

# BÀI TẬP VỀ NHÀ TUẦN 5

## Xử lý ảnh và video

**Yêu cầu:** *Viết chương trình C thực hiện nén và giải nén khối 8x8 ở trang 12 file Wallace.pdf*

### 1. PHẦN XÂY DỰNG MÃ NGUỒN C

- Khai báo các thư viện có sẵn trong C để nhập xuất giá trị, thư viện các hàm toán học, các hàm xử lý với chuỗi ký tự và khai báo hằng số  $N = 8$  là kích thước ma trận đầu vào.

```
#include <stdio.h>
#include <math.h>
#include <string.h>
#define N 8
```

- Hàm chuẩn hóa giá trị ma trận đầu vào 8x8 và kết quả mã hóa. Mã hóa 8 bit cho mỗi pixel nên giá trị từ 0 – 255, mà nhân với hàm cos vừa dương vừa âm nên trừ cho giá trị trung gian là 128 với điều kiện biến Scale là 1, còn khi Scale là 0 thì cộng 128 lại để trở lại giá trị ban đầu .

```
void scaleValue(int In[N][N], int Out[N][N], int Scale) {
    for (int x = 0; x < N; x++) {
        for (int y = 0; y < N; y++) {
            if (Scale) {
                Out[x][y] = In[x][y] - 128;
            } else {
                Out[x][y] = In[x][y] + 128;
                printf("%d ", Out[x][y]) ;
            }
        }
        printf("\n") ;
    }
}
```

Từ công thức biến đổi DCT và biến đổi ngược DCT được cho trong tài liệu như sau

$$F(u, v) = \frac{1}{4} C(u) C(v) \left[ \sum_{x=0}^7 \sum_{y=0}^7 f(x, y) * \cos \frac{(2x+1)u\pi}{16} \cos \frac{(2y+1)v\pi}{16} \right] \quad (1)$$

$$f(x, y) = \frac{1}{4} \left[ \sum_{u=0}^7 \sum_{v=0}^7 C(u) C(v) F(u, v) * \cos \frac{(2x+1)u\pi}{16} \cos \frac{(2y+1)v\pi}{16} \right] \quad (2)$$

where:  $C(u), C(v) = 1/\sqrt{2}$  for  $u, v = 0$ ;

$C(u), C(v) = 1$  otherwise.

- Hàm biến đổi DCT và xuất giá trị sau khi tính toán.

```
void DCT_trans(int input[N][N], double output[N][N]) {
    double Cu, Cv, F;
    double PI = 3.14159265358979323846;
    for (int u = 0; u < N; u++) {
        for (int v = 0; v < N; v++) {
            if (u == 0) Cu = 1.0 / sqrt(2); else Cu = 1.0;
            if (v == 0) Cv = 1.0 / sqrt(2); else Cv = 1.0;
            F = 0.0;
            for (int x = 0; x < N; x++) {
                for (int y = 0; y < N; y++) {
                    F += input[x][y] * cos(((2 * x + 1) * u * PI) / (2.0 * N)) *
cos(((2 * y + 1) * v * PI) / (2.0 * N));
                }
            }
            output[u][v] = 0.25 * Cu * Cv * F;
            printf("%.4f ", output[u][v]);
        }
        printf("\n");
    }
}
```

- Hàm đọc vào ma trận ảnh sau biến đổi DCT và các giá trị của bảng lượng tử hóa để thực

hiện lượng tử hóa. DCTcoeff chính mà ma trận sau biến đổi DCT, QuanTable là bảng các giá trị lượng tử hóa và Result là ma trận 8x8 các giá trị sau lượng tử hóa.

```
void Quantize(double DCTcoeff[N][N], int QuanTable[N][N], int Result[N][N]) {
    for (int x = 0; x < N; x++) {
        for (int y = 0; y < N; y++) {
            Result[x][y] = (int)(round(DCTcoeff[x][y] / QuanTable[x][y]));
            printf("%d ", Result[x][y]);
        }
        printf("\n");
    }
}
```

- Hàm đọc ma trận sau lượng tử hóa sang dạng vecto 1 chiều theo hướng Zig zag. Do chuyển từ ma trận 8x8 sang 1 chiều nên kích thước ra là N\*N.

```
void Zigzag(int QuantizeMatrix[N][N], int MatrixVal[N * N]) {
    int x = 0, y = 0, index = 0;
    int up = 1; // Điều hướng di chuyển: 1 là lên trên, -1 là xuống dưới

    for (int i = 0; i < N * N; i++) {
        MatrixVal[index] = QuantizeMatrix[x][y];
        index++;
        printf("%d ", QuantizeMatrix[x][y]);

        if (up) {
            if (y == N - 1) {
                x++; // Nếu đã đến cuối cột, chuyển sang hàng tiếp theo
                up = 0; // Đổi hướng
            } else if (x == 0) {
                y++; // Nếu đã đến đầu hàng, chuyển sang cột tiếp theo
                up = 0; // Đổi hướng
            } else {
                x--; y++; // Di chuyển chéo lên trên
            }
        } else {
            if (x == N - 1) {
                y++; // Nếu đã đến cuối hàng, chuyển sang cột tiếp theo
                up = 1; // Đổi hướng
            } else if (y == 0) {
                x++; // Nếu đã đến đầu cột, chuyển sang hàng tiếp theo
                up = 1; // Đổi hướng
            } else {
                x++; y--; // Di chuyển chéo xuống dưới
            }
        }
    }
    printf("\n");
}
```

}

- Các hàm thực hiện mã hóa từ vecto đọc từ ma trận sang sequence symbols. Đầu tiên là hàm con tính toán size cho giá trị biên độ. Sau đó mới là hàm mã hóa, được gọi chung là Baseline Sequential codec.

Trong hàm mã hóa Baseline Sequential (BaselineEncode) chứa các tham số DCpre là giá trị DC khối trước liền kề, input[N\*N] là vecto 1 chiều chứa các giá trị ma trận sau lượng tử hóa qua hàm trên; VLEcode (Variable-Length Entropy Coding) chứa các giá trị của Symbol 1 (Variable-length Code VLC) và Symbol 2 (Variable-length Integer VLI) và VLEsize là kích thước của vecto VLEcode chứa các giá trị này.

Đầu tiên khởi tạo cho biến vị trí lưu giá trị vào vecto VLEcode (VLE\_index) bằng 0, sau đó tính độ chênh lệch giữa 2 giá trị DC của khối đang xét với khối trước nó, gọi là DCdiff, sau đó tính size cho Dcdiff này. Và lưu giá trị DCsize vào vị trí đầu tiên (0) trong VLEcode và lưu DCdiff liền kề nó, vậy là mã hóa xong cho giá trị DC (Size)(Amplitude).

Vị trí 0 của input chứa DC nên với AC chỉ xét từ vị trí 1 trở đi ( $i = 1$  và  $\text{while } i < N*N$ ), nếu gặp giá trị 0 thì cộng dồn biến đếm Zero-run length thêm 1, cho tới khi gặp giá trị khác 0 thì lần lượt gán Zero-run length, size của giá trị đó và chính nó vào VLEcode thành bộ ba biến mã hóa cho Symbol 1 và 2 cho AC là (Zero-run length, Size)(Amplitude) và xét đến hết các giá trị.

```
int SizeCal(int value) {
    if (value == 0) return 0;
    return (int)log2(abs(value)) + 1;
}
//VLE: Variable-Length Entropy Coding
void BaselineEncode(int DCpre, int input[N * N], int *VLEcode, int *VLEsize) {
    int VLE_index = 0;
    int DCdiff = abs(input[0] - DCpre) ;
    int DCSize = SizeCal(DCdiff) ;
    VLEcode[VLE_index++] = DCSize ;
    VLEcode[VLE_index++] = DCdiff ;
    int i = 1;
    int ZeroRunLength = 0;
    while (i < N * N) {
        if (input[i] == 0) {
            ZeroRunLength++; // Tăng độ dài chuỗi số 0
        } else {
            int amp = input[i];
            int size = SizeCal(amp);
            VLEcode[VLE_index++] = ZeroRunLength;
            VLEcode[VLE_index++] = size;
            VLEcode[VLE_index++] = amp;
            ZeroRunLength = 0;
        }
        i++;
    }
}
```

```

    }
    VLEcode[VLE_index++] = 256;
    *VLEsize = VLE_index;
}

void printVLE(int *VLEcode, int VLEsize) {
    printf("(DC Size, DC Amplitude): (%d, %d)\n", VLEcode[0], VLEcode[1]);
    printf("(Zero-run length, Size)(Amplitude): \n");
    for (int i = 2; i < VLEsize; i += 3) {
        if (VLEcode[i] == 256) {
            printf("(0, 0) (EOB)\n");
            break;
        }
        printf("(%d, %d)(%d)\n", VLEcode[i], VLEcode[i + 1], VLEcode[i + 2]);
    }
}
}

```

Sau khi xét hết tất cả giá trị trong input, gán thêm giá trị 256 vào kế tiếp các giá trị đã được gán trước đó coi như là dấu hiệu của EOB (End of Block) do giá trị của pixel không thể đạt 256, tránh xảy ra lỗi. Sau là in ra các giá trị mã hóa, ở đây khi gặp giá trị 256 thì thay vì xuất ra 256 thì ghi ra là (0,0) EOB.

- Các hàm thực hiện mã hóa Huffman, đọc vào vecto VLEcode, kích thước của nó và ghi thành chuỗi ký tự theo mã hóa Huffman.

Đầu tiên là hai hàm mã hóa Symbol 1 (Zero-run length, Size) và mã hóa Symbol 2 (Amplitude). Từ bộ giá trị mã hóa trong tài liệu ở cuối mục 7.3 (cuối trang 12, đầu trang 13) mà thiết lập giá trị trả về tương ứng với bộ giá trị Zero-run length, Size và Amplitude tương ứng và để dễ dàng cho việc giải mã Huffman và so sánh kiểm tra thì thêm dấu cách “ ” sau giá trị mã hóa.

Ở hàm HuffmanEncode là hàm mã hóa chính thực hiện gọi lại hàm mã hóa Symbol 1 và 2 với BitStream là chuỗi các giá trị mã hóa Huffman.

```

const char* Symbol1Code(int zero_run, int size) {
    if (size == 1) {
        if (zero_run == 0) return "00 "; //(0,1)
        if (zero_run == 2) return "1100 "; //(2,1)
    }
    if (zero_run == 1 && size == 2) return "11011 "; //(1,2)
    return "011 "; //(2) của DC
}

const char* Symbol2Code(int amp) {
    if (amp == 2) return "011 ";
    if (amp == 3) return "11 ";
    if (amp == -2) return "01 ";
    if (amp == -1) return "0 ";
}

```

```

}

void HuffmanEncode(int *VLEcode, int VLEsize, char *BitStream) {
    BitStream[0] = '\\0';
    int DCSize = VLEcode[0];
    int DCamp = VLEcode[1];
    const char* dcSize = Symbol1Code(DCSize, DCamp);
    strcat(BitStream, dcSize);
    const char* dcAmp = Symbol2Code(DCAMP);
    strcat(BitStream, dcAmp);

    for (int i = 2; i < VLEsize; i += 3) {
        if (VLEcode[i] == 256) {
            strcat(BitStream, "1010 ");
            break;
        }
        int zeroRunLength = VLEcode[i];
        int size = VLEcode[i + 1];
        int amp = VLEcode[i + 2];
        const char* code = Symbol1Code(zeroRunLength, size);
        strcat(BitStream, code);
        const char* AmpCode = Symbol2Code(amp);
        strcat(BitStream, AmpCode);
    }
    printf("%s", BitStream) ;
}

```

Đầu tiên là gán giá trị kết thúc chuỗi (\0) vào vị trí đầu của chuỗi BitStream để khởi tạo nó là chuỗi rỗng, sau đó gọi các hàm mã hóa Symbol 1 và 2 và dùng lệnh strcat() để ghép chuỗi lần lượt theo thứ tự, đồng thời cũng xét điều kiện riêng cho EOB khi mà giá trị của VLEcode ở đó là 256.

- Hàm giải mã từ Huffman sang mã VLE, gọi vào chuỗi giá trị bit mã hóa Huffman từ hàm trên, so đó giải mã thành vecto VLEoutput có kích thước là VLEsize.

Đầu tiên là copy chuỗi sang biến encodedCopy, sau đó dùng hàm strtok() để tách lần lượt các giá trị từ chuỗi trước dấu cách (" "), với lệnh `token = strtok(encodedCopy, " ");` để tách lấy giá trị đầu tiên, còn `token = strtok(NULL, " ");` để tách các giá trị còn lại; đồng thời cũng so sánh giá trị được tách từ chuỗi dùng strtok() với mã Huffman tương ứng trong tài liệu để đọc lại giá trị Symbol (Zero-run length, Size) (Amplitude).

```

void HuffmanDecode(const char *encodedString, int *VLEoutput, int *VLEsize) {
    char *token;
    char encodedCopy[100];
    strcpy(encodedCopy, encodedString); // Sao chép chuỗi để sử dụng strtok
    int VLE_index = 0;

```

```

token = strtok(encodedCopy, " ");
while (token != NULL) {
    if (strcmp(token, "011") == 0) { // DC size = 2
        VLEoutput[VLE_index++] = 2;
    } else if (strcmp(token, "11") == 0) { // DC amplitude = 3
        VLEoutput[VLE_index++] = 3;
    } else if (strcmp(token, "01") == 0) { // Amp = -2
        VLEoutput[VLE_index++] = -2;
    } else if (strcmp(token, "0") == 0) { // Amp = -1
        VLEoutput[VLE_index++] = -1;
    } else if (strcmp(token, "00") == 0) { // (0,1)
        VLEoutput[VLE_index++] = 0;
        VLEoutput[VLE_index++] = 1;
    } else if (strcmp(token, "11011") == 0) { // (1,2)
        VLEoutput[VLE_index++] = 1;
        VLEoutput[VLE_index++] = 2;
    } else if (strcmp(token, "11100") == 0) { // (2,1)
        VLEoutput[VLE_index++] = 2;
        VLEoutput[VLE_index++] = 1;
    } else if (strcmp(token, "1010") == 0) { // (0,0) EOB
        VLEoutput[VLE_index++] = 0;
        VLEoutput[VLE_index++] = 0;
        break;
    } else {
        printf("Token không hợp lệ: %s\n", token);
    }
    token = strtok(NULL, " ");
}
*VLEsize = VLE_index;
for (int i = 0; i < VLE_index; i++) {
    printf("%d ", VLEoutput[i]);
}
}

```

- Hàm chuyển đổi từ mã VLE sang dạng ma trận 8x8, hàm này gọi vào DCpre là giá trị DC của khối 8x8 trước đó, vecto giá trị VLE tạo từ hàm trên với kích thước của nó (VLEsize) và tạo ra ma trận.

Ở đây tạo thêm 1 vecto zigzag để lưu trữ các giá trị của từng pixel 1 chiều trước khi đưa vào ma trận 8x8 theo chiều Zig zag.

```

void VLEtoMatrix(int DCpre, int VLEcode[], int VLEsize, int MatrixfromVLE[N][N]) {
    int zigzag[N * N] = {0}; //Vecto lưu trữ tạm các giá trị chuyển đổi trước khi
    chuyển sang dạng ma trận
    int zigzagIndex = 0;
    int DCcur = DCpre + VLEcode[1];
    zigzag[zigzagIndex++] = DCcur;
}

```

```
for (int i = 2; i < VLEsize; i += 3) {
    int zeroRun = VLEcode[i]; // Số lượng số 0
    int amp = VLEcode[i + 2]; // Giá trị amp
    if (zeroRun == 0 && amp == 0) {
        break;
    }
    for (int j = 0; j < zeroRun; j++) {
        zigzag[zigzagIndex++] = 0;
    }
    zigzag[zigzagIndex++] = amp;
}
while (zigzagIndex < N * N) {
    zigzag[zigzagIndex++] = 0;
}
int x = 0, y = 0, index = 0;
int up = 1; // Điều hướng di chuyển: 1 là lên trên, -1 là xuống dưới

for (int i = 0; i < N * N; i++) {
    MatrixfromVLE[x][y] = zigzag[index++];
    if (up) {
        if (y == N - 1) {
            x++;
            up = 0;
        } else if (x == 0) {
            y++;
            up = 0;
        } else {
            x--; y++;
        }
    } else {
        if (x == N - 1) {
            y++;
            up = 1;
        } else if (y == 0) {
            x++;
            up = 1;
        } else {
            x++; y--;
        }
    }
}
for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
        printf("%4d", MatrixfromVLE[i][j]);
    }
    printf("\n");
}
```



```
}  
}
```

- Hàm lượng tử hóa ngược bằng cách nhân giá trị ở từng vị trí của ma trận 8x8 từ hàm trên với vị trí tương ứng của bảng lượng tử hóa.

```
void deQuantize(int norCoeff[N][N], int QuanTable[N][N], int Result[N][N]) {  
    for (int x = 0; x < N; x++) {  
        for (int y = 0; y < N; y++) {  
            Result[x][y] = norCoeff[x][y] * QuanTable[x][y] ;  
            printf("%d ", Result[x][y]);  
        }  
        printf("\n");  
    }  
}
```

- Hàm biến đổi ngược DCT và xuất từng giá trị.

```
void INV DCT_trans(int input[N][N], int output[N][N]) {  
    double Cu, Cv, inv_F;  
    double PI = 3.14159265358979323846;  
  
    for (int x = 0; x < N; x++) {  
        for (int y = 0; y < N; y++) {  
            inv_F = 0.0;  
            for (int u = 0; u < N; u++) {  
                for (int v = 0; v < N; v++) {  
                    if (u == 0) Cu = 1.0 / sqrt(2); else Cu = 1.0;  
                    if (v == 0) Cv = 1.0 / sqrt(2); else Cv = 1.0;  
                    inv_F += Cu * Cv * input[u][v] * cos((2 * x + 1) * u * PI / (2.0  
* N)) * cos((2 * y + 1) * v * PI / (2.0 * N));  
                }  
            }  
            output[x][y] = (int)round(0.25 * inv_F);  
        }  
    }  
}
```

- Hàm main thực hiện khai báo giá trị cho ma trận đầu vào từ tài liệu như hình dưới và gọi các hàm trên

139	144	149	153	155	155	155	155
144	151	153	156	159	156	156	156
150	155	160	163	158	156	156	156
159	161	162	160	160	159	159	159
159	160	161	162	162	155	155	155
161	161	161	161	160	157	157	157
162	162	161	163	162	157	157	157
162	162	161	161	163	158	158	158

(a) source image samples

```
int main() {
    //Figure 10a
    int IN_img[N][N] = {
        {139, 144, 149, 153, 155, 155, 155, 155},
        {144, 151, 153, 156, 159, 156, 156, 156},
        {150, 155, 160, 163, 158, 156, 156, 156},
        {159, 161, 162, 160, 160, 159, 159, 159},
        {159, 160, 161, 162, 162, 155, 155, 155},
        {161, 161, 161, 161, 160, 157, 157, 157},
        {162, 162, 161, 163, 162, 157, 157, 157},
        {162, 162, 161, 161, 163, 158, 158, 158}
    };
    //Figure 10c
    int QuanTable[N][N] = {
        {16, 11, 10, 16, 24, 40, 51, 61},
        {12, 12, 14, 19, 26, 58, 60, 55},
        {14, 13, 16, 24, 40, 57, 69, 56},
        {15, 17, 22, 29, 51, 87, 80, 62},
        {18, 22, 37, 56, 68, 109, 103, 77},
        {24, 35, 55, 64, 81, 104, 113, 92},
        {49, 64, 78, 87, 103, 121, 120, 101},
        {72, 92, 95, 98, 112, 100, 103, 99}
    };
    int QuantCoeff[N][N], denorQuantCoeff[N][N], RecontImg[N][N], LinearString[N * N];
    int PreAmp = 12, VLEcode[N * 3], VLEsize = 0, MatrixVLE[N][N] = {0};
    double DCT_img[N][N];
    scaleValue(IN_img, IN_img, 1);
    printf("Ma trận sau khi khai triển DCT (Figure 10b):\n");
    DCT_trans(IN_img, DCT_img);
    printf("Ma trận sau khi lượng tử hóa (Figure 10d):\n");
    Quantize(DCT_img, QuanTable, QuantCoeff);
    printf("Chuỗi tuyến tính qua đọc Zig-zag:\n");
    Zigzag(QuantCoeff, LinearString);
}
```

```

BaselineEncode(PreAmp, LinearString, VLEcode, &VLEsize);
printVLE(VLEcode, VLEsize);
char Huffmanstr[N * N];
printf("Kết quả mã hóa Huffman:");
HuffmanEncode(VLEcode, VLEsize, Huffmanstr);
printf("\nKết quả giải mã Huffman: ");
HuffmanDecode(Huffmanstr, VLEcode, &VLEsize) ;
printf("\nKết quả chuyển từ chuỗi giải mã hóa Huffman sang ma trận: \n");
VLEtoMatrix(PreAmp, VLEcode, VLEsize, MatrixVLE) ;
printf("Ma trận giải mã từ VLE sau lượng tử hóa ngược (Figure 10e):\n");
deQuantize(MatrixVLE, QuanTable, denorQuantCoeff) ;
printf("Ma trận sau tái cấu trúc (Figure 10f): \n");
INVDCT_trans(denorQuantCoeff, RecontImg);
scaleValue(RecontImg, RecontImg, 0) ;
return 0;
}

```

**Ở vị trí hàng 4 cột 1 của ma trận các giá trị lượng tử hóa, thay đổi giá trị từ 14 sang 15 thì các kết quả phía sau mới đúng.**

Ở đây khai báo cho các biến, QuantCoeff là ma trận 8x8 tạo từ ma trận sau DCT sau khi lượng tử hóa (normalized quantized coefficients), denorQuantCoeff là ma trận 8x8 tạo từ bước lượng tử hóa ngược ma trận tạo bởi mã VLE sau khi đọc theo hướng zig zag (denormalized quantized coefficients), RecontImg là ma trận 8x8 từ ma trận denormalized quantized coefficients biến đổi ngược DCT và scale giá trị lại; LinearString là vecto 1 chiều chứa các giá trị đọc từ normalized quantized coefficients.

PreAmp là giá trị DC từ ma trận trước, VLEcode chứa các giá trị sau mã hóa VLE, VLEsize là kích thước vecto VLE, MatrixVLE là ma trận 8x8 từ giải mã VLE, ma trận này được khởi tạo gồm toàn bộ giá trị đều là 0.

DCT\_img là ma trận sau biến đổi DCT, Huffmanstr là chuỗi bit nhị phân mã hóa Huffman.

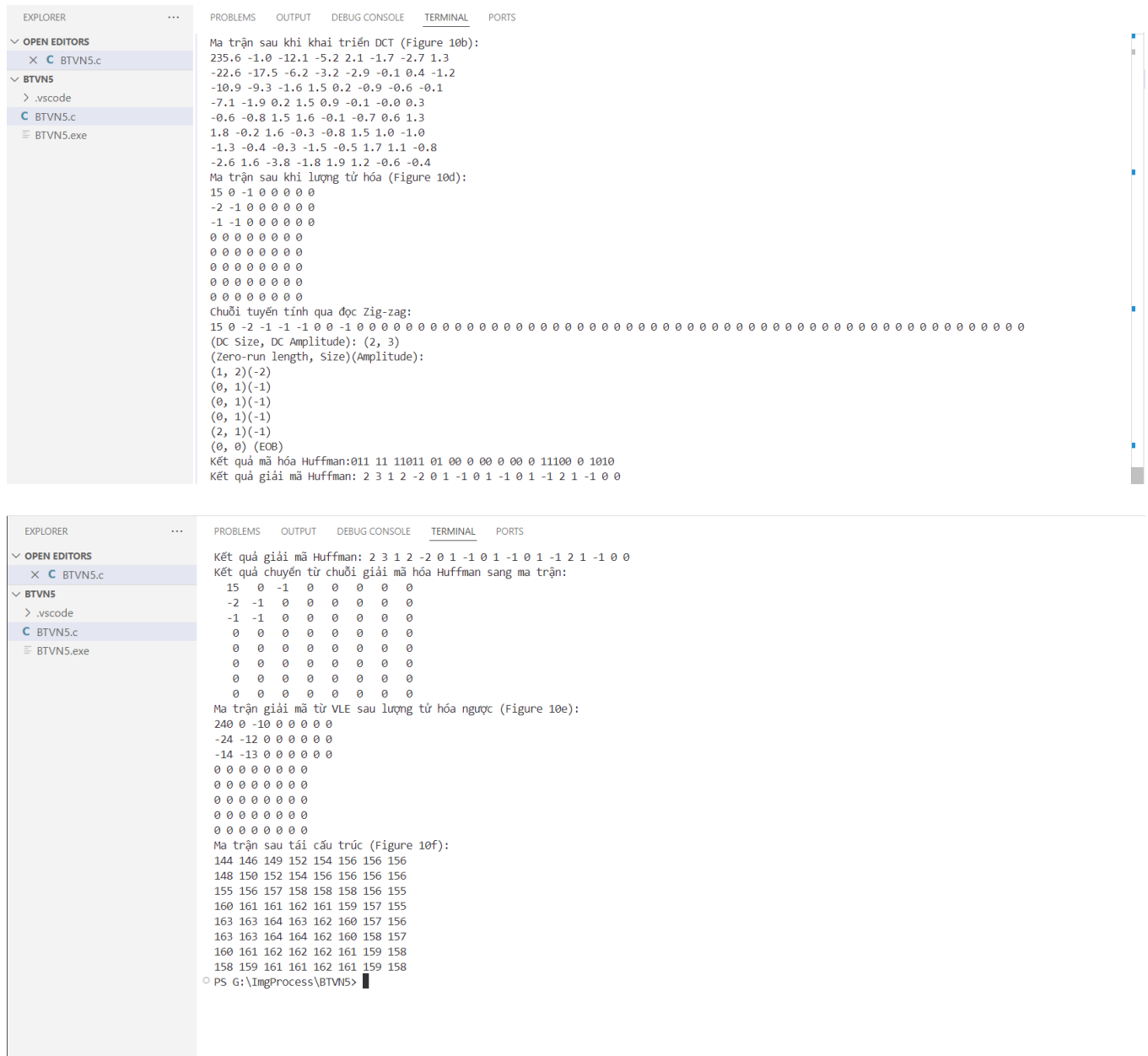
```

int QuantCoeff[N][N], denorQuantCoeff[N][N], RecontImg[N][N], LinearString[N * N];
int PreAmp = 12, VLEcode[N * 3], VLEsize = 0, MatrixVLE[N][N] = {0};
double DCT_img[N][N] ; char Huffmanstr[N * N];

```

## 2. KẾT QUẢ CHẠY THỬ TRÊN VS CODE

## Xử lý ảnh và video



The image shows two screenshots of a Visual Studio Code terminal window. The top screenshot displays the output of a Huffman encoding process. It starts with a message 'Ma trận sau khi khai triển DCT (Figure 10b):' followed by a 10x10 matrix of floating-point values. Then, it shows 'Ma trận sau khi lượng tử hóa (Figure 10d):' followed by a 10x10 matrix of integers. Next, it displays 'Chuỗi tuyến tính qua đọc Zig-zag:' followed by a long sequence of integers. This is followed by '(DC Size, DC Amplitude): (2, 3)', '(Zero-run length, Size)(Amplitude):', and a list of Huffman codes: (1, 2)(-2), (0, 1)(-1), (0, 1)(-1), (0, 1)(-1), (2, 1)(-1), and (0, 0) (EOB). The final lines show the Huffman encoded result: 'Kết quả mã hóa Huffman: 011 11 11011 01 00 0 00 0 00 0 11100 0 1010' and the Huffman decoded result: 'Kết quả giải mã Huffman: 2 3 1 2 -2 0 1 -1 0 1 -1 0 1 -1 2 1 -1 0 0'.

The bottom screenshot displays the output of a Huffman decoding process. It starts with the Huffman encoded result: 'Kết quả giải mã Huffman: 2 3 1 2 -2 0 1 -1 0 1 -1 0 1 -1 2 1 -1 0 0'. Then, it shows the Huffman decoded result: 'Kết quả chuyển từ chuỗi giải mã Huffman sang ma trận:'. This is followed by a 10x10 matrix of integers. Then, it shows 'Ma trận giải mã từ VLE sau lượng tử hóa ngược (Figure 10e):' followed by a 10x10 matrix of integers. Finally, it shows 'Ma trận sau tái cấu trúc (Figure 10f):' followed by a 10x10 matrix of integers. The terminal prompt is 'PS G:\ImgProcess\BTVN5>'.

Link folder chứa source code và các tài nguyên hình ảnh: [https://studenthcmusedu-my.sharepoint.com/:f:/g/personal/20200331\\_student\\_hcmus\\_edu\\_vn/Epr8-ikGHJAIXi-5vyoAw8B5Z-Nh1cs-zOUMDfqp93EnQ?e=WtiTIA](https://studenthcmusedu-my.sharepoint.com/:f:/g/personal/20200331_student_hcmus_edu_vn/Epr8-ikGHJAIXi-5vyoAw8B5Z-Nh1cs-zOUMDfqp93EnQ?e=WtiTIA)