Laboratory 4

Tiutiu Natan-Gabriel,937

Introduction

I will implement an algorithm (RSA) which will perform encryption for a plaintext and decryption for a ciphertext, based on some pre-generated keys. I also defined some functions for computing binary representaion(as a list) of a decimal number, for performing repeated squaring modular exponentiation algorithm (we will use rsme as a shortcut), for computing gcd using euclidean algorithm,for computing the modular inverse, using extended euclidean algorithm and for testing whether a number is prime (basic version - we just test whether the number is a multiple of the first prime numbers and the better one - in which we use miller rabin test)

In order to generate the documentation written using markdown format , run **pandoc -t latex -o main.pdf main.nw** .

In order to generate the source code file, run **notangle main.nw>main.py** .

Once you have the source code file, you just have to run the python source file, using **python main.py** . If you want to change the code you run, just modify the content of the main function. For example, you can encrypt and decrypt the message represented as a string, or if you uncomment the second line from the main function, you can encrypt and decrypt a message from a file (first of all please create a file in which you have just lowercase letters from the English alphabet and the blank (empty space)).

Repeated Squaring Modular Exponentiation & helper functions

You can find more details about the following functions in my previous projects.

```
<<generate_binary_number>>=
def generate_binary_number(n):
    if n == 0:
        return [0]
    l = []
    while n > 0:
        l.append(n % 2)
        n = n // 2
    return l
@

<<rsme>>=
def rsme(b, k, n):
    a = 1
    if (k == 0):
        return a
    c = b
    l = generate_binary_number(k)
    if l[0] == 1:
        a = b
    for i in range(1, len(l)):
        c = c * c % n
        if l[i] == 1:
            a = c * a % n
    return a
@
```

Euclidean algorithm

You can find more details about the following function in my previous projects.

```
<<euclidean>>=
def euclidean(a, b):
    if b>0:
        return euclidean(b, a % b)
    return a
@
```

Extended euclidean algorithm

Description:

I will use 3 arrays in this explanation,because I think that using arrays is somehow easier to unterstand this algorithm.

We can observe that the array defined as follows respects the definition of r (the variable r used in extended_euclidean function) :

r_0= a

r_1= b

r_2= r_0 % r_1 (1)

....

r_i= r_{i-2} % r_{i-1}

Let q_i be the quotient of r_{i-2} / r_{i-1} $\implies$ r_{i-2} = q_i * r_{i-1} + r_i

Hence, we have that:

r_i= r_{i-2} - q_i * r_{i-1} (*)

We will prove by induction that : P(k): r_k= s_k * a + t_k * b , where s_k= s_{k-2} - q_k * s_{k-1} and t_k= t_{k-2} - q_k * t_{k-1} , is true $\forall b \geq 2$, where s_0= 1, s_1= 0 and t_0= 0, t_1= 1

P(2):

s_2= s_0 - q_2 * s_1 = 1 - 0 = 1

t_2= t_0 - q_2 * t_1 = 0 - q_2 = - a/b

r_2 = a % b (using (1)) (2)

s_2 * a + t_2 * b = 1 * a - (a/b) * b = a - q*b = a % b (3)

Using (2) and (3) => P(2) is true

P(k) $\implies$ P(k+1)

P(k+1): r_{k+1}= s_{k+1} * a + t_{k+1} * b , where s_{k+1}= s_{k-1} - q_{k+1} * s_k and t_{k+1}= t_{k-1} - q_{k+1} * t_k

So, we have r_{k+1} = s_{k+1} * a + t_{k+1} * b = (s_{k-1} - q_{k+1} * s_k) * a + (t_{k-1} - q_{k+1} * t_k) * b = s_{k-1} * a - q_{k+1} * s_k * a + t_{k-1} * b - q_{k+1} * t_k * b = (s_{k-1} * a + t_{k-1} * b) - q_{k+1} * (s_k * a + t_k * b) = r_{k-1} - q_{k+1} * r_k (so we reached the definition of r_{k+1}, as you can see in (*)) $\implies$ P(k+1) is true

So,we have r_k= s_k * a + t_k * b , where s_k= s_{k-2} - q_k * s_{k-1} and t_k= t_{k-2} - q_k * t_{k-1} , $\forall b \geq 2$.

Once r_k becomes 0, the algorithm will stop. Then we will say that gcd(a,b)= r_{k-1} and gcd(a,b)= s_{k-1} * a + t_{k-1} * b .

In our program we just keep the last 2 values, not the entire arrays. We denoted u=s and v=t.

This is the algorithm:

Input: a, b $\in$ **N** , a,b $\leq$ n, a $\geq$ b.

Output: d=gcd(a,b) and u,v $\in$ **Z** such that au + bv = d.

Algorithm:

u2=1; u1=0; v2=0; v1=1;

while b>0 do:

q= [a / b]; r= a - q*b; u= u2 - q*u1; v= v2 - q*v1;

a= b; b= r; u2= u1; u1= u; v2= v1; v1= v;

d= a; u= u2; v= v2;

write(d, u, v)

```
<<extended_euclidean>>=
def extended_euclidean(a, b):
    u2 = 1
    u1 = 0
    v2 = 0
    v1 = 1
    while b > 0:
        q = a // b
        r = a - q * b
        u = u2 - q * u1
        v = v2 - q * v1

        a = b
        b = r
        u2 = u1
        u1 = u
        v2 = v1
        v1 = v
    d = a
    u = u2
    v = v2
    return d, u, v
@
```

Miller-Rabin test

The Miller-Rabin test is widely used in practice for RSA, so I decided to use it in my RSA implementation.

Description:

Extract the square roots from the previous congruence successively, that is, raise b to $\frac{n-1}{2}$, $\frac{n-1}{4}$, . . . , $\frac{n-1}{2^s}$, s , where t = $\frac{n-1}{2^s}$ is odd . Then the first result different of 1 has to be -1 if n is prime, because 1 and -1 are the only square roots modulo a prime of 1.

Remarks:

If the algorithm gives the answer composite(that is, False in our case), then this is composite for sure. If the algorithm gives the answer PRIME(that is, True in our case), then the probability of correct answer is 1 - $\frac{1}{4^k}$, where k is the number of repetitions. In my implementation I use k = 50 (you can change it in the function generate_large_prime in the second if branch) . The probability that the Miller-Rabin test gives the wrong answer is (at most):

$$\frac{1}{4^{50}} = \frac{1}{1267650600228229401496703205376}$$

So, this is much less than the probability to get the wrong answer due to an hardware error.

The function miller_rabin_test_wrapper compute s and t for a given n,where n - 1 = 2^s * t. We call the function miller_rabin_test at most k times. If the result is False, then we return False as well. Else,if we did not get any False, we will return True (after k iterations), that is, n may be prime.

This is the algorithm:

```
Write n - 1 = $2^s$ * t, where t is odd.

while k>0 do

  if miller_rabin_test(n, s, t)=False:

    return False

  k=k-1

return True
```

```
<<miller_rabin_test_wrapper>>=
def miller_rabin_test_wrapper(n, k):
    t = n - 1
    s = 0
    while t % 2 == 0:
        t = t // 2
        s += 1
    while k > 0:
        result = miller_rabin_test(n, s, t)
        if not result:
            return False  # the result is composite
        k -= 1
    return True  # the result may be prime
@
```

The function miller_rabin_test represents the main part of the test. This is the algorithm :

Choose (randomly) 1 < a < n

Compute (by the repeated squaring modular exponentiation) the following sequence (modulo n):

$$a^t, a^{2t}, a^{2^2 t}, ..., a^{2^s t}$$

If either the first number in the sequence is 1 or if one gets the value 1 and its previous number -1 (that is n-1), return True (that is, n may be prime) ,else return False

```
<<miller_rabin_test>>=
def miller_rabin_test(n, s, t):
    a = secrets.randbelow(n - 2) + 2
    # now let's compute the sequence
    sequence = []
    a_t = rsme(a, t, n)
    sequence.append(a_t)
    for i in range(1, s + 1):
        a_t = a_t * a_t % n
        sequence.append(a_t)

    if sequence[0] == 1:
        return True
    for i in range(1, len(sequence)):
        if sequence[i] == 1:
            if sequence[i - 1] == n - 1:
                return True
            else:
                return False
    return False
@
```

Key generation (public key & private key)

In the following section I will present the functions I use for the key generation: trivial__primality__check, generate__large__prime__wrapper, generate__large__prime and generate__key.

Of course, miller__rabin__test__wrapper and miller__rabin__test presented in the previous section are involved in the key generation, but I tought that miller-rabin test deserves a separate section.

The function trivial__primality__check just checks whether the number is a multiple of the first prime numbers (prime numbers less than 20 in our case). This function helps us to avoid running miller-rabin tests for trivial composite numbers.

This is the algorithm:

```
for i in [2, 3, 5, 7, 11, 13, 17, 19] do
  if i divides number :
    return False
return True
```

```
<<trivial_primality_check>>=
def trivial_primality_check(number):
    for i in [2, 3, 5, 7, 11, 13, 17, 19]:
        if number % i == 0:
            return False
    return True
@
```

The function generate_large_prime_wrapper computes 2^{order}. Because we use this wrapper (helper function), we will not have to compute this large number (2^{order}) at each recursive call in generate_large_prime, but just once.

The function generate_large_prime will generate a random prime number between 2^{order}+1 and $2^{order+1}$-1. The parameter order represents the number of bits of the number to be generated. We have that secrets.randbelow(2**order - 1) generates a natural number from interval $[0,2^{order} - 1)$ $\Longrightarrow$ secrets.randbelow(2**order - 1) + 2**order + 1 generates a natural number from interval $[2^{order} + 1,2^{order} + 2^{order})$ $\Longrightarrow$ a natural number from interval $[2^{order} + 1,2^{order+1} -1]$. In generate_large_prime we have: number = 2^{order} (as it was computed in generate_large_prime_wrapper) .

```
<<generate_large_prime_wrapper>>=
def generate_large_prime_wrapper(order):
    number = 2 ** order
    return generate_large_prime(number)
@
```

The algorithm for generate_large_prime function is :

```
Generate a random number from $[2^{order} + 1,2^{order+1} -1]$ .
if trivial_primality_check(random_number)=False
  return generate_large_prime(number)
if miller_rabin_test_wrapper(random_number,50)=False
  return generate_large_prime(number)
return random_number
```

```
<<generate_large_prime>>=
def generate_large_prime(number):
    random_number = secrets.randbelow(number - 1) + number + 1
    if not trivial_primality_check(random_number):
        return generate_large_prime(number)
    if not miller_rabin_test_wrapper(random_number, 50):
        return generate_large_prime(number)
    return random_number
@
```

The generate_key function generates a public and a private key (in a given interval - defined by the parameter order).

This is the algorithm:

Generates 2 random large distinct primes p, q of approximately same size (size is defined by the parameter order)

Computes n = pq and φ(n) = (p - 1)(q - 1) (the Euler function).

Randomly selects 1 < e < φ(n) with gcd(e, φ(n)) = 1

Computes d =e^{-1} mod φ(n).

The public key is K_E = (n, e); the private key is K_D = d

For generating a random number e, 1 < e < φ(n) we use secrets.randbelow(phi_n-2) + 2: secrets.randbelow(phi_n-2) generates a random natural number in range [0,φ(n) - 2) $\implies$ secrets.randbelow(phi_n-2) + 2 generates a random natural number in range [2,φ(n)) ,that is interval (1,φ(n))

```
<<generate_key>>=
def generate_key(order):
    p = generate_large_prime_wrapper(order)
    # print("p: ", p)
    q = generate_large_prime_wrapper(order)
    # sprint("q: ", q)
    while p == q:
        q = generate_large_prime_wrapper(order)
        print("q: ", q)
    n = p * q
    phi_n = (p - 1) * (q - 1)
    # secrets.randbelow(phi_n-2) generates a random in range [0,phi_n-2),then
    # secrets.randbelow(phi_n-2) + 2 generates a random in range [2,phi_n) ,that is(1,phi_n
    e = secrets.randbelow(phi_n - 2) + 2
    while euclidean(e, phi_n) != 1:
        e = secrets.randbelow(phi_n - 2) + 2
    _, d, _ = extended_euclidean(e, phi_n)
    d = (d + phi_n) % phi_n
    # (n,e) is public key and d is private
    return n, e, d
@
```

Alphabet

Because we will discuss about encryption and decryption in the following 2 sections, I think that this is the best time to introduce 2 dictionaries: alphabet and numbers.

We use a 27-letters alphabet for plaintext and ciphertext: the blank(" “) with numerical equivalent 0 and letters a - z (the English alphabet) with numerical equivalents 1-26 .

The dictionary numbers is obtained by inverting the dictionary alphabet.

```
<<alphabet>>=

alphabet = {" ": 0, "a": 1, "b": 2, "c": 3, "d": 4, "e": 5, "f": 6, "g": 7, "h": 8, "i": 9,
            "m": 13, "n": 14, "o": 15, "p": 16, "q": 17, "r": 18, "s": 19, "t": 20, "u": 21,
            "y": 25, "z": 26}

numbers = {0: " ", 1: "a", 2: "b", 3: "c", 4: "d", 5: "e", 6: "f", 7: "g", 8: "h", 9: "i", 1
           13: "m", 14: "n", 15: "o", 16: "p", 17: "q", 18: "r", 19: "s", 20: "t", 21: "u",
           25: "y", 26: "z"}

@
```

Encryption

Let us start with 2 helper functions: compute_numerical_equivalent and compute_literal_equivalent . They are doing what the names suggest.

Exapmle for compute_numerical_equivalent:

al -> $1 * 27 + 12 = 39 \implies$ numerical equivalent for “al” is 39.

The algorithm for compute_numerical_equivalent:

```
Input: a sequence of characters (that is, a text)
Output: numerical equivalent for a sequence of characters (that is, a text)
numerical_equivalent = 0
for each character in text do:
   numerical_equivalent = numerical_equivalent * 27 + alphabet[i]
return numerical_equivalent
```

```
<<compute_numerical_equivalent>>=
def compute_numerical_equivalent(text):
    numerical_equivalent = 0
    for i in text:
        numerical_equivalent = numerical_equivalent * 27 + alphabet[i]
    return numerical_equivalent
@
```

Exapmle for compute_literal_equivalent:

1428 = 1 * 27^2 + 25 * 27 + 24 -> AYX $\implies$ literal equivalent for 1428 is "AYX".

The algorithm for compute_literal_equivalent:

```
Input: number, iterations
Output: literal equivalent for a numerical sequence
literal_equivalent = ""
while iterations > 0 do
   literal_equivalent = numbers[number modulo 27] + literal_equivalent
   number = number / 27 (keep the integer part in result)
   iterations = iterations - 1
return literal_equivalent
```

```
<<compute_literal_equivalent>>=
def compute_literal_equivalent(number, iterations):
    literal_equivalent = ""
    while iterations > 0:
        literal_equivalent = numbers[number % 27] + literal_equivalent
        number = number // 27
        iterations -= 1
    return literal_equivalent
@
```

The algorithm for encrypt function is:

Input: plaintext, n , e (n and e form the public key, which is needed for encryption) , k (plaintext message units are blocks of k letters) , l (ciphertext message units are blocks of l letters)

Output: ciphertext

```
Check whether the constraint $27^k$ < n < $27^l$ holds
ciphertext = ""
Complete the plaintext with blanks (that is numbers[0]), if necessary
for each block of k characters in plaintext do
   compute m = numerical equivalent of the block
```

```
    compute c = $m^e$ mod n
    add to ciphertext the literal equivalent of c
return ciphertext
```

```
<<encrypt>>=
def encrypt(plaintext, n, e, k, l):
    if 27 ** k >= n or n >= 27 ** l:
        return "Please choose some appropriate values for k and l"

    ciphertext = ""
    while len(plaintext) % k != 0:
        plaintext += numbers[0]

    for i in range(0, len(plaintext) // k):
        numerical_equivalent = compute_numerical_equivalent(
            plaintext[k * i:k * (i + 1)])
        encrypted_number = rsme(numerical_equivalent, e, n)
        literal_equivalent = compute_literal_equivalent(encrypted_number, l)
        ciphertext = ciphertext + literal_equivalent

    return ciphertext
@
```

Decryption

The algorithm for decrypt function is:

```
Input: ciphertext, n , d (private key, which is needed for decryption) , k (plaintext message units are blocks of k letters) , l (ciphertext message units are blocks of l letters)
Output: plaintext
Check whether the constraint $27^k < n < 27^l$ holds
plaintext = “”
for each block of l characters in ciphertext do
    compute m = numerical equivalent of the block
    compute c = $m^d$ mod n
    add to plaintext the literal equivalent of c
remove trailing spaces from the plaintext
return plaintext
```

Remark

In the decrypt function we will not have any case in which we have to complete the ciphertext with blanks, because the encrypt function always returns a ciphertext of length multiple of “l”

```
<<decrypt>>=

def decrypt(ciphertext, n, d, k, l):
    if 27 ** k >= n or n >= 27 ** l:
        return "Please choose some appropriate values for k and l"
    plaintext = ""
    for i in range(0, len(ciphertext) // l):
        numerical_equivalent = compute_numerical_equivalent(ciphertext[l * i:l * (
                    i + 1)])
        decrypted_number = rsme(numerical_equivalent, d, n)
        literal_equivalent = compute_literal_equivalent(decrypted_number, k)
        plaintext = plaintext + literal_equivalent

    for i in range(len(plaintext) - 1, -1, -1):
        if plaintext[i] == numbers[0]:
            plaintext = plaintext[:-1]
        else:
            break

    return plaintext
@
```

RSA function

This function simply combines the functions defined above. It requires a message as paramater. You can also add some optional parameters:

1. order - p and q will be generated in the interval [2^{order} + 1,$2^{order+1}$ -1],so n will be approximately in the interval [$2^{2*order}$, $2^{2*order+2}$]
2. k - plaintext message units will be blocks of k letters
3. l - ciphertext message units will be blocks of l letters

The default order is 512, so n will have 1024 bits,so the security level will be 80 (An algorithm is said to have a security level of n bit if the best known attack requires 2^n steps).

The largest number factored (maybe not up-to-date): a 768-bit number with 232 decimal digits, announced on December 12, 2009, using hundreds of machines over two years

So, we are pretty safe using the default value for order.

If you do not provide some values for k and l, these values will be randomly generated such that the constraint 27^k < n < 27^l will hold .

We have that lower_bound is a little bit SMALLER than log_{27} 2. Also, upper_bound is a little bit LARGER than log_{27} 2.

Let us check whether 27^k < n < 27^l will hold , where the maximum value of k is int(2 * order * lower_bound) and the minimum value of l is int(2 * (order + 1) * upper_bound) + 1 (you can see the random generation of this values below, in the RSA function).

We have $27^{max(k)} = 27^{int(2*order*lower_bound)} \leq 27^{2*order*lower_bound} = (27^{lower\_bound})^{2*order} < (27^{log_{27} 2})^{2*order} = 2^{2*order} <$ n , because p and q are from interval [2^{order} + 1,$2^{order+1}$ -1] and n = pq

We have $27^{min(l)} = 27^{int(2*(order+1)*upper_bound)+1} > 27^{2*(order+1)*upper_bound} = (27^{upper\_bound})^{2*(order+1)} > (27^{log_{27} 2})^{2*(order+1)} = 2^{2*(order+1)} >$ n , because p and q are from interval [2^{order} + 1,$2^{order+1}$ -1] and n = pq

$\Longrightarrow$ it's safe to use the default values (auto-generated values) for k and l.

```
<<RSA>>=
def RSA(message, order=512, k=-1, l=-1):
    n, e, d = generate_key(order)
    lower_bound = 0.21030991785714
    upper_bound = 0.21030991785716
    if k == -1 or l == -1:
        # k = int(2 * order * lower_bound)
        # l = int(2 * (order + 1) * upper_bound) + 1
        k = random.randrange(2, int(2 * order * lower_bound)+1)
        aux = int(2 * (order + 1) * upper_bound) + 1
        l = random.randrange(aux, aux*4)
    print("Message to be encrypted: ", message)
    encrypted_message = encrypt(message, n, e, k, l)
    print("encrypted_message: ", encrypted_message)
    decrypted_message = decrypt(encrypted_message, n, d, k, l)
    print("decrypted_message: ", decrypted_message)

@
```

The function RSA_using_file is similar to RSA. The differences are: we read the message from a file and we write the ciphertext and plaintext obtained after decyption in 2 files, each of them having an additional extension. The additional extensions are ".encrypted" and ".decrypted" .

```
<<RSA_using_file>>=
def RSA_using_file(file_name, order=512, k=-1, l=-1):
    n, e, d = generate_key(order)
    lower_bound = 0.21030991785714
    upper_bound = 0.21030991785716
    if k == -1 or l == -1:
        # k = int(2 * order * lower_bound)
        # l = int(2 * (order + 1) * upper_bound) + 1
        k = random.randrange(2, int(2 * order * lower_bound)+1)
        aux = int(2 * (order + 1) * upper_bound) + 1
        l = random.randrange(aux, aux*4)

    f = open(file_name, "r")
    message = f.read()
    f.close()

    print("Message to be encrypted: ", message)


    encrypted_message = encrypt(message, n, e, k, l)
    encrypted_file = file_name+".encrypted"
    f = open(encrypted_file, "w")
    f.write(encrypted_message)
    f.close()
    print("encrypted_message: ", encrypted_message)



    decrypted_message = decrypt(encrypted_message, n, d, k, l)
    decrypted_file = file_name + ".decrypted"
    f = open(decrypted_file, "w")
    f.write(decrypted_message)
    f.close()
    print("decrypted_message: ", decrypted_message)
@

<<main>>=
def main():
    RSA("the best time to visit cancun is from december to april during the peak season")
    # RSA_using_file("message.txt")
@
```

Below we have some tests. We test function miller_rabin_test_wrapper with some basic values and also with some Mersenne primes. We also test it with some numbers having the form 2^n - 1,but which are not Mersenne primes. We use 50 repetitions for the tests, so they should not give a wrong result.

We also test other helper functions,such as extended_euclidean, euclidean and rsme functions .

Of course,we also test the encrypt and decrypt functions, testing whether the initial message is equal to the message obtained after decryption. (we run 20 tests for encrypt and decrypt functions)

```
<<tests>>=
def tests():
    assert miller_rabin_test_wrapper(101, 50)
    assert not miller_rabin_test_wrapper(123, 50)

    # testing miller_rabin_test_wrapper function (with 50 iterations)
    for i in [17, 19,31,61,89,107]:
        assert miller_rabin_test_wrapper(2**i - 1,50)
    for i in [21,29,49,80,99,123]:
        assert not miller_rabin_test_wrapper(2**i - 1, 50)

    # testing the extended_euclidean and euclidean functions
    for i in range(0, 20):
        a = random.randrange(10, 1000)
        b = random.randrange(10, 1000)
        l = extended_euclidean(a, b)
        assert a * l[1] + b * l[2] == euclidean(a, b)

    # testing the rsme function, which computes a^b mod n using repeated squaring modular e
    assert rsme(16, 10, 11) == pow(16, 10, 11)
    assert rsme(116, 107, 211) == pow(116, 107, 211)
    assert rsme(145, 129, 199) == pow(145, 129, 199)
    for i in range(0, 20):
        a = random.randrange(10, 1000)
        b = random.randrange(10, 1000)
        n = random.randrange(10, 1000)
        assert rsme(a, b, n) == pow(a, b, n)

    # testing the encrypt and decrypt functions

    for i in range(0, 20):
        # 27^ k >= n or n >= 27^ l
        message_length = random.randrange(10, 1000)
        characters = list(alphabet.keys())
        message = ''.join([random.choice(characters) for n in range(message_length)])
        # remove trailing spaces
        for i in range(len(message) - 1, -1, -1):
            if message[i] == numbers[0]:
                message = message[:-1]
            else:
                break
        order = 128
        n, e, d = generate_key(order)
        #lower bound is a little bit SMALLER than log 27 2
        # 27 ** k <= n and n <= 27 ** l
        # =>k * log 2 27 <= log 2 n and n <= 27 ** l and as we choose
        # n in interval 2^(order)+1,2^(order+1)-1 => k * log 2 27 <= log 2 n <= order   =>
        # it's safe to take k = (log 2 27)^(-1) * log 2 n = log 27 2 * log 2 n
        lower_bound = 0.21030991785714
        # lower bound is a little bit LARGER than log 27 2
        upper_bound = 0.21030991785716

        k = random.randrange(1, int(2 * order * lower_bound)+1)
        # aux is lower bound for l
        aux = int(2 * (order + 1) * upper_bound) + 1
        l = random.randrange(aux, aux*4)

        # print("Message to be encrypted: ", message)
        encrypted_message = encrypt(message, n, e, k, l)
        # print("encrypted_message: ", encrypted_message)
        decrypted_message = decrypt(encrypted_message, n, d, k, l)
        # print("message: ",message)
        # print("decrypted_message: ", decrypted_message)
        assert message == decrypted_message
    print("ALL TESTS PASSED")
@
```

Remarks

It is very import to set the recursion limit at least 2000 (using sys.setrecursionlimit(2000)), because otherwise we could get an error using order > 512 (i.e. 'RecursionError: maximum recursion depth exceeded in comparison').

Also,you can see that I used secrets module for generating random numbers in my program . I used this module whenever I had to generate a cryptographically strong random number. As the documentation states, secrets should be used in preference to the default pseudo-random number generator in the random module, which is designed for modelling and simulation, not security or cryptography. You can find more about this library here: https://docs.python.org/3/library/secrets.html .

```
<<*>>=
import secrets
import sys
import random

sys.setrecursionlimit(2000)
```

```
<<generate_binary_number>>
<<rsme>>

<<euclidean>>
<<extended_euclidean>>


<<miller_rabin_test_wrapper>>
<<miller_rabin_test>>
<<trivial_primality_check>>
<<generate_large_prime_wrapper>>
<<generate_large_prime>>
<<generate_key>>

<<alphabet>>
<<compute_numerical_equivalent>>
<<compute_literal_equivalent>>
<<encrypt>>
<<decrypt>>
<<RSA>>
<<RSA_using_file>>


<<main>>
<<tests>>



tests()

main()

@
```