EXPLICACION CODIGO RSA

```python
import random

# 1. Función para verificar si un número es primo
def es_primo(n):
    if n < 2:
        return False
    for i in range(2, int(n**0.5) + 1):
        if n % i == 0:
            return False
    return True
```

Aquí tienes la explicación de la función `es_primo` en español:

Esta función verifica si un número dado `n` es un número primo. Así es como funciona:

1. **Maneja números menores que 2**: Los números primos se definen como números naturales mayores que 1. Por lo tanto, si `n` es menor que 2, la función devuelve `False` inmediatamente.
2. **Itera a través de posibles divisores**: Luego, recorre los números a partir de 2 hasta la raíz cuadrada de `n` (inclusive). Solo necesitamos verificar hasta la raíz cuadrada porque si un número `n` tiene un divisor mayor que su raíz cuadrada, también debe tener un divisor menor que su raíz cuadrada.
3. **Verifica la divisibilidad**: Dentro del bucle, verifica si `n` es divisible por el número actual `i` utilizando el operador de módulo (`%`). Si `n % i` es igual a 0, significa que `i` es un divisor de `n` y, por lo tanto, `n` no es un número primo. En este caso, la función devuelve `False`.
4. **Devuelve True si no se encuentran divisores**: Si el bucle termina sin encontrar ningún divisor, significa que `n` no es divisible por ningún número desde 2 hasta su raíz cuadrada. Por lo tanto, `n` es un número primo y la función devuelve `True`.

```python
# 2. Generar un número primo aleatorio grande (para fines didácticos se usan menores)
# Aumentamos el rango para manejar mensajes más grandes
def generar_primo(minimo=100000, maximo=999999):
    while True:
        p = random.randint(minimo, maximo)
        if es_primo(p):
            return p
```

Para manejar textos potencialmente más grandes, lo que necesitamos es ampliar *aún más* ese rango para que los primos `p` y `q` que se generen aleatoriamente sean más grandes, y por lo tanto, su producto `n` también sea mayor.

Vamos a aumentar el rango de búsqueda de primos a uno mucho mayor. Por ejemplo, podemos ir de 100,000,000 a 999,999,999.

Esta función se encarga de generar un número primo aleatorio dentro de un rango especificado. Aquí te detallo cómo funciona:

1. **Define el rango de búsqueda**: La función toma dos argumentos opcionales, `minimo` y `maximo`, que establecen el rango dentro del cual se buscará el número primo. Por defecto, busca entre 1000 y 9999.
2. **Bucle infinito**: Utiliza un bucle `while True` para seguir buscando un número primo hasta que lo encuentre.
3. **Genera un número aleatorio**: Dentro del bucle, genera un número entero aleatorio `p` entre `minimo` y `maximo` (ambos inclusive) utilizando `random.randint(minimo, maximo)`.
4. **Verifica si es primo**: Llama a la función `es_primo(p)` (la que te expliqué anteriormente) para verificar si el número aleatorio `p` es primo.
5. **Devuelve el primo encontrado**: Si `es_primo(p)` devuelve `True`, significa que se ha encontrado un número primo. En ese momento, la función `return p` detiene el bucle y devuelve el número primo encontrado. Si no es primo, el bucle continúa y se genera un nuevo número aleatorio.

En resumen, la función `generar_primo` busca de forma aleatoria un número dentro de un rango hasta que encuentra uno que sea primo, utilizando la función `es_primo` para verificarlo. Los comentarios en el código indican que, aunque se podría generar primos muy grandes, para propósitos didácticos se usan rangos menores.

```python
# 3. MCD
def mcd(a, b):
    while b:
        a, b = b, a % b
    return a
```

Permíteme explicarte la función `mcd`.

Esta función calcula el Máximo Común Divisor (MCD) de dos números enteros, `a` y `b`, utilizando el algoritmo de Euclides. Así es como funciona paso a paso:

1. **Inicia un bucle**: La función entra en un bucle `while b:`. Este bucle continuará ejecutándose mientras el valor de `b` sea distinto de cero.
2. **Aplica el algoritmo de Euclides**: Dentro del bucle, se realiza la operación clave del algoritmo de Euclides: `a, b = b, a % b`. Esto significa que el nuevo valor de `a` será el valor actual de `b`, y el nuevo valor de `b` será el resto de la división de `a` entre `b` (calculado con `a % b`). Esta asignación múltiple se realiza simultáneamente.
3. **El bucle continúa hasta que b sea cero**: El proceso de reemplazar `a` por `b` y `b` por el resto de la división continúa. En cada iteración, el valor de `b` se hace más pequeño. Cuando `b` finalmente se convierte en 0, el buckle termina.
4. **Devuelve el MCD**: Cuando el bucle `while b:` finaliza (porque `b` es 0), el valor actual de `a` es el Máximo Común Divisor de los números originales. La función `return a` devuelve este valor.

En esencia, el algoritmo de Euclides se basa en el principio de que el Máximo Común Divisor de dos números no cambia si el número más grande se reemplaza por la diferencia entre los dos números. Repitiendo este proceso (o, de manera más eficiente, usando el resto de la división), eventualmente se llega a cero, y el último número no nulo es el MCD.

```python
# 4. Inverso modular
def inverso_modular(e, phi):
    d, x1, x2, y1 = 0, 0, 1, 1
    temp_phi = phi
    while e > 0:
        q = temp_phi // e
        temp_phi, e = e, temp_phi - q * e
        x1, x2 = x2 - q * x1, x1
        y1, d = d - q * y1, y1
    return d % phi
```

Te explicaré la función `inverso_modular`.

Esta función calcula el inverso multiplicativo modular de un número `e` módulo `phi`. En el contexto de RSA, esto es crucial para encontrar la clave privada `d` a partir de la clave pública `e` y el totient de Euler `phi`. La función utiliza el algoritmo extendido de Euclides.

Aquí te explico cómo funciona:

1. **Inicialización**: Se inicializan varias variables:

- `d`, `x1`, `x2`, `y1`: Estas variables se utilizan en el algoritmo extendido de Euclides para seguir el rastro de los coeficientes que nos permitirán encontrar el inverso modular.
- `temp_phi`: Se crea una copia de `phi` para no modificar el valor original durante el cálculo.

2. **Bucle del algoritmo extendido de Euclides**: El bucle `while e > 0:` continúa mientras el valor de `e` sea positivo. Este bucle implementa los pasos del algoritmo extendido de Euclides.
3. **Calcula el cociente y actualiza valores**: Dentro del bucle:
   - `q = temp_phi // e`: Se calcula el cociente de la división entera de `temp_phi` entre `e`.
   - `temp_phi, e = e, temp_phi - q * e`: Se actualizan los valores de `temp_phi` y `e` de manera similar a cómo se hace en el algoritmo de Euclides para el MCD. El nuevo `temp_phi` es el antiguo `e`, y el nuevo `e` es el resto de la división.
   - `x1, x2 = x2 - q * x1, x1` y `y1, d = d - q * y1, y1`: Estas líneas actualizan los coeficientes `x` y `y` que son fundamentales en el algoritmo extendido de Euclides para expresar el MCD (que al final del bucle será 1, si `e` y `phi` son coprimos) como una combinación lineal de `e` y `phi`. La variable `d` es la que nos interesa, ya que eventualmente contendrá el inverso modular.

```
# 5. Cifrado y descifrado
def cifrar_rsa(m, e, n):
    return pow(m, e, n)

def descifrar_rsa(mc, d, n):
    return pow(mc, d, n)
```

Te explicaré las funciones `cifrar_rsa` y `descifrar_rsa`.

Estas dos funciones son el corazón del algoritmo RSA, ya que implementan las operaciones de cifrado y descifrado utilizando las claves pública y privada. Ambas funciones hacen uso de la función incorporada de Python `pow(base, exp, mod)`.

Aquí te detallo cada una:

1. **Función `cifrar_rsa(m, e, n)`**:
   - **Propósito**: Esta función se utiliza para cifrar un mensaje.
   - **Argumentos**:
     - `m`: Es el mensaje original que se desea cifrar, representado como un número entero.
     - `e`: Es el exponente de la clave pública.
     - `n`: Es el módulo, que forma parte tanto de la clave pública como de la clave privada.

- **Funcionamiento**: La función simplemente calcula `m` elevado a la potencia de `e`, y luego toma el resultado de esa operación módulo `n`. Esto se logra eficientemente con `pow(m, e, n)`.
- **Resultado**: Devuelve el mensaje cifrado `mc` como un número entero. La fórmula matemática detrás de esto es `mc = m^e mod n`.

2. **Función `descifrar_rsa(mc, d, n)`**:
   - **Propósito**: Esta función se utiliza para descifrar un mensaje cifrado.
   - **Argumentos**:
     - `mc`: Es el mensaje cifrado que se desea descifrar, representado como un número entero.
     - `d`: Es el exponente de la clave privada (el inverso modular de `e` módulo `phi`).
     - `n`: Es el módulo, el mismo que se usó para cifrar.
   - **Funcionamiento**: La función calcula el mensaje cifrado `mc` elevado a la potencia de `d`, y luego toma el resultado de esa operación módulo `n`. Al igual que con el cifrado, se utiliza `pow(mc, d, n)` para realizar esta operación de manera eficiente.
   - **Resultado**: Devuelve el mensaje original `m` (en su forma numérica) como un número entero. La fórmula matemática detrás de esto es `m = mc^d mod n`.

La belleza del algoritmo RSA radica en que, gracias a las propiedades matemáticas del inverso modular y el teorema de Euler, elevar el mensaje cifrado a la potencia de la clave privada `d` (módulo `n`) revierte el proceso de cifrado y recupera el mensaje original.

```python
# 6. Convertir texto a binario
def texto_a_binario(texto):
    return ' '.join(format(ord(c), '08b') for c in texto)
```

Te explicaré la función `texto_a_binario`.

Esta función toma una cadena de texto como entrada y la convierte en una representación binaria, donde cada carácter del texto original se representa por su código ASCII (u otro estándar de codificación compatible) en binario, separado por espacios.

Aquí te detallo cómo funciona:

1. **Iteración sobre cada carácter**: La parte central de la función es una *comprensión de generador* `(format(ord(c), '08b') for c in texto)`. Esto itera sobre cada carácter `c` en la cadena de `texto` de entrada.
2. **Obtener el código numérico del carácter**: Para cada carácter `c`, `ord(c)` devuelve el valor numérico (código Unicode/ASCII) correspondiente a ese carácter. Por ejemplo, `ord('A')` devuelve 65.
3. **Convertir el número a binario**: `format(..., '08b')` toma el número obtenido de `ord(c)` y lo formatea como una cadena binaria.
   - `'b'` especifica que el formato de salida debe ser binario.

- `'08'` especifica que la cadena binaria debe tener un ancho mínimo de 8 dígitos, rellenando con ceros (`0`) a la izquierda si es necesario. Esto asegura que cada carácter se represente con 8 bits (un byte), lo cual es común en muchas codificaciones de caracteres.

4. **Unir los binarios con espacios**: Finalmente, `' '.join(...)` toma la secuencia de cadenas binarias generadas para cada carácter y las une en una sola cadena, separando cada representación binaria con un espacio.

Ejemplo:

Si el `texto` es "Hola":

- 'H' -> `ord('H')` es 72 -> `format(72, '08b')` es "01001000"
- 'o' -> `ord('o')` es 111 -> `format(111, '08b')` es "01101111"
- 'l' -> `ord('l')` es 108 -> `format(108, '08b')` es "01101100"
- 'a' -> `ord('a')` es 97 -> `format(97, '08b')` es "01100001"

La función `texto_a_binario("Hola")` devolverá: `"01001000 01101111 01101100 01100001"`.

En resumen, esta función convierte cada carácter de una cadena de texto en su representación binaria de 8 bits y las une con espacios para facilitar la lectura.

```python
# 7. Convertir texto a número decimal
def texto_a_decimal(texto):
    binario = ''.join(format(ord(c), '08b') for c in texto)
    return int(binario, 2)
```

Ahora te explicaré la función `texto_a_decimal`.

