Linear Algebra (0031)
Project 1

Yulwon Rhee (202211342)

Department of Computer Science and Engineering, Konkuk University

1. Write a computer program that performs the following:

(a) Take a greyscale image and form a matrix A.

(b) For given $n = 2^t, t = 0, 1, \cdots$, construct an n-point Haar matrix H.

(c) Perform DHWT $B = H^T AH$.

(d) For given $k = 2^s, s = 0, 1, \cdots$, construct an $n \times n$ matrix $\hat{B}$.

(e) Perform the IDHWT $\hat{A} = H\hat{B}H^T$.

(f) Save the reconstructed image $\hat{A}$ into a file.

Source Code 1.1: `problem1()` from `prj1.c`

```c
void problem1() {
    BITMAPHEADER outputHeader;
    int imgSize, imgWidth, imgHeight;

    double** A = getImageMatrixFromFileName(&outputHeader, &imgWidth, &imgHeight, &imgSize, "problem1/image_lena_24bit.bmp");

    doDHWT(A, imgHeight, imgWidth, imgSize, outputHeader, "problem1/image_lena_24bit");

    releaseMemory(A, imgHeight);
}
```

Source Code 1.2: `doDHWT()` from `prj1.c`

```c
void doDHWT(double** originalImageMatrix, int imgHeight, int imgWidth, int imgSize, BITMAPHEADER outputHeader, char** filePathToSave) {
    //Haar matrix H 구성 (orthonormal column을 갖도록 구성)
    int n = imgHeight; //이미지가 정사각형(Height==Width)이라고 가정; n = 2^t,t=0,1,2,...

    // 1. (b)
    double** H = constructHaarMatrixRecursive(n);
    double** normalisedH = normaliseMatrix(H, n, n);

    // 1. (c)
    double** transposedNormalisedH = transposeMatrix(normalisedH, n, n);
```

```c
    double** HTA = multiplyTwoMatrices(transposedNormalisedH, n, n, originalImageMatrix, n, n);
    double** B = multiplyTwoMatrices(HTA, n, n, normalisedH, n, n);

    // 1. (d)
    double** Bhat = allocateMemory(imgHeight, imgWidth);

    for (int s = 0; s <= 9; s++) { // 2^9 = 512
        int k = pow(2, s);

        // Construct Matrix B Hat
        for (int i = 0; i < imgHeight; i++) {
            for (int j = 0; j < imgWidth; j++) {
                if (i < k && j < k) Bhat[i][j] = B[i][j];
                else Bhat[i][j] = 0;
            }
        }

        // 1. (e)
        double** HBhat = multiplyTwoMatrices(normalisedH, n, n, Bhat, n, n);
        double** Ahat = multiplyTwoMatrices(HBhat, n, n, transposedNormalisedH, n, n);

        // 1. (f)
        // Write Reconstructed Image
        // Ahat을 이용해서 위의 image와 같은 형식이 되도록 구성 (즉, Ahat = [a b;c d]면 [a a a b b b c c c d d d]를 만들어야 함)
        BYTE* Are = (BYTE*)malloc(BYTES_PER_PIXEL * sizeof(BYTE) * imgSize);

        for (int i = 0; i < imgHeight; i++)
            for (int j = 0; j < imgWidth; j++)
                for (int k = 0; k < BYTES_PER_PIXEL; k++)
                    Are[(i * imgWidth + j) * BYTES_PER_PIXEL + k] = (BYTE)Ahat[i][j];

        char fileName[50] = "";
        strcat(fileName, filePathToSave);
        strcat(fileName, "_");
        char kStr[5];
        sprintf(kStr, "%d", k);
        strcat(fileName, kStr);
        strcat(fileName, ".bmp");
```

```c

        writeBitmapFile(BYTES_PER_PIXEL, outputHeader, Are, imgSize, fileName);
        printf("%s saved.\n", fileName);

        releaseMemory(HBhat, n);
        releaseMemory(Ahat, n);
        free(Are);
    }

    releaseMemory(H, n);
    releaseMemory(normalisedH, n);
    releaseMemory(transposedNormalisedH, n);
    releaseMemory(HTA, n);
    releaseMemory(B, n);
    releaseMemory(Bhat, n);
}
```

Source Code 1.3: `getImageMatrixFromFileName()` from `bitmapManager.h`

```c
double** getImageMatrixFromFileName(BITMAPHEADER* outputHeader, int* imgWidth, int* imgHeight, int* imgSize, char* filePath) {
    BITMAPHEADER originalHeader;
    BYTE* image = loadBitmapFile(BYTES_PER_PIXEL, &originalHeader, imgWidth, imgHeight, filePath);

    if (image == NULL) return 0;

    *imgSize = *imgWidth * *imgHeight;
    BYTE* output = (BYTE*)malloc(BYTES_PER_PIXEL * sizeof(BYTE) * *imgSize);
    *outputHeader = originalHeader;

    double** A = allocateMemory(*imgHeight, *imgWidth);

    for (int i = 0; i < *imgHeight; i++)
        for (int j = 0; j < *imgWidth; j++)
            A[i][j] = (double)image[(i * *imgWidth + j) * BYTES_PER_PIXEL];

    free(image);
    free(output);

    return A;
}
```

Fig. 1: Result Images After DHWT

(a) $k = 2^0$

(b) $k = 2^1$

(c) $k = 2^2$

(d) $k = 2^3$

(e) $k = 2^4$

(f) $k = 2^5$

(g) $k = 2^6$

(h) $k = 2^7$

(i) $k = 2^8$

(j) $k = 2^9$

(k) Original Image

2. Get any 2 grayscale images of any format from anywhere (Internet, your personal photos, etc.). It is recommended that you get one image with low frequency components, and one image filled with high frequency components. Use your computer program to do the following:

(a) As k increases, observe the quality of reconstructed image.

(b) Describe any difference between low-freq. image and high-freq. image.

(c) Discuss any findings or thoughts.

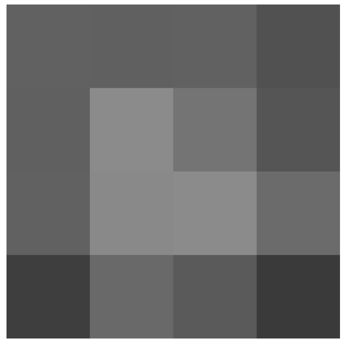
Source Code 1.4: `problem2()` from `prj1.c`

```c
void problem2() {
    BITMAPHEADER lowFreqOutputHeader, highFreqOutputHeader;
    int lowFreqWidth, lowFreqHeight, lowFreqSize, highFreqWidth, highFreqHeight, highFreqSize;

    double** lowFreqImgMatrix = getImageMatrixFromFileName(&lowFreqOutputHeader,
        &lowFreqWidth,
        &lowFreqHeight,
        &lowFreqSize,
        "problem2/low_freq.bmp");

    doDHWT(lowFreqImgMatrix, lowFreqHeight, lowFreqWidth, lowFreqSize, lowFreqOutputHeader, "problem2/low_freq");


    double** highFreqImgMatrix = getImageMatrixFromFileName(&highFreqOutputHeader,
        &highFreqWidth,
        &highFreqHeight,
        &highFreqSize,
        "problem2/high_freq.bmp");

    doDHWT(highFreqImgMatrix, highFreqHeight, highFreqWidth, highFreqSize, highFreqOutputHeader, "problem2/high_freq");

    releaseMemory(lowFreqImgMatrix, lowFreqHeight);
    releaseMemory(highFreqImgMatrix, highFreqHeight);
}
```

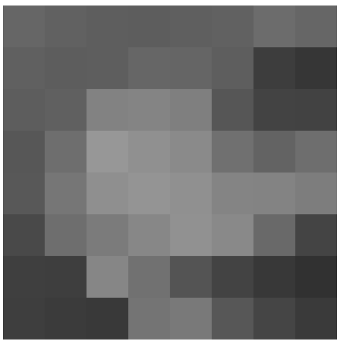
Fig. 2: Result of Low Frequency Image After DHWT

