Advanced Algorithms Assignment

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

Ramya C | **Ramya N Prabhu**

PES1UG19CS379 | PES1UG19CS380

5 F

INTRODUCTION

The aim of this assignment is to implement 1-D & 2-D Fourier Transform & RSA Encryption on a M x N matrix to achieve Fast Polynomial Multiplication, Secure transmission and Lossy Image compression and analyse their running time with the help of modules like numpy and random.

Implementing 1D DFT [using the DFT Matrix]

We implemented DFT using a special type of Vandermonde matrix called the DFT matrix. This transform matrix, when applied to any signal using matrix multiplication, gives its Fourier transform. This is generated using the following code:

$$\begin{bmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \omega^3 & \cdots & \omega^{N-1} \\ 1 & \omega^2 & \omega^4 & \omega^6 & \cdots & \omega^{2(N-1)} \\ 1 & \omega^3 & \omega^6 & \omega^9 & \cdots & \omega^{3(N-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{N-1} & \omega^{2(N-1)} & \omega^{3(N-1)} & \cdots & \omega^{(N-1)(N-1)} \end{bmatrix}$$

(A) the template for DFT matrix

```
omga=np.exp((-2j*np.pi)/v.shape[-1])
W=[]
for i in range(v.shape[-1]):
  p=[]
  for j in range(v.shape[-1]):
    p.append(omga**(i*j))
  W.append(p)
```

(B) Code for the same

The DFT matrix is then multiplied with the array of coefficients to produce the fourier transform.

Implementing 1D FFT

The fast Fourier transform is an algorithm that computes the discrete Fourier transform of a sequence in O(nlogn) time. Following is the code implementing it:

```
14 @arrayofarrays
15 def fft(x):
16     x=np.array(x,dtype=np.complex)
17     N = len(x)
18 
19     if N == 1:
20         return [x[0]]
21 
22     X = [0] * N
23 
24     even = fft(x[:N:2])
25     odd = fft(x[1:N:2])
26 
27     for k in range(N//2):
28         w = math.e**(-2j*math.pi * k/N)
29         X[k] = even[k] + w * odd[k]
30         X[k + N//2] = even[k] - w * odd[k]
31     return X
```

The FFT algorithm exploits 3 crucial facts about the DFT matrix:

Lemma 30.3 (Cancellation lemma)
For any integers $n \geq 0$, $k \geq 0$, and $d > 0$,
$\omega_{dn}^{dk} = \omega_n^k$.

,

Lemma 30.5 (Halving lemma)
If $n > 0$ is even, then the squares of the n complex nth roots of unity are the $n/2$ complex $(n/2)$th roots of unity.

And

Lemma 30.6 (Summation lemma)
For any integer $n \geq 1$ and nonzero integer k not divisible by n,

$$\sum_{j=0}^{n-1} \left(\omega_n^k\right)^j = 0 .$$

- The algorithm separately use the even-indexed and odd-indexed coefficients of A ,to define two new polynomials of half the degree-bound
- And then follows A(x)=A[0](x^2) + x*A[1](x^2)
- These results are then evaluated and recombined.

Checking the correctness of of our Implementation

The correctness of our implementation is check against the fft function from the numpy package

```
[93]  1 m=np.random.randint(100, size=(1024))
      2 np.allclose(dft(m), fft(m))
```

```
True
```

```
[55]  1 m=np.random.randint(100, size=(1024))
      2 np.allclose(dft(m), np.fft.fft(m))
```

```
True
```

```
[56]  1 m=np.random.randint(100, size=(1024))
      2 np.allclose(dft(m),fft(m))
```

```
True
```

Implementing 2D FFT

Implementation of 2D FFT is done using the 1D FFT we have generated. In this We apply 1D FFT to the Rows and then to the Columns.

```
def fft2_d(matrix):
   fftRows = np.asarray([np.fft.fft(row) for row in matrix],dtype=np.complex_)
   fftColumns=np.asarray(transpose([np.fft.fft(column) for column in transpose(fftRows)]),dtype=np.complex_)
   return fftColumns
```

Checking the correctness of of our Implementation

```
a =np.array([[1, 2, 3, 4], [5, 6, 7, 8]])
fft2_d(a)
```

```
array([[ 36.+0.j,  -4.+4.j,  -4.+0.j,  -4.-4.j],
       [-16.+0.j,   0.+0.j,   0.+0.j,   0.+0.j]])
```

```
np.fft.fft2(a)
```

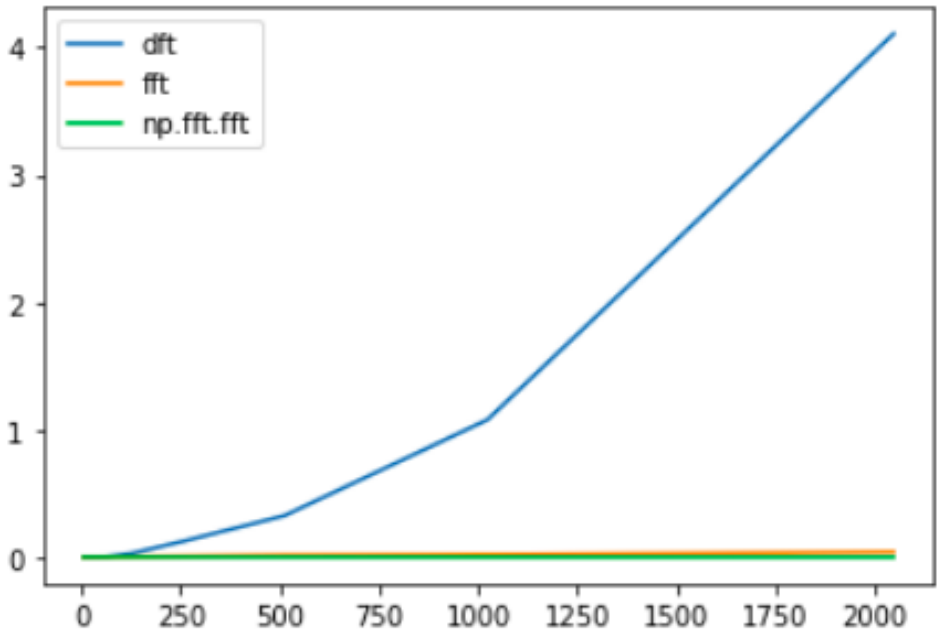
```
array([[ 36.+0.j,  -4.+4.j,  -4.+0.j,  -4.-4.j],
       [-16.+0.j,   0.+0.j,   0.+0.j,   0.+0.j]])
```

```
np.allclose(fft2_d(a), np.fft.fft2(a))
```

```
True
```

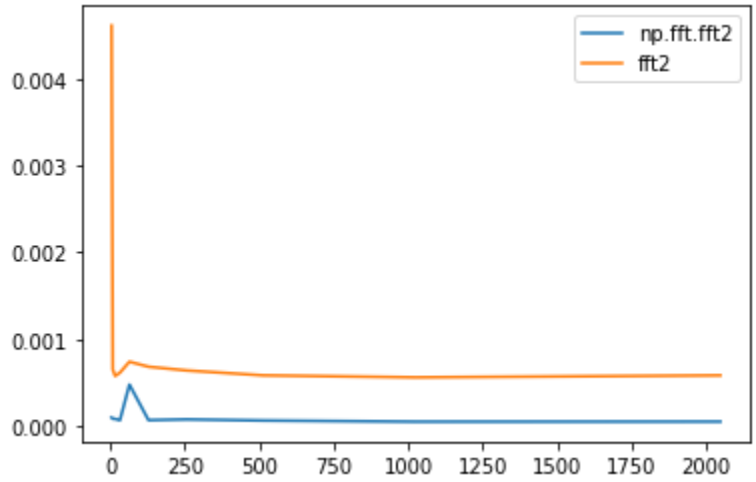
Running time of the three 1D fourier transform functions

Now that we know our function works, let's check how long it takes to transform vectors of sizes 4, 8, 16, 32, 64, … 1024 and 2048



From the graph it is clear that fft is much faster than dft [n/log(n) times faster]. So, fft will now be used to transform various coefficient vectors for arbitrary polynomials A and B of length 4, 8, 16, 32, 64, … 1024 and 2048

Running time of 2D FFT VS np.fft.fft2



Pointwise Multiply

The result of the transform of A and B is then point_multiplied using the code below.

```
def pointwise_mul(list1, list2):
  products=[]
  for num1, num2 in zip(list1, list2):
    products.append(num1 *num2)
  return products
```

It should be noted here that the below given code does not return the standard PV form. Rather, it only returns the value of A(x)*B(X) at these points. This is to save time that would have been otherwise lost in zipping the output into a list of tuples, which would have to be unzipped in further calculations

```
[66]  1 for i in range(len(entriesA)):
      2   print(pointwise_mul(fft(entriesA[i]), fft(entriesB[i])))
      3
```

```
[(7178+0j), (710.0000000000003-539.9999999999999j), (-1462-1.7194041060028884e-13j), (709.9999999999995+540.0000000000003j)]
[(146231+0j), (-4041.423868799799+1566.4751801064747j), (-2403.0000000000005-395.9999999999999j), (15743.423868799806-1153.5248198935233j), (-245-1.1803145850182191e-12
[(659016+0j), (12051.667967265996+33990.33132822239j), (2983.5984438143087-905.7710319181165j), (-7475.352994943844-10796.30263579838j), (-32622.999999999996+2408.99999
[(3248693+0j), (-9412.910630342862+5133.0709061958205j), (-17401.221687166566+16367.927190912698j), (-1158.7557976062956-469.89222423865436j), (-2742.799733257132+12153
[(7483320+0j), (32972.13232163631-122850.3662226159j), (-21196.766749956747-1156.904947907888j), (-24486.419591872163-14622.63709416721j), (-17997.444650827507-12529.783
[(36091703+0j), (-38151.30040005942+100575.48267009156j), (-518.4281003691058+47580.17234914916j), (-950.7617966200341+173948.1568194746j), (5698.305133437809-1603.4518
[(145210725+0j), (157.28562169577708+43343.2642927665j), (-7147.597124630274-8504.549195624386j), (724433.2844059415+249481.20842294284j), (-55083.84484627833-14825.850
[(616239162+0j), (-86451.07953147218-124110.79674935363j), (501695.83648425946+617605.7825523835j), (85253.03838004256-341096.2455561919j), (167991.07207965173-73223.830
[(2702966970+0j), (-78433.02638840591-419523.26407775324j), (1191558.2793049728-75918.1390105523j), (-538863.7399643447-809976.8687345114j), (-400904.8762671007+227052.6
[(10258524504+0j), (1059223.5410343066+98647.96381844359j), (221561.47620775807-1772084.8011191934j), (-334197.7481489994-1152281.0345379757j), (-183980.29117437592-6006
```

Implementing IFFT

The Inverse Fast Fourier Transform(IFFT) is a fast algorithm that performs reverse or backward or inverse Fourier Transform which undoes the process of FFT

The formula used here is as follows:

$$x_i = \frac{1}{N}\sum_{n=0}^{N-1} F_n e^{\frac{2\pi j}{N}ni}$$

Code Used

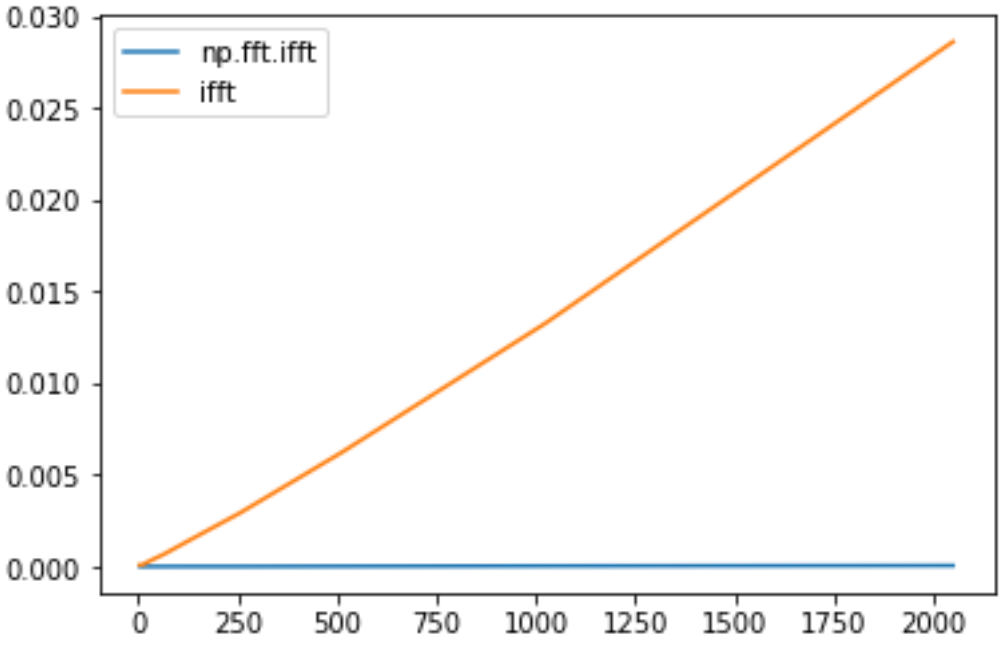
```
@arrayofarrayys
def ifft(x):
    x = np.asarray(x, dtype=complex)
    x_conjugate = np.conjugate(x)

    inverse = fft(x_conjugate)

    inverse = np.conjugate(inverse)
    inverse = inverse / x.shape[0]
    return inverse
```

Checking the correctness of of our Implementation

```
m=np.random.randint(100, size=(1024))
np.allclose(ifft(m), np.fft.ifft(m))
```

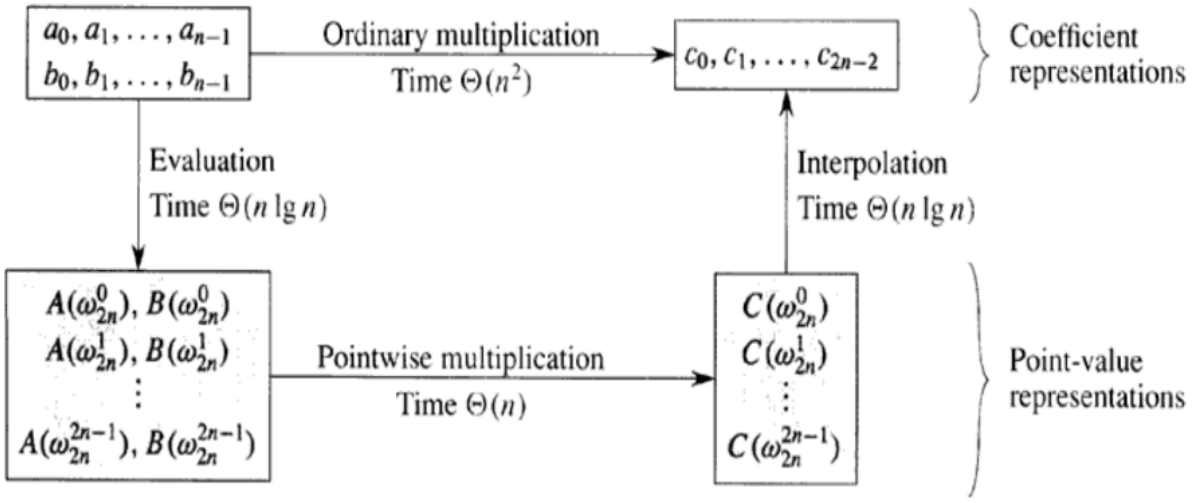
```
True
```

Running time of IFFT VS np.fft.ifft



Comparing Multiplication using FFT and IFFT vs regular multiplication.

Polynomial multiplication can be computed faster in the way indicated by the following block diagram:

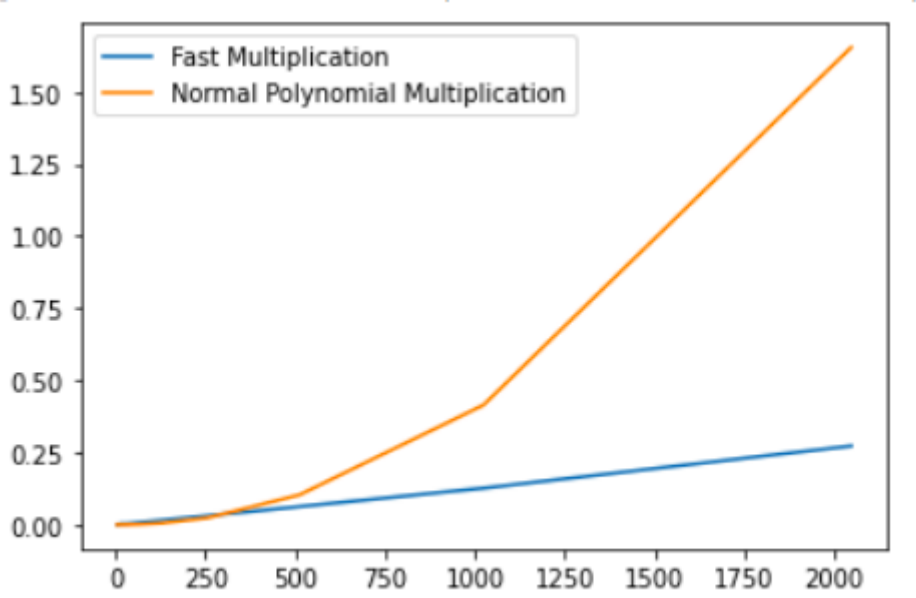


For the evaluation step, the fourier transform produces the point value forms of the two polynomials intended for multiplication.This is then multiplied using the function `pointwise_mul(a,b)`. Interpolation is carried out using the IFFT we implemented. The whole procedure is encapsulated in the below given function

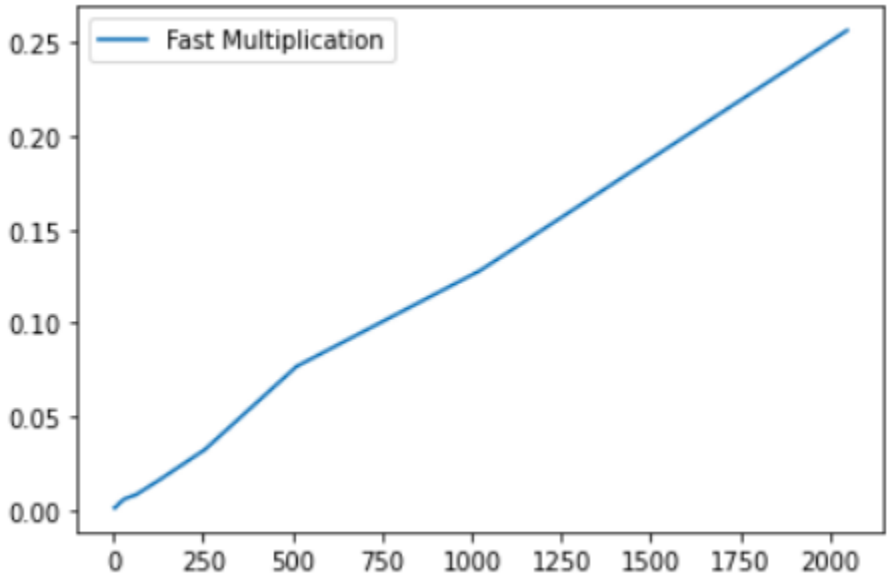
```
def fast_multiplication(arr1, arr2):
  n=len(arr1)
  for i in range(n, 2*n):
    arr1.append(0)
    arr2.append(0)
  arr3=np.asarray(fft(arr1), dtype=complex)
  arr4=np.asarray(fft(arr2), dtype=complex)
  products=pointwise_mul(arr3, arr4)
  return np.round(ifft(products))
```

The correctness of this implementation and this technique can be easily verified by checking its results against the output of an elementary convolution for loop

Now that we are sure of its veracity, we can now proceed to compare its speed with the speed of polynomial multiplication carried out in the traditional manner using the distributive property



As evident from the graph above, the running time of multiplication carried using extractions, pointwise multiplication and interpolation is much faster than its conventional counterpart



Implementing 2D IFFT

Implementation of 2D IFFT is done using the 1D IFFT we have generated. In this We apply 1D IFFT to the Rows and then to the Columns.

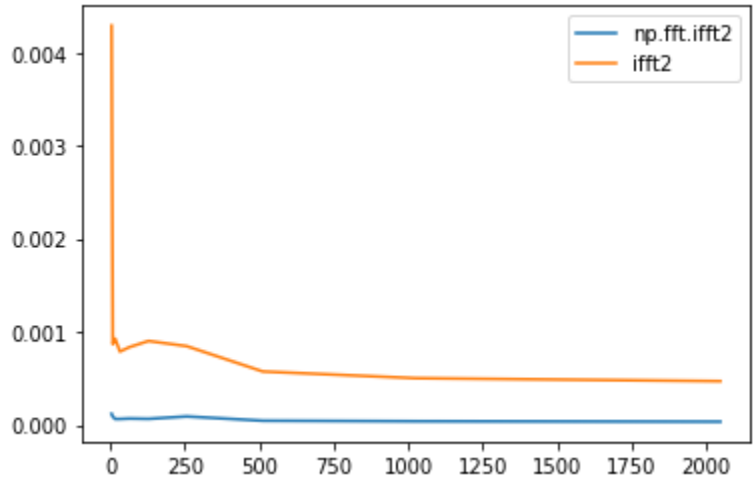
```
#2-D Inverse FFT
def ifft2d(matrix):
   fftRows = np.asarray([ifft(row) for row in matrix],dtype=np.complex_)
   fftcolumns=np.asarray(transpose([ifft(column) for column in transpose(fftRows)]),dtype=np.complex_)
   return fftcolumns
```

Checking the correctness of of our Implementation

```
#
a=[[1,2,3,4],[5,6,7,8]]
np.allclose(ifft2d(a), np.fft.ifft2(a))
```

```
True
```

Running time of 2DIFFT VS np.fft.ifft2



Generating Random Large Prime Numbers

We generate Large Numbers based on the number of bits and verify it while using the Miller-Rabin Test and also using Trial DIvision Loop

```
def generate_random_number(no_of_bits):
    return random.randrange(2**(no_of_bits-1)+1, 2**no_of_bits - 1)
```

```
def generate_prime_list(n):
    prime_list=list()
    for i in range(1,n+1):
        if i>1:
            for j in range(2,i):
                if( i % j )==0:
                    break
            else:
                prime_list.append(i)
    return prime_list
prime_number_list=generate_prime_list(350)
```

```
def low_level_prime(n):
    while True:
        num=generate_random_number(n)
        for i in prime_number_list:
            if num%i==0 and i**2<=num:
                break
            else:
                return num
```

Testing for primality

The Miller Rabin Test and Trial Composite Loop

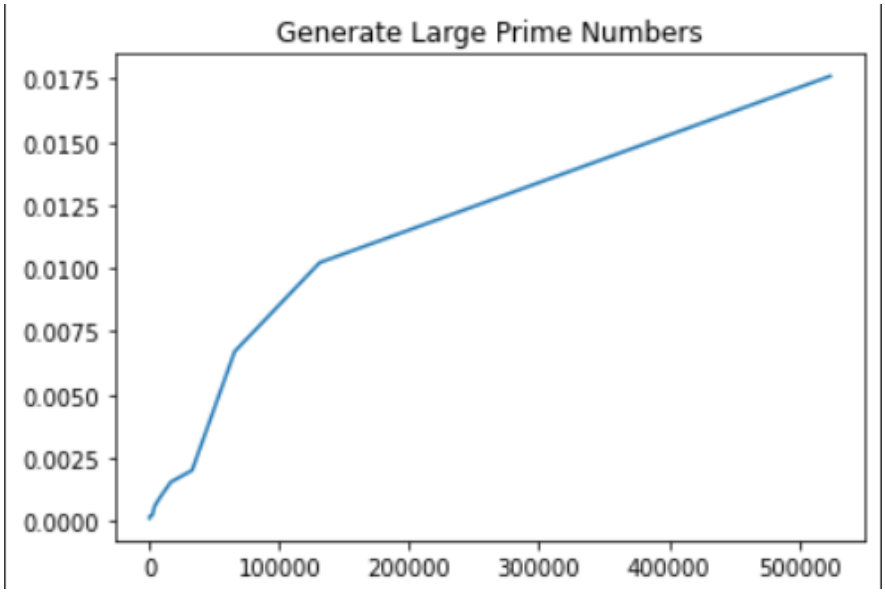
```
def Miller_Rabin(n):
  x = 0
  y = n-1
  while y % 2 == 0:
    y >>= 1
    x += 1
  assert(2**x * y == n-1)

  def trialComposite(t):
    if pow(t, y, n) == 1:
      return False
    for i in range(x):
      if pow(t, 2**i * y, n) == n-1:
        return False
    return True
  m = 20
  for i in range(m):
    t = random.randrange(2, n)
    if trialComposite(t):
      return False
  return True
```

Generation of Prime Numbers through the Miller Rabin test and appending it to the file

```
p_file=open('P.txt','w+')
i=100000
while i:
    P=16
    x=low_level_prime(P)
    if not Miller_Rabin(x):
        continue
    else:
        p_file.write("%d\n"%x)
        i=i-1
```

```
q_file=open('Q.txt','w+')
i=100000
while i:
    Q=16
    x=low_level_prime(P)
    if not Miller_Rabin(x):
        continue
    else:
        q_file.write("%d\n"%x)
        i=i-1
```

The graph for it's working time is as follows:



RSA

RSA is a public-key cryptosystem that is widely used for secure data transmission. It is also the oldest. Implementing RSA first required the implementation of a few helper functions.

Extended Euclidean GCD

This algorithm returns the gcd for any two numbers a and b and the values of x and y from the equation ax+by=gcd (the coefficients of Bézout's identity)

```
1 def egcd(a, b):
2     if b==0:
3       return (a,1,0)
4     else:
5       d1, x1, y1= egcd(b, a % b)
6       d= d1
7       x= y1
8       y= x1 - (a//b)*y1
9       return (d,x,y)
```

The standard Euclidean algorithm proceeds by a succession of Euclidean divisions whose quotients are not used. Only the remainders are kept. For the extended algorithm, the successive quotients are used. The implementation is based on the fact that gcd(a,b) is the smallest positive element from the set {ax+by: a,b are integers}.

Modular Inverse

A modular multiplicative inverse of an integer a is an integer x such that the product ax is congruent to 1 with respect to the modulus m that is $ax \equiv 1 \pmod{m}$

```
def modularInv(a, n):
    d, x, y = egcd(a, n)
    b=1
    if b%d==0:
        p= (x*(b/d))%n
        return p
```

This is basically the implementation of the modular linear equation solver, where b is 1. And so, it is driven by the theorem that $x_0 = x'(b/d) \bmod n$.

is a valid solution to the equation $ax \equiv b \pmod{n}$ if b is divisible by the gcd of a and n.

RSA Encryption

RSA encryption is carried out using the following procedure:

1. Select at random two large prime numbers p and q such that p ≠ q. The primes p and q might be, say, 1024 bits each.
2. Compute n = pq.
3. Select a small odd integer e , that is relatively prime to Φ(n) = (p-1)(q-1) .

   { Since by Euler's Theorem: Φ(n) = n (1 – 1/p) (1- 1/q) = (p-1)(q-1) }

4. Compute d, as the multiplicative inverse of e mod Φ(n) . [d exists and is uniquely defined. => ed ≡ 1 mod Φ(n)]
5. Publish the pair , P = (e , n) as the participant's RSA public key.
6. Keep secret the pair S = (d , n) as the participant's RSA secret key.

- To transform a message M associated with a public key P = (e , n) , compute

$$P(M) = M^e \bmod n$$

- To transform a ciphertext C associated with a secret key S = (d , n), compute

$$S(C) = C^d \bmod n \ .$$

This works because ,

$$\begin{aligned} M &= S_A(P_A(M)) \ , \\ M &= P_A(S_A(M)) \end{aligned}$$

We implemented RSA as a class. It creates an object that has the methods rsa, encrypt and decrypt.

The rsa method generates the public and private key, the encrypt method encrypts the message and the decrypt method decodes it.

The object is built for an array of values from the point value form of any given polynomial. The points are not encrypted because the points taken for the input polynomial are always the nth roots of unity [as the PV form is generated using fft.]

```
class RSA:
    def __init__(self, size):
        a,b=self.rsa()
        self.e = b[0]
        self.n= b[1]
        self.fullset=(a,b)
        self.d=a[0]
        self.size=size
```

The code for constructor

```
 9      def random_line(self,fname):
10          lines=open(fname).read().splitlines()
11          r= random.choice(lines)
12          return int(r)
```

Choosing random prime numbers

```
13  def rsa(self):
14      P=self.random_line("/content/P (1).txt")
15      Q=self.random_line("/content/Q (1).txt")
16      print(P)
17      print(Q)
18      e=7
19      tot=(P-1)*(Q-1)
20      while (not isCoprime(e, tot)):
21        el=[3, 5, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103, 107, 109, 11
22        e=random.choice(el)
23      n=P*Q
24      p=modularInv(e,tot)
25      prikey=(p,n)
26
27      pubkey=(e,n)
28      return (prikey,pubkey)
```

RSA

```
24      def encrypt(self, new):
25          msgasc=[]
26          for i in new:
27            #print(new)
28            c=str(complex(i))
29            c=c[1:-1]
30            # print("hi")
31            # print(c)
32            l=[]
33            for j in c:
34              l.append(ord(j))
35            msgasc.append(l)
36          enc=[]
37          for a in msgasc:
38            encs=[]
39            for z in a:
40              encs.append(pow(z, int(self.e), self.n))
41            enc.append(encs)
42          return enc
```

Encrypt method

```
def decrypt(self, ciph):
    b=[]
    for var in ciph:
      str1=''
      for pin in var:
        dvar=pow(pin, int(self.d), self.n)
        str1+=chr(dvar)
      b.append(str1)
    deciph=[]
    for i in b:
      deciph.append(complex(i))
    return deciph
```

Decrypt Method

```
1 L=RSA(6)
2 msg=pointwise_mul(fft(entriesA[0]),fft(entriesB[0]))
3 a=L.encrypt(msg)
4 w=L.decrypt(a)
5 np.allclose(msg,w)
```

```
[[3255, 3578, 1601, 1251, 592, 503, 1081, 2744], [1601, 1081, 1999, 1081, 503, 5, 3255, 3255, 592, 270
[(27456+0j), (4080+1226.0000000000014j), (-588+1.7830857395585468e-13j), (4080-1225.9999999999984j)]
True
```

For example:

```
]  1 print(msg)
```

```
[(27456+0j), (4080+1226.0000000000014j), (-588+1.7830857395585468e-13j), (4080-1225.9999999999984j)]
```

The test message to encrypt

```
1 for i in a:
2   print(i)
```

```
[3255, 3578, 1601, 1251, 592, 503, 1081, 2744]
[1601, 1081, 1999, 1081, 503, 5, 3255, 3255, 592, 2708, 1081, 1081, 1081, 1081, 1081, 1081, 1081, 1081, 1081, 1081, 5, 1601, 2744]
[3796, 1251, 1999, 1999, 503, 5, 2708, 3578, 1999, 4048, 1081, 1999, 1251, 3578, 4048, 3567, 1251, 1251, 1999, 1251, 1601, 592, 1999, 2279, 3796, 5, 4048, 2744]
[1601, 1081, 1999, 1081, 3796, 5, 3255, 3255, 1251, 2708, 3567, 3567, 3567, 3567, 3567, 3567, 3567, 3567, 3567, 3567, 1999, 1601, 2744]
```

Cipher

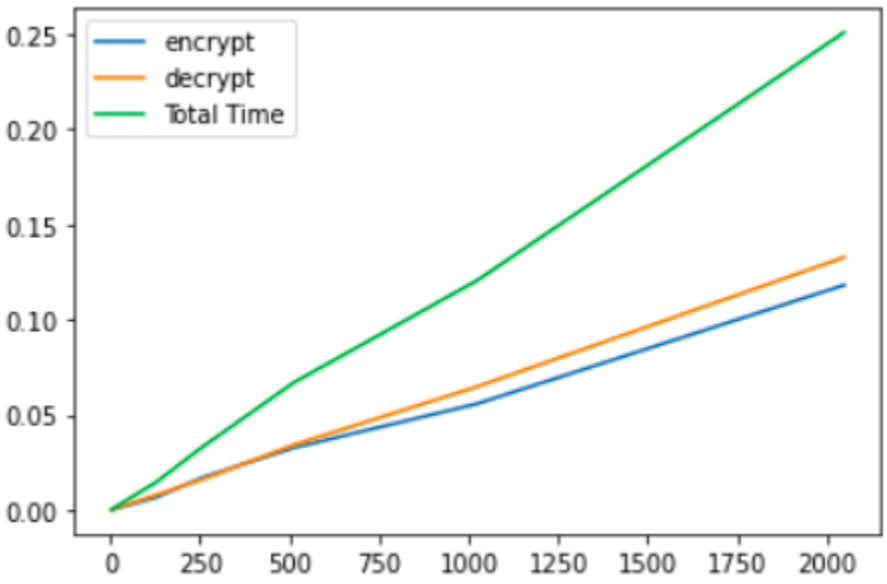
```
1 print(w)
```

```
[(27456+0j), (4080+1226.0000000000014j), (-588+1.7830857395585468e-13j), (4080-1225.9999999999984j)]
```

The decrypted message

Running time of RSA

For different values of C:



For different values of keys:

