An Implementation of the RSA cryptographic Algorithm

Candidate number: 11

Due Date of submission: 16th of December 2019

Abstract

More and more information is exchanged digitally through the world wide web, resulting in more information being possible to intercept for adversary agents. This development makes secure encryption of the information transmitted a necessity to maintain confidentiality. The RSA cryptosystem invented in the 1970s is a asymmetric algorithm where there is two different keys used for respectively encryption and decryption. The RSA cryptosystem is built upon important mathematical constructs; prime numbers, modular multiplicative inverse, greatest common divisior, Fermats little theorem and extended euclidean algorithm. This paper presents the approach of developing an RSA cryptosystem and provides an implementation of the aforementioned mathematical constructs. Further building them together as an RSA cryptosystem in Python, including examples of encrypted messages exchanges.

1 Introduction

With the evolution of computers and their rapid increase in computational power, maintaining the confidentiality and integrity of information has raised the public awareness of secure cryptographic algorithms. Daily, when browsing the world wide web, doing online banking or just sending an email, we are using these "magical" inventions, called cryptographic algorithms. They attempts to exploit the art of number theory and cryptography to protect us against cryptanalysis and vulnerabilities in todays digital world.

This article explains and describe an implementation of one of the magical inventions, namely the RSA cryptographic algorithm, which is named after the scientist that published the RSA algorithm first, Rivest, Shamir and Adleman (see [1] for the interesting fact that it was invented by another mathematician in secrecy before them). RSA is categorized as a asymmetric cryptosystem, as it uses both a public and a private key pair, i.e. that there is not a prerequisite that keys has been shared beforehand amongst the communicating parties. Simply phrased, there are two different keys, one to lock e.g. a door and another one to open it. While for symmetric cryptosystems, they require a pre-shared key.

RSA can be applied both for encryption/decryption as well as authentication signatures [2].

The article will firstly describe and explain the RSA algorithm, thereafter illustrating the algorithm with a textbook example of the RSA implemented using Python. Further, a short description of the security of the RSA and ending the paper with a conclusion.

2 The RSA algorithm

This parts describes and explains the RSA algorithm, which can be divided into six steps, when not considering padding. This is due to that padding is merely to add redundant data to enable the block ciphers to be of an even size of digits and to add confusion.

2.1 The main outlines of RSA

The main outlines of RSA can be illustrated by using the three crypto celebrities, Alice, Bob and Eve.

Bob sends a message to his new girlfriend Alice, however he is afraid that his ex-girlfriend Eve is eavesdropping on them. Therefore he decides to encrypt the message using the RSA algorithm, as him and Alice does not have any pre-shared key. Thus Bob dives into the theory of RSA. He quickly realizes that the three major builds blocks that he has to derive is the modulus (n), the public key component (e) and the secret key component (d) (the following section will describe the requirements of how to derive these components). When Bob is sending a message to Alice, he takes his message M and transforms it into a ciphertext C.

When he does this he is using the modular exponentiation with Alice's public key pair $({}_A,\ n_A)$ to generate C with the equation $C = M^e (mod\, n)$. Thereafter he sends the message using Whatsapp. When it arrives in Alice's inbox, she directly applies here private key pair $(\ d_A,\ n_A)$ to decrypt Bob's message using the also modular exponentiation. However she uses it to retrieve the plain text. Alice does that by calculating $M = C^d (mod\, n)$. Meanwhile, Eve is trying to decrypt the message from Bob to Alice, which she intercepted. However, as she does not know the secret key of Alice or the primes that the modulus is the product of, she is not able to decipher the message.

The following sections, describes what Bob and Alice had to do to generate their three components to fulfill the mathematical properties required to implement RSA.

2.2 The modulus, n

The first action Bob have to make before he encrypts the message to Alice, is to generate two random primes, p and q. The product of p and q is the modulus n, $n = pq$. The question arises, how to generate a **random prime** number?

2.2.1 Pseudorandom prime generator algorithm

To generate a random prime Bob needs to follow the steps outlined [2, p. 329]:

1. Generate a random number p within some range $p_{min} - p_{max}$. The Python script *rsa.py line 36* generates p using the *randint(* p_{min}, p_{max}*)* from the library package *random*.

2. By definition, even number cannot be prime numbers, the second step of generating a prime is to check if it is even. If that is true, increment p by one, to make it odd. Thus to set p within the range of $p_{min} - p_{max}$.

3. If p is incremented by one, it is possible that p becomes out of range ($p = p_{max}, p > p_{max}$ p_{max} is a even number. Thus, Bob needs to check that p is within the required range. If p is out of range, setting $p = p_{min} + p$ ($mod\, p_{max}$ $+$ 1 sets p to p_{min} *or* $p_{min} + 1$, *depending on the previous step of this algorithm.*

   Thereafter, it is necessary to go back to step 2, to ensure that the new p is odd.

4. *When both steps 2 and 3 is finished successfully, then Bob proceeds to this step, which is to check if the number is actually a prime. This is done by generating a random positive odd integer f, finding the greatest common divisor of p - 1 and f. If those two numbers are relative prime (*$\gcd(p-1, f) = 1$*), then Bob can check if p is a prime, using Fermat primality test. If the test concludes the number is a prime, then there is a high probability that it is a prime.*

   The primality test applies Fermat little theorem, which states that if p does not divide a, then $a^{p-1} \equiv 1\ (mod\, p)$*, meaning that p is a prime. Thus if the number Bob generates passes the primality test, he can know that his number is high likelihood a prime [1].*

   If the number p is not a prime, then p is incremented by two, and Bob have to move back to step 3 in the algorithm.

2.3 The public key component, e

After the modulus n is generated in RSA, the public key component e must be derived. The two condition that have to be satisfied, is that $\phi(n)$ and e are relative primes, i.e. $gcd(e, n) = 1$ and that e $\phi(n)$.

$$\phi(n) = (p-1)(q-1)\ , when\ q\ and\ p\ are\ primes.$$

In *rsa.py* 7.1, a procedure *gen_public* is performed repeatedly until it matches the conditions required for e.

THEOREM 1 If a and m are relatively prime integers and $m > 1$, then an inverse of a modulo m exists. Furthermore, this inverse is unique modulo m. (That is, there is a unique positive integer $\overline{a}$ less than m that is an inverse of a modulo m and every other inverse of a modulo m is congruent to $\overline{a}$ modulo m.)

Figure 1: Theorem for deriving the secret key components, d [1, p.275].

2.4 The secret key component, d

The secret key component in RSA, d is the modular multiplicative inverse of e, when the modulus is $\phi(n)$, $d = e^{-1}(mod\ \phi(n))$. According to Figure 1, the inverse of e, d is unique for the positive number less than the modulo, $\phi(n)$. The uniqueness ensures that there is only a number and not multiple numbers that can unlock the cipher text, which is a vital condition in the RSA's security.

The procedure to find the modular multiplicative inverse is the extended euclidean algorithm (see [2, p. 176] for the algorithm applied in this paper). Mainly, the extended euclidean algorithm derives each step in the gcd algorithm. Thereafter, it sets each iteration of gcd value equal to the value of the given iteration. Next, inserting for the remainder, and then you end up with the secret key, d. This can be illustrated with the following example, where $e = 7$ and $\phi(n) = 60$:

$$gcd(60, 7) : 60 = 8 \times 7 + 4$$
$$7 = 1 \times 4 + 3$$
$$4 = 1 \times 3 + 1$$
$$3 = 3 \times 1 + 0$$

Finding the inverse: $d \times e + s \times \phi(n) = 1$

$$1 = 4 - 3$$
$$3 = 7 - 4$$
$$4 = 60 - 8 \times 7$$

Insertion:

$$1 = 4 - 3 = 4 - (7 - 4) = 2 \times 4 - 7 = 2 \times (60 - 8 \times 7) - 7$$
$$1 = 120 - 17 \times 7$$

Inverse is $d = -17$, and the positive inverse $d = -17\ (mod\ 60) = 43$
Then the modular multiplicative inverse and secret key components of $7\ (mod\ 60)$ is 43.

2.5 Publish public key

After the key components are generated, then the public key pairs are exchanged, normally using a public key infrastructure.

2.6 Encrypt message, M

At this point, Bob can encrypt the message to Alice using Alice public key pair. To perform the encryption, and transform the plain text to cipher text, the following operation is performed; each message block (M) into the power of the public key of the message receiver (e), modulo the modulus of the receiver (n):

$$C = M^e (mod\, n)$$

2.7 Decrypt ciphertext, C

When Alice receives the ciphertext (C), she needs to decrypt it. Decrypting C is done by reversing the encryption, such that plain text is retrieved. That is performed by calculating C in the power of the receivers secret key, modulo the receivers modulus:

$$M = C^d (mod\, n)$$

3 Program Execution:

This chapter outlines the results of three sequential execution of the Python implementation of the RSA algorithm listed in the 7.1 of this paper. A step-wise output of the first execution of *rsa.py*, from plain text, to padded plain text, then next to padded plain text converted to numbers, the encryption and back from the cipher text can be found in 7.2.

3.1 Organization Source code

This part briefly explains the setup of the code. The *main.py* comprises the linking of the two other packages *rsa.py* and *text_padding.py* , which respectively implements the RSA algorithm described in section 2 and the padding of the result. The latter, as mentioned in section 2 does not provide any encryption, thus largely not addressed in this paper. *rsa.py* implements the parts described in 2.

3.2 Execution 1 - base case

Figure 2 lists the parameters of Alice and Bob in the base case of this paper; Bob sends a message to Alice, asking whether the encryption using RSA cryptosystem, in this program really works. The communication between Bob and Alice is listed in the text box below. The box displays the input (plain text), corresponding cipher text and decrypted text given the parameters derived by the program in Figure 2. Note, the reason why Alice and Bob always have their respective pseudo random prime numbers, p and q in the range 90,000-91,000 is due this range is set in the *gen_prime()* function in 7.1 in each execution of the program.

Name	Type	Size	Value
alice_d	int	1	1350439543
alice_e	int	1	6485941127
alice_n	int	1	8171195509
alice_p	int	1	90641
alice_q	int	1	90149
bob_d	int	1	1899877137
bob_e	int	1	971927185
bob_n	int	1	8209149061
bob_p	int	1	90263
bob_q	int	1	90947

Figure 2: Parameters of Execution 1

Bob's message to Alice: Alice, does this RSA encryption work?

Ciphertext: 414036696 1523330796 2173987070 595192032 879752267 398612489 5646173841 5379662166 6283838990 3464136775 7654978218 5714424823 345558973 5809187849 210625534 1875123913 7409654159 4760539384 6829424164 5450729414

Alice decrypts text: Alice, does this RSA encryption work?

Alice's replies to Bob: yes Bob, the RSA algorithm works for as a crypto algorithm

Ciphertext: 1856030455 5445326412 6935957528 147131935 4491656650 288887007 6971520128 2359593732 1311409484 2866700008 6334045738 1175028536 3291362151 1828857742 6334045738 2039550620 6470603201 6334045738 1311409484 5445326412 4889319957 2115599534 2721853053 3067268654 1311409484 2866700008 6334045738 1175028536 3291362151 5423913716 5423913716 5423913716

Bob decrypts text: yes Bob, the RSA algorithm works for as a crypto algorithm

Name	Type	Size	Value
alice_d	int	1	584367571
alice_e	int	1	5487679339
alice_n	int	1	8191208587
alice_p	int	1	90149
alice_q	int	1	90863
bob_d	int	1	4181123815
bob_e	int	1	6419628679
bob_n	int	1	8201106031
bob_p	int	1	90647
bob_q	int	1	90473

Figure 3: Parameters of Execution 2

3.3 Execution 2 - alteration of primes

This execution of the RSA implementation encrypts the identical plain text message as the previous section, except from one of Alice's prime numbers (*i.e. 90149*). The effect of altering the primes is seen in the two cipher texts, is seen in the text box. When comparing the cipher text in execution 2 with the base case execution, it is visible how only two digits in the last block cipher in Alice's reply is not altered (see bold digits Execution 1: 54**2**39**1**3716, Execution 2: 15**2**77**1**5357), despite that only one of Alice's primes is altered and the other hold constant (90149).

Bob's message to Alice: Alice, does this RSA encryption work?

Ciphertext: 6474272806 4945936162 622366098 3850050403 297356149 2693623627 2719815516 5485837151 546184425 7434575004 4142520869 3432974718 2466988932 934679196 3675073482 5266233883 1378662415 4358977967 6640266020 4148743658

Alice decrypts text: Alice, does this RSA encryption work?

Alice's replies to Bob: yes Bob, the RSA algorithm works for as a crypto algorithm

Ciphertext: 265489541 3457688594 4561667235 3902496074 4258013962 2070927295 5133467171 7049324784 6400127280 2977115126 3575293773 3929977683 2123541178 39208472 3575293773 3275426897 7153844057 3575293773 6400127280 3457688594 6534854590 509930591 873825044 5970789115 6400127280 2977115126 3575293773 3929977683 2123541178 1527715357 1527715357 1527715357

Bob decrypts text: yes Bob, the RSA algorithm works for as a crypto algorithm

3.4 Execution 3 - alteration of primes and plain text

The last execution of the RSA program implemented in this paper is an implementation where both the primes and the plain text that Bob sends Alice has been altered. This verifies that the program actually works, on multiple different plain texts and not just with different primes.

Name	Type	Size	Value
alice_d	int	1	4212435811
alice_e	int	1	954318091
alice_n	int	1	8217574901
alice_p	int	1	90481
alice_q	int	1	90821
bob_d	int	1	3356666321
bob_e	int	1	7821612713
bob_n	int	1	8186628551
bob_p	int	1	90437
bob_q	int	1	90523

Figure 4: Parameters of Execution 3

Bob's message to Alice: Alice, I am trying with another text, does it still work?

Ciphertext: 2481462288 7920703222 6687135788 6230915228 7327154345 6316449688 4211349955 4070002452 7275935373 2843391802 4787662787 6270340508 2388869069 7254878906 4879492804 2683578078 5606881603 1860691745 3575761169 2419340671 6426772539 3463623644 6478911077 7719426413 1559173551 7609899578 1732336517 428549086 3269033446 5251245704 5251245704 5251245704

Alice decrypts text: Alice, I am trying with another text, does it still work?

Alice's replies to Bob: yes Bob, the RSA algorithm works as a crypto algorithm

Ciphertext: 541944451 6423758366 4281248542 2417502274 4995820175 7910123482 6289550707 4834884128 6452591721 5282033820 4198977435 2763009698 7916746959 5254689355 4198977435 1535260465 6452591721 6423758366 2439354552 7804409815 1440120832 3054762866 6452591721 5282033820 4198977435 2763009698 7916746959 7146207229

Bob decrypts text: yes Bob, the RSA algorithm works as a crypto algorithm

4 The Security of RSA

This section briefly discusses the security of the RSA algorithm. For a more detailed description the readers is especially referred to the reference [2]

4.1 Large provide security in RSA

It is arguable that everything has to be large in RSA, for it to be secure. Example, if a small modulo is used, it is possible to factorize the modulus within reasonable time into the two primes that the modulus is the product of. Thereafter deriving the secret components, the modular multiplicative inverse of the public key component, hence an adversary can decrypt the ciphertext. Though, if a large modulus is used, i.e. that is it a product of two large primes, then this becomes computational infeasible within reasonable time.

Another momentum that supports that large numbers provides security in RSA, is that if the public key component is small then broadcast attacks is possible. The broadcast attacks are possible if the number of block ciphers, C are larger than the public key. [2] suggest that the public key, e size must be minimum 65537 and implementing padding in the RSA scheme. Thereby, avoiding the possibility that the plain text can be retrieved by $\sqrt[e]{C}$.

The third argument for large numbers providing security in RSA, is that in 1990 it was proven, that if the secret component of RSA is small, it can be derived if it is less than the square root of the modulus [2, p.322].

Note, also that if the secret key components become public, there is methods of factoring the modulus, and obtain the primes. Thus, again implying that the encryption can be broken, see [2] for the method of how to perform the factorization when the secret key is known.

4.2 Not commonly known modulus

If several actors in an RSA cryptosystem shares the same modulus when having published their public key pair. Then, it is possible for other actors with the same modulus to apply their public and secret key components to factorize the modulus into the two primes it is composed of. Implying, that the security of the RSA is breached. This advocates for even larger random primes, as the probability of actors in an RSA cryptosystem having a identical modulus diminishes as the size of the primes increases [2].

4.3 Hardware security of implementation

Another potential security vulnerability of the RSA, that do not strictly relate to the feasibility of "breaking" the math behind it, is the implementation into hardware. An example, is the implementation provided in this paper, as it does not implement measure to erase either Alice or Bob's secret keys or their random primes. Thus, a malicious agent, Eve may read the secret keys or the primes from the computer memory. Eve is then able to either decrypt directly or derive the public and private key pair of both Bob and Alice [2].

5 Conclusion

The conclusion of this paper that implements RSA in Python, is that the RSA cryptosystem have large computational requirements, and if those requirements are not fulfilled then the security of the algorithm may be severely reduced. This being especially related to the problem of factorization of large into prime numbers. Note, therefore the RSA cryptosystem shall not be implemented in system where the computational resources are scarce or the transmitted data is long, then there exist other options. In addition, special care must be taken when implementing RSA, to ensure that numbers are large enough to reduce the possibility of mathematical ”breaking” the encryption.

6

References

[1] Kenneth H. Rosen. *Discrete Mathematics and Its Applications*. 7th edition. McGrawHill, 2012. ISBN: 9780073383095.

[2] Michael Welschenbach. *Cryptography in C and C++*. Apress, 2001. ISBN: 189311595.

7 Appendix

7.1 Source code

main.py

```
#Main program file - running the RSA encryption code
import text_padding
import rsa

bob_plain_text = "b_msg.txt"
bob_pad_text = "b_pad_msg.txt"
bob_convert_to_num_text = "b_converted_num_file.txt"
bob_cipher_text = "b_cipher_text.txt"
alice_decrypt_convert_to_num_text = "a_decrypt_converted_num_file.txt"
alice_decrypt_pad_text = "a_d_pad_msg.txt"
alice_decrypted_plain_text = "a_d_message.txt"

alice_plain_text = "a_msg.txt"
```

```
alice_pad_text = "a_pad_msg.txt"
alice_convert_to_num_text = "a_converted_num_file.txt"
alice_cipher_text = "a_cipher_text.txt"
bob_decrypt_convert_to_num_text = "b_decrypt_converted_num_file.txt"
bob_decrypt_pad_text = "b_d_pad_msg.txt"
bob_decrypted_plain_text = "b_d_message.txt"


bob_p = rsa.gen_prime(90000, 91000)
bob_q = rsa.gen_prime(90000, 91000)
bob_n = rsa.gen_modulus(bob_p, bob_q)
bob_e = rsa.gen_public(bob_p, bob_q)
bob_d = rsa.gen_secret(bob_p, bob_q, bob_e)

alice_p = rsa.gen_prime(90000, 91000)
alice_q = rsa.gen_prime(90000, 91000)
alice_n = rsa.gen_modulus(alice_p, alice_q)
alice_e = rsa.gen_public(alice_p, alice_q)
alice_d = rsa.gen_secret(alice_p, alice_q, alice_e)

########## Bob sends message to Alice
message = "Alice, I am trying with another text, does it still work?"
print("Bob to Alice: ",message)
plain_txt = open (bob_plain_text, "w")
plain_txt.write(message)
plain_txt.close()

text_padding.pad_txt(bob_plain_text, bob_pad_text) # Bob pads the plain message
text_padding.convert_to_num(bob_pad_text, bob_convert_to_num_text, bob_n) # Bob converts the padded plain message to numbers
rsa.encrypt(alice_e, alice_n, bob_convert_to_num_text, bob_cipher_text, 4) # Bob encrypts the numbers to block ciphers


## Alice decrypts Bobs message
rsa.decrypt(alice_d, alice_n, bob_cipher_text, bob_decrypt_convert_to_num_text, 4)
text_padding.convert_to_char(bob_decrypt_pad_text, bob_decrypt_convert_to_num_text)
text_padding.unpad_txt(bob_decrypted_plain_text, bob_decrypt_pad_text)
```

```


########## Alice replies to Bob
plain_txt = open (bob_decrypted_plain_text , "r")
message = plain_txt.read()
print("Alice receives from Bob: ", message)
if message == "Alice, I am trying with another text, does it still work?":
    message = "yes Bob, the RSA algorithm works as a crypto algorithm"
else:
    message = "You sent some jewbrish Bob, can you send it again?"
plain_txt = open (alice_plain_text , "w")
print("Alice sends to Bob: ", message)
plain_txt.write(message)
plain_txt.close()

text_padding.pad_txt(alice_plain_text , alice_pad_text) # Alice pads the plain message
text_padding.convert_to_num(alice_pad_text , alice_convert_to_num_text , alice_n) # Alice converts the padded plain message to numbers
rsa.encrypt(bob_e, bob_n, alice_convert_to_num_text , alice_cipher_text , 4) # Alice encrypts the numbers to block ciphers

## Bob decrypts Alice message
rsa.decrypt(bob_d, bob_n, alice_cipher_text , alice_decrypt_convert_to_num_text , 4)
text_padding.convert_to_char(alice_decrypt_pad_text , alice_decrypt_convert_to_num_text)
text_padding.unpad_txt(alice_decrypted_plain_text , alice_decrypt_pad_text)
plain_txt = open (alice_decrypted_plain_text , "r")
message = plain_txt.read()
print("Bob receives from Alice: ", message)
```

rsa.py

```
#RSA encryption and decryption implementation
import random

def gcd(a, b): # for a, b => 0 # greatest common divisor
    c = a
    d = b
```

```
    while (d != 0):
        r = c % d
        c = d
        d = r
    return c

def if_even_make_odd(p): # 2. check if p number is even
    if (p % 2) == 0:
        p = p + 1
    return p

def is_p_bigger_pmax(p, p_min, p_max): # 3. check if p > p_max
    if p > p_max:
        p = p_min + p % (p_max + 1)
        return True
    else:
        return False

def check_if_prime(p):# 4.  primality test - fermats test
    a = random.randint(2, (p - 2)) # generate int between values
    if ((a**(p-1) % p) == 1):
        return True
    else:
        return False

def gen_prime(start, end): # cryptography in c & c++ p.329
                                    # start, end >= 1
    a = 1
    if (a == 1):
            p = random.randint(start, end) # generate int between values
            a = 2
    while(a != 5):
        if (a == 2):
            p = if_even_make_odd(p)
            a = 3
        elif (a == 3):
            b = is_p_bigger_pmax(p, start, end)
            if (b == True):
                a = 2
            else:
                a = 4
        elif (a == 4):
```

```
            f = random.randint(start, end)
            f = if_even_make_odd(f) # f needs to be an odd positive integer
            if (gcd(p - 1, f) == 1) and check_if_prime(p):
                a = 5
            else:
                p = p + 2
                a = 3
    return p


########## key pair generation
#1.1 Generate the modulus, n --> n = pq, where p and q are to random primes
def gen_modulus(p, q): # p and q are primes
    n = p*q
    return n

#1.2 derive the public key component, e --> relative prime to phi(n) and e < phi(n)
def gen_public(p, q):
    phi_n = (p - 1)*(q - 1)
    while True:
        e = random.randint(2, (phi_n - 1))
        if (gcd(e, phi_n) == 1): # run loop until gcd(e, phi_n) = 1
            return e
            break

#1.3 derive the secret key component, d --> modular inverse of e
def xtnd_gcd(a, b): # a = phi_n > b = e, from Crypto C/C++ p.176
    # gcd(a, b) = u*a + v*b, where v = secret component (the inverse of e)
    v_1 = -1
    v_3 = -1
    q = -1
    t_3 = -1
    t_1 = -1
    v = -1
    u = 1
    d = a
    if(b == 0):
        v = 0
```

```
        return v
    else:
        v_1 = 0
        v_3 = b
    while (v_3 != 0):
        t_3 = d % v_3
        q = int((d - t_3) / v_3)
        t_1 = int(u - q * v_1)
        u = v_1
        d = v_3
        v_1 = t_1
        v_3 = t_3
    v = int((d - u*a)/b)
    v = v % a
    return v

def gen_secret(prime_1, prime_2, public_key):
    phi_n = (prime_1 - 1)*(prime_2 - 1)
    private_key = xtnd_gcd(phi_n, public_key)
    return private_key

## message encryption
def encrypt(encryption_key, modulus, plain_file, cipher_file, block_size):
    plain_text = open (plain_file, "r")
    cipher_text = open (cipher_file, "w")
    while True:
        encrypt_block = plain_text.read(block_size + 1)
        if not encrypt_block:
            break
        encrypt_block = encrypt_block.rstrip(" ")
        encrypt_block = int(encrypt_block)
        cipher_block = pow(encrypt_block, encryption_key, modulus)
        cipher_block = str(cipher_block)
        cipher_text.write(cipher_block + " ")
    plain_text.close()
    cipher_text.close()


#3.1 decrypt the ciphertext
def decrypt(decryption_key, modulus, cipher_file, plain_file, block_size):
    plain_text = open (plain_file, "w")
    cipher_text = open (cipher_file, "r")
    decrypt_block = ""
```

```
    whitespace = " "
    while True:
        nxt_char = cipher_text.read(1) # read one char at the time
        if not nxt_char:
            break
        elif nxt_char == whitespace:
            decrypt_block = decrypt_block.rstrip(" ")
            decrypt_block = int(decrypt_block)
            plain_block = pow(decrypt_block, decryption_key, modulus)
            plain_block = str(plain_block)
            if len(plain_block) == 3: #pad with one digit
                plain_block = "0" + plain_block
            plain_text.write(plain_block + " ")
            decrypt_block = "" #reset decrypt block to read the next block
        else:
            decrypt_block = str(decrypt_block)
            decrypt_block = decrypt_block + nxt_char
    plain_text.close()
    cipher_text.close()
```

text_padding.py

```
#Text padding script

 # 1. pad original message
def pad_txt(plain_text, output_padded_text):
    msg_file = open(plain_text, "r")
    message = msg_file.read()
    message = message.rstrip("\n") #strip away the new line in end of file --> correct padding of padding char
    num_of_char = len(message)
    msg_file.close()

    # find length of padding
    len_blocks = 8 # plain text padding length
    if((num_of_char % len_blocks) > 0): # only pad if there is a remainder in one block
        len_padding = len_blocks - (num_of_char % len_blocks)
    else: # no remainder put lenngth of padding to zero
        len_padding = 0

```

```
    # pad file:
    char = "X"      # char used for padding
    pad_chars = ""
    pad_file = open(output_padded_text, "w")
    pad_file.write(message) # original message text

    pad_file = open(output_padded_text, "a") #start padding
    for i in range (len_padding):
       pad_chars+= char

    pad_file.write(pad_chars) # write all the padding to the padding file
    pad_file.close

# 2. convert blocks of chars into numbers (2 digit per char)
def convert_to_num(char_padded_file, converted_num_file, modulus): #convert chars to digits
    pad_file = open(char_padded_file, "r")
    convert_file = open(converted_num_file, "a")
    convert_file.truncate(0)     # clear write to file
    plain_txt_chars_per_block = 2
    whitespace = 0

    while True:
        whitespace += 1
        char = pad_file.read(1) # read one char at the time
        if not char: # end of file --> break loop
            break
        char_to_num = ord(char) - 28 # minus 28, to get all numbers to become two digit - cons: eliminate some ascii characters
        num_to_string = str(char_to_num)
        if char_to_num <= 9 and char_to_num >= 0:
            num_to_string = "0" + num_to_string
        convert_file.write(num_to_string)
        if (whitespace % plain_txt_chars_per_block == 0):
            convert_file.write(" ")
    pad_file.close()
    convert_file.close()

# 3. convert blocks of digits to padded text
def convert_to_char(convert_char_file, decrypted_num_file):
```

```
    convert_to = open(convert_char_file, "w")
    convert_from = open(decrypted_num_file, "r")

    plain_txt_chars_per_block = 2
    whitespace = " "
    counter = 0
    number = ""

    while True:
        counter += 1
        char = convert_from.read(1) # read one char at the time
        if not char: # end of file --> break loop
            break
        elif char == whitespace: # reduce counter to correct counts ascii chars - remember whitespace is a char
            counter -= 1
        else:
            number += char #store number before converting to ascii

        if (counter % plain_txt_chars_per_block == 0 and (number != "" or number != '') ): # write to file - you have a number correspodning to a ascii char
            number = number.lstrip('0')
            number = int(number)
            number = number + 28 # convert back to ascii --> therefore + 28
            number = chr(number)
            convert_to.write(number)
            number = ""

    convert_to.close()
    convert_from.close()

# 4. unpadding chars
def unpad_txt(plain_text, input_padded_text):
    plain_txt = open(plain_text, "w")
    padded_txt = open(input_padded_text, "r")
    pad_msg = padded_txt.read()
    plain_msg = pad_msg.rstrip("X")
    plain_txt.write(plain_msg)
```

7.2 Step-wise results of execution 1

7.2.1 Alice and Bob's parameters:

Name	Type	Size	Value
alice_d	int	1	1350439543
alice_e	int	1	6485941127
alice_n	int	1	8171195509
alice_p	int	1	90641
alice_q	int	1	90149
bob_d	int	1	1899877137
bob_e	int	1	971927185
bob_n	int	1	8209149061
bob_p	int	1	90263
bob_q	int	1	90947

7.2.2 Bob sends and encrypts message to Alice:

Bob's plain text

```
Alice , does this RSA encryption work?
```

Bob's padded plain text

```
Alice , does this RSA encryption work?XXX
```

Bob's plain text converted to numbers

```
3780 7771 7316 0472 8373 8704 8876 7787 0454 5537 0473
    8271 8693 8488 7783 8204 9183 8679 3560 6060
```

Bob's encryption to cipher text

```
414036696 1523330796 2173987070 595192032 879752267
    398612489 5646173841 5379662166 6283838990 3464136775
    7654978218 5714424823 345558973 5809187849 210625534
    1875123913 7409654159 4760539384 6829424164 5450729414
```

7.2.3 Alice's decrypts message from Bob:

Alice decrypts Bob's cipher text to original numbers

```
3780 7771 7316 0472 8373 8704 8876 7787 0454 5537 0473
    8271 8693 8488 7783 8204 9183 8679 3560 6060
```

Alice converts the numbers into padded plain text

```
Alice, does this RSA encryption work?XXX
```

Alice has decrypted Bob's message

```
Alice, does this RSA encryption work?
```

7.2.4 Alice's reply and encrypts message to Bob:

Alice's plain text

```
yes Bob, the RSA algorithm works for as a crypto
    algorithm
```

Alice's padded plain text

```
yes Bob, the RSA algorithm works for as a crypto
    algorithmXXXXXX
```

Alice's plain text converted to number

```
9373 8704 3883 7016 0488 7673 0454 5537 0469 8075 8386
    7788 7681 0491 8386 7987 0474 8386 0469 8704 6904 7186
     9384 8883 0469 8075 8386 7788 7681 6060 6060 6060
```

Alice's encryption to cipher text

```
1856030455 5445326412 6935957528 147131935 4491656650
    288887007 6971520128 2359593732 1311409484 2866700008
    6334045738 1175028536 3291362151 1828857742 6334045738
     2039550620 6470603201 6334045738 1311409484
    5445326412 4889319957 2115599534 2721853053 3067268654
     1311409484 2866700008 6334045738 1175028536
    3291362151 5423913716 5423913716 5423913716
```

7.2.5 Bob's decrypts message from Alice:

Alice decrypts Bob's cipher text to original numbers

```
9373 8704 3883 7016 0488 7673 0454 5537 0469 8075 8386
    7788 7681 0491 8386 7987 0474 8386 0469 8704 6904 7186
     9384 8883 0469 8075 8386 7788 7681 6060 6060 6060
```

Bob converts the numbers into padded plain text

```
yes Bob, the RSA algorithm works for as a crypto algorithmXXXXXX
```

Bob has decrypted Alice's message into plain text

```
yes Bob, the RSA algorithm works for as a crypto algorithm
```