ or Java).

Further, if the test (c1) is evaluated and it returns “yes” (to end recursion) i.e. $p' > p_{\max}$, the *data* is considered approximately *incompressible*, hence it is copied directly into the corresponding output component e1, skipping the entropy coding step **EC:2**. In order for this

* The “*depth of recursion*” λ is a variable attached to each node \mathbf{S}_i in a tree of \mathbf{S}_i nodes generated by the EP3 algorithm. Value $\lambda = \lambda(\mathbf{S}_i)$ is a ‘tree distance’ (# of parent-child hops) of \mathbf{S}_i from the root node \mathbf{S}_0 .

“incompressibility” assumption of (c1) to hold as a good approximation, the value p_{\max} should be set to satisfy the inequality*:

$$\frac{1}{2} - \frac{1}{\sqrt{n'}} \leq p_{\max} \leq \frac{1}{2} \quad (27)$$

The criteria (c2) and (c3) are dependent on the properties of the entropy coder **EC:2**. The (c3) parameter p_{\min} depends on the availability of fast coding modes for sparse data (such as run length coding) and (c2) parameter n_{\min} depends on whether the coder can gain efficiency for small enough input data (such as small quantized binomial tables for QI).

Tables 3 and 4 on the following pages show the EP3 performance properties for input sequence $\mathbf{S}[n]$ of length $n = 1 \text{ Mbit} = 2^{20}$ bits and for the recursion termination parameters p_{\min} , p_{\max} and n_{\min} as shown in the tables titles. The only difference between the tables is that Table 3 uses $n_{\min}=256$ and Table 4 uses $n_{\min}=1024$. Hence Table 3 shows gains in pump efficiencies f in return for larger redundancies and larger number of components.

Tables 4,5 columns:

p %	Probability of ones in the input sequence $\mathbf{S}[n]$: $p \cdot 100$
R %	EP output redundancy: $(L(\mathbf{S})/H(\mathbf{S})-1) \cdot 100$
R -bits	Excess bits in EP output: $L(\mathbf{S})-H(\mathbf{S})$
Depth	Maximum depth of recursion reached
$\#\mathbf{Z}$	Number of \mathbf{Z} components produced
$\#\text{Hi}$	Number of components output directly due to $p' \geq p_{\max} = 0.5 - 0.5/\sqrt{n}$
$\#\text{EC}$	Number of components encoded via regular entropy coder
$\#\text{Comp}$	Total number of (sub-)components output
f %	Overall pump efficiency = $100 \cdot [L(\mathbf{Z})+L(\text{Hi})]/H(\mathbf{S})$

The excess bits (R -bits) shown in Tables 3 and 4 arise from the two contributions:

- Bit fraction loss for the termination under the “incompressibility” condition (c1), where we copy the whole number of input bits into the output component.
- Bit fraction loss from the components of type \mathbf{Z} , which were stored into the next whole number of bits when the data size n_d is odd number (hence we have the non-integer number of blocks \mathbf{B}_j). This simplification is not necessary and the resulting small loss can be eliminated at some computational cost using the optimal coding of the ‘residual bit’ (cf. eq. (12) for component e3 coding).

Since these fractional bit losses (a) and (b) will have an average excess of 0.5 bits per component, the R -bits values shown are approximately $0.5 \cdot (\#\text{Hi} + \#\mathbf{Z})$.

* The lower bound in eq. (27) is based on the std. deviation σ for binomial distribution: $\sigma = (npq)^{1/2}$.

Performance of EP3 ($n=2^{20}$ bits, $n_{\min}=256$, $p_{\min}=1/n$, $p_{\max}=0.5-0.5/\sqrt{n}$)

	p %	R %	R-bits	Depth	#Z	#Hi	#EC	#Comp.	f %
1	0.1042	+88.299	13	155	5	151	311	94.3671	
2	0.0748	+110.921	13	208	9	200	417	95.6327	
3	0.0609	+124.153	13	244	12	233	489	95.9836	
4	0.0542	+137.822	13	266	14	253	533	96.3023	
5	0.0532	+159.627	13	290	16	275	581	96.6367	
6	0.0462	+158.708	13	294	17	278	589	96.8002	
7	0.0403	+154.518	13	298	21	278	597	96.9717	
8	0.0421	+177.429	13	313	24	290	627	97.2448	
9	0.0384	+175.608	13	308	22	287	617	97.2918	
10	0.0369	+181.458	13	320	26	295	641	97.4998	
11	0.0348	+182.636	13	320	28	293	641	97.6181	
12	0.0306	+169.914	13	321	27	295	643	97.6742	
13	0.0320	+187.185	13	330	28	303	661	97.7714	
14	0.0289	+176.960	13	332	29	304	665	97.7477	
15	0.0289	+184.500	13	326	30	297	653	97.9401	
16	0.0280	+185.959	13	328	30	299	657	97.9231	
17	0.0267	+184.258	13	326	28	299	653	97.9448	
18	0.0256	+182.339	13	312	28	285	625	98.0844	
19	0.0228	+167.612	13	310	31	280	621	98.2378	
20	0.0237	+179.784	13	313	31	283	627	98.1886	
21	0.0221	+171.573	13	311	32	280	623	98.2286	
22	0.0208	+165.440	13	302	32	271	605	98.3249	
23	0.0195	+159.420	13	283	31	253	567	98.4273	
24	0.0199	+165.989	13	286	34	253	573	98.4626	
25	0.0187	+159.106	13	275	34	242	551	98.4943	
26	0.0193	+167.004	12	268	34	235	537	98.5282	
27	0.0171	+150.849	12	258	33	226	517	98.6490	
28	0.0169	+151.787	12	249	34	216	499	98.6203	
29	0.0155	+141.229	12	241	34	208	483	98.7145	
30	0.0146	+134.703	12	231	29	203	463	98.7729	
31	0.0137	+128.631	12	221	33	189	443	98.8632	
32	0.0132	+124.995	12	210	31	180	421	98.9249	
33	0.0129	+124.213	12	207	31	177	415	98.9665	
34	0.0118	+114.335	12	202	32	171	405	98.9839	
35	0.0109	+106.953	12	182	33	150	365	99.0791	
36	0.0117	+115.687	12	177	29	149	355	99.1298	
37	0.0103	+102.207	12	162	33	130	325	99.2080	
38	0.0094	+93.933	11	151	30	122	303	99.1987	
39	0.0084	+84.564	11	140	31	110	281	99.3069	
40	0.0075	+76.037	11	123	27	97	247	99.3690	
41	0.0071	+73.063	11	119	28	92	239	99.3842	
42	0.0068	+70.489	11	107	27	81	215	99.4806	
43	0.0064	+66.117	11	97	23	75	195	99.5118	
44	0.0049	+50.701	11	80	21	60	161	99.6022	
45	0.0039	+41.119	10	68	19	50	137	99.6442	
46	0.0034	+35.094	10	56	19	38	113	99.7088	
47	0.0030	+31.191	10	43	18	26	87	99.7928	
48	0.0023	+23.983	9	37	18	20	75	99.8383	
49	0.0016	+16.604	9	18	11	8	37	99.9207	
50	0.0000	+0.000	1	0	1	0	1	100.0000	

Table 4.

Performance of EP3 ($n=2^{20}$ bits, $n_{\min}=1024$, $p_{\min}=1/n$, $p_{\max}=0.5-0.5/\sqrt{n}$)

p %	R %	R -bits	Depth	#Z	#Hi	#EC	#Comp.	f %
1	0.0557	+47.170	11	88	2	87	177	89.8520
2	0.0411	+60.993	11	116	3	114	233	91.4174
3	0.0323	+65.933	11	137	5	133	275	92.2347
4	0.0300	+76.144	11	150	4	147	301	92.4038
5	0.0290	+87.115	11	162	6	157	325	93.1657
6	0.0257	+88.203	11	163	8	156	327	93.8155
7	0.0209	+80.140	11	163	9	155	327	93.8793
8	0.0218	+92.060	11	170	9	162	341	94.2437
9	0.0203	+92.868	11	166	9	158	333	94.5927
10	0.0189	+92.957	11	174	11	164	349	94.7963
11	0.0182	+95.225	11	174	11	164	349	94.9166
12	0.0159	+88.123	11	173	12	162	347	95.2046
13	0.0164	+95.727	11	177	12	166	355	95.2582
14	0.0153	+93.859	11	179	13	167	359	95.3222
15	0.0149	+95.568	11	176	12	165	353	95.4172
16	0.0145	+96.540	11	175	13	163	351	95.5770
17	0.0142	+98.172	11	176	12	165	353	95.7733
18	0.0130	+92.854	11	167	13	155	335	96.0328
19	0.0121	+89.040	11	168	14	155	337	96.2080
20	0.0123	+92.741	11	168	14	155	337	96.0958
21	0.0111	+86.330	11	164	16	149	329	96.3353
22	0.0108	+86.137	11	159	16	144	319	96.4517
23	0.0107	+87.636	11	152	16	137	305	96.7224
24	0.0109	+91.024	11	154	17	138	309	96.6605
25	0.0094	+79.879	11	143	17	127	287	96.7771
26	0.0096	+83.422	10	138	20	119	277	96.9143
27	0.0087	+76.586	10	134	17	118	269	97.0477
28	0.0086	+77.186	10	131	18	114	263	96.9889
29	0.0081	+74.088	10	128	15	114	257	97.0558
30	0.0074	+68.533	10	117	16	102	235	97.4763
31	0.0069	+64.755	10	113	18	96	227	97.5592
32	0.0070	+66.074	10	107	16	92	215	97.6240
33	0.0071	+68.165	10	107	17	91	215	97.7651
34	0.0061	+59.466	10	107	17	91	215	97.7375
35	0.0058	+57.119	10	97	19	79	195	97.9769
36	0.0060	+59.445	10	93	17	77	187	98.0810
37	0.0059	+58.916	10	89	18	72	179	98.1394
38	0.0045	+45.110	9	79	17	63	159	98.1826
39	0.0043	+43.997	9	75	19	57	151	98.4048
40	0.0040	+40.751	9	66	16	51	133	98.5568
41	0.0037	+38.370	9	64	16	49	129	98.5319
42	0.0036	+37.264	9	58	15	44	117	98.6892
43	0.0035	+35.758	9	52	13	40	105	98.8126
44	0.0026	+27.147	9	44	12	33	89	99.0124
45	0.0023	+23.995	8	38	12	27	77	99.1592
46	0.0021	+21.887	8	34	12	23	69	99.2502
47	0.0019	+19.506	8	27	12	16	55	99.4392
48	0.0015	+15.847	8	23	13	11	47	99.5575
49	0.0011	+11.490	7	12	8	5	25	99.7877
50	0.0000	+0.000	1	0	1	0	1	100.0000

Table 5.

EP3 ENCODING

Although the processing steps of EP3 consist merely of multiple applications of the EP2 recursion, in order to describe the EP3 algorithm we need a more general formulation of the recursion step and the output component layout. The core of the EP3 engine is the 5 node *tree* \mathbf{G}_t depicted in Fig. 4:

$$\mathbf{G}_t \equiv \{ \mathbf{S}_t, \mathbf{T}_t, \mathbf{X}_t, \mathbf{Y}_t, \mathbf{Z}_t \} \quad (28)$$

The nodes of \mathbf{G}_t containing binary arrays are \mathbf{S}_t , \mathbf{X}_t , \mathbf{Y}_t and \mathbf{Z}_t . Each of these four nodes is characterized by the binary array length $n_g(t)$ and its probabilities of ones $p_g(t)$ and zeros $q_g(t) = 1 - p_g(t)$, where the node subscript $g = s, x, y$ or z . The relations between these four sets of array parameters $\{ n_g(t), p_g(t), q_g(t) \}$ for $g = s, x, y$ or z were already given within the description of EP2. In the generalized notation of EP3 (and omitting the label t which is common to all parameters) these relations become:

$\mathbf{S}_t[n_s] \rightarrow \mathbf{X}_t[n_x]$ via (e1.bx)

$$n_x = \lfloor n_s/2 \rfloor \quad (29a)$$

$$p_x = 2 p_s q_s \quad (29b)$$

$$q_x = 1 - p_x = p_s^2 + q_s^2 \quad (29c)$$

$\mathbf{S}_t[n_s] \rightarrow \mathbf{Y}_t[n_y]$ via (e1.by)

$$n_y = \# \text{ of zeros in } \mathbf{X}_t[n_x] \quad (30a)$$

$$p_y = p_s^2 / (p_s^2 + q_s^2) \quad (30b)$$

$$q_y = q_s^2 / (p_s^2 + q_s^2) \quad (30c)$$

$\mathbf{S}_t[n_s] \rightarrow \mathbf{Z}_t[n_z]$ via (e2)

$$n_z = \# \text{ of ones in } \mathbf{X}_t[n_x] \quad (31a)$$

$$p_z = 1/2 \quad (31b)$$

$$q_z = 1/2 \quad (31c)$$

Eqs. (29)-(31) show that “content” of the \mathbf{G}_t nodes \mathbf{X}_t , \mathbf{Y}_t and \mathbf{Z}_t is *built up* uniquely from the “content” of the \mathbf{G}_t root node \mathbf{S}_t (where the node “*content*” is defined as the node array and its parameters n, p and q). Further, while the \mathbf{X}_t parameters are computed using only the \mathbf{S}_t parameters, the parameters of \mathbf{Y}_t and \mathbf{Z}_t require also symbol counts of $\mathbf{X}_t[n_x]$.

The *recursion mechanism* of EP3 consists of the two “no”-paths, each going from the “no” output of its <end> test (applied to $\mathbf{X}_t[n_x]$ or $\mathbf{Y}_t[n_y]$) and looping back to the root of \mathbf{G}_t , which is node \mathbf{S}_t (see Fig. 4). This loopback is a graphical representation of the *generation of new trees* \mathbf{G}_t' from the tree \mathbf{G}_t . The initial tree \mathbf{G}_0 is defined in terms of the main input $\mathbf{S}[n]$ of EP3 and its parameters n, p and q as follows:

Initial Tree \mathbf{G}_0

$$n_s(0) = n, \quad p_s(0) = p, \quad q_s(0) = q \quad (32a)$$

$$\mathbf{S}_0[n_s(0)] = \mathbf{S}[n] \quad (32b)$$

The remaining parameters $n_g(0), p_g(0), q_g(0)$ for $g = x, y$ and z are computed via eqs. (29)-(31). Each “no”-path in Fig. 4 taken within some tree \mathbf{G}_t **generates a new tree** $\mathbf{G}_{t'}$, where t' is the **next available t -label** value (t -label is a global variable which is initialized to 0 and incremented each time a new $\mathbf{G}_{t'}$ tree is generated, taking thus values $t = 0, 1, \dots, t_{\max}$). Eqs. (33)-(34) show that the new tree $\mathbf{G}_{t'}$ is obtained from \mathbf{G}_t by **replicating the content** of \mathbf{X}_t or \mathbf{Y}_t (based on the “no”-path taken) as the content of $\mathbf{S}_{t'}$, then rebuilding the rest of $\mathbf{G}_{t'}$ from $\mathbf{S}_{t'}$ via eqs. (29)-(31).

