Trabajo Criptoanálisis:
Criptología de clave pública

Silviu Valentin Manolescu – MUCC

Contenido

Índice de ilustraciones

Índice de tablas

1. Aspectos generales de los algoritmos desarrollados

Hay que mencionar que para todos los códigos empleados se ha usado un método general main para poder sacar la tablas y así poder hacer los análisis pertinentes (se encuentra en el anexo). Esto produce una tabla CSV que para todos los algoritmos desarrollados van a tener el mismo formato:

Tamaño	Número	Solución	Tiempo	Memoria
24	11830219	(3011.0, 3929.0)	0	60.29
24	6255727	(2243.0, 2789.0)	0	60.56
24	12214001	(3463.0, 3527.0)	0	60.59

Tabla 1: Ejemplo de tabla CSV con resultados

Obviamente en este caso al ser tamaño 24 la ejecución es tan rápida que resulta en un tiempo insignificante que ni la librería "time" de Python es capaz de apreciar. Otro detalle es que la tabla presentada es solo un extracto de la original de Fermat, que para este caso cuenta en realidad con más de 100 filas desde tamaño 24 hasta 76 y por cuestiones de espacio se decide no incluir dada la cantidad de algoritmos y pruebas a mostrar. Finalmente, para medir el tiempo y memoria en los códigos de los algoritmos desarrollados, que se encuentran todos en el anexo, se han añadido las siguientes líneas (no en este concreto orden):

1. *start_timer = time.time()*
2. *if(time.time() - start_timer > 1200): return None*
3. *global memory*
4. *memory = round(psutil.Process(os.getpid()).memory_info().rss / 1024 ** 2, 2*

Con esto se puede medir la cantidad de memoria y en caso de que en una ejecución el algoritmo no encuentre una solución en al menos **20 minutos** procede a devolver un valor Nulo. Esto es aplicable a TODOS LOS ALGORITMOS que se van a desarrollar de ahora en adelante. La justificación de haber elegido 20 es que hay problemas que pueden resultar muy grandes para ciertos algoritmos según el tamaño y además no es un timeout que provoque que se tenga que dejar demasiadas horas el ordenador encendido.

Las especificaciones del equipo empleado en las pruebas se corresponden con un Portátil Lenovo Legión que cuenta con las siguientes características:

Procesador: AMD Ryzen 7 5800H (8 núcleos, N.º de subprocesos 16, Frecuencia Base 3.2 GHz)

Tarjeta Gráfica: Nvidia RTX 3060

Almacenamiento SSD: WDC PC SN730 SDBPNTY-1T00-1101 (1 Tb)

RAM: 16 GB DDR4 (3200 MHz)

Sistema Operativo: Windows 11

Para programar todo este código se ha escogido Python como lenguaje y Visual Studio Code como editor.

2. Factorización de enteros

Los algoritmos de factorización que se van a desarrollar en los próximos capítulos como Fermat, Pollard-rho, Pollard p-1 y Lenstra son relevantes en el contexto RSA a causa de que este se basa en la idea de que factorizar números grandes es una tarea sumamente compleja computacionalmente hablando.

En este sentido, si alguien pudiese factorizar el número n, empleado en RSA, se podría calcular la clave privada a partir de la clave pública y así romper el cifrado que se utiliza.

Todos estos algoritmos de factorización que se van a analizar ofrecen métodos alternativos para intentar factorizar números grandes con mejor eficiencia que algunas aproximaciones clásicas o brutas. Sin embargo, se tiene que comprender que cada uno tendrá sus virtudes como debilidades y eso es lo que se va a intentar mostrar poniéndolos a prueba en una serie de escenarios ante números n cada vez más grandes.

2.1 Implementación y experimentación base de Fermat

El algoritmo de Fermat propone una factorización basada en que n tiene dos factores que son próximos entre sí, por lo tanto, se puede aproximar a raíz de n. Por ello que a partir de esta raíz se quiere encontrar un cuadrado perfecto en su condición de bucle.

Para la implementación de Fermat se propone de este pseudocódigo:

```
1. def fermat(n):
2.     a = math.ceil(math.sqrt(n))
3.     b = a^2 - n
4.     while not b == math.isqrt(b) ^ 2:
5.         a += 1
6.         b = a^2 – n
7.         return (a - math.sqrt(b),  a + math.sqrt(b))
```

Como se puede observar se trata de una aproximación directa con el pseudocódigo propuesto en las diapositivas de clase, donde la clave reside en la línea 4 de la condición del bucle while. Aquí lo que se hace es llamar a la función *isqrt(b)* que devuelve la raíz cuadrada de b (en caso de que no sea exacta, por ejemplo, la de 10, se devuelve el valor truncado, es decir 3) y se eleva al cuadrado, comprobando que sea igual al valor original b. Esta aproximación en Python lo hace ideal para comprobar que se emplee un cuadrado perfecto.

Aunque cabe mencionar que se han probado otras condiciones del bucle previamente de conocer *isqrt(b):*

```
while not math.sqrt(b) * math.sqrt(b) == b:

while not math.sqrt(b).is_integer():
```

Estas aproximaciones resultaron incorrectas, la primera que hace la raíz cuadrada de b al cuadrado acaba devolviendo factores con muchos decimales. La segunda solución que se había pensado era hacer la raíz cuadrada y comprobar que fuese un resultado de tipo entero, es decir, exacto. A valores de tamaño de n pequeños los resultados eran correctos, ya que al multiplicar p*q resultaba n, sin embargo, a mayores valores como tamaño 68 empezaba a desviarse. Esto provocaba que para tamaños como 68 para n = 114216477245188960489 con una p y q reales (9363970571, 12197440859) diese un valor incorrecto (10265754278, 11125970304), resultando en 114216477245188960512. Con ello se tiene una desviación de 114216477245188960512 – 114216477245188960489 = 23 por encima del valor correcto. Se piensa que este problema de desviación ocasionada es muy probable que se deba por falta de precisión en los cálculos de los métodos empleados para números de gran tamaño. Por ello, usar la función math.isqrt(b) se ha elegido la alternativa definitiva al presentar soluciones exactas.

Visto las cuestiones generales del código se va a proceder a hacer un análisis de los resultados obtenidos desde tamaño 24 hasta 76 con 10 pruebas para cada uno que ha llevado más de 1.5 horas (archivo base de pruebas de factorización). En primer lugar, el tiempo sigue una evolución creciente exponencial a medida que aumenta el tamaño y eso se puede observar gracias al siguiente gráfico:

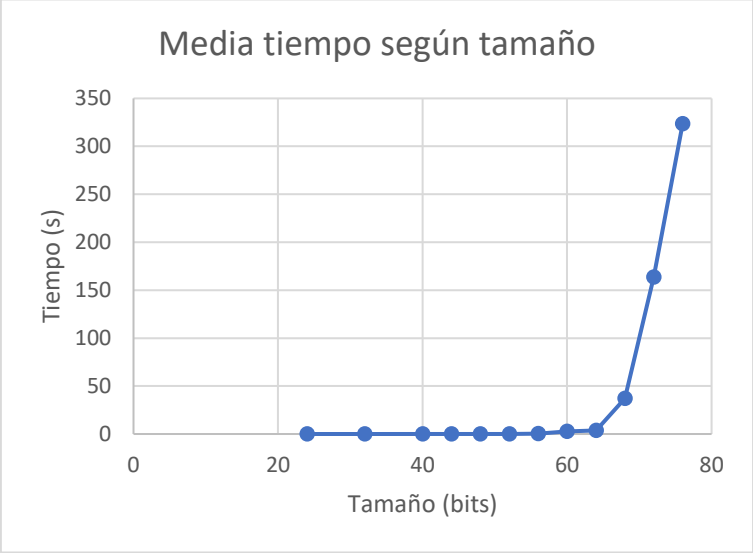

Ilustración 1: Media de tiempo según tamaño para Fermat ejecución base

Por su parte, la memoria no experimenta variación alguna a pesar del crecimiento del tamaño (se mantiene en 60 MB). Esto es lógico puesto que no hay ninguna estructura de datos como un diccionario que vaya almacenando gran cantidad de valores.

