

### Order-1 encoding

We start with the outer context byte, emitting the symbol if it is non-zero frequency. We perform the same run-length-encoding as we use for the Order-0 table and end the contexts with a nul byte. After each context byte we emit the Order-0 table relating to that context.

Consider ~~abracadabraabracadabraabracadabraabracadabr~~abraçadabraabracadabraabracadabraabracadabrad as example input. Note for the ~~last “a” was omitted from this string in order to demonstrate how the method works when the data is not a multiple of 4 long.~~additional trailing “d” giving us 45 characters instead of 44. This can be broken into 4 approximate equal portions ~~abracadabra abracadabra abracadabra~~abracadabrabraçadabrad. We operate one independent rANS stream per portion, providing us the opportunity to exploit CPU data parallelism.

Normalised (per Order-0 statistics):

Context	Symbol	Frequency
\0	a	4095
a	a	<del>646</del> 614
	b	<del>1725</del> 1639
	c	<del>862</del> 819
	d	<del>862</del> 1023
b	r	4095
c	a	4095
d	a	4095
r	a	4095

The above tables are encoded as:

```

0x61                                     # 'a' context
0x61 0x82 0x86 # a <646>
0x62 0x02 0x86 0xbd # b <+2: c,d> <1725>
0x83 0x5e # c (implicit) <862>
0x83 0x5e # d (implicit) <862>
0x61 0x82 0x66 # a <614>
0x62 0x02 0x86 0x66 # b <+2: c,d> <1638>

```

```

0x83 0x33 # c (implicit) <819>
0x84 0x00 # d (implicit) <1024>
0x00      # end of Order-0 table

0x62 0x02      # 'b' context, <+2: c, d>
0x72      0x8f 0xff # r <4095>
0x00      # end of Order-0 table

      # 'c' context (implicit)
0x61      0x8f 0xff # a <4095>
0x00      # end of Order-0 table

      # 'd' context (implicit)
0x61      0x8f 0xff # a <4095>
0x00      # end of Order-0 table

0x72      # 'r' context
0x61      0x8f 0xff # a <4095>
0x00      # end of Order-0 table

0x00      # end of contexts

```

## 2.2 rANS entropy encoding

The encoder takes a symbol  $s$  and a current state  $x$  (initially  $L$  below) to produce a new state  $x'$  with function  $C$ .

$$x' = C(s, x)$$

The decoding function  $D$  is the inverse of  $C$  such that  $C(D(x)) = x$ .

$$D(x') = (s, x)$$

The entire encoded message can be viewed as a series of nested  $C$  operations, with decoding yielding the symbols in reverse order, much like popping items off a stack. This is where the asymmetric part of ANS comes from.

As we encode into  $x$  the value will grow, so for efficiency we ensure that it always fits within known bounds. This is governed by

$$L \leq x < bL - 1$$

where  $b$  is the base and  $L$  is the lower-bound.

We ensure this property is true before every use of  $C$  and after every use of  $D$ . Finally to end the stream we flush any remaining data out by storing the end state of  $x$ .

### Implementation specifics

We use an unsigned 32-bit integer to hold  $x$ . In encoding it is initialised to  $L$ . For decoding it is read little-endian from the input stream.

Recall  $freq_i$  is the frequency of the  $i$ -th symbol  $s_i$  in alphabet  $\mathbb{A}$ . We define  $cfreq_i$  to be cumulative frequency of all symbols up to but not including  $s_i$ :

$$cfreq_i = \begin{cases} 0 & \text{if } i < 1 \\ cfreq_{i-1} + freq_{i-1} & \text{if } i \geq 1 \end{cases}$$

We have a reverse lookup table  $cfreq\_to\_sym_c$  from 0 to 4095 (0xfff) that maps a cumulative frequency  $c$  to a symbol  $s$ .

$$cfreq\_to\_sym_c = s_i \quad \text{where} \quad c : cfreq_i \leq c < cfreq_i + freq_i$$

The  $x' = C(s, x)$  function used for the  $i$ -th symbol  $s$  is:

$$x' = (x / freq_i) \times 0x1000 + cfreq_i + (x \bmod freq_i)$$

The  $D(x') = (s, x)$  function used to produce the  $i$ -th symbol  $s$  and a new state  $x$  is: