TUGAS KECIL 2 IF2211
Strategi Algoritma

Diampu oleh:

Dr. Ir. Rinaldi Munir, M.T.

Disusun oleh:

Sabilul Huda (13523072)

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG
2025

Aplikasi Algoritma Divide and Conquer

Dalam pembuatan program ini, digunakan algoritma Divide and Conquer sebagai algoritma utama. Sebagai ringkasan, berikut adalah persoalan yang dijawab pada program ini. Diberikan image yang sudah diubah bentuk menjadi *matrix of pixels.* Image tersebut harus dikompresi hingga memenuhi salah satu batasan berikut:

1. Threshold error: User memasukkan input float number 0-100. User juga memasukkan metode pengukuran error (salah satu dari: variance, mean absolute deviation, Max Pixel Difference, entropy, atau structural similarity index). Kompresi akan dilanjutkan selama hasil perhitungan error menggunakan metode yang dipilih lebih dari error yang dimasukkan.
2. Ukuran blok minimum: User memasukkan ukuran blok minimum. Kompresi akan dilanjutkan selama ukuran blok jika dibagi empat masih lebih dari atau sama dengan nilai ukuran blok minimum.

Dalam penyelesaian masalah tersebut, diterapkan algoritma divide and conquer sebagai berikut:

1. Inisialisasi: Baca gambar dan konversi ke matriks piksel RGB.
2. Pembagian Blok:
    - Hitung error (variansi, MAD, dll.) untuk blok saat ini.
    - Jika error > threshold dan ukuran blok > minimum block size, bagi blok menjadi 4 sub-blok.
    - Ulangi rekursif untuk setiap sub-blok.
3. Penghentian:
    - Berhenti jika error ≤ threshold atau ukuran blok tidak bisa dibagi lagi.
4. Normalisasi Warna: Ganti warna blok dengan rata-rata RGB jika blok menjadi leaf.
5. Rekonstruksi Gambar: Bangun gambar terkompresi dari struktur Quadtree.

Pseudocode

```
function buildQuadTree(block, threshold, minSize):
    if error(block) ≤ threshold or block.size ≤ minSize:
        return LeafNode(average_color(block))
    else:
        split block into 4 sub-blocks
        for each sub-block:
            children[i] = buildQuadTree(sub-block, threshold,
minSize)
        return InternalNode(children)
```

Source Code

Source code program secara keseluruhan dibagi menjadi satu program utama dan 4 program pendukung sebagai berikut:

a. Program utama (`main.cpp`)

```
#include <iostream>
#include <string>
#include <chrono>
#include "../header/ImagePixel.hpp"
#include "../header/ErrorCalculator.hpp"
#include "../header/QuadTreeNode.hpp"

int main(int argc, char* argv[]) {
    if (argc < 7) {
        std::cerr << "Usage: " << argv[0] << " <input_image>
<error_method> <threshold> "
                  << "<min_block_size> <compression_percentage>
<output_image> [output_gif]\n";
        return 1;
    }
        std::string inputPath;
        // std::cout << "input path: ";
        // std::cin >> inputPath;
        inputPath = "test/input.png";

        int methodNum;
        std::cout << "error method" << std::endl;
        std::cout << "1. Variance " << std::endl;
        std::cout << "2. Mean Absolute Deviation " <<
std::endl;
        std::cout << "3. Max Pixel Difference " << std::endl;
        std::cout << "4. Entropy " << std::endl;
        std::cout << "Enter method number (1-4): ";
        std::cin >> methodNum;

        double threshold;
        std::cout << "input treshold (0.0-1.0): ";
        std::cin >> threshold;
```

```
        int minBlockSize;
        std::cout << "input minimum block size: ";
        std::cin >> minBlockSize;

        std::string outputPath;
        std::cout << "output path: ";
        std::cin >> outputPath;

        // Convert method number to enum
        ErrorCalculator::ErrorMethod method;
        switch (methodNum) {
            case 1: method = ErrorCalculator::VARIANCE; break;
            case 2: method =
ErrorCalculator::MEAN_ABSOLUTE_DEVIATION; break;
            case 3: method =
ErrorCalculator::MAX_PIXEL_DIFFERENCE; break;
            case 4: method = ErrorCalculator::ENTROPY; break;
            default: throw std::invalid_argument("Invalid error
method");
        }

        // Load the image
        ImagePixel image;
        if (!image.loadImage(inputPath)) {
            std::cerr << "Failed to load image: " << inputPath
<< std::endl;
            return 1;
        }

        // Start timer
        auto start = std::chrono::high_resolution_clock::now();

        // Compress the image
        QuadTreeCompressor compressor(image, method, threshold,
minBlockSize);
        compressor.compress();
```

```
        // Reconstruct the compressed image
        ImagePixel compressedImage;

compressedImage.createFromMatrix(image.getPixelMatrix()); //
Initialize with same dimensions
        compressor.reconstruct(compressedImage);

        // Save the compressed image
        if (!compressedImage.saveImage(outputPath)) {
            std::cerr << "Failed to save compressed image: " <<
outputPath << std::endl;
            return 1;
        }

        // Stop timer
        auto end = std::chrono::high_resolution_clock::now();
        auto duration =
std::chrono::duration_cast<std::chrono::milliseconds>(end -
start);

        // Output results
        std::cout << "Execution time: " << duration.count() <<
" ms\n";
        std::cout << "Tree depth: " <<
compressor.getTreeDepth() << "\n";
        std::cout << "Node count: " <<
compressor.getNodeCount() << "\n";

        // TODO: Calculate and output compression percentage,
original and compressed sizes

    } catch (const std::exception& e) {
        std::cerr << "Error: " << e.what() << std::endl;
        return 1;
    }

    return 0;
}
```

b. Program `ErrorCalculator.hpp`

```
#ifndef ERROR_CALCULATOR_H
#define ERROR_CALCULATOR_H

#include <vector>
#include <cmath>
#include <algorithm>
#include <map>
#include <numeric>
#include "ImagePixel.hpp"
class ErrorCalculator {
public:
    enum ErrorMethod {
        VARIANCE = 1,
        MEAN_ABSOLUTE_DEVIATION = 2,
        MAX_PIXEL_DIFFERENCE = 3,
        ENTROPY = 4,
        SSIM = 5
    };
    static double calculateError(ErrorMethod method, const
std::vector<std::vector<Pixel>>& block, double& rValue, double&
gValue, double& bValue);
private:
    static double calculateVariance(const
std::vector<std::vector<Pixel>>& block, double& rMean, double&
gMean, double& bMean);
    static double calculateMAD(const
std::vector<std::vector<Pixel>>& block, double& rMean, double&
gMean, double& bMean);
    static double calculateMaxDiff(const
std::vector<std::vector<Pixel>>& block, double& rMean, double&
gMean, double& bMean);
    static double calculateEntropy(const
std::vector<std::vector<Pixel>>& block, double& rMean, double&
gMean, double& bMean);
    static double calculateSSIM(const
std::vector<std::vector<Pixel>>& block, double& rMean, double&
gMean, double& bMean);
    static void calculateMeans(const
std::vector<std::vector<Pixel>>& block, double& rMean, double&
gMean, double& bMean);
```

```
    static void calculateHistograms(const
std::vector<std::vector<Pixel>>& block, std::map<uint8_t, int>&
rHist, std::map<uint8_t, int>& gHist,std::map<uint8_t, int>&
bHist);
};

#endif
```

c. Program `ErrorCalculator.cpp`

```
#include "../header/ErrorCalculator.hpp"

double ErrorCalculator::calculateError(ErrorMethod method,
                               const
std::vector<std::vector<Pixel>>& block,
                               double& rValue, double& gValue,
double& bValue) {
    switch (method) {
        case VARIANCE: return calculateVariance(block, rValue,
gValue, bValue);
        case MEAN_ABSOLUTE_DEVIATION: return
calculateMAD(block, rValue, gValue, bValue);
        case MAX_PIXEL_DIFFERENCE: return
calculateMaxDiff(block, rValue, gValue, bValue);
        case ENTROPY: return calculateEntropy(block, rValue,
gValue, bValue);
        default: throw std::invalid_argument("Invalid error
method");
    }
}

double ErrorCalculator::calculateVariance(const
std::vector<std::vector<Pixel>>& block,
                                  double& rMean, double& gMean,
double& bMean) {
    calculateMeans(block, rMean, gMean, bMean);

    double maxVariance = 16256.25;
    double rVar = 0, gVar = 0, bVar = 0;
    int count = 0;
```

```
    for (const auto& row : block) {
        for (const Pixel& p : row) {
            rVar += (p.r - rMean) * (p.r - rMean);
            gVar += (p.g - gMean) * (p.g - gMean);
            bVar += (p.b - bMean) * (p.b - bMean);
            count++;
        }
    }

    rVar /= count;
    gVar /= count;
    bVar /= count;

    return (rVar + gVar + bVar) / (3.0 * maxVariance);
}

double ErrorCalculator::calculateMAD(const
std::vector<std::vector<Pixel>>& block,
                         double& rMean, double& gMean, double&
bMean) {
    calculateMeans(block, rMean, gMean, bMean);

    double maxMAD = 127.5;
    double rMad = 0, gMad = 0, bMad = 0;
    int count = 0;

    for (const auto& row : block) {
        for (const Pixel& p : row) {
            rMad += std::abs(p.r - rMean);
            gMad += std::abs(p.g - gMean);
            bMad += std::abs(p.b - bMean);
            count++;
        }
    }

    rMad /= count;
    gMad /= count;
    bMad /= count;
```

```
    return (rMad + gMad + bMad) / (3.0 * maxMAD);
}

double ErrorCalculator::calculateMaxDiff(const
std::vector<std::vector<Pixel>>& block,
                             double& rMean, double& gMean,
double& bMean) {
    const double diffMax = 255.0;
    if (block.empty() || block[0].empty()) return 0.0;

    uint8_t rMin = block[0][0].r, rMax = block[0][0].r;
    uint8_t gMin = block[0][0].g, gMax = block[0][0].g;
    uint8_t bMin = block[0][0].b, bMax = block[0][0].b;

    for (const auto& row : block) {
        for (const Pixel& p : row) {
            rMin = std::min(rMin, p.r);
            rMax = std::max(rMax, p.r);
            gMin = std::min(gMin, p.g);
            gMax = std::max(gMax, p.g);
            bMin = std::min(bMin, p.b);
            bMax = std::max(bMax, p.b);
        }
    }

    rMean = (rMax + rMin) / 2.0;
    gMean = (gMax + gMin) / 2.0;
    bMean = (bMax + bMin) / 2.0;

    double rDiff = rMax - rMin;
    double gDiff = gMax - gMin;
    double bDiff = bMax - bMin;

    return (rDiff + gDiff + bDiff) / (3.0 * diffMax);
}
```

```
double ErrorCalculator::calculateEntropy(const
std::vector<std::vector<Pixel>>& block, double& rMean, double&
gMean, double& bMean) {
    calculateMeans(block, rMean, gMean, bMean);

    std::map<uint8_t, int> rHist, gHist, bHist;
    calculateHistograms(block, rHist, gHist, bHist);

    double maxEntropy = 8.0;
    int totalPixels = block.size() * block[0].size();
    double rEntropy = 0, gEntropy = 0, bEntropy = 0;

    for (const auto& pair : rHist) {
        double prob = pair.second /
static_cast<double>(totalPixels);
        rEntropy -= prob * log2(prob);
    }

    for (const auto& pair : gHist) {
        double prob = pair.second /
static_cast<double>(totalPixels);
        gEntropy -= prob * log2(prob);
    }

    for (const auto& pair : bHist) {
        double prob = pair.second /
static_cast<double>(totalPixels);
        bEntropy -= prob * log2(prob);
    }

    return (rEntropy + gEntropy + bEntropy) / (3.0 *
maxEntropy);
}

void ErrorCalculator::calculateMeans(const
std::vector<std::vector<Pixel>>& block, double& rMean, double&
gMean, double& bMean) {
    rMean = gMean = bMean = 0.0;
    int count = 0;
```

```
    for (const auto& row : block) {
        for (const Pixel& p : row) {
            rMean += p.r;
            gMean += p.g;
            bMean += p.b;
            count++;
        }
    }

    if (count > 0) {
        rMean /= count;
        gMean /= count;
        bMean /= count;
    }
}

void ErrorCalculator::calculateHistograms(const
std::vector<std::vector<Pixel>>& block, std::map<uint8_t, int>&
rHist, std::map<uint8_t, int>& gHist,std::map<uint8_t, int>&
bHist) {
    rHist.clear();
    gHist.clear();
    bHist.clear();

    for (const auto& row : block) {
        for (const Pixel& p : row) {
            rHist[p.r]++;
            gHist[p.g]++;
            bHist[p.b]++;
        }
    }
}
```

d. Program `ImagePixel.hpp`

```
#ifndef IMAGE_PIXEL_H
#define IMAGE_PIXEL_H

#include <vector>
```

```
#include <string>
#include <cstdint>
#include <stdexcept>

// Forward declarations from stb
extern "C" {
    unsigned char* stbi_load(char const* filename, int* x, int*
y, int* comp, int req_comp);
    void stbi_image_free(void* retval_from_stbi_load);
    int stbi_write_png(char const* filename, int w, int h, int
comp, const void* data, int stride_in_bytes);
    int stbi_write_jpg(char const* filename, int w, int h, int
comp, const void* data, int quality);
}

// Represents a single pixel with RGB channels
struct Pixel {
    uint8_t r, g, b;
    Pixel() : r(0), g(0), b(0) {}
    Pixel(uint8_t red, uint8_t green, uint8_t blue) : r(red),
g(green), b(blue) {}
};

class ImagePixel {
public:
    ImagePixel();
    ~ImagePixel();
    bool loadImage(const std::string& filepath);
    bool saveImage(const std::string& filepath) const;
    int getWidth() const;
    int getHeight() const;
    const std::vector<std::vector<Pixel>>& getPixelMatrix()
const;
    std::vector<std::vector<Pixel>>& getPixelMatrix();
    Pixel getPixel(int x, int y) const;
    void setPixel(int x, int y, const Pixel& pixel);
    void createFromMatrix(const
std::vector<std::vector<Pixel>>& matrix);
```

```
private:
    std::vector<std::vector<Pixel>> pixelMatrix;
    int width;
    int height;
};

#endif
```

e. Program ImagePixel.cpp

```
#include "../header/ImagePixel.hpp"

ImagePixel::ImagePixel() : width(0), height(0) {}
ImagePixel::~ImagePixel() = default;

bool ImagePixel::loadImage(const std::string& filepath) {
    int channels;
    unsigned char* data = stbi_load(filepath.c_str(), &width,
&height, &channels, 3);
    if (!data) {
        return false;
    }

    pixelMatrix.resize(height, std::vector<Pixel>(width));

    for (int y = 0; y < height; y++) {
        for (int x = 0; x < width; x++) {
            int index = (y * width + x) * 3;
            pixelMatrix[y][x] = Pixel(data[index],
data[index+1], data[index+2]);
        }
    }

    stbi_image_free(data);
    return true;
}

bool ImagePixel::saveImage(const std::string& filepath) const {
```

```
    std::vector<unsigned char> data(width * height * 3);

    for (int y = 0; y < height; y++) {
        for (int x = 0; x < width; x++) {
            int index = (y * width + x) * 3;
            data[index] = pixelMatrix[y][x].r;
            data[index+1] = pixelMatrix[y][x].g;
            data[index+2] = pixelMatrix[y][x].b;
        }
    }

    std::string ext =
filepath.substr(filepath.find_last_of(".") + 1);
    if (ext == "png") {
        return stbi_write_png(filepath.c_str(), width, height,
3, data.data(), width * 3);
    } else if (ext == "jpg" || ext == "jpeg") {
        return stbi_write_jpg(filepath.c_str(), width, height,
3, data.data(), 90);
    }

    return false;
}

int ImagePixel::getWidth() const { return width; }
int ImagePixel::getHeight() const { return height; }

const std::vector<std::vector<Pixel>>&
ImagePixel::getPixelMatrix() const { return pixelMatrix; }
std::vector<std::vector<Pixel>>& ImagePixel::getPixelMatrix() {
return pixelMatrix; }

Pixel ImagePixel::getPixel(int x, int y) const {
    if (x < 0 || x >= width || y < 0 || y >= height) {
        throw std::out_of_range("Pixel coordinates out of
range");
    }
    return pixelMatrix[y][x];
```

```
}

void ImagePixel::setPixel(int x, int y, const Pixel& pixel) {
    if (x < 0 || x >= width || y < 0 || y >= height) {
        throw std::out_of_range("Pixel coordinates out of
range");
    }
    pixelMatrix[y][x] = pixel;
}

void ImagePixel::createFromMatrix(const
std::vector<std::vector<Pixel>>& matrix) {
    if (matrix.empty() || matrix[0].empty()) {
        width = height = 0;
        pixelMatrix.clear();
        return;
    }

    height = matrix.size();
    width = matrix[0].size();
    pixelMatrix = matrix;
}
```

f. Program `QuadTreeNode.hpp`

```
#ifndef QUADTREE_COMPRESSOR_H
#define QUADTREE_COMPRESSOR_H

#include "ImagePixel.hpp"
#include "ErrorCalculator.hpp"
#include <memory>
#include <queue>

struct QuadTreeNode {
    int x, y;
    int width, height;
    Pixel averageColor;
    bool isLeaf;
    std::unique_ptr<QuadTreeNode> children[4];
```

```
    QuadTreeNode(int x, int y, int w, int h)
        : x(x), y(y), width(w), height(h), isLeaf(false) {
        for (int i = 0; i < 4; i++) children[i] = nullptr;
    }
};

class QuadTreeCompressor {
public:
    QuadTreeCompressor(ImagePixel& image,
ErrorCalculator::ErrorMethod method, double threshold, int
minBlockSize);
    void compress();
    void reconstruct(ImagePixel& outputImage);
    int getTreeDepth() const;
    int getNodeCount() const;

private:
    ImagePixel& image;
    ErrorCalculator::ErrorMethod method;
    double threshold;
    int minBlockSize;
    std::unique_ptr<QuadTreeNode> root;
    int treeDepth;
    int nodeCount;

    std::unique_ptr<QuadTreeNode> buildQuadTree(int x, int y,
int width, int height, int currentDepth);
    void reconstructImage(QuadTreeNode* node,
std::vector<std::vector<Pixel>>& matrix);
    void getBlock(const ImagePixel& img, int x, int y, int
width, int height, std::vector<std::vector<Pixel>>& block);
};

#endif
```

g. Program `QuadTreeNode.cpp`

```
#include "../header/QuadTreeNode.hpp"
```

```
QuadTreeCompressor::QuadTreeCompressor(ImagePixel& image,
ErrorCalculator::ErrorMethod method, double threshold, int
minBlockSize)
    : image(image), method(method),
      threshold(threshold), minBlockSize(minBlockSize),
      treeDepth(0), nodeCount(0) {}

void QuadTreeCompressor::compress() {
    if (root) {
        root.reset();
        treeDepth = 0;
        nodeCount = 0;
    }

    root = buildQuadTree(0, 0, image.getWidth(),
image.getHeight(), 1);
}

void QuadTreeCompressor::reconstruct(ImagePixel& outputImage) {
    if (!root) return;

    std::vector<std::vector<Pixel>> matrix(image.getHeight(),

std::vector<Pixel>(image.getWidth()));
    reconstructImage(root.get(), matrix);
    outputImage.createFromMatrix(matrix);
}

int QuadTreeCompressor::getTreeDepth() const { return
treeDepth; }
int QuadTreeCompressor::getNodeCount() const { return
nodeCount; }

std::unique_ptr<QuadTreeNode>
QuadTreeCompressor::buildQuadTree(int x, int y, int width, int
height, int currentDepth) {
    auto node = std::make_unique<QuadTreeNode>(x, y, width,
height);
    nodeCount++;
```

```
    treeDepth = std::max(treeDepth, currentDepth);

    // Get the current block
    std::vector<std::vector<Pixel>> block;
    getBlock(image, x, y, width, height, block);

    // Calculate error and mean values
    double rMean, gMean, bMean;
    double error = ErrorCalculator::calculateError(method,
block, rMean, gMean, bMean);

    // Check if we should split
    bool shouldSplit = (error > threshold) &&
                      (width > minBlockSize && height >
minBlockSize) &&
                      (width/2 >= minBlockSize && height/2 >=
minBlockSize);

    if (shouldSplit) {
        // Split into 4 quadrants
        int halfWidth = width / 2;
        int halfHeight = height / 2;

        // Top-left
        node->children[0] = buildQuadTree(x, y, halfWidth,
halfHeight, currentDepth + 1);
        // Top-right
        node->children[1] = buildQuadTree(x + halfWidth, y,
width - halfWidth, halfHeight, currentDepth + 1);
        // Bottom-left
        node->children[2] = buildQuadTree(x, y + halfHeight,
halfWidth, height - halfHeight, currentDepth + 1);
        // Bottom-right
        node->children[3] = buildQuadTree(x + halfWidth, y +
halfHeight,
                                         width - halfWidth,
height - halfHeight, currentDepth + 1);
    } else {
        // Leaf node - store average color
```

```
        node->isLeaf = true;
        node->averageColor = Pixel(static_cast<uint8_t>(rMean),
                                   static_cast<uint8_t>(gMean),
                                   static_cast<uint8_t>(bMean));
    }

    return node;
}

void QuadTreeCompressor::reconstructImage(QuadTreeNode* node,
std::vector<std::vector<Pixel>>& matrix) {
    if (!node) return;

    if (node->isLeaf) {
        // Fill the block with average color
        for (int y = node->y; y < node->y + node->height; y++)
{
            for (int x = node->x; x < node->x + node->width;
x++) {
                if (static_cast<size_t>(y) < matrix.size() &&
static_cast<size_t>(x) < matrix[y].size()){
                    matrix[y][x] = node->averageColor;
                }
            }
        }
    } else {
        // Reconstruct children
        for (int i = 0; i < 4; i++) {
            reconstructImage(node->children[i].get(), matrix);
        }
    }
}

void QuadTreeCompressor::getBlock(const ImagePixel& img, int x,
int y, int width, int height,
             std::vector<std::vector<Pixel>>& block) {
    block.clear();
    block.reserve(height);
```

```
        for (int row = y; row < y + height; row++) {
            if (row >= img.getHeight()) break;

            std::vector<Pixel> rowPixels;
            rowPixels.reserve(width);

            for (int col = x; col < x + width; col++) {
                if (col >= img.getWidth()) break;
                rowPixels.push_back(img.getPixel(col, row));
            }

            block.push_back(rowPixels);
        }
    }
```

h. Program `stb_implementation.cpp`

Kode program ini diambil dari github https://github.com/nothings/stb untuk membantu proses baca dan tulis image file.

```
#define STB_IMAGE_IMPLEMENTATION
#include "../header/stb_image.h"

#define STB_IMAGE_WRITE_IMPLEMENTATION
#include "../header/stb_image_write.h"
```

Testing

No.	Penjelasan Kasus	Input Foto	Output Foto
1.	Metode perhitungan error: 1 Threshold: 0.3 Ukuran blok minimum: 4	Gambar 1	Execution time: 73 ms Tree depth: 8 Node count: 349 Output photo: Gambar 2
2.	Metode perhitungan error: 2 Threshold: 0.3 Ukuran blok minimum: 4	Gambar 1	Execution time: 194 ms Tree depth: 8 Node count: 1277 Output photo: Gambar 3
3.	Metode perhitungan error: 3 Threshold: 0.3 Ukuran blok minimum: 4	Gambar 1	Execution time: 75 ms Tree depth: 8 Node count: 6181 Output photo: Gambar 4
4.	Metode perhitungan error: 4 Threshold: 0.3 Ukuran blok minimum: 4	Gambar 1	Execution time: 464 ms Tree depth: 8 Node count: 20081 Output photo: Gambar 5
5.	Metode perhitungan error: 4 Threshold: 0.9 Ukuran blok minimum: 4	Gambar 1	Execution time: 198 ms Tree depth: 6 Node count: 85 Output photo: Gambar 6
6.	Metode perhitungan error: 4 Threshold: 0.5 Ukuran blok minimum: 4	Gambar 1	Execution time: 435 ms Tree depth: 8 Node count: 12845 Output photo: Gambar 7
7.	Metode perhitungan error: 4 Threshold: 0.1 Ukuran blok minimum: 50	Gambar 1	Execution time: 231 ms Tree depth: 4 Node count: 85 Output photo: Gambar 8

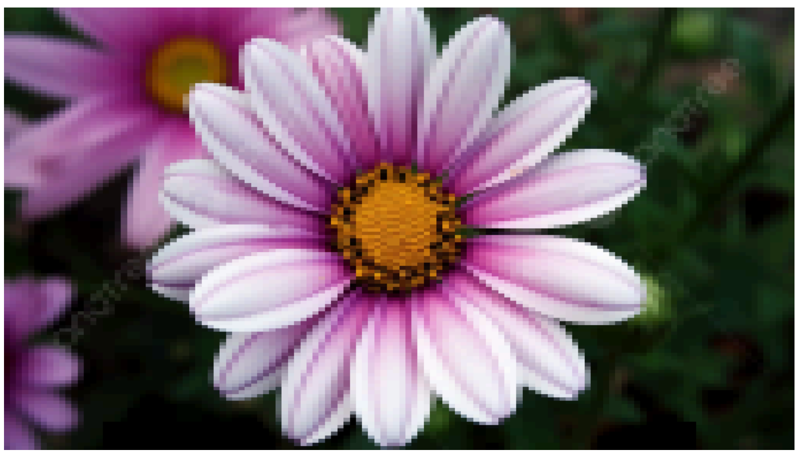
Gambar-gambar yang digunakan:

Gambar 1. Input file png

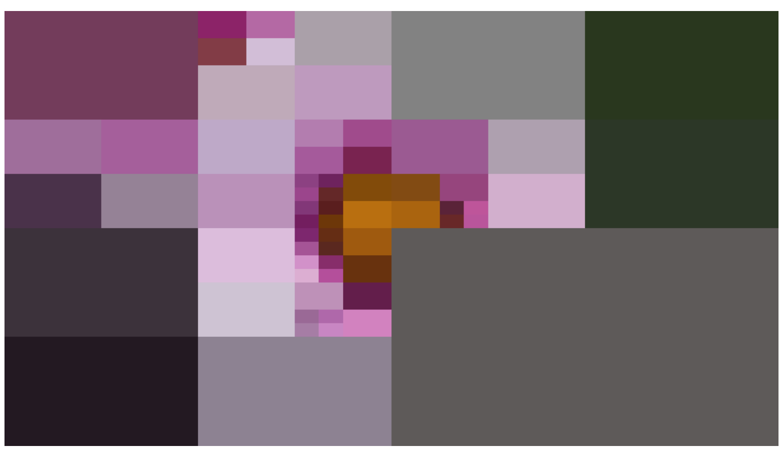
Gambar 2. Output test case 1

Gambar 3. Output test case 2

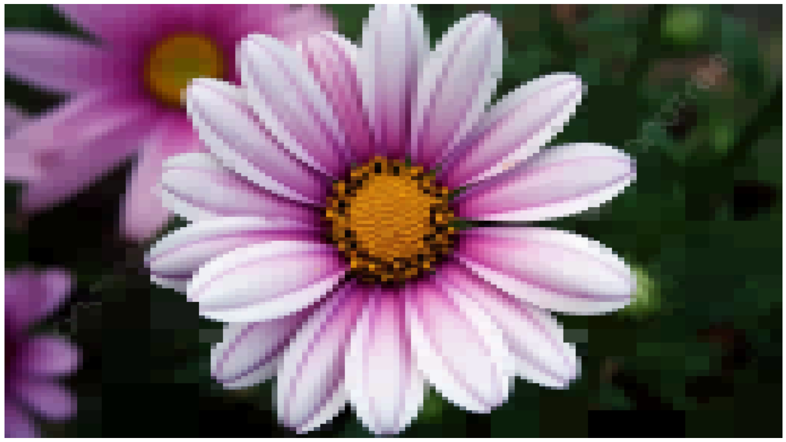
Gambar 4. Output test case 3



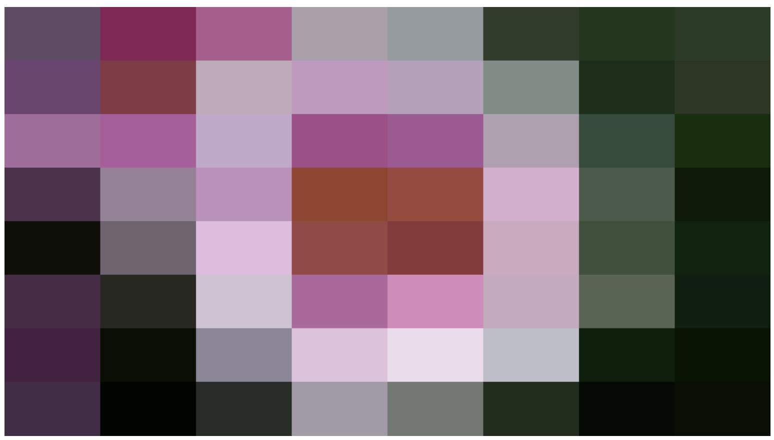
Gambar 5. Output test case 4



Gambar 6. Output test case 5



Gambar 7. Output test case 6



Gambar 8. Output test case 7

Hasil Analisis Percobaan Algoritma Divide and Conquer

Pranala Repositori Kode Program

Berikut adalah pranala ke repository github yang berisi kode program:
github.com/bill2247/Tucil2_1352307