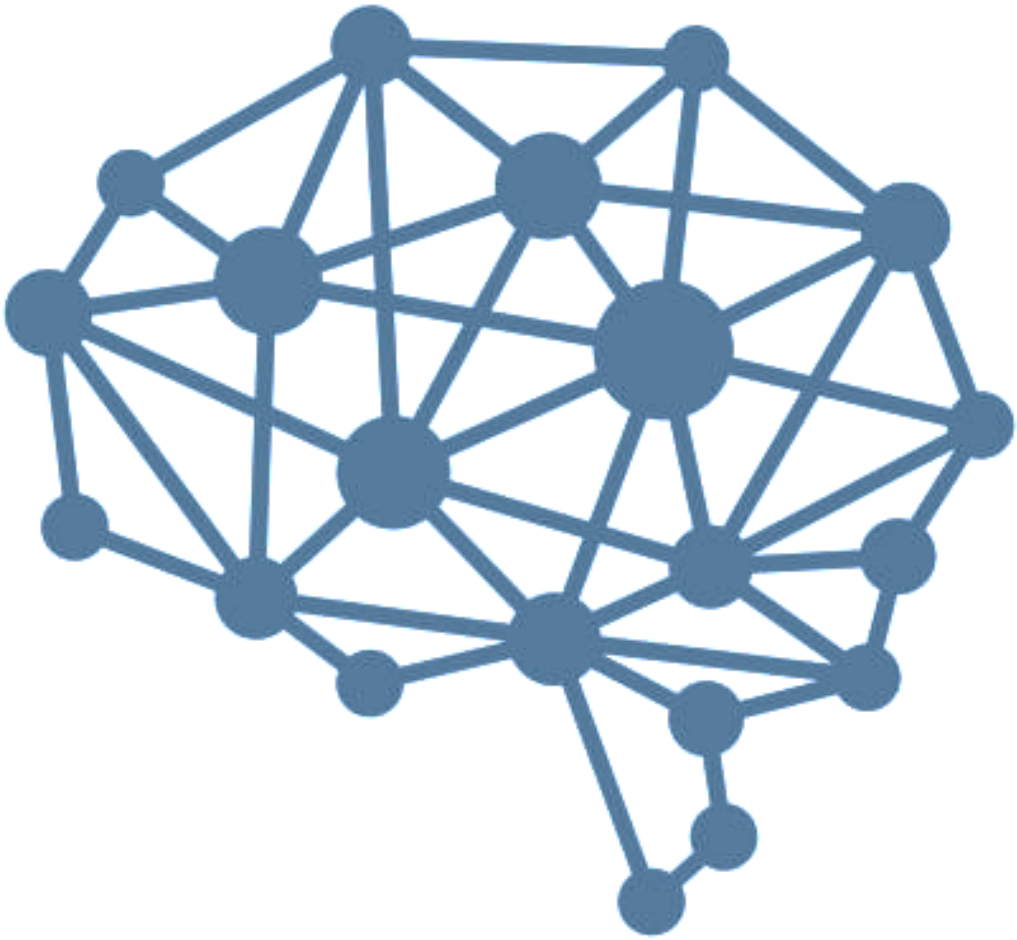
CSD. Trabajo 5

CRIPTOANÁLISIS



Autor: Javier García Ibáñez

DNI: 20952351H

Universidad Politécnica de Valencia

Índice de contenido

Índice de Ilustraciones

1. Introducción

El objetivo de esta práctica es implementar y analizar varios algoritmos estudiados en la asignatura, utilizados para atacar las funciones unidireccionales en las que los métodos criptográficos de clave publica basan su seguridad.

Se analizarán en este proyecto los dos problemas principales utilizados por los métodos criptográficos, el cálculo del logaritmo discreto y el de factorizar un entero, utilizando para su estudio varios algoritmos como Fermat, Pollard-Rho, Pollard P-1, Lenstra o Baby-Giant-Step.

Toda la programación requerida para esta práctica se realizará en lenguaje Python y mediante el editor de código Visual Studio Code.

El estudio de cada algoritmo se fundamentará en pruebas realizadas con anterioridad y cuyos resultados estarán disponibles en el siguiente repositorio GITHUB.

2. Factorización de enteros

La factorización de números enteros tiene una gran relación con la criptografía de clave publica, y es que, si consideremos un entero n, el grupo multiplicativo Z^*_n, y un elemento α de Z^*_n. Una vez calculado $y = \alpha^e \bmod n$, es posible obtener de nuevo α a partir de y si conocemos la factorización de n.

En esta sección de la práctica se pretenden estudiar diferentes técnicas de factorización de enteros, mediante la implementación de diferentes algoritmos, como *Fermat, Pollard-Rho, Pollard p-1 y Lenstra.*

Para cada uno de estos algoritmos se realizarán las pruebas dispuestas en poliformat, comenzando por describir cada uno de los algoritmos, su implementación y unas breves pruebas iniciales para comprobar y analizar su funcionamiento.

Una vez terminada la implementación básica de estos algoritmos se realizará un estudio más en profundidad sobre cada uno de los algoritmos desarrollados, utilizando un conjunto de datos mucho más amplio.

2.1. Características del ordenador

La experimentación de este algoritmo se ha realizado en un Mini PC con las siguientes características:

- Procesador: Intel de procesador de 11.ª gen N5095 (2,9GHz)
- Núcleos: 4 – Procesos: 4
- Gráfica: No contiene
- RAM: 16 GB DDR4
- Memoria: 256 GB SSD
- Sistema Operativo: Ubuntu 20.04

2.2. Implementación básica

La elección de los algoritmos utilizados en este estudio no tiene más trasfondo que el simple hecho de la curiosidad. Se decidió implementar primeramente los tres algoritmos más simples con la intención de realizar un primer acercamiento a estas técnicas factorización de enteros.

Más tarde se decidió implementar también el algoritmo de Lenstra, algo más complejo que los tres anteriores.

En cuanto a las pruebas que se realizarán sobre cada algoritmo en esta primera parte del estudio, serán muy simples, útiles únicamente para comprender funcionamiento de los algoritmos y comprobar el correcto funcionamiento del mismo.

Respecto a las pruebas a realizar en este primer apartado, para cada algoritmo se utilizarán 10 pruebas para cada tamaño de números, siendo los tamaños a probar [32, 40, 44, 48, 52, 56, 60, 64] bits.

El hecho de utilizar únicamente 10 pruebas y un tamaño de bits no demasiado grande es porque, como ya se dijo anteriormente, el objetivo de este primer apartado de implementación base es el de estudiar el funcionamiento de los algoritmos.

2.2.1. Fermat

Este algoritmo consiste en encontrar dos números a y b tal que $N = a^2 - b^2 = (a+b)\,(a-b)$.

El valor de a se va incrementando iterativamente en uno, hasta que, mediante el cálculo $b^2 = a^2 - n$, resulte una b^2 cuadrado perfecto, es decir, cuya raíz cuadrada sea un entero.

De esta forma, este método, prueba iterativamente, combinaciones de números enteros que al multiplicarse den como resultado N, siendo estos dos factores del mismo.

Require: n número a factorizar
Ensure: dos factores de n
a = math.ceil(√n)
b = a^2 - n
while not √b **es entero**:
    a = a + 1
    b = a^2 - n
return (a - √b, a + √b)

Experimentación Básica

En cuanto al funcionamiento del algoritmo, creo adecuado mostrar un ejemplo concreto del funcionamiento del mismo:

9227319586514486887 => (2836708781, 3252825827)

En estos algoritmos no tiene demasiado sentido mostrar la memoria utilizada, ya que únicamente utilizan variables temporales que se reescriben tras cada iteración, por lo que no se notaria un aumento de la misma.

Para los tamaños de [32, 40, 44, 48, 52, 56, 60, 64] bits y 10 pruebas para cada uno, se ha obtenido un 100% de acierto. En cuanto a los tiempos se puede observar (en la Ilustracion 1) un crecimiento exponencial en función del tamaño, aunque con cierta imprecisión debido a la reducida cantidad de pruebas realizadas.

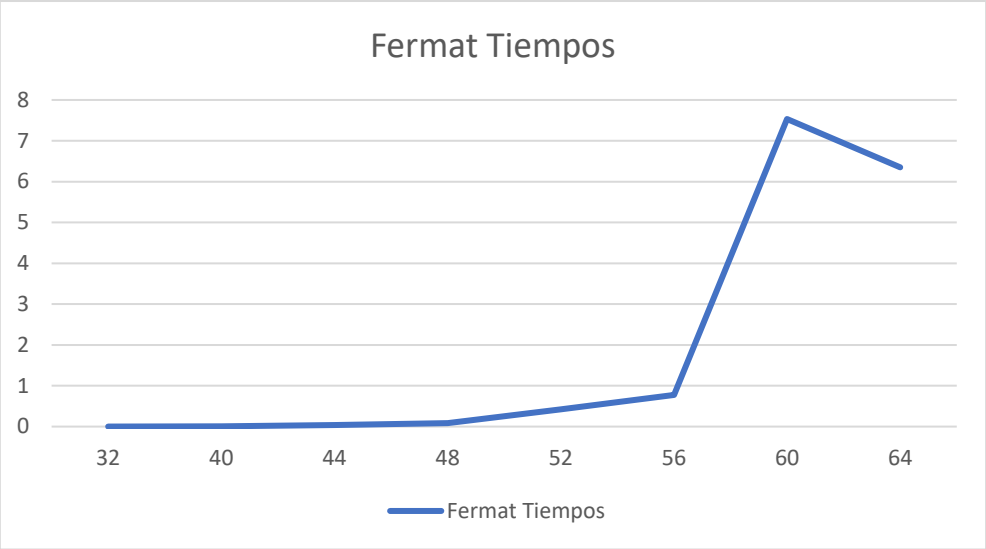


Ilustración 1. Algoritmo de Fermat pruebas simples

Esa pequeña bajada al final de la gráfica se debe, como ya he dicho, a la reducida cantidad de pruebas realizadas, ya que es posible que casualmente esos valores hayan sido más sencillos de que los de tamaño 60.

2.2.2. Pollard-rho

Es un algoritmo Montecarlo que busca encontrar valores congruentes con uno de los factores de n. Consiste en modificar iterativamente valores a y b aplicando una función pseudoaleatoria.

Si la diferencia entre a y b es congruente con el factor (desconocido) p (o bien con q), entonces: mcd(a − b, n) = p.

Require: Un número entero positivo compuesto n
Ensure: Un factor de n
A = B = random(2, n − 1)
while True **do**
  A = A2 + 1 mod n
  B = B2 + 1 mod n
  B = B2 + 1 mod n
  p = mcd(A − B, n)
  if 1 < p < n **then**
    return p
  end if
  if p == n **then**
    return n
  end if
end while

Experimentación Básica

Al igual que el caso anterior, mediante estas primeras pruebas iniciales, se comprueba el correcto funcionamiento del algoritmo, siendo prueba de esto:

17295603783688662937 => 4163610511

En cuanto a los tiempos obtenidos por este algoritmo son mucho mejores que los del algoritmo anterior. En la Ilustración 2 se puede observar como el tiempo crece también de manera exponencial, en este caso sin demasiada perturbación. En la ilustración 4 se puede observar la comparación entre los tiempos de Fermat y PollardRho.

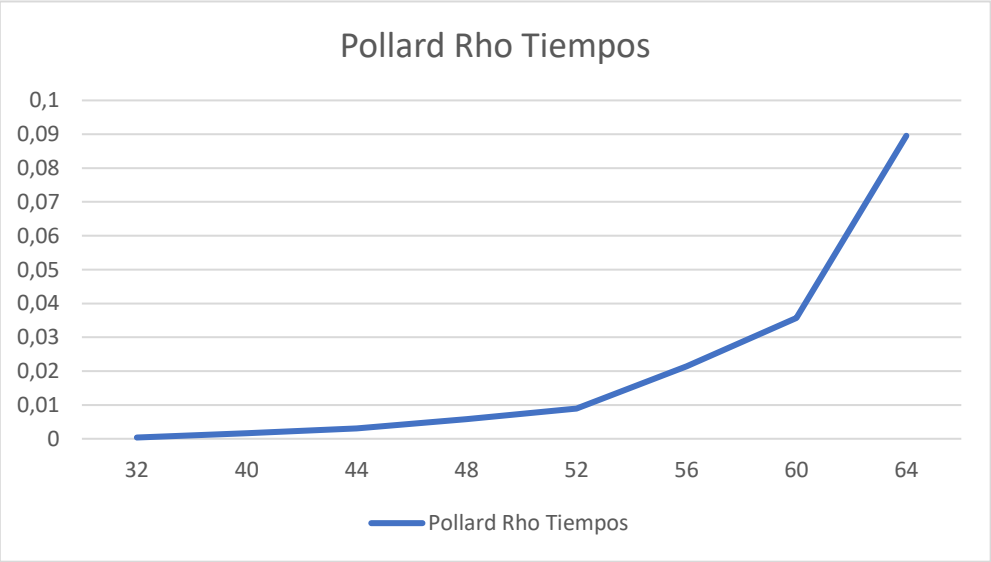


Ilustración 2. Algoritmo de Pollard Rho pruebas simples

Ya en base a la reducción del tiempo que se puede observar en los resultados se puede intuir que este algoritmo será más eficiente que el anterior para números grandes, aunque esto se analizará más en detalle en el estudio avanzado.

2.2.3. Pollard p-1

Este algoritmo se basa en la idea de que, si p es un factor primo de N y a es un entero aleatorio, entonces a^k es congruente a 1 (mod p) para algún k que es un múltiplo de p−1.

Seleccionamos un entero aleatorio a entre 2 y N−1.

Calculamos el máximo común divisor (MCD) de a^k−1 y N utilizando el algoritmo de Euclides. Si el MCD no es igual a 1, entonces hemos encontrado un factor no trivial de N.

Si el MCD es igual a 1, incrementamos a y repetimos el proceso.

```
Require: Un número entero positivo compuesto n
Ensure: Un factor de n
Escoger A aleatorio tal que 2 ≤ A ≤ n − 1
if 1 < mcd(A, n) < n then
    return mcd(A, n)
end if
k = 2;
while True do
    A = Ak mod n
    d = mcd(A − 1, n)
    if 1 < d < n then
        return d end if
    if d = n then
        return False
    end if
    k + +
end while
```

Experimentación Básica

En este caso también se mostrará un ejemplo del correcto funcionamiento del algoritmo:

763618151374994897 => 919522771

En este algoritmo se hace efectiva una restricción de tiempo impuesta en el propio código. Esta restricción impone que no será posible superar los 5 minutos por prueba, por lo que si en este tiempo no se ha encontrado una solución se devolverá False.

Es necesario tener esto en cuenta ya que la media de tiempo se hará únicamente con los tiempos para las pruebas para las que se encontró resultado, lo que podría dar una idea equivocada de los resultados obtenidos (Ilustración 3).

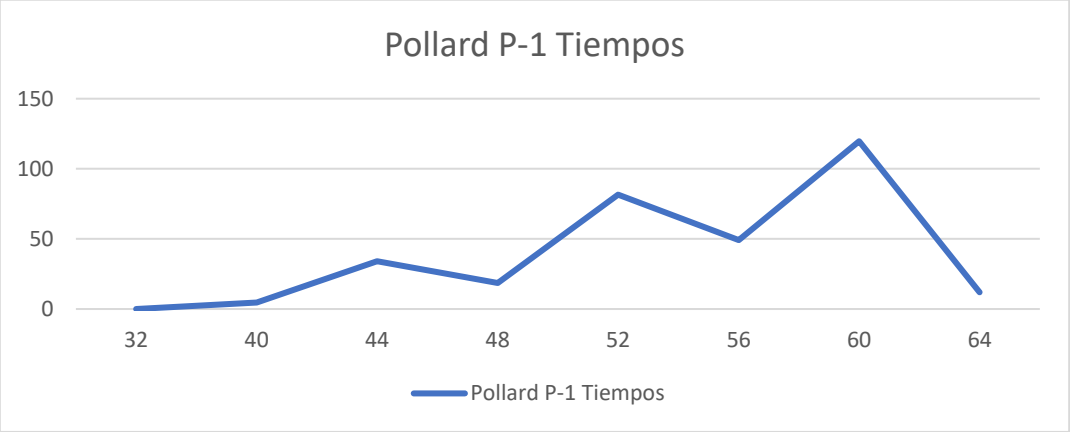


Ilustración 3. Algoritmo de Pollard P-1 pruebas simples

En la gráfica de la ilustración 3 se puede ver como para tamaño 64 bits tarda mucho menos que para 60, pero esto es debido a lo que ya comenté anteriormente acerca de que solo se tienen en cuenta las pruebas encontradas correctamente.

Como se puede observar en la tabla 1 para tamaño de 64 bits únicamente se encuentran 4 pruebas resueltas de las 10 que se han hecho, ya que un tiempo superior a 300 segundos implica que se ha alcanzado el timeout. Además, las 4 pruebas resueltas es casualmente en un tiempo muy reducido, por lo que al no tener en cuenta los fallos para la media, estas salen así de difusas.

Bits	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10
52	183.39	0.6469	69.092	217.30	88.909	1.283	0.062	3.214	170.54	>300
60	1.2668	>300	0.1432	197.309	>300	>300	282.90	>300	157.72	78.455
64	0.0757	0.0003	46.017	>300	>300	>300	>300	>300	1.4469	>300

2.2.4. Lenstra

A diferencia de los algoritmos mencionados anteriormente, el algoritmo de Lenstra se basa en la teoría de curvas elípticas. Utiliza una aproximación del algoritmo de Pollard (p-1) con el grupo definido por una curva elíptica.

Se elige aleatoriamente una curva elíptica sobre un cuerpo finito y un punto inicial en esa curva. Se realizan operaciones algebraicas en la curva elíptica, como la suma de puntos, para generar una secuencia de puntos. Si durante el proceso se encuentra un punto no trivial (un punto que no es el elemento neutro de la suma), se puede utilizar para obtener un factor no trivial del número que se está intentando factorizar.

Algoritmo de Lenstra
Require: Un número entero positivo compuesto n
Ensure: Un factor primo de n
Fijar cota B // Suponer que el orden de la curva es B-smooth
Escoger una CE módulo n cualquiera $y^2 = x^3 + ax + b \bmod n$.
Escoge un punto cualquiera de la curva $P = (x_0, y_0)$
k = 2
while k ≤ B **do**
    Obtener P = kP mod n #Idea de exponenciación de cuadrados sucesivos
    if es posible **then**
        k + +
    else
        return d #Hemos encontrado un t tal que mcd(t, n) = d
    end if
end while
if k > B **then** Probar con otra curva y otro punto inicial **end if**

```
---- kP mod n ---- (Exponenciación por cuadrado sucesivos)
Require: Enteros k y n
Require: p, un punto de una CE módulo n
Ensure: kA mod n
r = punto en el Infinito
while k > 0:
   if p[2] > 1:            #Esto es un flag de control
      return p
   if k % 2 == 1:          #Comprueba si el bit menos significativo es 1
      r = elliptic_add(p, r, a, b, m)
   k = k // 2              #Un bit a la izquierda en binario
   p = elliptic_add(p, p, a, b, m)
return r
```

```
Cálculo de X e Y
Require: Enteros k y n
Require: p y q, dos puntos de una CE módulo m
Ensure: kA mod n

if p[0] == q[0]:                  # Recta tangente a un punto
   if (p[1] + q[1]) % m == 0:
      return 0, 1, 0              # Infinity
   num = (3 * pow(p[0],2) + a) % m
   denom = (2 * p[1]) % m

else:                             # Secante a dos puntos
   num = (q[1] - p[1]) % m
   denom = (q[0] - p[0]) % m

inv = inverso(denom, m)
x = (pow((num * inv),2) - p[0] - q[0]) % m     # Xr = m^2 -xp -yq
y = ((num * inv) * (p[0] - x) - p[1]) % m      # Yr
return x, y, 1
```

Experimentación Básica

Para Lenstra, pese a ser bastante más complejo, se han realizado las mismas pruebas iniciales que para el resto, pues son más que suficiente para su objetivo.

Lo primero, al igual que en los casos anteriores, demostrar el correcto desempeño del algoritmo mediante un ejemplo resuelto:

17295603783688662937 => 4163610511

En este algoritmo se requiere de cierta configuración previa, ya que cada cierto número de iteraciones es recomendable cambiar la curva y el punto de inicio, es decir, reiniciar el algoritmo, si es que no se ha encontrado aun solución.

Este parámetro, al que yo he llamado "*limit*" en el seudocódigo, debe ser relativamente pequeño, por lo que, tras informarme acerca de cuanto podía ser relativamente pequeño, decidí dejarlo como 5000.

Ahora se mostrarán los tiempos medios de respuesta para este algoritmo.

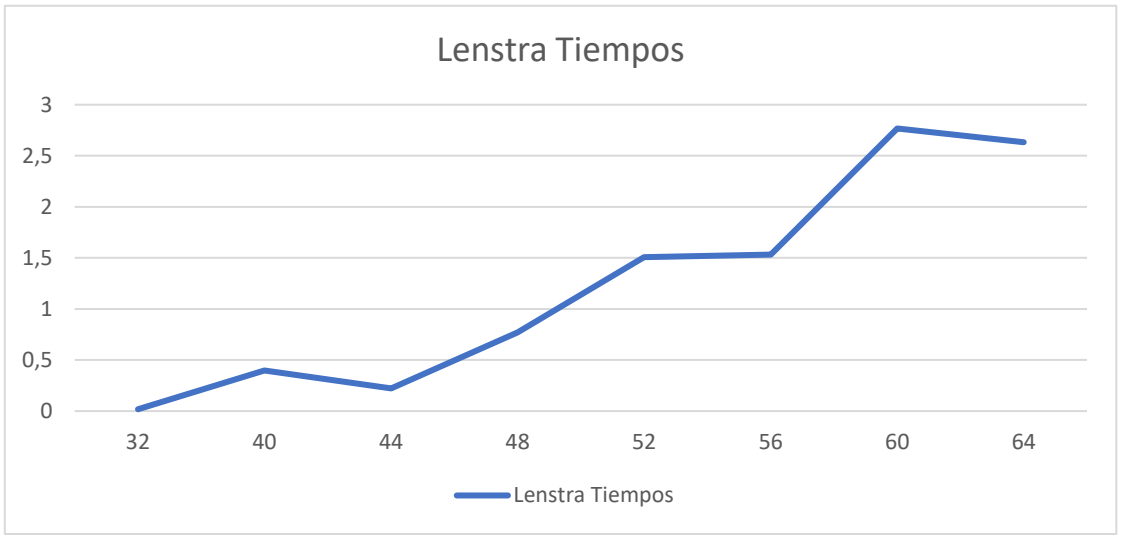


Ilustración 4. Algoritmo de Lenstra pruebas simples

Al contrario que con el algoritmo anterior, las discontinuidades de esta gráfica no se deben a pruebas no resueltas, ya que, para los tamaños utilizados en estas pruebas, se obtiene un acierto del 100%.

En este caso, las perturbaciones de la gráfica se deben únicamente a la aleatoriedad de este algoritmo y la reducida cantidad de pruebas realizadas.

2.3. Experimentación extendida

En cuanto a la experimentación extendida se pretende, no solo aumentar el número de pruebas para cada tamaño, sino aumentar también tamaños de prueba, pasando a realizar ahora 100 pruebas para cada uno de los siguientes tamaños:

[24,32,40,44,48,52,56,60,64,68,72,76,80,92,104]

De esta forma, al aumentar el número de pruebas por tamaño se aumentará la precisión de la media, obteniendo con esto graficas más certeras. Por otro lado, al aumentar la cantidad de tamaños de pruebas se consigue ampliar el rango de estudio, permitiendo observar cómo se comportan los algoritmos para tamaños de pruebas algo más grandes.

En este apartado se expondrán primero los resultados obtenidos para cada uno de los algoritmos con estas pruebas, comentando cualquier anomalía o rasgo de interés para el estudio. Más tarde se realizará una comparación grafica entre los 4 algoritmos, de manera que se pueda observar la eficiencia de cada uno de ellos en base a los tiempos de cálculo medio de cada uno.

2.3.1. Fermat

Tenemos aquí la gráfica de los tiempos medios para el algoritmo de Fermat en las pruebas extendidas.

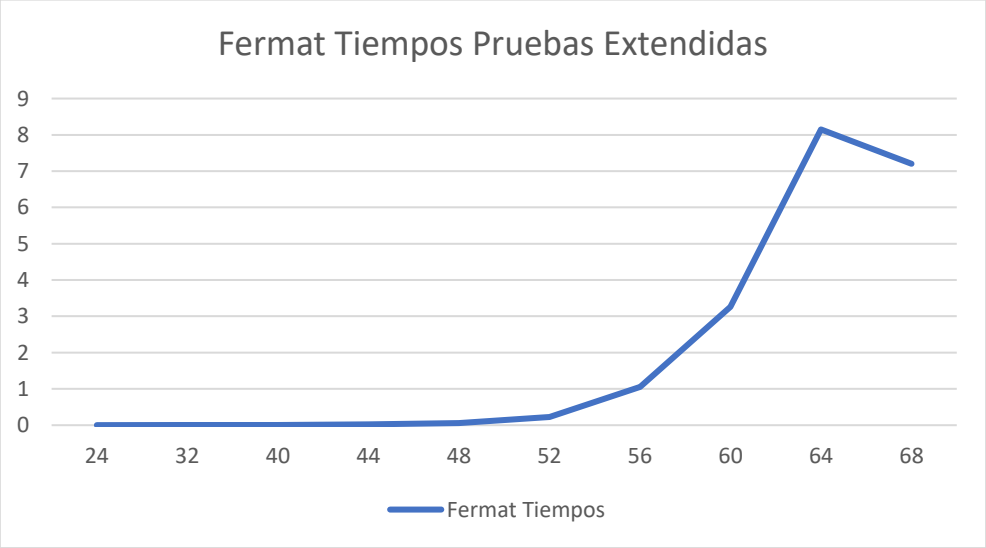


Ilustración 5. Algoritmo de Fermat pruebas extendidas

Puede observarse en la gráfica como, claramente, hay un error en los resultados obtenidos para las pruebas de tamaños 68 bits, ya que la gráfica estaba siguiendo una progresión exponencial hasta ese punto.

Este error se debe realmente a una cuestión puramente técnica. Cuando el algoritmo realiza la comprobación de si la raíz cuadrada de b es un entero para detectar el haber encontrado una solución, véase así:

while not $\sqrt{b}$ **es entero**:

Esta comprobación tiene un error bastante grave que no he sido capaz de solventar de ningún modo, y es que cuando esta $\sqrt{b}$ no es un entero, pero se aproxima mucho (ejemplo: 48,0000000345), python lo detecta como entero y devuelve esa solución como validad, cuando realmente no lo es. Esto hace que para números grandes, donde el tiempo debería ser cada vez mayor, sea cada vez menor, ya que encuentra más casos de decimales cercanos al entero.

Para intentar solventar esto se han probado las comprobaciones:

- is_integer()
- ((a - math.sqrt(b))*(a + math.sqrt(b))) == n
- (((a - math.sqrt(b))*(a + math.sqrt(b)))-n) != 0

Finalmente, tras desistir de seguir probando métodos sin éxito, se decidió implementar un *assert* que aviase cuando el algoritmo devolviese un resultado incorrecto y se detuviese ahí el programa, obteniendo ahora respuestas validas únicamente de hasta tamaño 64.

2.3.2. Pollard-rho

Aquí tenemos los tiempos medios obtenidos para el algoritmo Pollard Rho en las pruebas extendidas.

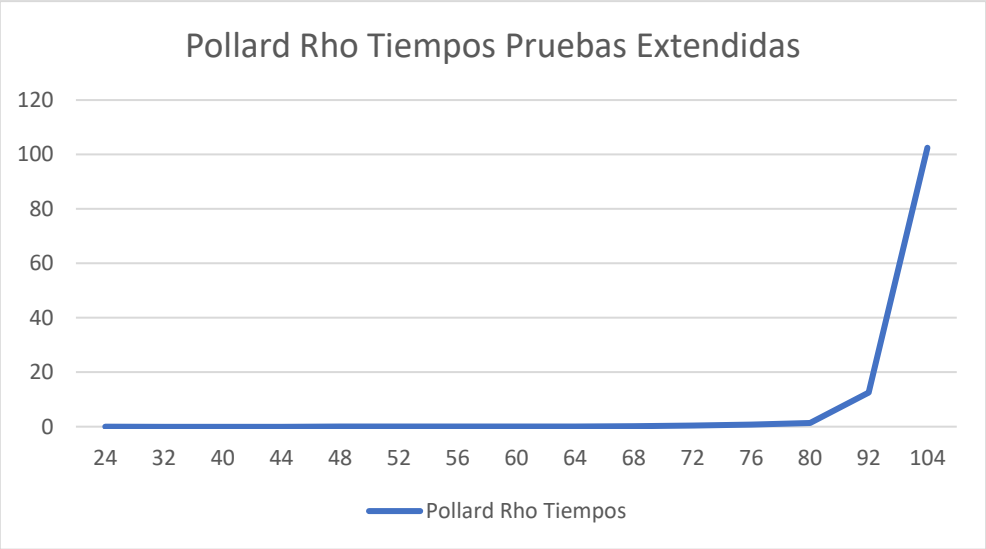


Ilustración 6. Algoritmo de Pollard Rho pruebas extendidas

A estos resultados poco hay que añadir, es sin duda mucho más eficiente que el algoritmo anterior y ya adelanto que también mejor que los dos siguientes.

Este algoritmo ha demostrado ser no solo eficiente, sino también capaz de factorizar números de gran tamaño, llegando en estas pruebas hasta tamaños de 104 bits en un tiempo medio de 100 segundos.

Si bien es cierto que las pruebas se detuvieron aquí ya que por la inercia de la curva para valores de tamaño superior a 104 el tiempo haría algo más que duplicarse, llegando a superar los 10 minutos por iteración.

Nótese también que la brusquedad de la inclinación de la curva de tiempo para los dos últimos tamaños se debe a que hasta ese punto los saltos de tamaños eran de 4 en 4, y desde ahí son de 12 en 12.

2.3.3. Pollard p-1

Este algoritmo ya dio ciertos problemas en su ejecución para las pruebas simples, generando una gráfica con bastantes perturbaciones debido a la cantidad de pruebas fallidas que se generaban. En estas pruebas extendidas no es diferente.

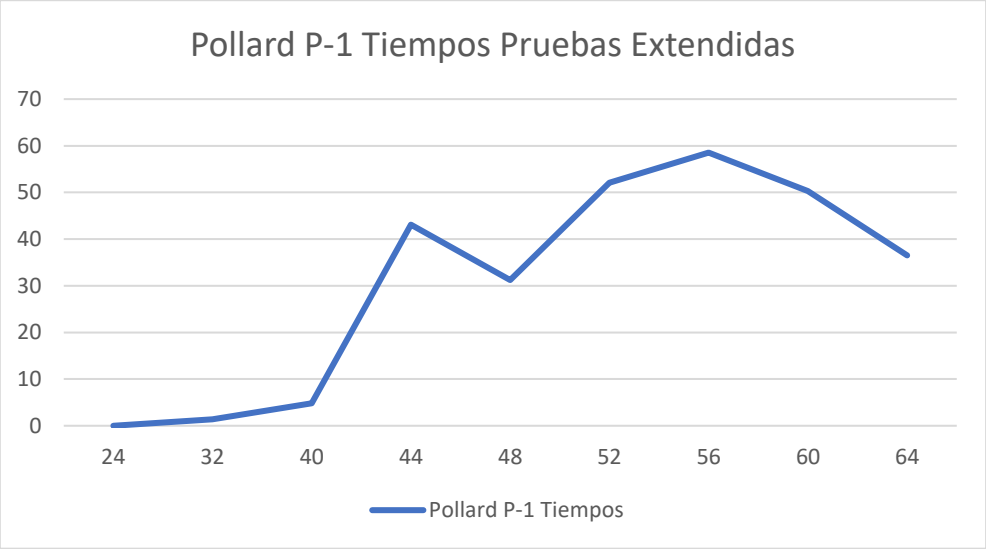


Ilustración 7. Algoritmo de Pollard P-1 pruebas extendidas

Al igual que en las pruebas simples se genera una gráfica con poco sentido, y el motivo es el mismo, la cantidad de pruebas que llegan a resultado nulo, pues a mayor es el tamaño del número a factorizar mayor es también el número de pruebas que no puede resolver el algoritmo en el tiempo puesto como límite.

Este algoritmo se estuvo ejecutando durante más de 32 horas, con un "*timeout*" de 15 minutos por prueba.

Se incluyo también para estas pruebas una condición extra de parada, que especificaba que cuando el número de fallos superase los 50 no se realizasen más pruebas. Esto se hizo para ahorrar tiempo de ejecución, pues una vez que para un tamaño se superasen los 50 fallos, con mucha probabilidad, para tamaños más grandes habría más fallos. Y unas pruebas en las que más del 50% de son fallos no sirven de mucho.

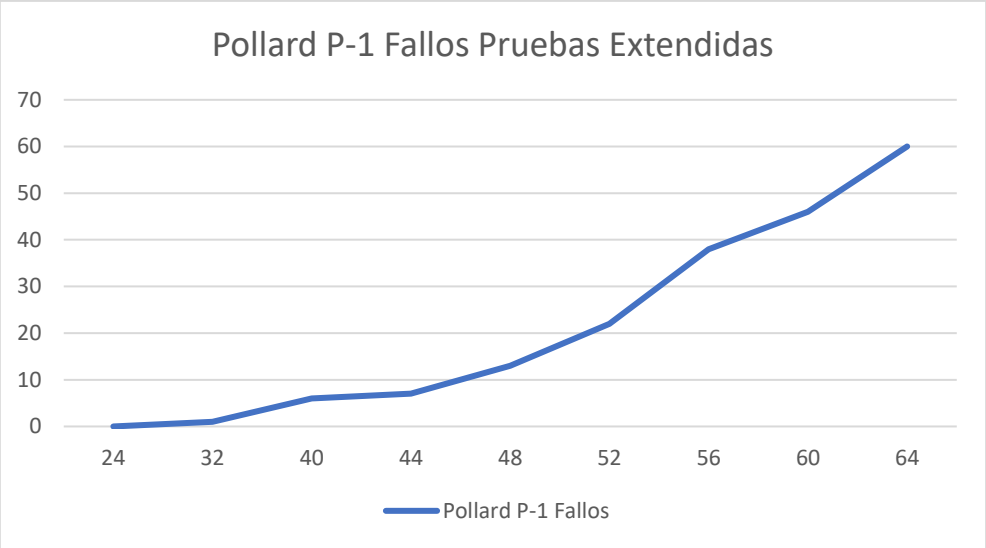


Ilustración 8. Algoritmo de Pollard P-1 pruebas extendidas Fallos

Se observa en la gráfica de la ilustración 8, como a medida que aumenta el tamaño de las pruebas aumenta también el número de errores, llegando a 60 pruebas sin resolver para tamaño 64.

2.3.4. Lenstra

En el siguiente grafico se muestran los tiempos medios obtenidos para Lenstra.

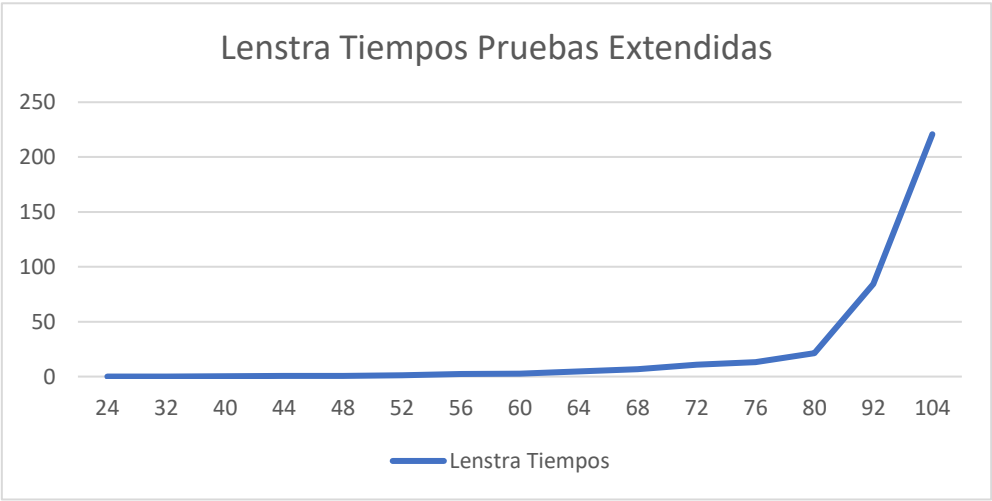


Ilustración 9. Algoritmo de Lenstra pruebas extendidas

Se observa en el gráfico de la ilustración 9 un perfecto crecimiento exponencial, demostrando mi hipótesis de que las perturbaciones del grafico generado por las pruebas simples de Lenstra (ilustración 4), se deben a la reducida cantidad de pruebas realizadas.

Se implementó Lenstra con la idea inicial de comprobar el funcionamiento de un algoritmo algo más complejo, esperando que superase con creces al resto, pero al observar los resultados no son muy diferentes de los obtenidos para Pollard Rho, siendo sus tiempos incluso peores, aunque esto se verá algo más en detalle en el siguiente apartado.

2.4. Comparación de algoritmos

Se tratará en este apartado de exponer la comparación entre estos 4 algoritmos desarrollados, aunque ya se han realizado algunas de estas comparaciones durante el desarrollo del estudio.

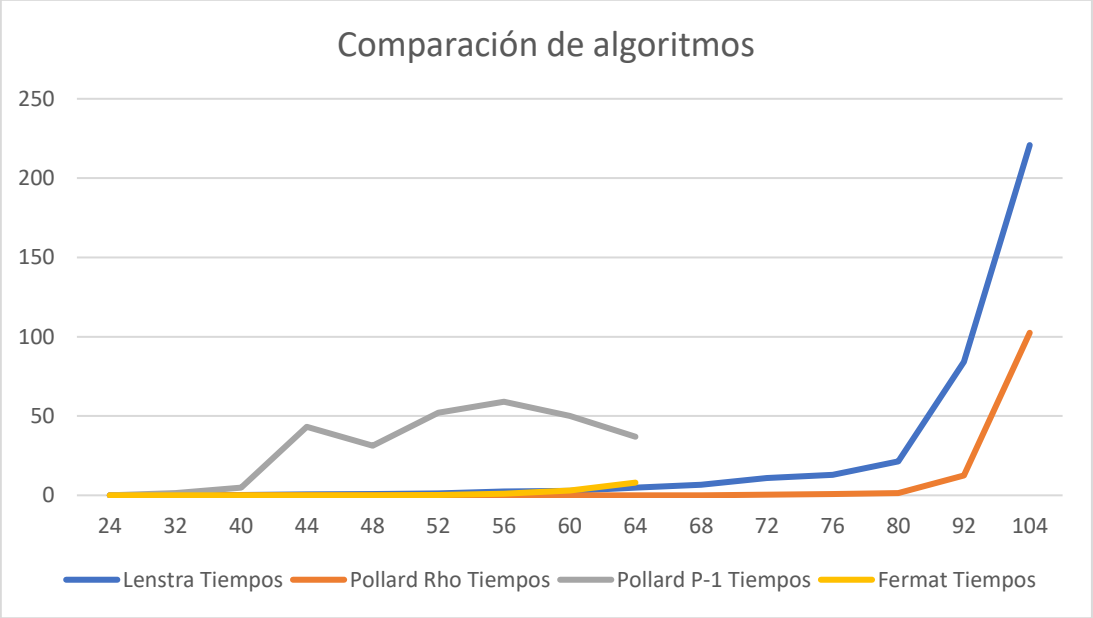


Ilustración 10. Comparación de tiempos de algoritmos

Su puede observar claramente en la ilustración 10 como el algoritmo más eficiente en cuanto a tiempos de ejecución es Pollard Rho, seguido de Lenstra.

No solo podemos deducir que estos dos algoritmos (Pollard Rho y Lenstra) son los más eficientes de los estudiados, sino que son los dos únicos que han podido llegar a resolver números de tamaño 104.

3. Cálculo del logaritmo discreto

Al igual que con la factorización de enteros, el calculo del logaritmo discreto es utilizado para criptografía de clave pública. Esto se hace debido a la dificulta que supone el calculo del logaritmo discreto de un número de gran tamaño.

Considerando un entero n, el grupo multiplicativo Z_n^*, y un elemento α de Z_n^*. Dado $\beta \in Z_p^*$, se busca el valor k tal que: $\beta = \alpha^k \bmod n$. A este valor k se le conoce como logaritmo discreto de β (en base α).

Para el calculo del logaritmo discreto, también existen varios algoritmos, de los cuales, en este trabajo, se desarrollarán dos, el algoritmo Baby-Step Giant-Step y el de Pollard-Rho para cálculo de logaritmo discreto.

Para cada uno de estos algoritmos se realizarán las pruebas dispuestas en poliformat, comenzando por describir cada uno de los algoritmos, su implementación y unas breves pruebas iniciales para comprobar y analizar su funcionamiento.

Una vez terminada la implementación básica de estos algoritmos se realizará un estudio más en profundidad sobre cada uno de los algoritmos desarrollados, utilizando un conjunto de datos mucho más amplio, también disponible en poliformat.

3.1. Características del ordenador

La experimentación de este algoritmo se ha realizado en un Mini PC con las siguientes características:

- Procesador: Intel de procesador de 11.ª gen N5095 (2,9GHz)
- Núcleos: 4 – Procesos: 4
- Gráfica: No contiene
- RAM: 16 GB DDR4
- Memoria: 256 GB SSD
- Sistema Operativo: Ubuntu 20.04

3.2. Implementación básica

En cuanto a las pruebas que se realizarán sobre cada algoritmo en esta primera parte del estudio, serán muy simples, útiles únicamente para comprender funcionamiento de los algoritmos y comprobar el correcto funcionamiento de los mismo.

Respecto a las pruebas a realizar en este primer apartado, para cada algoritmo se utilizarán 2 pruebas para cada tamaño de números, siendo los tamaños a probar [8, 10, 16, 24, 32, 40, 42, 48, 56] bits.

El hecho de utilizar únicamente 10 pruebas y un tamaño de bits no demasiado grande es porque, como ya se dijo anteriormente, el objetivo de este primer apartado de implementación base es el de estudiar el funcionamiento de los algoritmos.

3.2.1. Baby-Step Giant-Step

Dado *p* primo, el grupo Z_p^* y números $\alpha, \beta \in Z_n$, para obtener k $\beta = \alpha^k \bmod p$, se considera que:

Si $n = \sqrt{p-1}$, entonces $k = qn + r$, por lo que:

$$\alpha^k \equiv \alpha^{qn+r} \equiv \alpha^{qn}\,\alpha^r \pmod p,$$

por lo que, si podemos obtener α^{-qn}, entonces:

$$\alpha^k\,\alpha^{-qn} \equiv \alpha^{qn}\,\alpha^{-qn}\,\alpha^r \equiv \alpha^r \pmod p$$

Algoritmo Baby-Step Giant-Step
Require: Un valor primo p, un generador α de $Z * p$
Require: Un valor β de Z_p^*
Ensure: k tal que $\beta \equiv \alpha^k \pmod p$
$n = \lceil \sqrt{p} \rceil$
for r = 0 **to** n − 1 **do**
  Almacena en T el par $\langle r, \alpha^r \bmod p \rangle$ indexado por $\alpha^r \bmod p$
end for //El tiempo de acceso a la tabla T debe ser constante
Calcula $\alpha^{-n} \bmod p$ y asigna γ = β
for q = 0 **to** n − 1, q ++ **do**
  if Existe un par ⟨j, γ⟩ en la tabla T **then**
    $k = qn + j$
  end if
  $\gamma = \gamma\alpha^{-n} \bmod p$
end for

Experimentación Básica

En cuanto al funcionamiento del algoritmo, creo adecuado mostrar un ejemplo concreto del funcionamiento del mismo:

p = 157943476947589, α = 18, β = 64105273102320, resultado = 84789793767271

Para este algoritmo concreto si que es necesario controlar y medir el uso de memoria, pues utiliza una lista a la que se van añadiendo pares de valores de 0 a n-1, por lo que, a mayor tamaño del punto, mayor tamaño tendrá la lista, y por tanto mayor espacio de memoria utilizará.

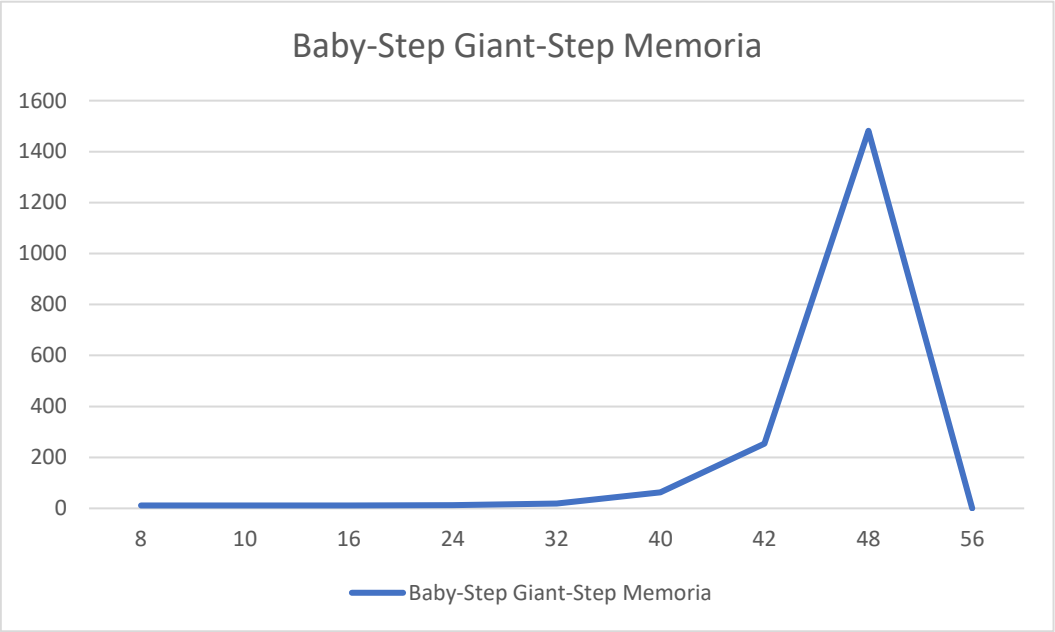


Ilustración 11. Algoritmo Baby-Step Giant-Step memoria

Se puede observar un crecimiento exponencial en cuanto al uso de memoria. Observando la gráfica de la ilustración 11, se puede ver un repentino descenso para el valor tamaño de 56 bits. Este descenso representa una interrupción del programa, ya que se ha superado la memoria RAM máxima del mismo.

Para tamaño de 42 bits se utilizan poco más de 200 megabytes, mientras que para 48 bits se utilizan aproximadamente 1,5 gigabytes. Teniendo en cuanta que la siguiente prueba supera en 8 bits el tamaño de esta, es de esperar que la memoria necesaria sea ligeramente superior a 16 gigabytes.

En cuanto al tiempo de ejecución también ha sido medido, ya que también es un factor relevante de cara a medir la eficiencia.

El tiempo (representado en la ilustración 12) sigue una curva de crecimiento muy similar a presentada por la memoria, y, evidentemente, presenta el mismo error para tamaño 56, pues al no haber podido completar la prueba por limitaciones de memoria, el tiempo no ha podido ser medido de forma correcta.

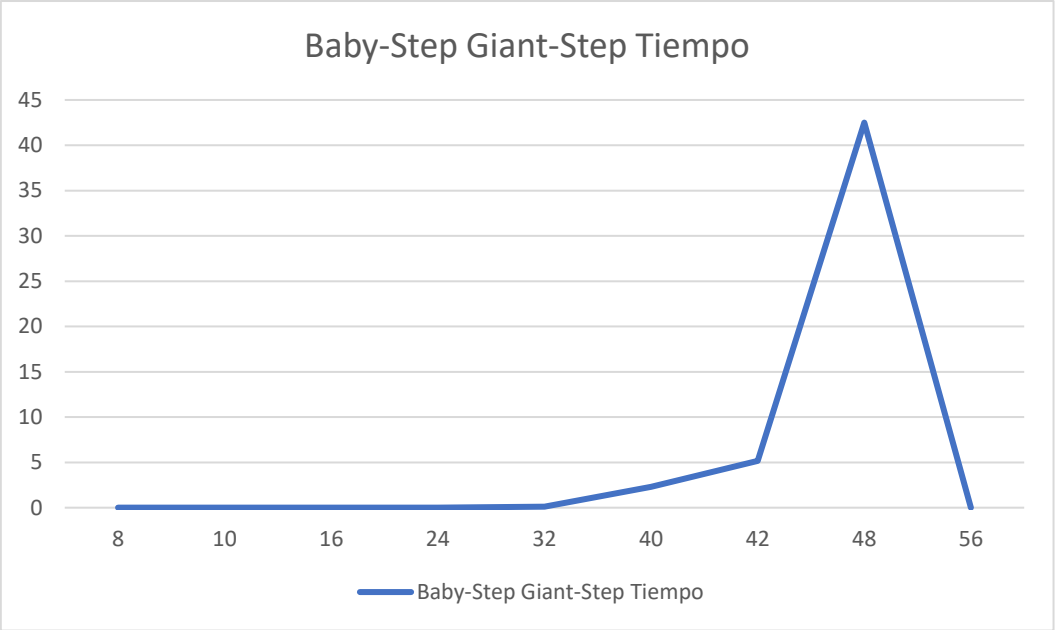


Ilustración 12. Algoritmo Baby-Step Giant-Step tiempo

En la experimentación extendida se tratará de estudiar con mayor precisión esta curva seguida por la memoria y el tiempo de este algoritmo.

3.2.2. Pollard-Rho para Logaritmo Discreto

Este algoritmo hace posible encontrar la solución con complejidad temporal O(√ p), sin los problemas de memoria que conlleva el algoritmo Baby-Step Giant-Step. Sin embargo, tal y como se vera más adelante, no siempre obtiene resultados. Pero antes introduzcamos de forma teórica este algoritmo.

Este algoritmo se basa en modificar iterativamente valores x_i, a_i y b_i que cumplen siempre que:

$$x_i = \alpha^{ai}\, \beta^{bi} \bmod p$$

Si se encuentran x_i y x_{2i} equivalentes, entonces podemos obtener el logaritmo discreto de β.

Para realizar el cálculo es necesario contar con o, el tamaño del grupo generado por α (el valor tal que $\alpha^o \bmod p = 1$).

Se clasifican los valores x_i en tres bloques de tamaño similar, aplicando una operación distinta para obtener el valor x_{i+1}

$$x_{i+1} = \begin{cases} \beta x_i \ mod\ p, & si\ x_i \in S_1 \\ x_i^2 \ mod\ p, & si\ x_i \in S_2 \\ \alpha x_i \ mod\ p, & si\ x_i \in S_3 \end{cases}$$

La modificación de los x_i implica modificar los valores asociados a_i y b_i para mantener la igualdad $x_i = \alpha^{ai}\ \beta^{bi}$ mod p.

Algoritmo Pollard-Rho

Require: Un entero p, un generador α de un subgrupo de Z_p^* de orden *o* y un valor *β* de < *α* >
Ensure: *k* tal que $\beta \equiv \alpha^k \pmod p$
a = b = aa = bb = 0
i = x = xx = 1
while i < p **do**
    x = operacion(x, a, b, p, o)
    xx = operacion (xx, aa, bb, p, o)
    xx = operacion (xx, aa, bb, p, o)
    if x == xx **then**
        if mcd(b − bb, *o*) ≠ 1 **then**
            return False
        end if
        return $(aa - a)(b - bb)^{-1} \bmod o$
    end if
end while
return False

Operacion

Require: Un entero p, un generador α de un subgrupo de Z_p^* de orden *o* y un valor *β* de < *α* >
Require: Valores *x, a* y *b* generados en el algoritmo.
Ensure: *x, a y b* actualizados.
x = (α^a * β^b) % p
switch_value = x % 3
if switch_value == 1:
    x = (x*β) % p; a = a; b = (b + 1) % (p-1)
elif switch_value == 0:
    x = (x*x) % p; a = (2*a) % (p-1); b = (2*b) % (p-1)
elif switch_value == 2:
    x = (x*α) % p; a = (a + 1) % (p-1); b = b
return x,a,b

Experimentación Básica

Para este algoritmo han surgido varios problemas a la hora de realizar las pruebas pues no se obtiene resultado para prácticamente ninguna de las pruebas.

Se comprobó que el algoritmo funcionase correctamente, y así es, las pocas soluciones que consigue obtener son correctas, véase el siguiente ejemplo:

$$p = 55243,\ \alpha = 22,\ \beta = 37205,\ orden = 27621 \Rightarrow 22484$$

Para las pruebas simples, puesto que solo eran 20 en total, se utilizó un timeout de 15 minutos y aun así solo se obtuvo resultado para 3 de esas 20 pruebas, y todas ellas en un tiempo bastante bajo, por lo que el tiempo de espera no es el problema.

No tiene sentido representar esto mediante una grafica por lo que se expondrán en el documento simplemente los tres aciertos obtenidos indicando su tiempo de ejecución.

Tamaño	p	α	β	orden	Resultado	Tiempo
10 bits	1109	19	274	277	206	0,0000937
10 bits	587	3	253	293	23	0.0001702
16 bits	55243	22	37205	27621	22484	0.4469106

3.3. Implementación extendida

En cuanto a la experimentación extendida, de igual forma que para el anterior problema, se pretende, no solo aumentar el número de pruebas para cada tamaño, sino aumentar también tamaños de prueba, pasando a realizar ahora 100 pruebas para cada uno de los siguientes tamaños:

$$[32,36,40,44,48,52,56,60,64,80,96]$$

De esta forma, al aumentar el número de pruebas por tamaño se aumentará la precisión de la media, obteniendo con esto graficas más certeras. Por otro lado, al aumentar la cantidad de tamaños de pruebas se consigue ampliar el rango de estudio, permitiendo observar cómo se comportan los algoritmos para tamaños de pruebas algo más grandes.

En este apartado se expondrán los resultados obtenidos para cada uno de los algoritmos con estas pruebas, comentando cualquier anomalía o rasgo de interés para el estudio.

3.3.1. Baby-Step Giant-Step

En estas pruebas trataremos de analizar de forma más precisa el aumento de memoria utilizada por el algoritmo para cada tamaño de problema.

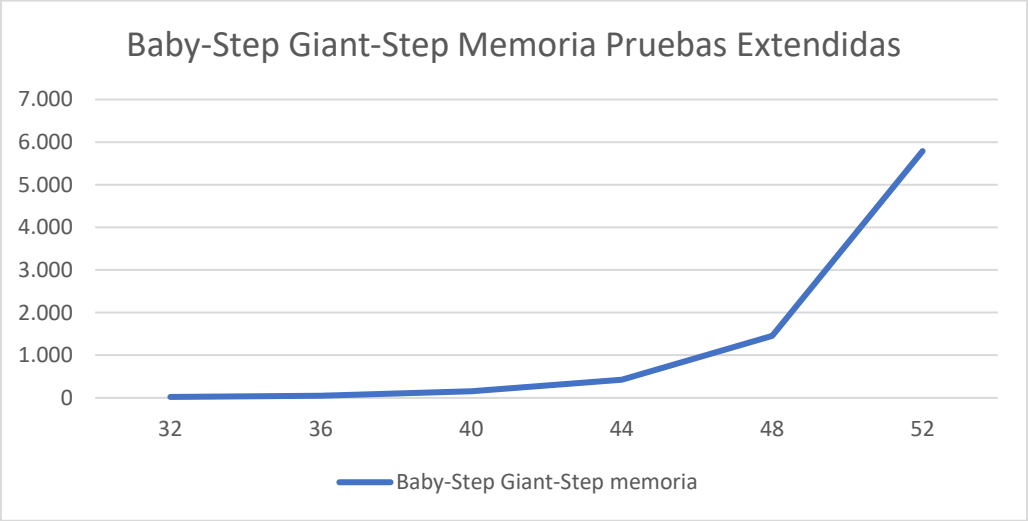


Ilustración 13. Algoritmo Baby-Step Giant-Step memoria Pruebas extendidas

Ahora sí, al haber realizado las pruebas con valores de tamaños con un crecimiento uniforme, podemos observar claramente y de forma precisa el crecimiento exponencial de la memoria utilizada por el algoritmo.

Teniendo en cuanta que, con cada aumento de 4 bits, la memoria utilizada aumenta más del triple y que el crecimiento es exponencial, podemos estimar, que, para 56 bits de tamaño, (calculo que ya no es capaz de realizar) necesitaría más de 20 gigabytes

En cuanto a las medias de tiempo obtenidas para estas pruebas tenemos la gráfica de la ilustración 14.

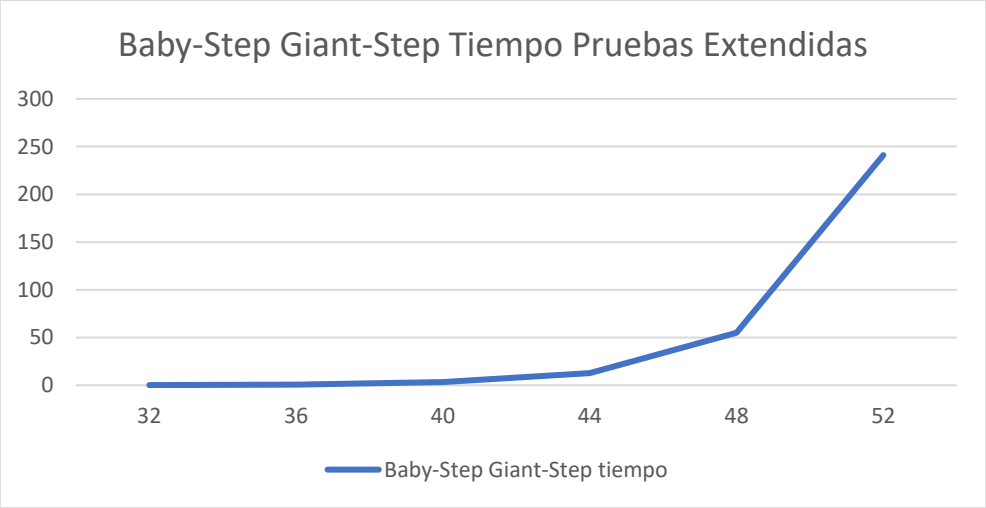


Ilustración 14. Algoritmo Baby-Step Giant-Step tiempo Pruebas extendidas

3.3.2. Pollard-Rho para Logaritmo Discreto

Como ya se comentó en la experimentación básica de este algoritmo, los resultados a las pruebas son prácticamente nulos ya para las pruebas básicas, ni hablar pues de los obtenidos para las pruebas extendidas.

En una primera instancia se lanzaron las pruebas extendidas utilizando un *timeout* de 15 minutos con la esperanza de que con tal cantidad tiempo encontrase al menos resultado para alguna, pero no obtuvo ni un solo resultado para el conjunto de 100 pruebas de tamaño 32 bits.

La ejecución de estas pruebas conllevó un total de 25 horas, pues, al alcanzar siempre el *timeout*, los 15 minutos se multiplicaron por las 100 pruebas a realizar.

Comprobar el resto de los valores con este *timeout*, y presuponiendo que siempre se alcanzará el *timeout*, es inconcebible ya que se tardaría otros 5 días en completar únicamente estas pruebas, por lo que se decidió reducir de forma drástica el tiempo de *timeout*, dejándolo finalmente en 3 minutos.

Tras, realizar de nuevo las pruebas, incluso con este nuevo *timeout*, y tras 30 horas de ejecución, no se obtuvieron resultados.

4. Conclusiones

A modo de conclusión general del trabajo, como ya se hizo en el trabajo anterior, me gustaría aprovechar este apartado del documento para aportar un comentario algo más personal acerca del estudio realizado.

En cuanto al problema de factorización decidí desarrollar y analizar primero los logaritmos sencillos, con la idea de más tarde analizar Lenstra y poder comentar como la complejidad de este conlleva unos mejores resultados. Para mi sorpresa, tras realizar todas las pruebas varias veces debido a mi incredulidad, decidí aceptar que uno de esos algoritmos sencillos supera en eficiencia al complejo.

Más tarde, en el estudio de algoritmos para el calculo del logaritmo discreto, encontré el problema del algoritmo Pollard-Rho. Mi planteamiento inicial para este apartado era poder comparar los algoritmos de Baby-Step Giant-Step y Pollard-Rho, pero tras horas de pruebas, el algoritmo de Pollard-Rho para logaritmo discreto no obtuvo ni un solo resultado. Esto me impidió estudiar este algoritmo correctamente y por consiguiente también poder compararlo con Baby-Step Giant-Step.

5. Repositorio GITHUB

En este repositorio se encuentra el código completo de todos los algoritmos, así como los resultados obtenidos para cada una de las pruebas mostradas en este documento Repositorio GitHub: https://github.com/Javi-GI/Practica5_CSD/tree/main/Practica_2

6. Bibliografía

Guion de la práctica,
https://poliformat.upv.es/access/content/group/DOC_34876_2023/Trabajos/MUCC%20-%20Criptologia%20-%20Criptoanalisis.pdf

Tema 10. Criptoanálisis de clave pública (2/2): Factorización,
https://poliformat.upv.es/access/content/group/DOC_34876_2023/Transparencias/MUCC%20-%20Criptolog%C3%ADa%20MUCC%20Tema-10-1.pdf

Tema 8. Criptoanálisis de clave pública (1/2): DL,
https://poliformat.upv.es/access/content/group/DOC_34876_2023/Transparencias/MUCC%20-%20Criptolog%C3%ADa%20MUCC%20Tema-08-1.pdf

Apéndices

fermat.py

```
def algoritmoFermat(n, limit):
    a = (math.trunc(math.sqrt(n)))+1
    b = pow(a,2) - n
    while not math.sqrt(b).is_integer():
        a = a + 1
        b = pow(a,2) - n
    return  (a - math.sqrt(b), a + math.sqrt(b))


def fermatAssert(n, limit):
    a = math.isqrt(n)
    b2 = a*a - n
    b = math.isqrt(n)
    count = 0
    while b*b != b2 and cronometro < limit:
        a = a + 1
        b2 = a*a - n
        b = math.isqrt(b2)
        count += 1

    p=a+b
    q=a-b
    assert n == p * q
    return p, q
```

pollard_rho.py

```
def algoritmoPollar(n, limit):
    a=b=randint(2, n-1)
    while cronometro < limit:
        a = (pow(a,2) + 1)%n
        b = (pow(b,2) + 1)%n
        b = (pow(b,2) + 1)%n
        p = math.gcd(a-b, n)
        if(p > 1 and p < n):
            return p
        if(p == n):
            return n
    return False
```

pollard_p1.py

```
def algoritmoPollar_1(n, limit):
    a=randint(2, n-1)
    if 1 < math.gcd(a, n) < n:
        return math.gcd(a,n)

    tiempoInicio = time.time()
    cronometro = 0
    k = 2
    while cronometro < limit:
        a = (pow(a,k))%n
        d = math.gcd(a-1,n)
        if(1 < d < n):
            return d
        if(d == n):
            return False
        k = k+1
        cronometro = time.time() - tiempoInicio
    return False
```

lenstra.py

```
def modular_inv(a, b):
    if b == 0:
        return 1, 0, a
    q, r = divmod(a, b)
    x, y, g = modular_inv(b, r)
    return y, x - q * y, g

def suma_eliptica(p, q, a, b, m):
    # Si el punto es infinito devuleve el otro
    if p[2] == 0: return q
    if q[2] == 0: return p
    if p[0] == q[0]: # Recta tangente a un punto
        if (p[1] + q[1]) % m == 0:
            return 0, 1, 0  # Infinity
        num = (3 * pow(p[0],2) + a) % m
        denom = (2 * p[1]) % m
    else: # Secante a dos puntos
        num = (q[1] - p[1]) % m
        denom = (q[0] - p[0]) % m
    inv, _, g = modular_inv(denom, m)

    # Arithmetic breaks. Imposible encontrar inverso
    if g > 1:
        #print("OCURRIO")
        return 0, 0, denom  # Failure

    x = (pow((num * inv),2) - p[0] - q[0]) % m # Xr = m^2 -xp -yq
    y = ((num * inv) * (p[0] - x) - p[1]) % m # Yr
    return x, y, 1


# Multiplicacion (repeated addition and doubling) la forma más eficiente de
multiplicación escalar
def multi_eliptica(k, p, a, b, m):
    r = (0, 1, 0)  # Infinito
    while k > 0:
        # Si el flag p este activo devolver
        if p[2] > 1:
            return p
        if k % 2 == 1:    # Si el número es impar, indica que en binario acabaria en
                          1 con lo cual se realiza la operacion. Si es cero no se hace
            r = suma_eliptica(p, r, a, b, m)
```

```
        k = k // 2
        #print(k)
        p = suma_eliptica(p, p, a, b, m)
        #print(p)
    return r

#----------- Algoritmo de Lenstra ----------
def lenstra(n, limit, timeout):
    cronometro = 0
    tiempoInicio = time.time()

    while cronometro < timeout:
        #----- Generar una curva sin anomalias y un punto --------
        mcd = n
        while mcd == n:
            p = randint(0, n - 1), randint(0, n - 1),1
            a = randint(0, n - 1)
            b = (pow(p[1],2) - pow(p[0],3) - a * p[0]) % n
            mcd = math.gcd(4 * pow(a,3) + 27 * pow(b,2), n)      # Comprobar
                                                                 existencia de
                                                                 discontinuidades

        # Si tenemos suerte podemos obtener resultado directo
        if mcd > 1:
            return mcd

        k = 2
        while k < limit:
            p = multi_eliptica(k, p, a, b, n)
            # Elliptic arithmetic breaks
            if p[2] > 1:
                return math.gcd(p[2], n)
            k = k + 1
        cronometro = time.time() - tiempoInicio

    return False
```

ld_baby_giant.py

```
def babyStepGiantStep(p, alpha, beta):
    n = math.ceil(math.sqrt(p))
    t = {}
    for r in range(n):
        t[pow(alpha, r, p)] = r

    alpha_inverso = pow(alpha, -n, p)
    gamma = beta
    for q in range(n):
        if gamma in t:
            j = t[gamma]
            k = q * n + j
            return k
        gamma = (gamma * alpha_inverso) % p

    return None
```

ld_pollard.py

```
def operacion(x, a, b, alpha, beta, p):
    x = (pow(alpha,a) * pow(beta,b)) % p
    switch_value = x % 3

    if switch_value == 1:
        x = (x*beta) % p
        a = a
        b = (b + 1) % (p-1)
    elif switch_value == 0:
        x = (x*x) % p
        a = (2*a) % (p-1)
        b = (2*b) % (p-1)
    elif switch_value == 2:
        x = (x*alpha) % p
        a = (a + 1) % (p-1)
        b = b

    return x,a,b
```

```
def pollard_rho(p, alpha, beta, o, limit):
    a = b = aa = bb = 0
    i = x = xx = 1

    cronometro = 0
    tiempoInicio = time.time()

    while i < p and cronometro < limit:
        x,a,b = operacion(x, a, b, alpha, beta, p)
        xx,aa,bb = operacion(xx, aa, bb, alpha, beta, p)
        xx,aa,bb = operacion(xx, aa, bb, alpha, beta, p)

        if x == xx:
            if math.gcd(b - bb, o) != 1:
                return False
            return ((aa - a) * pow(b - bb, -1, o)) % o
        i += 1
        cronometro = time.time() - tiempoInicio
    return False
```