06

Stream Codes

Notice

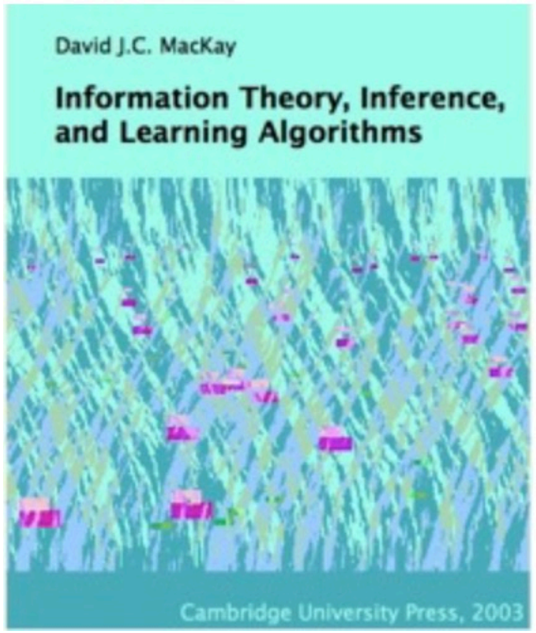
- **Author**
  - **João Moura Pires (jmp@fct.unl.pt)**

- **This material can be freely used for personal or academic purposes without any previous authorization from the author, provided that this notice is maintained/kept.**

- **For commercial purposes the use of any part of this material requires the previous authorization from the author.**

Bibliography

- **Many examples are extracted and adapted from:**



Information Theory, Inference, and Learning Algorithms
David J.C. MacKay
2005, Version 7.2

- **And some slides were based on Iain Murray course**
  - **http://www.inf.ed.ac.uk/teaching/courses/it/2014/**

Table of Contents

The guessing game

Two data compression schemes

- **Arithmetic coding**
  - Method that goes hand in hand with the philosophy that **compression of data from a source entails probabilistic modelling of that source**.
  - As of 1999, the best compression methods for text files use arithmetic coding
  - Several state-of-the-art image compression systems use it too.

- **Lempel–Ziv coding**
  - is a ‘universal’ method, designed under the philosophy that we would like a single compression algorithm that will do a reasonable job for any source.
  - Lempel–Ziv compression is widely used and often effective.

The guessing game

- Consider the redundancy in a typical English text file:
  - Characters with non-equal frequency;
  - Certain consecutive pairs of letters are more probable than others;
  - Entire words can be predicted given the context and a semantic understanding of the text.
- A guessing game
  - Assume an alphabet of the 26 upper case letters A, B, C, …, Z and a space ‘-’.
  - Repeatedly attempts to **predict the next character in a text file**.
    - The only **feedback** being whether the guess is **correct** or **not**, until the character is correctly guessed.
    - Then we note the **number of guesses** that were made when the character was identified

The guessing game

- Assume an alphabet of the 26 upper case letters A, B, C, …, Z and a space ‘-’.
- Repeatedly attempts to **predict the next character in a text file**.
  - The only feedback being whether the guess is correct or not, until the character is correctly guessed.
  - Then we note the number of guesses that were made when the character was identified.
- Example:

```
T H E R E - I S - N O - R E V E R S E - O N - A - M O T O R C Y C L E -
1 1 1 5 1 1 2 1 1 2 1 1 15 1 17 1 1 1 2 1 3 2 1 2 2 7 1 1 1 1 4 1 1 1 1 1
```

  - in **many cases**, the next **letter is guessed immediately, in one guess**.
  - In **other cases**, particularly at the start of syllables, **more guesses are needed**.

The guessing game

T	H	E	R	E	-	I	S	-	N	O	-	R	E	V	E	R	S	E	-	O	N	-	A	-	M	O	T	O	R	C	Y	C	L	E	-
1	1	1	5	1	1	2	1	1	2	1	1	15	1	17	1	1	1	2	1	3	2	1	2	2	7	1	1	1	1	4	1	1	1	1	1

- The **maximum number of guesses** that the subject will make **for a given letter is 27**
- So what the subject is doing for us is performing a **time-varying mapping**
  - {A, B, C, …, Z, –} ===> {1, 2, 3, …, 27}
  - A **twin** could decode a sequence 1 1 1 5 1 as THERE
    - **If we stop him whenever he has made a number of guesses equal to the given number**, then he will have just guessed the correct letter, and we can then say ‘yes, that’s right’, and move to the next character.

The guessing game

T	H	E	R	E	-	I	S	-	N	O	-	R	E	V	E	R	S	E	-	O	N	-	A	-	M	O	T	O	R	C	Y	C	L	E	-
1	1	1	5	1	1	2	1	1	2	1	1	15	1	17	1	1	1	2	1	3	2	1	2	2	7	1	1	1	1	4	1	1	1	1	1

- A **compression system with the help of just one human**:
  - A window length L (a number of characters of context) to show the human
  - For every one of the 27^L possible strings of length L, we ask them,
    - 'What would you predict is the next character?', and '
    - If that prediction were wrong, what would your next guesses be?'
  - After tabulating their answers to these 26×27^L questions, we could use two copies of these enormous tables at the encoder and the decoder.
- Such a language model is called an ***L*th order Markov model**.

Arithmetic codes

Arithmetic codes

- Arithmetic codes were invented by Elias, by Rissanen and by Pasco, and subsequently made practical by Witten et al. (1987).
- As **each symbol is produced by the source**, a probabilistic model supplies a **predictive distribution over all possible values of the next symbol**, that is, a list of positive numbers $\{p_i\}$ that sum to one.
- If we choose to model the **source as producing i.i.d. symbols with some known distribution**, then the **predictive distribution is the same every time**;
- Arithmetic coding can with equal ease handle **complex adaptive models that produce context-dependent predictive distributions**

Arithmetic codes

- Arithmetic codes were invented by Elias, by Rissanen and by Pasco, and subsequently made practical by Witten et al. (1987).
- As **each symbol is produced by the source**, a **probabilistic model** supplies a **predictive distribution over all possible values of the next symbol**, that is, a list of positive numbers $\{p_i\}$ that sum to one
- The **encoder** makes use of the model's predictions to create a **binary string**.
- The **decoder makes use of an identical twin of the model** to interpret the binary string.
- The predictive model is usually implemented in a computer program

Arithmetic codes

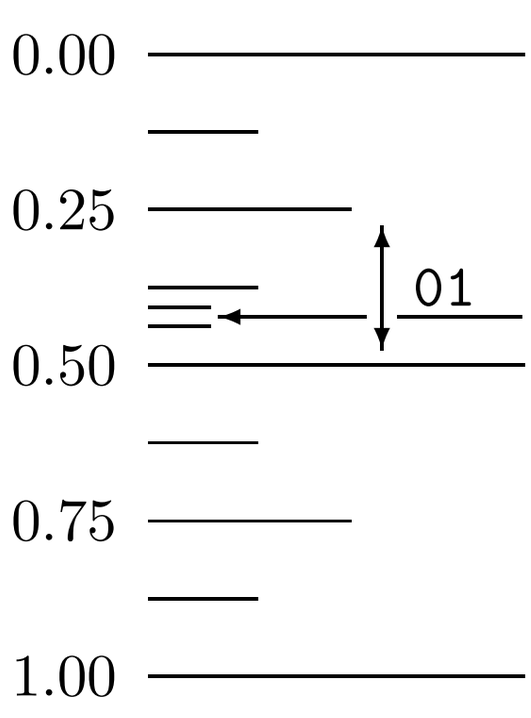
- Let the source alphabet be $A_X = \{a_1, \ldots, \boldsymbol{a_I}\}$
- Let the Ith symbol $\boldsymbol{a_I}$ have the special meaning '**end of transmission**'.
- The source spits out a sequence $x_1, x_2, \ldots, x_n, \ldots$.
- We will assume that a **computer program is provided to the encoder** that assigns a **predictive probability distribution** over a_i given the sequence that has occurred thus far, $P(x_n = a_i \mid x_1, \ldots, x_{n-1})$.
- The **receiver** has an **identical program** that produces the same predictive probability distribution $P(x_n = a_i \mid x_1, \ldots, x_{n-1})$.

Concepts for understanding arithmetic coding

- **Notation for intervals.**
  - The interval **[0.01, 0.10)** is all numbers between 0.01 and 0.10
    - including $0.01\dot{} \equiv 0.01000...$
    - but not $0.10\dot{} \equiv 0.1000...$
- A binary transmission defines an interval within the real line from 0 to 1.
- For example the string **01** is **interpreted as a binary real number 0.01**.
  - Lets make that corresponds to the interval **[0.01, 0.10)** in **binary**,
  - i.e., the interval [0.25, 0.50) in base ten

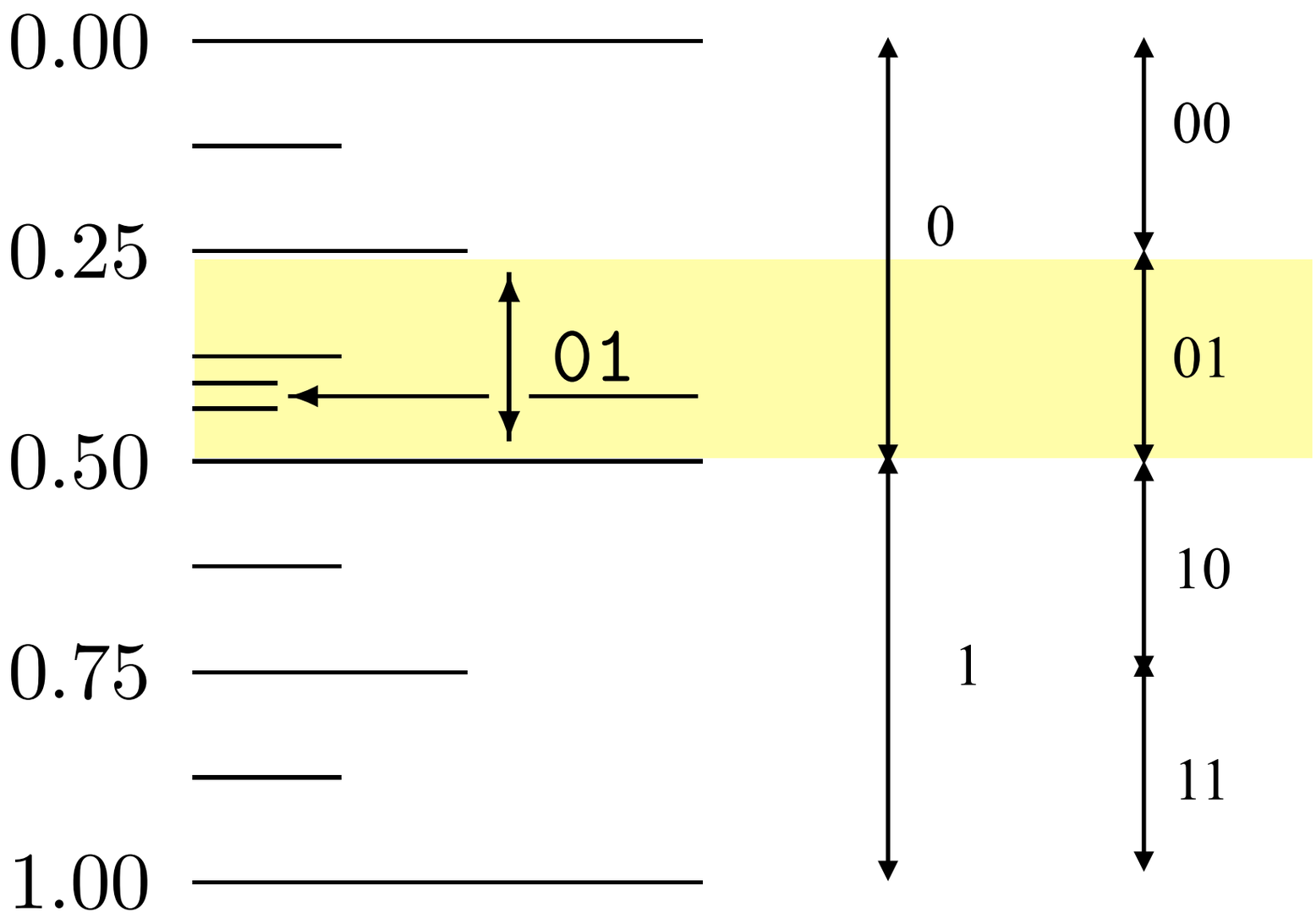
Concepts for understanding arithmetic coding

- A binary transmission defines an interval within the real line from 0 to 1.
- For example the string **01** is **interpreted as a binary real number 0.01**.
  - Lets make that corresponds to the interval **[0.01, 0.10)** in **binary**,
  - i.e., the interval **[0.25, 0.50)** in base ten



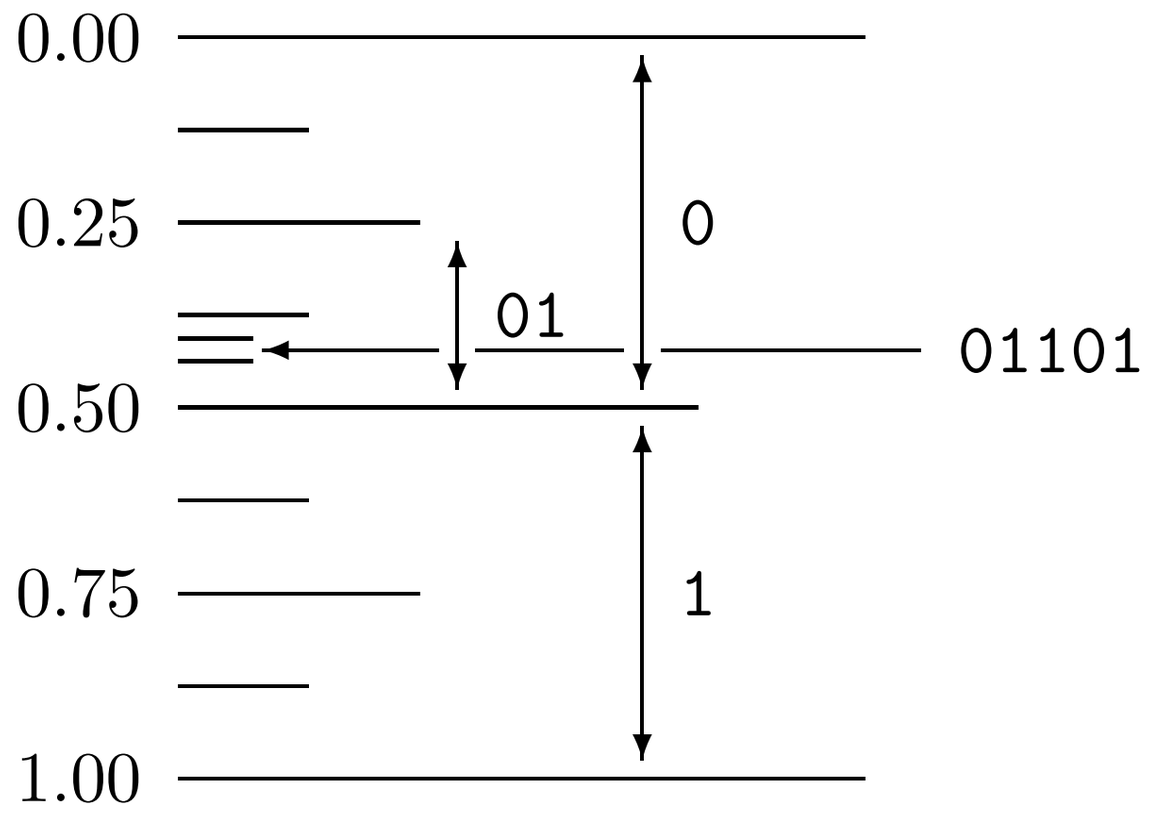
Concepts for understanding arithmetic coding

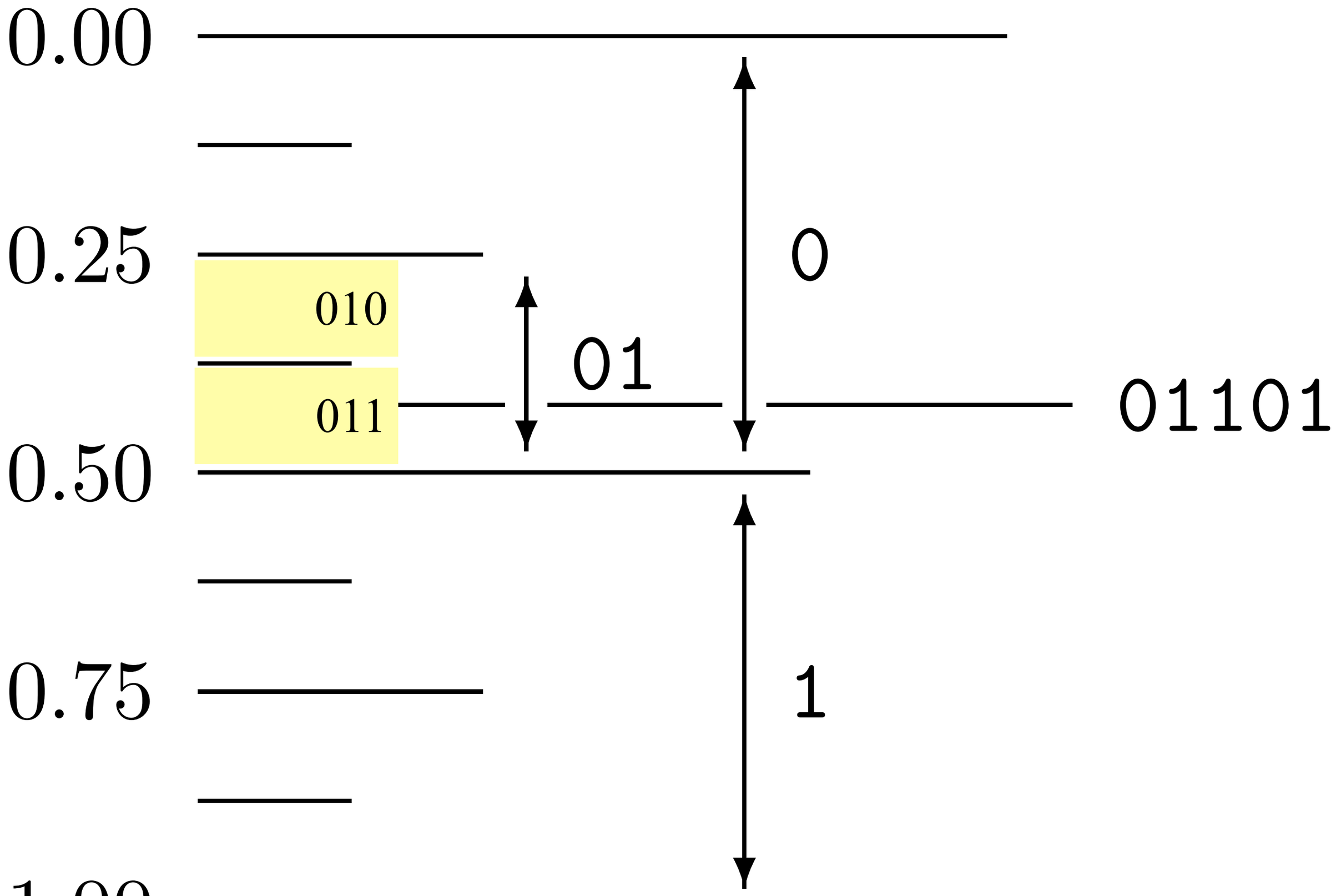
- A binary transmission defines an interval within the real line from 0 to 1.

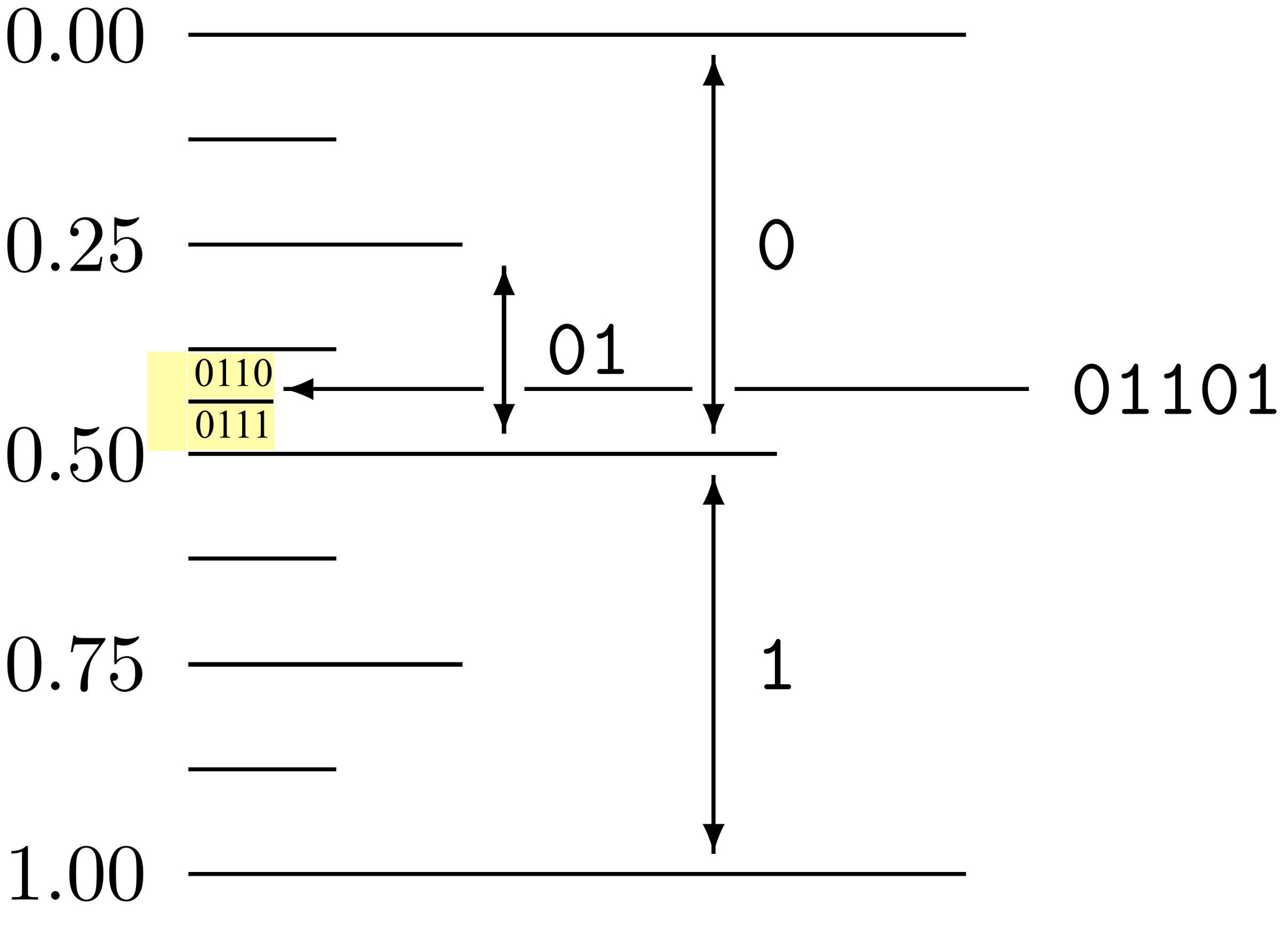


Concepts for understanding arithmetic coding

- **01** is **interpreted as a binary real number 0.01;** corresponds to the interval **[0.01, 0.10)**
  - **0.10 = 0.01 + 0.01**
- The **longer string 01101** is **interpreted as a binary real number 0. 01101** and corresponds to a **smaller interval** [0.**01101**, 0.**01110**).
  - 0.**01110** = **0.01101** + 0.**00001**

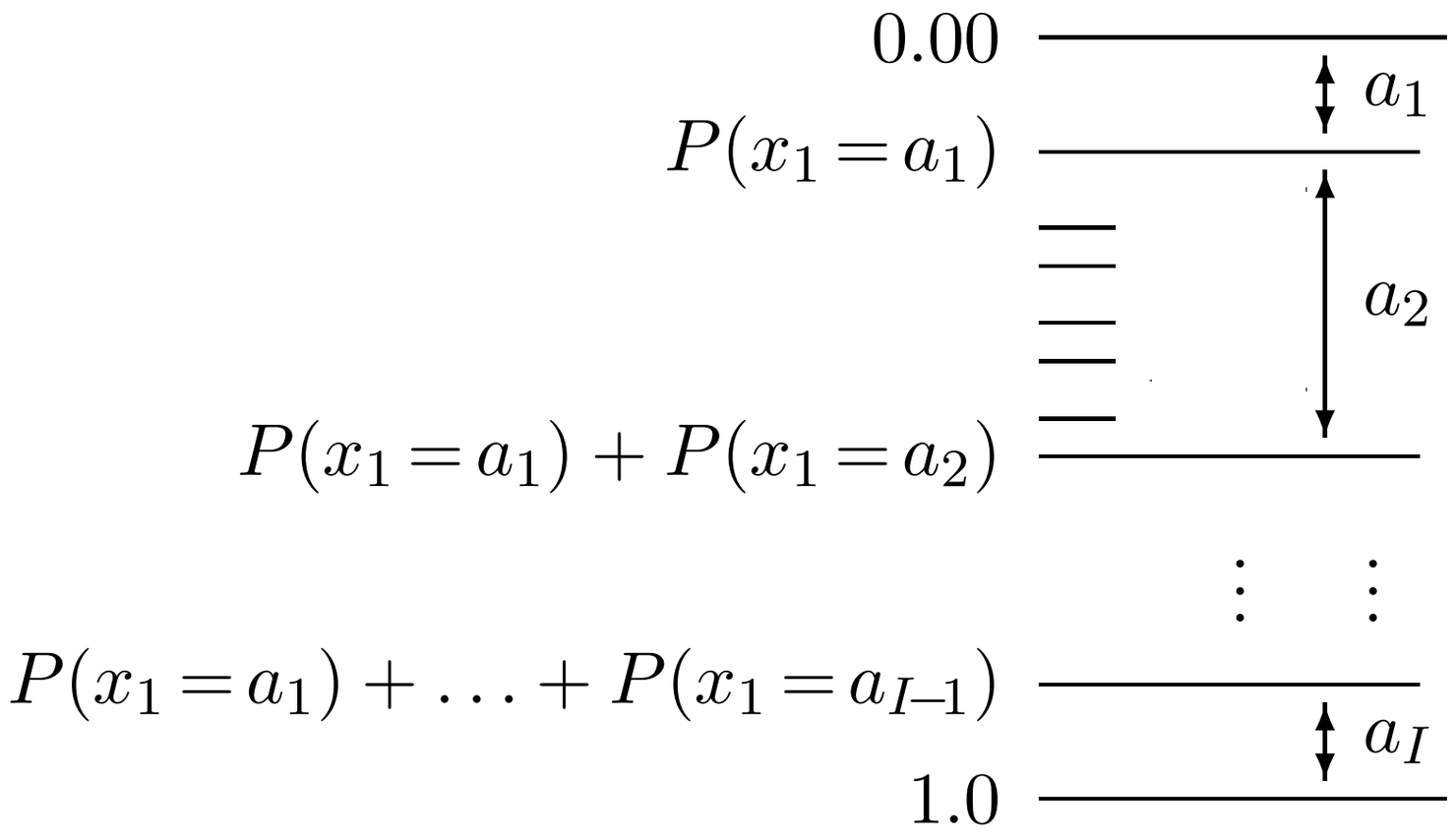



0.00
0.25
0.50
0.75
1.00
010
011
01
0
1
01101


0.00
0.25
0.50
0.75
1.00
0
01
01101
1
0110
0111

Concepts for understanding arithmetic coding

- The source alphabet be $A_X = \{a_1, \ldots, \boldsymbol{a_I}\}$
- We can divide the real line [0,1) into I intervals of lengths equal to the probabilities $P(x_1 = a_i)$.

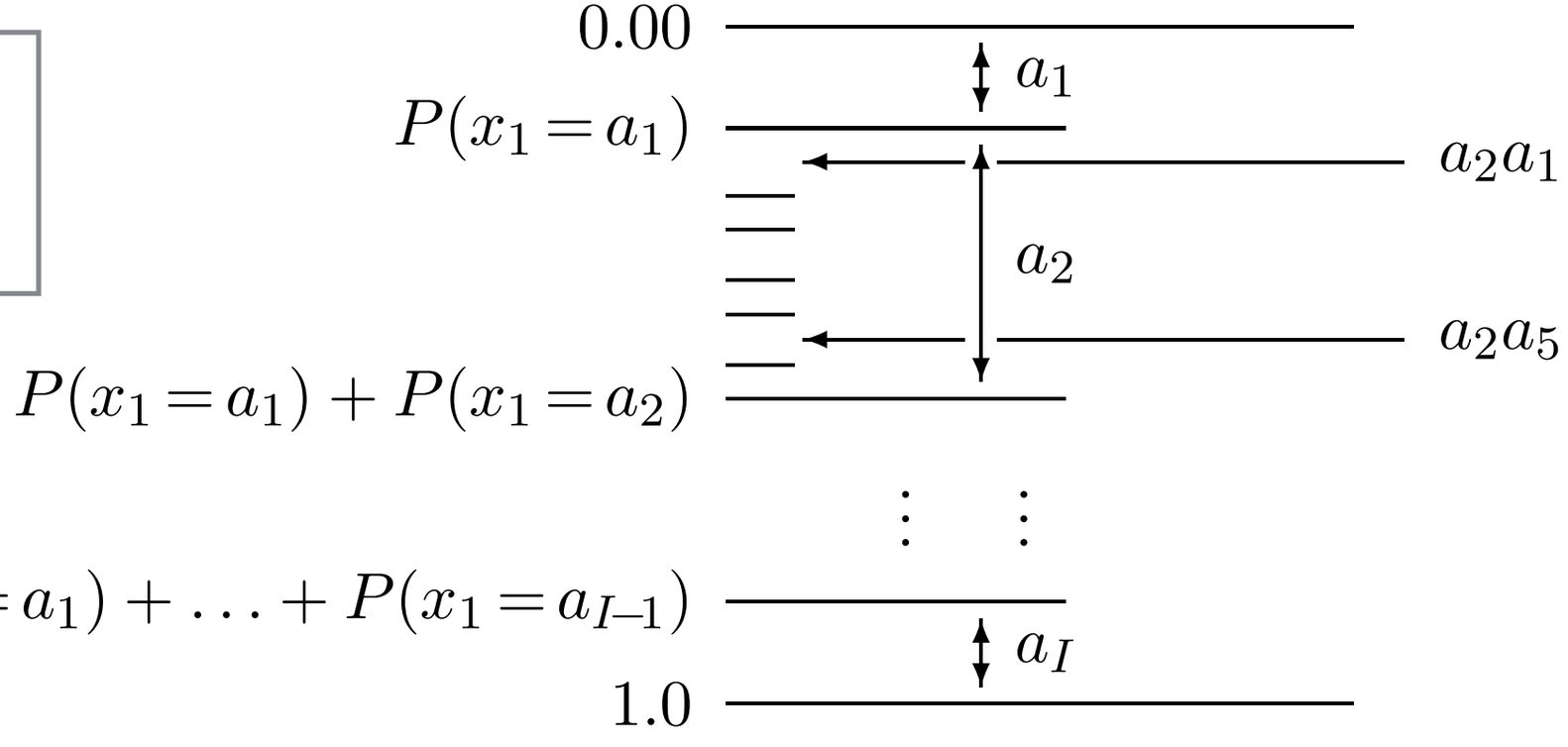


Concepts for understanding arithmetic coding

- We may then take each interval a_i and subdivide it into intervals denoted $a_i a_1, a_i a_2, \ldots, a_i a_I$, such that the length of $\boldsymbol{a_i a_j}$ is **proportional** to $\boldsymbol{P(x_2 = a_j \mid x_1 = a_i)}$**.**
- Indeed the length of the interval $\boldsymbol{a_i a_j}$ will be **precisely the joint probability**

$$P(x_1 = a_i, x_2 = a_j) = P(x_1 = a_i) P(x_2 = a_j \mid x_1 = a_i).$$

A probabilistic model defines real intervals within the real line [0,1).



Concepts for understanding arithmetic coding

- Iterating this procedure, the interval [0, 1) can be divided into a sequence of intervals corresponding to **all possible finite length strings** $x_1x_2 \ldots x_N$, such that the **length of an interval is equal to the probability of the string given our model**.

A probabilistic model defines real intervals within the real line [0,1).

Formulae describing arithmetic coding

- The process can be written explicitly as follows. The **intervals** are defined in terms of the **lower** and **upper cumulative probabilities**

$$Q_n(a_i \,|\, x_1,\ldots,x_{n-1}) = \sum_{i'=1}^{i-1} P(x_n = a_{i'} \,|\, x_1,\ldots,x_{n-1})$$

$$R_n(a_i \,|\, x_1,\ldots,x_{n-1}) = \sum_{i'=1}^{i} P(x_n = a_{i'} \,|\, x_1,\ldots,x_{n-1})$$

- As the nth symbol arrives, we sub-divide the n-1th interval at the points defined by Q_n and R_n

- Starting with the first symbol the intervals are:

$$a_1 \leftrightarrow [Q_1(a_1), R_1(a_1)) = [0, P(x_1 = a_1)),$$

$$a_2 \leftrightarrow [Q_1(a_2), R_1(a_2)) = [P(x = a_1), P(x = a_1) + P(x = a_2)),$$

$$a_I \leftrightarrow [Q_1(a_I), R_1(a_I)) = [P(x_1 = a_1) + \ldots + P(x_1 = a_{I-1}), 1.0).$$

Arithmetic coding. Iterative procedure

- Iterative procedure to find the interval **[u, v)** for the string $\boldsymbol{x_1\, x_2\, ...x_N}$

u := 0.0
v := 1.0
p := $v - u$
for n = 1 to N {
    Compute the cumulative probabilities Q_n and R_n
    $v := u + pR_n(x_n \mid x_1, \ldots, x_{n-1})$
    $u := u + pQ_n(x_n \mid x_1, \ldots, x_{n-1})$
    $p := v - u$
}

$$Q_n(a_i \mid x_1,\ldots,x_{n-1}) = \sum_{i'=1}^{i-1} P(x_n = a_{i'} \mid x_1,\ldots,x_{n-1}) \qquad R_n(a_i \mid x_1,\ldots,x_{n-1}) = \sum_{i'=1}^{i} P(x_n = a_{i'} \mid x_1,\ldots,x_{n-1})$$

- To encode a string $\boldsymbol{x_1\, x_2\, ...x_N}$, we locate the interval corresponding to $\boldsymbol{x_1\, x_2\, ...x_N}$, and send a binary string whose interval lies within that interval.

Example: compressing the tosses of a bent coin

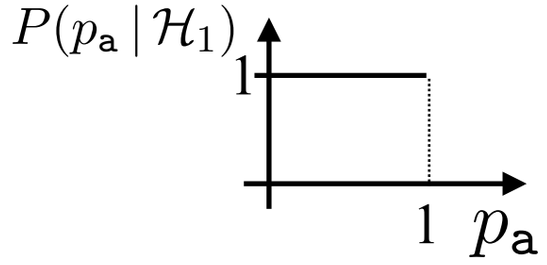
- Consider a **bent** coin !
- The **two outcomes when the coin is tossed** are denoted **a** and **b**. (**a** ≠ **b**)
- A **third possibility is that the experiment is halted**, an event denoted by the 'end of file' symbol, '□'
- Encoding
  - Source string '**b b b a** □'
  - We pass along the string **one symbol at a time** and use our model to **compute the probability distribution of the next symbol** given the string thus far
  - We do not know p_a and p_b and p □ .
  - We will assume the a *priori* probabilities are: $P(\mathbf{a}) = P(\mathbf{b}) = 0{,}425$ and $P(\square) = 0.15$

Understanding the tosses of a bent coin

- A **bent** coin is tossed F times. We observe a sequence **s** of heads (**a**) and tails (**b**)
- We want to know:
  - *i*) the bias of the coin (an *unknown* parameter)
  - *ii*) predict the **probability that the next toss** will result in a head, i.e, $P(\text{next toss} = \mathbf{a})$)

- We assume that $p_a = P(\mathbf{a})$ is uniformly distributed in the interval [0, 1]

$$P(p_\mathtt{a} \,|\, \mathcal{H}_1) = 1, \qquad p_\mathtt{a} \in [0,1] \qquad\qquad p_\mathtt{b} \equiv 1 - p_\mathtt{a}$$



- Let's denote our assumptions by H_1.
- The probability, given p_a, that F tosses result in a sequence **s** that contains $\{F_a, F_b\}$ counts of the two outcomes is

$$P(\mathbf{s} \,|\, p_a, F, H_1) = \; p_a^{F_a}(1-p_a)^{F_b}$$

Understanding the tosses of a bent coin

- Assuming H_1 to be true, the posterior probability of p_a, given a string $\mathbf{s}$ of length F that has counts $\{F_a, F_b\}$, is, by Bayes' theorem,

$$P(p_{\mathtt{a}} \,|\, \mathbf{s}, F, \mathcal{H}_1) \;=\; \frac{P(\mathbf{s} \,|\, p_{\mathtt{a}}, F, \mathcal{H}_1) P(p_{\mathtt{a}} \,|\, \mathcal{H}_1)}{P(\mathbf{s} \,|\, F, \mathcal{H}_1)}$$

- $P(\mathbf{s} \,|\, p_a, F, H_1) = \; p_a^{\,F_a} (1 - p_a)^{F_b}$

- $P(p_{\mathtt{a}} \,|\, \mathcal{H}_1) = 1, \qquad p_{\mathtt{a}} \in [0, 1]$

$$P(p_{\mathtt{a}} \,|\, \mathbf{s}, F, \mathcal{H}_1) \;=\; \frac{p_{\mathtt{a}}^{F_{\mathtt{a}}} (1 - p_{\mathtt{a}})^{F_{\mathtt{b}}}}{P(\mathbf{s} \,|\, F, \mathcal{H}_1)}$$

- The normalizing constant is given by the beta integral

$$P(\mathbf{s} \,|\, F, \mathcal{H}_1) = \int_0^1 \mathrm{d}p_{\mathtt{a}} \, p_{\mathtt{a}}^{F_{\mathtt{a}}} (1 - p_{\mathtt{a}})^{F_{\mathtt{b}}} = \frac{\Gamma(F_{\mathtt{a}} + 1) \Gamma(F_{\mathtt{b}} + 1)}{\Gamma(F_{\mathtt{a}} + F_{\mathtt{b}} + 2)} = \frac{F_{\mathtt{a}}! F_{\mathtt{b}}!}{(F_{\mathtt{a}} + F_{\mathtt{b}} + 1)!}.$$

Understanding the tosses of a bent coin

- Assuming H_1 to be true, the posterior probability of p_a, given a string $\mathbf{s}$ of length F that has counts $\{F_a, F_b\}$, is, by Bayes' theorem,

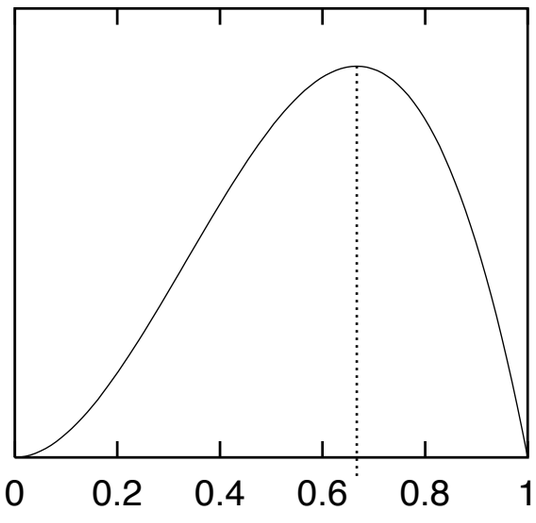
$$P(p_\mathtt{a} \,|\, \mathbf{s}, F, \mathcal{H}_1) \;=\; \frac{p_\mathtt{a}^{F_\mathtt{a}}(1-p_\mathtt{a})^{F_\mathtt{b}}}{P(\mathbf{s} \,|\, F, \mathcal{H}_1)}$$

  - The normalizing constant is given by the beta integral

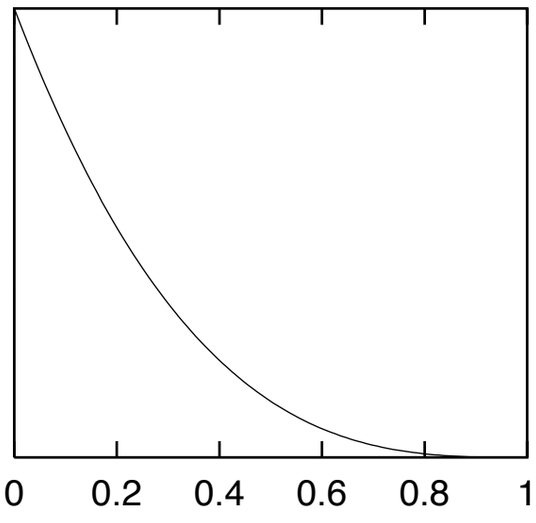
$$P(\mathbf{s} \,|\, F, \mathcal{H}_1) = \frac{F_\mathtt{a}! F_\mathtt{b}!}{(F_\mathtt{a} + F_\mathtt{b} + 1)!}.$$

For $\mathbf{s}$ = aba, the most probable value of p_a (i.e., the value that maximizes the posterior probability density) is 2/3.

The mean value of p_a is 3/5.



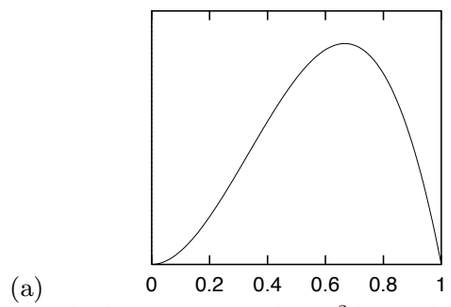
(a) $P(p_\mathtt{a} \,|\, \mathbf{s} = \mathtt{aba}, F = 3) \propto p_\mathtt{a}^2(1-p_\mathtt{a})$



(b) $P(p_\mathtt{a} \,|\, \mathbf{s} = \mathtt{bbb}, F = 3) \propto (1-p_\mathtt{a})^3$

Understanding the tosses of a bent coin

- A **bent** coin is tossed F times. We observe a sequence **s** of heads (**a**) and tails (**b**)
- We want to know:
  - *i*) the bias of the coin (an *unknown* parameter)
  - *ii*) predict the **probability that the next toss** will result in a head, i.e, P(next toss = **a**))



- The probability that the next toss is an **a**, is obtained by integrating over p_a. This has the effect of taking into account our uncertainty about p_a when making predictions. By the sum rule.

$$P(\mathtt{a} \,|\, \mathtt{s}, F) \quad = \quad \int \mathrm{d}p_\mathtt{a} \, P(\mathtt{a} \,|\, p_\mathtt{a}) P(p_\mathtt{a} \,|\, \mathtt{s}, F).$$

- The probability of an a given p_a is simply pa, so $P(\mathbf{a} \mid p_a) = p_a$

Understanding the tosses of a bent coin

- The probability that the next toss is an **a**, is obtained by integrating over p_a. This has the effect of taking into account our uncertainty about p_a when making predictions. By the sum rule.

$$P(\mathtt{a}\,|\,\mathbf{s},F) = \int \mathrm{d}p_{\mathtt{a}}\, p_{\mathtt{a}} \frac{p_{\mathtt{a}}^{F_{\mathtt{a}}}(1-p_{\mathtt{a}})^{F_{\mathtt{b}}}}{P(\mathbf{s}\,|\,F)}$$

$$= \int \mathrm{d}p_{\mathtt{a}}\, \frac{p_{\mathtt{a}}^{F_{\mathtt{a}}+1}(1-p_{\mathtt{a}})^{F_{\mathtt{b}}}}{P(\mathbf{s}\,|\,F)}$$

$$= \left[\frac{(F_{\mathtt{a}}+1)!\,F_{\mathtt{b}}!}{(F_{\mathtt{a}}+F_{\mathtt{b}}+2)!}\right] \Big/ \left[\frac{F_{\mathtt{a}}!\,F_{\mathtt{b}}!}{(F_{\mathtt{a}}+F_{\mathtt{b}}+1)!}\right] = \frac{F_{\mathtt{a}}+1}{F_{\mathtt{a}}+F_{\mathtt{b}}+2},$$

which is known as *Laplace's rule.*

Example: compressing the tosses of a bent coin

- The **two outcomes when the coin is tossed** are denoted **a** and **b**. (**a** ≠ **b**)
- A **third possibility is that the experiment is halted**, an event denoted by the 'end of file' symbol, '□'
- Encoding
  - Source string '**b b b a** □'
  - We pass along the string **one symbol at a time** and use our model to **compute the probability distribution of the next symbol** given the string thus far
  - We do not know p_a and p_b and p □ .
  - We will assume the a *priori* probabilities are: $P(\mathbf{a}) = P(\mathbf{b}) = 0{,}425$ and $P(\square) = 0.15$

Example: compressing the tosses of a bent coin

- We pass along the string one symbol at a time and use our model to compute the probability distribution of the next symbol given the string thus far.
- We will assume the a *priori* probabilities are: $P(\mathbf{a}) = P(\mathbf{b}) = 0{,}425$ and $P(\square) = 0.15$
- And a simple model that:
  - always assigns a probability of 0.15 to $\square$
  - assigns the remaining 0.85 to **a** and **b**, **divided in proportion to probabilities given by Laplace's rule**,

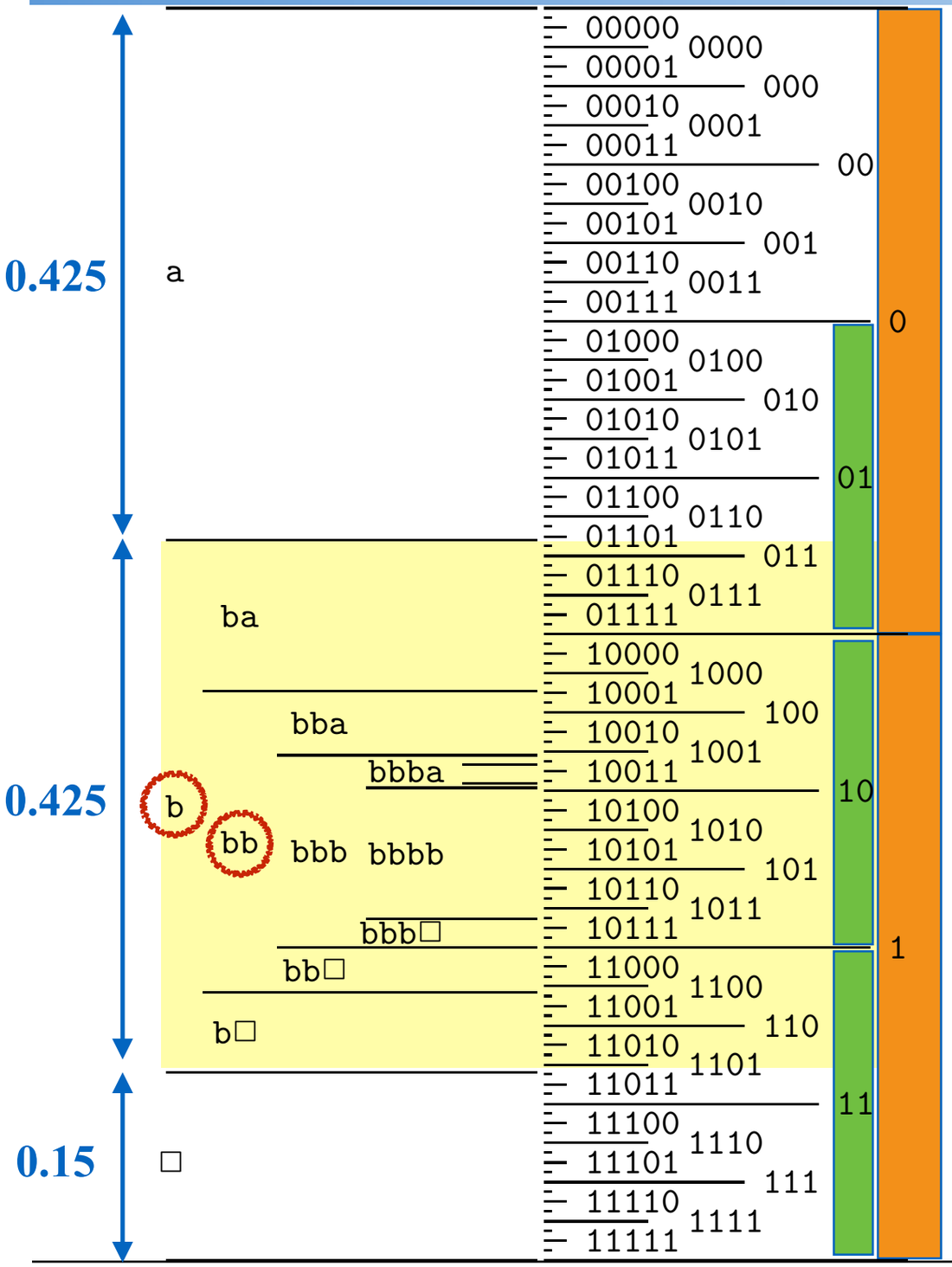
$$P_{\mathrm{L}}(\mathtt{a} \,|\, x_1, \ldots, x_{n-1}) = \frac{F_{\mathtt{a}} + 1}{F_{\mathtt{a}} + F_{\mathtt{b}} + 2}$$

Example: compressing the tosses of a bent coin

- We pass along the string one symbol at a time and use our model to compute the probability distribution of the next symbol given the string thus far.

Context (sequence thus far)	Probability of next symbol		
	$P(\mathtt{a}) = 0.425$	$P(\mathtt{b}) = 0.425$	$P(\square) = 0.15$
$\mathtt{b}$	$P(\mathtt{a} \mid \mathtt{b}) = 0.28$	$P(\mathtt{b} \mid \mathtt{b}) = 0.57$	$P(\square \mid \mathtt{b}) = 0.15$
$\mathtt{bb}$	$P(\mathtt{a} \mid \mathtt{bb}) = 0.21$	$P(\mathtt{b} \mid \mathtt{bb}) = 0.64$	$P(\square \mid \mathtt{bb}) = 0.15$
$\mathtt{bbb}$	$P(\mathtt{a} \mid \mathtt{bbb}) = 0.17$	$P(\mathtt{b} \mid \mathtt{bbb}) = 0.68$	$P(\square \mid \mathtt{bbb}) = 0.15$
$\mathtt{bbba}$	$P(\mathtt{a} \mid \mathtt{bbba}) = 0.28$	$P(\mathtt{b} \mid \mathtt{bbba}) = 0.57$	$P(\square \mid \mathtt{bbba}) = 0.15$

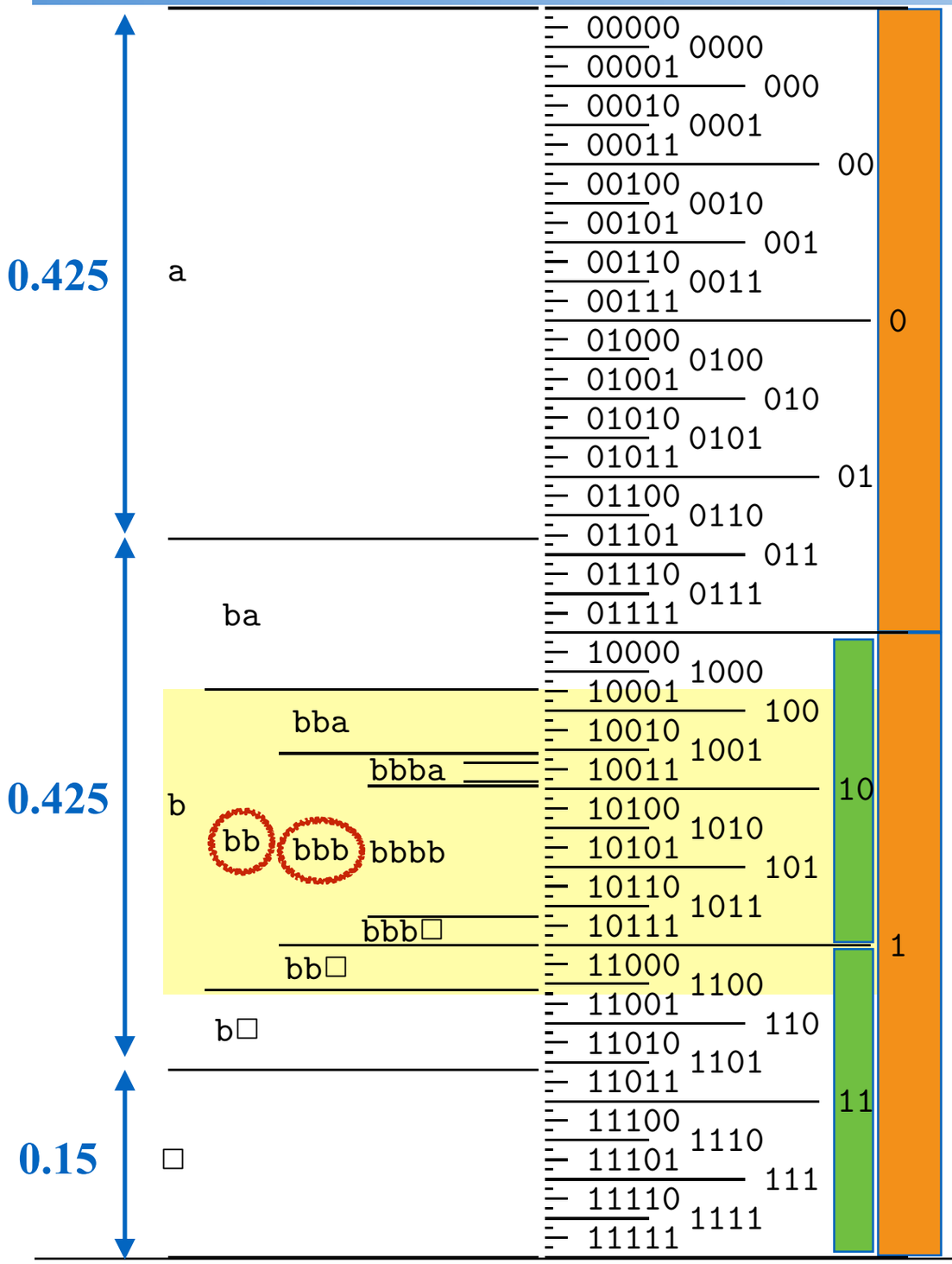
Example: compressing the tosses of a bent coin



- When the **first symbol** '**b**' is observed, the encoder knows that the encoded string will start '01', '10', or '11', but does not know which.
- Then examines the **next symbol**, which is '**b**'. The interval 'bb' lies wholly within interval '1', so the encoder can write the first bit: '1'

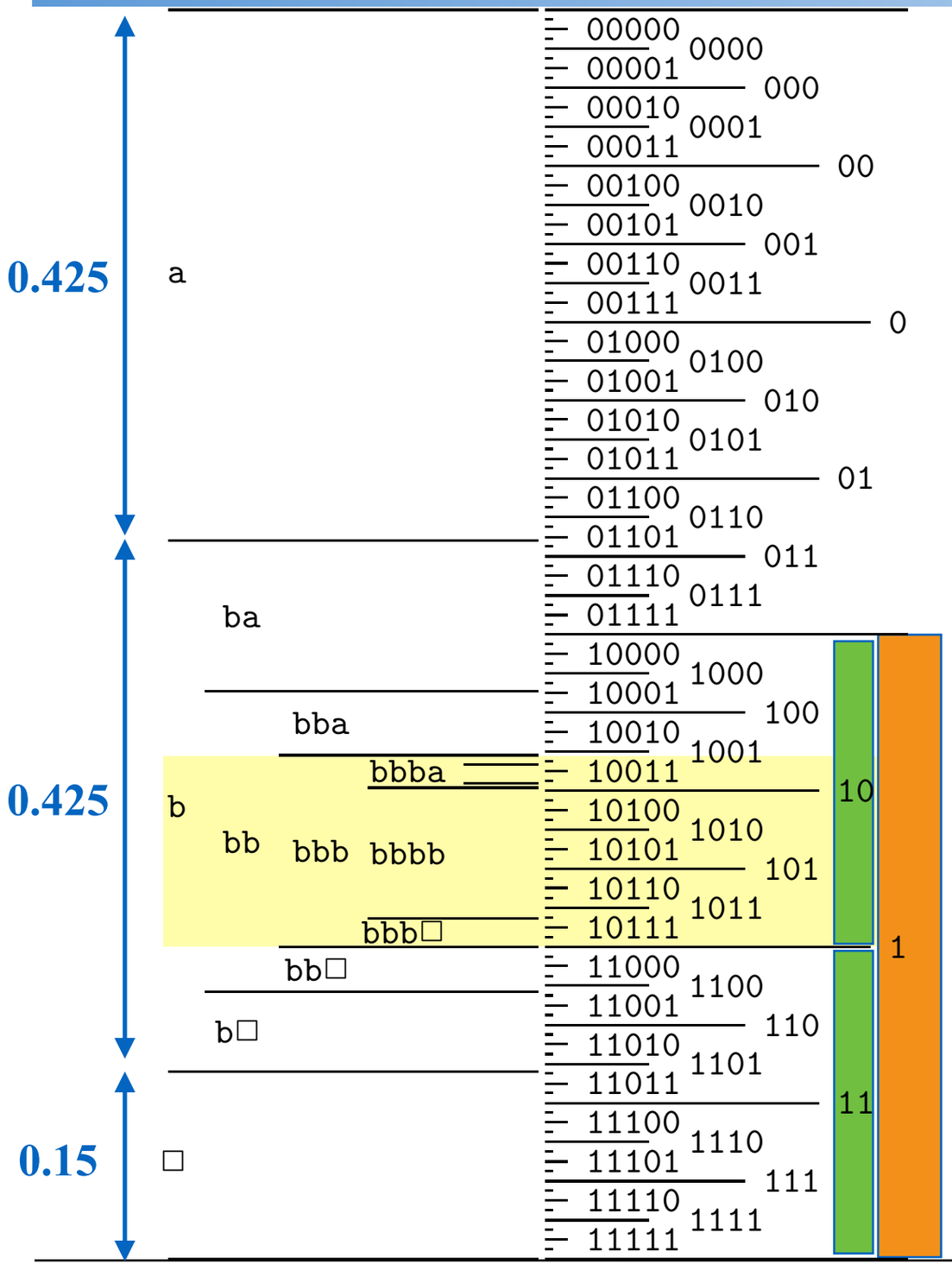
Context (sequence thus far)	Probability of next symbol		
	$P(\mathtt{a}) = 0.425$	$P(\mathtt{b}) = 0.425$	$P(\square) = 0.15$
b	$P(\mathtt{a} \mid \mathtt{b}) = 0.28$	$P(\mathtt{b} \mid \mathtt{b}) = 0.57$	$P(\square \mid \mathtt{b}) = 0.15$
bb	$P(\mathtt{a} \mid \mathtt{bb}) = 0.21$	$P(\mathtt{b} \mid \mathtt{bb}) = 0.64$	$P(\square \mid \mathtt{bb}) = 0.15$
bbb	$P(\mathtt{a} \mid \mathtt{bbb}) = 0.17$	$P(\mathtt{b} \mid \mathtt{bbb}) = 0.68$	$P(\square \mid \mathtt{bbb}) = 0.15$
bbba	$P(\mathtt{a} \mid \mathtt{bbba}) = 0.28$	$P(\mathtt{b} \mid \mathtt{bbba}) = 0.57$	$P(\square \mid \mathtt{bbba}) = 0.15$

Example: compressing the tosses of a bent coin



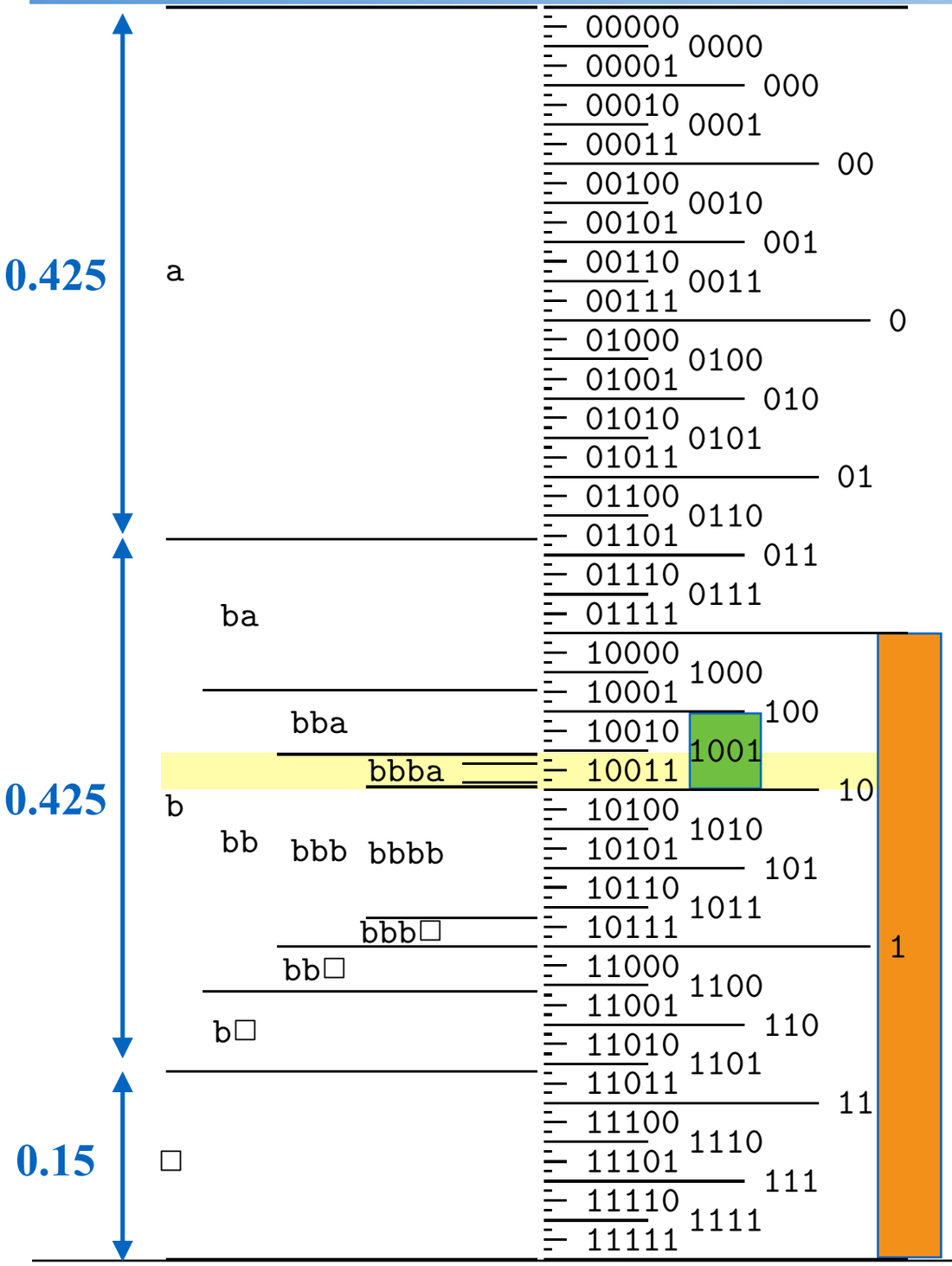
- When the **first symbol** '**b**' is observed, the encoder knows that the encoded string will start '01', '10', or '11', but does not know which.
- Then examines the **next symbol**, which is '**b**'. The interval 'bb' lies wholly within interval '1', so the encoder can write the first bit: '1'
- The **third symbol** '**b**' narrows down the interval a little.

Example: compressing the tosses of a bent coin



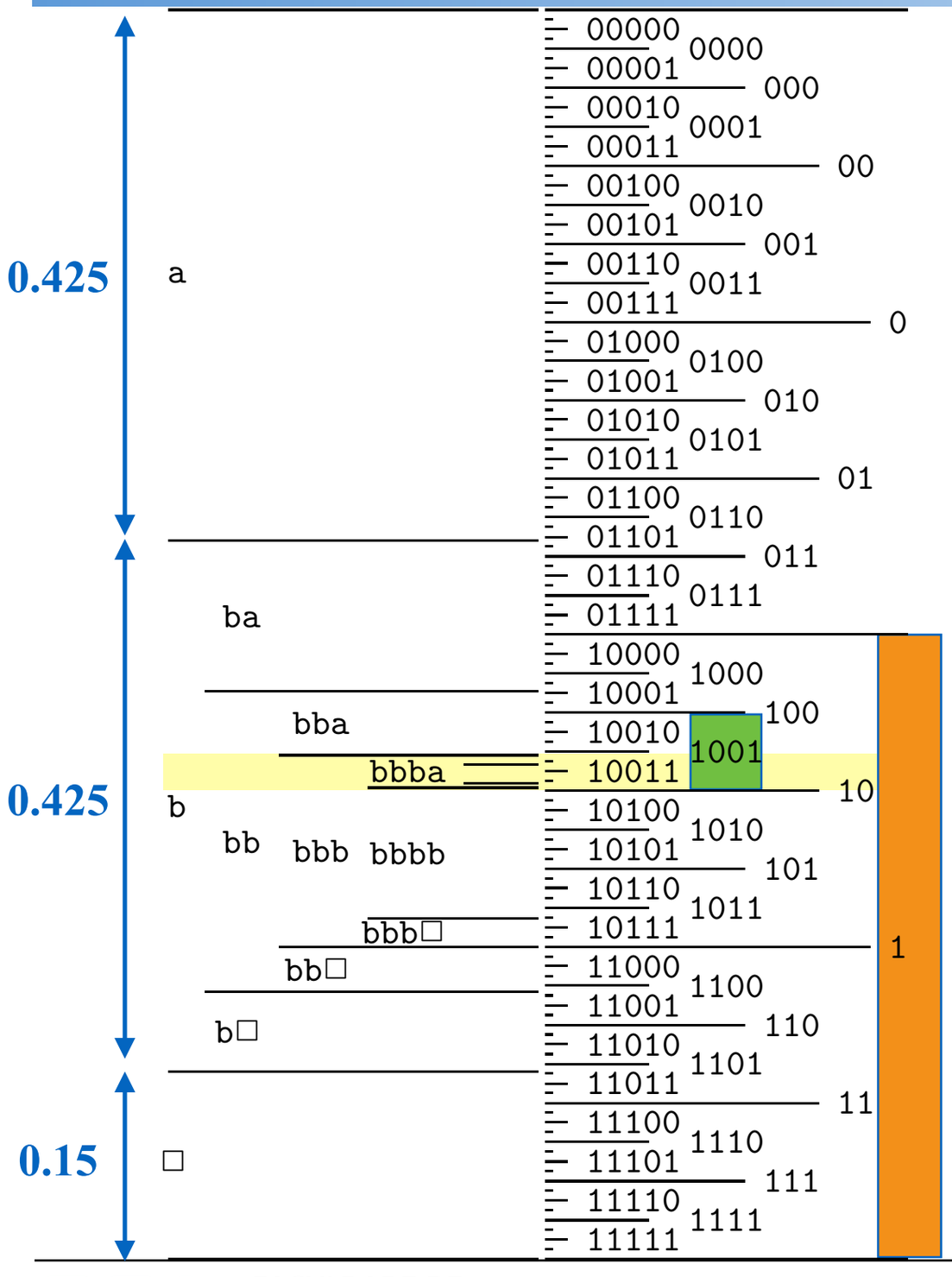
- When the **first symbol** '**b**' is observed, the encoder knows that the encoded string will start '01', '10', or '11', but does not know which.
- Then examines the **next symbol**, which is '**b**'. The interval 'bb' lies wholly within interval '1', so the encoder can write the first bit: '1'
- The **third symbol** '**b**' narrows down the interval a little, to lie wholly within interval '10'.
- When the next '**a**' is read from the source can we transmit some more bits.

Example: compressing the tosses of a bent coin



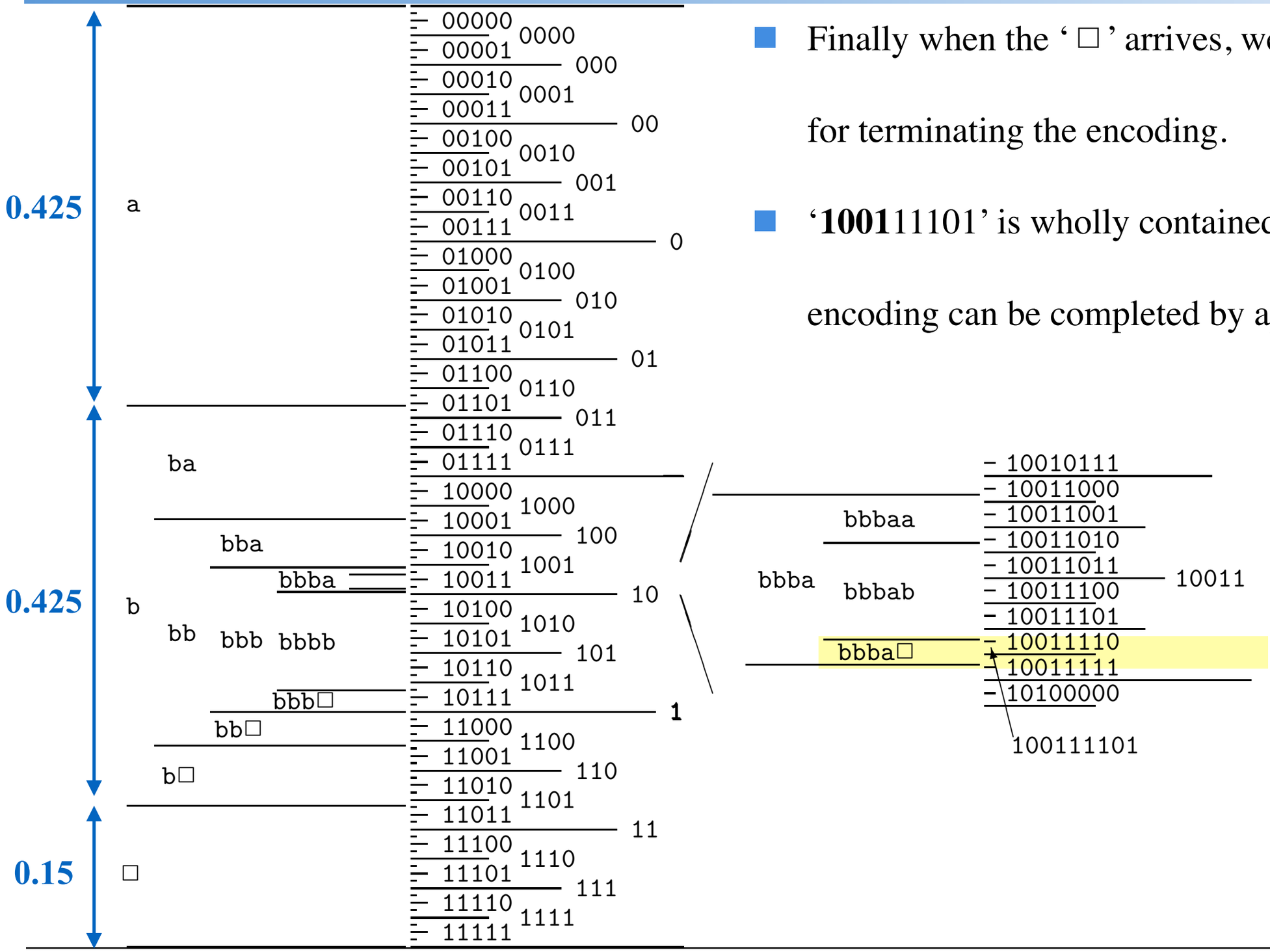
- When the next '**a**' is read from the source can we transmit some more bits.
- Interval '**bbba**' lies wholly within the interval '**1001**', so the encoder adds '01' to the '10' it has written.

Example: compressing the tosses of a bent coin



- When the next '**a**' is read from the source can we transmit some more bits.
- Interval '**bbba**' lies wholly within the interval '**1001**', so the encoder adds '01' to the '10' it has written.
- Finally when the '□' arrives, we need a procedure for terminating the encoding.

Example: compressing the tosses of a bent coin



- Finally when the '□' arrives, we need a procedure for terminating the encoding.
- '**1001**11101' is wholly contained by **bbba**□, so the encoding can be completed by appending '**11101**'.

Overhead to terminate a message is never more than 2 bits

- The overhead required to **terminate a message** is **never more than 2 bits**, relative to the ideal message length given the probabilistic model $\mathcal{H}$, $h(\mathbf{x} \mid \mathcal{H}) = \log[1/P(\mathbf{x} \mid \mathcal{H})]$.

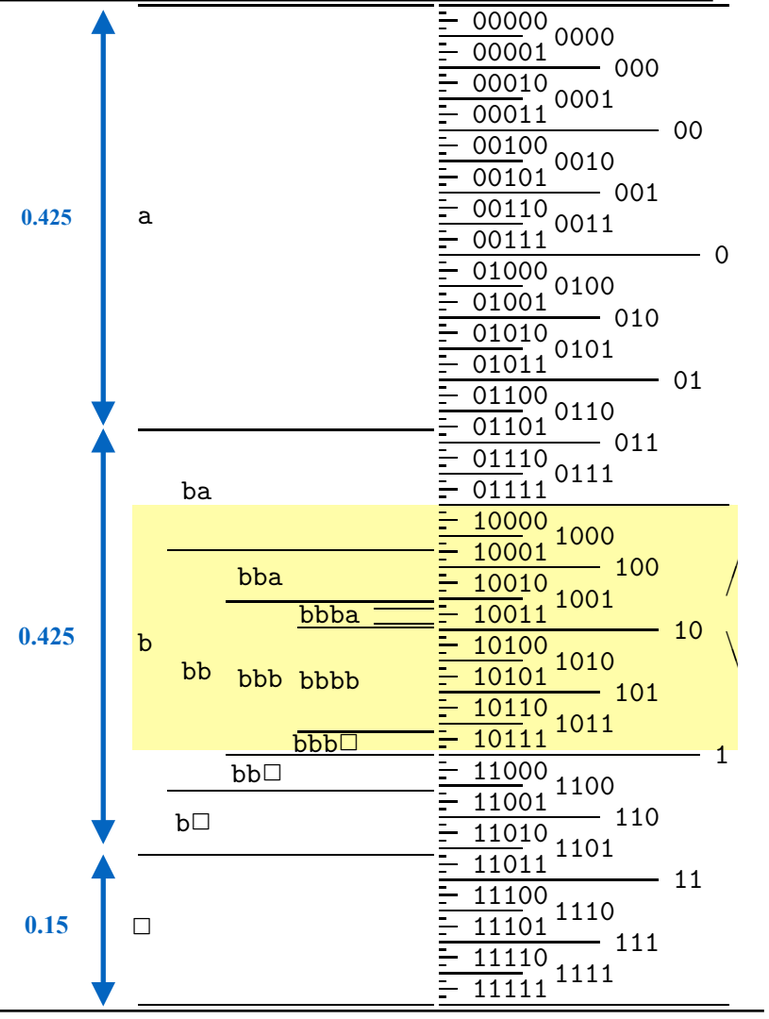
- **The message length is always within two bits of the Shannon information** content **of the entire source string,**

- **The expected message length is within two bits of the entropy of the entire message**

Decoding

- The decoder receives the string ‘100111101’ and passes along it one symbol at a time.
- First, the probabilities P(a), P(b), P(□) are computed using the identical program that the encoder used and the intervals ‘**a**’, ‘**b**’ and ‘□’ are deduced.

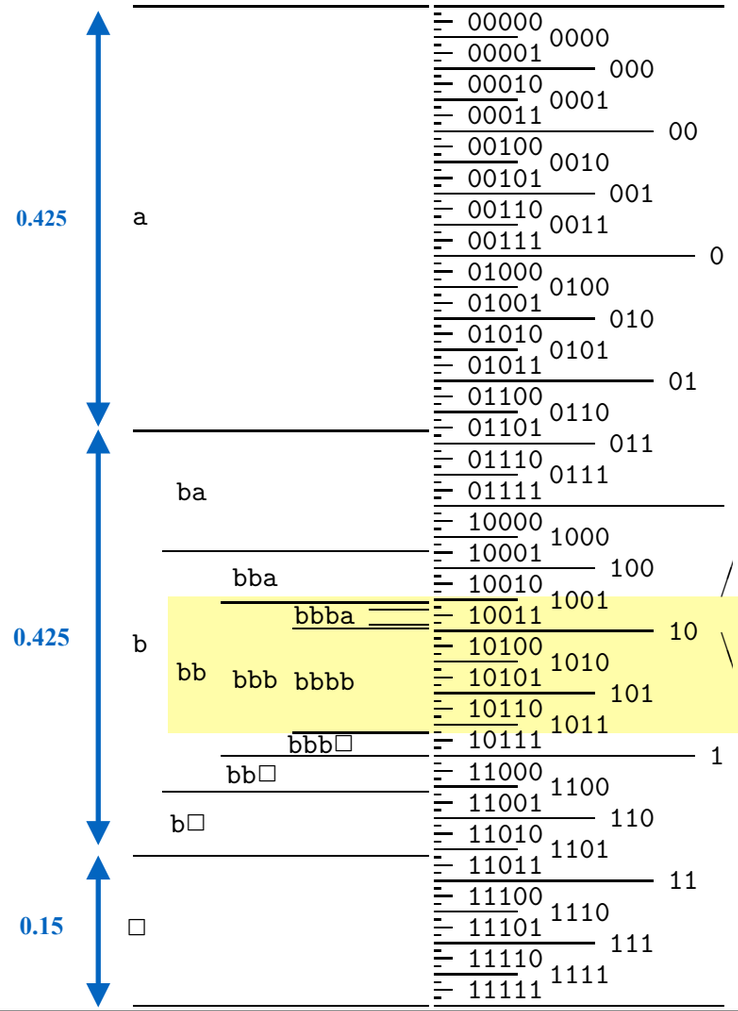
- Once the first two bits ‘**10**’ have been examined, it is certain that the original string must have been started with a ‘**b**’, since the interval ‘**10**’ lies wholly within interval ‘**b**’.



Decoding

- The decoder receives the string ‘100111101’ and passes along it one symbol at a time.

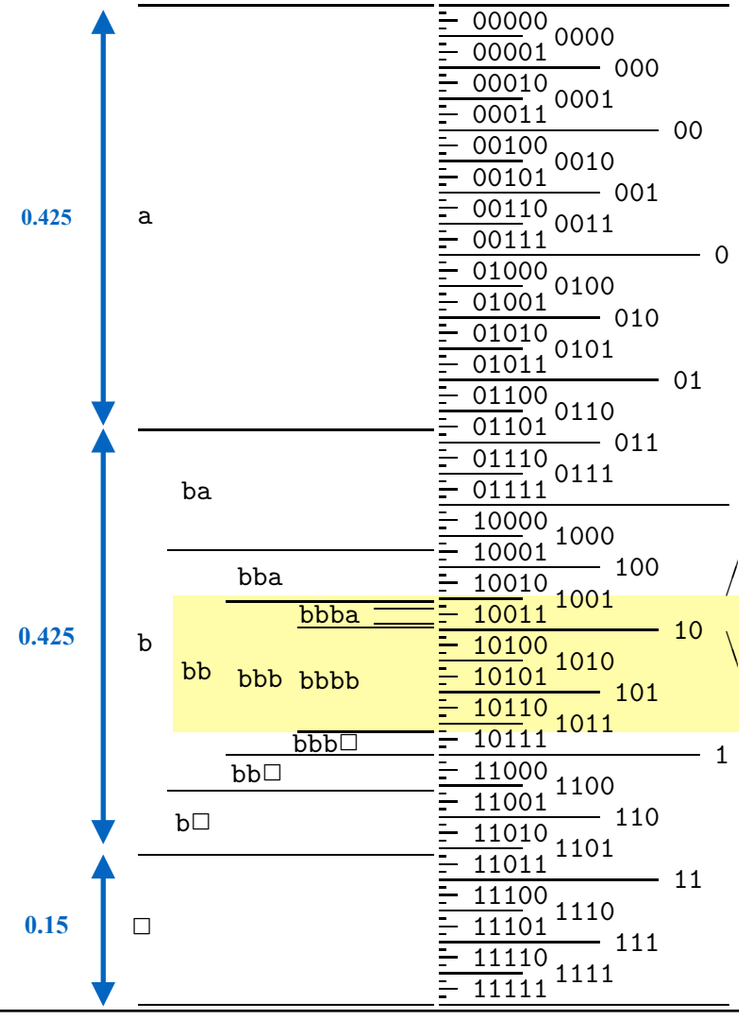
- The decoder can then use the model to compute $P(\mathbf{a} \mid \mathbf{b})$, $P(\mathbf{b} \mid \mathbf{b})$, $P(\square \mid \mathbf{b})$ and deduce the boundaries of the intervals ‘**ba**’, ‘**bb**’ and ‘**b** $\square$ ’.
- Continuing, we decode the second **b** once we reach ‘1001’



Decoding

- The decoder receives the string ‘100111101’ and passes along it one symbol at a time.

- The decoder can then use the model to compute $P(\mathbf{a} \mid \mathbf{b})$, $P(\mathbf{b} \mid \mathbf{b})$, $P(\square \mid \mathbf{b})$ and deduce the boundaries of the intervals ‘**ba**’, ‘**bb**’ and ‘**b** □’.
- Continuing, we decode the second **b** once we reach ‘1001’
- The third **b** once we reach ‘100111’,
- and so forth, with the unambiguous identification of ‘bbba □’.
- With the convention that ‘ □’ denotes the end of the message, the decoder knows to stop decoding.



Transmission of multiple files

- How might one use arithmetic coding to communicate several distinct files over the binary channel?
  - Once the □ character has been transmitted, the decoder is reset into its initial state.
  - **There is no transfer of the learnt statistics of the first file to the second file**.

- **If**, however, we did believe that **there is a relationship among the files** that we are going to compress, we could define our alphabet differently, **introducing a second end-of-file character that marks the end of the file but instructs the encoder and decoder to continue using the same probabilistic model.**

Comparing to Huffman - The big picture

- With arithmetic codes, notice that to communicate a string of N letters both the encoder and the decoder needed to compute only $N\,|A|$ conditional probabilities – the probabilities of each possible letter in each context actually encountered.
- This cost can be contrasted with the alternative of using a Huffman code with a large block size (in order to reduce the possible one-bit-per-symbol overhead), where **all block sequences that could occur** must be considered and their probabilities evaluated.
- Arithmetic coding is flexible:
  - It can be used with any source alphabet and any encoded alphabet
  - The size of the source alphabet and the encoded alphabet can change with time.
  - Can be used with any probability distribution, which can change from context to context.

Details of the Bayesian model

- The model is described using parameters $p_{\square}$, p_a and p_b,
- A bent coin labelled **a** and **b** is tossed some number of times l, which we don't know beforehand.
- The coin's probability of coming up a when tossed is p_a, and $p_b = 1 - p_a$; the parameters p_a, p_b are not known beforehand.
- The source string **s = baaba** $\square$ indicates that l was 5 and the sequence of outcomes was **baaba**.

Details of the Bayesian model

- It is assumed that the length of the string l has an exponential probability distribution

$$P(l) = (1 - p_{\square})^{l} p_{\square}$$

This distribution **corresponds to assuming a constant probability $p_{\square}$ for the termination symbol '□' at each character**.

- It is assumed that the non-terminal characters in the string are selected independently at random from an ensemble with probabilities $P = \{p_a, p_b\}$.

  - The probability p_a is **fixed throughout the string** to some **unknown value** that could be anywhere between 0 and 1.

  - The probability of an **a** occurring as the next symbol, given p_a, is $(1 - p_{\square})p_a$

Details of the Bayesian model

- The probability, given p_a, that an unterminated string of length F is a given string **s** that contains $\{F_a, F_b\}$ counts of the two outcomes is the Bernoulli distribution

$$P(\mathbf{s} \mid p_a, F) = p_a^{F_a}(1-p_a)^{F_b}$$

- We assume a uniform prior distribution for p_a,

$$P(p_\mathtt{a}) = 1, \qquad p_\mathtt{a} \in [0,1],$$

- The probability that the next character is **a** or **b** (assuming that it is not ‘ □ ’) is the Laplace’s rule

$$P_\mathrm{L}(\mathtt{a} \mid x_1, \ldots, x_{n-1}) = \frac{F_\mathtt{a} + 1}{F_\mathtt{a} + F_\mathtt{b} + 2}$$

Further applications of arithmetic coding

Efficient **generation of random samples**

- Arithmetic code also offers a way to generate random strings from a model.
- Imagine sticking a pin into the unit interval at random, that line having been divided into subintervals in proportion to probabilities p_i; the probability that your pin will lie in interval i is p_i.
- So to generate a sample from a model, all we need to do is **feed ordinary random bits into an arithmetic decoder for that model.**
- An infinite random bit sequence corresponds to the **selection of a point at random from the line [0,1),**
- **S**o the decoder will then select a string at random from the assumed distribution.

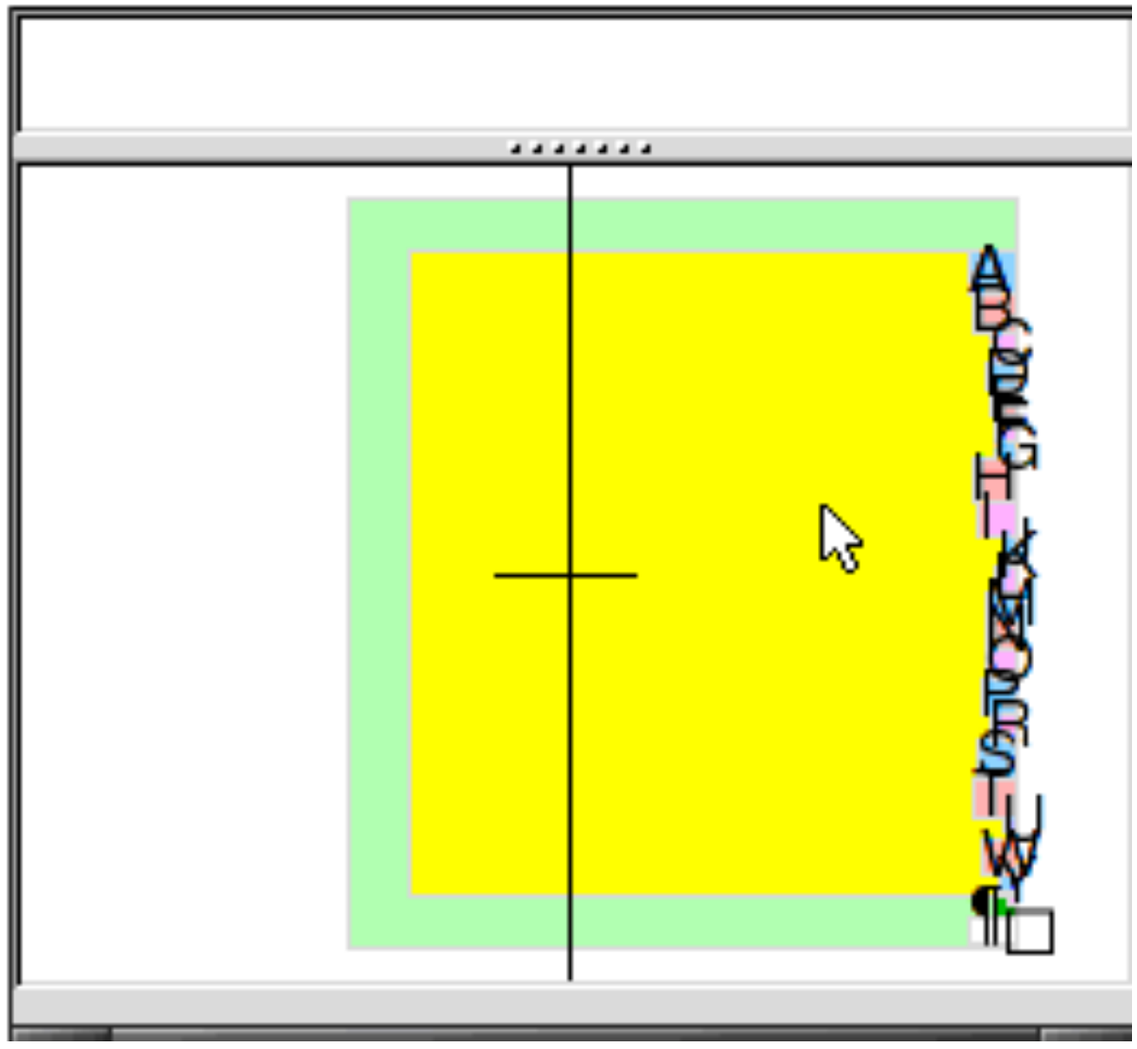
Take a look on random numbers generators

Efficient **data-entry devices**

- When we enter text into a computer, we make gestures of some sort – maybe we tap a keyboard, or scribble with a pointer, or click with a mouse;
- **an efficient text entry system** is one where the **number of gestures required to enter a given text string is small**.
- Writing can be viewed as an inverse process to data compression.
  - In **data compression**, the aim is to map a **given text string** into a **small number of bits**.
  - In **text entry**, we want a **small sequence of gestures** to produce our **intended text**.
- By **inverting an arithmetic coder**, we can obtain an information-efficient text entry device that is driven by continuous pointing gestures

Efficient data-entry devices

- By **inverting an arithmetic coder**, we can obtain an information-efficient text entry device that is driven by continuous pointing gestures (Ward et al. 2000)
- In this system, called Dasher, the user zooms in on the unit interval to locate the interval corresponding to their intended string,
- A language model (exactly as used in text compression) controls the sizes of the intervals.



See this video

More about Dasher And the impact

Dasher

Hell

Dasher’s Impact

Basic Lempel–Ziv algorithm

Basic Lempel–Ziv algorithm

- The method of compression is to **replace a substring** with a **pointer to an earlier occurrence of the same substring**.
- For example if the string is `1011010100010...`, we parse it into an ordered *dictionary* of **substrings that have not appeared before**:
  - λ - empty substring; λ as the first substring in the dictionary
  - 1, **1**011010100010
  - 0, ~~1~~**0**11010100010
  - 11, ~~10~~**11**010100010
  - 01, ~~1011~~**01**0100010
  - 010, 00, 10, . . . ~~101101~~**010**0010

Basic Lempel–Ziv algorithm

- For example if the string is `1011010100010`..., we parse it into an ordered *dictionary* of **substrings that have not appeared before**:
  - λ - empty substring λ as the first substring in the dictionary
  - 1, **1**011010100010
  - 0, ~~1~~**0**11010100010
  - 11, ..read a substring that has not been marked off before. **This substring is longer by one bit than a substring that has occurred earlier in the dictionary** ~~10~~**11**010100010
  - 01, ~~1011~~**01**0100010
  - 010, ~~101101~~**010**0010
  - 00, ~~10110101000~~**10**
  - 10, . . . ~~1011010100010~~

Basic Lempel–Ziv algorithm

- For example if the string is `1011010100010`..., we parse it into an ordered *dictionary* of **substrings that have not appeared before**:
  - λ - empty substring λ as the first substring in the dictionary
  - **1**, **1**011010100010
  - 0, ~~1~~**0**11010100010
  - **11**, ~~10~~**11**010100010
  - 01, ~~1011~~**01**00010
  - 010, ~~101101~~**010**0010
  - 00, ~~10110101000~~10
  - 10, . . . ~~1011010100010~~

Basic Lempel–Ziv algorithm

- For example if the string is `1011010100010`..., we parse it into an ordered *dictionary* of **substrings that have not appeared before**:
  - λ - empty substring λ as the first substring in the dictionary
  - 1, **1**011010100010
  - **0**, ~~1~~**0**11010100010
  - 11, ~~10~~**11**010100010
  - **01**, ~~1011~~**01**0100010
  - 010, ~~101101~~**010**0010
  - 00, ~~10110101~~**00**10
  - 10, . . . ~~10110101000~~**10**

Basic Lempel–Ziv algorithm

- For example if the string is `1011010100010`..., we parse it into an ordered *dictionary* of **substrings that have not appeared before**:
  - λ - empty substring λ as the first substring in the dictionary
  - 1, **1**011010100010
  - 0, ~~1~~**0**11010100010
  - 11, ~~10~~**11**010100010
  - **01**, ~~1011~~**01**0100010
  - **010**, ~~101101~~**010**0010
  - 00, ~~10110101~~**00**10
  - 10, . . . ~~10110101000~~**10**

Basic Lempel–Ziv algorithm

- For example if the string is `1011010100010...`, we parse it into an ordered *dictionary* of **substrings that have not appeared before**:
  - λ - empty substring λ as the first substring in the dictionary
  - 1, **1**011010100010
  - **0**, ~~1~~**0**11010100010
  - 11, ~~10~~**11**010100010
  - 01, ~~1011~~**01**0100010
  - 010, ~~101101~~**010**0010
  - **00**, ~~10110101000~~**10**
  - 10, . . .

Basic Lempel–Ziv algorithm

- For example if the string is `1011010100010`..., we parse it into an ordered *dictionary* of **substrings that have not appeared before**:
  - λ - empty substring λ as the first substring in the dictionary
  - **1**, ..read a substring that has not been marked off before. **This substring is longer by one bit than a substring that has occurred earlier in the dictionary**
  - 0,
  - 11,
  - 01,
  - 010,
  - 00,
  - **10**, . . .

1011010100010

~~1~~**0**11010100010

~~10~~**11**010100010

~~1011~~**01**0100010

~~101101~~**010**0010

~~10110101~~**00**10

~~101101010~~**10**

Basic Lempel–Ziv algorithm

- … read a substring that has not been marked off before. This **substring is longer by one bit than a substring that has occurred earlier in the dictionary**
- We can encode each substring by
  - giving a **pointer to the earlier occurrence** of that prefix and
  - then sending the **extra bit** by which the new **substring in the dictionary differs from the earlier** substring.
- If, at the nth bit, we have enumerated $s(n)$ substrings, then we can give the value of the pointer in $\lceil \log_2 s(n) \rceil$ bits.

Basic Lempel–Ziv algorithm

Source Substring	λ	1	0	11	01	010	00	10

1011010100010

~~1~~**0**11010100010

~~10~~**11**010100010

~~1011~~**01**0100010

~~101101~~**010**0010

~~10110101~~**00**10

~~1011010100~~**10**

Basic Lempel–Ziv algorithm

Source Substring	**λ**	**1**	**0**	**11**	**01**	**010**	**00**	**10**
$s(n)$	0	1	2	3	4	5	6	7
$s(n)_{\text{binary}}$	**000**	**001**	**010**	**011**	**100**	**101**	**110**	**111**

Basic Lempel–Ziv algorithm

Source Substring	**λ**	**1**	**0**	**11**	**01**	**010**	**00**	**10**
$s(n)$	0	1	2	3	4	5	6	7
$s(n)_{\text{binary}}$	**000**	**001**	**010**	**011**	**100**	**101**	**110**	**111**
(Pointer, bit)		**(, 1)**	**(0, 0)**	**(01, 1)**	**(10, 1)**	**(100, 0)**	**(010, 0)**	**(001, 0)**

1

1**00**

100**011**

100011**101**

100011101**1000**

1000111011000**0100**

10001110110000100**0010**

The encoding, in this simple case, is actually a **longer string than the source string. Why ?**

1011010100010

Basic Lempel–Ziv algorithm

- The encoding, in this simple case, is **actually a longer string than the source string**. Why ?
  - **It transmits unnecessary bits**
- Once **a substring in the dictionary has been joined there by both of its children, then we can be sure that it will not be needed** (except possibly as part of our protocol for terminating a message)
  - At that point we **could drop it from our dictionary of substrings** and **shuffle them all along one, thereby reducing the length of subsequent pointer messages.**
  - Equivalently, we could **write the second prefix into the dictionary at the point previously occupied by the parent**
- The transmission of the **new bit when the second time a prefix is used**, **we can be sure of the identity of the next bit**

Basic Lempel–Ziv algorithm

Source Substring	**λ**	**1**	**0**	**11**	**01**	**010**	**00**	**10**
$s(n)$	0	1	2	3	4	5	6	7
$s(n)_{\text{binary}}$	**000**	**001**	**010**	**011**	**100**	**101**	**110**	**111**
(Pointer, bit)		**(, 1)**	**(0, 0)**	**(01, 1)**	**(10, 1)**	**(100, 0)**	**(010, 0)**	**(001, 0)**

1

1**00**

100**011**

100011**101**

100011101**1000**

1000111011000**0100**

100011101100001000**0010**

The encoding, in this simple case, is actually a **longer string than the source string. Why ?**

101101010010

Basic Lempel–Ziv algorithm

Source Substring	λ	**1**	**0**	**11**	**01**	**010**	**00**	**10**
$s(n)$	0	1	2	3	4	5	6	7
$s(n)_{\text{binary}}$	**000**	**001**	**010**	**011**	**100**	**101**	**110**	**111**
(Pointer, bit)		**(, 1)**	**(0, 0)**	**(01, 1)**	**(10, 1)**	**(100, 0)**	**(010, 0)**	**(001, 0)**

1

1**00**

100**011**

100011**101**

100011101**1000**

1000111011000**0100**

100011101100001000**0010**

The encoding, in this simple case, is actually a **longer string than the source string. Why ?**

101101010010

Basic Lempel–Ziv algorithm

- The **decoder** again involves an identical twin at the decoding end who constructs the dictionary of substrings as the data are decoded.
- **100011101100001000010**

```
1 (0,0) (01,1) (10,1) (100,0) (010,0) (001,0)
```

- There are **many variations** on the Lempel–Ziv algorithm, all exploiting the same idea but using **different procedures for dictionary management**, etc.
- The resulting programs are fast, but their performance on compression of English text, although useful, **does not match the standards set in the arithmetic coding literature**.

Basic Lempel–Ziv: Theoretical properties

- The Lempel–Ziv algorithm is defined **without** making any mention of **a probabilistic model for the source**
- Yet, given any ergodic source (i.e., one that is memoryless on sufficiently long timescales), the Lempel–Ziv algorithm **can be proven asymptotically to compress down to the entropy of the source**.
  - This is why it is called a ‘universal’ compression algorithm
- It achieves its compression, however, only by memorizing substrings that have happened so that it has a short name for them the next time they occur. **The asymptotic timescale on which this universal performance is achieved may, for many sources, be unfeasibly long**, because the number of typical substrings that need memorizing may be enormous.

Basic Lempel–Ziv: Theoretical properties

- The Lempel–Ziv algorithm is defined **without** making any mention of **a probabilistic model for the source**
- The **useful performance of the algorithm in practice** is a reflection of the fact that **many files contain multiple repetitions of particular short sequences of characters, a form of redundancy to which the algorithm is well suited**.
- In principle, one can design **adaptive probabilistic models**, and thence arithmetic codes, that are '**universal**', that is, models that will asymptotically compress any source in some class to within some factor (preferably 1) of its entropy. **However**, for practical purposes such universal models can **only be constructed if the class of sources is severely restricted.**

Basic Lempel–Ziv: Theoretical properties

- In principle, one can design **adaptive probabilistic models**, and thence arithmetic codes, that are '**universal**', that is, models that will asymptotically compress any source in some class to within some factor (preferably 1) of its entropy. **However**, for practical purposes such universal models can **only be constructed if the class of sources is severely restricted.**

- **A general purpose compressor that can discover the probability distribution of any source would be a general purpose artificial intelligence!**

Demonstration

Check some resources

- An interactive aid for exploring arithmetic coding, dasher.tcl (in TCL)
- A demonstration arithmetic-coding software package written by Radford Neal consists of encoding and decoding modules to which the user adds a module defining the probabilistic model.
  - Radford Neal's package includes a simple adaptive model similar to the Bayesian model
- A state-of-the-art compressor for documents containing text and images, DjVu, uses arithmetic coding
  - It uses an approximate arithmetic coder for binary alphabets called the **Z-coder** (Bottou et al., 1998), which is much faster. The **adaptive model adapts only occasionally**, with the decision about when to adapt being pseudo-randomly controlled. (see also https://www.cuminas.jp/en/)

Check some resources

- The **JBIG image compression standard** for binary images uses **arithmetic coding** with a context-dependent model, which adapts using a rule similar to Laplace's rule
- **PPM** (Teahan, 1995) is a leading method for text compression, and it uses arithmetic coding
- There are many Lempel–Ziv-based programs.
  - **gzip** is based on a version of Lempel–Ziv called 'LZ77' (Ziv and Lempel, 1977).
  - **compress** is based on 'LZW' (Welch, 1984).
  - **bzip** is a *block-sorting file compressor* (Burrows and Wheeler, 1994). This method is not based on an explicit probabilistic model, and it only works well for files larger than several thousand characters; but in practice it is a very effective compressor for files in which the context of a character is a good predictor for that character.

Compression of a text file

- Compression achieved when these programs are applied to the LATEX file containing a text file, of size 20,942 bytes

Method	Compression time / sec	Compressed size (%age of 20,942)	Uncompression time / sec
Laplace model	0.28	12 974 (61%)	0.32
`gzip`	0.10	8 177 (39%)	**0.01**
`compress`	0.05	10 816 (51%)	0.05
`bzip`		7 495 (36%)	
`bzip2`		7 640 (36%)	
`ppmz`		**6 800 (32%)**	

Compression of a sparse file

- Compression achieved when these programs are applied to a text file containing 10^6 characters, each of which is either 0 and 1 with probabilities 0.99 and 0.01.

Method	Compression time / sec	Compressed size / bytes	Uncompression time / sec
Laplace model	0.45	14 143 (1.4%)	0.57
`gzip`	0.22	20 646 (2.1%)	0.04
`gzip --best+`	1.63	15 553 (1.6%)	0.05
`compress`	0.13	14 785 (1.5%)	0.03
`bzip`	0.30	10 903 (1.09%)	0.17
`bzip2`	0.19	11 260 (1.12%)	0.05
`ppmz`	533	**10 447 (1.04%)**	535

- An ideal model for this source would compress the file into about $10^6 \, H_2(0.01)/8 \simeq$ **10 100 bytes**.

Summary

Summary

Fixed-length block codes (Chapter 4). These are mappings from a fixed number of source symbols to a fixed-length binary message. Only a tiny fraction of the source strings are given an encoding. These codes were fun for identifying the entropy as the measure of compressibility but they are of little practical use.

Summary

Symbol codes (Chapter 5). Symbol codes employ a variable-length code for each symbol in the source alphabet, the codelengths being integer lengths determined by the probabilities of the symbols. Huffman's algorithm constructs an optimal symbol code for a given set of symbol probabilities.

Every source string has a uniquely decodeable encoding, and if the source symbols come from the assumed distribution then the symbol code will compress to an expected length per character L lying in the interval $[H, H+1)$. Statistical fluctuations in the source may make the actual length longer or shorter than this mean length.

If the source is not well matched to the assumed distribution then the mean length is increased by the relative entropy D_{KL} between the source distribution and the code's implicit distribution. For sources with small entropy, the symbol has to emit at least one bit per source symbol; compression below one bit per source symbol can be achieved only by the cumbersome procedure of putting the source data into blocks.

Summary

Stream codes. The distinctive property of stream codes, compared with symbol codes, is that they are not constrained to emit at least one bit for every symbol read from the source stream. So large numbers of source symbols may be coded into a smaller number of bits. This property could be obtained using a symbol code only if the source stream were somehow chopped into blocks.

- Arithmetic codes combine a probabilistic model with an encoding algorithm that identifies each string with a sub-interval of $[0, 1)$ of size equal to the probability of that string under the model. This code is almost optimal in the sense that the compressed length of a string $\mathbf{x}$ closely matches the Shannon information content of $\mathbf{x}$ given the probabilistic model. Arithmetic codes fit with the philosophy that good compression requires *data modelling*, in the form of an adaptive Bayesian model.

Summary

Stream codes. The distinctive property of stream codes, compared with symbol codes, is that they are not constrained to emit at least one bit for every symbol read from the source stream. So large numbers of source symbols may be coded into a smaller number of bits. This property could be obtained using a symbol code only if the source stream were somehow chopped into blocks.

- Lempel–Ziv codes are adaptive in the sense that they memorize strings that have already occurred. They are built on the philosophy that we don't know anything at all about what the probability distribution of the source will be, and we want a compression algorithm that will perform reasonably well whatever that distribution is.

Both arithmetic codes and Lempel–Ziv codes will fail to decode correctly if any of the bits of the compressed file are altered. So if compressed files are to be stored or transmitted over noisy media, error-correcting codes will be

Further Reading and Summary

Q&A

Further Reading

- **Recommend Readings**
  - Information Theory, Inference, and Learning Algorithms from David MacKay, 2015, pages 102 - 124.
- **Supplemental readings:**

What you should know

- The main ideia of arithmetic codes IS BASED ON predictive distribution over all possible values of the next symbol.
- The relation with the guessing (the next symbol) game
- Given a model and a source string how to code and decode.
- Given a model, how to compute the probability intervals.
- Understand the simple Bayesian model and its assumptions.
- Other applications of arithmetic coding

Further Reading and Summary

Q&A