Guía de funciones RSA: Explicaciones y fundamentos matemáticos

Proyecto `rsa_basic`

Introducción

Este documento recoge las funciones desarrolladas en el proyecto `rsa_basic`, junto con una explicación detallada de su funcionamiento y los fundamentos matemáticos que las respaldan. Se trata de una guía que actúa como memoria de prácticas y servirá de referencia para consultas futuras.

Funciones desarrolladas

1. `gcd(a: int, b: int) ->int`

Descripción: Calcula el máximo común divisor (MCD) entre dos números enteros positivos usando el algoritmo de Euclides.

Implementación:

```python
def gcd(a: int, b: int) -> int:
    while b != 0:
        a, b = b, a % b
    return a
```

Fundamento matemático:

Dado dos números enteros positivos a y b, el algoritmo de Euclides se basa en la propiedad:

$$\gcd(a, b) = \gcd(b, a \bmod b)$$

Iterando esta propiedad hasta que el segundo operando sea 0, se obtiene el MCD.

2. `modinv(a: int, m: int) ->int`

Descripción: Calcula el inverso modular de a módulo m, es decir, un entero x tal que:

$$(a \cdot x) \bmod m = 1$$

Implementación:

```python
def modinv(a: int, m: int) -> int:
    m0, x0, x1 = m, 0, 1
    while a > 1:
        q = a // m
        a, m = m, a % m
```

```
        x0, x1 = x1 - q * x0, x0
    if a != 1:
        raise ValueError("No existe inverso modular para los valores 
            ↪ elegidos")
    return x1 % m0
```

Fundamento matemático:

El inverso modular existe si y solo si $\gcd(a, m) = 1$. Se utiliza el algoritmo extendido de Euclides, que encuentra coeficientes x e y tales que:

$$ax + my = \gcd(a, m)$$

Si $\gcd(a, m) = 1$, entonces $ax \equiv 1 \mod m$, y por tanto x es el inverso modular buscado.

3. `generate_keys(bits: int = 16)`

Descripción: Genera un par de claves RSA (pública y privada) a partir de dos números primos de `bits` bits.

Implementación:

```
def generate_keys(bits: int = 16):
    p = getPrime(bits)
    q = getPrime(bits)
    while q == p:
        q = getPrime(bits)
    n = p * q
    phi = (p - 1) * (q - 1)
    e = random.randrange(2, phi)
    while gcd(e, phi) != 1:
        e = random.randrange(2, phi)
    d = modinv(e, phi)
    return (e, n), (d, n)
```

Fundamento matemático:

El sistema RSA se basa en:

- La dificultad de factorizar un número grande $n = pq$
- La existencia del inverso modular d de e módulo $\varphi(n) = (p - 1)(q - 1)$

La clave pública es (e, n), y la privada (d, n), donde:

$$d \equiv e^{-1} \mod \varphi(n)$$

4. `encrypt_message(message: str, public_key: tuple) ->list[int]`

Descripción: Cifra un mensaje utilizando la clave pública RSA. Transforma cada carácter en su código ASCII y aplica la fórmula de cifrado.

Implementación:

```
def encrypt_message(message: str, public_key: tuple) -> list[int]:
    e, n = public_key
    encrypted = []
```

```
    for char in message:
        m = ord(char)
        c = pow(m, e, n)
        encrypted.append(c)
    return encrypted
```

Fundamento matemático:
Cada carácter del mensaje se convierte en un entero m, y se cifra como:

$$c = m^e \mod n$$

El resultado es una lista de enteros cifrados.

5. `decrypt_message(ciphertext: list[int], private_key: tuple) ->str`

Descripción: Descifra una lista de enteros cifrados usando la clave privada RSA.
Implementación:

```
def decrypt_message(ciphertext: list[int], private_key: tuple) -> str
    ↪ :
    d, n = private_key
    decrypted = ""
    for c in ciphertext:
        if c >= n:
            raise ValueError(f"El␣valor␣cifrado␣{c}␣no␣puede␣
               ↪ descifrarse:␣debe␣ser␣menor␣que␣n={n}")
        m = pow(c, d, n)
        decrypted += chr(m)
    return decrypted
```

Fundamento matemático:
El descifrado aplica la función inversa:

$$m = c^d \mod n$$

Si $c = m^e \mod n$, entonces:

$$c^d \equiv (m^e)^d \equiv m^{ed} \mod n \equiv m \mod n$$

por la propiedad $ed \equiv 1 \mod \varphi(n)$.
Cada entero descifrado m se convierte nuevamente a su carácter con `chr(m)`.

Observaciones

- Este documento actúa como referencia viva. Se podrá ampliar con nuevas funciones, ejemplos de uso, casos de prueba y referencias bibliográficas.
- Se incluirá también información teórica clave sobre criptografía, RSA y seguridad de clave pública en futuras versiones.

0.1. Función **`getPrime`** (PyCryptodome)

Para generar números primos usados en RSA (Fase 1), empleamos:

```
from Crypto.Util.number import getPrime
```

La función `getPrime(N)` produce un primo probabilístico de N bits:

1. Genera un entero aleatorio de N bits (par o impar).
2. Aplica un test simple contra una tabla de cribado (primos pequeños).
3. Si pasa, aplica el test de primalidad de Miller–Rabin (pseudoprimalidad probabilística).
4. Si supera todas las rondas, se devuelve como primo.

Este método garantiza que la probabilidad de error (que un número compuesto sea considerado primo) es menor que 10^{-6} por defecto :contentReference[oaicite:3]index=3.

Significado matemático: Un número primo de N bits cumple:

$$2^{N-1} \leq p < 2^N, \quad \gcd(p, e) = 1$$

para cualquier exponente público e. La prueba Miller–Rabin confirma la primalidad de forma probabilística eficiente.

Fin de la Fase 1

---

1. Detalles teóricos y matemáticos de la fase 2: cifrado simétrico con Fernet

—

1.1. Criptografía simétrica con **`Fernet`**

Utilizamos la clase `cryptography.fernet.Fernet` para cifrar/descifrar. Internamente funciona así :contentReference[oaicite:4]index=4:

$$F = \text{Fernet}(k), \quad \text{token} = F.\text{encrypt}(m)$$

donde el token es:

$$\text{token} = \text{base64}(v \parallel t \parallel IV \parallel c \parallel \text{HMAC})$$

con:

- v: versión del protocolo (`0x80`) (1 byte)
- t: timestamp de generación (8 bytes)
- IV: vector de inicialización AES-CBC, 16 bytes
- $c = \text{AES-CBC}_{k_1}(m, IV)$: ciphertext
- $\text{HMAC} = \text{HMAC}_{k_2}(v \| t \| IV \| c)$: HMAC-SHA256 (32 bytes)

Proceso de cifrado

- $IV \xleftarrow{\$} \{0,1\}^{128}$
- $c = \text{AES-CBC}_{k_1}(m,\ IV)$
- $\text{tag} = \text{HMAC}_{k_2}(v\|t\|IV\|c)$
- $\text{Token} = v\|t\|IV\|c\|tag$

Proceso de descifrado

$$\text{verificar HMAC}_{k_2}(v\|t\|IV\|c) \stackrel{?}{=} tag \quad \Rightarrow \quad m = \text{AES-CBC}_{k_1}^{-1}(c, IV)$$

Si la verificación HMAC falla, no se descifra, garantizando integridad y autenticación.
—

1.2. Ventajas matemáticas y seguridad

- El IV aleatorio garantiza que c varía aún si m y k_1 son iguales.
- La HMAC protege contra modificaciones activas del mensaje.
- La combinación AES-CBC + HMAC cumple el principio de *encrypt-then-MAC*, el método más seguro :contentReference[oaicite:5]index=5.

Así, Fernet ofrece un esquema *cifrado autenticado simétrico* completo, basado en principios matemáticos sólidos.

—