Arithmetic coding with integer representation

1. Representation of numbers

Let's consider a fixed point representation of numbers in [0-1) interval

`0.bbbbbbbbbbbb...`

We can consider `1 = 0.11111111111111....` (an infinite number of 1)

We use a finite length representation by keeping (in an integer variable) only the first 32 bits.

`0.` `bbbbbbbbb...bbb` `bbbbbbbbbbbb...`
keep first 32 bits — drop the rest

We use in the arithmetic coder:

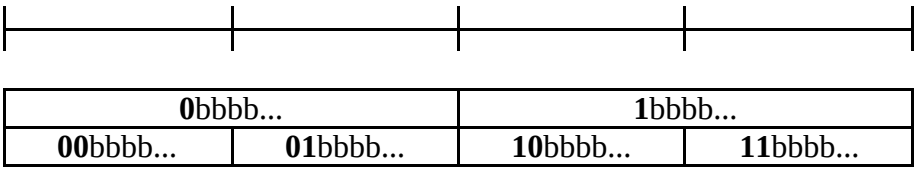
- Low – unsigned integer value on 32 bits
- High – unsigned integer value on 32 bits
- Range – unsigned integer value on 64 bits

We initialize:

High = `0xFFFFFFFF` and keep in mind that it is followed by an infinite number of 1's
Low = `0x00000000` and keep in mind that it is followed by an infinite number of 0's

In that case:

- the first bit of the representation describes the half of the [0-1) interval where the value lies
- the first 2 bits of the representation describes the quarter of the [0-1) interval where the value lies:



2. Shiftings to be done

in order to keep Range big enough.

2.1. First shifting: when both high and low are **in the same half**

before:

`High 0`$h_1h_2h_3h_4...h_{31}$ or `High 1`$h_1h_2h_3h_4...h_{31}$
`Low 0`$l_1l_2l_3l_4...l_{31}$ `Low 1`$l_1l_2l_3l_4...l_{31}$

action:

send **0** to output, shift left and add a bit — send **1** to output, shift left and add a bit

after:

`High` $h_1h_2h_3h_4...h_{31}$`1` or `High` $h_1h_2h_3h_4...h_{31}$`1`
`Low` $l_1l_2l_3l_4...l_{31}$`0` `Low` $l_1l_2l_3l_4...l_{31}$`0`

2.2. <u>Second shifting (underflow)</u>: when high and low are in 2'nd and 3'th quarters

before:
High **10**$h_1h_2h_3h_4...h_{30}$
Low **01**$l_1l_2l_3l_4...l_{30}$

action:
extract the second bit, increment a counter, shift 1 bit left the last 30 bits and add a bit

after:
High **1**$h_1h_2h_3h_4...h_{30}$**1**
Low **0**$l_1l_2l_3l_4...l_{30}$**0**

The exact value of the extracted bit will be known only at the next first shifting, when the corresponding number of bits (according to the counter) will be written to output.

3. Representation of MODEL

The simplest representation could be one based on 2 vectors: `counts[]` and `sums[]` indexed by the symbol.

An example...

symbol	**0**	**1**	**2**	**3**	**...**	**NrSymb-1**	**NrSymb**
counts[]	2	5	3	2	...	...	
sums[]	0	2	7	10	...	...	S

`count[]` = the count for each symbol (frequency of occurrence of the symbol).
`sums[]` = the sum of counts of symbols **before** the symbol.
`S` = the sum of counts for all the symbols.

The range for a symbol is now given by:
- for symbol 2 (by example)

```
7 / S – 10 / S
```

- generally

```
sums[symbol]/sums[NrSymb] – sums[symbol+1]/sums[symbol]
```

The standard floating point arithmetic coder updating formulas:

```
High = Low + Range * LimitH(symbol)
Low  = Low + Range * LimitL(symbol)
```

become:

```
High = Low + Range * sums[symbol+1]/sums[NrSymb]-1
Low  = Low + Range * sums[symbol  ]/sums[NrSymb]
```

4. Closing the output stream

In theory, at the end a value (any) from the range [Low-High) should be send to the decoder. The simplest case is to send the whole 32 bits representation of Low.

We can save 30 bits by sending only a 2 bit value corresponding to the left margin of the quarter that lies between Low and High (we have 3 cases described in the next figure). The decoder **must provide 0's** after the input stream ends (to complete the 32 bit required value).

Case	First quarter	Second quarter	Third quarter	Forth quarter
	00...	01...	10...	11...
1	**Low**	**Send 01**	**High**	
2		**Low**	**Send 10**	**High**
3	**Low**	**Send 01 or 10**		**High**

Implementation requirements:

- High, Low on **32 bits**.
- The number of symbols should be **configurable** (at least by a constant value). Adding an **EOF** symbol can be useful to inform the decoder about the end of the stream.
- The model should change after each symbol (i.e. we have to implement a **dynamic** approach).
- Fixed initialization of the model with **all counts 1 is acceptable**/enough. (You can try also to use a precomputed initialization).
- Encapsulate the coder/model in a class in order to easy implement later a **contextual approach** with different coders for different contexts in a data stream.
- Use **only bitwise** operations (shifting, logical) not arithmetic operations (multiplication, division).

Implementation advices:

- Start with a static approach until it works and only after that put code in an Update method.
- If you need debugging choose a small number of symbols in the alphabet.