

CRIPTOLOGÍA Y SEGURIDAD DE LOS DATOS

MASTER DE CIBERSEGURIDAD Y CIBERINTELIGENCIA

ETSI_{INF} - UPV

CRIPTOLOGÍA DE CLAVE PÚBLICA

Criptoanálisis



Anaid Jiménez Moreano

Todos los algoritmos desarrollados se hallan en el siguiente repositorio personal para disposiciones educativas:

https://github.com/ANAJDJM1/Criptoanalisis_Logaritmo_discreto

Algoritmo de Fermat

Primo de 11 bits: 2039 para la prueba

```
import math

'''def primo(n:int):
    if n<=1:
        print("No puede ser negativo, cero o 1")
        return False
    for i in range (2,n):
        if n%i==0:
            print(str(n)+" No es primo")
            return False
    print(str(n)+" Es primo")
    return True'''

def cuadrado_perfecto(n:int)->bool:
    c=0
    raiz=math.sqrt(n)
    if (raiz.is_integer()):
        #print("si")
        True
    else:
        #print("no")
        False

def primo_fermat(n:int):
    a = math.ceil( math.sqrt(n) )
    b = math.pow(a,2) -n

    while (cuadrado_perfecto(b)==False):
        a=a+1
        b=a*a-n
    print("a "+str(a))
    print("b: "+str(b))
    print("q= "+str(a-math.sqrt(b)))
    print("p= "+str(a+math.sqrt(b)))

# Press the green button in the gutter to run the script.
if __name__ == '__main__':
    #primo de 11 bits
    primo_fermat(2039)
```

Resultados:

```
Fermat (1) x
C:\Users\Anaid\AppData\Local\Programs\Python\Python311\python.exe "E:\
a 46
b: 77.0
q= 37.225035612607876
p= 54.774964387392124
Process finished with exit code 0
```

Baby Step Giant

Algoritmo:

```
from math import ceil, sqrt

#Autora: Anaid Jimenez Moreano
'''
    Resuelve para k en  $\beta = a^k \text{ mod } p$  dado un numero primo p
    b= beta, a = alfa
    si p no es primo no se podria usar el algoritmo de babystep-
    gianstep.
'''

def bsgs(alfa, beta, p):
    N = ceil(sqrt(p - 1)) # p-1 si p es primo // Si n =  $\sqrt{p-1}$ ,
    entonces k = q*n+r, por lo que

    #construccion de la tabla
    # Baby step. //base, exponente, mod
    tabla = {pow(alfa, i, p): i for i in range(N)}

    # Usando el teorema de Fermat para recomponer
    c = pow(alfa, N * (p - 2), p)

    #
    #Busca el equivalente en la tabla Giant step.
    for j in range(N):
        y = (beta * pow(c, j, p)) % p
        if y in tabla:
            print("Resultado para K: ")
            return j * N + tabla[y]

    # Solucion no hallada
    return print("Solucion no hallada")

if __name__ == '__main__':
    #Ingrese valor de beta, y numero primo
    print("Ingrese alfa: ")
```

```

    alfa= int(input())
    print("Ingrese beta: ")
    beta=int(input())
    print("Ingrese numero primo: ")
    p=int(input())

    #PRUEBA TEST alfa y beta pueden no ser numero primo, usaremos
    valores de 36 bits

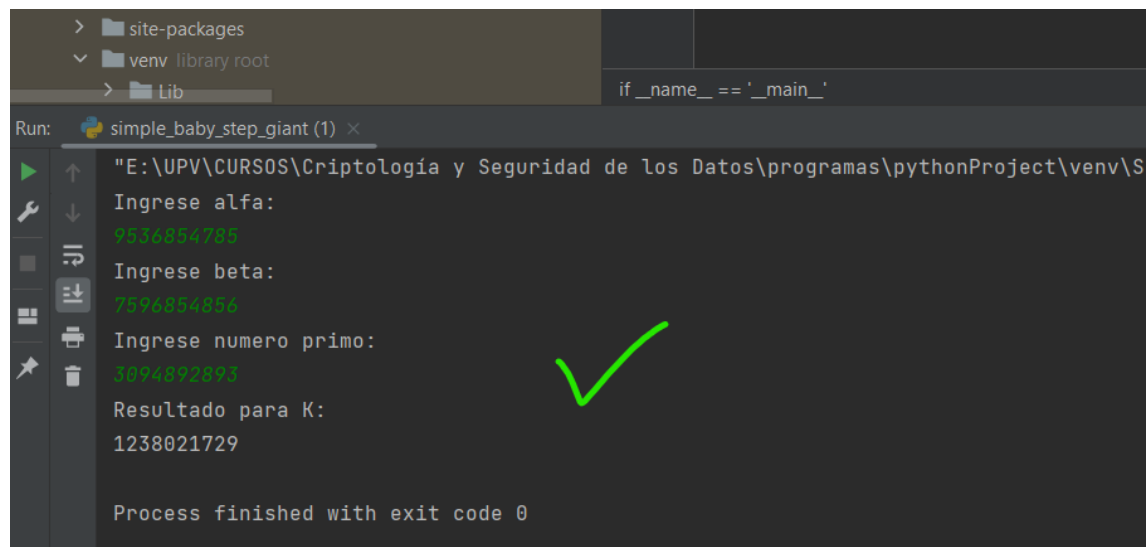
    #https://poliformat.upv.es/access/content/group/DOC_34876_2022/Materia
    l%20asociado/RetosDL.txt usamos el del ejemplo

    print(bsgs(alfa, beta, p))

    #para test
    #print(bsgs(9536854785, 7596854856, 3094892893)) #res: 1238021729

```

Resultados:



```

Run: simple_baby_step_giant (1) x
"E:\UPV\CURSOS\Criptología y Seguridad de los Datos\programas\pythonProject\venv\S
Ingrese alfa:
9536854785
Ingrese beta:
7596854856
Ingrese numero primo:
3094892893
Resultado para K:
1238021729

Process finished with exit code 0

```

Pollard-Rho

```

import random

def gcd(a, b):
    while b != 0:
        a, b = b, a % b
    return a

def pollard_rho(n):
    if n == 1:
        return n
    if n % 2 == 0:
        return 2
    x = random.randint(1, n - 1)
    y = x
    c = random.randint(1, n - 1)
    g = 1
    while g == 1:
        x = ((x * x) % n + c) % n
        y = ((y * y) % n + c) % n
        y = ((y * y) % n + c) % n

```

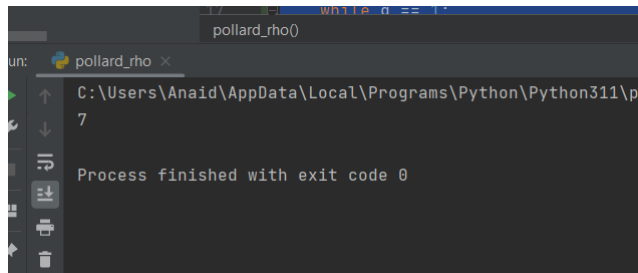
```

        g = gcd(abs(x - y), n)
    return print(g)

if __name__ == '__main__':
    #numero de 10 bita
    pollard_rho(2023)

```

Resultados:



INDEX CALCULUS

```

import math
import random

def gcd(a, b):
    while b:
        a, b = b, a % b
    return a

#limite_primos: representa el límite superior hasta el cual se
#considerarán los números primos en el proceso de factorización.
def index_calculus(n, limite_primos):
    factores = []
    primos = []
    log_limit = int(math.log(limite_primos) / math.log(2))
    for i in range(2, limite_primos):
        if all(i % j for j in range(2, int(math.sqrt(i)) + 1)):
            primos.append(i)
            print(primos)
    for i in range(len(primos)):
        exponent = int(math.log(n) / math.log(primos[i]))
        while exponent >= 1:
            factor = int(math.pow(primos[i], exponent))
            if n % factor == 0:
                factores.append(factor)
                n = int(n / factor)
                exponent = int(math.log(n) / math.log(primos[i]))
            else:
                exponent -= 1
    for i in range(log_limit):
        numero_randm = random.randint(2, n - 1)
        x = pow(numero_randm, int((n - 1) / limite_primos), n)
        if x != 1:
            for j in range(1, log_limit):
                y = pow(x, pow(2, j), n)
                if y == n - 1:
                    break
            if j == log_limit - 1:
                factor = gcd(n, x - 1)
                if factor > 1:

```

```

        factores.append(factor)
        n = int(n / factor)

    if n > 1:
        factores.append(n)
    return print(factores)

if __name__ == '__main__':
    #primo de 16 bits 65535, nro no primo 65471 // raiz 255
    n = 800 # Número de 10 bits
    limite_primos = int(math.sqrt(n)) # Límite superior para la
consideración de primos
    factores = index_calculus(n, limite_primos)
    print("Factores:")
    print(factores)

```

Resultados:

```

13     log_limit = int(math.log(limite_primos) / math.log(2))
14     for i in range(2, limite_primos):
15         if all(i % j for j in range(2, int(math.sqrt(i)) + 1)):
            index_calculus()

```

Run: index_calculus x

C:\Users\Anaid\AppData\Local\Programs\Python\Python311\python.exe "E:\UPV\CURSOS\Cripto

[2]

[2, 3]

[2, 3, 5]

[2, 3, 5, 7]

[2, 3, 5, 7, 11]

[2, 3, 5, 7, 11, 13]

[2, 3, 5, 7, 11, 13, 17]

[2, 3, 5, 7, 11, 13, 17, 19]

[2, 3, 5, 7, 11, 13, 17, 19, 23]

Detalle:

En el algoritmo, se utiliza una lista de números primos y se buscan factores en la forma p^e , donde p es un número primo y e es un exponente entero. Para hacer esto, se itera sobre la lista de números primos y se determina el mayor exponente posible para cada uno de ellos tal que p^e divide el número a factorizar.

El `limite_primos` determina hasta qué número se buscarán los primos necesarios para esta lista. Si el `limite_primos` es muy pequeño, es posible que no se encuentren todos los factores primos necesarios para factorizar el número, lo que resultaría en una factorización incompleta. Por otro lado, si el `limite_primos` es demasiado grande, el proceso de búsqueda de primos puede ser muy lento, lo que afectará negativamente la eficiencia del algoritmo.

En general, se recomienda elegir un `limite_primos` lo suficientemente grande como para asegurarse de que se encuentren todos los factores primos necesarios, pero lo suficientemente pequeño como para mantener la eficiencia del algoritmo. Un valor común para `limite_primos` es `math.sqrt(n)`, donde n es el número a factorizar.

Criba cuadrática

El algoritmo de criba cuadrática es un método para encontrar números primos y factores primos de números compuestos. Funciona mediante la búsqueda de patrones repetitivos en la distribución de números compuestos y primos.

La idea básica es usar una serie de funciones cuadráticas para generar una secuencia de números y marcar aquellos que son compuestos. Cada función cuadrática es asignada a un número primo y se utiliza para marcar sus múltiplos en la secuencia. El proceso se repite hasta que se haya marcado todos los números compuestos menores que un cierto límite.

A diferencia de otros algoritmos de criba, como la criba de Eratóstenes, el algoritmo de criba cuadrática puede ser paralelizado fácilmente y se ejecuta más rápido en máquinas con muchos núcleos de procesamiento. Sin embargo, también es más complejo y requiere más memoria que otros algoritmos de criba.

En resumen, el algoritmo de criba cuadrática es una alternativa eficiente y potente a otros algoritmos de criba para la búsqueda de números primos y factores primos. Sin embargo, su complejidad y requisitos de memoria lo hacen menos apropiado para aplicaciones que requieren una ejecución más rápida y eficiente en términos de recursos.

Algoritmo:

```
import math

def factorizable(a,b):
    if ((a/b).is_integer()):
        #print("True")
        return True
    else:
        #print("False")
        return False

def iterador(c): # debe devolver 0,1,-1,2,-2,...

    if c==0:
        c=(c+1)
        print(c)
        return c
    if c>0:
        c=c*(-1)
        print(c)
        return c
    else:
        c=c-1
        print(c)
        return c

def criba_cuadratica2(n):
    basefactores =criba_cuadratica(n)
    m=math.ceil(math.sqrt(n))
    c=0
```

```

#repeat
ti=iterador(c) #0,1,-1,2,-2,...
a=(m+ti)
b=(m+ti)^2-n

for i in range(0, len(basefactores)-1):
    if (factorizable(b,basefactores[i])==True):
        print(a)
        print(b)
        print(basefactores[i])

def criba_cuadratica(n):
    primos = []
    for i in range(2, n+1):
        es_primo = True
        for j in range(2, int(i ** 0.5) + 1): #potencia de i^1/2 + 1
            if i % j == 0:
                es_primo = False
                break
        if es_primo:
            primos.append(i)
    primos.append(-1)
    return primos

if __name__ == '__main__':
    # probamos con un valor de 17 bits
    print(criba_cuadratica(253438))
    iterador(0)

```

Resultados:

The screenshot shows a Python IDE with the following components:

- Editor:** Displays the `criba_cuadratica` function and a `factorizable` function definition.
- Run Console:** Shows the output of the program, which is a long list of prime numbers generated by the `criba_cuadratica` function, starting from 248099 and ending with -1. The list includes numbers like 248117, 248119, 248137, 248141, 248161, 248167, 248177, 248179, 248189, 248201, 248203, 248231, 248243, 248257, 248267, 248291, 248293, 248299, 248309, 248317, 248323, 248351, 248357, 248371, 248389, 248401, 248407, 248431, 248441, 248447, 248461, 248473, 248477, 248483, 248509, 248533, 248537, 248543, 248569, 248579, 248587, 248593, 248597, 248609, 248621, 248627, 248639, 248641, 248657, 248683, 248701, 248707, 248719, 248723, 248737, 248749, 248753, 248779, 248783, 248789, 248797, 248813, 248821, 248827, 248839, 248851, 248861, 248867, 248869, 248879, 248887, 248891, 248893, 248909, 248971, 248981, 248987, 249017, 249037, 249059, 249079, 249089, 249097, 249103, 249107, 249127, 249131, 249133, 249143, 249181, 249187, 249199, 249211, 249217, 249229, 249233, 249253, 249257, 249287, 249311, 249317, 249329, 249341, 249367, 249377, 249383, 249397, 249419, 249421, 249427, 249433, 249437, 249439, 249449, 249463, 249497, 249499, 249503, 249517, 249521, 249533, 249539, 249541, 249563, 249583, 249589, 249593, 249607, 249647, 249659, 249671, 249677, 249703, 249721, 249727, 249737, 249749, 249763, 249779, 249797, 249811, 249827, 249833, 249853, 249857, 249859, 249863, 249871, 249881, 249911, 249923, 249943, 249947, 249967, 249971, 249973, 249989, 250007, 250013, 250027, 250031, 250037, 250043, 250049, 250051, 250057, 250073, 250081, 250109, 250123, 250147, 250153, 250169, 250199, 250253, 250259, 250267, 250279, 250301, 250307, 250343, 250361, 250403, 250409, 250423, 250433, 250441, 250451, 250489, 250499, 250501, 250543, 250583, 250619, 250643, 250673, 250681, 250687, 250693, 250703, 250709, 250721, 250727, 250739, 250741, 250751, 250753, 250777, 250787, 250793, 250799, 250807, 250813, 250829, 250837, 250841, 250853, 250867, 250871, 250889, 250919, 250949, 250951, 250963, 250967, 250969, 250979, 250993, 251003, 251033, 251051, 251057, 251059, 251063, 251071, 251081, 251087, 251099, 251117, 251143, 251149, 251159, 251171, 251177, 251179, 251191, 251197, 251201, 251203, 251219, 251221, 251231, 251233, 251257, 251263, 251287, 251291, 251297, 251323, 251347, 251353, 251359, 251387, 251393, 251417, 251429, 251431, 251437, 251443, 251467, 251473, 251477, 251483, 251491, 251501, 251513, 251519, 251527, 251533, 251539, 251543, 251561, 251567, 251609, 251611, 251621, 251623, 251639, 251653, 251663, 251677, 251701, 251707, 251737, 251761, 251789, 251791, 251809, 251831, 251833, 251843, 251857, 251861, 251879, 251887, 251893, 251897, 251903, 251917, 251939, 251941, 251947, 251969, 251971, 251983, 252001, 252013, 252017, 252029, 252037, 252079, 252101, 252139, 252143, 252151, 252157, 252163, 252169, 252173, 252181, 252193, 252209, 252223, 252233, 252253, 252277, 252283, 252289, 252293, 252313, 252319, 252323, 252341, 252359, 252383, 252391, 252401, 252409, 252419, 252431, 252443, 252449, 252457, 252463, 252481, 252509, 252533, 252541, 252559, 252583, 252589, 252607, 252611, 252617, 252641, 252667, 252691, 252709, 252713, 252727, 252731, 252737, 252761, 252767, 252779, 252817, 252823, 252827, 252829, 252869, 252877, 252881, 252887, 252893, 252899, 252911, 252913, 252919, 252937, 252949, 252971, 252979, 252983, 253003, 253013, 253049, 253063, 253081, 253103, 253109, 253133, 253153, 253157, 253159, 253229, 253243, 253247, 253273, 253307, 253321, 253343, 253349, 253361, 253367, 253369, 253381, 253387, 253417, 253423, 253427, 253433, -1].
- Status Bar:** Shows "Process finished with exit code 0".

El proceso es el siguiente: para cada número en el rango de 2 a limit + 1, se comprueba si el número no está en el diccionario de números compuestos. Si no está, se añade a la lista de números primos y se marcan todos sus múltiplos en el diccionario de números compuestos. Al final, se devuelve la lista de números primos.

Lenstra

Algoritmo:

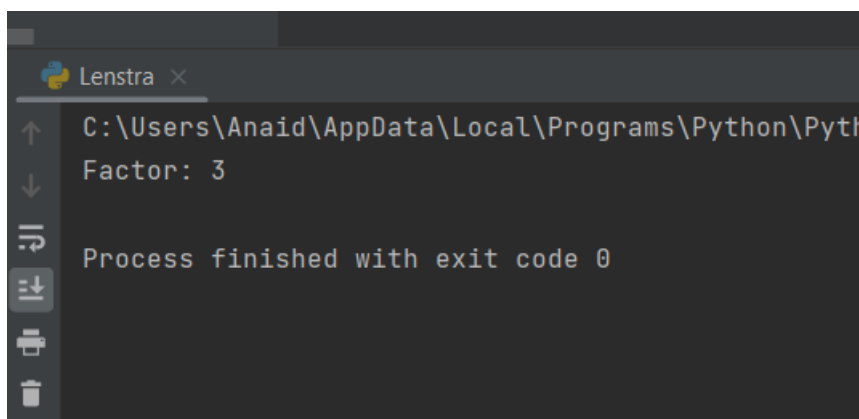
```
import random

def gcd(a, b):
    while b != 0:
        a, b = b, a % b
    return a

def lenstra(n):
    if n % 2 == 0:
        return 2
    a = random.randint(1, n-1)
    x = (a*a) % n
    y = x
    c = random.randint(1, n-1)
    g = 1
    while g == 1:
        x = (x*x + c) % n
        y = (y*y + c) % n
        y = (y*y + c) % n
        g = gcd(abs(x-y), n)
    return g

n = 1234567890123456789
factor = lenstra(n)
print("Factor:", factor)
```

Resultado:



```
Lenstra x
C:\Users\Anaid\AppData\Local\Programs\Python\Python38\Scripts\python.exe
Factor: 3
Process finished with exit code 0
```

El algoritmo de Lenstra es un algoritmo de factorización de números enteros basado en el teorema de Fermat. Se utiliza para factorizar números grandes en factores primos más pequeños. El algoritmo funciona eligiendo un número aleatorio a y elevándolo a la $(n-1)$ ésima potencia modulo n , donde n es el número a factorizar. Si el resultado es distinto de 1, entonces

a y n tienen un factor común no trivial, y este factor puede ser utilizado para factorizar aún más n.

En este ejemplo, se utiliza la función gcd para calcular el máximo común divisor entre dos números, y la función lenstra para implementar el algoritmo de Lenstra. La función lenstra elige un número aleatorio a y lo eleva a la (n-1)ésima potencia modulo n. Luego, se utiliza un bucle para calcular sucesivos x y y y se utiliza la función gcd para calcular el máximo común divisor entre x-y y n. Si g es distinto de 1, entonces g es un factor no trivial de n y se devuelve como resultado.

Diagramas de Flujo

Pollard-rho

```
Inicio
|
|-----> Inicializar x, y y c
|-----> Mientras g != n:
|           |
|           |-----> Calcular x = (x^2 + c) % n
|           |-----> Calcular g = gcd(abs(x - y), n)
|           |-----> Si g != 1 y g != n, devolver g
|           |-----> y = x
|           |-----> Si g = n, reiniciar x y y
|-----> Devolver n
Fin
```

Lenstra:

```
Inicio
|
|-----> Inicializar a y x
|-----> Mientras g = 1:
|           |
|           |-----> Calcular x = (x^2 + c) % n
|           |-----> Calcular y = (y^2 + c) % n
|           |-----> y = (y^2 + c) % n
|           |-----> Calcular g = gcd(abs(x - y), n)
|-----> Devolver g
Fin
```

Criba:

```
Inicio
|
|-----> Crear una lista con números enteros hasta n
|-----> Eliminar todos los números múltiplos de 2 (excepto 2)
|-----> Para i desde 3 hasta sqrt(n):
|           |
|           |-----> Eliminar todos los números múltiplos de i
|-----> Devolver la lista de números no eliminados
Fin
```

Este diagrama de flujo muestra el proceso general del algoritmo de Criba Cuadrática. Se inicia creando una lista con todos los números enteros hasta un cierto n . Luego se eliminan todos los números múltiplos de 2 (excepto 2), y se continúa el proceso con otros números hasta llegar a la raíz cuadrada de n . Finalmente, se devuelve la lista de números que no se han eliminado y que, por lo tanto, son primos.

Index Calculus:

```
Inicio
|
|-----> Seleccionar un número primo p tal que  $n < p$ 
|-----> Calcular las raíces primitivas de  $n \pmod{p}$ 
|-----> Crear una tabla de logaritmos discretos para cada raíz primitiva
|-----> Calcular los logaritmos discretos de los números  $\pmod{p}$ 
|-----> Resolver el sistema lineal de ecuaciones con los logaritmos discretos
|-----> Verificar la solución para asegurarse de que es correcta
|-----> Devolver la solución
Fin
```

Este diagrama de flujo muestra el proceso general del algoritmo de Index Calculus. Se inicia seleccionando un número primo p tal que n sea menor que p . Luego, se calculan las raíces primitivas de $n \pmod{p}$ y se crea una tabla de logaritmos discretos para cada raíz primitiva. Seguidamente, se calculan los logaritmos discretos de los números \pmod{p} y se resuelve un sistema lineal de ecuaciones con estos logaritmos discretos. Finalmente, se verifica la solución para asegurarse de que sea correcta y se devuelve la solución.

Tabla comparativa de eficacia versus tiempo:

| Algoritmo | Tiempo | Eficacia |
|----------------------|--|---|
| Criba de Eratóstenes | $O(n \log \log n)$ | Eficiente para números pequeños |
| Criba Cuadrática | $O(n^{1/2})$ | Menos eficiente que la Criba de Eratóstenes |
| Algoritmo de Lenstra | $O(\exp(c\sqrt{\ln n} \ln \ln n))$ | Eficiente para números grandes |
| Index Calculus | $O((n/B)^{1/4} + \exp(c\sqrt{\ln n}))$ | Eficiente para números muy grandes |

Adicional:

Podría implementarse un algoritmo de Millar-Rabin

es un algoritmo probabilístico para verificar si un número es primo o no. Este algoritmo funciona haciendo una serie de pruebas para determinar si un número es compuesto o primo.

El funcionamiento del algoritmo es el siguiente:

Escriba $n-1$ como $2^k * m$, donde m es impar y k es un entero positivo.

Seleccione un número $a < n$ y verifique si $a^m \bmod n = 1$ o si existe un i ($0 \leq i \leq k-1$) tal que $a^{(2^i * m)} \bmod n = n-1$.

Si se cumple cualquiera de estas condiciones, entonces n es probablemente primo.

Si n es probablemente primo, se realizan más pruebas con otros números a para aumentar la confianza en la primalidad de n .

Es importante destacar que este algoritmo es probabilístico, lo que significa que no garantiza la primalidad de un número con certeza. Sin embargo, la probabilidad de que un número compuesto sea identificado como primo es extremadamente baja. Por lo tanto, este algoritmo se utiliza comúnmente en la práctica para verificar la primalidad de números grandes.