Lecture 7

Asymmetric Numeral Systems

RECAP -> Arithmetic coding

1. Given *any* distribution P, achieves *optimal* compression. Thus, Arithmetic coding allows for `model` and `entropy coding` separation.
2. Can work very well with changing distribution P.
   i.e. Adaptive algorithms work well with Arithmetic coding

RECAP -> Arithmetic coding in practice

Lots of Variants of Arithmetic coding; mainly come from how they implement the rescaling.

1. **Arithmetic coding:** Bit-based rescaling -> keeping a count of the mid-ranges etc.
   SCL Arithmetic coding
2. **Range Coding** Byte (8-bit based rescaling), word-based rescaling ->
   SCL range coding

RECAP -> Arithmetic/Range coders in practice

Used almost everywhere! (either as Range coder or Arithmetic coding)

1. JPEG2000, BPG, H265, H266, VP8
2. CMIX, `tensorflow-compress`, NNCP

RECAP -> Arithmetic/Range coders in practice

Although Arithmetic coding algorithms are quite efficient, they are not fast enough! (especially when compared with Huffman coding)

Codec	Encode speed	Decode speed	compression
Huffman coding	252 Mb/s	300 Mb/s	1.66
Arithmetic coding	120 Mb/s	69 Mb/s	1.24

NOTE -> Speed numbers from: Charles Bloom's blog

RECAP -> Arithmetic/Range coders in practice

Codec	Encode speed	Decode speed	compression
Huffman coding	252 Mb/s	300 Mb/s	1.66
Arithmetic coding	120 Mb/s	69 Mb/s	1.24
rANS	76 Mb/s	140 Mb/s	1.24
tANS	163 Mb/s	284 Mb/s	1.25

NOTE -> Speed numbers from: Charles Bloom's blog

ANS: Asymmetric Numeral System

Among others, ANS is used in the Facebook Zstandard compressor[6][7] (also used e.g. in Linux kernel,[8] Android[9] operating system, was published as RFC 8478 for MIME[10] and HTTP[11]), Apple LZFSE compressor,[12] Google Draco 3D compressor[13] (used e.g. in Pixar Universal Scene Description format[14]) and PIK image compressor,[15] CRAM DNA compressor[16] from SAMtools utilities,[17] Dropbox DivANS compressor,[18] Microsoft DirectStorage BCPack texture compressor,[19] and JPEG XL[20] image compressor.

ANS: Asymmetric Numeral System

- `rANS` : *range-ANS*: drop-in replacement for Arithmetic coding, but faster
- `tANS/FSE` : *table-ANS* (or *Finite State Engine*): drop-in replacement for Huffman coding, but better in terms of compression (and similar speed)

Why *"Asymmetric Numeral System"*?

Why *"Asymmetric Numeral System"*?

Lets assume inputs are digits `[0,9]`

```
data_input = [3,2,4,1,5]
```

Quiz-1: Form a single number `x` (*state*) representing `data_input`?

Why *"Asymmetric Numeral System"*?

Lets assume inputs are digits `[0,9]`

```
data_input = [3,2,4,1,5]
```

Quiz-1: Form a single number `x` (*state*) representing `data_input`?

Ans: `[3,2,4,1,5] -> 32415`

Symmetric Numeral System: *Encoding*

```
# given
data_input = [3,2,4,1,5]

## "encoding" process
x = 0  # <-- initial state
x = x*10 + 3 #x = 3
x = x*10 + 2 #x = 32
x = x*10 + 4 #x = 324
x = x*10 + 1 #x = 3241
x = x*10 + 5 # x = 32415 <- final state
```

Symmetric Numeral System: *Decoding*

Quiz-2: Given the *state* `x`, how can you retrieve the `data_input`?

```
symbols = []
x = 32145 # <- final state
n = 5
```

Symmetric Numeral System: *Decoding*

Quiz-2: Given the *state* `x`, and number of elements `n=5` how can you retrieve the `data_input`?

```
symbols = []
x = 32145 # <- final state
n = 5

# repeat n=5 times
s, x = x%10, x//10  # s,x = 5, 3214
s, x = x%10, x//10  # s,x = 4, 321
s, x = x%10, x//10  # s,x = 1, 32
s, x = x%10, x//10  # s,x = 2, 3
s, x = x%10, x//10  # s,x = 3, 0
```

