ADVANCED ALGORITHMS ASSIGNMENT
UE19CS311

Fast Polynomial Multiplication with DFT/FFT implementation, RSA Encryption , Image compression

Team details:
Smruthi B.T : PES1UG19CS487
Anurita Bose : PES1UG19CS072

Objective:

Implementing 1-D & 2-D Fourier Transform & RSA Encryption on a MxN matrix to achieve Fast Polynomial Multiplication, Secure transmission N matrix to achieve Fast Polynomial Multiplication, Secure transmission and Lossy Image compression.

I. Problem — Given two polynomials- A(x) =a0+a1x+a2x^2+… anx^n and B(x) =b0+b1x+b2x^2+… bnx^n, find the polynomial C(x) = A(x)*B(x).

Solution - More generally, when C(x) = A(x)*B(x) and C(x) =c0+c1x+c2x^2+... c2nx^2n .

If we think of A and B as vectors, then C vector is called '**Convolution**' of A and B (represented as **{A⊗B}**). The polynomials A and B are appropriately appended with 0s to make each of their degrees equal to 2n (degree of C). Straightforward computation is **O(n^2)** time.
Another way of representing a polynomial is called **point-value representation**. A polynomial of degree n-1 can be uniquely represented by its values at at-least n different points. This is a result of the **Fundamental Theorem of Algebra** that states — **"A degree n-1 polynomial A(x) is uniquely specified by its evaluation at n distinct values of x"**.

A polynomial A(x) =a0+a1x+a2x^2+… anx^n can be represented as:

1. A set of n pairs {(x0, y0),(x1, y1),...,(xn, yn)} such that
2. for all i != j, xi != xj. ie, the points are unique.
3. for every k, yk = A(xk);

In point-value form, multiplication $C(x) = A(x)B(x)$ is given by $C(xk) = A(xk) .B(xk)$ for any point (xk).

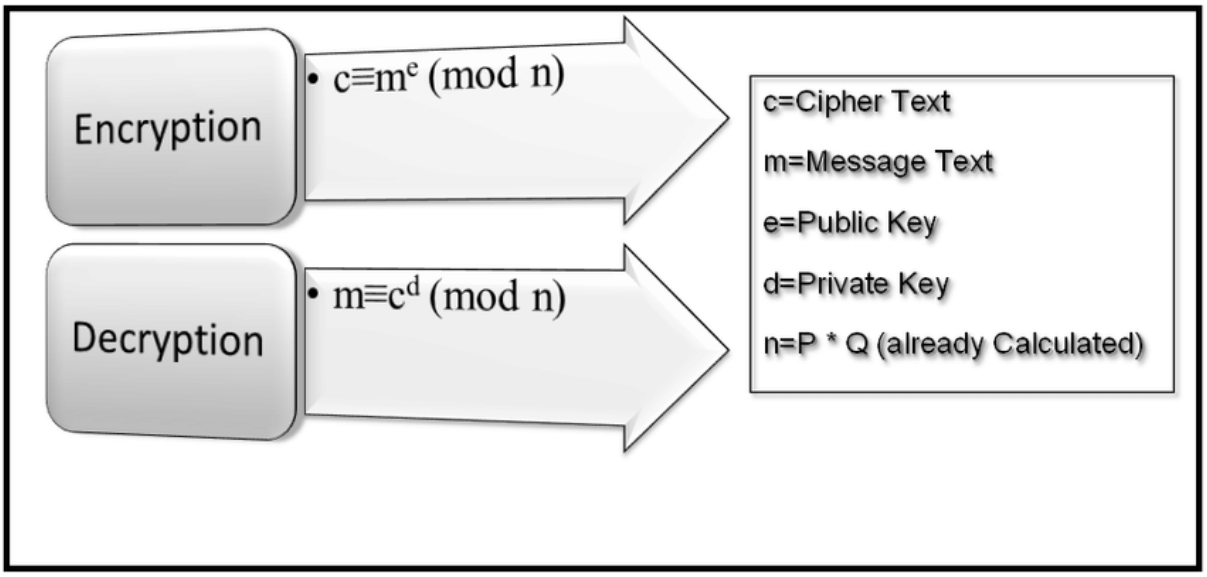
The time to multiply two polynomials of degree-bound *n* in point-value form is **Θ(n)**.The inverse of evaluation--determining the coefficient form of a polynomial from a point-value representation--is called ***interpolation.*** This is what the Convolution Theorem states with respect to Fourier Transform of a function.

II. Objective behind RSA algorithm —- RSA is a public key cryptography system. Public key cryptography, also known as asymmetric cryptography, uses two different but mathematically linked keys -- one public and one private.

The public key can be shared with everyone, whereas the private key must be kept secret.

In RSA cryptography, both the public and the private keys can encrypt a message. The opposite key from the one used to encrypt a message is used to decrypt it.

