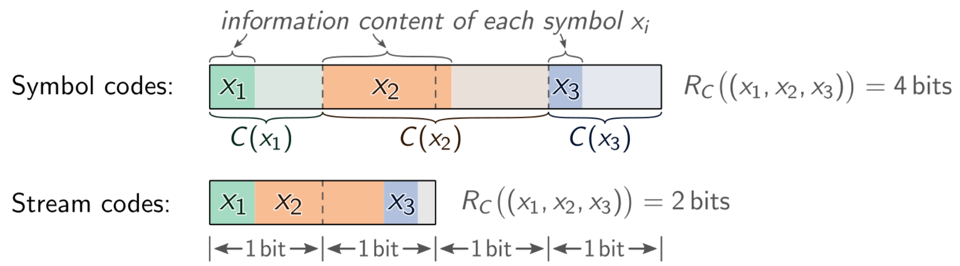
Stream Codes II: Asymmetric Numeral Systems (ANS)

Lecture 6 (2 June 2022); lecturer: Robert Bamler
more course materials online at https://robamler.github.io/teaching/compress22/

[Duda et al., 2015]

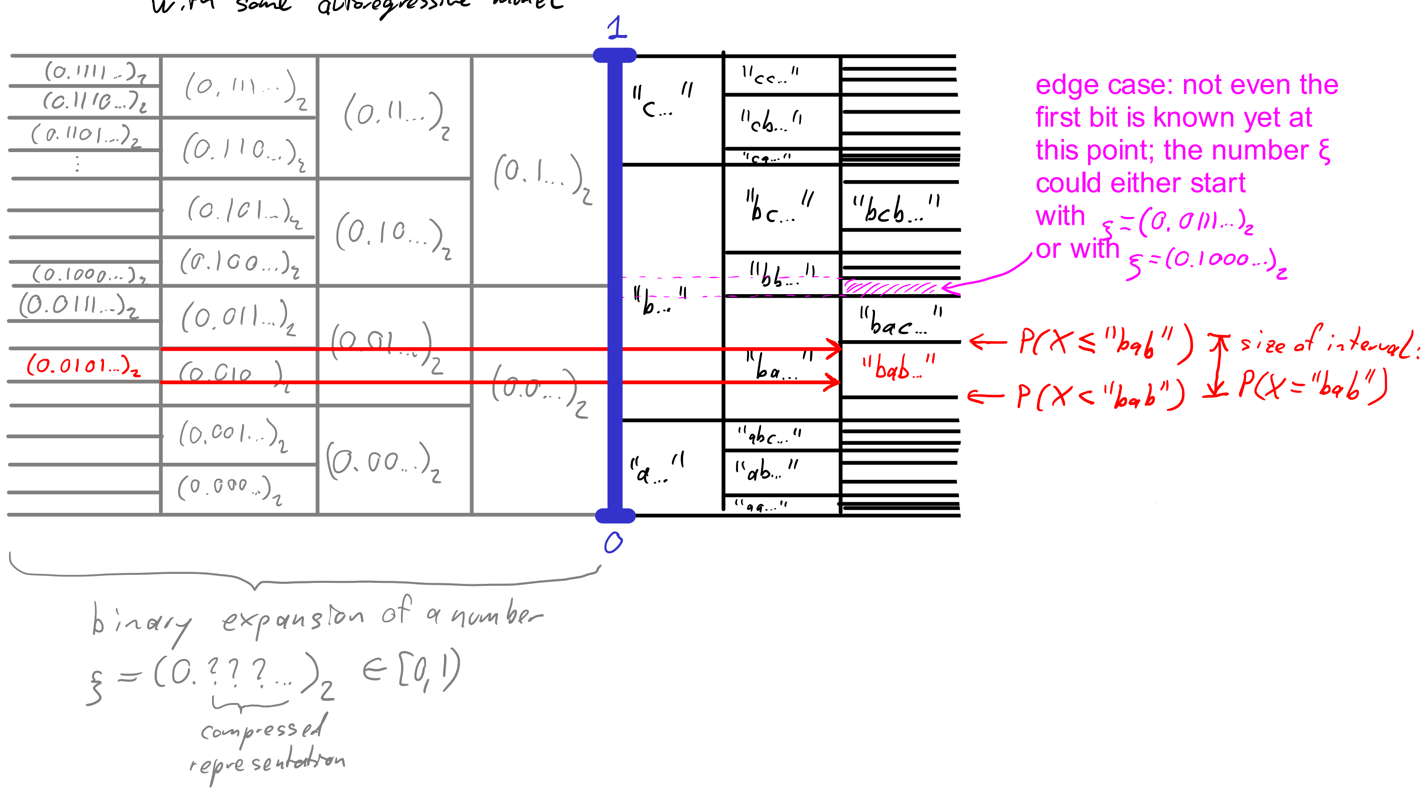
Recap from last lecture

- symbol codes vs. stream codes:



- two stream codes: arithmetic coding and range coding
(≈ 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 $x \in \mathfrak{X}^*$ with alphabet $\mathfrak{X} = \{\text{"a"}, \text{"b"}, \text{"c"}\}$ 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 very easy to implement.

Asymmetric Numeral Systems

[Duda et al., 2015]

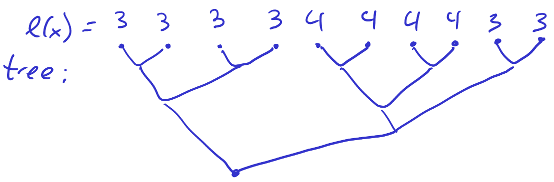
Exercise

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

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

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

Huffman tree:



(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? $\lim_{k\to\infty} \frac{1}{k} \mathbb{E}_P[R_C(\mathbf{X})] =$

largest possible number made up of k decimals: $\underbrace{99 9\ldots 9}_{k \text{ times}} = 10^k - 1$

$\Rightarrow$ length of binary representation:

for rounding; $\varepsilon < 1$

$$R_C((9,9,9,\ldots,9)) = \log_2(10^k - 1) + \varepsilon \le \log_2(10^k) + \varepsilon = k \log_2 10 + \varepsilon$$

$$\Rightarrow \lim_{k\to\infty} \frac{1}{k} \mathbb{E}_P[R_C(X)] \le \lim_{k\to\infty} \frac{k \log_2 10 + \varepsilon}{k} = \log_2 10 = H_P[X_i]$$

```python
def encode_uniform(message, base):
    compressed = 0
    for symbol in reversed(message):
        compressed = compressed * base + symbol
    return compressed

def decode_uniform(compressed, base, message_length):
    message = []
    for _ in range(message_length):
        message.append(compressed % base)
        compressed //= base
    return message
```

```python
compressed = encode_uniform([3, 5, 6], 10)
print(f'compressed: {compressed:b}')
reconstruction = decode_uniform(compressed, 10, 3)
print(f'reconstruction: {reconstruction}')
```

```
compressed: 1010001101
reconstruction: [3, 5, 6]
```

Observations from our implementation of positional numeral systems:

- encoding and decoding (or "parsing and generating") operates as a stack → i.e., "last in first out"

- conversion from, e.g., decimal to binary system amortizes the bit rate over several symbols (i.e., each bit in the compresse representation may correspond to more than just a single symbol). This differentiates stream codes from symbol codes.

```
print(f'[3, 5, 7] ==> {encode_uniform([3, 5, 7], 10):b}')
print(f'[3, 5, 6] ==> {encode_uniform([3, 5, 6], 10):b}')
print(f'[3, 4, 6] ==> {encode_uniform([3, 4, 6], 10):b}')
```

```
[3, 5, 7] ==> 1011110001
[3, 5, 6] ==> 1010001101
[3, 4, 6] ==> 1010000011
```

change only last symbol

change only second symbol

red underlined bits change in both cases (this is unlike 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 statistically independent.

→ i.e. amortized bit rate per symbol is $\log_2 |\mathcal{X}|$ (= entropy of a uniformly distributed random variable)

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

→ Observation: it just works (see code)

```
class UniformCoder:
    def __init__(self, compressed=0):
        self.compressed = compressed

    def push(self, symbol, base): # encodes one symbol
        self.compressed = self.compressed * base + symbol

    def pop(self, base):          # decodes one symbol
        symbol = self.compressed % base
        self.compressed //= base
        return symbol
```

```
coder = UniformCoder()

coder.push( 3, base=10) # uses alphabet {0, 1, ..., 9}
coder.push( 6, base=10) # uses alphabet {0, 1, ..., 9}
coder.push(12, base=15) # uses alphabet {0, 1, ..., 14}
coder.push( 5, base=15) # uses alphabet {0, 1, ..., 14}
print(f'compressed: {coder.compressed:b}')

print(f"decoded: {coder.pop(base=15)}")
print(f"decoded: {coder.pop(base=15)}")
print(f"decoded: {coder.pop(base=10)}")
print(f"decoded: {coder.pop(base=10)}")
```

```
compressed: 10000001011101
decoded: 5
decoded: 12
decoded: 6
decoded: 3
```

Limitation (ii): non-uniformly distributed symbols

- consider a single symbol $x_i \in \mathfrak{X}_i$ ← some finite alphabet
- assume some (fixed) probabilistic model $P(X_i)$

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

Probabilistic model $Q(X_i)$ with

$$Q(X_i = x_i) = \frac{m(x_i)}{n} \qquad \forall x_i \in \mathfrak{X}_i \qquad (*)$$

where $m(x_i) \in \mathbb{N}$; $\sum_{x_i \in \mathfrak{X}_i} m(x_i) = n$ ($\Rightarrow Q$ is properly normalized); $n = 2^{\text{precision}}$

choose $m(x_i)\ \forall x_i \in \mathfrak{X}_i$ so that $Q(X_i)$ approximates $P(X_i)$: minimize $D_{KL}(\underbrace{P(X_i)}_{\text{fixed}} \| Q(X_i))$

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

Assume 5 bit precision (toy example), i.e., $n = 2^5 = 32$

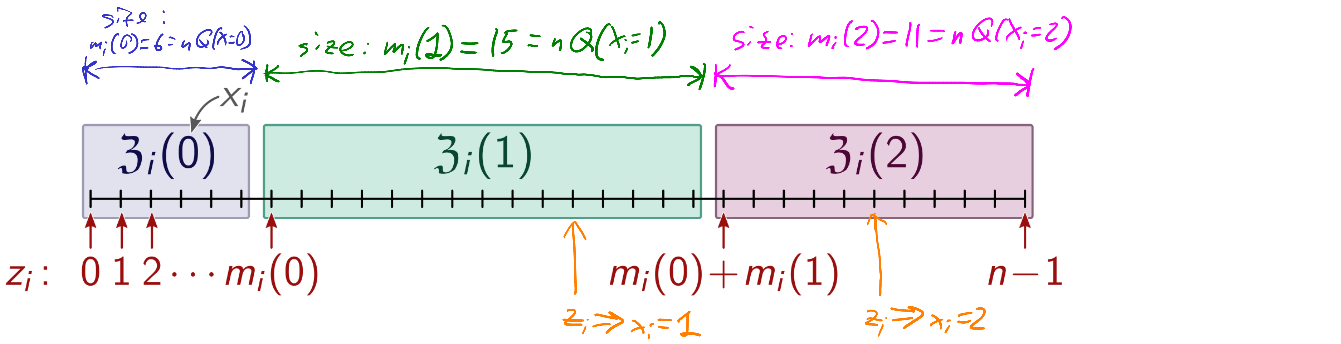
x_i	0	1	2
$P(X_i = x_i)$	0.2	0.45	0.35
$Q(X_i = x_i)$ for optimal approximation	$\frac{6}{32} = 0.1875$	$\frac{15}{32} = 0.46875$	$\frac{11}{32} = 0.34375$

$m_i(2) = 11$

(approximation improves for increasing precision)

Step 2: interpret Q as a latent variable model

→ Observation: since $\sum_{x_i \in \mathfrak{X}_i} m_i(x_i) = n$, the $m_i(x_i)$ values define a partitioning of the range $\{0, 1, \ldots, n-1\}$ into pairwise disjoint subranges $\mathfrak{Z}_i \subset \{0, \ldots, n-1\}$



Def. subrange: $\mathfrak{Z}_i(x_i) := \left\{ \sum_{x_i' < x_i} m(x_i'), \ldots, \left(\sum_{x_i' \leq x_i} m(x_i') \right) - 1 \right\}$

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

→ idea: • draw a uniformly distributed random number $z_i \in \{0, \dots, n-1\}$

• then identify the unique $x_i \in \mathfrak{X}_i$ s.t. $z \in \mathcal{Z}_i(x_i)$

→ joint probability of z_i & x_i: $Q(Z_i, X_i) = Q(Z_i)\, Q(X_i \mid Z_i)$

with • $Q(Z_i = z_i) = \frac{1}{n} \quad \forall z_i \in \{0, \dots, n-1\}$

• $Q(X_i = x_i \mid Z_i = z_i) = \begin{cases} 1 & \text{if } z_i \in \mathcal{Z}_i(x_i) \\ 0 & \text{otherwise} \end{cases}$

→ thus, the marginal distribution of X_i is:

$$Q(X_i = x_i) = \sum_{z_i=0}^{n-1} Q(Z_i = z_i, X_i = x_i) = \frac{m(x_i)}{n}$$

→ recovers (*)

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

$\boxed{\text{encoder}}$

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

$\downarrow \quad \downarrow \quad \downarrow$

$z_1 \quad z_2 \quad z_k$

$\forall i$: pick some (arbitrary) $z_i \in \mathcal{Z}_i(x_i)$

encode this with our Uniform Coder, using alphabet $\{0, \dots, n-1\}$

$\boxed{\text{decoder}}$

decode with Uniform Coder: $z_1, z_2, \dots, z_k$

then identify $\forall i$ the unique x_i s.t. $z_i \in \mathcal{Z}(x_i)$

Problem: - resulting bitrate per symbol: $\log_2 n$

- compare to information content symbol i (under approximate model Q):

$$-\log_2 Q(X_i = x_i) = -\log_2 \frac{m(x_i)}{n} = \log_2 n - \log_2 m(x_i)$$

$$\Rightarrow \text{overhead: } (\log_2 n) - (\log_2 n - \log_2 m(x_i)) = \log_2 m(x_i) = \underbrace{\log_2 |\mathcal{Z}_i(x_i)|}$$

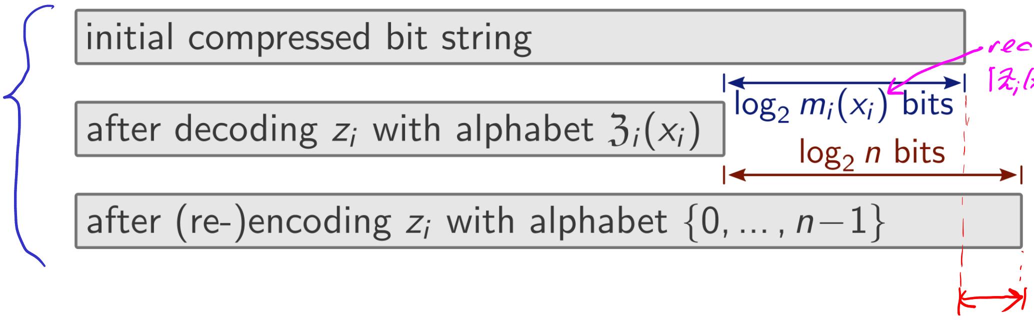
information that's "hidden" in our arbitrary choice of $z_i \in \mathcal{Z}_i(x_i)$

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

consume some part of the already compressed data by decoding from it with our Uniform Coder & alphabet $\mathcal{Z}_i(x_i)$

encoding a single symbol x_i:



recall $|\mathcal{Z}_i(x_i)| = m_i(x_i)$

$\log_2 n - \log_2 m_i(x_i) =$

$= -\log_2 \frac{m_i(x_i)}{n} =$

$= -\log_2 Q(X_i = x_i)$

→ coder is optimal (w.r.t. Q)

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

    def push(self, symbol, base): # encodes one symbol
        z = (self.uniform_coder.pop(base=m[symbol])
             + sum(m[:symbol]))
        self.uniform_coder.push(z, base=self.n)

    def pop(self, base):          # decodes one symbol
        z = self.uniform_coder.pop(base=self.n)
        for symbol, m_symbol in enumerate(m):
            if z >= m_symbol:
                z -= m_symbol
            else:
                break # found the symbol
        self.uniform_coder.push(z, base=m[symbol])
        return symbol

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

```python
precision = 4      # thus, n = 2^4 = 16
m1 = [7, 3, 6]     # implies alphabet {0, 1, 2}
m2 = [4, 2, 3, 7] # implies alphabet {0, 1, 2, 3}

encoder = SlowAnsCoder(precision)
encoder.push(2, m1)
encoder.push(1, m2)
encoder.push(0, m1)
encoder.push(2, m1)

compressed = encoder.get_compressed()
print(f'compressed: {compressed:b}')

decoder = SlowAnsCoder(precision, compressed)
print(f'decoded: {decoder.pop(m1)}')
print(f'decoded: {decoder.pop(m1)}')
print(f'decoded: {decoder.pop(m2)}')
print(f'decoded: {decoder.pop(m1)}')
```

```
compressed: 101011100
decoded: 2
decoded: 0
decoded: 1
decoded: 2
```

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)

↳ works in principle but the "stack" semantics make it difficult in practice

↳ for autoregressive models, use range coding instead (Problem 6.3)

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

↳ next lecture (tomorrow!) & Problem Set 7

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 ∈ ℨ_i(symbol).
        self.compressed = self.compressed * m_symbol + z
        return symbol

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

Those arithmetic operations are expensive because self.compressed will eventually be a very large number with $O(k)$ bits
↑ message length

⇒ $O(k)$ runtime per symbol

⇒ $O(k^2)$ runtime for the full message

☹

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):

↳ general case: expensive

↳ exception: if b is a power of 2, e.g., $b = n = 2^{\text{precision}}$ then we just need to append "precision" zeros

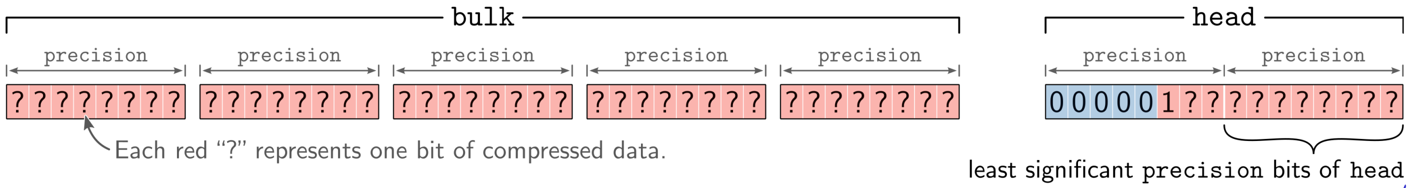
→ can be done in $O(1)$ (amortized) time if we store "a" in a dynamic array (a.k.a. "vector")

⇒ Strategy: allow arithmetic operations like a*b, a/b, or a mod b only if:

↳ either b is a power of 2

↳ or if a (and b) is small

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



- growable as needed (dynamic array)
- size is always a multiple of "precision"

- bounded size (e.g., max 64 bit)
- Contains most recently compressed data (the top of the stack)

Streaming ANS algorithm:

- Most encoding/decoding operations involve only on the "head" (⇒ 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*precision. More formally, we uphold the following two invariants:

  (i) $\text{head} < 2^{2\times\text{precision}}$ (always)

  (ii) $\text{head} \geq 2^{\text{precision}}$ unless bulk is empty

  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.

Complete implementation of streaming ANS in python:
(usage example on next page)

```python
class AnsCoder:
    def __init__(self, precision, compressed=[]):
        self.precision = precision
        self.mask = (1 << precision) - 1 # (a string of precision one-bits)
        self.bulk = compressed.copy() # (We will mutate bulk below.)
        self.head = 0

        # Establish invariant (ii):
        while len(self.bulk) != 0 and (self.head >> precision) == 0:
            self.head = (self.head << precision) | self.bulk.pop()

    def push(self, symbol, m):
        # Check if encoding directly onto head would violate invariant (i):
        if (self.head >> self.precision) >= m[symbol]:
            # Transfer one word of compressed data from head to bulk:
            self.bulk.append(self.head & self.mask)
            self.head >>= self.precision
            # At this point, invariant (ii) is definitely violated,
            # but the operations below will restore it.

        z = self.head % m[symbol] + sum(m[0:symbol])
        self.head //= m[symbol]
        self.head = (self.head << self.precision) | z # (This is
        # equivalent to " self.head * n + z", just slightly faster.)

    def pop(self, m):
        z = self.head & self.mask
        self.head >>= self.precision
        for symbol, m_symbol in enumerate(m):
            if z >= m_symbol:
                z -= m_symbol
            else:
                break
        self.head = self.head * m_symbol + z

        # Restore invariant (ii) if it is violated:
        if (self.head >> self.precision) == 0 and len(self.bulk) != 0:
            # Transfer data back from bulk to head (" |" is bitwise or):
            self.head = (self.head << self.precision) | self.bulk.pop()
        return symbol

    def get_compressed(self):
        compressed = self.bulk.copy() # (We will mutate compressed below.)
        head = self.head
        # Chop head into precision-sized words and append to compressed:
        while head != 0:
            compressed.append(head & self.mask)
            head >>= self.precision
        return compressed
```

Usage example:

```
precision = 4
m1 = [7, 3, 6]
m2 = [4, 2, 3, 7]

encoder = AnsCoder(precision)
encoder.push(1, m1)
encoder.push(1, m2)
encoder.push(0, m1)
encoder.push(2, m1)

compressed = encoder.get_compressed()
print(f'compressed: {[bin(word) for word in compressed]}')

decoder = AnsCoder(precision, compressed)
print(f"decoded: {decoder.pop(m1)}")
print(f"decoded: {decoder.pop(m1)}")
print(f"decoded: {decoder.pop(m2)}")
print(f"decoded: {decoder.pop(m1)}")
```

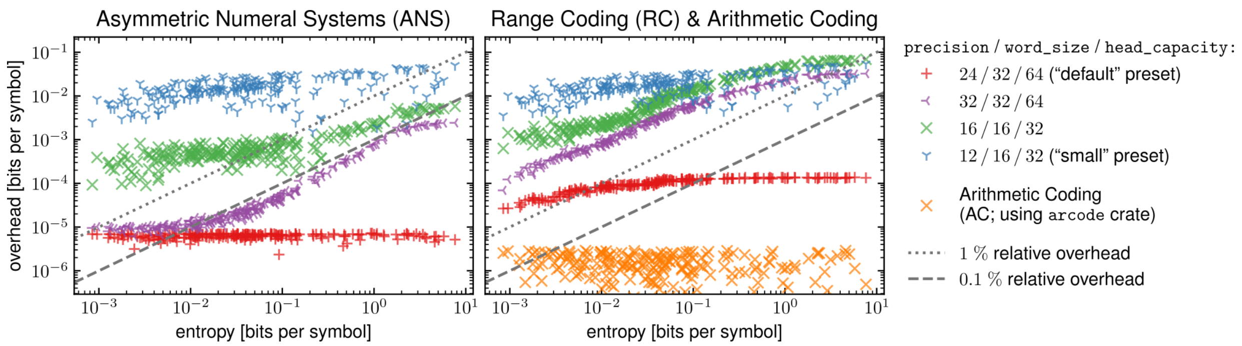
```
compressed: ['0b100', '0b1011', '0b1']
decoded: 2
decoded: 0
decoded: 1
decoded: 1
```

Empirical Performance And Efficiency

[plots taken from Bamler, 2022 (arXiv:2201.01741)]

- comparison of ANS, Range Coding, And Arithmetic Coding
- results for ANS and Range Coding were obtained with a library called "constriction", with which you'll experiment in Problem Sets 6 and 7

Compression Performance:



Runtime:

(Take these results with a grain of salt because the runtime depends on implementation details.)