Arithmetic coding & rANS recap + a bit more!

EE 274, Fall 2025

Issues with symbol codes:

1. `P = {A: 0.1, B: 0.9}`, `H(P) = 0.47`
   Huffman code can only compress this to `1 bit`.
2. For any symbol s, ideally we want to use $l(s) = \log_2 \frac{1}{P(s)}$ bits. But, as we are using a symbol code, we cant use fractional bits.

Thus, there is always going to be `~1 bit` overhead per symbol with symbol codes

Solution -> use block codes

We can do better by considering blocks of size 2: `P = {AA: 0.01, AB: 0.09, BA: 0.09, BB: 0.81}`, this way the overhead is `~1 bit` per 2 symbol!

(or in case of blocks of size `B`, the overhead is `~1 bit` per symbol)

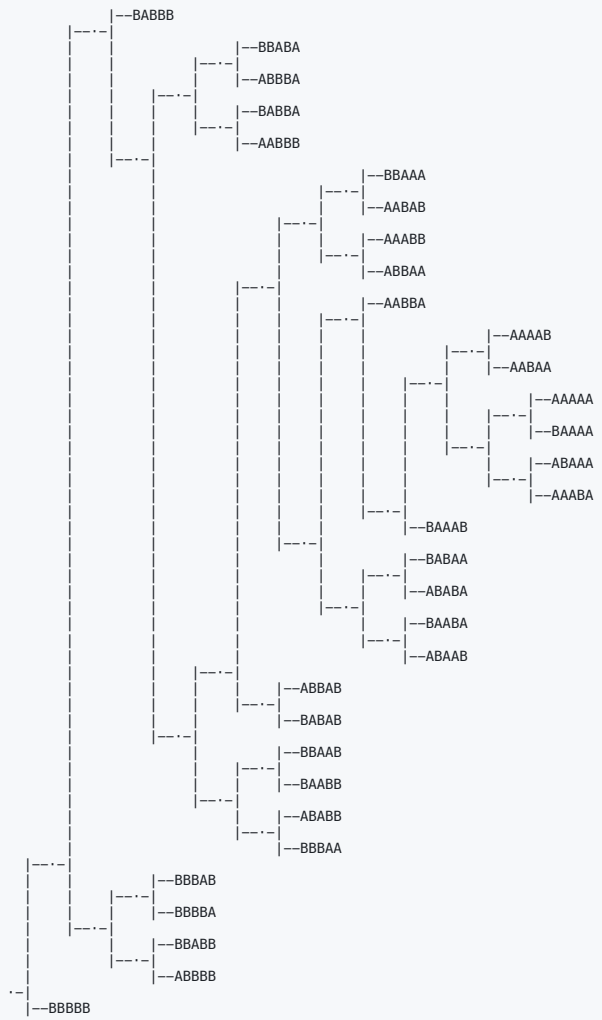
```
## Huffman code for blocks
block_size: 1, entropy: 0.47, avg_codelen: 1.00 bits/symbol
block_size: 2, entropy: 0.47, avg_codelen: 0.65 bits/symbol
block_size: 3, entropy: 0.47, avg_codelen: 0.53 bits/symbol
block_size: 4, entropy: 0.47, avg_codelen: 0.49 bits/symbol
block_size: 5, entropy: 0.47, avg_codelen: 0.48 bits/symbol
```

Huffman codes on blocks

```
block_size: 1, entropy: 0.47, avg_codelen: 1.00 bits/symbol

  |--A
·-|
  |--B
block_size: 2, entropy: 0.47, avg_codelen: 0.65 bits/symbol

       |--BA
  |--·-|
  |    |    |--AA
  |    |--·-|
  |         |--AB
·-|
  |--BB
block_size: 3, entropy: 0.47, avg_codelen: 0.53 bits/symbol

                     |--AAA
                |--·-|
                |    |--BAA
           |--·-|
           |    |    |--ABA
           |    |--·-|
           |         |--AAB
       |--·-|
       |    |--ABB
  |--·-|
  |    |    |--BBA
  |    |--·-|
  |         |--BAB
·-|
  |--BBB
```

Huffman codes on blocks

block_size: 5, entropy: 0.47, avg_codelen: 0.48 bits/symbol



Huffman codes on blocks

1. Huffman codes

$$H(X) \leq \mathbb{E}[l(X)] \leq H(X) + 1$$

2. Huffman codes on blocks of size B

$$H(X) \leq \frac{\mathbb{E}[l(X_1^B)]}{B} \leq H(X) + \frac{1}{B}$$

Question: If block Huffman gives us *optimal* compression, then what is the issue?

Huffman codes on blocks

1. Convergence to entropy $H(X)$ is quite fast -> `1/B`
2. But, not very practical, as the codebook size needed is large (exponential):

$$size = |\mathcal{X}|^B$$

3. *Larger codebook* -> difficult to handle,
   block size limited by device memory,
   higher latency, ...

Arithmetic coding

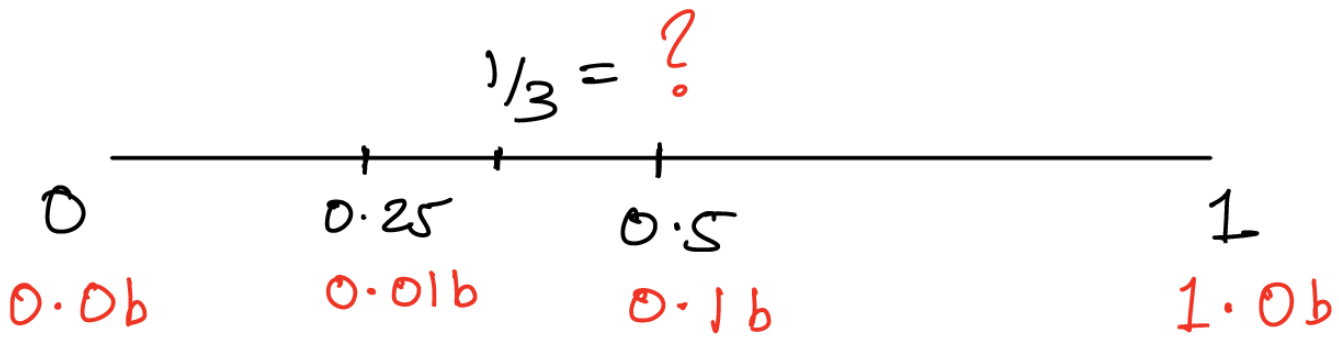
1. For data x_1^n, the `block_size = n`
   i.e. the entire data is a single block!
2. Codeword is computed on *on the fly*
   No need to pre-compute the codebook beforehand
3. Very Efficient! -> *theoretically* the performance can be shown to be:

$$H(X) \leq \frac{\mathbb{E}[l(X_1^n)]}{B} \leq H(X) + \frac{2}{n}$$

i.e. `~2 bits` of overhead for the *entire* sequence

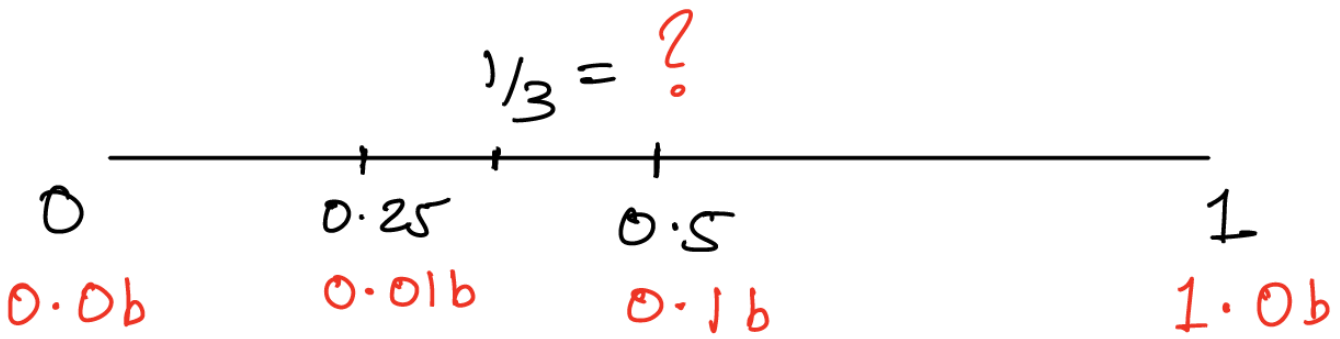
How does Arithmetic coding work?

Primer: the number line (in binary)




```
# floating point values in binary
0.3333 = # b0.... ?
0.6666 = #?
```

Primer: the number line (in binary)




```
# floating point values in binary
from utils.bitarray_utils import float_to_bitarrays
_, binary_exp = float_to_bitarrays(0.3333333, 20)

0.3333 = b0.010101...
0.6666 = b0.101010...
```

Arithmetic Encoding

We will consider the following running example:

```
P = ProbabilityDist({A: 0.3, B: 0.5, C: 0.2})
x_input = BACB
```

We want to encode the sequence $x_1^n = BACB$ sampled from the distribution P.

Arithmetic Encoding

1. **STEP I**: Find an *interval* (or a *range*) `[L,H)`, corresponding to the *entire sequence* x_1^n

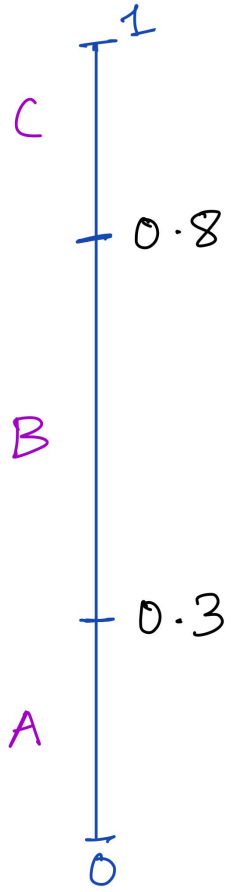
```
x_input -> |------------------[.....)------------|
           0                  low     high       1
```

2. **STEP II**: Communicate the *interval* `[L,H)` *efficiently*
   (i.e. using less number of bits)

```
x_input -> [L,H) -> 011010
```

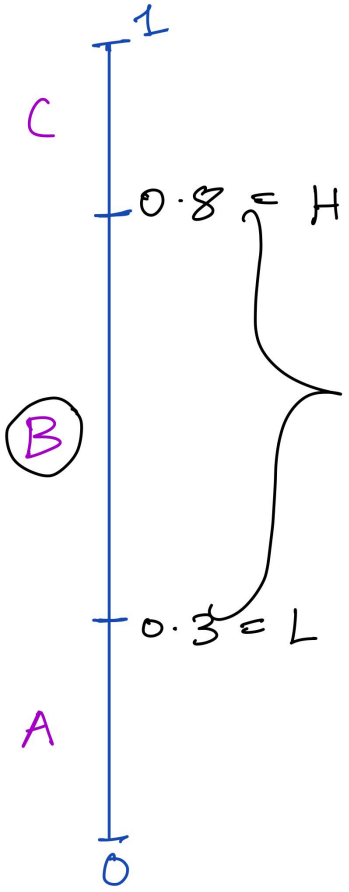
Arithmetic coding example

$P = \{A : 0.3, B : 0.5, C : 0.2\}$, $X_1^4 = BACB$



Arithmetic coding example

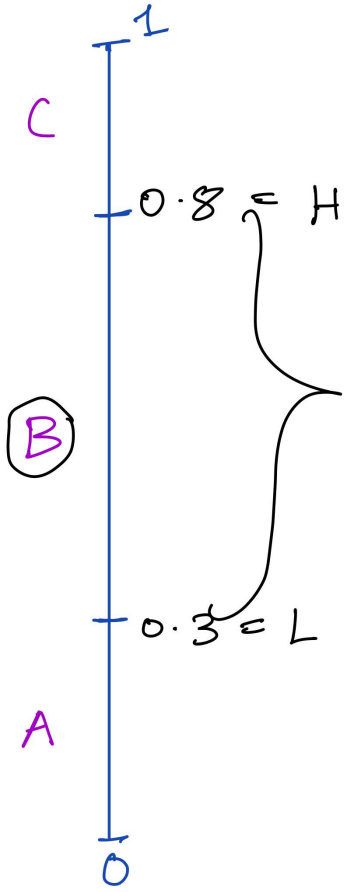
$P = \{A : 0.3, B : 0.5, C : 0.2\}$, $X_1^n = BACB$



B → Interval $[0.3, 0.8)$

Arithmetic coding example

$P = \{A : 0.3, B : 0.5, C : 0.2\}$, $X_1^n = BACB$

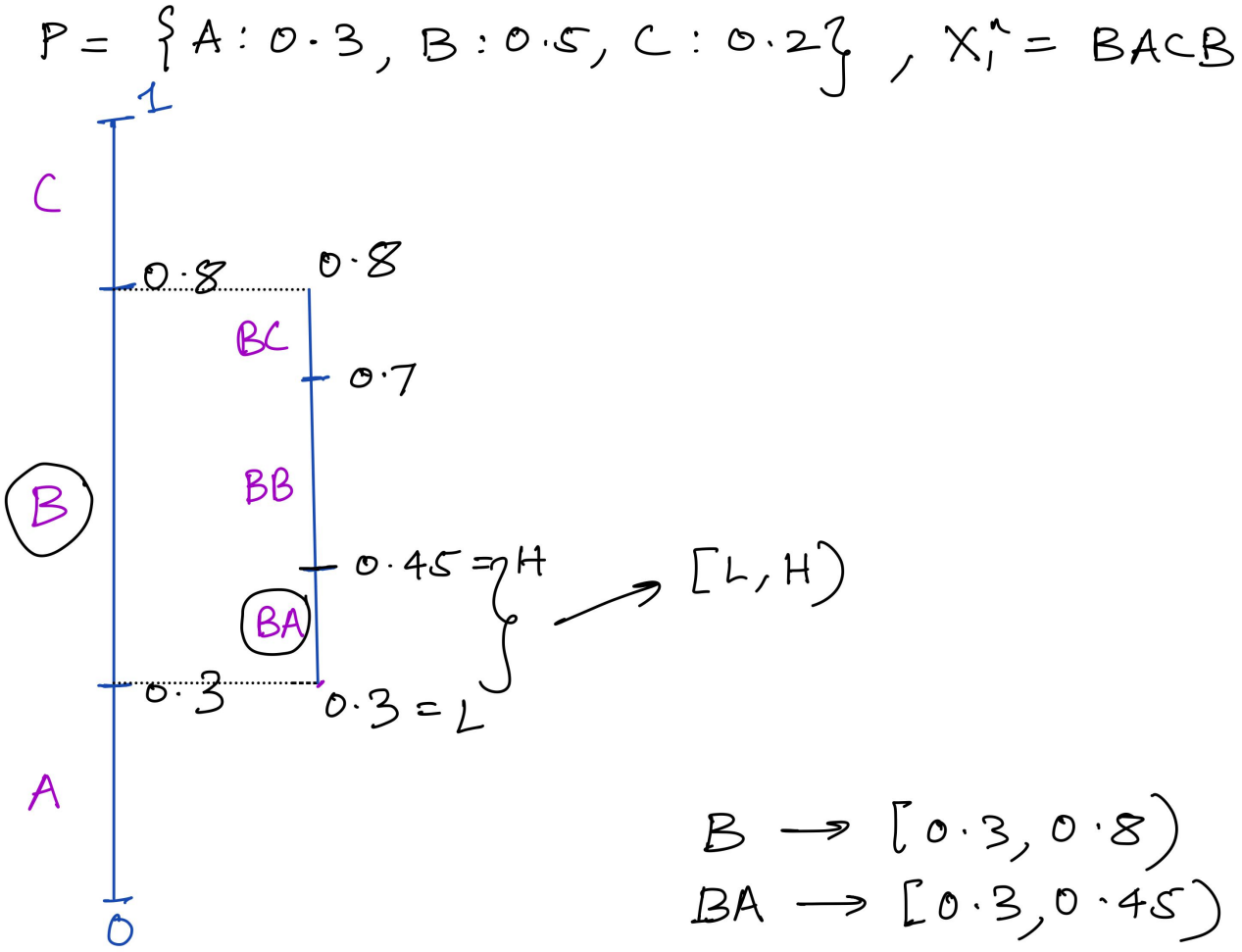


B → interval $[0.3, 0.8)$

	A	B	C
P :	0.3	0.5	0.2
C :	0.0	0.3	0.8

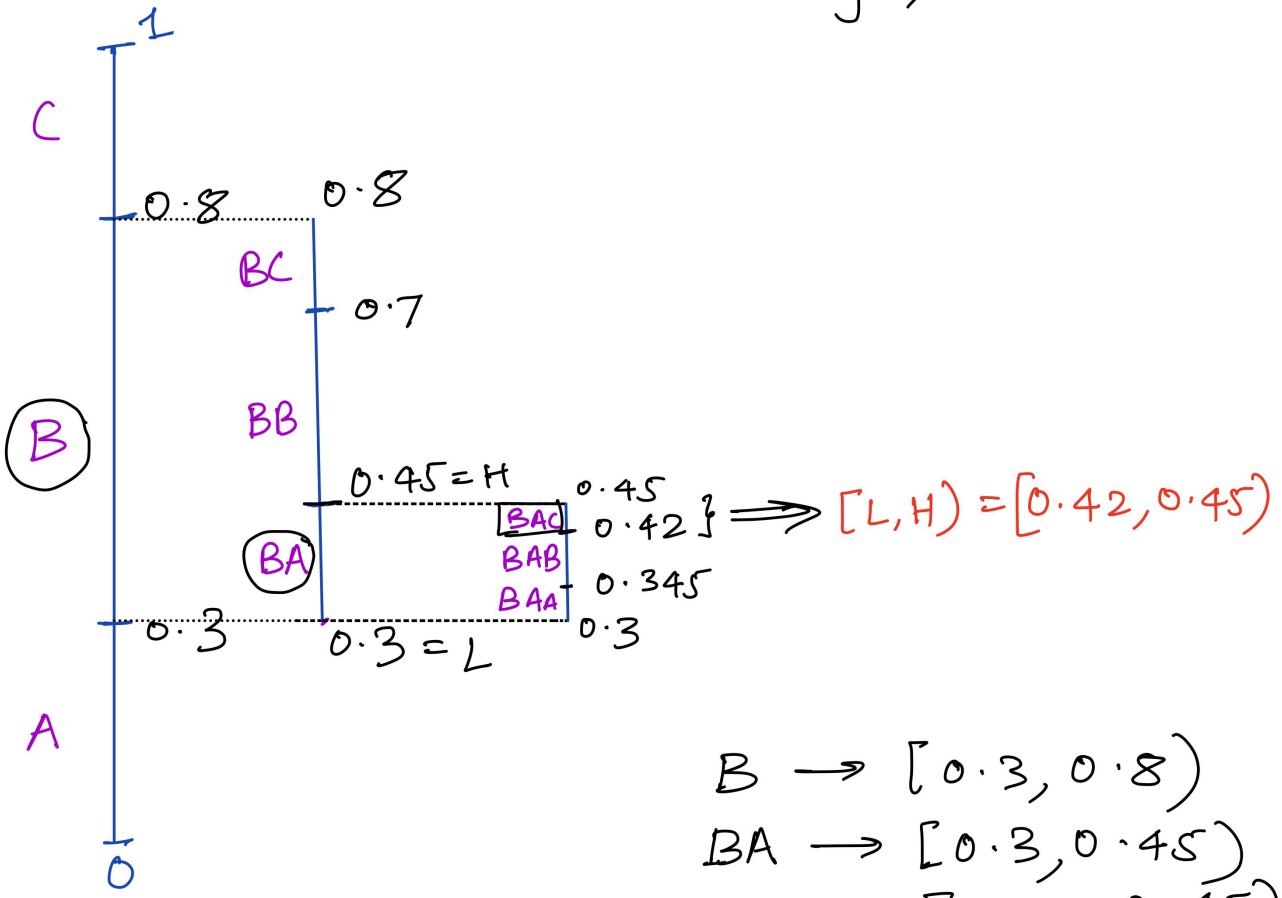
$[L, H) = [C(B), C(B) + P(B))$

Arithmetic coding example



Arithmetic coding example

$P = \{A : 0.3, B : 0.5, C : 0.2\}$, $X_1^n = BACB$



$\Rightarrow [L, H) = [0.42, 0.45)$

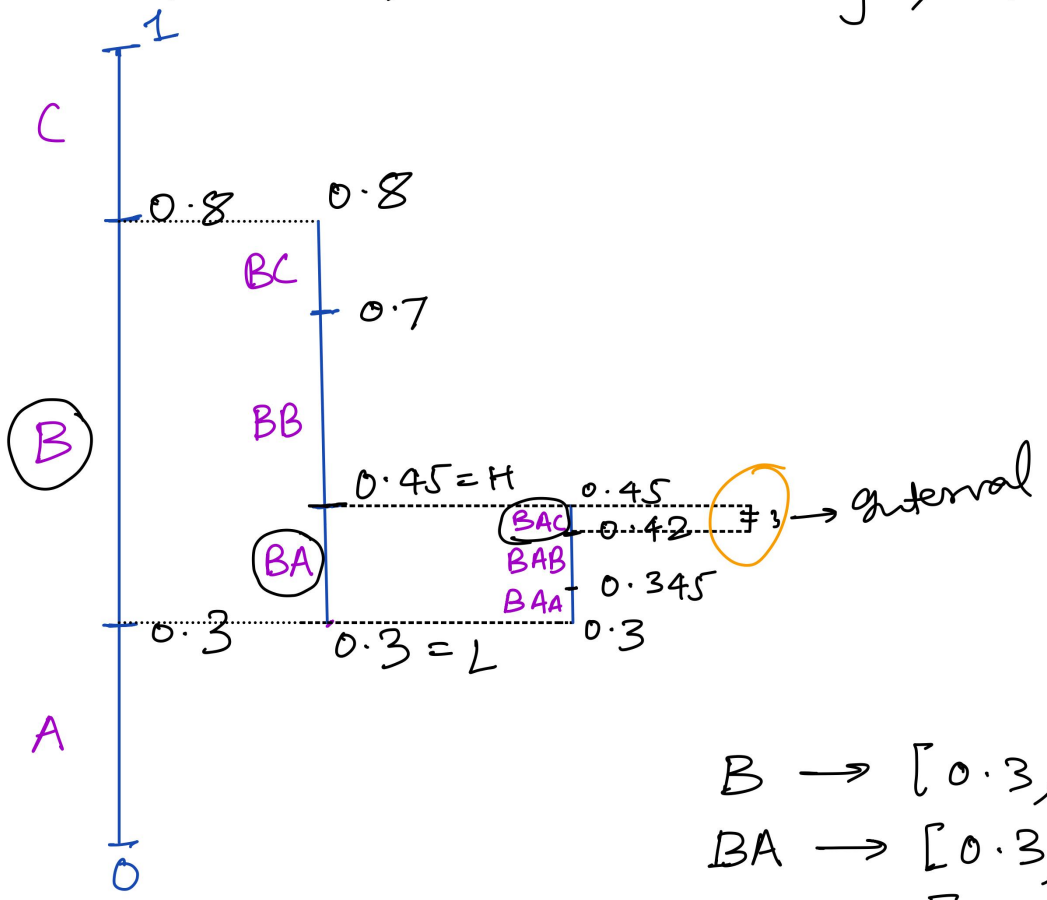
$B \rightarrow [0.3, 0.8)$

$BA \rightarrow [0.3, 0.45)$

$BAC \rightarrow [0.42, 0.45)$

Arithmetic coding example

$P = \{A : 0.3, B : 0.5, C : 0.2\}$, $X_1^n = BACB$

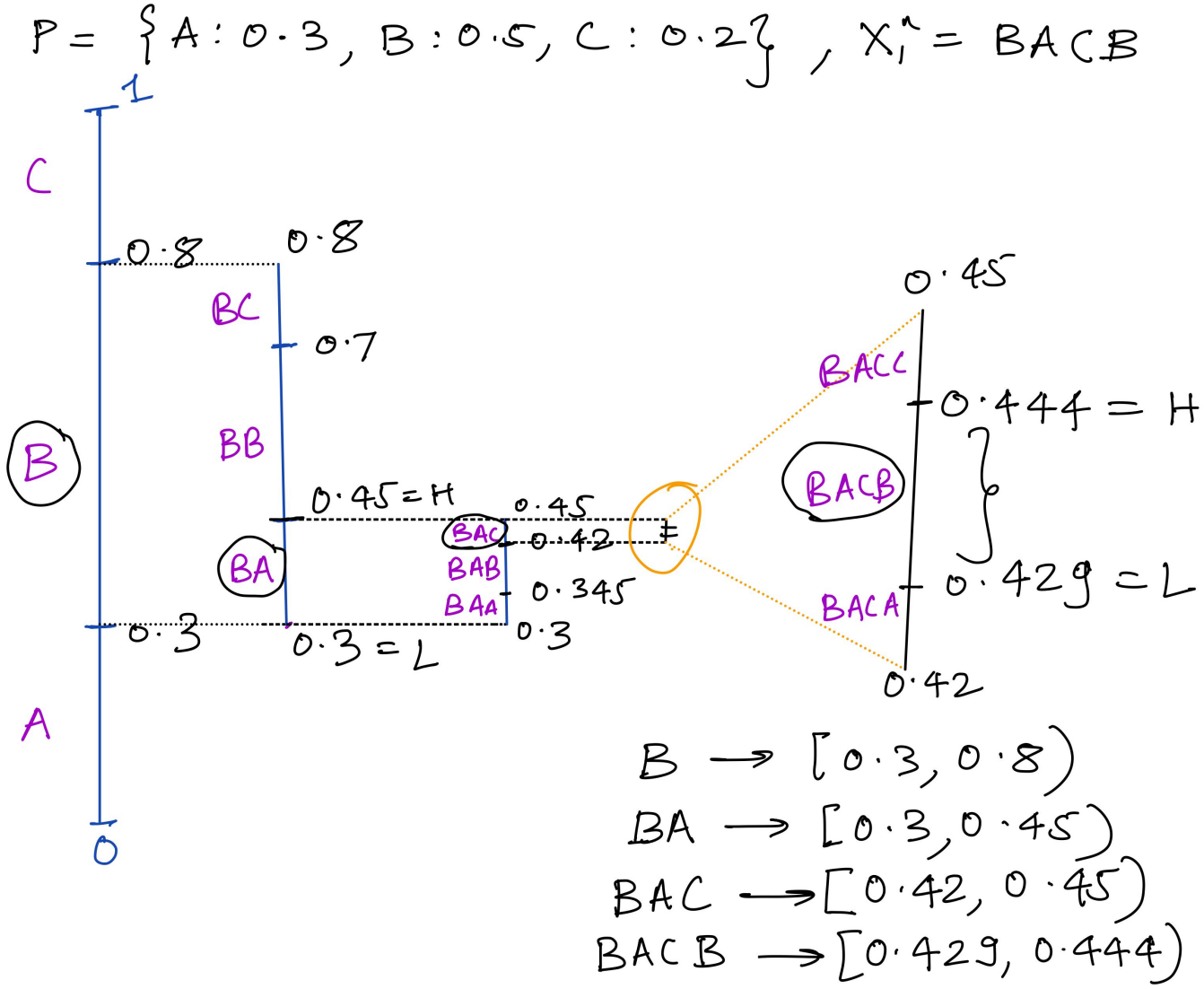


$B \rightarrow [0.3, 0.8)$

$BA \rightarrow [0.3, 0.45)$

$BAC \rightarrow [0.42, 0.45)$

Arithmetic coding example



Arithmetic coding example

1. **STEP I**: Find an *interval* (or a *range*) `[L,H)`, corresponding to the *entire sequence* x_1^n

```
prob = ProbabilityDist({A: 0.3, B: 0.5, C: 0.2})
x_input = BACB

# find interval corresp to BACB
ENCODE: B -> [L,H) = [0.30000,0.80000)
ENCODE: A -> [L,H) = [0.30000,0.45000)
ENCODE: C -> [L,H) = [0.42000,0.45000)
ENCODE: B -> [L,H) = [0.42900,0.44400)
```

Thus, the final interval is: `x_input -> [0.429, 0.444)`

Arithmetic coding pseudo-code

```
class ArithmeticEncoder:
    ...

    def shrink_range(self, L, H, s):
        rng = H - L
        new_L = L + (rng * self.P.cumul[s])
        new_H = new_L + (rng * self.P.probs(s))
        return new_L, new_H

    def find_interval(self, x_input):
        L,H = 0.0, 1.0
        for s in x_input:
            L,H = self.shrink_range(L,H,s)
        return L,H

    def encode_block(self, x_input):
        # STEP1
        L,H = self.find_interval(x_input)

        # STEP-II
        ...
```

STEP-I: Find the interval [L,H)

1. **Observation**: Interval size reduces as we encode more symbols
2. **QUIZ-1**: What is the size of the interval (`H–L`) for the input X_1^n?

STEP-I: Find the interval [L,H)

1. **Observation**: Interval size reduces as we encode more symbols
2. **QUIZ-1**: What is the size of the interval (`H-L`) for the input X_1^n?

$$
\begin{aligned}
(H - L) &= p(x_1) * p(x_2) \ldots p(x_n) \\
&= \prod_{i=1}^{n} p(x_i) \\
&= p(x_1^n)
\end{aligned}
$$

Arithmetic Encoding

```
P = {A: 0.3, B: 0.5, C: 0.2}
x_input = BACB
L = [0.429, 0.444)
```

1. **STEP-I**: Find an *interval* (or a *range*) `[L,H)`
   corresponding to the *entire sequence* x_1^n
2. **STEP-II**: Communicate the interval $[L, H)$ using a value $Z \in [L, H)$

For example: $Z = \frac{(L+H)}{2}$, i.e. the midpoint of the range.
(in our example `Z = 0.4365`)

Arithmetic encoding

https://tinyurl.com/ee274-aec -> Demo time!

Arithmetic decoding

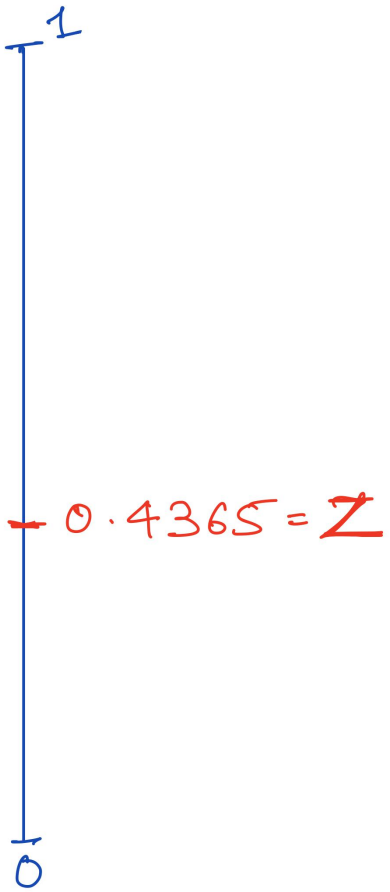
Quiz-2: If the decoder knows:

- `n=4`
- `P = {A: 0.3, B: 0.5, C: 0.2}`
- `Z = 0.4365`

How can it decode the entire input sequence? X_1^n.

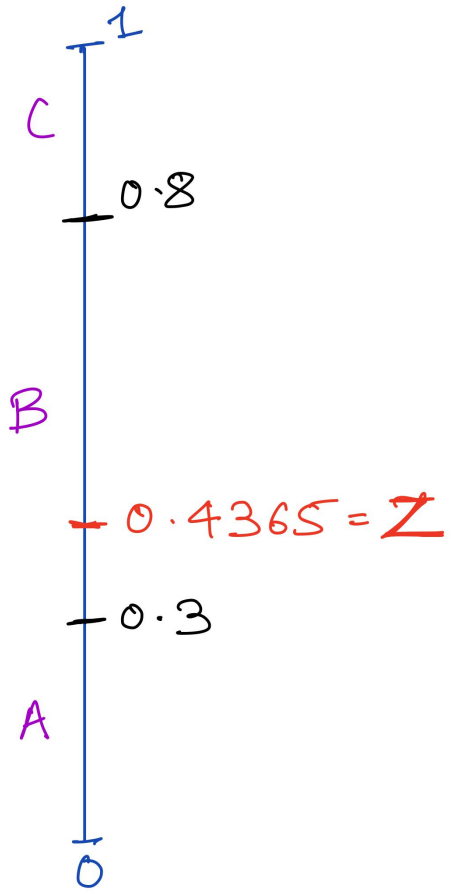
Arithmetic decoding - example

$P = \{A : 0.3, B : 0.5, C : 0.2\}$, $Z = 0.4365$, $n = 4$



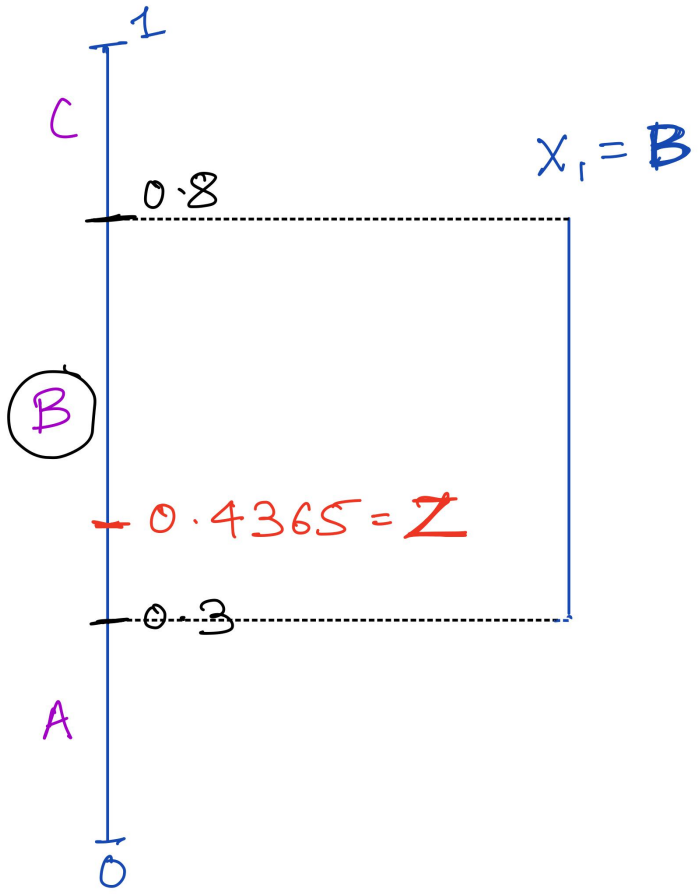
Arithmetic decoding - example

$P = \{A : 0.3, B : 0.5, C : 0.2\}$, $Z = 0.4365$, $n = 4$



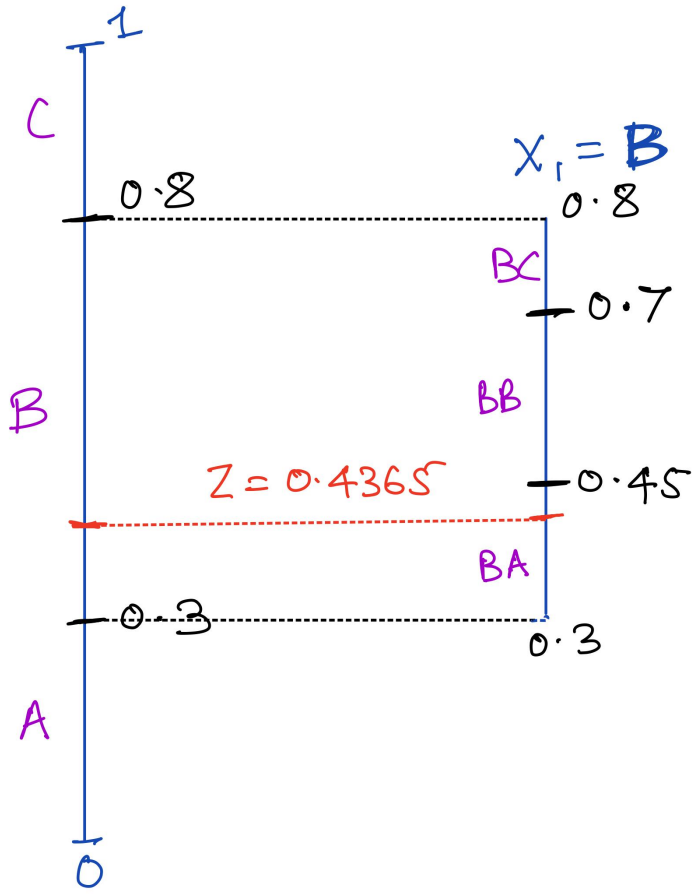
Arithmetic decoding - example

$P = \{A : 0.3, B : 0.5, C : 0.2\}$, $Z = 0.4365$, $n = 4$



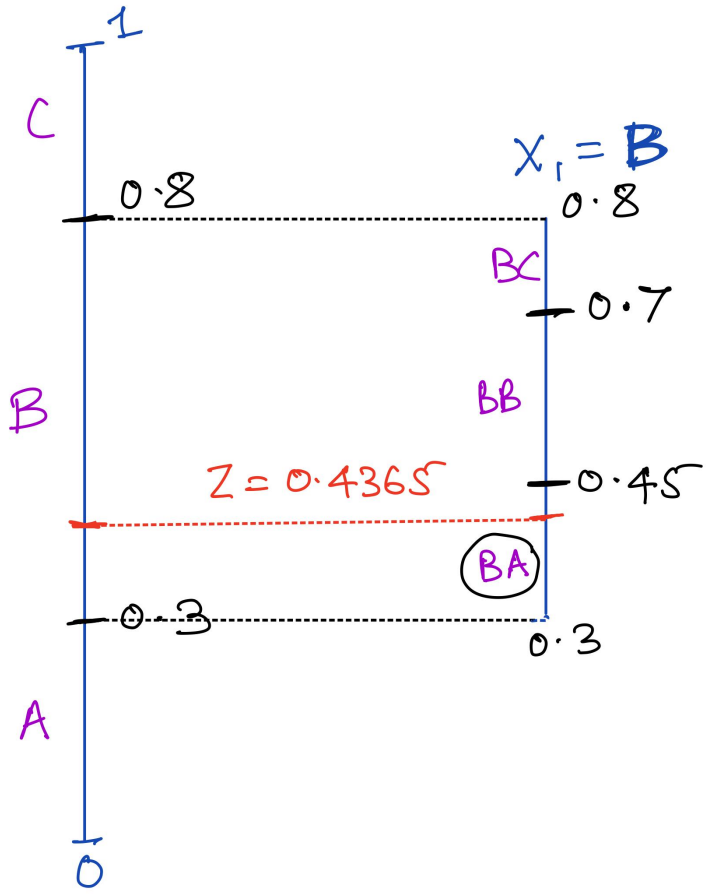
Arithmetic decoding - example

$P = \{A : 0.3, B : 0.5, C : 0.2\}$, $Z = 0.4365$, $n = 4$

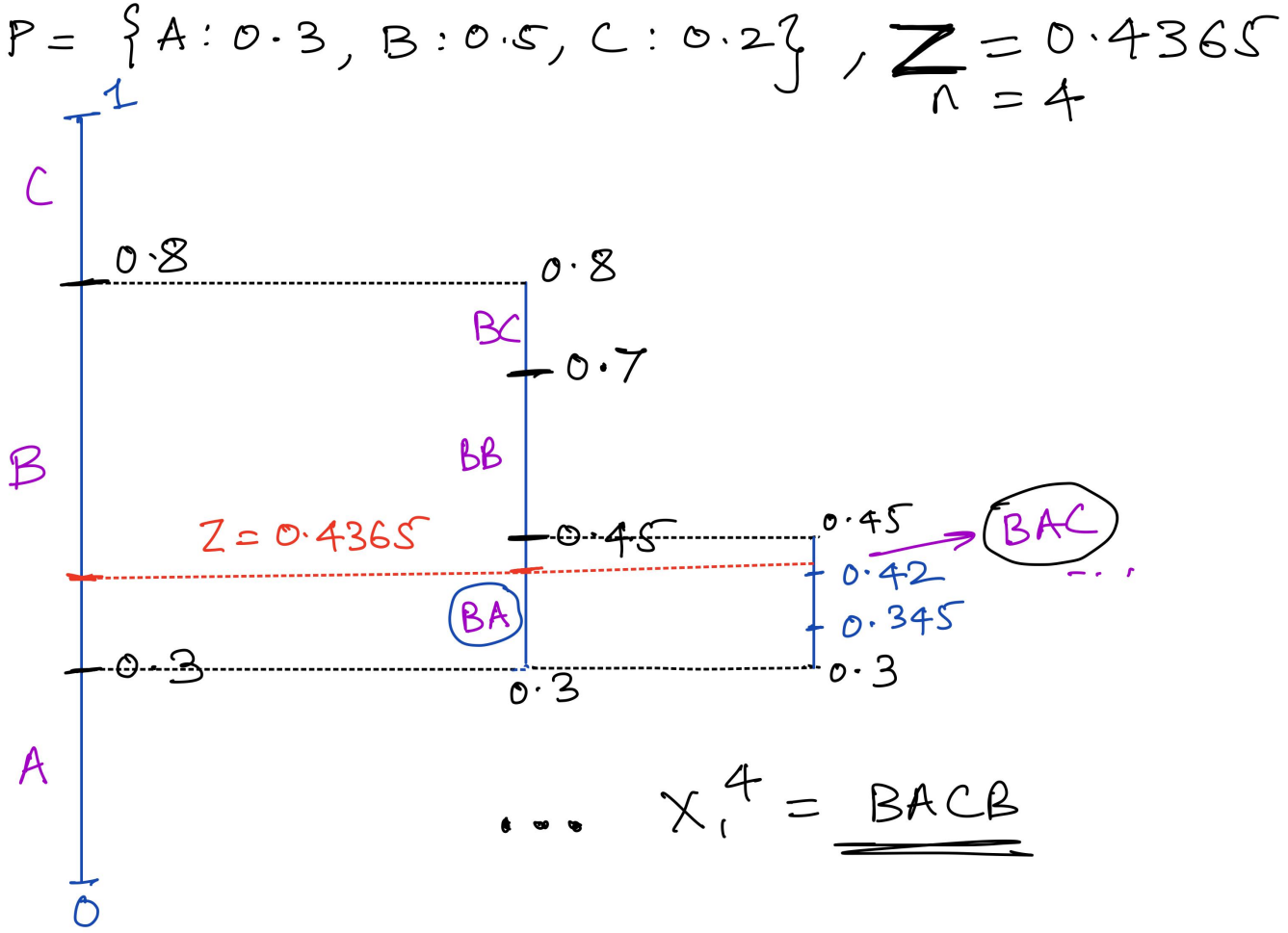


Arithmetic decoding - example

$P = \{A : 0.3, B : 0.5, C : 0.2\}$, $Z = 0.4365$, $n = 4$



Arithmetic decoding - example



Arithmetic decoding

```
Z = 0.4365
ENCODE: B -> [L,H) = [0.30000,0.80000)
ENCODE: A -> [L,H) = [0.30000,0.45000)
ENCODE: C -> [L,H) = [0.42000,0.45000)
ENCODE: B -> [L,H) = [0.42900,0.44400)
---------------------------
DECODE: B -> [L,H) = [0.30000,0.80000)
DECODE: A -> [L,H) = [0.30000,0.45000)
DECODE: C -> [L,H) = [0.42000,0.45000)
DECODE: B -> [L,H) = [0.42900,0.44400)
```

Arithmetic decoding-pseudocode

```
class ArithmeticDecoder:
    ...
    def shrink_range(self, L, H, s):
        ...
        return new_L, new_H

    def decode_symbol(self, L, H, Z):
        rng = H - L
        search_list = L + (self.P.cumul * rng)
        symbol_ind = np.searchsorted(search_list, Z)
        return self.P.alphabet[symbol_ind]

    def decode_block(self, Z, n):
        L,H = 0.0, 1.0
        for _ in range(n): #main decoding loop
            s = self.decode_symbol(L, H, Z)
            L,H = self.shrink_range(L,H,s)
```

Arithmetic decoding:

https://tinyurl.com/ee274-aec -> Demo time!

Arithmetic encoding

1. **STEP-I**: Find an *interval* (or a *range*) `[L,H)`
   corresponding to the *entire sequence* x_1^n (`[0.429, 0.444]`)
2. **STEP-II**: Find the midpoint of the interval $[L, H), Z = \frac{(L+H)}{2}$. (`Z =0.4365`)
3. **STEP-III:** Write the binary expansion of Z to the bitstream ->
   eg: `Z = 0.4365 = b0.01101111101...`
   then the final `encoded_bitstream = 01101111101...`

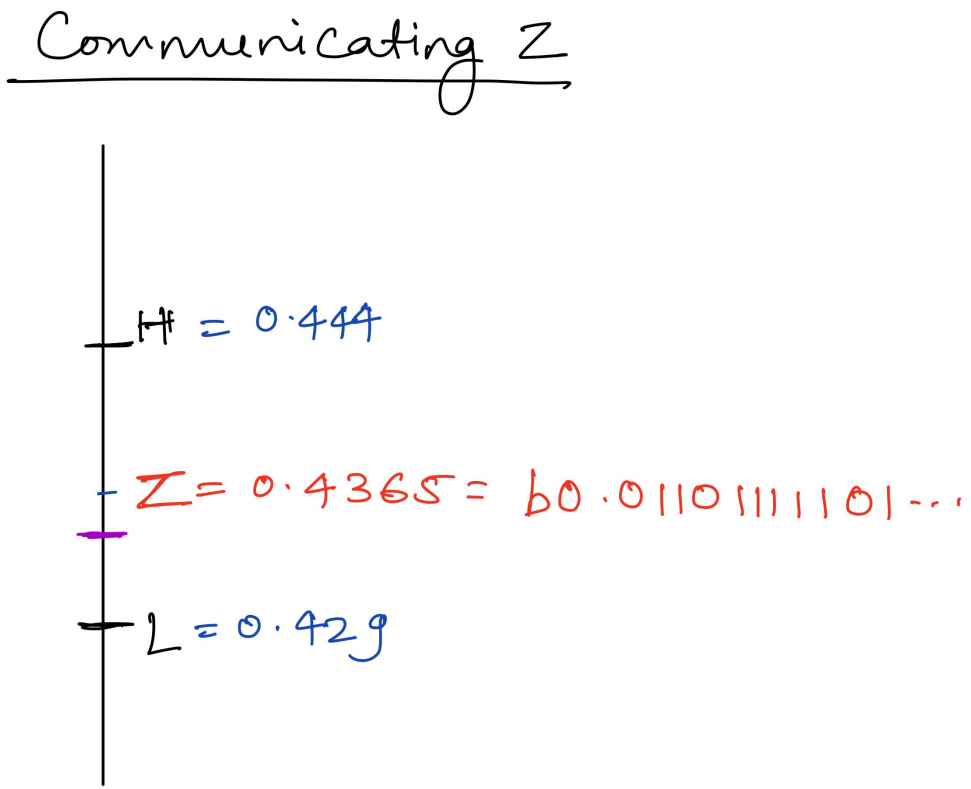
Arithmetic encoding

1. **STEP-I**: Find an *interval* (or a *range*) `[L,H)`
   corresponding to the *entire sequence* x_1^n (`[0.429, 0.444]`)
2. **STEP-II**: Find the midpoint of the interval $[L, H), Z = \frac{(L+H)}{2}$. (`Z =0.4365`)
3. **STEP-III:** Write the binary expansion of Z to the bitstream ->
   eg: `Z = 0.4365 = b0.01101111101...`
   then the final `encoded_bitstream = 01101111101...`

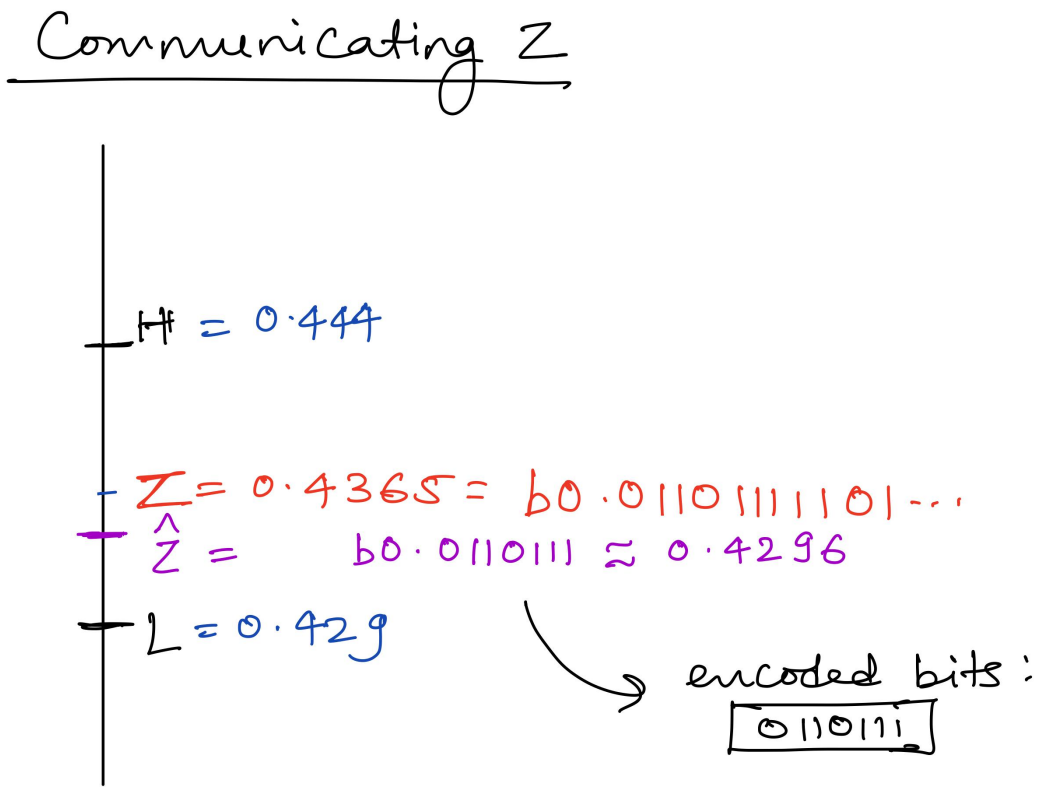
Question: Z's binary representation can be long, can also have infinite bits.

How can we fix this?

Communicating the interval [L, H)



Communicating the interval $[L, H)$



We just need a $\hat{Z}$ such the $\hat{Z} \in [L, H)$
$\hat{Z}$ should have a short binary representation.

Arithmetic coding example:

1. **STEP-I**: Find an *interval* (or a *range*) $[L, H)$
   corresponding to the *entire sequence* x_1^n (`[0.429, 0.444]`)
2. **STEP-II**: Find the midpoint of the interval $[L, H), Z = \frac{(L+H)}{2}$. (`Z =0.4365`)
3. **STEP-III:** Truncate Z to k bits ($\hat{Z}$)
   e.g:

```
L,H = 0.429, 0.444
Z = 0.4365 = b0.01101111101...
Z_hat = b0.011011111 ~ 0.4296
```

Final Encoding = `encoded_bitstream = 011011111`

Communicating the interval `[L, H)`

1. **Cond 1:** Truncate Z to $\hat{Z}$ with k bits, so that $\hat{Z} \in [L, H)$
2. **Cond 2:** If $\hat{Z}$ has binary representation: `Z_hat = b0.011011111` for example, then we also need, any extension of it $Z_{ext} \in [L, H)$.

For eg:

```
Z_hat = b0.011011111
Z_ext = b0.01101111111011110101..
```

Question: Why so?

Communicating the interval `[L, H)`

1. **Cond 1:** Truncate Z to $\hat{Z}$ with k bits, so that $\hat{Z} \in [L, H)$
2. **Cond 2:** If $\hat{Z}$ has binary representation: `Z_hat = b0.011011111` for example, then we also need, any extension of it $Z_{ext} \in [L, H)$.

The two conditions can be written together as:

$$[\hat{Z}, \hat{Z} + 2^{-k}) \in [L, H)$$

Communicating the interval `[L, H)`

Given the interval $[L, H)$, and $Z = \frac{(L+H)}{2}$, truncate Z to k bits so that:

$$[\hat{Z}, \hat{Z} + 2^{-k}) \in [L, H)$$

Question: What should the k be?

```
## Examples
# Ex1: L=0.429, H=0.444, Z = 0.4365.. how many digits we can truncate from Z?

## Ex2: L=0.552398714,H=0.5524123
Z = 0.5524058..., how many digits we can truncate Z from?
```

Communicating the interval `[L, H)`

Given the interval $[L, H)$, and $Z = \frac{(L+H)}{2}$, truncate Z to k bits so that:

$$[\hat{Z}, \hat{Z} + 2^{-k}) \in [L, H)$$

Question: What should the k be?

- Shorter the interval, $|H - L|$, the more the number of bits we need to use.

Communicating the interval `[L, H)`

Given the interval $[L, H)$, and $Z = \frac{(L+H)}{2}$, truncate Z to k bits so that:

$$[\hat{Z}, \hat{Z} + 2^{-k}) \in [L, H)$$

Question: What should the k be?

- Shorter the interval, $|H - L|$, the more the number of bits we need to use.
- the numbers of bits we need to truncate Z by is:

$$k \leq \left\lceil log_2 \frac{1}{(H - L)} \right\rceil + 1$$

Arithmetic Encoding pseudo-code

```
class ArithmeticEncoder:
    def shrink_range(self, L, H, s):
        ...
    def find_interval(self, x_input):
        L,H = 0.0, 1.0
        for s in x_input:
            L,H = self.shrink_range(L,H,s)
        return L,H

    def encode_block(self, x_input):
        # STEP-1 find interval
        L,H = self.find_interval(x_input)

        # STEP-II,III communicate interval
        Z = (L+H)/2
        num_bits = ceil(log2((H-L))) + 1
        _, code = float_to_bitarray(Z, num_bits)
        return code
```

Arithmetic decoding-pseudocode

```
class ArithmeticDecoder:
    ...
    def shrink_range(self, L, H, s):
        ...

    def decode_symbol(self, L, H, Z):
        ...

    def decode_block(self, code, n):
        Z = bitarray_to_float(code)

        # start decoding
        L,H = 0.0, 1.0
        for _ in range(n): #main decoding loop
            s = self.decode_symbol(L, H, Z)
            L,H = self.shrink_range(L,H,s)

        # add code to remove additional bits read
```

Arithmetic coding compression performance:

- Size of interval $|H - L| = \log_2 \frac{1}{p(x_1^n)}$
- $k \leq \log_2 \frac{1}{H-L} + 2$

Question: What is the codelength for arithmetic coding?

Arithmetic coding compression performance:

THEOREM: Arithmetic coding achieves average codelength:

$$H(X) \leq \frac{\mathbb{E}[l(X_1^n)]}{B} \leq H(X) + \frac{2}{n}$$

Arithmetic coding Summary

1. Given *any* distribution P, achieves *optimal* compression. Thus, Arithmetic coding allows for `model` and `entropy coding` separation.
2. Encoding, decoding is linear time and quite efficient!
3. As we are not saving a large codebook, memory requirements are not very high
4. Can work very well with changing distribution P.
   i.e. Adaptive algorithms work well with Arithmetic coding

Arithmetic coding in practice

What are the practical issues with Arithmetic coding?

https://tinyurl.com/ee274-aec -> Demo time!

Arithmetic coding in practice

Quiz-8: Whar are the practical issues with our Arithmetic encoding/decoding?

Ans -> The interval becomes too small very quickly

and we run out of bits to represent `L,H` .

```
prob = ProbabilityDist({A: 0.3, B: 0.5, C: 0.2})
x_input = BACBBCCBA

# find interval corresp to BACB
ENCODE: B -> [L,H) = [0.30000,0.80000)
ENCODE: A -> [L,H) = [0.30000,0.45000)
ENCODE: C -> [L,H) = [0.42000,0.45000)
ENCODE: B -> [L,H) = [0.42900,0.44400)
...
```

Arithmetic coding in practice

Quiz-9: What can we do to avoid the interval `[L,H)` from getting too small?

Hint ->

```
L = 0.429 = b0.0110110...
H = 0.444 = b0.01110001...
```

Arithmetic coding in practice

Quiz-9: What can we do to avoid the interval `[L,H)` from getting too small?

Idea: If `L` , `H` start with `011` then any value lying inside the interval `[L,H)` also will start with `011` !

```
L = 0.429 = b0.0110110...
H = 0.444 = b0.01110001...
Z_hat = b0.011...
```

Arithmetic coding in practice

Quiz-9: What can we do to avoid the interval `[L,H)` from getting too small?

Idea: If `L`, `H` start with `011` then any value lying inside the interval `[L,H)` also will start with `011`!

Rescale:: Already output bits `011`, and rescale `L,H`

```
L = 0.429 = b0.0110110...
H = 0.444 = b0.01110001...

Rescaled: L=0.8580, H=0.8880, bitarray='0'
Rescaled: L=0.7160, H=0.7760, bitarray='01'
Rescaled: L=0.4320, H=0.5520, bitarray='011'
ENCODE: B -> [L,H) = [0.42900,0.44400)
```

Arithmetic Encoding with rescaling

```
class ArithmeticEncoder:
    def shrink_range(self, L, H, s):
        ...
    def rescale_range(self, L, H):
        ...
    def find_interval(self, x_input):
        L,H, bitarray = 0.0, 1.0, Bitarray("")
        for s in x_input:
            L,H = self.shrink_range(L,H,s)
            L,H, bits = self.rescale_range(L,H)
            bitarray += bits
        return L,H, bitarray
```

Arithmetic Encoding with rescaling

```
def rescale_range(self, L, H):
    bitarray = ""
    while (L >= 0.5) or (H < 0.5):
        if (L < 0.5) and (H < 0.5):
            bitarray+= "0"
            L,H = L*2, H*2
        elif ((L >= 0.5) and (H >= 0.5)):
            bitarray += "1"
            L,H = (L - 0.5)*2, (H - 0.5)*2
    return L, H, bitarray
```

Arithmetic Encoding with rescaling

Rescale:: Already output bits which are same between L, H (`011`), and rescale L, H.

```
L = 0.429 = b0.0110110...
H = 0.444 = b0.01110001...

Rescaled: L=0.8580=b0.11011..., H=0.8880, bitarray='0'
Rescaled: L=0.7160=b0.1011... , H=0.7760, bitarray='01'
Rescaled: L=0.4320=b0.011...  , H=0.5520, bitarray='011'
ENCODE: B -> [L,H) = [0.42900,0.44400)
```

Question: There is one case in which our algorithm can still have L,H being really close. What is that?

Arithmetic Encoding with rescaling

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 coder
2. **Range Coding** Byte (8-bit based rescaling), word-based rescaling
   SCL range coder
3. Variants on the above based on how compressors handle the edge case (L starts with `b0.0` and H starts with `b0.1..`, but the interval is very small)

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 etc.

Arithmetic coding in video compression (265):

https://datatracker.ietf.org/doc/html/rfc7798

https://datatracker.ietf.org/doc/draft-ietf-avtcore-rtp-vvc/02/

Question: What is CABAC? (context adaptive ...)?

What are the problems with Arithmetic coding

Although Arithmetic coding algorithms are fast, 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

Beyond Arithmetic 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
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

- `rANS` : *range-ANS*: drop-in replacement for Arithmetic coding, but faster
- `tANS` : *table-ANS*: 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`?

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=32415`, 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

```
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]))`.
(eg. `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*

```
## 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):
   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?

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`?

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)
```

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?

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: Flush out a few bits intermittently, and ensure that the state x remains within a range!

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
```

rANS practical considerations:

- **Streaming**: Like in Arithmetic coding case, is necessary to encode long inputs
- **Speed:**: Caching rANS -> tANS, Alias-rANS etc.

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

Alias rANS

Question: Where do we spend most of the time in rANS decoding?

```
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)
```

Alias rANS

Question: Where do we spend most of the time in rANS decoding?

```
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)
```

Ans: The `find_bin(cumul, slot)` function takes $O(log|k|)$
(binary search of `slot` over `cumul` sorted array)

Alias rANS

Question: `find_bin(cumul, slot)` is going to be $O(log|k|)$, as it is difficult to improve upon binary search. Can we "change" the problem?

Example:

freq -> [3,3,4,5,12,3,2], M = 8, cumul -> [0,3,6,10,15,27,30,32]

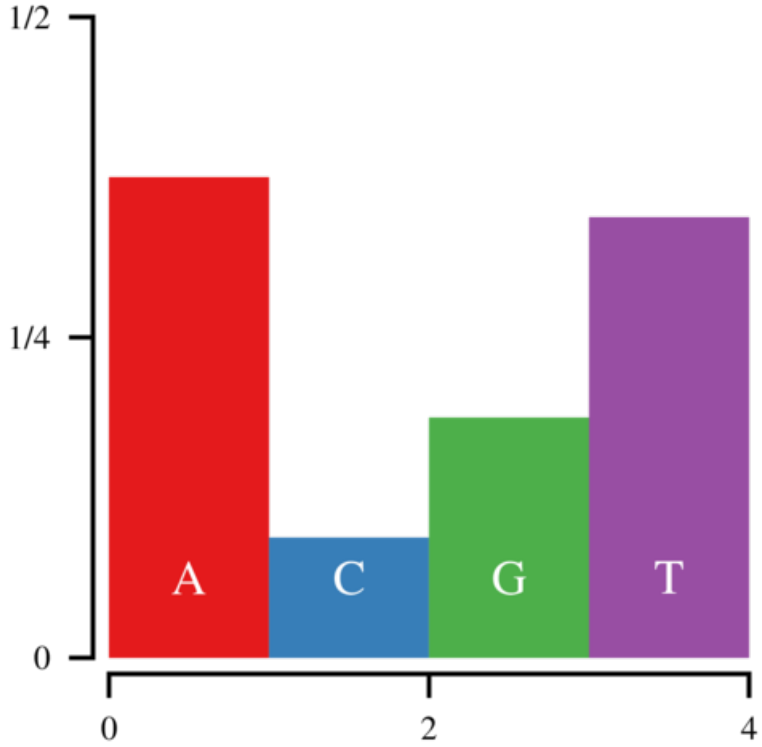
Alias rANS

Question: `find_bin(cumul, slot)` is going to be $O(log|k|)$, as it is difficult to improve upon binary search. Can we "change" the problem?

Ans: `cumul` array assigns contiguous regions for each of the symbols. But do we need to?

Aside: sampling!

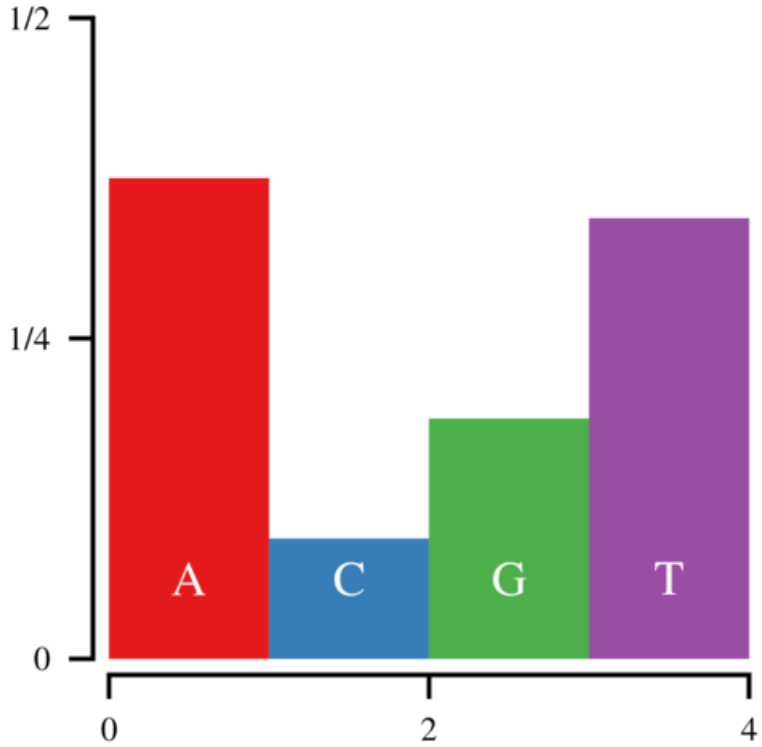
Question: How would you go about sampling from a distribution?



https://pandasthumb.org/archives/2012/08/lab-notes-the-a.html

Aside: sampling!

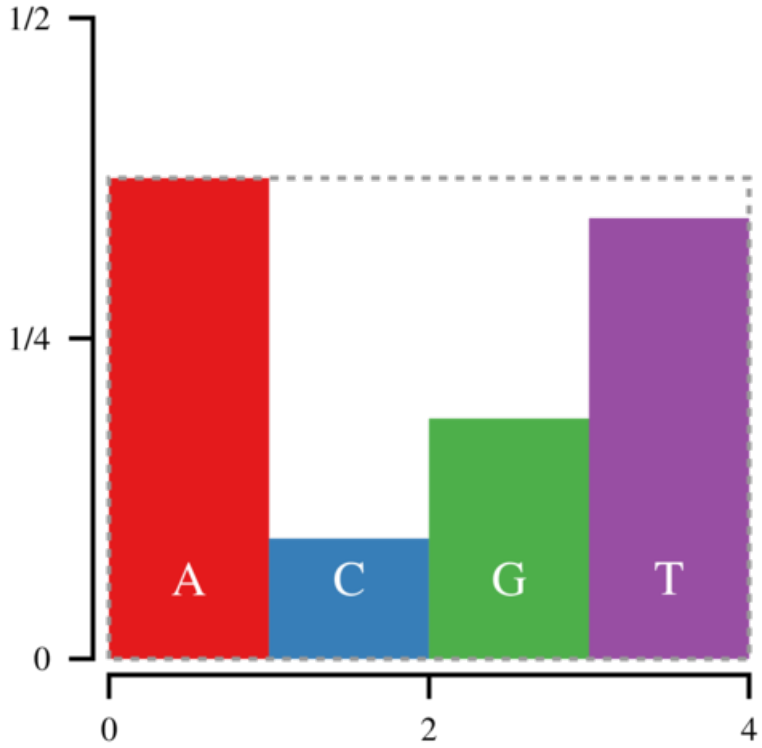
Question: How would you go about sampling from a distribution?



Idea-1: Binary search over unit line

Aside: sampling!

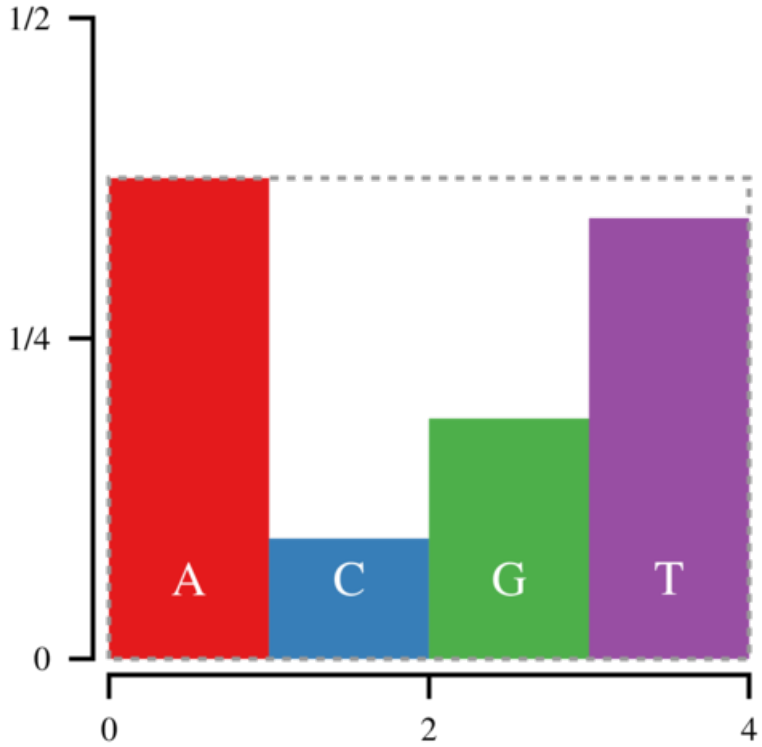
Question: How would you go about sampling from a distribution?



Idea-2: Rejection sampling! (if sample falls within the bars, output the symbol, else repeat)

Aside: sampling!

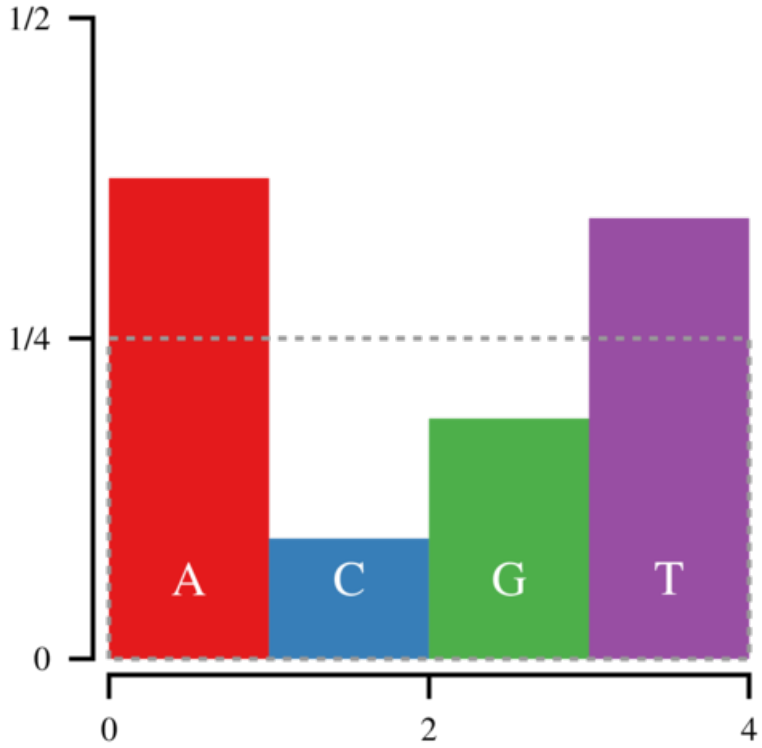
Question: How would you go about sampling from a distribution?



Idea-2: Rejection sampling! very wasteful!

Aside: sampling!

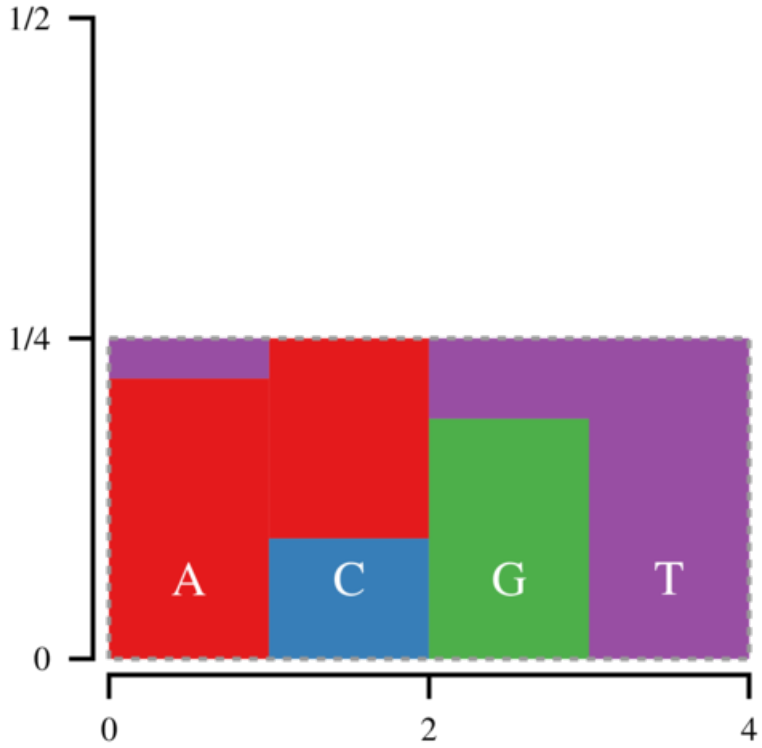
Question: How would you go about sampling from a distribution?



Idea-2: Rejection sampling! very wasteful!

Aside: sampling!

Question: How would you go about sampling from a distribution?



Idea-3: Re-arrange the "bars/buckets" to fill in a rectangle.

Alias method for sampling

- rearrange probability mass to be in a 2D rectangle
- *Voss method*: Possible to ensure that each vertical bucket contains mass from at max two symbols (primary, alias).
- **Question**: How can we use this for speeding up rANS?

rANS vs rANS-alias decoding

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

```
def rans_alias_base_decode_step(x):
    block_id, slot = x//M, x%M # Step I
    ###############
    bucket_id = slot/K # bit-shift if K is pow-2
    if slot > divider[bucket_id]:
        s = bucket_id
    else:
        s = alias_symbol[bucket_id]
    ...
   return (s,x_prev)
```

rANS vs rANS-alias decoding

```
def rans_alias_base_decode_step(x):
    block_id, slot = x//M, x%M # Step I
    ###############
    bucket_id = slot/K # bit-shift if K is pow-2
    if slot > divider[bucket_id]:
        s = bucket_id
    else:
        s = alias_symbol[bucket_id]
    ...
   return (s,x_prev)
```

Question: How should the encoding speed change due to this?

Thank you!