

# **LAPORAN TUGAS KECIL 2**

## **IF2211 STRATEGI ALGORITMA**

### **Kompresi Gambar Dengan Metode Quadtree**



Oleh Ghaisan Zaki Pratama

NIM : 10122078

Semester II Tahun 2024/2025

INSTITUT TEKNOLOGI BANDUNG

BANDUNG

2025

## Daftar Isi

<b>Daftar Isi</b>	2
<b><i>Algoritma Divide and Conquer</i></b>	3
<b><i>Source Program</i></b>	4
<b><i>Test Case</i></b>	22
<b>Hasil Analisis</b>	30
<b>Penjelasan Bonus</b>	31
<b><i>Pranala Repository</i></b>	31
<b>Lampiran</b>	32

## Algoritma *Divide and Conquer*

Pada program ini, *input* dibaca dari file berekstensi .JPG, .PNG, dan .JPEG yang akan diterapkan algoritma *divide and conquer* untuk mengkompresi file-file tersebut. Gambar di input dalam format Red Green Blue (RGB) dan dikonversi menjadi matriks piksel. Parameter-parameter seperti metode penghitungan error, threshold, ukuran blok, dan target kompresi juga menjadi input. Gambar direpresentasikan menjadi matriks tiga dimensi [*tinggi*][*lebar*][3] dengan 3 merupakan komponen untuk RGB.

Proses *Divide* (pembagian) dilakukan dengan seluruh gambar dianggap sebagai blok besar (*root node* dalam *quadtree*) terlebih dahulu. Blok tersebut akan diperiksa homogen atau tidak dengan menghitung error. Jika nilai error di atas threshold dan ukuran blok melebihi ukuran minimum, blok dibagi menjadi empat sub-blok dengan cara membagi lebar dan tinggi blok. Prosesnya dilakukan secara rekursif untuk setiap sub-blok yang masih heterogen (error tinggi).

Basis rekursif dalam algoritma nya terjadi ketika blok gambar tidak memenuhi syarat untuk dibagi lagi. Hal tersebut dapat terjadi jika error kurang dari threshold, atau tinggi blok (dapat bermakna ukuran dari sub-blok) sudah kurang dari minimum blok, atau lebar blok sudah kurang dari minimum blok (dapat bermakna ukuran dari sub-blok), atau ketika tinggi blok dibagi dua sudah kurang dari minimum blok, atau ketika lebar blok dibagi dua sudah kurang dari minimum blok. Blok tersebut dianggap homogen atau sudah tidak akan dibagi lagi. Dalam *quadtree* dia akan menjadi daun atau *leaf*.

Proses *Conquer* (penyelesaian sub-masalah) dilakukan proses rekursif untuk setiap blok yang memenuhi kondisi pembagian. Setiap pemanggilan rekursif memecah masalah kompresi untuk blok yang lebih kecil, sehingga setiap blok telah menjadi sub-masalah baru yang memiliki struktur yang sama dengan masalah awal. Proses rekursif terus dilakukan sampai blok dianggap homogen (error berada di bawah threshold) atau ukurannya telah mencapai batas minimum, maka algoritma tidak akan membagi blok tersebut lebih lanjut. Pada kondisi tersebut, blok dianggap sebagai simpul daun atau *leaf* dan nilai rata-rata warna dari blok tersebut dihitung sebagai representasi.

Proses *Combine* (penggabungan hasil) dilakukan dengan cara mengkombinasikan hasil dari proses rekursif ke dalam struktur *quadtree*. Setiap node menyimpan referensi ke 4 anaknya (untuk blok yang telah dibagi) atau nilai rata-rata warna (untuk simpul daun). Proses rekonstruksi menjadi gambar lagi dilakukan dengan traversing *quadtree* dan menggambar setiap simpul daun dengan blok berwarna rata-rata. Hasil rekonstruksi merupakan gabungan dari semua blok yang telah diproses.

Secara keseluruhan, langkah-langkah algoritmanya adalah:

1. Mulai dengan gambar utuh sebagai *input*.
2. Bagi gambar secara rekursif dan selesaikan setiap sub-blok nya dengan cek error.
3. Gabungkan hasil.
4. Tampilkan *output*.

## Source Program

```
import java.awt.image.BufferedImage;
import java.io.File;
import java.io.IOException;
import java.util.Scanner;
import javax.imageio.ImageIO;

public class Main {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        String folderPath = "test/";
        System.out.print("Masukkan nama file gambar yang akan dikompresi:");
        String inputFileName = scanner.nextLine();
        String inputImagePath = folderPath + inputFileName;
        System.out.println("Nama File: " + inputImagePath);
        BufferedImage image = null;
        try {
            image = ImageIO.read(new File(inputImagePath));
            if (image == null){
                System.out.println("Format gambar tidak didukung atau file tidak ditemukan");
                System.exit(1);
            }

        } catch (IOException e) {
            System.out.println("Gagal membaca gambar: " + e.getMessage());
            System.exit(1);
        }

        System.out.println("Gambar berhasil dimuat dengan ukuran: " + image.getWidth() + "x" + image.getHeight());

        String extension = "";
        int dotIndex = inputFileName.lastIndexOf('.');
        if (dotIndex > 0 && dotIndex < inputFileName.length()-1){
            extension = inputFileName.substring(dotIndex + 1);
        } else {
```

```

        System.out.println("Nama file input tidak valid, tidak
terdapat ekstensi");
        System.exit(1);
    }

    System.out.println("Pilih metode perhitungan variansi");
    System.out.println("1.Variance");
    System.out.println("2.Mean Absolute Deviation (MAD)");
    System.out.println("3.Max Pixel Difference");
    System.out.println("4.Entropy");
    System.out.print("Masukkan pilihan (nomor): ");
    int errorMethod = scanner.nextInt();

    System.out.print("Masukkan nilai ambang batas (threshold): ");
    double threshold = scanner.nextDouble();

    System.out.print("Masukkan ukuran blok minimum: ");
    int minBlockSize = scanner.nextInt();

    System.out.print("Masukkan target persentase kompresi (nilai dari
0 sampai 1): ");
    double targetCompression = scanner.nextDouble();

    scanner.nextLine();

    System.out.print("Masukkan nama file untuk gambar hasil kompresi:
");

    String outputFileName = scanner.nextLine();
    String outputImagePath = folderPath + outputFileName;

    System.out.print("Masukkan nama file untuk file GIF (tekan Enter
untuk melewati): ");
    String gifFileName = scanner.nextLine();
    String outputGifPath = "";
    if (!gifFileName.isEmpty()){
        outputGifPath = folderPath + gifFileName;
    }

    int width = image.getWidth();
    int height = image.getHeight();

```

```

int[][][] pixelMatrix = new int[height][width][3];

for (int y=0; y < height; y++){
    for (int x=0; x < width; x++){
        int pixel = image.getRGB(x,y);
        int red = (pixel >> 16) &0xff;
        int green = (pixel >> 8) &0xff;
        int blue = pixel &0xff;
        pixelMatrix[y][x][0] = red;
        pixelMatrix[y][x][1] = green;
        pixelMatrix[y][x][2] = blue;
    }
}

long originalSize = new File(inputImagePath).length();
long startTime = System.currentTimeMillis();

if (targetCompression != 0){
    threshold =
Bonus.CompressionUtils.adjustThreshold(pixelMatrix, width, height,
originalSize, minBlockSize, errorMethod, extension, targetCompression);
}

QuadTreeNode root = QuadtreeCompressor.buildQuadtree(pixelMatrix,
0, 0, width, height, threshold, minBlockSize, errorMethod);
System.out.println("Quadtree telah berhasil dibangun");

BufferedImage compressedImage =
QuadtreeCompressor.reconstructImage(root, width, height);

try {
    File outputFile = new File(outputImagePath);
    ImageIO.write(compressedImage, extension, outputFile);
    System.out.println("Gambar hasil kompresi berhasil disimpan
di: " + outputImagePath);
} catch (IOException e) {
    System.out.println("Gagal menyimpan gambar hasil kompresi: " +
e.getMessage());
}

long endTime = System.currentTimeMillis();

```

```

        long executionTime = endTime - startTime;

        long compressedSize = new File(outputImagePath).length();
        double compressionPercentage = (1-((double) compressedSize /
originalSize)) * 100;

        int treeDepth = QuadtreeCompressor.getTreeDepth(root);
        int totalNodes = QuadtreeCompressor.countNodes(root);

        System.out.println("Waktu eksekusi: " + executionTime);
        System.out.println("Ukuran gambar sebelum kompresi: " +
originalSize);
        System.out.println("Ukuran gambar setelah kompresi: " +
compressedSize);
        System.out.println("Persentase kompresi: " +
compressionPercentage);
        System.out.println("Kedalaman pohon: " + treeDepth);
        System.out.println("Banyak simpul pada pohon: " + totalNodes);

        double ssim = Bonus.CompressionUtils.computeSSIM(image,
compressedImage);
        System.out.printf("Nilai SSIM: %.4f\n", ssim);

        if (!outputGifPath.isEmpty()){
            Bonus.createCompressionGIFByLevel(root, outputGifPath, width,
height);
        }

        scanner.close();
    }
}

public class ErrorMeasurementMethods {
    public static double computeVariance(int[][][] block){
        int height = block.length;
        int width = block[0].length;
        int N = height * width;
        double[] sum = new double[3];

```

```

        for (int i = 0; i < height; i++){
            for (int j = 0; j < width; j++){
                for (int c = 0; c < 3; c++){
                    sum[c] += block[i][j][c];
                }
            }
        }

        double[] mean = new double[3];
        for (int c = 0; c < 3; c++){
            mean[c] = sum[c] / N;
        }

        double[] variance = new double[3];
        for (int i = 0; i < height; i++){
            for (int j = 0; j < width; j++){
                for (int c = 0; c < 3; c++){
                    variance[c] += Math.pow(block[i][j][c] - mean[c], 2);
                }
            }
        }

        for (int c = 0; c < 3; c++){
            variance[c] /= N;
        }

        double combinedVariance = (variance[0] + variance[1] +
variance[2])/3;
        return combinedVariance;
    }

    public static double computeMAD(int[][][] block){
        int height = block.length;
        int width = block[0].length;
        int N = height * width;
        double[] sum = new double[3];

        for (int i = 0; i < height; i++){
            for (int j = 0; j < width; j++){
                for (int c = 0; c < 3; c++){

```



```

        sum[c] += block[i][j][c];
    }
}

double[] mean = new double[3];
for (int c = 0; c < 3; c++){
    mean[c] = sum[c] / N;
}

double[] mad = new double[3];
for (int i = 0; i < height; i++){
    for (int j = 0; j < width; j++){
        for (int c = 0; c < 3; c++){
            mad[c] += Math.abs(block[i][j][c] - mean[c]);
        }
    }
}

for (int c = 0; c < 3; c++){
    mad[c] /= N;
}

double combinedMAD = (mad[0] + mad[1] + mad[2])/3;
return combinedMAD;
}

public static double computeMaxPixelDiffenrece(int[][][] block){
    int height = block.length;
    int width = block[0].length;
    int[] maxVal = {0,0,0};
    int[] minVal = {255,255,255};

    for (int i = 0; i < height; i++){
        for (int j = 0; j < width; j++){
            for (int c = 0; c < 3; c++){
                if (block[i][j][c] < minVal[c]){
                    minVal[c] = block[i][j][c];
                }
                if (block[i][j][c] > maxVal[c]){

```

```

        maxVal[c] = block[i][j][c];
    }
}

double[] diff = new double[3];
for (int c = 0; c < 3; c++){
    diff[c] = maxVal[c] - minVal[c];
}

double combinedDiff = (diff[0] + diff[1] + diff[2])/3;
return combinedDiff;
}

public static double computeEntropy(int[][][] block){
    int height = block.length;
    int width = block[0].length;
    int N = height * width;
    double combinedEntropy = 0.0;

    for (int c = 0; c < 3; c++){
        int[] hist = new int[256];
        for (int i = 0; i < height; i++){
            for (int j = 0; j < width; j++){
                int value = block[i][j][c];
                hist[value]++;
            }
        }

        double entropy = 0.0;
        for (int k = 0; k < 256; k++){
            if (hist[k] > 0){
                double p = (double) hist[k] / N;
                entropy += p * (Math.log(p)/Math.log(2));
            }
        }
        entropy = -entropy;
        combinedEntropy += entropy;
    }
}

```

```

        combinedEntropy /= 3;
        return combinedEntropy;
    }
}

public class QuadtreeNode {
    int startX, startY, width, height;
    double error;
    QuadtreeNode[] children;
    boolean isLeaf;
    int avgR, avgG, avgB;
    public boolean expanded = false;

    public QuadtreeNode(int startX, int startY, int width, int height){
        this.startX = startX;
        this.startY = startY;
        this.width = width;
        this.height = height;
        this.isLeaf = false;
        this.children = null;
    }
}

import java.awt.Color;
import java.awt.Graphics2D;
import java.awt.image.BufferedImage;

public class QuadtreeCompressor {
    public static QuadtreeNode buildQuadtree(int[][][] pixels, int startX,
int startY, int width, int height, double threshold, int minBlockSize, int
errorMethod) {
        int[][][] block = new int[height][width][3];
        for (int i=0; i < height; i++){
            for (int j=0; j < width; j++){
                block[i][j][0] = pixels[startY + i][startX + j][0];
                block[i][j][1] = pixels[startY + i][startX + j][1];
                block[i][j][2] = pixels[startY + i][startX + j][2];
            }
        }
    }
}

```

```

    }
}

double error;
switch (errorMethod) {
    case 1:
        error = ErrorMeasurementMethods.computeVariance(block);
        break;
    case 2:
        error = ErrorMeasurementMethods.computeMAD(block);
        break;
    case 3:
        error = ErrorMeasurementMethods.computeMaxPixelDiffenrece(block);
        break;
    case 4:
        error = ErrorMeasurementMethods.computeEntropy(block);
        break;
    default:
        error = ErrorMeasurementMethods.computeVariance(block);
        break;
}

QuadtreeNode node = new QuadtreeNode(startX, startY, width,
height);
node.error = error;

if (error > threshold && width > minBlockSize && height >
minBlockSize &&
    (width / 2) >= minBlockSize && (height / 2) >= minBlockSize) {

    int halfWidth = width / 2;
    int halfHeight = height / 2;
    // Menangani sisa piksel untuk dimensi ganjil
    int width2 = width - halfWidth;
    int height2 = height - halfHeight;

    node.children = new QuadtreeNode[4];
    node.children[0] = buildQuadtree(pixels, startX, startY,
halfWidth, halfHeight, threshold, minBlockSize, errorMethod);

```

```

        node.children[1] = buildQuadtree(pixels, startX + halfWidth,
startY, width2, halfHeight, threshold, minBlockSize, errorMethod);
        node.children[2] = buildQuadtree(pixels, startX, startY +
halfHeight, halfWidth, height2, threshold, minBlockSize, errorMethod);
        node.children[3] = buildQuadtree(pixels, startX + halfWidth,
startY + halfHeight, width2, height2, threshold, minBlockSize,
errorMethod);

        node.isLeaf = false;
    } else {
        int[] avgColor = computeAverageColor(block);
        node.avgR = avgColor[0];
        node.avgG = avgColor[1];
        node.avgB = avgColor[2];
        node.isLeaf = true;
    }

    return node;
}

public static int[] computeAverageColor(int[][][] block){
    int height = block.length;
    int width = block[0].length;
    long sumR = 0, sumG = 0, sumB = 0;
    int totalPixels = height * width;

    for (int i = 0; i < height; i++){
        for (int j = 0; j < width; j++){
            sumR += block[i][j][0];
            sumG += block[i][j][1];
            sumB += block[i][j][2];
        }
    }

    int avgR = (int) (sumR/totalPixels);
    int avgG = (int) (sumG/totalPixels);
    int avgB = (int) (sumB/totalPixels);
    return new int[]{avgR, avgG, avgB};
}

```

```

    public static BufferedImage reconstructImage(QuadtreeNode root, int
imageWidth, int imageHeight){
        BufferedImage reconstructed = new BufferedImage(imageWidth,
imageHeight, BufferedImage.TYPE_INT_RGB);
        Graphics2D g = reconstructed.createGraphics();
        drawNode(g, root);
        g.dispose();
        return reconstructed;
    }

    public static void drawNode(Graphics2D g, QuadtreeNode node){
        if (node.isLeaf){
            Color avgColor = new Color(node.avgR, node.avgG, node.avgB);
            g.setColor(avgColor);
            g.fillRect(node.startX, node.startY, node.width, node.height);
        }else {
            for (QuadtreeNode child : node.children){
                drawNode(g, child);
            }
        }
    }

    public static int getTreeDepth(QuadtreeNode node){
        if (node.isLeaf){
            return 1;
        } else {
            int maxDepth = 0;
            for (QuadtreeNode child : node.children){
                int childDepth = getTreeDepth(child);
                if (childDepth > maxDepth){
                    maxDepth = childDepth;
                }
            }
            return maxDepth + 1;
        }
    }

    public static int countNodes(QuadtreeNode node){
        if (node.isLeaf){
            return 1;
        }
    }

```

```

        } else {
            int count = 1;
            for (QuadtreeNode child : node.children) {
                count += countNodes(child);
            }
            return count;
        }
    }
}

import java.awt.Color;
import java.awt.Graphics2D;
import java.awt.image.BufferedImage;
import java.awt.image.RenderedImage;
import java.io.ByteArrayOutputStream;
import java.io.File;
import java.io.IOException;
import java.util.Iterator;
import javax.imageio.IIOImage;
import javax.imageio.ImageIO;
import javax.imageio.ImageTypeSpecifier;
import javax.imageio.ImageWriteParam;
import javax.imageio.ImageWriter;
import javax.imageio.metadata.IIOMetadata;
import javax.imageio.metadata.IIOMetadataNode;
import javax.imageio.stream.FileImageOutputStream;
import javax.imageio.stream.ImageOutputStream;

public class Bonus {
    public static class CompressionUtils{
        public static long getCompressedSize(int[][][] pixelMatrix, int
imageWidth, int imageHeight, double threshold, int minBlockSize, int
errorMethod, String formatName){
            QuadtreeNode root =
QuadtreeCompressor.buildQuadtree(pixelMatrix, 0, 0, imageWidth,
imageHeight, threshold, minBlockSize, errorMethod);
            BufferedImage compressedImage =
QuadtreeCompressor.reconstructImage(root, imageWidth, imageHeight);
            try(ByteArrayOutputStream baos = new ByteArrayOutputStream()){
                ImageIO.write(compressedImage, formatName, baos);
            }
        }
    }
}

```

```

        baos.flush();
        return baos.size();
    } catch (IOException e) {
        e.printStackTrace();
        return Long.MAX_VALUE;
    }
}

public static double adjustThreshold(int[][][] pixelMatrix, int
imageWidth, int imageHeight, double originalSize, int minBlockSize, int
errorMethod, String formatName, double targetCompression) {
    double lower = 0.1;
    double upper = 1000.0;
    double bestThreshold = lower;
    double epsilon = 0.1;
    int maxIterations = 30;
    for (int iter = 0; iter < maxIterations; iter++) {
        double mid = (lower + upper) / 2.0;
        long compSize = getCompressedSize(pixelMatrix, imageWidth,
imageHeight, mid, minBlockSize, errorMethod, formatName);
        double compPerc = 1 - ((double) compSize / originalSize);
        if (Math.abs(compPerc - targetCompression) < 0.02) {
            bestThreshold = mid;
            break;
        } else if (compPerc < targetCompression) {
            lower = mid;
        } else {
            upper = mid;
        }
        bestThreshold = mid;
        if (upper - lower < epsilon) {
            break;
        }
    }

    System.out.println("Threshold disesuaikan menjadi: " +
bestThreshold);
    return bestThreshold;
}

```



```

        public static double computeSSIM(BufferedImage original,
        BufferedImage compressed) {
            if (original.getWidth() != compressed.getWidth() ||
            original.getHeight() != compressed.getHeight()) {
                throw new IllegalArgumentException("Gambar harus memiliki
            dimensi yang sama");
            }
            int width = original.getWidth();
            int height = original.getHeight();
            double L = 255;
            double C1 = Math.pow(0.01 * L, 2);
            double C2 = Math.pow(0.03 * L, 2);
            double[] muOrig = new double[3];
            double[] muComp = new double[3];
            double[] sigmaOrig2 = new double[3];
            double[] sigmaComp2 = new double[3];
            double[] sigmaOrigComp = new double[3];
            for (int y = 0; y < height; y++) {
                for (int x = 0; x < width; x++) {
                    Color cOrig = new Color(original.getRGB(x,y));
                    Color cComp = new Color(compressed.getRGB(x,y));
                    muOrig[0] += cOrig.getRed();
                    muOrig[1] += cOrig.getGreen();
                    muOrig[2] += cOrig.getBlue();
                    muComp[0] += cComp.getRed();
                    muComp[1] += cComp.getGreen();
                    muComp[2] += cComp.getBlue();
                }
            }
            int N = width * height;
            for (int c = 0; c < 3; c++) {
                muOrig[c] /= N;
                muComp[c] /= N;
            }
            for (int y = 0; y < height; y++) {
                for (int x = 0; x < width; x++) {
                    Color cOrig = new Color(original.getRGB(x,y));
                    Color cComp = new Color(compressed.getRGB(x,y));
                    for (int c = 0; c < 3; c++) {

```

```

        int origVal = (c == 0 ? cOrig.getRed() : (c == 1 ?
cOrig.getGreen() : cOrig.getBlue()));
        int compVal = (c == 0 ? cComp.getRed() : (c == 1 ?
cComp.getGreen() : cComp.getBlue()));
        sigmaOrig2[c] += Math.pow(origVal - muOrig[c], 2);
        sigmaComp2[c] += Math.pow(compVal - muComp[c], 2);
        sigmaOrigComp[c] += (origVal - muOrig[c]) *
(compVal - muComp[c]);
    }
}
}
for (int c = 0; c < 3; c++){
    sigmaOrig2[c] /= (N-1);
    sigmaComp2[c] /= (N-1);
    sigmaOrigComp[c] /= (N-1);
}
double ssimTotal = 0;
for (int c = 0; c < 3; c++){
    double ssim = ((2 * muOrig[c] * muComp[c] + C1) * (2 *
sigmaOrigComp[c] + C2))
/ ((muOrig[c] * muOrig[c] + muComp[c] * muComp[c] + C1)
* (sigmaOrig2[c] + sigmaComp2[c] + C2));
    ssimTotal += ssim;
}
return ssimTotal / 3.0;
}
}

public static void createCompressionGIFByLevel(QuadtreeNode root,
String outputGifPath, int imageWidth, int imageHeight) {
    try {
        int maxDepth = QuadtreeCompressor.getTreeDepth(root);

        ImageOutputStream output = new FileImageOutputStream(new
File(outputGifPath));

        int delay = 500; // delay 500 ms per frame
        GifSequenceWriter gifWriter = new GifSequenceWriter(output,
BufferedImage.TYPE_INT_RGB, delay, true);

        for (int level = 1; level <= maxDepth; level++) {

```

```

        BufferedImage frame = new BufferedImage(imageWidth,
imageHeight, BufferedImage.TYPE_INT_RGB);

        Graphics2D g = frame.createGraphics();
        g.setColor(Color.WHITE);
        g.fillRect(0, 0, imageWidth, imageHeight);

        drawQuadtreeAtLevel(g, root, 1, level);

        g.dispose();
        gifWriter.writeToSequence(frame);
    }

    gifWriter.close();
    output.close();
    System.out.println("GIF proses kompresi berhasil disimpan di:
" + outputGifPath);
    } catch (Exception e) {
        e.printStackTrace();
    }
}

public static void drawQuadtreeAtLevel(Graphics2D g, QuadtreeNode
node, int currentLevel, int targetLevel) {
    if (node == null) return;

    if (node.isLeaf || currentLevel >= targetLevel) {
        Color avgColor = new Color(node.avgR, node.avgG, node.avgB);
        g.setColor(avgColor);
        g.fillRect(node.startX, node.startY, node.width, node.height);
    } else {
        if (node.children != null) {
            for (QuadtreeNode child : node.children) {
                drawQuadtreeAtLevel(g, child, currentLevel + 1,
targetLevel);
            }
        }
    }
}

public static class GifSequenceWriter {

```

```

private ImageWriter gifWriter;
private ImageWriteParam imageWriteParam;
private IIOMetadata imageMetaData;

    public GifSequenceWriter(ImageOutputStream outputStream, int
imageType, int timeBetweenFramesMS, boolean loopContinuously) throws
IOException {
    gifWriter = getWriter();
    imageWriteParam = gifWriter.getDefaultWriteParam();
        ImageTypeSpecifier imageTypeSpecifier =
ImageTypeSpecifier.createFromBufferedImageType(imageType);
        imageMetaData =
gifWriter.getDefaultImageMetadata(imageTypeSpecifier, imageWriteParam);

        String metaFormatName =
imageMetaData.getNativeMetadataFormatName();
        IIOMetadataNode root = (IIOMetadataNode)
imageMetaData.getAsTree(metaFormatName);

        IIOMetadataNode graphicsControlExtensionNode = getNode(root,
"GraphicControlExtension");
        graphicsControlExtensionNode.setAttribute("delayTime",
Integer.toString(timeBetweenFramesMS / 10));
        graphicsControlExtensionNode.setAttribute("disposalMethod",
"none");
        graphicsControlExtensionNode.setAttribute("userInputFlag",
"FALSE");

graphicsControlExtensionNode.setAttribute("transparentColorFlag",
"FALSE");

graphicsControlExtensionNode.setAttribute("transparentColorIndex", "0");

        IIOMetadataNode appExtensionsNode = getNode(root,
"ApplicationExtensions");
        IIOMetadataNode appExtensionNode = new
IIOMetadataNode("ApplicationExtension");
        appExtensionNode.setAttribute("applicationID", "NETSCAPE");
        appExtensionNode.setAttribute("authenticationCode", "2.0");

```

```

        int loop = loopContinuously ? 0 : 1;
        appExtensionNode.setUserObject(new byte[]{0x1, (byte) (loop &
0xFF), (byte) ((loop >> 8) & 0xFF)});
        appExtensionsNode.appendChild(appExtensionNode);

        imageMetaData.setFromTree(metaFormatName, root);

        gifWriter.setOutput(outputStream);
        gifWriter.prepareWriteSequence(null);
    }

    public void writeToSequence(RenderedImage img) throws IOException
    {
        gifWriter.writeToSequence(new IIOMImage(img, null,
imageMetaData), imageWriteParam);
    }

    public void close() throws IOException {
        gifWriter.endWriteSequence();
    }

    private static ImageWriter getWriter() throws IOException {
        Iterator<ImageWriter> iter =
ImageIO.getImageWritersBySuffix("gif");
        if (!iter.hasNext()){
            throw new IOException("Tidak ada Image Writer untuk format
GIF");
        } else {
            return iter.next();
        }
    }

    private static IIOMetadataNode getNode(IIOMetadataNode rootNode,
String nodeName) {
        int nNodes = rootNode.getLength();
        for (int i = 0; i < nNodes; i++){
            if
(rootNode.item(i).getNodeName().equalsIgnoreCase(nodeName)) {
                return (IIOMetadataNode) rootNode.item(i);
            }
        }
    }

```


```

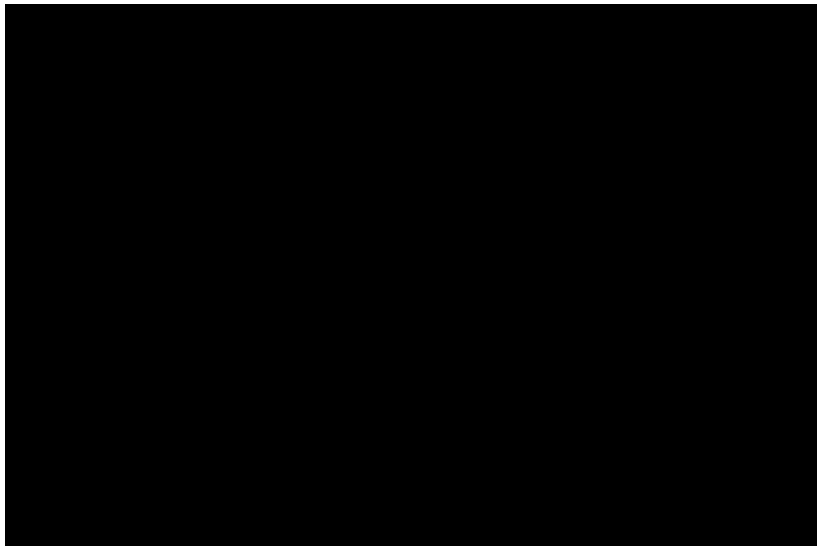
    }

    IIOMetadataNode node = new IIOMetadataNode(nodeName);
    rootNode.appendChild(node);
    return node;
}
}
}

```

## Test Case

No	Input	Output
1	 <p>Gambar 1. Input <i>Gambar1.JPG</i></p>	<pre> Masukkan nama file gambar yang akan dikompresi: Gambar1.JPG Nama File: test/Gambar1.JPG Gambar berhasil dimuat dengan ukuran: 1000x668 Pilih metode perhitungan variansi 1.Variance 2.Mean Absolute Deviation (MAD) 3.Max Pixel Difference 4.Entropy Masukkan pilihan (nomor): 1 Masukkan nilai ambang batas (threshold): 0.01 Masukkan ukuran blok minimum: 4 Masukkan target persentase kompresi (nilai dari 0 sampai 1): 0 Masukkan nama file untuk gambar hasil kompresi: Output1.JPG Masukkan nama file untuk file GIF (tekan Enter untuk melewati): GifOutput1.gif Quadtrees telah berhasil dibangun Gambar hasil kompresi berhasil disimpan di: test/Output1.JPG Waktu eksekusi: 533 Ukuran gambar sebelum kompresi: 166698 Ukuran gambar setelah kompresi: 85886 Persentase kompresi: 48.47808611980947 Kedalaman pohon: 8 Banyak simpul pada pohon: 21813 Nilai SSIM: 0.9317 GIF proses kompresi berhasil disimpan di: test/GifOutput1.gif </pre> <p>Gambar 2. Output</p>



Gambar3. *GifOutput1*



Gambar4. Output *Output1.JPG*

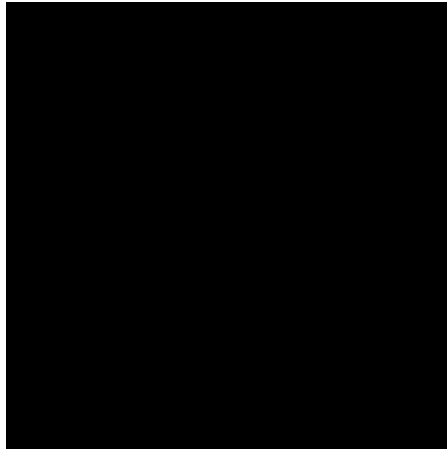
2



Gambar 5. Input  
*Gambar2.PNG*

```
Masukkan nama file gambar yang akan dikompresi: Gambar2.PNG
Nama File: test/Gambar2.PNG
Gambar berhasil dimuat dengan ukuran: 225x225
Pilih metode perhitungan variansi
1.Variance
2.Mean Absolute Deviation (MAD)
3.Max Pixel Difference
4.Entropy
Masukkan pilihan (nomor): 2
Masukkan nilai ambang batas (threshold): 0.1
Masukkan ukuran blok minimum: 3
Masukkan target persentase kompresi (nilai dari 0 sampai 1): 0
Masukkan nama file untuk gambar hasil kompresi: Output2.PNG
Masukkan nama file untuk file GIF (tekan Enter untuk melewati): GifOutput2.gif
Quadtree telah berhasil dibangun
Gambar hasil kompresi berhasil disimpan di: test/Output2.PNG
Waktu eksekusi: 98
Ukuran gambar sebelum kompresi: 12676
Ukuran gambar setelah kompresi: 8965
Persentase kompresi: 29.27579678131903
Kedalaman pohon: 7
Banyak simpul pada pohon: 3709
Nilai SSIM: 0.8676
GIF proses kompresi berhasil disimpan di: test/GifOutput2.gif
```

Gambar 6. Output



Gambar 7. Gif *GifOutput2.gif*  
Gambar 8. Output *Output2.PNG*



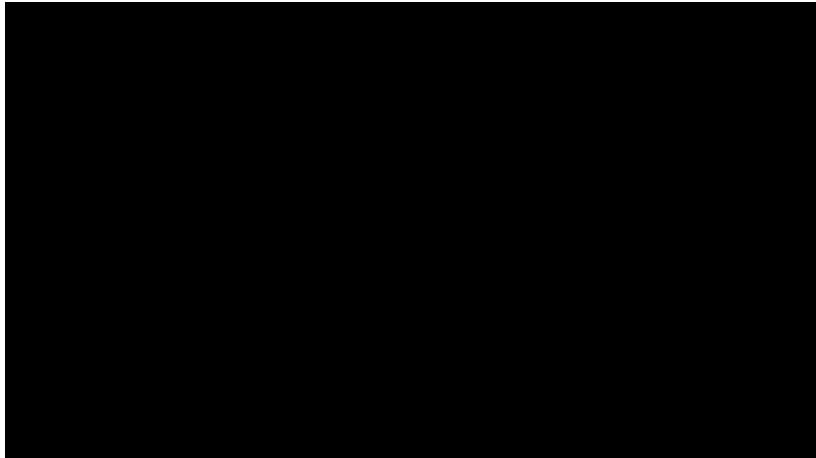
3



Gambar 9. Input  
*Gambar3.JPEG*

```
Masukkan nama file gambar yang akan dikompresi: Gambar3.JPEG
Nama File: test/Gambar3.JPEG
Gambar berhasil dimuat dengan ukuran: 600x338
Pilih metode perhitungan variansi
1.Variance
2.Mean Absolute Deviation (MAD)
3.Max Pixel Difference
4.Entropy
Masukkan pilihan (nomor): 3
Masukkan nilai ambang batas (threshold): 0.01
Masukkan ukuran blok minimum: 5
Masukkan target persentase kompresi (nilai dari 0 sampai 1): 0
Masukkan nama file untuk gambar hasil kompresi: Output3.JPEG
Masukkan nama file untuk file GIF (tekan Enter untuk melewati): GifOutput3.gif
Quadtree telah berhasil dibangun
Gambar hasil kompresi berhasil disimpan di: test/Output3.JPEG
Waktu eksekusi: 139
Ukuran gambar sebelum kompresi: 65394
Ukuran gambar setelah kompresi: 26459
Persentase kompresi: 59.539101446615895
Kedalaman pohon: 7
Banyak simpul pada pohon: 4669
Nilai SSIM: 0.9526
GIF proses kompresi berhasil disimpan di: test/GifOutput3.gif
```

Gambar 10. Output



Gambar 11. Gif *GifOutput3.gif*  
Gambar 12. Output *Output2.JPEG*

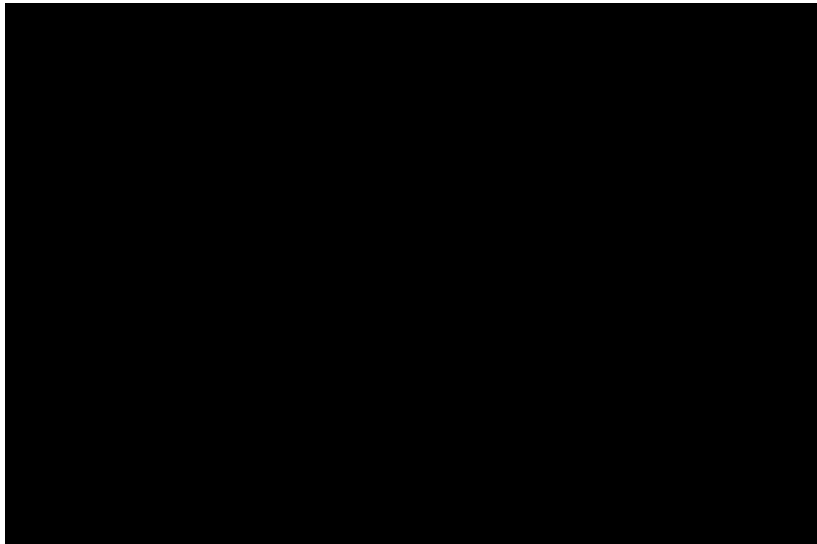
4







Gambar 13. Input  
*Gambar1.JPG*

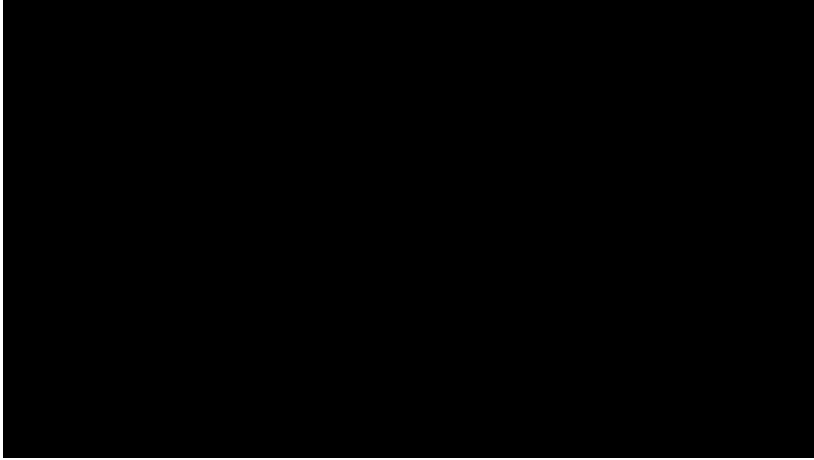


```
Masukkan nama file gambar yang akan dikompresi: Gambar1.JPG
Nama File: test/Gambar1.JPG
Gambar berhasil dimuat dengan ukuran: 1000x668
Pilih metode perhitungan variansi
1.Variance
2.Mean Absolute Deviation (MAD)
3.Max Pixel Difference
4.Entropy
Masukkan pilihan (nomor): 3
Masukkan nilai ambang batas (threshold): 0.1
Masukkan ukuran blok minimum: 4
Masukkan target persentase kompresi (nilai dari 0 sampai 1): 0.8
Masukkan nama file untuk gambar hasil kompresi: Output4.JPG
Masukkan nama file untuk file GIF (tekan Enter untuk melewati): GifOutput4.gif
Threshold disesuaikan menjadi: 234.45156250000002
Quadtree telah berhasil dibangun
Gambar hasil kompresi berhasil disimpan di: test/Output4.JPG
Waktu eksekusi: 1167
Ukuran gambar sebelum kompresi: 166698
Ukuran gambar setelah kompresi: 35126
Persentase kompresi: 78.928361468044
Kedalaman pohon: 8
Banyak simpul pada pohon: 905
Nilai SSIM: 0.7719
GIF proses kompresi berhasil disimpan di: test/GifOutput4.gif
```

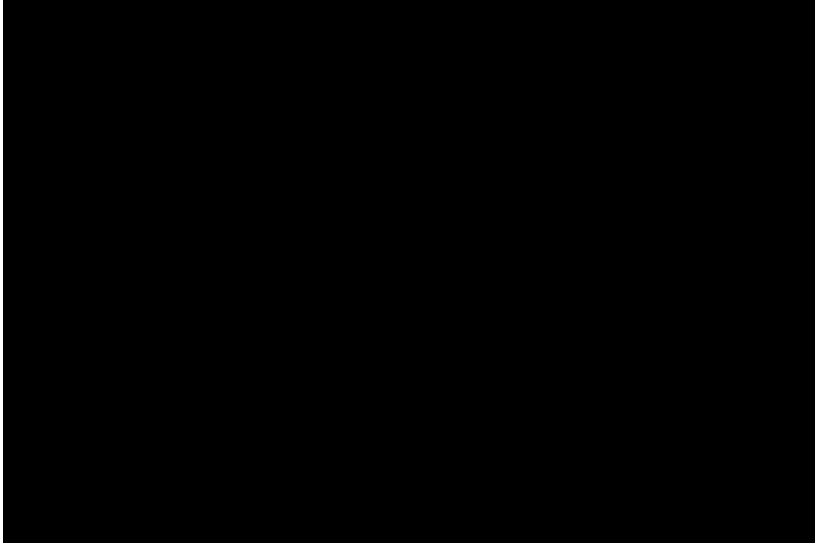

Gambar 14. Output



		 <p>Gambar 15. Gif <i>GifOutput2.gif</i>          Gambar 16. Output <i>Output2.JPG</i></p>
5	 <p>Gambar 17. Input  <i>Gambar2.PNG</i></p>	<pre> Masukkan nama file gambar yang akan dikompresi: Gambar2.PNG Nama File: test/Gambar2.PNG Gambar berhasil dimuat dengan ukuran: 225x225 Pilih metode perhitungan variansi 1.Variance 2.Mean Absolute Deviation (MAD) 3.Max Pixel Difference 4.Entropy Masukkan pilihan (nomor): 1 Masukkan nilai ambang batas (threshold): 0.1 Masukkan ukuran blok minimum: 4 Masukkan target persentase kompresi (nilai dari 0 sampai 1): 0.75 Masukkan nama file untuk gambar hasil kompresi: Output5.PNG Masukkan nama file untuk file GIF (tekan Enter untuk melewati): GifOutput5.gif Threshold disesuaikan menjadi: 999.9389709472656 Quadtree telah berhasil dibangun Gambar hasil kompresi berhasil disimpan di: test/Output5.PNG Waktu eksekusi: 420 Ukuran gambar sebelum kompresi: 12676 Ukuran gambar setelah kompresi: 4789 Persentase kompresi: 62.219943199747554 Kedalaman pohon: 6 Banyak simpul pada pohon: 1061 Nilai SSIM: 0.7558 GIF proses kompresi berhasil disimpan di: test/GifOutput5.gif </pre> <p>Gambar 18. Output</p>

		  <p>Gambar 19. Gif <i>GifOutput2.gif</i>          Gambar 20. Output <i>Output2.PNG</i></p>
6	 <p>Gambar 11. Input  <i>Gambar21.JPEG</i></p>	<pre> Masukkan nama file gambar yang akan dikompresi: Gambar3.JPEG Nama File: test/Gambar3.JPEG Gambar berhasil dimuat dengan ukuran: 600x338 Pilih metode perhitungan variansi 1.Variance 2.Mean Absolute Deviation (MAD) 3.Max Pixel Difference 4.Entropy Masukkan pilihan (nomor): 2 Masukkan nilai ambang batas (threshold): 0.1 Masukkan ukuran blok minimum: 4 Masukkan target persentase kompresi (nilai dari 0 sampai 1): 0.7 Masukkan nama file untuk gambar hasil kompresi: Output6.JPEG Masukkan nama file untuk file GIF (tekan Enter untuk melewati): GifOutput6.gif Threshold disesuaikan menjadi: 23.53515625 Quadtree telah berhasil dibangun Gambar hasil kompresi berhasil disimpan di: test/Output6.JPEG Waktu eksekusi: 435 Ukuran gambar sebelum kompresi: 65394 Ukuran gambar setelah kompresi: 20191 Persentase kompresi: 69.12407866165091 Kedalaman pohon: 7 Banyak simpul pada pohon: 1525 Nilai SSIM: 0.9346 GIF proses kompresi berhasil disimpan di: test/GifOutput6.gif </pre> <p>Gambar 22. Output</p>

		  <p>Gambar 23. Gif <i>GifOutput2.gif</i>          Gambar 24. Output <i>Output2.JPEG</i></p>
7	 <p>Gambar 25. Input  <i>Gambar1.JPG</i></p>	<pre> Masukkan nama file gambar yang akan dikompresi: Gambar1.JPG Nama File: test/Gambar1.JPG Gambar berhasil dimuat dengan ukuran: 1000x668 Pilih metode perhitungan variansi 1.Variance 2.Mean Absolute Deviation (MAD) 3.Max Pixel Difference 4.Entropy Masukkan pilihan (nomor): 4 Masukkan nilai ambang batas (threshold): 0.01 Masukkan ukuran blok minimum: 4 Masukkan target persentase kompresi (nilai dari 0 sampai 1): 0.5 Masukkan nama file untuk gambar hasil kompresi: Output7.JPG Masukkan nama file untuk file GIF (tekan Enter untuk melewati): GifOutput7.gif Threshold disesuaikan menjadi: 4.005859375 Quadtree telah berhasil dibangun Gambar hasil kompresi berhasil disimpan di: test/Output7.JPG Waktu eksekusi: 1113 Ukuran gambar sebelum kompresi: 166698 Ukuran gambar setelah kompresi: 85127 Persentase kompresi: 48.93340052070211 Kedalaman pohon: 8 Banyak simpul pada pohon: 17609 Nilai SSIM: 0.9306 GIF proses kompresi berhasil disimpan di: test/GifOutput7.gif PS C:\Users\HP\OneDrive\Documents\Stima\Tucil2&gt; </pre>

		<p>Gambar 26. Output</p>   <p>Gambar 27. Gif <i>GifOutput2.gif</i>          Gambar 28. Output <i>Output2.JPG</i></p>
--	--	---

## Hasil Analisis

Algoritma bekerja dengan membagi gambar secara rekursif menjadi empat sub-blok (*divide*) berdasarkan perhitungan error (*conquer*). Setiap blok yang error-nya melebihi threshold dan masih memenuhi syarat ukuran dibagi lagi. Proses berlanjut hingga terjadi kondisi basis (blok homogen atau ukuran blok sudah mencapai batas minimum). Seluruh hasil rekursif digabung dalam struktur quadtree yang kemudian dikonstruksi menjadi gambar terkompresi (*combine*). Waktu eksekusi, ukuran file asli dan gambar kompresi, persentase pengurangan ukuran, kedalaman pohon, dan jumlah simpul dihitung untuk mengevaluasi efektivitas algoritma.

Misalkan  $n$  adalah jumlah piksel dalam gambar (dengan  $n = W \cdot H$ ) dengan  $W \cdot H$  adalah ukuran gambar. Setiap pemanggilan rekursif dilakukan pemeriksaan satu blok gambar

dengan menghitung error dan dilakukan loop ganda (sesuai ukuran gambar yaitu  $n$ ). Maka didapat waktu yang diperlukan untuk memproses blok adalah  $O(n)$  dengan  $c$  dianggap suatu faktor tetap. Setelah memproses satu blok, jika syarat pembagian dipenuhi maka blok tersebut dibagi menjadi 4 sub-blok. Setiap sub-blok memiliki jumlah piksel  $n/4$ . Waktu yang dibutuhkan untuk setiap sub-blok adalah  $T(\frac{n}{4})$  dan karena ada 4 sub-blok maka didapat  $4T(\frac{n}{4})$ . Setelah dilakukan *Divide and Conquer* didapat total waktu  $T(n) = 4T(\frac{n}{4}) + cn$  dengan  $c > 0$ .

Teorema master dapat digunakan untuk menentukan notasi asimtotik kompleksitas waktu yang berbentuk relasi rekurens dengan mudah tanpa menyelesaikannya secara iteratif. Misalkan  $T(n)$  adalah fungsi monoton naik yang memenuhi relasi rekurens  $T(n) = aT(\frac{n}{b}) + cn^d$  yang dalam hal ini  $n = b^k$ ,  $k = 1, 2, \dots$ ,  $a \geq 1$ ,  $b \geq 2$ ,  $c \geq 0$ , dan  $d \geq 0$ , maka  $T(n) = O(n^d)$  jika  $a < b^d$ ,  $T(n) = O(n^d \log n)$  jika  $a = b^d$ , dan  $T(n) = O(n^{\log_b a})$  jika  $a > b^d$ .

Dari total waktu didapat  $a = 4$ ,  $b = 4$ ,  $d = 1$ . Maka didapat  $a = 4 = 4^1 = 4 = b^d$ . Maka berdasarkan teorema master kompleksitas waktunya adalah  $O(n^1 \log n) = O(n \log n)$ .

### Penjelasan Bonus

- Penyesuaian threshold otomatis  
User dapat menentukan target persentase kompresi. Ketika target diaktifkan (diberi nilai lebih dari 0 kurang dari sama dengan 1), algoritma secara dinamis menyesuaikan nilai threshold menggunakan metode pencarian bisection. Fungsi `adjustThreshold` melakukan iterasi nilai threshold sehingga ukuran gambar terkompresi mendekati target yang ditentukan.
- Penghitungan Structural Similarity Index (SSIM)  
SSIM merupakan angka yang dapat merepresentasikan kualitas hasil kompresi secara struktural. Perhitungan nilai SSIM dilakukan dengan memproses setiap kanal Red, Green, dan Blue dari gambar asli dan gambar terkompresi, lalu menggabungkan nilainya dengan rata-rata (memilih setiap bobot  $w$  rata menjadi  $\frac{1}{3}$ ). Dengan asumsi gambar berformat 24-bit (8-bit masing-masing kanal) dan menggunakan  $L=255$  sebagai nilai maksimum, implementasi memberikan gambaran objektif mengenai seberapa mirip kedua gambar dari segi struktur.
- Pembuatan GIF  
Metode yang diimplementasikan (`createCompressionGIFByLevel`) menggambar ulang gambar secara global berdasarkan level pembagian (frame pertama 1 blok, frame kedua 4 blok, dan seterusnya). Hasil yang didapat ialah animasi berekstensi .gif.

### Pranala Repository

[https://github.com/GhaisanZP/Tucil2\\_10122078](https://github.com/GhaisanZP/Tucil2_10122078)

## Lampiran

Poin	Ya	Tidak
1. Program berhasil dikompilasi tanpa kesalahan	✓	
2. Program berhasil dijalankan	✓	
3. Program berhasil melakukan kompresi gambar sesuai parameter yang ditentukan	✓	
4. Mengimplementasi seluruh metode perhitungan error wajib	✓	
5. <b>[Bonus]</b> Implementasi persentase kompresi sebagai parameter tambahan	✓	
6. <b>[Bonus]</b> Implementasi Structural Similarity Index (SSIM) sebagai metode pengukuran error	✓	
7. <b>[Bonus]</b> Output berupa GIF Visualisasi Proses pembentukan Quadtree dalam Kompresi Gambar	✓	
8. Program dan laporan dibuat (kelompok) sendiri	✓	