ADVANCED ALGORITHM ASSIGNMENT REPORT

UE19CS311

TEAM MEMBERS:

Ananya Uppal	PES1UG19CS058
Nihal Ramaswamy	PES1UG19CS297

ABOUT THE ASSIGNMENT

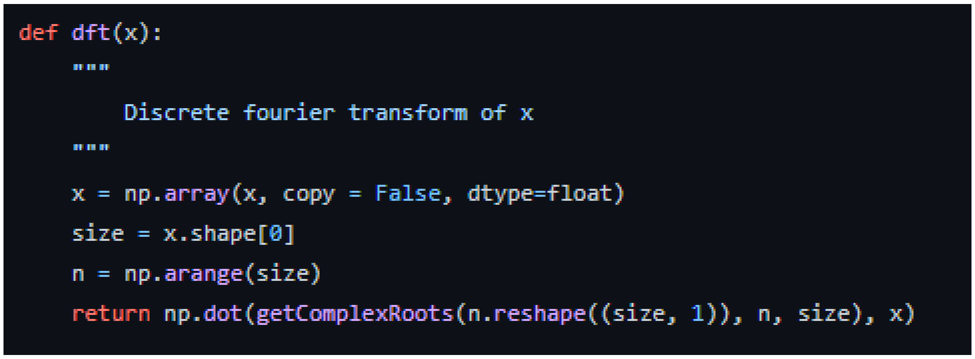
This assignment encompassed learning about and implementing in python several complex algorithms such as DFT , FFT and Inverse FFT. We also explored RSA cryptography and lossless decomposition of images. Each step of the process will be explained in detail further.

STEP 1 : 1D - DFT

The discrete Fourier transform (DFT) converts a finite sequence of equally-spaced samples of a function into a same-length sequence of equally-spaced samples of the discrete-time Fourier transform (DTFT), which is a complex-valued function of frequency. The samples used in a DFT are the nth roots of unity which provide several useful properties such as :

- $1 + w + w^2 = 0$
- The roots are in a geometric progression with r = i * 2* pi / n

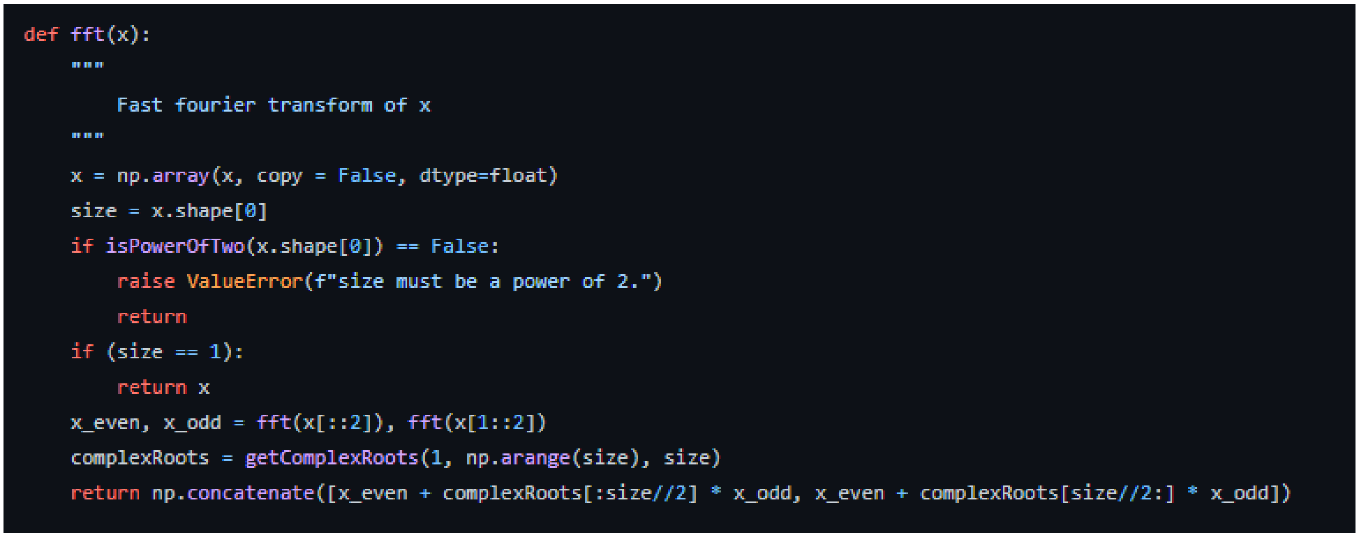
The DFT transformation can be applied for polynomial multiplication i.e. multiplying two polynomial functions A(x) and B(x) in Point value form.



```
def dft(x):
    """
        Discrete fourier transform of x
    """
    x = np.array(x, copy = False, dtype=float)
    size = x.shape[0]
    n = np.arange(size)
    return np.dot(getComplexRoots(n.reshape((size, 1)), n, size), x)
```

The code gets the n complex roots of unity and then converts the array (which is the CR representation to PV representation). The polynomial is essentially evaluated at the n roots of unity so that complexity of the operation is **O(n^2)**.

STEP 2 : 1-D FFT



```
def fft(x):
    """
        Fast fourier transform of x
    """
    x = np.array(x, copy = False, dtype=float)
    size = x.shape[0]
    if isPowerOfTwo(x.shape[0]) == False:
        raise ValueError(f"size must be a power of 2.")
        return
    if (size == 1):
        return x
    x_even, x_odd = fft(x[::2]), fft(x[1::2])
    complexRoots = getComplexRoots(1, np.arange(size), size)
    return np.concatenate([x_even + complexRoots[:size//2] * x_odd, x_even + complexRoots[size//2:] * x_odd])
```

This solution uses a divide and conquer strategy where each polynomial is broken down into its constituent parts of even and odd powers.

