crypto

November 24, 2023

[1]:
```
# Una lista con algunos números primos pequeños.

primos = [ 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61,␣
 ↪67, 71,
73, 79, 83, 89, 97, 101, 103, 107, 109, 113, 127, 131, 137, 139, 149, 151, 157,␣
 ↪163, 167, 173,
179, 181, 191, 193, 197, 199, 211, 223, 227, 229, 233, 239, 241, 251, 257, 263,␣
 ↪269, 271, 277, 281,
283, 293, 307, 311, 313, 317, 331, 337, 347, 349, 353, 359, 367, 373, 379, 383,␣
 ↪389, 397, 401, 409,
419, 421, 431, 433, 439, 443, 449, 457, 461, 463, 467, 479, 487, 491, 499, 503,␣
 ↪509, 521, 523, 541,
547, 557, 563, 569, 571, 577, 587, 593, 599, 601, 607, 613, 617, 619, 631, 641,␣
 ↪643, 647, 653, 659 ]

def primo(indice = 0)-> int:
    if indice < 0: return None
    if indice > len(primos): return None
    return primos[indice]

print(primo(5)) #13
```

```
13
```

[2]:
```
# Algoritmo extendido de Euclides para el cálculo del MCD de dos números␣
 ↪naturales a y b
def MCD(a,b):
    while b != 0:
        a,b = b, a % b
    return a

display(MCD(3,6)) # MCD(3,6) = 3
display(MCD(5,17)) # MCD(5,17) = 1 (co-primos o primos relativos)
```

```
3
```

```
1
```

[3]:
```python
# Comprobar si dos números son co-primos:
def son_coprimos(a,b):
    return MCD(a,b) == 1

print(son_coprimos(14,15)) # True
print(son_coprimos(14,28)) # False
```

```
True
False
```

[4]:
```python
# Optimización usando la función gcd incorporada en math
from math import gcd

def son_coprimos_opt(a,b):
    return gcd(a,b) == 1

print(son_coprimos_opt(14,15)) # True
print(son_coprimos_opt(14,28)) # False
```

```
True
False
```

[5]:
```python
# Cálculo de co-primos de un número
def coprimos(a, maximo):
    coprimos = []
    for i in range(1, maximo + 1):
        if son_coprimos_opt(a,i):
            coprimos.append(i)
    return coprimos

print(coprimos(55,100))
```

```
[1, 2, 3, 4, 6, 7, 8, 9, 12, 13, 14, 16, 17, 18, 19, 21, 23, 24, 26, 27, 28, 29,
31, 32, 34, 36, 37, 38, 39, 41, 42, 43, 46, 47, 48, 49, 51, 52, 53, 54, 56, 57,
58, 59, 61, 62, 63, 64, 67, 68, 69, 71, 72, 73, 74, 76, 78, 79, 81, 82, 83, 84,
86, 87, 89, 91, 92, 93, 94, 96, 97, 98]
```

La función ϕ de Euler (también llamada función indicatriz de Euler o función totiente) es una función importante en teoría de números. Si n es un número entero positivo, entonces ϕ(n) se define como la cantidad de enteros positivos menores a n y coprimos con n, es decir, formalmente se puede definir como:

$\phi(n) = \ |\{m \in \mathbb{N} \mid m \leq n \wedge mcd(m,n) = 1\}|$

Donde $|\cdot|$ es la cardinalidad del conjunto de valores (o el número de elementos en el conjunto).

[6]:
```python
def indicatriz(a):
    if a < 1: return None
    return len(coprimos(a,a-1))
```

```
print(indicatriz(55)) # 40
```

40

Con φ es la función de Euler, se puede calcular la indicatriz como $\varphi(n) = (p-1)\cdot(q-1)$ para dos primos p y q aprovechando las dos propiedades de la función de Euler siguientes:

1. $\varphi(p) = p-1$ si p p es primo.
2. Si m y n son primos entre sí, entonces $\varphi(mn) = \varphi(m)\varphi(n)$

[7]:
```
def indicatriz_coprimos(p1,p2):
    if p1 < 1 or p2 < 1: return None
    return (p1 - 1) * (p2 - 1)

print(indicatriz_coprimos(5,11)) # 40
```

40

Para buscar el producto modular inverso de $a \ mod \ \varphi(n)$, buscamos un número k tal que $ak \equiv 1 \ (mod \ \varphi(n))$. Esto se puede hacer con el algoritmo extendido de Euclides, pero también se puede usar la regla $ak \equiv [(a \ mod \ \varphi(n))(k \ mod \ \varphi(n))] \ (mod \ \varphi(n))$ y buscar un k tal que $[(a \ mod \ \varphi(n))(k \ mod \ \varphi(n))] \ mod \ \varphi(n) = 1$.

[8]:
```
def inversa_modular(a,phin):
    # Buscamos un número k tal que ak sea congruente con 1 (mod phi(n)) donde b
 ↪= phi(n), b > 1 y phi(n) es la función de la indicatriz de Euler.
    if phin <= 1: return None
    for k in range(0,phin):
        # Aquí se usa el hecho de que ak es congruente con [(a mod phi(n)) (k
 ↪mod phi(n))] (mod phi(n)). Como ak tiene que ser congruente con 1 (mod
 ↪phi(n)),
        # buscamos un k tal que  [(a mod phi(n)) (k mod phi(n))] mod phi(n) = 1.
        if (a % phin) * (k % phin) % phin == 1:
            return k
    return None

print(inversa_modular(3, 40)) #27
```

27

[9]:
```
from random import randint
def generar_claves():
    NUMEROS_PRIMOS = len(primos)
    # Se obtienen dos números primos p y q distintos
    p = primo(randint(0,NUMEROS_PRIMOS - 1))
    q = p
    while q == p:
        q = primo(randint(0,NUMEROS_PRIMOS - 1))
    # Se calcula n = p * q. n es el módulo de la clave pública.
    n = p*q
```

```
    # Se calcula la indicatriz de n
    phin = indicatriz_coprimos(p,q)
    # Se escoge un entero positivo e menor que phi(n), e es coprimo de phi(n). e es el exponente de la clave pública.
    e = 2
    while son_coprimos_opt(phin, e) == False:
        e += 1
    # Se calcula d como el multiplicador modular inverso de e mod phi(n). d es el exponente de la clave privada.
    d = inversa_modular(e, phin)
    # Clave pública: (e,n). Clave privada: (d, n).
    return [(n,e), (n,d)]

print(generar_claves())
```

```
[(4571, 5), (4571, 1565)]
```

0.1 Cifrado

El cifrado se realiza mediante la siguiente fórmula:

$C = M^e \ mod \ n$

Donde C es el mensaje codificado, M es el mensaje original, e el exponente de la clave pública, y n el módulo. Para descifrarlo:

$M = C^d \ mod \ n$

Donde M es el mensaje original, C el mensaje codificado, d es el exponente de la clave privada, y n el módulo.

[12]:
```
def cifrado(mensaje: str, clave_publica: tuple)->list:
    return [ord(i)**clave_publica[1] % clave_publica[0] for i in mensaje]

claves = generar_claves()
mensaje = "¡Hola mundo!"
mensaje_cifrado = cifrado(mensaje, claves[0])
print(mensaje_cifrado)
```

```
[740, 147, 252, 324, 274, 206, 580, 351, 191, 247, 252, 690]
```

[13]:
```
def descifrar(mensaje: list, clave_privada: tuple)->str:
    return ''.join([chr(i**clave_privada[1] % clave_privada[0]) for i in mensaje])

mensaje_descifrado = descifrar(mensaje_cifrado, claves[1])
print(mensaje_descifrado)
```

```
¡Hola mundo!
```

[ ]: