Solutions to Problem Set 6

discussed: 7 June 2023

Data Compression With And Without Deep Probabilistic Models
Prof. Robert Bamler, University of Tübingen

Course materials available at https://robamler.github.io/teaching/compress23/

Note: This problem set focuses entirely on the Asymmetric Numeral Systems (ANS) algorithm as there seemed to be quite a bit of confusion about this algorithm in the lecture. I decided to defer the promised exercise that uses range coding with our autoregressive machine-learning model of English text to a later problem set.

Problem 6.1: Positional Numeral Systems

In the lecture, we introduced the Asymmetric Numeral Systems entropy coder as a generalization of positional numeral systems. Listing 1 on the next page shows our implementation of a class `UniformCoder`, which is an optimal entropy coder for a sequence of independent and uniformly distributed symbols, using a positional numeral system with a position-dependent `base`. The listing also contains a simple usage example.

(a) **Correctness:** prove that `UniformCoder` is indeed a correct entropy coder, i.e., that decoding (i.e., `pop`) is the inverse of encoding (i.e., `push`):

(i) convince yourself that calling `coder.push(symbol, base)` and then calling `coder.pop(base)` returns `symbol` and restores the original state of `coder`, regardless of its original state (assuming that all method arguments are valid, i.e., `base` and `symbol` are integers with $0 \leq$ `symbol` $<$ `base`);

Solution: We concatenate the two method bodies and then rewrite the resulting code in static single assignment (SSA) form by introducing subscript indices to all variables that are not method parameters:

```
1 # coder.push(symbol, base):
2 self.compressed₂ = self.compressed * base + symbol
3
4 # coder.pop(base):
5 symbol₂ = self.compressed₂ % base
6 self.compressed₃ //= base    # ("//" denotes integer division.)
7 return symbol₂
```

Since `self.compressed` is a nonnegative integer, and $0 \leq$ `symbol` $<$ `base` by assumption, dividing `self.compressed`$_2$ (defined in Line 2) by `base` has integer part `self.compressed` and remainder `symbol`. Lines 5-6 extract these two parts, thus setting `symbol`$_2$ = `symbol` (which gets returned on Line 7), `self.compressed`$_3$ = `self.compressed` (restoring the original state). ■

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

    def push(self, symbol, base):  # Encodes a symbol ∈ {0, ..., base − 1}.
        self.compressed = self.compressed * base + symbol

    def pop(self, base):           # Decodes a symbol ∈ {0, ..., base − 1}.
        symbol = self.compressed % base
        self.compressed //= base   # ("//" denotes integer division.)
        return symbol

# Usage example:
coder = UniformCoder()

coder.push(6, base=10)
coder.push(13, base=16)
coder.push(7, base=8)
print(bin(coder.compressed))    # Prints: "0b1101101111"

print(coder.pop(base=8))  # Prints: "7"
print(coder.pop(base=16)) # Prints: "13"
print(coder.pop(base=10)) # Prints: "6"
```

Listing 1: A “stack”-like entropy coder that is optimal for uniformly distributed symbols.

(ii) convince yourself that setting `symbol = coder.pop(base)` with any positive integer `base` and then calling `coder.push(symbol, base)` restores the original state of `coder`, regardless of its original state (even if the coder was originally empty).

Solution: We again concatenate the two method bodies and then rewrite the resulting code in SSA form:

```
1  # symbol = coder.pop(base):
2  symbol₁ = self.compressed % base
3  self.compressed₂ //= base    # ("//" denotes integer division.)
4
5  # coder.push(symbol₁, base):
6  self.compressed₃ = self.compressed₂ * base + symbol₁
```

Here, it's even easier to see that Line 6 reverts Lines 2-3, i.e., the final value $\texttt{self.compressed}_3$ is equal to the original value `self.compressed`. Note that this even holds if the coder was originally empty, i.e., if $\texttt{self.compressed} = 0$.

■

(b) **Compression performance:** assume you use the `UniformCoder` to encode a sequence of `symbols` $x_1, x_2, \ldots, x_k$ with respective `bases` (i.e., alphabet sizes) $|\mathfrak{X}_1|, |\mathfrak{X}_2|, \ldots, |\mathfrak{X}_k|$. Assuming the alphabet sizes are fixed, which sequence of symbols leads to the largest possible value of `coder.compressed` after encoding? What is this largest possible value and how long is its binary representation? Compare to the information content of the message assuming the symbols are statistically independent, and each one is uniformly distributed over the respective alphabet $\mathfrak{X}_i = \{0, 1, \ldots, |\mathfrak{X}_i| - 1\}$ for each symbol x_i.

Solution: We obtain the largets possible final value of `coder.compressed` by encoding the largest possible symbols values, i.e., $x_i = |\mathfrak{X}_i| - 1 \ \forall i \in \{1, \ldots, k\}$. Analogous to the decimal system where the largest k-digit number (written as a sequence of k digits "9") is $999\ldots9 = 10^k - 1$, the largest possible final value of the coder state is $\texttt{coder.compressed} = \left(\prod_{i=1}^k |\mathfrak{X}_i|\right) - 1$, i.e., the product of all `bases` minus one. Its binary representation is thus $\left\lceil \log_2 \left(\prod_{i=1}^k |\mathfrak{X}_i|\right)\right\rceil = \left\lceil \sum_{i=1}^k \log_2 |\mathfrak{X}_i| \right\rceil$ bits long (see Problem 2.4 (b)). By comparison, the information content of each symbol x_i is $-\log_2 P(X_i = x_i) = \log_2 |\mathfrak{X}_i|$ since we assumed a uniform distribution $P(X_i = x_i) = 1/|\mathfrak{X}_i| \ \forall x_i \in \mathfrak{X}_i$. And since we assumed that the symbols are statistically independent, the information content of the message is the sum of the information contents of the symbols, $\sum_{i=1}^k \log_2 |\mathfrak{X}_i|$. Thus, `UniformCoder` is asymptotically optimal since its bit rate has less than one bit of overhead over the information content, regardless of the length of the message.

Note: Technically, `UniformCoder` does not implement a uniquely decodable code, i.e., if a sender were to encode two messages with two separate `UniformCoders`

named `coder1` and `coder2` and then concatenate the binary representations of `coder1.compressed` and `coder2.compressed`, then a receiver of this concatenated bit string would in general not be able to identify where the concatenation happend. In practice, however, it is quite unusual to concatenate compressed representations of *entire messages* (as opposed to concatenating compressed representations of *single symbols*, as one does in symbol codes). But for completeness, we note (leaving the proof to the reader) that the `StreamingAnsCoder` in Listing 3 (discussed in Problem 6.3 below) can be made uniquely decodable by initializing it with `StreamingAnsCoder(compressed = [0, 1])`. ■

Problem 6.2: Naive Asymmetric Numeral Systems

Listing 2 shows the `SlowAnsCoder` from the lecture, which is a naive version of the Asymmetric Numeral Systems (ANS) algorithm (we will improve it in Problem 6.3).

(a) **Correctness:** similar to Problem 6.1 (a), prove that `SlowAnsCoder` is a correct entropy coder, i.e., that decoding (i.e., `pop`) is the inverse of encoding (i.e., `push`):

(i) convince yourself that calling `coder.push(symbol, m)` followed by calling `coder.pop(m)` returns `symbol` and restores the original state of `coder`, regardless of its original state (assuming that all method arguments are valid, i.e., `m` is a list of nonnegative integers that sum to $2^{\texttt{precision}}$ and $0 \leq \texttt{symbol} < \texttt{len(m)}$; also, $\texttt{m[symbol]} \neq 0$; why is the last condition necessary?);

Solution: We again concatenate the two method bodies, and we rename the local variable `symbol` in `pop` to `decoded_symbol` to resolve name clashes (we don't use full SSA-form here since this would be somewhat complicated for the `for`-loop, requiring a so-called ϕ-expression):

```
# coder.push(symbol, base):
z = self.uniform_coder.pop(base=m[symbol]) + sum(m[0:symbol])
self.uniform_coder.push(z, base=self.n)

# coder.pop(base):
z = self.uniform_coder.pop(base=self.n)
# Find the unique symbol that satisfies z ∈ 𝔷_i(symbol)
# (using linear search just to simplify exposition):
for decoded_symbol, m_symbol in enumerate(m):
    if z >= m_symbol:
        z -= m_symbol
    else:
        break
self.uniform_coder.push(z, base=m_symbol)
return decoded_symbol
```

Here, Lines 3 and 6 are inverses of each other as we've shown in Problem 6.1 (a), so we can simplify:

```
1 z = self.uniform_coder.pop(base=m[symbol]) + sum(m[0:symbol])
2 for decoded_symbol, m_symbol in enumerate(m):
3     if z >= m_symbol:
4         z -= m_symbol
5     else:
6         break
7 self.uniform_coder.push(z, base=m_symbol)
8 return decoded_symbol
```

The `for`-loop on Lines 2-6 iterates over `decoded_symbol` from 0 to (at most) `len(m)` $-\,1$, setting `m_symbol` = `m[decoded_symbol]` in each iteration. Each iteration reduces `z` (which was initialized on Line 1) by `m_symbol` as long as the result would not be negative. Thus, the first `symbol` iterations of the loop reduce `z` by $\sum_{i=0}^{\texttt{symbol}-1}$ `m[i]` = `sum(m[0:symbol])`, which reverts the part that reads "`+ sum(m[0:symbol])`" on Line 1. Therefore, after the first `symbol` iterations of the loop, we have `decoded_symbol` = `symbol` $-\,1$ and `z` = `self.uniform_coder.pop(base=m[symbol])`, which is less than `m[symbol]` because the method `pop` on `UniformCoder` always returns an integer in the range from 0 to `base` $-\,1$. The next iteration of the `for`-loop increments `decoded_symbol` to `symbol` and then `break`s because, at this point, `z` $<$ `m_symbol` = `m[decoded_symbol]` = `m[symbol]`. In summary, the `for`-loop is equivalent to Lines 4-6 in the following simplified code:

```
1 z = self.uniform_coder.pop(base=m[symbol]) + sum(m[0:symbol])
2
3 # Effect of the for-loop:
4 z -= sum(m[0:symbol])
5 decoded_symbol = symbol
6 m_symbol = m[decoded_symbol]
7
8 self.uniform_coder.push(z, base=m_symbol)
9 return decoded_symbol
```

Using the fact that the methods `push` and `pop` on `UniformCoder` coder are inverses of each other (see Problem 6.1 (a)), it is now easy to see that Lines 4 and 8 revert Line 1, and that the code `return`s the original `symbol`. ■

(ii) convince yourself that setting `symbol = coder.pop(m)` (where `m` is again a list of nonnegative integers that sum to $2^{\texttt{precision}}$), followed by calling `coder.push(symbol, m)` restores the original state of `coder`, regardless of its original state (even if the coder was originally empty).

Solution: Concatenating the two method bodies, we obtain:

```
1  # symbol = coder.pop(base):
2  z = self.uniform_coder.pop(base=self.n)
3  for symbol, m_symbol in enumerate(m):
4      if z >= m_symbol:
5          z -= m_symbol
6      else:
7          break
8  self.uniform_coder.push(z, base=m_symbol)
9
10 # coder.push(symbol, base):
11 z = self.uniform_coder.pop(base=m[symbol]) + sum(m[0:symbol])
12 self.uniform_coder.push(z, base=self.n)
```

Using the correctness of `UniformCoder`, we can simplify Lines 8-11:

```
1  z = self.uniform_coder.pop(base=self.n)
2  for symbol, m_symbol in enumerate(m):
3      if z >= m_symbol:
4          z -= m_symbol
5      else:
6          break
7  z += sum(m[0:symbol]) # equivalent to Lines 8-11 in last code
8  self.uniform_coder.push(z, base=self.n)
```

We then note that, no matter how many iterations the `for`-loop performs, its effect is to reduce `z` by `sum(m[0:symbol])`, where `symbol` denotes the number of performed iterations, i.e., the value that the variable `symbol` has after the `for`-loop (i.e., on Line 8). Thus, Line 8 reverts the `for`-loop, and we can simplify:

```
1  z = self.uniform_coder.pop(base=self.n)
2  self.uniform_coder.push(z, base=self.n)
```

This simplifies to a no-op by correctness of `UniformCoder`. Note that this is, again, even true if `self.uniform_coder` is initially empty (in which case Line 1 above leaves `self.uniform_coder` unaltered and sets `z = 0`, and Line 2 does nothing). ■

You may build your arguments on the fact that `coder.uniform_coder` is a correct entropy coder, which you showed in Problem 6.1 (a).

(b) **Compression performance:** in Problem 6.1 (b), you showed that encoding (i.e., `push`ing) a symbol with a `UniformCoder` and some given `base` contributes

$\log_2$(`base`) bits to the amortized bit rate. Since `pop` inverts `push`, it therefore reduces the amortized bit rate by the same amount. Using these results, by how much does calling the method `push` on a `SlowAnsCoder` increase the amortized bit rate? Compare to the information content of the symbol under the approximate probability distribution $Q(X_i{=}x_i) = \frac{m_i(x_i)}{n}$ where $m_i(x_i)$ is `m[symbol]` in python.

Solution: The method `push` in Listing 2 performs a `pop` call on the `UniformCoder` `self.uniform_coder` with argument `base = m[symbol]`, and a `push` call with argument `base = self.n`. As shown in Problem 6.1 (b), the `pop` call reduces the (amortized) bit rate of the `UniformCoder` by $\log_2$(`m[symbol]`) bits,[1] and the `push` call increases it by $\log_2$(`self.n`) = `precision` bits. Translating the python notation to the mathematical notation used in the lecture, we find that the combined amortized bit rate of both internal method calls is

$$\begin{aligned} -\log_2\big(\texttt{m[symbol]}\big) + \log_2\big(\texttt{self.n}\big) &= -\log_2 m_i(x_i) + \log_2 n \\ &= -\log_2 \frac{m_i(x_i)}{n} = -\log_2 Q(X_i{=}x_i). \end{aligned}$$

Thus, apart from a (usually tiny) approximation overhead $D_{\mathrm{KL}}(P\,||\,Q)$ (and a constant overhead of at most `precision` bits per message, see Footnote 1), the `SlowAnsCoder` has asymptotically optimal compression performance. ■

Problem 6.3: Streaming Asymmetric Numeral Systems

The `SlowAnsCoder` that we implemented in the lecture and discussed in Problem 6.2 is a correct entropy coder with a very close to optimal bit rate. But it has a problem: it is slow. More precisely, the runtime for encoding a single symbol grows linearly with the information content that has already been encoded onto the `SlowAnsCoder` before.

[1]except for the first symbol in a message, where `self.uniform_coder` is still empty, and thus `pop`ping from it does not actually remove any bits; this leads to a constant overhead of `SlowAnsCoder` of at most `precision` bits per message, which is negligible for anything but very short messages.

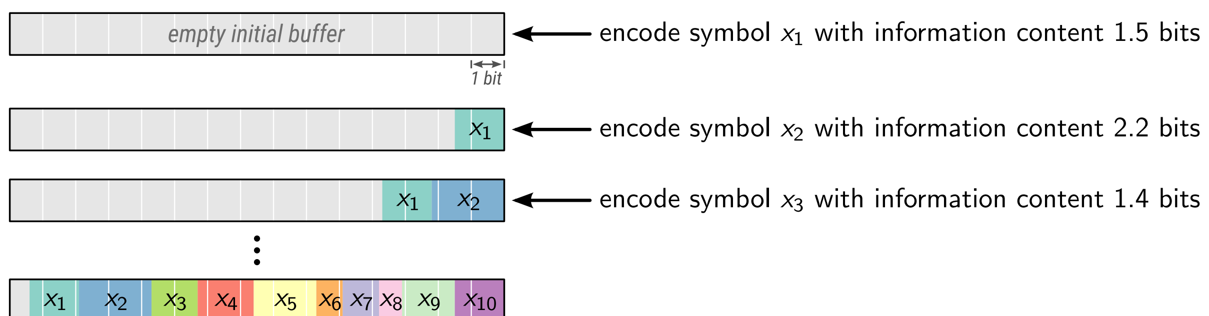


Figure 1: Stream coding with the naive `SlowAnsCoder` from Listing 2.

