Q1.

The pixels in an image are scanned from left to the right and from the top to the bottom. Each new pixel is predicted by the average of the pixel above and the one to the left. Let f and F represent the original and the predicted values, and e = f – F is the prediction error. The prediction error is quantized to “0”, “B”, or “-B” according to:

Find the optimum weights while predicting the image such that mean square error is minimum. Repeat the process if you use all nearest neighbor to predict the pixel value. Repeat the process on any image of your choice

1. The pixels in an image are scanned from left to the right and from the top to the bottom. Each new pixel is predicted by the average of the pixel above and the one to the left. Let f and F represent the original and the predicted values, and e = f – F is the prediction error. The prediction error is quantized to “0”, “B”, or “-B” according to:

$$\hat{e} = \begin{cases} -B & e < -T \\ 0 & -T \le e \le T \\ B & e > T \end{cases}$$

0	0	1	5	6
0	0	1	5	6
2	2	4	7	8
3	3	7	4	2
6	6	5	1	0

Find the optimum weights while predicting the image such that mean square error is minimum.
Repeat the process if you use all nearest neighbor to predict the pixel value.
Repeat the process on any image of your choice

In [1]:
```
import numpy as np
import cv2 as cv
from matplotlib import pyplot as plt
```

In [2]:
```
img = np.array([
    [0,0,1,5,6],
    [0,0,1,5,6],
    [2,2,4,7,8],
    [3,3,7,4,2],
    [6,6,5,1,0]
])
```

In [3]:
```
pad = np.pad(img,1)
pad
```

Out[3]:
```
array([[0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 1, 5, 6, 0],
       [0, 0, 0, 1, 5, 6, 0],
       [0, 2, 2, 4, 7, 8, 0],
       [0, 3, 3, 7, 4, 2, 0],
       [0, 6, 6, 5, 1, 0, 0],
       [0, 0, 0, 0, 0, 0, 0]])
```

In [4]:
```
w = pad[1:-1, :-2]
e = pad[1:-1, 2: ]
n = pad[:-2 ,1:-1]
s = pad[2:  ,1:-1]
```

```python
nw = pad[:-2, :-2]
ne = pad[:-2 , 2:]
se = pad[2: ,  2:]
sw = pad[2:,  :-2]
```

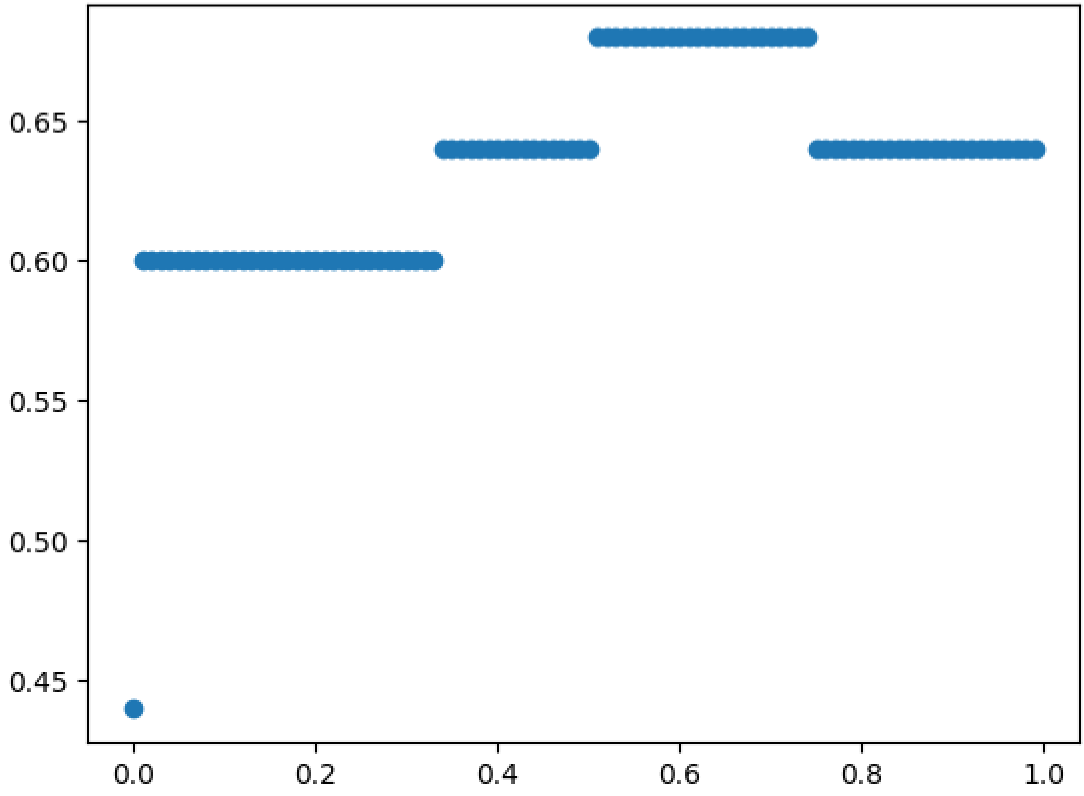
In [5]:
```python
def average(arrays, weights):
    shape = arrays[0].shape
    out = np.zeros(shape)
    norm = np.sum(np.array(weights))
    for a,w in zip(arrays, weights):
        out += a*w/norm
    return out
```

In [6]:
```python
def error(orig, pred, B=1, T=1):
    e = orig - pred
    r1 = e < -T
    r2a = -T <= e
    r2b = e <= T
    r3 = T < e
    e[r2a] = 0
    e[r2b] = 0
    e[r1] = -B
    e[r3] = B
    return e
```

In [7]:
```python
def mse(arr):
    return np.mean(np.square(arr))
```

In [8]:
```python
def find2optimum():
    x = []
    y = []
    for weight in np.arange(start=0, stop=1, step=0.01):
        pred = average([n,w], [weight, 1-weight])
        E = error(img, pred)
        e = mse(E)
        x.append(weight)
        y.append(e)
    plt.scatter(x,y)
```

In [9]:
```python
find2optimum()
```



Result (left and top)

We obtain that the optimal weight is 0 for top and 1 for left. The MSE will be 0.4, taking B=1 and T=1

In [10]:
```
def find8optimum(start=0, stop=1, step=0.25):
    err_Arr = []
    weight_Arr = []
    for w1 in np.arange(start, stop, step):
        for w2 in np.arange(start, stop, step):
            for w3 in np.arange(start, stop, step):
                for w4 in np.arange(start, stop, step):
                    for w5 in np.arange(start, stop, step):
                        for w6 in np.arange(start, stop, step):
                            for w7 in np.arange(start, stop, step):
                                for w8 in np.arange(start, stop, step):
                                    pred = average([n, ne, e, se, s, sw,
                                    E = error(img, pred)
                                    err = mse(E)
                                    if not np.isnan(err):
                                        err_Arr.append(err)
                                        weight_Arr.append([w1,w2,w3,w4,w5
    min_idx = np.argmin(err_Arr)
    min_val = err_Arr[min_idx]
    min_weight = weight_Arr[min_idx]
    print(f"Minimum error is {min_val} with weights {min_weight}")
```

In [11]:
```
find8optimum(step=0.5)
```

```
Minimum error is 0.36 with weights [0.0, 0.0, 0.0, 0.0, 0.0, 0.5, 0.0, 0.
0]
```

```
/var/folders/v_/7h9hf8f91m9cxmhg0xrmbstr0000gn/T/ipykernel_25892/249885976
3.py:6: RuntimeWarning: invalid value encountered in divide
  out += a*w/norm
```

In [12]:
```
find8optimum(step=0.25)
```

```
/var/folders/v_/7h9hf8f91m9cxmhg0xrmbstr0000gn/T/ipykernel_25892/249885976
3.py:6: RuntimeWarning: invalid value encountered in divide
  out += a*w/norm
Minimum error is 0.32 with weights [0.0, 0.0, 0.0, 0.0, 0.25, 0.0, 0.5, 0.
0]
```

In [13]:
```
find8optimum(step=0.2)
```

```
/var/folders/v_/7h9hf8f91m9cxmhg0xrmbstr0000gn/T/ipykernel_25892/249885976
3.py:6: RuntimeWarning: invalid value encountered in divide
  out += a*w/norm
Minimum error is 0.32 with weights [0.0, 0.0, 0.0, 0.0, 0.2, 0.0, 0.4, 0.
0]
```

In [14]:
```
find8optimum(step=0.15)
```

```
/var/folders/v_/7h9hf8f91m9cxmhg0xrmbstr0000gn/T/ipykernel_25892/249885976
3.py:6: RuntimeWarning: invalid value encountered in divide
  out += a*w/norm
Minimum error is 0.32 with weights [0.0, 0.0, 0.0, 0.0, 0.15, 0.0, 0.3, 0.
0]
```

Result (all nearest neighbors)

Minimum error is 0.32 with weights [0.0, 0.0, 0.0, 0.0, 0.15, 0.0, 0.3, 0.0]

[Open in Colab]

We are following OpenCV's format for run-length encoding where tuples are (start_col, end_col, row)

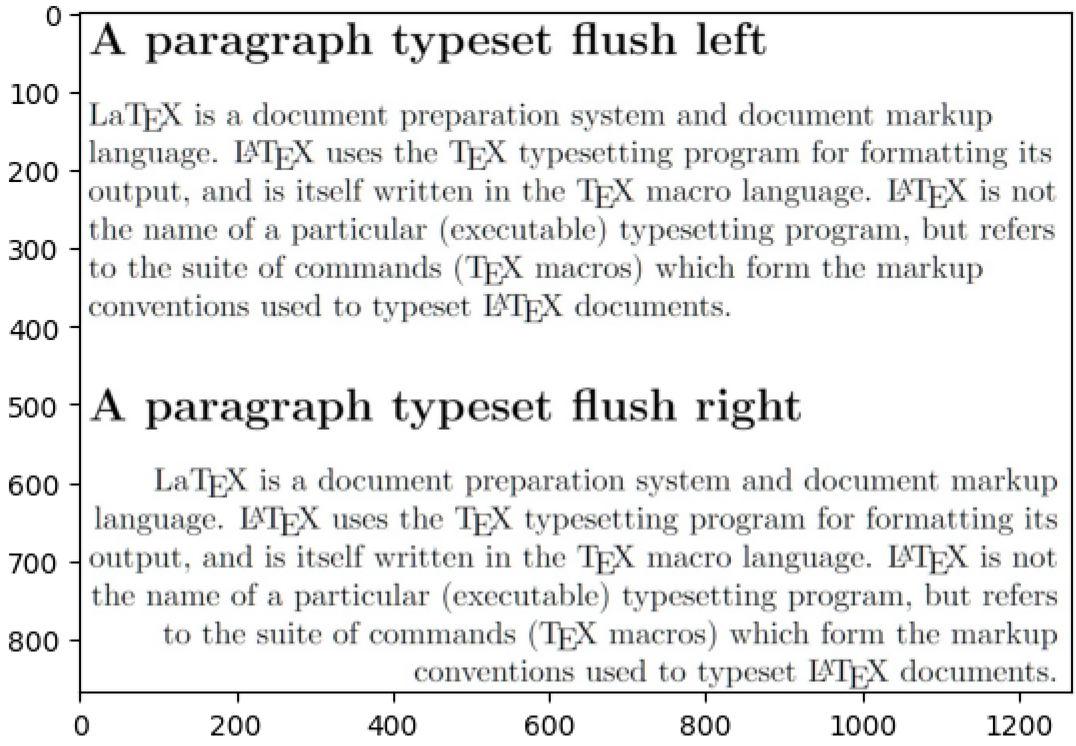
In [1]:
```
def runlength(mat):
    a = []

    for r in range(len(mat)):
        arr=mat[r]
        start=0
        end=0
        isCounting=False
        for i in range(len(arr)):
            if arr[i]==255 and isCounting ==False:
                isCounting=True
                start=i
#                 print("starting",i)

            if arr[i]==0 and isCounting==True:
                end=i-1
                isCounting=False
#                 print("ending", i)
                a.append((start,end,r))

        if arr[len(arr)-1] == 255:
            end=len(arr)-1
            a.append((start,end,r))

    return a
```

In [2]:
```
import cv2 as cv
import numpy as np
import matplotlib.pyplot as plt
```

In [3]:
```
img = cv.imread("img.jpeg")
_, b = cv.threshold(img, 30, 255, cv.THRESH_BINARY)
```

In [6]:
```
plt.imshow(img)
```

Out[6]:
```
<matplotlib.image.AxesImage at 0x17db05610>
```



In [4]:
```
b = np.mean(b,axis=-1,dtype=int)
```

In [5]:
```
runlength(b)
```

```
Out[5]: [(0, 1266, 0),
 (0, 1266, 1),
 (0, 1266, 2),
 (0, 1266, 3),
 (0, 1266, 4),
 (0, 1266, 5),
 (0, 1266, 6),
 (0, 1266, 7),
 (0, 1266, 8),
 (0, 1266, 9),
 (0, 1266, 10),
 (0, 1266, 11),
 (0, 1266, 12),
 (0, 35, 13),
 (37, 647, 13),
 (652, 851, 13),
 (854, 1266, 13),
 (0, 34, 14),
 (38, 342, 14),
 (351, 642, 14),
 (659, 731, 14),
 (740, 789, 14),
 (799, 847, 14),
 (857, 1266, 14),
 (0, 34, 15),
 (38, 342, 15),
 (351, 639, 15),
 (644, 653, 15),
 (659, 731, 15),
 (740, 789, 15),
 (799, 845, 15),
 (850, 854, 15),
 (858, 1266, 15),
 (0, 34, 16),
 (39, 345, 16),
 (351, 638, 16),
 (643, 652, 16),
 (659, 734, 16),
 (740, 793, 16),
 (799, 844, 16),
 (848, 853, 16),
 (859, 1266, 16),
 (0, 33, 17),
 (39, 346, 17),
 (351, 408, 17),
 (410, 591, 17),
 (593, 637, 17),
 (642, 652, 17),
 (659, 734, 17),
 (740, 793, 17),
 (799, 843, 17),
 (848, 853, 17),
 (859, 865, 17),
 (867, 1266, 17),
 (0, 33, 18),
 (40, 346, 18),
 (351, 408, 18),
 (410, 591, 18),
 (593, 636, 18),
 (641, 652, 18),
```

```
(659, 735, 18),
(740, 793, 18),
(799, 842, 18),
(847, 853, 18),
(859, 865, 18),
(867, 1266, 18),
(0, 32, 19),
(40, 346, 19),
(351, 408, 19),
(410, 591, 19),
(593, 635, 19),
(641, 652, 19),
(659, 735, 19),
(740, 793, 19),
(799, 842, 19),
(847, 853, 19),
(859, 865, 19),
(867, 1266, 19),
(0, 32, 20),
(40, 346, 20),
(351, 408, 20),
(410, 591, 20),
(593, 635, 20),
(640, 653, 20),
(659, 735, 20),
(740, 793, 20),
(799, 842, 20),
(847, 854, 20),
(858, 865, 20),
(867, 1266, 20),
(0, 32, 21),
(41, 346, 21),
(351, 407, 21),
(410, 590, 21),
(593, 635, 21),
(640, 653, 21),
(659, 735, 21),
(740, 793, 21),
(799, 841, 21),
(847, 864, 21),
(867, 1266, 21),
(0, 31, 22),
(41, 346, 22),
(351, 407, 22),
(410, 590, 22),
(593, 635, 22),
(640, 653, 22),
(659, 735, 22),
(740, 793, 22),
(799, 841, 22),
(847, 864, 22),
(867, 1266, 22),
(0, 31, 23),
(33, 34, 23),
(42, 346, 23),
(351, 407, 23),
(410, 590, 23),
(593, 635, 23),
(640, 653, 23),
(659, 735, 23),
```

```
(740, 793, 23),
(799, 841, 23),
(847, 864, 23),
(867, 1266, 23),
(0, 30, 24),
(33, 35, 24),
(42, 346, 24),
(351, 406, 24),
(410, 589, 24),
(593, 635, 24),
(640, 653, 24),
(659, 735, 24),
(740, 793, 24),
(799, 841, 24),
(847, 863, 24),
(867, 1266, 24),
(0, 30, 25),
(32, 35, 25),
(42, 346, 25),
(351, 405, 25),
(410, 588, 25),
(593, 635, 25),
(640, 653, 25),
(659, 735, 25),
(740, 793, 25),
(799, 841, 25),
(847, 862, 25),
(867, 1266, 25),
(0, 30, 26),
(32, 35, 26),
(43, 346, 26),
(351, 405, 26),
(410, 587, 26),
(593, 635, 26),
(640, 653, 26),
(659, 735, 26),
(740, 793, 26),
(799, 841, 26),
(847, 862, 26),
(867, 1266, 26),
(0, 29, 27),
(32, 36, 27),
(43, 346, 27),
(351, 403, 27),
(410, 586, 27),
(593, 635, 27),
(640, 653, 27),
(659, 735, 27),
(740, 793, 27),
(799, 841, 27),
(847, 860, 27),
(867, 1266, 27),
(0, 29, 28),
(31, 36, 28),
(44, 85, 28),
(95, 100, 28),
(108, 128, 28),
(140, 154, 28),
(163, 168, 28),
(174, 187, 28),
```

```
(199, 221, 28),
(229, 235, 28),
(241, 246, 28),
(255, 260, 28),
(265, 279, 28),
(290, 305, 28),
(314, 320, 28),
(328, 346, 28),
(351, 358, 28),
(365, 401, 28),
(418, 423, 28),
(425, 434, 28),
(436, 445, 28),
(448, 453, 28),
(455, 459, 28),
(468, 473, 28),
(481, 506, 28),
(516, 532, 28),
(542, 544, 28),
(547, 562, 28),
(572, 584, 28),
(601, 631, 28),
(659, 668, 28),
(678, 686, 28),
(696, 710, 28),
(719, 721, 28),
(724, 735, 28),
(740, 747, 28),
(754, 793, 28),
(799, 815, 28),
(825, 837, 28),
(854, 858, 28),
(875, 1266, 28),
(0, 28, 29),
(31, 37, 29),
(44, 85, 29),
(95, 98, 29),
(102, 104, 29),
(111, 126, 29),
(130, 136, 29),
(142, 154, 29),
(163, 167, 29),
(170, 171, 29),
(175, 185, 29),
(189, 195, 29),
(201, 218, 29),
(223, 227, 29),
(232, 233, 29),
(236, 238, 29),
(242, 246, 29),
(255, 258, 29),
(261, 263, 29),
(267, 277, 29),
(281, 287, 29),
(293, 305, 29),
(314, 317, 29),
(321, 324, 29),
(330, 346, 29),
(351, 356, 29),
(359, 362, 29),
```

```
(367, 399, 29),
(418, 423, 29),
(436, 445, 29),
(455, 459, 29),
(468, 471, 29),
(475, 478, 29),
(484, 503, 29),
(508, 513, 29),
(518, 530, 29),
(533, 541, 29),
(547, 559, 29),
(564, 569, 29),
(574, 582, 29),
(601, 631, 29),
(659, 668, 29),
(678, 686, 29),
(696, 707, 29),
(711, 718, 29),
(724, 735, 29),
(740, 745, 29),
(747, 751, 29),
(756, 793, 29),
(799, 813, 29),
(818, 823, 29),
(828, 837, 29),
(854, 856, 29),
(875, 1266, 29),
(0, 28, 30),
(30, 37, 30),
(44, 88, 30),
(95, 96, 30),
(99, 106, 30),
(112, 125, 30),
(129, 138, 30),
(143, 157, 30),
(163, 171, 30),
(176, 184, 30),
(188, 197, 30),
(203, 217, 30),
(221, 229, 30),
(234, 238, 30),
(242, 249, 30),
(255, 257, 30),
(259, 262, 30),
(268, 276, 30),
(281, 289, 30),
(295, 308, 30),
(314, 315, 30),
(318, 326, 30),
(332, 346, 30),
(351, 354, 30),
(356, 363, 30),
(368, 404, 30),
(410, 427, 30),
(433, 447, 30),
(451, 461, 30),
(468, 469, 30),
(472, 480, 30),
(486, 502, 30),
(507, 515, 30),
```

```
(519, 529, 30),
(532, 543, 30),
(547, 558, 30),
(563, 571, 30),
(575, 587, 30),
(593, 635, 30),
(641, 653, 30),
(659, 671, 30),
(678, 689, 30),
(696, 706, 30),
(709, 720, 30),
(724, 735, 30),
(740, 743, 30),
(745, 752, 30),
(757, 793, 30),
(799, 811, 30),
(816, 825, 30),
(829, 841, 30),
(847, 861, 30),
(867, 1266, 30),
(0, 28, 31),
(30, 37, 31),
(45, 89, 31),
(97, 108, 31),
(113, 124, 31),
(130, 139, 31),
(145, 158, 31),
(163, 165, 31),
(167, 170, 31),
(177, 183, 31),
(189, 198, 31),
(204, 216, 31),
(221, 230, 31),
(234, 238, 31),
(242, 250, 31),
(255, 262, 31),
(268, 275, 31),
(281, 290, 31),
(296, 309, 31),
(317, 327, 31),
(333, 346, 31),
(351, 353, 31),
(355, 363, 31),
(369, 404, 31),
(410, 427, 31),
(434, 448, 31),
(450, 463, 31),
(470, 481, 31),
(487, 500, 31),
(506, 516, 31),
(520, 528, 31),
(531, 544, 31),
(547, 556, 31),
(562, 572, 31),
(576, 587, 31),
(593, 635, 31),
(640, 653, 31),
(659, 672, 31),
(678, 690, 31),
(696, 705, 31),
```

```
(708, 721, 31),
(724, 735, 31),
(740, 742, 31),
(744, 752, 31),
(758, 793, 31),
(799, 810, 31),
(815, 825, 31),
(830, 841, 31),
(847, 861, 31),
(867, 1266, 31),
(0, 27, 32),
(30, 38, 32),
(45, 89, 32),
(96, 108, 32),
(114, 124, 32),
(130, 140, 32),
(145, 158, 32),
(163, 164, 32),
(166, 170, 32),
(177, 183, 32),
(189, 199, 32),
(204, 215, 32),
(220, 230, 32),
(235, 250, 32),
(255, 262, 32),
(268, 275, 32),
