The missing step of arithmetic encoding is to connect source alphabet intervals with code alphabet intervals. There are a few different ways to do this. Our solution is to map a source interval $[l, r)$ into the largest dyadic interval that is completely contained in $[l, r)$.

Now we have the full encoding procedure:

source string $\mapsto$ source interval $\mapsto$ code interval $\mapsto$ code string

Example 1.32: The string `cab` is encoded as 101110:

`cab` $\mapsto [.712, .742) \mapsto [.71875, .734375) = [.101110, .101111) \mapsto 101110$

The mapping from source intervals to code intervals ensures that the code intervals do not overlap. Therefore, we have a prefix code for Σ^n.

The prefix code is usually not optimal or even complete, because there are gaps between the code intervals. However, the length of the code interval for a source interval of length p is always more than $p/4$. Thus the code length for a string X is less than $-\log P_n(X) + 2$, and the average code length is less than $H(P_n) + 2$.

Since
$$\begin{aligned} H(P_n) &= -\sum_{X\in\Sigma^n} P_n(X)\log P_n(X) \\ &= -\sum_{X=x_1\dots x_n\in\Sigma^n} P_n(X)\big(\log P(x_1) + \cdots + \log P(x_n)\big) \\ &= -\sum_{i=1}^{n}\sum_{X=x_1\dots x_n\in\Sigma^n} P_n(X)\log P(x_i) \\ &= -\sum_{i=1}^{n}\sum_{s\in\Sigma}\sum_{X=x_1\dots x_n\in\Sigma^n, x_i=s} P_n(X)\log P(s) \\ &= -\sum_{i=1}^{n}\sum_{s\in\Sigma} P(s)\log P(s) \;=\; nH(P), \end{aligned}$$
the average code length per symbol is less than $H(P) + 2/n$. For a large n, this is very close to the entropy.

In principle, an even better code could be obtained by applying the Huffman algorithm on the probability distribution P_n. However, the large size of the set Σ^n makes this approach infeasible except for very small n. Because of the incremental computation of the intervals, arithmetic coding is not hampered by this problem.

A common description of arithmetic coding in the literature maps the source interval to any number in that interval and uses the binary representation of the number as the code. However, this is not necessarily a prefix code for Σ^n, and one has to be careful about how to end the code sequence.

Example 1.33: For strings of length 1 in our running example, we might obtain the following codes:

source string	`a`	`b`	`c`
source interval	$[0, 0.2)$	$[0.2, 0.7)$	$[0.7, 1.0)$
code number	0	$.5_{10} = .1_2$	$.75_{10} = .11_2$
code string	0	1	11

There are two ways of turning the code for Σ^n into a code for Σ^*:

1. Prepend $\gamma(n)$ or another encoding of n to the code string.
2. Add a special EOM (End-Of-Message) symbol to the alphabet (with a very small probability), and finish the source string with the EOM symbol.

A problematic issue with arithmetic coding is the potentially high precision that may be needed in the interval computations. The number of digits required is proportional to the length of the final code string.

Practical implementations avoid high precision numbers using two techniques.

- *Rouding:* Replace high precision numbers with low precision approximations. As long as the final intervals for two distinct strings can never overlap, the coding still works correctly. Rounding may increase the average code length, though.
- *Scaling:* Construct the source interval and the code interval in parallel. When both intervals get small enough, we can scale the numbers, essentially "zooming in". This way the intervals never get extremely small.

Practical implementations commonly use *integer arithmetic* both to improve performance and to have full control of precision and the details of rounding.

Let us look at a complete example with full details.

We use integer arithmetic with integers $[0, 64]$ representing the unit interval $[0, 1.0]$. For example, 13 represents $13/64 = 0.203125$.

Due to limited precision, the interval sizes are not a perfect representation of the true probabilities. The alphabet intervals for our running example are:

symbol	a	b	c
true probability	0.2	0.5	0.3
self-information	2.32	1.0	1.74
interval	$[0, 13)$	$[13, 45)$	$[45, 64)$
implied probability	0.203125	0.5	0.296875
code length	2.30	1.0	1.75

This may have a tiny effect on the compression rate, but much smaller than using a prefix code would have.

Now let us start encoding the string `cab`:

$[l, r) \leftarrow [0, 64)$ // initialize
$[l', r') \leftarrow [45, 64)$ // interval for symbol `c`
$p \leftarrow 64 - 0 = 64$
$l \leftarrow l + \frac{p \cdot l'}{64} = 0 + \frac{64 \cdot 45}{64} = 45$
$r \leftarrow l + \frac{p \cdot r'}{64} = 0 + \frac{64 \cdot 64}{64} = 64$

When multiplying two probabilities, we have to divide by 64 in order to keep the right scale. In a moment, we will see what happens when the result is not an integer.

The source interval $[45, 64)$ is now completely in the second half of the unit interval $[0, 64)$. This means that all the subintervals including the final code interval are in the second half too. Thus we know that the first bit of the code string is 1.

Now we output 1, i.e., set the first bit of the code string to 1. The corresponding code interval is $[32, 64)$. Then we zoom in the second half by translating and scaling the numbers:

output 1
$l \leftarrow (l - 32) \times 2 = 26$
$r \leftarrow (r - 32) \times 2 = 64$

The code interval becomes $[0, 64)$. This is always the case after scaling, so there is no need to store it.

If the source interval had been completely in the first half, we would have output 0 and multiplied both end points by 2.

Now $[l, r) = [26, 64)$ and we process the next symbol.

$[l', r') \leftarrow [0, 13)$ // interval for symbol `a`
$p \leftarrow 64 - 26 = 38$
$l \leftarrow l + \left\lfloor \frac{p \cdot l'}{64} \right\rfloor = 26 + \left\lfloor \frac{38 \cdot 0}{64} \right\rfloor = 26$
$r \leftarrow l + \left\lfloor \frac{p \cdot r'}{64} \right\rfloor = 26 + \left\lfloor \frac{38 \cdot 13}{64} \right\rfloor = 26 + \lfloor 7.7 \rfloor = 33$

Here we had to do some rounding, because the exact values could not be represented with the limited precision. This can again cause a tiny increase in the average code length.

Other rounding schemes are possible. Fast execution is often important, even at the cost of worse compression, which may result from rounding more than necessary.

When rounding, one has to be careful about not to create an overlap between intervals that should be distinct. Our rounding scheme is OK in this respect. (Why?)

Now $[l, r) = [26, 33)$. This interval is not completely in the first half nor in the second half. However, it is completely in the *"middle" half* $[16, 48)$. Now we do not know what to output yet, but we still zoom in. Instead of outputting, we increment a variable called delayed_bits (initialized to 0):

delayed_bits $\leftarrow$ delayed_bits $+ 1 = 1$
$l \leftarrow (l - 16) \times 2 = 20$
$r \leftarrow (r - 16) \times 2 = 34$

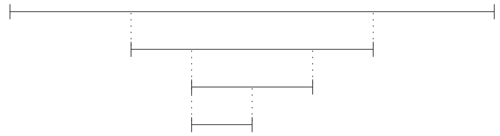
$[20, 34)$ is still in the middle half, so increment delayed_bits and scale again:

delayed_bits $\leftarrow$ delayed_bits $+ 1 = 2$
$l \leftarrow (l - 16) \times 2 = 8$
$r \leftarrow (r - 16) \times 2 = 36$

Zooming in the middle half ensures that the length of the source interval remains larger than one quarter of the unit interval, 16 in this case.

The delayed_bits variable keeps track of how many times we have scaled without outputting anything. The output to produce will be resolved when we zoom in to the first or the second half.

Suppose the next scaling goes left. Then source intervals involved look like this:



The first interval, from before zooming in the middle, was a proper dyadic interval. The next two intervals, resulting from zooming in the middle, are not dyadic due to wrong alignment. Thus we could not determine the next bits. The last interval is dyadic again, and the transition from the first to the last interval corresponds to the bits 011.

The general procedure when zooming in the first half is to output 0 followed by delayed_bits 1's, and then to set delayed_bits to zero.

Similarly, when zooming in the second half, output 1 followed by delayed_bits 0's, and set delayed_bits to zero.

Let us return to the computation. The source interval is $[l, r) = [8, 36)$ and we process the next source symbol.

$[l', r') \leftarrow [13, 45)$ // interval for symbol $\mathtt{b}$
$p \leftarrow 36 - 8 = 28$
$l \leftarrow l + \left\lfloor \frac{p \cdot l'}{64} \right\rfloor = 8 + \left\lfloor \frac{28 \cdot 13}{64} \right\rfloor = 8 + \lfloor 5.7 \rfloor = 13$
$r \leftarrow l + \left\lfloor \frac{p \cdot r'}{64} \right\rfloor = 8 + \left\lfloor \frac{28 \cdot 45}{64} \right\rfloor = 8 + \lfloor 19.7 \rfloor = 27$

$[13, 27)$ is in the first half, so we output and scale:

output 011
delayed_bits $\leftarrow 0$
$l \leftarrow l \times 2 = 26$
$r \leftarrow r \times 2 = 54$

Now there is no more source symbols. The final scaled source interval is $[26, 54)$. The current scaled code interval is $[0, 64)$. We need two more bits:

output 10

Now the code interval is $[32, 48)$, which is completely inside the source interval.

The general rule for the last two bits is:

- If the source interval contains the second quadrant $[16, 32)$, output 01.
- If the source interval contains the third quadrant $[32, 48)$, output 10.

If the source interval does not contain either quadrant, it must be inside the first, middle or last half and we can scale. If the source interval contains both quadrants, either output is OK.

The final output is 101110. This the same we got with exact arithmetic, so in this case the rounding did not change the result.

Decoding arithmetic code performs much the same process, but the process is now controlled by the code string and it produces the source string as output. Here is an outline of the decoding procedure in our example:

- We start with the code interval $[0, 64)$. Each bit read from the code string halves the code interval.

- In the beginning, the source interval is $[0, 64)$ too, and it is divided into three candidate intervals: $[0, 13)$, $[13, 45)$ and $[45, 64)$. Whenever the code interval falls within one of the candidate intervals, that interval becomes the new source interval and we output the corresponding symbol. If the new source interval is within one of the three halves of the unit interval, we zoom in scaling both the source and the code interval. Then the new source interval is divided into three candidate intervals and the process continues.

- The process ends, when n symbols have been output.

The full details are left as an exercise.

Summary

We have seen two encoding algorithms, Huffman coding and arithmetic coding, that perform the same task:

- Turn a sequence of symbols with a given probability distribution into a uniquely decodable binary code sequence.

- Try to minimize the average length of the code sequence.

This task is often known as *entropy encoding* reflecting the fact that entropy is the ideal average code length and the algorithms try to get as close to that as possible.

Huffman coding and arithmetic coding cannot be easily used when the alphabet is large or even infinite. Fixed codes such as γ or Golomb–Rice codes address this problem in the case of integer alphabet.

The Huffman algorithm computes the optimal prefix code. But there are still other algorithms for constructing prefix codes that can sometimes be useful. For example:

- The Shannon-Fano algorithm (Exercise 1.3) is simpler to implement than the Huffman algorithm, and usually produces an optimal or nearly optimal prefix code. See also Exercise 2.1.

- The package-merge algorithm constructs the optimal prefix code under the constraint that the codeword is longer than L. Such a code is sometimes known as a length-limited Huffman code.

- The Hu-Tucker algorithm constructs an optimal alphabetic prefix code, which has the property that the lexicographical order of codewords is the same as the order of the symbols they encode.