```python
class SlowAnsCoder:
    def __init__(self, precision, compressed=0):
        self.n = 2**precision    # ("**" denotes exponentiation.)
        self.uniform_coder = UniformCoder(compressed)  # See Listing 1.

    def push(self, symbol, m):   # Encodes one symbol.
        z = self.uniform_coder.pop(base=m[symbol]) + sum(m[0:symbol])
        self.uniform_coder.push(z, base=self.n)

    def pop(self, m):            # Decodes one symbol.
        z = self.uniform_coder.pop(base=self.n)
        # Find the unique symbol that satisfies z ∈ ℨ_i(symbol)
        # (using linear search just to simplify exposition):
        for symbol, m_symbol in enumerate(m):
            if z >= m_symbol:
                z -= m_symbol
            else:
                break
        self.uniform_coder.push(z, base=m_symbol)
        return symbol

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

# Usage example:
precision = 4 # (for demonstration purpose only)
m = [7, 6, 3] # (adds up to 16 = 2^precision, as required)
coder = SlowAnsCoder(precision)

coder.push(0, m)
coder.push(2, m)
coder.push(1, m)
print(bin(coder.get_compressed())) # Prints: "0b101000"

print(coder.pop(m)) # Prints: 1
print(coder.pop(m)) # Prints: 2
print(coder.pop(m)) # Prints: 0
```

Listing 2: Our naive (slow) implementation of Asymmetric Numeral Systems (ANS) from the lecture.

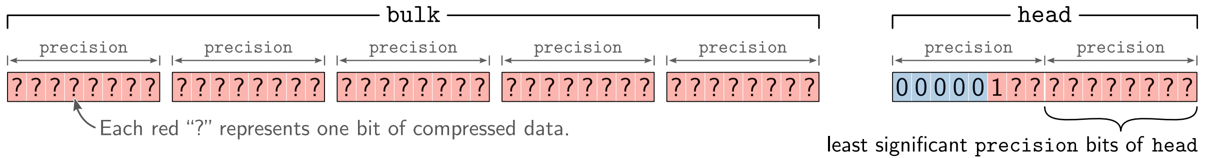


Figure 2: Representation of compressed data in streaming ANS with `precision` $= 8$.

Therefore, the runtime for encoding a sequence of k symbols scales quadratically in k, which makes this naive algorithm impractical for most common use cases. Let's fix this.

(a) Let's first understand exactly why the `SlowAnsCoder` from Listing 2 is slow. Figure 1 on page 7 illustrates how compressed data accumulates as we encode a sequence of 10 symbols $x_1, x_2, \ldots, x_{10}$ with a `SlowAnsCoder`. Different to, e.g., arithmetic coding or range coding, each new symbol that we encode conceptually pushes any previously encoded data to the left (i.e., towards more significant bits of `coder.uniform_coder.compressed`). This operation becomes increasingly expensive as the amount of previously encoded data grows. Identify the lines of code in Listing 2 (and, more precisely, inside method calls that jump to Listing 1) that implement this increasingly expensive operation.

Solution: The increasingly expensive operations are the method calls that delegate to `self.uniform_coder.pop` and `self.uniform_coder.push` on Lines 7, 8, 11, and 19 in Listing 2. More precisely, the culprits are Lines 6, 9, and 10 in the implementation of `UniformCoder` in Listing 1. Each of these lines performs a multiplication or division involving `self.compressed`. If you recall the algorithm for long multiplication and division from grade school, you will remember that its run time is bilinear in the length of both arguments.[2] ■

In order to speed up ANS and to make its runtime complexity linear in the length of the message, one uses the following trick (called "streaming ANS"): instead of representing the entire compressed data as a single (giant) integer, one splits the compressed data into a `bulk` part that holds most of the data in a dynamically sized array (aka "vector") and a `head` part with a fixed (and small) capacity. Figure 2 illustrates the simplest variant of this approach where each item of the vector `bulk` holds `precision` bits (referred to as "one word" of compressed data in the following), and `head` has a capacity of 2 words, i.e., $2 \times$ `precision` bits (thus, `head` can represent only integers from zero to $2^{2\times\texttt{precision}} - 1$).

Figure 3 illustrates what happens when we encode a sequence of symbols using such a streaming ANS coder. We use `precision` $= 4$ here for demonstration purpose.[3] While the first 4 symbols fit into `head` in the example illustrated in Figure 3, encoding the fifth symbol x_5 would lead to an overflow, i.e., it would lead to `head` $\geq 2^{2\times\texttt{precision}}$, which

[2] Strictly speaking, there exist more efficient algorithms than long multiplication, but these wouldn't be practical here. See, e.g.: Harvey and Van Der Hoeven (2021). Integer multiplication in time $O(n \log n)$. Annals of Mathematics, 193(2), 563-617.

[3] In production, one would set `precision` to about 16 or 32, so that `head` just fits into a CPU register.

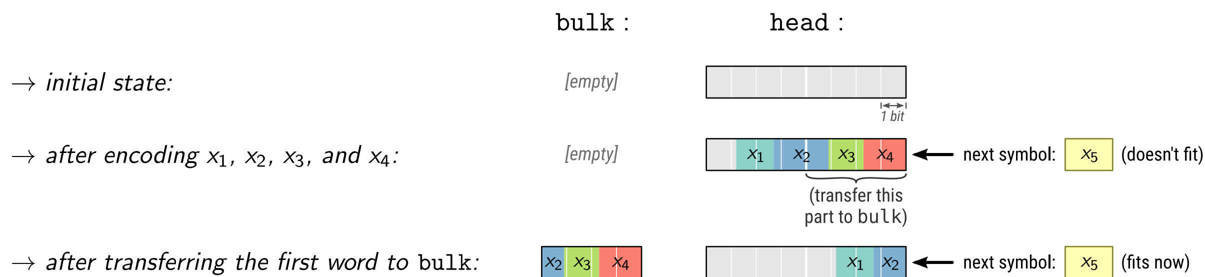


Figure 3: Stream coding with a `StreamingAnsCoder` (Listing 3) with `precision` = 4.

is not allowed. Thus, before we encode x_5, we transfer the least significant `precision` bits of `head` to `bulk`, and we move the remaining part of `head` by `precision` bits to the right. Assuming a good vector implementation, transferring *an integer number of bits* from `head` to `bulk` requires only constant (amortized) runtime because even if there was already some data on `bulk`, we wouldn't need to move it around.

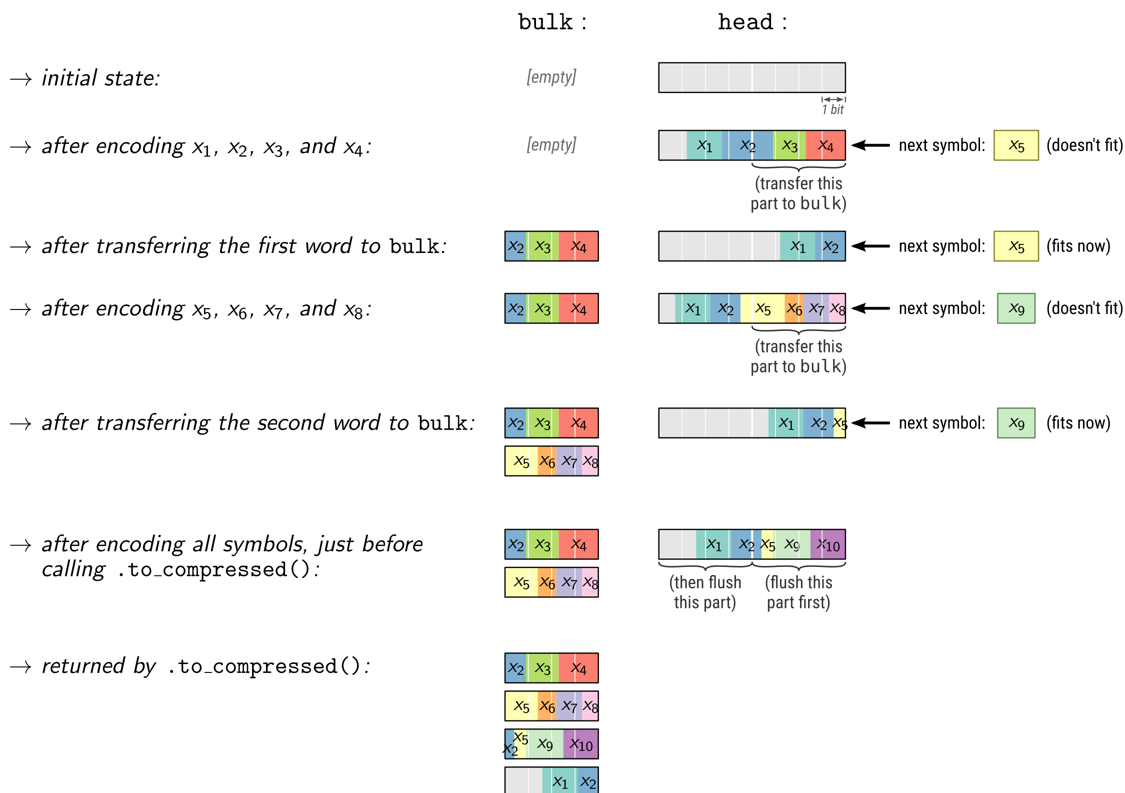
(b) Continue Figure 3 until you have encoded 10 symbols $x_1, \ldots, x_{10}$. Assume that the symbols have the following information contents:

- x_1 has information content 1.5 bits;
- x_2 has information content 2.2 bits;
- x_3 has information content 1.4 bits;
- x_4 has information content 1.7 bits;
- x_5 has information content 1.9 bits;
- x_6 has information content 0.8 bits;
- x_7 has information content 1.1 bits;
- x_8 has information content 0.7 bits;
- x_9 has information content 1.6 bits; and
- x_{10} has information content 1.5 bits.

You may ignore the fact that the corresponding probabilities, $2^{-\text{information content}}$, cannot be precisely represented in fixed point precision.

You should find that some of the symbols get logically "split up" into two or even three not necessarily consecutive parts. Further, the first symbol x_1 stays completely on `head` until the very end in this example.

Solution: See figure below, which also illustrates how the method `to_compressed` exports the final compressed bit string by chopping up `head` into `precision`-bit words and appending them to `bulk`:



In case you're curious, Listing 3 implements streaming ANS in Python. The main differences to our `SlowAnsCoder` class from Listing 2 are that (i) Listing 3 adds a potential transfer of one word from `head` to `bulk` at the beginning of the method `push` as discussed above and, correspondingly, a potential transfer of one word from `bulk` back to `head` at the end of the method `pop`; (ii) all remaining encoding and decoding operations are performed on `head`; and (iii) these encoding and decoding operations are no longer delegated to the methods `push` and `pull` of an internal `UniformCoder` because we've manually inlined these method calls so that Listing 3 is self-contained.

The inlining also allows us to perform some optimizations by replacing multiplications and divisions involving $n = 2^{\texttt{precision}}$ by equivalent and faster bit shifts. This eliminates divisions during decoding, which are by far the slowest arithmetic operations on CPUs.[4]

[4]see, e.g., measurements by Fog: https://www.agner.org/optimize/instruction_tables.pdf

```
class StreamingAnsCoder:
    def __init__(self, precision, compressed=[]):
        self.precision = precision
        self.mask = (1 << precision) - 1 # a string of precision 1-bits
        self.bulk = compressed.copy() # (We will mutate bulk below.)
        self.head = 0
        # Ensure that head is at least half filled unless bulk is empty.
        while len(self.bulk) != 0 and (self.head >> precision) == 0:
            self.head = (self.head << precision) | self.bulk.pop()

    def push(self, symbol, m):       # Encodes one symbol.
        # Check if encoding onto head would lead to an overflow:
        if (self.head >> self.precision) >= m[symbol]:
            # Transfer one word of compressed data from head to bulk:
            self.bulk.append(self.head & self.mask) # (bitwise and)
            self.head >>= self.precision
            # This is the only point where head is (temporarily) less
            # than half filled despite bulk not being empty.

        # Below as in SlowAnsCoder (Listing 2), just with inlined calls:
        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):                # Decodes one symbol.
        # Begin as in SlowAnsCoder (see Listing 2):
        z = self.head & self.mask # (same as "self.head % n" but faster)
        self.head >>= self.precision # (same as "//= n" but faster)
        for symbol, m_symbol in enumerate(m):
            if z >= m_symbol:
                z -= m_symbol
            else:
                break
        self.head = self.head * m_symbol + z

        # Detect whether push transferred data from head to bulk here:
        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
```

Listing 3: A complete streaming ANS entropy coder in Python. For a usage example, see lower part of Listing 2 (replace `SlowAnsCoder` with `StreamingAnsCoder`).