Unit 5: Data Compression (8 LH)

5.1 Basic Knowledge About Storage Space

Storage space refers to the capacity of a storage medium to hold data. In the context of data compression, understanding storage space is essential as it determines how efficiently digital data, including multimedia like images, videos, and animations, can be stored and transmitted.

Key Concepts

1. Definition of Storage Space

- **Storage Space**: The total amount of data that can be stored on a storage medium, measured in units such as bytes, kilobytes (KB), megabytes (MB), gigabytes (GB), and terabytes (TB).
- Storage is available in various forms like:
  - Hard Drives (HDD)
  - Solid State Drives (SSD)
  - Cloud Storage
  - External Storage Devices (USB drives, memory cards)

2. Importance of Storage Space

- **Cost Efficiency**: Reducing file sizes saves storage costs.
- **Improved Transmission**: Smaller files are faster to transmit over networks.
- **Device Compatibility**: Low-storage devices like mobile phones benefit from compressed files.

3. Storage and Multimedia

- Multimedia files like videos, audio, and images are often large and consume significant storage space.
- Example file sizes:
  - Uncompressed image: 10–50 MB
  - Uncompressed video: 1 GB for a few minutes
  - High-quality audio: 50–100 MB per song

4. Data Compression and Storage Optimization

- **Data Compression**: Reduces the size of digital files while maintaining acceptable quality.
- Compression techniques balance:
  - File size reduction
  - Data quality preservation
- Examples:
  - Lossless Compression (e.g., ZIP, PNG)
  - Lossy Compression (e.g., MP3, MP4, JPEG)

5. Storage Medium Hierarchy

- **Primary Storage**: RAM, faster but temporary storage.
- **Secondary Storage**: HDDs and SSDs, used for permanent data storage.
- **Tertiary Storage**: Backup systems or archival storage (e.g., tapes, Blu-rays).
- **Cloud Storage**: Internet-based storage, scalable and accessible from anywhere.

6. Factors Affecting Storage Space

- **Resolution and Quality**:
  - High-resolution images and videos require more space.
  - Compression can optimize quality vs. size trade-offs.
- **File Format**:
  - Some formats (e.g., PNG, FLAC) retain more data and take more space.
  - Others (e.g., JPEG, MP3) are optimized for smaller sizes.
- **Redundancy**:
  - Duplicate or unnecessary data increases storage usage.
  - Compression removes redundancy.

Examples of File Size Reduction Through Compression

Type of Data	Original Size	Compressed Size
Image (TIFF to JPEG)	5 MB	500 KB
Video (RAW to MP4)	1 GB	100 MB
Audio (WAV to MP3)	50 MB	5 MB

Applications of Optimized Storage Space

1. **Multimedia Storage**:
   - Saving movies, games, and high-quality images on limited devices.
2. **Cloud-Based Services**:
   - Efficiently storing user data across servers.
3. **Data Archiving**:
   - Long-term storage of records, reducing redundancy.
4. **IoT Devices**:
   - Low-storage devices benefit from compressed data.

5.2 Basic Knowledge About Coding Requirements

Coding requirements refer to the set of guidelines, standards, and expectations that define how software should be written and structured. These requirements ensure that the code is efficient, maintainable, readable, and secure. Understanding coding requirements is essential for developers to produce high-quality software.

Key Concepts

1. Coding Standards

- **Definition**: Coding standards are a set of rules that dictate how code should be written, formatted, and organized. These standards help make code more consistent, readable, and maintainable.
- **Examples**:
  - Naming conventions (e.g., camelCase for variables, PascalCase for class names).
  - Indentation style (e.g., 4 spaces vs. tabs).
  - Function/method documentation (using comments and docstrings).

2. Code Readability

- **Clarity**: The code should be easy to understand and self-explanatory. Developers should use meaningful variable and function names.
- **Consistency**: Following the same style and naming conventions throughout the codebase.
- **Comments**: Proper comments help explain complex logic, making it easier for other developers to understand and modify the code.

3. Modularity and Reusability

- **Modular Code**: Breaking down the code into smaller, reusable functions or classes. Each function should have a single responsibility.
- **Reusability**: Writing code that can be reused in different parts of the program or even across different projects. This reduces redundancy and improves maintainability.

4. Code Optimization

- **Efficiency**: Writing code that performs well and uses minimal system resources (e.g., memory, processing power).
- **Algorithm Optimization**: Using efficient algorithms and data structures to improve the performance of the application.
- **Avoiding Code Duplication**: Repeated code should be refactored into functions or methods to avoid redundancy.

5. Security Requirements

- **Input Validation**: Ensuring that all user inputs are validated and sanitized to prevent security vulnerabilities like SQL injection and cross-site scripting (XSS).
- **Encryption**: Using proper encryption methods to protect sensitive data.
- **Authentication & Authorization**: Ensuring that only authorized users have access to specific parts of the application.

6. Error Handling

- **Graceful Handling**: Handling errors in a way that the program continues to run smoothly or provides meaningful feedback to the user.
- **Logging**: Using logging mechanisms to track errors and application behavior, which is helpful for debugging.

7. Testing

- **Unit Testing**: Writing tests for individual components of the code to verify their correctness.
- **Integration Testing**: Ensuring that different parts of the system work well together.
- **Automated Testing**: Setting up automated test suites that run tests each time the code changes to ensure reliability.

8. Version Control

- **Git**: Using a version control system like Git to track changes in the codebase, collaborate with other developers, and manage different versions of the software.
- **Branching and Merging**: Organizing code changes into branches for features, bug fixes, or experiments and then merging them into the main codebase after review.

9. Documentation

- **Code Documentation**: Writing detailed documentation for the code to explain how functions, classes, and modules work.
- **Project Documentation**: Including instructions for how to set up, use, and maintain the software, especially for open-source projects or collaborative work environments.

10. Compliance with Standards

- **Industry Standards**: Following widely accepted standards for coding in specific programming languages (e.g., PEP 8 for Python, Java Code Conventions for Java).
- **Regulatory Compliance**: Ensuring that the software complies with relevant laws and regulations (e.g., GDPR for data protection).

Best Practices for Coding

- **Plan and Design**: Before writing code, plan and design the structure of the program.
- **Write Clean, Simple Code**: Avoid overly complex code, and aim for simplicity and readability.
- **Refactor Regularly**: Continuously improve the code to make it more efficient and readable.
- **Collaborate and Review**: Code reviews by peers help catch mistakes and improve the quality of the code.

5.3 Source, Entropy, and Hybrid Coding

1. Source Coding

Source coding refers to the process of converting the information from a source (like text, audio, or video) into a binary representation that can be efficiently transmitted or stored. The goal is to represent the data in the most compact form without losing the original content.

- **Objective**: Minimize the number of bits used to represent a given source while maintaining the integrity of the information.
- **Examples**: Text compression (e.g., Huffman encoding), image compression (e.g., JPEG), and audio compression (e.g., MP3).

---

2. Entropy Coding

Entropy coding is a type of lossless data compression technique that assigns variable-length codes to input characters based on their frequencies of occurrence. It is grounded in the concept of entropy, which is a measure of the uncertainty or randomness of information.

- **Objective**: Reduce the size of data by assigning shorter codes to more frequent symbols and longer codes to less frequent ones.
- **Application**: Used in various file compression formats like ZIP, PNG, and JPEG.

---

3. Huffman Encoding

Huffman Encoding is one of the most widely used entropy coding techniques that assigns shorter codes to more frequently occurring symbols and longer codes to less frequent

symbols. It is a variable-length prefix coding method, meaning no codeword is a prefix of another codeword.

- **How it Works**:
    1. **Frequency Analysis**: First, calculate the frequency of each symbol in the source data.
    2. **Build a Tree**: Construct a binary tree where the least frequent symbols are placed at the leaves, and the most frequent symbols are closer to the root.
    3. **Generate Codes**: Assign binary codes based on the tree structure (moving left adds a '0', and moving right adds a '1').
    4. **Assign Codes to Symbols**: The more frequent symbols will have shorter codes, and less frequent symbols will have longer codes.
- **Example**: If the source symbols are A, B, and C, with frequencies:
    - A: 50%
    - B: 30%
    - C: 20%

    The Huffman tree may assign:
    - A: 0
    - B: 10
    - C: 11

    This results in a compressed form of the data where symbols are represented by fewer bits.
- **Advantages**:
    - Efficient for compression when symbol frequencies vary significantly.
    - Guarantees an optimal prefix code for the data.
- **Disadvantages**:
    - Requires knowledge of the symbol frequencies beforehand.

