

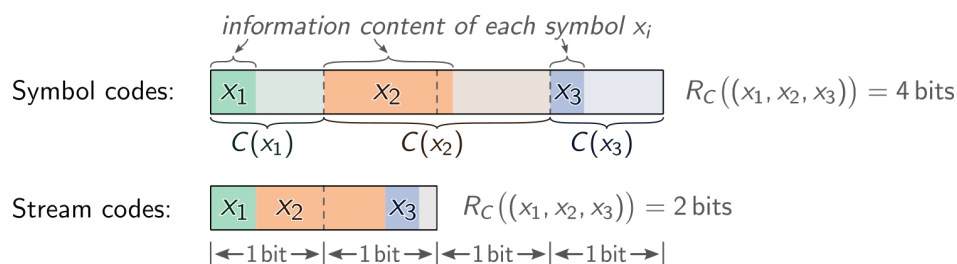
# Stream Codes II: Asymmetric Numeral Systems (ANS)

Lecture 6 (2 June 2022); lecturer: Robert Bamler

[Duda et al., 2015]

more course materials online at <https://robamler.github.io/teaching/compress22/>

## Recap from last lecture



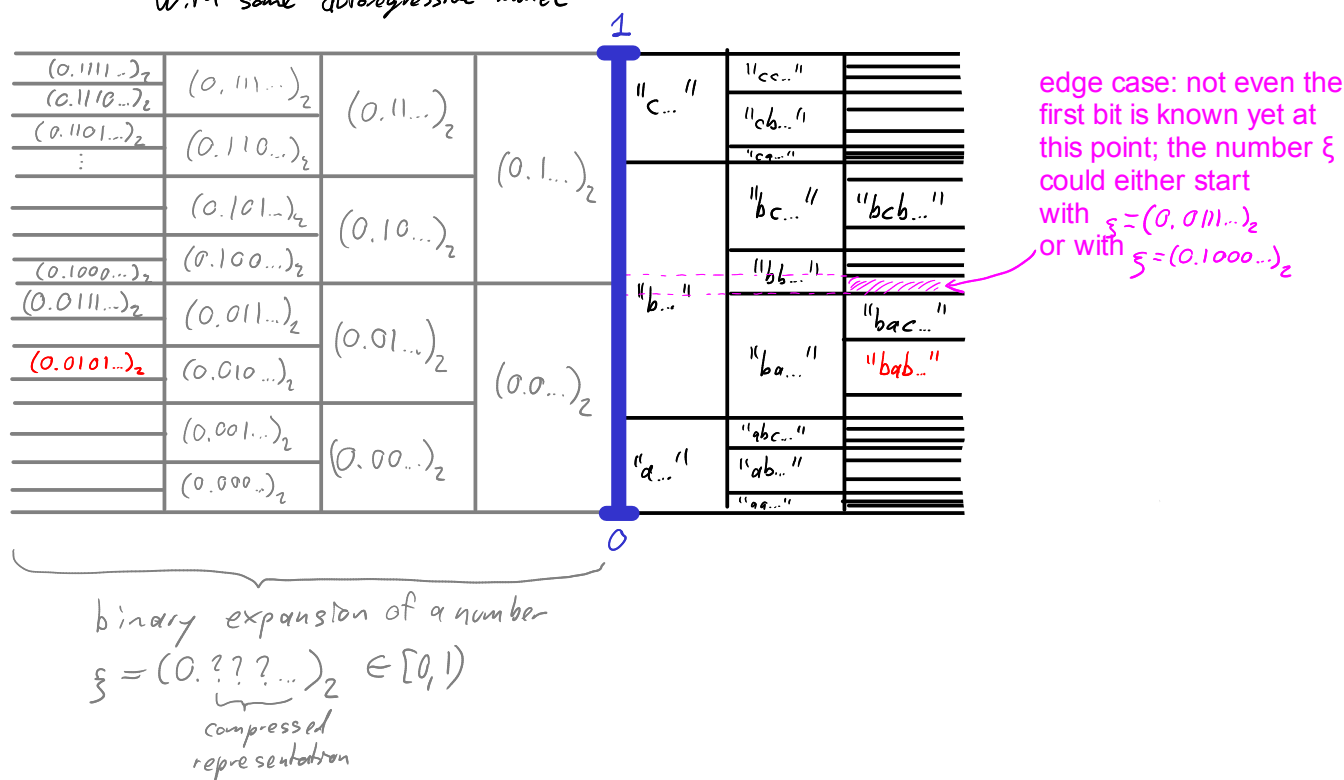
### - symbol codes vs. stream codes:


### - two stream codes: arithmetic coding and range coding

( $\approx$  computationally efficient variant of Shannon coding for sequences of symbols)

iterate over symbols in message; each symbol refines an a subinterval of  $[0, 1)$ .

Example: encoding a sequence  $x \in \mathcal{X}^*$  with alphabet  $\mathcal{X} = \{ "a", "b", "c" \}$  and with some autoregressive model



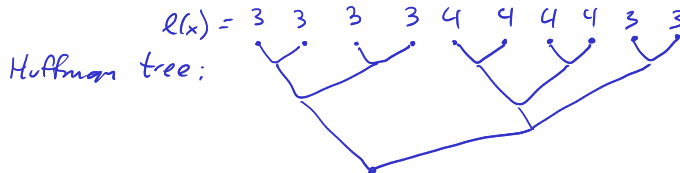
- Range coding is similar to arithmetic coding but it uses a larger base (e.g.,  $2^{32}$  instead of 2) on the left side of the above illustration. This makes range coding more computationally efficient in practice since operations with single bits typically require manipulating larger registers anyway on standard computing hardware.
- Arithmetic coding and range coding are conceptionally relatively simple, but a complete implementation is somewhat involved because of a number of edge cases (like  above).
- You'll use a range coder from a library in Problem 6.3.
- Today, we'll discuss and live-implement a different stream code that is conceptionally a bit more involved, but that turns out to be relatively easy to implement.

## Exercise

Consider a data source that generates a random message  $\mathbf{X} \equiv (X_1, X_2, \dots, X_k)$  of length  $k$ , where each symbol  $X_i, i \in \{1, \dots, k\}$  is drawn independently from all other symbols from a uniform probability distribution over the alphabet  $\mathcal{X} = \{0, 1, 2, \dots, 9\}$ .

(a) What is the entropy per symbol?  $\frac{1}{k} H_P[\mathbf{X}] = H_P[X_i] = \mathbb{E}_P[-\log_2 \frac{1}{10}] = \log_2 10 \approx 3.32 \text{ bit}$

(b) What is the expected code word length of an *optimal* symbol code for this data source?  $L := \mathbb{E}_P[\ell_{\text{Huff}}(X_i)] = 3.4 \text{ bit} > H_P[X_i]$



(c) Can you do better than an optimal symbol code? Describe your approach first in words, then implement it in Python or in pseudo code. (about 4 lines of code for encoding and 4 lines of code for decoding; no library function calls necessary.)

→ interpret the sequence of symbols as a number in the decimal system and convert it to binary. See code.

(d) What is the expected bit rate per symbol of your method from part (c) in the limit of long messages?  $\lim_{k \rightarrow \infty} \frac{1}{k} \mathbb{E}_P[R_C(\mathbf{X})] =$

largest possible number made up of  $k$  decimals:  $\underbrace{999\dots 9}_{k \text{ times}} = 10^k - 1$

⇒ length of binary representation:

$$R_C(\underbrace{(9, 9, 9, \dots, 9)}_{k \text{ times}}) = \log_2(10^k - 1) + \epsilon \leq \log_2(10^k) + \epsilon = k \log_2 10 + \epsilon$$

for rounding:  $\epsilon < 1$

$$\Rightarrow \lim_{k \rightarrow \infty} \frac{1}{k} \mathbb{E}_P[R_C(\mathbf{X})] \leq \lim_{k \rightarrow \infty} \frac{k \log_2 10 + \epsilon}{k} = \log_2 10 = H_P[X_i]$$

## Observations from our implementation of positional numeral systems (see python code):

- encoding and decoding (or "parsing and generating") operates as a stack
- conversion from, e.g., decimal to binary system amortizes the bit rate over several symbols (i.e., each bit in the compressed representation may correspond to more than just a single symbol). This differentiates stream codes from symbol codes.
- positional numeral systems are an optimal compression method for sequences of symbols if the symbols satisfy the following three requirements:
  - (i) all symbols are from the same (finite) alphabet;
  - (ii) all symbols are uniformly distributed over this alphabet; and
  - (iii) all symbols are (therefore) statistically independent.

**Idea: generalize the concept of positional numeral systems by lifting all three of these limitations (i)-(iii).**

Limitation (i): positional numeral systems with a different base for each symbol

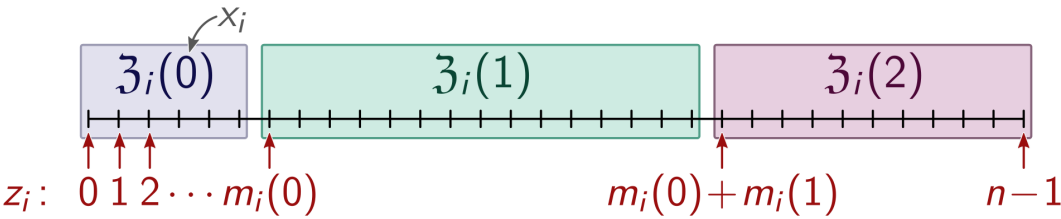
Limitation (ii): non-uniformly distributed symbols

Step 1: approximate the probabilistic model  $P$  with fixed-point precision.

Example:  $\mathcal{X}_i = \{0, 1, 2\}$ .

$x_i$	0	1	2
$P(x_i = x_i)$	0.2	0.45	0.35

Step 2: interpret  $Q$  as a latent variable model



Question: How would you draw a random sample  $x_i$  from the distribution  $Q(x_i)$ ?

Step 3: since the latents  $z_i$  uniquely identify the symbols  $x_i$ , we can encode the sequence of  $z_i$ 's instead of the sequence of  $x_i$ 's

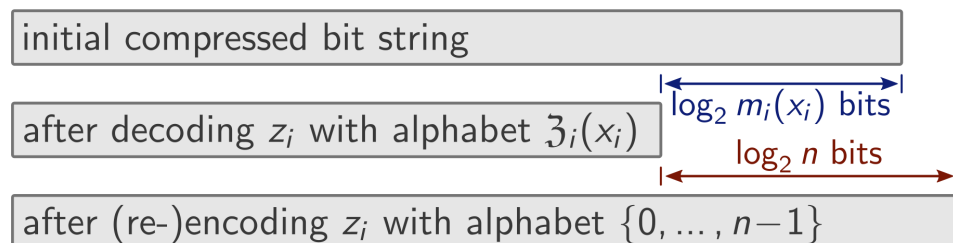
encoder  
message  $x = (x_1, x_2, \dots, x_k)$

decoder

Problem: - resulting bitrate per symbol:  
- compare to information content symbol  $i$  (under approximate model  $Q$ ):

#### Step 4: The Bits-Back Trick

Idea: rather than choosing  $z_i$  arbitrarily from  $\mathcal{Z}_i(x_i)$ , encode some "side information" into our choice of  $z_i$



#### Recap: what have we achieved so far?

Our goal is still to lift the three limitations of positional numeral systems:

- positional numeral systems are an optimal compression method for sequences of symbols if the symbols satisfy the following three requirements:
  - (i) all symbols are from the same (finite) alphabet;
  - (ii) all symbols are uniformly distributed over this alphabet; and
  - (iii) all symbols are (therefore) statistically independent.

#### Limitation (iii): modeling correlations between symbols

(a) use an autoregressive model (as we did in Problem 3.2 with Huffman Coding)

(b) use a latent variable model and generalize the Bits-Back trick

## Computational efficiency of the algorithm we have so far:

```
class SlowAnsCoder:
    def __init__(self, precision, compressed=0):
        self.n = 2**precision
        self.compressed = compressed

    def push(self, symbol, m):          # Encodes one symbol.
        z = self.compressed % m[symbol] + sum(m[0:symbol])
        self.compressed //= m[symbol]
        self.compressed = self.compressed * self.n + z

    def pop(self, m):                  # Decodes one symbol.
        z = self.compressed % self.n
        self.compressed //= self.n
        # Identify symbol and subtract sum(m[0:symbol]) from z:
        for symbol, m_symbol in enumerate(m):
            if z >= m_symbol:
                z -= m_symbol
            else:
                break # We found the symbol that satisfies  $z \in \sum_{j=0}^{symbol-1} m[j]$ .
        self.compressed = self.compressed * m_symbol + z
        return symbol

    def get_compressed(self):
        return self.compressed
```

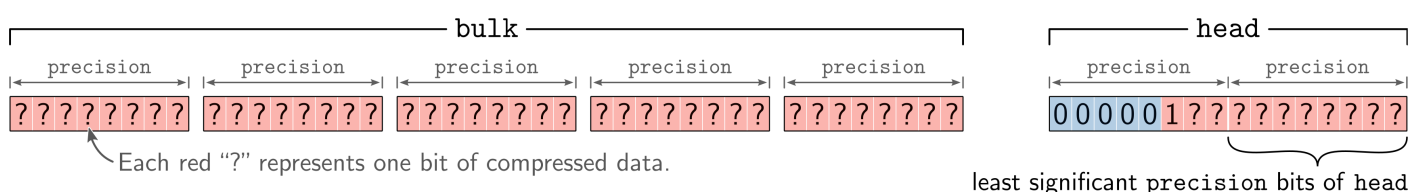
Note: these orange parts are just for demonstration purpose. In a production setup, you should do something more efficient here (e.g., a binary search or a lookup table); the "constriction" library that we'll use on the problem sets provides several efficient alternatives for these parts depending on the nature of your model.

## Improving Computational Efficiency: Streaming ANS

Consider the task of multiplying  $a*b$  where  $a$  is a very large number (similar for division and modulo):

⇒ Strategy: allow arithmetic operations like  $a*b$ ,  $a/b$ , or  $a \bmod b$  only if:

We'll represent the (so far) compressed data by a "bulk" and a "head" part



Streaming ANS algorithm:

- Most encoding/decoding operations involve only on the "head" ( $\Rightarrow$  fast since head size is bounded).
- Only if "head" overflows (during encoding) or underflows (during decoding) do we transfer some bits between "bulk" and "head". Here, we always transfer an integer number of bits, so this is also fast.
- The encoder and the decoder must agree on the exact point in time where they transfer data between "bulk" and "head". To ensure this, a common approach is to keep the number of valid bits on "head" always between precision and  $2 \times \text{precision}$ . More formally, we uphold the following two invariants:

A violation of invariant (i) triggers a data transfer from "head" to "bulk" that restores both invariants.  
A violation of invariant (ii) triggers a data transfer from "bulk" to "head" that restores both invariants.