```
import random

'''
Euclid's algorithm for determining the greatest common
divisor
This implementation uses iteration to efficiently compute
the GCD for larger integers.
'''
def gcd(a, b):
    while b != 0:
        a, b = b, a % b  # Update a and b to b and the remainder
of a divided by b
    return a  # Return the GCD

'''
Euclid's extended algorithm for finding the multiplicative
inverse of two numbers.
This function computes the multiplicative inverse using
the Extended Euclidean Algorithm.
'''
def multiplicative_inverse(e, phi):
    d = 0  # Initialize d
    x1 = 0  # Initialize x1
    x2 = 1  # Initialize x2
    y1 = 1  # Initialize y1
    temp_phi = phi  # Store the original value of phi

    # Loop until e becomes 0
    while e > 0:
        temp1 = temp_phi // e  # Integer division
        temp2 = temp_phi - temp1 * e  # Remainder
        temp_phi = e  # Update temp_phi
        e = temp2  # Update e

        # Update x and y values
```

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

        # Shift x and y
        x2 = x1
        x1 = x
        d = y1
        y1 = y

    # If temp_phi is 1, return the positive value of d
    if temp_phi == 1:
        return d + phi  # Ensure d is positive

'''
Function to test if a number is prime.
This function checks for primality by testing divisibility.
'''
def is_prime(num):
    if num == 2:
        return True  # 2 is prime
    if num < 2 or num % 2 == 0:
        return False  # Exclude even numbers and numbers
less than 2
    # Check for factors from 3 to the square root of num
    for n in range(3, int(num**0.5) + 2, 2):
        if num % n == 0:
            return False  # Not prime if divisible by n
    return True  # Return true if no divisors found

'''
Function to generate RSA key pairs.
It takes two prime numbers p and q and returns the public
and private keys.
'''
def generate_key_pair(p, q):
```

```
    # Validate that both p and q are prime
    if not (is_prime(p) and is_prime(q)):
        raise ValueError('Both numbers must be prime.')
    elif p == q:
        raise ValueError('p and q cannot be equal')  # Ensure p
and q are distinct

    n = p * q  # Calculate n = p * q
    phi = (p - 1) * (q - 1)  # Calculate the totient 蠻(n)

    # Choose an integer e such that 1 < e < 蠻(n) and e is
coprime to 蠻(n)
    e = random.randrange(1, phi)
    g = gcd(e, phi)  # Check GCD of e and 蠻(n)
    while g != 1:  # Ensure e and 蠻(n) are coprime
        e = random.randrange(1, phi)
        g = gcd(e, phi)
    # Calculate the private key d using the multiplicative
inverse
    d = multiplicative_inverse(e, phi)

    # Return public key (e, n) and private key (d, n)
    return ((e, n), (d, n))

'''
Function to encrypt input_text using the public key.
The input_text is converted into ciphertext using modular
exponentiation.
'''
def encrypt(pk, input_text):
    key, n = pk  # Unpack the public key
    # Convert each character in the input_text to ciphertext
using a^b mod m
    ciphertext = [pow(ord(char), key, n) for char in input_text]
```

```
    return ciphertext  # Return the list of ciphertext integers

'''
Function to decrypt ciphertext using the private key.
The ciphertext is converted back into output_text.
'''
def decrypt(pk, ciphertext):
    key, n = pk  # Unpack the private key
    # Generate the output_text based on the ciphertext and
key using a^b mod m
    aux = [str(pow(char, key, n)) for char in ciphertext]  #
Convert ciphertext to string
    # Convert the integer values back to characters
    output_text = [chr(int(char2)) for char2 in aux]
    return ''.join(output_text)  # Return the decrypted
output_text as a string

if __name__ == '__main__':
    '''
    Main execution block to run the RSA encryption/
decryption process.
    This block is executed only if the script is run directly.
    '''

print("==========================================
==========================================
====================")
    print("==================================RSA
ALGORITHM==================================
==========")
    print(" ")

    # Input for prime numbers with validation
    while True:
        try:
```

```
            p = int(input(" - Enter a prime number: "))
            if is_prime(p):
                break  # Exit loop if p is prime
            else:
                print(f" - {p} is not a prime number. Please try
again.")
        except ValueError:
            print(" - Invalid input. Please enter a valid integer.")

    while True:
        try:
            q = int(input(" - Enter another prime number
(different prime): "))
            if is_prime(q) and p != q:
                break  # Exit loop if q is prime and not equal to p
            else:
                print(f" - {q} is not a valid prime number or is equal
to {p}. Please try again.")
        except ValueError:
            print(" - Invalid input. Please enter a valid integer.")

    print(" - Generating your public/private key-pairs
now . . .")

    # Generate the RSA key pairs
    public, private = generate_key_pair(p, q)

    print(" - Your public key is ", public, " and your private key
is ", private)

    # Input for the message to be encrypted
    input_text = input(" - Enter a message to encrypt: ")
    encrypted_msg = encrypt(public, input_text)  # Encrypt
the message
```

```
    # Display the encrypted message
    print(" - Your encrypted message is: ", ".join(map(lambda
x: str(x), encrypted_msg)))

    # Decrypt the message using the private key
    print(" - Decrypting message with private key ", private,
" . . .")
    print(" - Your message is: ", decrypt(private,
encrypted_msg))  # Show the decrypted message

    print(" ")

print("=======================================
=== END
================================================
============")

print("=======================================
================================================
====================")
```