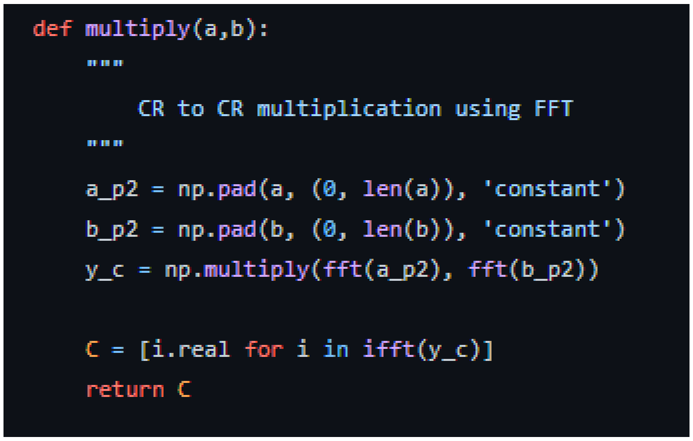
The FFT function is then called recursively with these two components. The recursive function will look something like this :

$$T(n) = 2T(n/2) + O(n)$$

It has a complexity of **O(nlogn)**.

STEP 3 : POINTWISE MULTIPLICATION

Multiply the two polynomials point by point to produce the result C(x) in point-value form. The time complexity of this operation is **O(n)**.



```
def multiply(a,b):
    """
        CR to CR multiplication using FFT
    """
    a_p2 = np.pad(a, (0, len(a)), 'constant')
    b_p2 = np.pad(b, (0, len(b)), 'constant')
    y_c = np.multiply(fft(a_p2), fft(b_p2))

    C = [i.real for i in ifft(y_c)]
    return C
```

STEP 4 : RSA ENCRYPTION

RSA is an asymmetric encryption-decryption algorithm which uses the concept of public key and secret key. The idea of RSA is that the prime number chosen for factorisation is too large to be factored efficiently. If this is possible , the integrity of the secret key is compromised.

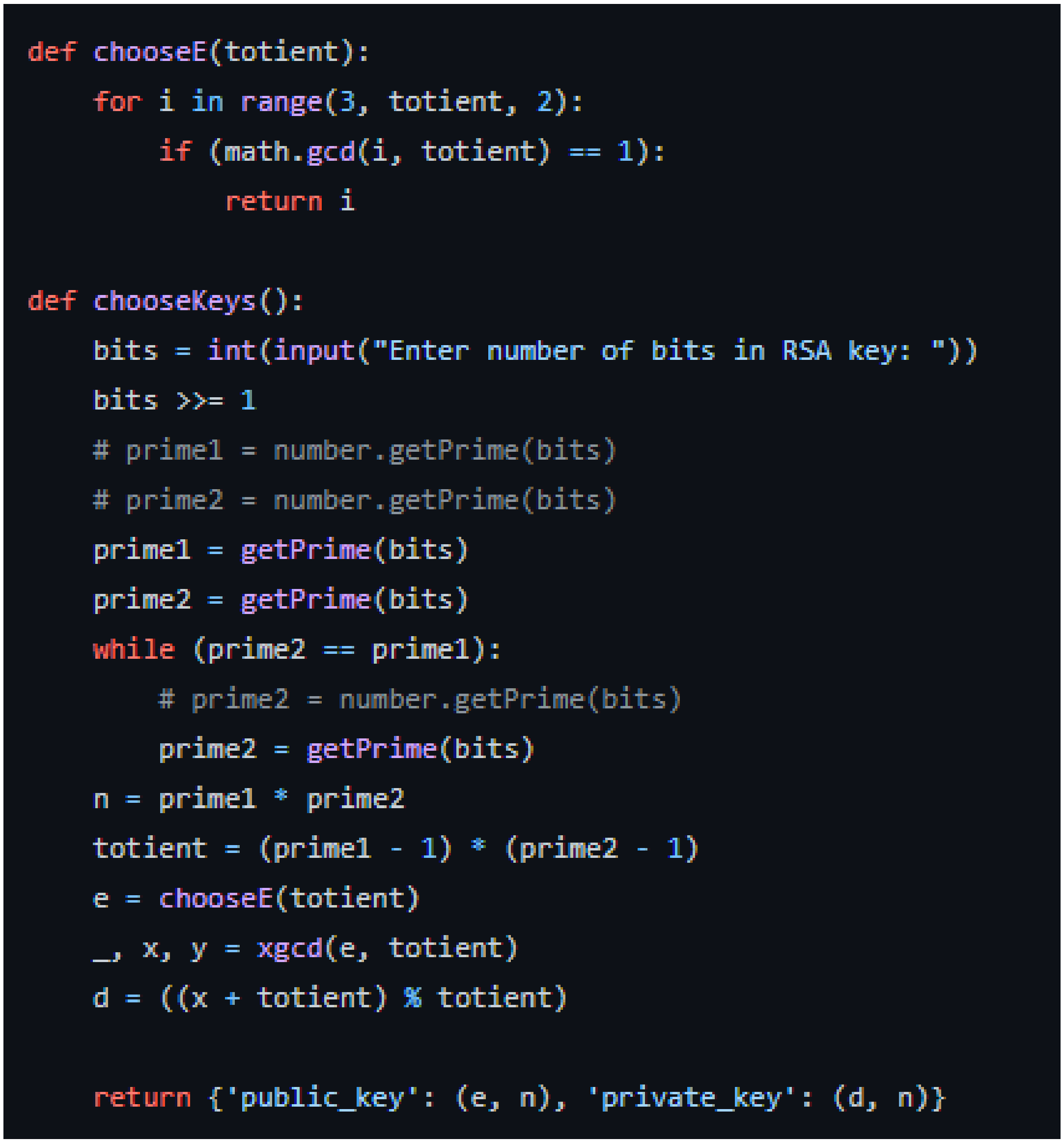
Thus the strength of encryption depends on the size of the key which is why we try encryption with 3 different sizes - 128 bit , 256 bit and 512 bit.

The public key is made of 2 integers - n and e where n = p * q (p and q are 2 prime numbers) and e is a small integer.

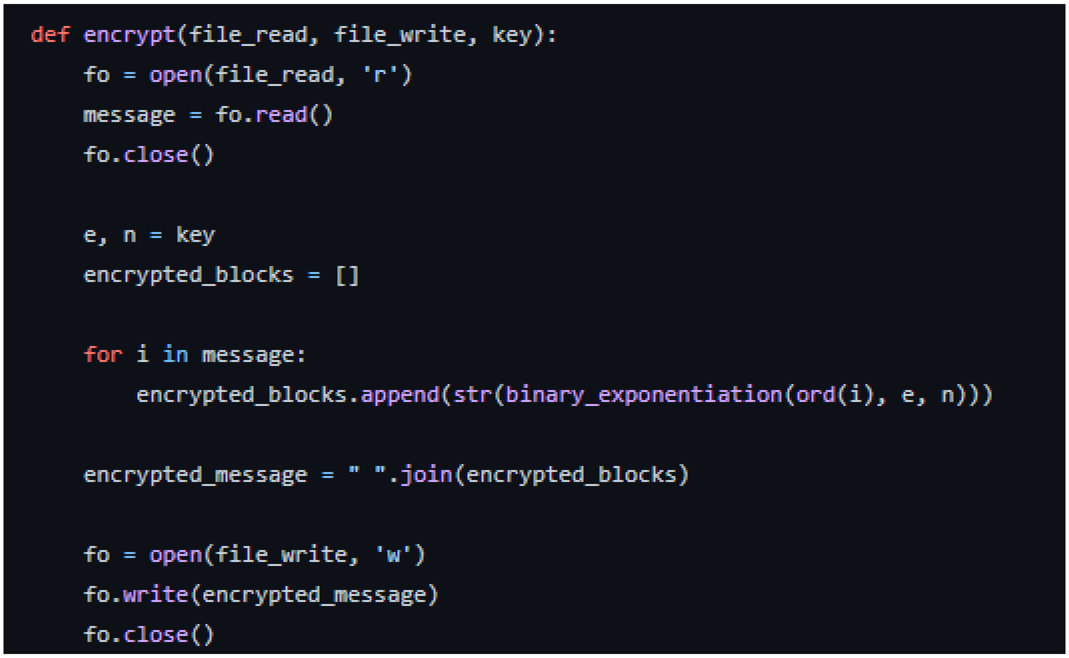
The secret key d is derived from k*ϕ(n) + 1 where ϕ = (p-1)(q-1).

The encrypted message is c = M^e mod n and the decrypted message is c’ = c^d mod n.

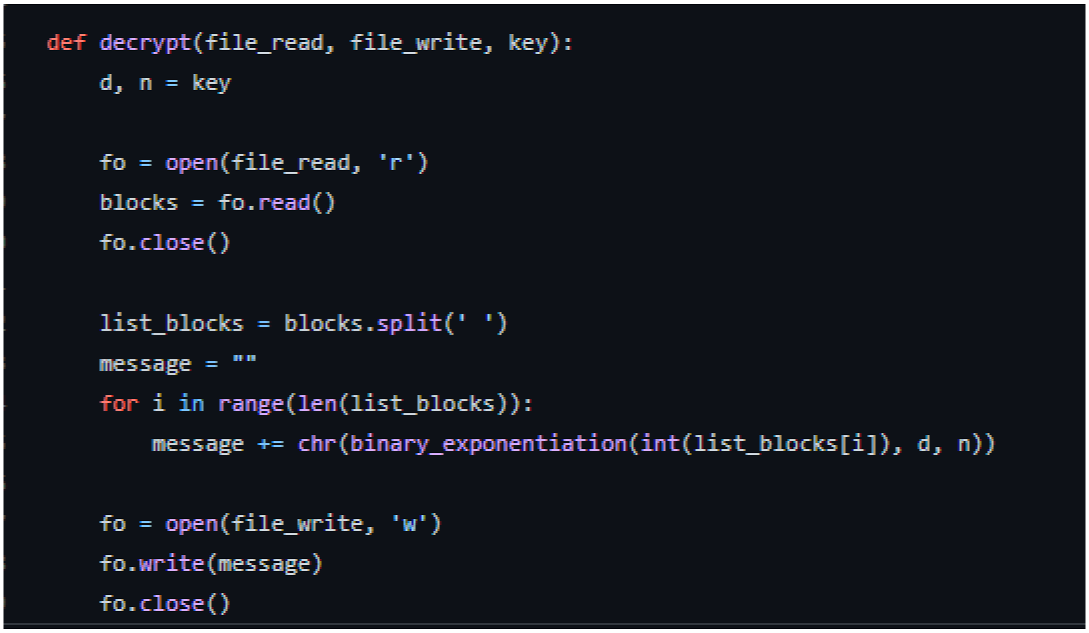
The generation of prime numbers can be done in 2 ways : either using built in python functions or generating them from scratch and validating them using Miller Rabin Method.



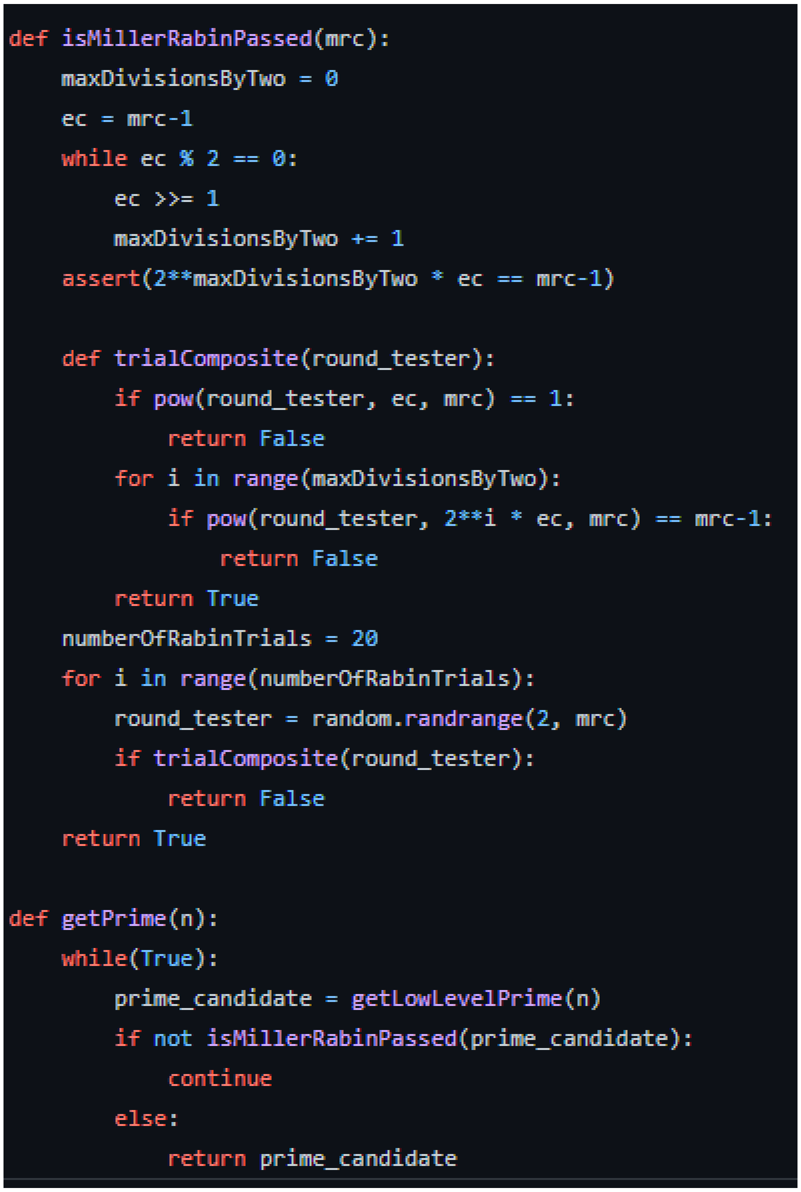
```python
def chooseE(totient):
    for i in range(3, totient, 2):
        if (math.gcd(i, totient) == 1):
            return i

def chooseKeys():
    bits = int(input("Enter number of bits in RSA key: "))
    bits >>= 1
    # prime1 = number.getPrime(bits)
    # prime2 = number.getPrime(bits)
    prime1 = getPrime(bits)
    prime2 = getPrime(bits)
    while (prime2 == prime1):
        # prime2 = number.getPrime(bits)
        prime2 = getPrime(bits)
    n = prime1 * prime2
    totient = (prime1 - 1) * (prime2 - 1)
    e = chooseE(totient)
    _, x, y = xgcd(e, totient)
    d = ((x + totient) % totient)

    return {'public_key': (e, n), 'private_key': (d, n)}
```



```python
def encrypt(file_read, file_write, key):
    fo = open(file_read, 'r')
    message = fo.read()
    fo.close()

    e, n = key
    encrypted_blocks = []

    for i in message:
        encrypted_blocks.append(str(binary_exponentiation(ord(i), e, n)))

    encrypted_message = " ".join(encrypted_blocks)

    fo = open(file_write, 'w')
    fo.write(encrypted_message)
    fo.close()
```



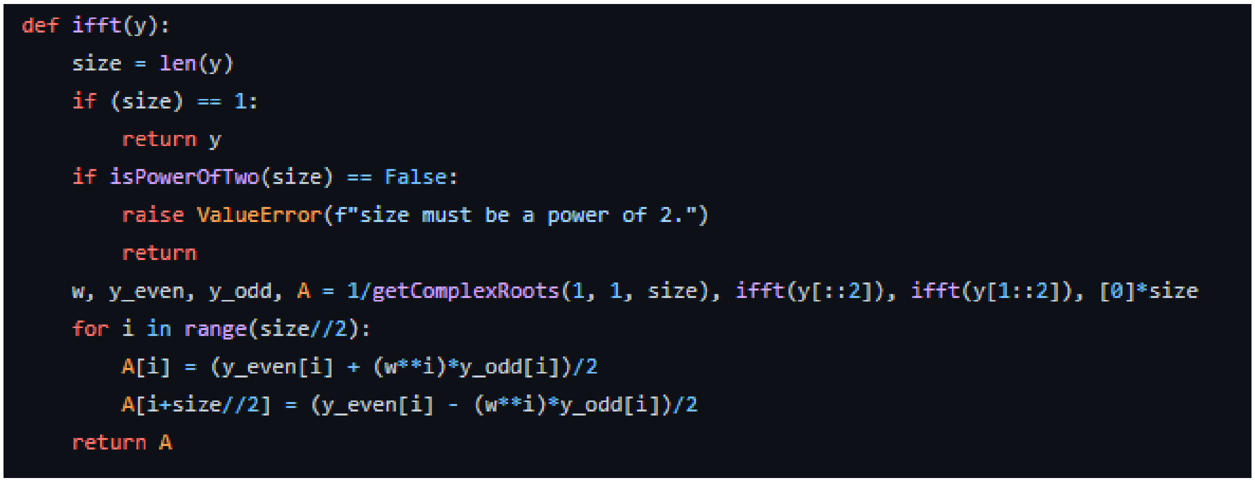
```python
def decrypt(file_read, file_write, key):
    d, n = key

    fo = open(file_read, 'r')
    blocks = fo.read()
    fo.close()

    list_blocks = blocks.split(' ')
    message = ""
    for i in range(len(list_blocks)):
        message += chr(binary_exponentiation(int(list_blocks[i]), d, n))

    fo = open(file_write, 'w')
    fo.write(message)
    fo.close()
```



```python
def isMillerRabinPassed(mrc):
    maxDivisionsByTwo = 0
    ec = mrc-1
    while ec % 2 == 0:
        ec >>= 1
        maxDivisionsByTwo += 1
    assert(2**maxDivisionsByTwo * ec == mrc-1)

    def trialComposite(round_tester):
        if pow(round_tester, ec, mrc) == 1:
            return False
        for i in range(maxDivisionsByTwo):
            if pow(round_tester, 2**i * ec, mrc) == mrc-1:
                return False
        return True
    numberOfRabinTrials = 20
    for i in range(numberOfRabinTrials):
        round_tester = random.randrange(2, mrc)
        if trialComposite(round_tester):
            return False
    return True

def getPrime(n):
    while(True):
        prime_candidate = getLowLevelPrime(n)
        if not isMillerRabinPassed(prime_candidate):
            continue
        else:
            return prime_candidate
```

STEP 5 : 1-D INVERSE FFT

Once we multiply the 2 polynomials in PV form , we need to convert them back to the CV form. This can be done using Inverse FFT by a process called Interpolation. The time complexity of this process is **O(nlogn)**.



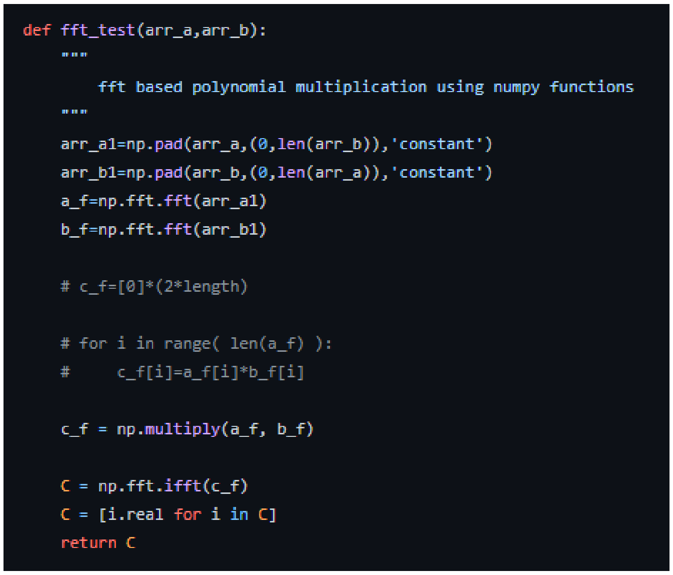
```python
def ifft(y):
    size = len(y)
    if (size) == 1:
        return y
    if isPowerOfTwo(size) == False:
        raise ValueError(f"size must be a power of 2.")
        return
    w, y_even, y_odd, A = 1/getComplexRoots(1, 1, size), ifft(y[::2]), ifft(y[1::2]), [0]*size
    for i in range(size//2):
        A[i] = (y_even[i] + (w**i)*y_odd[i])/2
        A[i+size//2] = (y_even[i] - (w**i)*y_odd[i])/2
    return A
```

This function has been implemented using recursion, separating the odd and even degree polynomials.

The parameter should be a power of 2 so as to compute the roots of unity till n. We take the inverse of the roots to construct the Vandermonde matrix.

STEP 6 : VERIFY IFFT RESULTS

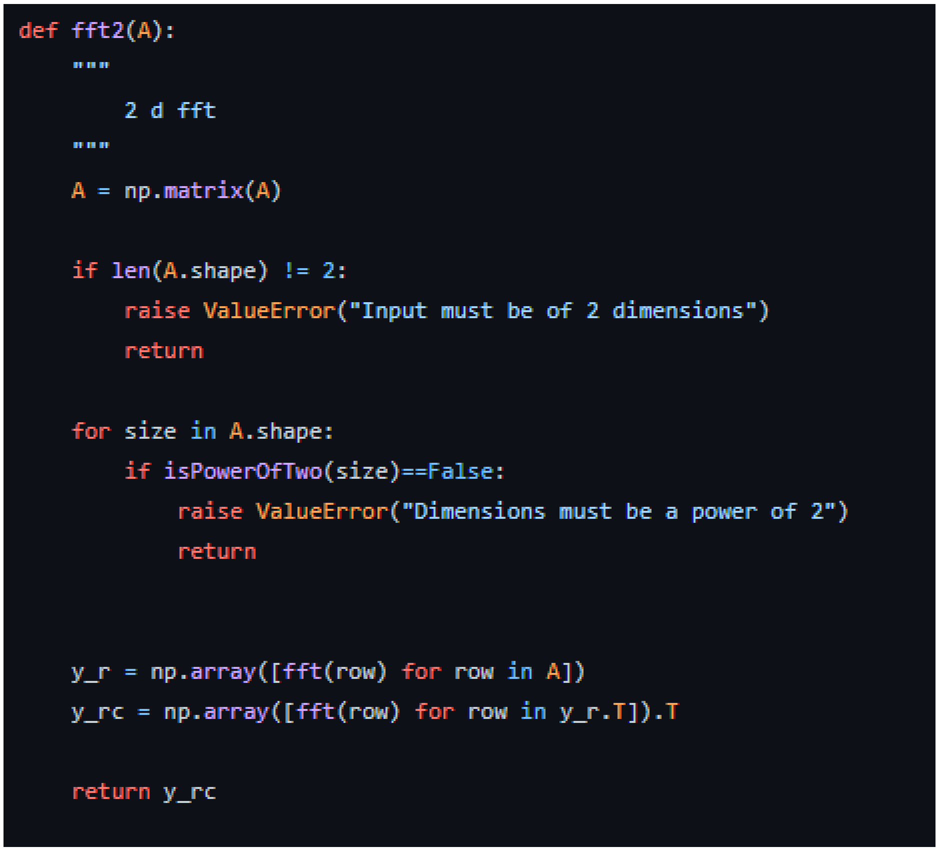
Compare each coefficient of the generated C(x) with the list produced by the inbuilt numpy function and validate the output C(x).



```python
def fft_test(arr_a,arr_b):
    """
        fft based polynomial multiplication using numpy functions
    """
    arr_a1=np.pad(arr_a,(0,len(arr_b)),'constant')
    arr_b1=np.pad(arr_b,(0,len(arr_a)),'constant')
    a_f=np.fft.fft(arr_a1)
    b_f=np.fft.fft(arr_b1)

    # c_f=[0]*(2*length)

    # for i in range( len(a_f) ):
    #     c_f[i]=a_f[i]*b_f[i]

    c_f = np.multiply(a_f, b_f)

    C = np.fft.ifft(c_f)
    C = [i.real for i in C]
    return C
```

STEP 7 : 2D FFT AND 2D IFFT

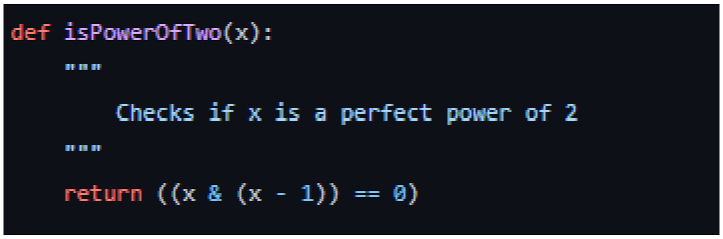
This step is done using 1D FFT. The FFT operation is applied



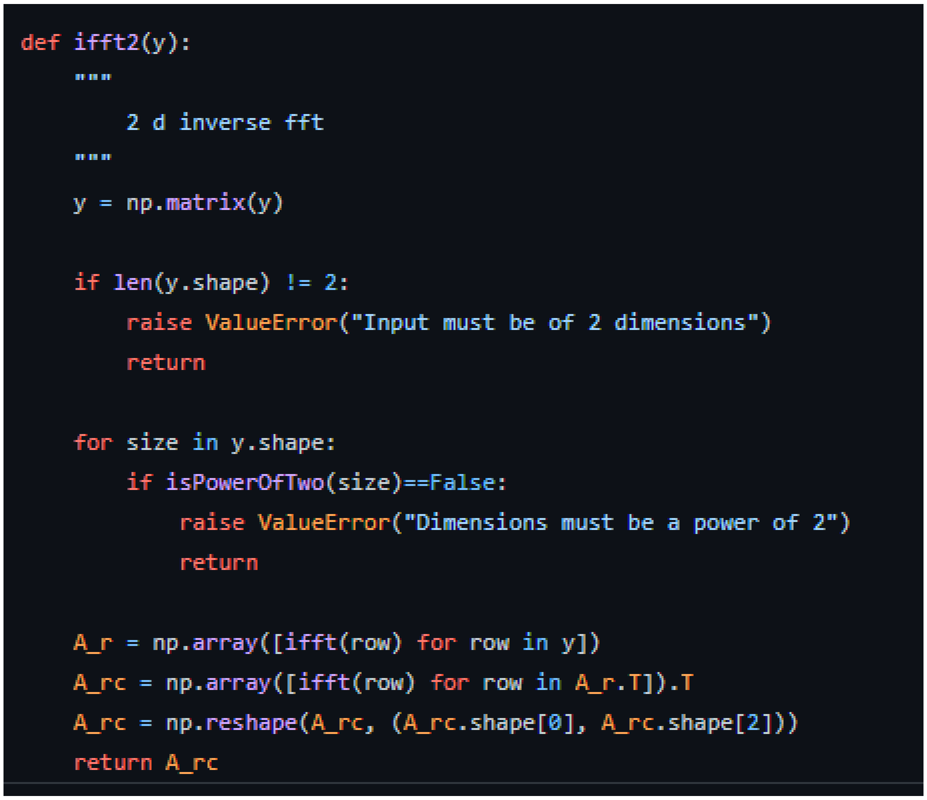
```python
def fft2(A):
    """
        2 d fft
    """
    A = np.matrix(A)

    if len(A.shape) != 2:
        raise ValueError("Input must be of 2 dimensions")
        return

    for size in A.shape:
        if isPowerOfTwo(size)==False:
            raise ValueError("Dimensions must be a power of 2")
            return


    y_r = np.array([fft(row) for row in A])
    y_rc = np.array([fft(row) for row in y_r.T]).T

    return y_rc
```

The FFT operation is applied to each row in the matrix and then applied to each column by taking a transpose of the matrix.

In the Inverse FFT operation , the IFFT is taken in a similar manner and the array is reshaped.



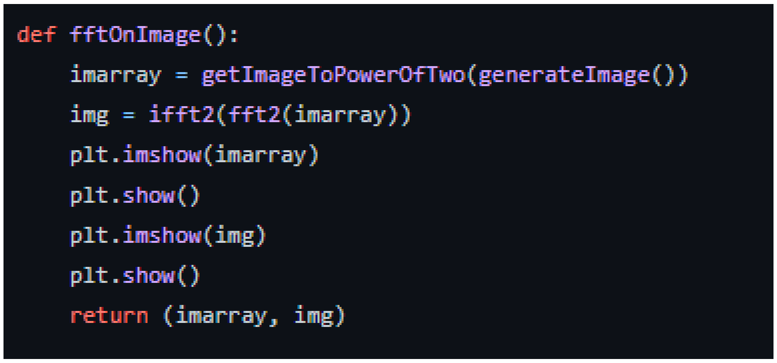
```python
def isPowerOfTwo(x):
    """
        Checks if x is a perfect power of 2
    """
    return ((x & (x - 1)) == 0)
```



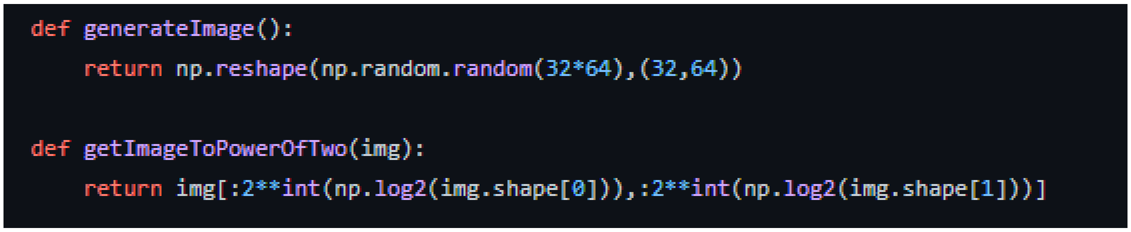
```
def ifft2(y):
    """
        2 d inverse fft
    """
    y = np.matrix(y)

    if len(y.shape) != 2:
        raise ValueError("Input must be of 2 dimensions")
        return

    for size in y.shape:
        if isPowerOfTwo(size)==False:
            raise ValueError("Dimensions must be a power of 2")
            return

    A_r = np.array([ifft(row) for row in y])
    A_rc = np.array([ifft(row) for row in A_r.T]).T
    A_rc = np.reshape(A_rc, (A_rc.shape[0], A_rc.shape[2]))
    return A_rc
```

STEP 8 : VERIFY 2D FFT AND IFFT ON GRAYSCALE IMAGE

First a random image is generated by choosing values between 0-256 for all pixels. Then FFT and IFFT operations are performed and the two images are displayed for comparison.



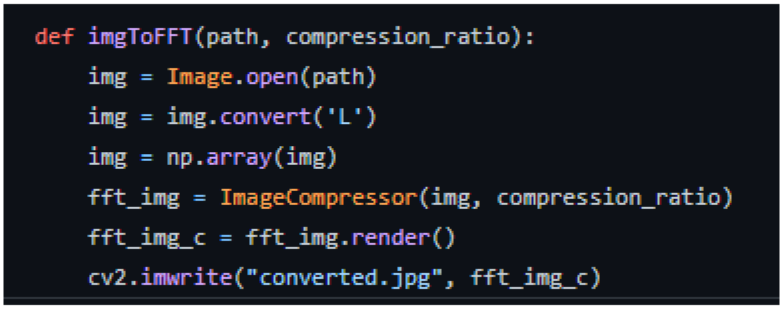
```
def fftOnImage():
    imarray = getImageToPowerOfTwo(generateImage())
    img = ifft2(fft2(imarray))
    plt.imshow(imarray)
    plt.show()
    plt.imshow(img)
    plt.show()
    return (imarray, img)
```



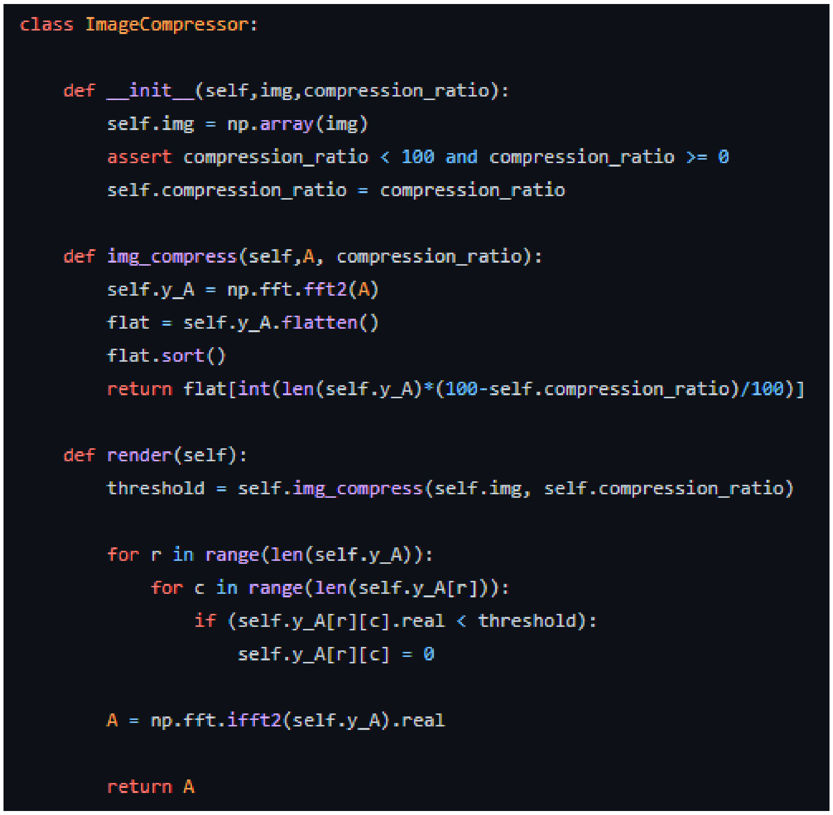
```
def generateImage():
    return np.reshape(np.random.random(32*64),(32,64))

def getImageToPowerOfTwo(img):
    return img[:2**int(np.log2(img.shape[0])),:2**int(np.log2(img.shape[1]))]
```

STEP 9 : 2-D FFT ON LOSSLESS GRAYSCALE IMAGE

The RGB image is first converted to a grayscale image by using python function.The compression ratio is taken and the ImageCompressor class is called.



```
def imgToFFT(path, compression_ratio):
    img = Image.open(path)
    img = img.convert('L')
    img = np.array(img)
    fft_img = ImageCompressor(img, compression_ratio)
    fft_img_c = fft_img.render()
    cv2.imwrite("converted.jpg", fft_img_c)
```

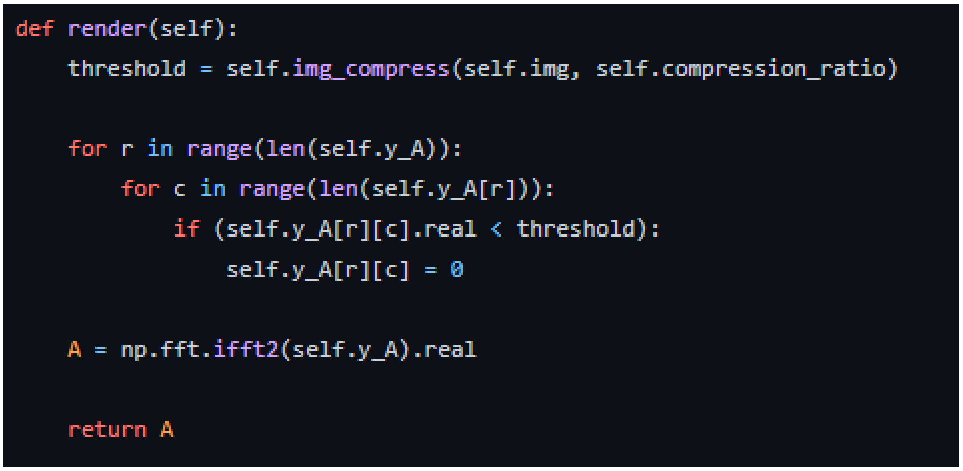


```
class ImageCompressor:

    def __init__(self,img,compression_ratio):
        self.img = np.array(img)
        assert compression_ratio < 100 and compression_ratio >= 0
        self.compression_ratio = compression_ratio

    def img_compress(self,A, compression_ratio):
        self.y_A = np.fft.fft2(A)
        flat = self.y_A.flatten()
        flat.sort()
        return flat[int(len(self.y_A)*(100-self.compression_ratio)/100)]

    def render(self):
        threshold = self.img_compress(self.img, self.compression_ratio)

        for r in range(len(self.y_A)):
            for c in range(len(self.y_A[r])):
                if (self.y_A[r][c].real < threshold):
                    self.y_A[r][c] = 0

        A = np.fft.ifft2(self.y_A).real

        return A
```

First we check if the compression ratio lies between 0-100 and then the FFT function is applied on the image passed. The returned object is flattened to a 1D array and sorted in ascending order. Then the first (100-compression_ratio)% elements are chosen from the list and this proves to be the threshold. After this , only those values are chosen from the matrix whose values are less than the threshold value and a compressed image is generated.

STEP 10 : 2-D IFFT ON GRAYSCALE

IFFT operation is performed on the image obtained after filtering out values less than the threshold value. Then the real and compressed images are rendered and compared.



```
def render(self):
    threshold = self.img_compress(self.img, self.compression_ratio)

    for r in range(len(self.y_A)):
        for c in range(len(self.y_A[r])):
            if (self.y_A[r][c].real < threshold):
                self.y_A[r][c] = 0

    A = np.fft.ifft2(self.y_A).real

    return A
```

TIMINGS FOR OPERATIONS

Input Size : 2048 bits

RSA bits : 512

Brute Force Multiplication : 0.0746634 ~ 0.075 seconds

Numpy FFT : 0.001382083 ~ 0.00138 seconds

Custom FFT : 0.1443710327 ~ 0.14437 seconds

DFT : 7.556415 seconds

LEARNING OUTCOMES

During the course of this assignment , we obtained in depth knowledge of the inner workings of Discrete Fourier Transformations and its optimization Fast Fourier Transform. We also understood how to implement these algorithms from scratch in python. We also learned the basics of cryptography and the RSA algorithm and its applications. We also understood the basics of image processing and decompression using FFT and IFFT.