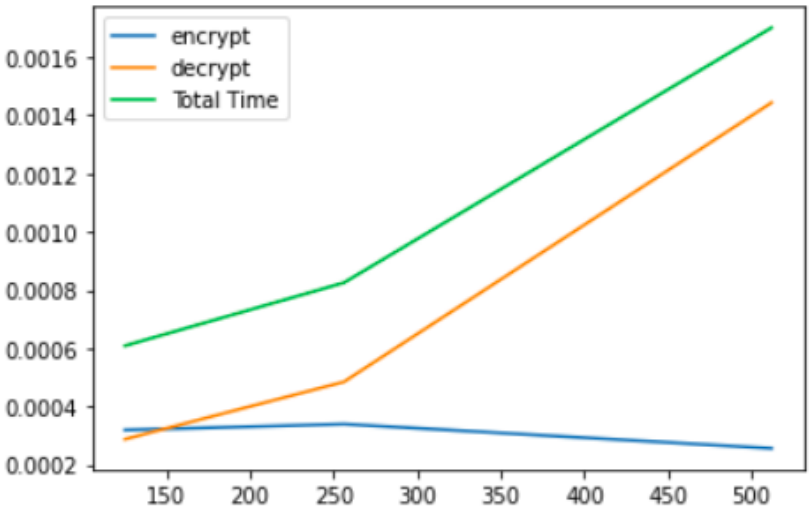


Image Compression: Applying IFFT/FFT to the real world

The use of digital images has grown exponentially in the last 40 years all thanks to the storage transmission modification of these images. The fast fourier transform can be used to compress an image to almost a 10th of its original size without losing any of its clarity.So far we have managed to implement FFT and IFFT for both 1D and 2D arrays . We are no completely equipped to perform this compression, it is only a matter of these simple steps:

1. Convert a grayscale image of your choosing into a matrix
2. Apply 2D FFT to the matrix
3. Acquire the value from the matrix that corresponds with the top c% [c is the compression rate] [this tells you all the signals from the image that carry 100-c % of the information]
4. Remove all values from the matrix that are lower than this threshold value by replacing it with 0
5. Apply 2D IFFT to the matrix that is remaining.
6. Read the result using imshow()

The below given code implements the same

```
def compress(img, cmpr):
 # print(img.shape)
  b=fft2_d(img)
  b_sort=np.sort(np.abs(b.ravel()))
  thresh=b_sort[round(abs(np.floor(((1-cmpr)*len(b_sort)))))]
  print(thresh)
  print(max(b.ravel()))
  b=b.ravel()
  for i in range(len(b)):
    if abs(b[i])<thresh:
      b[i]=0
  #np.testing.assert_array_equal(abs(np.fft.ifft2(b.reshape(img.shape))),img)
  plt.imshow(abs(ifft2d(b.reshape(img.shape))), "gray")
```

Let's take an image and try.

```
[82]  1 image=imread("/content/gray.jpeg")
      2 plt.imshow(image, "gray")
```

```
<matplotlib.image.AxesImage at 0x7fe3aa2b0310>
```

This on being compress to the compression rate 10%:

```
[114]  1 compress(image, .1)
```

```
18278.807361876577
(274395457+0j)
```

Or to 1 %:

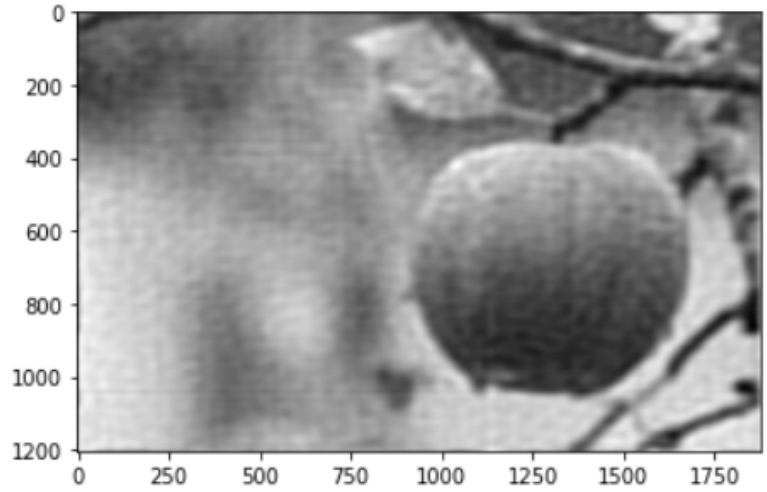
```
1 compress(image,.01)
```

```
75518.44969301533
(274395457+0j)
```

Or to .1%:

```
[103]  1 compress(image, .001)
```

```
356596.45392414974
(274395457+0j)
```



It is quite clear that even at a compression rate of 1%, the image quality isn't hampered.

Observations and Learning Outcomes

While writing code for these algorithms did inform us of how these algorithms were designed to accomplish their tasks, what sets the assignment apart from just studying course work from slides is getting to see these algorithms in action.

Implementing these algorithms gave us a chance to truly appreciate the beauty of these algorithms. They manage to leverage simple to understand concepts to accomplish things of great relevance. In this era of increased web traffic, digital image compression and secure information transmission are imperative and key to a good online experience.

It is fascinating to see how these algorithms manage to accomplish this in an efficient manner. What is more noteworthy, is that these algorithms were created more than 50 years back and still continue to be quite relevant and popular. It goes to show what sound understanding of simple mathematical concepts and good engineering skills can accomplish.