Symmetric Numeral System: *Encoding, Decoding*

`encode_step`, `decode_step` are exact inverses of each other

```
def encode_step(x_prev,s):
    x = x_prev*10 + s
    return x

def decode_step(x)
    s = x%10 # decode symbol
    x_prev = x//10 # retrieve the previous state
    return (s,x_prev)
```

Symmetric Numeral System

```
symbols = []
x = 32145 # <- final state
n = 5
```

```
# Encoding
state x: 0 -> 3 -> 32 -> 321 -> 3214 -> 32145

# decoding
state x: 0 <- 3 <- 32 <- 321 -> 3214 <- 32145
```

Symmetric Numeral System: Full *Encoding*

Transmit the final state `x` in binary using `ceil(log_2[x+1]))`. (ex: `32145 = b111110110010001`).

```
## Encoding
def encode_step(x_prev,s):
   x = x_prev*10 + s
   return x

def encode(data_input):
    x = 0 # initial state
    for s in data_input:
        x = encode_step(x,s)

    return to_binary(x) # takes ~log_2(x) bits
```

Symmetric Numeral System: Full *Decoding*

```python
## Decoding
def decode_step(x):
    s = x%10 # <- decode symbol
    x_prev = x//10 # <- retrieve the previous state
    return (s,x_prev)

def decode(bits, num_symbols):
   x = to_uint(bits) # convert bits to final state

   # main decoding loop
   symbols = []
   for _ in range(num_symbols):
       s, x = decode_step(x)
       symbols.append(s)
   return reverse(symbols) # need to reverse to get original sequence
```

Symmetric Numeral System: Compression performance

- `x <- 10*x_prev + s`, i.e `x ~ x_prev*10`
- Per symbol we are using approximately:

$$\log_2(10) = \log_2 \frac{1}{0.1}$$

- Our compressor is "optimal", only if all our symbol `[0,9]` have equal probability of `0.1`.

Asymmetric Numeral Systems:

Quiz 2: If that digits `[0,9]` have different probabilities, how can we modify our algorithm to achieve better compression?

Asymmetric Numeral Systems:

If that digits `[0,9]` have different probabilities, how can we modify our algorithm to achieve better compression?

Ans: Instead of scaling `x ~ 10*x_prev`, we want to scale it as `x ~ (1/prob[s])*x_prev`

rANS: Notation

- represent probabilities in terms of integer `frequencies`

```
prob[s] = freq[s]/M
```

- For example:

```
# prob in terms of integers
alphabet  -> {0,1,2}, prob -> [3/8, 3/8, 2/8]
freq -> [3,3,2], M = 8
cumul -> [0,3,6]
```

rANS: Encoding

```
def rans_base_encode_step(x_prev,s):
   # TODO: add details here
   return x

def encode(data_input):
    x = 0 # initial state
    for s in data_input:
        x = rans_base_encode_step(x,s)

    return to_binary(x)
```

Quiz 4: What is the `rans_base_encode_step`?

(*HINT*: `x ~ (1/prob[s])*x_prev`)

rANS: Encoding

```
def rans_base_encode_step(x_prev,s):
   x = (x_prev//freq[s])*M + cumul[s] + x_prev%freq[s]
   return x

def encode(data_input):
    x = 0 # initial state
    for s in data_input:
        x = rans_base_encode_step(x,s)

    return to_binary(x)
```

(*HINT*: `x ~ (1/prob[s])*x_prev`)

rANS: Encoding

```
def rans_base_encode_step(x_prev,s):
    x = (x_prev//freq[s])*M + cumul[s] + x_prev%freq[s]
    return x
```

rANS: Encoding

```
def rans_base_encode_step(x_prev,s):
    x = (x_prev//freq[s])*M + cumul[s] + x_prev%freq[s]
    return x
```

Example-1

```
alphabet  -> {0,1,2}
freq -> [3,3,2], M = 8, cumul -> [0,3,6]
```

Quiz 4: What is the final encoding for `data_input = [1,0,2,1]` ?

rANS: Encoding

```
def rans_base_encode_step(x_prev,s):
    x = (x_prev//freq[s])*M + cumul[s] + x_prev%freq[s]
    return x
```

Example-1:

```
freq -> [3,3,2], M = 8, cumul -> [0,3,6]
```

```
step 0: x = 0
step 1: s = 1 -> x = (0//3)*8 + 3 + (0%3) = 3
step 2: s = 0 -> x = (3//3)*8 + 0 + (3%3) = 8
step 3: s = 2 -> x = (8//2)*8 + 6 + (8%2) = 38
step 4: s = 1 -> x = (38//3)*8 + 3 + (38%3) = 101
```

rANS: Encoding

```
def rans_base_encode_step(x_prev,s):
   x = (x_prev//freq[s])*M + cumul[s] + x_prev%freq[s]
   return x
```

Example-2

```
alphabet = {0,1,2,3,4,5,6,7,8,9}
freq[s] = 1, M = 10
prob[s] = 1/10
```

Quiz 5: How does the `rans_base_encode_step` function look for the input above?

rAN S: Encoding

`x = (x_prev//freq[s])*M + cumul[s] + x_prev%freq[s]`

```python
def rans_base_encode_step(x_prev,s):
   # find block_id, slot
   block_id = x_prev//freq[s]
   slot = cumul[s] + (x_prev % freq[s])

   # compute next state x
   x = block_id*M + slot
   return x
```

Quiz 6: What can you say about the `slot` ?

rANS: Encoding

```
x = (x_prev//freq[s])*M + cumul[s] + x_prev%freq[s]
```

```
def rans_base_encode_step(x_prev,s):
    # find block_id, slot
    block_id = x_prev//freq[s]
    slot = cumul[s] + (x_prev % freq[s])

    # compute next state x
    x = block_id*M + slot
    return x
```

Quiz 6: What can you say about the `slot`?

Ans: `0 <= slot < M`

rANS: Decoding

```
def rans_base_encode_step(x_prev,s):
   block_id = x_prev//freq[s]
   slot = cumul[s] + (x_prev % freq[s])
   x = block_id*M + slot
   return x

def rans_base_decode_step(x):
   # TODO -> fill in the decoder here
   return (s,x_prev)
```

rANS: Decoding

```
def rans_base_encode_step(x_prev,s):
   block_id = x_prev//freq[s]
   slot = cumul[s] + (x_prev % freq[s])
   x = block_id*M + slot
   return x

def rans_base_decode_step(x):
   # Step I: find block_id, slot
   block_id = ?
   slot = ?


   ...
   return (s,x_prev)
```

STEP-I: Decode `block_id, slot`

rANS: Decoding

```
def rans_base_encode_step(x_prev,s):
    block_id = x_prev//freq[s]
    slot = cumul[s] + (x_prev % freq[s])
    x = block_id*M + slot
    return x

def rans_base_decode_step(x):
    block_id, slot = x//M, x%M # Step I

    s = ? # Step II: decode symbol s
    ...
    return (s,x_prev)
```

STEP-II: Decode symbol `s`

rANS: Decoding

```
def rans_base_encode_step(x_prev,s):
    block_id = x_prev//freq[s]
    slot = cumul[s] + (x_prev % freq[s])
    x = block_id*M + slot
    return x

def rans_base_decode_step(x):
    block_id, slot = x//M, x%M # Step I

    s = ? # Step II: decode symbol s
    ...
    return (s,x_prev)
```

HINT: `cumul[s] <= slot < cumul[s+1]`

rANS: Decoding

```
def rans_base_encode_step(x_prev,s):
   block_id = x_prev//freq[s]
   slot = cumul[s] + (x_prev % freq[s])
   x = block_id*M + slot
   return x

def rans_base_decode_step(x):
   block_id, slot = x//M, x%M # Step I
   s = find_bin(cumul, slot) # Step II: decode symbol s
   ...
   return (s,x_prev)
```

HINT: `cumul[s] <= slot < cumul[s+1]`

rANS: Decoding

```
def rans_base_encode_step(x_prev,s):
    block_id = x_prev//freq[s]
    slot = cumul[s] + (x_prev % freq[s])
    x = block_id*M + slot
    return x

def rans_base_decode_step(x):
    block_id, slot = x//M, x%M # Step I
    s = find_bin(cumul, slot) # Step II: decode symbol s

    x_prev = ? # Step-III
    return (s,x_prev)
```

STEP-III: Decode prev state `x_prev`

rANS: Decoding

```
def rans_base_encode_step(x_prev,s):
   block_id = x_prev//freq[s]
   slot = cumul[s] + (x_prev % freq[s])
   x = block_id*M + slot
   return x

def rans_base_decode_step(x):
   block_id, slot = x//M, x%M # Step I
   s = find_bin(cumul, slot) # Step II: decode symbol s
   x_prev = block_id*freq[s] + slot - cumul[s] # Step-III
   return (s,x_prev)
```

rANS: Decoding example

```
def rans_base_decode_step(x):
   block_id, slot = x//M, x%M # Step I
   s = find_bin(cumul, slot) # Step II: decode symbol s
   x_prev = block_id*freq[s] + slot - cumul[s] # Step-III
   return (s,x_prev)
```

Example-1:

```
freq -> [3,3,2], M = 8, cumul -> [0,3,6]
```

```
# Decode
bitarray = b1100101 -> x_final = 101
n = 4
```

rANS: Encoder, Decoder

Interactive snippet: https://kedartatwawadi.github.io/post--ANS/

rANS: Full Encoder

```
def rans_base_encode_step(x_prev,s):
   block_id = x_prev//freq[s]
   slot = cumul[s] + (x_prev % freq[s])
   x = block_id*M + slot
   return x

def encode(data_input):
    x = 0 # initial state
    for s in data_input:
        x = rans_base_encode_step(x,s)
    return to_binary(x)
```

rANS: Full Decoder

```
def rans_base_decode_step(x):
    block_id, slot = x//M, x%M # Step I
    s = find_bin(cumul, slot) # Step II: decode symbol s
    x_prev = block_id*freq[s] + slot - cumul[s] # Step-III
   return (s,x_prev)

def decode(bits, num_symbols):
   x = to_uint(bits) # convert bits to final state

   # main decoding loop
   decoded_symbols = []
   for _ in range(num_symbols):
       s, x = rans_base_decode_step(x)
       decoded_symbols.append(s)
   return reverse(decoded_symbols) # need to reverse to get original sequence
```

rANS Compression performance:

- state increases as: $x \approx x_{prev} \cdot \frac{M}{freq[s]} = x_{prev} \cdot \frac{1}{P(s)}$
- Final state x_{final} represented using $\approx \log_2(x_{final})$ bits.

Quiz 8: What is the approximate encode length for input s^n?

$$L(s^n) \approx \log_2 \frac{1}{P(s^n)}$$

rANS Compression performance:

- state increases as: $x \approx x_{prev} \cdot \frac{M}{freq[s]} = x_{prev} \cdot \frac{1}{P(s)}$
- Final state x_{final} represented using $\approx \log_2(x_{final})$ bits.

Quiz 8: What is the approximate encode length for input s^n?

$$L(s^n) \approx \log_2 \frac{1}{P(s^n)}$$

i.e. `rANS` is *optimal*!

rANS vs Arithmetic coding

- **Optimal Compression**: Compression performance *optimal* and similar to Arithmetic coding in practice
- **Reverse decoding**: Decoding is in the *reverse* order unlike Arithmetic coding, which can be a bit of a problem for Adaptive schemes
- **Simpler encoding/decoding**: Encoding, Decoding operations are a bit simpler, making it easier to modify and optimize for speed

rANS in practice

Quiz 9: What is the practical problem with our rANS Encoding/Decoding?

