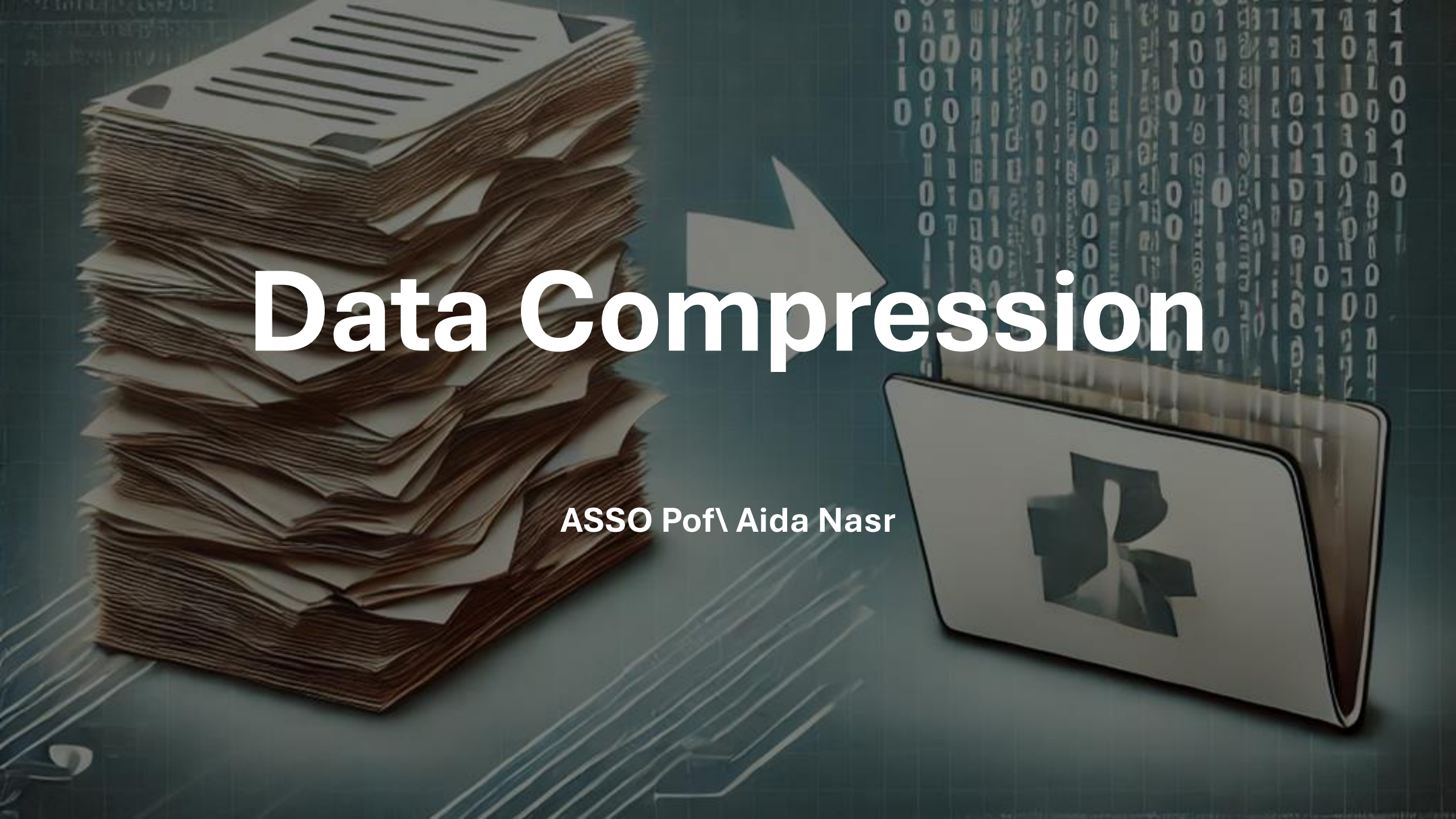


Data Compression

ASSO Pof\ Aida Nasr

Definition

- **Data compression** is a fundamental concept in computer science that involves reducing the size of data to optimize storage, transmission, and computation
- **Save Storage Space**
- **Reduce Time for Transmission**
- **Faster to Encode, Send, then Decode than to Send the Original**
- **Progressive Transmission**
  - Some compression techniques allow us to send the most important bits first so we can get a low resolution version of some data before getting the high fidelity version

What is Data and Information

- Data and information are not the same!
- Data is the medium through which information is transmitted.
- Data is what we use to send and share information.

❑**Analog data**

- Also called continuous data
- Represented by real numbers (or complex numbers)

❑ **Digital data**

- Finite set of symbols {a1, a2, … , am}
- All data represented as sequences (strings) in the symbol set.
- Example: {a,b,c,d,r} abracadabra
- Digital data can be an approximation to analog data

What is Data and Information

- The same amount of information can be represented by various amount of data
- Ex 1: Your friend, Ali, will meet you at the Airport in Cairo at 5 minutes past 6:00 pm tomorrow night
- Ex 2: Your friend will meet you at the Airport at 5 minutes past 6:00 pm tomorrow night
- Ex 3: Ali will meet you at Airport at 6:00 pm tomorrow night

Why Data Compression is Possible

- Most natural data has extra, unnecessary details.
- There is usually more data than actual useful information.
- Removing the extra data is called **compression**.
- But to understand the data, we need to **restore** it (decompression).
- **Information theory** helps us understand how much we can compress data and the best way to do it.

Data Redundancy

•**Inter Data Redundancy** means that some parts of the data are repeated or unnecessary.

•In **audio**, nearby sound samples are often similar (predictive encoding),

and silent parts can be removed.

•In **digital images**, nearby pixels on the same line are usually similar (spatial redundancy).

•In **digital videos**, in addition to spatial redundancy, frames close to each

•other in a video often look similar (temporal redundancy).

•**Compression** reduces one or more of these redundancies to make data smaller.

Types of Data Redundancy

- **Code:** a list of symbols (letters, numbers, bits etc.)
- **Code word:** a sequence of symbols used to represent a piece of information or an event (e.g., gray levels).
- **Code word length:** the number of symbols in each code word

Example: (binary code, symbols: 0,1, length: 3)

0: 000	4: 100
1: 001	5: 101
2: 010	6: 110
3: 011	7: 111

Code Redundancy

r_k: k-th Event or Code word

$P(r_k)$: probability of r_k

$l(r_k)$: # (the number) of bits for r_k

Expected value: $$E(X) = \sum_x xP(X = x)$$

Average # of bits: $$L_{avg} = E(l(r_k)) = \sum_{k=0}^{L-1} l(r_k)P(r_k)$$

For N symbols total number of bits: NL_{avg}

Code Redundancy

- $l(r_k)$ = **constant length**
- **Example:** for N x M image with 8 gray level (L=8)

r_k	$p_r(r_k)$	*Code 1*	$l_1(r_k)$
$r_0 = 0$	0.19	000	3
$r_1 = 1/7$	0.25	001	3
$r_2 = 2/7$	0.21	010	3
$r_3 = 3/7$	0.16	011	3
$r_4 = 4/7$	0.08	100	3
$r_5 = 5/7$	0.06	101	3
$r_6 = 6/7$	0.03	110	3
$r_7 = 1$	0.02	111	3

$$\text{Assume } l(r_k) = 3,\ L_{avg} = \sum_{k=0}^{7} 3P(r_k) = 3\sum_{k=0}^{7} P(r_k) = 3 \text{ bits}$$

Total number of bits: $3NM$

Code Redundancy

- $l(r_k)$ = variable length
- Consider the probability of the gray levels:

Table 6.1 Variable-Length Coding Example

variable length

r_k	$p_r(r_k)$	*Code 1*	$l_1(r_k)$	*Code 2*	$l_2(r_k)$
$r_0 = 0$	0.19	000	3	11	2
$r_1 = 1/7$	0.25	001	3	01	2
$r_2 = 2/7$	0.21	010	3	10	2
$r_3 = 3/7$	0.16	011	3	001	3
$r_4 = 4/7$	0.08	100	3	0001	4
$r_5 = 5/7$	0.06	101	3	00001	5
$r_6 = 6/7$	0.03	110	3	000001	6
$r_7 = 1$	0.02	111	3	000000	6

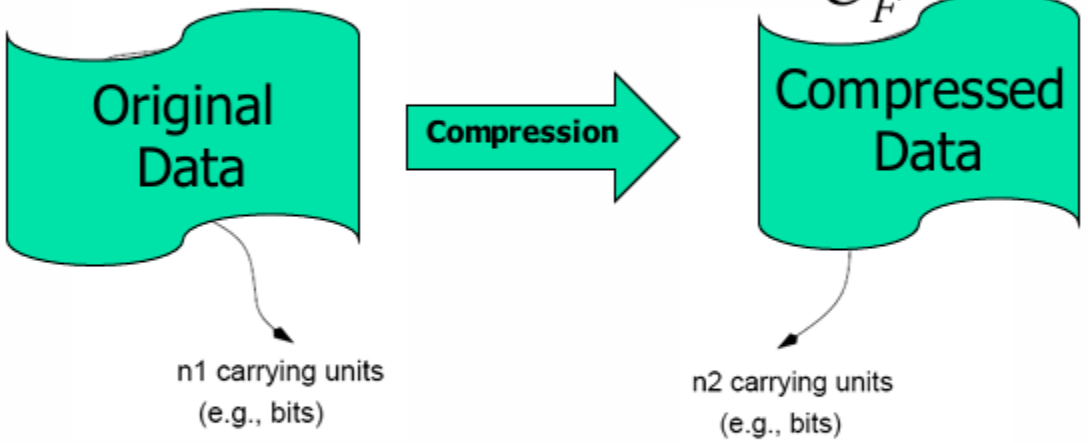
$$L_{avg} = \sum_{k=0}^{7} l(r_k)P(r_k) = 2.7 \text{ bits}$$

Total number of bits: $2.7NM$

Code Redundancy

- **Compression Factor:** $C_F = \frac{n_1}{n_2}$
- **Relative data redundancy:** $R_D = 1 - \frac{1}{C_F}$

Example:



if $C_F = \frac{10}{1}$ then $R_D = 1 - \frac{1}{10} = 0.9$

(90% of the data in dataset1 is redundant)

if $n_2 = n_1$ then $C_F = 1,\ R_D = 0$

if $n_2 \ll n_1$ then $C_F \to \infty,\ R_D \to 1$

if $n_2 \gg n_1$ then $C_F \to 0,\ R_D \to -\infty$

Code Redundancy

- $l(r_k)$ = **variable length**
- **Consider the probability of the gray levels:**

Table 6.1 Variable-Length Coding Example

variable length

r_k	$p_r(r_k)$	*Code 1*	$l_1(r_k)$	*Code 2*	$l_2(r_k)$
$r_0 = 0$	0.19	000	3	11	2
$r_1 = 1/7$	0.25	001	3	01	2
$r_2 = 2/7$	0.21	010	3	10	2
$r_3 = 3/7$	0.16	011	3	001	3
$r_4 = 4/7$	0.08	100	3	0001	4
$r_5 = 5/7$	0.06	101	3	00001	5
$r_6 = 6/7$	0.03	110	3	000001	6
$r_7 = 1$	0.02	111	3	000000	6

$$L_{avg}^{Code1} = \sum_{k=0}^{7} l_1(r_k) P_r(r_k) = 3\,bits$$

$$\text{Total number of bits(Code1)} : 3NM$$

$$L_{avg}^{Code2} = \sum_{k=0}^{7} l_2(r_k) P_r(r_k) = 2.7\,bits$$

$$\text{Total number of bits(Code2)} : 2.7NM$$

$$C_F = \frac{3}{2.7} = 1.11 \ (about 10\%) \qquad R_D = 1 - \frac{1}{1.11} = 0.099$$

Summary

- Variable-length coding reduces the number of bits required compared to fixed-length coding.
- Shorter codes are assigned to values with higher probabilities, improving storage efficiency and reducing data size.
- By using variable-length coding in images and videos, effective compression can be achieved without losing information.

Quiz

- **1. What is the primary goal of data compression?**

- A) Increase data size

B) Reduce storage space and transmission time

C) Introduce more redundancy in data

D) Make data harder to process

- [redacted]

Quiz

- **What does Inter Data Redundancy refer to?**
  A) The process of adding extra data for error detection
  B) The repetition or unnecessary presence of some parts of the data
  C) The process of encrypting data to improve security
  D) The method of storing data in multiple locations
- [redacted]

Quiz

- **What is the main advantage of variable-length coding over fixed-length coding?**
  A) It increases the number of bits required for storage
  B) It assigns longer codes to frequently occurring values
  C) It reduces the number of bits required by assigning shorter codes to more probable values
  D) It ensures that all values have the same code length
- [redacted]

Quiz

- **How does variable-length coding contribute to data compression?**
  A) By increasing redundancy in the data
  B) By assigning shorter codes to less probable values
  C) By using the same number of bits for all values
  D) By reducing storage size while preserving information
- [redacted]

What is the information?

- Information adds to people's knowledge.
- The more unknown or surprising a message is, the more informative it becomes.
- **Key Concept:**
- Surprise, unpredictability, and unexpectedness increase the amount of information in a message.
- **Example Comparison:**
- *Message 1:* "Two-thirds of the students passed all subjects."
- ***Message** 2:* "Two-thirds of the students scored 100% in all subjects."
- Both messages have similar word counts, but Message 2 is more informative because it presents an unexpected fact.

Data Compression

ASSO Pof\ Aida Nasr

Modeling Information

Key Concepts:

- The expectation of an event is measured by its probability.

High probability → High expectation (more likely to occur).

Low probability → Rare event (less likely to occur).

Zero probability → The event never happens.

Self-Information Formula:

- For a random event E with probability P(E), the amount of information (self-information) is:

- **Idea: For a random event E with probability P(E) the amount of information (self-information) is:**

$$I(E) = log(\frac{1}{P(E)}) = -log(P(E)) \text{ units of information}$$

  - **Note: I(E)=0 when P(E)=1**

First Order Information

🔵 **Definition of Events and Alphabet:**

- A set of random events $S = (r_1, r_2, ..., r_n)$ forms an **alphabet** in a source message.
- Each event r_k has a probability p_k.
- The total probability of all events sums to **1**:

$$\sum p_k = 1$$

- The source is **memoryless** (each event occurs independently).

🟢 **Measuring Self-Information:**

- The **amount of surprise (self-information)** in an event r_k is given by:

$$I(r_k) = -\log(P(r_k))$$

- **Lower probability = Higher information content** (rare events carry more information).

First Order Information

Key Insight:

- Frequent events (high probability) provide **less information.**
- Rare events (low probability) provide **more information.**

First Order Information

Self-Information:

- Measures information in **individual events or symbols.**
- To analyze an **entire source**, we need to consider **all possible symbols.**

First-Order Entropy:

- Measures the **average self-information** of all symbols.
- Defined mathematically as:

$$H = -\sum_{k=0}^{L-1} P(r_k) log(P(r_k))$$

- Represents **uncertainty** or **expected information** in a source.
- **Higher entropy** = More uncertainty (symbols are equally probable).

Entropy and Redundancy

Entropy (H):

- Represents the **average number of bits** needed to encode a symbol based on its probability distribution.
- It is the **Shannon lower bound** on the number of bits required per symbol in a **source model**.

Redundancy (R):

- Measures **the extra bits used beyond entropy** when encoding symbols.
- Defined as:

$$R = L_{avg} - H$$

- **where:** $$L_{avg} = E(l(r_k)) = \sum_{k=0}^{L-1} l(r_k)P(r_k)$$
- **Note: of L_{avg}= H, the R=0 (no redundancy)**

First Order Information

🔵 Probability of Symbols:

- Given an **alphabet** {a, b, c} with probabilities:
  - $P(a) = \frac{1}{8}$
  - $P(b) = \frac{1}{4}$
  - $P(c) = \frac{5}{8}$

🟢 Self-Information Calculation:

- **Self-information** is calculated using **log base 2**:
  - $I(a) = \log_2(8) = 3$
  - $I(b) = \log_2(4) = 2$
  - $I(c) = \log_2(8/5) = 0.678$

🔴 Conclusion:

- Receiving symbol "**a**" provides **more information** than receiving "**b**" or "**c**", since its probability is the lowest.

First Order Information

🔵 Case 1: Given Probabilities

- Consider an **alphabet** {a, b, c} with probabilities:
  - $P(a) = \frac{1}{8}$
  - $P(b) = \frac{1}{4}$
  - $P(c) = \frac{5}{8}$
- **Self-Information Calculation:**
  - $I(a) = \log_2(8) = 3$
  - $I(b) = \log_2(4) = 2$
  - $I(c) = \log_2(8/5) = 0.678$
- **Entropy Calculation:**
  - $H = \frac{1}{8} \times 3 + \frac{1}{4} \times 2 + \frac{5}{8} \times 0.678$
  - $H = 1.3$ **bits/symbol**

🟢 Case 2: Uniform Probabilities (Worst Case Scenario)

- Consider an **alphabet** {a, b, c} with equal probabilities:
  - $P(a) = \frac{1}{3}$
  - $P(b) = \frac{1}{3}$
  - $P(c) = \frac{1}{3}$
- Self-Information Calculation:
  - $I(a) = I(b) = I(c) = \log_2(3) = 1.6$
- Entropy Calculation (Worst Case):
  - $H = 3 \times (\frac{1}{3} \times \log_2(3))$
  - $H = 1.6$ bits/symbol

First Order Entropy

💡 Insights:

- ✔ Entropy is lower when probabilities are uneven (1.3 bits/symbol).
- ✔ Maximum entropy occurs in the worst case (equal probabilities) (1.6 bits/symbol).
- ✔ More uncertainty = higher entropy 🚀.

First Order Entropy Estimation

- **It is not easy to estimate H reliably!**

21	21	21	95	169	243	243	243
21	21	21	95	169	243	243	243
21	21	21	95	169	243	243	243
21	21	21	95	169	243	243	243

image

Gray Level	*Count*	*Probability*
21	12	3/8
95	4	1/8
169	4	1/8
243	12	3/8

3/8=12/32

- **First order estimate of H:**

$$H = -\sum_{k=0}^{3} P(r_k) log(P(r_k)) = 1.81 \text{ bits/pixel}$$

Total bits: 4 x 8 x 1.81 = 58 bits

First Order Entropy Estimation

💡 Conclusions:

✔ The more distinct gray levels there are and the more evenly they are distributed, the higher the entropy (H).

✔ If the image contains only a few gray levels, the entropy will be lower.

✔ Entropy represents **the level of uncertainty or randomness in the distribution of values**, which is crucial for data compression and efficient image representation. 🚀

Second order entropy Estimation

- **Second order estimate of H:**
  - **Use relative frequencies of pixel blocks :**

21	21	21	95	169	243	243	243
21	21	21	95	169	243	243	243
21	21	21	95	169	243	243	243
21	21	21	95	169	243	243	243

image

Gray Level Pair	*Count*	*Probability*
(21, 21)	8	1/4
(21, 95)	4	1/8
(95, 169)	4	1/8
(169, 243)	4	1/8
(243, 243)	8	1/4
(243, 21)	4	1/8

$$H = -\sum_{k=0}^{5} P(r_k) log(P(r_k)) = 1.25 \text{ bits/pixel}$$

Total bits: $4 \times 8 \times 1.25 = 40$ bits

Entropy Estimation

First-order Estimate:

- Provides a **lower bound** on achievable compression.
- Based only on **direct frequency counts** of elements.

Higher-order Estimates:

- Reveal **inter-data redundancy** in the dataset.
- Offer a more **accurate** entropy estimation.

Why Apply Transformations?

- **Redundant data** indicates potential for better compression.
- **Mathematical transformations & compression techniques** improve efficiency.

💡 Conclusions:

✔ **Second-order entropy** considers pixel blocks, reducing redundancy and leading to lower entropy (H).

✔ **Lower entropy** means the image is more predictable, which improves **compression efficiency**.

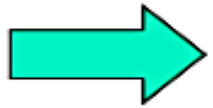
✔ Compared to first-order entropy (1.81 bits/pixel), second-order entropy (1.25 bits/pixel) reduces the total bits needed to represent the image. 🚀

Entropy Estimation

- **For example, consider differences:**

Original Image Data:

21	21	21	95	169	243	243	243
21	21	21	95	169	243	243	243
21	21	21	95	169	243	243	243
21	21	21	95	169	243	243	243



Applying Difference Transformation:

21	0	0	74	74	74	0	0
21	0	0	74	74	74	0	0
21	0	0	74	74	74	0	0
21	0	0	74	74	74	0	0

- **High variation between values → Higher entropy (H = 1.81 bits/pixel)**

Entropy After Transformation:

- New entropy = 1.41 bits/pixel (lower than 1.81).
- Compression improves as redundancy increases.

Gray Level or Difference	*Count*	*Probability*
0	16	1/2
21	4	1/8
74	12	3/8

$$H = -\sum_{k=0}^{2} P(r_k) log(P(r_k)) = 1.41 \text{ bits/pixel}$$

Entropy Estimation

- **Further Improvement Possible:**
  - Second-order entropy estimate suggests **H = 1.25 bits/pixel** is achievable.
  - **Better transformations** can further reduce entropy for **optimal compression**.

Estimating Entropy & Data Compression

- 🔵 **Understanding Entropy in Compression**
- **Entropy** measures the amount of **information or randomness** in data.
- Lower entropy → **better compression** is possible.
- First-order entropy provides a **lower bound** on compression efficiency.
- Higher-order entropy helps detect **hidden data redundancy**.

Estimating Entropy & Data Compression

- ✅ Key Takeaways
- ✓ Transformations (like difference calculation) reduce entropy.

  ✓ Lower entropy means better data compression.

  ✓ Higher-order entropy helps find optimal compression strategies.

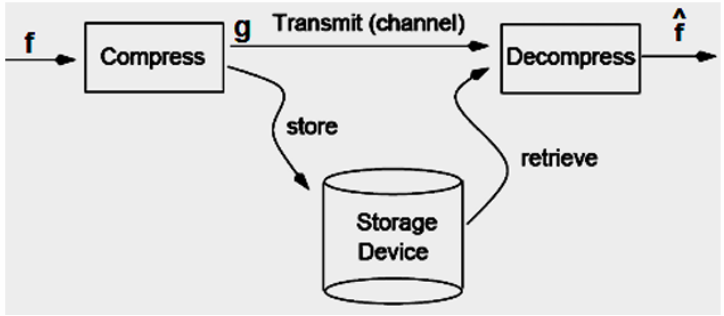
Data Compression

ASSO Pof\ Aida Nasr

Goal of Compression

Goals of Compression:

- **Conserve storage space** – Reduces data size, allowing more information to be stored.
- **Reduce transmission time** – Smaller data sizes lead to faster transmission over networks.
- **Progressive transmission** – Enables data to be gradually restored while being transmitted.
- **Reduce computation** – Minimizes processing power required to handle compressed data.
- The original data (**f**) is compressed using a **Compress** unit.
- The compressed data (**g**) is either **transmitted** through a channel or **stored** in a storage device.
- When needed, the data is **retrieved** from storage or received through the transmission channel.
- The **Decompress** unit restores the original data (**^f**).



Basic Data Types Explanation

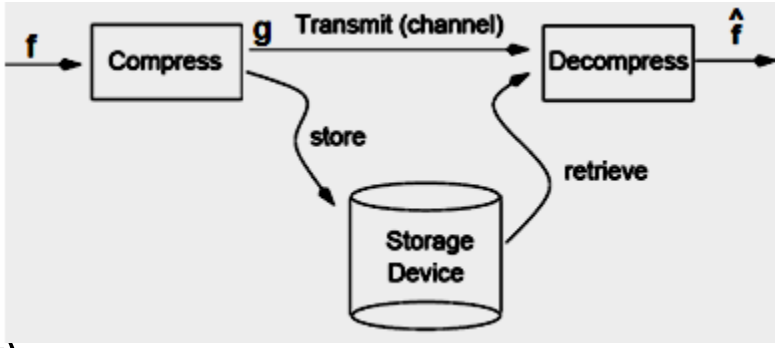
- **Definition of Data in Compression:**
  Data in the context of compression includes any digital form of factual information that a computer program can process.
- **Types of Data:**
  Data can be classified into different types:
  - **Text**
  - **Binary**
  - **Graphic**
  - **Image**
  - **Audio**
  - **Video**
  - **Multimedia**
- **Binary Representation:**
  All digital data is ultimately stored as **0s and 1s** in a **binary format**.

Basic Data Types

- 🟣 **Text**: Represented by **8-bit ASCII code**.
  - Found in files with **.txt, .tex, .doc** extensions.
- 🔴 **Binary**: Includes **database files, spreadsheets, executables, and program codes**.
  - Typically stored in **.bin** files.
- 🟠 **Image**: Stored as a **2D array of pixels**, where each pixel has a color code.
  - Examples: **.bmp** (bitmap image), **.psd** (Photoshop format).
- 🔵 **Graphics**: Based on **vectors or mathematical equations**.
  - Example: **.png** (Portable Network Graphics).
- 🟢 **Sound**: Represented by **wave functions**.
  - Example: **.wav** sound files.

Basic Data Compression Concepts

- **Lossless Compression (f=f^)**
- Also called **entropy coding** or **reversible coding**.
- **Information preserving** (original data is perfectly restored).
- **Low compression ratios** (less reduction in size).



- 🔴 **Lossy Compression (f≠f^)**
- Also called **irreversible coding**.
- **Not information preserving** (some data is lost permanently).
- **High compression ratios** (better size reduction but lower quality).
- ⚙️ **Compression Ratio**
- |f| is the number of bits in the original data.

$$\text{Compression Ratio} = \frac{|g|}{|f|}$$

⚖️ **Trade-off: Quality vs. Compression Ratio**

- Higher compression ratio = **Smaller file size but lower quality.**
- Lower compression ratio = **Larger file size but better quality.**

Lossless Compression

- ✅ **Data is not lost - The original data is fully restored.**
  - **Text compression** (e.g., documents, source code).
  - **Compression of computer binary files** (e.g., executable files).
- ⚙️ **Compression Ratio**
  - Typically **no better than 4:1** (modest reduction in size).
- 📊 **Statistical-Based Techniques**
  - 🔷 **Huffman coding**
  - 🔷 **Arithmetic coding**
  - 🔷 **Golomb coding**

📚 **Dictionary-Based Techniques**

- 🔷 **LZW, LZ77**
- 🔷 **Sequitur**
- 🔷 **Burrows-Wheeler Method**

📌 **Existing Standards**

•Morse code, Braille, gzip, zip, bzip, GIF, JBIG, Lossless JPEG

Lossy Compression

- ❌ **Data is lost, but not too much**
  - 🎵 **Audio** (e.g., MP3, AAC).
  - 🎥 **Video** (e.g., MP4, MPEG).
  - 🖼️ **Still images, medical images, photographs** (e.g., JPEG).
- ⚙️ **Compression Ratio10:1**
  - compression often maintains **high fidelity** results.

📊 **Major Techniques**
- 🔷 **Vector Quantization**
- 🔷 **Wavelets**
- 🔷 **Block Transforms**

📌 **Standards**
- 🖼️ **JPEG** (for images).
- 🎥 **MPEG** (for videos).

Compression Algorithm Design

- 🛠️ **Systematic Approach (8 Stages)**

- 1 **Description of the Problem** – Identify the need for compression.
  2 **Mathematical Modeling** – Develop a theoretical framework.
  3 **Design of the Algorithm** – Define the logic and structure.
  4 **Verification of the Algorithm** – Ensure correctness and efficiency.
  5 **Estimation of Computational Complexity** – Analyze performance.
  6 **Implementation** – Convert the design into code.
  7 **Program Testing** – Validate with real-world data.
  8 **Documentation** – Record details for future use.

Description of the Problem

- 🔷 **Objective:** Develop an efficient algorithm to **remove redundancy** from specific data types.
- 📝 **Key Considerations:**
- ✅ **Understanding the problem –** Identify data format and output constraints.
  ✅ **Asking the right questions –** What redundancies exist? What efficiency is required?
  ✅ **Clarifying vague problems –** Apply **Divide and Conquer** to break down complex issues.
  ✅ **Deliverables:** A **precise problem specification** detailing **input & output** requirements.

Mathematical Modeling

- 🔷 **Definition:**
  Modeling sets up an **environment** to observe variables and explore system behavior. It extends **problem description** into a **mathematical formulation** using rules & relationships.
- 📊 **Importance in Algorithm Design:**
- ✅ **Determines algorithm approach –** The right model influences **design choices**.
  ✅ **Describes source data –** Helps identify **redundancies** in the data.
  ✅ **Forms the knowledge base –** The model is the **foundation** on which compression algorithms operate.
- 📌 **In data compression:**
  Mathematical models **analyze redundancy** and define **effective ways to represent** data.

Mathematical Modeling in Compression Algorithm Design

- 🔷 **Mathematical modeling** is a crucial field in **mathematics, statistics, and computer science**, widely researched in **data compression**.
- 📊 **Common Mathematical Models in Data Compression:**
  - 1️⃣ **Physical Model**
  - Uses **known physics** of the data source.
  - Based on **data generation processes** or **empirical observations**.
  - 2️⃣ **Probability Model**
  - Applies **probability theory** to describe the data source.
  - Helps in **predicting data patterns**.

Mathematical Modeling in Compression Algorithm Design

- 3️⃣ **Markov Model**
- Utilizes **Markov chain theory**.
- Assumes **future states depend only on the present**, not past history.
- 4️⃣ **Composite Model**
- **Combines multiple models** dynamically.
- Uses a **switching mechanism** to select the appropriate model.

- 📌 **Why Modeling?**
  ✅ Helps in understanding **data redundancy**.
  ✅ Provides a structured way to **optimize compression efficiency**.
  ✅ Supports the design of **advanced algorithms** like **Huffman, Arithmetic, and Lempel-Ziv coding**.

Design of the Algorithm in Data Compression

- **Algorithm design** is a crucial step where we apply **all available algorithmic knowledge and techniques** to develop an effective compression solution.
- 🛠 **Key Considerations in Algorithm Design:**
  - 1 **Mathematical Model Selection**
  - The chosen **mathematical model** (e.g., probability, Markov, physical) directly influences the algorithm design.
  - 2 **Refining the Model**
  - Adding further **details** to enhance accuracy.
  - Incorporating **feedback mechanisms** for optimization.
  - 3 **Use of Data Structures & Tools**
  - Selecting efficient **data structures** (e.g., trees, hash tables, stacks).
  - Utilizing **abstract data types** and **existing libraries** for efficiency.

- 4 **Top-Down Approach**
- Breaking the problem into **smaller subproblems**.
- Identifying and integrating **existing efficient algorithms** to solve parts of the problem.
- 📌 **Example Approaches:**
    - ✅ **Huffman coding** uses **priority queues and trees**.
    - ✅ **Lempel-Ziv compression (LZ77, LZ78)** uses **sliding windows and hash tables**.
    - ✅ **Arithmetic coding** relies on **probability models and range division**.
- **Final Goal:** Develop an optimized **compression algorithm** that balances **speed, accuracy, and efficiency**!

Verification of the Algorithm in Data Compression

- **Why is verification important?**

- Ensures the **correctness** of the algorithm before implementation.

- Helps avoid **fundamental flaws** that could waste time during coding.

Verification of the algorithm

- ✅ **Key Aspects of Verification:**
- 1 **Correctness of the Algorithm**
- Mathematically prove that the algorithm works as expected.
- Use **logical reasoning** or **comparisons with existing solutions**.
- 2 **Compression Quality Assessment**
- Measure the **compression ratio**

$$\text{Compression Ratio} = \frac{\text{Original Size}}{\text{Compressed Size}}$$

- Calculate **saving percentage**:

$$\text{Saving Percentage} = \left(1 - \frac{\text{Compressed Size}}{\text{Original Size}}\right) \times 100$$

Verification of the algorithm

- 3️⃣ **Coder Efficiency Analysis**
- Compare the **average length of codewords** to **entropy** (Shannon's theorem).
- Lower difference = **higher efficiency**.

- **Deliverables at the End of Verification:**
  - ✅ **Correctness proofs** or logical reasoning.
    ✅ **Performance metrics** (compression ratio, efficiency).
    ✅ **Comparisons** with other algorithms to validate effectiveness.
- 📌 **Final Goal:** Ensure the algorithm is **correct, efficient, and achieves the expected compression quality!** 🎯

📌 Estimation of Computational Complexity in Data Compression

- **Why is Computational Complexity Important?**
- Helps predict **algorithm behavior** before implementation.
- Saves **time and resources** by avoiding inefficient designs.

- ✅ **Key Aspects of Complexity Estimation:**
  - 1️⃣ **Efficiency Analysis**
  - Evaluate **time complexity** (how execution time grows with input size).
  - Evaluate **space complexity** (how much memory is required).
  - 2️⃣ **Comparing Candidate Algorithms**
  - Choose at least **two compression algorithms** and compare their:
    - **Best-case, worst-case, and average-case time complexity.**
    - **Compression speed** (encoding/decoding time).
    - **Memory consumption** (RAM usage, disk space).

Implementation Phase

- 🔷 **Programming Languages & Tools**
- **No restriction** on programming language choice.
- **MATLAB** is popular for prototyping due to its built-in functions.
- Other options: **Python (NumPy, SciPy), C++, Java, or Rust** for high-performance implementations.
- 🔷 **Best Practices in Implementation**
  1. **Modular Code** – Break algorithms into **reusable functions**.
  2. **Code Optimization** – Use **efficient data structures** and minimize redundant computations.
  3. **Scalability** – Ensure the algorithm works for **small and large datasets**.

🛠️ Program Testing

- 🔷 **Key Aspects of Testing**
  - ✅ **Unit Testing** – Test individual functions for correctness.
  - ✅ **Integration Testing** – Ensure different parts of the algorithm work together.
  - ✅ **Performance Testing** – Measure **compression speed & memory usage**.
  - ✅ **Edge Case Testing** – Test with **very large files, random data, and worst-case inputs**.
- 🔷 **Standard Testing Methods**
  - **Benchmarking**: Compare against known compression algorithms.
  - **Using Test Benches**: Standard datasets for evaluation (e.g., **Calgary Corpus, Silesia Corpus**).
  - **Error Checking**: Ensure no unintended data loss in **lossless compression**.

📄 Documentation Phase

- 🔷 **Why is Documentation Important?**
- Helps in **future maintenance** and improvements.
- Enables **other researchers** to understand and replicate the work.
- 🔷 **What to Document?**
  - ✅ **Algorithm Description** – Explain **compression steps and techniques used**.
  - ✅ **Code Comments** – Provide **clear explanations in the source code**.
  - ✅ **Testing Results** – Include **performance metrics, compression ratios, and benchmarks**.
- 📌 **Final Goal:** A well-documented and tested compression algorithm ready for real-world use! 🎯

Data Compression

ASSO Pof\ Aida Nasr

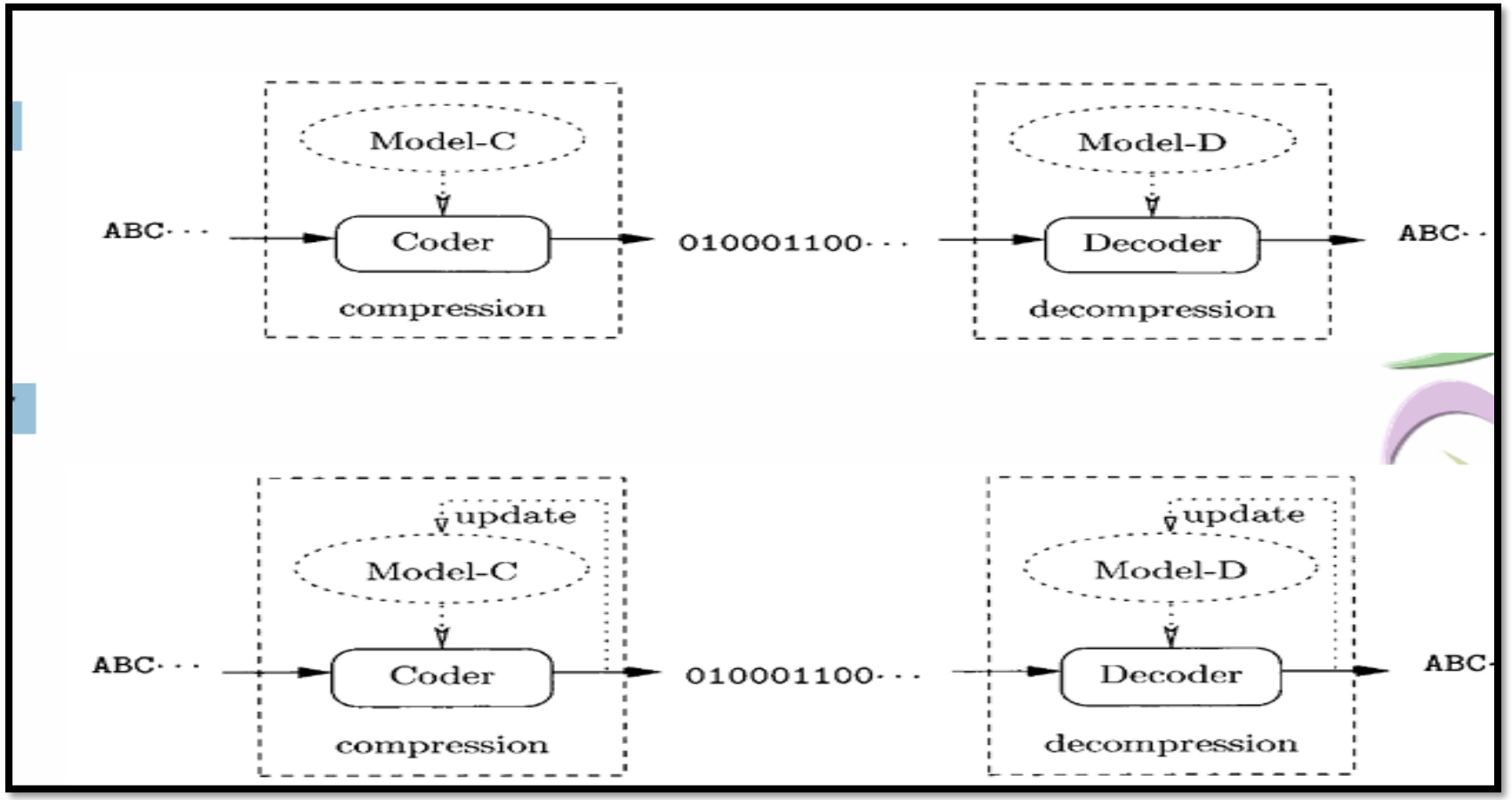
Key Points of Compression Methods 🎯

- 🔷 **Modeling:** Represents knowledge about the source domain and identifies data redundancy.
  🔷 **Coding:** Uses an **encoder (coder)** to compress the input based on the model.
  🔷 **Independent Coder:** The coder can function **with or without** a model.
  🔷 **Decoding Structure:** Similar to compression, consisting of a **model and a decoder**.
  🔷 **Compression Process:** The **model analyzes** data patterns, while the **coder generates** the compressed output.
  🔷 **Decompression Process:** A **decoder reconstructs** the original data using the corresponding model.
  🔷 **Unified Structure:** Both compression and decompression rely on a **model-coder framework** for efficiency.

Types of Compression Algorithms

- **Static (Non-Adaptive) System:**
- The **model remains unchanged** throughout the compression and decompression process.
- It does not adapt to variations in input data.
- **Adaptive System:**
- The **model evolves dynamically** based on the input data or feedback from the output.
- Some adaptive algorithms **start with an empty model** and build it as they process the data.

Types of Compression Algorithms



Types of Compression Algorithms Based on Codeword Length

- **Fixed-to-Fixed:**
- Each symbol **before** compression has a fixed number of bits.
- After compression, it is **encoded** using a fixed-length bit sequence.
- **Example:**
- **Before compression:** A → 8 bits, B → 8 bits, C → 8 bits, D → 8 bits (ASCII)
- **After compression:** A → 00, B → 01, C → 10, D → 11

Types of Compression Algorithms Based on Codeword Length

- **Fixed-to-Variable:**
- Each symbol **before** compression has a fixed number of bits.
- After compression, it is **encoded** using a variable-length bit sequence.
- **Example:**
  - **Before compression**: A → 8 bits, B → 8 bits, C → 8 bits, D → 8 bits
  - **After compression**: A → 0, B → 10, C → 101, D → 0101

Types of Compression Algorithms Based on Codeword Length

- **Variable-to-Fixed:**
- A sequence of symbols with **variable-length** representation before compression.
- After compression, it is **encoded** using a fixed-length bit sequence.
- **Example:**

•**Before compression**: ABCD → 32 bits, ABCDE → 40 bits, BC → 16 bits

•**After compression**: ABCD → 00, ABCDE → 01, BC → 11

Types of Compression Algorithms Based on Codeword Length

- **Variable-to-Variable:**
- A sequence of symbols with **variable-length** representation before compression.
- After compression, it remains **variable-length** but optimized.
- **Example:**

•Before compression: ABCD → 32 bits, ABCDE → 40 bits, BC → 16 bits, BBB → 24 bits

•After compression: ABCD → 0, ABCDE → 01, BC → 1, BBB → 0001

Summary 

- ✅ **Fixed-to-Fixed:** Same length before and after compression.
- ✅ **Fixed-to-Variable:** Fixed before compression, variable after.
- ✅ **Variable-to-Fixed:** Variable before compression, fixed after.
- ✅ **Variable-to-Variable:** Variable before and after compression.

Symbols & Encoding in Compression

- 🔷 **Definition of a Symbol:**
- A **symbol** is a unit of input data in a compression algorithm.
- It can be:
  - A **letter** or a **special character** in text.
  - An **audio sample** in sound data.
  - A **pixel value** in an image.
- **Data Representation:**
- A text, image, audio, or video file is a **sequence of symbols**, either **one-dimensional** (text) or **multi-dimensional** (image, audio, video).

Symbols & Encoding in Compression

- **Encoding & Decoding:**

- **Encoding:** Assigning **codewords** to symbols in a source.
- **Decoding:** Reconstructing the **original symbols** from the compressed data.
- **Symbol Representation in Compression:**
- If the source alphabet is $\mathbf{S = (r_1, r_2, \ldots, r_n)}$, its digital representation is a **code set** $\mathbf{S' = (c_1, c_2, \ldots, c_n)}$.
- Each **symbol** $\mathbf{r_j}$ is assigned a unique **codeword** $\mathbf{c_j}$ where $\mathbf{j = 1, 2, \ldots, n}$.

Symbols & Encoding in Compression

- **Compression as Encoding & Decompression as Decoding:**
- **Compression = Encoding** (mapping symbols to compressed codewords).
- **Decompression = Decoding** (recovering original symbols from codewords).

Compression vs. Data Reliability

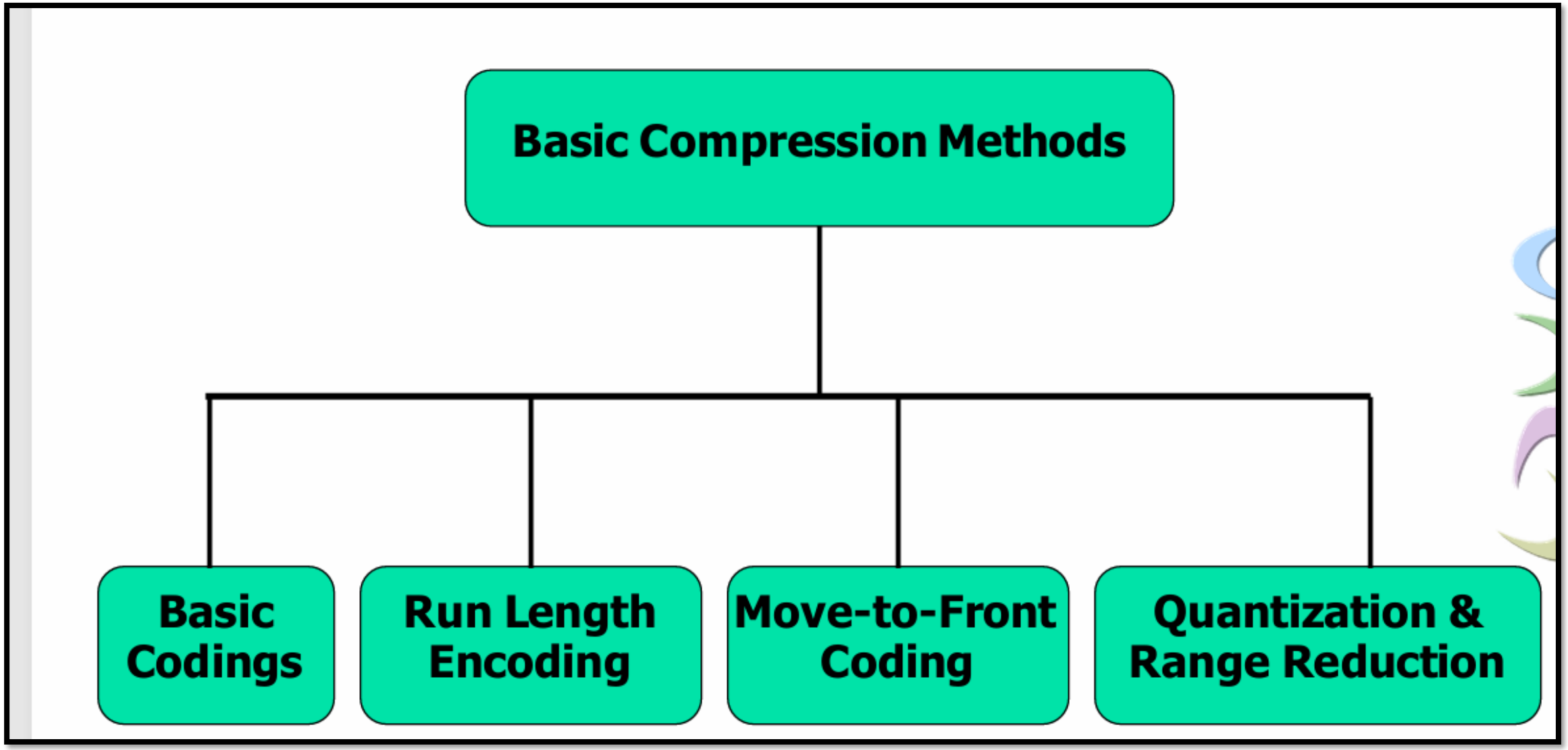
- **Compression & Reliability Are Opposites:**
- **Compression = Removes redundancy → Less reliable.**
- **Data integrity = Adds redundancy → More reliable.**

(such as **check bits** and **parity bits**)

- **error detection & correction** is a newer field, while **compression techniques have existed for centuries**.

Basic Methods

Asso. Prof. Aida Nasr


Basic Compression Methods
Basic Codings
Run Length Encoding
Move-to-Front Coding
Quantization & Range Reduction

Basic Coding Methods 🎯

- Digital data can be represented using two main types of **codewords**:

- 1️⃣ **Fixed-Length Codewords**
- Each symbol is assigned a **code of the same length**.
- Simple but inefficient for variable-frequency symbols.

- **Examples:**
  - ✅ **CDC Display Code** – Used in early computing displays.
  - ✅ **ASCII Codes** – Standard encoding for text characters (8-bit per character).
  - ✅ **Braille Coding** – Fixed-dot patterns for visually impaired readers.
  - ✅ **Baudot Code** – Early telegraph and teletype encoding (5-bit per character).
  - ✅ **Tunstall Code** – Used in lossless data compression (fixed-length dictionary).

Basic Coding Methods 🎯

- 2️⃣ **Variable-Length Codewords**
- Symbols are **represented by codes of different lengths** based on frequency.
- More efficient for compression and transmission.

- **Examples**:
  - ✅ **Self-Delimiting Coding** – Ensures no code overlaps another.
  - ✅ **Suffix Coding** – Codes are structured based on suffix patterns.
  - ✅ **Prefix Coding** – No code is a prefix of another (e.g., Huffman coding).
  - ✅ **Unary Coding** – Represents numbers using a series of 1s followed by 0.
  - ✅ **Phase-in Coding** – A structured encoding method.
  - ✅ **Golomb Coding** – Used in data compression and image coding.

- ✅ **Fixed-length coding is simple but inefficient, while variable-length coding optimizes space by adapting to symbol frequency**

CDC Display Code (Fixed-Length Coding)

- **Overview:**
- The **CDC (Control Data Corporation) Display Code** is a **6-bit fixed-length encoding system**.
- It was used in **second-generation computers** and some **third-generation** ones.
- Suitable for representing **uppercase letters, digits, and basic symbols**.

CDC Display Code (Fixed-Length Coding)

- **Why Only 6 Bits?**
- Early computers didn’t have display monitors.
- Output was sent to **printers**, which could only print a **limited** **set of characters** (≤ 64 symbols).
- 6 bits = 2^6 **= 64 possible characters**, which was sufficient

CDC Display Code (Fixed-Length Coding)

- **Structure:**
- Uses a **fixed-length 6-bit encoding** for each character.
- The table you provided shows **bit positions and their corresponding characters**.
- 🔷 **Example Characters in CDC Code:**
- **Letters:** A, B, C, … Z
- **Digits:** 0, 1, 2, … 9
- **Symbols:** +, -, *, /, =, $, (,)
- Fixed-length codes like CDC are simple but inefficient for large character sets. Modern systems use variable-length codes like UTF-8

$2^3=8$

2	1	0

Bits 543	Bit positions 210							
	0	1	2	3	4	5	6	7
0		A	B	C	D	E	F	G
1	H	I	J	K	L	M	N	O
2	P	Q	R	S	T	U	V	W
3	X	Y	Z	0	1	2	3	4
4	5	6	7	8	9	+	-	*
5	/	(	)	$	=	sp	,	.
6	≡	[	]	:	≠	_	∨	∧
7	↑	↓	<	>	≤	≥	¬	;

3
4
5

CDC Display Code (Fixed-Length Coding)

- **Understanding "Bit positions 210" and "Bits 543" in the Table**
- 1 **"Bit positions 210"**
- **Represents the least significant bits (LSB).**
  - These **3 bits determine the column** in the table.
  - Possible values: **0 to 7** (since 3 bits = 2^3 = 8 values).
- 2 **"Bits 543"**
- **Represents the most significant bits (MSB).**
  - These **3 bits determine the row** in the table.
  - Possible values: **0 to 7** (3 bits = 8 values).

CDC Display Code (Fixed-Length Coding)

- **How to Find a Character Using the Table?**
- Each **character** is determined by **3 bits from Bits 543 (row) and 3 bits from Bit positions 210 (column).**
  This allows **64 unique symbols** (since $2^3 \times 2^3 = 8 \times 8 = 64$).
- 🔷 **Example 1:**

0	1	1	1	0	0

- ✅ Given:
- **Bits 543 = 3 → Row 3**(since counting starts from 0).
- **Bit positions 210 = 4 → Column 4.**
  🔷 **The corresponding symbol is "1".**

Bits 543	Bit positions 210 0	1	2	3	4	5	6	7
0		A	B	C	D	E	F	G
1	H	I	J	K	L	M	N	O
2	P	Q	R	S	T	U	V	W
3	X	Y	Z	0	1	2	3	4
4	5	6	7	8	9	+	-	*
5	/	(	)	$	=	sp	,	.
6	≡	[	]	:	≠	_	∨	∧
7	↑	↓	<	>	≤	≥	¬	;

CDC Display Code (Fixed-Length Coding)

- 🔷 **Example 2:**

- ✅ **"HELLO 123"**
  ➞ H = 001000, E = 000101 …

Braille Coding (Fixed-Length Coding) ⠿

- **Developed by:** Louis Braille in the 1820s.
  - **Used for:** Reading and writing for the visually impaired.

- **Structure:** ⠿
- Each character is represented by a **3×2 grid** of dots (a "Braille cell").
- Dots can be **raised** or **flat**, functioning like **binary data (6 bits)**.
- Since $2^6 = 64$, there are **64 possible Braille symbols**.

A	B	C	D	E	F	G	H	I	J	K	L	M
N	O	P	Q	R	S	T	U	V	W	X	Y	Z

Table 1.1: The 26 Braille Letters.

and	for	of	the	with	ch	gh	sh	th

Table 1.2: Some Words and Strings in Braille.

Braille Coding (Fixed-Length Coding) ⠿

- **Encoding Letters and Words:**
- **Alphabet letters** (A-Z) are mapped to unique Braille cells.
- **Digits and punctuation** also use unique patterns.
- Some **common words and letter sequences** (like "the", "and", "th") have dedicated symbols to improve efficiency.
- ✅ **Braille is a fixed-length coding system where each symbol is always represented by 6 dots, making it both simple and efficient for tactile reading.**

Braille Coding (Fixed-Length Coding) ⠿

Examples of Braille and Baudot Code

🔷 Braille Code Examples ⠿

Braille represents letters, numbers, and common words using **6-dot cells**.

📌 Example Sentence in Braille:

🔷 **"HELLO"** → ⠓ ⠑ ⠇ ⠇ ⠕

Character	Braille Representation
A	⠁ (1)
B	⠃ (1-2)
C	⠉ (1-4)
D	⠙ (1-4-5)
E	⠑ **(1-5)**
and	⠯ (Shortcut for "and")
the	⠮ (Shortcut for "the")

Baudot Code (Fixed-Length Coding) 📡

- 🔷 **Developed by: J. M. E. Baudot** in **1880** for telegraph communication.
  🔷 **Character Length: 5-bit encoding per character.**
  🔷 **Adoption:** Became **International Telegraph Code No. 1** by **1950** and was used in **early computers (first & second generation).**

Baudot Code (Fixed-Length Coding) 📡

- **How It Works**
- 🔷 **5-bit length** allows for **32 unique combinations** (since $2^5 = 32$).
  🔷 However, **Baudot encodes more than 32 characters** by introducing:
- **Letter Shift (LS)** → **Switch to letter mode.**
- **Figure Shift (FS)** → **Switch to number/symbol mode.**
  🔷 This **doubles** the number of possible characters:

32 (letters) + 32 (figures) - 2 (LS & FS codes) = 62 total characters.

Baudot Code (Fixed-Length Coding) 

Example of Encoding:

- **Baudot Code was an early yet efficient fixed-length encoding system, paving the way for modern digital communication!**

5-bit Code	Letter Mode	Figure Mode
10100	A	1
11001	Z	%
01011	J	?
11111	(Letter Shift - LS)	(Figure Shift - FS)

Baudot Code Examples 📡

Baudot uses **5-bit binary codes** for letters and symbols. It switches between **Letter Mode** and **Figure Mode** using special shift codes.

📌 **Example Encoding in Baudot Code:**

🔹 **"HELLO 123"**

➡ **Letter Mode:** 10111 10000 10011 10011 10101 **(HELLO)**

➡ **Figure Shift:** 11111 (Switch to Numbers)

➡ **Figure Mode:** 10000 10001 10010 (1 2 3)

✅ **These examples show how both Braille and Baudot use fixed-length codes to efficiently encode**

5-bit Code	Letter Mode (A-Z)	Figure Mode (Numbers /Symbols)
10000	A	1
10001	B	2
10010	C	3
11111	Letter Shift (LS)	Figure Shift (FS)

Data Compression

ASSO Pof\ Aida Nasr

Quantization

- **Definition:**

  Quantization means **restricting a variable to discrete values** instead of a continuous range.

- **Quantization in Data Compression:**
- It is used in two primary ways:
- 🔷 **1) Reducing Large Numbers to Small Numbers**
- Large numbers take up more space, while small numbers require less storage.
- This process **reduces file size**, leading to **compression**.
- However, **smaller numbers hold less information**, making this a **lossy compression** method.

Quantization

- 🔷 **2) Digitizing Analog Data**
- Analog signals (e.g., voltages changing over time) are **converted into digital values**.
- Using **smaller numbers** improves compression but increases **data loss**.
- This technique is widely used in **speech compression methods**.
- In summary, **quantization balances compression efficiency and data loss**—the more compression, the more information is sacrificed.

Scalar Quantization: Discrete Quantization in Number-Based Compression

- 🔷 **Example: Naïve Discrete Quantization of 8-bit Numbers Method**:
- 1 Take an 8-bit number (range: **0 to 255**).

  2 Remove the **least significant 4 bits** (zero them out).

  3 The result is a **4-bit quantized value** (only 16 possible values instead of 256).

  4 The compression factor is **fixed at 2** (since each quantized value requires half the original storage).

Example Calculation:

Original 8-bit Value (Decimal)	Binary Representation (8-bit)	Quantized Value (Keep Upper 4 Bits, Set LSBs to 0)	Quantized Value (Decimal)
43	`00101011`	`00100000`	32
156	`10011100`	`10010000`	144
230	`11100110`	`11100000`	224

Scalar Quantization: Discrete Quantization in Number-Based Compression

- 🔷 **Limitations of This Method**

❌ **Too much information loss** – Removing 4 bits significantly reduces data accuracy.

❌ **Low compression gain** – A factor of **2** isn't impressive compared to more advanced techniques.

❌ **Not practical** – The trade-off between compression and information loss makes this method inefficient. While simple, this approach **sacrifices too much detail for minimal compression benefits**.

Uniform Quantization Sequences

- **Improved Approach : Uniform Quantization Sequences**
- Uniform quantization divides the input range into equal-sized intervals, where each input value is mapped to the nearest representative value. The spacing between quantized values is determined by a parameter **s**, known as the **quantization step size**.
- We refine our approach by assuming **unsigned 8-bit numbers**, meaning the input values range from **0 to 255**.

Uniform Quantization Sequences

- **Defining the Quantization Process**

1. We choose a **spacing parameter s**.
2. We create a **uniform sequence** of quantized values:

$$0, s, 2s, \ldots, ks$$

such that $(k+1)s > 255$ and $ks \leq 255$.

3. Each input symbol n is mapped to its nearest value in this sequence.

Uniform Quantization Sequences

Example 1:Here, every input number is rounded to the nearest multiple of **3**.
0,3,6,9,12,...,252,255

Original Value	Quantized Value
5	6
23	24
151	150

Example 2: **Uniform Quantization with s=4**
If we set **s=4**, the quantized values are:
0,4,8,12,...,252,255

Original Value	Quantized Value
5	4
23	24
151	152

Example 3: **Uniform Quantization with s=8 (More Aggressive Quantization)**
For **s=8**, the quantized values are:
0,8,16,24,...,248,255

Original Value	Quantized Value
5	8
23	24
151	152

Uniform Quantization Sequences

- **Examples of Uniform Quantization Sequences**

✓ **If** s=3→ **Quantized values:** 0,3,6,9,12,...,

✓ **If** s=4 → **Quantized values:** 0,4,8,12,...,252,255

Uniform Quantization Sequences

- **Advantages of This Method**

- ✅ **More control over precision** – Adjusting s changes the balance between compression and accuracy.
  ✅ **Better than naïve bit removal** – Instead of blindly discarding bits, we **map values smartly**, reducing information loss.
  ✅ **Flexible compression** – A larger sss increases compression but also increases quantization error.
- By choosing an optimal **s**, we achieve a better trade-off between **compression efficiency and data retention**.

General Formula for Uniform Quantization

Each input value x is quantized to:

$$Q(x) = s \times \text{round}\left(\frac{x}{s}\right)$$

where s is the quantization step size.

```python
def uniform_quantization(value, step_size):
    """Perform uniform quantization with given step size."""
    return round(value / step_size) * step_size

# Example values
original_values = [5, 23, 151]
step_sizes = [3, 4, 8]

# Display results for different step sizes
for s in step_sizes:
    quantized_values = [uniform_quantization(v, s) for v in original_values]
    print(f"Step size {s}: {quantized_values}")
```

Summary

- ✅ **Smaller s → More detail retained** but **less compression**
  ✅ **Larger s → More compression** but **more loss of information**
  ✅ **Uniform quantization is simple but may not be optimal for non-uniform data distributions**

Distance-Based Quantization Approach

- Instead of using **uniform quantization**, a **distance-based quantization** approach ensures that each number is **at most** a certain distance (d) away from a quantized value. This allows for **better distribution of values** and **reduced quantization error** while maintaining a small set of representative values.

Example: Quantizing 8-bit Values into Only 8 Values (Using 3 Bits)

Since 3-bit values can represent $2^3 = 8$ **quantized values**, we need to choose **8 representative values** that cover the range **[0, 255]** efficiently.

🔹 **Step 1:** Divide the range into **8 segments**.

🔹 **Step 2:** Select one representative value for each segment, ensuring that no original value is more than d units away.

Chosen Quantized Values:

$$\{8, 41, 74, 107, 140, 173, 206, 239\}$$

Each input value will be mapped to the closest value in this set.

Distance-Based Quantization Approach

- **Example with Only 8 Quantized Values (Expressed in 3 Bits)**
- If we limit to **8 values**, we compute the sequence: **8, 41, 74, 107, 140, 173, 206, 239**
- 🔷 **When This Method Works Best**

  ✓ **Useful when input symbols are equally probable** (memoryless sources).

  ✓ **Good for controlled compression** – Balances precision and storage.

  ✓ **Better than uniform spacing** in some cases, since it **minimizes worst-case quantization error**.
- By carefully selecting **d and the number of quantized values**, we can achieve a **better trade-off between accuracy and compression**!

Example Mappings

Original 8-bit Value	Closest Quantized Value	3-bit Representation
5	8	000
39	41	001
72	74	010
105	107	011
138	140	100
172	173	101
205	206	110
238	239	111

```python
import numpy as np

# Define the 8 chosen quantized values
quantized_values = np.array([8, 41, 74, 107, 140, 173, 206, 239])

def distance_based_quantization(value):
    """Map input value to the closest quantized value."""
    return quantized_values[np.abs(quantized_values - value).argmin()]

# Example values
original_values = [5, 39, 72, 105, 138, 172, 205, 238]
quantized_results = [distance_based_quantization(v) for v in original_values]

# Display results
for orig, quant in zip(original_values, quantized_results):
    print(f"Original: {orig:3d} → Quantized: {quant:3d}")
```

Distance-Based Quantization Approach

- 🔷 **Example with d=16**
- The range **[0, 255]** is divided into **7 segments of size 33** each.
- The remaining **25 numbers** are adjusted to start **12 units** from 0.
- This produces the **quantized sequence:**

12, 33, 45, 78, 111, 144, 177, 210, 243, 255

- **Every input number is at most 16 units away** from one of these values

📝 Example of Quantization Using These Values:

Original 8-bit Value	Closest Quantized Value
5	12
40	33
55	45
90	78
120	111
160	144
190	177
220	210
250	243

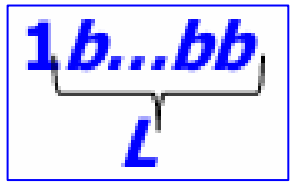
Quantization for Non-Uniformly Distributed Data

- 📌 **Problem:** If most input values are **small** (close to zero) and only a few are **large**, a **uniform** quantization method may not be efficient.

- 📌 **Solution:** The sequence of quantized values should **follow the same distribution as the input data**, meaning:
  ✓ **Many small values**
  ✓ **Few large values**

Method: Window-Based Quantization

- Step 1: Choose a length parameter L.
- Step 2: Construct a "window" of the form: 1b...bb(L bits)



- Step 3: Align this window under each bit position in the 8-bit number.
- Step 4: If the window extends beyond 8 bits, truncate the excess bits.
- Step 5: Shift the window left and pad with zeros.

📝 Example Calculation

Given 8-bit input data:

$$[100, 110, 95, 105, 180, 190, 170, 185]$$

- We divide it into **two windows** (since window size = 4):
  - **Window 1:** [100, 110, 95, 105]
  - **Window 2:** [180, 190, 170, 185]

Step 3: Compute the Local Reference for Each Window

Let's choose the **mean** as the reference value:

- **Window 1 Reference:**

$$(100 + 110 + 95 + 105)/4 = 102.5 \approx 102$$

- **Window 2 Reference:**

$$(180 + 190 + 170 + 185)/4 = 181.25 \approx 181$$

Step 4: Quantize Each Value Relative to the Reference

Each value in the window is now stored as its **difference from the reference** instead of the absolute value.

Original Value	Reference	Quantized Value (Relative Difference)
100 → 102 → **-2**	102	-2
110 → 102 → **+8**	102	+8
95 → 102 → **-7**	102	-7
105 → 102 → **+3**	102	+3
180 → 181 → **-1**	181	-1
190 → 181 → **+9**	181	+9
170 → 181 → **-11**	181	-11
185 → 181 → **+4**	181	+4

Method: Window-Based Quantization

Instead of storing **absolute values**, we only store the **relative differences**:

[−2,+8,−7,+3,−1,+9,−11,+4]

Since these values are **smaller** than the original values, they require **fewer bits to encode**, leading to **compression**

💡 **Why is Window-Based Quantization Useful?**

✅ **Reduces data range** → Smaller values require fewer bits to store.

✅ **Adaptive to local variations** → Better for **non-uniform data** like images and audio.

✅ **Can improve compression efficiency** in **predictive coding** techniques.

Vector Quantization (VQ)

- **What is Vector Quantization?**
- Vector quantization is a **generalization of scalar quantization**, where data is compressed **in blocks** rather than as **individual values**.
- **Where is Vector Quantization Used?**
- ✅ **Image compression**
  ✅ **Audio compression**
  ✅ **Compressing digitally sampled analog data (DSAD)** such as:
- Audio samples
- Scanned images
- This type of data originates from analog sources but has been digitized

Vector Quantization (VQ)

- **Why is Vector Quantization More Efficient than Scalar Quantization?**
- 🔷 **Scalar quantization** compresses **each value separately** (e.g., each pixel in an image or each audio sample).
  🔷 **Vector quantization** compresses **groups of values together as a unit** (blocks).
- Since **natural data contains repetitive patterns**, compressing **blocks** instead of individual values **leads to better compression ratios** without significant quality loss.

Vector Quantization (VQ)

- **Relationship Between Vector Quantization and Block Coding**
- 🟢 When compressing data using **blocks instead of individual symbols**, we use **block coding techniques**.

  🟢 As the **block size increases**, compression methods based on block coding can **approach the entropy limit**, achieving **optimal compression efficiency** while preserving important information.

Vector Quantization (VQ)

- **Simple Example**
- 🔷 **Scalar Quantization:**
- If an image consists of **individual pixels**, each pixel is compressed separately.
  🔷 **Vector Quantization:**
- Instead of compressing each pixel, the image is **divided into small blocks (e.g., 2×2 pixels)**, and each block is compressed as a unit.
- Since **neighboring blocks often have similar patterns**, **higher compression efficiency** is achieved.

Vector Quantization (VQ)

- Why is Vector Quantization Effective?
- ✅ Captures relationships between adjacent values
  ✅ Improves compression ratio without excessive information loss
  ✅ Approaches optimal performance as defined by entropy theory
- 🎯 Would you like a Python example demonstrating vector quantization?