8 Clever Codes

2 December 2024

Prof. Dr. Sebastian Wild

CS566 (Wintersemester 2024/25)
Philipps-Universität Marburg

version `2024-12-03 13:01`

Learning Outcomes

Unit 8: ***Clever Codes***

1. Know the principles and performance characteristics of *arithmetic coding*.
2. Judge the use of arithmetic coding in applications.
3. Understand the context of *error-prone communication*.
4. Understand concepts of *error-detecting codes* and *error-correcting codes*.
5. Know and understand *Hamming codes*, in particular (7,4) Hamming code.
6. Reason about the *suitability of a code* for an application.

Outline

8.1 Arithmetic Coding

Stream Codes

- **Recall:** (binary) character encoding $E : \Sigma \to \{0,1\}^\star$
  - Huffman codes *optimal* for any given character frequencies
  - $\rightsquigarrow$ encoding all characters with that code *minimizes* compressed size

Stream Codes

- **Recall:** (binary) character encoding $E : \Sigma \to \{0, 1\}^\star$
  - Huffman codes *optimal* for any given character frequencies
  - $\rightsquigarrow$ encoding all characters with that code *minimizes* compressed size

. . . ***if we assume*** *that all characters must be encoded individually by a codeword!*

Stream Codes

- **Recall:** (binary) character encoding $E : \Sigma \to \{0,1\}^\star$
  - Huffman codes *optimal* for any given character frequencies
  - $\leadsto$ encoding all characters with that code *minimizes* compressed size

… ***if we assume*** *that all characters must be encoded individually by a codeword!*

- Stream codes instead compress entire **sequence** of characters
  - RLE and LZW are examples of stream codes $\leadsto$ can sometimes do better
- Two indicative examples
  1. **"Low entropy bits:"** $\Sigma = \{0,1\}$, highly skewed: $p_0 = 0.99$
     - $\leadsto$ entropy $\mathcal{H}(\frac{1}{100}, \frac{99}{100}) \approx 0.08$ bits per character, Huffman code must use 1 bit per character!
     - $\leadsto$ "optimal" Huffman code gives 12-fold space increase over entropy!

Stream Codes

- **Recall:** (binary) character encoding $E : \Sigma \to \{0,1\}^\star$
  - Huffman codes *optimal* for any given character frequencies
  - $\rightsquigarrow$ encoding all characters with that code *minimizes* compressed size

… ***if we assume*** *that all characters must be encoded individually by a codeword!*

- Stream codes instead compress entire **sequence** of characters
  - RLE and LZW are examples of stream codes $\rightsquigarrow$ can sometimes do better
- Two indicative examples
  1. **"Low entropy bits:"** $\Sigma = \{0,1\}$, highly skewed: $p_0 = 0.99$
     - $\rightsquigarrow$ entropy $\mathcal{H}(\frac{1}{100}, \frac{99}{100}) \approx 0.08$ bits per character,
       Huffman code must use 1 bit per character!
     - $\rightsquigarrow$ "optimal" Huffman code gives 12-fold space increase over entropy!
     - Can certainly do better here (RLE!)
  2. **"Trits":** $\Sigma = \{0,1,2\}$, equally likely
     - $\rightsquigarrow$ entropy $\mathcal{H}(\frac{1}{3}, \frac{1}{3}, \frac{1}{3}) = \lg(3) \approx 1.58$ bits per character,
       Huffman code uses average of $\frac{1}{3} \cdot 1 + \frac{2}{3} \cdot 2 = \frac{5}{3} \approx 1.67$
- Can we do better?

A Decent Hack: Block Codes

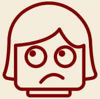
- Huffman on trits wastes ≈ 0.0817 bits per character and over $5\,\%$ of space
- A simple trick can reduce this substantially!
  - treat 5 trits as one "supercharacter", e. g., `21101`
  - $\rightsquigarrow$ $3^5 = 243$ possible combinations
  - $\rightsquigarrow$ encode these using 8 bits (with $2^8 = 256$ possible combinations)
  - entropy $\lg(3^5) \approx 7.92$ bits, so less than $0.1\,\%$ wasted space!

A Decent Hack: Block Codes

- Huffman on trits wastes ≈ 0.0817 bits per character and over $5\,\%$ of space
- A simple trick can reduce this substantially!
  - treat 5 trits as one "supercharacter", e. g., `21101`
  - $\rightsquigarrow$ $3^5 = 243$ possible combinations
  - $\rightsquigarrow$ encode these using 8 bits (with $2^8 = 256$ possible combinations)
  - entropy $\lg(3^5) \approx 7.92$ bits, so less than $0.1\,\%$ wasted space!
- We can even use a Huffman code for the supercharacters to handle nonuniformity!
- For the low-entropy bits, could use 3 bits
  - $\rightsquigarrow$ probabilities:
    `000` : 0.97
    `001`, `010`, `100` : 0.0098
    `011`, `101`, `110` : 0.000099
    `111`: 0.000001
  - $\rightsquigarrow$ with Huffman code, 1.06 bits per superchar of 3 input bits
  - $\rightsquigarrow$ almost factor 3 better; can improve with larger blocks!

Block Codes – A Panacea?

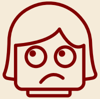
- Using supercharacters works well in our examples.



Hmmm . . . so why don't we treat the entire source text as one large block? Wouldn't that be even better!?

Block Codes – A Panacea?

- Using supercharacters works well in our examples.

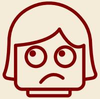


Hmmm . . . so why don't we treat the entire source text as one large block?
Wouldn't that be even better!?

⇝ *We can optimally compress any text, without doing anything intelligent!*

Block Codes – A Panacea?

- Using supercharacters works well in our examples.

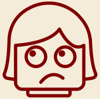


Hmmm . . . so why don't we treat the entire source text as one large block?
Wouldn't that be even better!?

⇝ *We can optimally compress any text, without doing anything intelligent!?*

Block Codes – A Panacea?

- Using supercharacters works well in our examples.



Hmmm ... so why don't we treat the entire source text as one large block? Wouldn't that be even better!?

⇝ *We can optimally compress any text, without doing anything intelligent!?*

ϟ For general case, need to *communicate* the supercharacter encoding

- Blocks of k characters need $\Omega(\sigma^k)$ space for code
- Huffman code has to be part of coded message

⇝ Can only sensibly use block codes for small σ and k

There is no such thing as a free lunch ...

Arithmetic Coding

except in isolated lucky cases

- Also: Block codes still had $\Theta(n)$ wasted space for sequences of n symbols

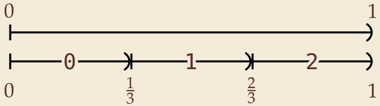
Arithmetic Coding

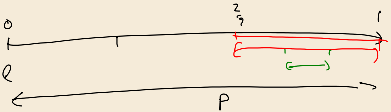
except in isolated lucky cases

- Also: Block codes still had $\Theta(n)$ wasted space for sequences of n symbols
- ***Arithmetic Coding:***
  - **0.** Maintain $[\ell, \ell + p) \subseteq [0, 1)$; initially $\ell = 0$, $p = 1$
  - **1.** Zoom into subinterval for each character
  - **2.** Output dyadic encoding of final interval
- ***Step 1:*** "Zoom" for each character (trit) in $S[0..n)$:
  - Of the current subinterval $[\ell, \ell + p)$, take first, second or last third depending whether $S[i] = \mathtt{0}$, $\mathtt{1}$, resp. $\mathtt{2}$:

    $\ell := \ell + S[i] \cdot \frac{1}{3} \cdot p$

    $p := p \cdot \frac{1}{3}$





$S[0] = 2 \rightsquigarrow \ell = \frac{2}{3} \quad p = \frac{1}{3}$

$S[1] = 1 \rightsquigarrow \ell = \frac{2}{3} + \frac{1}{9} \quad p = \frac{1}{3} \cdot \frac{1}{3}$

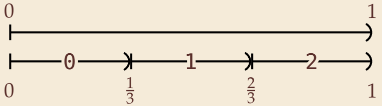
Arithmetic Coding

except in isolated lucky cases

- Also: Block codes still had $\Theta(n)$ wasted space for sequences of n symbols

- ***Arithmetic Coding:***
  0. Maintain $[\ell, \ell + p) \subseteq [0, 1)$; initially $\ell = 0$, $p = 1$
  1. Zoom into subinterval for each character
  2. Output dyadic encoding of final interval

- ***Step 1:*** "Zoom" for each character (trit) in $S[0..n)$:
  - Of the current subinterval $[\ell, \ell + p)$, take first, second or last third depending whether $S[i] = \mathtt{0}$, $\mathtt{1}$, resp. $\mathtt{2}$:
    $\ell := \ell + S[i] \cdot \frac{1}{3} \cdot p$
    $p := p \cdot \frac{1}{3}$



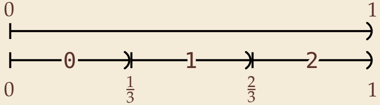
[ℓ, ℓ+p) not "nice" to encode

- ***Step 2:*** Dyadic encoding
  - Find smallest m so that $\exists x \in \mathbb{N}_0$ with $\left[\frac{x}{2^m}, \frac{x+1}{2^m}\right) \subseteq [\ell, \ell + p)$
  - Output x in binary using m bits.

Arithmetic Coding

except in isolated lucky cases

- Also: Block codes still had $\Theta(n)$ wasted space for sequences of n symbols
- ***Arithmetic Coding:***
  - **0.** Maintain $[\ell, \ell + p) \subseteq [0, 1)$; initially $\ell = 0$, $p = 1$
  - **1.** Zoom into subinterval for each character
  - **2.** Output dyadic encoding of final interval
- ***Step 1:*** "Zoom" for each character (trit) in $S[0..n)$:
  - Of the current subinterval $[\ell, \ell + p)$, take first, second or last third depending whether $S[i] = \texttt{0}$, $\texttt{1}$, resp. $\texttt{2}$:
    $\ell := \ell + S[i] \cdot \frac{1}{3} \cdot p$
    $p := p \cdot \frac{1}{3}$



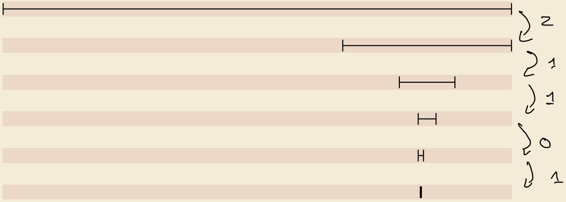
- ***Step 2:*** Dyadic encoding
  - Find smallest m so that $\exists x \in \mathbb{N}_0$ with $\left[\frac{x}{2^m}, \frac{x+1}{2^m}\right) \subseteq [\ell, \ell + p)$
  - Output x in binary using m bits.

$\rightsquigarrow$ Encode n trits in $n \lg(3) + 2$ bits(!) without cheating

Arithmetic Coding – Encode Trits Example

- $S[0..n) = \texttt{21101} \quad (n = 5)$
- **Step 1:** Zoom into subintervals

Iteration	ℓ	p	Interval (rounded)
0	0	1	$[0.00000, 1.00000)$
1	$\frac{2}{3}$	$\frac{1}{3}$	$[0.66667, 1.00000)$
2	$\frac{7}{9}$	$\frac{1}{9}$	$[0.77778, 0.88889)$
3	$\frac{22}{27}$	$\frac{1}{27}$	$[0.81482, 0.85185)$
4	$\frac{66}{81}$	$\frac{1}{81}$	$[0.81482, 0.82716)$
5	$\frac{199}{243}$	$\frac{1}{243}$	$[0.81893, 0.82305)$



Arithmetic Coding – Encode Trits Example

- $S[0..n) = \texttt{21101} \quad (n = 5)$
- **Step 1:** Zoom into subintervals

Iteration	ℓ	p	Interval (rounded)
0	0	1	$[0.00000, 1.00000)$
1	$\frac{2}{3}$	$\frac{1}{3}$	$[0.66667, 1.00000)$
2	$\frac{7}{9}$	$\frac{1}{9}$	$[0.77778, 0.88889)$
3	$\frac{22}{27}$	$\frac{1}{27}$	$[0.81482, 0.85185)$
4	$\frac{66}{81}$	$\frac{1}{81}$	$[0.81482, 0.82716)$
5	$\frac{199}{243}$	$\frac{1}{243}$	$[0.81893, 0.82305)$

- **Step 2:** Dyadic encoding for interval $[\ell, \ell + p) = \left[\frac{199}{243}, \frac{200}{243}\right)$ $\quad 2^{-m} \leq p$
  - Must have $\underline{m \geq \lg(1/p)} > 7$

Arithmetic Coding – Encode Trits Example

- $S[0..n) = \texttt{21101} \quad (n = 5)$
- **Step 1:** Zoom into subintervals

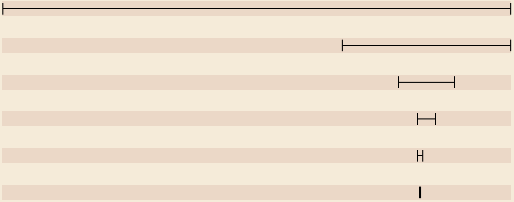
Iteration	ℓ	p	Interval (rounded)
0	0	1	$[0.00000, 1.00000)$
1	$\frac{2}{3}$	$\frac{1}{3}$	$[0.66667, 1.00000)$
2	$\frac{7}{9}$	$\frac{1}{9}$	$[0.77778, 0.88889)$
3	$\frac{22}{27}$	$\frac{1}{27}$	$[0.81482, 0.85185)$
4	$\frac{66}{81}$	$\frac{1}{81}$	$[0.81482, 0.82716)$
5	$\frac{199}{243}$	$\frac{1}{243}$	$[0.81893, 0.82305)$

- **Step 2:** Dyadic encoding for interval $[\ell, \ell + p) = \left[\frac{199}{243}, \frac{200}{243}\right)$
  - Must have $m \geq \lg(1/p) > 7$
    - $m = 8$: smallest $x/2^m \geq \frac{199}{243}$ is $x = 210$, but $[210/256, 211/256) \approx [0.82031, 0.82422) \not\subseteq [\ell, \ell + p)$

Arithmetic Coding – Encode Trits Example

- $S[0..n) = \texttt{21101} \quad (n = 5)$
- **Step 1:** Zoom into subintervals

Iteration	ℓ	p	Interval (rounded)
0	0	1	$[0.00000, 1.00000)$
1	$\frac{2}{3}$	$\frac{1}{3}$	$[0.66667, 1.00000)$
2	$\frac{7}{9}$	$\frac{1}{9}$	$[0.77778, 0.88889)$
3	$\frac{22}{27}$	$\frac{1}{27}$	$[0.81482, 0.85185)$
4	$\frac{66}{81}$	$\frac{1}{81}$	$[0.81482, 0.82716)$
5	$\frac{199}{243}$	$\frac{1}{243}$	$[0.81893, 0.82305)$



- **Step 2:** Dyadic encoding for interval $[\ell, \ell + p) = \left[\frac{199}{243}, \frac{200}{243}\right)$
  - Must have $m \ge \lg(1/p) > 7$
    - $m = 8$: smallest $x/2^m \ge \frac{199}{243}$ is $x = 210$, but $[210/256, 211/256) \approx [0.82031, 0.82422) \not\subset [\ell, \ell + p)$
    - $m = 9$: smallest $x/2^m \ge \frac{199}{243}$ is $x = 420$ and $[420/512, 421/512) \approx [0.82031, 0.82227) \subset [\ell, \ell + p)$ ✓

Arithmetic Coding – Encode Trits Example

- $S[0..n) = \texttt{21101} \quad (n = 5)$
- **Step 1:** Zoom into subintervals

Iteration	ℓ	p	Interval (rounded)
0	0	1	$[0.00000, 1.00000)$
1	$\frac{2}{3}$	$\frac{1}{3}$	$[0.66667, 1.00000)$
2	$\frac{7}{9}$	$\frac{1}{9}$	$[0.77778, 0.88889)$
3	$\frac{22}{27}$	$\frac{1}{27}$	$[0.81482, 0.85185)$
4	$\frac{66}{81}$	$\frac{1}{81}$	$[0.81482, 0.82716)$
5	$\frac{199}{243}$	$\frac{1}{243}$	$[0.81893, 0.82305)$

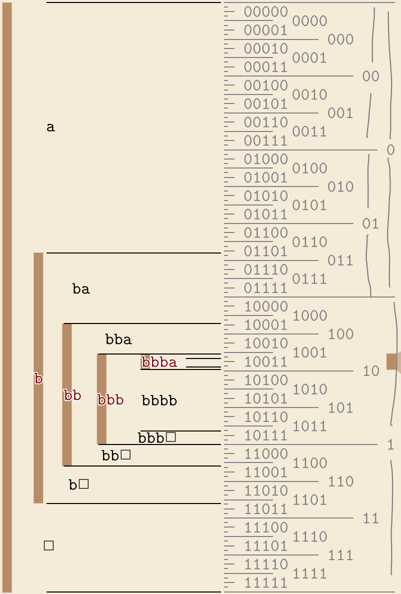
- **Step 2:** Dyadic encoding for interval $[\ell, \ell + p) = \left[\frac{199}{243}, \frac{200}{243}\right)$
  - Must have $m \ge \lg(1/p) > 7$
    - $m = 8$: smallest $x/2^m \ge \frac{199}{243}$ is $x = 210$, but $[210/256, 211/256) \approx [0.82031, 0.82422) \not\subset [\ell, \ell + p)$
    - $m = 9$: smallest $x/2^m \ge \frac{199}{243}$ is $x = 420$ and $[420/512, 421/512) \approx [0.82031, 0.82227) \subset [\ell, \ell + p)$ ✓

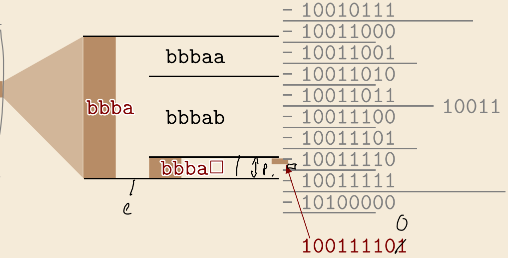
$\rightsquigarrow$ Output $x = 420$ in binary with $m = 9$ digits: `110100100`

Versatility of Arithmetic Coding – Adaptive Model

bbba□

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





adapted from Figure 6.4 of MacKay: *Information Theory, Inference, and Learning Algorithms* 2003

Arithmetic Coding – General framework

- Note: Arithmetic coder *doesn't care* if probabilities or even σ change all the time!
  - As long as encoder and decoder know from context what they are!

Arithmetic Coding – General framework

- Note: Arithmetic coder *doesn't care* if probabilities or even σ change all the time!
  - As long as encoder and decoder know from context what they are!

General stochastic sequence:
Sequence of random variables $X_0, X_1, X_2, \ldots$ such that

1. $X_i \in [0..U_i) \cup \{\$\}$ (We use $\$$ to signal "end of text")
2. $\mathbb{P}[X_i = j] = P_{ij}$
3. both U_i and P_{ij} are random variables as they *depend* on $X_0, \ldots X_{i-1}$, but conditioned on $X_0, \ldots, X_{i-1}$, they are fixed and known:
   $$P_{ij} = P_{ij}(X_0, \ldots, X_{i-1}) = \mathbb{P}[X_i = j \mid X_0, \ldots, X_{i-1}]$$
   $$U_i = U_i(X_0, \ldots, X_{i-1}) = \max\{j : P_{ij}(X_0, \ldots, X_{i-1}) > 0\}$$

Arithmetic Coding – General framework

- Note: Arithmetic coder *doesn't care* if probabilities or even σ change all the time!
  - As long as encoder and decoder know from context what they are!

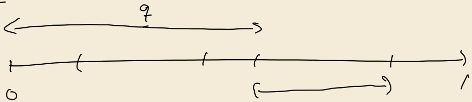
General stochastic sequence:
Sequence of random variables $X_0, X_1, X_2, \ldots$ such that

1. $X_i \in [0..U_i) \cup \{\$\}$ (We use $\$$ to signal "end of text")
2. $\mathbb{P}[X_i = j] = P_{ij}$
3. both U_i and P_{ij} are random variables as they *depend* on $X_0, \ldots X_{i-1}$,
   but conditioned on $X_0, \ldots, X_{i-1}$, they are fixed and known:
   $$P_{ij} = P_{ij}(X_0, \ldots, X_{i-1}) = \mathbb{P}[X_i = j \mid X_0, \ldots, X_{i-1}]$$
   $$U_i = U_i(X_0, \ldots, X_{i-1}) = \max\{j : P_{ij}(X_0, \ldots, X_{i-1}) > 0\}$$

- Can model arbitrary dependencies on previous outcomes
- Assume here that random process is known by both encoder and decoder (fixed coding)
  otherwise extra space needed to encode model!

Arithmetic Coding – Encoding

procedure arithmeticEncode($X_0, \ldots, X_n$):
  // Assume model U_i and P_{ij} are fixed.
  // Assume $X_i \in [0..U_i)$ for $i < n$ and $X_n = \$$
  // Step 1: Interval zooming
  $\ell := 0; \quad p := 1$
  for $i := 0, \ldots, n-1$ **do**
    $$q := \sum_{j=0}^{X_i - 1} P_{ij};$$
    $q = P_{i0} + P_{i1} + \ldots + P_{i,X_i-1}$
    $\ell := \ell + q \cdot p; \quad p := p \cdot P_{i,X_i}$
  end for
  $q := 1 - P_{n,\$}$ *// encode \$ as last character*
  $\ell := \ell + q \cdot p; \quad p := p \cdot P_{n,\$}$
  // Step 2: Dyadic encoding
  $m := \lceil \lg(1/p) \rceil - 1$
  do
    $m := m + 1; \quad x := \lceil \ell \cdot 2^m \rceil$
  while $(x+1)/2^m > \ell + p$
  return x **in** binary using m bits

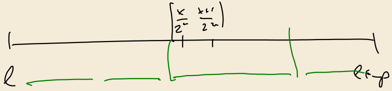


Arithmetic Coding – Decoding

procedure arithmeticDecode($C[0..m)$):
    // Assume model U_i and P_{ij} are fixed.
    // $C[0..m)$ bit string produced by arithmeticEncode
    $x = \sum_{i=0}^{m-1} C[i] \cdot 2^{m-1-i}$ *// final interval* $[x/2^m, (x+1)/2^m)$
    $\ell := 0;\ \ p := 1;\ \ i := 0$
    while true
        $c := 0;\ \ q := 0$ *// Decode next character c*
        while $\ell + q \cdot p < x/2^m$ *// Iterate through characters until final interval*
            if $c == U_i + 1$ *// reached \$*
                $X[i] := \$$
                return $X[0..i]$
            else
                $q := q + P_{i,c};\ \ c := c + 1$
        end while
        $c := c - 1;\ \ q := q - P_{i,c}$ *// we overshot by 1*
        $X[i] := c$
        $\ell := \ell + q \cdot p;\ \ p := p \cdot P_{i,c}$
        $i := i + 1$
    end for

Example: adaptive model on $\Sigma = \{a, b\}$

$$\mathbb{P}[S[i]=a \mid S[0..i)] = \frac{|S[0..i)|_a + 1}{i+2}$$



8.2 Practical Arithmetic Coding

Arithmetic Coding – Numerics

- As implemented above, p usually gets smaller by a constant factor with *each character*
  - $\rightsquigarrow$ p gets exponentially small in n!
  - ℓ does not get smaller in absolute terms, but we need it to ever higher accuracy

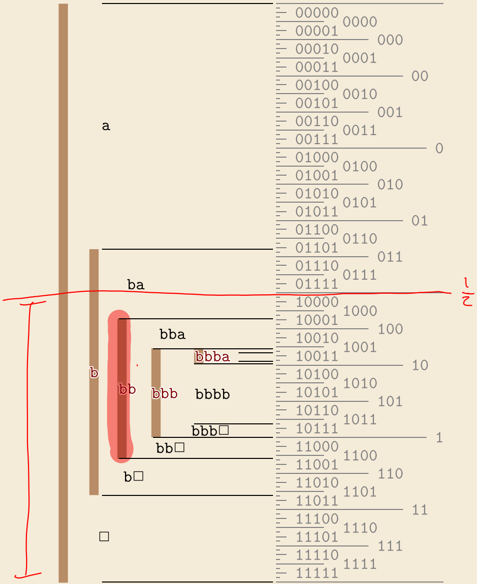
$\rightsquigarrow$ requires $\Omega(n)$ bit precision and exact arithmetic!

Arithmetic Coding – Numerics

- As implemented above, p usually gets smaller by a constant factor with *each character*
  - $\rightsquigarrow$ p gets exponentially small in n!
  - ℓ does not get smaller in absolute terms, but we need it to ever higher accuracy

$\rightsquigarrow$ requires $\Omega(n)$ bit precision and exact arithmetic!

- *With a clever trick, this can be avoided!*
  - If $[\ell, \ell + p) \subseteq [0, \frac{1}{2})$, we know:
    - Our final x with $\left[\frac{x}{2^m}, \frac{x+1}{2^m}\right) \subseteq [\ell, \ell + p)$ *must start with a* `0`*-bit*!
  - $\rightsquigarrow$ Output a `0` and renormalize interval: $\ell := 2\ell;\ p := 2p$

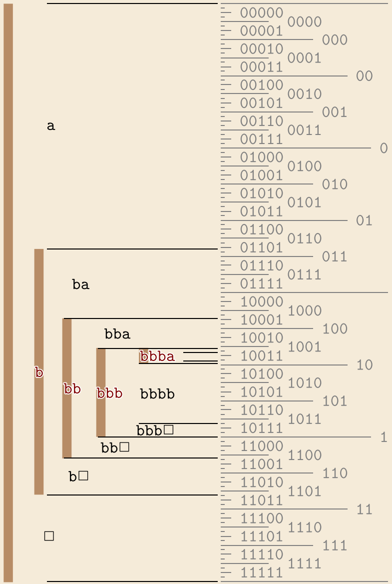


Arithmetic Coding – Numerics

- As implemented above, p usually gets smaller by a constant factor with *each character*
  - $\rightsquigarrow$ p gets exponentially small in n!
  - ℓ does not get smaller in absolute terms, but we need it to ever higher accuracy

$\rightsquigarrow$ requires $\Omega(n)$ bit precision and exact arithmetic!

- *With a clever trick, this can be avoided!*
  - If $[\ell, \ell + p) \subseteq [0, \frac{1}{2})$, we know:
    - Our final x with $\left[\frac{x}{2^m}, \frac{x+1}{2^m}\right) \subseteq [\ell, \ell + p)$ *must start with a `0`-bit*!
    - $\rightsquigarrow$ Output a `0` and renormalize interval: $\ell := 2\ell;\ p := 2p$
  - If $[\ell, \ell + p) \subseteq [\frac{1}{2}, 1)$, similarly:
    - Output `1` and renormalize: $\ell := \ell - \frac{1}{2}$ $\ell := 2\ell;\ p := 2p$



Arithmetic Coding – Renormalization

Does this guarantee ℓ and p stay in a reasonable range?

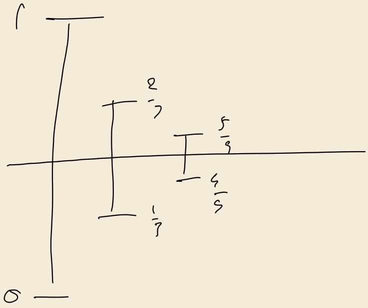
Arithmetic Coding – Renormalization

Does this guarantee ℓ and p stay in a reasonable range?

- No! Consider (uniform) trits in $\{0, 1, 2\}$ again and encode $1111111111111111\ldots$

$$\rightsquigarrow\ p = \left(\tfrac{1}{3}\right)^n, \quad \ell = \frac{1}{3} + \frac{1}{9} + \frac{1}{27} + \cdots = \sum_{i=1}^{n} 3^{-i} = \frac{1}{2} - \frac{3^{-n}}{2}$$

$\rightsquigarrow$ $\ell < \frac{1}{2}$ and $\ell + p > \frac{1}{2}$ $\rightsquigarrow$ next bit unknown as of yet



Arithmetic Coding – Renormalization

Does this guarantee ℓ and p stay in a reasonable range?

- No! Consider (uniform) trits in $\{0, 1, 2\}$ again and encode `1111111111111111...`

$$\rightsquigarrow\ p = \left(\tfrac{1}{3}\right)^n, \quad \ell = \frac{1}{3} + \frac{1}{9} + \frac{1}{27} + \cdots = \sum_{i=1}^{n} 3^{-i} = \frac{1}{2} - \frac{3^{-n}}{2}$$

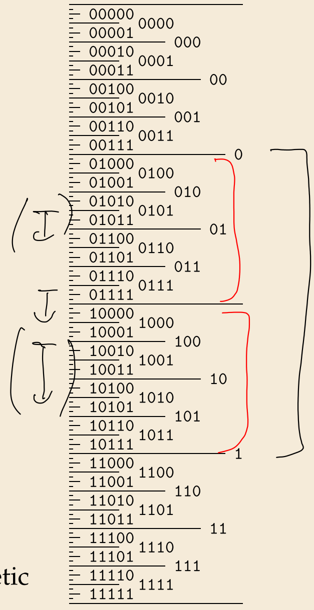
$\rightsquigarrow$ $\ell < \frac{1}{2}$ and $\ell + p > \frac{1}{2}$ $\rightsquigarrow$ next bit unknown as of yet

But: If $[\ell, \ell + p) \subseteq [\frac{1}{4}, \frac{3}{4})$, next **two** bits are either `01` or `10`

- Remember an *"outstanding opposite bit"* (increment counter)
- Renormalize:
  $\ell := \ell - \frac{1}{\mathbf{4}}$
  $\ell := 2\ell;\ p := 2p$

$\rightsquigarrow$ ℓ and p remain in range of P_{ij}

$\rightsquigarrow$ round P_{ij} to integer multiple of 2^{-F} $\rightsquigarrow$ fixed-precision arithmetic



Fixed Precision Arithmetic Encode

Detailed code from Moffat, Neal, Witten, *Arithmetic Coding Revisited*, ACM Trans. Inf. Sys. 1998

Note: L is our ℓ, R is our p, $b \leq w$ is #bits for variables

arithmetic_encode(l, h, t)

/* Arithmetically encode the range $[l/t, h/t)$ using low-precision arithmetic. The state variables R and L are modified to reflect the new range, and then renormalized to restore the initial and final invariants $2^{b-2} < R \leq 2^{b-1}$, $0 \leq L < 2^b - 2^{b-2}$, and $L + R \leq 2^b$ */

(1) Set $r \leftarrow R$ div t

(2) Set $L \leftarrow L + r$ times l

(3) If $h < t$ then

    set $R \leftarrow r$ times $(h - l)$

  else

    set $R \leftarrow R - r$ times l

(4) While $R \leq 2^{b-2}$ do

    Use Algorithm ENCODER RENORMALIZATION (Figure 7) to renormalize R, adjust L, and output one bit

Fixed Precision Renormalize

In *arithmetic_encode*()

/* Reestablish the invariant on R, namely that $2^{b-2} < R \le 2^{b-1}$. Each doubling of R corresponds to the output of one bit, either of known value, or of value opposite to the value of the next bit actually output */

(4) While $R \le 2^{b-2}$ do

    If $L + R \le 2^{b-1}$ then

        bit_plus_follow(0)

    else if $2^{b-1} \le L$ then

        bit_plus_follow(1)

        Set $L \leftarrow L - 2^{b-1}$

    else

        Set $bits\_outstanding \leftarrow bits\_outstanding + 1$ and $L \leftarrow L - 2^{b-2}$

    Set $L \leftarrow 2L$ and $R \leftarrow 2R$

bit_plus_follow(x)

/* Write the bit x (value 0 or 1) to the output bit stream, plus any outstanding following bits, which are known to be of opposite polarity */

(1) *write_one_bit*(x).

(2) While $bits\_outstanding > 0$ do

    write_one_bit($1 - x$)

    Set $bits\_outstanding \leftarrow bits\_outstanding - 1$

Fixed Precision Arithmetic Decode

Functions *decode_target* and *arithmetic_decode* to be called alternatingly.

decode_target(t)

/* Returns an integer *target*, $0 \le target < t$ that is guaranteed to lie in the range $[l, h)$ that was used at the corresponding call to *arithmetic_encode*() */

(1) Set $r \leftarrow R$ div t

(2) Return ($\min\{t-1, D \text{ div } r\}$)

$\left[\frac{l}{t}, \frac{h}{t}\right)$

arithmetic_decode(l, h, t)

/* Adjusts the decoder's state variables $\underline{R}$ and $\underline{D}$ to reflect the changes made in the encoder during the corresponding call to *arithmetic_encode*(). Note that, compared with Algorithm CACM Coder (Figure 6), the transformation $D = V - L$ is used. It is also assumed that r has been set by a prior call to *decode_target*() */

(1) Set $D \leftarrow D - r$ times l

(2) If $h < t$ then

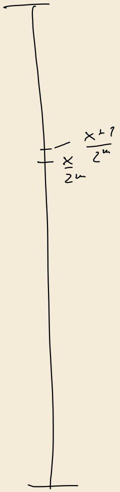
  set $R \leftarrow r$ times $(h - l)$

else

  set $R \leftarrow R - r$ times l

(3) While $R \le 2^{b-2}$ do

  Set $R \leftarrow 2R$ and $\underline{D \leftarrow 2D + \textit{read\_one\_bit}()}$



Arithmetic Coding Discussion

- 👎 Subtle code (⇝ libraries!)
- 👎 Typically slower to encode/decode than Huffman codes
- 👎 Encoded bits can be produced/consumed in bursts
- 👍 Extremely versatile w. r. t. random process
- 👍 Almost optimal space usage / compression
- 👍 Widely used (instead of Huffman) in JPEG, zip variants, . . .

8.3 Error Correcting Codes

Noisy Communication

- most forms of communication are “noisy”
  - humans: acoustic noise, unclear pronunciation, misunderstanding, foreign languages

Noisy Communication

- most forms of communication are “noisy”
  - humans: acoustic noise, unclear pronunciation, misunderstanding, foreign languages

- How do humans cope with that?
  - slow down and/or speak up
  - ask to repeat if necessary

Noisy Communication

- most forms of communication are "noisy"
  - humans: acoustic noise, unclear pronunciation, misunderstanding, foreign languages

- How do humans cope with that?
  - slow down and/or speak up
  - ask to repeat if necessary

- But how is it possible (for us) to decode a message in the presence of noise & errors?

*Bcaesue it semes taht ntaurul lanaguge has a lots fo **redundancy** bilt itno it!*

Noisy Communication

- most forms of communication are "noisy"
  - humans: acoustic noise, unclear pronunciation, misunderstanding, foreign languages

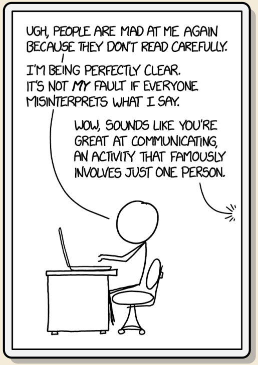
- How do humans cope with that?
  - slow down and/or speak up
  - ask to repeat if necessary
- But how is it possible (for us) to decode a message in the presence of noise & errors?

Bcaesue it semes taht ntaurul lanaguge has a lots fo ***redundancy*** *bilt itno it!*

⇝ We can

1. **detect errors** "This sentence has aao pi dgsdho gioasghds."
2. **correct** (some) **errors** "Tiny errs ar corrrected automaticly."

(sometimes too eagerly as in the Chinese Whispers / Telephone)



Noisy Channels

- computers: copper cables &
  electromagnetic interference
- transmit a binary string
- but occasionally bits can "flip"

⇝ want a robust code

Noisy Channels

- computers: copper cables & electromagnetic interference
- transmit a binary string
- but occasionally bits can "flip"

⇝ want a robust code

- We can aim at
  1. **error detection** ⇝ can request a re-transmit
  2. **error correction** ⇝ avoid re-transmit for common types of errors

Noisy Channels

- computers: copper cables & electromagnetic interference
- transmit a binary string
- but occasionally bits can "flip"

⇝ want a robust code

- We can aim at
  1. **error detection** ⇝ can request a re-transmit
  2. **error correction** ⇝ avoid re-transmit for common types of errors
- This will require *redundancy*: sending *more* bits than plain message
  - ⇝ **goal:** robust code with lowest redundancy

that's the opposite of compression!

Clicker Question

What do you think, how many extra bits do we need to **detect** a **single bit error** in a message of 100 bits?

→ sli.do/cs566

Clicker Question

What do you think, how many extra bits do we need to **correct** a **single bit error** in a message of 100 bits?

→ sli.do/cs566