Advanced Algorithms - UE19CS311
Assignment : Fast Polynomial Multiplication with DFT/FFT implementation, RSA Encryption, Image compression

Date: 3/12/2021

Sumukh Raju Bhat	PES1UG19CS519
Saileshwar Kartik	PES1UG19CS421

I) Polynomial Multiplication:

1. 1-D DFT:
    - We first obtain the vander-monde matrix M, where M is of the form as show below
    - Next we do a matrix multiplication with the 1D vector A to obtain the PV form of the polynomial A

```
## 1-d DFT
def DFT(x):
    x = np.asarray(x, dtype=float)
    N = len(x)
    M = np.ones((len(x), len(x)), dtype=complex)
    w = np.exp((-2*np.pi*1J)/N)
    for i in range(N):
        for j in range(N):
            M[i][j] = np.power(w, i*j)
    return np.dot(M,x)
```

- Now let's verify the algorithm on a 1D vector filled with 512 random values

```
# Verify
x = np.random.rand(512)
np.allclose(np.fft.fft(x), DFT(x))
```

```
True
```

- np.fft.fft in numpy's fft function which can be used to cross verify our dft and fft functions
- Now let’s observe the time complexity

```
%timeit DFT(x)
%timeit np.fft.fft(x)
```

```
1 loop, best of 5: 948 ms per loop
The slowest run took 115.26 times longer than the fastest. This could mean that an intermediate result is being cached.
100000 loops, best of 5: 9.54 µs per loop
```

- Here we observe that DFT took 950 ms to execute one loop while FFT took 9.54 µs which is almost 100,000 times slower.

2. 1-D FFT

- DFT runs in $O(n^2)$, while FFT runs in O(nlog(n))
- FFT(recursive) is implemented as shown below

```
def FFT(x):
    #x = np.asarray(x, dtype=float)
    n = len(x)
    if n==1:
        return x
    wn = np.exp(2*np.pi*1J/n)
    w = 1
    even = FFT(x[::2])
    odd = FFT(x[1::2])
    y = np.zeros_like(x, dtype=complex)
    for k in range(n//2):
        y[k] = even[k] + w*odd[k]
        y[k+(n//2)] = even[k] - w*odd[k]
        w = w*wn
    return y
```

- Now let's verify the algorithm on a 1D vector filled with 512 random values

```
# Verify
x = np.random.rand(512)
np.allclose(np.fft.fft(x), FFT(x))
```

```
True
```

- Now let's compare the time complexity of DFT and FFT

```
x = np.random.rand(1024)
%timeit DFT(x)
%timeit FFT(x)
%timeit np.fft.fft(x)
```

```
1 loop, best of 5: 3.59 s per loop
10 loops, best of 5: 18.5 ms per loop
The slowest run took 82.52 times longer than the fastest. This could mean that an intermediate result is being cached.
100000 loops, best of 5: 16.2 µs per loop
```

- DFT: 3.69s, FFT: 18.5 ms, numpy's FFT: 16.2 µs
- Let's compare values generated by DFT and FFT on the same coefficient representation vectors

```
# Make sure FFT and DFT have same value
# testcases
failure = 0
j = 0
max_iters = 10
n = [4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048]
for i in tqdm (range (max_iters), desc="Loading..."):
    x = np.random.rand(n[j%len(n)])
    x1 = double_db(x)

    m1 = DFT(x1)
    m3 = FFT(x1)

    if(np.allclose(m1, m3) == False):
        print(False)
        failure = failure+1

print("Failed",failure,"times!")
```

```
Loading...: 100%|██████████| 10/10 [01:17<00:00,  7.76s/it]Failed 0 times!
```

- Thus, they match as our code failed 0 times!

3. 1-D IFFT:

- Obtain PV form of the two polynomials by applying FFT on them
- Multiply A(x) and B(x) to obtain C(x) in PV form
- Then apply IFFT on it to get C(x) in CR form. The overall polynomial multiplication will be in O(logn) time.

- Code for recursive IFFT:

```
[40] import math
     def rifft(y):
         n = y.size
         if n == 1:
             return y
         i = 1j
         w_n = np.exp(- 2 * i * np.pi / float(n))
         w = 1
         a_0 = rifft(y[::2])
         a_1 = rifft(y[1::2])
         a = np.zeros(n, dtype=np.complex_)
         for k in range(0, n // 2):
             a[k] = (a_0[k] + w * a_1[k])
             a[k + n // 2] = (a_0[k] - w * a_1[k])
             w = w * w_n
         return a

     def IFFT(x):
         return rifft(x)/len(x)
```

- We test our functions against the ones in the numpy module for various shapes: (We see that it failed 0 times!)

```
# Testing our functions with numpy's
failure = 0
n = [4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048]
for i in tqdm (range (1000), desc="Loading..."):
    x = double_db(np.random.rand(n[j%len(n)]))
    y = double_db(np.random.rand(n[j%len(n)]))

    m1 = FFT(x)
    m2 = FFT(y)
    c = m1*m2

    CRc = IFFT(c)
    v = np.fft.ifft(np.fft.fft(x)*np.fft.fft(y))
    if(np.allclose(CRc, v) == False):
        failure = failure+1

print("Failed",failure,"times!")
```

```
Loading...: 100%|██████████| 1000/1000 [00:00<00:00, 2456.91it/s]Failed 0 times!
```

4. Multiplication using convolution for-loop:
    - We test the multiplication using fft-ifft against the result obtained from convolution-for loop which runs in $O(n^2)$ time.
    - Code:

```
[33] # Multiplication using convolution for loop
     def convolution(x, y):
       ck = np.zeros(y.shape[0] + x.shape[0])
       x = double_db(x)
       y = double_db(y)

       for i in range(ck.shape[0]):
           for j in range(0, i+1):
               ck[i] += x[j]*y[i-j]

       return ck
```

- As we see below, our code failed 0 times!

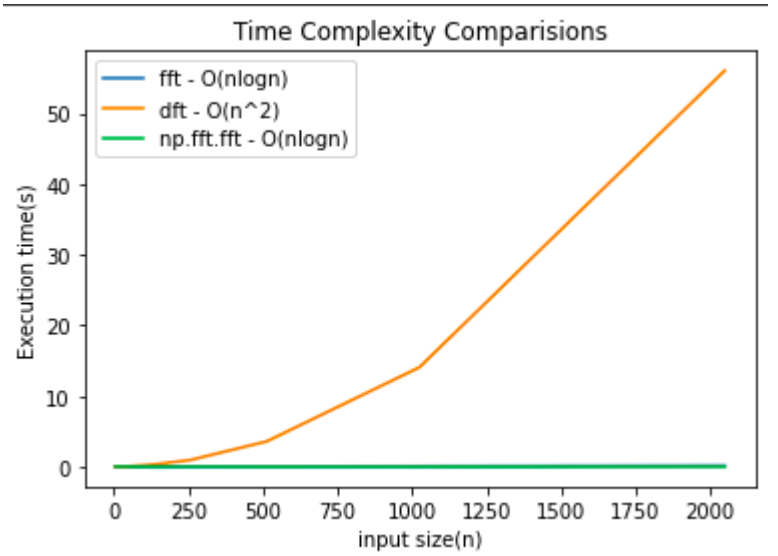
```
[56] # Verifying FFT-IFFT multiplication with regards to convolution loop multiplication
     failure = 0
     j = 0
     max_iters = 10000
     n = [4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048]
     for i in tqdm (range (max_iters), desc="Loading..."):
         x = np.random.rand(n[j%len(n)])
         y = np.random.rand(n[j%len(n)])
         x1 = double_db(x)
         y1 = double_db(y)

         m1 = FFT(x1)
         m2 = FFT(y1)

         Cx = IFFT(m1*m2)
         Cx1 = convolution(x, y)

         if(np.allclose(Cx1, Cx) == False):
             failure = failure+1

     print("Failed",failure,"times!")

     Loading...: 100%|██████████| 10000/10000 [00:04<00:00, 2171.82it/s]Failed 0 times!
```

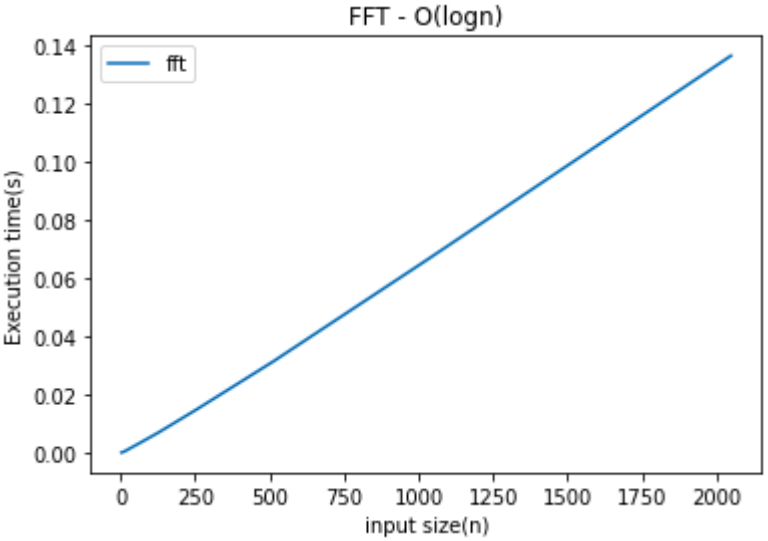
5. Running time analysis:
   - We notice the following trend on varying input size after averaging out the values got in multiple iterations for fft(O(nlogn)), dft(O(n^2)) and numpy's fft.fft: (in seconds)

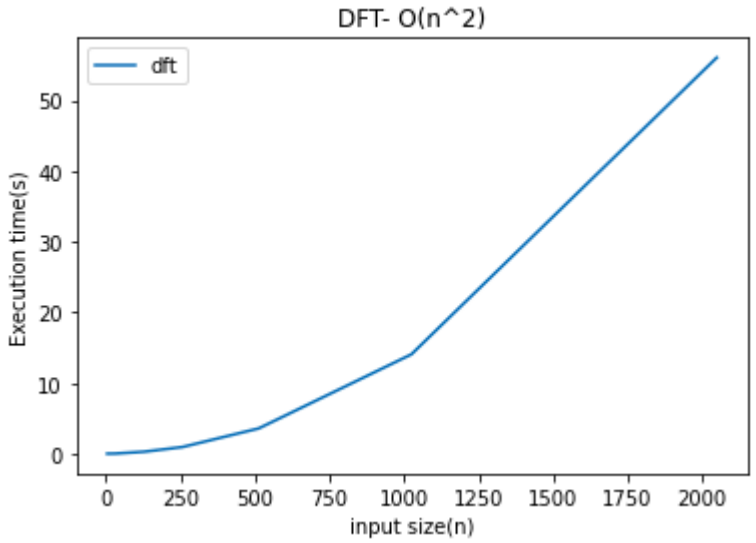
n	**FFT**	**DFT**	**np.fft.fft**
4	0.0001427	0.0002718	1.64e-5
8	0.0002728	0.0008826	2.49e-5
16	0.0006066	0.0034498	4.18e-5
32	0.0015368	0.0139558	2.62e-5

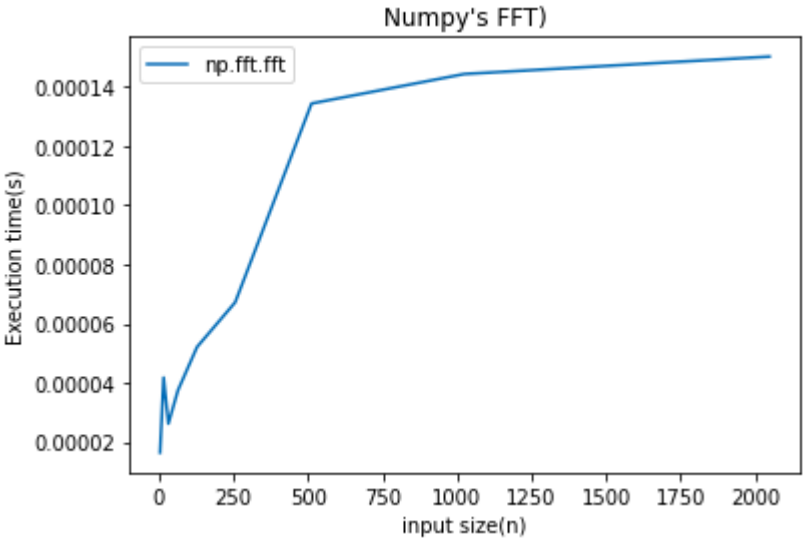
64	0.003327	0.103826	3.74e-5
128	0.006996	0.274915	5.21e-5
256	0.014927	0.936672	6.723e-5
512	0.031341	3.57252	0.0001344
1024	0.065961	14.0604	0.0001443
2048	0.136353	56.06768	0.0001502

- Plots:









II) RSA Encryption:

- We have a class called transmission common to both receiver and sender which has public and secret keys as class members and encrypt(), decrypt() as member functions.
- On instantiating an object of the class by passing the number of bits used as a parameter to the constructor, both secret and public keys are calculated and stored as class members.
- To encrypt, the receiver's public key has to be passed as a parameter to the sender’s encrypt function.
- To decrypt, the receiver uses his secret key to decrypt the message sent.
- Steps used:

- Generate random numbers p and q of specified bits and perform the Fermat's little theorem and Rabin-Miller test for primality. Brute-force way checking primality is also implemented, but commented as it was observed that it takes too much time to complete execution, especially for > 50 bits. Keep generating random numbers until they pass the primality tests.
- n = p*q and we find a number e which is relative prime to phi(n) = (p-1)(q-1).
  [To find relative prime, we initially used a neat mathematical trick that, for all integers x,
  gcd(nx+1, n) = gcd(n, 1) = gcd(1, 0) = 1.
  Therefore, nx+1 is relatively prime to n. But this also means that d = 1 always in the further steps. Hence we dropped this method and we used a brute force method instead]
- We find d which is the modular inverse of 5 with respect to (p-1)(q-1)
- Save the public key, P = (e, n) and
  secret key, S = (d, n).
- Encryption: $M' = (M^{e})(\text{mod } n)$ and
  Decryption: $M = (M'^{d})(\text{mod } n)$

- Some of the utils/helper functions:

```
# utils for RSA algo
def MODULAR_EXPONENTIATION(a, b, n):
  c = 0
  d = 1
  binary = bin(b)[2:]
  for i in binary:
    c = 2*c
    d = (d*d) % n
    if(i == '1'):
      c+=1
      d = (d*a)%n
  return d

def EXTENDED_EUCLIDS_GCD(m, n):
  if(n == 0):
    return (m, 1, 0)
  else:
```

```
    d1, x1, y1 = EXTENDED_EUCLIDS_GCD(n, m%n)
    d, x, y = d1, y1, (x1 - (m//n)*y1)
    return (d, x, y)

def MODULAR_INVERSE(a, m):
  g, x, y = EXTENDED_EUCLIDS_GCD(a, m)
  if g != 1:
      return -1
  else:
      return x % m

def WITNESS(a, n):
  binary = bin(n-1)[2:]
  t = 0
  for i in binary[-1::-1]:
    if(i == '1'):
      break
    else:
      t+=1
  u = (n-1) >> t
  xcur = MODULAR_EXPONENTIATION(a, u, n)
  for i in range(t):
    xprev = xcur
    xcur = (xprev*xprev) % n
    if(xcur == 1 and xprev != 1 and xprev != n-1):
      return 1
  if(xcur != 1):
    return 1
  return 0

# Fermat's little theorem
def FERMATS_TEST(n):
  if(MODULAR_EXPONENTIATION(2, n - 1, n) != 1 % n):
    return 0
  return 1

def MILLER_RABIN_TEST(n, s):
  for i in range(s):
    a = random.randint(1, n-1)
    if(WITNESS(a, n)):
      return 0
  return 1
```

```
def BRUTEFORCE_TEST(n):
  if(n > 1):
      for i in range(2, int(sqrt(n)) + 1):
          if (n % i == 0):
              return 0
      return 1
  else:
      return 0

def EUCLIDS_GCD(m, n):
  if(n == 0):
    return m
  return EUCLIDS_GCD(n, m%n)
```