Esta función toma una cadena de texto y la convierte en un único número entero decimal. Esencialmente, convierte el texto a su representación binaria (concatenada) y luego interpreta esa cadena binaria como un gran número en base 2 para obtener su equivalente en base 10 (decimal).

Aquí te explico los pasos:

1. **Convierte el texto a una cadena binaria concatenada**:
   - `binario = ''.join(format(ord(c), '08b') for c in texto)`: Esta parte es muy similar a la función `texto_a_binario` que te expliqué antes, con una diferencia clave: en lugar de usar `' '.join(...)` para separar los binarios de cada carácter con espacios, usa `''.join(...)`. Esto significa que las representaciones binarias de 8 bits de cada carácter se unen *sin espacios*, formando una única y larga cadena binaria.
2. **Convierte la cadena binaria a un número decimal**:

- `return int(binario, 2)`: Aquí es donde ocurre la conversión final. La función incorporada de Python `int()` se utiliza para convertir una cadena a un número entero.
  - El primer argumento (`binario`) es la cadena que se va a convertir.
  - El segundo argumento (`2`) especifica la base de la cadena de entrada. Al pasar `2`, le estamos diciendo a `int()` que la cadena `binario` debe interpretarse como un número en base 2 (binario).

Ejemplo:

Si el `texto` es "Hi":

- 'H' -> `format(ord('H'), '08b')` es "01001000"
- 'i' -> `format(ord('i'), '08b')` es "01101001"

La línea `binario = ''.join(...)` resultará en la cadena binaria `"0100100001101001"`.

Luego, `int("0100100001101001", 2)` interpretará `"0100100001101001"` como un número binario y lo convertirá a su equivalente decimal.

En resumen, esta función convierte el texto a una larga secuencia de bits y luego trata esa secuencia como un solo número binario gigante, obteniendo su valor decimal. Este número decimal es el "mensaje" `M` que se utiliza en el cifrado RSA.

```
# 8. Convertir decimal a texto
def decimal_a_texto(decimal):
    binario = bin(decimal)[2:]
    while len(binario) % 8 != 0:
        binario = '0' + binario
    chars = [chr(int(binario[i:i+8], 2)) for i in range(0, len(binario), 8)]
    return ''.join(chars)
```

Ahora te explicaré la última función, `decimal_a_texto`.

Esta función es el proceso inverso de `texto_a_decimal`. Toma un número entero decimal (que representa el mensaje descifrado) y lo convierte de nuevo a su formato de texto original.

Aquí te detallo cómo funciona:

1. **Convierte el número decimal a binario**:
   - `binario = bin(decimal)[2:]`: La función incorporada de Python `bin(decimal)` convierte el número entero `decimal` en una cadena que representa su valor binario. Esta cadena comienza con el prefijo "0b" para indicar que es un número binario (por ejemplo, `bin(65)` devuelve

"0b1000001"). `[2:]` es una operación de slicing que elimina este prefijo "0b", dejándonos solo con la secuencia de bits (por ejemplo, "1000001").

2. **Asegura que la cadena binaria tenga longitud múltiplo de 8**:
   - `while len(binario) % 8 != 0:`: Este bucle `while` comprueba si la longitud de la cadena binaria es un múltiplo de 8.
   - `binario = '0' + binario`: Si la longitud no es un múltiplo de 8, significa que faltan ceros iniciales que se perdieron durante la conversión de decimal a binario. Este paso agrega un cero a la izquierda de la cadena `binario` hasta que su longitud sea un múltiplo de 8. Esto es crucial porque cada carácter original estaba representado por exactamente 8 bits.
3. **Divide la cadena binaria en bloques de 8 bits y convierte a caracteres**:
   - `chars = [chr(int(binario[i:i+8], 2)) for i in range(0, len(binario), 8)]`: Esta es una *comprensión de lista* que procesa la cadena binaria en bloques de 8 bits.
     - `range(0, len(binario), 8)` genera una secuencia de índices que empiezan en 0 y avanzan de 8 en 8 (0, 8, 16, etc.) hasta el final de la cadena binaria.
     - Para cada índice `i`, `binario[i:i+8]` extrae una subcadena de 8 caracteres de la cadena binaria, comenzando en el índice `i`.
     - `int(..., 2)` convierte esta subcadena binaria de 8 bits de nuevo a su valor numérico decimal (base 10).
     - `chr(...)` toma este valor numérico decimal y lo convierte de nuevo al carácter correspondiente (utilizando la tabla Unicode/ASCII).
     - La comprensión de lista recopila todos estos caracteres convertidos en una lista llamada `chars`.
4. **Une los caracteres para formar el texto final**:
   - `return ''.join(chars)`: Finalmente, `''.join(chars)` toma todos los caracteres en la lista `chars` y los une para formar una única cadena de texto, que es el mensaje descifrado original.

En resumen, esta función toma el número decimal descifrado, lo convierte de nuevo a binario (asegurándose de tener el número correcto de bits), divide la secuencia de bits en grupos de 8, convierte cada grupo de 8 bits de nuevo a su carácter correspondiente, y une estos caracteres para reconstruir el texto original.

```python
# === PROGRAMA PRINCIPAL ===
print("=== CIFRADO RSA CON TEXTO ===")
texto = input("Ingresa el texto a cifrar: ")

print("\nTexto original:", texto)
print("Texto en binario:", texto_a_binario(texto))

mensaje_decimal = texto_a_decimal(texto)
print("Mensaje M (decimal):", mensaje_decimal)

# Generar primos p y q
p = generar_primo()
q = generar_primo()
n = p * q
phi = (p - 1) * (q - 1)
```

Ahora te explicaré la sección del "PROGRAMA PRINCIPAL" que has resaltado. Aquí es donde se juntan todas las funciones que hemos revisado para realizar el proceso de cifrado RSA con texto.

Esta parte del código realiza los siguientes pasos:

1. **Título y entrada del usuario**:
   - `print("=== CIFRADO RSA CON TEXTO ===")`: Simplemente imprime un título para el programa en la consola.
   - `texto = input("Ingresa el texto a cifrar: ")`: Le pide al usuario que ingrese el texto que desea cifrar. El texto ingresado por el usuario se almacena en la variable `texto`.
2. **Mostrar información del texto original**:
   - `print("\nTexto original:", texto)`: Imprime el texto que el usuario ingresó.
   - `print("Texto en binario:", texto_a_binario(texto))`: Llama a la función `texto_a_binario` (que te expliqué antes) para convertir el texto ingresado a su representación binaria y luego imprime esta representación.
   - `mensaje_decimal = texto_a_decimal(texto)`: Llama a la función `texto_a_decimal` (que también te expliqué) para convertir el texto a un único número entero decimal. Este número es el mensaje `M` que se utilizará en el cifrado RSA y se almacena en la variable `mensaje_decimal`.
   - `print("Mensaje M (decimal):", mensaje_decimal)`: Imprime el valor decimal del mensaje.
3. **Generación de las claves RSA (parte inicial)**:
   - `p = generar_primo()`: Llama a la función `generar_primo` para obtener un número primo aleatorio grande. Este primo se almacena en la variable `p`.

- `q = generar_primo()`: Llama nuevamente a la función `generar_primo` para obtener otro número primo aleatorio grande, distinto del anterior. Este primo se almacena en la variable `q`. Es importante que `p` y `q` sean primos y diferentes para la seguridad de RSA.
- `n = p * q`: Calcula el producto de los dos primos `p` y `q`. Este valor `n` es el módulo y es una parte fundamental tanto de la clave pública como de la clave privada. Se almacena en la variable `n`.
- `phi = (p - 1) * (q - 1)`: Calcula el totient de Euler (φ) para `n`. Para un número `n` que es el producto de dos primos distintos `p` y `q`, el totient de Euler es simplemente `(p-1) * (q-1)`. Este valor `phi` es esencial para encontrar la clave privada `d`. Se almacena en la variable `phi`.

En resumen, esta sección inicial del programa principal obtiene el texto del usuario, lo prepara para el cifrado convirtiéndolo a un número decimal, y luego genera los componentes iniciales necesarios para las claves RSA: los dos números primos `p` y `q`, el módulo `n`, y el totient de Euler `phi`. Los siguientes pasos en el código continuarán con la generación de las claves `e` y `d`, y luego el cifrado y descifrado

```
# Elegir e tal que 1 < e < phi y MCD(e, phi) = 1
e = 3
while mcd(e, phi) != 1:
    e += 2

d = inverso_modular(e, phi)

# Cifrado y descifrado
mc = cifrar_rsa(mensaje_decimal, e, n)
m_descifrado = descifrar_rsa(mc, d, n)
texto_descifrado = decimal_a_texto(m_descifrado)
```

Continuemos con la explicación de la siguiente parte del programa principal. Esta sección se centra en la selección de la clave pública `e`, el cálculo de la clave privada `d`, y finalmente, la aplicación del cifrado y descifrado RSA.

Aquí te detallo los pasos:

1. **Selección del exponente público `e`**:
   - `e = 3`: Se inicializa el valor de `e` a 3. `e` es el exponente de la clave pública. Debe cumplir dos condiciones importantes: 1 < `e` < `phi` y el Máximo Común Divisor (MCD) entre `e` y `phi` debe ser 1 (es decir, deben ser coprimos). Un valor pequeño como 3 se usa a menudo porque hace que el cifrado sea más rápido.
   - `while mcd(e, phi) != 1:`: Este bucle `while` verifica si el valor actual de `e` es coprimo con `phi` utilizando la función `mcd` que te expliqué antes. El bucle continúa mientras el MCD de `e` y `phi` sea diferente de 1.

- `e += 2`: Si el MCD no es 1, el valor de `e` se incrementa en 2 (solo se prueban números impares, ya que si `phi` es par, cualquier `e` par tendría un MCD de al menos 2 con `phi`). El bucle vuelve a verificar la condición hasta encontrar un `e` que sea coprimo con `phi`.

2. **Cálculo del exponente privado `d`**:
   - `d = inverso_modular(e, phi)`: Una vez que se ha encontrado un valor adecuado para `e`, se llama a la función `inverso_modular` (que también te expliqué) para calcular el inverso multiplicativo modular de `e` módulo `phi`. El resultado, que es el exponente de la clave privada `d`, se almacena en la variable `d`. Este `d` es la clave que permitirá descifrar los mensajes cifrados con la clave pública `e`.
3. **Cifrado del mensaje**:
   - `mc = cifrar_rsa(mensaje_decimal, e, n)`: Aquí se realiza el cifrado del mensaje. Se llama a la función `cifrar_rsa` con el mensaje original en formato decimal (`mensaje_decimal`), el exponente público `e`, y el módulo `n`. La función aplica la fórmula `mc = mensaje_decimal^e mod n`. El resultado, el mensaje cifrado `mc`, se almacena en la variable `mc`.
4. **Descifrado del mensaje**:
   - `m_descifrado = descifrar_rsa(mc, d, n)`: Ahora se descifra el mensaje cifrado. Se llama a la función `descifrar_rsa` con el mensaje cifrado (`mc`), el exponente privado `d`, y el módulo `n`. La función aplica la fórmula `m_descifrado = mc^d mod n`. El resultado, el mensaje descifrado en formato decimal `m_descifrado`, se almacena en la variable `m_descifrado`.
5. **Conversión del mensaje descifrado de vuelta a texto**:
   - `texto_descifrado = decimal_a_texto(m_descifrado)`: Finalmente, se llama a la función `decimal_a_texto` (que te expliqué por último) para convertir el número decimal descifrado (`m_descifrado`) de nuevo a su formato de texto original. El texto resultante se almacena en la variable `texto_descifrado`.

En resumen, esta parte del código completa la generación de claves encontrando el exponente público `e` y calculando el exponente privado `d`. Luego, utiliza estas claves para cifrar el mensaje original y descifrar el mensaje cifrado, demostrando que el proceso de descifrado recupera el mensaje original. La última parte del programa principal probablemente mostrará los resultados de estas operaciones.

```
# === RESULTADOS ===
print(f"\nPrimos generados: p = {p}, q = {q}")
print("\n=== CIFRADO ===")
print(f"Clave pública (e, n): ({e}, {n})")
print(f"Clave privada (d, n): ({d}, {n})")
print(f"Mensaje cifrado (mc): {mc}")

print("\n=== DESCIFRADO ===")
print("Texto original (para comparar):", texto) # Added this line
print(f"Mensaje descifrado (decimal): {m_descifrado}")
print(f"Texto descifrado: {texto_descifrado}")
```

Ahora te explicaré la última sección del "PROGRAMA PRINCIPAL", que es donde se muestran todos los resultados importantes del proceso de cifrado y descifrado RSA.

Esta parte del código simplemente imprime en la consola la información relevante calculada en los pasos anteriores, organizándola para que sea fácil de entender.

Aquí te detallo lo que hace cada línea de `print`:

1. **Mostrar los primos generados**:
   - `print(f"\nPrimos generados: p = {p}, q = {q}")`: Imprime los valores de los dos números primos `p` y `q` que fueron generados aleatoriamente al inicio. Esto muestra las "bases" secretas sobre las que se construyó el sistema de claves.
2. **Sección de resultados del Cifrado**:
   - `print("\n=== CIFRADO ===")`: Imprime un encabezado para la sección que muestra los resultados del cifrado.
   - `print(f"Clave pública (e, n): ({e}, {n})")`: Imprime la clave pública, que consiste en el par de números `(e, n)`. Esta es la clave que se compartiría para que cualquiera pueda cifrar mensajes dirigidos a ti.
   - `print(f"Clave privada (d, n): ({d}, {n})")`: Imprime la clave privada, que consiste en el par de números `(d, n)`. Esta clave debe mantenerse en secreto, ya que es necesaria para descifrar los mensajes.
   - `print(f"Mensaje cifrado (mc): {mc}")`: Imprime el valor numérico del mensaje después de haber sido cifrado utilizando la clave pública.
3. **Sección de resultados del Descifrado**:
   - `print("\n=== DESCIFRADO ===")`: Imprime un encabezado para la sección que muestra los resultados del descifrado.
   - `print(f"Mensaje descifrado (decimal): {m_descifrado}")`: Imprime el valor numérico del mensaje después de haber sido descifrado utilizando la clave privada. Este valor debería ser el mismo que el `Mensaje M (decimal)` que se mostró al inicio, antes del cifrado.
   - `print(f"Texto descifrado: {texto_descifrado}")`: Imprime el mensaje después de haber sido convertido de nuevo a texto a partir del valor

decimal descifrado. Si todo el proceso funcionó correctamente, este texto debería ser idéntico al "Texto original" que el usuario ingresó al principio.

En conjunto, esta sección de resultados permite verificar visualmente que el proceso de cifrado y descifrado se realizó correctamente, mostrando las claves generadas, el mensaje cifrado y, lo más importante, el mensaje descifrado que debe coincidir con el original.

Resultado final

```
=== CIFRADO RSA CON TEXTO ===
Ingresa el texto a cifrar: LA TECNOLOGIA ES INNOVADORA

Texto original: LA TECNOLOGIA ES INNOVADORA
Texto en binario: 01001100 01000001 00100000 01010100 01000101 01000011 01001110 010011
Mensaje M (decimal): 31369240495543357612301101200602911080401703207964230187238773313

Primos generados: p = 4909, q = 3847

=== CIFRADO ===
Clave pública (e, n): (5, 18884923)
Clave privada (d, n): (11325701, 18884923)
Mensaje cifrado (mc): 12828228

=== DESCIFRADO ===
Mensaje descifrado (decimal): 15278977
Texto descifrado: é#
```

NOTA EL RESULTADO DEL TEXTO EN BINARIO ES MAS LARGO