Coding Memoryless Sources

(Chapter 3)

Chapter Overview

After this chapter you should have understood a number of algorithms, which means:

- you are able to execute by hand a (compression or decompression) algorithm on a given input.
- you are able to describe the (compression or decompression) algorithm in your own words, in a reasonably precise manner.
- you are able to understand and explain, why these algorithms perform well. Your explanation and understanding should convey the intuition in a reasonably precise manner but need not be mathematically rigorous.
- you should know examples of the use of these algorithms in software products, including an explanation of why they are effective in these application areas.

The algorithms you should have understood are:

- Huffman coding, including adaptive and blocked variants.
- Arithmetic coding, including adaptive variants.

Lectures Overview

This chapter is scheduled to take about four lectures. The main topics are:

- Huffman coding.
- Adaptive Huffman coding.
- Limitations of Huffman coding.
- Arithmetic coding.

Outline

- Huffman coding algorithm.
- Huffman coding's performance.
- Resolving ambiguities: canonical minimium-variance codes.

Huffman coding

Given a memoryless source S:

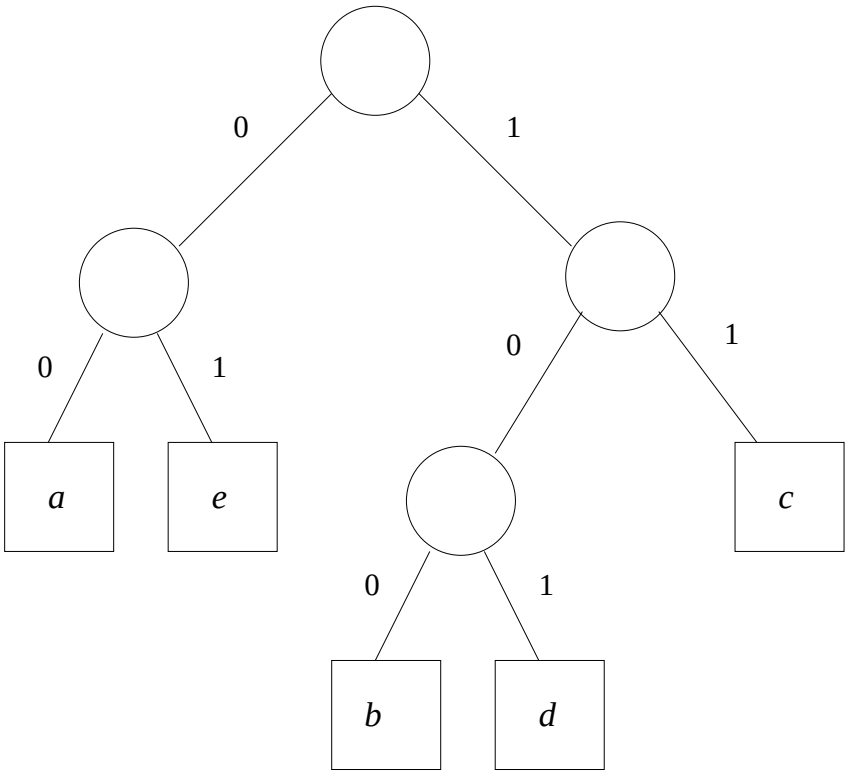
- Symbols $\sigma_1, \ldots, \sigma_n$;
- Probabilities $p_1, \ldots, p_n$, entropy $H(S)$.

Huffman coding outputs a "complete binary tree":

1. Internal nodes (non-leaves) have exactly two children and are not labelled with a symbol. The edges out of each internal node are labelled 0 and 1.
2. There are n leaves, each labelled with a symbol.

▷ The code for a symbol σ is obtained by walking down from the root of the tree to σ and reading off the bits.

Example of Huffman Output



a	00
b	100
c	11
d	101
e	01

▷ HC produces prefix codes (uniquely decodable)

Algorithm Overview

- Algorithm, as we describe it, has two main phases: a *combination phase* and a *tree-build* phase.
- The combination phase takes the input symbols, and combines them pairwise until one symbol is left.
- The tree-build phase "uncombines" them, reversing the combination steps, and obtains the complete binary tree.
- After the tree is obtained, the codes are read off.

The combination phase

1. Create a list of all the symbols, placed in decreasing order of probability.
2. If there is only one symbol in list, then OUTPUT the symbol. The combination phase is finished and exits.
3. If there is more than one symbol in the list, then take the last two symbols σ and σ' in the list, and let their probabilities p be and p'. (Since the list is ordered by probability, the last elements in the list have the lowest probability.)
   - 3.1 Delete σ and σ' from the the list.
   - 3.2 Create a new symbol $[\sigma\sigma']$.
   - 3.3 Give the new symbol the probability $p + p'$.
   - 3.4 Insert the new symbol into the list, ensuring the list remains in decreasing order of probability.
4. Go to (2).

Combination phase example

MLS, symbols $\{a, b, c, d, e, f\}$, probs $0.25, 0.2, 0.15, 0.15, 0.15, 0.1$.

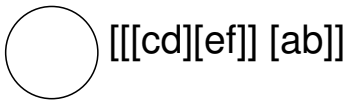
a	0.25	$[ef]$	0.25	$[cd]$	0.30	$[ab]$	0.45	$[[cd][ef]]$	0.55
b	0.20	a	0.25	$[ef]$	0.25	$[cd]$	0.30	$[ab]$	0.45
c	0.15	b	0.20	a	0.25	$[ef]$	0.25		
d	0.15	c	0.15	b	0.20				
e	0.15	d	0.15						
f	0.10								

Final combination step: $[[[cd][ef]][ab]]$, probability 1.00.

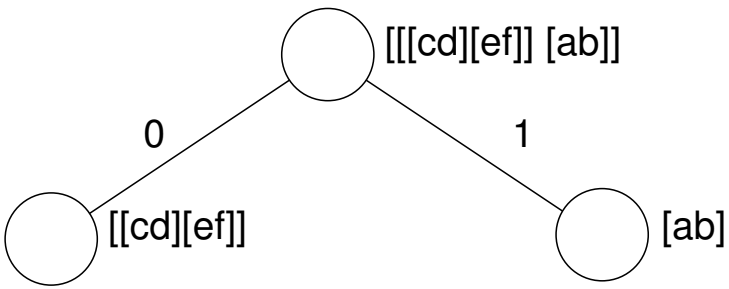
Tree-Build Phase

1. If the current symbol is a symbol of the original alphabet (i.e. not a combined symbol) output a leaf labelled with this symbol.
2. Otherwise, the current symbol must be a combined symbol and must look like $[\sigma\sigma']$.
   - 2.1 Create a new internal node v
   - 2.2 Recurse using σ as the current symbol (the left part of the combined symbol). Take the tree returned by the recursive call and make it the left child of the node v, and label the edge with a **0**.
   - 2.3 Recurse using σ' as the current symbol (the right part of the combined symbol). Take the tree returned by the recursive call and make it a the right child of the node v, and label the edge with a **1**.

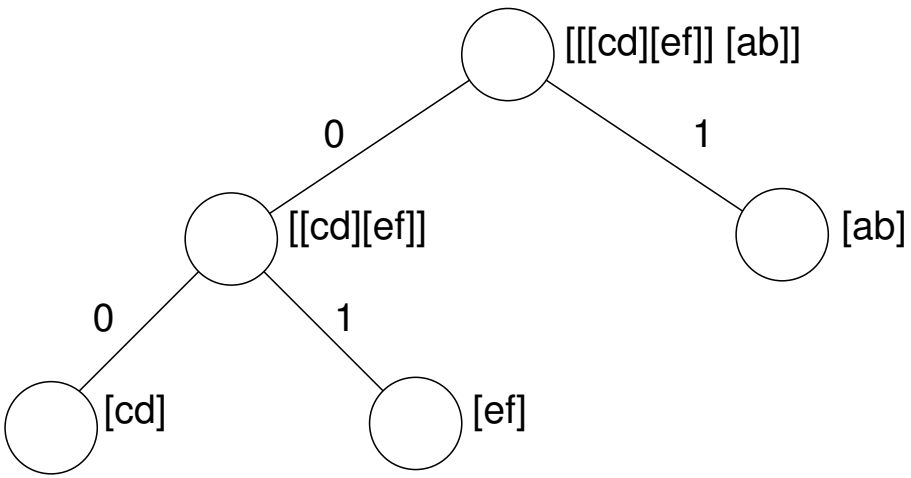
Example tree-build phase



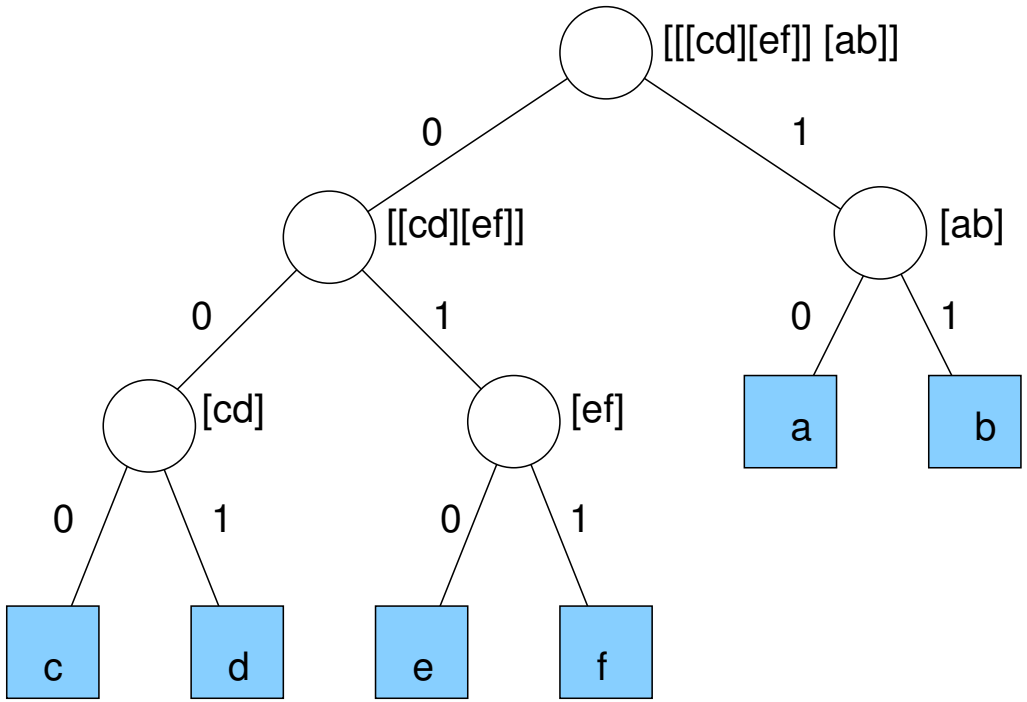
Example tree-build phase



Example tree-build phase



Example tree-build phase



a	10
b	11
c	000
d	001
e	010
f	011

Final output

Symbol	Code	Probability
a	10	0.25
b	11	0.20
c	000	0.15
d	001	0.15
e	010	0.15
f	011	0.10

- If a symbol is frequent, it is combined *later*. This means it is "un-combined" *earlier*, so its code is shorter.
  - ▷ More frequent symbols have shorter codes.
- The average code length $0.25 \times 2 + 0.2 \times 2 + 0.15 \times 3 + 0.15 \times 3 + 0.15 \times 3 + 0.1 \times 3 = 2.55$ bits/symbol.

Huffman Coding Performance

Input: a memoryless source S, with symbols $\sigma_1, \ldots, \sigma_n$, probabilities $p_1, \ldots, p_n$, entropy $H(S)$.

Suppose HC outputs codes of lengths $l_1, \ldots, l_n$ for n input symbols. Let $\bar{l} = p_1 l_1 + \ldots + p_n l_n$ (avg code length of HC).

(a) $H(S) \leq \bar{l} < H(S) + 1$ always.

▷ In the example above, $H(S) = 2.53$ bits/symbol (2dp), $\bar{l} = 2.55$ bits/symbol.

$$H(S) = 2.53 \leq \bar{l} = 2.55 < 3.53$$

(b) $\bar{l} = H(S)$ when for all i, $l_i = -\log_2 p_i$. (Happens only if **all** probabilities are powers of 2.)

(c) HC codes are *minimum-redundancy*, i.e. no symbol-by-symbol codes can be better.

Ambiguities

- There can be items with equal probabilities.
- The roles of 0 and 1 are interchangeable.
- In many cases (e.g. adaptive Huffman coding) it is useful for coder and decoder to obtain *the same code* starting from the same probabilities.
    - We need *canonical* codes.

0 or 1?

1. When combining two symbols the higher/first one comes first.

a	0.25	$[ef]$	0.25	a	0.25	$[fe]$	0.25 $\times$
b	0.20	a	0.25	b	0.20	a	0.25
c	0.15	b	0.20	c	0.15	b	0.20
d	0.15	c	0.15	d	0.15	c	0.15
e	0.15	d	0.15	e	0.15	d	0.15
f	0.10			f	0.10		

2. When un-combining, the first gets the 0 branch.

Ambiguities

There can be items with equal probabilities.

▷A newly-created element goes as high in the list as possible.

a	0.25	$[ef]$	0.25	a	0.25	a	0.25
b	0.20	a	0.25	b	0.20	$[ef]$	0.25 ×
c	0.15	b	0.20	c	0.15	b	0.20
d	0.15	c	0.15	d	0.15	c	0.15
e	0.15	d	0.15	e	0.15	d	0.15
f	0.10			f	0.10		

▷ Produces *minimum-variance* codes.

▷ This rule does not change *average* code-length.

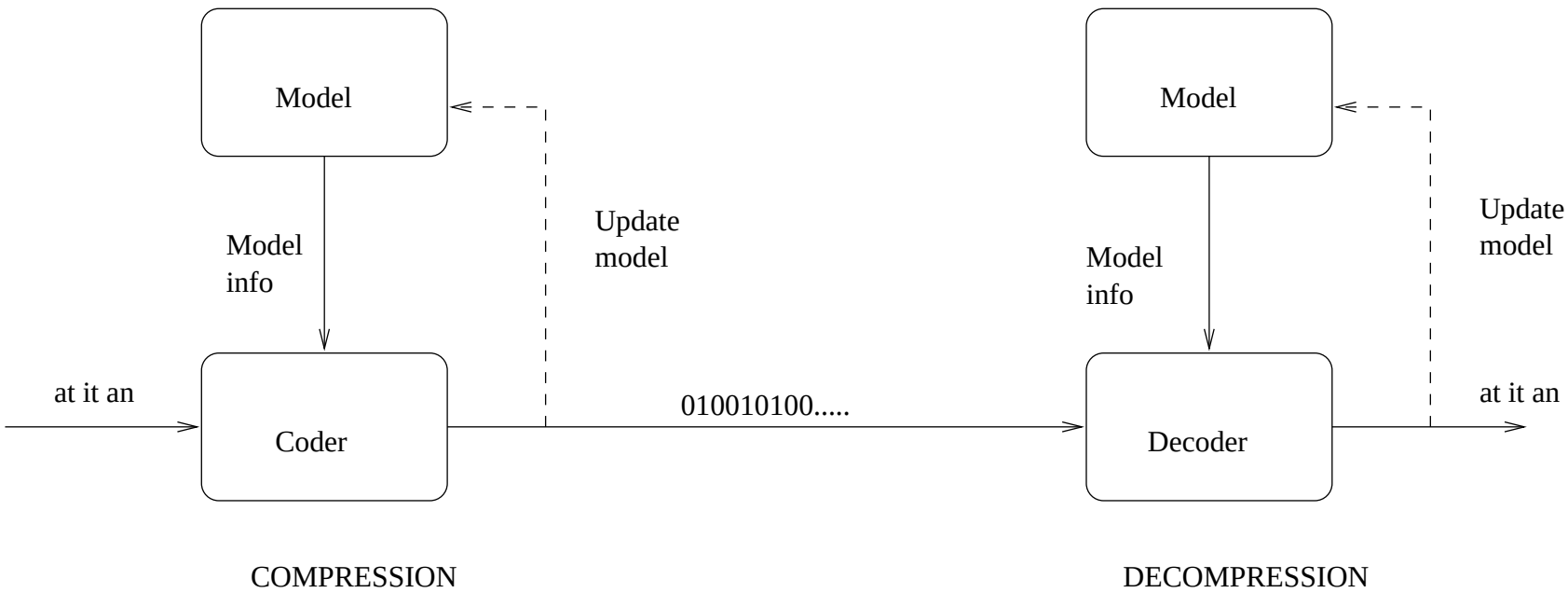
Outline

- Adaptive algorithms (review).
- Adaptive Huffman coding.

Adaptiveness (review)

- How to deal with inputs that do not fit the model?
  - Letter frequencies in model from English, but input is Polish text.
- Adaptive algorithms learn model parameters from input.
  - Often start with "empty" model;
  - Have start-up cost but are flexible;
  - Maintaining synchronization with de-coder is tricky (de-coder only sees compressed data).

Architecture of Adaptive Compressors



Adaptive Huffman Coding

Input is a sequence of symbols from some source S. The source has an alphabet $A = \{\sigma_1, \ldots, \sigma_n\}$ and probabilities $p_1, \ldots, p_n$.

- Assume that coder does not know either A or $p_1, \ldots, p_n$.
- However, for this to work, the coder should know something about A. E.g.:
  - The raw data is a set of ASCII characters, but the source is modelling a file full of numbers. So the alphabet A of the source could be $0, \ldots, 9$, the space character, the newline character, and say a decimal point, comma and $+/-$ signs.
  - The raw data is a set of UTF-8 characters, but the source is modelling a Spanish text, so A comprises the Latin characters together with accented characters such as á, etc.

Adaptive Huffman Coding

- The coder starts with an empty model, and uses set $\text{SEEN} \subset A$ of symbols seen by the coder so far.
- Instead of probabilities, the coder uses *weights*.
  - Weight of symbol: # times symbol seen so far.
    - Weight of symbol i is denoted by W_i.
  - Prob of symbol $\propto$ weight of symbol.

$$p_i = \frac{W_i}{W_1 + W_2 + \ldots + W_n}$$

- For HC, only the *order* of the list matters during combination stage.
  - ▷ Use only weights, not probabilities.
  - ▷ Update weights $\equiv$ update model.

Problems

- Coder and decoder start with empty model. All weights zero, no symbols seen.
- How can the coder add symbols to SEEN and tell the decoder?
- Normal mode of operation of coder and decoder:
  - Coder sends a series of Huffman codes, decoder decodes a series of Huffman codes.
  - If coder sees a new symbol and sends an ASCII (e.g. 01000000 = ASCII(d) or UTF-8 code, then the decoder will decode incorrectly
  - Solution: use symbol $\dagger \notin A$ as 'escape' symbol.

Encoding

Encoder works as follows. Assume raw input is ASCII.

- Keeps Huffman codes for all symbols in SEEN(including †).
- Reads the next input symbol *a* and then:
  - Normal-mode processing: output Huffman code for *a*.
  - New symbol mode processing, where $a \notin$ SEEN. Output *Huffman code for* † and then ASCII code of *a*. Add *a* to SEEN.
- Recompute the model: update the weights and rebuild the tree (called `update_tree` in pseudocode).
  - Weight of † is always 0. This is a design decision, but is also consistent with the definition that the weight of a symbol is the number of times it is seen in the input.

Encoding (Pseudo-code)

```
1. Initialize:
    set Huffman tree T to a tree with one node;
    set SEEN := {†};
2. while (more characters remain) do
     a := next_symbol_in_text();
     if (a in SEEN) then
       output h(a);
     else
       output h(†) followed by ASCII(a);
     T := update_tree(T);
   end
```

NOTE: `h(a)` means Huffman code of `a`.

Encoding (Example)

Pick up the algorithm after it has read 5 symbols, 3 *a*s and 2 *b*s.
Next input: *bc* . . .

```
CURRENT STATE: SEEN = {a, b, †}
               w(a) = 3, w(b) = 2, w(†) = 0.
               h(a) = 0, h(b) = 10, h(†) = 11.

1. READ b, b in SEEN
OUTPUT 10                    // h(b) = 10
SEEN stays the same.
w(a) = 3, w(b) = 3, w(†) = 0
h(a) = 1, h(b) = 00, h(†) = 01.

2. READ c, c not in SEEN.
OUTPUT 01 ASCII(c)           // h(†) = 01
SEEN = {a, b, c, †}
w(a) = 3, w(b) = 3, w(c) = 1, w(†) = 0
h(a) = 1, h(b) = 00, h(c) = 010, h(†) = 011.
```

Decoding

Decoder works as follows. Assume raw input is ASCII.

- Keeps Huffman codes for all symbols in SEEN(including †).
- Reads the next Huffman code, and decodes it. Let the answer be *a*.
  - *a* is a normal symbol: OUTPUT it.
  - *a* is †. *Read an ASCII code from input*. Let the symbol you have read be *x*. Add *x* to SEEN.
- Recompute the model: update the weights and rebuild the tree (called `update_tree` in pseudocode).

Decoding (Pseudo-code)

```
1. Initialize:
     set Huffman tree T to a tree with one node;
     set SEEN := {†};
2. while (more bits remain) do
     a := huffman_next_sym();
     if (a == †) then a := read_unencoded_sym();
     output a;
     T := update_tree(T);
   end
```

Decoding (Example)

Pick up the algorithm after it has read 5 symbols, 3 *a*s and 2 *b*s.
Next input: 100100111111...

```
CURRENT STATE: SEEN = {a, b, †}
               w(a) = 3, w(b) = 2, w(†) = 0.
               h(a) = 0, h(b) = 10, h(†) = 11.

1. READ 10, decode b.  // h(b) = 10
OUTPUT b
SEEN stays the same.
w(a) = 3, w(b) = 3, w(†) = 0
h(a) = 1, h(b) = 00, h(†) = 01.
2. READ 01, decode †  // h(†) = 01
READ 8 bit ASCII code 00111111 = c
OUTPUT c
SEEN = {a, b, c, †}
w(a) = 3, w(b) = 3, w(c) = 1, w(†) = 0
h(a) = 1, h(b) = 00, h(c) = 010, h(†) = 011.
```

Outline

- Limitations of Huffman codes.
- Blocked Huffman codes.

Limitations of Huffman coding

- Optimal only when all probabilities are powers of 2.
- Not too far from entropy for large alphabets and roughly even distribution.
  E.g a, b, c, d, e, f, probs $0.25, 0.2, 0.15, 0.15, 0.15, 0.10$.
  $H(S) = 2.53$ bits/symbol.
  Codes: $a = 10$; $b = 11$; $c = 000$; $d = 001$; $e = 010$; $f = 011$.
  Avg. len.: $2 \times 0.25 + 2 \times 0.2 + 3 \times 3 \times 0.15 + 3 \times 0.1 = 2.55$ bits/symbol
  $\triangleright \sim 1\%$ non-optimal.
- Binary alphabet?
  E.g a, b, probs $0.8, 0.2$. $H(S) = 0.72$ bits/symbol.
  Codes: $a = 0$; $b = 1$; $\Rightarrow$ Avg len $= 1$ bit/symbol.
  $\triangleright \sim 39\%$ non-optimal.

Blocked Huffman coding

Increase alphabet size by combining multiple symbols.
E.g. $\Pr[a] = 0.8, \Pr[b] = 0.2$.
$w = aa, x = ab, y = ba, z = bb$.
$\Pr[w] = \Pr[a] \times \Pr[a] = 0.64$.
$\Pr[x] = \Pr[a] \times \Pr[b] = 0.16$.
$\Pr[y] = \Pr[b] \times \Pr[a] = 0.16$.
$\Pr[z] = \Pr[b] \times \Pr[b] = 0.04$.
Huffman Code: $w = 0$, $x = 11$, $y = 100$, $z = 101$.
Avg. length $= 0.64 \times 1 + 0.16 \times 2 + 0.16 \times 3 + 0.04 \times 3 = 1.56$ bits/doubled-up symbol (2dp).
Average bits/original symbol: $1.56/2 = 0.78$. $H(S) = 0.78$ bits/symbol (2dp)

Limitations of Blocking

- Huffman coding a block of k symbols at a time gets closer to entropy.
- MLS S, $\Pr[a] = 0.8, \Pr[b] = 0.2$.
  - Coding 2 symbols at a time 0.78 bits/symbol. $H(S) = 0.72$ bits/symbol.
- MLS S, $\Pr[a] = 0.9, \Pr[b] = 0.1$.
  - Coding 2 symbols at a time 0.65 bits/symbol. $H(S) = 0.46$ bits/symbol.
  - Need to code 3 symbols at a time.
- The more skewed the original MLS, the more the block size.
- Alphabet size increases exponentially with the block size :(
- Not clear how to use blocking with an adaptive algorithm.
- Not clear how to use blocking to handle a Markov source.

Arithmetic Coding

- HC is non-optimal because it codes one symbol at a time, but no algorithm is better if coding one symbol at a time.
  - ▷ Arithmetic coding codes sequence of symbols at a time and is optimal.
- Huffman can't adapt rapidly to drastic probability changes (e.g Markov sources).
  - ▷ Arithmetic coding can.

Simplified Arithmetic Coding

In this module we teach a simplified version of arithmetic coding.

- We output the coded file only after seeing *entire* input.
  - Unrealistic for a number of reasons. Holding coded file in memory (?), precision limitations of floating-point arithmetic.
- We output a sequence of *decimal* digits, which is viewed as decimal number between 0 and 1.
  $\triangleright$ $563212\ldots \equiv 0.563212\ldots$.
  - Unrealistic since the output should be in binary.
- We assume that the decoder is told the length of the input sequence of symbols.
  - Unrealistic e.g. for streaming data.
- Some notation:
  - "One-sided open" real interval $[r, s) = \{x \mid r \leq x < s\}$
  - $0.43 \in [0.1, 0.6), 0.1 \in [0.1, 0.6), 0.6 \notin [0.1, 0.6)$
  - *Width* or *size* of interval $[r, s) = s - r$. Size of $[0.1, 0.6) = 0.5$.

Basic Idea

MLS with $A = \{a, b\}$, $\Pr[a] = 1/4$, $\Pr[b] = 3/4$.
We want to encode 2 symbols as a number in $[0, 1)$.

- $aa \rightarrow$ output a value from $[0, .0625)$;
- $ab \rightarrow$ output a value from $[.0625, 0.25)$;
- $ba \rightarrow$ output a value from $[.25, .4375)$;
- $bb \rightarrow$ output a value from $[.4375, 1)$.

We want to code ab.

Basic Idea

MLS with $A = \{a, b\}$, $\Pr[a] = 1/4$, $\Pr[b] = 3/4$.
We want to encode 2 symbols as a number in $[0, 1)$.

- $aa \rightarrow$ output a value from $[0, .0625)$;
- $ab \rightarrow$ output a value from $[.0625, 0.25)$;
- $ba \rightarrow$ output a value from $[.25, .4375)$;
- $bb \rightarrow$ output a value from $[.4375, 1)$.

We want to code ab. Output $1 \rightarrow 0.1$

Basic Idea

MLS with $A = \{a, b\}$, $\Pr[a] = 1/4$, $\Pr[b] = 3/4$.
We want to encode 2 symbols as a number in $[0, 1)$.

- $aa \rightarrow$ output a value from $[0, .0625)$;
- $ab \rightarrow$ output a value from $[.0625, 0.25)$;
- $ba \rightarrow$ output a value from $[.25, .4375)$;
- $bb \rightarrow$ output a value from $[.4375, 1)$.

We want to code aa.

Basic Idea

MLS with $A = \{a, b\}$, $\Pr[a] = 1/4$, $\Pr[b] = 3/4$.
We want to encode 2 symbols as a number in $[0, 1)$.

- $aa \rightarrow$ output a value from $[0, .0625)$;
- $ab \rightarrow$ output a value from $[.0625, 0.25)$;
- $ba \rightarrow$ output a value from $[.25, .4375)$;
- $bb \rightarrow$ output a value from $[.4375, 1)$.

We want to code aa. Output $0 \rightarrow 0.0$.

Basic Idea

MLS with $A = \{a, b\}$, $\Pr[a] = 1/4$, $\Pr[b] = 3/4$.
We want to encode 2 symbols as a number in $[0, 1)$.

- $aa \rightarrow$ output a value from $[0, .0625)$;
- $ab \rightarrow$ output a value from $[.0625, 0.25)$;
- $ba \rightarrow$ output a value from $[.25, .4375)$;
- $bb \rightarrow$ output a value from $[.4375, 1)$.

We want to decode $5 \rightarrow .5$.

Basic Idea

MLS with $A = \{a, b\}$, $\Pr[a] = 1/4$, $\Pr[b] = 3/4$.
We want to encode 2 symbols as a number in $[0, 1)$.

- $aa \rightarrow$ output a value from $[0, .0625)$;
- $ab \rightarrow$ output a value from $[.0625, 0.25)$;
- $ba \rightarrow$ output a value from $[.25, .4375)$;
- $bb \rightarrow$ output a value from $[.4375, 1)$.

We want to decode $5 \rightarrow .5$. Output bb.

Basic Idea

MLS with $A = \{a, b\}$, $\Pr[a] = 1/4$, $\Pr[b] = 3/4$.
We want to encode 2 symbols as a number in $[0, 1)$.

- $aa \rightarrow$ output a value from $[0, .0625)$;
- $ab \rightarrow$ output a value from $[.0625, 0.25)$;
- $ba \rightarrow$ output a value from $[.25, .4375)$;
- $bb \rightarrow$ output a value from $[.4375, 1)$.

We want to decode $25 \rightarrow 0.25$

Basic Idea

MLS with $A = \{a, b\}$, $\Pr[a] = 1/4$, $\Pr[b] = 3/4$.
We want to encode 2 symbols as a number in $[0, 1)$.

- $aa \rightarrow$ output a value from $[0, .0625)$;
- $ab \rightarrow$ output a value from $[.0625, 0.25)$;
- $ba \rightarrow$ output a value from $[.25, .4375)$;
- $bb \rightarrow$ output a value from $[.4375, 1)$.

We want to decode $25 \rightarrow 0.25$ Output ba.

Basic Idea

MLS with $A = \{a, b\}$, $\Pr[a] = 1/4$, $\Pr[b] = 3/4$.
We want to encode 2 symbols as a number in $[0, 1)$.

- $aa \rightarrow$ output a value from $[0, .0625)$;
- $ab \rightarrow$ output a value from $[.0625, 0.25)$;
- $ba \rightarrow$ output a value from $[.25, .4375)$;
- $bb \rightarrow$ output a value from $[.4375, 1)$.

Works if all intervals are *disjoint*, i.e. have no numbers in common.

Basic Idea: 2

MLS with $A = \{a, b\}$, $\Pr[a] = 1/4$, $\Pr[b] = 3/4$.
We want to encode 2 symbols as a number in $[0, 1)$.

- $aa \rightarrow$ output a value from $[0, .0625)$;
- $ab \rightarrow$ output a value from $[.0625, 0.25)$;
- $ba \rightarrow$ output a value from $[.25, .4375)$;
- $bb \rightarrow$ output a value from $[.4375, 1)$.

- Why is the size of the interval of $ab = .25 - 0.0625 = 0.1875$?

Basic Idea: 2

MLS with $A = \{a, b\}$, $\Pr[a] = 1/4$, $\Pr[b] = 3/4$.
We want to encode 2 symbols as a number in $[0, 1)$.

- $aa \rightarrow$ output a value from $[0, .0625)$;
- $ab \rightarrow$ output a value from $[.0625, 0.25)$;
- $ba \rightarrow$ output a value from $[.25, .4375)$;
- $bb \rightarrow$ output a value from $[.4375, 1)$.

- Why is the size of the interval of $ab = .25 - 0.0625 = 0.1875$?
- The probability of $ab = 1/4 \times 3/4 = 3/16 = 0.1875$.

Basic Idea: 2

MLS with $A = \{a, b\}$, $\Pr[a] = 1/4$, $\Pr[b] = 3/4$.
We want to encode 2 symbols as a number in $[0, 1)$.

- $aa \rightarrow$ output a value from $[0, .0625)$;
- $ab \rightarrow$ output a value from $[.0625, 0.25)$;
- $ba \rightarrow$ output a value from $[.25, .4375)$;
- $bb \rightarrow$ output a value from $[.4375, 1)$.

- Why is the size of the interval of $ab = .25 - 0.0625 = 0.1875$?
- The probability of $ab = 1/4 \times 3/4 = 3/16 = 0.1875$.
- RULE 1: Size of an interval for an input is *always* equal to its probability.

Basic Idea: 2

MLS with $A = \{a, b\}$, $\Pr[a] = 1/4$, $\Pr[b] = 3/4$.
We want to encode 2 symbols as a number in $[0, 1)$.

- $aa \rightarrow$ output a value from $[0, .0625)$;
- $ab \rightarrow$ output a value from $[.0625, 0.25)$;
- $ba \rightarrow$ output a value from $[.25, .4375)$;
- $bb \rightarrow$ output a value from $[.4375, 1)$.

- Why is the size of the interval of $ab = .25 - 0.0625 = 0.1875$?
- The probability of $ab = 1/4 \times 3/4 = 3/16 = 0.1875$.
- RULE 1: Size of an interval for an input is *always* equal to its probability.
- Why output $3 \rightarrow 0.3$ for ba and not $31415926 \rightarrow 0.31415926$?

Basic Idea: 2

MLS with $A = \{a, b\}$, $\Pr[a] = 1/4$, $\Pr[b] = 3/4$.
We want to encode 2 symbols as a number in $[0, 1)$.

- $aa \rightarrow$ output a value from $[0, .0625)$;
- $ab \rightarrow$ output a value from $[.0625, 0.25)$;
- $ba \rightarrow$ output a value from $[.25, .4375)$;
- $bb \rightarrow$ output a value from $[.4375, 1)$.

- Why is the size of the interval of $ab = .25 - 0.0625 = 0.1875$?
- The probability of $ab = 1/4 \times 3/4 = 3/16 = 0.1875$.
- RULE 1: Size of an interval for an input is *always* equal to its probability.
- Why output $3 \rightarrow 0.3$ for ba and not $31415926 \rightarrow 0.31415926$?
- The first output is much shorter, so better compresson.

Basic Idea: 2

MLS with $A = \{a, b\}$, $\Pr[a] = 1/4$, $\Pr[b] = 3/4$.
We want to encode 2 symbols as a number in $[0, 1)$.

- $aa \rightarrow$ output a value from $[0, .0625)$;
- $ab \rightarrow$ output a value from $[.0625, 0.25)$;
- $ba \rightarrow$ output a value from $[.25, .4375)$;
- $bb \rightarrow$ output a value from $[.4375, 1)$.

- Why is the size of the interval of $ab = .25 - 0.0625 = 0.1875$?
- The probability of $ab = 1/4 \times 3/4 = 3/16 = 0.1875$.
- RULE 1: Size of an interval for an input is *always* equal to its probability.
- Why output $3 \rightarrow 0.3$ for ba and not $31415926 \rightarrow 0.31415926$?
- The first output is much shorter, so better compresson.
- RULE 2: Encoding of an input is the *shortest* decimal in its interval. (NB, not "smallest", shortest.)

Arithmetic Coding Algorithm

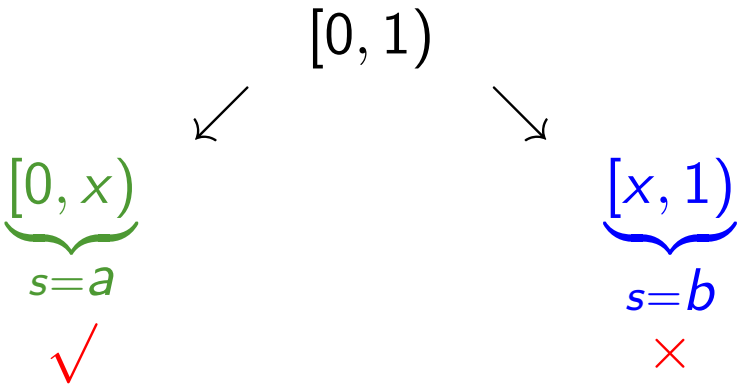
For simplicity assume alphabet has only two symbols say *a* and *b*.

- Reads input one symbol at a time.
- Keep only the *current interval* (the interval for the input seen so far).
- Takes the current interval and splits it into two intervals, one for *a* and one for *b*.
  - Remember RULE 1: Size of an interval for an input is *always* equal to its probability.
  - Split of current interval is usually not equal.

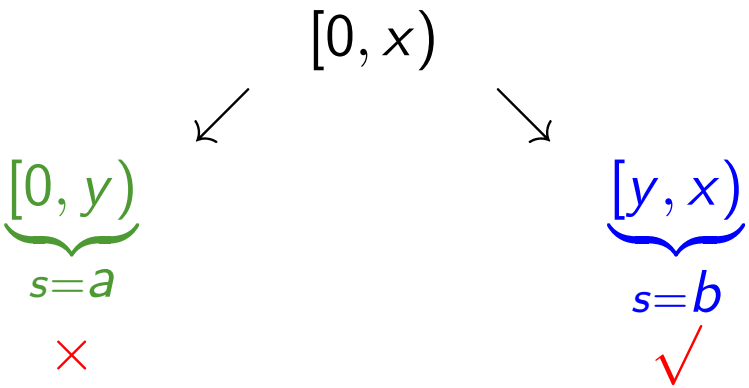
Coding

Input: *abbbab*.

1. Start with $[0, 1)$.
2. Read next symbol s. Calculate intervals for s



3. Read next symbol s. Calculate intervals for as:



Dividing intervals

- Current interval width: δ; Current left endpoint: L.
  - ▷ Interval is $[L, L+\delta)$.
    - e.g. $[0.3, 0.5) \rightarrow L = 0.3, \delta = 0.2$.
- Assume $\sigma_1 =$ 'a', $\sigma_2 =$ 'b', $\Pr[a] = p_1$, $\Pr[b] = p_2$.
- Divide $[L, L+\delta)$ into two pieces that are proportional to p_1 and p_2 respectively:

$$[L, L+\delta)$$

$$\underbrace{[L, L + p_1 \times \delta)}_{s=a} \qquad \underbrace{[L + p_1 \times \delta, L+\delta)}_{s=b}$$

a	L stays same	$\delta = \delta \times p_1$
b	$L = L + \delta \times p_1$	$\delta = \delta \times p_2$

Coding algorithm

1. [INITIALISE] Set $L \leftarrow 0$ and $\delta \leftarrow 1$.
2. [GET PROBABILITIES] For the next symbol, let $\Pr[\sigma_1] = p_1$ and $\Pr[\sigma_2] = p_2$.
3. [RECOMPUTE]
   If next symbol is σ_1 then leave L as is, and set $\delta \leftarrow \delta \times p_1$
   else set $L \leftarrow L + \delta \times p_1$ and $\delta \leftarrow \delta \times p_2$.
4. [DONE?] If no more symbols left then output decimal from $[L, L + \delta)$ else go to 2.

Example

Take $\sigma_1 = a$ and $\sigma_2 = b$, and with $p_1 = 1/4$ and $p_2 = 3/4$. We code *aba* follows:

- Initially $L = 0$ and $\delta = 1$.
- First symbol is a, so leave $L = 0$ and set $\delta = 0.25$.
- Second symbol is b, so set $L = 0.0625$ and $\delta = 0.1875$.
- Third symbol is a, so leave $L = 0.0625$ and set $\delta = .046875$.
- Done. Final interval is $[L, L + \delta) = [0.0625, 0.109375)$.
  - Choose the *shortest* decimal ≥ 0.0625 and < 0.109375.

Example

Take $\sigma_1 = a$ and $\sigma_2 = b$, and with $p_1 = 1/4$ and $p_2 = 3/4$. We code *aba* follows:

- Initially $L = 0$ and $\delta = 1$.
- First symbol is a, so leave $L = 0$ and set $\delta = 0.25$.
- Second symbol is b, so set $L = 0.0625$ and $\delta = 0.1875$.
- Third symbol is a, so leave $L = 0.0625$ and set $\delta = .046875$.
- Done. Final interval is $[L, L + \delta) = [0.0625, 0.109375)$.
  - Choose the *shortest* decimal ≥ 0.0625 and < 0.109375.
  - Choose .1 and output 1.

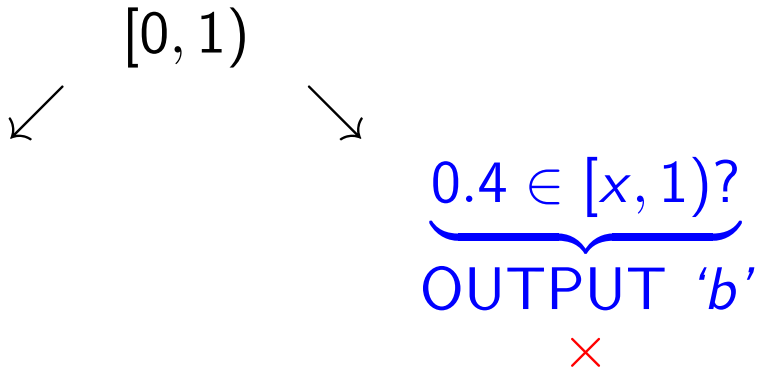
Decoding

- Decoding is basically the inverse of encoding.
- Input is a string of digits which we treat as a real number x in $[0, 1)$.
- Decoder also told the *length of the raw input*
- From the current interval, we create two sub-intervals.
- If x is in the first sub-interval, output a, make the first sub-interval the current interval.
- If x is in the second sub-interval, output b, make the second sub-interval the current interval.
- Continue until all symbols have been decoded.

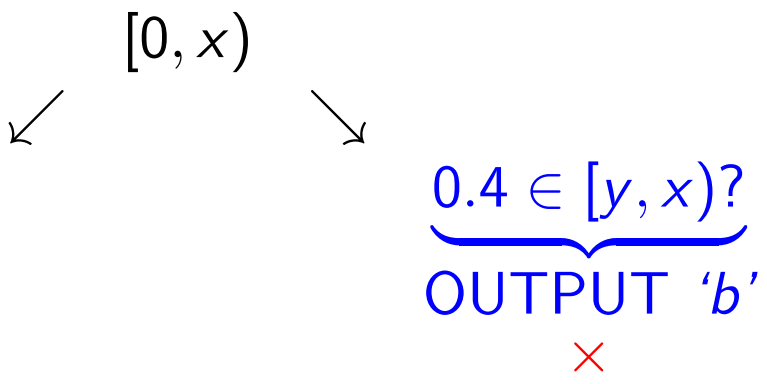
Decoding

Input: $4 \rightarrow 0.4$

1. Start with $[0, 1)$.



2.



3.

Decoding

1. [INITIALISE] Set $L \leftarrow 0$ and $\delta \leftarrow 1$.
2. [GET PROBABILITIES] For the next symbol, let $\Pr[\sigma_1] = p_1$ and $\Pr[\sigma_2] = p_2$.
3. [COMPUTE INTERVALS]
   If $x \in [L, L + \delta \times p_1)$ then output σ_1, leave L unchanged, and set $\delta \leftarrow \delta \times p_1$;
   else if $x \in [L + \delta \times p_1, L + \delta)$ then output σ_2, set $L \leftarrow L + \delta \times p_1$ and $\delta \leftarrow \delta \times p_2$.
4. [DONE?] If we have not decoded the required number of symbols go to 2.

Example

Take $\sigma_1 = a$ and $\sigma_2 = b$, and with $p_1 = 1/4$ and $p_2 = 3/4$. Input is 3 symbols long, and encoded as $1 \rightarrow 0.1$.

- Initially $L = 0$ and $\delta = 1$.
- Create intervals $[0, 0.25)$ for a and $[0.25, 1)$ for b.
  - $0.1 \in [0, 0.25)$ so first symbol is an a. $L = 0$, $\delta = 0.25$.
- Create intervals $[0, 0.0625)$ for a and $[0.0625, 0.25)$ for b.
  - $0.1 \in [0.0625, 0.25)$ so second symbol is a b. $L = 0.0625$, $\delta = 0.1875$.
- Create intervals $[0.0625, 0.109375)$ for a and $[0.109375, 0.25)$ for b.
  - $0.1 \in [0.0625, 0.109375)$ so third symbol is an a. $L = 0.0625$, $\delta = 0.046875$.
- Decoded 3 symbols, so stop.

NOTE: We could have decoded an infinite number of (garbage) symbols from 0.1 so we need to know when to stop.

Optimality of Arithmetic Coding

We now prove (mathematically) that arithmetic coding is optimal. Recall:

- RULE 1: Size of an interval for an input is *always* equal to its probability.
- RULE 2: Encoding of an input is the *shortest* decimal in its interval.
- Shannon's theorem gives optimality when coding in *binary*. When coding in *decimal* we need to modify it as given below.

Theorem (Shannon's Theorem)

Given a MLS with alphabet $\{\sigma_1, \ldots, \sigma_n\}$ *and with* $\Pr[\sigma_i] = p_i$, *the optimal variable-length code (i.e. that has the lowest possible average code length, and gives lossless compression) uses* $\log_{10}(1/p_i)$ *digits to code* σ_i, *for all* $i = 1, \ldots, n$.

Optimality of Arithmetic Coding: 1

Suppose $A = \{a, b\}$, $\Pr[a] = p_1$, $\Pr[b] = p_2$. Input $s = s_1 s_2 \ldots s_k$

- At the end, $\delta = \Pr[s] = \Pr[s_1] \times \ldots \Pr[s_k]$. Why?
  - if $s_i = a$ then $\delta = \delta \times \Pr[a]$.
  - if $s_i = b$ then $\delta = \delta \times \Pr[b]$.

We are obeying RULE 1: size of an interval for an input is *always* equal to its probability.

Optimality of Arithmetic Coding: 2

Suppose $A = \{a, b\}$, $\Pr[a] = p_1$, $\Pr[b] = p_2$. Input $s = s_1 s_2 \ldots s_k$.
RULE 2: Encoding of an input is the *shortest* decimal in its interval.

- Suppose that the interval for $s = [L, L + \delta)$.
- ▷ The shortest decimal in $[L, L + \delta)$ is *at most* $\lceil \log_{10}(1/\delta) \rceil$ digits.

Example: Let $\delta = 0.001$.

- The shortest decimal in $[L, L + 0.001)$ needs at most $\lceil \log_{10}(1/.001) \rceil = 3$ digits.
  E.g. $L = 0.43875$, $L + \delta = 0.43975$, choose 0.439.
  E.g. $L = 0.299$, $L + \delta = 0.3$, choose 0.299.
  E.g. $L = 0.2995$, $L + \delta = 0.3005$, choose 0.3.

Optimality of Arithmetic Coding: 3

Suppose $A = \{a, b\}$, $\Pr[a] = p_1$, $\Pr[b] = p_2$. Input $s = s_1 s_2 \ldots s_k$.

- An input s gets an interval of size $\delta = \Pr[s]$.
- The shortest decimal in $[L, L + \delta)$ has at most $\lceil \log_{10}(1/\Pr[s]) \rceil$ digits.
- The coded output size is $\lceil \log_{10}(1/\Pr[s]) \rceil$ digits.

However:

$$\begin{aligned} & \log_{10}(1/\Pr[s]) \\ = \; & \log_{10}(1/(\Pr[s_1] \times \Pr[s_2] \times \ldots \times \Pr[s_k])) \\ = \; & \log_{10}((1/\Pr[s_1]) \times (1/\Pr[s_2]) \times \ldots \times (1/\Pr[s_k])) \\ = \; & \log_{10}(1/\Pr[s_1]) + \log_{10}(1/\Pr[s_2]) + \ldots + \log_{10}(1/\Pr[s_k]) \end{aligned}$$

Since each symbol is coded using the optimal number of bits, the overall output size is optimal (except for ceiling).

Conclusion

- We covered a simplified version of arithmetic coding.
- We showed that the output size is optimal in the decimal case (the same reasoning holds in the binary case).
- We focussed on binary alphabets. If the alphabet size is larger?
  ▷ 3 symbols $\rightarrow$ 3 intervals.
- In each step we have a [GET PROBABILITIES] step. This is pointless for an MLS since the probabilities never change.
  - However, if we are coding an output from a *known* Markov source, there is ▷ no change in the algorithm.
  - Arithmetic coding can code the output of a *known* Markov source optimally.