- Code:

```
# RSA Encryption
class Transmission:
  def __init__(self, bits):
    s = random.randint(5,20)  # iters in miller rabin
    while(1):
      p = random.getrandbits(bits)
      if(p <= 1):
        continue
      if(p%2 == 0):
        p+=1
      if(not FERMATS_TEST(p)):
        # print("Failed fermat's little theorem test for p = ",
p)
        continue
      if(not MILLER_RABIN_TEST(p, s)):
        # print("Failed miller rabin test for p = ", p)
        continue
      # if(not BRUTEFORCE_TEST(p)):
        # print("Failed Bruteforce for p = ", p)
      break

    while(1):
      q = random.getrandbits(bits)
      if(q <= 1):
        continue
      if(p == q):
        continue
      if(q%2 == 0):
```

```
        q+=1
      if(not FERMATS_TEST(q)):
        # print("Failed fermat's little theorem test for q = ",q)
        continue
      if(not MILLER_RABIN_TEST(q, s)):
        # print("Failed miller rabin test for q = ", q)
        continue
      # if(not BRUTEFORCE_TEST(q)):
        # print("Failed Bruteforce for q = ", q)
        # continue
      break
    n = p*q
    x = random.randint(1, 10)
    e = 3
    while(EUCLIDS_GCD(e, (p-1)*(q-1)) != 1):
      e+=2

    d = MODULAR_INVERSE(e, (p-1)*(q-1))
    self.P = (e, n)
    self.S = (d, n)

  def encrypt(self, Perceiver, M):
    e = Perceiver[0]
    n = Perceiver[1]
    return MODULAR_EXPONENTIATION(M, e, n)

  def decrypt(self, C):
    d = self.S[0]
    n = self.S[1]
    return MODULAR_EXPONENTIATION(C, d, n)

  def encrypt_c(self, Perceiver, M):
    e = Perceiver[0]
    n = Perceiver[1]
    cipher = [MODULAR_EXPONENTIATION(ord(chr), e, n) for chr in
M]
    return cipher

  def decrypt_c(self, C):
    d = self.S[0]
    n = self.S[1]
    aux = [chr(MODULAR_EXPONENTIATION(x, d, n)) for x in C]
    return ''.join(aux)
```

- We encrypt the C(x) obtained using the previous part of the assignment and decrypt it. We consider it as an array of ascii values for convenience. We notice that we get the same C(x) back after a round of encrypting and decrypting it back!

```
# encrypting C(X) using 512 bit keys

sender = Transmission(512)
receiver = Transmission(512)

x = np.random.rand(512)
y = np.random.rand(512)
x1 = double_db(x)
y1 = double_db(y)

m1 = FFT(x1)
m2 = FFT(y1)

Cx = m1*m2
CRc = IFFT(Cx)
V = np.array([str(x) for x in Cx])
C = np.array([sender.encrypt_c(receiver.P, x) for x in V])
M = np.array([complex(receiver.decrypt_c(x)) for x in C])
np.allclose(M, Cx)
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:1
True
```

- We also tested our code with some randomly generated integers and of different bit sizes for multiple times and as seen below, our code failed 0 times!

```
# Testcases for RSA Encryption
bits = [128, 256, 512]
failure = 0

for bit in bits:
  sender = Transmission(bit)
  reciever = Transmission(bit)
  for i in range(1000):
    M = random.randint(-100000000, 100000000)
    C = sender.encrypt(reciever.P, M)
    M1 = reciever.decrypt(C)
    if(M < 0):
      M1 = M1 - reciever.P[1]
    if(M1 != M):
      failure+=1

print("Failed",failure,"times!")
```

```
Failed 0 times!
```

-

- Execution time comparison for encrypting and decrypting using various sized keys, each run for 1000 iterations

Size	Time(in seconds)
128	0.266
256	1.294
512	6.604

III) Image Compression:

1. FFT2D:
    - Implementation of 2D FFT
    - It is done by applying 1D FFT on each row of an input matrix and then on the columns of the resultant matrix.
    - In the below code, axis = 1 indicates rows of a matrix and axis = 0 indicates columns of a matrix.

```
[126] # 2D FFT
      def FFT2D(matrix):
        matrix = np.apply_along_axis(FFT, 1, matrix)
        matrix = np.apply_along_axis(FFT, 0, matrix)

        return matrix
```

2. IFFT2D:
    - Implementation of 2D IFFT
    - It is done by applying 1D IFFT on each row of an input matrix and then on the columns of the resultant matrix.
    - In the below code, axis = 1 indicates rows of a matrix and axis = 0 indicates columns of a matrix.

```
  # 2D IFFT
  def IFFT2D(matrix):
    matrix = np.apply_along_axis(IFFT, 1, matrix)
    matrix = np.apply_along_axis(IFFT, 0, matrix)

    return matrix
```

3. Testcase:

- We input a random 4x4 matrix with elements <= 256 and >=0 mimicking a grayscale image to our FFT2D and IFFT2D functions and verify their correctness

```
[ ]  # testing 2D-fft and 2D-ifft using a random grayscale image
     grayscale_img = np.random.randint(0, 256, size = (4,4))
     ffted = FFT2D(grayscale_img)
     iffted = IFFT2D(ffted)
     np.allclose(iffted, grayscale_img)

     True
```

4. Compression:

- We input a grayscale image to our function along with the threshold of keeping the information or clarity of the inputted image
- 'Threshold' parameter decides how much the the image will be compressed i.e. more the magnitude of threshold parameter, more will the image be compressed
- However, it should be noted that this is a lossy compression. Meaning, we cannot get the lost information and lost clarity back.

```
# Lossy compression -> apply fft2d, drop some values, apply ifft2d
def compression(image, threshold = 0.5):
  img_fft = FFT2D(image)
  img_fft_sorted = np.sort(np.abs(img_fft.reshape(1, img_fft.shape[0]*img_fft.shape[1])[0]))
  upper_bound = img_fft_sorted[int(threshold*(len(img_fft_sorted) - 1))]
  compressed = img_fft *( np.abs(img_fft) >= upper_bound )

  return np.real(IFFT2D(compressed)).astype('int')
```

5. Outputs/Test Cases:

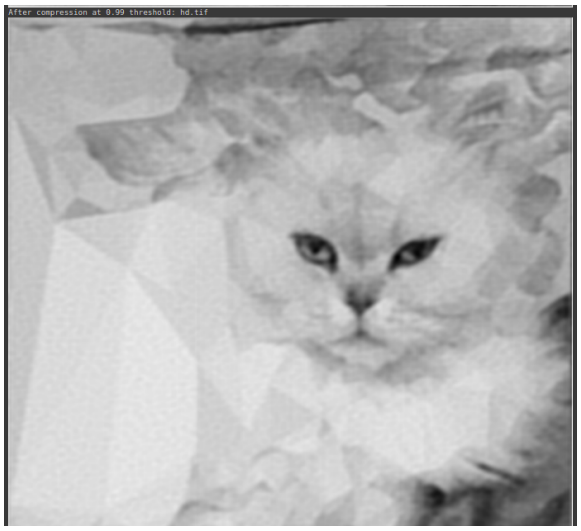
 - We have 4 images of varying dimensions in Tag Image File Format(tiff) already in grayscale image called hd.tif, cat.tif, tiger.tif, scene.tif in 1024x1024, 512x512, 512x512 1024x2048 respectively.
    - We compress each of these images with threshold values of 0.1, 0.5, 0.8, 0.9, 0.95, 0.99, 0.999
    - 0.1 should result in least compression and 0.999 should result in highest compression
    - We save the result of compression in the format <original_image_name>_compressed_<threshold>.tif
    - We observe the decrease in the file size as follows:

```
] !ls -l

 total 14548
 -rw-r--r-- 1 root root  230220 Dec  4 11:15 cat_compressed_0.1.tif
 -rw-r--r-- 1 root root  248298 Dec  4 11:15 cat_compressed_0.5.tif
 -rw-r--r-- 1 root root  248958 Dec  4 11:15 cat_compressed_0.8.tif
 -rw-r--r-- 1 root root  213644 Dec  4 11:16 cat_compressed_0.95.tif
 -rw-r--r-- 1 root root   68648 Dec  4 11:17 cat_compressed_0.999.tif
 -rw-r--r-- 1 root root  141106 Dec  4 11:16 cat_compressed_0.99.tif
 -rw-r--r-- 1 root root  236898 Dec  4 11:16 cat_compressed_0.9.tif
 -rw-r--r-- 1 root root  262266 Dec  4 11:04 cat.tif
 -rw-r--r-- 1 root root  507820 Dec  4 11:06 hd_compressed_0.1.tif
 -rw-r--r-- 1 root root  514908 Dec  4 11:08 hd_compressed_0.5.tif
 -rw-r--r-- 1 root root  494130 Dec  4 11:09 hd_compressed_0.8.tif
 -rw-r--r-- 1 root root  389268 Dec  4 11:12 hd_compressed_0.95.tif
 -rw-r--r-- 1 root root  233194 Dec  4 11:14 hd_compressed_0.999.tif
 -rw-r--r-- 1 root root  303290 Dec  4 11:13 hd_compressed_0.99.tif
 -rw-r--r-- 1 root root  444492 Dec  4 11:10 hd_compressed_0.9.tif
 -rw-r--r-- 1 root root 1048698 Dec  4 11:04 hd.tif
 drwxr-xr-x 1 root root    4096 Nov 18 14:36 sample_data
 -rw-r--r-- 1 root root  798846 Dec  4 11:22 scene_compressed_0.1.tif
 -rw-r--r-- 1 root root  805544 Dec  4 11:25 scene_compressed_0.5.tif
 -rw-r--r-- 1 root root  946592 Dec  4 11:27 scene_compressed_0.8.tif
 -rw-r--r-- 1 root root  964256 Dec  4 11:33 scene_compressed_0.95.tif
 -rw-r--r-- 1 root root  417442 Dec  4 11:38 scene_compressed_0.999.tif
 -rw-r--r-- 1 root root  699350 Dec  4 11:36 scene_compressed_0.99.tif
 -rw-r--r-- 1 root root  971414 Dec  4 11:30 scene_compressed_0.9.tif
 -rw-r--r-- 1 root root 2097274 Dec  4 11:05 scene.tif
 -rw-r--r-- 1 root root  226116 Dec  4 11:17 tiger_compressed_0.1.tif
 -rw-r--r-- 1 root root  228058 Dec  4 11:17 tiger_compressed_0.5.tif
 -rw-r--r-- 1 root root  219340 Dec  4 11:18 tiger_compressed_0.8.tif
 -rw-r--r-- 1 root root  181160 Dec  4 11:18 tiger_compressed_0.95.tif
 -rw-r--r-- 1 root root   81518 Dec  4 11:19 tiger_compressed_0.999.tif
 -rw-r--r-- 1 root root  134924 Dec  4 11:19 tiger_compressed_0.99.tif
 -rw-r--r-- 1 root root  204648 Dec  4 11:18 tiger_compressed_0.9.tif
 -rw-r--r-- 1 root root  262266 Dec  4 11:04 tiger.tif
```

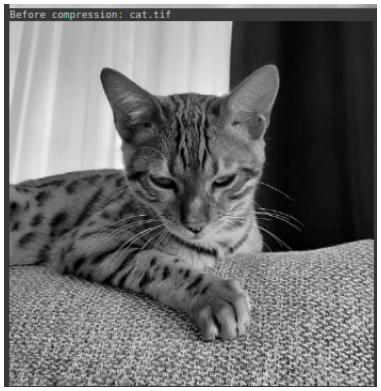
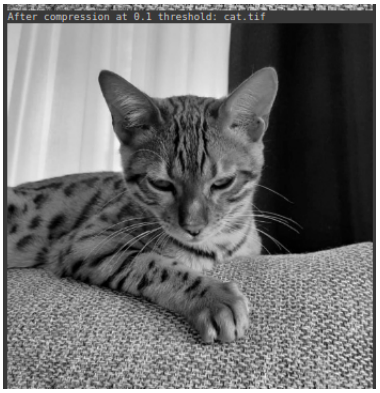
- Compression of the image hd.tif:

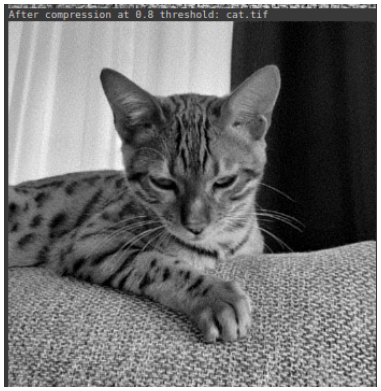
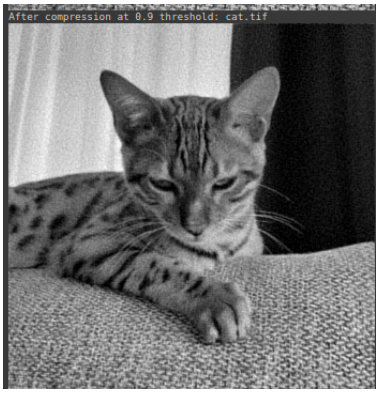
Original	Threshold = 0.1	Threshold = 0.5
		

Threshold = 0.8	Threshold = 0.9	Threshold = 0.95
		

Threshold = 0.99	Threshold = 0.999
	

- Compression of the image cat.tif:

Original	Threshold = 0.1	Threshold = 0.5
		

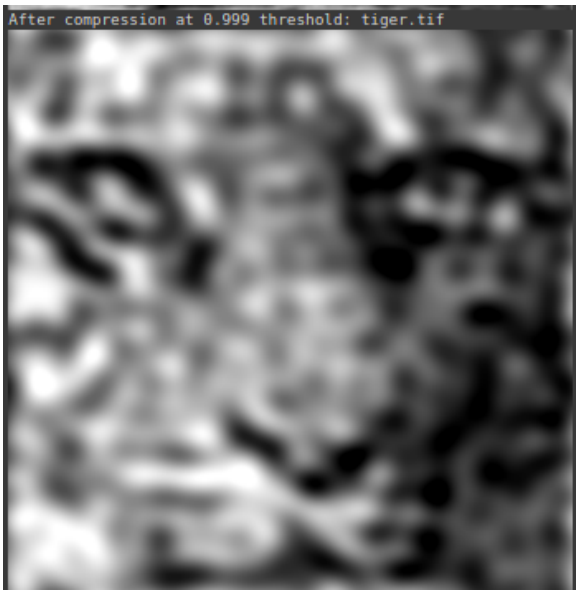
Threshold = 0.8	Threshold = 0.9	Threshold = 0.95
		

Threshold = 0.99	Threshold = 0.999
	

- Compression of the image tiger.tif:

Original	Threshold = 0.1	Threshold = 0.5
		

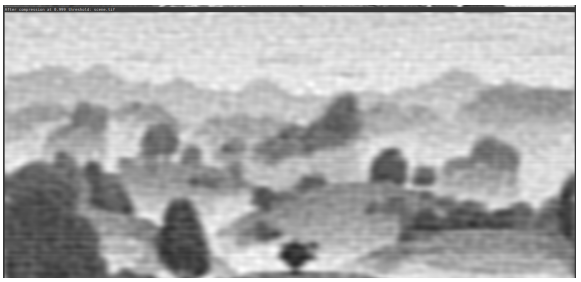
Threshold = 0.8	Threshold = 0.9	Threshold = 0.95
		

Threshold = 0.99	Threshold = 0.999
	

- Compression of the image scene.tif:

Original	Threshold = 0.1	Threshold = 0.5
		

Threshold = 0.8	Threshold = 0.9	Threshold = 0.95
		

Threshold = 0.99	Threshold = 0.999
	

- Summary of compressed file sizes (in Bytes)

Name	Original	0.1	0.5	0.8	0.9	0.95	0.99	0.999
hd.tif	1048698	507820	514908	494130	444492	389268	303290	233194
cat.tif	262266	230220	248298	248958	236898	213644	141106	141106
tiger.tif	262266	226116	228058	219340	204648	181160	134924	81518
scene.tif	2097274	798846	805544	946592	971414	964256	699350	417442

IV) Learning Outcomes:

The assignment was a great experience in terms of watching the asymptotic time complexity coming into picture of various algorithms, which although performs the same task, but with different execution time. The assignment showcases these execution time differences of polynomial multiplication in a smooth manner. The assignment also contributed positively to the practise of writing good code, given an algorithm or pseudo-code. Coding an image compression algorithm which is so simple in theory, yet so effective and changes can be seen through the bare eyes was a great experience. Witnessing the execution of RSA encryption, which again is so simple in theory and also so accurate in practise was fun. All-in-all, it was a great learning experience!

V) Source code of all implementation

VI) References:

1. Introduction to Algorithms, 3rd Edition, CLRS
2. Image Compression and the FFT - Steve Brunton on Youtube
3. Some of the articles on GFG on primality testing