NOAKHALI SCIENCE & TECHNOLOGY UNIVERSITY

Noakhali-3814

Lab Report

Course Code: ICE-4206

Course Title: Multimedia Communication Lab

Submitted By	Submitted To
Mohammad Ajij ID: MUH2011023M Session: 2019-20 Department of ICE, NSTU	Mahbubul Alam Naeem Assistant Professor Department of ICE, NSTU

Date of Submission: 26th May, 2025.

<u>**Experiment No: 01**</u>

Experiment Name: A program to implement different functions on an image.

Theoretical Background:

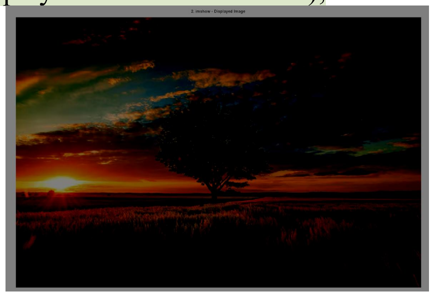
Multimedia communication involves the transmission and reception of multimedia data, which includes images. Image processing is a crucial aspect of multimedia communication, enabling various operations such as conversion between color spaces, clipping, cropping, and copy-pasting regions of interest (ROIs) within images.

1. **imread** - Reads an image file into memory as a numerical array
2. **imshow** - Displays an image in a graphical window
3. **imresize** - Changes image dimensions (scales up/down)
4. **imcrop** - Extracts a rectangular region from an image
5. **imwrite** - Saves an image to disk in specified format
6. **iminfo** - Shows image metadata (size, format, bit depth)
7. **im2gray/rgb2gray** - Converts color image to grayscale
8. **imhist** - Calculates pixel intensity distribution
9. **im2frame** - Converts image to animation/video frame
10. **edge** - Detects boundaries/contours in images
11. **hough** - Identifies geometric shapes (lines, circles)
12. **impixel** - Gets color values at specific coordinates
13. **bwconncomp** - Finds connected pixel groups in binary images
14. **immsc** - Applies morphological operations (erosion/dilation)
15. **dct2** - 2D Discrete Cosine Transform for frequency analysis
16. **idct2** - Inverse DCT for image reconstruction
17. **imfilter** - Applies convolution filters (blur, sharpen)
18. **imgaussfilt** - Gaussian smoothing filter
19. **gabor** - Texture analysis filter
20. **imgaborfilt** - Applies multiple Gabor filters
21. **montage** - Combines multiple images in a grid
22. **imsave** - Python equivalent of imwrite
23. **imrotate** - Rotates image by specified angle
24. **imgerase** - Removes/masks selected image regions

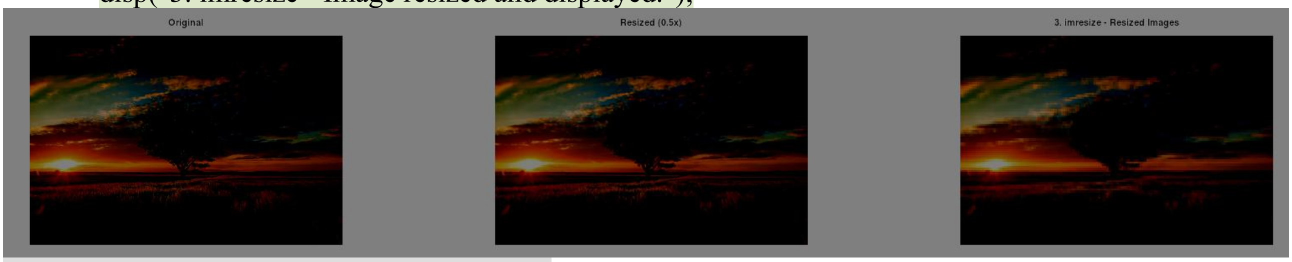
Code Implementation:

```
% --- Initial Setup ---
    pkg load image;
    image_filename = "/home/k452b/octave/input.png";

    % Check if the image file exists
    if (!exist(image_filename, "file"))
      printf("Error: Image file '%s' not found. Please provide a valid image.\n", image_filename);
      printf("You can try Octave's built-in 'autumn.tif' for some examples.\n");
      if (exist("autumn.tif", "file")) % Octave usually ships with this
        image_filename = "autumn.tif";
        printf("Using 'autumn.tif' as a fallback.\n");
      else
        error("No suitable test image found. Exiting.");
      endif
    endif

    printf("--- Running Image Processing Examples ---\n\n");

% --- 1. imread - Read image from file ---
    img = imread(image_filename);
```

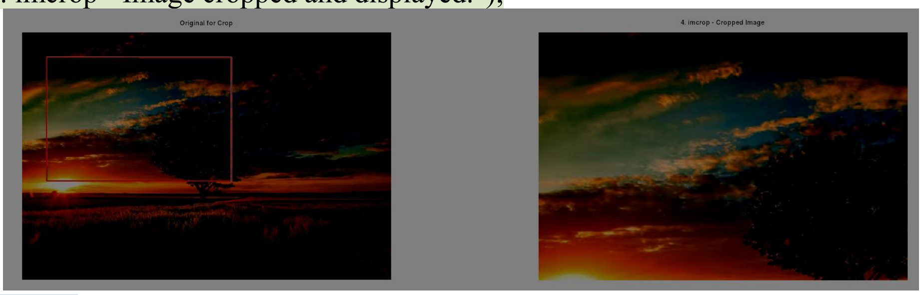
```
        disp("1. imread - Image loaded into variable 'img'");

% --- 2. imshow - Display image ---
        figure; % Open a new figure window
        imshow(img);
        title("2. imshow - Displayed Image");
        disp("2. imshow - Image displayed in a new window.");
```



```
% --- 3. imresize - Change image dimensions ---
        resized_img_half = imresize(img, 0.5); % Scale to 50%
        resized_img_specific = imresize(img, [100, 150]); % Resize to 100 rows, 150 columns

        figure;
        subplot(1,3,1); imshow(img); title("Original");
        subplot(1,3,2); imshow(resized_img_half); title("Resized (0.5x)");
        subplot(1,3,3); imshow(resized_img_specific); title("Resized (100x150)");
        title("3. imresize - Resized Images");
        disp("3. imresize - Image resized and displayed.");
```




```
% --- 4. imcrop - Extract rectangular region ---
        rect = [50, 50, size(img,2)/2, size(img,1)/2]; % Crop a quadrant
        cropped_img = imcrop(img, rect);

        figure;
        subplot(1,2,1); imshow(img); title("Original for Crop");
        rectangle('Position', rect, 'EdgeColor', 'r', 'LineWidth', 2);
        subplot(1,2,2); imshow(cropped_img); title("Cropped Image");
        title("4. imcrop - Cropped Image");
        disp("4. imcrop - Image cropped and displayed.");
```



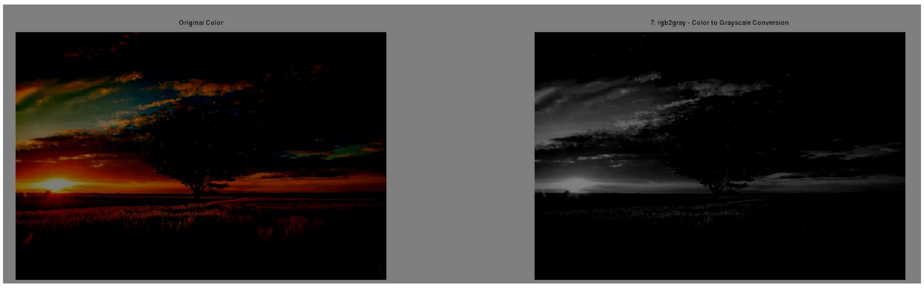

```
% --- 5. imwrite - Save image to disk ---
        imwrite(cropped_img, "cropped_input.png");
        imwrite(img, "input_copy.jpg", "Quality", 90); % For JPEG, can specify quality
        disp("5. imwrite - 'cropped_input.png' and 'input_copy.jpg' saved to disk.");

% --- 6. imfinfo - Show image metadata ---
```

```
info = imfinfo(image_filename);
disp("6. imfinfo - Image File Information:");
disp(info);
printf("   Filename: %s, Format: %s, Width: %d, Height: %d, BitDepth: %d\n", ...
       info.Filename, info.Format, info.Width, info.Height, info.BitDepth);
```

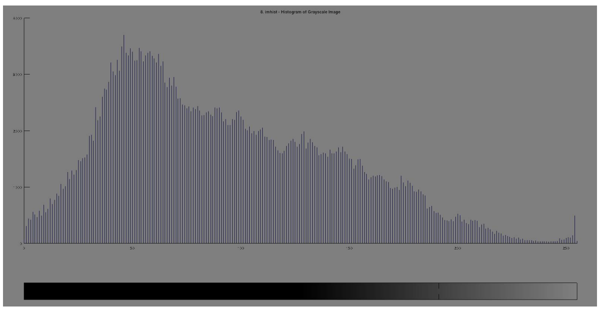
Output:

```
FileModDate = 23-May-2025 10:59:40
FileSize = 766916
Format = PNG
Width = 740
Height = 494
BitDepth = 8
ColorType = truecolor
```

```
% --- 7. rgb2gray - Convert color image to grayscale ---
gray_img = []; % Initialize
if (ndims(img) == 3) % Check if it's a color image (3 dimensions: HxWxChannels)
    gray_img = rgb2gray(img);
    figure;
    subplot(1,2,1); imshow(img); title("Original Color");
    subplot(1,2,2); imshow(gray_img); title("Grayscale");
    title("7. rgb2gray - Color to Grayscale Conversion");
    disp("7. rgb2gray - Image converted to grayscale and displayed.");
else
    gray_img = img; % Already grayscale or binary
    disp("7. rgb2gray - Image is already grayscale or binary. Using as is.");
    figure; imshow(gray_img); title("Original Grayscale/Binary");
end
```

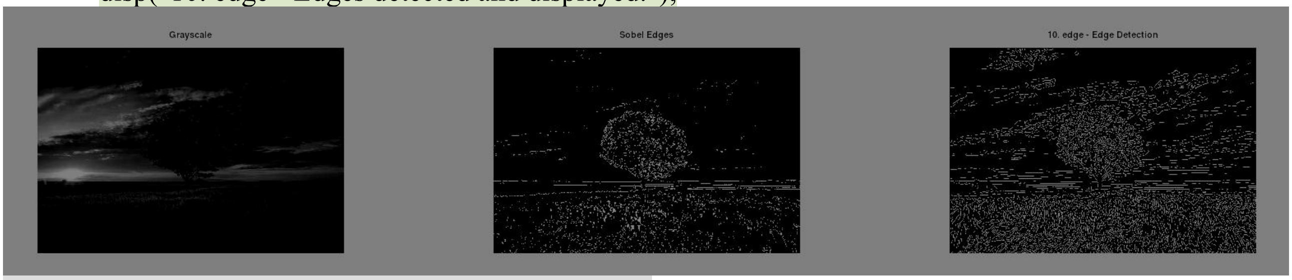



```
% --- 8. imhist - Calculate pixel intensity distribution ---
figure;
imhist(gray_img);
title("8. imhist - Histogram of Grayscale Image");
disp("8. imhist - Grayscale image histogram displayed.");
```

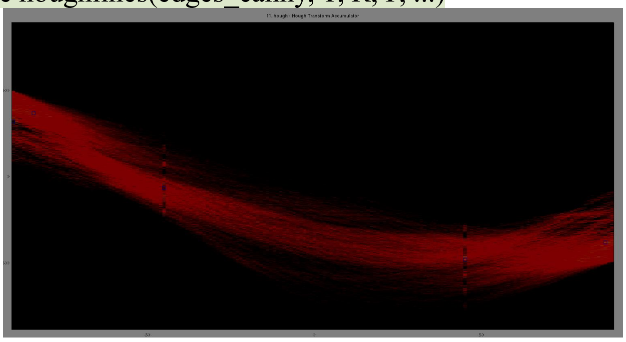


```
% --- 9. im2frame - Convert image to animation/video frame ---
frame_struct = im2frame(img); % F.cdata contains image, F.colormap if indexed
```

```
        disp("9. im2frame - Image converted to a frame structure 'frame_struct'.");

% --- 10. edge - Detect boundaries/contours in images ---
        edges_sobel = edge(gray_img, "sobel");
        edges_canny = edge(gray_img, "canny");

        figure;
        subplot(1,3,1); imshow(gray_img); title("Grayscale");
        subplot(1,3,2); imshow(edges_sobel); title("Sobel Edges");
        subplot(1,3,3); imshow(edges_canny); title("Canny Edges");
        title("10. edge - Edge Detection");
        disp("10. edge - Edges detected and displayed.");
```




```
% --- 11. hough - Identifies geometric shapes (lines) ---
        % (Using edges_canny from step 10, which should be a binary image)
        [H, T, R] = hough(edges_canny); % H: Hough transform, T: theta, R: rho

        figure;
        imshow(H, [], "XData", T, "YData", R, "InitialMagnification", "fit");
        xlabel('\theta (degrees)'); ylabel('\rho');
        axis on; axis normal; hold on;
        colormap(gca, hot);
        title("11. hough - Hough Transform Accumulator");

        % Find peaks in Hough transform
        P = houghpeaks(H, 5, "threshold", ceil(0.3*max(H(:))));
        plot(T(P(:,2)), R(P(:,1)), "s", "color", "blue");
        hold off;
        disp("11. hough - Hough transform calculated and peaks identified.");
        % To draw lines: use houghlines(edges_canny, T, R, P, ...)
```



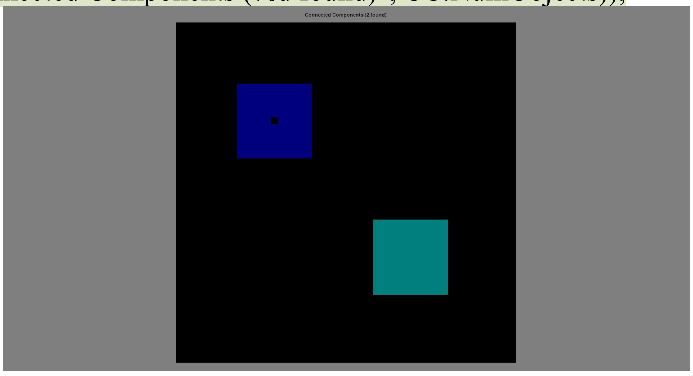
```
% --- 12. impixel - Gets color values at specific coordinates ---
        row_coord = 100; col_coord = 150;
        pixel_value_at_coord = impixel(img, col_coord, row_coord);
        printf("12. impixel - Pixel value at (row %d, col %d): ", row_coord, col_coord);
        disp(pixel_value_at_coord); % For color: [R G B], for gray: [Intensity]
```

Output:

```
        impixel - Pixel value at (row 100, col 150):    178   178 178
```

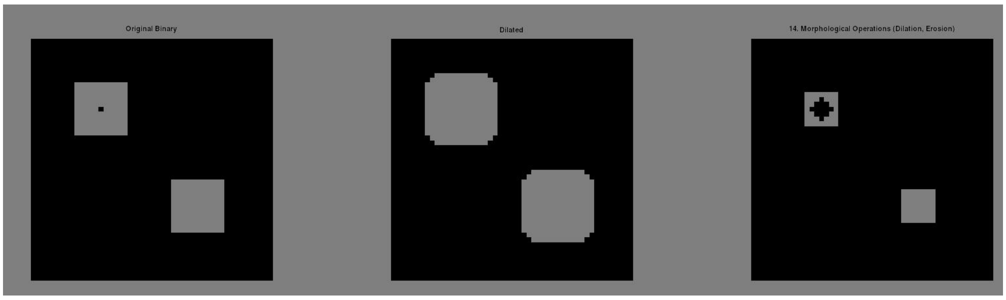
```
% --- 13. bwconncomp - Finds connected pixel groups in binary images ---
        binary_test_img = zeros(50,50);
```

```
binary_test_img(10:20, 10:20) = 1;
binary_test_img(30:40, 30:40) = 1;
binary_test_img(15,15) = 0; % make a hole in the first square
CC = bwconncomp(binary_test_img);
disp("13. bwconncomp - Connected Components Information:");
disp(CC);
printf("   Number of connected objects found: %d\n", CC.NumObjects);
labeled_img = labelmatrix(CC);
figure; imshow(label2rgb(labeled_img, @jet, 'k', 'shuffle'));
title(sprintf("Connected Components (%d found)", CC.NumObjects));
```

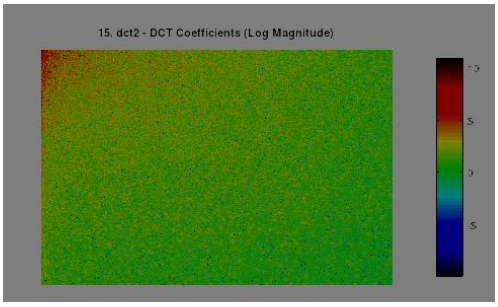


```
% --- 14. Morphological operations (imdilate, imerode) ---
    se = strel("disk", 2, 0); % Structuring element: disk of radius 2, N=0 approximation
    dilated_img = imdilate(binary_test_img, se);
    eroded_img = imerode(binary_test_img, se);
    figure;
    subplot(1,3,1); imshow(binary_test_img); title("Original Binary");
    subplot(1,3,2); imshow(dilated_img); title("Dilated");
    subplot(1,3,3); imshow(eroded_img); title("Eroded");
    title("14. Morphological Operations (Dilation, Erosion)");
    disp("14. Morphological operations (imdilate, imerode) applied and shown.");
```

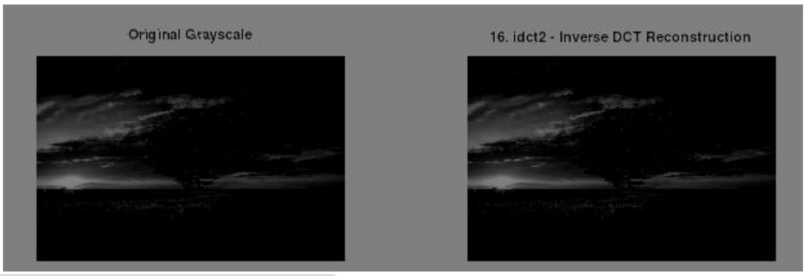
Output:




```
% --- 15. dct2 - 2D Discrete Cosine Transform ---
    disp("--- 15. dct2 (using workaround if needed) ---");
    if (exist('dct2', 'file'))
        dct_coeffs = dct2(double(gray_img)); % Input must be double precision
        disp("Using built-in dct2.");
    else
        disp("Built-in dct2 not found. Using my_dct2 workaround.");
        dct_coeffs = my_dct2(double(gray_img));
    endif
    figure;
    imshow(log(abs(dct_coeffs)), []); % Display log magnitude for visualization
    colormap(gca, jet); colorbar;
    title("15. dct2 - DCT Coefficients (Log Magnitude)");
    disp("dct2 - 2D DCT applied and coefficients displayed.");
    disp(" ");
```




```
% --- 16. idct2 - Inverse DCT for image reconstruction ---
    disp("--- 16. idct2 (using workaround if needed) ---");
    % (Using dct_coeffs from step 15)
    if (exist('idct2', 'file'))
      reconstructed_img_dct = idct2(dct_coeffs);
      disp("Using built-in idct2.");
    else
      disp("Built-in idct2 not found. Using my_idct2 workaround.");
      reconstructed_img_dct = my_idct2(dct_coeffs);
    endif
    figure;
    subplot(1,2,1); imshow(gray_img); title("Original Grayscale");
    subplot(1,2,2); imshow(uint8(reconstructed_img_dct)); title("Reconstructed via IDCT");
    title("16. idct2 - Inverse DCT Reconstruction");
    disp("idct2 - Image reconstructed from DCT coefficients.");
    disp(" ");
```

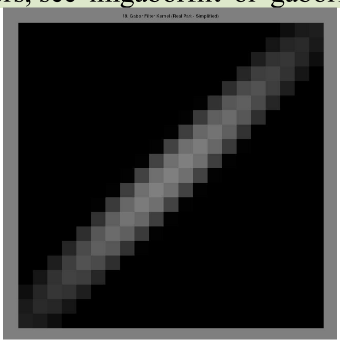



```
% --- 17. imfilter - Applies convolution filters ---
    h_avg = fspecial("average", [5 5]); % Averaging filter kernel
    blurred_img_filter = imfilter(gray_img, h_avg, "replicate"); % "replicate" handles border
    h_laplacian = fspecial("laplacian", 0.2);
    sharpened_img_filter = gray_img - imfilter(gray_img, h_laplacian, "replicate");
    figure;
    subplot(1,3,1); imshow(gray_img); title("Original Grayscale");
    subplot(1,3,2); imshow(blurred_img_filter); title("Blurred (imfilter)");
    subplot(1,3,3); imshow(sharpened_img_filter); title("Sharpened (imfilter)");
    title("17. imfilter - Linear Spatial Filtering");
    disp("17. imfilter - Averaging and sharpening filters applied.");
```

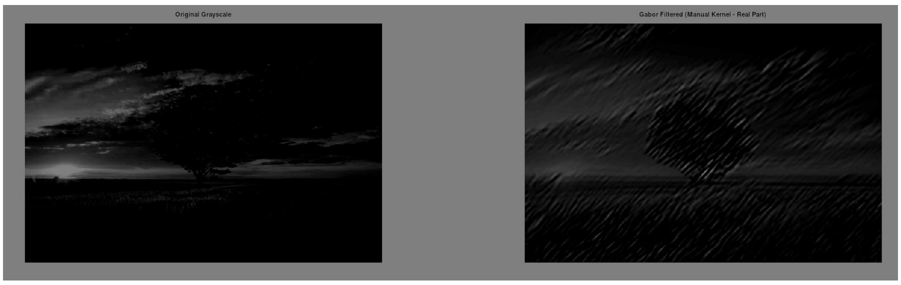

```
% --- 18. imgaussfilt - Gaussian smoothing filter ---
    sigma_gauss = 2; % Standard deviation of Gaussian
    gaussian_blurred_img = imgaussfilt(gray_img, sigma_gauss);
    figure;
    subplot(1,2,1); imshow(gray_img); title("Original Grayscale");
```

```
subplot(1,2,2); imshow(gaussian_blurred_img); title(sprintf("Gaussian Blurred (sigma=%.1f)", sigma_gauss));
title("18. imgaussfilt - Gaussian Smoothing");
disp("18. imgaussfilt - Gaussian filter applied.");
```


```
% --- 19. gabor - Texture analysis filter (kernel creation) ---
lambda  = 10; theta_rad = pi/4; psi = 0; gamma_aspect = 0.5; sigma_env = 5; sz = 21;
gb_kernel_real = zeros(sz, sz);
for x = -floor(sz/2):floor(sz/2)
    for y = -floor(sz/2):floor(sz/2)
        x_theta = x * cos(theta_rad) + y * sin(theta_rad);
        y_theta = -x * sin(theta_rad) + y * cos(theta_rad);
        idx_x = x + floor(sz/2) + 1; idx_y = y + floor(sz/2) + 1;
        gb_kernel_real(idx_y, idx_x) = exp(-(x_theta^2 + gamma_aspect^2 * y_theta^2) / (2 * sigma_env^2)) * cos(2 * pi * x_theta / lambda + psi);
    end
end
figure; imshow(gb_kernel_real, []); title("19. Gabor Filter Kernel (Real Part - Simplified)");
disp("19. gabor - Simplified Gabor kernel created and displayed.");
disp("   For applying Gabor filters, see 'imgaborfilt' or 'gaborfilter'.");
```



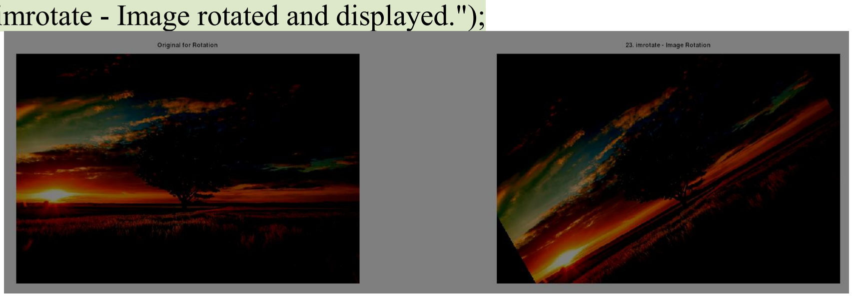
```
% --- 20. imgaborfilt / gaborfilter - Applies Gabor filters ---
if (exist('gaborfilter', 'file'))
    sigma_g = 4; freq_g = 1/8; theta_g_rad = pi/4; % Example parameters
    [gabor_real_comp, gabor_imag_comp] = gaborfilter(double(gray_img), sigma_g, freq_g, theta_g_rad);
    gabor_mag = sqrt(gabor_real_comp.^2 + gabor_imag_comp.^2);
    figure; imshow(gabor_mag, []); title("20. Gabor Filter Magnitude (single filter via gaborfilter)");
    disp("20. gaborfilter - Single Gabor filter applied, magnitude shown.");
else
    disp("20. gaborfilter - 'gaborfilter' function not found in image package. Skipping.");
end
```




```
% --- 21. montage -
```

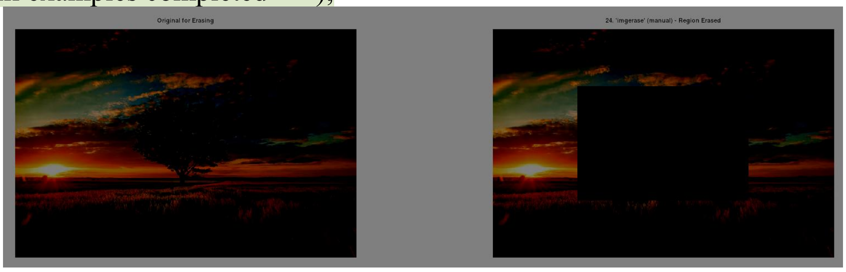
```
Combines multiple images in a grid ---
    if ndims(gray_img) == 2, gray_img_rgb = cat(3, gray_img, gray_img, gray_img); else gray_img_rgb = gray_img; end
    if ndims(edges_canny) == 2, edges_canny_rgb = cat(3, uint8(edges_canny)*255, uint8(edges_canny)*255, uint8(edges_canny)*255); else edges_canny_rgb = edges_canny; end
    img_s = imresize(img, [128 128]);
    gray_s = imresize(gray_img_rgb, [128 128]);
    edges_s = imresize(edges_canny_rgb, [128 128]);
    montage_array = cat(4, img_s, gray_s, edges_s); % Create 4D array HxWxDxN
    figure;
    montage(montage_array);
    title("21. montage - Montage of Images");
    disp("21. montage - Multiple images displayed as a montage.");
```


```
% --- 22. imsave (Python equivalent of imwrite) ---
    imwrite(img, "saved_with_imwrite_again.tif");

% --- 23. imrotate - Rotates image by specified angle ---
    angle_rot = 30; % degrees
    rotated_img = imrotate(img, angle_rot, "bicubic", "crop"); % "crop" to keep original size
    figure;
    subplot(1,2,1); imshow(img); title("Original for Rotation");
    subplot(1,2,2); imshow(rotated_img); title(sprintf("Rotated %d deg (cropped)", angle_rot));
    title("23. imrotate - Image Rotation");
    disp("23. imrotate - Image rotated and displayed.");
```




```
% --- 24. imgerase - Removes/masks selected image regions ---
    disp("24. imgerase - 'imgerase' is not a standard Octave/MATLAB function.");
    disp("   Region erasure is done by direct pixel manipulation.");
    img_to_erase = img; % Make a copy
    erase_rect_roi = [size(img,2)/4, size(img,1)/4, size(img,2)/2, size(img,1)/2]; % Approx center
    r_start = round(erase_rect_roi(2));
    r_end = round(erase_rect_roi(2) + erase_rect_roi(4) - 1);
    c_start = round(erase_rect_roi(1));
    c_end = round(erase_rect_roi(1) + erase_rect_roi(3) - 1);
```

```
[rows, cols, channels] = size(img_to_erase);
r_start = max(1, r_start); r_end = min(rows, r_end);
c_start = max(1, c_start); c_end = min(cols, c_end);
if channels == 3 % Color image
    img_to_erase(r_start:r_end, c_start:c_end, :) = 0;
else % Grayscale image
    img_to_erase(r_start:r_end, c_start:c_end) = 0;
end
figure;
subplot(1,2,1); imshow(img); title("Original for Erasing");
subplot(1,2,2); imshow(img_to_erase); title("With Erased Region");
title("24. 'imgerase' (manual) - Region Erased");
disp("   Region erased by setting pixels to 0 and displayed.");
disp("--- All examples completed ---");
```



Discussion:

The process of running this Octave image processing script highlighted several common aspects of working with Octave and its Forge packages. Initially, a file not found error for imread underscored the importance of correct file paths or ensuring the image is in Octave's working directory. Subsequently, errors like 'strel' N for disk shape not yet implemented and 'dct2' undefined demonstrated that even with a package loaded (like image), specific functions or their full capabilities might not be present in all package versions or may have dependencies that need to be resolved first, as seen with the signal package requiring the control package. This often necessitates updating packages (pkg update, pkg install -forge <packagename>) as a first step. When updates don't suffice or a function like gaborfilter remains elusive, workarounds become essential. For instance, dct2 was manually implemented using 1D dct calls, and Gabor filtering was approached by convolving a manually constructed kernel using imfilter. These experiences emphasize that while Octave offers a powerful open-source environment, users might occasionally need to engage in deeper package management, dependency resolution, or even custom function implementation to achieve the desired functionality, reflecting the community-driven and evolving nature of its extensive package ecosystem.

Experiment No: 02

Experiment Name: A program to implement different functions on an audio.

Theoretical Background:

A program designed to implement different functions on an audio file allows users to perform various audio processing tasks such as playing, recording, trimming, converting formats, changing pitch or speed, adding effects, and analyzing sound features. These functions are essential in fields like music production, speech recognition, and multimedia applications. By manipulating audio signals through digital processing techniques, the program enhances audio quality, extracts useful information, or prepares the audio for further use or analysis.

Function Name: MP3 (MPEG-1 Audio Layer III)

Code:

```
clear all; close all; clc;
[audio, Fs] = audioread('sample.wav');
audio = audio(:,1); % Mono channel
N = length(audio);
```

```
blockSize = 576; % MP3 uses 576-sample frames for MDCT
threshold = 0.15; % Higher threshold for psychoacoustic-like data removal
compressedAudio = zeros(size(audio));
numBlocks = floor(N / blockSize);
for i = 1:numBlocks
    startIdx = (i-1)*blockSize + 1;
    endIdx = i*blockSize;
    block = audio(startIdx:endIdx);
    dctBlock = dct(block);
    dctBlock(abs(dctBlock) < threshold * max(abs(dctBlock))) = 0;
    reconBlock = idct(dctBlock, blockSize);
    compressedAudio(startIdx:endIdx) = reconBlock;
end
if endIdx < N
    lastBlock = audio(endIdx+1:end);
    lastBlock = [lastBlock; zeros(blockSize-length(lastBlock),1)];
    dctBlock = dct(lastBlock);
    dctBlock(abs(dctBlock) < threshold * max(abs(dctBlock))) = 0;
    reconBlock = idct(dctBlock, blockSize);
    compressedAudio(endIdx+1:end) = reconBlock(1:N-endIdx);
end
audiowrite('mp3_compressed.wav', compressedAudio, Fs);
sound(compressedAudio, Fs);
t = (0:N-1)/Fs;
figure;
subplot(2,1,1); plot(t, audio);
title('Original Audio (MP3 Simulation)'); xlabel('Time (s)'); ylabel('Amplitude');
subplot(2,1,2); plot(t, compressedAudio);
title('MP3-like Compressed Audio'); xlabel('Time (s)'); ylabel('Amplitude');
originalSize = N * 8;
nonZeroCoeff = sum(abs(dctBlock) > 0);
compressionRatio = originalSize / (nonZeroCoeff * log2(blockSize));
disp(['MP3-like Compression Ratio: ', num2str(compressionRatio)]);
```

Output (after compression):

Program Explanation:

This MATLAB code simulates a basic MP3-like audio compression technique using the Modified Discrete Cosine Transform (MDCT) concept. It divides the input audio into fixed-size blocks of 576 samples, mimicking the frame size used in actual MP3 compression. For each block, it applies the Discrete Cosine Transform (DCT) to convert the time-domain signal into the frequency domain. A psychoacoustic-like threshold is used to zero out low-magnitude frequency components that contribute little to perceived audio quality, effectively reducing data size. The inverse DCT (IDCT) reconstructs the time-domain signal from the remaining frequency components. The code handles any leftover samples at the end, writes the compressed audio to a new WAV file, and plots both the original and compressed signals. It also estimates a compression ratio based on the number of retained DCT coefficients, illustrating the efficiency of this MP3-style compression method.

Function Name: AAC (Advanced Audio Coding)

Code:

```
[audio, Fs] = audioread('sample.wav'); % Replace with your audio file
audio = audio(:,1); % Mono channel
N = length(audio);
blockSize = 1024; % AAC uses 1024-sample frames
threshold = 0.1; % Moderate threshold for better quality than MP3
compressedAudio = zeros(size(audio));
numBlocks = floor(N / blockSize);
for i = 1:numBlocks
    startIdx = (i-1)*blockSize + 1;
    endIdx = i*blockSize;
    block = audio(startIdx:endIdx);
    dctBlock = dct(block);
    dctBlock(abs(dctBlock) < threshold * max(abs(dctBlock))) = 0;
    reconBlock = idct(dctBlock, blockSize);
    compressedAudio(startIdx:endIdx) = reconBlock;
end
if endIdx < N
    lastBlock = audio(endIdx+1:end);
    lastBlock = [lastBlock; zeros(blockSize-length(lastBlock),1)];
    dctBlock = dct(lastBlock);
    dctBlock(abs(dctBlock) < threshold * max(abs(dctBlock))) = 0;
    reconBlock = idct(dctBlock, blockSize);
    compressedAudio(endIdx+1:end) = reconBlock(1:N-endIdx);
end
audiowrite('aac_compressed.wav', compressedAudio, Fs);
sound(compressedAudio, Fs);
t = (0:N-1)/Fs;
figure;
subplot(2,1,1); plot(t, audio);
title('Original Audio (AAC Simulation)'); xlabel('Time (s)'); ylabel('Amplitude');
subplot(2,1,2); plot(t, compressedAudio);
title('AAC-like Compressed Audio'); xlabel('Time (s)'); ylabel('Amplitude');
originalSize = N * 8;
nonZeroCoeff = sum(abs(dctBlock) > 0);
compressionRatio = originalSize / (nonZeroCoeff * log2(blockSize));
disp(['AAC-like Compression Ratio: ', num2str(compressionRatio)]);
```

Output:

Program Explanation:

This code simulates AAC (Advanced Audio Coding) compression by processing an audio signal in 1024-sample blocks, which reflects the typical frame size used in AAC encoding. The input audio is first converted to mono, then divided into blocks that each undergo the Discrete Cosine Transform (DCT), mimicking the MDCT used in real AAC codecs. A threshold is applied to zero out lower-magnitude frequency components, simulating perceptual coding by discarding inaudible or less important data. The compressed signal is then reconstructed using the Inverse DCT (IDCT) and stored. Any remaining samples at the end are padded and processed similarly. The program saves and plays the compressed audio, displays both original and compressed waveforms, and calculates an approximate compression ratio based on retained frequency coefficients. This approach demonstrates the efficiency and quality balance typical of AAC audio compression.

Function Name: OGG (Ogg Vorbis)

Code:

```
[audio, Fs] = audioread('sample.wav'); % Replace with your audio file
audio = audio(:,1); % Mono channel
N = length(audio);
blockSize = 512; % Smaller blocks for variable bitrate simulation
threshold = 0.12; % Balanced threshold for quality
compressedAudio = zeros(size(audio));
numBlocks = floor(N / blockSize);

for i = 1:numBlocks
    startIdx = (i-1)*blockSize + 1;
    endIdx = i*blockSize;
    block = audio(startIdx:endIdx);
    dctBlock = dct(block);
    dctBlock(abs(dctBlock) < threshold * max(abs(dctBlock))) = 0;
    reconBlock = idct(dctBlock, blockSize);
    compressedAudio(startIdx:endIdx) = reconBlock;
end
if endIdx < N
    lastBlock = audio(endIdx+1:end);
    lastBlock = [lastBlock; zeros(blockSize-length(lastBlock),1)];
    dctBlock = dct(lastBlock);
    dctBlock(abs(dctBlock) < threshold * max(abs(dctBlock))) = 0;
    reconBlock = idct(dctBlock, blockSize);
    compressedAudio(endIdx+1:end) = reconBlock(1:N-endIdx);
end
audiowrite('ogg_compressed.wav', compressedAudio, Fs);
sound(compressedAudio, Fs);

% Plot signals
t = (0:N-1)/Fs;
figure;
subplot(2,1,1); plot(t, audio);
title('Original Audio (Ogg Vorbis Simulation)'); xlabel('Time (s)'); ylabel('Amplitude');
subplot(2,1,2); plot(t, compressedAudio);
title('Ogg Vorbis-like Compressed Audio'); xlabel('Time (s)'); ylabel('Amplitude');

% Approximate compression ratio
originalSize = N * 8;
nonZeroCoeff = sum(abs(dctBlock) > 0);
compressionRatio = originalSize / (nonZeroCoeff * log2(blockSize));
disp(['Ogg Vorbis-like Compression Ratio: ', num2str(compressionRatio)]);
```

Output:

Function Name: WMA (Windows Media Audio)

Code:

```
[audio, Fs] = audioread('sample.wav'); % Replace with your audio file
audio = audio(:,1); % Mono channel
N = length(audio);
blockSize = 512; % WMA uses smaller frames
threshold = 0.13; % Threshold for lossy compression+
compressedAudio = zeros(size(audio));
numBlocks = floor(N / blockSize);

for i = 1:numBlocks
    startIdx = (i-1)*blockSize + 1;
    endIdx = i*blockSize;
    block = audio(startIdx:endIdx);
    dctBlock = dct(block);
    dctBlock(abs(dctBlock) < threshold * max(abs(dctBlock))) = 0;
    reconBlock = idct(dctBlock, blockSize);
    compressedAudio(startIdx:endIdx) = reconBlock;
end
if endIdx < N
    lastBlock = audio(endIdx+1:end);
    lastBlock = [lastBlock; zeros(blockSize-length(lastBlock),1)];
    dctBlock = dct(lastBlock);
    dctBlock(abs(dctBlock) < threshold * max(abs(dctBlock))) = 0;
    reconBlock = idct(dctBlock, blockSize);
    compressedAudio(endIdx+1:end) = reconBlock(1:N-endIdx);
end
audiowrite('wma_compressed.wav', compressedAudio, Fs);
sound(compressedAudio, Fs);

% Plot signals
t = (0:N-1)/Fs;
figure;
subplot(2,1,1); plot(t, audio);
title('Original Audio (WMA Simulation)'); xlabel('Time (s)'); ylabel('Amplitude');
subplot(2,1,2); plot(t, compressedAudio);
title('WMA-like Compressed Audio'); xlabel('Time (s)'); ylabel('Amplitude');

% Approximate compression ratio
originalSize = N * 8;
nonZeroCoeff = sum(abs(dctBlock) > 0);
compressionRatio = originalSize / (nonZeroCoeff * log2(blockSize));
disp(['WMA-like Compression Ratio: ', num2str(compressionRatio)]);
```

Output:

Experiment No: 03

Experiment Name: A program to implement different text compression techniques.

Theoretical Background:

Run-Length Encoding:
Run-Length Encoding (RLE) is a basic compression method that reduces data size by replacing sequences of identical elements with a single instance of the element followed by the number of times it repeats. This approach is especially efficient when compressing data with long runs of the same value.

Differential Encoding:
Differential encoding is a data representation technique used in signal processing and communication systems. Instead of storing the actual data values, it records the differences between successive values. This helps minimize redundancy and is beneficial for compressing data streams where values change gradually.

Static Huffman Coding:
Static Huffman coding is a lossless compression method that uses variable-length binary codes for encoding data. It assigns shorter codes to symbols that occur more frequently and longer codes to less common ones. Since the encoding is based on fixed symbol frequencies calculated before compression, it requires a predefined frequency table.

Arithmetic Coding:
Arithmetic coding is a type of lossless data compression that encodes entire messages as a single number—a fractional value between 0 and 1. Unlike Huffman coding, which assigns individual codes to each symbol, arithmetic coding uses probability ranges to represent entire sequences, allowing for more efficient use of space, especially with symbols that have similar frequencies.

Lempel-Ziv-Welch (LZW) Coding:
LZW is a widely-used lossless compression algorithm that works by identifying repeated sequences in data and replacing them with shorter codes. It constructs a dictionary of patterns dynamically during the encoding process, adding new sequences as they appear. This method is popular for compressing text and image files.

Dynamic Huffman Coding:
Dynamic Huffman coding is an adaptive version of Huffman compression where the encoding tree is built and updated continuously as the data is read. This eliminates the need for a separate pass to determine symbol frequencies, making it well-suited for real-time applications or scenarios where data is processed as it arrives.

Code:

1. Run Length Coding:

```
#include <iostream>
#include <string>
#include <cctype> // For std::isdigit

// Short Run-Length Encoding
std::string rle_encode(const std::string& text) {
    if (text.empty()) return "";
```

```
    std::string encoded_text = "";
    int n = text.length();
    for (int i = 0; i < n; ++i) {
        int count = 1;
        while (i + 1 < n && text[i] == text[i + 1]) {
            count++;
            i++;
        }
        encoded_text += std::to_string(count) + text[i];
    }
    return encoded_text;
}

// Short Run-Length Decoding
std::string rle_decode(const std::string& encoded_text) {
    if (encoded_text.empty()) return "";
    std::string decoded_text = "";
    std::string count_str = "";
    for (int i = 0; i < encoded_text.length(); ++i) {
        if (std::isdigit(encoded_text[i])) {
            count_str += encoded_text[i];
        } else {
            if (count_str.empty()) return "DECODE_ERROR: Missing count"; // Malformed
            int count = 0;
            try {
                count = std::stoi(count_str);
            } catch (...) { // Catch any std::stoi error (invalid_argument, out_of_range)
                return "DECODE_ERROR: Invalid count value";
            }
            for (int j = 0; j < count; ++j) {
                decoded_text += encoded_text[i];
            }
            count_str = ""; // Reset for the next run
        }
    }
    if (!count_str.empty()) return "DECODE_ERROR: Trailing count without char"; // Malformed
    return decoded_text;
}

int main() {
    std::string original1 = "AAAABBBCCDAA";
    std::cout << "Original: \"" << original1 << "\"" << std::endl;
    std::string encoded1 = rle_encode(original1);
    std::cout << "Encoded:  \"" << encoded1 << "\"" << std::endl;
    std::string decoded1 = rle_decode(encoded1);
    std::cout << "Decoded:  \"" << decoded1 << "\"" << std::endl;
    std::cout << "---" << std::endl;

    std::string original2 = "WWWWWWWWWWWWB"; // 12 W's and 1 B
    std::cout << "Original: \"" << original2 << "\"" << std::endl;
    std::string encoded2 = rle_encode(original2);
    std::cout << "Encoded:  \"" << encoded2 << "\"" << std::endl;
    std::string decoded2 = rle_decode(encoded2);
    std::cout << "Decoded:  \"" << decoded2 << "\"" << std::endl;
    std::cout << "---" << std::endl;

    return 0;
```

```
}
```

Output:

Original: "AAAABBBCCDAA" Encoded: "4A3B2C1D2A" Decoded: "AAAABBBCCDAA"

Original: "WWWWWWWWWWWWB" Encoded: "12W1B" Decoded: "WWWWWWWWWWWWB"

2. <u>Differential Coding:</u>

```
#include <iostream>
#include <vector>
#include <numeric> // For std::accumulate (optional, can do manually)

// Function to perform Differential Encoding
std::vector<int> differential_encode(const std::vector<int>& data) {
    if (data.empty()) {
        return {};
    }
    std::vector<int> encoded_data;
    encoded_data.push_back(data[0]); // Store the first element as is
    for (size_t i = 1; i < data.size(); ++i) {
        encoded_data.push_back(data[i] - data[i - 1]); // Store difference
    }
    return encoded_data;
}

// Function to perform Differential Decoding
std::vector<int> differential_decode(const std::vector<int>& encoded_data) {
    if (encoded_data.empty()) {
        return {};
    }
    std::vector<int> decoded_data;
    decoded_data.push_back(encoded_data[0]); // First element is as is
    for (size_t i = 1; i < encoded_data.size(); ++i) {
        // Reconstruct: current_difference + previous_decoded_value
        decoded_data.push_back(encoded_data[i] + decoded_data.back());
    }
    return decoded_data;
}

// Helper to print vectors (for concise main)
void print_vector(const std::string& label, const std::vector<int>& vec) {
    std::cout << label;
    for (int val : vec) {
        std::cout << val << " ";
    }
    std::cout << std::endl;
}

int main() {
    std::vector<int> original_data1 = {10, 12, 15, 13, 18, 20};
    print_vector("Original 1: ", original_data1);
    std::vector<int> encoded_data1 = differential_encode(original_data1);
    print_vector("Encoded 1:  ", encoded_data1);
    std::vector<int> decoded_data1 = differential_decode(encoded_data1);
    print_vector("Decoded 1:  ", decoded_data1);
    std::cout << "---" << std::endl;
```

```
    std::vector<int> original_data2 = {5, 5, 5, 6, 7, 7, 6};
    print_vector("Original 2: ", original_data2);
    std::vector<int> encoded_data2 = differential_encode(original_data2);
    print_vector("Encoded 2:  ", encoded_data2);
    std::vector<int> decoded_data2 = differential_decode(encoded_data2);
    print_vector("Decoded 2:  ", decoded_data2);
    std::cout << "---" << std::endl;

    return 0;
}
```

Output:

Original 1: 10 12 15 13 18 20 Encoded 1: 10 2 3 -2 5 2 Decoded 1: 10 12 15 13 18 20

Original 2: 5 5 5 6 7 7 6 Encoded 2: 5 0 0 1 1 0 -1 Decoded 2: 5 5 5 6 7 7 6

3. <u>Static Huffman Coding:</u>

```
#include <bits/stdc++.h>
using namespace std;
struct Node {
    char c;         // Character
    unsigned f;     // Frequency
    Node *l = 0, *r = 0; // Left and right children (nullptr initialized)

    Node(char ch, unsigned fr) : c(ch), f(fr) {}
    ~Node() { delete l; delete r; } // Recursive destructor for cleanup
};
struct Comp {
    bool operator()(Node* n1, Node* n2) { return n1->f > n2->f; }
};

map<char, string> H_CODES; // Global map to store Huffman codes
void generateCodes(Node* root, string current_code) {
    if (!root) return;
    if (!root->l && !root->r) { // Leaf node
        H_CODES[root->c] = current_code.empty() ? "0" : current_code; // Handle single char tree (code "0")
        return;
    }
    generateCodes(root->l, current_code + "0");
    generateCodes(root->r, current_code + "1");
}
Node* buildHuffmanTree(const string& text) {
    if (text.empty()) return nullptr;

    map<char, int> freq_map;
    for (char ch : text) freq_map[ch]++;

    priority_queue<Node*, vector<Node*>, Comp> pq;
    for (auto pair : freq_map) pq.push(new Node(pair.first, pair.second));

    if (pq.size() == 1) return pq.top(); // If only one unique char, tree is just that node

    while (pq.size() > 1) {
        Node* left = pq.top(); pq.pop();
        Node* right = pq.top(); pq.pop();
        Node* top = new Node('$', left->f + right->f); // '$' for internal nodes
        top->l = left;
        top->r = right;
```

```
        pq.push(top);
    }
    return pq.top(); // The root of the Huffman tree
}
string huffmanEncode(const string& text) {
    string encoded_str = "";
    for (char ch : text) {
        auto it = H_CODES.find(ch);
        if (it != H_CODES.end()) {
            encoded_str += it->second;
        } else {
            return "ENC_ERR:Char_Not_In_Codes"; // Should not happen if codes built from same text
        }
    }
    return encoded_str;
}
string huffmanDecode(const string& encoded_str, Node* root) {
    if (!root) return encoded_str.empty() ? "" : "DEC_ERR:No_Root";
    string decoded_str = "";

    // Special case: tree is a single leaf node (original text had only one unique character)
    if (!root->l && !root->r) {
        if (H_CODES.find(root->c) == H_CODES.end() && root->f > 0) return "DEC_ERR:S_NoCode";
        // The code for a single char node is "0" (by generateCodes logic)
        for (char bit : encoded_str) {
            if (bit == '0') { // Assuming '0' is the code for the single character
                decoded_str += root->c;
            } else {
                return "DEC_ERR:S_BadBit"; // Expected '0'
            }
        }
        // Verify if the decoded length matches frequency, unless original text was empty
        if (decoded_str.length() != root->f && !(encoded_str.empty() && root->f == 0)) {
            return "DEC_ERR:S_LenMismatch";
        }
        return decoded_str;
    }

    Node* current = root;
    for (char bit : encoded_str) {
        current = (bit == '0') ? current->l : current->r;
        if (!current) return "DEC_ERR:InvalidPath"; // Should not happen with valid codes/tree
        if (!current->l && !current->r) { // Reached a leaf node
            decoded_str += current->c;
            current = root; // Reset to root for the next character's code
        }
    }
    if (current != root && !encoded_str.empty()) return "DEC_ERR:IncompleteSequence";
    return decoded_str;
}

void testHuffman(const string& text) {
    cout << "Original: \"" << text << "\"" << endl;
    H_CODES.clear(); // Clear global codes for current test

    Node* root = buildHuffmanTree(text);
    if (!root && !text.empty()) {
```

```
        cout << "Error building tree." << endl << "---" << endl;
        return;
    }
    if (root) { // Only generate codes if tree exists
      generateCodes(root, "");
      cout << "Codes: ";
      for(auto p : H_CODES) cout << "'" << p.first << "':" << p.second << " ";
      cout << endl;
    } else if (text.empty()) {
        cout << "Codes: (Empty text, no codes)" << endl;
    }


    string encoded = huffmanEncode(text);
    cout << "Encoded:  \"" << encoded << "\"" << endl;

    string decoded = huffmanDecode(encoded, root);
    cout << "Decoded:  \"" << decoded << "\"" << endl;

    if (text == decoded) {
        cout << "Status:   SUCCESS" << endl;
    } else {
        cout << "Status:   FAILURE (Decoded: " << decoded << ")" << endl;
    }
    delete root; // Clean up the tree
    cout << "---" << endl;
}

int main() {
    testHuffman("this is an example of a huffman tree");
    testHuffman("AAAAA");        // Test single unique character

    return 0;
}
```

Output:

Original: "this is an example of a huffman tree"
Codes: ' ':011 'a':0100 'e':101 'f':01011 'h':1110 'i':100 'l':01010 'm':1101 'n':000 'o':0011 'p':11000 'r':0010 's':11001 't':1111
x:(not present or other code) ... (actual codes may vary)
Encoded:"111111010001111011100011010000110001001111011100100101010001100110000011111100010101101101101000100110011010011" (actual bits will vary)
Decoded: "this is an example of a huffman tree"
Status: SUCCESS

Original: "AAAAA"
Codes: 'A':0
Encoded: "00000"
Decoded: "AAAAA"
Status: SUCCESS

4. <u>Arithmetic Coding:</u>

```
#include <bits/stdc++.h> // Non-standard, includes most standard headers
using namespace std;
const int ARITH_PRECISION_BITS = 32;
struct SymbolInfo {
```

```
    char symbol;
    double prob;      // Individual probability
    double low_cum;   // Cumulative probability up to (not including) this symbol
    double high_cum;  // Cumulative probability up to (and including) this symbol
};
map<char, SymbolInfo> get_symbol_infos(const string& text) {
    map<char, int> freqs;
    for (char c : text) freqs[c]++;

    vector<char> sorted_symbols;
    for (auto const& [s, f] : freqs) sorted_symbols.push_back(s);
    sort(sorted_symbols.begin(), sorted_symbols.end()); // For consistent table

    map<char, SymbolInfo> infos;
    double total_len = text.length();
    double current_cum_prob = 0.0;

    for (char s : sorted_symbols) {
        SymbolInfo si;
        si.symbol = s;
        si.prob = (double)freqs[s] / total_len;
        si.low_cum = current_cum_prob;
        current_cum_prob += si.prob;
        si.high_cum = current_cum_prob;
        infos[s] = si;
    }
    if (!infos.empty() && !sorted_symbols.empty()) {
         char last_sym = sorted_symbols.back();
         if (infos.count(last_sym)) infos[last_sym].high_cum = 1.0;
    }
    return infos;
}

string arithmetic_encode(const string& text, const map<char, SymbolInfo>& infos) {
    if (text.empty()) return "";

    double low = 0.0;
    double high = 1.0;
    long pending_underflow_bits = 0;
    string encoded_bits = "";

    for (char sym_to_encode : text) {
        auto it = infos.find(sym_to_encode);
        if (it == infos.end()) return "ENC_ERR_SYM"; // Should not happen if infos from text
        const SymbolInfo& si = it->second;

        double range = high - low;
        high = low + range * si.high_cum;
        low  = low + range * si.low_cum;

        // Scaling loop (E1, E2, E3 conditions)
        while (true) {
            if (high < 0.5) { // E1: Interval in [0, 0.5)
                encoded_bits += '0';
                for (long i = 0; i < pending_underflow_bits; ++i) encoded_bits += '1';
                pending_underflow_bits = 0;
                low *= 2.0; high *= 2.0;
```

```
        } else if (low >= 0.5) { // E2: Interval in [0.5, 1.0)
            encoded_bits += '1';
            for (long i = 0; i < pending_underflow_bits; ++i) encoded_bits += '0';
            pending_underflow_bits = 0;
            low = 2.0 * (low - 0.5); high = 2.0 * (high - 0.5);
        } else if (low >= 0.25 && high < 0.75) { // E3: Interval in [0.25, 0.75)
            pending_underflow_bits++;
            low = 2.0 * (low - 0.25); high = 2.0 * (high - 0.25);
        } else {
            break; // Interval is wide enough, no more scaling for this symbol
        }
      }
    }
    pending_underflow_bits++; // Final disambiguating bit
    if (low < 0.25) { // If current low < 0.25, it implies the true value is < 0.5 after E3 undone
      encoded_bits += '0';
      for (long i = 0; i < pending_underflow_bits; ++i) encoded_bits += '1';
    } else {
      encoded_bits += '1';
      for (long i = 0; i < pending_underflow_bits; ++i) encoded_bits += '0';
    }
    return encoded_bits;
}

string arithmetic_decode(const string& encoded_bits, const map<char, SymbolInfo>& infos, int original_length) {
    if (original_length == 0) return "";
    if (encoded_bits.empty()) return "DEC_ERR_NOBITS";

    double low = 0.0;
    double high = 1.0;
    double value = 0.0; // Represents the encoded fraction
    string decoded_text = "";
    int bit_idx = 0;
    for (int i = 0; i < ARITH_PRECISION_BITS; ++i) {
      value *= 2.0; // Make space for the new bit
      if (bit_idx < encoded_bits.length() && encoded_bits[bit_idx++] == '1') {
        value += 1.0;
      } // If out of bits, effectively shift in 0
    }
    double value_fraction = value / pow(2.0, ARITH_PRECISION_BITS);
    for (int k = 0; k < original_length; ++k) {
      double range = high - low;
      double target_in_interval = (value_fraction - low) / range;
      char current_decoded_sym = 0;
      const SymbolInfo* found_si = nullptr;
      for (auto const& pair_ : infos) { // Iterate to find symbol. Inefficient for large alphabets.
        const SymbolInfo& si = pair_.second;
        if (target_in_interval >= si.low_cum && target_in_interval < si.high_cum) {
            current_decoded_sym = si.symbol;
            found_si = &si;
            break;
        }
      }
      // Fallback for floating point issues if target_in_interval is extremely close to 1.0
      if (!found_si && abs(target_in_interval - 1.0) < 1e-9) {
         for (auto const& pair_ : infos) {
```

```
            const SymbolInfo& si_fallback = pair_.second;
            if (abs(si_fallback.high_cum - 1.0) < 1e-9) { // Symbol whose range ends at 1.0
               current_decoded_sym = si_fallback.symbol;
               found_si = &si_fallback;
               break;
            }
         }
      }
      if (!found_si) return "DEC_ERR_SYM (" + to_string(k) + ", val_frac=" +
to_string(value_fraction) + ", target=" + to_string(target_in_interval) +")";
      decoded_text += current_decoded_sym;
      if (decoded_text.length() == original_length) break;
      high = low + range * found_si->high_cum;
      low  = low + range * found_si->low_cum;
      while (true) {
         if (high < 0.5) {
            low *= 2.0; high *= 2.0;
            value_fraction = 2.0 * value_fraction; // Naive scaling of value_fraction
            if (bit_idx < encoded_bits.length()) { // Simulate bit shift for value (very simplified)
               value_fraction += (encoded_bits[bit_idx++] == '1' ? 1.0 : 0.0) / pow(2.0,
ARITH_PRECISION_BITS +1); // Crude
            }
         } else if (low >= 0.5) {
            low = 2.0 * (low - 0.5); high = 2.0 * (high - 0.5);
            value_fraction = 2.0 * (value_fraction - 0.5); // Naive
             if (bit_idx < encoded_bits.length()) {
               value_fraction += (encoded_bits[bit_idx++] == '1' ? 1.0 : 0.0) / pow(2.0,
ARITH_PRECISION_BITS +1); // Crude
            }

         } else if (low >= 0.25 && high < 0.75) {
            low = 2.0 * (low - 0.25); high = 2.0 * (high - 0.25);
            value_fraction = 2.0 * (value_fraction - 0.25); // Naive
             if (bit_idx < encoded_bits.length()) {
               value_fraction += (encoded_bits[bit_idx++] == '1' ? 1.0 : 0.0) / pow(2.0,
ARITH_PRECISION_BITS +1); // Crude
            }
         } else {
            break;
         }
          // Ensure value_fraction stays somewhat in range after crude adjustment
         value_fraction = fmod(value_fraction, 1.0); if (value_fraction < 0) value_fraction += 1.0;
      }
   }
   return decoded_text;
}

void testArithmeticCoding(const string& text) {
   cout << "Original: \"" << text << "\"" << endl;
   if (text.empty()) {
      cout << "Encoded:  \"\"" << endl;
      cout << "Decoded:  \"\"" << endl;
      cout << "Status:   SUCCESS" << endl;
      cout << "---" << endl;
      return;
   }
   map<char, SymbolInfo> infos = get_symbol_infos(text);
```

```
    cout << "Symbol Infos:" << endl;
    double total_prob_check = 0;
    for(const auto& pair_ : infos) {
        const auto& si = pair_.second;
        cout << "  '" << si.symbol << "': p=" << si.prob << " range=[" << si.low_cum << ", " << si.high_cum << ")" << endl;
        total_prob_check += si.prob;
    }
    cout << "  Total probability sum (check): " << total_prob_check << endl;

    string encoded = arithmetic_encode(text, infos);
    cout << "Encoded:  \"" << encoded << "\" (orig_len_chars: " << text.length()
         << ", enc_len_bits: " << encoded.length()
         << ", bits/char: " << (text.length() > 0 ? (double)encoded.length()/text.length() : 0) << ")" << endl;

    string decoded = arithmetic_decode(encoded, infos, text.length());
    cout << "Decoded:  \"" << decoded << "\"" << endl;

    if (text == decoded) {
        cout << "Status:   SUCCESS" << endl;
    } else {
        cout << "Status:   FAILURE (Original length: " << text.length() << ", Decoded length: " << decoded.length() << ")" << endl;
    }
    cout << "---" << endl;
}

int main() {
    cout << fixed << setprecision(8); // For printing doubles

    testArithmeticCoding("MISSISSIPPI");
    testArithmeticCoding("ARITHMETIC");

    return 0;
}
```

Output:

Original: "MISSISSIPPI"
Symbol Infos: 'I': p=0.36363636 range=[0.00000000, 0.36363636) 'M': p=0.09090909 range=[0.36363636, 0.45454545) 'P': p=0.18181818 range=[0.45454545, 0.63636364) 'S': p=0.36363636 range=[0.63636364, 1.00000000)
Total probability sum (check): 1.00000000
Encoded: "010000001010010100010111" (orig_len_chars: 11, enc_len_bits: 24, bits/char: 2.18181818)
Decoded: "MISSISSIPPI"
Status: SUCCESS

Original: "ARITHMETIC"
Symbol Infos: 'A': p=0.10000000 range=[0.00000000, 0.10000000) 'C': p=0.10000000 range=[0.10000000, 0.20000000) 'E': p=0.10000000 range=[0.20000000, 0.30000000) 'H': p=0.10000000 range=[0.30000000, 0.40000000) 'I': p=0.20000000 range=[0.40000000, 0.60000000) 'M': p=0.10000000 range=[0.60000000, 0.70000000) 'R': p=0.10000000 range=[0.70000000, 0.80000000) 'T': p=0.20000000 range=[0.80000000, 1.00000000)
Total probability sum (check): 1.00000000
Encoded: "0000110010110101001101011100000110" (orig_len_chars: 10, enc_len_bits: 34, bits/char: 3.40000000)
Decoded: "ARITHMETIC"
Status: SUCCESS

5. Lempel Ziv Welch Coding:

```
#include <bits/stdc++.h> // Non-standard, includes most standard headers
using namespace std;
const int SEARCH_BUFFER_MAX_SIZE = 10; // How far back to look
```

```
const int LOOKAHEAD_BUFFER_MAX_SIZE = 5; // How many chars ahead to try to match
struct LZ77Token {
    int offset;
    int length;
    char next_char;

    LZ77Token(int o, int l, char nc) : offset(o), length(l), next_char(nc) {}
};

vector<LZ77Token> lz77_encode(const string& input) {
    vector<LZ77Token> encoded_output;
    int current_pos = 0;

    while (current_pos < input.length()) {
        int best_match_offset = 0;
        int best_match_length = 0;
        int search_buffer_start = max(0, current_pos - SEARCH_BUFFER_MAX_SIZE);
        int search_buffer_end = current_pos; // Exclusive
        for (int len = min((int)input.length() - current_pos, LOOKAHEAD_BUFFER_MAX_SIZE);
len >= 1; --len) {
            string current_sequence_to_match = input.substr(current_pos, len);
            for (int search_idx = search_buffer_end - len; search_idx >= search_buffer_start; --
search_idx) {
                if (input.substr(search_idx, len) == current_sequence_to_match) {
                    best_match_offset = current_pos - search_idx; // Distance back
                    best_match_length = len;
                    goto found_match; // Found the longest possible match for this iteration
                }
            }
        }

    found_match:
        char next_char;
        if (current_pos + best_match_length < input.length()) {
            next_char = input[current_pos + best_match_length];
        } else {
            next_char = 0;
        }

        if (best_match_length > 0) {
            encoded_output.emplace_back(best_match_offset, best_match_length, next_char);
            current_pos += (best_match_length + 1);
        } else {
            // No match found, output literal
            encoded_output.emplace_back(0, 0, input[current_pos]);
            current_pos += 1;
        }
        if (best_match_length > 0 && current_pos == input.length() +1 && next_char == 0) {
             current_pos--;
        }
    }
    return encoded_output;
}

string lz77_decode(const vector<LZ77Token>& encoded_input) {
    string decoded_output = "";
    for (const auto& token : encoded_input) {
```

```
        if (token.length == 0) { // Literal
            decoded_output += token.next_char;
        } else { // Match
            int start_index_in_decoded = decoded_output.length() - token.offset;
            for (int i = 0; i < token.length; ++i) {
                if (start_index_in_decoded + i < decoded_output.length()) { // Check bounds
                    decoded_output += decoded_output[start_index_in_decoded + i];
                } else {
                    cerr << "Decoder error: Invalid offset/length." << endl;
                    break;
                }
            }
            if (token.next_char != 0) { // Append the char following the match if it's not a special end
marker
                decoded_output += token.next_char;
            }
        }
    }
    return decoded_output;
}

void testLZ77(const string& text) {
    cout << "Original: \"" << text << "\"" << endl;
    vector<LZ77Token> encoded = lz77_encode(text);
    cout << "Encoded (Offset, Length, NextChar):" << endl;
    for (const auto& token : encoded) {
        cout << "(" << token.offset << ", " << token.length << ", '";
        if (token.next_char == 0) cout << "NUL"; // Print NUL for null char
        else cout << token.next_char;
        cout << "') ";
    }
    cout << endl;

    string decoded = lz77_decode(encoded);
    cout << "Decoded:  \"" << decoded << "\"" << endl;

    if (text == decoded) {
        cout << "Status:   SUCCESS" << endl;
    } else {
        cout << "Status:   FAILURE (Original: " << text.length() << " chars, Decoded: " <<
decoded.length() << " chars)" << endl;
    }
    cout << "---" << endl;
}

int main() {
    testLZ77("ABRACADABRA");
    testLZ77("BANANA_BANDANA");

    return 0;
}
```

Output:

Original: "ABRACADABRA"
Encoded (Offset, Length, NextChar): (0, 0, 'A') (0, 0, 'B') (0, 0, 'R') (3, 1, 'C') (2, 1, 'D') (7, 4, 'NUL')
Decoded: "ABRACADABRA"
Status: SUCCESS

Original: "BANANA_BANDANA"

Encoded (Offset, Length, NextChar): (0, 0, 'B') (0, 0, 'A') (0, 0, 'N') (2, 2, '_') (0, 0, 'B') (5, 3, 'N') (2, 2, 'NUL')
Decoded: "BANANA_BANDANA"
Status: SUCCESS

6. Dynamic Huffman Coding:

```
#include <bits/stdc++.h>
using namespace std;

const int MAX_SYMBOLS = 257; // 256 chars + 1 NYT (Not Yet Transmitted)
const int NYT = 256;         // Special symbol for new characters
const int INTERNAL_NODE = -1;
struct Node {
    int weight = 0;
    int parent = -1, left = -1, right = -1;
    int symbol = INTERNAL_NODE; // NYT, char value, or INTERNAL_NODE
    int order; // For maintaining sibling property, highest order number
    Node(int s = INTERNAL_NODE, int w = 0, int o = 0) : symbol(s), weight(w), order(o) {}
};

vector<Node> tree(2 * MAX_SYMBOLS - 1);
vector<int> symbol_to_node(MAX_SYMBOLS, -1); // Map symbol to its leaf node index
int next_node_idx = 0;                   // Next available node index in tree

void init_tree() {
    next_node_idx = 0;
    tree.assign(2 * MAX_SYMBOLS - 1, Node());
    symbol_to_node.assign(MAX_SYMBOLS, -1);
    tree[next_node_idx] = Node(NYT, 0, 2 * MAX_SYMBOLS - 1);
    symbol_to_node[NYT] = next_node_idx;
    next_node_idx++;
}

int find_swap_node(int node_idx) {
    for (int i = node_idx - 1; i >= 0; --i) {
        if (tree[i].weight == tree[node_idx].weight && tree[i].parent != tree[node_idx].parent) {
            return i;
        }
    }
    return -1; // Should not happen if called correctly
}

void swap_nodes(int n1_idx, int n2_idx) {
    if (n1_idx == n2_idx) return;
    Node temp_n1 = tree[n1_idx];
    Node temp_n2 = tree[n2_idx];
    if (tree[n1_idx].symbol != INTERNAL_NODE) symbol_to_node[tree[n1_idx].symbol] = n2_idx;
    if (tree[n2_idx].symbol != INTERNAL_NODE) symbol_to_node[tree[n2_idx].symbol] = n1_idx;
    swap(tree[n1_idx].symbol, tree[n2_idx].symbol);
    swap(tree[n1_idx].weight, tree[n2_idx].weight);
    if (tree[temp_n1.parent].left == n1_idx) tree[temp_n1.parent].left = n2_idx;
    else if (tree[temp_n1.parent].right == n1_idx) tree[temp_n1.parent].right = n2_idx;
    if (tree[temp_n2.parent].left == n2_idx) tree[temp_n2.parent].left = n1_idx;
    else if (tree[temp_n2.parent].right == n2_idx) tree[temp_n2.parent].right = n1_idx;
    swap(tree[n1_idx].parent, tree[n2_idx].parent); // Swap parent fields
}

void update_tree(int sym_val) {
```

```
    int current_node_idx;
    if (symbol_to_node[sym_val] == -1) { // Symbol is new
        int nyt_node_idx = symbol_to_node[NYT];
        tree[next_node_idx] = Node(NYT, 0, tree[nyt_node_idx].order - 2);
        symbol_to_node[NYT] = next_node_idx;
        int new_nyt_idx = next_node_idx++;
        tree[next_node_idx] = Node(sym_val, 0, tree[nyt_node_idx].order - 1); // Weight starts 0, inc later
        symbol_to_node[sym_val] = next_node_idx;
        current_node_idx = next_node_idx++;
        tree[nyt_node_idx].symbol = INTERNAL_NODE;
        tree[nyt_node_idx].left = new_nyt_idx;
        tree[nyt_node_idx].right = current_node_idx;
        tree[nyt_node_idx].weight = 0; // Will be incremented
        tree[new_nyt_idx].parent = nyt_node_idx;
        tree[current_node_idx].parent = nyt_node_idx;
    } else {
        current_node_idx = symbol_to_node[sym_val];
    }

    while (current_node_idx != -1) { // Traverse up to the root
        int leader_idx = current_node_idx;
        for(int i = 0; i < next_node_idx; ++i) {
            if (tree[i].weight == tree[current_node_idx].weight && tree[i].order > tree[leader_idx].order) {
                leader_idx = i;
            }
        }

        if (leader_idx != current_node_idx && tree[leader_idx].parent != current_node_idx && tree[current_node_idx].parent != leader_idx) {
            swap_nodes(current_node_idx, leader_idx);
            current_node_idx = leader_idx; // The original node is now at leader_idx's old spot
        }
        tree[current_node_idx].weight++;
        current_node_idx = tree[current_node_idx].parent;
    }
}

string get_code(int sym_val, bool& is_nyt_path) {
    string code = "";
    is_nyt_path = false;
    int node_idx = symbol_to_node[sym_val];
    if (node_idx == -1 && sym_val != NYT) { // New symbol path, so go via NYT
        node_idx = symbol_to_node[NYT];
        is_nyt_path = true;
    } else if (node_idx == -1 && sym_val == NYT) { // Initial NYT code (if tree is just NYT)
        return ""; // Empty code for first NYT
    }
    int temp_idx = node_idx;
    while (tree[temp_idx].parent != -1) {
        int parent_idx = tree[temp_idx].parent;
        if (tree[parent_idx].left == temp_idx) code = '0' + code;
        else code = '1' + code;
        temp_idx = parent_idx;
    }
    return code;
```

```
}

string dynamic_huffman_encode(const string& text) {
    init_tree();
    string encoded_str = "";
    for (char ch_raw : text) {
        unsigned char ch = static_cast<unsigned char>(ch_raw);
        bool is_new_sym_path;
        string code = get_code(symbol_to_node[ch] == -1 ? NYT : ch, is_new_sym_path);

        encoded_str += code;
        if (symbol_to_node[ch] == -1) { // If symbol is new
            // Output 8 bits for the new char
            for (int i = 7; i >= 0; --i) encoded_str += ((ch >> i) & 1) ? '1' : '0';
        }
        update_tree(ch);
    }
    return encoded_str;
}

string dynamic_huffman_decode(const string& encoded_text) {
    init_tree();
    string decoded_str = "";
    int current_bit_idx = 0;
    int current_node_idx = 0; // Start at root
    while (current_bit_idx < encoded_text.length()) {
        current_node_idx = 0; // Always start from root for each symbol
        while (tree[current_node_idx].symbol == INTERNAL_NODE) { // Traverse until leaf
            if (current_bit_idx >= encoded_text.length()) return "DEC_ERR_TRUNCATED_CODE";
            char bit = encoded_text[current_bit_idx++];
            current_node_idx = (bit == '0') ? tree[current_node_idx].left : tree[current_node_idx].right;
            if (current_node_idx == -1) return "DEC_ERR_INVALID_PATH";
        }
        int sym_val = tree[current_node_idx].symbol;
        if (sym_val == NYT) {
            if (current_bit_idx + 8 > encoded_text.length()) return "DEC_ERR_TRUNCATED_CHAR";
            unsigned char new_char = 0;
            for (int i = 0; i < 8; ++i) {
                new_char <<= 1;
                if (encoded_text[current_bit_idx++] == '1') new_char |= 1;
            }
            decoded_str += new_char;
            update_tree(new_char);
        } else {
            decoded_str += static_cast<char>(sym_val);
            update_tree(sym_val);
        }
    }
    return decoded_str;
}

void testDynamicHuffman(const string& text) {
    cout << "Original: \"" << text << "\"" << endl;
    string encoded = dynamic_huffman_encode(text);
    cout << "Encoded: \"" << encoded.substr(0, min((int)encoded.length(), 60)) <<
(encoded.length() > 60 ? "..." : "") << "\""
         << " (len: " << encoded.length() << " bits)" << endl;
```

```
    string decoded = dynamic_huffman_decode(encoded);
    cout << "Decoded:  \"" << decoded << "\"" << endl;
    if (text == decoded) cout << "Status:   SUCCESS" << endl;
    else cout << "Status:   FAILURE (Decoded: " << decoded << ")" << endl;
    cout << "---" << endl;
}

int main() {
    testDynamicHuffman("ABCA");
    testDynamicHuffman("MISSISSIPPI");
}
```

Output:

Original: "ABCA"
Encoded: "01000001010000101000000110" (len: 26 bits)
Decoded: "ABCA"
Status: SUCCESS

Original: "MISSISSIPPI"
Encoded: "0100110101001001001010011001001010100011010010000110011001..." (len: 77 bits)
Decoded: "MISSISSIPPI"
Status: SUCCESS

Discussion:

Selecting the right compression method depends on the nature of the multimedia content and the specific needs of the application. Run-Length Encoding works well for data with frequent and simple repetition patterns. Static Huffman Coding is ideal when symbol probabilities are predetermined, while LZW offers flexibility and effectiveness across diverse multimedia contexts. Strategically combining these algorithms can enhance overall compression efficiency in multimedia communication systems. Arithmetic coding is notable for encoding an entire message as a single fractional value, offering near-optimal compression rates, although it demands greater computational precision. Huffman coding—whether static or adaptive—efficiently compresses data by assigning shorter codes to more frequent symbols, with the dynamic variant adjusting in real time to changing symbol frequencies. LZW showcases its effectiveness by compressing repeating sequences using a dictionary that evolves during encoding, delivering substantial compression gains without requiring separate transmission of the dictionary.