Document version: 1.1 (2015-11-15)

Curtin University – Department of Computing

Assignment Cover Sheet / Declaration of Originality

Complete this form if/as directed by your unit coordinator, lecturer or the assignment specification.

Last name:	Mehdi	Student ID:	20337270
Other name(s):	Hussain		
Unit name:	Fundamental Concepts of Cryptography	Unit ID:	ISEC2000
Lecturer / unit coordinator:	Mr. Salih	Tutor:	
Date of submission:	30/07/2022	Which assignment?	2 (Leave blank if the unit has only one assignment.)

I declare that:

- The above information is complete and accurate.
- The work I am submitting is *entirely my own*, except where clearly indicated otherwise and correctly referenced.
- I have taken (and will continue to take) all reasonable steps to ensure my work is *not accessible* to any other students who may gain unfair advantage from it.
- I have *not previously submitted* this work for any other unit, whether at Curtin University or elsewhere, or for prior attempts at this unit, except where clearly indicated otherwise.

I understand that:

- Plagiarism and collusion are dishonest, and unfair to all other students.
- Detection of plagiarism and collusion may be done manually or by using tools (such as Turnitin).
- If I plagiarise or collude, I risk failing the unit with a grade of ANN ("Result Annulled due to Academic Misconduct"), which will remain permanently on my academic record. I also risk termination from my course and other penalties.
- Even with correct referencing, my submission will only be marked according to what I have done myself, specifically for this assessment. I cannot re-use the work of others, or my own previously submitted work, in order to fulfil the assessment requirements.
- It is my responsibility to ensure that my submission is complete, correct and not corrupted.

Signature: Hussain Mehdi　　Date of signature: 30/07/2022

(By submitting this form, you indicate that you agree with all the above text.)

ASSIGNMENT 2 REPORT

DATE: 30/07/2022

COURSE TITLE:
Fundamental Concepts of Cryptography (ISEC2000)

STUDENT'S NAME: Hussain Mehdi

STUDENT ID: 20337270

Table of Contents

Introduction

This report is for Assignment 2 and contains answers to the questions asked along with brief on running the RSA code.

Question 1:

The Euclidean algorithm is based on the following assertion. Given two integers a, b, (a > b),

gcd (a, b) = gcd (b, a mod b).

Prove the assertion (1) mathematically. (Note that proof by example is NOT appropriate here)

To mathematically prove gcd (a,b) = gcd(b,amodb):

a mod b is the remainder when a is divided by b. ∴ For some integer c, the statement is a-bc and it is between 0 and b – 1.

Lets let f = gcd(a,b)

f divides both a and b therefore, it divides both a -bc and b which in turn divides gcd(b, a mod b).

Let g = gcd(b, a mod b), where g divides both b and a-bc. Which means that it divides both a and b and so it divides gcd(a,b) respectively.

(math.stackexchange.com, 2012)

Alternative Proof:

As per the division algorithm, there exists some x and r < b such that a = bx + r :

r = a mod b and r = a-bx

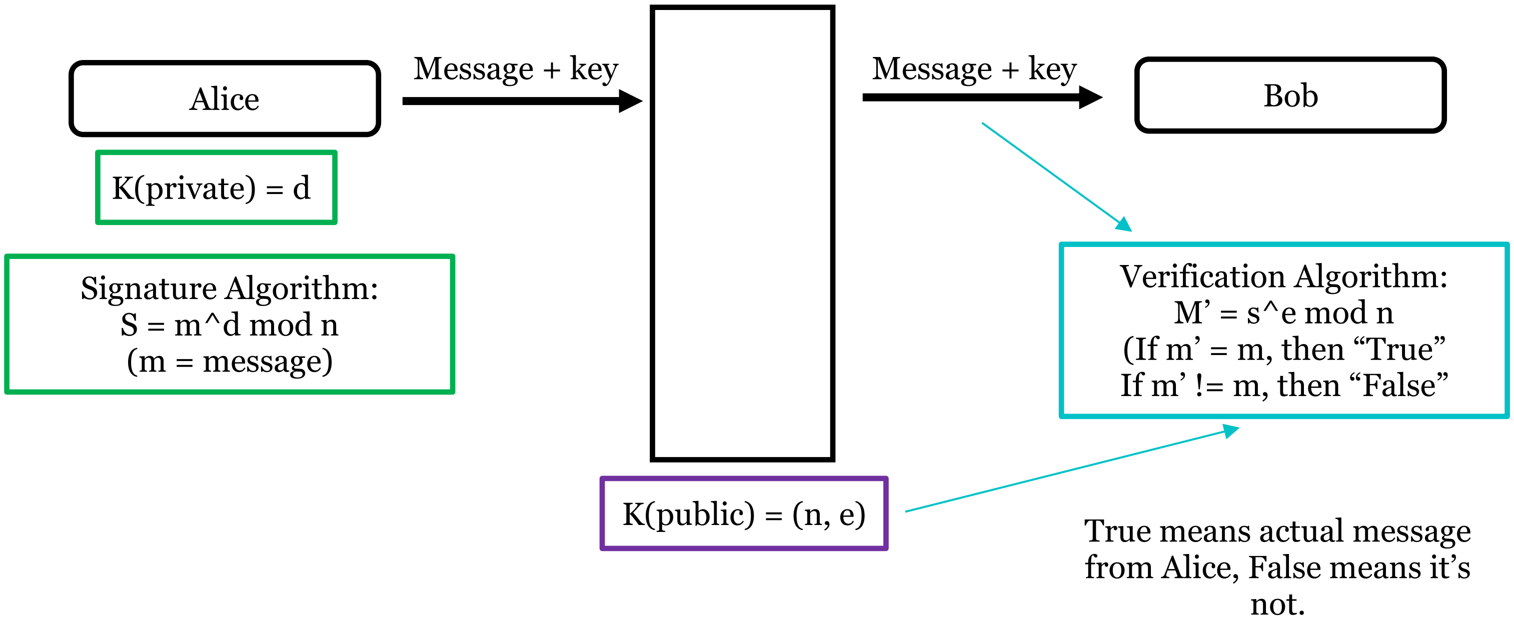
Any divisor (common) of a and b will also be a common divisor of a mod b. This is particularly the case with gcd(a,b) is a divisor of r and of b, which implies that gcd(a,b) | gcd(b,r).

Therefore, gcd(a,b) = gcd(b,r) which in-turn means that gcd(a,b) = gcd(b,a mod b)

(quora.com, 2016)

Question 2:

Assuming that Alice signed a document m using the RSA signature scheme. (You should describe the RSA signature structure first with a diagram and explain the authentication principle). The signature is sent to Bob. Accidentally Bob found one message m0 (m 6= m') such that H(m) = H(m'), where H() is the hash function used in the signature scheme. Describe clearly how Bob can forge a signature of Alice with such m'.



Sender (Alice) has a private key which is denoted by 'd' and a public key which has (n, e). Private key is only known by Alice, but public key is known by everyone. Alice takes the message (m) and puts it in the signature algorithm which takes input of the message and private key (d). The signature does modular exponentiation by taking message to the power of d multiplied by mod n. Alice is the only one to theoretically produce her message algorithm as she is the only one who knows m and d.
The result of signature algorithm is the signature. The Message and the signature are sent to Bob through a channel (ideally with encryption).
Once Bob receives the message, he checks it using the verification algorithm which takes in the message, signature to the power of e and mod n (known by the public). If m' is equal to the message, then True is returned. If not, False is returned and this means that the sender is not authentic, and the message might have been changed.

Hash collisions although not very common can and do occur. A hash for one message can result in the same hash for a totally different message. In this case, Bob was able to find a message that was signed by Alice that had the same hash as another message. The hash function applied is no secret. Bob can attempt an Existential forgery using a chose message attack. Bob sends m to Alice and obtains SigK(H(m)). Afterwhich, (m',SigK(h(m)) is a valid signed message.

Reference Material: (UoA, 2020)

Question 3:

Below are the RSA Code Implementation Screenshots along with description.

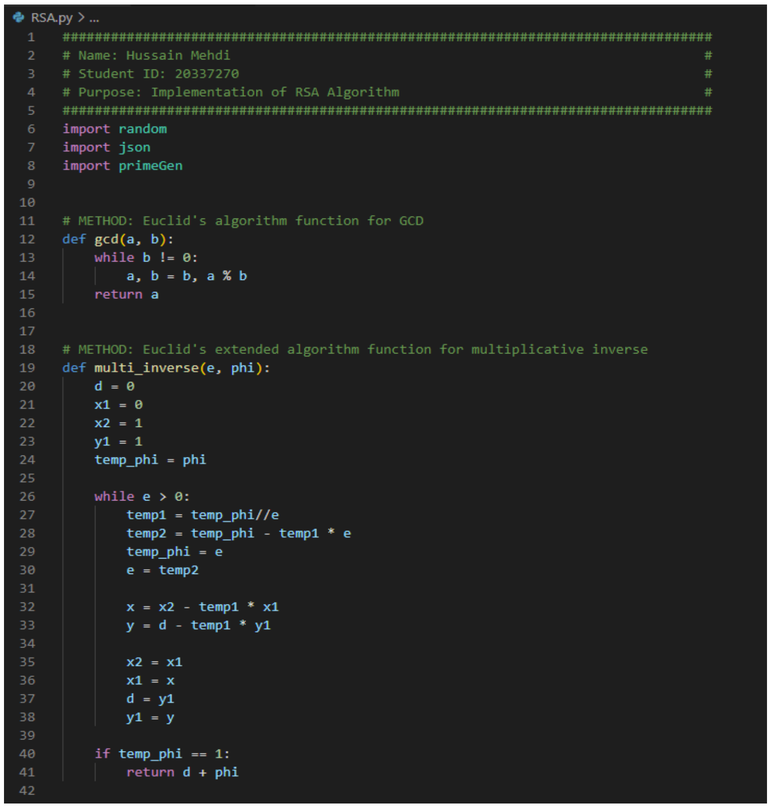
Files:

- RSA.py
- primeGen.py

Running Code:

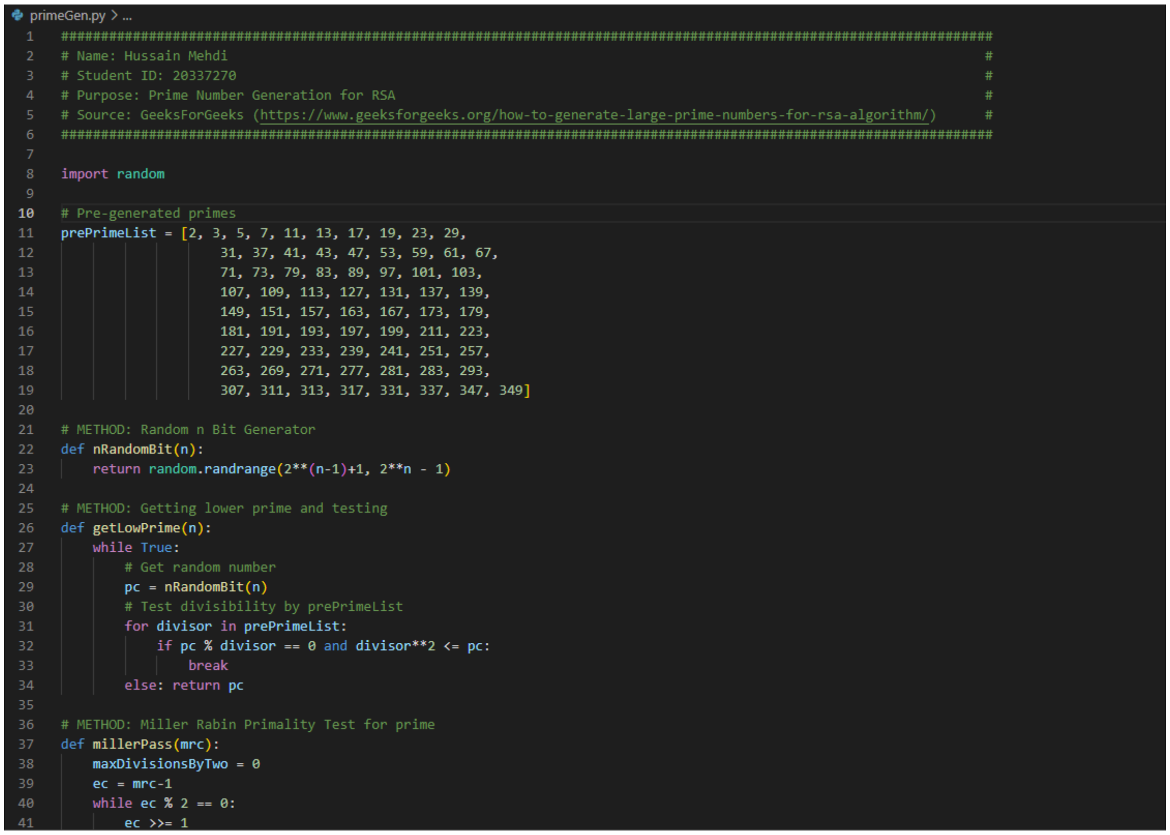
- python3 RSA.py

RSA.py File code snippets:



```
###############################################################################
# Name: Hussain Mehdi                                                         #
# Student ID: 20337270                                                        #
# Purpose: Implementation of RSA Algorithm                                    #
###############################################################################
import random
import json
import primeGen


# METHOD: Euclid's algorithm function for GCD
def gcd(a, b):
    while b != 0:
        a, b = b, a % b
    return a


# METHOD: Euclid's extended algorithm function for multiplicative inverse
def multi_inverse(e, phi):
    d = 0
    x1 = 0
    x2 = 1
    y1 = 1
    temp_phi = phi

    while e > 0:
        temp1 = temp_phi//e
        temp2 = temp_phi - temp1 * e
        temp_phi = e
        e = temp2

        x = x2 - temp1 * x1
        y = d - temp1 * y1

        x2 = x1
        x1 = x
        d = y1
        y1 = y

    if temp_phi == 1:
        return d + phi

```

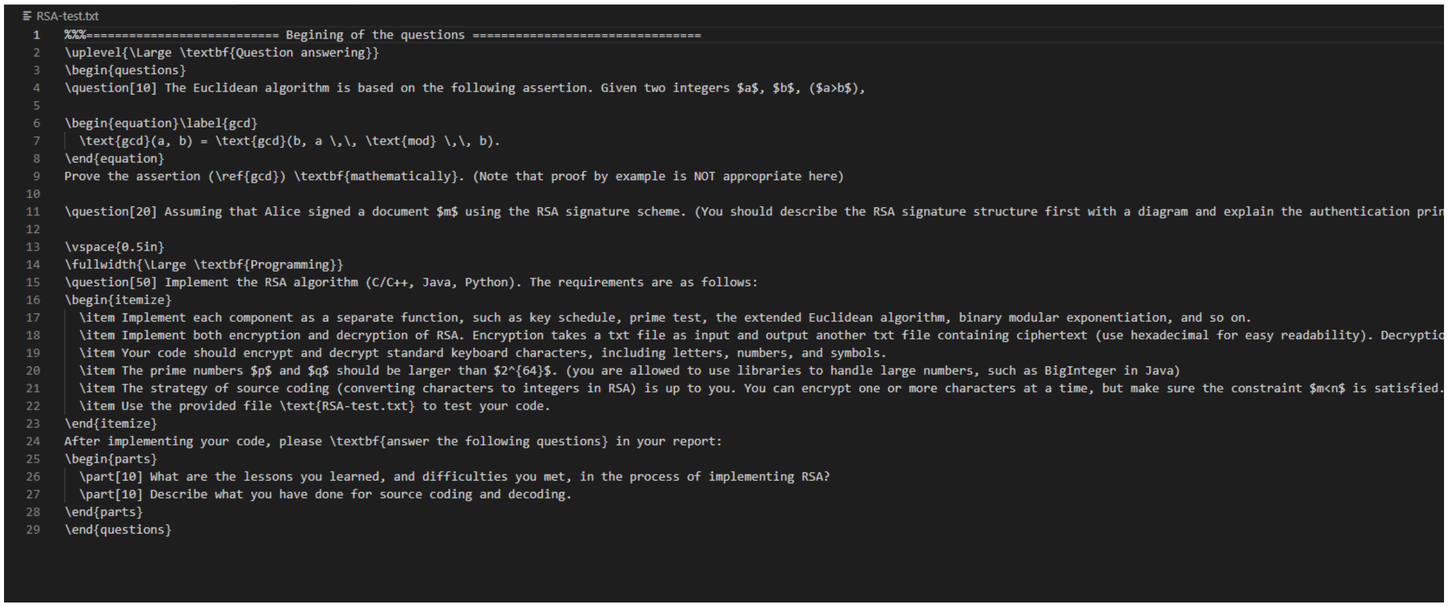
primeGen.py File code snippet:



primeGen.py > ...

```python
#####################################################################################################################
# Name: Hussain Mehdi                                                                                               #
# Student ID: 20337270                                                                                              #
# Purpose: Prime Number Generation for RSA                                                                          #
# Source: GeeksForGeeks (https://www.geeksforgeeks.org/how-to-generate-large-prime-numbers-for-rsa-algorithm/)       #
#####################################################################################################################

import random

# Pre-generated primes
prePrimeList = [2, 3, 5, 7, 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, 113, 127, 131, 137, 139,
                    149, 151, 157, 163, 167, 173, 179,
                    181, 191, 193, 197, 199, 211, 223,
                    227, 229, 233, 239, 241, 251, 257,
                    263, 269, 271, 277, 281, 283, 293,
                    307, 311, 313, 317, 331, 337, 347, 349]

# METHOD: Random n Bit Generator
def nRandomBit(n):
    return random.randrange(2**(n-1)+1, 2**n - 1)

# METHOD: Getting lower prime and testing
def getLowPrime(n):
    while True:
        # Get random number
        pc = nRandomBit(n)
        # Test divisibility by prePrimeList
        for divisor in prePrimeList:
            if pc % divisor == 0 and divisor**2 <= pc:
                break
        else: return pc

# METHOD: Miller Rabin Primality Test for prime
def millerPass(mrc):
    maxDivisionsByTwo = 0
    ec = mrc-1
    while ec % 2 == 0:
        ec >>= 1
```

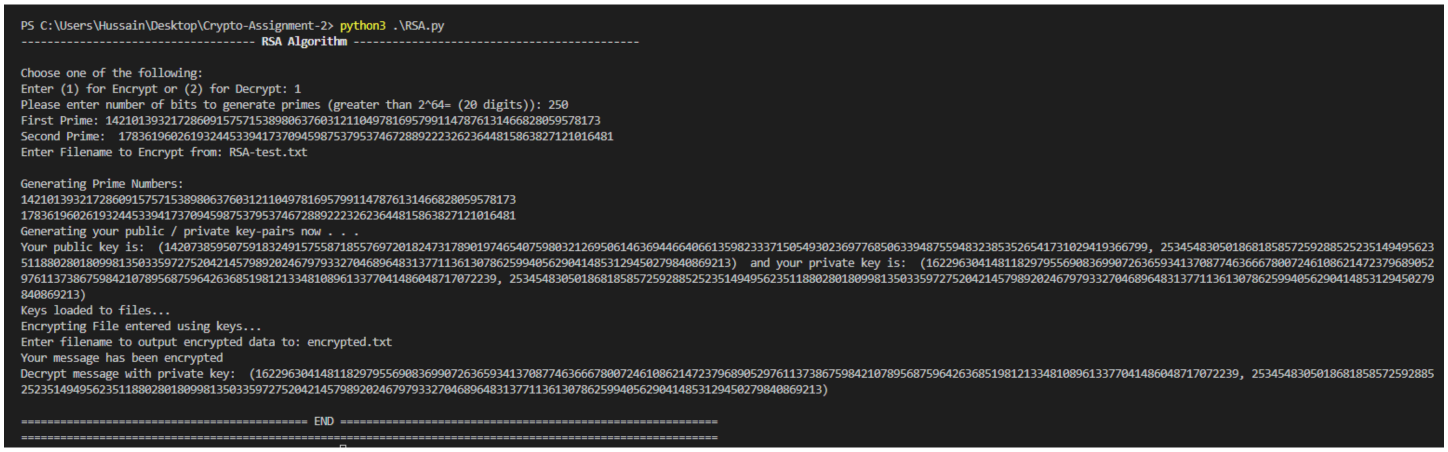
RSA-test.txt Test file snippet:



RSA-test.txt

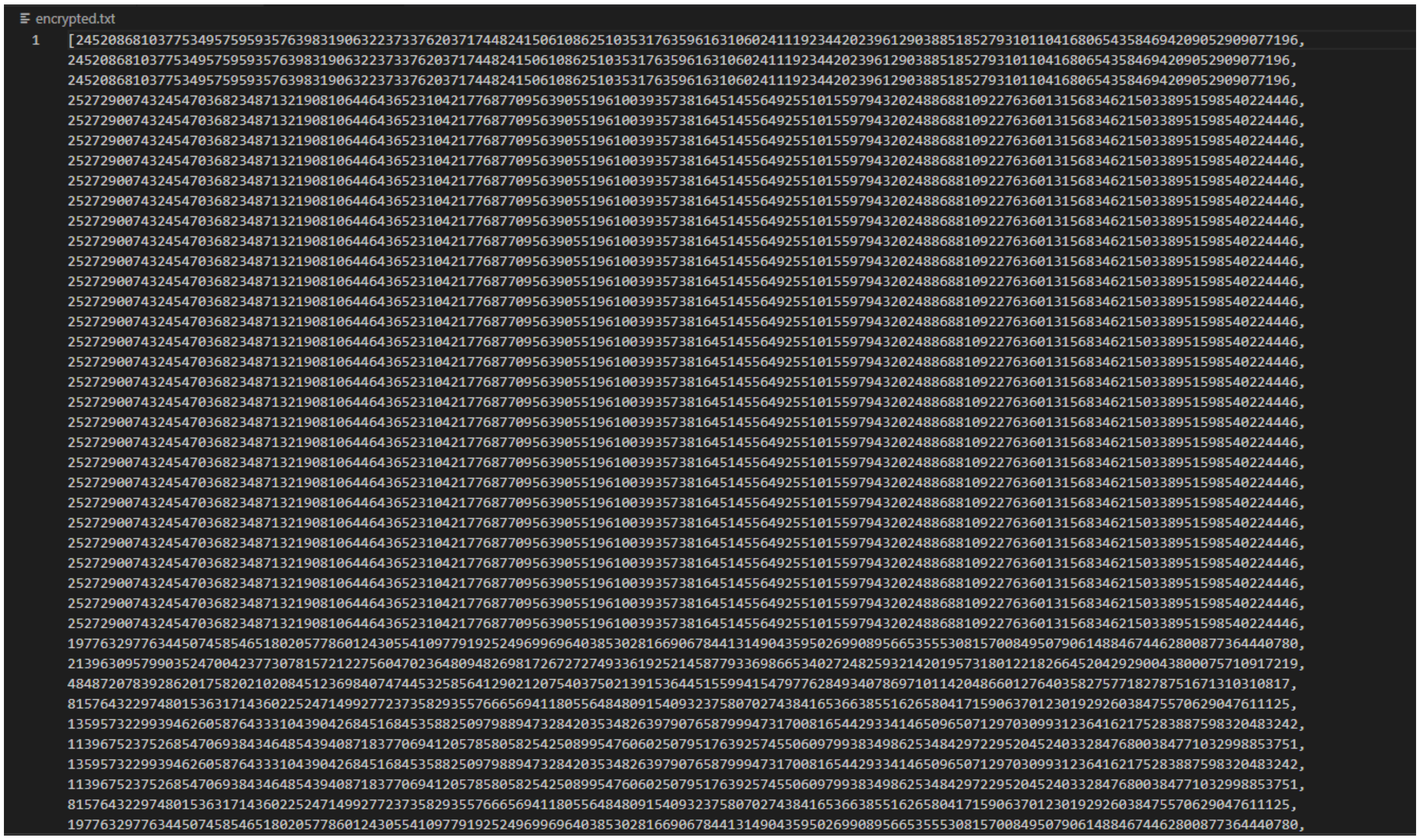
```latex
%%%=========================== Begining of the questions ===============================
\uplevel{\Large \textbf{Question answering}}
\begin{questions}
\question[10] The Euclidean algorithm is based on the following assertion. Given two integers $a$, $b$, ($a>b$),

\begin{equation}\label{gcd}
  \text{gcd}(a, b) = \text{gcd}(b, a \,\, \text{mod} \,\, b).
\end{equation}
Prove the assertion (\ref{gcd}) \textbf{mathematically}. (Note that proof by example is NOT appropriate here)

\question[20] Assuming that Alice signed a document $m$ using the RSA signature scheme. (You should describe the RSA signature structure first with a diagram and explain the authentication prin

\vspace{0.5in}
\fullwidth{\Large \textbf{Programming}}
\question[50] Implement the RSA algorithm (C/C++, Java, Python). The requirements are as follows:
\begin{itemize}
  \item Implement each component as a separate function, such as key schedule, prime test, the extended Euclidean algorithm, binary modular exponentiation, and so on.
  \item Implement both encryption and decryption of RSA. Encryption takes a txt file as input and output another txt file containing ciphertext (use hexadecimal for easy readability). Decryptio
  \item Your code should encrypt and decrypt standard keyboard characters, including letters, numbers, and symbols.
  \item The prime numbers $p$ and $q$ should be larger than $2^{64}$. (you are allowed to use libraries to handle large numbers, such as BigInteger in Java)
  \item The strategy of source coding (converting characters to integers in RSA) is up to you. You can encrypt one or more characters at a time, but make sure the constraint $m<n$ is satisfied.
  \item Use the provided file \text{RSA-test.txt} to test your code.
\end{itemize}
After implementing your code, please \textbf{answer the following questions} in your report:
\begin{parts}
  \part[10] What are the lessons you learned, and difficulties you met, in the process of implementing RSA?
  \part[10] Describe what you have done for source coding and decoding.
\end{parts}
\end{questions}
```

Demo Encryption:



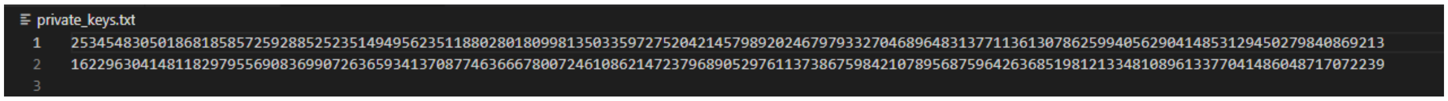
```
PS C:\Users\Hussain\Desktop\Crypto-Assignment-2> python3 .\RSA.py
----------------------------------- RSA Algorithm -------------------------------------------

Choose one of the following:
Enter (1) for Encrypt or (2) for Decrypt: 1
Please enter number of bits to generate primes (greater than 2^64= (20 digits)): 250
First Prime: 1421013932172860915757153898063760312110497816957991147876131466828059578173
Second Prime:  1783619602619324453394173709459875379537467288922232623644815863827121016481
Enter Filename to Encrypt from: RSA-test.txt

Generating Prime Numbers:
1421013932172860915757153898063760312110497816957991147876131466828059578173
1783619602619324453394173709459875379537467288922232623644815863827121016481
Generating your public / private key-pairs now . . .
Your public key is:  (1420738595075918324915755871855769720182473178901974654075980321269506146369446640661359823337150549302369776850633948755948323853526541731029419366799, 253454830501868185857259288525235149495623
5118802801809981350335972752042145798920246797933270468964831377113613078625994056290414853129450279840869213)  and your private key is:  (16229630414811829795569083699072636593413708774636667800724610862147237968905 2
97611373867598421078956875964263685198121334810896133770414860487172239, 2534548305018681858572592885252351494956235118802801809981350335972752042145798920246797933270468964831377113613078625994056290414853129450279
840869213)
Keys loaded to files...
Encrypting File entered using keys...
Enter filename to output encrypted data to: encrypted.txt
Your message has been encrypted
Decrypt message with private key:  (1622963041481182979556908369907263659341370877463666780072461086214723796890529761137386759842107895687596426368519812133481089613377041486048717072239, 2534548305018681858572592885
2523514949562351188028018099813503359727520421457989202467979332704689648313771136130786259940562904148531294502798408 69213)

========================================== END =======================================================
======================================================================================================
```

Encrypted File (encrypted.txt):



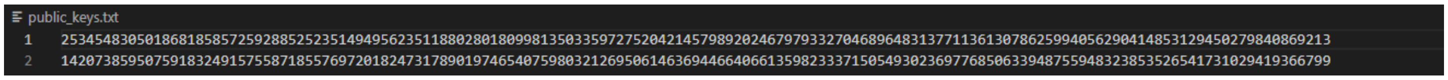
```
encrypted.txt
[245208681037753495759593576398319063223733762037174482415061086251035317635961631060241119234420239612903885185279310110416806543584694209052909077196,
245208681037753495759593576398319063223733762037174482415061086251035317635961631060241119234420239612903885185279310110416806543584694209052909077196,
245208681037753495759593576398319063223733762037174482415061086251035317635961631060241119234420239612903885185279310110416806543584694209052909077196,
2527290074324547036823487132190810644643652310421776877095639055196100393573816451455649255101559794320248868810922763601315683462150338951598540224446,
2527290074324547036823487132190810644643652310421776877095639055196100393573816451455649255101559794320248868810922763601315683462150338951598540224446,
2527290074324547036823487132190810644643652310421776877095639055196100393573816451455649255101559794320248868810922763601315683462150338951598540224446,
2527290074324547036823487132190810644643652310421776877095639055196100393573816451455649255101559794320248868810922763601315683462150338951598540224446,
2527290074324547036823487132190810644643652310421776877095639055196100393573816451455649255101559794320248868810922763601315683462150338951598540224446,
2527290074324547036823487132190810644643652310421776877095639055196100393573816451455649255101559794320248868810922763601315683462150338951598540224446,
2527290074324547036823487132190810644643652310421776877095639055196100393573816451455649255101559794320248868810922763601315683462150338951598540224446,
2527290074324547036823487132190810644643652310421776877095639055196100393573816451455649255101559794320248868810922763601315683462150338951598540224446,
2527290074324547036823487132190810644643652310421776877095639055196100393573816451455649255101559794320248868810922763601315683462150338951598540224446,
2527290074324547036823487132190810644643652310421776877095639055196100393573816451455649255101559794320248868810922763601315683462150338951598540224446,
2527290074324547036823487132190810644643652310421776877095639055196100393573816451455649255101559794320248868810922763601315683462150338951598540224446,
2527290074324547036823487132190810644643652310421776877095639055196100393573816451455649255101559794320248868810922763601315683462150338951598540224446,
2527290074324547036823487132190810644643652310421776877095639055196100393573816451455649255101559794320248868810922763601315683462150338951598540224446,
2527290074324547036823487132190810644643652310421776877095639055196100393573816451455649255101559794320248868810922763601315683462150338951598540224446,
2527290074324547036823487132190810644643652310421776877095639055196100393573816451455649255101559794320248868810922763601315683462150338951598540224446,
2527290074324547036823487132190810644643652310421776877095639055196100393573816451455649255101559794320248868810922763601315683462150338951598540224446,
2527290074324547036823487132190810644643652310421776877095639055196100393573816451455649255101559794320248868810922763601315683462150338951598540224446,
2527290074324547036823487132190810644643652310421776877095639055196100393573816451455649255101559794320248868810922763601315683462150338951598540224446,
2527290074324547036823487132190810644643652310421776877095639055196100393573816451455649255101559794320248868810922763601315683462150338951598540224446,
2527290074324547036823487132190810644643652310421776877095639055196100393573816451455649255101559794320248868810922763601315683462150338951598540224446,
2527290074324547036823487132190810644643652310421776877095639055196100393573816451455649255101559794320248868810922763601315683462150338951598540224446,
2527290074324547036823487132190810644643652310421776877095639055196100393573816451455649255101559794320248868810922763601315683462150338951598540224446,
2527290074324547036823487132190810644643652310421776877095639055196100393573816451455649255101559794320248868810922763601315683462150338951598540224446,
2527290074324547036823487132190810644643652310421776877095639055196100393573816451455649255101559794320248868810922763601315683462150338951598540224446,
2527290074324547036823487132190810644643652310421776877095639055196100393573816451455649255101559794320248868810922763601315683462150338951598540224446,
2527290074324547036823487132190810644643652310421776877095639055196100393573816451455649255101559794320248868810922763601315683462150338951598540224446,
2527290074324547036823487132190810644643652310421776877095639055196100393573816451455649255101559794320248868810922763601315683462150338951598540224446,
1977632977634450745854651802057786012430554109779192524969969640385302816690678441314904359502699089566535553081570084950790614884674462800877364440780,
2139630957990352470042377307815721227560470236480948269817267272749336192521458779336986653402724825932142019573180122182664520429290043800075710917219,
484872078392862017582021020845123698407474453258564129021207540375021391536445155994154797762849340786971011420486601276403582757718278751671310310817,
815764322974801536317143602252471499277237358293557666569411805564848091540932375807027438416536638551626580417159063701230192926038475570629047611125,
1359573229939462605876433310439042684516845358825097988947328420353482639790765879994731700816544293341465096507129703099312364162175283887598320483242,
1139675237526854706938434648543940871837706941205785805825425089954760602507951763925745506097993834986253484297229520452403328476800384771032998853751,
1359573229939462605876433310439042684516845358825097988947328420353482639790765879994731700816544293341465096507129703099312364162175283887598320483242,
1139675237526854706938434648543940871837706941205785805825425089954760602507951763925745506097993834986253484297229520452403328476800384771032998853751,
815764322974801536317143602252471499277237358293557666569411805564848091540932375807027438416536638551626580417159063701230192926038475570629047611125,
1977632977634450745854651802057786012430554109779192524969969640385302816690678441314904359502699089566535553081570084950790614884674462800877364440780,
```

private_keys.txt (Generated file):



```
private_keys.txt
2534548305018681858572592885252351494956235118802801809981350335972752042145798920246797933270468964831377113613078625994056290414853129450279840869213
1622963041481182979556908369907263659341370877463666780072461086214723796890529761137386759842107895687596426368519812133481089613377041486048717072239

```

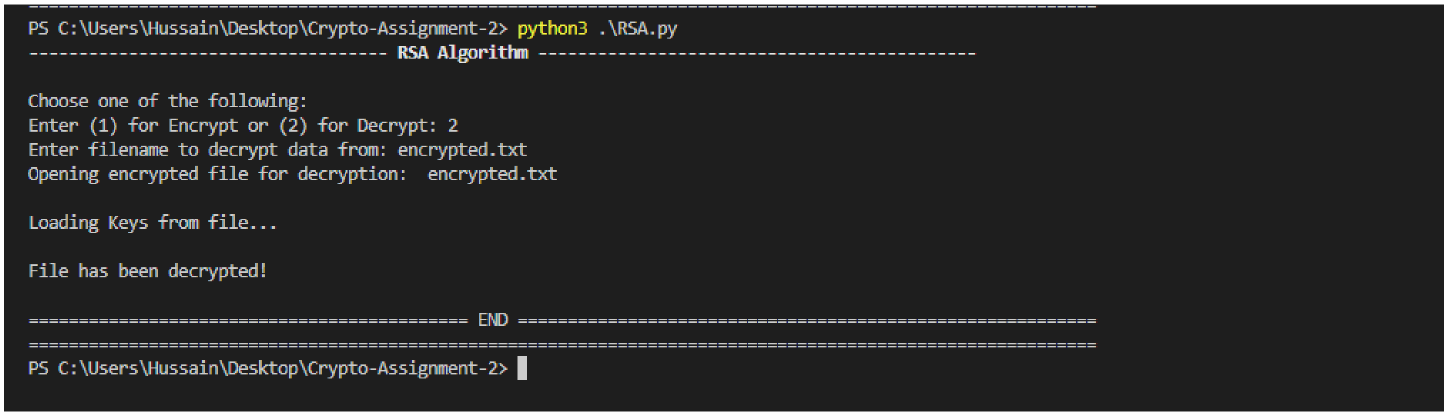
public_keys.txt (Generated File):



public_keys.txt

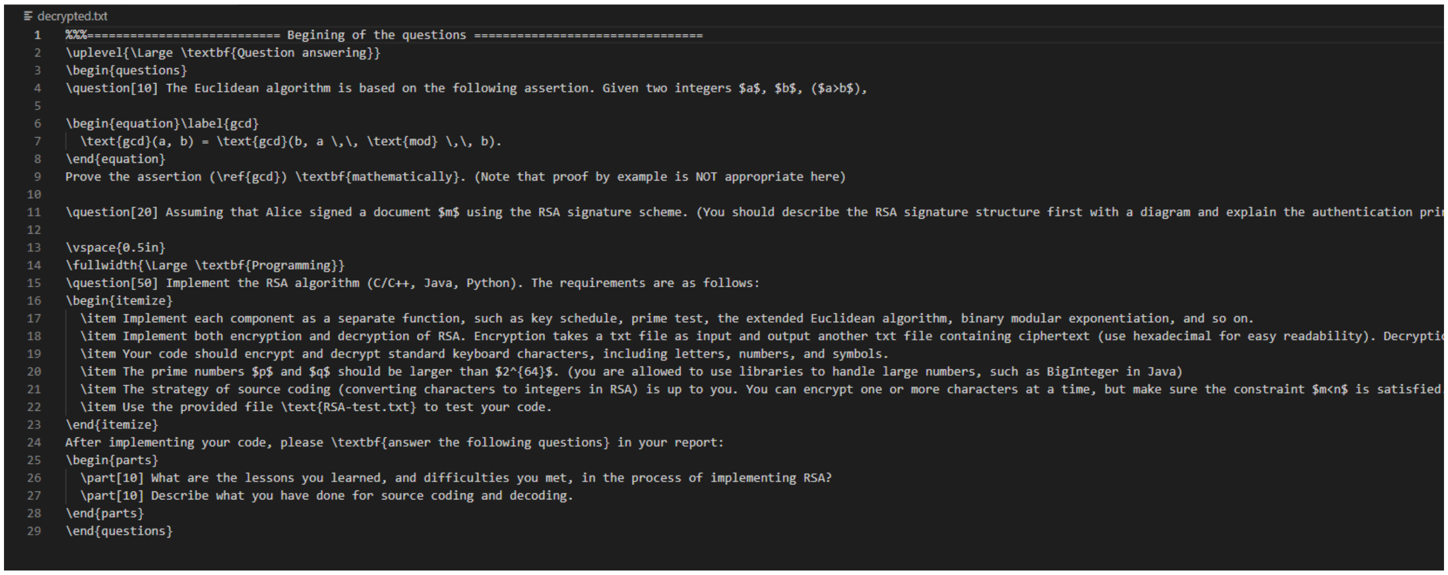
```
2534548305018681858572592885252351494956235118802801809981350335972752042145798920246797933270468964831377113613078625994056290414853129450279840869213
1420738595075918324915755871855769720182473178901974654075980321269506146369446640661359823337150549302369776850633948755948323853526541731029419366799
```

Demo Decryption:



```
-----------------------------------------------------------------------------------------------------------------
PS C:\Users\Hussain\Desktop\Crypto-Assignment-2> python3 .\RSA.py
------------------------------------ RSA Algorithm --------------------------------------------

Choose one of the following:
Enter (1) for Encrypt or (2) for Decrypt: 2
Enter filename to decrypt data from: encrypted.txt
Opening encrypted file for decryption:  encrypted.txt

Loading Keys from file...

File has been decrypted!

============================================= END ===========================================================
=============================================================================================================
PS C:\Users\Hussain\Desktop\Crypto-Assignment-2> █
```

Decrypted File (decrypted.txt)



decrypted.txt

```
%%%========================== Begining of the questions ===============================
\uplevel{\Large \textbf{Question answering}}
\begin{questions}
\question[10] The Euclidean algorithm is based on the following assertion. Given two integers $a$, $b$, ($a>b$),

\begin{equation}\label{gcd}
  \text{gcd}(a, b) = \text{gcd}(b, a \,\, \text{mod} \,\, b).
\end{equation}
Prove the assertion (\ref{gcd}) \textbf{mathematically}. (Note that proof by example is NOT appropriate here)

\question[20] Assuming that Alice signed a document $m$ using the RSA signature scheme. (You should describe the RSA signature structure first with a diagram and explain the authentication pri

\vspace{0.5in}
\fullwidth{\Large \textbf{Programming}}
\question[50] Implement the RSA algorithm (C/C++, Java, Python). The requirements are as follows:
\begin{itemize}
  \item Implement each component as a separate function, such as key schedule, prime test, the extended Euclidean algorithm, binary modular exponentiation, and so on.
  \item Implement both encryption and decryption of RSA. Encryption takes a txt file as input and output another txt file containing ciphertext (use hexadecimal for easy readability). Decryptio
  \item Your code should encrypt and decrypt standard keyboard characters, including letters, numbers, and symbols.
  \item The prime numbers $p$ and $q$ should be larger than $2^{64}$. (you are allowed to use libraries to handle large numbers, such as BigInteger in Java)
  \item The strategy of source coding (converting characters to integers in RSA) is up to you. You can encrypt one or more characters at a time, but make sure the constraint $m<n$ is satisfied
  \item Use the provided file \text{RSA-test.txt} to test your code.
\end{itemize}
After implementing your code, please \textbf{answer the following questions} in your report:
\begin{parts}
  \part[10] What are the lessons you learned, and difficulties you met, in the process of implementing RSA?
  \part[10] Describe what you have done for source coding and decoding.
\end{parts}
\end{questions}
```

3a) What are the lessons you learned, and difficulties you met, in the process of implementing RSA?

I learned about Prime number generation and the complexity behind accurately generating large prime numbers. Additionally, I learned about Public and Private Key pairs and techniques of generating them. Lastly, I learned about the security and complexity associated with RSA and importance of large key sizes.

The difficulty I faced when implementing RSA was the generation of prime numbers for Key Pair generation. I tried multiple techniques however, efficiently generating prime number greater than 2^64 was difficult. I was able to figure out an effective method as explained on GeeksForGeeks website for efficient and quick generation of RSA Prime Numbers.

Lastly, I faced a little issue with time management due to multiple submission within the same week.

3b) Describe what you have done for source coding and decoding.

I start off by generating prime numbers which are required for the generation of key pairs. The primes are sent to the key generation function (gen_key_pair ()) where phi, e, gcd and multiplicative inverse are all calculated and Public Key (e, n) and Private Key (d, n) are generated.

Encryption Function (Source Coding):

The encryption function takes in plain text and checks for public key file. N and e are read from the file and put into the function. We use ord () to convert each character to its ASCII value and pow () function to do the calculation of a^b mod n. This is repeated for each digit in the plain text and an array of bytes is returned.

Decryption Function (Decoding):

The decryption function takes in cipher text and searches for the private keys file. D and n are read from the file and put into the function. The pow () function is used and a^b mod n is done, the result is casted to an str () and stored in a list. The list is then parsed and converted to int, the result of which is converted to a character using chr () function. The array of bytes is then returned as a string and written to a file.

References

https://www.geeksforgeeks.org/how-to-generate-large-prime-numbers-for-rsa-algorithm/

https://math.stackexchange.com/questions/59147/why-gcda-b-gcdb-a-bmod-b-understanding-euclidean-algorithm

https://www.quora.com/What-are-the-proofs-for-gcd-a-b-gcd-b-a-mod-b