Huffman Coding Process in Detail

Huffman coding is a **lossless data compression** algorithm used to reduce the size of data by assigning shorter codes to more frequent symbols and longer codes to less frequent symbols. It is widely used in **file compression** (like in ZIP, JPEG, etc.) and in **data transmission** to improve efficiency.

The process of **Huffman coding** is based on the principle of **frequency-based encoding**, where symbols with higher frequencies are given shorter binary codes, and those with lower frequencies are given longer binary codes.

Steps in Huffman Coding:

1. **Create a frequency table for each symbol**:
    - The first step is to calculate the frequency of each symbol (e.g., characters in a text) in the data to be compressed.
2. **Build a priority queue (min-heap)**:

- A **priority queue** (often implemented as a **min-heap**) is created to hold all the symbols and their frequencies. The queue is sorted by frequency, with the least frequent symbols at the top.

3. **Construct the Huffman Tree**:
   - The algorithm builds the **Huffman tree** by repeatedly merging the two nodes with the least frequency. Each merged node is assigned a frequency equal to the sum of the frequencies of the two nodes.
   - This process continues until all nodes are merged into one tree. The tree's structure represents the shortest binary codes for the most frequent symbols.
4. **Assign binary codes to each symbol**:
   - Starting from the root of the tree, assign **0** to one branch and **1** to the other at each node. The binary code for each symbol is determined by the path taken from the root to the leaf node that represents that symbol.
   - Symbols at deeper levels of the tree (i.e., less frequent symbols) will have longer binary codes, while symbols near the root will have shorter binary codes.
5. **Generate the compressed bitstream**:
   - Once the binary codes are assigned, the original data is replaced with the corresponding binary code for each symbol, creating the compressed bitstream.

Example of Huffman Coding Process:

Refer to GeekForGeek: https://www.geeksforgeeks.org/huffman-coding-greedy-algo-3/

Summary of the Huffman Coding Process:

1. Calculate the frequency of each symbol in the data.
2. Build a priority queue (min-heap) sorted by frequency.
3. Create the Huffman tree by repeatedly merging the two least frequent nodes.
4. Assign binary codes to each symbol based on the tree.
5. Replace each symbol in the original data with its binary code to create the compressed bitstream.

Huffman coding ensures that more frequent symbols are assigned shorter codes, achieving efficient compression while retaining the ability to perfectly reconstruct the original data.

---

4. Arithmetic Encoding (Introduction)

Arithmetic Encoding is another form of entropy coding that encodes the entire message as a single number in the interval [0, 1). Unlike Huffman coding, which uses individual symbols to build a code, arithmetic encoding encodes the entire sequence of symbols into a fraction.

- **How it Works**:

1. Each symbol in the message is represented by a subrange of the interval [0, 1).
2. The message is encoded by narrowing down the range for each symbol based on the cumulative probability distribution of the symbols.
3. The final range after processing all symbols gives the encoding for the entire message.

- **Example**: For a message "ABAC" with the following probability distribution:
  - A: 0.5
  - B: 0.3
  - C: 0.2

  Arithmetic encoding would map the message "ABAC" to a fractional value between 0 and 1, which would be its encoded representation.

- **Advantages**:
  - Can achieve better compression than Huffman coding for some types of data.
  - Efficient for data with varying symbol probabilities.

- **Disadvantages**:
  - More computationally intensive than Huffman encoding.
  - Requires a floating-point number to represent the encoded value, which can lead to precision issues.

---

5. Run-Length Encoding (RLE)

Run-Length Encoding (RLE) is a simple compression algorithm that works by reducing the size of data sequences where symbols repeat consecutively (runs). It is most effective when the data contains long sequences of repeated symbols.

- **How it Works**:
  1. Identify consecutive occurrences (runs) of the same symbol.
  2. Replace the run with the symbol followed by the count of its occurrences.

- **Example**: For the data sequence: "AAAABBBCCDAA", the RLE encoding would be:
  - A4B3C2D1A2

  This means "A" occurs 4 times, "B" occurs 3 times, and so on.

- **Advantages**:
  - Very efficient for data with large runs of repeated symbols (e.g., simple graphics, text files with lots of spaces).
  - Simple to implement and understand.

- **Disadvantages**:
  - Ineffective for data with little or no repetition.
  - The compressed data might actually be larger if there are no long sequences of repeated symbols.

Summary of Encoding Techniques

Technique	Description	Best for
Huffman Encoding	Variable-length prefix code based on symbol frequencies	Data with varying symbol frequencies
Arithmetic Encoding	Encodes the entire message as a single number in the interval [0,1)	Data with complex symbol distributions
Run-Length Encoding (RLE)	Compresses sequences of repeated symbols into a count and symbol	Data with long runs of repeated symbols

---

Conclusion

- **Source coding** and **entropy coding** are crucial for efficient data compression, and the methods like Huffman encoding, Arithmetic encoding, and Run-Length Encoding provide different strategies depending on the data structure.
- Each method has its strengths and weaknesses, making them suitable for various types of data and applications.

5.4 Lossy Sequential DCT-based Mode

The **Lossy Sequential DCT-based Mode** is commonly used in **image compression**, particularly in the **JPEG (Joint Photographic Experts Group)** standard. It involves transforming an image into the frequency domain using the **Discrete Cosine Transform (DCT)** and then quantizing and compressing the resulting frequency coefficients. The goal is to reduce the amount of data needed to represent an image, but at the cost of some loss of quality.

Key **steps** involved in the **Lossy Sequential DCT-based Mode**:

Steps of Lossy Sequential DCT-based Mode

1. **Image Dividing into Blocks**
    - **Divide the image** into small, non-overlapping **8x8 pixel blocks**. This segmentation helps reduce the computational complexity and isolates local image features for processing.
    - These blocks allow the DCT to work more effectively, as local frequency information within a block is typically more relevant than across larger areas.
2. **Applying Discrete Cosine Transform (DCT)**
    - For each **8x8 block**, apply the **2D Discrete Cosine Transform (DCT)**.
    - The DCT transforms the spatial domain (pixel values) into the frequency domain, representing the image in terms of **low-frequency** (smooth areas) and **high-frequency** (edges and noise) components.
    - The result of the DCT is a set of **frequency coefficients** that represent the image block. These coefficients include both **DC (average) coefficient** and **AC (higher frequency) coefficients**.
3. **Quantization**

- The frequency coefficients obtained from the DCT are then **quantized** to reduce the precision of the coefficients. This step is the primary source of **lossy compression** and results in a loss of image quality.
- **Quantization** involves dividing the DCT coefficients by a **quantization matrix** and rounding them to the nearest integer.
- The **quantization matrix** determines how much each frequency component is compressed, and it typically gives more weight to lower-frequency components (which are more visually important) and less weight to higher-frequency components (which are often less noticeable).

4. **Zigzag Scanning and Encoding**

   - After quantization, the quantized DCT coefficients are **scanned in a zigzag order** to prioritize the most significant coefficients (which typically appear in the top-left corner of the block after DCT).
   - The coefficients are then grouped and **encoded** using **Run-Length Encoding (RLE)** and **Huffman Encoding** to further compress the data.
     - **Run-Length Encoding (RLE)** is used to encode sequences of zeros (which are common in the quantized data).
     - **Huffman Encoding** is used to represent frequently occurring values with shorter code words, optimizing the data for storage or transmission.

5. **Compression and Bitstream Formation**

   - The quantized and encoded data is then compressed and converted into a **bitstream** for storage or transmission.
   - This bitstream contains all the compressed information about the image, which can be decoded and reconstructed at a later time.

6. **Image Reconstruction (Decoding)**

   - During decompression, the bitstream is **decoded**, and the encoded DCT coefficients are restored.
   - The coefficients are **inverse quantized** (approximated to their original values) and then passed through the **Inverse Discrete Cosine Transform (IDCT)** to return to the spatial domain, approximating the original image.
   - **Loss** occurs during quantization, so the reconstructed image may not exactly match the original.

7. **Post-Processing (Optional)**

   - Some post-processing techniques, such as **deblocking filters**, may be applied to reduce visible artifacts like blockiness in the reconstructed image.

Summary of Key Steps

1. **Divide the image** into 8x8 blocks.
2. Apply **DCT** to each block to transform into the frequency domain.
3. **Quantize** the DCT coefficients (lossy step).
4. **Scan** the quantized coefficients in a zigzag order and **encode** them.
5. **Compress** and form the bitstream for storage or transmission.
6. **Decode** the bitstream and apply **IDCT** to reconstruct the image.

7. Optionally, **post-process** the image to improve visual quality.

Advantages of DCT-based Compression

- **Efficient compression**: DCT effectively reduces the amount of data by focusing on important frequency components and discarding less important ones.
- **Visual quality**: By prioritizing low-frequency components, this method helps retain important visual information while allowing some loss.
- **Widely used**: The DCT-based mode is a fundamental part of JPEG and other image compression standards.

Disadvantages

- **Loss of quality**: Quantization introduces loss, which can lead to visible artifacts like blurring and blocking, especially at high compression ratios.
- **Artifacts**: The quantization process can cause blockiness or ringing effects in the image if the compression is too aggressive.

5.5 Expanded Lossy DCT-based Mode

The **Expanded Lossy DCT-based Mode** is an extension of the **Lossy Sequential DCT-based Mode** commonly used in **JPEG compression**. It allows more flexibility in terms of compression, enabling better quality and higher compression ratios. This extended method includes additional techniques and optimizations, resulting in improvements over the standard DCT-based compression approach.

In this mode, a more detailed and efficient process is used to handle image data for compression, including **progressive encoding** and **adaptive quantization**. The steps involved are essentially an elaboration of the basic DCT-based compression process, with enhancements for better performance in terms of quality and compression ratio.

Expanded steps involved in the **Expanded Lossy DCT-based Mode**:

Steps of Expanded Lossy DCT-based Mode

1. **Image Dividing into Blocks**
    - Similar to the standard DCT-based compression, the image is **divided into non-overlapping 8×8 blocks**. This segmentation of the image allows DCT to be applied more effectively on local areas, preserving the spatial coherence of the image.
2. **Applying Discrete Cosine Transform (DCT)**
    - Each **8x8 block** undergoes a **2D Discrete Cosine Transform (DCT)**. This converts the image data from the spatial domain (pixel values) to the frequency domain.
    - The result of DCT produces **frequency coefficients**, which represent the image in terms of both **low-frequency** (smooth areas) and **high-frequency** (edges, textures) components.
3. **Improved Quantization (Adaptive Quantization)**
    - In the **Expanded Lossy DCT-based Mode**, quantization is enhanced through **adaptive quantization**. This technique adjusts the **quantization step**

sizes dynamically based on the characteristics of the image data.
    - **Higher frequency components** (which often correspond to less visually important details) are quantized more aggressively, while **lower frequency components** (which correspond to important structural and perceptible information) are quantized less aggressively.
    - **Quantization matrix adaptation**: The quantization matrix is adjusted dynamically based on the local characteristics of each block, enabling better compression while maintaining more of the perceptible image quality.

4. **Zigzag Scanning and Run-Length Encoding**

    - After quantization, the **quantized coefficients** are **scanned in a zigzag order** to prioritize the most significant values (typically starting from the DC coefficient and moving towards higher frequency AC coefficients).
    - This scanning order allows for better compression, as many coefficients, especially higher frequency ones, will often be zero or near-zero after quantization.
    - The **Run-Length Encoding (RLE)** technique is then applied to compress sequences of repeated values (especially zeros) more efficiently.

5. **Entropy Coding (Huffman Encoding)**

    - The quantized and RLE-compressed data is then passed through **entropy coding** (typically **Huffman Encoding**) to further compress the bitstream.
    - **Huffman Encoding** assigns shorter code words to more frequent values, optimizing the data for storage or transmission by reducing the overall size.

6. **Progressive Encoding (Optional in Expanded Mode)**

    - A major feature of the **Expanded Lossy DCT-based Mode** is the **progressive encoding** capability. In this mode, the image is encoded in multiple passes.
    - In the first pass, a **low-quality version** of the image is encoded and transmitted, followed by **subsequent passes** that progressively refine the image. This allows for faster loading of a rough image preview, with higher quality details appearing as more passes are processed.
    - This feature is particularly useful in applications like **web images** (JPEG2000) or streaming, where users can view a lower quality image quickly and wait for the full image to load progressively.

7. **Compression and Bitstream Formation**

    - After entropy coding, the compressed data is formatted into a **bitstream** for storage or transmission. The bitstream contains all the information needed to reconstruct the image (or its progressively refined versions).
    - The **header** of the bitstream includes metadata such as the image dimensions, quantization matrix, and other parameters that aid in decoding.

8. **Image Reconstruction (Decoding)**

    - During decompression, the bitstream is **decoded**, and the entropy-coded data is recovered.
    - The quantized DCT coefficients are **inverse quantized** (approximated to their original values), and the **Inverse Discrete Cosine Transform (IDCT)**

is applied to convert the data back to the spatial domain.
    - Depending on whether the image was encoded progressively, the **final image** may be progressively refined after each pass.

9. **Post-Processing and Filtering**
    - Once the image is decoded, **post-processing techniques** like **deblocking filters** can be applied to reduce the appearance of visible blockiness, which may occur due to the block-based compression method.
    - This is especially important in images with high compression ratios, where the loss of fine details can lead to visible artifacts.

Summary of Expanded Lossy DCT-based Mode Steps

1. **Divide the image** into 8x8 blocks.
2. Apply **DCT** to each block to transform into the frequency domain.
3. Use **adaptive quantization** to quantize the DCT coefficients, dynamically adjusting based on image content.
4. **Scan** the quantized coefficients in zigzag order and apply **Run-Length Encoding (RLE)**.
5. Apply **Huffman Encoding** for **entropy coding**.
6. **Progressive encoding** (optional) for progressive image refinement.
7. **Compress** and generate the **bitstream**.
8. **Decode** the bitstream, apply **inverse DCT**, and reconstruct the image.
9. Optionally, apply **post-processing** to improve image quality.

Advantages of Expanded Lossy DCT-based Mode

- **Adaptive quantization** allows better handling of different image areas, preserving important details while achieving higher compression.
- **Progressive encoding** enables faster image loading and smoother visual experience, especially for web applications.
- **Better image quality** at higher compression ratios compared to standard DCT-based methods.

Disadvantages

- **Computational complexity** increases due to adaptive quantization, progressive encoding, and post-processing.
- **Loss of quality**: While adaptive quantization improves quality retention, it still results in some visible artifacts, especially at higher compression levels.

Conclusion

The **Expanded Lossy DCT-based Mode** offers improvements over the basic DCT compression method by introducing adaptive quantization, progressive encoding, and post-processing techniques. These enhancements make the compression more efficient and the resulting image quality better at higher compression ratios. This mode is particularly useful in applications where both compression efficiency and visual quality are important, such as web image compression and streaming applications.

5.6 JPEG and MPEG Compression Process

JPEG (Joint Photographic Experts Group) and MPEG (Moving Picture Experts Group) are two widely used compression standards for image and video formats, respectively. While JPEG is used primarily for still images, MPEG is designed for video compression. Both use similar techniques like **Discrete Cosine Transform (DCT)**, but they differ in their specific applications and how they handle multimedia data.

Overview of the **compression processes** for both **JPEG** (for still images) and **MPEG** (for video compression):

JPEG Compression Process

JPEG is a standard for **image compression** that uses **lossy compression** techniques to reduce file sizes while maintaining visual quality. It is typically used for compressing photographic images and is commonly used in digital photography and web images.

Steps in JPEG Compression Process:

1. **Color Space Conversion (Optional)**
    - JPEG typically operates in the **YCbCr** color space instead of the RGB color space. The conversion from **RGB** (Red, Green, Blue) to **YCbCr** (Luminance and Chrominance components) allows better compression, as human vision is more sensitive to luminance (Y) than to chrominance (Cb, Cr).
    - This step is optional but improves compression efficiency.
2. **Image Dividing into 8x8 Blocks**
    - The image is divided into **non-overlapping 8x8 pixel blocks**. The reason for this is that the **Discrete Cosine Transform (DCT)** works more efficiently on small, localized blocks rather than the entire image.
3. **Discrete Cosine Transform (DCT)**
    - The **DCT** is applied to each **8x8 block** to convert the image from the spatial domain to the frequency domain. This transforms the pixel values into **frequency coefficients**.
    - In the DCT domain, the image is represented by **low-frequency components** (which contain most of the image's information) and **high-frequency components** (which represent fine details and noise).
4. **Quantization**
    - The **DCT coefficients** are quantized to reduce the precision of the frequency values. This step introduces loss and is the main source of compression.
    - **Quantization matrices** are used, which apply different quantization levels based on the frequency. Lower frequencies are quantized with less loss (maintaining quality), while higher frequencies are quantized more aggressively (discarding less important details).
    - The result of quantization is a smaller set of coefficients with a loss in image quality but a significant reduction in size.
5. **Zigzag Scanning**

- After quantization, the **quantized coefficients** are **scanned in a zigzag pattern**. This scanning order ensures that the **most important coefficients** (low-frequency components) come first, which aids in more efficient compression.

6. **Entropy Coding**
   - The **zigzag-scanned quantized coefficients** are then encoded using **Huffman encoding** or **Run-Length Encoding (RLE)**.
   - Huffman encoding is a lossless compression technique that assigns shorter codes to frequently occurring values, thereby reducing the file size.
7. **Compression and Bitstream Generation**
   - The result of this process is a **compressed bitstream** that contains the encoded image data. The bitstream can be stored or transmitted.
8. **Image Reconstruction (Decompression)**
   - To decompress the image, the inverse of each step is performed:
     1. **Huffman decoding** is applied to reconstruct the quantized DCT coefficients.
     2. **Inverse zigzag scanning** is performed.
     3. The **quantized DCT coefficients** are **dequantized** (approximating the original values).
     4. The **Inverse DCT (IDCT)** is applied to convert the data back to the spatial domain.
     5. The **RGB conversion** is done if needed (from YCbCr back to RGB).
   - The decompressed image is an approximation of the original image, with some loss of quality due to quantization.

JPEG Compression Example

Let’s assume the image is small and only has a few pixels:

1. **Original Image (RGB):**

```
R=255 G=0 B=0 (Red)
R=0 G=255 B=0 (Green)
R=0 G=0 B=255 (Blue)
```

2. **Convert to YCbCr:**

```
Y=76 Cb=85 Cr=255 (for Red)
Y=150 Cb=43 Cr=21 (for Green)
Y=29 Cb=255 Cr=107 (for Blue)
```

3. **Chroma Subsampling (4:2:0):** The chroma components (Cb and Cr) are reduced in resolution.
4. **Block Splitting (8x8):** The image is divided into 8x8 blocks (if the image is larger, it gets divided accordingly).
5. **DCT and Quantization:** After applying DCT and quantizing the coefficients, you get a matrix of numbers with reduced precision.

6. **Zig-Zag Scanning and Run-Length Encoding:** The quantized values are scanned and encoded.
7. **Huffman Coding:** The final step is to apply Huffman coding to the data to further compress it.

MPEG Compression Process

MPEG is a family of standards used for **video and audio compression**, primarily for streaming and storage of video content. MPEG video compression uses many of the same techniques as JPEG but with additional techniques to handle video data and motion.

Steps in MPEG Compression Process:

1. **Preprocessing**
    - **Color space conversion**: Like JPEG, MPEG typically converts the image from **RGB** to **YCbCr** to take advantage of the human visual system's greater sensitivity to brightness (luminance) over color information (chrominance).
2. **Motion Compensation (For Video)**
    - A key component of video compression in MPEG is **motion compensation**. Video frames are not treated as independent images, but instead, **inter-frame compression** is used by referencing previous or future frames.
    - **Motion estimation** identifies areas of the image that move between frames. Instead of encoding these areas as independent frames, the encoder only stores the **difference** between consecutive frames (motion vectors) and the **static background**.
    - This allows for significant compression, as large sections of the frame can be referenced from neighboring frames, reducing redundancy.
3. **Dividing the Video into Blocks (Macroblocks)**
    - Just like JPEG, the video frame is divided into **16x16 macroblocks** (instead of 8x8 as in JPEG). Each macroblock is processed separately for **spatial compression**.
4. **Discrete Cosine Transform (DCT)**
    - **DCT** is applied to each **macroblock** to transform the image data from the spatial domain to the frequency domain.
    - This allows for better compression by separating important low-frequency information from less significant high-frequency information.
5. **Quantization**
    - Similar to JPEG, the **DCT coefficients** are quantized to reduce precision, introducing some loss but achieving compression.
    - In MPEG, there are **different quantization tables** for different types of macroblocks (e.g., intra-coded or inter-coded blocks), allowing for more flexibility in the compression process.
6. **Encoding (I, P, and B Frames)**

- **Intra-coded (I) frames**: These are key frames that store the full image and are compressed using the standard DCT-based techniques similar to JPEG.
- **Predictive (P) frames**: These frames store the difference (motion vectors and residual data) from a previous frame (I or P frame).
- **Bidirectional (B) frames**: These frames store the difference from both the previous and the next frames, allowing for even more efficient compression by using bidirectional prediction.

7. **Entropy Coding (Huffman Encoding)**
   - The quantized and motion-compensated data are then encoded using **Huffman encoding** to further compress the video stream.
   - Huffman coding reduces the bitstream size by assigning shorter codes to more frequent values.
8. **Bitstream Generation**
   - The final result is a **compressed bitstream** that contains the video data, including I, P, and B frames, motion vectors, and other information. This bitstream can be stored or transmitted for playback.
9. **Decompression (Video Decoding)**
   - During playback or decompression, the process is reversed:
     1. **Huffman decoding** is applied to recover the quantized data.
     2. **Motion compensation** reconstructs the video frames based on the motion vectors and references to I, P, and B frames.
     3. **Inverse DCT (IDCT)** is applied to reconstruct the spatial data from the frequency coefficients.
     4. The reconstructed video frames are combined and displayed on the screen.

Summary of JPEG and MPEG Compression Processes:

- **JPEG**:
  - Used for **still image compression**.
  - Involves **DCT**, **quantization**, **Huffman encoding**, and **inverse DCT**.
  - Primarily **lossy** compression.
- **MPEG**:
  - Used for **video compression**.
  - Utilizes **motion compensation**, **DCT**, **quantization**, and **Huffman encoding**.
  - Compresses both **intra-frame** and **inter-frame** data (with I, P, and B frames).
  - Supports **audio compression** as well.

Example: MPEG Compression Process

Let's consider a 3-frame video clip that needs to be compressed using the MPEG process.

1. **Frame 1 (I-frame)**:
   - This is the **keyframe** that stores a full image of a car in a parking lot.

    - The entire image is encoded (spatial compression using techniques like DCT and quantization).

2. **Frame 2 (P-frame)**:
    - The car in Frame 2 has moved slightly to the right.
    - Instead of storing the whole image, MPEG stores the **difference** (motion vector) between Frame 2 and Frame 1. This is **temporal compression**, where only the changes are encoded.

3. **Frame 3 (B-frame)**:
    - The car moves further in Frame 3.
    - MPEG stores the **difference** between Frame 3 and both Frame 1 and Frame 2, as the car's motion can be predicted from both previous frames. This is **bi-directional prediction** (B-frame), further increasing compression.

Steps in Summary:

1. **I-frame** (full image of the first frame).
2. **P-frame** (stores motion between frames, only the difference).
3. **B-frame** (stores differences from both previous and future frames, using bi-directional prediction).

This way, instead of encoding entire frames, MPEG compresses the video by storing **only the changes** between frames, achieving high compression while maintaining reasonable quality.

Both JPEG and MPEG are powerful compression methods that rely on **transform coding**, **quantization**, and **entropy coding** to reduce file sizes while maintaining acceptable quality, with JPEG focusing on images and MPEG on video and audio.