Generation $\mathbf{G}_t \rightarrow \mathbf{G}_{t'}$ via \mathbf{X}_t “no”-path

$$n_s(t') = n_x(t) \quad (33a)$$

$$p_s(t') = p_x(t) \quad (33b)$$

$$q_s(t') = q_x(t) \quad (33c)$$

$$\mathbf{S}_{t'}[n_s(t')] = \mathbf{X}_t[n_x(t)] \quad (33d)$$

$$\mathbf{S}_{t'} \rightarrow \mathbf{X}_{t'}, \mathbf{Y}_{t'}, \mathbf{Z}_{t'} \text{ via (29)-(31)} \quad (33e)$$

Generation $\mathbf{G}_t \rightarrow \mathbf{G}_{t'}$ via \mathbf{Y}_t “no”-path

$$n_s(t') = n_y(t) \quad (34a)$$

$$p_s(t') = p_y(t) \quad (34b)$$

$$q_s(t') = q_y(t) \quad (34c)$$

$$\mathbf{S}_{t'}[n_s(t')] = \mathbf{Y}_t[n_y(t)] \quad (34d)$$

$$\mathbf{S}_{t'} \rightarrow \mathbf{X}_{t'}, \mathbf{Y}_{t'}, \mathbf{Z}_{t'} \text{ via (29)-(31)} \quad (34e)$$

Since each \mathbf{G}_t can give rise to maximum two trees, \mathbf{G}_t^x (via (33)) and \mathbf{G}_t^y (via (34)), we can represent the **full recursion** of EP3 as a **binary tree** $\Gamma = \Gamma(\mathbf{S}[n], n, p, n_{\min}, p_{\min}, p_{\max})$, with \mathbf{G}_t (for $t = 0, 1, \dots, t_{\max}$) as its nodes*. Construction of the Γ tree and of the mapping $\{\mathbf{G}_t: \text{for } t = 0, 1, \dots, t_{\max}\} \rightarrow \Gamma$ is accomplished as follows:

* To avoid the notation clutter with formal distinctions between a “tree \mathbf{G}_t ” and the “corresponding Γ node $N(\mathbf{G}_t)$ ”, we use a single symbol \mathbf{G}_t for both, distinguishing them by context or by prefix ‘tree’/‘node’.

Construction* of Γ tree

(35)

- a) Set G_0 as the root of Γ .
- b) For each node G_t of Γ , its left child is set to G_t^x and right child to G_t^y if **both** G_t^x and G_t^y were generated by G_t .
- c) Any Γ node G_t which generates neither G_t^x nor G_t^y is a leaf of Γ .
- d) If a Γ node G_t has only a single left child (G_t^x) we add node (of G_t tree) Y_t as the right child of the Γ node G_t . This Y_t is a leaf of Γ .
- e) If a Γ node G_t has only a single right child (G_t^y) we add node (of G_t tree) X_t as the left child of the Γ node G_t . This X_t is a leaf of Γ .

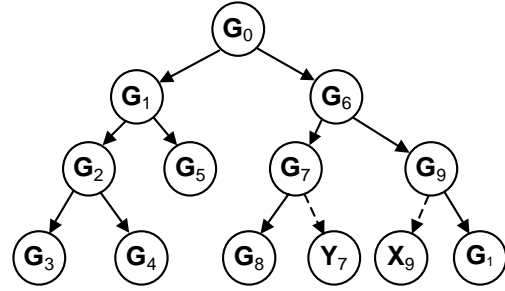
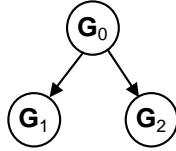


Fig. 5

Fig. 6

Fig. 7

Figs. 5, 6 and 7 depict Γ trees for EP1, EP2 and a 13 node (depth 3) Γ tree for EP3. The assignments of labels t to Γ nodes G_t in Fig. 7 is done in depth-first order (*preorder*[†]). The nodes Y_7 and X_9 are generated by rules (35d) and (35e) i.e. they do not use new t values but retain t values of their parent G_t node.

The encoder processing within each G_t tree, which computes nodes X_t, Y_t, Z_t and the “residual bit” rb_t from the root node S_t , is identical to that described in steps (e1.bx), (e1.by), (e2), (e3) of EP2 and eqs. (29)-(31). Unlike the EP2 case, the presence of an arbitrary number of G_t trees in EP3 requires (due to the constraints of decoder logistics) a more systematic approach to the output component layout than was needed for EP2.

* Γ tree is not explicitly created or stored in memory. It is implicit in the recursive function calls creating G_t trees via eqs. (29)-(34) i.e. Γ tree is a formal characterization of these calls performing eqs. (29)-(34).

† To generate *preorder* labels, imagine a worm crawling around the tree without crossing any lines, starting at the root G_0 , taking left and visiting each node until reaching the root again. The *preorder* labels are assigned sequentially to the G_t nodes on the first visit, while the *postorder* labels are assigned on the last visit, to a given G_t node. The t values are used as array indices to allocate and access G_t content.

EP3 COMPONENT LAYOUT

The principal decoder constraint is that it cannot reconstruct arrays $\mathbf{Y}_t[n_y]$ and $\mathbf{Z}_t[n_z]$ (which are needed to produce array $\mathbf{S}_t[n_s]$ and move up the $\mathbf{\Gamma}$ tree) until it has reconstructed array $\mathbf{X}_t[n_x]$, due to eqs. (30a) and (31a) where the n_y and n_z are computed as the counts of zeros and ones in $\mathbf{X}_t[n_x]$. Therefore, for a tree \mathbf{G}_t , decoder cannot know the parameters n_y and n_z of nodes \mathbf{Y}_t and \mathbf{Z}_t until it has decoded node \mathbf{X}_t and produced $\mathbf{X}_t[n_x]$. Since the parameter n_y is used by the $\langle \text{end} \rangle$ test criteria (c1)-(c3) applied to \mathbf{Y}_t , decoder cannot know upfront the complete layout of the $\mathbf{\Gamma}$ tree (computed via eqs. (29)-(34), the mapping $\{ \mathbf{G}_t \} \rightarrow \mathbf{\Gamma}$ and the $\langle \text{end} \rangle$ test criteria (c1)-(c3)) from the known $\mathbf{S}[n]$ parameters n , p and q and the agreed upon $\langle \text{end} \rangle$ test parameters p_{\max} , p_{\min} and n_{\min} . The only branch decoder can completely know upfront is the leftmost sequence of \mathbf{G}_t^x nodes, since its n_x values are computed by successive halving of n (cf. eq. 29a). Decoder can also compute upfront all of the $p_g(t)$ values, for $g=s,x,y,z$ via (29b,c)-(31b,c).

Therefore, the encoder output layout has to provide components which allow decoder to reconstruct the structure of $\mathbf{\Gamma}$ tree incrementally. To obtain such layout we need to revisit the \mathbf{G}_t output components in the $\mathbf{\Gamma}$ tree context.

The \mathbf{G}_t nodes produce up to 4 output components (see Fig 4): $e1.\mathbf{X}_t$, $e1.\mathbf{Y}_t$, $e2.\mathbf{Z}_t$ and $e3.rb_t$. The component $e2.\mathbf{Z}_t$ is produced by all \mathbf{G}_t nodes unconditionally. The component $e3.rb_t$ is produced by \mathbf{G}_t nodes with odd $n_s(t)$. The components $e1.\mathbf{X}_t$ or $e1.\mathbf{Y}_t$ are produced by \mathbf{G}_t nodes which have “yes” outcome of the \mathbf{X}_t or $\mathbf{Y}_t \langle \text{end} \rangle$ test. Hence, these are the \mathbf{G}_t nodes which have fewer than 2 child nodes, such as leaves \mathbf{G}_3 , \mathbf{G}_4 , \mathbf{G}_5 , \mathbf{G}_8 , \mathbf{G}_{10} and the single child (non-leaf) nodes \mathbf{G}_7 and \mathbf{G}_9 in Fig. 7. The $\mathbf{\Gamma}$ rules (35d), (35e) allow us to remove this non-uniformity for the single-child nodes by extending any such $\mathbf{\Gamma}$ node into a two child node. Hence, the components $e1.\mathbf{Y}_7$ and $e1.\mathbf{X}_9$ are produced by the leaf nodes \mathbf{Y}_7 and \mathbf{X}_9 of the $\mathbf{\Gamma}$ tree of Fig 7. Thus we can say uniformly that the $e1.\mathbf{X}_t$ and $e1.\mathbf{Y}_t$ output components are produced only by the leaf nodes of the $\mathbf{\Gamma}$ tree.

Recalling the EP2 decoding procedure, EP3 decoder will need to decode $e1.\mathbf{X}_t$ of the leftmost $\mathbf{\Gamma}$ leaf (which is a \mathbf{G}_t leaf or \mathbf{X}_t leaf) first, since that is the only $\mathbf{\Gamma}$ node for which decoder will have all parameters $p_x(t)$, $q_x(t)$ and $n_x(t)$ (computed via eqs. (29), (32), (33) and criteria (c1)-(c3), knowing upfront the $\mathbf{S}[n]$ parameters n , p , q and the formulas for p_{\min} , p_{\max} and n_{\min}). From the obtained $\mathbf{X}_t[n_x]$, it can compute $n_y(t)$, $n_z(t)$ which allow it to check for the presence of the right child of \mathbf{G}_t and decode \mathbf{Y}_t and \mathbf{Z}_t . Then, from the decoded $\mathbf{X}_t[n_x]$, $\mathbf{Y}_t[n_y]$ and $\mathbf{Z}_t[n_z]$ it can compute $\mathbf{S}_t[n_s]$ (see EP2 decoder), which in turn allows it to recreate $\mathbf{X}_t''[n_x]$ or $\mathbf{Y}_t''[n_y]$ (by reversing copying direction of eqs. (33d) or (34d)) of the \mathbf{G}_t 's parent node \mathbf{G}_t'' .

This type of tree traversal (visit left subtree, then right subtree, then root) is known as the *postorder traversal* (see Appendix A for an explicit construction algorithm). It can be easily visualized via the ‘*crawling worm*’ model: the worm starts at the root and goes left and around the tree without crossing any lines. It assigns the next available sequence number to a node when that node is encountered *for the last time*. The Fig. 8 shows the postorder visit sequence numbers for our Γ example of Fig. 7.

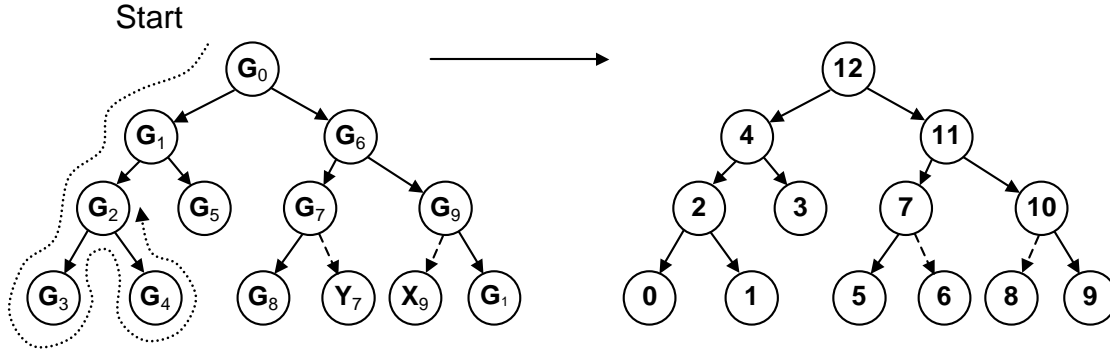


Fig. 7

Fig. 8

Therefore, the EP3 encoder in the example of Fig 7, outputs the encoded components in the order shown in Fig 8. In general case, EP3 encoder does not automatically send or generate its 4-component output for the **out**> stream when it first encounters some node Γ_t of Γ (where Γ_t is of \mathbf{G} , \mathbf{X} or \mathbf{Y} type), but *postpones the output* until Γ_t is *visited for the last time*. If the node Γ_t is a leaf, the first and the last visit is the same visit, hence the output is produced immediately. Otherwise, the *last visit* to Γ_t will be its *third visit* (by virtue of our Γ rules (35d) and (35e) which imply that all internal Γ_t nodes have exactly 2 child nodes).

Note also that since Γ_t nodes of \mathbf{X} or \mathbf{Y} type give rise to the *double request for output* of $e1.X_t$ or $e1.Y_t$ components (e.g. Γ nodes Y_7 and G_7 in Fig. 7, will both request the output of $e1.Y_7$ component), EP3 encoder outputs $e1.X_t$ or $e1.Y_t$ only once: the first time that output is requested. It then marks the second request (from the G_t node, which in the postorder sequencing always comes after the requests from its child nodes) to produce and output $e1.X_t$ or $e1.Y_t$ component as redundant/inactive. The remaining components of this G_t are not affected by the specific component ‘inactive flag’ and are output when requested.

EP3 DECODING

- 1) The decoder constructs a partial node \mathbf{G}_0 from the known n , p and q (via eqs. (32) and (29)), with only the parameters of \mathbf{S}_0 and \mathbf{X}_0 completed (note that \mathbf{Y}_0 and \mathbf{Z}_0 parameters lack lengths n_y and n_z since these require symbol counts in $\mathbf{X}_0[n_x]$; parameters p_y and q_y are computed using eqs. (30b), (30c)).
- 2) The **<end>** test parameters p_{\min} , p_{\max} are computed (using \mathbf{X}_0 parameter n_x) and the **<end>** test (c1)-(c3) is invoked for \mathbf{X}_0 (denoted here as boolean function **end**(\mathbf{X}_0)). Due to the lack of n_y we cannot perform \mathbf{Y}_0 test, or even construct p_{\min} , p_{\max} for that test. Hence we do not know whether there is a right subtree of \mathbf{G}_0 .
- 3) If **end**(\mathbf{X}_0) = **false** (“no”), we proceed creating new (partial) nodes \mathbf{G}_t ($t=1,2,\dots$) (with only \mathbf{S}_t and \mathbf{X}_t parameters and p_{\min} , p_{\max} fully computed) and applying **end**(\mathbf{X}_t) until we obtain **end**(\mathbf{X}_t) = **true**. For each t we also continue computing $p_y(t)$ and $q_y(t)$ via eqs. (30b), (30c) and $n_s(t)$, $p_s(t)$ and $q_s(t)$ via eqs. (33).

In the Fig. 7 example, this step will traverse nodes \mathbf{G}_0 , \mathbf{G}_1 and \mathbf{G}_2 .

- 4) When **end**(\mathbf{X}_t) eventually becomes **true** (for some $t=0,1,\dots$) we have reached the leftmost Γ leaf (e.g. \mathbf{G}_3 in Fig. 7) of the current subtree (rooted at \mathbf{G}_0). The component e1. \mathbf{X}_t is decoded via EC:2 decoder, yielding array $\mathbf{X}_t[n_x(t)]$. Count of zeros in $\mathbf{X}_t[n_x(t)]$ provides parameter $n_y(t)$ and count of ones parameter $n_z(t)$. With $n_y(t)$ and the already computed $p_y(t)$ and $q_y(t)$ we can compute **<end>** test parameters p_{\min} , p_{\max} for the \mathbf{Y}_t and invoke the test **end**(\mathbf{Y}_t).

In Fig. 7, **end**(\mathbf{X}_t) becomes true for $t=3$ and the test **end**(\mathbf{Y}_3) = **true**.

- 5) If **end**(\mathbf{Y}_t) in step (4) returns **true**, the encoded components e1. \mathbf{Y}_t , e2. \mathbf{Z}_t and e3. rb_t are available as the next encoded items and they are decodable (since we know: $p_y(t)$, $q_y(t)$, $n_y(t)$ for $\mathbf{Y}_t[n_y]$, $n_z(t)$ for $\mathbf{Z}_t[n_z]$ extraction and $p_s(t)$, $q_s(t)$, $n_s(t)$ for rb_t decoding if $n_s(t)$ is odd). From the decoded $\mathbf{X}_t[n_x]$, $\mathbf{Y}_t[n_y]$, $\mathbf{Z}_t[n_z]$ and rb_t , we reconstruct array $\mathbf{S}_t[n_s(t)]$, which is via eq. (33d) a copy of $\mathbf{X}_{t''}[n_x(t'')]$ of the \mathbf{G}_t 's parent node $\mathbf{G}_{t''}$. Hence we have achieved the state of step (4), but for the $\mathbf{G}_{t''}$ with $t'' < t$, and thus we can proceed with step (4) and (5), recovering further arrays $\mathbf{X}_t[n_x]$, each closer to the root of this subtree \mathbf{G}_0 than the previous one.

In Fig. 7, $t''=2$, hence we have $\mathbf{X}_2[n_x]$ as a copy of $\mathbf{S}_3[n_s]$.

- 6) If **end**(\mathbf{Y}_t) in step (4) returns **false**, the node \mathbf{G}_t has a right child $\mathbf{G}_{t'}$ (generally a right subtree). Using eqs. (34), we construct partial $\mathbf{G}_{t'}$, with only $\mathbf{S}_{t'}$ and $\mathbf{X}_{t'}$ parameters completed, while missing the array $\mathbf{X}_{t'}[n_x]$, hence missing n_y and n_z , as in step (1), hence we can proceed with steps (1), (2),... (replacing the initial \mathbf{G}_0 with $\mathbf{G}_{t'}$) to process the \mathbf{G}_t 's right subtree rooted at $\mathbf{G}_{t'}$.

In Fig. 7, $\mathbf{G}_t = \mathbf{G}_2$ and $\mathbf{G}_{t'} = \mathbf{G}_4$. Since \mathbf{G}_4 is a leaf, it is decoded immediately, which then provides decoded $\mathbf{Y}_2[n_y]$ (via eqs. (34) in reverse direction). With $\mathbf{X}_2[n_x]$ and $\mathbf{Y}_2[n_y]$ available, we can compute $\mathbf{S}_2[n_s]$, which via eqs. (33) is the same array as $\mathbf{X}_1[n_x]$, ... i.e. we are decoding nodes in the order shown in Fig. 8.

Since the steps (1)-(6) above walk along each path and probe each decision branch of the EP3 flow diagram in Fig. 4, demonstrating for each of the paths & decisions a reduction in the number of remaining undecoded components, and since the number of such components is finite, this procedure terminates when no undecoded components are left.

The steps (1)-(6) are focused with the logistics of the decoding recursions along the different flow paths in EP3 flow diagram of Fig 4. Just the EP3 *output decodability* is simpler to demonstrate. We will use *mathematical induction* and note first that the EP1 and EP2 Γ trees of depths $\lambda=0$ and 1 (see Fig 6) is decodable. Now, we assume that all Γ trees up to some depth λ are decodable and we take two trees Γ_1 and Γ_2 so that one has depth λ and the other has depth not exceeding λ . We join Γ_1 and Γ_2 as the left and right subtrees of some new root node \mathbf{G}_0 , forming thus a new tree Γ_3 of depth $\lambda+1$. Since Γ_1 and Γ_2 are decodable and since the EP3 postorder output layout will result in the output components ordered into three contiguous sections as Γ_1 , Γ_2 , \mathbf{G}_0 , the EP3 decoder can decode Γ_1 and Γ_2 components (since each has depth $\leq \lambda$ and each is contiguous). Hence, decoder will produce decoded content of root nodes \mathbf{G}_{01} and \mathbf{G}_{02} of subtrees Γ_1 and Γ_2 , including their \mathbf{S} components: $\mathbf{S}_{01}[n_s]$ and $\mathbf{S}_{02}[n_s]$. Since $\mathbf{S}_{01}[n_s] = \mathbf{X}_0[n_x]$ and $\mathbf{S}_{02}[n_s] = \mathbf{Y}_0[n_y]$, and since $\mathbf{Z}_0[n_z]$ and rb_0 are the next remaining undecoded components, we can decode them and construct $\mathbf{S}_0[n_s]$ which is the input $\mathbf{S}[n]$ for the tree Γ_3 . Since all trees of depth $\lambda+1$ can be formed this way (as Γ_3), we can decode all trees of depth $\lambda+1$. Since we can decode trees of depth $\lambda \leq 1$, we can decode all trees of depth 2, 3, 4, ...

EP4: EP with block lengths $d > 2$ symbols

We generalize here our steps (a)-(e) for the 2-symbols/block EP to a d -symbols/block EP:

a) Virtual dictionary \mathbf{D} (cf. eqs. (2)-(4)) contains all d -symbol sequences:

$$\mathbf{D} = \{ 00..0, 00..1, 00..10, \dots, 11..1 \} \quad (40)$$

$$|\mathbf{D}| = \# \text{ of entries in } \mathbf{D} = 2^d \quad (41)$$

b) \mathbf{D} is partitioned into $\alpha = d + 1$ enumerative classes \mathbf{D}_c , where $c = 0, 1, \dots, d$:

$$\mathbf{D} = \mathbf{D}_0 + \mathbf{D}_1 + \dots + \mathbf{D}_d \quad (42)$$

where \mathbf{D}_c is defined as a subset of \mathbf{D} containing sequences with c ones:

$$\mathbf{D}_0 = \{ 00..00 \} \quad (43a)$$

$$\mathbf{D}_1 = \{ 00..01, 00..10, \dots, 10..00 \} \quad (43b)$$

$$\dots \dots \dots \dots \dots \dots \dots \quad \dots \dots$$

$$\mathbf{D}_d = \{ 11..11 \} \quad (43c)$$

The number of elements $|\mathbf{D}_c|$ in the class \mathbf{D}_c is binomial coefficient $C(d, c)$:

$$|\mathbf{D}_c| = \binom{d}{c} \equiv \frac{d!}{c!(d-c)!} \equiv C(d, c) \quad (44)$$

EP4 ENCODING

c) We *segment the input* $\mathbf{S}[n]$ into $m = \lfloor n/d \rfloor$ d -symbol blocks* as: $\mathbf{B} = \mathbf{B}_1 \mathbf{B}_2 \dots \mathbf{B}_m$ and generate the corresponding sequence of *class tags*: $\mathbf{T}[m] \equiv \mathbf{T} = T_1 T_2 \dots T_m$, where each T_j is a symbol $0, 1, \dots, d$ identifying the enumerative class \mathbf{E}_c ($c = 0..d$) to which the block \mathbf{B}_j belongs. Hence \mathbf{T} is a sequence in alphabet of size $\alpha = d + 1$ defined via:

$$\mathbf{B}_j \in \mathbf{E}_c \Rightarrow T_j = c \quad \text{where: } c = 0..d \text{ and } j = 1, 2, \dots, m \quad (45)$$

The probabilities p_c of occurrence of tags T_j in \mathbf{T} with values $c = 0..d$ are:

$$p_c \equiv P(T_j = c) = p^c q^{d-c} \binom{d}{c} \quad (46a)$$

$$q_c \equiv 1 - p_c \quad (46b)$$

We denote counts in \mathbf{T} of tags T_j with value c as m_c and define: $\underline{m}_c \equiv m - m_c$ (47)

The counts m_c satisfy constraints:

* For sequence lengths n for which $rb \equiv (n \bmod d) > 0$ we need to send the last rb bits of $\mathbf{S}[n]$ separately from the blocks \mathbf{B}_j e.g. in a manner similar to the residual bit for EP1.

$$m_0 + m_1 + \dots + m_d = m \equiv \lfloor n/d \rfloor \quad (48a)$$

$$0 \leq m_c \leq m \quad \text{for } c = 0..d \quad (48b)$$

The probability that a given m_c has value μ is binomial distribution:

$$P(m_c = \mu) = p_c^\mu q_c^{m-\mu} \binom{m}{\mu} \quad (49)$$

d) For each block \mathbf{B}_i from \mathbf{B} , such that if $\mathbf{B}_i \in \mathbf{E}_c$ then $|\mathbf{E}_c| > 1^*$, we compute enumerative index[†] Z_i within its class \mathbf{E}_c . Hence the index Z_i is an integer in the interval $[0, |\mathbf{E}_c|)$ and all values of Z_i in this interval are equiprobable. We denote resulting sequence of Z_i as:

$$\mathbf{Z}[n_z] = Z_1 Z_2 \dots Z_{n_z} \quad \text{where } n_z = m - m_0 - m_d \quad (50)$$

e) Encoder output for the single cycle version EP4.1 of the algorithm consists of the following 3 components:

e1) Optimally coded sequence of class tags $\mathbf{T} = T_1 T_2 \dots T_m$

$\mathbf{T}[m] \equiv \mathbf{T}$ is a sequence of m symbols in alphabet of size $\alpha = d + 1$, with known symbol probabilities (eq. (46a)) and known symbol count $m = \lfloor n/d \rfloor$. The output size $L(\mathbf{T})$ of the optimally encoded sequence \mathbf{T} , with known length m and symbol probabilities is the Shannon entropy for \mathbf{T} :

$$L(\mathbf{T}) = -m \sum_{c=0}^d p_c \log(p_c) = -m \sum_{c=0}^d p_c q^{d-c} \binom{d}{c} \log \left(p_c q^{d-c} \binom{d}{c} \right) \quad (51)$$

e2) Enumerative indices sequence $\mathbf{Z}[n_z]$

Since \mathbf{Z} is a maximum entropy sequence, it is incompressible provided the source alphabet size is an integer power of the output channel alphabet size. In our case the output channel uses binary alphabet $\alpha=2$. The source alphabet size $R_i = R_i(c)$ for index Z_i whose block $\mathbf{B}_i \in \mathbf{E}_c$ is:

$$R_i(c) = \binom{d}{c} \quad (51)$$

Hence, for $R_i(c)$ which are integer powers of 2 we can output Z_i without further encoding (as was the case for e2.Z component). Otherwise we can use **mixed radix codes** (QI provides practical form of mixed radix coding for inputs of any length [2] pp. 46-51; see Appendix B for more details). Since QI coding in fixed radix can use **precomputed tables**

* Hence we omit $\mathbf{E}_c = \mathbf{E}_0$ and $\mathbf{E}_c = \mathbf{E}_d$, which are the only cases for which $|\mathbf{E}_c| = 1$.

† For "smaller" values of d regular enumerative coding can be used. For "larger" d (such as $d > 64$) we can use Quantized Indexing.

of quantized radix powers, while the mixed radix coding for a random sequence of radices R_i requires computation of quantized radix products specific for each coding task, it is more practical to encode distinct radices $R_i(c)$ for $c = 1, 2, \dots, \lfloor d/2 \rfloor^*$ into **separate fixed radix output streams** (the size of each fixed radix stream is determined from the quantized radix powers tables, since the tag sequence \mathbf{T} provides digit count for each fixed radix). Further reduction in sizes and number of distinct fixed radices needed is obtained by factoring powers of 2 from the R_i and by grouping remaining factors into a smaller number of smaller radices (which helps the QI table based fixed radix computations). The fixed radix separation also allows for much more efficient tables for larger n_z (cf. Appendix B, the HRC methods).

The output length for index Z_i belonging to radix $R_i(c)$ is $\log(R_i(c))$ and its probability is p_c . Hence the average length of output \mathbf{Z} is:

$$L(\mathbf{Z}) = m \sum_{c=0}^d p_c \log \left(\binom{d}{c} \right) \quad (52)$$

e3) Residual bits

This component has $rb \equiv n \pmod{d}$ bits and the cost of its transmission is:

$$L(rb) = rb \cdot h(p) = H(rb) \quad (53)$$

The total length $L(\mathbf{S})$ of the EP4 output for input $\mathbf{S}[n]$ is (via eqs. (51)-(53)) optimal:

$$L(\mathbf{S}) = L(\mathbf{T}) + L(\mathbf{Z}) + L(rb) = H(\mathbf{B}) + H(rb) = H(\mathbf{S}) \quad (54)$$

EP4 DECODING

The EP4 decoding works similarly to EP1, with the only differences that the entropy coder in step (e1) codes in different alphabet $\alpha = d + 1$ (instead of ternary coder of EP1) and that the sequence \mathbf{Z} is decoded via mixed radix decoder into values $0 \leq Z_i < R_i(c)$, hence Z_i identifies d -bit patterns from the \mathbf{E}_c classes (instead of 2-bit patterns of EP1).

* Radices $R_i(c)=C(d,c)$ for $c=1..d-1$ have at most $\lfloor d/2 \rfloor$ distinct values due to symmetry $C(d,c)=C(d,d-c)$.

EP4 PERFORMANCE

Unlike EP1 where the $\mathbf{Z}[n_z]$ array was simply stored into output e2. \mathbf{Z} , the EP4 requires fixed radix R_i (or the small factors of R_i) coding for at least some of the symbols Z_i , which is more expensive (due to possible multiplications & divisions). At the same time, EP4 operates on larger blocks of bits at a time, which goes in its favor. Therefore a direct comparison of the pump efficiencies is not related to the EP performance gains as directly as in the comparisons between EP1, EP2 and EP3 methods. The EP4 single cycle efficiency is the ratio between the encoded size of \mathbf{Z} and the entropy $H(\mathbf{S})$ (cf. eq. (15)):

$$f(p, d) = m \sum_{c=0}^d p^c q^{d-c} \log \left(\binom{d}{c} \right) / nh(p) = \sum_{c=0}^d p^c q^{d-c} \log \left(\binom{d}{c} \right) / dh(p) \quad (55)$$

The most practical block sizes d are powers of 2 since for $d = 2^k$ the binomial coefficients $C(d, c)$ have only few small ($< d$) prime factors, after dividing out factors 2 (see Table 6 below), allowing for efficient precomputed tables based fixed radix coding of \mathbf{Z} (since only the fixed radices equal to the distinct prime factors of $C(d, c)$ are needed).

d	Prime Factors ($\neq 2$) of $C(d, c)$									
4	3									
8	5	7								
16	3	5	7	11	13					
32	3	5	7	13	17	19	23	29	31	

Table 6

Table 7 shows the single cycle EP4 pump efficiencies (in percents) $f(p, d)$ for several block sizes d and probabilities p (in percents).

MULTICYCLE EP4 VARIANTS

The multicycle EP extensions, such EP2 and EP3, work exactly the same way for the general block size d since the EP cycles affect only the higher level coding logistics and output layouts from multiple elemental single cycle EPs, while the block length d affects the low level coding tasks within EP1. The only difference is that the binarization steps (e1.b) of EP2 and EP3 will in general case produce $d > 2$ binary components instead of only two, \mathbf{X} and \mathbf{Y} , for $d = 2$ case, thus requiring d -ary Γ tree instead of binary Γ tree. Since, as Table 7 shows, the EP4 efficiencies for practically interesting block sizes ($d = 8, 16, 32, 64, 128$ bits) are already quite high (e.g. at $p = 25\%$, $f(p)$ ranges from 64% to 95%), the multicycle extensions of EP4 are not as useful as they are for EP1.

Single Cycle EP4 Efficiencies (in %) for Block Lengths d

p % \ d:	2	3	4	8	16	24	32	64	128
1	12.25	19.42	24.49	36.62	48.51	55.28	59.96	70.58	79.85
2	13.86	21.96	27.68	41.26	54.32	61.57	66.45	77.02	85.37
3	14.97	23.73	29.89	44.40	58.12	65.55	70.45	80.62	88.10
4	15.85	25.12	31.62	46.84	60.97	68.45	73.28	82.99	89.77
5	16.59	26.29	33.07	48.84	63.24	70.70	75.44	84.69	90.91
6	17.22	27.30	34.33	50.55	65.12	72.52	77.14	85.97	91.75
7	17.79	28.20	35.44	52.04	66.72	74.03	78.53	86.97	92.39
8	18.30	29.01	36.44	53.35	68.09	75.30	79.69	87.78	92.90
9	18.76	29.74	37.34	54.53	69.29	76.40	80.67	88.46	93.32
10	19.19	30.42	38.17	55.60	70.35	77.34	81.51	89.02	93.66
11	19.58	31.04	38.93	56.56	71.29	78.18	82.24	89.50	93.96
12	19.95	31.62	39.64	57.45	72.13	78.91	82.88	89.92	94.21
13	20.29	32.16	40.30	58.26	72.89	79.56	83.45	90.29	94.44
14	20.61	32.66	40.91	59.01	73.58	80.15	83.95	90.61	94.63
15	20.91	33.14	41.49	59.71	74.20	80.67	84.40	90.90	94.80
16	21.19	33.58	42.03	60.35	74.77	81.15	84.81	91.15	94.96
17	21.45	34.00	42.54	60.95	75.29	81.58	85.17	91.38	95.09
18	21.70	34.40	43.01	61.51	75.77	81.97	85.50	91.59	95.22
19	21.94	34.77	43.47	62.03	76.21	82.33	85.81	91.78	95.33
20	22.16	35.13	43.89	62.52	76.62	82.66	86.08	91.95	95.43
21	22.37	35.46	44.29	62.98	76.99	82.97	86.33	92.10	95.52
22	22.57	35.78	44.67	63.40	77.34	83.24	86.57	92.25	95.61
23	22.76	36.08	45.03	63.80	77.66	83.50	86.78	92.38	95.69
24	22.94	36.36	45.37	64.18	77.95	83.74	86.98	92.50	95.76
25	23.11	36.63	45.69	64.53	78.23	83.95	87.16	92.61	95.82
26	23.27	36.89	46.00	64.85	78.48	84.16	87.33	92.71	95.88
27	23.42	37.13	46.28	65.16	78.72	84.34	87.48	92.81	95.94
28	23.57	37.35	46.55	65.45	78.94	84.52	87.62	92.89	95.99
29	23.70	37.57	46.80	65.72	79.15	84.68	87.76	92.98	96.04
30	23.83	37.77	47.04	65.97	79.34	84.83	87.88	93.05	96.08
31	23.95	37.96	47.27	66.21	79.51	84.96	87.99	93.12	96.12
32	24.06	38.14	47.48	66.43	79.68	85.09	88.10	93.18	96.16
33	24.17	38.30	47.68	66.63	79.83	85.21	88.19	93.24	96.19
34	24.26	38.46	47.86	66.82	79.97	85.32	88.28	93.30	96.22
35	24.36	38.60	48.03	66.99	80.09	85.42	88.37	93.35	96.25
36	24.44	38.74	48.19	67.16	80.21	85.51	88.44	93.39	96.28
37	24.52	38.86	48.34	67.31	80.32	85.59	88.51	93.43	96.30
38	24.59	38.98	48.47	67.44	80.42	85.67	88.57	93.47	96.33
39	24.66	39.08	48.60	67.57	80.51	85.74	88.63	93.50	96.35
40	24.72	39.18	48.71	67.68	80.59	85.80	88.68	93.54	96.37
41	24.77	39.26	48.81	67.78	80.66	85.86	88.73	93.56	96.38
42	24.82	39.34	48.90	67.87	80.72	85.91	88.77	93.59	96.40
43	24.86	39.41	48.98	67.95	80.78	85.95	88.80	93.61	96.41
44	24.90	39.46	49.05	68.01	80.83	85.99	88.83	93.63	96.42
45	24.93	39.51	49.10	68.07	80.87	86.02	88.86	93.64	96.43
46	24.96	39.55	49.15	68.12	80.90	86.04	88.88	93.65	96.43
47	24.97	39.58	49.19	68.15	80.93	86.06	88.90	93.66	96.44
48	24.99	39.61	49.21	68.18	80.94	86.07	88.91	93.67	96.44
49	25.00	39.62	49.23	68.19	80.96	86.08	88.91	93.68	96.45
50	25.00	39.62	49.23	68.20	80.96	86.09	88.92	93.68	96.45

Table 7.

QI/EC MODELING OF MARKOV SOURCES

Markov source of order r ($\mathbf{M}|r$) is a probabilistic source for which the probabilities of the next symbol are determined by the value of previous r symbols^{*}. Bernoulli source is an order 0 or ($\mathbf{M}|0$) source. Our input sequence generated by $\mathbf{M}|r$ is $\mathbf{S} \equiv \mathbf{S}[n] = a_1 a_2 \dots a_n$, where a_i belong to alphabet $\mathbf{A}=[0, \alpha]$ of size α . The $\mathbf{M}|r$ property of $\mathbf{S}[n]$ for the i -th symbol can be expressed in **conditional probabilities** notation $P(\mathbf{Event}|\mathbf{Condition})$:

$$P_{\mathbf{M}|r}(a_i) \equiv P(r, a_i) = P(a_i | a_{i-1} a_{i-2} \dots a_{i-r}) \quad (60)$$

which says that the probability for i -th symbol to have some value $a_i \in \mathbf{A}$ is a function of $r+1$ variables: a_i and the r symbols $a_{i-1}, a_{i-2} \dots a_{i-r}$ preceding[†] a_i in $\mathbf{S}[n]$. We call this r symbol sequence preceding a_i the **r -context** $C(r, a_i) \equiv C(a_i) \equiv C \equiv c_1 c_2 \dots c_r$ of a_i :

$$\begin{aligned} & | \leftarrow C(r, a_i) \rightarrow | \\ & \dots a_{i-r} a_{i-r+1} \dots a_{i-2} a_{i-1} \mathbf{a}_i \dots & (61a) \\ & c_r \ c_{r-1} \ \dots \ c_2 \ c_1 & (61b) \end{aligned}$$

While the definition (60) of $\mathbf{M}|r$ is unambiguous within the ‘*infinite sequence*’ source parametrization of the conventional probabilistic modeling, it is ambiguous within the ‘*finite sequence*’ parametrization of the QI/EC modeling. Namely, for a finite sequence $\mathbf{S}[n]$ we also need to specify **boundary/initial conditions** at the ends of the array $\mathbf{S}[n]$. In this case we need to specify the results of (60) for $i < r$, where some sequence subscripts, such as $i-r$, become invalid. We will use **periodic boundary conditions** convention[‡], in which the copies $\mathbf{S}^-[n], \mathbf{S}^+[n]$ of $\mathbf{S}[n]$ are assumed to be repeated before and after $\mathbf{S}[n]$:

$$\begin{aligned} \dots | \leftarrow \mathbf{S}^-[n] \rightarrow | \leftarrow \mathbf{S}[n] \rightarrow | \leftarrow \mathbf{S}^+[n] \rightarrow | \dots \\ \dots a_1 a_2 \dots a_{n-3} a_{n-2} a_{n-1} a_n \mathbf{a}_1 \mathbf{a}_2 \mathbf{a}_3 \dots \mathbf{a}_{n-1} \mathbf{a}_n a_1 a_2 \dots a_{n-3} a_{n-2} a_{n-1} a_n \dots \end{aligned} \quad (62)$$

Therefore, in this convention the $\mathbf{M}|r$ probabilities for a_1 are defined by the r -context consisting of the last r characters of $\mathbf{S}[n]$:

$$C(r, a_1) = a_{n-r+1} a_{n-r+2} \dots a_{n-1} a_n \quad (63)$$

^{*} The type of source considered in this section is also called Finite Context (FC) source. More general Markov source is a Finite State (FS) source which includes FC sources as a special case (e.g. FS can model multi-alphabet sequence from its binary representation while FC cannot). Since FC provides a more concrete model description, we use it to present our Markov modeling, even though the methods presented work the same way (with a more elaborate notation) for general FS sources. Further, we also restrict this section to stationary Markov sources (the general probabilistic and BWT modelers remove this restriction).

[†] In many practical instances of $\mathbf{M}|r$ the “ r preceding symbols a_i ” are interpreted as “some r symbols” with indices lower than i , but which are not necessarily the symbols immediately preceding a_i in $\mathbf{S}[n]$. For example, in image compression the “ r preceding symbols” are only spatially, in 2-D, adjacent to the pixel a_i but they are not adjacent to a_i in the 1-D image representation via $\mathbf{S}[n]$.

[‡] This convention is equivalent to viewing $\mathbf{S}[n]$ as a **cyclic array**, with a_{n+1} ‘wrapping back’ as a_1 .

Combinatorial Properties of Contexts

CP1. Array $\mathbf{C} \equiv \mathbf{C}[m] \equiv \mathbf{C}^r[m] \equiv \{C_u: u=1..m\}$ is a sequence of all *unique r-contexts* in $\mathbf{S}[n]$. Since the number of *all r-contexts* $\mathbf{C}(r, a_i)$ (for $i=1..n$) in $\mathbf{S}[n]$ is n , the number of unique contexts m is constrained as:

$$1 \leq m \leq n \quad (64)$$

CP2. An implication of eq. (60) relevant for the EC/QI modeling of M|r is that for any *fixed context* $C = c_1 c_2 \dots c_r$ the *probabilities* $P(a | C)$ for all $a \in \mathbf{A}$ *are fixed* through all occurrences of context C in $\mathbf{S}[n]$. Therefore, the sequence $\mathbf{B}_c \equiv \mathbf{B}_c[n_c] \equiv b_1 b_2 \dots b_{n_c}$ of symbols which follow the 1st, 2nd, ... n_c th occurrence of C in $\mathbf{S}[n]$ is an M|0 sequence (n_c is the total number of occurrences of C in $\mathbf{S}[n]$). Hence, \mathbf{B}_c is an element of *enumerative class* \mathbf{E}_c consisting of all strings $\mathbf{W}_c \equiv \mathbf{W}[n_c]$ which have the same *array of symbol counts** $\mathbf{K}(\mathbf{W}_c) \equiv \mathbf{K}_c[\alpha] \equiv \mathbf{K}_c \equiv (k_0, k_1, \dots, k_{\alpha-1})$ as the sequence \mathbf{B}_c :

$$\mathbf{E}_c = \{ \text{all } \mathbf{W}_c: \mathbf{K}(\mathbf{W}_c) = \mathbf{K}(\mathbf{B}_c) \equiv (k_0, k_1, \dots, k_{\alpha-1}) \} \quad (65)$$

where k_a is the *count of symbol a* in \mathbf{B}_c (i.e. $k_a \equiv \mathbf{K}_c[a]$). Since $k_0 + k_1 + \dots + k_{\alpha-1} = n_c$, the size $|\mathbf{E}_c|$ of the class \mathbf{E}_c , which is multinomial (cf. [1] p. 11), can be written as:

$$|\mathbf{E}_c| = \binom{n_c}{k_0, k_1, \dots, k_{\alpha-1}} \equiv \frac{n_c!}{k_0! k_1! \dots k_{\alpha-1}!} \equiv M(n_c; k_0, k_1, \dots, k_{\alpha-1}) \quad (66)$$

CP3. Since a sequence $\Sigma_c \mathbf{B}_c$ concatenating all sequences \mathbf{B}_c for $C \in \mathbf{C}$ contains all n symbols, exactly once each, of the original input array $\mathbf{S}[n]$, the counts n_c satisfy relation:

$$\sum_{c \in \mathbf{C}} n_c = n \quad (67)$$

CP4. Sequences \mathbf{B}_c along with the corresponding context strings $C \in \mathbf{C}$ can be used to *reconstruct* not just n but *the original sequence* $\mathbf{S}[n]$ as well. To show this reconstruction, we define a working dictionary Δ whose entries (e.g. OOP objects) $S(C)$ contain context $C = c_1 c_2 \dots c_r$ and the corresponding $S(C) \cdot \mathbf{B}_c \equiv \mathbf{B}_c$ (which contains sequence $b_1 b_2 \dots b_{n_c}$ and its length n_c) for all $C \in \mathbf{C}[m]$. Given a context C , the dictionary Δ can retrieve (a pointer to) entry $S(C)$. For convenience, each entry $S(C)$ also contains a working variable $j \equiv S(C).j$ which represents the current position within \mathbf{B}_c (initialized as $j=0$). The following steps show reconstruction of $\mathbf{S}[n]$ from Δ and the starting context $\mathbf{C}(r, a_1)$ (cf. eq.(63)):

1. Set the *current context* $C = C(a_1)$ and output index $i = 0$
2. Retrieve from Δ $S(C)$ (containing \mathbf{B}_c and count $j \equiv S(C).j$).

* In the '*Method of Types*' the array of symbol counts $\mathbf{K}(\mathbf{S})$ of some sequence \mathbf{S} is called "type of \mathbf{S} ". Hence, \mathbf{E}_c consists of all strings \mathbf{W}_c of the *same 'type'* as \mathbf{B}_c . Note that *enumerative classes* \mathbf{E}_c are combinatorial category *distinct from 'types'* e.g. an \mathbf{E}_c can include sequences with differing symbol counts and lengths (as long as they are equiprobable i.e. have the same entropy $p \log(1/p)$). The two categories coincide only for order-0 stationary sequences.

3. Get symbol $a = \mathbf{B}_c[j]$ and increment count $S(C).j$.
4. Output symbol a as: $\mathbf{S}[i] = a$, increment i and terminate if $i = n$.
5. Compute the ‘next context’ $X = x_1 x_2 \dots x_r$ from the current context $C = c_1 c_2 \dots c_r$ using the mapping (a right shift of C): $x_1 x_2 \dots x_r \leftarrow a c_1 c_2 \dots c_{r-1}$.
6. Assign $C = X$ and go to step 2.

CP5. While counts $k_a \in \mathbf{K}_c$ are simple counts of symbol a in \mathbf{B}_c , in the input sequence $\mathbf{S}[n]$ the counts k_a count occurrences of the $s \equiv (r+1)$ symbol substrings D of $\mathbf{S}[n]$ as follows:

$$D \equiv D[s] \equiv D_{a.c} \equiv d_1 d_2 d_3 \dots d_s \equiv a c_1 c_2 \dots c_r \quad (68)$$

where: $s \equiv r+1$ and $d_1 \equiv a, d_{1+t} \equiv c_t$ for $t=1..r$

$$\#d_1 d_2 \dots d_s \equiv \#D_{a.c} \equiv \mathbf{K}_c[a] = k_a \text{ for } a \in A \quad (69)$$

We call s -symbol substrings D of $\mathbf{S}[n]$ the ***s-grams***^{*} of $\mathbf{S}[n]$ and denote their ***occurrence counts*** in $\mathbf{S}[n]$ by prefixing the string or its label with ‘#’ (e.g. #abc is count of occurrence of ‘abc’ in \mathbf{S}). Hence the 1-gram # a is simply the symbol count of a in $\mathbf{S}[n]$. For an ***empty string*** ϕ , # ϕ defines a count of 0-grams, which we set by convention: # $\phi \equiv n$. We also denote the set of all ***unique s-grams*** occurring in $\mathbf{S}[n]$ as $\mathbf{G} \equiv \mathbf{G}^s \equiv \mathbf{G}^s(\mathbf{S})$ and the array of their corresponding counts as # \mathbf{G} , with count # D of s -gram D retrieved[†] as # $D = \#\mathbf{G}[D]$.

CP6. The objects \mathbf{G} and # \mathbf{G} can be constructed from the array (of unique contexts) $\mathbf{C}[m]$ and a set $\{\mathbf{K}\} \equiv \{\text{arrays } \mathbf{K}_c \text{ for all } C \in \mathbf{C}[m]\}$. Namely, for each $C = c_1 c_2 \dots c_r \in \mathbf{C}[m]$ we scan the corresponding array $\mathbf{K}_c[\alpha] \in \{\mathbf{K}\}$ for $a = 0..r-1$, examining counts $k_a = \mathbf{K}_c[a]$. For each nonzero count k_a found, we append s -gram $D \equiv a c_1 c_2 \dots c_r$ to \mathbf{G} and the count k_a to # \mathbf{G} .

CP7. Similarly, the objects $\mathbf{C}[m]$ and $\{\mathbf{K}\}$ can be constructed from \mathbf{G} and # \mathbf{G} as follows: we split \mathbf{G} into classes \mathbf{T} , each class consisting of all s -grams $D \in \mathbf{G}$ with the same r -symbol ***postfix*** $T(D) = d_2 d_3 \dots d_s$. Hence, the s -grams $D \in \mathbf{T}$ differ only in their first symbol d_1 . We set context count $m = 0$ and loop through all classes $\mathbf{T} \subset \mathbf{G}$ and for each \mathbf{T} :

1. We assign \mathbf{T} ’s common r -symbol postfix $T(D) = d_2 d_3 \dots d_s \equiv C \equiv c_1 c_2 \dots c_r$ to $\mathbf{C}[m] = C$ and increment number m of unique r -contexts created so far.
2. We create array $\mathbf{K}_c[\alpha]$ as a new element of $\{\mathbf{K}\}$, set its values to 0. Then for each s -gram $D = d_1 d_2 \dots d_s \in \mathbf{T}$ we set $\mathbf{K}_c[d_1]$ to the s -gram D count, the number # $\mathbf{G}[D]$ from the set # \mathbf{G} .

The important consequence of the reconstructions (CP6,7) for coding is that transmitting objects $\{\mathbf{C}\}$ and $\{\mathbf{K}\}$ costs exactly the same as transmitting objects \mathbf{G} and # \mathbf{G} . Since the latter two

^{*} We interpret “substring of $\mathbf{S}[n]$ ” under the ***cyclic boundary conditions*** (63) for $\mathbf{S}[n]$ i.e. s -grams D may wrap around the ends of $\mathbf{S}[n]$. The s -grams (of cyclic and non-cyclic type) are used extensively in data mining of large data sets (linguistics, biology, search engines,...) where they are called n -grams or q -grams.

[†] For brevity, we use array notation for # $\mathbf{G}[D]$ searches. Implementation would use hash tables, trees,... etc.

objects are simpler and algorithmically more convenient, we examine below some of their properties.

CP8. The size $|\mathbf{G}|$ of \mathbf{G} (which is the same as the size $|\#\mathbf{G}|$ of $\#\mathbf{G}$) can be upper bound by noting that \mathbf{G} is a subset of a set $\mathbf{A}^s \equiv \mathbf{A}^s(\alpha)$ of all strings with s symbols from alphabet \mathbf{A} of size α , hence:

$$|\mathbf{G}| = |\#\mathbf{G}| \leq |\mathbf{A}^s| = \alpha^s \quad (70a)$$

$$|\mathbf{G}| \leq n \quad (70b)$$

$$s=1 \Rightarrow |\mathbf{G}| \leq \text{Min}\{ n, \alpha \} \quad (70c)$$

CP9. The s -gram counts are constrained by the following hierarchy of linear equations:

$$\sum_{x=0}^{\alpha-1} \#x a_1 a_2 \cdots a_t = \#a_1 a_2 \cdots a_t = \sum_{x=0}^{\alpha-1} \#a_1 a_2 \cdots a_t x \quad (71)$$

Eqs. (71) follow by considering how the counts $A \equiv \#a_1 a_2 \dots a_t$ and $B[x] \equiv \#a_1 a_2 \dots a_t x$ (for $x = 0.. \alpha-1$) are computed – a loop `for(i=0; i<n; ++i)` scans the cyclic string $\mathbf{S}[n]$, keeping track in each step the last t symbols $L[t]$. Whenever $L[t] = a_1 a_2 \dots a_t$, we increment counter A and one of the counters $B[x]$, using the value of the symbol x which follows a_t as the index for the $B[\alpha]$ counters. Hence, the sum of the $B[\alpha]$ counters (right side sum in (71)) is the same as the A counter (middle term in (71)). For the left side sum in (71), we use the symbol x which preceded a_1 as index for the $B[x]$ counter in the reasoning above and arrive at the same conclusion.

Eqs. (71) hold for any $t = 0, 1, \dots, n-1$ and any sequence $a_1 a_2 \dots a_t$. For $t=0$, the left and right sums in (71) become $\sum_x \#x = \#\phi \equiv n$. Thus our 0-gram convention (for count of empty substring ϕ): $\#\phi \equiv n$ agrees with the sum of 1-grams $\#x$, since values $\#x$ are the counts of symbol x (where $x = 0, 1, \dots, \alpha-1$) in $\mathbf{S}[n]$, and the sum all $\#x$ is also equals n .

Example: $s=2, \alpha=2$. Parameter t in (71) can be 0 or 1. Since $t=0$ was already discussed for general case, the remaining value $t=1$ in (71) yields the equations:

$$\#0a + \#1a = \#a = \#a0 + \#a1, \quad \text{for } a=0,1$$

which produce a single independent constraint: $\#01 = \#10$ i.e. the numbers of substrings 01 and 10 must be the same (for cyclic boundary conditions of eq. (62)). Since there are four 2-gram counts $\#00, \#01, \#10$ and $\#11$, we need to transmit only 3 independent numbers: $\#00, \#01$ and $\#11$ to specify $\#\mathbf{G}$ (which in turn implies \mathbf{G}).

MS1: Modeling Markov Source of Known Order r

The M|r case of known order r and possibly unknown probabilities $P(a \in \mathbf{A} \mid C \in \mathbf{C})$ is of practical importance since we often know the relevant range of a context influencing the next symbol, although we may not know the exact degree of that influence (e.g. in image compression, we may know in advance the size of the relevant pixel neighborhood but not its quantitative effects on the pixel value).

ENCODING

- 1) We create an empty *context dictionary* Δ with functionality to add new contexts (pointers to context state) to Δ , check whether a context C is in Δ and if found return its state $S(C)$, and finally to extract the list of all $S(C)$ in Δ . The *context state* $S(C)$ includes output array for the enumerative index $I(\mathbf{B}_c)$ within the class \mathbf{E}_c (of the array \mathbf{B}_c for that context), along with the necessary QI/EC coding state variables, the array \mathbf{K}_c and misc. state variables. We denote any variable x local to $S(C)$ as $S(C).x$.
- 2) We set the ‘*current context*’ $C = C(a_1)$ and loop (for $i=1,2, \dots n-r$) through the following steps:
 - a. If C is not in Δ , we create new context state $S(C)$ and add it to Δ , otherwise we retrieve $S(C)$ from Δ . The new $S(C)$ constructor allocates space for arrays \mathbf{K}_c and $I(\mathbf{B}_c)$, clears both arrays and integer counters (e.g. n_c) to 0. It initializes EC/QI coder variables for encoding \mathbf{B}_c into $I(\mathbf{B}_c)$.
 - b. We read the next input symbol $a_i = \mathbf{S}[i]$ and increment the counts $\mathbf{K}_c[a_i]$ and $n_c \equiv S(C).n_c$. With QI’s colex enumeration, we encode here the symbol a_i into the output index $I(\mathbf{B}_c)$ (i.e. without accumulating \mathbf{B}_c at all).
 - c. From $C = c_1 c_2 \dots c_r$ and a_i we form the ‘*next context*’ $X = x_1 x_2 \dots x_r$ via mapping:

$$x_1 x_2 \dots x_r \leftarrow a_i c_1 c_2 \dots c_{r-1}.$$
 - d. We increment i and if ($i \leq n-r$) we update the ‘current state’ as $C \leftarrow X$ and go back to step (a). Otherwise (when $i = n-r+1$) we terminate the loop.
- 3) We extract from dictionary Δ the *full list* $\mathbf{C}[m]$ of m unique contexts C (with associated context state objects $S(C)$).
- 4) For each \mathbf{B}_c , the accumulated symbol counts $\mathbf{K}(\mathbf{B}_c) \equiv \mathbf{K}_c[\alpha] \equiv (k_0, k_1, \dots, k_{\alpha-1})$ identify the enumerative class \mathbf{E}_c to which \mathbf{B}_c belongs and with respect to which the index $I_c(\mathbf{B}_c)$ is computed in step (2.b). The output size of I_c is obtained from:

$$0 \leq I_c \leq |\mathbf{E}_c| - 1 = M(n_c; k_0, k_1, \dots, k_{\alpha-1}) - 1 \quad (72)$$

The bit fraction packaging of the sequence of indices I_c is done as described in Appendix B. The output is a component QMR-TH($\mathbf{I}[m]$), where $\mathbf{I}[m]$ is the sequence of m indices and QMR-TH($\mathbf{I}[m]$) is its mixed radix code (App. B).

- 5) Encode the class tags $\mathbf{T} = T_1 T_2 \dots T_m$ which indicate the sequence of \mathbf{E}_c 's to be used by decoder to unrank the sequence of indices I_c and reconstruct arrays \mathbf{B}_c . The unencoded class tags here are the m count arrays \mathbf{K}_c . Recalling the $\mathbf{S}[n]$ reconstruction (see (CP4)) from the unique contexts array $\mathbf{C}[m]$ and the set $\{\mathbf{K}\}$ of associated counts arrays \mathbf{K}_c , we need to send $\mathbf{C}[m]$ and $\{\mathbf{K}\}$. Since, by (CP7), the combined information of $\mathbf{C}[m]$ and $\{\mathbf{K}\}$ is the same as that of set \mathbf{G} (of all unique s -grams encountered) and $\#\mathbf{G}$ (the counts of the s -grams from \mathbf{G}) and since the former pair can be computed from the latter (via CP7), we encode \mathbf{G} and $\#\mathbf{G}$.

The MS1 output thus consists of encoded components: e1. \mathbf{G} , e1. $\#\mathbf{G}$ and e2.QMR($\mathbf{I}[m]$).

DECODING

1. Decoder expands the s -grams \mathbf{G} and their counts $\#\mathbf{G}$ from e1. \mathbf{G} , e1. $\#\mathbf{G}$
2. Using algorithm (CP7), decoder computes from \mathbf{G} and $\#\mathbf{G}$ the context array $\mathbf{C}[m]$ and the set $\{\mathbf{K}\}$ of corresponding \mathbf{K}_c (symbol counts of \mathbf{B}_c) arrays.
3. From the e2.QMR($\mathbf{I}[m]$), the mixed radix decoder of Appendix B decodes the sequence $\mathbf{I}[m]$ of indices I_c (for \mathbf{B}_c sequences).
4. From each pair \mathbf{K}_c (from step #2) and I_c decoder unranks the sequence \mathbf{B}_c , for all $C \in \mathbf{C}[m]$.
5. From the set of \mathbf{B}_c 's and the corresponding contexts $C \in \mathbf{C}[m]$ decoder computes the input sequence $\mathbf{S}[n]$ using (CP4) algorithm.

* Numerous techniques exist for n -gram coding from various field, including those developed for PPM implemetations for use with AC. The choice of n -gram coding algorithm depends on the type of inputs.

USE OF GENERAL PROBABILISTIC MODELS BY QI/EC

The general probabilistic models (**GPM**) are a generalization of Markov M|r models in which the probability distribution $P(a_i)$ of the next symbol a_i of the input sequence $\mathbf{S}[n]$ depends on the entire preceding section of the sequence, its symbols *and* the position i for which the prediction is being made. Hence, eq. (60) characterizing Markov source M|r is replaced by a more general functional dependency:

$$P_{\text{GPM}}(a_i) \equiv P(a_i) = P(a_i | a_{i-1} a_{i-2} \dots a_1) \equiv P(a_i | C_{i-1}) \quad (80)$$

The conditional probability in (80) is a general function of $i-1$ variables a_j for $j=1..i-1$. But GPM also introduces an assumption (i.e. a restriction) that was not present in the M|r models: GPM assumes that coder and decoder know function $P(a_i | C_{i-1})$, which is an assumption we didn't make for M|r models (we only assumed that coder knows the model order r).

In the finite sequence setting, where the full sequence length is also given, our periodic (cyclic) boundary conditions can express eq. (80) as a special case of M|n model, i.e. Markov model with a fixed length r context, where r equal to the length n of $\mathbf{S}[n]$:

$$P_{\text{M|n}}(a_i) \equiv P(a_i) = P(a_i | a_{i-1} a_{i-2} \dots a_{i-n}) \quad (81)$$

with an additional 'predictive modeling' restriction on the n -parameter function $P(a_i | C_n)$ in (81): the GPM case of M|n in (81) ignores the parameters a_j in (81) for $j \geq i$. The general M|n modeler (81) for finite sequences takes these ignored parameters into account. Therefore, our M|r discussion and algorithms from the previous section apply for GPM as well. Since GPM has an additional restriction – the probabilities $P(a_i)$ are assumed to be known to encoder/decoder – the QI/EC coder for GPM source can make use of this restriction and improve coding efficiency and speed.

The GPM form of QI/EC modeling algorithms constructs enumerative classes \mathbf{E}_c (cf. [1], p. 27) using the function $P(a_i | C_{i-1})$, resulting in the following algorithm:

GPM1: QI/EC GPM Modeler

- a) We define an interval called **range** R_i of function $P(a_i | C_{i-1})$ as follows:

$$R_i \equiv [P_0(i), P_1(i)] \equiv [P_0, P_0 + \Delta R(i)] \quad (82)$$

$$\begin{aligned} \text{where: } P_0(i) &\equiv P_0 \equiv \text{Min} \{ P(a_i): a_i \in [0, \alpha] \} \\ P_1(i) &\equiv P_1 \equiv \text{Max} \{ P(a_i): a_i \in [0, \alpha] \} \\ \Delta R(i) &\equiv \Delta R \equiv P_1(i) - P_0(i) \end{aligned} \quad (83)$$

- b) We partition range R_i into $m = m(i|C_{i-1})$ sections (non-overlapping subintervals) denoted as:

$$D_j \equiv [d_j, d_j + \Delta_j) \quad \text{for } j=1..m \quad (84)$$

$$\text{where: } d_{j+1} = d_j + \Delta_j \quad \text{for } j=1..m-1 \text{ and } d_1 = P_0 \quad (85)$$

where d_j and Δ_j are functions of i and C_{i-1} (just as m is). The values m and Δ_j are selected to be ‘small’ with respect to $\Delta R(i)$, such as follows:

$$m = \sqrt{n} \quad (86)$$

$$\Delta_j \equiv \delta = \Delta R / \sqrt{n} \quad (87)$$

which yields equal subintervals of R_i :

$$D_j = [P_0 + (j-1) \cdot \delta, P_0 + j \cdot \delta) \quad (88)$$

- c) We maintain m output blocks $\mathbf{B}_1 \mathbf{B}_2 \dots \mathbf{B}_m$ and send ‘next symbol’ a_i to block \mathbf{B}_j , where j is defined by the subinterval D_j containing GPM prediction $P(a_i | C_{i-1})$.
- d) Since each \mathbf{B}_j will contain symbols a_i which are *approximately equiprobable*, the block \mathbf{B}_j belongs to enumerative class \mathbf{E}_j consisting of all symbol sequences \mathbf{W} that have the same symbol counts $\mathbf{K}(\mathbf{W}) \equiv (k_0, k_1, \dots, k_{\alpha-1})$ as \mathbf{B}_j (cf. eq. (65)):

$$\mathbf{E}_j = \{ \text{all } \mathbf{W}: \mathbf{K}(\mathbf{W}) = \mathbf{K}(\mathbf{B}_j) \equiv (k_0, k_1, \dots, k_{\alpha-1}) \} \quad (89)$$

The encoding/decoding logistics of \mathbf{B}_j is similar to MS1 algorithm for the M|r models (GPM is a special case of finite sequence formulation of M|n).

The GPM1 redundancy per symbol (δH) depends on the probability errors δP , which are upper bound by the sizes Δ_j of the subintervals D_j . The general form of this dependency in the limit of $\delta P \rightarrow 0$ is:

$$\delta H = \sum_{a=0}^{\alpha-1} \frac{\delta P(a)^2}{P(a)} = \sum_{a_i=0}^{\alpha-1} \frac{\Delta_j^2(a_i)}{d_j(a_i)} \approx O\left(\frac{\alpha}{n}\right) \quad (90)$$

where we have used eq. (87) to estimate the asymptotic value of δH for $n \rightarrow \infty$. This δH is of the same order as the effects of typical sampling errors on symbol statistics.

Appendix A: Postorder Tree Traversal (for EP3 Component Layout)

//-- Binary tree node structure

```
typedef struct _node { // Node data type (for binary tree)
    _struct _node *left; // pointer to left child X(0 if none)
    _struct _node *right; // pointer to right child Y (0 if none)
    void *content; // generic "content" pointer of this node
} NODE;
```

//-- Store node pointers into output buffer in postorder layout

```
int list_tree(NODE *root, NODE *out)
{ int i=0;
  visit(root,out,&i); // recurse from given root
  return i; // return # of nodes stored into out[]
}
```

//-- Postorder (bottom up) tree traversal

```
void visit(NODE *parent, NODE *out, int *ip)
{
  if (parent->left) // visit left subtree if available
    visit(parent->left, out, ip);
  if (parent->right) // visit right subtree if available
    visit(parent->right, out, ip);
  out[*ip]=parent; // output parent of these subtrees
  ip[0]++; // advance output position (count
} // of node pointers stored)
```

Fig. A.1 shows output of `list_tree()` for a 15-node tree, where the root (label 15) represents the original input sequence **S**. The nodes 1..14 represent components **X** (left children) and **Y** (right children). The leaves (1, 2, 3, 5, 8, 9, 11 and 12) represent the **X** and **Y** components which are encoded via **EC:2** (in the order of increasing leaf labels). The EP3 decoder visits and produces decoded components in the order 1, 2, ..., 15.

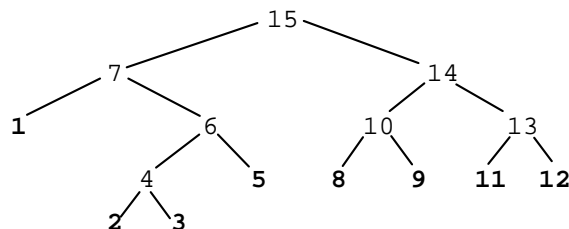


Fig. A.1

Appendix B: Bit Fractions Removal via Mixed Radix Coding

The input for *mixed radix (MR) coding* (cf. [2]) is a sequence $\mathbf{A} \equiv \mathbf{A}[n] = a_1 a_2 \dots a_n$, where “digits” a_i are integers constrained via integer “radices” R_i as:

$$0 \leq a_i < R_i \text{ for } i=1,2,\dots, n \quad (\text{B.1})$$

The MR coder also assumes that for each digit a_i all its values consistent with constraint (B.1) are equiprobable (i.e. \mathbf{A} is a *high entropy limit*/max entropy sequence). The exact *enumerative coding (EC)* of \mathbf{A} produces an integer (index) $I(\mathbf{A})$ computed as follows:

$$I(\mathbf{A}) = a_1 V_0 + a_2 V_1 + a_3 V_2 + \dots + a_n V_{n-1} \quad (\text{B.2})$$

$$\text{where: } V_0 \equiv 1, V_1 \equiv R_1, V_2 \equiv R_1 R_2, \dots, V_n \equiv R_1 R_2 \dots R_n \quad (\text{B.3})$$

For different n digit sequences \mathbf{A} satisfying (B.1), the index $I(\mathbf{A})$ takes V_n distinct values $0, 1, \dots, V_n - 1$ and all these values of I are equiprobable. The quantities V_i defined via (B.3) are called *enumerative volumes* (or *radix powers* for *fixed radix* case: $R_i = R$ for $i=1..n$, since volumes V_i become regular i -th powers of the common radix R : $V_i = R^i$ for $i=0..n$). The problem with index computation (B.2) is that it becomes quickly impractical due to the increasing arithmetic precision required as n increases.

Quantized Indexing (QI) solves the precision problem of EC by replacing exact volumes V_i with quantized volumes Q_i , which are of special numeric type called *Sliding Window Integers (SW integers)*, cf. [3] p. 7). An SW integer $Q = SW(g, w, s)$ is an integer defined via three integer parameters g (precision), w (mantissa) and s (exponent), with w limited to g bits, as follows*:

$$Q = SW(g, w, s) \equiv (w, s) = w \cdot 2^s \quad (\text{B.4})$$

$$\text{where: } 0 \leq w < 2^g \text{ and for } s > 0 \Rightarrow 2^{g-1} \leq w < 2^g \quad (\text{B.5})$$

$$Q = SW(g, w, s) = \overbrace{1xx \dots x}^{g\text{-bit } w} \overbrace{0000 \dots 0000}^{s \text{ bits}} \text{ for } (s > 0) \quad (\text{B.6})$$

While SW integers are similar in form to floating point (FP) numbers[†], the SW arithmetic is exact integer arithmetic i.e. unlike FP arithmetic, the SW arithmetic operations are *decoupled from rounding*. With SW integers, the rounding is a separate operator $\lceil X \rceil_{sw}$ applicable to general integers X :

$$Q = \lceil X \rceil_{sw} \Leftrightarrow Q = \text{Min}\{ SW(g, w, s) \geq X, \text{ for all } w, s \} \quad (\text{B.7})$$

In words, SW rounding $\lceil X \rceil_{sw}$ is *rounding up* to the nearest g -bit SW integer $Q \geq X$. The QI Mixed Radix coding (QMR) replaces (B.2)-(B.3) with (cf. [2], p. 64) :

* We also abbreviate function $SW(g, w, s)$ as (w, s) since g , being normally fixed, can be left implicit.

† Note also that SW *normalization* in eq. (B.5) applies only to $Q = SW(g, w, s)$ integers for which the exponent s is nonzero, while the FP normalization is done for all values of FP exponent.

$$I(\mathbf{A}) = a_1 Q_0 + a_2 Q_1 + a_3 Q_2 + \dots + a_n Q_{n-1} \quad (\text{B.8})$$

$$\text{where: } Q_0 \equiv 1, Q_1 \equiv \lceil R_1 \rceil_{\text{sw}}, Q_2 \equiv \lceil Q_1 R_2 \rceil_{\text{sw}}, \dots, Q_n \equiv \lceil Q_{n-1} R_n \rceil_{\text{sw}} \quad (\text{B.9a})$$

$$\text{or: } Q_0 \equiv 1, Q_i \equiv \lceil Q_{i-1} R_i \rceil_{\text{sw}} \quad \text{for } i = 1..n \quad (\text{B.9b})$$

Multiplications in eqs. (B8-9) require only the multiplies of two g -bit integers (where g is usually a register size, such as $g=32$), unlike eqs. (B2-3) where one integer is of the size of output I (i.e. one operand has $O(\log(I))$ bits). Since the index values I are limited as:

$$0 \leq I \leq Q_n - 1 \equiv I_{\text{max}} \quad (\text{B.10})$$

and since all I values, being equiprobable, are encoded in a fixed number of bits (just large enough to fit the I_{max}), the QMR output length in bits, $L(I)$, is (denoting $Q_n \equiv (w,s)$):

$$L(I) = \lfloor \log(Q_n - 1) \rfloor + 1 = \lfloor \log(w - 1) \rfloor + s + 1 = L(w - 1) + s \quad (\text{B.11})$$

The output redundancy $\delta(g)$ (in bits per input symbol), resulting from the limited g -bit SW mantissa precision, has upper bound (cf. [3], p. 8) as follows:

$$\delta(g) \equiv [L(I_{\text{QMR}}) - L(I_{\text{MR}})]/n < 1/2^{g-1} \text{ bits/symbol} \quad (\text{B.12})$$

Block Bit Fractions Removal

The bit fraction loss occurs for sequences of blocks $\mathbf{B} = \mathbf{B}_1 \mathbf{B}_2 \dots \mathbf{B}_n$ where each block contains bit patterns restricted in a particular way: if the bit patterns are interpreted as integers N_i for blocks \mathbf{B}_i (in some high-low bit order convention for a block), then:

$$0 \leq N_i < M_i \quad \text{for } i=1..n \quad (\text{B.13})$$

where decoder knows the bounds M_i and where M_i have the following properties: *

a) Integers M_i are the *lowest integer bounds* on patterns in \mathbf{B}_i i.e. we have

$$0 \leq N_i \leq M_i - 1 \quad (\text{B.14})$$

b) The values for N_i are *distributed uniformly* in the interval $[0, M_i)$.

The minimum whole number of bits L_i that can fit any N_i satisfying (a) is:

$$L_i = \lfloor \log(M_i - 1) \rfloor + 1 \quad (\text{B.15})$$

which for $M_i = 2^c$, yields $L_i = c$. If we store \mathbf{B}_i in L_i whole bits, then the bit fraction loss is $\delta_i = L_i - \log(M_i)$, and $\delta_i > 0$ for M_i which are not power of 2 i.e.:

$$2^{L_i - 1} < M_i < 2^{L_i} \quad (\text{B.16})$$

due to the existence of unused bit patterns of all integers in the interval $[M_i, 2^{L_i})$. Hence we will assume that all blocks \mathbf{B}_i of interest here satisfy (B.16).

When there is only a single block \mathbf{B} with properties (a)-(b), one can partially remove the bit fraction loss $\delta = L - \log(M)$ (we drop subscript i) via *Tapered Huffman (TH)* codes as follows:

TH.1) Encode $n_0 \equiv 2^L - M$ values of N using $L - 1$ bits per value.

TH.2) Encode $n_1 \equiv M - n_0 = 2 \cdot M - 2^L$ values of N using L bits per value.

The average length \underline{N} of TH(N) codes and the average redundancy $\delta(M)$ are:

$$\underline{N} = [n_0(L - 1) + n_1 L] / M = L + 1 - 2^L / M \quad (\text{B.17})$$

$$\delta(M) = \underline{N} - \log(M) \quad (\text{B.18})$$

For example if $M = 6$, we have $L = \lfloor \log(6-1) \rfloor + 1 = \lfloor 2.32 \rfloor + 1 = 3$, hence $n_0 = 2^3 - 6 = 2$ (the # of 2 bit codes) and $n_1 = 6 - n_0 = 4$ (the # of 3 bit codes). The TH(N) codes are:

N	TH(N)
-----	-----------

* Note that the low bound 0 of N_i in (B.13) is a convention i.e. if the low bound for N_i is integer $L_i > 0$, then we define some new integers $N'_i = N_i - L_i$ and $M'_i = M_i - L_i$ so that (B.13) holds for N'_i and M'_i .