(a) $k = 2^0$

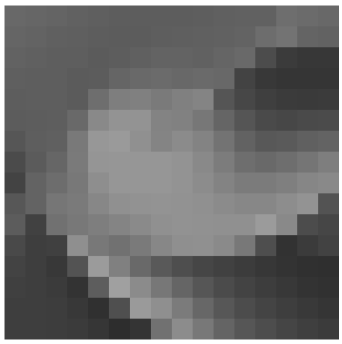
(b) $k = 2^1$



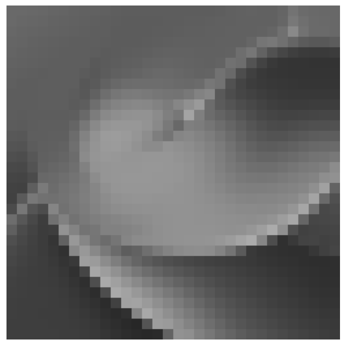
(c) $k = 2^2$



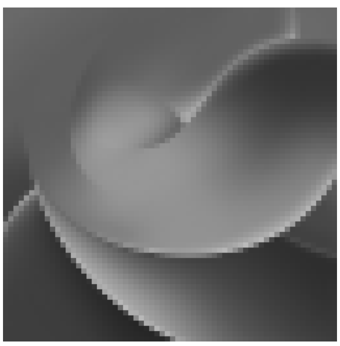
(d) $k = 2^3$



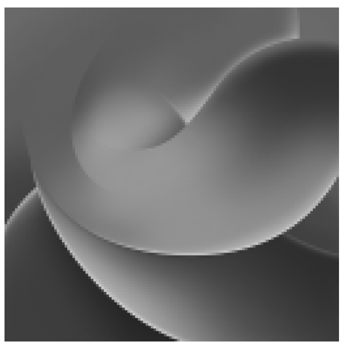
(e) $k = 2^4$



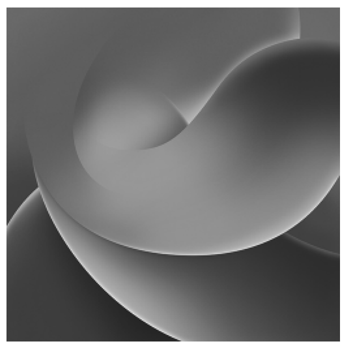
(f) $k = 2^5$



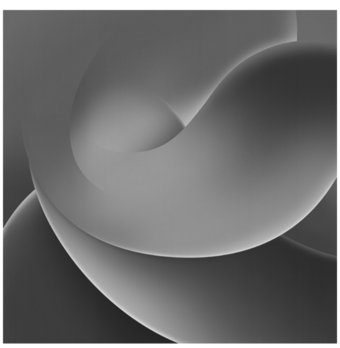
(g) $k = 2^6$



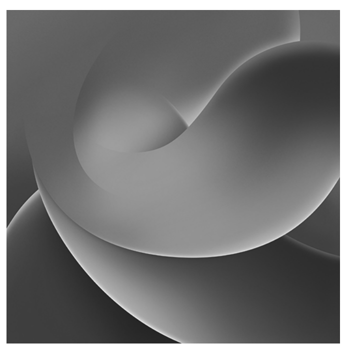
(h) $k = 2^7$



(i) $k = 2^8$



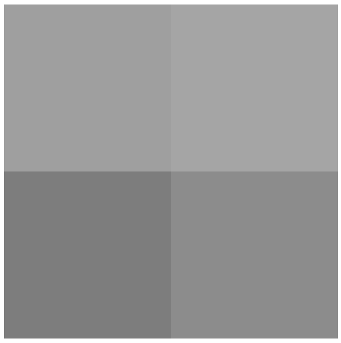
(j) $k = 2^9$



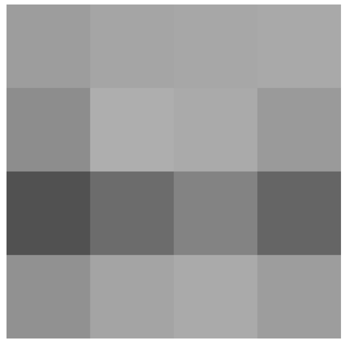
(k) Original Image

Fig. 3: Result of High Frequency Image After DHWT

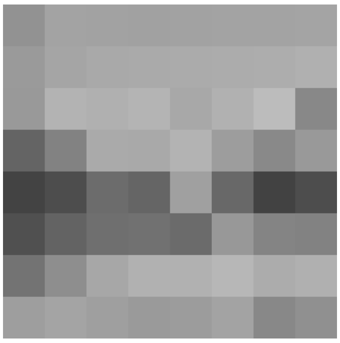
(a) $k = 2^0$



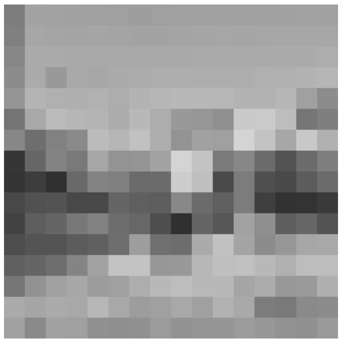
(b) $k = 2^1$



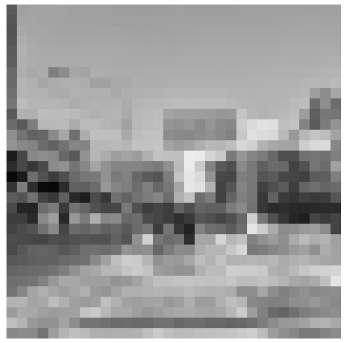
(c) $k = 2^2$



(d) $k = 2^3$



(e) $k = 2^4$



(f) $k = 2^5$

(g) $k = 2^6$

(h) $k = 2^7$

(i) $k = 2^8$

(j) $k = 2^9$

(k) Original Image

High frequency image의 경우, k의 값이 낮아질 수록 품질이 급격하게 하락하여 $k = 2^7$인 경우 글자를 읽을 수 없게 되고, $k = 2^6$부터는 사물을 정확히 판별하기 어려워진다. 반면 Low frequency image의 경우, k의 값이 낮아지더라도 크게 품질이 하락하지 않으며, 위 이미지의 $k = 2^6$인 경우에도 원본과 큰 차이 없이 형체를 인식할 수 있다.

3. (a), (b), (c), (d)

Source Code 1.5: `problem3()` from `prj1.c`

```c
void problem3() {
    BITMAPHEADER outputHeader;
    int imgSize, imgWidth, imgHeight;

    double** A = getImageMatrixFromFileName(&outputHeader, &imgWidth, &imgHeight, &imgSize, "problem3/image_lena_24bit.bmp");

    int n = imgHeight;

    // Split H^T into H_l and H_h
    double** H = constructHaarMatrixRecursive(n);
    double** normalisedH = normaliseMatrix(H, n, n);
    double** transposedNormalisedH = transposeMatrix(normalisedH, n, n);

    double** Hl = allocateMemory(n / 2, n);
    double** Hh = allocateMemory(n / 2, n);

    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (i < n / 2) Hl[i][j] = transposedNormalisedH[i][j];
            else Hh[i - n / 2][j] = transposedNormalisedH[i][j];
        }
    }
    double** HlT = transposeMatrix(Hl, n / 2, n);
    double** HhT = transposeMatrix(Hh, n / 2, n);

    // 3. (a) LHS
    double** HTA = multiplyTwoMatrices(transposedNormalisedH, n, n, A, n, n);
    double** B = multiplyTwoMatrices(HTA, n, n, normalisedH, n, n);

    // 3. (a) RHS
    double** HlA = multiplyTwoMatrices(Hl, n / 2, n, A, n, n);
    double** HhA = multiplyTwoMatrices(Hh, n / 2, n, A, n, n);

    double** HlAHlT = multiplyTwoMatrices(HlA, n / 2, n, HlT, n, n / 2);
```

```c
    double** HlAHhT = multiplyTwoMatrices(HlA, n / 2, n, HhT, n, n / 2);
    double** HhAHlT = multiplyTwoMatrices(HhA, n / 2, n, HlT, n, n / 2);
    double** HhAHhT = multiplyTwoMatrices(HhA, n / 2, n, HhT, n, n / 2);

    // 3. (a) Check LHS == RHS
    bool isASame = true;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            double cmp = 0;
            if (i < n / 2 && j < n / 2) cmp = HlAHlT[i][j];
            else if (i < n / 2 && j >= n / 2) cmp = HlAHhT[i][j - n / 2];
            else if (i >= n / 2 && j < n / 2) cmp = HhAHlT[i - n / 2][j];
            else cmp = HhAHhT[i - n / 2][j - n / 2];

            if (!doubleEquals(cmp, B[i][j])) isASame = false;
        }
    }

    printf("3. (a): %s\n", isASame ? "true" : "false");


    // 3. (b) LHS
    double** HB = multiplyTwoMatrices(normalisedH, n, n, B, n, n);
    double** HBHT = multiplyTwoMatrices(HB, n, n, transposedNormalisedH, n, n);

    // 3. (b) RHS
    double** HlTHlAHlT = multiplyTwoMatrices(HlT, n, n / 2, HlAHlT, n / 2, n / 2);
    double** HlTHlAHlTHl = multiplyTwoMatrices(HlTHlAHlT, n, n / 2, Hl, n / 2, n);

    double** HlTHlAHhT = multiplyTwoMatrices(HlT, n, n / 2, HlAHhT, n / 2, n / 2);
    double** HlTHlAHhTHh = multiplyTwoMatrices(HlTHlAHhT, n, n / 2, Hh, n / 2, n);

    double** HhTHhAHlT = multiplyTwoMatrices(HhT, n, n / 2, HhAHlT, n / 2, n / 2);
    double** HhTHhAHlTHl = multiplyTwoMatrices(HhTHhAHlT, n, n / 2, Hl, n / 2, n);

    double** HhTHhAHhT = multiplyTwoMatrices(HhT, n, n / 2, HhAHhT, n / 2, n / 2);
```

```
    double** HhTHhAHhTHh = multiplyTwoMatrices(HhTHhAHhT, n, n / 2, Hh, n /
    ↪ 2, n);

    double** LURU = addTwoMatrices(HlTHlAHlTHl, n, n, HlTHlAHhTHh, n, n);
    double** LURULL = addTwoMatrices(LURU, n, n, HhTHhAHlTHl, n, n);
    double** RHS = addTwoMatrices(LURULL, n, n, HhTHhAHhTHh, n, n);

    // 3. (b) Check LHS == RHS
    bool isBSame = true;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (!doubleEquals(HBHT[i][j], RHS[i][j])) isBSame = false;
        }
    }

    printf("3. (b): %s\n", isBSame ? "true" : "false");

    // 3. (c)
    BYTE* term1 = (BYTE*)malloc(BYTES_PER_PIXEL * sizeof(BYTE) * imgSize);

    for (int i = 0; i < imgHeight; i++)
        for (int j = 0; j < imgWidth; j++)
            for (int k = 0; k < BYTES_PER_PIXEL; k++)
                term1[(i * imgWidth + j) * BYTES_PER_PIXEL + k] =
    ↪ (BYTE)HlTHlAHlTHl[i][j];

    writeBitmapFile(BYTES_PER_PIXEL, outputHeader, term1, imgSize,
    ↪ "problem3/term1.bmp");


    BYTE* term2 = (BYTE*)malloc(BYTES_PER_PIXEL * sizeof(BYTE) * imgSize);

    for (int i = 0; i < imgHeight; i++)
        for (int j = 0; j < imgWidth; j++)
            for (int k = 0; k < BYTES_PER_PIXEL; k++)
                term2[(i * imgWidth + j) * BYTES_PER_PIXEL + k] =
    ↪ (BYTE)HlTHlAHhTHh[i][j];

    writeBitmapFile(BYTES_PER_PIXEL, outputHeader, term2, imgSize,
    ↪ "problem3/term2.bmp");


    BYTE* term3 = (BYTE*)malloc(BYTES_PER_PIXEL * sizeof(BYTE) * imgSize);

```

```
    for (int i = 0; i < imgHeight; i++)
        for (int j = 0; j < imgWidth; j++)
            for (int k = 0; k < BYTES_PER_PIXEL; k++)
                term3[(i * imgWidth + j) * BYTES_PER_PIXEL + k] =
↪   (BYTE)HhTHhAHlTHl[i][j];

    writeBitmapFile(BYTES_PER_PIXEL, outputHeader, term3, imgSize,
↪   "problem3/term3.bmp");


    BYTE* term4 = (BYTE*)malloc(BYTES_PER_PIXEL * sizeof(BYTE) * imgSize);

    for (int i = 0; i < imgHeight; i++)
        for (int j = 0; j < imgWidth; j++)
            for (int k = 0; k < BYTES_PER_PIXEL; k++)
                term4[(i * imgWidth + j) * BYTES_PER_PIXEL + k] =
↪   (BYTE)HhTHhAHhTHh[i][j];

    writeBitmapFile(BYTES_PER_PIXEL, outputHeader, term4, imgSize,
↪   "problem3/term4.bmp");


    // 3. (d)

    double** Hll = allocateMemory(n / 4, n);
    double** Hlh = allocateMemory(n / 4, n);

    for (int i = 0; i < n / 2; i++) {
        for (int j = 0; j < n; j++) {
            if (i < n / 4) Hll[i][j] = Hl[i][j];
            else Hlh[i - n / 4][j] = Hl[i][j];
        }
    }

    double** HllT = transposeMatrix(Hll, n / 4, n);
    double** HlhT = transposeMatrix(Hlh, n / 4, n);

    // 3. (d) RHS
    double** HllA = multiplyTwoMatrices(Hll, n / 4, n, A, n, n);
    double** HlhA = multiplyTwoMatrices(Hlh, n / 4, n, A, n, n);

    double** HllAHllT = multiplyTwoMatrices(HllA, n / 4, n, HllT, n, n /
↪   4);
```

```
    double** HllAHlhT = multiplyTwoMatrices(HllA, n / 4, n, HlhT, n, n / 4);
    double** HlhAHllT = multiplyTwoMatrices(HlhA, n / 4, n, HllT, n, n / 4);
    double** HlhAHlhT = multiplyTwoMatrices(HlhA, n / 4, n, HlhT, n, n / 4);

    double** HllTHllAHllT = multiplyTwoMatrices(HllT, n, n / 4, HllAHllT, n / 4, n / 4);
    double** HllTHllAHllTHll = multiplyTwoMatrices(HllTHllAHllT, n, n / 4, Hll, n / 4, n);

    double** HllTHllAHlhT = multiplyTwoMatrices(HllT, n, n / 4, HllAHlhT, n / 4, n / 4);
    double** HllTHllAHlhTHlh = multiplyTwoMatrices(HllTHllAHlhT, n, n / 4, Hlh, n / 4, n);

    double** HlhTHlhAHllT = multiplyTwoMatrices(HlhT, n, n / 4, HlhAHllT, n / 4, n / 4);
    double** HlhTHlhAHllTHll = multiplyTwoMatrices(HlhTHlhAHllT, n, n / 4, Hll, n / 4, n);

    double** HlhTHlhAHlhT = multiplyTwoMatrices(HlhT, n, n / 4, HlhAHlhT, n / 4, n / 4);
    double** HlhTHlhAHlhTHlh = multiplyTwoMatrices(HlhTHlhAHlhT, n, n / 4, Hlh, n / 4, n);

    double** dLURU = addTwoMatrices(HllTHllAHllTHll, n, n, HllTHllAHlhTHlh, n, n);
    double** dLURULL = addTwoMatrices(dLURU, n, n, HlhTHlhAHllTHll, n, n);
    double** dRHS = addTwoMatrices(dLURULL, n, n, HlhTHlhAHlhTHlh, n, n);

    // 3. (d) Check LHS == RHS
    bool isDSame = true;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (!doubleEquals(HlTHlAHlTHl[i][j], dRHS[i][j])) isDSame = false;
        }
    }

    printf("3. (d): %s\n", isDSame ? "true" : "false");

    // 3. (d) Save Image
```

```
    BYTE* dTerm1 = (BYTE*)malloc(BYTES_PER_PIXEL * sizeof(BYTE) * imgSize);

    for (int i = 0; i < imgHeight; i++)
        for (int j = 0; j < imgWidth; j++)
            for (int k = 0; k < BYTES_PER_PIXEL; k++)
                dTerm1[(i * imgWidth + j) * BYTES_PER_PIXEL + k] =
    ↪ (BYTE)HllTHllAHllTHll[i][j];

    writeBitmapFile(BYTES_PER_PIXEL, outputHeader, dTerm1, imgSize,
    ↪ "problem3/dTerm1.bmp");


    BYTE* dTerm2 = (BYTE*)malloc(BYTES_PER_PIXEL * sizeof(BYTE) * imgSize);

    for (int i = 0; i < imgHeight; i++)
        for (int j = 0; j < imgWidth; j++)
            for (int k = 0; k < BYTES_PER_PIXEL; k++)
                dTerm2[(i * imgWidth + j) * BYTES_PER_PIXEL + k] =
    ↪ (BYTE)HllTHllAHlhTHlh[i][j];

    writeBitmapFile(BYTES_PER_PIXEL, outputHeader, dTerm2, imgSize,
    ↪ "problem3/dTerm2.bmp");


    BYTE* dTerm3 = (BYTE*)malloc(BYTES_PER_PIXEL * sizeof(BYTE) * imgSize);

    for (int i = 0; i < imgHeight; i++)
        for (int j = 0; j < imgWidth; j++)
            for (int k = 0; k < BYTES_PER_PIXEL; k++)
                dTerm3[(i * imgWidth + j) * BYTES_PER_PIXEL + k] =
    ↪ (BYTE)HlhTHlhAHllTHll[i][j];

    writeBitmapFile(BYTES_PER_PIXEL, outputHeader, dTerm3, imgSize,
    ↪ "problem3/dTerm3.bmp");


    BYTE* dTerm4 = (BYTE*)malloc(BYTES_PER_PIXEL * sizeof(BYTE) * imgSize);

    for (int i = 0; i < imgHeight; i++)
        for (int j = 0; j < imgWidth; j++)
            for (int k = 0; k < BYTES_PER_PIXEL; k++)
                dTerm4[(i * imgWidth + j) * BYTES_PER_PIXEL + k] =
    ↪ (BYTE)HlhTHlhAHlhTHlh[i][j];

```

```
    writeBitmapFile(BYTES_PER_PIXEL, outputHeader, dTerm4, imgSize,
    ↪ "problem3/dTerm4.bmp");


    // Release Allocated Memory
    free(dTerm4);
    free(dTerm3);
    free(dTerm2);
    free(dTerm1);
    releaseMemory(HllA, n / 4);
    releaseMemory(HllAHllT, n / 4);
    releaseMemory(HllAHlhT, n / 4);
    releaseMemory(HlhAHllT, n / 4);
    releaseMemory(HlhAHlhT, n / 4);
    releaseMemory(HllTHllAHllT, n);
    releaseMemory(HllTHllAHllTHll, n);
    releaseMemory(HllTHllAHlhT, n);
    releaseMemory(HllTHllAHlhTHlh, n);
    releaseMemory(HlhTHlhAHllT, n);
    releaseMemory(HlhTHlhAHllTHll, n);
    releaseMemory(HlhTHlhAHlhT, n);
    releaseMemory(HlhTHlhAHlhTHlh, n);
    releaseMemory(dLURU, n);
    releaseMemory(dLURULL, n);
    releaseMemory(dRHS, n);
    free(term4);
    free(term3);
    free(term2);
    free(term1);
    releaseMemory(RHS, n);
    releaseMemory(LURULL, n);
    releaseMemory(LURU, n);
    releaseMemory(HhTHhAHhTHh, n);
    releaseMemory(HhTHhAHhT, n);
    releaseMemory(HhTHhAHlTHl, n);
    releaseMemory(HhTHhAHlT, n);
    releaseMemory(HlTHlAHhTHh, n);
    releaseMemory(HlTHlAHhT, n);
    releaseMemory(HlTHlAHlTHl, n);
    releaseMemory(HlTHlAHlT, n);
    releaseMemory(HBHT, n);
    releaseMemory(HB, n);
    releaseMemory(HhAHhT, n / 2);
    releaseMemory(HhAHlT, n / 2);
```

```
    releaseMemory(HlAHhT, n / 2);
    releaseMemory(HlAHlT, n / 2);
    releaseMemory(HhA, n / 2);
    releaseMemory(HlA, n / 2);
    releaseMemory(B, n);
    releaseMemory(HTA, n);
    releaseMemory(Hh, n / 2);
    releaseMemory(Hl, n / 2);
    releaseMemory(transposedNormalisedH, n);
    releaseMemory(normalisedH, n);
    releaseMemory(H, n);

}
```

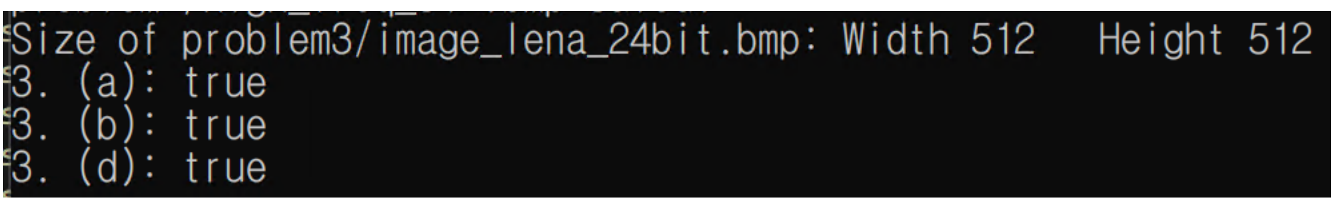


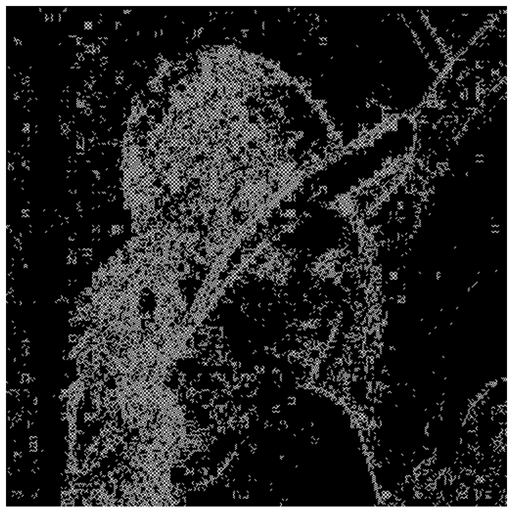
Fig. 4: Program Execution Result

Fig. 5: Result Images of (c)

(a) First Term in (b)

(b) Second Term in (b)

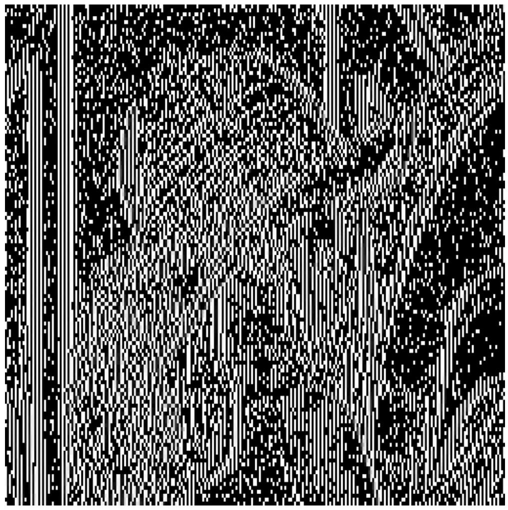
(c) Third Term in (b)



(d) Fourth Term in (b)

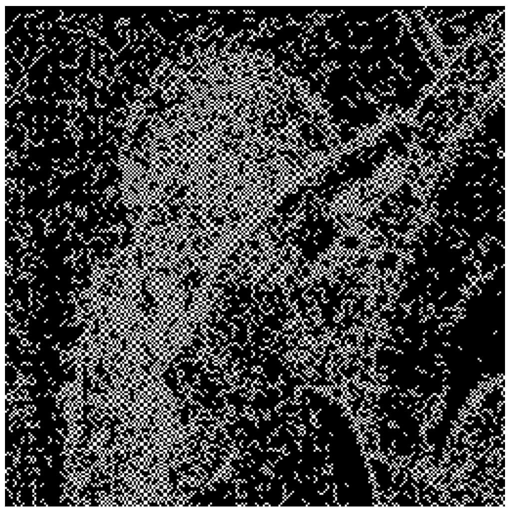
Fig. 6: Result Images of (d)

(a) First Term in (d)



(b) Second Term in (d)

(c) Third Term in (d)



(d) Fourth Term in (d)

첫번째 항에 대한 이미지는 이미지의 대부분의 정보를, 두번째 항에 대한 이미지는 이미지의 세로 방향의 성분을, 세번째 항에 대한 이미지는 이미지의 가로 방향 성분을, 네번째 항에 대한 이미지는 이미지의 Contour 정보를 담고 있는 것을 확인할 수 있다.