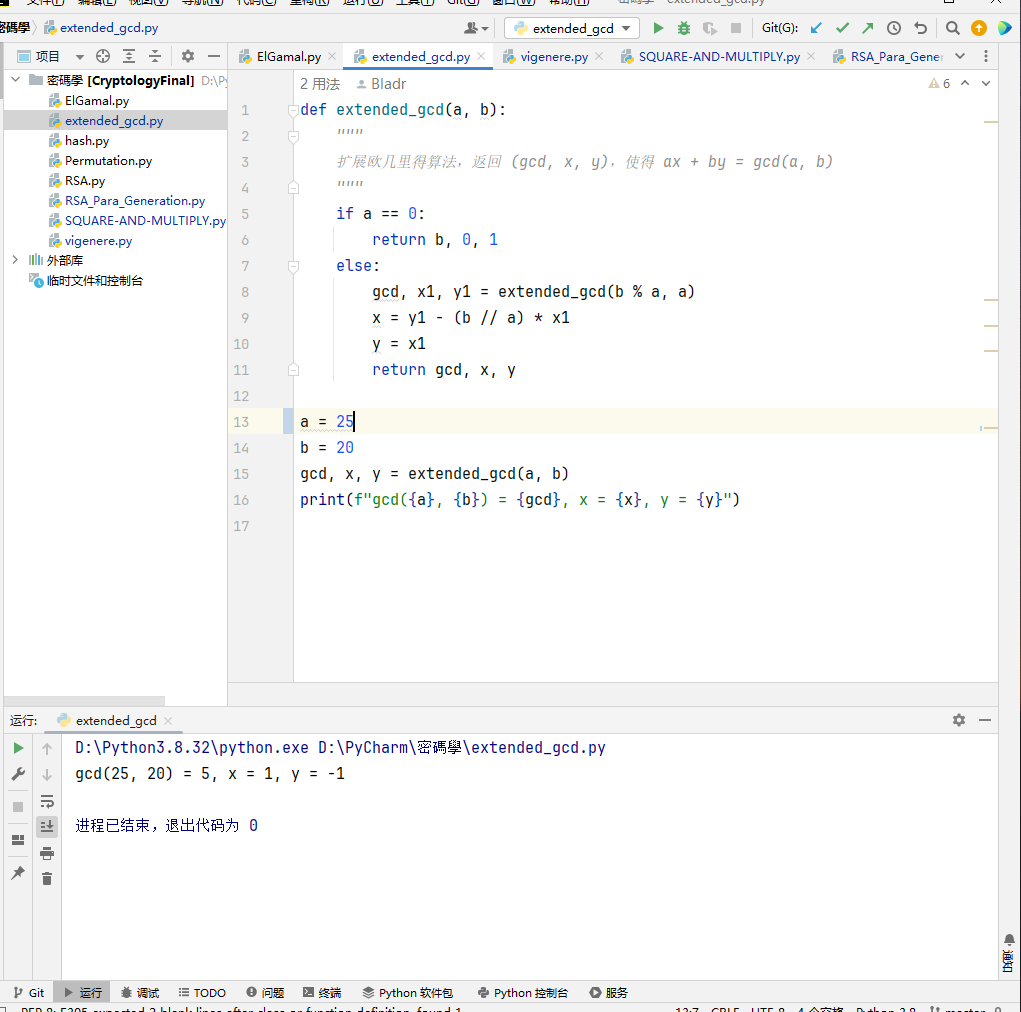
# Cryptography and Information Security Final Report

WangYihan 2230017050

**Algorithm 6.2: EXTENDED EUCLIDEAN ALGORITHM(a, b)**

|  |
| --- |
| def extended\_gcd(a, b):  if a == 0:  return b, 0, 1  else:  gcd, x1, y1 = extended\_gcd(b % a, a)  x = y1 - (b // a) \* x1  y = x1  return gcd, x, y |

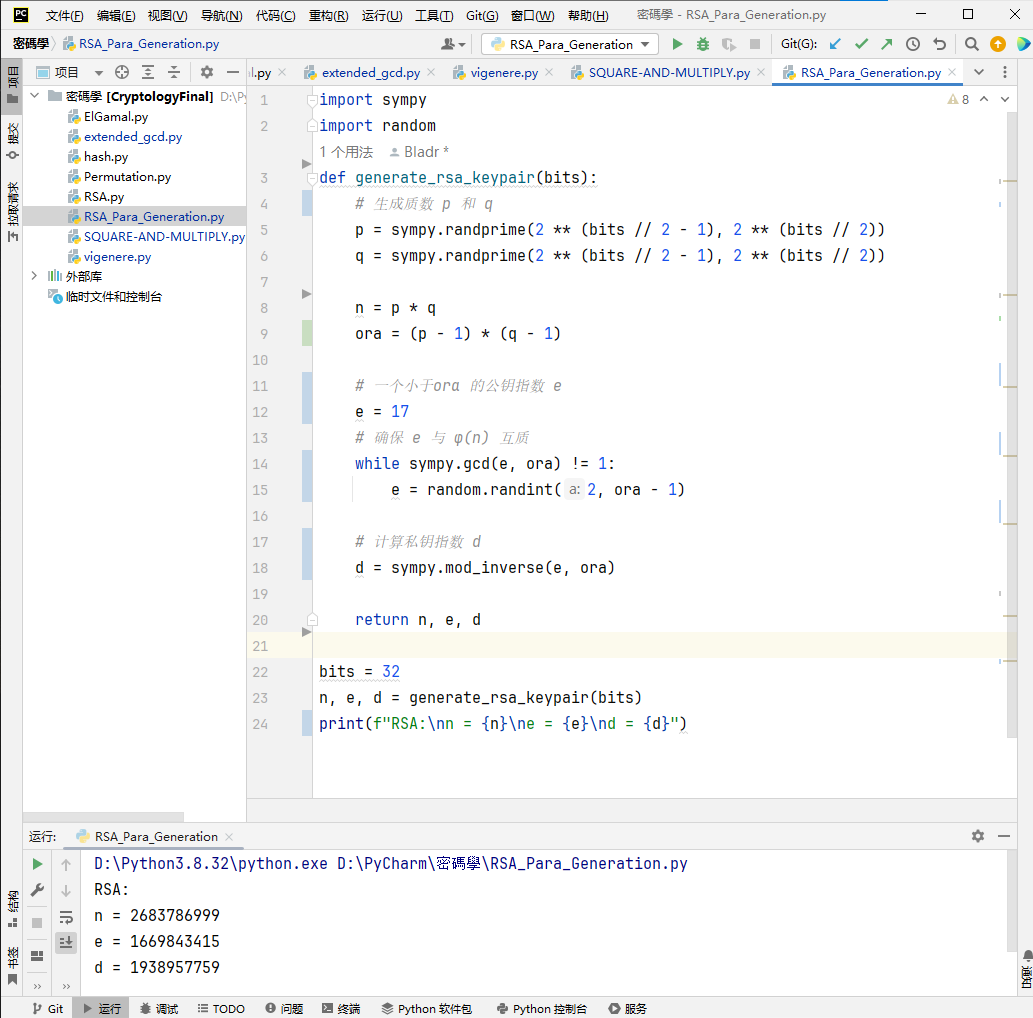
run result：



**Algorithm 6.4: RSA PARAMETER GENERATION**

|  |
| --- |
| import sympy  import random  def generate\_rsa\_keypair(bits):  # 生成质数 p 和 q  p = sympy.randprime(2 \*\* (bits // 2 - 1), 2 \*\* (bits // 2))  q = sympy.randprime(2 \*\* (bits // 2 - 1), 2 \*\* (bits // 2))  n = p \* q  ora = (p - 1) \* (q - 1)  e = 17  while sympy.gcd(e, ora) != 1:  e = random.randint(2, ora - 1)  d = sympy.mod\_inverse(e, ora)  return n, e, d |

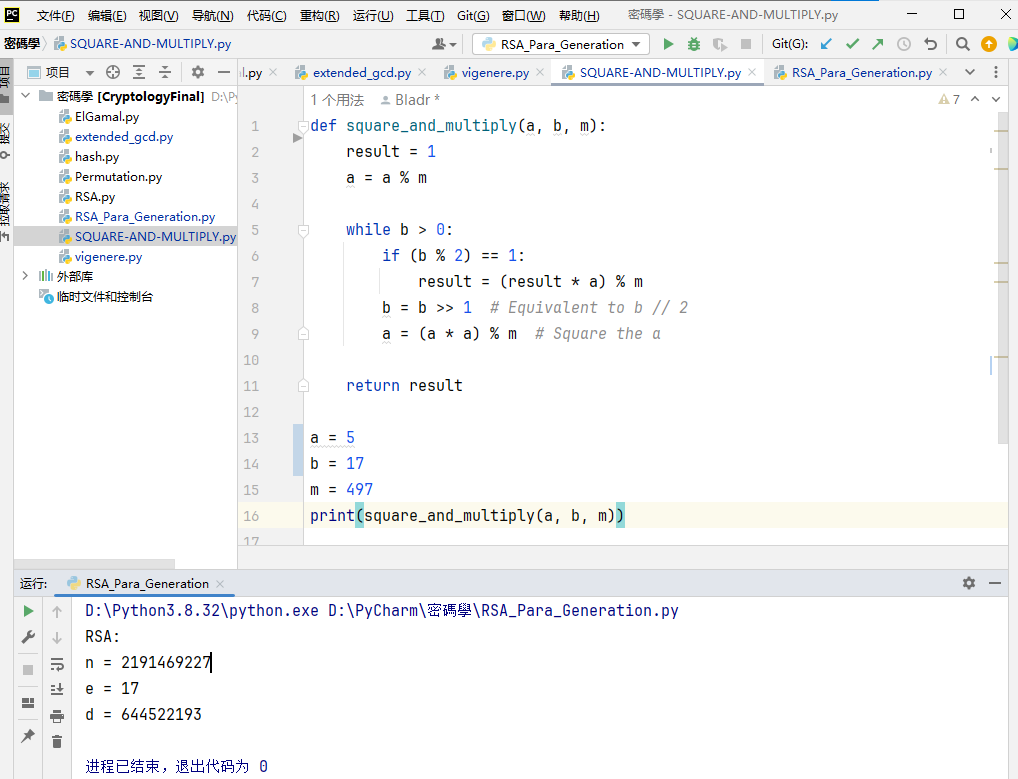
run result：



**Algorithm6.5: SQUARE-AND-MULTIPLY(x,c,n)**

|  |
| --- |
| def square\_and\_multiply(a, b, m):  result = 1  a = a % m  while b > 0:  if (b % 2) == 1:  result = (result \* a) % m  b = b >> 1 # Equivalent to b // 2  a = (a \* a) % m # Square the a  return result |

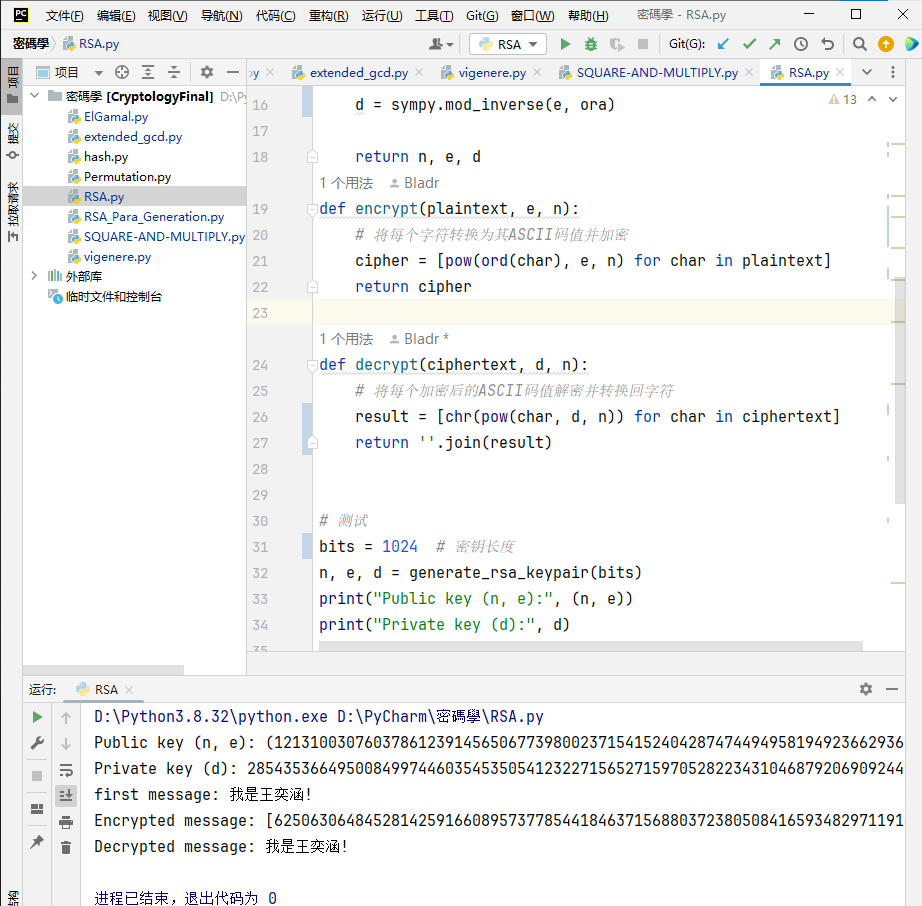
run result：



**Algorithm for Cryptosystem 6.1: RSA Cryptosystem**

|  |
| --- |
| import random  import sympy  def generate\_rsa\_keypair(bits):  p = sympy.randprime(2 \*\* (bits // 2 - 1), 2 \*\* (bits // 2))  q = sympy.randprime(2 \*\* (bits // 2 - 1), 2 \*\* (bits // 2))  n = p \* q  ora = (p - 1) \* (q - 1)  e = 17  while sympy.gcd(e, ora) != 1:  e = random.randint(2, ora - 1)  d = sympy.mod\_inverse(e, ora)  return n, e, d  def encrypt(plaintext, e, n):  cipher = [pow(ord(char), e, n) for char in plaintext]  return cipher  def decrypt(ciphertext, d, n):  result= [chr(pow(char, d, n)) for char in ciphertext]  return ''.join(result) |

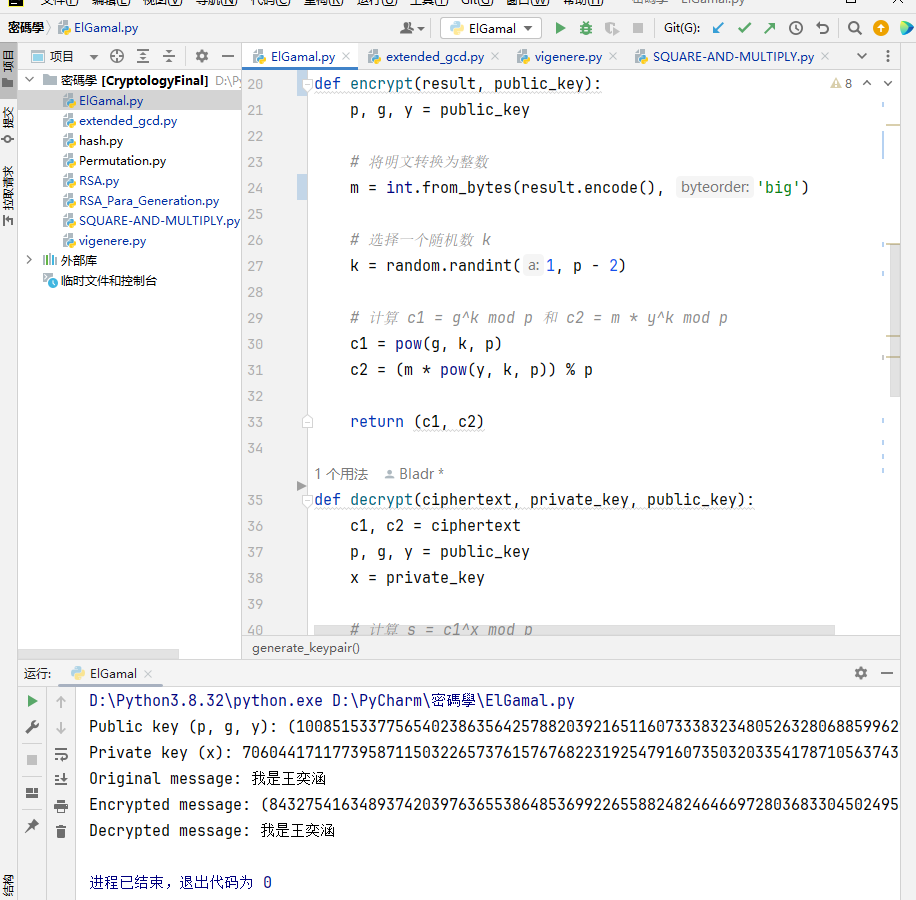
run result



**Algorithm for Cryptosystem 7.1: ElGamal Public-key Cryptosystem**

|  |
| --- |
| import random  import sympy  def generate\_keypair(bits):  while True:  p = random.getrandbits(bits)  if sympy.isprime(p):  break  g = random.randint(2, p - 1)  x = random.randint(1, p - 2)  y = pow(g, x, p)  return (p, g, y), x  def encrypt(result, public\_key):  p, g, y = public\_key  m = int.from\_bytes(result.encode(), 'big')  k = random.randint(1, p - 2)  c1 = pow(g, k, p)  c2 = (m \* pow(y, k, p)) % p  return (c1, c2)  def decrypt(ciphertext, private\_key, public\_key):  c1, c2 = ciphertext  p, g, y = public\_key  x = private\_key  s = pow(c1, x, p)  s\_inverse = sympy.mod\_inverse(s, p)  m = (c2 \* s\_inverse) % p  result = m.to\_bytes((m.bit\_length() + 7) // 8, 'big').decode()  return result |

run result：



**Algorithm for The Permutation Cipher**

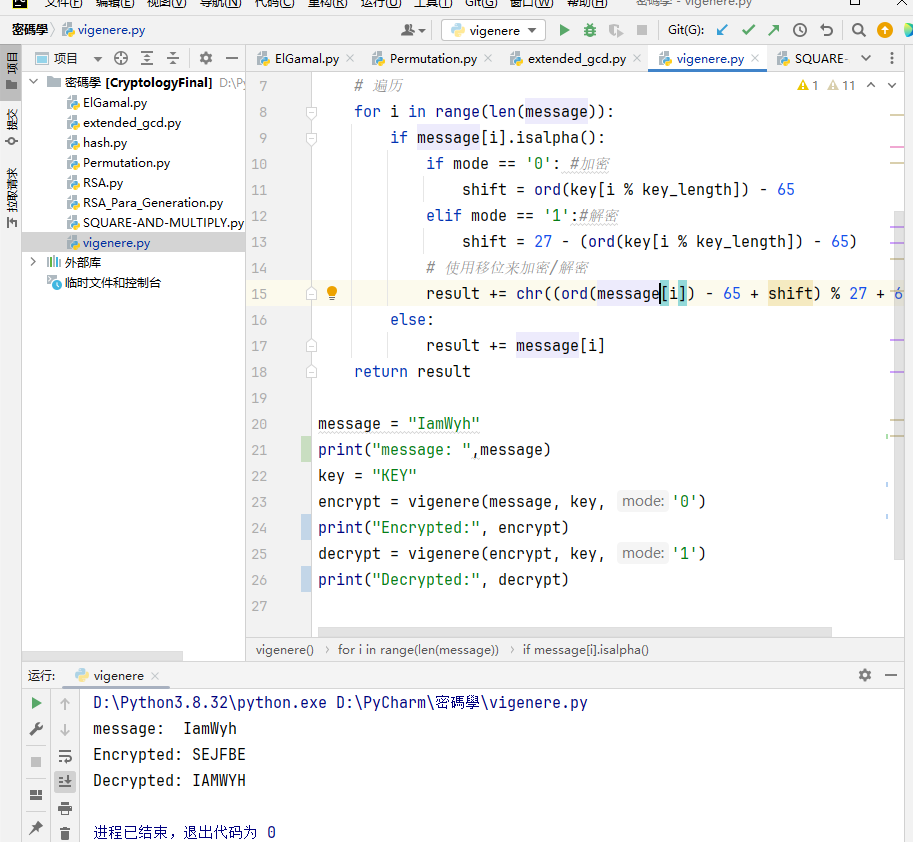
|  |
| --- |
| import random  def generate\_key(length):  key = list(range(length))  random.shuffle(key)  return key  def permutation\_encrypt(plaintext, key):  ciphertext = [''] \* len(plaintext)  for i, char in enumerate(plaintext):  ciphertext[key[i]] = char  return ''.join(ciphertext)  def permutation\_decrypt(ciphertext, key):  plaintext = [''] \* len(ciphertext)  for i, char in enumerate(ciphertext):  plaintext[key[i]] = char  return ''.join(plaintext) |

Run result：

**Algoritm for the vigenere cipher**

|  |
| --- |
| def vigenere(message, key, mode):  message = message.upper()  key = key.upper()  key\_length = len(key)  result = ''  for i in range(len(message)):  if message[i].isalpha():  if mode == '0':  shift = ord(key[i % key\_length]) - 65  elif mode == '1':  shift = 27 - (ord(key[i % key\_length]) - 65)  result += chr((ord(message[i]) - 65 + shift) % 27 + 65)  else:  result += message[i]  return result |

Run result:



**The application scenarios of Harsh function and MAC**

|  |
| --- |
|  |

Run result: