

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 $n = 2^t, t = 0, 1, \dots$, construct an n -point Haar matrix H .
 - (c) Perform DHWT $B = H^T A H$.
 - (d) For given $k = 2^s, s = 0, 1, \dots$, 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

```
1 void problem1() {
2     BITMAPHEADER outputHeader;
3     int imgSize, imgWidth, imgHeight;
4
5     double** A = getImageMatrixFromFileName(&outputHeader, &imgWidth,
→    &imgHeight, &imgSize, "problem1/image_lena_24bit.bmp");
6
7     doDHWT(A, imgHeight, imgWidth, imgSize, outputHeader,
→    "problem1/image_lena_24bit");
8
9     releaseMemory(A, imgHeight);
10 }
```

Source Code 1.2: doDHWT() from prj1.c

```
1 void doDHWT(double** originalImageMatrix, int imgHeight, int imgWidth, int
→    imgSize, BITMAPHEADER outputHeader, char** filePathToSave) {
2     //Haar matrix H 구성 (orthonormal column을 갖도록 구성)
3     int n = imgHeight; //이미지가 정사각형(Height==Width)이라고 가정; n =
→    2^t, t=0,1,2,...
4
5     // 1. (b)
6     double** H = constructHaarMatrixRecursive(n);
7     double** normalisedH = normaliseMatrix(H, n, n);
8
9     // 1. (c)
10    double** transposedNormalisedH = transposeMatrix(normalisedH, n, n);
```

```

11     double** HTA = multiplyTwoMatrices(transposedNormalisedH, n, n,
→ originalImageMatrix, n, n);
12     double** B = multiplyTwoMatrices(HTA, n, n, normalisedH, n, n);
13
14     // 1. (d)
15     double** Bhat = allocateMemory(imgHeight, imgWidth);
16
17     for (int s = 0; s <= 9; s++) { // 2^9 = 512
18         int k = pow(2, s);
19
20         // Construct Matrix B Hat
21         for (int i = 0; i < imgHeight; i++) {
22             for (int j = 0; j < imgWidth; j++) {
23                 if (i < k && j < k) Bhat[i][j] = B[i][j];
24                 else Bhat[i][j] = 0;
25             }
26         }
27
28         // 1. (e)
29         double** HBhat = multiplyTwoMatrices(normalisedH, n, n, Bhat, n,
→ n);
30         double** Ahat = multiplyTwoMatrices(HBhat, n, n,
→ transposedNormalisedH, n, n);
31
32         // 1. (f)
33         // Write Reconstructed Image
34         // 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)
35         BYTE* Are = (BYTE*)malloc(BYTES_PER_PIXEL * sizeof(BYTE) *
→ imgSize);
36
37         for (int i = 0; i < imgHeight; i++)
38             for (int j = 0; j < imgWidth; j++)
39                 for (int k = 0; k < BYTES_PER_PIXEL; k++)
40                     Are[(i * imgWidth + j) * BYTES_PER_PIXEL + k] =
→ (BYTE)Ahat[i][j];
41
42         char fileName[50] = "";
43         strcat(fileName, filePathToSave);
44         strcat(fileName, "_");
45         char kStr[5];
46         sprintf(kStr, "%d", k);
47         strcat(fileName, kStr);
48         strcat(fileName, ".bmp");

```

```

49
50     writeBitmapFile(BYTES_PER_PIXEL, outputHeader, Are, imgSize,
→   fileName);
51     printf("%s saved.\n", fileName);
52
53     releaseMemory(HBhat, n);
54     releaseMemory(Ahat, n);
55     free(Are);
56 }
57
58 releaseMemory(H, n);
59 releaseMemory(normalisedH, n);
60 releaseMemory(transposedNormalisedH, n);
61 releaseMemory(HTA, n);
62 releaseMemory(B, n);
63 releaseMemory(Bhat, n);
64 }

```

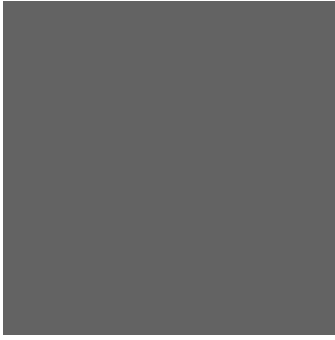
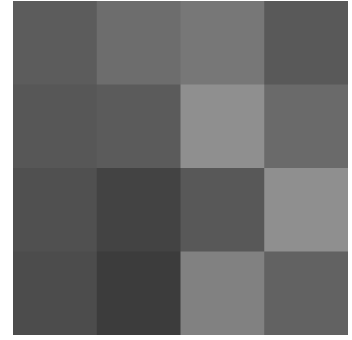
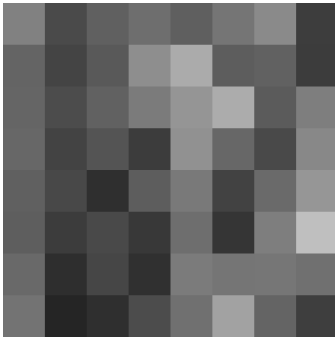
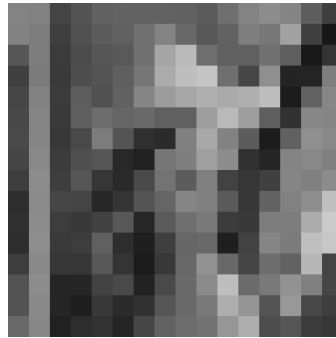
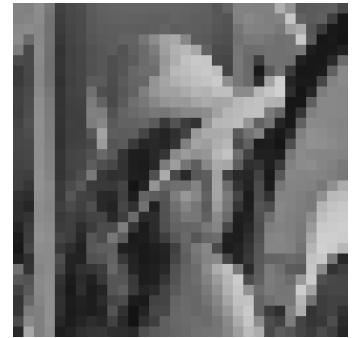
Source Code 1.3: getImageMatrixFromFile() from bitmapManager.h

```

1  double** getImageMatrixFromFileName(BITMAPHEADER* outputHeader, int*
→   imgWidth, int* imgHeight, int* imgSize, char* filePath) {
2     BITMAPHEADER originalHeader;
3     BYTE* image = loadBitmapFile(BYTES_PER_PIXEL, &originalHeader,
→   imgWidth, imgHeight, filePath);
4
5     if (image == NULL) return 0;
6
7     *imgSize = *imgWidth * *imgHeight;
8     BYTE* output = (BYTE*)malloc(BYTES_PER_PIXEL * sizeof(BYTE) *
→   *imgSize);
9     *outputHeader = originalHeader;
10
11     double** A = allocateMemory(*imgHeight, *imgWidth);
12
13     for (int i = 0; i < *imgHeight; i++)
14         for (int j = 0; j < *imgWidth; j++)
15             A[i][j] = (double)image[(i * *imgWidth + j) * BYTES_PER_PIXEL];
16
17     free(image);
18     free(output);
19
20     return A;
21 }

```

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

```

1 void problem2() {
2     BITMAPHEADER lowFreqOutputHeader, highFreqOutputHeader;
3     int lowFreqWidth, lowFreqHeight, lowFreqSize, highFreqWidth,
    → highFreqHeight, highFreqSize;
4
5     double** lowFreqImgMatrix =
    → getImageMatrixFromFileName(&lowFreqOutputHeader,
6         &lowFreqWidth,
7         &lowFreqHeight,
8         &lowFreqSize,
9         "problem2/low_freq.bmp");
10
11     doDHWt(lowFreqImgMatrix, lowFreqHeight, lowFreqWidth, lowFreqSize,
    → lowFreqOutputHeader, "problem2/low_freq");
12
13
14     double** highFreqImgMatrix =
    → getImageMatrixFromFileName(&highFreqOutputHeader,
15         &highFreqWidth,
16         &highFreqHeight,
17         &highFreqSize,
18         "problem2/high_freq.bmp");
19
20     doDHWt(highFreqImgMatrix, highFreqHeight, highFreqWidth, highFreqSize,
    → highFreqOutputHeader, "problem2/high_freq");
21
22     releaseMemory(lowFreqImgMatrix, lowFreqHeight);
23     releaseMemory(highFreqImgMatrix, highFreqHeight);
24 }

```

Fig. 2: Result of Low Frequency Image After DHWT

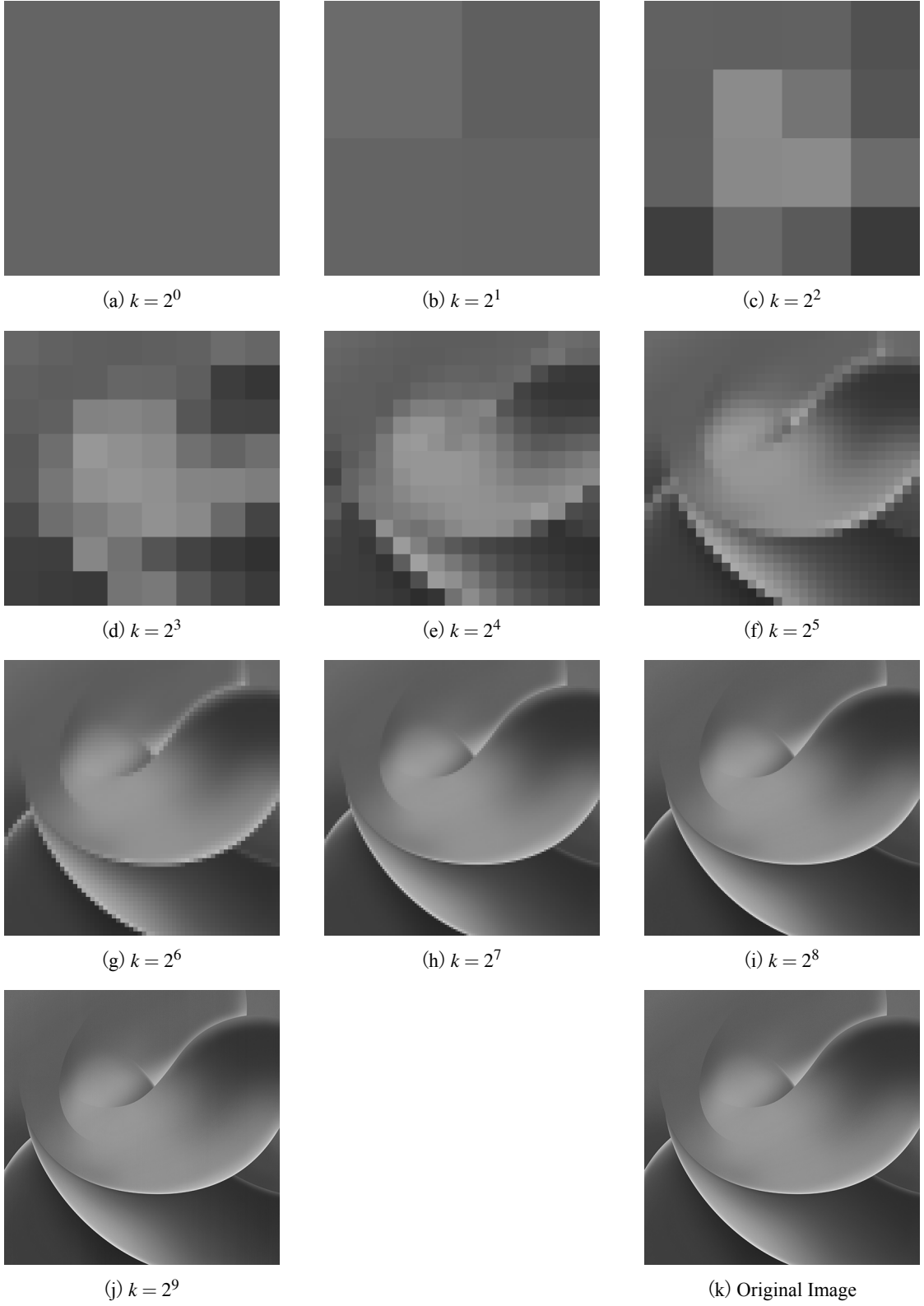
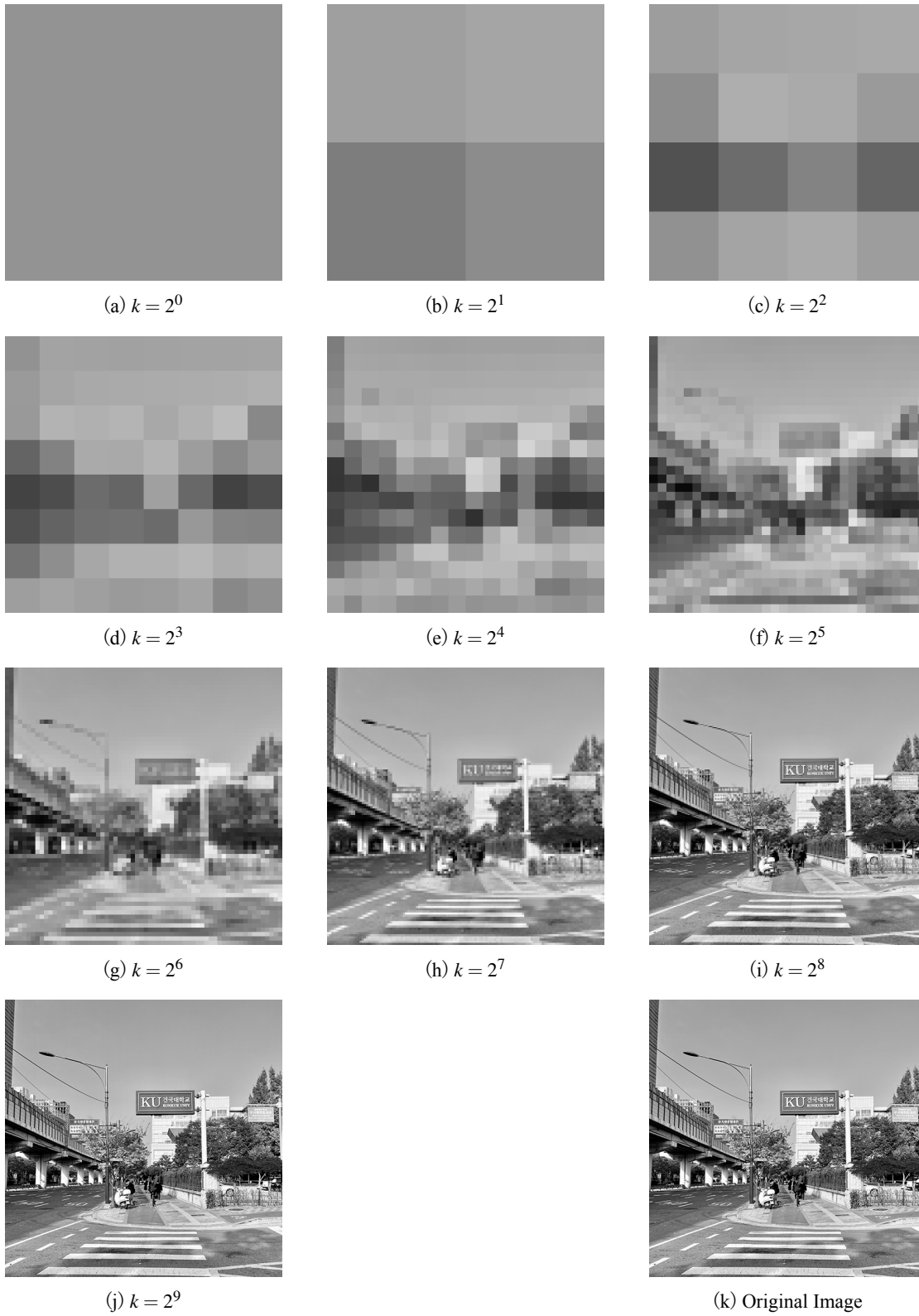


Fig. 3: Result of High Frequency Image After DHWT



High frequency image의 경우, k 의 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, k 의 값이 낮아지더라도 크게 품질이 하락하지 않으며, 위 이미지의 $k = 2^6$ 인 경우에도 원본과 큰 차이 없이 형태를 인식할 수 있다.

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

Source Code 1.5: problem3() from prj1.c

```

1 void problem3() {
2     BITMAPHEADER outputHeader;
3     int imgSize, imgWidth, imgHeight;
4
5     double** A = getImageMatrixFromFileName(&outputHeader, &imgWidth,
→   &imgHeight, &imgSize, "problem3/image_lena_24bit.bmp");
6
7     int n = imgHeight;
8
9     // Split  $H^T$  into  $H_l$  and  $H_h$ 
10    double** H = constructHaarMatrixRecursive(n);
11    double** normalisedH = normaliseMatrix(H, n, n);
12    double** transposedNormalisedH = transposeMatrix(normalisedH, n, n);
13
14    double** Hl = allocateMemory(n / 2, n);
15    double** Hh = allocateMemory(n / 2, n);
16
17    for (int i = 0; i < n; i++) {
18        for (int j = 0; j < n; j++) {
19            if (i < n / 2) Hl[i][j] = transposedNormalisedH[i][j];
20            else Hh[i - n / 2][j] = transposedNormalisedH[i][j];
21        }
22    }
23    double** HlT = transposeMatrix(Hl, n / 2, n);
24    double** HhT = transposeMatrix(Hh, n / 2, n);
25
26    // 3. (a) LHS
27    double** HTA = multiplyTwoMatrices(transposedNormalisedH, n, n, A, n,
→   n);
28    double** B = multiplyTwoMatrices(HTA, n, n, normalisedH, n, n);
29
30    // 3. (a) RHS
31    double** HlA = multiplyTwoMatrices(Hl, n / 2, n, A, n, n);
32    double** HhA = multiplyTwoMatrices(Hh, n / 2, n, A, n, n);
33
34    double** HlAHlT = multiplyTwoMatrices(HlA, n / 2, n, HlT, n, n / 2);

```



```

35 double** H1AHhT = multiplyTwoMatrices(H1A, n / 2, n, HhT, n, n / 2);
36 double** HhAHlT = multiplyTwoMatrices(HhA, n / 2, n, HlT, n, n / 2);
37 double** HhAHhT = multiplyTwoMatrices(HhA, n / 2, n, HhT, n, n / 2);
38
39 // 3. (a) Check LHS == RHS
40 bool isASame = true;
41 for (int i = 0; i < n; i++) {
42     for (int j = 0; j < n; j++) {
43         double cmp = 0;
44         if (i < n / 2 && j < n / 2) cmp = H1AHlT[i][j];
45         else if (i < n / 2 && j >= n / 2) cmp = H1AHhT[i][j - n / 2];
46         else if (i >= n / 2 && j < n / 2) cmp = HhAHlT[i - n / 2][j];
47         else cmp = HhAHhT[i - n / 2][j - n / 2];
48
49         if (!doubleEquals(cmp, B[i][j])) isASame = false;
50     }
51 }
52
53 printf("3. (a): %s\n", isASame ? "true" : "false");
54
55
56 // 3. (b) LHS
57 double** HB = multiplyTwoMatrices(normalisedH, n, n, B, n, n);
58 double** HBHT = multiplyTwoMatrices(HB, n, n, transposedNormalisedH, n,
→ n);
59
60 // 3. (b) RHS
61 double** HlTHlAHlT = multiplyTwoMatrices(HlT, n, n / 2, H1AHlT, n / 2,
→ n / 2);
62 double** HlTHlAHlTHl = multiplyTwoMatrices(HlTHlAHlT, n, n / 2, Hl, n /
→ 2, n);
63
64 double** HlTHlAHhT = multiplyTwoMatrices(HlT, n, n / 2, H1AHhT, n / 2,
→ n / 2);
65 double** HlTHlAHhTHh = multiplyTwoMatrices(HlTHlAHhT, n, n / 2, Hh, n /
→ 2, n);
66
67 double** HhTHhAHlT = multiplyTwoMatrices(HhT, n, n / 2, HhAHlT, n / 2,
→ n / 2);
68 double** HhTHhAHlTHl = multiplyTwoMatrices(HhTHhAHlT, n, n / 2, Hl, n /
→ 2, n);
69
70 double** HhTHhAHhT = multiplyTwoMatrices(HhT, n, n / 2, HhAHhT, n / 2,
→ n / 2);

```

```

71     double** HhTHhAHhTHh = multiplyTwoMatrices(HhTHhAHhT, n, n / 2, Hh, n /
→ 2, n);
72
73     double** LURU = addTwoMatrices(H1TH1AH1TH1, n, n, H1TH1AHhTHh, n, n);
74     double** LURULL = addTwoMatrices(LURU, n, n, HhTHhAH1TH1, n, n);
75     double** RHS = addTwoMatrices(LURULL, n, n, HhTHhAHhTHh, n, n);
76
77     // 3. (b) Check LHS == RHS
78     bool isBSame = true;
79     for (int i = 0; i < n; i++) {
80         for (int j = 0; j < n; j++) {
81             if (!doubleEquals(HBHT[i][j], RHS[i][j])) isBSame = false;
82         }
83     }
84
85     printf("3. (b): %s\n", isBSame ? "true" : "false");
86
87     // 3. (c)
88     BYTE* term1 = (BYTE*)malloc(BYTES_PER_PIXEL * sizeof(BYTE) * imgSize);
89
90     for (int i = 0; i < imgHeight; i++)
91         for (int j = 0; j < imgWidth; j++)
92             for (int k = 0; k < BYTES_PER_PIXEL; k++)
93                 term1[(i * imgWidth + j) * BYTES_PER_PIXEL + k] =
→ (BYTE)H1TH1AH1TH1[i][j];
94
95     writeBitmapFile(BYTES_PER_PIXEL, outputHeader, term1, imgSize,
→ "problem3/term1.bmp");
96
97
98     BYTE* term2 = (BYTE*)malloc(BYTES_PER_PIXEL * sizeof(BYTE) * imgSize);
99
100    for (int i = 0; i < imgHeight; i++)
101        for (int j = 0; j < imgWidth; j++)
102            for (int k = 0; k < BYTES_PER_PIXEL; k++)
103                term2[(i * imgWidth + j) * BYTES_PER_PIXEL + k] =
→ (BYTE)H1TH1AHhTHh[i][j];
104
105    writeBitmapFile(BYTES_PER_PIXEL, outputHeader, term2, imgSize,
→ "problem3/term2.bmp");
106
107
108    BYTE* term3 = (BYTE*)malloc(BYTES_PER_PIXEL * sizeof(BYTE) * imgSize);
109

```

```

110     for (int i = 0; i < imgHeight; i++)
111         for (int j = 0; j < imgWidth; j++)
112             for (int k = 0; k < BYTES_PER_PIXEL; k++)
113                 term3[(i * imgWidth + j) * BYTES_PER_PIXEL + k] =
→ (BYTE)HhTHhAHlTHl[i][j];
114
115     writeBitmapFile(BYTES_PER_PIXEL, outputHeader, term3, imgSize,
→ "problem3/term3.bmp");
116
117
118     BYTE* term4 = (BYTE*)malloc(BYTES_PER_PIXEL * sizeof(BYTE) * imgSize);
119
120     for (int i = 0; i < imgHeight; i++)
121         for (int j = 0; j < imgWidth; j++)
122             for (int k = 0; k < BYTES_PER_PIXEL; k++)
123                 term4[(i * imgWidth + j) * BYTES_PER_PIXEL + k] =
→ (BYTE)HhTHhAHhTHh[i][j];
124
125     writeBitmapFile(BYTES_PER_PIXEL, outputHeader, term4, imgSize,
→ "problem3/term4.bmp");
126
127
128     // 3. (d)
129
130     double** H1l = allocateMemory(n / 4, n);
131     double** H1h = allocateMemory(n / 4, n);
132
133     for (int i = 0; i < n / 2; i++) {
134         for (int j = 0; j < n; j++) {
135             if (i < n / 4) H1l[i][j] = H1[i][j];
136             else H1h[i - n / 4][j] = H1[i][j];
137         }
138     }
139
140     double** H1lT = transposeMatrix(H1l, n / 4, n);
141     double** H1hT = transposeMatrix(H1h, n / 4, n);
142
143     // 3. (d) RHS
144     double** H1lA = multiplyTwoMatrices(H1l, n / 4, n, A, n, n);
145     double** H1hA = multiplyTwoMatrices(H1h, n / 4, n, A, n, n);
146
147     double** H1lAH1lT = multiplyTwoMatrices(H1lA, n / 4, n, H1lT, n, n /
→ 4);

```

```

148     double** H1lAH1hT = multiplyTwoMatrices(H1lA, n / 4, n, H1hT, n, n /
→ 4);
149     double** H1hAH1lT = multiplyTwoMatrices(H1hA, n / 4, n, H1lT, n, n /
→ 4);
150     double** H1hAH1hT = multiplyTwoMatrices(H1hA, n / 4, n, H1hT, n, n /
→ 4);
151
152     double** H1lTH1lAH1lT = multiplyTwoMatrices(H1lT, n, n / 4, H1lAH1lT, n
→ / 4, n / 4);
153     double** H1lTH1lAH1lTH1l = multiplyTwoMatrices(H1lTH1lAH1lT, n, n / 4,
→ H1l, n / 4, n);
154
155     double** H1lTH1lAH1hT = multiplyTwoMatrices(H1lT, n, n / 4, H1lAH1hT, n
→ / 4, n / 4);
156     double** H1lTH1lAH1hTH1h = multiplyTwoMatrices(H1lTH1lAH1hT, n, n / 4,
→ H1h, n / 4, n);
157
158     double** H1hTH1hAH1lT = multiplyTwoMatrices(H1hT, n, n / 4, H1hAH1lT, n
→ / 4, n / 4);
159     double** H1hTH1hAH1lTH1l = multiplyTwoMatrices(H1hTH1hAH1lT, n, n / 4,
→ H1l, n / 4, n);
160
161     double** H1hTH1hAH1hT = multiplyTwoMatrices(H1hT, n, n / 4, H1hAH1hT, n
→ / 4, n / 4);
162     double** H1hTH1hAH1hTH1h = multiplyTwoMatrices(H1hTH1hAH1hT, n, n / 4,
→ H1h, n / 4, n);
163
164     double** dLURU = addTwoMatrices(H1lTH1lAH1lTH1l, n, n, H1lTH1lAH1hTH1h,
→ n, n);
165     double** dLURULL = addTwoMatrices(dLURU, n, n, H1hTH1hAH1lTH1l, n, n);
166     double** dRHS = addTwoMatrices(dLURULL, n, n, H1hTH1hAH1hTH1h, n, n);
167
168     // 3. (d) Check LHS == RHS
169     bool isDSame = true;
170     for (int i = 0; i < n; i++) {
171         for (int j = 0; j < n; j++) {
172             if (!doubleEquals(H1TH1AH1TH1[i][j], dRHS[i][j])) isDSame =
→ false;
173         }
174     }
175
176     printf("3. (d): %s\n", isDSame ? "true" : "false");
177
178     // 3. (d) Save Image

```

```

179     BYTE* dTerm1 = (BYTE*)malloc(BYTES_PER_PIXEL * sizeof(BYTE) * imgSize);
180
181     for (int i = 0; i < imgHeight; i++)
182         for (int j = 0; j < imgWidth; j++)
183             for (int k = 0; k < BYTES_PER_PIXEL; k++)
184                 dTerm1[(i * imgWidth + j) * BYTES_PER_PIXEL + k] =
→ (BYTE)H11TH11AH11TH11[i][j];
185
186     writeBitmapFile(BYTES_PER_PIXEL, outputHeader, dTerm1, imgSize,
→ "problem3/dTerm1.bmp");
187
188
189     BYTE* dTerm2 = (BYTE*)malloc(BYTES_PER_PIXEL * sizeof(BYTE) * imgSize);
190
191     for (int i = 0; i < imgHeight; i++)
192         for (int j = 0; j < imgWidth; j++)
193             for (int k = 0; k < BYTES_PER_PIXEL; k++)
194                 dTerm2[(i * imgWidth + j) * BYTES_PER_PIXEL + k] =
→ (BYTE)H11TH11AH1hTH1h[i][j];
195
196     writeBitmapFile(BYTES_PER_PIXEL, outputHeader, dTerm2, imgSize,
→ "problem3/dTerm2.bmp");
197
198
199     BYTE* dTerm3 = (BYTE*)malloc(BYTES_PER_PIXEL * sizeof(BYTE) * imgSize);
200
201     for (int i = 0; i < imgHeight; i++)
202         for (int j = 0; j < imgWidth; j++)
203             for (int k = 0; k < BYTES_PER_PIXEL; k++)
204                 dTerm3[(i * imgWidth + j) * BYTES_PER_PIXEL + k] =
→ (BYTE)H1hTH1hAH11TH11[i][j];
205
206     writeBitmapFile(BYTES_PER_PIXEL, outputHeader, dTerm3, imgSize,
→ "problem3/dTerm3.bmp");
207
208
209     BYTE* dTerm4 = (BYTE*)malloc(BYTES_PER_PIXEL * sizeof(BYTE) * imgSize);
210
211     for (int i = 0; i < imgHeight; i++)
212         for (int j = 0; j < imgWidth; j++)
213             for (int k = 0; k < BYTES_PER_PIXEL; k++)
214                 dTerm4[(i * imgWidth + j) * BYTES_PER_PIXEL + k] =
→ (BYTE)H1hTH1hAH1hTH1h[i][j];
215

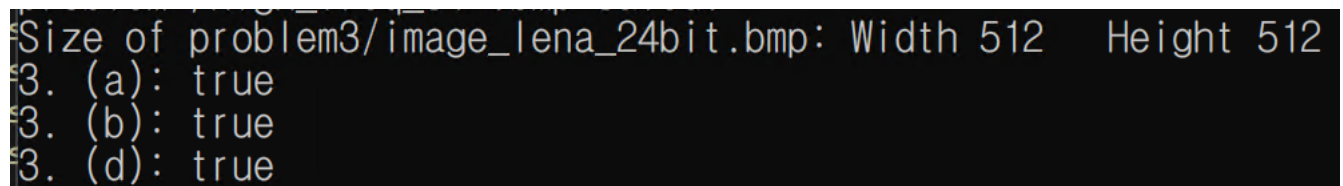
```

```

216     writeBitmapFile(BYTES_PER_PIXEL, outputHeader, dTerm4, imgSize,
→    "problem3/dTerm4.bmp");
217
218
219     // Release Allocated Memory
220     free(dTerm4);
221     free(dTerm3);
222     free(dTerm2);
223     free(dTerm1);
224     releaseMemory(H11A, n / 4);
225     releaseMemory(H11AH11T, n / 4);
226     releaseMemory(H11AH1hT, n / 4);
227     releaseMemory(H1hAH11T, n / 4);
228     releaseMemory(H1hAH1hT, n / 4);
229     releaseMemory(H11TH11AH11T, n);
230     releaseMemory(H11TH11AH11TH11, n);
231     releaseMemory(H11TH11AH1hT, n);
232     releaseMemory(H11TH11AH1hTH1h, n);
233     releaseMemory(H1hTH1hAH11T, n);
234     releaseMemory(H1hTH1hAH11TH11, n);
235     releaseMemory(H1hTH1hAH1hT, n);
236     releaseMemory(H1hTH1hAH1hTH1h, n);
237     releaseMemory(dLURU, n);
238     releaseMemory(dLURULL, n);
239     releaseMemory(dRHS, n);
240     free(term4);
241     free(term3);
242     free(term2);
243     free(term1);
244     releaseMemory(RHS, n);
245     releaseMemory(LURULL, n);
246     releaseMemory(LURU, n);
247     releaseMemory(HhTHhAHhTHh, n);
248     releaseMemory(HhTHhAHhT, n);
249     releaseMemory(HhTHhAH1TH1, n);
250     releaseMemory(HhTHhAH1T, n);
251     releaseMemory(H1TH1AHhTHh, n);
252     releaseMemory(H1TH1AHhT, n);
253     releaseMemory(H1TH1AH1TH1, n);
254     releaseMemory(H1TH1AH1T, n);
255     releaseMemory(HBHT, n);
256     releaseMemory(HB, n);
257     releaseMemory(HhAHhT, n / 2);
258     releaseMemory(HhAH1T, n / 2);

```

```
259     releaseMemory(HlAHhT, n / 2);
260     releaseMemory(HlAHlT, n / 2);
261     releaseMemory(HhA, n / 2);
262     releaseMemory(HlA, n / 2);
263     releaseMemory(B, n);
264     releaseMemory(HTA, n);
265     releaseMemory(Hh, n / 2);
266     releaseMemory(Hl, n / 2);
267     releaseMemory(transposedNormalisedH, n);
268     releaseMemory(normalisedH, n);
269     releaseMemory(H, n);
270
271 }
```



```
Size of problem3/image_lena_24bit.bmp: Width 512 Height 512
3. (a): true
3. (b): true
3. (d): true
```

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)

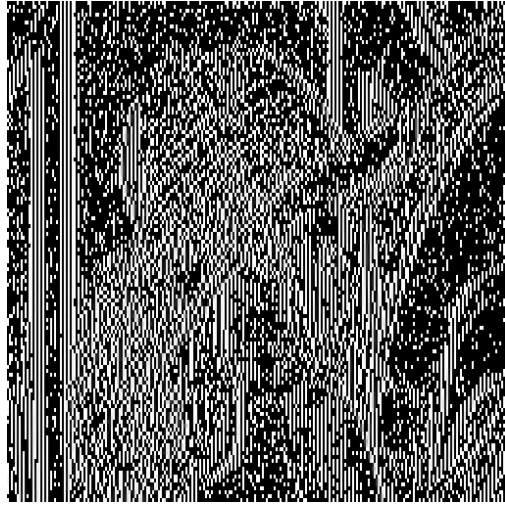


(d) Fourth Term in (b)

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.