Deniz Aytemiz

ECE 5560: LAB 3

Task 1.

Code:

```
[3] #given p,q and e there is a unique d that can be found with extended eucluidian algorithm
    p = 877120207828103588060123669605304803636762908805750390255929453581934082498 97
    q = 102835471351264451708400576484301274347085188629221996951152314010256656047547
    e = 65537

[4] ##extended euclidian function gives modular inverse.
    def egcd(a, b):
        if a == 0:
            return (b, 0, 1)
        g, y, x = egcd(b%a,a)
        return (g, x - (b//a) * y, y)

    def modinv(a, m):
        g, x, y = egcd(a, m)
        if g != 1:
            raise Exception('No modular inverse')
        return x%m

[5] tic = time.perf_counter()
    d = modinv(e, (p-1)*(q-1))
    toc = time.perf_counter()

[9] print(d)
    print(d.bit_length())
    print(toc-tic)

    2777113439026967813870345260359839597823226328146295814538363064080293099090869828193345932139225213849393071682039628960771675474665521453093256445593377
    510
    6.079399997815926e-05
```

Figure 1: Calculating private key with size of 510 bits

a. There is a unique private key for a specific public key, p and q values. As the math suggest in order to find the associated private key, we use the formula;
   $(e \times d)\, mod\, (p-1) \times (q-1)$ which indicates;
   $d = e^{-1} mod\, (p-1) \times (q-1)$
   With the code I found online that calculates the modular inverse; d can be found as shown in Figure 1.

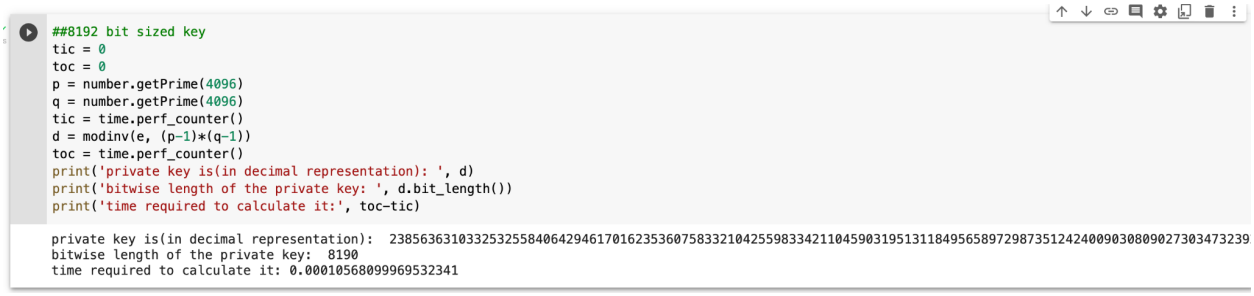
b. To get different sizes of keys I changed the sizes of p and q values respectively with the help of number.getPrime(n) method. The results are shown in Figures 2,3 and 4.

```
[29] ##256 bit sized key
     tic = 0
     toc = 0
     p = number.getPrime(128)
     q = number.getPrime(128)
     tic = time.perf_counter()
     d = modinv(e, (p-1)*(q-1))
     toc = time.perf_counter()
     print('private key is(in decimal representation): ', d)
     print('bitwise length of the private key: ', d.bit_length())
     print('time required to calculate it:', toc-tic)

     private key is(in decimal representation):  348994042706091416850592901860138431983147800946216352349002684355907083065 53
     bitwise length of the private key:  255
     time required to calculate it: 6.696599939459702e-05

     ##1024 bit sized key
     tic = 0
     toc = 0
     p = number.getPrime(512)
     q = number.getPrime(512)
     tic = time.perf_counter()
     d = modinv(e, (p-1)*(q-1))
     toc = time.perf_counter()
     print('private key is(in decimal representation): ', d)
     print('bitwise length of the private key: ', d.bit_length())
     print('time required to calculate it:', toc-tic)

     private key is(in decimal representation):  254037693547721691140498709228382457503175645372330721121535613862579857618706753881436267367504143449560456359 8
     bitwise length of the private key:  1022
     time required to calculate it: 9.825400047702715e-05
```

Figure 2: Private key and duration to calculate it for sizes 256 and 1024 bits.

```
[24] ##2048 bit sized key
     tic = 0
     toc = 0
     p = number.getPrime(1024)
     q = number.getPrime(1024)
     tic = time.perf_counter()
     d = modinv(e, (p-1)*(q-1))
     toc = time.perf_counter()
     print('private key is(in decimal representation): ', d)
     print('bitwise length of the private key: ', d.bit_length())
     print('time required to calculate it:', toc-tic)

     private key is(in decimal representation):  139305342655580830877036720398015110756056182769274717917368263581234596626628982423627985617829465159741462209 53
     bitwise length of the private key:  2047
     time required to calculate it: 8.674899981997442e-05

[25] ##4096 bit sized key
     tic = 0
     toc = 0
     p = number.getPrime(2048)
     q = number.getPrime(2048)
     tic = time.perf_counter()
     d = modinv(e, (p-1)*(q-1))
     toc = time.perf_counter()
     print('private key is(in decimal representation): ', d)
     print('bitwise length of the private key: ', d.bit_length())
     print('time required to calculate it:', toc-tic)

     private key is(in decimal representation):  500874790709747152070639319475732765794385619538589739908756653829433842695338044034085695267605598508430995719 30
     bitwise length of the private key:  4095
     time required to calculate it: 8.610299983047298e-05
```

Figure 3: Private key and duration to calculate it for sizes 2048 and 4096 bits.



```
##8192 bit sized key
tic = 0
toc = 0
p = number.getPrime(4096)
q = number.getPrime(4096)
tic = time.perf_counter()
d = modinv(e, (p-1)*(q-1))
toc = time.perf_counter()
print('private key is(in decimal representation): ', d)
print('bitwise length of the private key: ', d.bit_length())
print('time required to calculate it:', toc-tic)

private key is(in decimal representation):  238563631033253255840642946170162353607583321042559833421104590319513118495658972987351242400903080902730347323 9
bitwise length of the private key:  8190
time required to calculate it: 0.00010568099969532341
```

Figure 4: Private key and duration to calculate it for size 8192 bits.

To summarize; 256 bits take 6.69×10^{-5} seconds.

510 bits take 6.079×10^{-5} seconds.

1022 bits take 9.825×10^{-5} seconds.

2047 bits take 8.281×10^{-5} seconds.

4095 bits take 8.610×10^{-5} seconds.

8190 bits take 10.568×10^{-5} seconds.

As the key size increases the calculation process takes longer time but the differences are very slight. However, as p and q increases in size number.getPrime() method takes much longer time to complete with respect to calculation of d.

Task 2.

a. The encryption code for the first message is given in Figure 5.

```
[48]  s = 'Hello, this is my first RSA message!'.encode('utf-8')
      m = s.hex()
      ##verifying the hex version of the message
      print(m)

48656c6c6f2c207468697320697320697320686d792066697273742052534120 6d65737361676521

[49] base16INT = int(m, 16)
     hex_value = hex(base16INT)
     print(hex_value)

0x48656c6c6f2c207468697320697320686d792066697273742052534120 6d65737361676521

[50] N = 0xEF38064573FC9B1DF7BD8415B6BFB64402E5DB284FE8CAD9A85F0785BC3E3D07A3CFCFCEE6C8B64C37966982472C36604EF8B5A4AA5178CD2758D0E443126C19
     e = 0x010001
     ##converting the hex value of the message to integer so we can involve the number in calculations.
     hex_int = int(hex_value, 16)

[51] print(hex_int)
     print(type(hex_int))

1406418178021519876848582144199233836725815171252273591641142323056807403784209343050 57
<class 'int'>

     ##creating the ciphertext
     C =  pow(hex_int, e, N)
     print(hex(C))

0x722732586ffb8b6639aed189a822d92d16630fac0d54c9f7a7df9810e2f2200476fad29db5de2aaa2c82f1f706cbd674e09cbaf5e7215cb179933b359fca8997
```

Figure 5: First sentence is encrypted successfully

The string message is first converted from ASCII string to hexadecimal string and then hexadecimal number which is later converted to integer. With the message and given (e,N), the ciphertext is calculated as follows;

$C = (message)^{e} mod\ N$

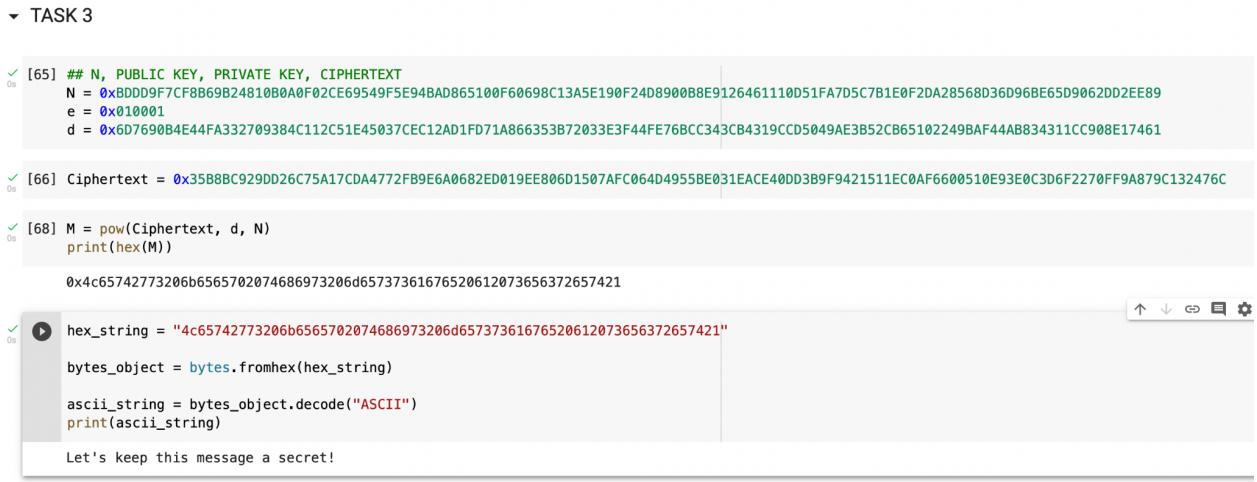
b. No error occurred while encrypting the second sentence.

```
[60] new_str = 'This is a much longer second RSA message. I am having so much fun!'.encode('utf-8')

[61] m = new_str.hex()
     base16INT = int(m, 16)
     hex_value = hex(base16INT)
     hex_int = int(hex_value, 16)
     C =  pow(hex_int, e, N)

[63] print(hex(C))

     0xd7b171885415c1eb8a45bf863be956744068ab5b6b2d9371b5dcde1f96d57319ad0bd9f67b8be5256676062c71dab968aeaa0824474fadf8749573c0aa88b59c
```

Figure 6: Second sentence is encrypted successfully

Task 3.

a. With given (d,N) and ciphertext we decrypt the message as follows; $message = (C)^{d} mod\ N$. As seen in Figure 7, once we get the hex version of the message I created an Ascii string from its hex string and hence we could decrypt the message.



```
TASK 3

[65] ## N, PUBLIC KEY, PRIVATE KEY, CIPHERTEXT
     N = 0xBDDD9F7CF8B69B24810B0A0F02CE69549F5E94BAD865100F60698C13A5E190F24D8900B8E9126461110D51FA7D5C7B1E0F2DA28568D36D96BE65D9062DD2EE89
     e = 0x010001
     d = 0x6D7690B4E44FA332709384C112C51E45037CEC12AD1FD71A866353B72033E3F44FE76BCC343CB4319CCD5049AE3B52CB65102249BAF44AB834311CC908E17461

[66] Ciphertext = 0x35B8BC929DD26C75A17CDA4772FB9E6A0682ED019EE806D1507AFC064D4955BE031EACE40DD3B9F9421511EC0AF6600510E93E0C3D6F2270FF9A879C132476C

[68] M = pow(Ciphertext, d, N)
     print(hex(M))

     0x4c65742773206b6565702074686973206d65737361676520612073656372657421

     hex_string = "4c65742773206b6565702074686973206d65737361676520612073656372657421"

     bytes_object = bytes.fromhex(hex_string)

     ascii_string = bytes_object.decode("ASCII")
     print(ascii_string)

     Let's keep this message a secret!
```

Figure 7: Decrypting the ciphertext

b. First we want to create public/private key pairs. In Figure 8, we select 1024 bits lengthed 2 prime numbers and determine N.

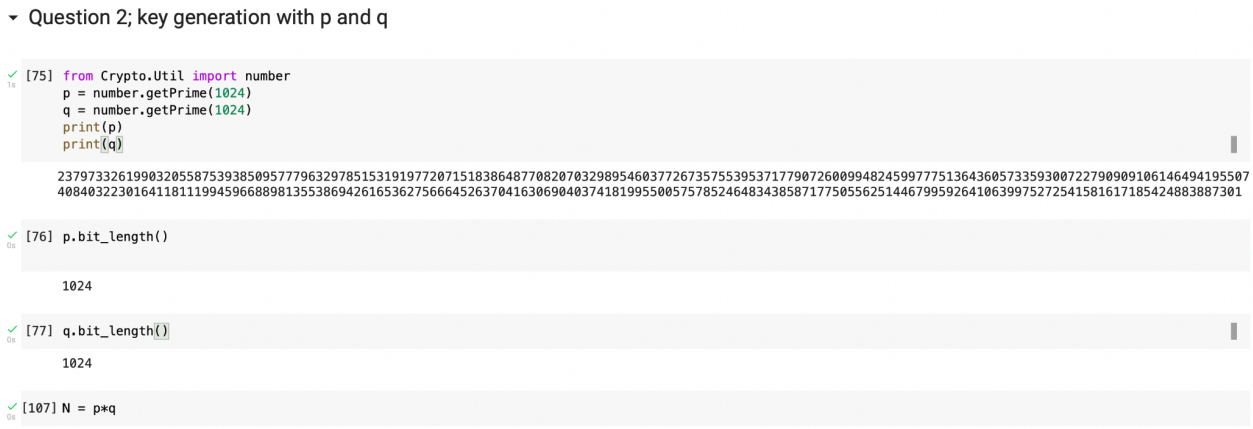


Figure 8: Determining p and q with 1024 bits

Then, we select a public key such that the public key is smaller than (p-1)x(q-1) and coprimes with (p-1)x(q-1). The code in Figure 9 selects the smallest number that satisfies these conditions hence the public key is selected small. However since p and q values are very large the private key is very large too and that is more important since private key should not be obtained but public key is already known.

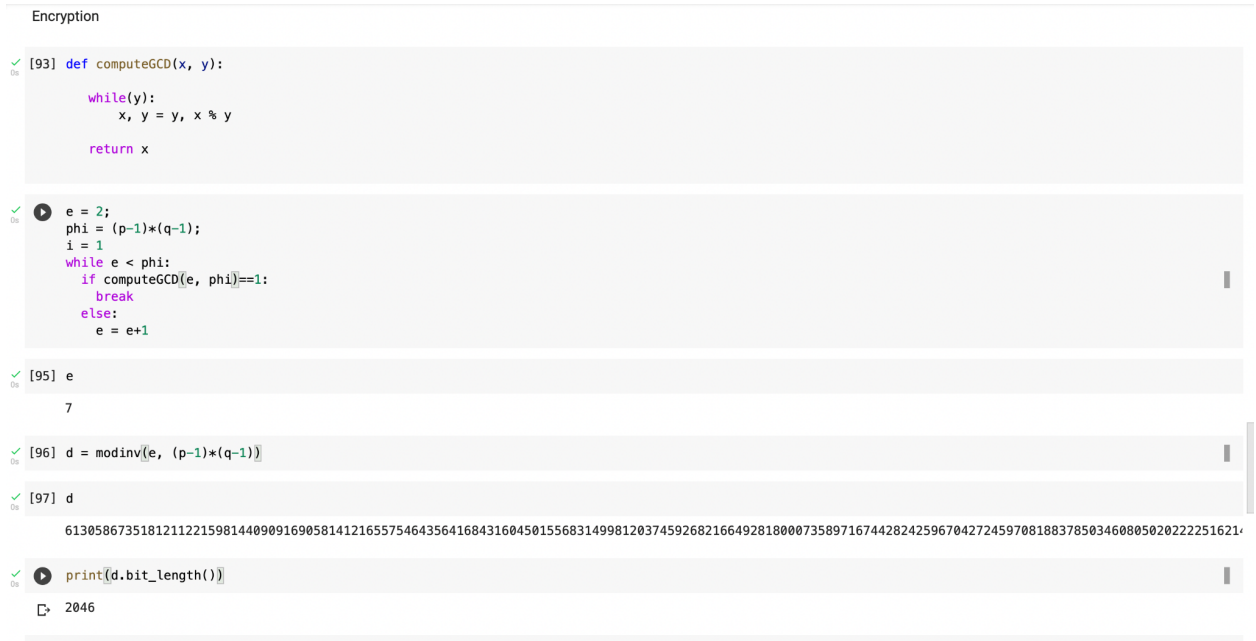


Figure 9: Finding public and private key pairs

In Figure 10, again converting the message string into integer we perform modular arithmetics and complete the encryption and decryption.

Figure 10: Encryption and Decryption

Task 4.

a. In order to digitally sign a message with RSA, the sender of the message encrypts it with his/her private key. So when it is verified(decrypted) with the public key the receiver knows the public and private key used in the scheme are pairs meaning the message came from an authenticated source. The signing of the first message is shown in Figure 11.

Figure 11: Digital signing of the first message

b. In Figure 12, one can observe the altered message is digitally signed.

We changed 20000 to 100000 in the message. No error occurred but as expected the ciphertext was changed.

Figure 12: Altering the message and digitally signing

Task 5.

a. For digitally signing with an RSA sender encrypts his/her message with his/her private key. The receiver decrypts the message he/she got with the public key both parties have. Hence when it is decrypted correctly the receiver knows that message was coming from the right source. In Figure 13, the whole process is demonstrated.

Task 5

```
[103] message =  'This is a contact for $20,000'.encode('utf-8')
      ## N, PUBLIC KEY, PRIVATE KEY
      N = 0xBDDD9F7CF8B69B24810B0A0F02CE69549F5E94BAD865100F60698C13A5E190F24D8900B8E9126461110D51FA7D5C7B1E0F2DA28568D36D96BE65D9062DD2EE89
      e = 0x010001
      d = 0x6D7690B4E44FA332709384C112C51E45037CEC12AD1FD71A866353B72033E3F44FE76BCC343CB4319CCD5049AE3B52CB65102249BAF44AB834311CC908E17461

[104] ## digitally signing the message(encrpyting it with private key)
      m = message.hex()
      base16INT = int(m, 16)
      hex_value = hex(base16INT)
      hex_int = int(hex_value, 16)
      S =  pow(hex_int, d, N)

[105] ##verifying the signing. The signed message is verified(decrpyted) with the public key.
      V = pow(S,e,N)
      print(hex(V))

      0x546869732069732061206 36f6e7461637420666f72202432302c303030

      ##converting the hex string to ascii to observe we acquire the original message
      hex_string = "546869732069732061206 36f6e7461637420666f72202432302c303030"

      bytes_object = bytes.fromhex(hex_string)

      ascii_string = bytes_object.decode("ASCII")
      print(ascii_string)

      This is a contact for $20,000
```

Figure 13: Sign and verify the message

b. The key size will be crucial in using the RSA algorithm. As we encrypt/decrypt large files, the encrypted/decrypted information's size should be smaller than the effective key size. This will be demonstrated in the following figures. In Figure 14,

we import RSA library and create public and private keys and check their length in bytes. The shorter key is the private key with size 130 bytes So the block sizes will be smaller than 130 bytes. Hence for reading data from the file we gather 128 bytes of information for 1 block.

```
[4]  ## importing RSA library for key generation
     from Crypto.PublicKey import RSA
     import time

[5]  ## public and private keys are generated
     keyPair = RSA.generate(bits=1024)
     print(f"Public key:  (n={hex(keyPair.n)}, e={hex(keyPair.e)})")
     print(f"Private key: (n={hex(keyPair.n)}, d={hex(keyPair.d)})")

     Public key:  (n=0xcce674647c4b9f74c050ffcd1853a3bb42f972d2bfe2af49610120dd9876b5c5034dbf636cf5fbbc7ea58e6ae26b35e69d56e5a6b7914acf8e9489243a13cf6d9611290d69
     Private key: (n=0xcce674647c4b9f74c050ffcd1853a3bb42f972d2bfe2af49610120dd9876b5c5034dbf636cf5fbbc7ea58e6ae26b35e69d56e5a6b7914acf8e9489243a13cf6d9611290d69

     ## the limiting key size is public key, its size is 130 bytes. Whereas private key size is 255 bytes.
     print('key sizes in bytes')
     print((keyPair.e.bit_length()+keyPair.n.bit_length())/8)
     print((keyPair.d.bit_length()+keyPair.n.bit_length())/8)

     key sizes in bytes
     130.125
     255.875

     + Code   + Text
```

Figure 14: RSA public/private key created and their sizes are observed

```
[16] ##code for generating fixed size file with random content
     def generate_big_random_bin_file(filename,size):
         """
         generate big binary file with the specified size in bytes
         :param filename: the filename
         :param size: the size in bytes
         :return:void
         """
         import os
         with open('%s'%filename, 'wb') as fout:
             fout.write(os.urandom(size)) #1

         print ('big random binary file with size %f generated ok'%size)
         pass
```

```
if __name__ == '__main__':
    generate_big_random_bin_file("temp_big_bin.dat",1024)
```

```
big random binary file with size 1024.000000 generated ok
```

```
[18] ##the file size in bytes is 1024
     txt_file =open("temp_big_bin.dat","rb")
     print(len(txt_file.read()))
```

```
1024
```

Figure 15: 1 KB file is created

In Figure 15, we create 1 KB sized file with random content. The length of the file is checked which is 1024 bytes(in binary) is 1 KB.

```
[11] byte_list = []

     with open("temp_big_bin.dat", "rb") as f:
         while True:
             byte = f.read(128)
             if not byte:
                 break
             byte_list.append(byte)

     #print(byte_list)
```

```
## byte_list stores the blocks we sign and verify its length is 1024/128 = 8
len(byte_list)
```

```
8
```

Figure 16: The file is divided into blocks of size 128 bytes

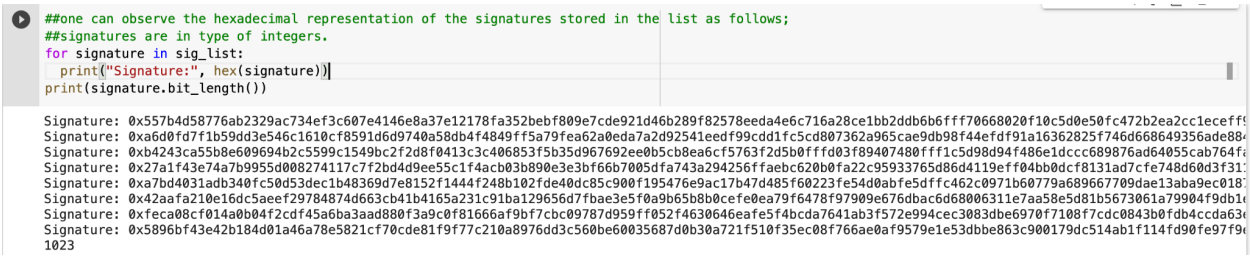
In Figure 16, we read files by 128 bytes and save those blocks into byte_list. There are 8 blocks stored in byte_list.

```
## sign_list will store the signatures performed on 1 block of information.
## variable signature stores the 1 block of information that is signed with private key.
tic = time.perf_counter()
sig_list = []
for message in byte_list:
  m = int.from_bytes(message, byteorder='big')
  print(hex(m))
  signature = pow(m, keyPair.d, keyPair.n)
  sig_list.append(signature)
toc = time.perf_counter()
print('Time passed to sign the file:(seconds)',toc-tic)
```

```
0x5467406900fe2012c7a235225a332d6eb002099ca31495a84d455867cd3ee26d1767d572e64c34a69add9e294f1882172f615975626939b6ec2830ddce7a248eee64b845fe9adbd12b0d7cc00b
0x5410572f3f7eb8f57061558de89c28e7f6752ea889d823f3906dea88dd0374d25fd9e9dc0bf56f20ee6daa06262886c8c87fb1490c3ec1582fbf519bf25717133af202f088777907064ae1c1b0
0xbe2046c39106edb22ed763ce58790b451b5256edcf9e3dd62077fed17539081aa3280f9cbfe7acbfb16432a9a12780deb31cde6ed520e65d91a7e03e1da285520b300c158e405e6d3f4efe68dd
0x936e728bbf41cd09a94d7ec3424a3e13833d4d4cddcca5ac37000ad5fe4617176bf8353254c6183a2577a44828951de0ba2f608d659a13cd63ee2df45f3aedcaf7dc95315050f374514f8f3a27
0x6ea5ddaca12e92901ad1fb232f0710d6b0808a059374a38d6ef5053e3cb18952cdf066d53c772b577cfe80d735bdd20abffc0b8dbbab5edadc956f1218e34509c972b2a2941dd7b677ae490414
0x903a78f6ed5eff18a3ce3e3e4f87d3f57258a160f6df2d4b06a70d0c5c7ab9e03c38ffb1d6e85062702c243fcd9727d24401834052a42c6aa1f91312ca9003234631634617b69ce62d2b5b3ea98
0xc56d26eb10b71377d8dcf906756b4a7ad62df1ea0c470a69b5d10e89556c21ec2cef3f982d2c299d1f1c2bac22d2ee92920c4033cce9b358a62d0ed0215e52fc7de3c62962b4568973d13ee2070
0x64cfea0b74bce2bbc0c15385d48988b4d4e29f25905a89638dd98e2a90053ddd298bbadc3b22bd16a77f7330e621791407f1284eb94a92620124b4a0e57566590721ca469a099c6c4dffef13865
Time passed to sign the file:(seconds) 0.04604021400001557
```

Figure 17: The time needed to sign all the blocks is demonstrated.

In Figure 17, we see it took 0.046 seconds to sign the 1KB file. The hex values of the blocks are printed for verification later on.



```
##one can observe the hexadecimal representation of the signatures stored in the list as follows;
##signatures are in type of integers.
for signature in sig_list:
  print("Signature:", hex(signature))
print(signature.bit_length())
```

```
Signature: 0x557b4d58776ab2329ac734ef3c607e4146e8a37e12178fa352bebf809e7cde921d46b289f82578eeda4e6c716a28ce1bb2ddb6b6fff70668020f10c5d0e50fc472b2ea2cc1eceff5
Signature: 0xa6d0fd7f1b59dd3e546c1610cf8591d6d9740a58db4f4849ff5a79fea62a0eda7a2d92541eedf99cdd1fc5cd807362a965cae9db98f44efdf91a16362825f746d668649356ade884
Signature: 0xb4243ca55b8e609694b2c5599c1549bc2f2d8f0413c3c406853f5b35d967692ee0b5cb8ea6cf5763f2d5b0fffd03f89407480fff1c5d98d94f486e1dccc689876ad64055cab764fa
Signature: 0x27a1f43e74a7b9955d008274117c7f2bd4d9ee55c1f4acb03b890e3e3bf66b7005dfa743a294256ffaebc620b0fa22c95933765d86d4119eff04bb0dcf8131ad7cfe748d60d3f31
Signature: 0xa7bd4031adb340fc50d53dec1b48369d7e8152f1444f248b102fde40dc85c900f195476e9ac17b47d485f60223fe54d0abfe5dffc462c0971b60779a689667709dae13aba9ec018
Signature: 0x42aafa210e16dc5aeef29784874d663cb41b4165a231c91ba129656d7fbae3e5f0a9b65b8b0cefe0ea79f6478f97909e676dbac6d68006311e7aa58e5d81b5673061a79904f9db1e
Signature: 0xfeca08cf014a0b04f2cdf45a6ba3aad880f3a9c0f81666af9bf7cbc09787d959ff052f4630646eafe5f4bcda7641ab3f572e994cec3083dbe6970f7108f7cdc0843b0fdb4ccda63e
Signature: 0x5896bf43e42b184d01a46a78e5821cf70cde81f9f77c210a8976dd3c560be60035687d0b30a721f510f35ec08f766ae0af9579e1e53dbbe863c900179dc514ab1f114fd90fe97f9e
1023
```