RANS ENCODING EXAMPLE

Symbol Counts, $\mathcal{F}$ 3,3,2

Input Symbol String: 1,0,2,1,0,2,2,1,0,1, 2,2,2,2

Try it

Input	State
1	3
0	8
2	38
1	101
0	266
2	1070
2	4286
1	11429
0	30474
1	81267
2	325071
2	1300287
2	5201151
2	20804607

rANS in practice

Quiz 9: What is the practical problem with our rANS Encoding/Decoding?

- The state increases as: $x \approx x_{prev} \cdot \frac{M}{freq[s]} = x_{prev} \cdot \frac{1}{P(s)}$
- After `~30-40` symbols we are going to need more than `64` bits

Solution: Restrict the state `x` to lie in an interval `[L,H]`

Streaming rANS: Encoding

```
def rans_base_encode_step(x_shrunk, s):
    ...

def shrink_state(x_prev,s):
    ...
    return x_shrunk, out_bits

def encode_step(x_prev,s):
   # shrink state x before calling base encode
   x_shrunk, out_bits = shrink_state(x, s)

   # perform the base encoding step
   x = rans_base_encode_step(x_shrunk,s)
   assert x in [L,H]
   return x, out_bits
```

Streaming rANS: shrink state

```
def encode_step(x_prev,s):
   # shrink state x before calling base encode
   x_shrunk, out_bits = shrink_state(x, s)

   # perform the base encoding step
   x = rans_base_encode_step(x_shrunk,s)
   assert x in [L,H]
   return x, out_bits
```

- Before encoding a symbol into the state `x <- base_step(x_prev,s)`, we shrink the state `x_prev -> x_shrunk`, so that `base_step(x_shrunk,s) in [L,H]`

Streaming rANS: shrink state function

```
def shrink_state(x_prev,s):
    ...
    return x_shrunk, out_bits
```

```
# input state
x_prev = 21 = 10110b

# stream out bits from x_prev until we are sure base_step(x_shrunk,s) in [L,H]
x_shrunk = 10110b = 22, out_bits = ""
x_shrunk = 1011b = 11, out_bits = "0"
x_shrunk = 101b = 5, out_bits = "10"
...
```

Streaming rANS: shrink state function

```
def shrink_state(x_prev,s):
    ...
    while rans_base_encode_step(x_shrunk,s) not in Interval[L,H]:
        out_bits.prepend(x_shrunk%2)
        x_shrunk = x_shrunk//2
    x_shrunk = x
    return x_shrunk, out_bits
```

```
# input state
x_prev = 21 = 10110b

# stream out bits from x_prev until we are sure base_step(x_shrunk,s) in [L,H]
x_shrunk = 10110b = 22, out_bits = ""
x_shrunk = 1011b = 11, out_bits = "0"
x_shrunk = 101b = 5, out_bits = "10"
...
```

How many bits to stream out?

- We need to choose `[L,H]` so that it is always possible to stream out bits from the state `x_prev` and guarantee that a at some point `base_encode(x_shrunk, s) lies in [L,H]`
- For unique decoding, we need to ensure that forthere only one such `x_shrunk` for which `base_encode(x_shrunk, s) lies in [L,H]`
- Condition satisfied for `[L,H] = [M*t, 2*M*t - 1]`

Streaming rANS: Full Encoding

```
## streaming params
t = 1 # can be any uint
L = M*t
H = 2*M*t - 1

########################### Streaming rANS Encoder ###########################
def rans_base_encode_step(x,s):
   x_next = (x//freq[s])*M + cumul[s] + x%freq[s]
   return x_next

def shrink_state(x,s):
   # initialize the output bitarray
   out_bits = BitArray()

   # shrink state until we are sure the encoded state will lie in the correct interval
   while rans_base_encode_step(x,s) not in Interval[freq[s]*t,2*freq[s]*t - 1]:
       out_bits.prepend(x%2)
       x = x//2
   x_shrunk = x
   return x_shrunk, out_bits

def encode_step(x,s):
   # shrink state x before calling base encode
   x_shrunk, out_bits = shrink_state(x, s)

   # perform the base encoding step
   x_next = rans_base_encode_step(x_shrunk,s)

   return x_next, out_bits
```