This attribute is one reason why RSA has become the most widely used asymmetric algorithm: It provides a method to assure the confidentiality, integrity, authenticity, and non-repudiation of electronic communications and data storage.



Implementation, Design and Results

i) Implement 1-D DFT ,on coefficient vectors of two polynomials A(x), B(x) by multiplication of Vander-Monde matrix . (O(n2) - Complexity)

```
def dft(x):
    x = np.asarray(x, dtype = np.cdouble)
    N = x.shape[0]
    n = np.arange(N)
    k = n.reshape((N, 1))
    M = np.exp(-2j * np.pi * k * n / N)
    return np.dot(M, x)

def inv_dft(C):
    xa = np.asarray(C, dtype = np.cdouble)
    N = xa.shape[0]
    n = np.arange(N)
    k = n.reshape((N, 1))
    M = np.exp(2j * np.pi * k * n / N)
    M_inverse = np.linalg.inv(M)
    VC = np.dot(M_inverse,C)
    return VC

#code to do pointwise multiplication
def dft_multiply(a,b):
    VA=dft(a)
    VB=dft(b)
    C = []
    for i in range(VA.shape[0]):
        C.append(VA[i]*VB[i])
    return C
```

In the dft function, we found the Vander-Monde matrix corresponding to the nth roots of unity where n is the dimension of the coefficient vector x that was passed as an argument to the function. We got the dot product of the matrix with x (Vx).

On passing A and B, we obtain VA and VB.

In the inverse dft function, we pass VC (which is obtained when we do pointwise multiplication of the values in VA and VB), to get the final result C, which is V^(-1)*VC.

```
#driver code
i = 4
poly_a = list(np.random.randint(low = -1000, high = 1000, size = i))
poly_b = list(np.random.randint(low = -1000, high = 1000, size = i))
dft_res_a=dft(poly_a)
dft_res_b=dft(poly_b)
C = dft_multiply(dft_res_a,dft_res_b)
VC=inv_dft(C)
```

Checking the return value of DFT with numpy.fft.fft function, using randomly generated polynomial coefficient vectors for A(x) and B(x) of varying degree-bound sizes namely, n = 4, 8, 16, 32, 64, … 1024 and 2048

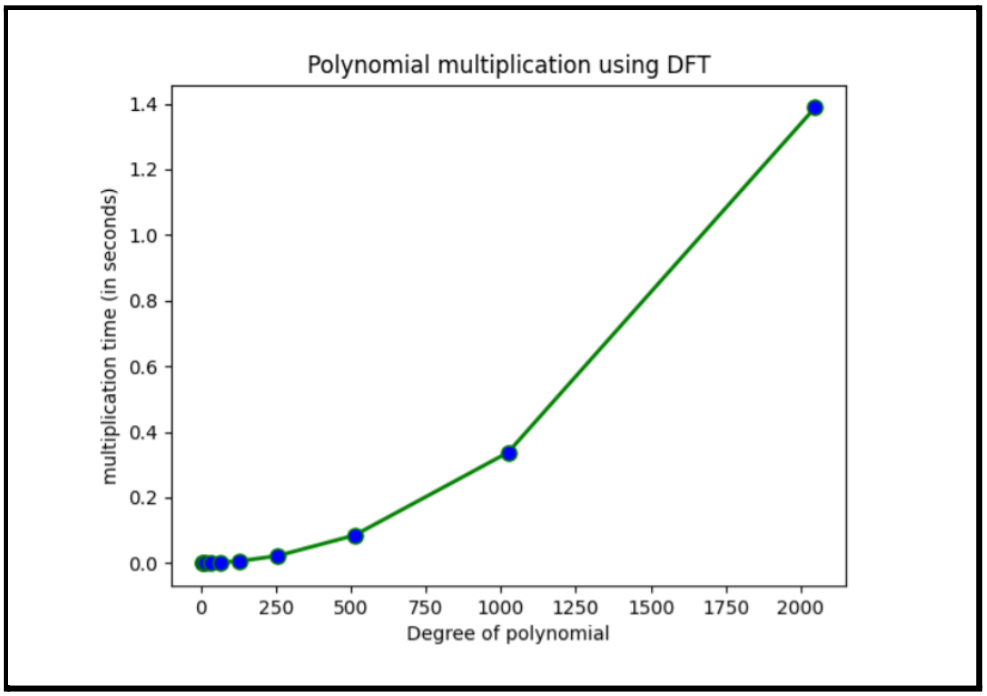
```
#arguments for dft, fft, inverse dft and inverse fft
parameters = []
parameters = []
i=4
while i<2048:
    param1 = list(np.random.randint(low = 0, high = 1000, size = i))
    parameters.append(param1)
    param2 = list(np.random.randint(low = 0, high = 1000, size = i))
    parameters.append(param2)
    i=i*2
```

```
def test_case():
    try:
        for test in range(len(parameters)):
            VA = dft(parameters[test])
            print(np.allclose(VA, np.fft.fft(parameters[test])))
            print("Test Case", (test+1), "for the function DFT passed")
    except:
        print("Test Case", (test+1), "for the function DFT failed")
```

Which results in,

```
True
Test Case 1 for the function DFT passed
True
Test Case 2 for the function DFT passed
True
Test Case 3 for the function DFT passed
True
Test Case 4 for the function DFT passed
True
Test Case 5 for the function DFT passed
True
Test Case 6 for the function DFT passed
True
Test Case 7 for the function DFT passed
True
Test Case 8 for the function DFT passed
True
Test Case 9 for the function DFT passed
True
Test Case 10 for the function DFT passed
True
Test Case 11 for the function DFT passed
True
Test Case 12 for the function DFT passed
True
Test Case 13 for the function DFT passed
True
Test Case 14 for the function DFT passed
True
Test Case 15 for the function DFT passed
True
Test Case 16 for the function DFT passed
True
Test Case 17 for the function DFT passed
True
Test Case 18 for the function DFT passed
```

Time taken for the multiplication of two polynomials of varying **degree bound sizes** vs **multiplication time in seconds**



ii) Implementing 1-D FFT on the same vectors, of A(x) and B(x)(O(n logn) – Complexity)

```
def fft(x):
    x = np.asarray(x, dtype= np.cdouble)
    N = x.shape[0]
    if N % 2 > 0:
        raise ValueError("must be a power of 2")
    elif N <= 2:
        return dft(x)
    else:
        X_even = fft(x[::2])
        X_odd = fft(x[1::2])
        terms = np.exp((-2j * np.pi * np.arange(N)) / N)
        return np.concatenate([X_even + terms[:int(N/2)] * X_odd,
                               X_even + terms[int(N/2):] * X_odd])
```

The plan is to use a divide-and-conquer strategy, called
Fast Fourier Transform (FFT), to compute DFTn, in θ(nlog n) time.

In the fft function, we are using the recursive method for evaluating the point value representation of a coefficient vector at the nth complex roots of unity. We separately use the even-indexed and odd-indexed coefficients of A, to define two new polynomials of half the degree-bound.

Therefore, evaluating the A(x) at the n complex nth roots of unity (i.e., DFTn), has been reduced to evaluating two polynomials at the n/2 complex, n/2th roots of unity (i.e., two instances of DFTn/2). Once the A(x) and B(x) are evaluated at n complex roots of unity, we get 2 vectors PA and PB.

Verifying that our implementation of FFT and numpy.fft.fft produces the same results, using randomly generated polynomial coefficient vectors for A(x) and B(x) of varying degree-bound sizes namely, n = 4, 8, 16, 32, 64, … 1024 and 2048.

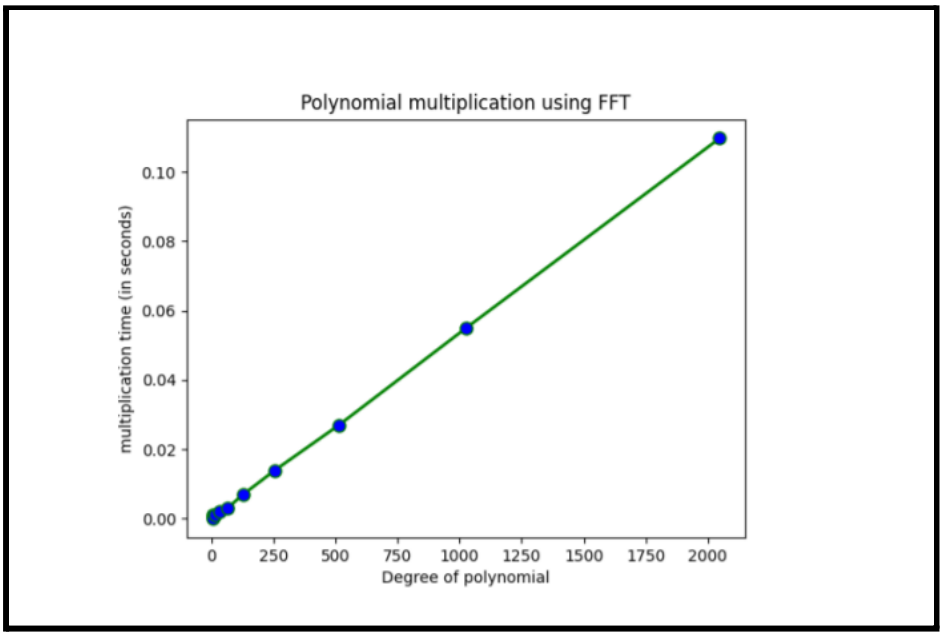
```
#arguments for dft, fft, inverse dft and inverse fft
parameters = []
parameters = []
i=4
while i<2048:
    param1 = list(np.random.randint(low = 0, high = 1000, size = i))
    parameters.append(param1)
    param2 = list(np.random.randint(low = 0, high = 1000, size = i))
    parameters.append(param2)
    i=i*2
```

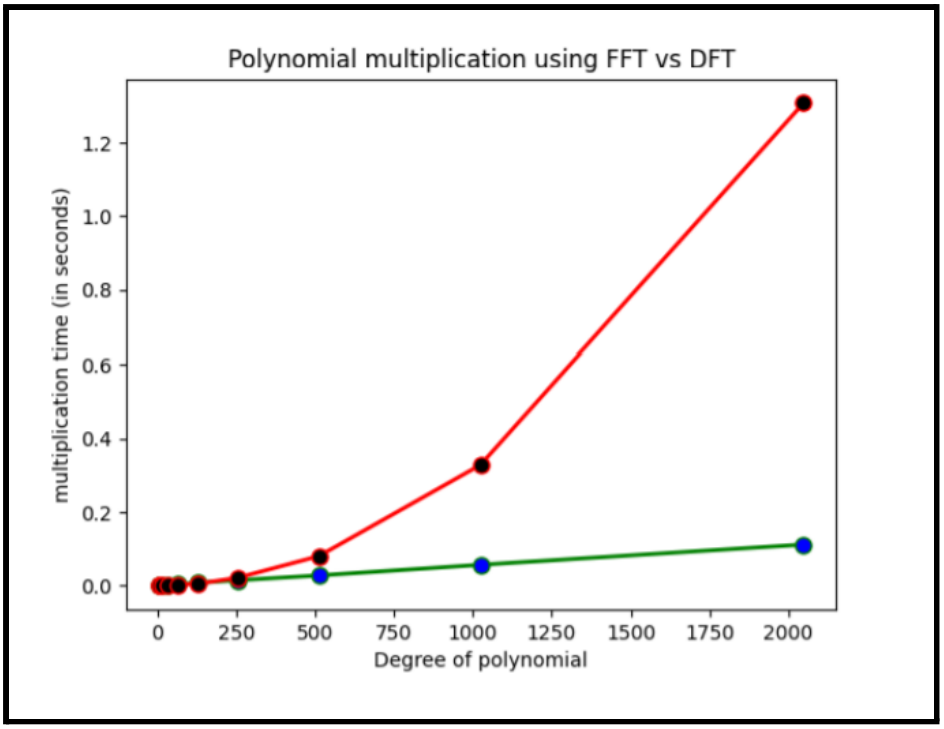
```
try:
    for test in range(len(parameters)):
            VA = fft(parameters[test])
            print(np.allclose(VA, np.fft.fft(parameters[test])))
            print("Test Case", (test+1), "for the function FFT passed")
    except:
        print("Test Case", (test+1), "for the function FFT failed")
```

```
True
Test Case 1 for the function FFT passed
True
Test Case 2 for the function FFT passed
True
Test Case 3 for the function FFT passed
True
Test Case 4 for the function FFT passed
True
Test Case 5 for the function FFT passed
True
Test Case 6 for the function FFT passed
True
Test Case 7 for the function FFT passed
True
Test Case 8 for the function FFT passed
True
Test Case 9 for the function FFT passed
True
Test Case 10 for the function FFT passed
True
Test Case 11 for the function FFT passed
True
Test Case 12 for the function FFT passed
True
Test Case 13 for the function FFT passed
True
Test Case 14 for the function FFT passed
True
Test Case 15 for the function FFT passed
True
Test Case 16 for the function FFT passed
True
Test Case 17 for the function FFT passed
True
Test Case 18 for the function FFT passed
```

iii) Pointwise multiplying results of Step (ii) to produce C(x) in P-V form

```
C = []
for i in range(PA.shape[0]):
    C.append(PA[i]*PB[i])
```





iv) RSA encrypt (128-bit , 256-bit and 512-bit) ,with a public key, the C(x) in PV form, for transmission security and decrypt it with a private key and verify.

In the RSA algorithm for encryption and decryption of messages, we have followed certain steps in order to make the algorithm as efficient as possible:

1. We started off with generating two large odd numbers (128 bit, 256 bit, 512 bit). For this, we used the getrandbits() function from Python's random library by passing the bit length as a parameter. This function returned an integer on which we performed bitwise or, in order to make it odd. Here is a snippet of he function:

```
#generate two large odd numbers of a specific bit size
def generateLargeOdd(bitSize):
    a = random.getrandbits(128) | 1
    #print(a)
    return a
```

2. We then performed a primality check on these two numbers. We first checked if the numbers were pseudoprimes by using base-2 Fermat's theorem. This algorithm used modular exponentiation for efficient execution (since it is faster than conventional evaluation method). The modular exponentiation speeds up the process of finding modulus by multiplying and dividing (by bitwise shifting) as shown below

```
#Primality check using Fermat Theorem to identify base 2 pseudoprimes
def checkPseudoprimeFermat(num):
   base = 2
   if (checkPrimeModularExponentiation(num-1, base, num)) == 1:
       return True #returns true if prime
   return False #returns false if composite


#Primality check using Modular Exponentiation to the base 2
def checkPrimeModularExponentiation(num2, base, num):
   res = 1
   base%=num
   if (base == 0) :
       res = 0
       return res
   while (num2 > 0) :
```

```
        #print("base:", base)
        if ((int(num2) & 1) == 1) :
            res = (res * base) % num
        base = (base * base) % num
        num2 = int(num2) >> 1 # num = num/2
    #print("/n/n/nRESULTTTTTT:", res)
    return res #if res is 1 or n-1 it is prime
```

3. Since the Fermat's theorem is not quite efficient, due to its limit on the base value, we improved our primality check by using the Rabin Miller algorithm for primality checking. Here, we choose a base value from a given set of values. The basic idea that we followed was that even if there is a single base for which the relation the relation $a^{n-1} \equiv 1 mod(n)$ holds true, it indicates that n is a prime number. We implement this algorithm using two functions, namely, checkPrimeMillerRabin() and millerHelper().

```
#Helper function for Miller Rabin test
def millerHelper(d, n):
    a = 2 + random.randint(1, n - 4)
    #print("base:", a)
    x = checkPrimeModularExponentiation(d, a, n)
    if (x == 1 or x == n - 1):
        return True
    while (d != n - 1):
        x = (x * x) % n
        d *= 2
        if (x == 1):
            return False
        if (x == n - 1):
            return True
    return False


#Primality check using Miller Rabin test
def checkPrimeMillerRabin(n):
    k = 4 #no. of iterations
    if (n <= 1 or n == 4):
        return False
    if (n <= 3):
        return True
```

```
    d = n - 1
    while (d % 2 == 0):
        d //= 2
    for i in range(k): #Iterating k times
        if not millerHelper(d, n):
            return False
    return True
```

4. In order to add more surety in terms of whether the two large numbers are prime or not, we implemented the trial division loop as well. This took comparatively more time to run due to its nested nature. Below is the function to implement the same.

```
#Primality check using Trial Division
def checkPrimeTrialDivision(a):
    for i in range(2, math.ceil(pow(a, 0.5))):
        if a%i == 0:
            return False
    return True
```

5. The next step was finding ‘e’, a small odd integer which is relatively prime to phi(n) where phi(n) = (p-1)(q-1) assuming p and q are our two large numbers. For this, we iterated through a loop starting with 3 with a step count of 2, and checking if the gcd(phi(n), e) = 1.

```
#Finding gcd between two numbers
def gcd(a, b):
    if(b == 0): return a
    if a > b: gcd(b, a%b)
    else: gcd(a, b%a)

#Find relative prime of a number
def relativePrime(p,q):
    phi = (p-1)*(q-1)
    for e in range(3, phi, 2):
        if gcd(phi, e) == 1:
            print("e:", e)
            return e
```

6. Once we found e value, we found the modular multiplicative inverse of e, namely d. They satisfy the equation ed $\equiv 1 mod(phi(n))$. d was found using extended Euclid theorem on e and phi(n). e and phi(n) satisfy the equation es + phi(n)t = 1 (since gcd(e, phi(n))=1) using extended Euclid's would give us the value of d.

Below are the functions to implement this:

```
#Extended Euclid
def egcd(a, b):
    s = 0; old_s = 1
    t = 1; old_t = 0
    r = b; old_r = a
    while r != 0:
        quotient = old_r // r
        old_r, r = r, old_r - quotient * r
        old_s, s = s, old_s - quotient * s
        old_t, t = t, old_t - quotient * t
    return old_r, old_s, old_t # return gcd, x, y

#To find modular multiplicative inverse of e
def modularMultiplicativeInverse(e, p, q):
    phi = (p-1)*(q-1)
    gcd, x, y = egcd(e, phi)
    if x < 0:
        x += phi
    return x
```

7. Once we get e and d, The pair (e, n) is used to generate the public key used for RSA encryption. Let M be the message we wish to encrypt. To transform the message M associated with a public key P = (e,n), we compute $P(M) = M^e \bmod(n)$

```
def rsaEncrypt(M, e, n):
    C = []
    for m in M:
        C_inner = []
        for i in m:
            ex = checkPrimeModularExponentiation(e, i, n)
            #ex = (pow(m, e))%n
            C_inner.append(ex)
        C.append(C_inner)
    return C
```

8. We get the cyphertext C after RSA encryption. To decrypt this, we use the pair (d, N) in order to get our message back. To transform the cyphertext C associated with the secret key S = (d, n), compute $S(C) = C^d \bmod(n)$

```
def rsaDecrypt(C, d, n):
    MD = []
    for c in C:
        MD_inner = []
        for i in c:
            de = checkPrimeModularExponentiation(d, i, n)
            #de = (pow(c, d))%n
            MD_inner.append(chr(de))
        MD_in = "".join(MD_inner)
        MD.append(MD_in)
    return MD
```

In order to deal with complex numbers as well, since the point-value representation of C(x) may be in complex form, we are converting the message to its ASCII equivalent and then encrypting the same using Public Key. Once encrypted, the ASCII values, or the crypt text, are decrypted using Private Key to get back the original message.

Test Case 1:
Message M- [5,7, 8, 10]

```
(base) Anuritas-MacBook-Air-6:Advanced-Algorithm-FFT-RSA_Encryption anuritabose$ python rsa.py
First large prime number p: 142083649243986852182301293212099402573

Second large prime number q: 269388348948388122586513853014537025333

n = p * q: 38275679682399512399855779318992136141769632544333644961751293726317866381809

e: 5

d: 765513593647990247997115586379842722827163210922825399739649571601824599078l

phi(n): 382756796823995123998557793189921361413581605461412699869824785800912299539O4

Original message M: [5, 7, 8, 10]

Encrypted message (Cyphertext) C: [[418195493], [503284375], [550731776], [282475249, 254803968]]

Decrypted message M_decrypted: [5, 7, 8, 10]
```

Test Case 2:
Message M- [0j, (2+1.1102230246251565e-16j), (1.4997597826618576e-32-2.4492935982947064e-16j), (2+4.440892098500626e-16j)]

```
(base) Anuritas-MacBook-Air-6:Advanced-Algorithm-FFT-RSA_Encryption anuritabose$ python rsa.py
First large prime number p: 32609860780045427484027279205490597294 9

Second large prime number q: 8206146970639864971702947147731928660 9

n = p * q: 267601310253157528855665994108498609840230418699098051659621021455897319399 41

e: 3

d: 178400873502105019237110662738999073224099211949353014942698665880383377869 23

phi(n): 267601310253157528855665994108498609836148817924029522414047998820575066803 84

Original message M: [0j, (2+1.1102230246251565e-16j), (1.4997597826618576e-32-2.4492935982947064e-16j), (2+4.440892098
500626e-16j)]

Encrypted message (Cyphertext) C: [[110592, 1191016], [64000, 125000, 79507, 117649, 97336, 117649, 117649, 110592, 12
5000, 125000, 132651, 110592, 125000, 140608, 157464, 125000, 148877, 117649, 148877, 157464, 148877, 1030301, 91125,
117649, 157464, 1191016, 68921], [64000, 117649, 97336, 140608, 185193, 185193, 166375, 148877, 185193, 166375, 175616
, 125000, 157464, 157464, 117649, 175616, 148877, 166375, 157464, 1030301, 91125, 132651, 125000, 91125, 125000, 97336
, 140608, 140608, 185193, 125000, 185193, 132651, 148877, 185193, 175616, 125000, 185193, 140608, 166375, 110592, 1574
64, 140608, 1030301, 91125, 117649, 157464, 1191016, 68921], [64000, 125000, 79507, 140608, 97336, 140608, 140608, 110
592, 175616, 185193, 125000, 110592, 185193, 175616, 148877, 110592, 110592, 157464, 125000, 157464, 1030301, 91125, 1
17649, 157464, 1191016, 68921]]

Decrypted message M_decrypted: [0j, (2+1.1102230246251565e-16j), (1.4997597826618576e-32-2.4492935982947064e-16j), (2+
4.440892098500626e-16j)]
```

v) Implement 1-D Inverse FFT (I-FFT) on C(x), in PV form (Interpolation) to get C(x) in Coefficient form (CR) Polynomial.

```
def inv_fft(x):
    x = np.asarray(x, dtype=complex)
    x_conj = np.conjugate(x)

    y = fft(x_conj)

    y = np.conjugate(y)
    y = y / x.shape[0]

    return y
```

To get back coefficients of C(x) , we carry out "Interpolation" at the same n complex roots of unity.
Interpolation is the process of multiplying by "inverse of VanderMonde Matrix" V. VanderMonde matrix V, is constructed with the same n complex roots

of unity, which are the evaluation points for A(x) and B(x)

vi) Verify correctness of C(x) , by comparing with the coefficients generated by a Elementary "Convolution For Loop" on the Coefficients of A(x) and B(x).

```
import numpy as np
from fft import dft,inv_dft,inv_fft,fft

def multiply(A, B):
    m=len(A)
    n=len(B)
    prod = [0] * (m + n - 1);
    for i in range(m):
        for j in range(n):
            prod[i + j] += A[i] * B[j];
    return prod;
```

```
#driver code
#Randomly generating the two coefficient arrays with size 4
param1 = list(np.random.randint(low = 0, high = 1000, size = 4))
param2 = list(np.random.randint(low = 0, high = 1000, size = 4))
#The result generated by "convolution for loop" method
print("brute force multiplication: \n", multiply(param1,param2),"\n")

#degree bound extension
l = [0 for i in range(0,4)]
param1.extend(l)
param2.extend(l)

PA = fft(param1)
PB = fft(param2)

#pointwise multiply
C = []
for i in range(PA.shape[0]):
    C.append(PA[i]*PB[i])

#print the answer obtained by inverse fft
print("The answer obtained by calling inverse fft function
\n",inv_fft(C),"\n")
```

Result output verification,

```
The answer obtained by brute force multiplication:
 [32760, 205806, 401208, 364521, 410825, 270366, 19175]

The answer obtained by calling inverse fft function
 [3.27600000e+04-6.54186911e-11j 2.05806000e+05-3.77975828e-11j
 4.01208000e+05-3.64514589e-12j 3.64521000e+05-2.36849732e-11j
 4.10825000e+05+8.60544870e-11j 2.70366000e+05+2.95050252e-11j
 1.91750000e+04+1.40365727e-11j 5.82076609e-11+3.45226877e-11j]
```

vii) Implement a 2-D FFT and 2-D I-FFT module using your 1-D version

```
def twodfft(matrix):
    # randomly generate two matrices of the same size
    # one matrix to store the result of FFT of row vectors
    # another one to store the result of FFT of column vectors of the
matrix generated in the previous step
    z_rows = np.zeros((mat_size,mat_size),dtype=complex)
    z_cols = np.zeros((mat_size,mat_size),dtype=complex)

    # store the FFT of row vectors in z_rows
    # every row of z_rows contains FFT of row vector of "matrix"
    for i in range(len(matrix)):
        z_rows[i] = fft(matrix[i])

    # apply FFT on column vectors and store the resultant column vector
in z_cols as columns
    # every column of z_cols contains the FFT of column vectors of
z_rows
    for i in range(len(z_rows[0])):
        z_cols[:,i] = fft(z_rows[:,i])
    return z_cols

#print(z_cols)
def inv_2dfft(z_cols):
    i_rows = np.zeros((mat_size,mat_size),dtype=complex)
    i_cols = np.zeros((mat_size,mat_size),dtype=complex)

    for i in range(len(z_cols[0])):
        i_rows[i] = inv_fft(z_cols[i])
    for i in range(len(z_cols[0])):
        i_cols[:,i] = inv_fft(i_rows[:,i])
```

```
    return i_cols
```

viii) Verify your of Step (vii) correctness on a Grayscale matrix (which has random integer values in the range 0-255; 255 → White & 0 → Black))

Taking the following parameters,
Where lower limit is 0 and upper limit is 255

```
mat_size = 512
upper_limit = 255
lower_limit=0
matrix =
np.random.randint(lower_limit,upper_limit,size=(mat_size,mat_size))
```

Verified the output of 2dfft with with numpy's np.fft.fft2 function and the outputs of inv_2dfft with np.fft.ifft2,

```
#driver program
z_cols = twodfft(matrix)
i_cols = inv_2dfft(z_cols)

if(np.allclose(z_cols, np.fft.fft2(matrix))):
    print("The output of the 2D fft matches the numpy function's output
for the 2D fft")

if(np.allclose(i_cols, np.fft.ifft2(z_cols))):
    print("The output of the 2D fft matches the numpy function's output
for the 2D fft")
```

Runtime log:

```
The output of the 2D fft matches the numpy function's output for the 2D fft
The output of the 2D fft matches the numpy function's output for the 2D fft
```

ix) Apply your 2D-FFT on TIFF/JPG (lossless) Grayscale image and drop Fourier coefficients below some specified magnitude and save the 2D-image to a new file.

```
from google.colab.patches import cv2_imshow
img=cv2.imread('img3.jpg',0)
resize_img = cv2.resize(img  , (512, 512))

cv2_imshow(resize_img)
```

View the image using matplotlib.pyplot

% compression – permanent Lossy compression, by sorting and retaining only coefficients greater than some(quantization) value. Rest are made 0.

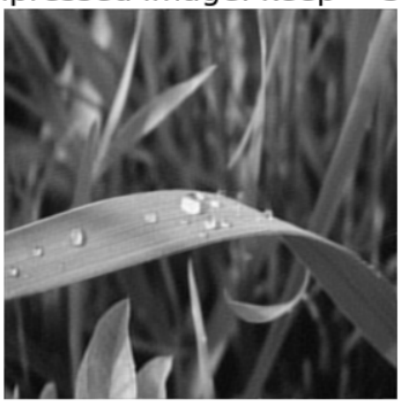
Applying 2D I-FFT, on the Quantized Grayscale image and rendering it to observe Image Quality.

We convert the fourier transform into a vector and sort the values from largest to smallest and obtain only the top 10,5,1 and 0.2 percent of the threshold.

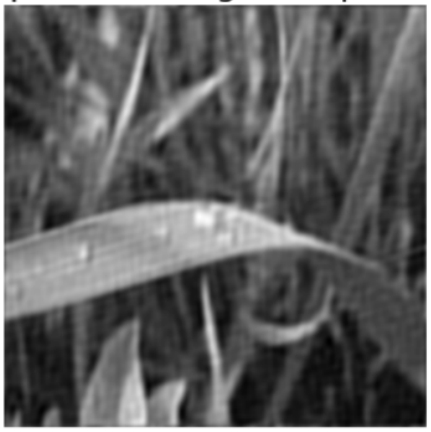
```
from google.colab.patches import cv2_imshow
img=cv2.imread('img3.jpg',0)
resize_img = cv2.resize(img  , (512, 512))

new_img = twodfft(resize_img)
newimg_sort = np.sort(np.abs(new_img.reshape(-1)))

#Keeping largest 10,5,1,and 0.2 percent of the fourier coefficients
#Convert fourier transform into a vector and sort all the values from largest to smallest
#Obtain only the top 10,5,1 and 0.2 percent as the threshold
for keep in (0.1,0.05,0.01,0.002):
    thresh = newimg_sort[int(np.floor((1-keep)*len(newimg_sort)))]
    ind = np.abs(new_img)>thresh
    BLow = new_img*ind
    Alow = inv_2dfft(BLow).real
    plt.figure()
    plt.imshow(Alow,cmap='gray')
    plt.axis('off')
    plt.title('Compressed image: keep = ' + str(keep*100) + "%")
```

Compressed image: keep = 10.0%

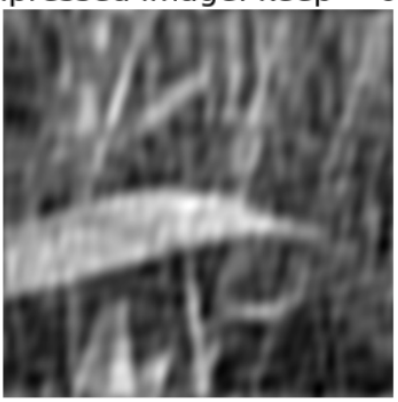
Compressed image: keep = 5.0%



Compressed image: keep = 1.0%



Compressed image: keep = 0.2%



We obtained similar results when we use numpy's fft (np.fft.ftt2()) and ifft (np.fft.ifft2()) function

```
Bt = np.fft.fft2(B)
Btsort = np.sort(np.abs(Bt.reshape(-1)))

for keep in (0.1,0.05,0.01,0.02):
    thresh = Btsort[int(np.floor((1-keep)*len(Btsort)))]
    ind = np.abs(Bt)>thresh
    Atlow = Bt*ind
    Alow = np.fft.ifft2(Atlow).real
    plt.figure()
    plt.imshow(256-Alow,cmap='gray')
    plt.axis('off')
    plt.title('Compressed image: keep = ' + str(keep*100) + "%")
```

Compressed image: keep = 10.0%

Compressed image: keep = 5.0%

Compressed image: keep = 1.0%

Compressed image: keep = 2.0%

Observations and Learning Outcomes

1. The 1-D DFT has a time complexity of O(n^2) time as compared to the 1-D FFT which has a time complexity of O(n log n) time. DFTn, which is evaluating the coefficient vector A(x) at the n complex nth units of unity, has been reduced to evaluating two polynomials at the n/2 complex, n/2th roots of unity (i.e., two instances of DFTn/2).
2. 1-D FFT is then performed on the row vectors followed by 1-D FFT on the column vectors of the transformed matrix to get 2-D FFT.
3. RSA is a public key cryptographic algorithm in which two different keys are used to encrypt and decrypt the message. Each user has to generate two keys: a private key and a public key. The public key is circulated or published to all and hence others are aware of it whereas, the private key is secretly kept with the user only. A sender has to encrypt the message using the intended receiver's public key. Only the intended receiver can crack the message. In between the communication no one can harm the confidentiality of the message as the message can only be decrypted by the intended receiver’s private key which is only known to that receiver.
4. Image compression - The quantized grayscale image is obtained by considering the largest 10%, 5% , 1% and 0.2% of fourier coefficients. As the percentage of largest fourier coefficients considered is reduced, image quality reduces.