
# 8 Clever Codes

*1 December 2025*

Prof. Dr. Sebastian Wild

CS566 (Wintersemester 2025–26)
Philipps-Universität Marburg

*version* `2025-12-02 13:49`

# 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 Clever Codes

## 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
  - $\rightsquigarrow$ 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!

# Stream Codes

- **Recall:** (binary) character encoding $E : \Sigma \to \{0,1\}^\star$
  - Huffman codes *optimal* for any given character frequencies
  - ⇝ 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 ⇝ can sometimes do better
- Two indicative examples
  1. **"Low entropy bits:"** $\Sigma = \{0,1\}$, highly skewed: $p_0 = 0.99$
     - ⇝ entropy $\mathcal{H}(\frac{1}{100}, \frac{99}{100}) \approx 0.08$ bits per character,
       Huffman code must use 1 bit per character!
     - ⇝ "optimal" Huffman code gives 12-fold space increase over entropy!
     - Can certainly do better here (RLE!)
  2. **"Trits":** $\Sigma = \{0,1,2\}$, equally likely
     - ⇝ 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

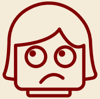
- Huffman on trits wastes $\approx 0.0817$ bits per character and over $5\,\%$ of space
- A simple trick can reduce this substantially!
  - treat 5 trits as one "supercharacter", e. g., `21101`
  - $\rightsquigarrow$ $3^5 = 243$ possible combinations
  - $\rightsquigarrow$ encode these using $8$ bits (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 $\approx 0.0817$ bits per character and over $5\,\%$ of space
- A simple trick can reduce this substantially!
  - treat 5 trits as one "supercharacter", e. g., `21101`
  - ⇝ $3^5 = 243$ possible combinations
  - ⇝ 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
  - ⇝ probabilities:
    `000` : $0.97$
    `001`, `010`, `100` : $0.0098$
    `011`, `101`, `110` : $0.000099$
    `111`: $0.000001$
  - ⇝ with Huffman code, $1.06$ bits per superchar of 3 input bits
  - ⇝ almost factor 3 better; can improve with larger blocks!

# Block Codes – A Panacea?

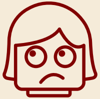
- Using supercharacters works well in our examples.



*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.

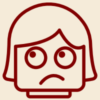


*Hmmm . . . so why don't we treat the entire source text as one large block? Wouldn't that be even better!?*

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

# Block Codes – A Panacea?

- Using supercharacters works well in our examples.

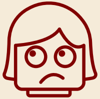


*Hmmm . . . so why don't we treat the entire source text as one large block? Wouldn't that be even better!?*

⇝ *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!?*

$\leadsto$ *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

  $\leadsto$ Can only sensibly use block codes for small $\sigma$ and $k$

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

# Arithmetic Coding

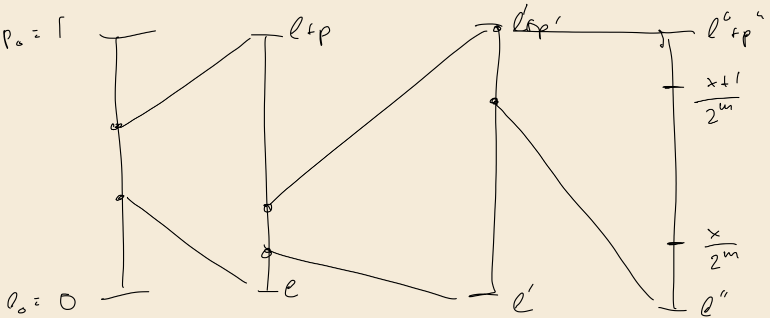
except in isolated lucky cases

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

# Arithmetic Coding

except in isolated lucky cases

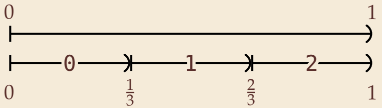
- Also: Block codes still had $\Theta(n)$ wasted space for sequences of $n$ symbols
- ***Arithmetic Coding:***
  - ***0.*** Maintain $[\ell, \ell + p) \subseteq [0, 1)$; initially $\ell = 0$, $p = 1$
  - ***1.*** Zoom into subinterval for each character
  - ***2.*** Output dyadic encoding of final interval



# Arithmetic Coding

except in isolated lucky cases

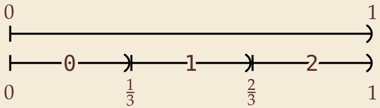
- Also: Block codes still had $\Theta(n)$ wasted space for sequences of $n$ symbols
- ***Arithmetic Coding:***
  ***0.*** Maintain $[\ell, \ell + p) \subseteq [0,1)$; initially $\ell = 0$, $p = 1$
  ***1.*** Zoom into subinterval for each character
  ***2.*** Output dyadic encoding of final interval
- ***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. $\mathtt{2}$:
    $\ell := \ell + S[i] \cdot \frac{1}{3} \cdot p$
    $p := p \cdot \frac{1}{3}$



# 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 $S[0..n)$:

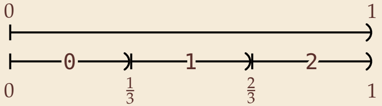



  - Of the current subinterval $[\ell, \ell + p)$, take first, second or last third depending whether $S[i] =$ `0`, `1`, resp. `2`:
    $\ell := \ell + S[i] \cdot \frac{1}{3} \cdot p$
    $p := p \cdot \frac{1}{3}$
- ***Step 2:*** Dyadic encoding
  - Find smallest $m$ so that $\exists x \in \mathbb{N}_0$ with $\left[\frac{x}{2^m}, \frac{x+1}{2^m}\right) \subseteq [\ell, \ell + p)$
  - Output $x$ in binary using $m$ bits.

# 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. $\texttt{2}$:
    $\ell := \ell + S[i] \cdot \frac{1}{3} \cdot p$
    $p := p \cdot \frac{1}{3}$



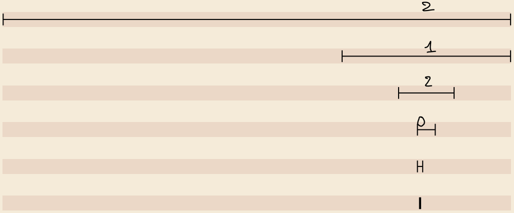

- ***Step 2:*** Dyadic encoding
  - Find smallest $m$ so that $\exists x \in \mathbb{N}_0$ with $\left[\frac{x}{2^m}, \frac{x+1}{2^m}\right) \subseteq [\ell, \ell + p)$
  - Output $x$ in binary using $m$ bits.

$\rightsquigarrow$ 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 | $\ell$ | $p$ | 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 | $\ell$ | $p$ | 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 $m \geq \lg(1/p) > 7$

# Arithmetic Coding – Encode Trits Example

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

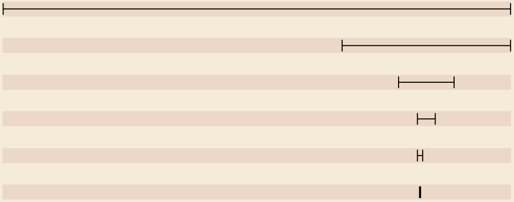
| Iteration | $\ell$ | $p$ | Interval (rounded) |
|---|---|---|---|
| 0 | $0$ | $1$ | $[0.00000, 1.00000)$ |
| 1 | $\frac{2}{3}$ | $\frac{1}{3}$ | $[0.66667, 1.00000)$ |
| 2 | $\frac{7}{9}$ | $\frac{1}{9}$ | $[0.77778, 0.88889)$ |
| 3 | $\frac{22}{27}$ | $\frac{1}{27}$ | $[0.81482, 0.85185)$ |
| 4 | $\frac{66}{81}$ | $\frac{1}{81}$ | $[0.81482, 0.82716)$ |
| 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 | $\ell$ | $p$ | 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 $[210/256, 211/256) \approx [0.82031, 0.82422) \not\subset [\ell, \ell + p)$
    - $m = 9$: smallest $x/2^m \geq \frac{199}{243}$ is $x = 420$ and $[420/512, 421/512) \approx [0.82031, 0.82227) \subset [\ell, \ell + p)$ ✓

# Arithmetic Coding – Encode Trits Example

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

| Iteration | $\ell$ | $p$ | 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`

## 8.2 Arithmetic Coding Beyond Trits

# Beyond Trits

*In the example above, we always split the interval into thirds.*

*But there's nothing special about thirds.*

- $\rightsquigarrow$ Any subdivision of $[0, 1)$ works!

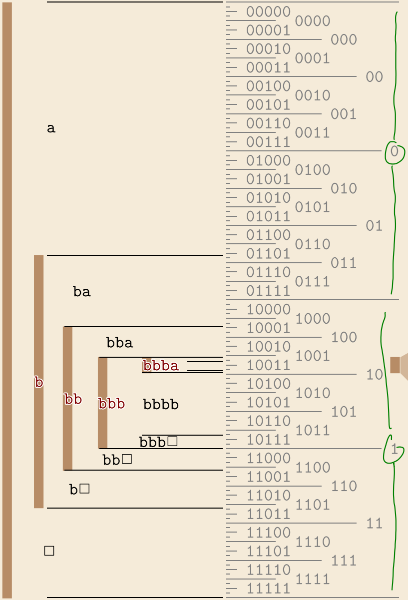
# Versatility of Arithmetic Coding – Adaptive Model

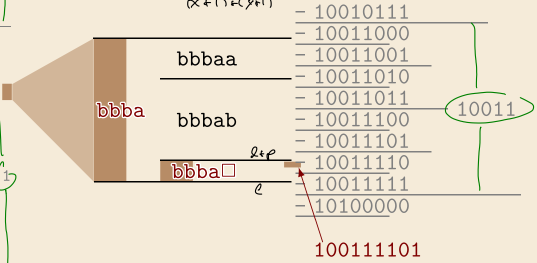
end of word '□'

| 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 | $P(\mathtt{a} \mid \mathtt{bbba}) = 0.28$ | $P(\mathtt{b} \mid \mathtt{bbba}) = 0.57$ | $P(\square \mid \mathtt{bbba}) = 0.15$ |

$$\mathbb{P}[a \mid \#a \text{ before} = x,\ \#b \text{ before } y] = 0.85 \cdot \frac{x+1}{(x+1)+(y+1)}$$

15% chance to stop




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 $\sigma$ 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 $\sigma$ 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 $\sigma$ change all the time!
  - As long as encoder and decoder know from context what they are!

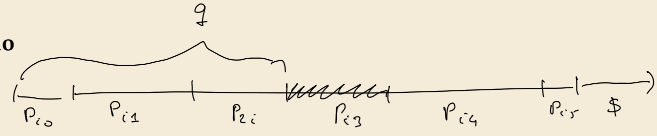
**General stochastic sequence:**
Sequence of random variables $X_0, X_1, X_2, \ldots$ such that

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:
   $$\begin{aligned} P_{ij} &= P_{ij}(X_0, \ldots, X_{i-1}) = \mathbb{P}[X_i = j \mid X_0, \ldots, X_{i-1}] \\ U_i &= U_i(X_0, \ldots, X_{i-1}) = \max\{j : P_{ij}(X_0, \ldots, X_{i-1}) > 0\} \end{aligned}$$

- 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*
  $\ell := 0; \;\; p := 1$
  **for** $i := 0, \ldots, n-1$ **do**
    $q := \sum_{j=0}^{X_i - 1} P_{ij};$
    $\ell := \ell + q \cdot p; \;\; p := p \cdot P_{i,X_i}$
  **end for**
  $q := 1 - P_{n,\$}$ *// encode \$ as last character*
  $\ell := \ell + q \cdot p; \;\; p := p \cdot P_{n,\$}$
  *// Step 2: Dyadic encoding*
  $m := \lceil \lg(1/p) \rceil - 1$
  **do**
    $m := m + 1; \;\; x := \lceil \ell \cdot 2^m \rceil$
  **while** $(x+1)/2^m > \ell + p$
  **return** $x$ in binary using $m$ bits




$X_i = 3$

$\leq 2$ repetitions

# 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 = ∑_{i=0}^{m−1} C[i] · 2^{m−1−i} // final interval [x/2^m, (x+1)/2^m)
    ℓ := 0;  p := 1;  i := 0
    while true
        c := 0;  q := 0 // Decode next character c
        while ℓ + q · 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
        ℓ := ℓ + q · p;  p := p · P_{i,c}
        i := i + 1
    end for
```

## 8.3 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 $n$!
  - $\ell$ does not get smaller in absolute terms, but we need it to ever higher accuracy

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

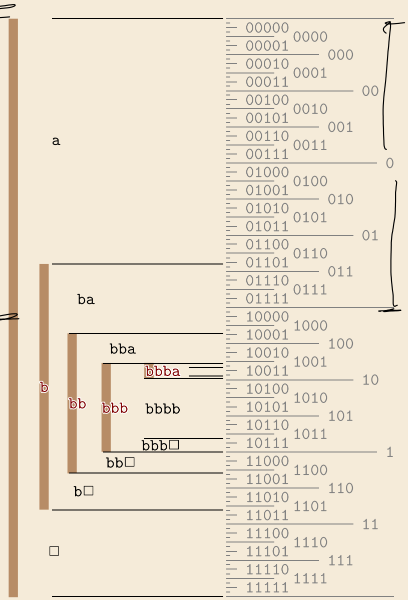
# Arithmetic Coding – Numerics

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

⇝ requires $\Omega(n)$ bit precision and exact arithmetic! $\ell+p$

- *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*!

    ⇝ Output a `0` and renormalize interval: $\ell := 2\ell;\ p := 2p$

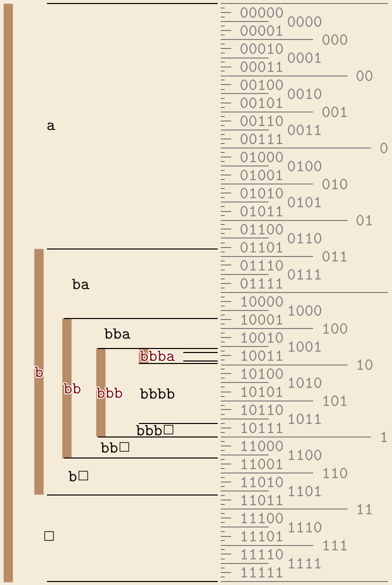


# Arithmetic Coding – Numerics

- As implemented above, $p$ usually gets smaller by a constant factor with *each character*
  - $\rightsquigarrow$ $p$ gets exponentially small in $n$!
  - $\ell$ 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* $\ell$ *and* $p$ *stay in a reasonable range?*

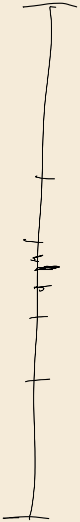
# Arithmetic Coding – Renormalization

*Does this guarantee $\ell$ and $p$ stay in a reasonable range?*

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

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

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



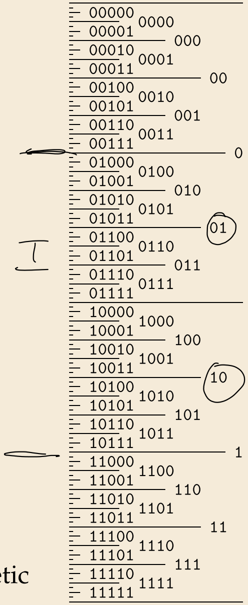
# Arithmetic Coding – Renormalization

*Does this guarantee $\ell$ and $p$ stay in a reasonable range?*

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

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

$\rightsquigarrow\ \ell < \frac{1}{2}$ 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:
  $\ell := \ell - \frac{1}{\mathbf{4}}$
  $\ell := 2\ell;\ p := 2p$

$\rightsquigarrow$ $\ell$ 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: Their $L$ is our $\ell$, $R$ is our $p$, $b \le w$ 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 $2^{b-2} < R \le 2^{b-1}$, $0 \le L < 2^b - 2^{b-2}$, and $L + R \le 2^b$ */

(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 $R \le 2^{b-2}$ 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 $2^{b-2} < R \leq 2^{b-1}$. 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 $R \leq 2^{b-2}$ do

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

        *bit_plus_follow*(0)

    else if $2^{b-1} \leq L$ 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*, $0 \leq target < t$ 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 $(\min\{t-1, D \text{ div } r\})$

*arithmetic_decode*$(l, h, t)$

/* Adjusts the decoder's state variables $R$ and $D$ 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 $R \leq 2^{b-2}$ do
 Set $R \leftarrow 2R$ and $D \leftarrow 2D +$ *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, ...

## 8.4 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

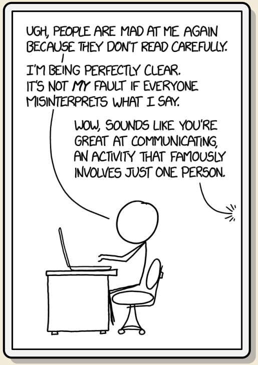
- How do humans cope with that?
  - slow down and/or speak up
  - ask to repeat if necessary
- But how is it possible (for us) to decode a message in the presence of noise & errors?

*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"

$\rightsquigarrow$ want a robust code

# Noisy Channels


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

$\rightsquigarrow$ want a robust code

- We can aim at
  1. **error detection** $\rightsquigarrow$ can request a re-transmit
  2. **error correction** $\rightsquigarrow$ 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

## 8.5 Coding Theory

# Block codes

- **model:**
  - want to send message $S \in \{0,1\}^\star$ (bitstream) across a *(communication) channel*
  - any bit transmitted through the channel might ***flip*** ($0 \to 1$ resp. $1 \to 0$)
    **no other errors** occur (no bits lost, duplicated, inserted, etc.)
  - instead of $S$, we send *encoded bitstream* $C \in \{0,1\}^\star$
    sender *encodes* $S$ to $C$, receiver *decodes* $C$ to $S$ (hopefully)
  - ⇝ what errors can be detected and/or corrected?

# Block codes

- **model:**
  - want to send message $S \in \{\mathtt{0},\mathtt{1}\}^{\star}$ (bitstream) across a *(communication) channel*
  - any bit transmitted through the channel might ***flip*** ($\mathtt{0} \to \mathtt{1}$ resp. $\mathtt{1} \to \mathtt{0}$)
    **no other errors** occur (no bits lost, duplicated, inserted, etc.)
  - instead of $S$, we send *encoded bitstream* $C \in \{\mathtt{0},\mathtt{1}\}^{\star}$
    sender *encodes* $S$ to $C$, receiver *decodes* $C$ to $S$ (hopefully)
  - ⇝ what errors can be detected and/or corrected?
- all codes discussed here are ***block codes***
  - divide $S$ into *messages* $m \in \{\mathtt{0},\mathtt{1}\}^{k}$ of $k$ bits each ($k$ = *message length*)
  - encode each message (separately) as $C(m) \in \{\mathtt{0},\mathtt{1}\}^{n}$ ($n$ = *block length*, $n \geq k$)
  - ⇝ can analyze everything block-wise

# Block codes

- **model:**
  - want to send message $S \in \{0,1\}^\star$ (bitstream) across a *(communication) channel*
  - any bit transmitted through the channel might ***flip*** ($0 \to 1$ resp. $1 \to 0$)
    **no other errors** occur (no bits lost, duplicated, inserted, etc.)
  - instead of $S$, we send *encoded bitstream* $C \in \{0,1\}^\star$
    sender *encodes* $S$ to $C$, receiver *decodes* $C$ to $S$ (hopefully)
  - $\rightsquigarrow$ what errors can be detected and/or corrected?
- all codes discussed here are ***block codes***
  - divide $S$ into *messages* $m \in \{0,1\}^k$ of $k$ bits each ($k$ = *message length*)
  - encode each message (separately) as $C(m) \in \{0,1\}^n$ ($n$ = *block length*, $n \geq k$)
  - $\rightsquigarrow$ can analyze everything block-wise
- between $0$ and $n$ bits might be flipped

  invalid code
  - how many flipped bits can we definitely **detect**?
  - how many flipped bits can we **correct** without retransmit?

  i. e. decoding $m$ still possible

# Clicker Question

What is the Hamming distance between `heart` and `beard`?

beard

→ *sli.do/cs566*

# Code distance

$m \neq m' \implies C(m) \neq C(m')$

- each block code is an *injective* function $C : \{\mathtt{0}, \mathtt{1}\}^k \to \{\mathtt{0}, \mathtt{1}\}^n$

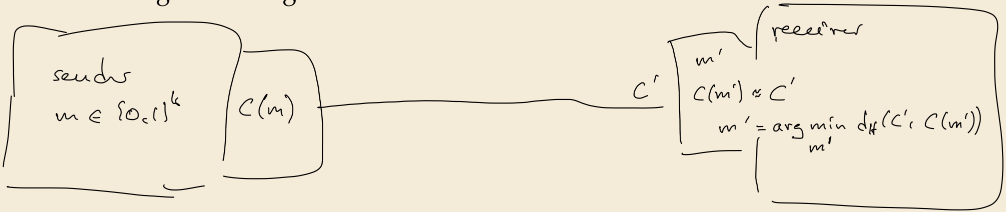
# Code distance

$m \neq m' \implies C(m) \neq C(m')$

- each block code is an *injective* function $C : \{0,1\}^k \to \{0,1\}^n$
- define $\mathcal{C}$ = set of all codewords = $C(\{0,1\}^k)$

$\rightsquigarrow \mathcal{C} \subseteq \{0,1\}^n$ $\boxed{|\mathcal{C}| = 2^k \text{ out of } 2^n\ n\text{-bit strings are valid codewords}}$

- decoding = finding closest valid codeword



# Code distance

$m \neq m' \implies C(m) \neq C(m')$

- each block code is an *injective* function $C : \{0,1\}^k \to \{0,1\}^n$
- define $\mathcal{C}$ = set of all codewords = $C(\{0,1\}^k)$

$\rightsquigarrow$ $\mathcal{C} \subseteq \{0,1\}^n$ $\quad$ $\boxed{|\mathcal{C}| = 2^k \text{ out of } 2^n\ n\text{-bit strings are valid codewords}}$

- decoding = finding closest valid codeword

$= \min_{m, m' \in \{0,1\}^k} d_H(C(m), C(m'))$

- ***distance of code:***
  $d$ = minimal Hamming distance of any two codewords = $\min_{x,y \in \mathcal{C}} d_H(x, y)$

# Code distance

$m \neq m' \implies C(m) \neq C(m')$

- each block code is an *injective* function $C : \{\texttt{0},\texttt{1}\}^k \to \{\texttt{0},\texttt{1}\}^n$
- define $\mathcal{C}$ = set of all codewords = $C(\{\texttt{0},\texttt{1}\}^k)$

$\rightsquigarrow$ $\mathcal{C} \subseteq \{\texttt{0},\texttt{1}\}^n$ $\boxed{|\mathcal{C}| = 2^k \text{ out of } 2^n\ n\text{-bit strings are valid codewords}}$

- decoding = finding closest valid codeword

- ***distance of code:***
  $d$ = minimal Hamming distance of any two codewords = $\min_{x,y \in \mathcal{C}} d_H(x, y)$

**Implications for codes**

1. Need distance $d$ to **detect** all errors flipping up to $d-1$ bits.
2. Need distance $d$ to **correct** all errors flipping up to $\left\lfloor \frac{d-1}{2} \right\rfloor$ bits.

# Lower Bounds

- Main advantage of concept of code distance:
  can *prove* lower bounds on block length

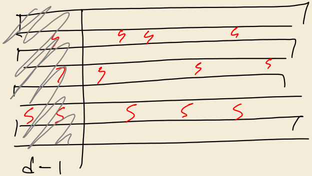
# Lower Bounds

- Main advantage of concept of code distance:
  can *prove* lower bounds on block length

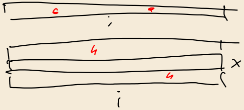
  otherwise no such code exists

  Given block length $n$, message length $k$, code distance $d$, we must have:

- ***Singleton bound:*** $2^k \le 2^{n-(d-1)} \quad \rightsquigarrow \quad n \ge k + \underline{d-1}$
  - *proof sketch:* We have $2^k$ codeswords with distance $d$
    after deleting the first $d-1$ bits, all are still distinct
    but there are only $2^{n-(d-1)}$ such shorter bitstrings.



# Lower Bounds

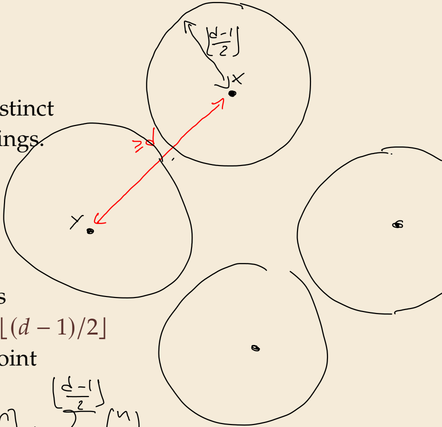


- Main advantage of concept of code distance:
  can *prove* lower bounds on block length

  otherwise no such code exists

  Given block length $n$, message length $k$, code distance $d$, we must have:

- ***Singleton bound:*** $2^k \leq 2^{n-(d-1)} \rightsquigarrow n \geq k + d - 1$
  - *proof sketch:* We have $2^k$ codeswords with distance $d$
    after deleting the first $d-1$ bits, all are still distinct
    but there are only $2^{n-(d-1)}$ such shorter bitstrings.




- ***Hamming bound:*** $2^k \leq \dfrac{2^n}{\sum_{f=0}^{\lfloor (d-1)/2 \rfloor} \binom{n}{f}}$
  - *proof idea:* consider "balls" of bitstrings around codewords
    count bitstrings with Hamming-distance $\leq t = \lfloor (d-1)/2 \rfloor$
    correcting $t$ errors means all these balls are disjoint
    so $2^k \cdot$ ball size $\leq 2^n$

$$\sum_{f=0}^{\lfloor (d-1)/2 \rfloor} [\#\text{strings w/ } d_H(x, \cdot) = f] = \sum_{f=0}^{\lfloor \frac{d-1}{2} \rfloor} \binom{n}{f}$$

$\rightsquigarrow$ We will come back to these.

## 8.6 Hamming Codes

# Parity Bit

- simplest possible error-detecting code: add a **parity bit**

$$\boxed{0\ 1\ 1\ 0\ 1\ 1\ 1\ 1\ 0}\boxed{0} \quad = \begin{cases} 0 & \text{if number of ones is even} \\ 1 & \text{if number of ones is odd} \end{cases}$$

$$0 \oplus 1 \oplus 1 \oplus 0 \oplus 1 \oplus 1 \oplus 1 \oplus 1 \oplus 0 \quad = 0$$

XOR

# Parity Bit

- simplest possible error-detecting code: add a **parity bit**

`0 1 1 0 1 1 1 1 0 | 0`

$$= \begin{cases} 0 & \text{if number of ones is even} \\ 1 & \text{if number of ones is odd} \end{cases}$$

$$0 \oplus 1 \oplus 1 \oplus 0 \oplus 1 \oplus 1 \oplus 1 \oplus 1 \oplus 0 \; = 0$$

XOR

**Implications for codes**

1. Need distance $d$ to **detect** all errors flipping up to $d-1$ bits.
2. Need distance $d$ to **correct** all errors flipping up to $\lfloor \frac{d-1}{2} \rfloor$ bits.

$\leadsto$ code distance 2

- can detect any single-bit error (actually, any odd number of flipped bits)
- used in many hardware (communication) protocols
  - PCI buses, serial buses
  - caches
  - early forms of main memory

# Parity Bit

- simplest possible error-detecting code: add a **parity bit**

`0 1 1 0 1 1 1 1 0 | 0`

$$= \begin{cases} 0 & \text{if number of ones is even} \\ 1 & \text{if number of ones is odd} \end{cases}$$

$$0 \oplus 1 \oplus 1 \oplus 0 \oplus 1 \oplus 1 \oplus 1 \oplus 1 \oplus 0 \quad = 0$$

XOR

⇝ code distance 2

- can detect any single-bit error (actually, any odd number of flipped bits)
- used in many hardware (communication) protocols
  - PCI buses, serial buses
  - caches
  - early forms of main memory

👍 very simple and cheap

👎 cannot correct any errors

# 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?

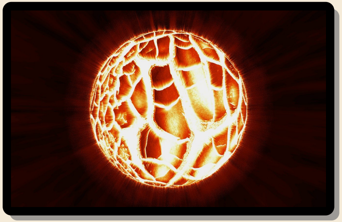
→ sli.do/cs566

# Error-correcting codes

any downtime is expensive!

- typical application: heavy-duty server RAM
  - bits can randomly flip (e. g., by cosmic rays)
  - individually very unlikely,
    but in always-on server with lots of RAM, it happens!

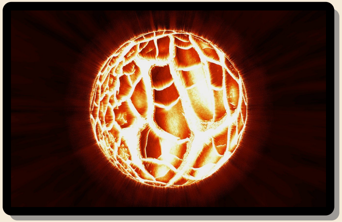
    https://blogs.oracle.com/linux/attack-of-the-cosmic-rays-v2

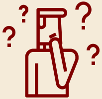


# Error-correcting codes

any downtime is expensive!

- typical application: heavy-duty server RAM
  - bits can randomly flip (e. g., by cosmic rays)
  - individually very unlikely,
    but in always-on server with lots of RAM, it happens!
    https://blogs.oracle.com/linux/attack-of-the-cosmic-rays-v2



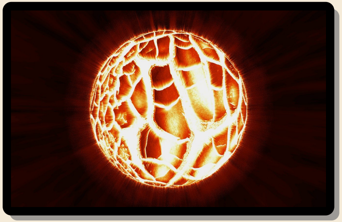


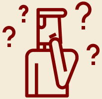
Can we **correct** a bit error without knowing where it occurred? How?

# Error-correcting codes

any downtime is expensive!

- typical application: heavy-duty server RAM
  - bits can randomly flip (e. g., by cosmic rays)
  - individually very unlikely,
    but in always-on server with lots of RAM, it happens!
    `https://blogs.oracle.com/linux/attack-of-the-cosmic-rays-v2`





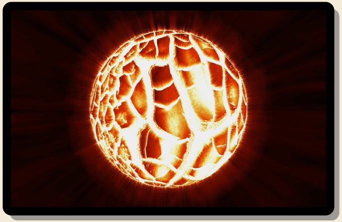
Can we **correct** a bit error without knowing where it occurred? How?

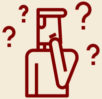
- Yes! store every bit *three times!*
  - upon read, do majority vote
  - if only one bit flipped, the other two (correct) will still win

# Error-correcting codes

any downtime is expensive!

- typical application: heavy-duty server RAM
  - bits can randomly flip (e. g., by cosmic rays)
  - individually very unlikely,
    but in always-on server with lots of RAM, it happens!
    `https://blogs.oracle.com/linux/attack-of-the-cosmic-rays-v2`





Can we **correct** a bit error without knowing where it occurred? How?

- Yes! store every bit *three times!*
  - upon read, do majority vote
  - if only one bit flipped, the other two (correct) will still win

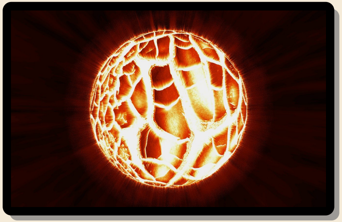
*triples* the cost!

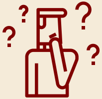
*You want WHAT!?!*

# Error-correcting codes

any downtime is expensive!

- typical application: heavy-duty server RAM
  - bits can randomly flip (e. g., by cosmic rays)
  - individually very unlikely,
    but in always-on server with lots of RAM, it happens!
    
    https://blogs.oracle.com/linux/attack-of-the-cosmic-rays-v2





Can we **correct** a bit error without knowing where it occurred? How?

- Yes! store every bit *three times!*
  - upon read, do majority vote
  - if only one bit flipped, the other two (correct) will still win

*triples* the cost!

*You want WHAT!?!*

instead of 200% (!)

Can do it with 11% extra memory!

# How to locate errors?

- **Idea:** Use several parity bits
  - each covers a **subset** of bits
  - clever subsets $\rightsquigarrow$ violated/valid parity bit pattern narrows down error

# How to locate errors?

- **Idea:** Use several parity bits
  - each covers a **subset** of bits
  - clever subsets $\rightsquigarrow$ violated/valid parity bit pattern narrows down error

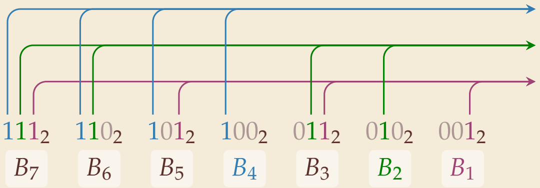
⚠ flipped bit can be one of the parity bits!

# How to locate errors?

- **Idea:** Use several parity bits
  - each covers a **subset** of bits
  - clever subsets $\rightsquigarrow$ violated/valid parity bit pattern narrows down error

⚠ flipped bit can be one of the parity bits!

- Consider $n = 7$ bits $B_1, \ldots, B_7$ with the following constraints:

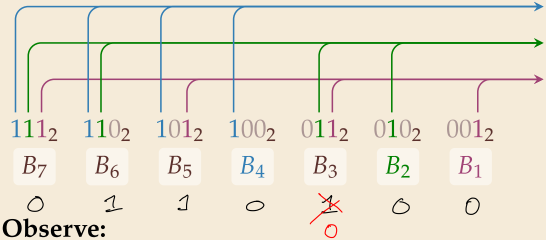



$$C_2 = B_4 \oplus B_5 \oplus B_6 \oplus B_7 \overset{!}{=} 0$$
$$C_1 = B_2 \oplus B_3 \oplus B_6 \oplus B_7 \overset{!}{=} 0$$
$$C_0 = B_1 \oplus B_3 \oplus B_5 \oplus B_7 \overset{!}{=} 0$$

# How to locate errors?

- **Idea:** Use several parity bits
  - each covers a **subset** of bits
  - clever subsets $\rightsquigarrow$ violated/valid parity bit pattern narrows down error
  - ⚠ flipped bit can be one of the parity bits!

- Consider $n = 7$ bits $B_1, \ldots, B_7$ with the following constraints:




$$C_2 = B_4 \oplus B_5 \oplus B_6 \oplus B_7 \overset{!}{=} 0$$
$$C_1 = B_2 \oplus B_3 \oplus B_6 \oplus B_7 \overset{!}{=} \cancel{0}\ 1$$
$$C_0 = B_1 \oplus B_3 \oplus B_5 \oplus B_7 \overset{!}{=} \cancel{0}\ 1$$

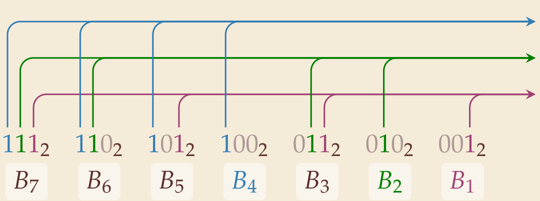
**Observe:**

- No error (all 7 bits correct) $\rightsquigarrow$ $C = C_2C_1C_0 = 000_2 = 0$ ✓
- What happens if (exactly) 1 bit, say $B_i$ flips?

# How to locate errors?

- **Idea:** Use several parity bits
  - each covers a **subset** of bits
  - clever subsets $\rightsquigarrow$ violated/valid parity bit pattern narrows down error
- ⚠ flipped bit can be one of the parity bits!

- Consider $n = 7$ bits $B_1, \ldots, B_7$ with the following constraints:




$$C_2 = B_4 \oplus B_5 \oplus B_6 \oplus B_7 \overset{!}{=} 0$$
$$C_1 = B_2 \oplus B_3 \oplus B_6 \oplus B_7 \overset{!}{=} 0$$
$$C_0 = B_1 \oplus B_3 \oplus B_5 \oplus B_7 \overset{!}{=} 0$$

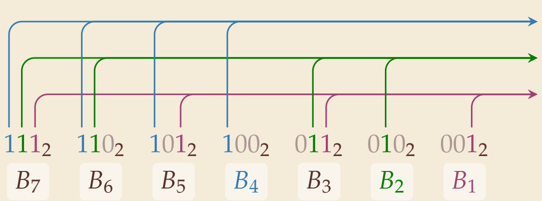
**Observe:**

- No error (all 7 bits correct) $\rightsquigarrow$ $C = C_2C_1C_0 = 000_2 = 0$ ✓
- What happens if (exactly) 1 bit, say $B_i$ flips?

$\boxed{C_j = 1 \text{ iff } j\text{th bit in binary representation of } i \text{ is } 1}$

# How to locate errors?

- **Idea:** Use several parity bits
  - each covers a **subset** of bits
  - clever subsets $\rightsquigarrow$ violated/valid parity bit pattern narrows down error
  - ⚠ flipped bit can be one of the parity bits!
- Consider $n = 7$ bits $B_1, \ldots, B_7$ with the following constraints:




$$C_2 = B_4 \oplus B_5 \oplus B_6 \oplus B_7 \stackrel{!}{=} 0$$
$$C_1 = B_2 \oplus B_3 \oplus B_6 \oplus B_7 \stackrel{!}{=} 0$$
$$C_0 = B_1 \oplus B_3 \oplus B_5 \oplus B_7 \stackrel{!}{=} 0$$

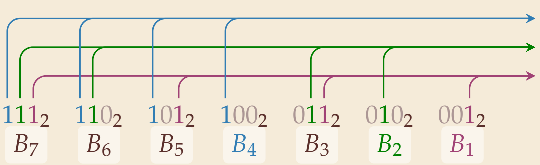
**Observe:**

- No error (all 7 bits correct) $\rightsquigarrow$ $C = C_2C_1C_0 = 000_2 = 0$ ✓
- What happens if (exactly) 1 bit, say $B_i$ flips?

$\boxed{C_j = 1 \text{ iff } j\text{th bit in binary representation of } i \text{ is } 1}$ $\rightsquigarrow$ *$C$ encodes **position of error**!*

# (7, 4) Hamming Code

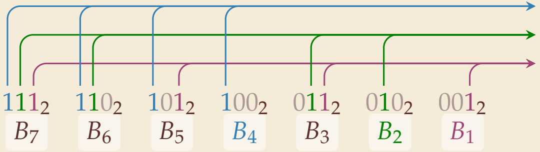
- *How can we turn this into a code?*




$$C_2 = B_4 \oplus B_5 \oplus B_6 \oplus B_7 \overset{!}{=} 0$$
$$C_1 = B_2 \oplus B_3 \oplus B_6 \oplus B_7 \overset{!}{=} 0$$
$$C_0 = B_1 \oplus B_3 \oplus B_5 \oplus B_7 \overset{!}{=} 0$$

# (7, 4) Hamming Code

- *How can we turn this into a code?*

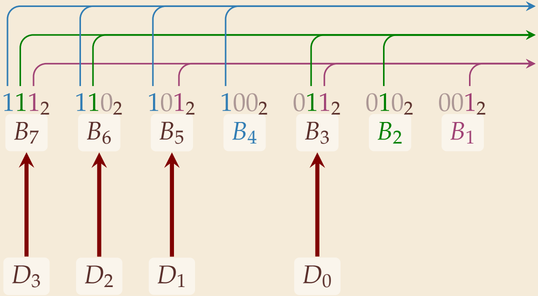



$$
\begin{aligned}
C_2 &= B_4 \oplus B_5 \oplus B_6 \oplus B_7 \stackrel{!}{=} 0\\
C_1 &= B_2 \oplus B_3 \oplus B_6 \oplus B_7 \stackrel{!}{=} 0\\
C_0 &= B_1 \oplus B_3 \oplus B_5 \oplus B_7 \stackrel{!}{=} 0
\end{aligned}
$$

- $B_4$, $B_2$ and $B_1$ occur only in one constraint each $\rightsquigarrow$ **define** them based on rest!
- **(7, 4)** ***Hamming Code*** **– Encoding**
  1. **Given:** message $D_3D_2D_1D_0$ of length $k = 4$

# (7, 4) Hamming Code

- *How can we turn this into a code?*

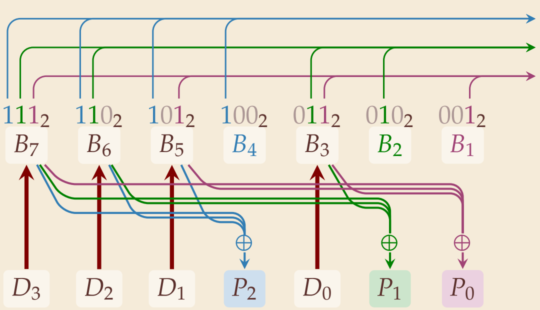



$$C_2 = B_4 \oplus B_5 \oplus B_6 \oplus B_7 \overset{!}{=} 0$$
$$C_1 = B_2 \oplus B_3 \oplus B_6 \oplus B_7 \overset{!}{=} 0$$
$$C_0 = B_1 \oplus B_3 \oplus B_5 \oplus B_7 \overset{!}{=} 0$$

- $B_4$, $B_2$ and $B_1$ occur only in one constraint each $\rightsquigarrow$ **define** them based on rest!
- **(7, 4) *Hamming Code* – Encoding**
  1. **Given:** message $D_3D_2D_1D_0$ of length $k = 4$
  2. copy $D_3D_2D_1D_0$ to $B_7B_6B_5B_3$

# (7, 4) Hamming Code

- *How can we turn this into a code?*



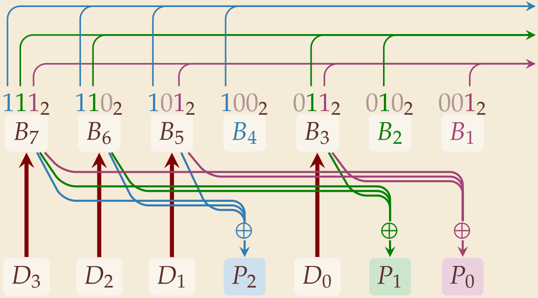

$$C_2 = B_4 \oplus B_5 \oplus B_6 \oplus B_7 \overset{!}{=} 0$$
$$C_1 = B_2 \oplus B_3 \oplus B_6 \oplus B_7 \overset{!}{=} 0$$
$$C_0 = B_1 \oplus B_3 \oplus B_5 \oplus B_7 \overset{!}{=} 0$$

$$P_2 = D_3 \oplus D_2 \oplus D_1$$
$$P_1 = D_3 \oplus D_2 \oplus D_0$$
$$P_0 = D_3 \oplus D_1 \oplus D_0$$

- $B_4$, $B_2$ and $B_1$ occur only in one constraint each $\rightsquigarrow$ **define** them based on rest!
- **(7, 4) *Hamming Code* – Encoding**
  1. **Given:** message $D_3D_2D_1D_0$ of length $k = 4$
  2. copy $D_3D_2D_1D_0$ to $B_7B_6B_5B_3$
  3. compute $P_2P_1P_0 = B_4B_2B_1$ so that $C = 0$

# (7, 4) Hamming Code

- *How can we turn this into a code?*




$$C_2 = B_4 \oplus B_5 \oplus B_6 \oplus B_7 \overset{!}{=} 0$$
$$C_1 = B_2 \oplus B_3 \oplus B_6 \oplus B_7 \overset{!}{=} 0$$
$$C_0 = B_1 \oplus B_3 \oplus B_5 \oplus B_7 \overset{!}{=} 0$$

$$P_2 = D_3 \oplus D_2 \oplus D_1$$
$$P_1 = D_3 \oplus D_2 \oplus D_0$$
$$P_0 = D_3 \oplus D_1 \oplus D_0$$

- $B_4$, $B_2$ and $B_1$ occur only in one constraint each $\rightsquigarrow$ **define** them based on rest!
- **(7, 4) *Hamming Code* – Encoding**
  1. **Given:** message $D_3D_2D_1D_0$ of length $k = 4$
  2. copy $D_3D_2D_1D_0$ to $B_7B_6B_5B_3$
  3. compute $P_2P_1P_0 = B_4B_2B_1$ so that $C = 0$
  4. send $D_3D_2D_1P_2D_0P_1P_0$

# (7, 4) Hamming Code – Decoding

- **(7, 4)** ***Hamming Code*** **– Decoding**

  1. **Given:** block $B_7B_6B_5B_4B_3B_2B_1$ of length $n = 7$
  2. compute $C$ (as above)
  3. if $C = 0$ no (detectable) error occurred
     otherwise, flip $B_C$ (the $C$th bit was twisted)
  4. return 4-bit message $B_7B_6B_5B_3$

# Clicker Question

What is the code distance of $(7,4)$ Hamming code?

- **A** $0$
- **B** $1$
- **C** $2$
- **D** $3$
- **E** $4$
- **F** $5$
- **G** $6$
- **H** $\geq 7$

→ sli.do/cs566

# Clicker Question

from 4 message bits ⇝ get 1 valid codeword

What is the code distance of $(7, 4)$ Hamming code?

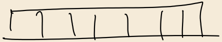
- **A** ~~0~~
- **B** ~~1~~
- **C** ~~2~~
- **D** 3 ✓
- **E** ~~4~~
- **F** ~~5~~
- **G** ~~6~~
- **H** ~~$\geq 7$~~

**Implications for codes**

1. Need distance $d$ to **detect** all errors flipping up to $d - 1$ bits.
2. Need distance $d$ to **correct** all errors flipping up to $\left\lfloor \frac{d-1}{2} \right\rfloor$ bits.

→ sli.do/cs566

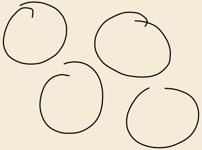
# (7, 4) Hamming Code – Properties



$t = \lfloor \frac{d-1}{2} \rfloor = 1$

ball size $= 1 + 7 = 8$

- **Hamming bound:**
  - $2^4$ valid 7-bit codewords (on per message)
  - any of the 7 single-bit errors corrected towards valid codeword
  - ⇝ each codeword covers 8 of all possible 7-bit strings
  - $2^4 \cdot 2^3 = 2^7$ ⇝ exactly cover space of 7-bit strings



# (7, 4) Hamming Code – Properties

- **Hamming bound:**
  - $2^4$ valid $7$-bit codewords (on per message)
  - any of the $7$ single-bit errors corrected towards valid codeword
  - $\rightsquigarrow$ each codeword covers $8$ of all possible $7$-bit strings
  - $2^4 \cdot 2^3 = 2^7 \quad \rightsquigarrow \quad$ exactly cover space of $7$-bit strings

- distance $d = 3$
- can *correct* any 1-bit error

# (7, 4) Hamming Code – Properties

- **Hamming bound:**
  - $2^4$ valid $7$-bit codewords (on per message)
  - any of the $7$ single-bit errors corrected towards valid codeword
  - ⇝ each codeword covers $8$ of all possible $7$-bit strings
  - $2^4 \cdot 2^3 = 2^7$ ⇝ exactly cover space of $7$-bit strings

- distance $d = 3$

- can *correct* any 1-bit error

- How about 2-bit errors?
  - We can *detect* that *something* went wrong.
  - **But:** above decoder mistakes it for a (different!) 1-bit error and "corrects" that
  - Variant: store one additional parity bit for entire block
  - ⇝ Can *detect* any 2-bit error, but *not correct* it.

# Hamming Codes – General recipe

- construction can be generalized:
  - Start with $n = 2^\ell - 1$ bits for $\ell \in \mathbb{N}$ (we had $\ell = 3$)
  - use the $\ell$ bits whose index is a power of 2 as parity bits
  - the other $n - \ell$ are data bits

# Hamming Codes – General recipe

- construction can be generalized:
  - Start with $n = 2^\ell - 1$ bits for $\ell \in \mathbb{N}$ (we had $\ell = 3$)
  - use the $\ell$ bits whose index is a power of 2 as parity bits
  - the other $n - \ell$ are data bits
- Choosing $\ell = 7$ we can encode entire word of memory (64 bit) with 11% overhead (using only 64 out of the 120 possible data bits)

# Hamming Codes – General recipe

- construction can be generalized:
  - Start with $n = 2^\ell - 1$ bits for $\ell \in \mathbb{N}$ (we had $\ell = 3$)
  - use the $\ell$ bits whose index is a power of 2 as parity bits
  - the other $n - \ell$ are data bits
- Choosing $\ell = 7$ we can encode entire word of memory (64 bit) with 11% overhead (using only 64 out of the 120 possible data bits)

👍 simple and efficient coding / decoding

👍 fairly space-efficient

# Outlook

- Indeed: $(2^\ell - 1,\ 2^\ell - \ell - 1)$ Hamming Code is "*perfect*" code

= matches Hamming lower bound

$\rightsquigarrow$ cannot use fewer bits . . .

  - **if** message length is $2^\ell - \ell - 1$ for $\ell \in \mathbb{N}_{\geq 2}$
    i. e., one of $1, 4, 11, 26, 57, 120, 247, 502, 1013, \ldots$
  - **and** we want to correct 1-bit errors

# Outlook

- Indeed: $(2^\ell - 1,\ 2^\ell - \ell - 1)$ Hamming Code is "*perfect*" code

  = matches Hamming lower bound

$\rightsquigarrow$ cannot use fewer bits . . .

  - **if** message length is $2^\ell - \ell - 1$ for $\ell \in \mathbb{N}_{\geq 2}$
    i. e., one of $1, 4, 11, 26, 57, 120, 247, 502, 1013, \ldots$
  - **and** we want to correct 1-bit errors

- For other scenarios, finding good codes is an active research area
  - information theory predicts that *almost all* randomly chosen codes are good(!)
  - but these are inefficient to decode

  $\rightsquigarrow$ clever tricks and constructions needed
  e. g. *low density parity check codes*

$\notin$ exam