0	00
1	01
2	110
3	101
4	110
5	111

The average code length $\underline{N} = 4 - 8/6 = 2.666$ and the optimal length is $\log(6) = 2.585$, hence the average redundancy $\delta(6) = 0.0817$ bits/symbols, which is a 3.2% excess. The worst TH performance case (max δ for M satisfying B.16) occurs for integer M nearest to real number $M_0 = 2^L/\log(e) = 2^L/1.4427... \approx 2^L \cdot 2/3 \approx 2^L \cdot 192/277$ where the redundancy reaches its maximum $\delta(M_0) = 1 + \log(\log(e)) - \log(e) = 0.086071$ bits/symbol.

When there is more than one block ($n > 1$) with such block bit fraction losses, QMR method described here reduces the redundancy to $\delta_Q(n) < \delta(M_0)/n + 1/2^{g-1}$ bits/symbol. We define SW integers $Q_i \equiv \lceil M_i \rceil_{sw}$ for $i=1..n$, hence we have:

$$Q_i \equiv (w_i, s_i) \equiv \lceil M_i \rceil_{sw} \geq M_i \geq N_i + 1 > N_i \quad (\text{B.19})$$

Recalling the SW integer representation (B.6) and using (B.19) we can express the bit patterns of Q_i , Q_i-1 and N_i as follows (bit significance increases from right to left):

$$\begin{aligned}
Q_i &= \overbrace{\text{xxx}\cdots\text{x}}^{g\text{-bit } w_i} \overbrace{\text{0000}\cdots\text{0000}}^{s_i \text{ bits}} \\
0 \leq N_i \leq Q_i - 1 &= \overbrace{\text{yyy}\cdots\text{y}}^{g\text{-bit } w_i-1} \overbrace{\text{1111}\cdots\text{1111}}^{s_i \text{ bits}} \\
N_i &= \overbrace{\text{zzz}\cdots\text{z}}^{\text{top } g \text{ bits}} \overbrace{\text{tttt}\cdots\text{tttt}}^{s_i \text{ bit tail}}
\end{aligned} \quad (\text{B.20})$$

The bit patterns $\text{xx}\dots\text{x}$ and $\text{yy}\dots\text{y}$ in (B.20) represent integers w_i and w_i-1 . Denoting the top g bits of N_i (the pattern $\text{zz}\dots\text{z}$ in (B.20)) as d_i , the integers d_i have properties:

$$0 \leq d_i \leq w_i - 1 < w_i \quad (\text{B.21})$$

and values of d_i are uniformly distributed in the interval $[0, w_i)$. Therefore we can encode the sequence d_i taken as digits of an integer D in radices $R_i = w_i$ via QMR eqs. (B.8-9):

$$D = d_1 + d_2 U_1 + d_3 U_2 + \cdots + d_n U_{n-1} \quad (\text{B.22})$$

$$\text{where: } U_0 \equiv 1, \quad U_i \equiv \lceil U_{i-1} w_i \rceil_{sw} \quad \text{for } i = 1..n \quad (\text{B.23})$$

at the redundancy $\delta(g) < 1/2^{g-1}$ bits per digit d_i . Since the s_i tail bits \mathbf{T}_i of N_i (denoted as pattern $\text{tt}\dots\text{t}$ in eq. (B.20)) are uniformly distributed in the interval $[0, 2^{s_i})$, their optimal encoding requires s_i whole bits, hence they need no further processing. Therefore, the full redundancy per block introduced by QMR encoding is $\delta(g) < 1/2^{g-1}$ bits per block \mathbf{B}_i . The resulting encoded output consists of $\text{QMR}(\mathbf{B}) = D \mathbf{T}_1 \mathbf{T}_2 \dots \mathbf{T}_n$, where the length of D is determined via (B.11) from the value U_n computed in (B.23).

Further, since the SW integer $U_n = (w_u, s_u)$ limits the max value of integer D as:

$$0 \leq D \leq U_n - 1 \quad (\text{B.24})$$

then, applying (B.20) to U_n and D , we obtain the following constraint on the top g bits of D (denoted here as integer d):

$$0 \leq d \leq w_u \quad (\text{B.25})$$

If the block sequence \mathbf{B} is the sole coding task^{*} which has to be transmitted in the whole number of bits, then we would encode D via Tapered Huffman code, hence our output would consist of sequence: QMR-TH(\mathbf{B}) = TH(D) $\mathbf{T}_1 \mathbf{T}_2 \dots \mathbf{T}_n$.

Decoding of QMR-TH(\mathbf{B}) assumes that decoder knows array of M_i for $i = 1..n$, along with the block sizes[†]. Decoder computes $Q_i = (w_i, s_i) \equiv \lceil M_i \rceil_{sw}$ via (B.19), and computes sequence U_i via (B.21). From $U_n = (w_u, s_u)$ it computes the length of D and decodes TH(D), which provides it a bit pointer into the sequence of tail bits $\mathbf{T}_1 \mathbf{T}_2 \dots \mathbf{T}_n$ of $\mathbf{B}_1 \mathbf{B}_2 \dots \mathbf{B}_n$. QMR decoder decodes then D into sequence of digits $d_1 d_2 \dots d_n$ which are then concatenated with the corresponding tail blocks \mathbf{T}_i for $i = 1..n$. The length of each tail block \mathbf{T}_i and the number of bits from the ‘top g bits’ of \mathbf{B}_i [‡] is known to decoder from the knowledge of Q_i (cf. eq. (B.20)).

^{*} If there are additional components, they could be e.g. included in the original sequence of locks \mathbf{B} . Note also that eqs. (B.22-24) establish properties (a) & (b) for the entire output as block of type \mathbf{B}_i , allowing thus inclusion of the output in further application of the same method hierarchically.

[†] The blocks \mathbf{B}_i contain typically indices from some other coding sub-task e.g. from binary QI coder output, where the $M_i = Q_i$ is the top binomial coefficient for the count of ones, which decoder would have to know.

[‡] The ‘top g bits’ may be fewer than g bits if the $Q_i = (w_i, s_i)$ has $s_i=0$, thus w_i may have fewer than g significant bits. The number of bits assigned to $d_i < w_i$ is, via (B.20), the number of significant bits in w_i-1 .

Hierarchical Radix Coding (HRC)

Although QMR coding (B8-9) reduces the coding work and tables memory size by a factor $O(n)$ of the conventional mixed radix coding, the memory use for the array of Q_i values may still become a problem for large enough n . Namely, while the QMR encoder can use recurrence (B.9b) without storing Q_i 's into an array (since the additions in (B.8) can be done in the same order as computations in (B.9): $i=1,2,..n$), the QMR decoder uses Q_i 's in order $i=n, n-1,..1$, which is reverse from the recurrence (B.9), hence the decoder needs to store all Q_i 's generated by (B.9) into an array of n SW integers. **HRC** method solves this problem of table space for large n in case of *fixed radix* (or cases of 'few' distinct radices).

In HRC method we assume fixed radix for the input sequence $\mathbf{A}[n]$:

$$R_i = R \text{ for } i=1..n \quad (\text{B.26})$$

The table for quantized volumes $Q_i = \lceil Q_{i-1}R \rceil_{\text{sw}}$ requires space for n SWI entries. The 2-level HRC method, presented first, reduces the table size to $\approx 2\sqrt{n}$ entries, while general k -level HRC variant reduces the table size to $\approx k n^{1/k}$ entries.

2-LEVEL HRC

ENCODING

a) We segment the input sequence $\mathbf{A}[n]$ into m blocks: $\mathbf{B} = \mathbf{B}_1 \mathbf{B}_2 \dots \mathbf{B}_m$, of lengths (in symbols) $d_j = d$ for $j=1..m-1$, and the last block of length $d_m \leq d$. We compute m , d and d_m from n and $r \equiv \lfloor \sqrt{n} \rfloor$ as follows*:

$$n - r^2 \leq r \Rightarrow m=r+1, \quad d=r, \quad d_m = n - r^2 \quad (\text{B.26a})$$

$$n - r^2 > r \Rightarrow m=r+1, \quad d=r+1, \quad d_m = n - r^2 - r \quad (\text{B.26b})$$

b) We compute via eqs. (B8-9) the QMR indices $\mathbf{I}[m] \equiv I_1, I_2, \dots, I_m$ for blocks $\mathbf{B}_1 \mathbf{B}_2 \dots \mathbf{B}_m$. Since the radix R is the same for all blocks and block sizes $d \leq r+1$, we need a single Q_i table (from (B.9)) common to all blocks and this table needs at most $r+1$ entries (for $i=1..d$). The maximum quantized volumes (Q_n of eq. (B.8)) are the same SWI value $Q \equiv (w, s)$ for all blocks \mathbf{B}_i , $i=1..m-1$, and possibly different value $Q' \equiv (w', s')$ for the last block \mathbf{B}_m , where $Q' \leq Q$.

c) Since the $\mathbf{I} \equiv I_1, I_2, \dots, I_m$ have uniformly distributed values in the interval $[0, Q)$ (or $[0, Q')$ for the I_m), and the values Q and Q' are known to decoder, we can apply QMR-TH(\mathbf{I}) method of the previous section to sequence $\mathbf{I}[m]$, with blocks I_1, I_2, \dots, I_m . Hence we encode the 'top g bits' d_j of each index I_j for $j=1..m-1$ using radix w (mantissa of Q) for and d_m (for the last block) using radix w' (mantissa of Q'). The table requirements for this computation are m SWI entries of eq. (B.9), which by (B.26) is limited to at most $r+1$ entries (the same as for step (b)).

* The lengths (B.26) are selected to minimize the total table sizes, which will have of $d+m$ SWI entries. Without the table minimization requirement, the sizes d_j and m can be arbitrary.

Therefore, the HRC de/encoding require 2 tables with at most $r+1$ SWI entries per table, which is $2 \sqrt[n+2]{n}$ entries instead of n entries used by the direct QMR coding of $\mathbf{A}[n]$.

DECODING

HRC decoder computes m , d and d_m values via (B.26) and known n . Then it computes quantized volumes Q and Q' from the known radix R , d and d_m via (B.9). With these it invokes the QMR-TH(I) decoder to reconstruct sequence of indices $\mathbf{I}[m] \equiv I_1, I_2, \dots, I_m$. Then regular QMR it decodes each of the indices I_j back into the block \mathbf{B}_j (sequence of symbols in radix R) recovering thus the original sequence $\mathbf{A}[n]$.

***k*-LEVEL HRC**

For brevity, we will consider the input length n which is a k -th power of an integer*:

$$n = m^k \tag{B.27}$$

- a) We split the input sequence $\mathbf{A}[n]$ into m_1 equal blocks $\mathbf{B}_1(j_1)$, for $j_1=1..m_1$ with length (in symbols) $d = m$ symbols per block. Hence there are $m_1 = m^{k-1}$ blocks. We then encode each block via QMR, with radix R and block length m symbols. Since all blocks are of equal size and equal radix, we need only a single Q table (from eq. (B.9)) of m entries. We obtain m_1 indices $I_1(j_1)$ for $j_1=1..m_1$. All indices have the same quantized upper bound $Q_1 = (w_1, s_1)$ (which was denoted as Q_n in eqs. (B.9),(B.20)).
- b) With known upper bound Q_1 for indices, we can split each index $I_1(j_1)$ into the ‘top g bits’, integer $d_1(j_1)$, and a block of tail bits $\mathbf{T}_1(j_1)$. The latter is at maximum entropy, hence it can be output without further encoding into the tail sequence buffer (which accepts m_1 such tails). The $m_1 = m^{k-1}$ integers $d_1(j_1)$ form a sequence of the $\mathbf{A}[n]$ type used in step (a) (equiprobable symbols in a fixed radix), which after replacement: $m_1 = m^{k-1} = n/m \rightarrow n$ and $w_1 \rightarrow R$ is passed back to step (a), provided the new $n > 1$. Otherwise the encoding loop terminates.

The termination condition $n=1$ at the end of step (b) is reached after k executions of step (b), since each execution of (a) and (b) reduces the input n by factor m . Since step (a) requires a new m -entry Q table, the total number of tables at the termination is k , hence the total number of Q table entries is $k \cdot m \approx k \cdot n^{1/k}$. The table *space reduction factor* $f(k,n)$ is thus:

$$f(k,n) = n / (k \cdot n^{1/k}) = n^{1-1/k} / k \tag{B.28}$$

* More generally one would define $r = \lfloor n^{1/k} \rfloor$, as for the 2-level HRC, and compute from n and r the number of blocks and block sizes. But as we have seen in the $k=2$ case, that level of detail merely makes the last block shorter, without affecting the maximum table size (which is the topic of optimization here).

which achieves its maximum for $k = \lceil [L + (L^2 - 4L)^{1/2}] / 2 \rceil$, where $L = \ln(n)$.

MIXED RADIX HRC

The table space savings of the HRC methods above was based on the reuse of the same Q_i tables for different blocks (which also resulted in the same limits on the ‘top g bits’ of block indices, which allowed for hierarchical reuse). While the fixed radix for the entire input $\mathbf{A}[n]$ is sufficient condition to have such table reusability, it is not a necessary condition i.e. the tables may be reusable even for variable radices, provided the radix variations are repeated from block to block. For example, $\mathbf{A}[n]$ may have radices R_i which simply flip between two (or a few) values depending on whether i is even or odd, and the previous HRC methods would still work (with minor extra attention in setting the block sizes).

Generally, one would not have such convenient radix variations in the array $\mathbf{A}[n]$. The method presented here allows HRC coder to greatly extend usefulness of the previous HRC methods. We note first that in this coding setup coder & decoder know the radices R_i (which now may vary for $i = 1..n$). Consider now a variant of our even/odd radix regularity example mentioned above, but allow now that there are two radices R_1 and R_2 which vary in some ‘random’ fashion (albeit known to de/coder in any given coding instance).

The solution for this case is to have a convention between coder & decoder on how to *permute the input array $\mathbf{A}[n]$ based on the radix values R_i* in the given coding instance (or any other info about the input array which may make the permutation construction simpler). In the ‘random’ (but known) R_1 and R_2 switches, this convention could be to separate the coding task and code R_1 symbols as an input array $\mathbf{A}_1[n_1]$ and R_2 symbols as an input array $\mathbf{A}_2[n_2]$, where n_1 and n_2 are counts of occurrences of R_1 and R_2 symbols (which decoder knows, thus it knows how to find the boundary between \mathbf{A}_1 and \mathbf{A}_2).

Appendix C: Removing SWI Exponents from QI Tables

The QI tables of quantized enumerative addends use SW integers (cf. eq. (B4)) for table entries. In many cases the explicit formulas for enumerative addends within the exact enumerative coding (EC) are also known. The examples of such cases include quantized binomials for binary & multi-alphabet coders, quantized powers for fixed radix coder, quantized factorials for permutation coders, quantized radix products for mixed radix coders, quantized Ballot and Catalan numbers for tree coders... etc (cf [2],[3]). The RXL method below utilizes these exact formulas to compute SWI exponents from the integer part of the exact logs (stored in tables) of addends.