Figure 18: Signed blocks in hex form

The signed blocks are stored in the list sig_list.

```
## verification of the signatured blocks will be done individually on the blocks.
## each verified message is converted to its ASCII string and printed in order to verify
## we get the same message as before. In addition to message there is also some decoding on
## font and text scaling and font type etc.
ver_list = []
for signature in sig_list:
  verification = pow(signature,keyPair.e,keyPair.n)
  ver_list.append(verification)
  print(hex(verification))
```

```
0x5467406900fe2012c7a235225a332d6eb002099ca31495a84d455867cd3ee26d1767d572e64c34a69add9e294f1882172f615975626939b6ec2830ddce7a248eee64b845fe9adbd12b0d7cc00b
0x5410572f3f7eb8f57061558de89c28e7f6752ea889d823f3906dea88dd0374d25fd9e9dc0bf56f20ee6daa06262886c8c87fb1490c3ec1582fbf519bf25717133af202f088777907064ae1c1b0
0xbe2046c39106edb22ed763ce58790b451b5256edcf9e3dd62077fed17539081aa3280f9cbfe7acbfb16432a9a12780deb31cde6ed520e65d91a7e03e1da285520b300c158e405e6d3f4efe68dd
0x936e728bbf41cd09a94d7ec3424a3e13833d4d4cddcca5ac37000ad5fe4617176bf8353254c6183a2577a44828951de0ba2f608d659a13cd63ee2df45f3aedcaf7dc95315050f374514f8f3a27
0x6ea5ddaca12e92901ad1fb232f0710d6b0808a059374a38d6ef5053e3cb18952cdf066d53c772b577cfe80d735bdd20abffc0b8dbbab5edadc956f1218e34509c972b2a2941dd7b677ae490414
0x903a78f6ed5eff18a3ce3e4f87d3f57258a160f6df2d4b06a70d0c5c7ab9e03c38ffb1d6e85062702c243fcd9727d24401834052a42c6aa1f91312ca9003234631634617b69ce62d2b5b3ea9
0xc56d26eb10b71377d8dcf906756b4a7ad62df1ea0c470a69b5d10e89556c21ec2cef3f982d2c299d1f1c2bac22d2ee92920c4033cce9b358a62d0ed0215e52fc7de3c62962b4568973d13ee207
0x64cfea0b74bce2bbc0c15385d48988b4d4e29f25905a89638dd98e2a90053ddd298bbadc3b22bd16a77f7330e621791407f1284eb94a92620124b4a0e57566590721ca469a099c6c4dffef1386
```

Figure 19: The verified blocks

In Figure 19, we can see the verified blocks are the same with the hex representations of the message blocks which means the we successfully signed and verified the whole file. In the following figures we change the size of the file and find the time it takes to sign the whole file. The block sizes will remain 128 bytes.

```
[16] ##code for generating fixed size file with random content
     def generate_big_random_bin_file(filename,size):
         """
         generate big binary file with the specified size in bytes
         :param filename: the filename
         :param size: the size in bytes
         :return:void
         """
         import os
         with open('%s'%filename, 'wb') as fout:
             fout.write(os.urandom(size)) #1

         print ('big random binary file with size %f generated ok'%size)
         pass
```

```
[19] if __name__ == '__main__':
         generate_big_random_bin_file("temp_big_bin.dat",1024*100)
```

```
big random binary file with size 102400.000000 generated ok
```

```
##the file size in bytes is 1024
txt_file =open("temp_big_bin.dat","rb")
print(len(txt_file.read()))
```

```
102400
```

Figure 20: 100 KB file created

The file is divided into 800 blocks this time. The time it takes to sign the whole file is 4.4327 seconds as shown in Figure 21.

```
## sign_list will store the signatures performed on 1 block of information.
## variable signature stores the 1 block of information that is signed with private key.
tic = time.perf_counter()
sig_list = []
for message in byte_list:
  m = int.from_bytes(message, byteorder='big')
  #print(hex(m))
  signature = pow(m, keyPair.d, keyPair.n)
  sig_list.append(signature)
toc = time.perf_counter()
print('Time passed to sign the file:(seconds)',toc-tic)
```

```
Time passed to sign the file:(seconds) 4.432702094999968
```

Figure 21: 100 KB file is signed

```
[25] if __name__ == '__main__':
         generate_big_random_bin_file("temp_big_bin.dat", 1048576)
```

```
big random binary file with size 1048576.000000 generated ok
```

```
[26] ##the file size in bytes is 1024
     txt_file =open("temp_big_bin.dat","rb")
     print(len(txt_file.read()))
```

```
1048576
```

```
[27] byte_list = []

     with open("temp_big_bin.dat", "rb") as f:
         while True:
             byte = f.read(128)
             if not byte:
                 break
             byte_list.append(byte)

     #print(byte_list)
```

```
## byte_list stores the blocks we sign and verify its length is 1024/128 = 8
len(byte_list)
```

```
8192
```

Figure 22: 1 MB file is created

The time it takes to sign 1 MB file is shown in Figure 23. It is 45.254 seconds

```
## sign_list will store the signatures performed on 1 block of information.
## variable signature stores the 1 block of information that is signed with private key.
tic = time.perf_counter()
sig_list = []
for message in byte_list:
  m = int.from_bytes(message, byteorder='big')
  #print(hex(m))
  signature = pow(m, keyPair.d, keyPair.n)
  sig_list.append(signature)
toc = time.perf_counter()
print('Time passed to sign the file:(seconds)',toc-tic)
```

```
Time passed to sign the file:(seconds) 45.25468423700022
```

Figure 23: 1 MB file is signed

```
[30] if __name__ == '__main__':
         generate_big_random_bin_file("temp_big_bin.dat", 1048576*10)

     big random binary file with size 10485760.000000 generated ok
```

```
     ##the file size in bytes is 1024
     txt_file =open("temp_big_bin.dat","rb")
     print(len(txt_file.read()))

     10485760
```

```
[32] byte_list = []

     with open("temp_big_bin.dat", "rb") as f:
         while True:
             byte = f.read(128)
             if not byte:
                 break
             byte_list.append(byte)

     #print(byte_list)
```

```
     ## byte_list stores the blocks we sign and verify its length is 1024/128 = 8
     len(byte_list)

     81920
```

Figure 24: 10 MB file is created

The time it takes to sign the whole file is shown in Figure 25. It is 438.2367 seconds.

```
     ## sign_list will store the signatures performed on 1 block of information.
     ## variable signature stores the 1 block of information that is signed with private key.
     tic = time.perf_counter()
     sig_list = []
     for message in byte_list:
       m = int.from_bytes(message, byteorder='big')
       #print(hex(m))
       signature = pow(m, keyPair.d, keyPair.n)
       sig_list.append(signature)
     toc = time.perf_counter()
     print('Time passed to sign the file:(seconds)',toc-tic)

     Time passed to sign the file:(seconds) 438.236710745
```

Figure 25: The signing of 10 MB file

Now, we will demonstrate that if block size is 256 bytes for instance which exceeds the key size(130 bytes) the RSA will not work properly. We will show this on a 1 KB file.

```
if __name__ == '__main__':
    generate_big_random_bin_file("temp_big_bin.dat", 1024)
```

```
big random binary file with size 1024.000000 generated ok
```

```
[36] ##the file size in bytes is 1024
     txt_file =open("temp_big_bin.dat","rb")
     print(len(txt_file.read()))
```

```
1024
```

```
[37] byte_list = []

     with open("temp_big_bin.dat", "rb") as f:
         while True:
             byte = f.read(128*2)
             if not byte:
                 break
             byte_list.append(byte)

     #print(byte_list)
```

```
## byte_list stores the blocks we sign and verify its length is
len(byte_list)
```

```
4
```

Figure 26: The block size is changed to 256 bytes and 1KB file is created again

In Figure 27, one can observe the hex versions of the blocks and in Figure 28 one can observe hex representations of the verified blocks which are not the same. Hence we can say RSA did not work properly.

```
## sign_list will store the signatures performed on 1 block of information.
## variable signature stores the 1 block of information that is signed with private key.
tic = time.perf_counter()
sig_list = []
for message in byte_list:
  m = int.from_bytes(message, byteorder='big')
  print(hex(m))
  signature = pow(m, keyPair.d, keyPair.n)
  sig_list.append(signature)
toc = time.perf_counter()
print('Time passed to sign the file:(seconds)',toc-tic)
```

```
0xac55d9f63d58095c19c35a3dcbce38bd8bbbc84954e44e66cfa929cebb5aed2f7b5476416181221f88c66268aa9856fffb4b3d0323204ff80ae35766fddf0bcf3e76f470f8809e72bae3b6f7
0x3ee79253f0fe1c3cb0cd88113cf3c1a2d4a227d0a99bbad79a356ab9b3c71c365c03f9d195fe7044fad2082ad1d322648ed72acac38332b1ab77d9ec91bf60b9523194b71d8d3129cd7eb2e8
0xea56f96d82adf23a90c8a92e27fc7eebc40ed3068bc442c457e312192908a97773cbd8dbe654759e812421d29b411cc17c4ec264c2c82c2a38d30d7650035c842be83ac0b1a0e7e0152e9410
0xd07a6e080dadc229882a8d1e84671d6649e040a7f362be186b28c4325f39d3af2dd6f1c824f70d7a5a8379c69c16080d79ebb45537108f57440347ebee3cda88f05d357f42916adb21750c71
Time passed to sign the file:(seconds) 0.028457886999603943
```

Figure 27: Hex of the file blocks

```
## verification of the signatured blocks will be done individually on the blocks.
## each verified message is converted to its ASCII string and printed in order to verify
## we get the same message as before. In addition to message there is also some decoding on
## font and text scaling and font type etc.
ver_list = []
for signature in sig_list:
  verification = pow(signature,keyPair.e,keyPair.n)
  ver_list.append(verification)
  print(hex(verification))
```

```
0xc35cf184fd3a0186507474a3d75794134c2dbc9b697c587bcd3f4465d28c7d7363ad7779b8a86a6ad9072feabe47d4bd6986528d56e390a8ad600542519d8e2da35e843bcdd45bb2e18bb57369
0x7b0829dfb36d9f5bcaa3bd8a07c42de796241b6f0c92c70c58a104161cf2e8a30fb223d52f78ac397c0365011d604fa2d314f6c1c268dce07806b455b203fe81e9dc7ccbd84d87fb150086929d
0x61fb78c8849d2675a401953ba1a7333f4de37e6cff27931b910aae22d756e2263f6c3f0fd2be4a8827610f94281937d10e35446ad0731b12e62d89d95f934022ede575a0bb1611c62f27018c76
0x1caac0ddb932612630241ac390b0613cdcb0b70fd1fda4139b3fb05a3685fc0bbd54359f7b857f8ff11d4d87a95c4ca85a1f3a36ca271bef94995016454cdbc7770e4123ddca5c3b30f3458af4
```

Figure 28: Hex of the verified blocks

One can conclude that as the file size increases the time it takes to sign/verify will increase with acceleration. The key size in such applications is crucial since it determines the block size which in turn determines the time it takes to do the whole process.

Extra Credit: