IMAGE COMPRESSION

Maitri S. Khatwala

Data Science 06

—

November 1, 2024

—

CC-402 Text, Image & Video Analytics

1. Explain the need for image compression in multimedia applications. How does compression impact storage and transmission efficiency?

A. <u>Importance of Image Compression</u>

Image compression is essential in multimedia applications because it significantly reduces file sizes, which saves storage space and lowers costs. It also speeds up the transmission of images over networks, enhancing user experience by reducing loading times. Additionally, compressed images improve the performance of software applications by allowing faster processing and ensuring compatibility across various devices and platforms. Modern compression techniques achieve this while maintaining high image quality, making them indispensable for efficient and effective multimedia management.

Although modern computers can store an increasing number of files, file size is still an issue. We can store more files if our files are smaller. We use various compression algorithms to reduce the amount of space required to represent a file. Lossless and lossy compression algorithms are the two types of compression algorithms. Lossless compression algorithms reduce file size without hampering any information in the file. In contrast, lossy compression algorithms reduce the file size by removing unnecessary or less important information in the file. We need data compression algorithms for various reasons, some of which are mentioned below.

Data compression reduces the amount of space that files take up on a hard drive and the amount of time taken to transfer or download them.
To reduce storage costs, we need compression algorithms that allow us to store large amounts of data at a lower cost.
For example, in a 2:1 compression ratio, a 20 MB file takes up 10 MB of space, allowing us to store large files in less storage space.

---

2. What is redundancy? Explain three types of Redundancy.

A. <u>Redundancy</u>

Redundancy refers to “storing extra information to represent a quantity of information.” So, that is the redundancy of data. Now, apply this concept to digital images. We know that computers store images in pixel values, so sometimes an image has duplicate pixel values, or maybe if we remove some of the pixel values, they don't affect the information of an actual image. Data Redundancy is one of the fundamental components of Data Compression.

Types of Redundancy in the context of Neighbouring Pixels:

1. Coding redundancy: Coding redundancy in image compression refers to the inefficiency where more bits are used than necessary to represent image data. By identifying and eliminating this redundancy, compression algorithms can reduce file sizes without significant loss of quality. For instance, Huffman coding assigns shorter codes to frequently occurring pixel values, optimizing the data representation. This process ensures efficient storage and transmission of images by minimizing unnecessary data.

2. Interpixel redundancy: Interpixel redundancy in image compression refers to the redundancy that arises from the correlation between neighbouring pixels in an image. Since adjacent pixels often have similar values, this redundancy can be exploited to reduce the amount of data needed to represent the image. Compression techniques like run-length encoding (RLE) and predictive coding take advantage of this by encoding the differences between neighbouring pixels rather than the pixel values themselves, leading to more efficient storage and transmission of image data.

3. Psychovisual redundancy: Psychovisual redundancy in image compression leverages the human visual system's limitations to reduce data without noticeable loss of quality. Compression algorithms like JPEG can discard or simplify this information since the human eye is less sensitive to certain visual details, such as high-frequency components or subtle colour variations. By focusing on preserving perceptually important details and eliminating less noticeable ones, psychovisual redundancy helps achieve efficient compression while maintaining visual fidelity.

---

3. Define coding redundancy. Provide examples of how coding redundancy is used to reduce image file sizes.

A. Coding redundancy

Code redundancy relates to how information is expressed through codes representing data, such as the gray levels within an image. When these codes use excessive symbols to represent each gray level, more than what's required, the resultant image is described as having coding redundancy.

Examples:

Predictive coding encodes only the new information in each pixel, which is the difference between the predicted and actual pixel values. Since prediction errors are usually smaller than the original pixel values, the VLC encoder generates shorter codewords.

Bit-plane coding decomposes an image into a series of binary images and compresses each binary image using a binary compression method.

WebP - A versatile image format that supports both lossless and lossy compression. Lossy WebP uses predictive coding to encode images, and lossless WebP uses entropy coding on the transformed image data.

Run-Length Encoding (RLE) is effective for images with large areas of uniform colour. Instead of storing each pixel individually, RLE stores the colour value and the number of consecutive pixels with that colour. For example, a row of 10 white pixels might be stored as “10W” instead of “WWWWWWWWWW”.