RXL: Exponents Removal via Log Tables

The present method (**RXL**) removes the exponent fields present in each SWI table entry and replaces them with much smaller global interpolation parameters (which apply to the entire table). In a typical QI implementation (optimized for speed), the SWI exponents and mantissas are 32-bit integers stored in a structure such as (cf. [4], file: qi.h):

```
typedef unsigned int dword; // 32 bit unsigned

typedef struct _swi {
    dword m; // SWI mantissa
    int e; // SWI exponent
} SWI;
```

Code C.1.

The removal of the exponents reduces thus the table sizes in half. For example, the table for quantized binomials with max blocks size n requires $n^2/4$ SWI entries. The RXL SWI exponent replacement tables require n integers while removing $n^2/4$ integers (SWI.e fields) from the tables. For typical $n = 1024$ to 4096 , the space savings range from 1 to 16 MBytes of memory.

The basic idea of RXL is to compute the integer part of the $\log(V)$, where V is the exact enumerative addend as a replacement for the $Q.e$ field of quantized addend Q (where Q is of type SWI, Code C.1). The mathematical basis for applicability of the RXL method is the relatively small quantization error $\delta(g) < 1/2^{g-1}$ bits/symbol (cf. [3] p. 8). Thus for each iterative step which computes quantized addends Q , on average 0.5 is added to the integer in range $2^{31}-2^{32}$. In order for the exponent to be affected by such small addends (which gets incremented when mantissa doubles), one would need on average more than 2^{30} additions. Of course, if mantissa is close to 2^{32} , e.g. $Q.m = 0xFFFF,FFFF$, even a single add of +1 will cause mantissa overflow and a need to increment exponent. But such cases are easily recognized (since the resulting mantissa will be close to $0x8000,0000$ after renormalization), and corrected as shown in the actual implementation below. Table C.1 shows the typical quantization errors for $n=1024$ and some values of k .

We examine RXL for the important case of quantized binomial coefficients (cf. eqs. (5)-(7) [2]). The exact addend $V(n, k)$ for some number n of input bits processed and the count k of ones encountered so far, is a binomial coefficient:

$$V(n, k) = \binom{n}{k} \equiv \frac{n!}{k!(n-k)!} \quad (\text{C.1})$$

$$\log(V(n, k)) = \log(n!) - \log(k!) - \log((n-k)!) \quad (\text{C.2})$$

From (C.2) it follows that we can compute integer part of $\log(V)$ using a single table of n integer entries, each entry containing fixed point form of logarithms of factorials $F[i!]$, for $i = 1..n$. Since SWI use integer mantissa normalized to fall between 2^{g-1} and 2^g (where g is the mantissa precision, usually $g=32$), as shown in eq. (B.5), while regular FP numbers normalize mantissa to fall between 0.5 and 1.0, in order to obtain the SWI exponent from FP exponent, we need to rescale the FP value by subtracting 32 from the FP exponent, as shown in the Code C.2 (from [4], file `Swi.c`):

```

//-- Convert FP number to nearest SWI

SWI fp2swi(double x,      // FP number to convert
           double Round  // rounding: 0.0=down, 0.5 nearest
           )             // 0.99 round up
{ SWI Q;                // SWI used for conversion
  double xm;            // FP mantissa
  int k=32;             // scaling via exponent subtraction
  xm=frexp(x,&Q.e);     // separate FP mantissa & exp
  if (Q.e<=32)         // the whole x fits in 32 bits?
    k=Q.e;             // yes, use denormalized SWI
  Q.e-=k;              // rescale FP via its exponent
  if (Round>0.5001)    // Check type of rounding requested
    Q.m=(dword)
      ceil(ldexp(xm,k)); // Rounding mantissa up
  else
    Q.m=(dword)
      floor(ldexp(xm,k)+Round); // Round down/nearest
  return Q;            // return computed SWI
}

```

Code C.2

The Code C.3 ([4] file `Qi.c`) shows the computation of the $\log(x!)$ in fixed point format (thus treated as integers) used by the coding algorithm. The fixed point format reserves low 12 bits (value `XP32`) of integer for fractional part and upper 20 bits for integer part of $\log(x!)$. Code C.3 converts the precomputed log factorial table of type `double lf[i]` (computed elsewhere, cf. [4], function `lg_start` in file `Qi.c`) via recurrence:

$$\log(x!) = \log((x-1)!) + \log(x) = lf[x-1] + \log(x)$$

```

/-- Init exponents interpolation table

#define BM(b) ((dword)1<<(b)) // 1 bit bitmask
#define BML(b) (BM(b)-1) // low part bitmask
#define XP32 12 // 12 fractional bits to
// keep + 20 bits for integer part
#define XP32ER (-15.99999) // roundoff adjustment for exp
// table (Ok to n=256K, at least)
#define XP32M BML(XP32) // mask for fractional part

int bcexp_start(int n, // Maximum n for the table
                dword *expTbl // Destination array for the fixed
                ) // point log(x!) entries
{ int i;
  double x;
  double *lf=lgf; // lf=logs of factorials table in FP double
  dword *t=expTbl; // logs in fixed point format and the SWI bias

  if (!lf || !t)
    return 0;
  if (n>maxlg)
    return 0;

  t[0]=t[1]=0;
  for(i=2; i<=n+1; ++i)
  {
    x=lf[i]; // calc 32 bit fixed point logs of factorials
    x=ldexp(x,XP32); // for C(n,k) exponents
    t[i]=(dword)floor(x+XP32ER);
  }
  return 1;
}

```

Code C.3

The actual computation of the SWI exponent for quantized binomial $Q(n,k)$ is shown in Code C.4 (from [4], file Qi.h). The macro BCLOG uses eq. (C.2) to compute the fixed point $\log(x!)$, then the macro EXP32 shifts out the fixed point fraction to get integer part of FP exponent. It then rescales FP number (to match SWI exponent bias) by subtracting 32 from the FP exponent. The overflow adjustments occur at the end of the macro, where the mantissa is checked against the overflow threshold via **((b).m<0xFF000000)** and exponent is adjusted if mantissa goes above the threshold. Similarly, the fractional part of the fixed point $\log(bc)$ is checked for its own overflow **((te&XP32M)>30)** and the same adjustments to the SWI exponent (b).e is done. Note that the tests result of “no overflow” falls into ++(b).e branch, i.e. the exponent was preadjusted (pre-decremented) for any overflow, then adjusted back if tests return that no overflow occurred.

```

/-- Computation of Quantized Binomial exponent
#define BCLOG(t,n,k,l) ((t[n]-t[k]-t[l])) // calc log of binomial
// via log(k!) table as
// log(C(n,k))=log(n!)-log(k!)
// -log((n-k)!)

#define EXP32(t,n,k,l,b) \ // Table int t[] is contains
{ dword te,ti; \ // fixed point log(x!)values
  te=BCLOG((t),(n),(k),(l)); \ // Compute fixed point log bc(n,k)
  ti=te>>XP32; \ // Shift out fraction of fixed
  (b).e=0; \ // point result
  if ((int)(ti-=32)>=0) \ // check for denormalized SWI
  { \
    (b).e=ti; \
    if ((te&XP32M)>30 || \ // check for FP exp overflow
        (b).m<0xFF000000) \ // and mantissa vicinity to 2^32
        ++(b).e; \ // adjust computed SWI exponent
  } \ // if no overflow
}

```

Code C.4

Table C.1 Columns

The table shows the mantissa errors due to quantization of binomials $Q(n,k)$. The computation was done for $n=1024$ and various values of k .

QBC gives the quantized binomial coefficient (mantissa in hex and exponent in decimal)
XBC column the same for the more accurately (within the 53 bit precision of C double)
 computed binomial coefficient.

The columns **dm16** and **dm10** show differences $QBC - XBC$.mantissa (exponents will always be normalized to the same value) in hex and decimal. The column **ddm** (delta dm) shows the difference between the current dm10 value and the one from the previous line.

The column **Extra Bits** shows the total excess bits for the block of size n due to the expansion of the mantissa (the small fractional values may yield a whole bit difference in multi-block coding or even statistically on a single block). The column **Gap** shows '*' symbol next to the QBC mantissas which were rounded up from the immediate previous add of the binomials: $C(n,k) = C(n-1,k) + C(n-1,k-1)$.

Gaps & Mantissa Excess in QI Binomial Tables ($n=1024$)

k	Gap	QBC.mant:exp	-	XBC.mant:exp	=	dm16	dm10	ddm	Extra Bits		
1.		00000400:	0	-	00000400:	0	=	0	0	+0	0.00000e+000
2.		0007FE00:	0	-	0007FE00:	0	=	0	0	+0	0.00000e+000
3.		0AA2AC00:	0	-	0AA2AC00:	0	=	0	0	+0	0.00000e+000
4.	*	A9AB202F:	4	-	A9AB1FF0:	4	=	3f	63	+63	3.19296e-008
5.	*	87345DD0:	12	-	87345D73:	12	=	5d	93	+30	5.91488e-008
6.	*	B3647B52:	19	-	B3647AA8:	19	=	aa	170	+77	8.14891e-008
7.	*	CBD197B2:	26	-	CBD196C9:	26	=	e9	233	+63	9.83029e-008
8.	*	CA6CE909:	33	-	CA6CE801:	33	=	108	264	+31	1.12149e-007
9.	*	B2872A13:	40	-	B287290F:	40	=	104	260	-4	1.25234e-007
10.	*	8D912E6D:	47	-	8D912D8F:	47	=	de	222	-38	1.34848e-007
11.	*	CBE7A80C:	53	-	CBE7A6B6:	53	=	156	342	+120	1.44229e-007
12.	*	8679F208:	60	-	8679F11D:	60	=	eb	235	-107	1.50271e-007
13.	*	A391DDCD:	66	-	A391DCAA:	66	=	123	291	+56	1.52983e-007
14.	*	B8904D75:	72	-	B8904C21:	72	=	154	340	+49	1.58412e-007
15.	*	C22D26DF:	78	-	C22D256D:	78	=	172	370	+30	1.63855e-007
16.	*	BF54FD9B:	84	-	BF54FC21:	84	=	17a	378	+8	1.69887e-007
17.	*	B1437283:	90	-	B143711E:	90	=	165	357	-21	1.73182e-007
18.	*	9AF39FE6:	96	-	9AF39EA8:	96	=	13e	318	-39	1.76476e-007
19.	*	80312159:	102	-	8031204F:	102	=	10a	266	-52	1.78433e-007
20.	*	C94D2663:	107	-	C94D24BD:	107	=	1a6	422	+156	1.80269e-007
...		...		-	...		=
100.	*	821A287F:	437	-	821A2748:	437	=	137	311	-228	2.05556e-007
101.	*	94C7BC40:	440	-	94C7BADD:	440	=	163	355	+44	2.05181e-007
102.	*	A84A1540:	443	-	A84A13AE:	443	=	192	402	+47	2.05411e-007
103.	*	BC4DEC4A:	446	-	BC4DEA86:	446	=	1c4	452	+50	2.06411e-007
104.	*	D07290F3:	449	-	D0728EFF:	449	=	1f4	500	+48	2.06266e-007
105.	*	E44CB722:	452	-	E44CB4FF:	452	=	223	547	+47	2.06033e-007
106.	*	F76A14F4:	455	-	F76A12A4:	455	=	250	592	+45	2.05755e-007
107.		84AAD39C:	459	-	84AAD25F:	459	=	13d	317	-275	2.05471e-007
108.	*	8CCE31C6:	462	-	8CCE3076:	462	=	150	336	+19	2.05199e-007
109.	*	93E90A03:	465	-	93E908A1:	465	=	162	354	+18	2.05807e-007
110.	*	99CB08F5:	468	-	99CB0785:	468	=	170	368	+14	2.05762e-007
...		...		-	...		=
300.		9AED3564:	857	-	9AED3435:	857	=	12f	303	+51	1.68179e-007
301.	*	BA52DCB3:	858	-	BA52DB48:	858	=	16b	363	+60	1.67530e-007
302.	*	DF087C50:	859	-	DF087A9D:	859	=	1b3	435	+72	1.67716e-007
303.		84DCEC45:	861	-	84DCEB43:	861	=	102	258	-177	1.66982e-007
304.	*	9D8E6757:	862	-	9D8E6624:	862	=	133	307	+49	1.67555e-007
305.	*	B9F7D21B:	863	-	B9F7D0B1:	863	=	16a	362	+55	1.67388e-007
306.		DA7B6908:	864	-	DA7B6760:	864	=	1a8	424	+62	1.66880e-007
307.	*	FF7D226E:	865	-	FF7D207F:	865	=	1ef	495	+71	1.66605e-007
308.	*	94B0839F:	867	-	94B0827F:	867	=	120	288	-207	1.66559e-007
309.	*	AC44A26F:	868	-	AC44A122:	868	=	14d	333	+45	1.66224e-007
310.	*	C6A9FD63:	869	-	C6A9FBE3:	869	=	180	384	+51	1.66214e-007
...		...		-	...		=
500.		9A2EDF6B:	987	-	9A2EDE91:	987	=	da	218	+10	1.21583e-007
501.		A142E8A6:	987	-	A142E7C4:	987	=	e2	226	+8	1.20513e-007
502.	*	A801E20F:	987	-	A801E124:	987	=	eb	235	+9	1.20280e-007
503.		AE5A82C4:	987	-	AE5A81D2:	987	=	f2	242	+7	1.19355e-007
504.		B43C0A38:	987	-	B43C093F:	987	=	f9	249	+7	1.18800e-007
505.		B9968940:	987	-	B9968841:	987	=	ff	255	+6	1.18153e-007
506.	*	BE5B299D:	987	-	BE5B2898:	987	=	105	261	+6	1.17904e-007
507.		C27C7236:	987	-	C27C712C:	987	=	10a	266	+5	1.17611e-007
508.		C5EE865F:	987	-	C5EE8552:	987	=	10d	269	+3	1.16867e-007
509.	*	C8A75E79:	987	-	C8A75D69:	987	=	110	272	+3	1.16567e-007
510.		CA9EF87E:	987	-	CA9EF76C:	987	=	112	274	+2	1.16284e-007

Table C.1

References

1 2 3 4

-
- ¹ **R. V. Tomic** *Quantized indexing: Background information* 1stWorks TR05-0625, 1-39, 2005
<http://www.1stworks.com/ref/TR/tr05-0625a.pdf>
 - ² **R. V. Tomic** *Fast, optimal entropy coder* 1stWorks TR04-0815, 1-52, 2004
<http://www.1stworks.com/ref/TR/tr04-0815b.pdf>
 - ³ **R. V. Tomic** *Quantized Indexing: Beyond Arithmetic Coding* arXiv cs.IT/0511057, Nov 2005
<http://arxiv.org/abs/cs.IT/0511057>
 - ⁴ **R. V. Tomic** *QIC Research kit*, Version 1.03 (QI source code)
<http://www.1stworks.com/ref/C/QIC100.zip>