(281, 291, 32),
(296, 309, 32),
(315, 328, 32),
(334, 346, 32),
(351, 352, 32),
(354, 364, 32),
(369, 404, 32),
(410, 428, 32),
(434, 448, 32),
(450, 463, 32),
(469, 482, 32),
(488, 500, 32),
(505, 516, 32),
(521, 528, 32),
(531, 544, 32),
(547, 556, 32),
(561, 572, 32),
(577, 587, 32),
(593, 635, 32),
(640, 653, 32),
(659, 672, 32),
(678, 690, 32),
(696, 705, 32),
(708, 721, 32),
(724, 735, 32),
(740, 741, 32),
(743, 753, 32),
(758, 793, 32),
(799, 809, 32),
(814, 826, 32),
(831, 841, 32),
(847, 861, 32),
(867, 1266, 32),
(0, 27, 33),
```

```
(29, 38, 33),
(46, 89, 33),
(95, 109, 33),
(115, 124, 33),
(130, 140, 33),
(146, 158, 33),
(163, 171, 33),
(176, 183, 33),
(189, 199, 33),
(205, 215, 33),
(220, 230, 33),
(236, 250, 33),
(255, 262, 33),
(268, 275, 33),
(281, 291, 33),
(297, 309, 33),
(315, 328, 33),
(335, 346, 33),
(352, 364, 33),
(370, 404, 33),
(410, 428, 33),
(435, 447, 33),
(449, 463, 33),
(469, 482, 33),
(488, 499, 33),
(504, 517, 33),
(521, 527, 33),
(531, 545, 33),
(547, 555, 33),
(560, 573, 33),
(577, 587, 33),
(593, 635, 33),
(640, 653, 33),
(659, 672, 33),
(678, 690, 33),
(696, 705, 33),
(708, 722, 33),
(724, 735, 33),
(741, 753, 33),
(758, 793, 33),
(799, 809, 33),
(814, 827, 33),
(831, 841, 33),
(847, 861, 33),
(867, 1266, 33),
(0, 26, 34),
(29, 39, 34),
(46, 89, 34),
(95, 110, 34),
(116, 125, 34),
(130, 140, 34),
(146, 158, 34),
(165, 171, 34),
(176, 183, 34),
(189, 199, 34),
(205, 214, 34),
(220, 230, 34),
(236, 250, 34),
(256, 263, 34),
(267, 276, 34),
```

```
(281, 291, 34),
(297, 309, 34),
(315, 329, 34),
(335, 346, 34),
(353, 364, 34),
(370, 404, 34),
(410, 429, 34),
(435, 447, 34),
(449, 463, 34),
(469, 483, 34),
(489, 498, 34),
(504, 517, 34),
(522, 527, 34),
(531, 545, 34),
(547, 554, 34),
(560, 573, 34),
(578, 587, 34),
(593, 635, 34),
(640, 653, 34),
(659, 672, 34),
(678, 690, 34),
(696, 704, 34),
(708, 722, 34),
(724, 735, 34),
(741, 753, 34),
(759, 793, 34),
(799, 808, 34),
(814, 827, 34),
(832, 841, 34),
(847, 861, 34),
(867, 1266, 34),
(0, 26, 35),
(28, 39, 35),
(46, 89, 35),
(95, 110, 35),
(116, 126, 35),
(129, 140, 35),
(146, 158, 35),
(164, 185, 35),
(188, 199, 35),
(205, 214, 35),
(220, 230, 35),
(236, 250, 35),
(256, 277, 35),
(280, 291, 35),
(297, 309, 35),
(315, 329, 35),
(336, 346, 35),
(352, 364, 35),
(370, 404, 35),
(410, 429, 35),
(435, 446, 35),
(449, 463, 35),
(469, 483, 35),
(489, 498, 35),
(504, 517, 35),
(522, 527, 35),
(532, 554, 35),
(560, 573, 35),
(578, 587, 35),
```

```
(593, 635, 35),
(640, 653, 35),
(659, 672, 35),
(678, 690, 35),
(696, 704, 35),
(709, 735, 35),
(741, 753, 35),
(759, 793, 35),
(799, 808, 35),
(813, 827, 35),
(832, 841, 35),
(847, 861, 35),
(867, 1266, 35),
(0, 25, 36),
(28, 39, 36),
(47, 89, 36),
(95, 110, 36),
(116, 140, 36),
(146, 158, 36),
(164, 199, 36),
(205, 214, 36),
(220, 230, 36),
(236, 250, 36),
(256, 291, 36),
(297, 309, 36),
(315, 330, 36),
(336, 346, 36),
(352, 364, 36),
(370, 404, 36),
(410, 430, 36),
(436, 446, 36),
(448, 463, 36),
(469, 483, 36),
(490, 498, 36),
(504, 518, 36),
(522, 527, 36),
(534, 554, 36),
(560, 574, 36),
(578, 587, 36),
(593, 635, 36),
(640, 653, 36),
(659, 672, 36),
(678, 690, 36),
(696, 704, 36),
(711, 735, 36),
(741, 753, 36),
(759, 793, 36),
(799, 807, 36),
(813, 827, 36),
(832, 841, 36),
(847, 861, 36),
(867, 1266, 36),
(0, 25, 37),
(27, 40, 37),
(47, 89, 37),
(95, 110, 37),
(117, 140, 37),
(146, 158, 37),
(164, 199, 37),
(205, 214, 37),
```

```
(220, 230, 37),
(236, 250, 37),
(256, 291, 37),
(297, 309, 37),
(315, 330, 37),
(336, 346, 37),
(352, 364, 37),
(370, 404, 37),
(410, 430, 37),
(436, 445, 37),
(448, 463, 37),
(469, 484, 37),
(490, 497, 37),
(504, 518, 37),
(523, 528, 37),
(538, 553, 37),
(560, 574, 37),
(579, 587, 37),
(593, 635, 37),
(640, 653, 37),
(659, 672, 37),
(678, 690, 37),
(696, 705, 37),
(715, 735, 37),
(741, 753, 37),
(759, 793, 37),
(799, 807, 37),
(813, 827, 37),
(832, 841, 37),
(847, 861, 37),
(867, 1266, 37),
(0, 25, 38),
(27, 40, 38),
(48, 89, 38),
(95, 110, 38),
(117, 140, 38),
(146, 158, 38),
(164, 199, 38),
(205, 214, 38),
(220, 230, 38),
(236, 250, 38),
(256, 291, 38),
(297, 309, 38),
(315, 330, 38),
(336, 346, 38),
(352, 364, 38),
(370, 404, 38),
(410, 431, 38),
(437, 445, 38),
(447, 463, 38),
(469, 484, 38),
(490, 497, 38),
(503, 518, 38),
(523, 528, 38),
(543, 553, 38),
(559, 574, 38),
(579, 587, 38),
(593, 635, 38),
(640, 653, 38),
(659, 672, 38),
```

```
(678, 690, 38),
(696, 705, 38),
(720, 735, 38),
(740, 753, 38),
(759, 793, 38),
(799, 807, 38),
(813, 828, 38),
(832, 841, 38),
(847, 861, 38),
(867, 1266, 38),
(0, 24, 39),
(27, 41, 39),
(48, 89, 39),
(95, 111, 39),
(117, 134, 39),
(146, 158, 39),
(164, 193, 39),
(205, 215, 39),
(220, 230, 39),
(235, 250, 39),
(255, 285, 39),
(297, 309, 39),
(315, 330, 39),
(337, 346, 39),
(352, 364, 39),
(370, 404, 39),
(410, 431, 39),
(437, 444, 39),
(447, 463, 39),
(469, 484, 39),
(490, 497, 39),
(504, 518, 39),
(523, 529, 39),
(545, 553, 39),
(560, 574, 39),
(579, 587, 39),
(593, 635, 39),
(640, 653, 39),
(659, 672, 39),
(678, 690, 39),
(696, 706, 39),
(722, 735, 39),
(740, 753, 39),
(759, 793, 39),
(799, 807, 39),
(813, 827, 39),
(832, 841, 39),
(847, 861, 39),
(867, 1266, 39),
(0, 24, 40),
(49, 89, 40),
(95, 111, 40),
(117, 129, 40),
(136, 140, 40),
(146, 158, 40),
(164, 188, 40),
(195, 199, 40),
(205, 215, 40),
(220, 230, 40),
(235, 250, 40),
```

```
(255, 280, 40),
(287, 291, 40),
(297, 309, 40),
(315, 330, 40),
(337, 346, 40),
(352, 364, 40),
(370, 404, 40),
(410, 432, 40),
(438, 444, 40),
(446, 463, 40),
(469, 484, 40),
(490, 497, 40),
(523, 530, 40),
(546, 553, 40),
(579, 587, 40),
(593, 635, 40),
(640, 653, 40),
(659, 672, 40),
(678, 690, 40),
(696, 707, 40),
(723, 735, 40),
(740, 753, 40),
(759, 793, 40),
(799, 807, 40),
(832, 841, 40),
(847, 861, 40),
(867, 1266, 40),
(0, 23, 41),
(49, 89, 41),
(95, 111, 41),
(117, 127, 41),
(133, 140, 41),
(146, 158, 41),
(164, 186, 41),
(192, 199, 41),
(205, 216, 41),
(221, 229, 41),
(234, 250, 41),
(255, 278, 41),
(284, 291, 41),
(297, 309, 41),
(315, 330, 41),
(337, 346, 41),
(352, 364, 41),
(370, 404, 41),
(410, 432, 41),
(438, 444, 41),
(446, 463, 41),
(469, 484, 41),
(490, 497, 41),
(503, 532, 41),
(547, 553, 41),
(559, 587, 41),
(593, 635, 41),
(640, 653, 41),
(659, 672, 41),
(678, 690, 41),
(696, 710, 41),
(724, 735, 41),
(740, 753, 41),
```

```
(759, 793, 41),
(799, 807, 41),
(813, 841, 41),
(847, 861, 41),
(867, 1266, 41),
(0, 23, 42),
(25, 42, 42),
(49, 89, 42),
(95, 110, 42),
(117, 125, 42),
(131, 140, 42),
(146, 158, 42),
(164, 184, 42),
(190, 199, 42),
(205, 217, 42),
(222, 228, 42),
(233, 250, 42),
(255, 276, 42),
(282, 291, 42),
(297, 309, 42),
(315, 330, 42),
(337, 346, 42),
(352, 364, 42),
(370, 404, 42),
(410, 432, 42),
(439, 443, 42),
(445, 463, 42),
(469, 484, 42),
(490, 497, 42),
(503, 537, 42),
(547, 553, 42),
(559, 587, 42),
(593, 635, 42),
(640, 653, 42),
(659, 672, 42),
(678, 690, 42),
(696, 714, 42),
(724, 735, 42),
(740, 753, 42),
(759, 793, 42),
(799, 807, 42),
(813, 841, 42),
(847, 861, 42),
(867, 1266, 42),
(0, 23, 43),
(25, 42, 43),
(50, 89, 43),
(95, 110, 43),
(117, 124, 43),
(130, 140, 43),
(146, 158, 43),
(164, 183, 43),
(189, 199, 43),
(205, 216, 43),
(218, 219, 43),
(231, 250, 43),
(255, 275, 43),
(280, 291, 43),
(297, 309, 43),
(315, 330, 43),
```

```
(336, 346, 43),
(352, 364, 43),
(370, 404, 43),
(410, 433, 43),
(439, 443, 43),
(445, 463, 43),
(469, 484, 43),
(490, 497, 43),
(504, 541, 43),
(548, 553, 43),
(560, 587, 43),
(593, 635, 43),
(640, 653, 43),
(659, 672, 43),
(678, 690, 43),
(696, 718, 43),
(725, 735, 43),
(740, 753, 43),
(759, 793, 43),
(799, 807, 43),
(813, 841, 43),
(847, 861, 43),
(867, 1266, 43),
(0, 22, 44),
(25, 43, 44),
(50, 89, 44),
(95, 110, 44),
(117, 123, 44),
(129, 140, 44),
(146, 158, 44),
(164, 182, 44),
(188, 199, 44),
(205, 250, 44),
(255, 274, 44),
(280, 291, 44),
(297, 309, 44),
(315, 330, 44),
(336, 346, 44),
(352, 364, 44),
(370, 404, 44),
(410, 433, 44),
(440, 442, 44),
(444, 463, 44),
(469, 484, 44),
(490, 497, 44),
(504, 543, 44),
(548, 553, 44),
(560, 587, 44),
(593, 635, 44),
(640, 653, 44),
(659, 672, 44),
(678, 690, 44),
(696, 720, 44),
(725, 735, 44),
(740, 753, 44),
(759, 793, 44),
(799, 807, 44),
(813, 841, 44),
(847, 861, 44),
(867, 1266, 44),
```

```
(0, 22, 45),
(24, 43, 45),
(51, 89, 45),
(95, 110, 45),
(116, 122, 45),
(128, 140, 45),
(146, 158, 45),
(164, 181, 45),
(187, 199, 45),
(205, 215, 45),
(217, 250, 45),
(255, 273, 45),
(279, 291, 45),
(297, 309, 45),
(315, 329, 45),
(336, 346, 45),
(352, 364, 45),
(370, 404, 45),
(410, 434, 45),
(440, 442, 45),
(444, 463, 45),
(469, 483, 45),
(490, 498, 45),
(504, 527, 45),
(529, 544, 45),
(548, 554, 45),
(560, 587, 45),
(593, 635, 45),
(640, 653, 45),
(659, 672, 45),
(678, 690, 45),
(696, 704, 45),
(706, 721, 45),
(725, 735, 45),
(740, 753, 45),
(759, 793, 45),
(799, 807, 45),
(813, 841, 45),
(847, 861, 45),
(867, 1266, 45),
(0, 21, 46),
(24, 43, 46),
(51, 89, 46),
(95, 110, 46),
(116, 122, 46),
(128, 140, 46),
(146, 158, 46),
(164, 181, 46),
(187, 199, 46),
(205, 215, 46),
(217, 250, 46),
(255, 273, 46),
(279, 291, 46),
(297, 309, 46),
(315, 329, 46),
(336, 346, 46),
(352, 364, 46),
(370, 404, 46),
(410, 434, 46),
(441, 441, 46),
```

```
(443, 463, 46),
(469, 483, 46),
(489, 498, 46),
(504, 527, 46),
(530, 545, 46),
(548, 554, 46),
(560, 587, 46),
(593, 635, 46),
(640, 653, 46),
(659, 672, 46),
(678, 690, 46),
(696, 704, 46),
(707, 722, 46),
(725, 735, 46),
(740, 753, 46),
(759, 793, 46),
(799, 808, 46),
(814, 841, 46),
(847, 861, 46),
(867, 1266, 46),
(0, 21, 47),
(23, 44, 47),
(51, 89, 47),
(95, 109, 47),
(116, 122, 47),
(128, 140, 47),
(146, 158, 47),
(164, 181, 47),
(187, 199, 47),
(205, 215, 47),
(218, 250, 47),
(255, 273, 47),
(279, 291, 47),
(297, 309, 47),
(315, 329, 47),
(335, 346, 47),
(352, 364, 47),
(370, 404, 47),
(410, 417, 47),
(419, 435, 47),
...]
```

In [ ]:

```

```

In [ ]:

```

```

Huffman coding is a variable-length prefix coding technique. It relies on creating a variable-length code for each symbol (in this case, pixel values) based on their frequency of occurrence in the image. It does not involve any mathematical transforms.

In [60]:
```
import heapq
import collections
from PIL import Image

image_path = r"img.jpeg"
image = Image.open(image_path)
image = image.resize((1024, 1024))

pixel_values = list(image.getdata())

frq = collections.Counter(pixel_values)

heap = [[weight, [pixel, ""]] for pixel, weight in frq.items()]
heapq.heapify(heap)

while len(heap) > 1:
    lo = heapq.heappop(heap)
    hi = heapq.heappop(heap)
    for pair in lo[1:]:
        pair[1] = '0' + pair[1]
    for pair in hi[1:]:
        pair[1] = '1' + pair[1]
    heapq.heappush(heap, [lo[0] + hi[0]] + lo[1:] + hi[1:])

huffman_dict = dict(heap[0][1:])

encoding = ''.join(huffman_dict[pixel] for pixel in pixel_values)

original_size = len(pixel_values) * 8  # Assuming 8 bits per pixel
compressed_size = len(encoding)

huffman_compression_ratio = original_size / compressed_size
print(f"Huffman Compression Ratio: {huffman_compression_ratio:}")
```

```
Huffman Compression Ratio: 2.488950814740001
```

DCT is a mathematical transform that converts image data into a frequency-domain representation. It is widely used in JPEG compression. DCT captures the frequency components of an image, allowing for efficient compression by quantizing high-frequency components.

In [65]:
```
import numpy as np
from scipy.fftpack import dct
from PIL import Image

image_path = r'img.jpeg'
image = Image.open(image_path)
# print(image.size)
image = image.resize((1024, 1024))

gray = image.convert("L")
```

```
imarr = np.array(gray)
# print(imarr.shape)
block_size = 8

quantmat = np.array([[16, 11, 10, 16, 24, 40, 51, 61],
                     [12, 12, 14, 19, 26, 58, 60, 55],
                     [14, 13, 16, 24, 40, 57, 69, 56],
                     [14, 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]])

def quantize(k, Q):
    return np.round(k / Q)

def dequantize(l, Q):
    return l * Q

height, width = imarr.shape
dctb = np.zeros_like(imarr)

for i in range(0, height, block_size):
    for j in range(0, width, block_size):
        block = imarr[i:i + block_size, j:j + block_size]
        dct_block = dct(dct(block, axis=0), axis=1)

        qb = quantize(dct_block, quantmat)
        dctb[i:i + block_size, j:j + block_size] = qb

original = height * width * 8
compressed = dctb.size * np.log2(quantmat.max())

compression_ratio = original / compressed

print(f"Compression Ratio: {compression_ratio:}")
```

```
Compression Ratio: 1.1562593052715515
```

KL transform (Karhunen-Loève transform) is a linear transformation that converts data into a set of uncorrelated variables (principal components). It is used for decorrelation and dimensionality reduction.

In [62]:

```
import matplotlib.pyplot

image_path = r'img.jpeg'
image = Image.open(image_path)
image = image.resize((1024, 1024))

gray = image.convert("L")
imarr = np.array(gray)
covariance_matrix = np.cov(imarr.astype(float))
e1, e = np.linalg.eigh(covariance_matrix)

ind = np.argsort(e1)[::-1]
e1 = e1[ind]
e = e[:, ind]

compression_ratio = 0.1
```

```
eign = int(compression_ratio * len(e1))
sel = e[:, :eign]
compressed_image = np.dot(sel.T, imarr.T).T
newimage = np.dot(compressed_image, sel.T)

original = imarr.size * 8
compressed = eign * (len(e1) + 1) * 64

compression_efficiency = original / compressed

print(f"Compression Efficiency: {compression_efficiency:.2f}")

# matplotlib.pyplot.imshow(image, cmap='gray')
# matplotlib.pyplot.imshow(newimage, cmap='gray')
```

```
Compression Efficiency: 1.25
```

Haar wavelet transform is a mathematical technique that decomposes an image into wavelet coefficients representing different levels of detail.

In [64]:

```
import numpy as np
from PIL import Image
import pywt
import sys

image_path = r'img.jpeg'
image = Image.open(image_path)
image = image.resize((1024, 1024))
gray = image.convert("L")

imarr = np.array(gray)

coeffs = pywt.dwt2(imarr, 'haar')
threshold = 15.0
qc = [np.where(np.abs(coef) < threshold, 0, coef) for coef in coeffs]

reconstructed_image = pywt.idwt2(qc, 'haar')

original_size_bits = imarr.size * 8
compressed_size_bits = sum([np.sum(np.abs(coef) > 0) * np.ceil(np.log2(np

compression_efficiency = original_size_bits / compressed_size_bits

print(f"Compression Efficiency: {compression_efficiency:.2f}")
```

```
Compression Efficiency: 2.79
```

Conclusion:

Huffman Coding: Huffman coding is typically used for lossless compression, so it preserves image quality but may not achieve very high compression ratios.

DCT Coding: DCT coding is often used for lossy compression. The trade-off between image quality and compression ratio can be adjusted by varying the quantization step size.

KL Transform-Based Coding: KL transform can be used for both lossless and lossy compression, offering a flexible trade-off between image quality and compression ratio.

Haar Wavelet Compression: Haar wavelet compression can also be used for both lossless and lossy compression, providing control over image quality and compression ratio.