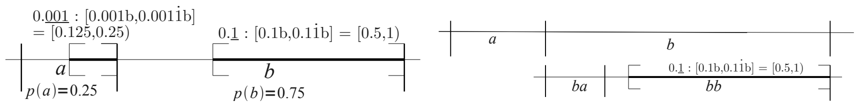

0.001 : [0.001b,0.001i̇b]
= [0.125,0.25)
a
p(a)=0.25
0.1 : [0.1b,0.1i̇b] = [0.5,1)
b
p(b)=0.75
a
b
ba
0.1 : [0.1b,0.1i̇b] = [0.5,1)
bb

(a) Representing a and b by 001 and 1. (b) Representing bb by 1.

Figure 2.4: Representing a symbol using an interval. $[0, 1)$ is partitioned into regions with lengths proportionally to the probabilities of the symbols. Any interval (that in turn can be represented by a number) that lies within the region corresponding to a symbol can be used to generate the codeword of the symbol.

2.3.1 Arithmetic Coding

Targetted to conventional source coding, arithmetic coding is a type of entropy coding, where a symbol is compressed on average to its optimum length $-\log p(x)$. The key idea of arithmetic coding is illustrated in Figure 2.4(a), where the interval $[0, 1)$ is partitioned into intervals with lengths proportionally to the probabilities of the symbols of a source. Let us call each of these intervals the *probability interval* of the corresponding symbol. The main trick is that a symbol can be uniquely represented by a real number within the probability interval of the symbol. Moreover, the longer the probability interval, the less number of bit needed to represent the number.

As in Figure 2.4(a), $0.001b$ (0.125 in decimal representation) lies inside the probability interval of a and thus can be used to represent a uniquely with its bits after the decimal point (i.e., 001). Similarly, b can be identified by the number $0.1b$ and thus can be coded as 1. Note that we are really referring to the interval $[0.a_1a_2\cdots a_n, 0.a_1a_2\cdots a_n\dot{1}]$ when we consider the real number $0.a_1a_2\cdots a_n$. Therefore, it is important to require this region to lie completely inside the probability interval of the corresponding symbol.

Apparently, as the probability interval gets shorter, a real number with higher precision is needed to represent the interval. It is easy to verify that it needs approximately $-\log p(x)$ bits to represent an probability interval of x. The length of the codeword does approximate to the optimum code length.

We can easily see why this coding approach allows unique decoding. By construction, the regions that identifies any two symbols cannot overlap since both regions lie completely inside the non-overlapping probability intervals of the two symbols. As an immediate consequence, the following lemma provides a sufficient condition for unique decoding.

Lemma 2.2 (Prefix Condition)**.** *Any codeword of the code that constructed above cannot be a prefix of another codeword.*

Proof. Assuming that the Lemma is false, there exists two codewords $\mathbf{a} = a_1a_2\cdots a_n$ and $\mathbf{b} = b_1b_2\cdots b_m$ that $\mathbf{a}$ is a prefix of $\mathbf{b}$. This means that $m \geq n$ and $a_i = b_i$, for $i = 1, \cdots n$. Then, we have $0.b_1b_2\cdots b_m$ lie inside both intervals $([0.a_1a_2\cdots a_n, 0.a_1a_2\cdots a_n\dot{1}]$ and $[0.b_1b_2\cdots b_n, 0.b_1b_2\cdots b_n\dot{1}]$ specified by $\mathbf{a}$ and $\mathbf{b}$. This contradicts with the assumption that the corresponding intervals for each symbols cannot overlap. □

Actually, the prefix condition not only guarantees the uniqueness of the codewords. With little inspection, the condition ensures that each symbol can be decoded instantenously once the codeword of the symbol is received by the decoder.

If the coding trick described above is only used for one symbol at a time, we will be compressing the source rather inefficiently. (In fact, we are "expanding" rather than "compressing" in our example since the source bit rate is 1 bit/sample but the average "compressed" bit rate is $0.25 * 3 + 0.75 * 1 = 1.5$ bit/sample.) The real power of arithmetic coding is that it can handle a group of symbols with arbitrary size easily. Let say we are going to encode *bb*. Instead of sending a codeword right away for the first symbol (b), we can first further partition the interval of b into intervals for *ba* ($[0.25, 0.4375)$) and *bb* ($[0.4375, 1)$). Upon receiving the second b, the encoder realizes that *code interval* is $[0.4375, 1)$ and can represent this code interval as described before. In this case, $0.1b$ and its corresponding region lie inside the probability interval of *bb* and thus *bb* can be compressed as a single bit 1. Of course, we do not need to stop at the second symbol, we can continue to partition the code interval into finer and finer region and ultimately the interval will become so short and converge to a real number. This real number corresponds precisely to the compressed symbols.

Due to finite precision of any computing device, scaling is an required step to avoid underflow problems. Let LOW and $HIGH$ be the lower and upper bound of the code interval, respectively. During encoding, the encoder knows that the next bit has to be 0 whenever a) $HIGH < 0.5$. Therefore, the encoder can safely output a bit 0 and scales up the region by two. That is, $LOW \leftarrow 2LOW$ and $HIGH \leftarrow 2HIGH$. On the other hand, if b) $LOW > 0.5$, the encoder knows that next output bit has to be 1 and thus it can output 1 and scales up the region by setting $LOW \leftarrow 2(LOW - 0.5)$ and $HIGH \leftarrow 2(HIGH - 0.5)$.

The scaling steps described above are not complete. With the only two scaling rules above, it may occur that $LOW = 0.5 - \epsilon_1$ and $HIGH = 0.5 + \epsilon_2$ with very small ϵ_1 and ϵ_2. To avoid this kind of underflow, the code interval should be scaled by two also when c) $HIGH < 0.75$ and $LOW > 0.25$. In that case, $LOW \leftarrow 2(LOW - 0.25)$ and $HIGH \leftarrow 2(HIGH - 0.25)$. Note that the encoder cannot tell whether the next output bit should be 1 or 0 at this stage. However, for each scaling-up, the encoder expects the code interval to be closer to 0.5. Actually, one can easily verify that if the code interval turns out to be less than 0.5 after one scaling, the next two output bits should be 01 as the code interval will lie in the second quandrant if no scaling has been applied. Similarly, if the code interval turns out to be larger than 0.5, the next two output bits should be 10.

In general, the encoder will store the number of times that Case c) happens and let the number be BITS-TO-FOLLOW. When finally the encoder realizes the code interval is smaller than 0.5 (i.e, Case a) happens), the next BITS-TO-FOLLOW+1 output bits should be $011 \cdots 1$ and BITS-TO-FOLLOW should be reset to 0. If the opposite (Case b) happens, the next BITS-TO-FOLLOW+1 output bits will be $100 \cdots 0$ and again BITS-TO-FOLLOW should be reset to 0.

Decoding is very similar to encoding except that an extra variable VALUE comes into the picture. VALUE can be interpreted as the location of the code interval up to the maximum available precision. For example, let the bit precision of the coders be 6. At the beginning of decoding, VALUE is constructed from the first 6 bits of the incoming bit stream. From VALUE, the decoder will know precisely which symbol the code interval lies into and thus such symbol

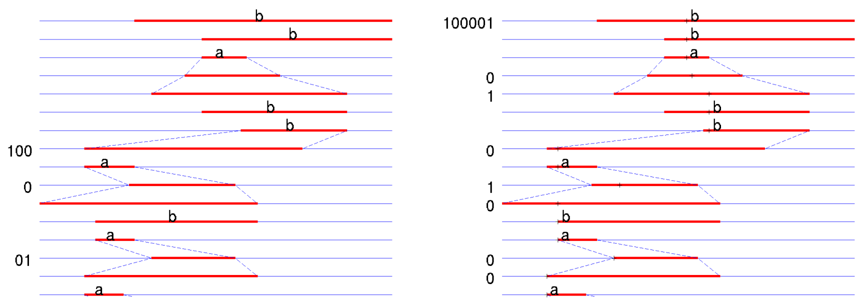

Figure 2.5: An example of encoding (left) and decoding (right) of arithmetic coding. A sequence of symbols $bbabbabaa\cdots$ is compressed into a bitstream $1000010101000\cdots$.

will be the next output symbol. As the output symbol is known, the code interval will be partitioned and shrinked as in encoding. The scaling procedure is almost identical except that the decoder needs to update VALUE as well during scaling. Moreover, VALUE is scaled up by two, an incoming bit is read as the least significant bit of VALUE. This step is important to maintain the precision of VALUE.

Example 2.9 (Arithmetic Coding)**.** A detail coding example is shown in Figure 2.5. A sequence of symbols $bbabbabaa\cdots$ is compressed into $1000010101000\cdots$.

The left and right figures in Figure 2.5 describe the status of encoding and decoding, respectively. The thick horizontal line is the code interval. In the left figure, we can see the code interval shrinks to lie within the range $[0.25, 0.75)$ as symbols b, b, a are coded. The code interval scales up twice afterward and thus BITS-TO-FOLLOW becomes two. The scaling up steps are highlighted by the dash lines connecting the corresponding LOWs and HIGHs of code intervals of adjacent time steps. As two symbols b, b are coded afterward, the code interval now falls in the range of $[0.5, 1)$, thus the code interval expands and 100 are output. The explanation of rest of the left figure is similar and thus is omitted.

The right figure looks very similar to the left figure except that there is a cross mark within the code interval for each iteration step. The cross mark corresponds to VALUE described previously. Since 6 bit precision is used in this example, 6 bits 100001 are read by the decoder to generate VALUE at the beginning of decoding. As VALUE falls inside the probability interval of b, the decoder knows that the first output symbol is b. Similarly, it can tell from the next two iteration steps that b and a are the next two symbols. Case c) scaling occurs in the fourth and fifth iterations and two bits are read by the decoder to keep VALUE to maintain the precision of VALUE. The explanation for rest of the figure is similar and will be left as an exercise to the readers.

Due to space limitation, we only describe some fundamental elements of arithmetic coding. For detail implementation issue, we recommend the classic reference by Witten *et al.* [4].