DAFTAR PUSTAKA

Asamoah, D., Ofori, E., Opoku, S., & Danso, J. (2018). Measuring the Performance of Image Contrast Enhancement Technique. *International Journal of Computer Applications*, *181*(22), 6–13. https://doi.org/10.5120/ijca2018917899

Awcock, G. J., & Thomas, R. (1995). *Applied Image Processing*. Macmillan Education UK. https://doi.org/10.1007/978-1-349-13049-8

Dorland, W. A. N. (2010). *Kamus Kedokteran Dorland Edisi 31* (R. N. Elseria & A. A. Mahode (eds.)). EGC.

Esakandari, H., Nabi-Afjadi, M., Fakkari-Afjadi, J., Farahmandian, N., Miresmaeili, S. M., & Bahreini, E. (2020). A comprehensive review of COVID-19 characteristics. *Biological Procedures Online*, *22*(1), 1–10. https://doi.org/10.1186/s12575-020-00128-2

Kaundal, A. K. (2014). *Feistel Inspired Structure for DNA Cryptography* (Issue June). Thapar University.

Maniyath, S. R., & Supriya, M. (2011). An Uncompressed Image Encryption Algorithm Based on DNA Sequences. *Image (Rochester, N.Y.)*, 258–270. https://doi.org/10.5121/csit.2011.1224

Menezes, A. J., Oorschot, P. C. van, & Vanstone, S. A. (1996). *Handbook of Applied Cryptography*. CRC Press.

National Institute of Biomedical Imaging. (2019). Computed Tomography Fact Sheet. *National Institutes of Health*. www.nibib.nih.gov

Omer, A., & Farooq, M. I. (2015). *DNA Cryptography Algorithms and Applications*. *July*. https://doi.org/10.13140/RG.2.1.4892.5289

Paar, C., & Pelzl, J. (2009). *Understanding Cryptography*. Springer-Verlag Berlin Heidelberg.

Paul, S., Anwar, T., & Kumar, A. (2016). An Innovative DNA Cryptography Technique for Secure Data Transmission. *International Journal of Bioinformatics Research and Applications*, *12*(3), 238–262. https://doi.org/10.1504/IJBRA.2016.078235

Putra, D. (2010). *Pengolahan Citra Digital*. CV. Andi Offset.

Stallings, W. (2017). *Cryptography and Network Security: Principles and Practice* (7th ed.). Pearson Education.

Terec, R., Vaida, M., Alboaie, L., & Chiorean, L. (2011). *DNA Security using Symmetric and Asymmetric Cryptography*. *1*(1), 34–51.

Tim Administrator Situs Kawal COVID 19. (2020). *Mencegah dan Menangani Stigma Sosial Seputar COVID-19*. Kawal COVID19.Id. https://kawalcovid19.id/

Wang, X. Y., Zhang, Y. Q., & Bao, X. M. (2015). A Novel Chaotic Image Encryption Scheme using DNA Sequence Operations. *Optics and Lasers in Engineering*, *73*, 53–61. https://doi.org/10.1016/j.optlaseng.2015.03.022

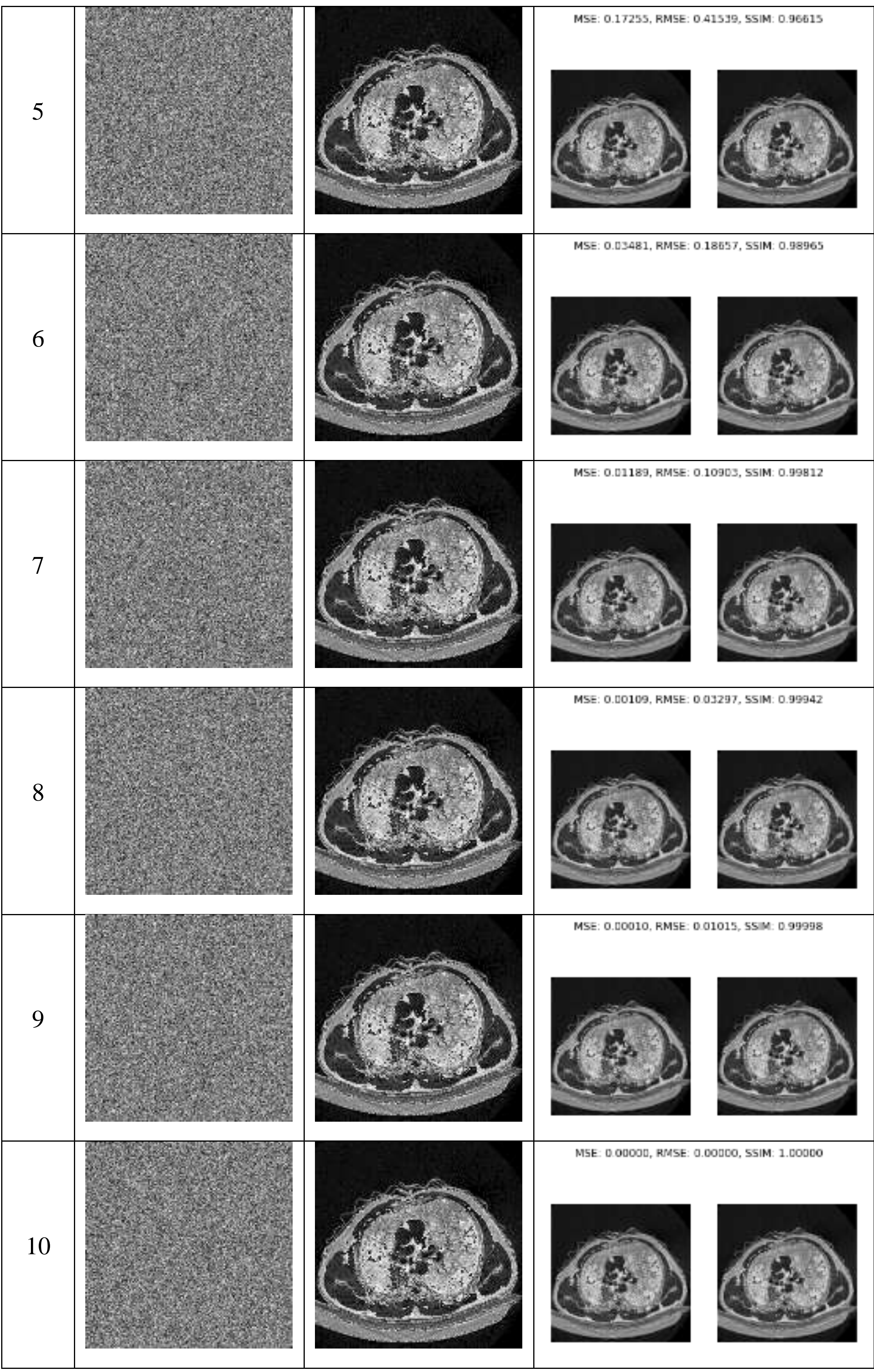
Wang, Z., Bovik, A. C., Sheikh, H. R., & Simoncelli, E. P. (2004). Image Quality Assessment: From Error Visibility to Structural Similarity. *IEEE Transactions on Image Processing*, *13*(4), 600–612.

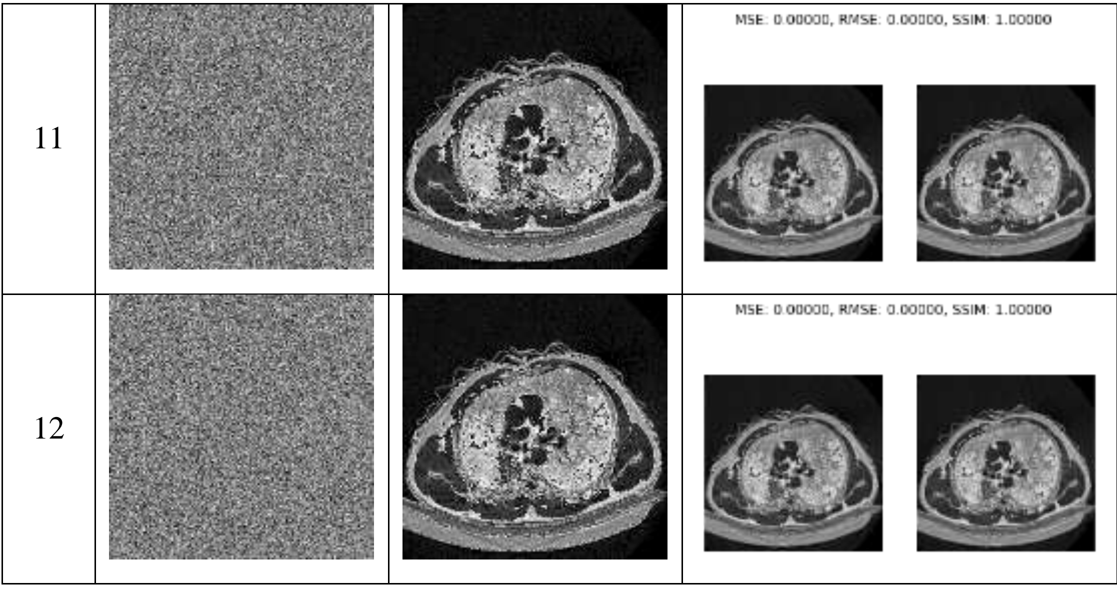
World Health Organization (2020). *Coronavirus disease (COVID-19) Q&As: What are the symptomps of COVID-19?*. October 12, 2020. https://www.who.int/emergencies/diseases/novel-coronavirus-2019/question-and-answers-hub/q-a-detail/coronavirus-disease-covid-19

LAMPIRAN

Lampiran 1: Tabel citra hasil enkripsi, dekripsi, dan uji kerusakan masing-masing nilai *K*

Nilai *K*	Hasil enkripsi	Hasil dekripsi	Uji kerusakan
1			MSE: 76.20198, RMSE: 8.72937, SSIM: 0.10370
2			MSE: 13.88363, RMSE: 3.72607, SSIM: 0.40668
3			MSE: 2.81999, RMSE: 1.67928, SSIM: 0.66531
4			MSE: 0.64149, RMSE: 0.80093, SSIM: 0.87871


5
MSE: 0.17255, RMSE: 0.41539, SSIM: 0.96615
6
MSE: 0.03481, RMSE: 0.18657, SSIM: 0.98965
7
MSE: 0.01189, RMSE: 0.10903, SSIM: 0.99812
8
MSE: 0.00109, RMSE: 0.03297, SSIM: 0.99942
9
MSE: 0.00010, RMSE: 0.01015, SSIM: 0.99998
10
MSE: 0.00000, RMSE: 0.00000, SSIM: 1.00000


11
MSE: 0.00000, RMSE: 0.00000, SSIM: 1.00000
12
MSE: 0.00000, RMSE: 0.00000, SSIM: 1.00000

Lampiran 2: Generate_numbers.ipynb

```
g = [23] # g is prime number
n = [29] # n is prime number
x, y, K = [], [], []

for a in n:
  for b in g:
    for c in range (1, 10):
      for d in range (1,10):
        x.append(c)
        y.append(d)
        R = (b*c) % a
        S = (b*d) % a
        K1 = (S*c) % a
        K2 = (R*d) % a
        K.append(K1)
print(g)
print(n)
print(x)
print(y)
print(K)
```

```
Output:

#2 and 3

[2]
[3]
[1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 3, 3,
3, 3, 3, 3, 3, 3, 3, 4, 4, 4, 4, 4, 4, 4, 4, 4, 5, 5, 5, 5, 5,
5, 5, 5, 5, 6, 6, 6, 6, 6, 6, 6, 6, 6, 7, 7, 7, 7, 7, 7, 7, 7,
7, 8, 8, 8, 8, 8, 8, 8, 8, 8, 9, 9, 9, 9, 9, 9, 9, 9, 9]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 1, 2, 3, 4, 5, 6, 7, 8, 9, 1, 2,
3, 4, 5, 6, 7, 8, 9, 1, 2, 3, 4, 5, 6, 7, 8, 9, 1, 2, 3, 4, 5,
6, 7, 8, 9, 1, 2, 3, 4, 5, 6, 7, 8, 9, 1, 2, 3, 4, 5, 6, 7, 8,
9, 1, 2, 3, 4, 5, 6, 7, 8, 9, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[2, 1, 0, 2, 1, 0, 2, 1, 0, 1, 2, 0, 1, 2, 0, 1, 2, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 2, 1, 0, 2, 1, 0, 2, 1, 0, 1, 2, 0, 1, 2,
0, 1, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 1, 0, 2, 1, 0, 2, 1,
0, 1, 2, 0, 1, 2, 0, 1, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]


#5 and 7

[5]
[7]
[1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 3, 3,
3, 3, 3, 3, 3, 3, 3, 4, 4, 4, 4, 4, 4, 4, 4, 4, 5, 5, 5, 5, 5,
5, 5, 5, 5, 6, 6, 6, 6, 6, 6, 6, 6, 6, 7, 7, 7, 7, 7, 7, 7, 7,
7, 8, 8, 8, 8, 8, 8, 8, 8, 8, 9, 9, 9, 9, 9, 9, 9, 9, 9]
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 1, 2, 3, 4, 5, 6, 7, 8, 9, 1, 2,
3, 4, 5, 6, 7, 8, 9, 1, 2, 3, 4, 5, 6, 7, 8, 9, 1, 2, 3, 4, 5,
6, 7, 8, 9, 1, 2, 3, 4, 5, 6, 7, 8, 9, 1, 2, 3, 4, 5, 6, 7, 8,
9, 1, 2, 3, 4, 5, 6, 7, 8, 9, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[5, 3, 1, 6, 4, 2, 0, 5, 3, 3, 6, 2, 5, 1, 4, 0, 3, 6, 1, 2,
3, 4, 5, 6, 0, 1, 2, 6, 5, 4, 3, 2, 1, 0, 6, 5, 4, 1, 5, 2, 6,
3, 0, 4, 1, 2, 4, 6, 1, 3, 5, 0, 2, 4, 0, 0, 0, 0, 0, 0, 0, 0,
0, 5, 3, 1, 6, 4, 2, 0, 5, 3, 3, 6, 2, 5, 1, 4, 0, 3, 6]


#11 and 13

[11]
[13]
[1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 3, 3,
3, 3, 3, 3, 3, 3, 3, 4, 4, 4, 4, 4, 4, 4, 4, 4, 5, 5, 5, 5, 5,
5, 5, 5, 5, 6, 6, 6, 6, 6, 6, 6, 6, 6, 7, 7, 7, 7, 7, 7, 7, 7,
7, 8, 8, 8, 8, 8, 8, 8, 8, 8, 9, 9, 9, 9, 9, 9, 9, 9, 9]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 1, 2, 3, 4, 5, 6, 7, 8, 9, 1, 2,
3, 4, 5, 6, 7, 8, 9, 1, 2, 3, 4, 5, 6, 7, 8, 9, 1, 2, 3, 4, 5,
6, 7, 8, 9, 1, 2, 3, 4, 5, 6, 7, 8, 9, 1, 2, 3, 4, 5, 6, 7, 8,
9, 1, 2, 3, 4, 5, 6, 7, 8, 9, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[11, 9, 7, 5, 3, 1, 12, 10, 8, 9, 5, 1, 10, 6, 2, 11, 7, 3, 7,
1, 8, 2, 9, 3, 10, 4, 11, 5, 10, 2, 7, 12, 4, 9, 1, 6, 3, 6,
9, 12, 2, 5, 8, 11, 1, 1, 2, 3, 4, 5, 6, 7, 8, 9, 12, 11, 10,
9, 8, 7, 6, 5, 4, 10, 7, 4, 1, 11, 8, 5, 2, 12, 8, 3, 11, 6,
1, 9, 4, 12, 7]
```

Lampiran 3: DNA_cryptography.ipynb

```
# UPLOAD DATASET
from google.colab import files
uploaded = files.upload()

# IMPORT LIBRARY
import random

import time

import psutil

from PIL import Image

from numpy import array
import numpy as np

import math

from skimage.metrics import structural_similarity as ssim
import matplotlib.pyplot as plt

#show image before preprocessing
rawImage = Image.open(r"01.tif")

#convert image to matrix
im2arr1 = array(rawImage)

#preprocessing
imPreP = np.uint8(rawImage)
rawIm = Image.fromarray(imPreP)

# GENERATE SYMMETRIC KEY K
#input g
cpu_start = time.process_time()

g = 0
prima = False

g = int(input("Enter g value: "))

while (prima == False):
  if g > 1:
    for i in range(2,g):
```

```
        if (g % i) == 0:
          print (g, "is not a prime number. Please input again."
)
          g = int(input("Enter g value: "))
          break
      else:
        print (g, "is a prime number")
        prima = True
        break
    else:
      print (g, "is not a prime number. Please input again.")
      g = int(input("Enter g value: "))
      break

#input n
n = 0
prima = False

n = int(input("Enter n value: "))

while (prima == False):
  if n > 1:
    for i in range(2,n):
      if (n % i) == 0:
        print (n, "is not a prime number. Please input again."
)
        n = int(input("Enter n value: "))
        break
    else:
      print (n, "is a prime number")
      prima = True
      break
  else:
    print (n, "is not a prime number. Please input again.")
    n = int(input("Enter n value: "))
    break

#input x and y
x = int(input("Enter x value: "))
y = int(input("Enter y value: "))

R = (g*x) % n
S = (g*y) % n

K1 = (S*x) % n
```

```
K2 = (R*y) % n
K = K1

print("K is", K)
cpu_end = time.process_time()
cpu_time = cpu_end-cpu_start
print("CPU time: %.3f s" % (cpu_time))
print('CPU usage is: ', psutil.cpu_percent(cpu_end), "%")
print('RAM memory % used:', psutil.virtual_memory()[2])

# GENERATE DNA OTP SEQUENCES
cpu_start = time.process_time()

OTP_length = imsize[0] * imsize[1] * K * 8

def get_OTP(OTP_length, K):
  length = OTP_length
  letters = "ACGT"
  rand_otp = ''.join((random.choice(letters) for i in range (l
ength)))

  #split
  split1 = []
  for i in range(1,(length//K)+1):
    str = ""
    for j in range((i*K)-K, i*K):
      str += rand_otp[j]
    split1.append(str)

  #reverse
  rev = []
  for i in reversed(split1):
    rev.append(i)

  #unsplit
  unsplit = ''.join(rev)

  #find complementary
  complement1 = ""
  for i in unsplit:
    if i == 'A':
      complement1 += 'T'
    elif i == 'T':
      complement1 += 'A'
    elif i == 'C':
```

```
      complement1 += 'G'
    elif i == 'G':
      complement1 += 'C'

  #split
  split2 = []
  for i in range(1,(length//K)+1):
    str2 = ""
    for j in range((i*K)-K, i*K):
      str2 += complement1[j]
    split2.append(str2)

  #add base
  added = []
  n = 0
  for i in split2:
    if n < 1:
      added.append(i+'A')
      n += 1
    elif n < 2:
      added.append(i+'C')
      n += 1
    elif n < 3:
      added.append(i+'G')
      n += 1
    elif n < 4:
      added.append(i+'T')
      n += 1
    n = n % 4
  return added

OTP = get_OTP(OTP_length, K)
OTP_str = ''.join(OTP)

OTPbin = ""
for i in OTP_str:
  if i == 'A':
    OTPbin+='00'
  elif i == 'T':
    OTPbin+='11'
  elif i == 'C':
    OTPbin+='10'
  elif i == 'G':
    OTPbin+='01'
```

```
cpu_end = time.process_time()
cpu_time = cpu_end-cpu_start
print("CPU time: %.3f s" % (cpu_time))
print('CPU usage is: ', psutil.cpu_percent(cpu_end), "%")
print('RAM memory % used:', psutil.virtual_memory()[2])

# ENCRYPTION
cpu_start = time.process_time()

#convert decimal to binary
im2bin1 = np.unpackbits(imPreP)

#find m,n for matrix size
larr = len(im2bin1)
factor = []
for i in range (1, larr + 1):
    if(larr % i==0):
        factor.append(i)
lfactor = len(factor)
n = 0
m = 0
if (lfactor%2==0):
    n = factor[(lfactor//2)-1]
    m = factor[(lfactor//2)]
else :
    n = factor[(lfactor//2)]
    m = n

#arrange the binary into a matrix named Z
Z = np.zeros((m,n))
count = 0
for i in range(m):
  for j in range(n):
    Z[i,j] = im2bin1[count]
    count+=1
Z = Z.transpose()

#flip Z as a new matrix named M
M = np.flip(Z,axis=1)
M = M.astype(np.int64)

#generate random DNA sequence
letters = "ACGT"
rand_dna = ''.join((random.choice(letters) for i in range (lar
r//2)))
```

```
#convert DNA to binary
dna2bin = ""
for i in rand_dna:
  if i == 'A':
    dna2bin += '00'
  elif i == 'T':
    dna2bin += '11'
  elif i == 'C':
    dna2bin += '10'
  elif i == 'G':
    dna2bin += '01'

#arrange the DNA binary into a matrix named N
N = np.zeros((m,n))
count = 0
for i in range(m):
  for j in range(n):
    N[i,j] = dna2bin[count]
    count+=1
N = N.transpose()
N = N.astype(np.int64)

#xor operation
MN = (M ^ N).transpose()

#convert 2d array to string
def tostr (mn):
  x,y = mn.shape
  st = ""
  for i in range(x):
    for j in range(y):
      st += str(mn[i,j])
  return st

mn = tostr(MN)

#scanning for 1 occurence
scanned = []
for i in range(len(mn)):
  if (mn[i] == '1'):
    scanned.append(OTP[i])

#delete additional base
deleted = []
```

```
for i in range (len(scanned)):
  lst = scanned[i]
  dlt = ""
  for j in range (len(lst) - 1):
    dlt += lst[j]
  deleted.append(dlt)
#find complementary
comp = []
for i in range(len(deleted)):
  a = deleted[i]
  b = ""
  for j in a:
    if j == 'A':
      b+='T'
    elif j == 'T':
      b+='A'
    elif j == 'C':
      b+='G'
    elif j == 'G':
      b+='C'
  comp.append(b)
compstr = ''.join(comp)

#convert DNA to binary
comp2bin = []
for i in range(len(comp)):
  a = comp[i]
  b = ""
  for j in a:
    if j == 'A':
      b+='00'
    elif j == 'T':
      b+='11'
    elif j == 'C':
      b+='10'
    elif j == 'G':
      b+='01'
  comp2bin.append(b)
cipher_temp = ''.join(comp2bin)

# BINARY MODIFICATION
#find n x n image size
n = len(cipher_temp)
sisi = 0
while ((sisi*sisi)*8 < n):
```

```
  sisi += 1

#convert OTP length to binary
def decimalToBinary(n):
    return bin(n).replace("0b", "")

lenOTP = decimalToBinary(len(OTPbin))
lencipher = decimalToBinary(len(cipher_temp))
print("Jumlah bit panjang OTP:", len(lenOTP))
print("Biner panjang ciphertext:", lencipher)
print("Jumlah bit panjang ciphertext:", len(lencipher))

#add 0 in front
plenOTP = 0
while (plenOTP * 8 < len(lenOTP)):
  plenOTP += 1

new_cipher = ""
for i in range ((plenOTP*8) - len(lencipher)):
  new_cipher += "0"

new_cipher += lencipher + cipher_temp

additional = 0
for i in range ((sisi*sisi*8)-len(new_cipher)):
  new_cipher += "0"
  additional += 1

#divide binary into 8 each
st2dec = []
a = ""
index = 0
for i in range(len(new_cipher)):
  a += new_cipher[index]
  index += 1
  if (len(a) == 8):
    st2dec.append(a)
    a = ""

#delete unused zero
st2dec_len = len(st2dec)
dec = []
for i in range(st2dec_len):
  dec.append(int(st2dec[i]))
```

```
#convert binary to decimal
def binaryToDecimal(binary):
    binary1 = binary
    decimal, i, n = 0, 0, 0
    while(binary != 0):
        dec = binary % 10
        decimal = decimal + dec * pow(2, i)
        binary = binary//10
        i += 1
    return decimal

bin2dec = []
for i in dec:
  bin2dec.append(binaryToDecimal(i))

final = np.zeros((sisi, sisi))
aa = 0
for i in range(sisi):
  for j in range(sisi):
    final[i,j] = bin2dec[aa]
    aa+=1

#extract ciphertext into cipher image
encrypt = np.array(final, dtype=np.uint8)
cipher_img = Image.fromarray(encrypt)

cpu_end = time.process_time()
cpu_time = cpu_end-cpu_start
print("CPU time: %.3f s" % (cpu_time))
print('CPU usage is: ', psutil.cpu_percent(cpu_time), "%")
print('RAM memory % used:', psutil.virtual_memory()[2])

# DECRYPTION
#convert image to array
cpu_start = time.process_time()

im2arr2 = array(cipher_img)

#convert decimal to binary
im2bin2 = np.unpackbits(im2arr2)
im2byte = im2bin2.tobytes()

str3 = ""
bytelist = bytes(im2byte)
```

```
for i in bytelist:
  str3 += str(i)

#get cipher image information
OTPlen = decimalToBinary(len(OTPbin))
print("INFORMASI CIPHERTEXT")
print("Panjang ciphertext:", len(str3))

q = 0
while (q*8 < len(OTPlen)):
  q += 1

selisih = 8*q - len(OTPlen)
print("Total bit inisiasi:", 8*q)

init_bit = str3[:(8*q)]
print("Bit inisiasi:", init_bit)

init_dec = int(init_bit,2)
print("Panjang ciphertext asli:", init_dec)

added = len(str3) - init_dec - len(init_bit)
print("Panjang 0 sisipan:", added)

#convert binary to DNA
str4 = str3[len(init_bit):(init_dec)+len(init_bit)]
ciphertxt = ""
for i in range(0,len(str4)-1,2):
  inext = i+1
  a = str4[i]+str4[inext]
  if a == "00":
    ciphertxt+='A'
  elif a == "11":
    ciphertxt+='T'
  elif a == "10":
    ciphertxt+='C'
  elif a == "01":
    ciphertxt+='G'

#find complementary
comp1 = ""
for i in ciphertxt:
  if i == 'A':
    comp1 += 'T'
  elif i == 'T':
```

```
    comp1 += 'A'
  elif i == 'C':
    comp1 += 'G'
  elif i == 'G':
    comp1 += 'C'

#split DNA sequence with K base each index
split2 = []
for i in range(1,(len(comp1)//K)+1):
  str5 = ""
  for j in range((i*K)-K, i*K):
    str5 += comp1[j]
  split2.append(str5)

#scanning DNA OTP sequence
cipher2bin = ''
for h in range(len(OTP)):
  cipher2bin += '0'
cipher2bin = list(cipher2bin)

cipher_index = []

now = 0
for i in split2:
  for j in range(now,len(OTP)):
    a = OTP[j]
    a = a[:-1]
    if i == a:
      cipher_index.append(j)
      cipher2bin[j] = '1'
      now = j
      break

cipherstr = ''.join(cipher2bin)

#add additional base
for i in range(len(cipher_index)):
  a = cipher_index[i] % 4
  if a == 0:
    split2[i] += 'A'
  elif a == 1:
    split2[i] += 'C'
  elif a == 2:
    split2[i] += 'G'
  elif a == 3:
```

```
    split2[i] += 'T'

#find e,f for matrix size
lenarr = len(cipher2bin)
fact = []
for i in range (1, lenarr + 1):
    if(lenarr % i==0):
        fact.append(i)
lfact = len(fact)
f = 0
e = 0
if (lfact%2==0):
    f = fact[(lfact//2)-1]
    e = fact[(lfact//2)]
else :
    f = fact[(lfact//2)]
    e = f

#arrange ciphertext binary into a matrix named MN1
MN1 = np.zeros((e,f))
count = 0
for i in range(e):
  for j in range(f):
    MN1[i,j] = cipher2bin[count]
    count+=1
MN1 = MN1.transpose()
MN1 = MN1.astype(np.int64)

#xor operation
M1 = (MN1 ^ N)

#flip M1 into a new matrix named Y
Y = np.flip(M1,axis=1)
Y = Y.transpose()

#convert 2d array to string
efn = tostr(Y)

#divide binary into 8 each
str2dec = []
b = ""
ind = 0
for i in range(len(efn)):
  b += efn[ind]
  ind += 1
```

```
  if (len(b) == 8):
    str2dec.append(b)
    b = ""
#convert binary to decimal
str2dec_len = len(str2dec)
decimal = []
for i in range(str2dec_len):
  decimal.append(int(str2dec[i]))

todec = []
for i in decimal:
  todec.append(binaryToDecimal(i))

#find s for image size
s = math.sqrt(len(decimal))
s = int(s)

im_final = np.zeros((s, s))
aarggh = 0
for i in range(s):
  for j in range(s):
    im_final[i,j] = todec[aarggh]
    aarggh+=1

#extract binary into decrypted image
decrypt = np.array(im_final, dtype=np.uint8)
plain_img = Image.fromarray(decrypt)

cpu_end = time.process_time()
cpu_time = cpu_end-cpu_start
print("CPU time: %.3f s" % (cpu_time))
print('CPU usage is: ', psutil.cpu_percent(cpu_time), "%")
print('RAM memory % used:', psutil.virtual_memory()[2])

#IMAGE ERROR TEST
cpu_start = time.process_time()

def mse(imageA, imageB):
  err = np.sum((np.asarray(imageA) - np.asarray(imageB)) ** 2)
  err /= float(imageA.shape[0] * imageA.shape[1])
  return err

def compare_images(imageA, imageB, title):
  m = mse(imageA, imageB)
  r = math.sqrt(m)
```

```
  s = ssim(np.asarray(imageA), np.asarray(imageB))

  # setup the figure
  fig = plt.figure(title)
  plt.suptitle("MSE: %.5f, RMSE: %.5f, SSIM: %.5f" % (m, r, s)
)

  # show first image
  ax = fig.add_subplot(1, 2, 1)
  plt.imshow(imageA, cmap = plt.cm.gray)
  plt.axis("off")

  # show the second image
  ax = fig.add_subplot(1, 2, 2)
  plt.imshow(imageB, cmap = plt.cm.gray)
  plt.axis("off")

  # show the images
  plt.show()

#testing
compare_images(imPreP, plain_img, "IMAGE ERROR TEST")

cpu_end = time.process_time()
cpu_time = cpu_end-cpu_start
print("CPU time: %.3f s" % (cpu_time))
print('CPU usage is: ', psutil.cpu_percent(cpu_time), "%")
print('RAM memory % used:', psutil.virtual_memory()[2])
```