Streaming rANS: Full decoder

```
############################ Streaming rANS Decoder
def rans_base_decode_step(x):
   ...
   return (s,x_prev)

def expand_state(x_shrunk, enc_bitarray):
    # init
    num_bits_step = 0

    # read in bits to expand x_shrunk -> x
    x = x_shrunk
    while x not in Interval[L,H]:
        x = x*2 + enc_bitarray[num_bits_step]
        num_bits_step += 1
    return x, num_bits_step

def decode_step(x, enc_bitarray):
    # decode s, retrieve prev state
    s, x_shrunk = rans_base_decode_step(x)

    # expand back x_shrunk to lie in Interval[L,H]
    x_prev, num_bits_step = expand_state(x_shrunk, enc_bitarray)
    return s, x_prev, num_bits_step

def decode(encoded_bitarray, num_symbols):
    ...
    return reverse(symbols) # need to reverse to get original sequence
```

Streaming rANS in practice:

- `M = power of 2`
- `[L,H] = Mt, 2Mt−1` -> `t` chosen as large as possible.
- typically byte/words are streamed out instead of a bit
  (NOTE -> in case of byte streaming, the interval is `[L,H] = Mt, 256*Mt - 1`)

Byte/word streaming:

typically byte/words are streamed out instead of a bit

(numbers from https://github.com/rygorous/ryg_rans)

```
# reading/writing one byte at a time
rANS encode: 9277880 clocks, 12.1 clocks/symbol (299.6MiB/s)
rANS decode 14754012 clocks, 19.2 clocks/symbol (188.4MiB/s)

# reading/writing 32 bits at a time:
rANS encode: 7726256 clocks, 10.1 clocks/symbol (359.8MiB/s)
rANS decode: 12159778 clocks, 15.8 clocks/symbol (228.6MiB/s)
```

Caching rANS computations

- `x in [L,H]` -> for small values interval size, can we cache the `encoding` ?

```
def encode_step(x,s):
   # shrink state x before calling base encode
   x_shrunk, out_bits = shrink_state(x, s)

   # perform the base encoding step
   x_next = rans_base_encode_step(x_shrunk,s)

   return x_next, out_bits
```

Caching rANS computation:

- `rans_base_encode_step(x_shrunk,s)` goes from

$$[L, H] \times \{alphabet\} \rightarrow [L, H]$$

STREAMING-RANS AS A FSE

Symbol Counts, $\mathcal{F}$ 3,3,2

Try it

Output State

Input State	A	B	C
8	9	12	14
9	9	12	14
10	10	13	14
11	10	13	14
12	8	11	15
13	8	11	15
14	8	11	15
15	8	11	15

BitStream Output

Input State	A	B	C
8	0	0	00
9	1	1	10
10	0	0	01
11	1	1	11
12	00	00	00
13	10	10	10
14	01	01	01
15	11	11	11

Caching rANS computation -> tANS

- Both encoding, decoding can be appropriately modified to be basically in terms of lookup-tables
- Lookup-tables are fast, so the encoding/decoding speeds for `tANS` are similar to Huffman coding
- Cannot use a very large interval `[L,H]`, so there is some compression loss (based on how much memory you can use)

Asymmetric Numeral Systems:

- ANS class of algorithms -> very flexible and can tradeoff compression ratio a bit for very fast implementation:
  (`rANS` -> `tANS`)

Codec	Encode speed	Decode speed	compression
Huffman coding	252 Mb/s	300 Mb/s	1.66
Arithmetic coding	120 Mb/s	69 Mb/s	1.24
rANS	76 Mb/s	140 Mb/s	1.24
tANS	163 Mb/s	284 Mb/s	1.25

NOTE -> Speed numbers from: Charles Bloom's blog

rANS, tANS implementations in SCL

- rANS: https://github.com/kedartatwawadi/stanford_compression_library/blob/main/compressors/rANS.py
- tANS: https://github.com/kedartatwawadi/stanford_compression_library/blob/main/compressors/tANS.py

What next?

We have been discussing compression of i.i.d data.

What if the data was dependent?