Huffman Coding assigns shorter codes to more frequently occurring pixel values and longer codes to less frequent values. By using variable-length codes based on the frequency of occurrence, Huffman coding reduces the overall number of bits needed to represent the image.

Arithmetic Coding like Huffman coding, arithmetic coding represents frequently occurring pixel values with fewer bits. However, it encodes the entire image as a single number, which can be more efficient for certain types of data.

---

4. Discuss inter-pixel redundancy and how it is exploited in image compression algorithms. Provide examples of common methods to reduce inter-pixel redundancy.

A. <u>Inter-pixel Redundancy</u>

Inter-pixel redundancy is the redundancy arising from the correlation between neighbouring pixels in an image. Since adjacent pixels often have similar values, this redundancy can be exploited to reduce the amount of data needed to represent the image. By encoding the differences between neighbouring pixels rather than the pixel values themselves, compression algorithms can achieve significant data reduction.

Exploitation in Image Compression Algorithms

1. <u>Run-Length Encoding (RLE)</u>:
    - How it works: RLE compresses sequences of identical pixels by storing the pixel value and the count of its consecutive occurrences.
    - Example: In a black-and-white image, a row of ten black pixels followed by five white pixels can be encoded as (10, black), and (5, white) instead of storing each pixel individually.

2. <u>Predictive Coding</u>:
    - How it works: Predictive coding uses the values of neighbouring pixels to predict the value of a given pixel. Only the difference (or error) between the predicted and actual value is encoded.
    - Example: Differential Pulse Code Modulation (DPCM) predicts each pixel value based on the previous pixel and encodes the difference, which is usually smaller and requires fewer bits.

3. <u>Transform Coding</u>:
    - How it works: Transform coding, such as the Discrete Cosine Transform (DCT) used in JPEG, converts spatial domain data (pixel values) into frequency domain data. This often results in many coefficients being zero or close to zero, which can be efficiently encoded.
    - Example: In JPEG compression, the image is divided into blocks, and each block is transformed using DCT. The resulting coefficients are quantized and encoded, reducing the overall data size.

By leveraging inter-pixel redundancy, these methods help achieve efficient image compression, making storage and transmission more effective without significantly compromising image quality.

5. Compare and contrast lossy and lossless image compression techniques. Provide examples of when each type of compression is more appropriate.

A.
Lossy and lossless image compression techniques are two fundamental approaches to reducing the file size of images. Each method has its advantages and disadvantages, making them suitable for different applications.

Lossy Compression
Definition: Lossy compression reduces file size by permanently eliminating some data, which can lead to a decrease in image quality. The goal is to minimize file size while maintaining acceptable visual fidelity.

Key Features:

- Data Loss: Some original data is irretrievably discarded, resulting in a loss of quality.
- Higher Compression Ratios: Generally, achieves much smaller file sizes compared to lossless compression.
- Human Perception: Designed to remove information that is less perceptible to human eyes, ensuring that the quality remains acceptable for most uses.

Examples:

- JPEG: Widely used for photographs and web images, JPEG compression can significantly reduce file size while maintaining good visual quality, especially for images with gradual color transitions.
- WebP: Developed by Google, it ours lossy and lossless options but is optimized for web use with smaller file sizes and faster load times.

Appropriate Use Cases:

- Photography: Where slight quality loss is acceptable in exchange for reduced file size.
- Web Images: For faster loading times on websites, where bandwidth is limited.

Lossless Compression

Definition: Lossless compression reduces file size without losing any data, meaning the original image can be perfectly reconstructed from the compressed file.

Key Features:

- No Data Loss: The original image can be fully restored, maintaining perfect quality.
- Lower Compression Ratios: Typically achieves less compression than lossy methods,
- resulting in larger file sizes.
- Preservation of Quality: Essential for applications where image quality is critical.

Examples:

- PNG: Used for images that require transparency and high quality, such as logos and graphics. It compresses images without any loss of quality.
- TIFF: Often used in professional photography and printing, TIFF files can be compressed without losing any data.

Appropriate Use Cases:

- Medical Imaging: Where every detail is crucial, and any loss of data could impact diagnosis.
- Archiving: For preserving original images without loss, such as in art and historical records.

---

6. Explain Compression Ratio with an Example. What other metrics helps in understanding the quality of the compression.

A. Compression Ratio

The Compression Ratio quantifies how much a file or data has been compressed. It is calculated as the ratio of the original file size to the compressed file size, and it tells us how effectively the data has been reduced.

Formula:
Compression Ratio=Original Size/Compressed Size

A higher compression ratio indicates better compression (e.g., 10:1 means the file size was reduced by a factor of 10).

Example:
Suppose we have an original image file that is 10 MB, and after compression, it becomes 2 MB.
Compression Ratio=10 MB/2 MB=5
This means the file has been compressed by a factor of 5, or in other words, it is only 20% of its original size (since 1/5=0.2).

Other Metrics for Compression Quality:
While compression ratio is important, other metrics are also crucial for understanding the quality of the compression, especially for lossy compression where some data may be lost:

1. Compression Efficiency:
   - Bits per Symbol (or Bits per Pixel): This measures the average number of bits used per symbol (or pixel, in the case of images). Lower bits per symbol indicate higher efficiency.
   - Entropy: This is the theoretical limit of how much data can be compressed, based on the probability distribution of symbols. The closer a compression method gets to the entropy, the more efficient it is.

2. Distortion or Error Metrics (for Lossy Compression):
   - Mean Squared Error (MSE): Measures the average of the squared differences between the original and decompressed data. Lower MSE indicates higher quality.
   - Peak Signal-to-Noise Ratio (PSNR): Often used in image and audio compression, PSNR compares the maximum possible signal value of the original data to the noise introduced by compression. A higher PSNR usually means better quality (less distortion).
   - Structural Similarity Index (SSIM): Common in image compression, SSIM assesses the perceived visual quality by measuring changes in structure, luminance, and contrast between the original and compressed image. SSIM values closer to 1 indicate higher quality.

3. Compression Speed:

- This measures how fast the compression and decompression processes are. Compression speed is particularly important for real-time applications or large datasets where processing time needs to be minimized

---

7. Identify the Pros and Cons of the following algorithms
I. Huffman coding
II. Arithmetic coding
III. LZW coding
IV. Transform coding
V. Run length coding

A. <u>Pros and Cons of the following</u>:

I. <u>Huffman Coding</u>:

Pros:

- Efficient Compression: Reduces bit size for frequent characters.
- Lossless: Allows full data reconstruction.
- Simple Decoding: Unique paths in the tree make decoding easy.

Cons:

- Complex Tree Building: Requires sorting and structure creation.
- Less Effective for Small Files: Limited benefit with few or uniform characters.
- Variable-Length Codes: Needs extra storage for code management.

II. <u>Arithmetic coding</u>:

Pros:

- High Compression: Achieves efficient compression by using probabilities instead of fixed codes.
- Adaptable: Works well for files with complex or varied character frequencies.
- Precision: Encodes the entire message as a single unique number, allowing exact reconstruction.

Cons:

- Computationally Intensive: Requires more processing power and precision for encoding and decoding.
- Not Suitable for Fixed Registers: Needs variable precision, which can be challenging on fixed-size registers.
- Requires Known Probabilities: Needs predefined probabilities for each symbol, adding setup complexity.

III. **LZW coding:**

Pros:

- The algorithm is straightforward, simple, and effective.
- LZW does not require any prior knowledge of the input data stream.
- The LZW algorithm has a 60 – 70% compression ratio for some text files.
- The LZW algorithm performs better for files with a lot of repetitive data.
- The LZW algorithm compresses the data in a single pass.

Cons:

- Inefficient for Non-Repetitive Data: LZW performs poorly when data lacks repetition.
- Memory-Intensive: Requires a dictionary to store patterns, which can grow large.
- Slower Compression for Unique Patterns: Encoding takes longer with unique or random data due to lack of pattern reuse.

IV. **Transform coding:**

Pros:

- Efficient Compression: Reduces file size by focusing on essential data.
- Redundancy Removal: Discards unnecessary information for better compression.
- Broad Applications: Commonly used in compressing images, audio, and video.

Cons:

- High Computational Cost: Requires significant processing power and time.

- Potential Quality Loss: Lossy transform coding can degrade data quality.
- Data-Type Dependence: Most effective on media files, less so for other data types.

V. **Run length coding:**

Pros:

- Simple and Efficient: Easy to implement and understand.
- Effective for Repetitive Data: Compresses data well when there are long runs of identical values.
- Low Memory Requirement: Requires minimal memory for encoding and decoding.

Cons:

- Limited to Repetitive Data: Ineffective for data with few or no repeated values.
- Not Ideal for Complex Data: Poor compression performance on images or files with high variability.
- Can Increase File Size: May lead to larger files if applied to non-repetitive data.

---

8. Perform Huffman coding on a given set of pixel values. Show the step-by-step process and calculate the compression ratio achieved.

A. **Huffman coding**

To perform Huffman coding on a given set of pixel values, we need a list of pixel values and their respective frequencies

Let us assume we have the following pixel values and their frequencies:

Pixel Value	Frequency
0	10
1	15
2	12
3	3
4	4
5	13
6	1

Pixel Value	Frequency
7	1

Step 1: Build the Huffman Tree

1. Sort the pixels by frequency:
    - Initial list (sorted): (6,1), (7,1), (3,3), (4,4), (0,10), (2,12), (5,13), (1,15)
2. Combine the two nodes with the lowest frequencies:
    - Combine (6,1) and (7,1) to form a new node with frequency two.
    - Updated list: (3,3), (4,4), (2,12), (0,10), (5,13), (1,15), (combined,2)
3. Repeat the process:
    - Combine (3,3) and (combined,2) → new node with frequency 5.
    - Combine (4,4) and (combined,5) → new node with frequency 9.
    - Continue this process until you have a single tree.

Each time we combine two nodes, we assign a zero to the left branch and a one to the right branch, building the binary code for each pixel value.

Step 2: Assign Codes to Pixel Values

After completing the tree, assign binary codes to each pixel value based on the path taken from the root to the leaf node. Each path direction represents one bit in the binary code.

For example:

- Pixel zero might get code 00.
- Pixel one might get code 101.
- Pixel two might get code one hundred.
- And so on...

Step 3: Calculate the Compression Ratio

1. Original size: Calculate the original number of bits needed to represent the data without compression.
    - If we have 8 unique pixel values, each pixel would typically be represented by 3 bits (since 2^3 = 8).
    - Total bits without compression = Total pixels×3
2. Compressed size: Calculate the number of bits used with Huffman encoding.
    - Multiply the frequency of each pixel by the length of its Huffman code, then sum these values.
3. Compression Ratio:

Compression Ratio=Original Size/Compressed Size

Example Calculation
Let us assume the following hypothetical Huffman codes:

Pixel Value	Frequency	Huffman Code	Code Length	Total Bits (Freq × Code Length)
0	10	00	2	20
1	15	101	3	45
2	12	100	3	36
3	3	1100	4	12
4	4	1101	4	16
5	13	111	3	39
6	1	11100	5	5
7	1	11101	5	5

- Total bits (Compressed) = 20 + 45 + 36 + 12 + 16 + 39 + 5 + 5 = 178 bits
- Total bits (Original) = (10+15+12+3+4+13+1+1) ×3=177 bits

Calculate Compression Ratio
Compression Ratio=Original Size/Compressed Size=177/178≈0.99approx

In this hypothetical example, the compression ratio is close to one, indicating only slight compression due to low redundancy. Actual ratios vary based on data distribution and redundancy.

---

9. Explain the concept of arithmetic coding and how it differs from Huffman coding. Why is arithmetic coding considered more efficient in some cases?

A. Arithmetic coding

In the world of dictionary coding and probability-based encoding, the floating-point weirdness that is arithmetic coding is a refreshing and surprisingly efficient lossless compression algorithm. The algorithm takes the form of two stages, the first stage translates a string into a floating-point range and the second stage translates this into a binary sequence.

Arithmetic coding and Huffman coding are both lossless compression methods, but they work differently and have unique strengths.

<u>Differences between Arithmetic Coding and Huffman Coding:</u>

1. <u>Encoding Method</u>:

- Huffman Coding: Each symbol in the input data is assigned a unique binary code based on its frequency, with more frequent symbols having shorter codes. This leads to a fixed-length prefix-free code for each symbol.
- Arithmetic Coding: Rather than assigning a distinct code to each symbol, arithmetic coding represents the entire message as a single number in the form of a fractional interval [0,1) [0, 1) [0,1). This interval is progressively refined as each symbol is processed.

2. <u>Output Representation</u>:

- Huffman Coding: Produces a unique binary code for each individual symbol, resulting in a sequence of binary codes that together represent the entire message.
- Arithmetic Coding: Encodes the entire message into a single, continuous floating-point number that lies within a defined range. This is often more compact as it allows encoding fractions of bits.

<u>Why Arithmetic Coding is Considered More Efficient in Some Cases:</u>

1. <u>Better Compression with Complex Probabilities</u>:
   Because arithmetic coding does not rely on fixed-length codes, it can compress data with non-uniform symbol distributions more effectively. It is not limited to a power-of-2 probability distribution, so it provides near-optimal compression ratios regardless of the exact symbol probabilities.

2. <u>Efficiency for High Entropy Data</u>:
   For data with many symbols or where symbols have close probabilities, Huffman coding can be inefficient because it cannot always assign a perfectly optimized code to each symbol. Arithmetic coding, however, assigns intervals based on precise probabilities, leading to more efficient compression in high-entropy scenarios.

3. Flexible Code Lengths:
   Since arithmetic coding does not rely on assigning discrete codes to individual symbols, it can compress data at fractional-bit levels. This means it can achieve compression ratios closer to the theoretical limit (entropy) than Huffman coding, especially in cases where the data is highly compressible.

---

10. Provide an example of LZW coding on a simple sequence of image pixel values.

A. An example of LZW (Lempel-Ziv-Welch) coding on a simple sequence of pixel values to understand the compression process.

Step-by-Step Example of LZW Coding
Suppose we have a sequence of pixel values represented as follows:

A B A B C A B A B C A B

This sequence represents a simple pattern of colours in an image.

Step 1: Initialize the Dictionary

LZW starts with a dictionary of all unique symbols (single pixels in this case) in the sequence. Each symbol is mapped to a unique code. For simplicity, let us assign each symbol an integer code, starting from zero.

Symbol	**Code**
A	0
B	1
C	2

Step 2: Encoding the Sequence

- Read the first symbol, "A":
- "A" is already in the dictionary, so we output its code, 0.
- Current sequence: A
- Move to the next symbol, "B":

- "AB" is not in the dictionary, so we add it with the next available code, 3.
- Output the code for "A" (0), then move to the next symbol.
- Add to dictionary: AB → 3
- Now, the current symbol is "B":
- "B" is in the dictionary, so we keep it as the current sequence.
- Current sequence: B
- Move to the next symbol, "A":
- "BA" is not in the dictionary, so we add it with the next code, 4.
- Output the code for "B" (1), then move to the next symbol.
- Add to the dictionary: BA → 4
- Now, the current symbol is "A":
- "A" is in the dictionary, so we keep it as the current sequence.
- Current sequence: A
- Move to the next symbol, "B":
- "AB" is already in the dictionary with code 3, so we keep "AB" as the current sequence.
- Current sequence: AB
- Move to the next symbol, "C":
- "ABC" is not in the dictionary, so we add it with the next code, 5.
- Output the code for "AB" (3), then move to the next symbol.
- Add to the dictionary: ABC → 5
- Now, the current symbol is "C":
- "C" is in the dictionary, so we keep it as the current sequence.
- Current sequence: C
- Move to the next symbol, "A":
- "CA" is not in the dictionary, so we add it with the next code, 6.
- Output the code for "C" (2), then move to the next symbol.
- Add to the dictionary: CA → 6
- Now, the current symbol is "A":
- "A" is in the dictionary, so we keep it as the current sequence.
- Current sequence: A
- Move to the next symbol, "B":
- "AB" is already in the dictionary with code 3, so we keep "AB" as the current sequence.
- Current sequence: AB
- Move to the next symbol, "C":

- "ABC" is already in the dictionary with code 5, so we keep "ABC" as the current sequence.
- Current sequence: ABC
- Move to the last symbol, "A":
- "ABCA" is not in the dictionary, so we add it with the next code, 7.
- Output the code for "ABC" (5).
- Add to dictionary: ABCA → 7
- End of Sequence:
- Output the code for the remaining symbol, "A" (0).

Final Output
The LZW-encoded output for this sequence is:

0 1 3 2 3 5 0

Final Dictionary
After encoding, the dictionary looks like this:

Symbol	Code
A	0
B	1
C	2
AB	3
BA	4
ABC	5
CA	6

11. What is transform coding? Explain how it helps in compressing image data by reducing redundancies in the frequency domain.

A. Transform coding

Transform coding is a type of compression technique that converts data from its original domain (usually spatial or time domain) to a different domain, typically the frequency domain. This approach is widely used in image and video compression formats, like JPEG, because it allows for more efficient representation of data, focusing on reducing redundancies and irrelevant information, especially in images.

How Transform Coding Works in Image Compression:
In images, data is represented by pixel intensities in the spatial domain, which can contain a lot of spatial redundancy—repeating patterns or similar pixel values across neighbouring pixels. Transform coding seeks to represent this data in a way that clusters related information together, making it easier to reduce irrelevant data.

Here's how it works step-by-step:

Divide the Image into Blocks:

- The image is divided into small blocks, typically 8×8 pixels in JPEG compression, to apply the transform coding on each block independently. This localized approach makes it easier to remove redundancies within each block.

Apply a Transform to Each Block:

- A mathematical transform, like the Discrete Cosine Transform (DCT), is applied to each block. The DCT converts the pixel intensities into frequency coefficients.
- In the frequency domain, the image data is separated into different frequency components. The low-frequency components represent the overall structure or smooth parts of the image, while the high-frequency components represent the fine details and sharp edges.

Reduce Redundancy in the Frequency Domain:

- After the transformation, most of the important visual information is concentrated in the low-frequency components. High-frequency components, which usually correspond to minor details or noise, are often less perceptible to the human eye.
- By removing or reducing these high-frequency coefficients (often by quantizing them to zero), we can compress the image with minimal impact on perceived quality. This process of zeroing out less important coefficients is known as quantization.

12. Discuss the significance of sub-image size selection and blocking in image compression. How do these factors impact compression efficiency and image quality?

A. Sub-image size selection and blocking are crucial aspects of image compression, significantly influencing both compression efficiency and image quality. Here is a breakdown of their significance:

<u>Sub-Image Size Selection</u>

1. <u>Compression Efficiency:</u>
   - Smaller Sub-Images: Using smaller sub-images (or blocks) can lead to higher compression efficiency because they allow for more precise modelling of local image features. This can reduce redundancy and improve the compression ratio.
   - Larger Sub-Images: Larger sub-images may not capture fine details as effectively, potentially leading to less efficient compression. However, they can reduce the overhead of storing block boundaries, which might be beneficial in some contexts.

2. <u>Image Quality:</u>
   - Smaller Sub-Images: These can preserve more detail and texture, leading to higher image quality, especially in areas with high detail or sharp edges.
   - Larger Sub-Images: While they might introduce some blurring or loss of detail, they can be more effective in compressing smooth or uniform areas of an image.

<u>Blocking</u>

1. <u>Compression Efficiency:</u>
   - Block-Based Compression: Techniques like JPEG use fixed-size blocks (e.g., 8x8 pixels). This approach simplifies the compression process and can be highly efficient for certain types of images.
   - Adaptive Blocking: Some advanced methods use variable block sizes, adapting to the image content. This can improve efficiency by using larger blocks for smooth areas and smaller blocks for detailed regions.

2. <u>Image Quality:</u>

- Blocking Artifacts: Fixed-size blocks can introduce visible artifacts, especially at lower bit rates. These artifacts appear as discontinuities or “blockiness” between adjacent blocks.
- Adaptive Blocking: By adjusting block sizes based on image content, adaptive methods can reduce blocking artifacts and improve overall image quality.

Balancing Act

The choice of sub-image size and blocking strategy involves a trade-off between compression efficiency and image quality. Smaller, adaptive blocks tend to offer better quality but may require more computational resources and storage for block information. Larger, fixed-size blocks can be more efficient but might compromise on quality, especially in detailed regions.

---

13. Explain the process of implementing Discrete Cosine Transform (DCT) using Fast Fourier Transform (FFT). Why is DCT preferred in image compression?

A. <u>Implementing DCT Using FFT</u>

The Discrete Cosine Transform (DCT) can be efficiently implemented using the Fast Fourier Transform (FFT) due to their mathematical similarities.

Here is a simplified outline of the process:

1. <u>Symmetry Exploitation:</u>

   The DCT can be derived from the Discrete Fourier Transform (DFT) by exploiting the symmetry properties of the cosine function. Specifically, the DCT can be seen as the real part of a DFT of a symmetrically extended sequence.

2. <u>Sequence Extension:</u>

   Extend the input sequence to twice its length by mirroring it. For an input sequence (x[n]) of length (N), create a new sequence (y[n]) of length (2N)

3. <u>Apply FFT:</u>

   Compute the FFT of the extended sequence (y[n]). This step leverages the efficiency of the FFT algorithm, which has a computational complexity of (O (N \log N)).

4. Extract Real Part:
   The DCT coefficients are obtained by taking the real part of the FFT results. Specifically, the DCT of the original sequence (x[n]) corresponds to the real part of the first (N) elements of the FFT of (y[n]).

Why DCT is Preferred in Image Compression

1. Energy Compaction:
   The DCT has strong energy compaction properties, meaning it can represent most of the signal's energy in a few coefficients. This is particularly useful in image compression, where most of the image information can be captured with fewer coefficients, leading to efficient compression.

2. Reduction of Correlation:
   DCT helps in reducing the correlation between pixels. In an image, neighbouring pixels are often highly correlated. The DCT transforms these correlations into a form where they can be more easily quantized and compressed.

3. Simplicity and Efficiency:
   The DCT is computationally efficient and can be implemented using fast algorithms like the FFT. This makes it suitable for real-time applications and devices with limited processing power.

4. Compatibility with Human Vision:
   The DCT aligns well with the characteristics of human vision. It tends to concentrate the image information in the lower frequencies, which are more perceptually significant to human eyes. This allows for higher compression ratios without significant loss of perceived image quality.

---

14. Describe how run-length coding is used in image compression, particularly for images with large areas of uniform colour. Provide an example to illustrate your explanation.

A. Run-length coding (RLC) is a simple and effective compression technique used to reduce the size of images, especially those with large areas of uniform colour. Here is how it works and an example to illustrate:

How Run-Length Coding Works:

1. Identify Runs:
   RLC identifies sequences of consecutive pixels (or runs) that have the same colour or intensity value. Instead of storing each pixel individually, RLC stores the value of the pixel and the length of the run.

2. Encode Runs:
   Each run is encoded as a pair: the pixel value and the run length. This reduces the amount of data needed to represent the image, especially when there are long runs of the same colour.

3. Compression Efficiency:
   The efficiency of RLC depends on the image content. It works best for images with large uniform areas, such as simple graphics, icons, or scanned documents with large white spaces.

Example

Consider a simple grayscale image row with the following pixel values:
[255, 255, 255, 255, 0, 0, 0, 255, 255, 255, 0, 0]

Using run-length coding, this row can be compressed as:
[(255, 4), (0, 3), (255, 3), (0, 2)]

Here is the step-by-step breakdown:
- The first run consists of four pixels with the value 255.
- The second run consists of three pixels with the value 0.
- The third run consists of three pixels with the value 255.
- The fourth run consists of two pixels with the value 0.