Otro aspecto interesante es que se ha observado una correlación entre el tiempo y la diferencia de p con q. Para las veinte pruebas de tamaño 72 y 76 se tienen estos resultados:

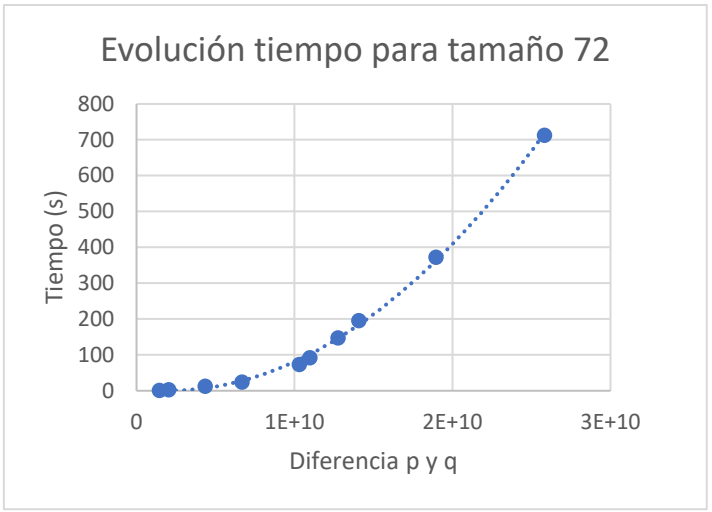

Ilustración 2: Evolución del tiempo según la diferencia de p y q, tamaño 72

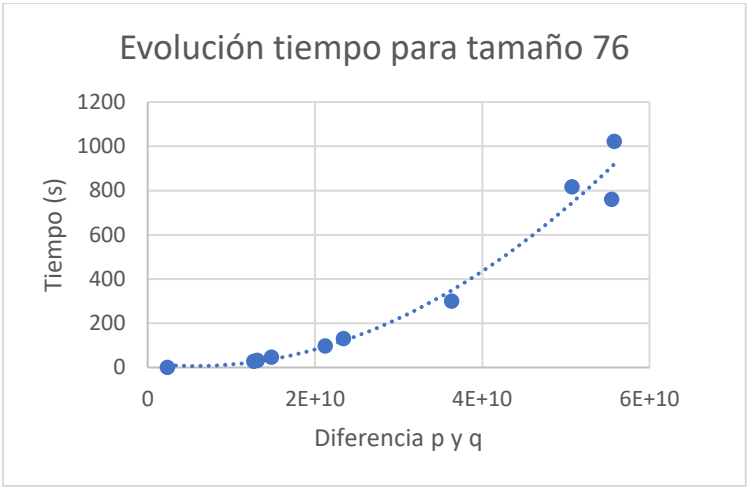

Ilustración 3: Evolución del tiempo según la diferencia de p y q, tamaño 72

Como se puede observar de las 20 pruebas, al agregar una línea de tendencia que intente pasar lo más cercano a los puntos esta sigue un patrón creciente. Por lo tanto, se puede concluir que **a mayor diferencia entre los factores p y q más tiempo tardará Fermat en encontrar la solución.**

2.2 Implementación y experimentación base de Pollard-rho

El pseudocódigo es el siguiente:

```
1     pollard_rho_factorizacion(n):
2       a = b = Generar número aleatorio de 2 a n-1
3       while True:
4         a = (a^2 + 1) modulo n
5         b = (b^2 + 1) modulo n
6         b = (b^2 + 1) modulo n
7         p = MCD(a-b, n)
8         if(p > 1 y p < n):
9           Devolver p
10        if(p == n):
11          Devolver n
```

Para la implementación del algoritmo de Pollard-rho también se ha seguido una aproximación directa con las diapositivas dada su baja complejidad. Se ha usado la librería "random" para emplear la función "radint" que genera números aleatorios entre 2 y n-1, así como "math.gcd" para el Máximo Común Divisor. Esto simplifica el desarrollo del código de Pollard-rho al no tener que crear desde cero funciones básicas esenciales.

En las pruebas realizadas de tamaño 24 a 92 han obtenido los siguientes resultados de tiempo:

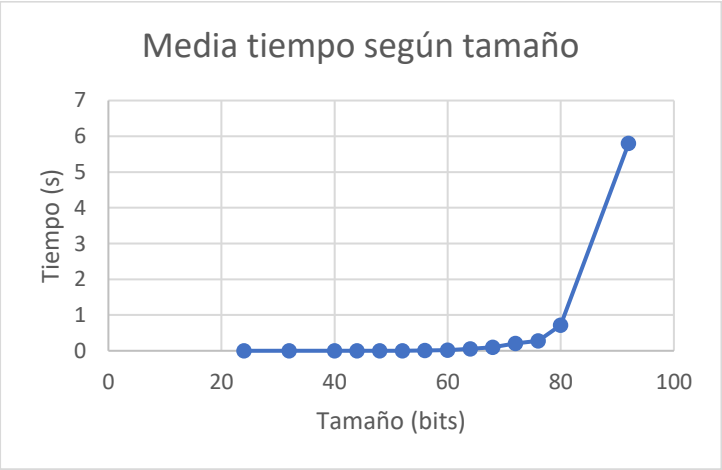

Ilustración 4: Media de tiempo según tamaño para Pollard-rho ejecución base

Como se puede observar, Pollard-rho tiene un mejor comportamiento que Fermat en estos resultados preliminares al tener un tiempo mucho menor. Sin embargo, se observa un crecimiento muy similar a Fermat que **a tamaños grandes puede resultar en un problema similar**.

Por su parte, la memoria no experimenta variación alguna a pesar del crecimiento del tamaño. (60 MB constantes)

2.3 Implementación y experimentación base de Pollard p-1

El pseudocódigo es el siguiente:

```
1     pollard_P_1(n):
2       a = Generar número aleatorio de 2 a n-1
3       if(MCD(a, n) > 1 y MCD (a, n) < n): Devolver MCD(a, n)
4       k = 2
5       while True:
6         a = (a^k) modulo n
7         d = MCD(a-1, n)
8         if(d > 1 y d < n): Devolver d
9         if(d == n): Devolver False
10        k = k + 1
```

Para el caso de Pollard p-1 se han realizado unas pruebas de tamaño 24 a 56 que han llevado varias horas y presenta estos resultados:

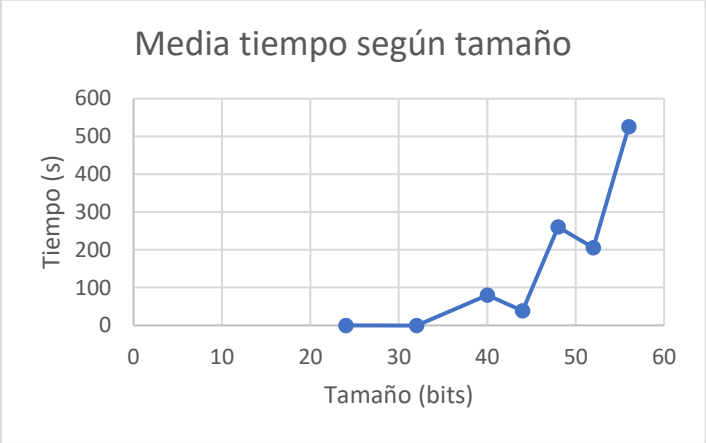

Ilustración 5: Media de tiempo según tamaño para Pollard P-1 ejecución base

Destaca desde el primer momento la pendiente tan irregular que presenta. Esto se debe a que ha habido en muchas ocasiones que para cierta talla no ha encontrado a tiempo, una solución para el problema. Ha habido tamaños que han tenido solo un valor sin descubrir o por ejemplo, para tamaño 56 se presentan 4 valores de 10 que no tienen solución a tiempo. Esto provoca un crecimiento drástico del tiempo medio.

Por su parte, la memoria no experimenta variación alguna a pesar del crecimiento del tamaño. (60 MB constantes)

Cabe recalcar que a pesar de haber una condición *"if(d == n): return False"* no se ha activado en ningún momento y en los resultados todas las ejecuciones aparecen con una solución o bien None por el timeout.

2.4 Implementación y experimentación base de Lenstra

Lenstra se ha mostrado como un algoritmo con una estructura mucho más elaborada, ya que como se verá a continuación supone la implementación de varios conceptos desarrollados a lo largo de la asignatura. Como el código de Lenstra resulta bastante extenso se va a explicar de forma general las funciones que lo conforman y qué consideraciones se han tenido en cuenta:

```
1   lenstra(n, cota):
2       while timer < 1200:
3           while curva discontinua:
4               Generar un punto P aleatorio entre 0 y n-1
5               Generar coeficiente A aleatorio entre 0 y n-1
6               Crear curva elíptica en formato y^2 = x^3 + ax + b mod n
7               Comprobar existencia discontinuidades 4a^3 + 27b^2  ¡= 0
8           k = 2
9           while k < cota:
10              Llamada a exponCuadradoSuc (k, p, a, b, n)
11              If MCD > 1:
12                  return solución
13              k += 1
```

Como se puede observar se han aplicado diversas propiedades de las curvas elípticas como su fórmula general, así como asegurarse que no se trata de una que sea discontinua, con el objetivo de obtener mejores resultados. Asimismo, destaca dentro del segundo bucle while que se llama a una función de la exponenciación de cuadrados sucesivos, su pseudocódigo es el siguiente:

```
1   exponCuadradosSuc(k, p, a, b, m):
2           result = (0,1,0) #Para el infinito
3           While k < 0:
4               If flag > 1:
5                   Return p
6               If k es impar:
7                   result = Llamar a sumaEnCurvasElipticas (p, result, a, b, m)
8               k = k // 2    #Devuelve parte entera de la división
9               p = Llamar a sumaEnCurvasElipticas(p, p, a, b, m)
10          return result
```

Del código anterior de la exponenciación por cuadrados sucesivos en la línea 6 se comprueba si es impar desde el tipo entero simplemente viendo el módulo 2, ya que si es 1 significa que el bit será 1. Además, se crea un flag para detectar cuando no es posible realizar los cálculos y dejar constancia de que ha habido un error. Por último, se vuelve a llamar una función de la cual aún no se ha visto, se trata de la "sumaEnCurvasElipticas". Esta función no es más que la aplicación de la diapositiva de suma (en este caso consigo mismo) cuando la recta es tangente a un punto o secante a dos:

```
1   sumaEnCurvasElipticas(p, q, a, b, m):
2       If el punto se encuentra en el infinito devuelve el otro
3       If coordenada x es igual en ambos puntos (es la tangente):
4           If la suma de coordenadas y módulo m
5               Considerar punto en infinito
6           Calcular numerador
7           Calcular denominador
8       else: (es la secante)
9           Calcular numerador
10          Calcular denominador
11      Calcular el inverso del denominador llamando a inverso(denominador, m)
12      If mcd > 1:
13          Imposible encontrar el inverso
14      Calcular pendiente
15      Calcular Xr
16      Calcular Yr
17      Return Xr, Yr, 1
```

Adicionalmente como se ha visto se ha llamado a una función adicional para calcular el inverso. Dentro de esta función destaca el uso de divmod(a,b) que devuelve tanto el cociente como resto de la división, de gran utilidad para este caso.

Pasando ya a las pruebas realizadas, se ha ejecutado inicialmente con una cota de 6000. Realmente se desconoce momentáneamente si es un valor adecuado para los tamaños que se calculan. Sin embargo, más adelante en experimentación extensa se revisará qué efecto tiene esta variable sobre el tiempo de ejecución. Continuando con la experimentación base se han hecho pruebas de tamaño 24 hasta 92:

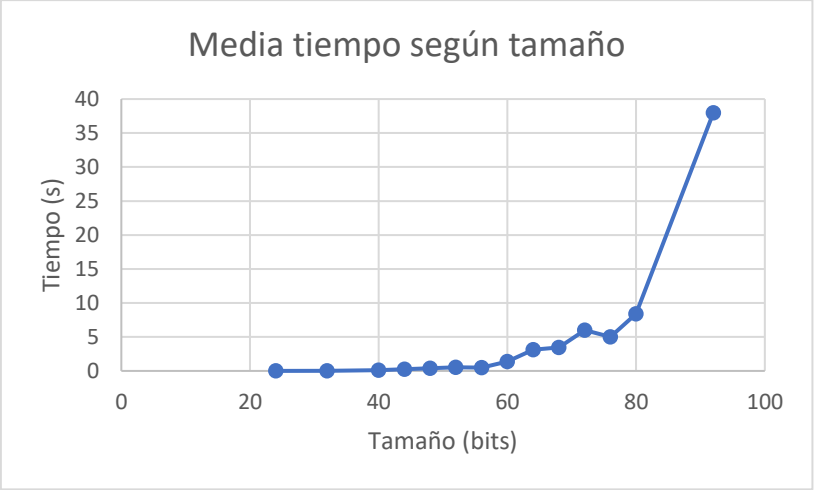

Ilustración 6: Media de tiempo según tamaño para Lenstra ejecución base

La media de tiempo presenta unos resultados crecientes a medida que aumenta el tamaño en bits de los números introducidos. Se esperaban tener inicialmente unos resultados mejores dada la complejidad del algoritmo de Lenstra. Una hipótesis inicial que se deriva de esta situación es que la cota puede tener grandes efectos sobre el tiempo medio, por ello que se analizará más adelante.

Por su parte, la memoria no experimenta variación alguna a pesar del crecimiento del tamaño. (60 MB constantes)

2.5 Experimentación extendida

A lo largo de este capítulo se van a tratar diferentes aspectos. En primer lugar, se quiere ver qué efectos tiene la cota sobre los resultados en Lenstra y si es posible mejorar la media de tiempo obtenida para cada tamaño de los resultados del anterior capítulo. En segundo lugar, se quiere hacer una comparación entre todos los algoritmos vistos hasta el momento para elegir el mejor según las pruebas efectuadas.

Obviamente, para estos apartados como se trata de llegar en mayor profundidad se va a emplear el nuevo archivo de datos más extenso que cuenta con 100 pruebas para cada tamaño.

2.5.1 Elección cota en Lenstra

Previamente en el capítulo de experimentación base de Lenstra se eligió el límite/cota de 6000 para la ejecución. Sin embargo, realmente no se siguió ningún criterio a la hora de escoger ese valor, puesto que se desconoce cuál escoger según el tamaño del problema. Por este motivo, en este apartado se va a tratar de determinarlo con el objetivo de que más adelante se comparen los algoritmos entre ellos la ejecución de Lenstra esté en óptimas condiciones.

En las siguientes pruebas se ha dejado toda la noche el ordenador ejecutando el algoritmo con valores desde 100 hasta 100 000 con tal de ver su impacto. La siguiente tabla recoge dicho comportamiento:

Tamaño	Cota 100	Cota 500	Cota 1000	Cota 10000	Cota 50000	Cota 100000
24	0,001	0,002	0,003	0,004	0,004	0,006
32	0,007	0,017	0,020	0,035	0,036	0,076
40	0,026	0,046	0,053	0,211	0,611	1,361
44	0,057	0,084	0,112	0,580	1,081	1,365
48	0,110	0,108	0,190	0,766	1,851	2,910
52	0,270	0,230	0,307	0,929	3,383	6,371
56	0,521	0,377	0,385	1,601	6,069	8,900
60	1,119	0,561	0,854	2,360	7,884	12,232
64	2,390	1,224	1,265	4,473	9,955	13,613
68	4,925	2,107	1,778	4,646	16,919	21,787
72	12,775	3,492	3,526	7,217	20,793	29,590

Tabla 2: Tiempo según la cota y tamaño para Lenstra

Ahora se va a crear un gráfico que recoja estas evoluciones:

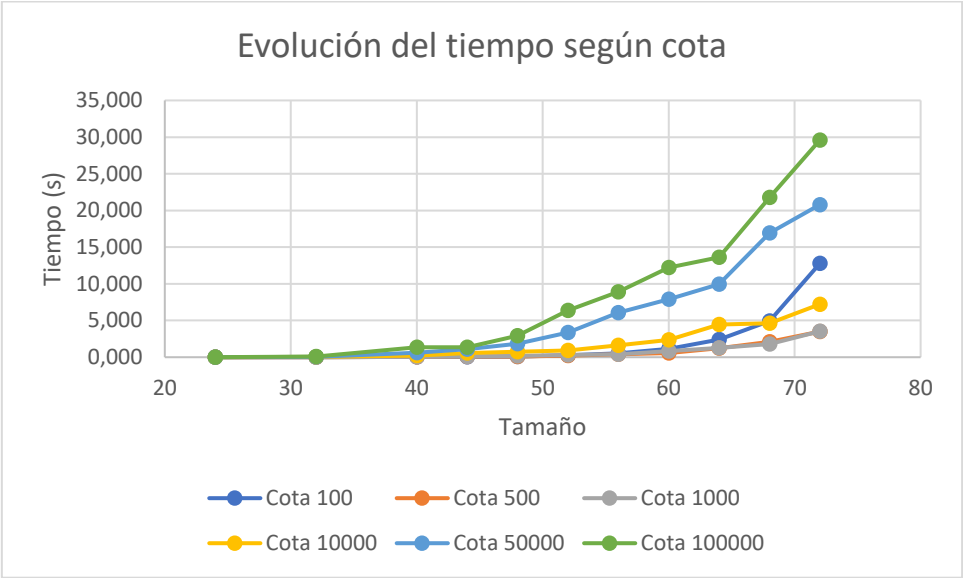

Ilustración 7: Comparación tiempo según la cota y tamaño para Lenstra

Como se puede notar del anterior gráfico se cumplen las sospechas de que es sumamente importante elegir correctamente la cota. Se observa que cotas demasiado altas implican una gran cantidad de trabajo para los pequeños tamaños de bits que se están manejando. El caso más claro es el de la cota 100 000 que supera con creces a la que se encuentra por debajo, la cota 50 000, en cuanto a tiempo promedio se refiere. Las cotas de menor valor parecen tener un mejor comportamiento, siendo las mejores que se pueden observar la cota de 500 y de 1000. Resalta cómo la cota de 100 tiene un gran aumento de tiempo cuando se pasa de tamaño 68 a 72, aspecto que reafirma que una cota de tan pequeño valor no es adecuada para tamaños de n tan grandes. Lo contrario pasa con la cota 10 000, donde al principio se sitúa por encima del tiempo de la cota 100, pero en cuanto el tamaño aumenta al final muestra un mejor resultado. Con el objetivo de elegir la mejor cota para la comparación global de algoritmos de factorización se va a sacar una nueva tabla con los promedios del tiempo:

Cota	Promedio (s)
100	2,018
500	0,749
1000	0,772
10000	2,074
50000	6,234
100000	8,928

Tabla 3: Promedio de tiempo según la cota en Lenstra

Ante estos datos del promedio de tiempo queda claro que lo mejor para esta experimentación es elegir la cota 500 al tener el menor valor. Se concluye que la cota es dependiente del tamaño de bits de los problemas que se quiera resolver.

2.5.2 Comparación algoritmos factorización

A lo largo de este trabajo se han ido viendo multitud de algoritmos de factorización. Cabe mencionar que cada uno hasta ahora se ha visto de forma aislada para asegurar su correcto funcionamiento y ver matices de su comportamiento. Sin embargo, se ha llegado al punto de que pueda resultar interesante de elegir aquél que presente unos mejores resultados. Mediante el archivo de experimentación extendida se han realizado pruebas desde tamaño 24 hasta 72, consiguiendo los siguientes resultados:

Tamaño	Fermat	Pollard-rho	Pollard p-1	Lenstra
24	0,007	0,000	0,003	0,002
32	0,000	0,000	0,960	0,017
40	0,002	0,000	82,887	0,046
44	0,009	0,001	112,164	0,084
48	0,029	0,002	212,562	0,108
52	0,112	0,004	-	0,230
56	0,508	0,008	-	0,377
60	1,947	0,014	-	0,561
64	11,153	0,037	-	1,224
68	47,544	0,073	-	2,107
72	177,434	0,157	-	3,492

Tabla 4: Tiempos para cada algoritmo de factorización

Recogido en un gráfico resulta de esta forma:

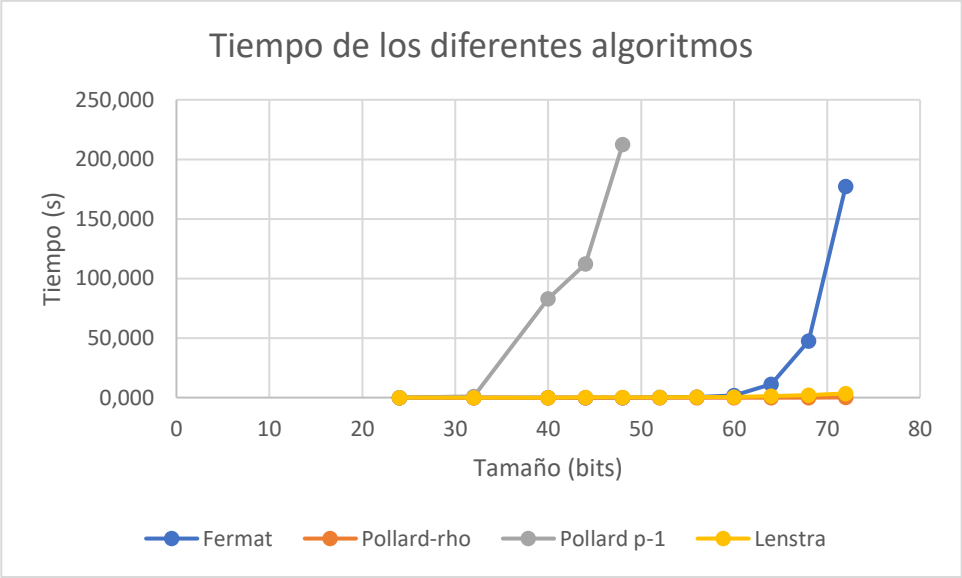

Ilustración 8: Comparación algoritmos de factorización

Los resultados son interesantes dada las grandes diferencias que se presentan entre algunos algoritmos. Destaca a primera vista como Pollard p-1 es incapaz de rivalizar ante las otras soluciones. A partir de tamaño 40 se muestra un crecimiento agigantado que ha obligado a interrumpir la ejecución, puesto que después de toda la noche de ejecución solo para este algoritmo se había alcanzado tamaño 48 bits y de muchas veces se llegaba el timeout de 20 minutos. Similar ocurre con el algoritmo de Fermat que al acercarse a las últimas pruebas de tamaño 72 empieza a tener un gran crecimiento en el promedio

del tiempo, aunque no llegando al timeout. Ante estos resultados queda claro que Fermat y Pollard p-1 no son muy buenas elecciones, pero ¿qué ocurre con Lenstra y Pollard-rho? Habrá que verlo con un gráfico aislando estos dos comportamientos:

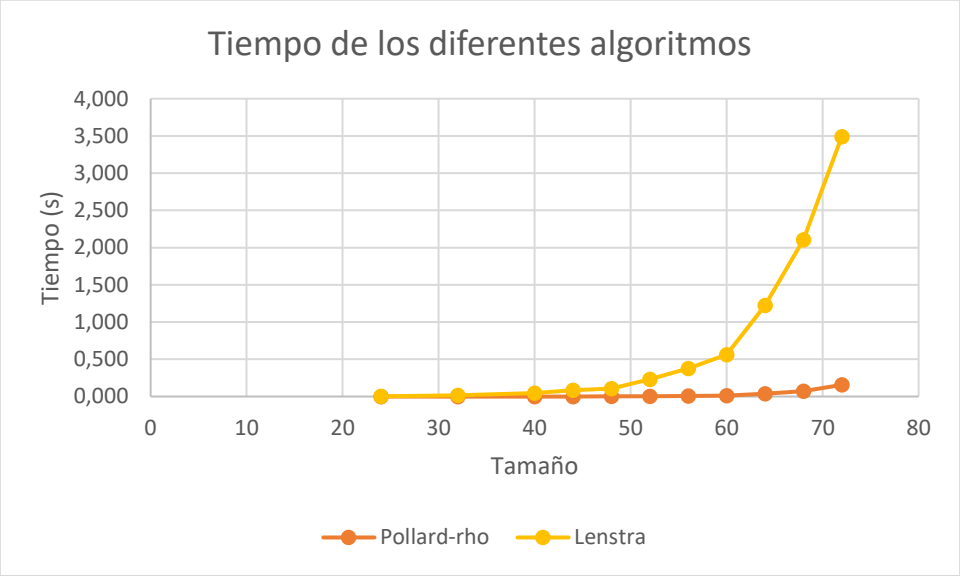

Ilustración 9: Comparación Pollard-rho y Lenstra

Antes estos resultados se ven que Lenstra tiene un peor comportamiento dada la gran pendiente que muestra ya al final de las pruebas. Pollard-rho mantiene un menor promedio de tiempo que lo hace quedar como una mejor elección de algoritmo para factorización. Cabe destacar que estos resultados se han obtenido para los algoritmos que hay en los anexos de este documento. Si se implementasen nuevas mejoras de eficiencia cabría la posibilidad de que Lenstra mostrase un mejor comportamiento.

Aun así, eligiendo Pollard-rho como algoritmo de factorización, por lo menos para las pruebas efectuadas puede surgir una nueva pregunta**: ¿Si se emplease para números de tamaños reales empleados hoy en día cuánto se tardaría?** Se va a suponer que se quiera atacar a un algoritmo RSA de tamaño 2048 bits, para hacer una estimación de tiempo se va a necesitar un gráfico que obtenga una ecuación capaz de predecir el comportamiento de Pollard-rho:

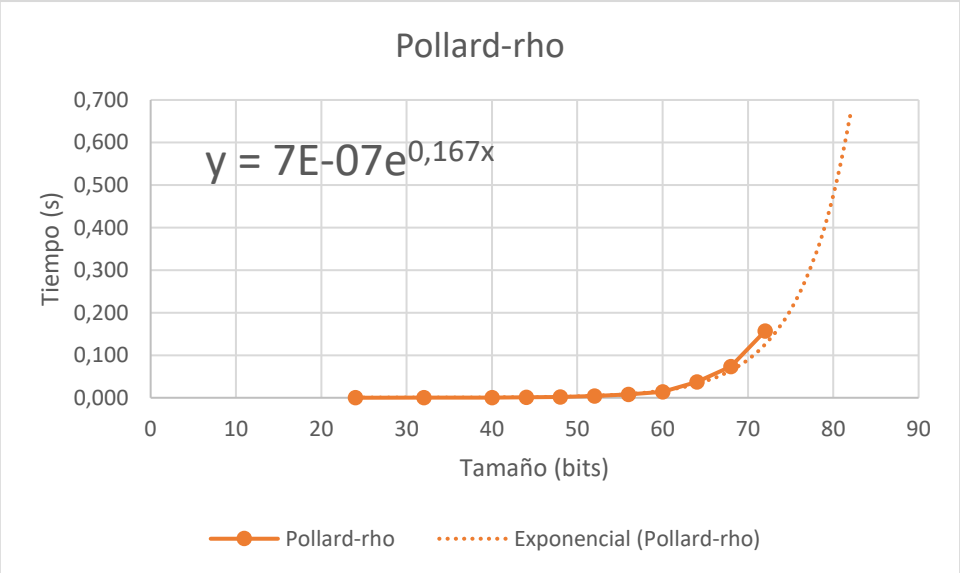

Ilustración 10: Línea tendencia Pollard-rho a futuro

A pesar de no haber realizado pruebas de tamaño 2048 bits, gracias al agregar una línea de tendencia que se adapte lo mejor al comportamiento de la evolución de tiempo de Pollard-rho se va a poder obtener una aproximación. Se ha optado por emplear una línea exponencial ya que es la que tenía el comportamiento más parecido. La ecuación de esta se muestra en el gráfico obtenido gracias a Excel. **Aplicando dicha ecuación se obtiene que para el tamaño de 2048 bits se tardaría aproximadamente 7,61473E134 AÑOS**, una cifra extremadamente elevada que supera la vida humana.

3. Logaritmo discreto

3.1 Implementación y experimentación base de Baby-step Giant-step

El pseudocódigo definido para el algoritmo de Baby-step Giant-step es el siguiente:

```
1   babyStepGiantStep(p, alpha, beta):
2       n = ⌈√p⌉
3       Crear diccionario t vacío
4       For r de 0 a n-1:
5           Llenar diccionario con el par t[alpha^r mod p] = r
6       Definir alpha_invN = alpha^-n mod p
7       gamma = beta
8       for q de 0 a n-1:
9           if gamma se encuentra en diccionario t:
10              j = t[gamma]
11              k = q*n+j
12              return k
13          gamma = (gamma * alpha_invN) mod p
14      return False
```

Como se puede observar el algoritmo desarrollado es muy similar al código proporcionado en clase, pero traducido a Python. Gracias al haber escogido este lenguaje se pueden llamar a funciones preconstruidas que pueden realizar por ejemplo la línea 6 en un momento, pow(alpha, -n, p), simplificando todo el proceso. No se han incluido las líneas que guardan las variables de memoria ni tampoco las variables de reloj que miden si se ha alcanzado el timeout. Estos detalles y el código exacto se pueden ver en el anexo de este trabajo.

Se han realizado pruebas del código desde el tamaño 8 a 56. Los resultados son los siguientes:

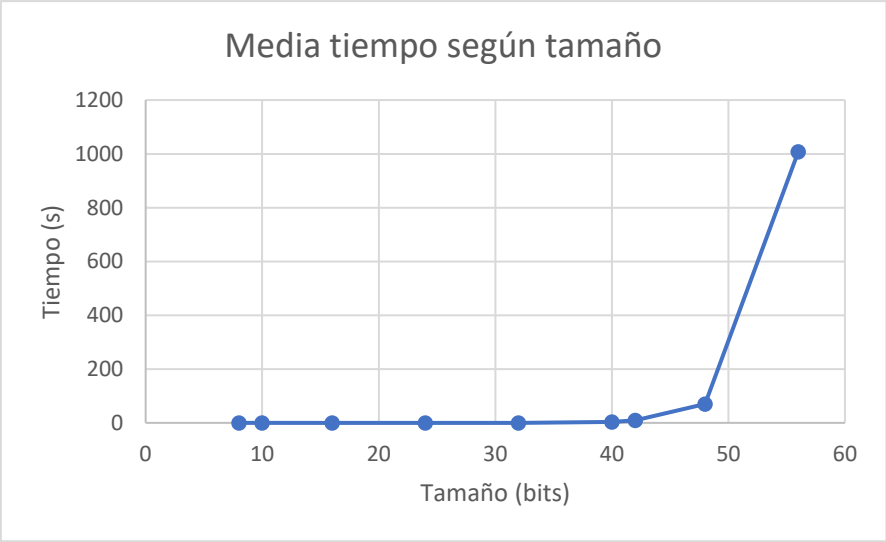

Ilustración 11: Media de tiempo según tamaño para Baby-Step Giant-Step ejecución base

Los resultados son bastante interesantes, lo que se observa en los anteriores resultados es que al intentar sobrepasar ya a los 56 bits se produce un salto enorme de tiempo. De los 2 cálculos que se tienen para tamaño 56 el primero se alcanza a averiguar la solución al problema después de 13 minutos, mientras que en el segundo después de 20 minutos ha superado el timeout sin tener lista la solución.

Otro aspecto por recalcar es que a pesar de haber una salida a False en ningún momento se alcanza, es decir, salvo para el último cálculo que se ha interrumpido por timeout, todos los anteriores han presentado una solución.

Por último, se ha detectado un crecimiento importante de la memoria a medida que aumenta según el tamaño en bits de las ejecuciones. Este aspecto se tratará más en profundidad en el capítulo de pruebas extensas.

3.2 Implementación y experimentación base de Pollard-rho logaritmo discreto

El pseudocódigo definido para el algoritmo de Pollard-rho del logaritmo discreto es el siguiente:

```
1   Pollard_rho_logDisc(p, alpha, beta, o):
2       a = b = aa = bb = 0
3       i = x = xx = 1
4       Mientras i < p:
5           x, a , b = llamar a la función f(x, a, b, alpha, beta, p)
6           xx, aa, bb = llamar a la función f(xx, aa, bb, alpha, beta, p)
7           xx, aa, bb = llamar a la función f(xx, aa, bb, alpha, beta, p)
8
9           if se cumple x == xx:
10              if MCD(b - bb, o) diferente de 1:
11                  return False
12              return ((aa - a) * inverso(b-bb)) módulo o
13          i = i+1
14      return False
```

El código anterior no tiene gran complejidad, puesto que en la mayoría de las líneas se declaran variables o se realizan operaciones matemáticas básicas. Sin embargo, lo más costoso de este proceso es darse cuenta de que se debe llamar a una función externa f que va a albergar toda la lógica de actualización a los nuevos valores de x, a , b, xx, aa y bb. Esta es la función f en su pseudocódigo:

```
f(x, a, b, alpha, beta, p):
        x = ((alpha^a) * (beta^b)) módulo p
        mod = x módulo 3
        if mod es 1:
                x_ii = (beta * x) módulo p
                a_ii = a
                b_ii = (b+1) módulo (p-1)
        elif mod es 0:
                x_ii = (x**2) módulo p
                a_ii = (2*a) módulo (p-1)
                b_ii = (2*b) módulo (p-1)
        else: # mod es 2
                x_ii = (alpha * x) módulo p
                a_ii = (a+1) módulo (p-1)
                b_ii = b

        return x_ii, a_ii, b_ii
```

Se han realizado pruebas del código desde el tamaño 8 a 56, aunque el código no ha podido llegar a valores tan altos interrumpiéndose en 40. Los resultados son los siguientes:

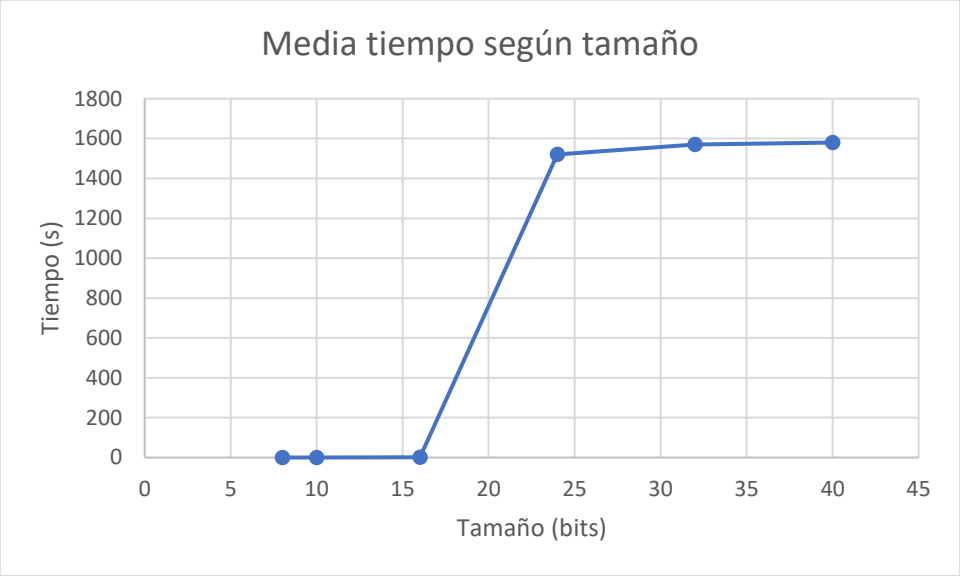

Ilustración 12: Media de tiempo según tamaño para Pollard-rho (log d) ejecución base

El anterior gráfico muestra unos resultados muy inusuales teniendo en cuenta todos los algoritmos anteriores. Sorprende el hecho de que al pasar de tamaño 16 a 24 el algoritmo ya no es capaz de encontrar una solución en el tiempo de timeout inferior a 20 minutos. Esto ha impedido obtener suficientes datos con la profundidad deseada. Para llegar a una mejor comprensión hay que ver la tabla CSV generada:

Tamaño	Número	Solución	Tiempo
8	487	False	0.0
8	239	False	0.0
10	1109	206	0.0
10	587	23	0.001
16	55243	22484	0.507
16	33967	False	0.445
24	14005903	None	1483.359
24	11841217	None	1597.019
32	3094892893	None	1613.641
32	3326067959	None	1584.691
32	3477400219	None	1590.554
40	585690427969	None	1610.876
40	1014737275411	None	1554.565
40	806063375303	None	1595.543

Tabla 5: Resultados CSV de Pollard-rho en experimentación base

Hay que hacer una aclaración de la tabla anterior: el valor None de la solución significa que se ha superado el timeout y el valor False es que no se ha descubierto la solución dentro del tiempo. De 14 ejecuciones solo se tiene la solución a 3 pruebas, es decir, un 21 % de acierto con los parámetros actuales.

En último lugar, la memoria no presenta ningún problema ya que se mantiene en 60 MB de forma estable.

3.3 Experimentación extendida

Se van a realizar las pruebas con el archivo de experimentación extendida que hay publicado con el objetivo de llegar a unos resultados más precisos en los análisis y así poder realizar una comparación.

3.3.1 Análisis memoria Baby-step Giant-step

Como ya se presenció en las ejecuciones del capítulo anterior, el algoritmo de Baby-step Giant-step sufre de un problema de memoria que puede acabar generando interrupciones. Por ello se van a realizar las ejecuciones pertinentes y ver la evolución de esta variable. El siguiente gráfico muestra la media de memoria según el tamaño:

Tamaño	Memoria
< 32	62,61
32	68,51
36	89,37
40	163,41
44	401,26
48	1414,7
52	5544,41
56	> 10000

Tabla 6: Evolución memoria Baby-Step Giant-Step

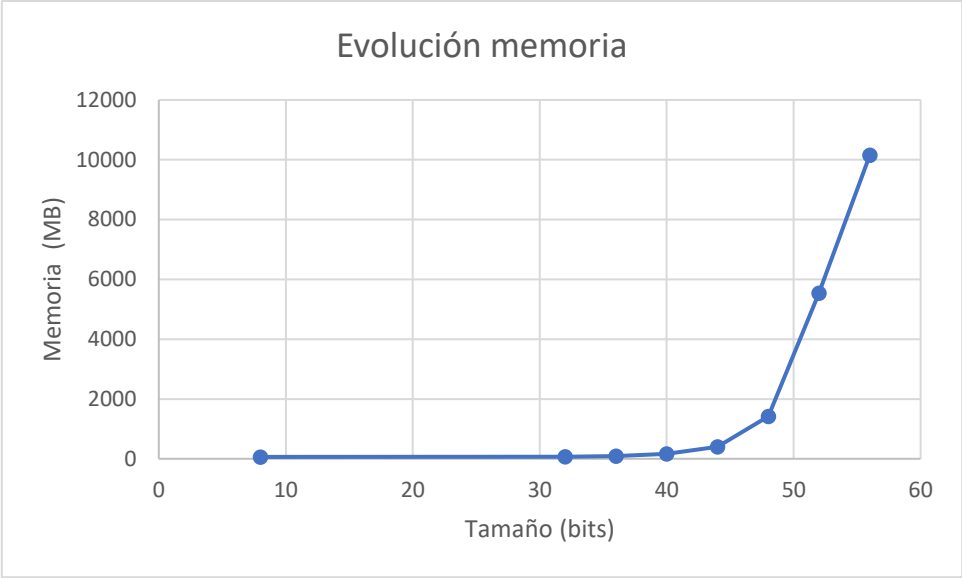

Ilustración 13: Evolución memoria 2

El presente gráfico recoge la media de memoria consumida por el algoritmo en las ejecuciones del archivo extenso, es decir de 32 a 56. Además de ello, se ha incluido los datos extraídos de las ejecuciones base, ya que se visto que de media por debajo de 32 bits el valor se sitúa alrededor de 62,61 MB. Esto es a causa de que al igual que todos los demás algoritmos de este trabajo el proceso de Python consume de base, al importar todas las librería y elementos necesarios de la ejecución, unos recursos imprescindibles para llevar a cabo la tarea. Sin embargo, a diferencia de los demás, Baby-Step Giant-Step hace uso de un diccionario que crece según el bucle "for r in range(n):" de la línea 4, que a bajos tamaños no se nota su presencia, pero a medida que el tamaño de bits crece también lo hace el diccionario. Ya a partir de los 48 bits se nota su evolución con más de 1 GB, aunque a los 56 alcanza los 10 GB. Esto resulta en un grave problema ya que, en este tamaño de 56 bits, ha habido muchas ejecuciones que no se han podido realizar por ello que en la tabla se ha dejado como > 10000 MB. No se ha conseguido hacer una media con todas las pruebas de ese tamaño al tener muchas ocasiones que terminaba en el siguiente error de memoria:

```
Traceback (most recent call last):
  File "c:\Users\silvi\Desktop\Master\CSD\Trabajo criptoanalisis\algoritmos1.py", line 325, in <module>
    result = babyStepGiantStep(int(row['m']), int(row['alfa']), int(row['beta']))
             ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "c:\Users\silvi\Desktop\Master\CSD\Trabajo criptoanalisis\algoritmos1.py", line 98, in babyStepGiantStep
    t[pow(alpha, r, p)] = r
    ~^^^^^^^^^^^^^^^^^^^
MemoryError
```

Ilustración 14: Resultado de la consola

Como se puede ver en la parte inferior se ha efectuado un "MemoryError" durante la ejecución de la línea "t[pow(alpha, r, p)] = r", es decir, justo en el momento que al diccionario se le añadía una entrada adicional. A causa de estos errores se ha decido no seguir calculando el para 56 bits o mayores. **La conclusión de este algoritmo es que no es óptimo para ser usado en tamaños grandes.**

3.3.2 Comparación algoritmos logaritmo discreto

Se ha intentado ejecutar el archivo extenso de logaritmo extenso, tanto para Baby-Step Giant-Step como Pollard-rho. Sin embargo, este último se ha sido incapaz de lograr un resultado satisfactorio con un timeout de 20 minutos. Ante esta situación se creía que el timeout era demasiado pequeño como para lograr alcanzar una solución, pero al aumentar el timeout a 40 minutos (2400 en segundos) se sigue sin obtener alguna solución. Las tablas de ejecución de Pollard-rho son las siguientes después de dejarlo durante más de 10 horas[1]:

Tamaño	Número	Solución	Tiempo	Memoria
32	2703258601	None	2886.609	60.8
32	2703258601	None	2467.253	61.16
32	2703258601	None	4297.83	61.03
32	2703258601	None	2666.521	61.45
32	2703258601	None	4649.58	61.41
32	2703258601	None	3211.093	61.42
32	2703258601	None	2793.009	61.0
32	2703258601	None	3459.643	62.47
32	2703258601	None	9639.871	62.81

Tabla 7: Solución Pollard-rho experimentación extensa

Ante esta situación después de todas estas ejecuciones se ha optado por interrumpir la ejecución y no seguir con un análisis en mayor profundidad. Como posible explicación ante estos resultados se intuye que es posible que sea a causa de que el algoritmo pueda ser ineficiente al resolver el logaritmo discreto en grupos con tamaños grandes a causa de su complejidad computacional o que solo sea posible calcularlo para entradas que tengan un patrón matemático determinado.

Por la otra parte , para el otro algoritmo de Baby-Step Giant-Step se ha conseguido ejecutar desde 32 a 52 bits en casi 7 horas:

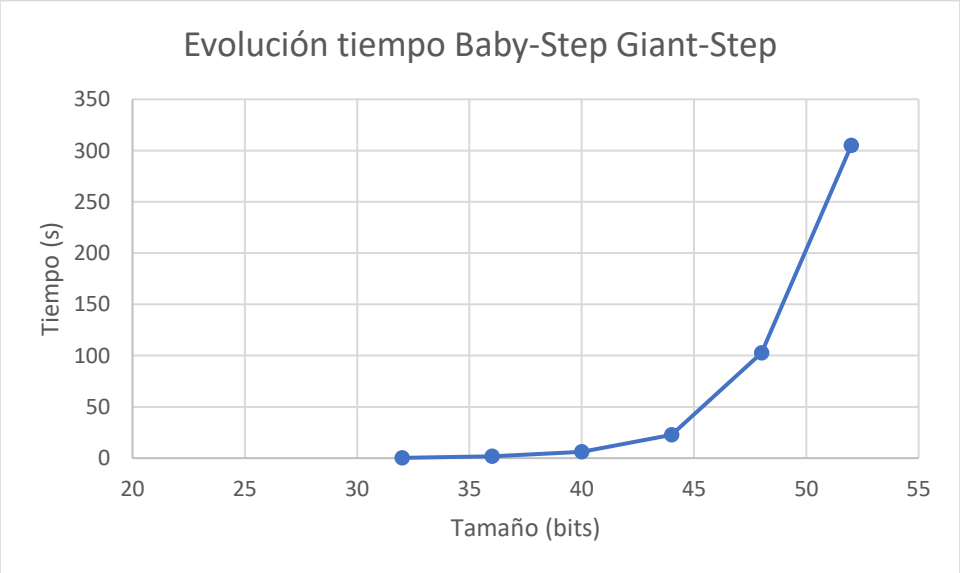

Ilustración 15: Evolución tiempo según tamaño según Baby-Step Giant-Step experimentación extensa

[1] Se desconoce porque a pesar de usar los mismos comandos de timeout en Pollard-rho de algoritmo discreto no se respeta los 40 minutos fijados como límite. No obstante, esto ha proporcionado un mayor rango de tiempo para encontrar una solución para cada de esos tamaños y aun así no se ha logrado obtenerla.

Cabe recordar que no se ha seguido la ejecución a tamaños mayores de 56 al presentar los problemas de memoria ya comentados anteriormente. Aun así, se gozan de datos suficientes para hacer una predicción de un problema real y con esto se va a crear un nuevo gráfico en Excel con una función exponencial que es la que más se adapta al crecimiento observado:

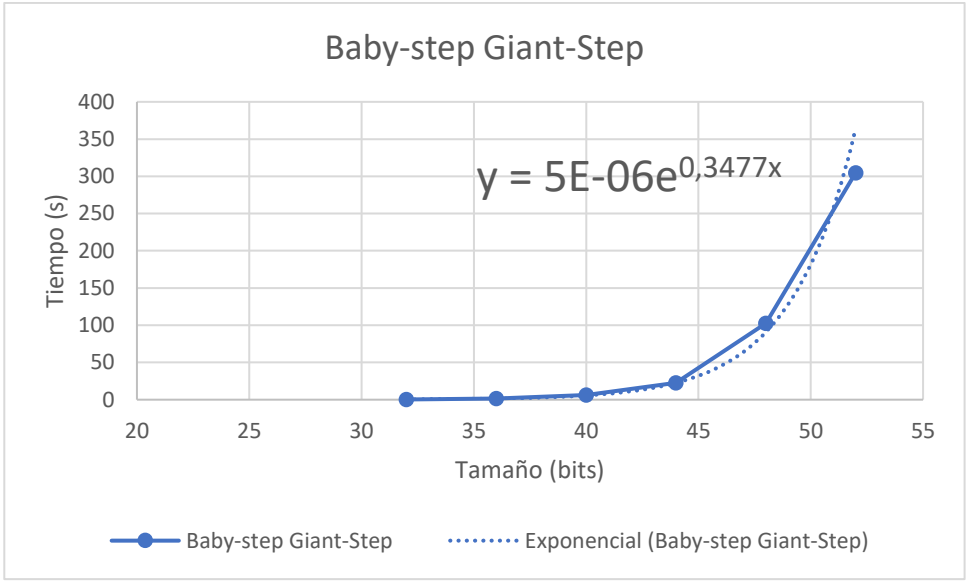

Ilustración 16: : Línea tendencia Baby-Step Giant-Step a futuro

Con esto se puede aplicar ya a problemas reales para poder hacer una predicción aproximada. **¿Si se emplease para números de 2048 bits cuánto se tardaría?** Aplicando la ecuación obtenida se tiene que para 2048 bits se necesitan $1{,}5844E - 13e^{712{,}09}$ AÑOS (para el hardware de estas pruebas). Una cifra que se sitúa muy por encima de lo que una persona podría esperar para romper los 2048 bits.

Anexo

Fermat

```
def fermat(n):
    start_timer = time.time()
    a = math.ceil(math.sqrt(n))
    b = a**2 - n
    while not b == math.isqrt(b) ** 2:
        a += 1
        b = a**2 - n
        if(time.time() - start_timer > 1200): return None

    global memory
    memory = round(psutil.Process(os.getpid()).memory_info().rss / 1024 ** 2, 2)
    return a - math.sqrt(b), a + math.sqrt(b)
```

Pollard-rho Factorización

```
def pollard_rho_factorizacion(n):
    start_timer = time.time()

    randomNum = random.randint(2, n-1)
    a = randomNum
    b = randomNum
    while True:
        a = (a**2 + 1) % n
        b = (b**2 + 1) % n
        b = (b**2 + 1) % n
        p = math.gcd(a-b, n)

        if(time.time() - start_timer > 1200): return None
        global memory
        memory = round(psutil.Process(os.getpid()).memory_info().rss / 1024 ** 2, 2)
        if(p > 1 and p < n):
            return p
        if(p == n):
            return n
```

Pollard P-1

```
def pollard_P_1(n):
    start_timer = time.time()
    a = random.randint(2, n-1)
    if(math.gcd(a, n) > 1 and math.gcd(a, n) < n): return math.gcd(a, n)
    k = 2
    while True:
        a = (a**k) % n
        d = math.gcd(a-1, n)
        if(d > 1 and d < n): return d
        if(d == n): return False
        k += 1

        global memory
        memory = round(psutil.Process(os.getpid()).memory_info().rss / 1024 ** 2, 2)
        if(time.time() - start_timer > 1200): return None
```

Lenstra

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

def sumaEnCurvasElipticas(p, q, a, b, m):
    if p[2] == 0: return q
    if q[2] == 0: return p
    if p[0] == q[0]:
        if (p[1] + q[1]) % m == 0:
            return 0, 1, 0
        numerador = (3 * pow(p[0],2) + a) % m
        denominador = (2 * p[1]) % m
    else:
        numerador = (q[1] - p[1]) % m
        denominador = (q[0] - p[0]) % m
    inv, _, gcd = inverso(denominador, m)

    if gcd > 1:
        return 0, 0, denominador

    pendiente = numerador * inv
    xr = (pendiente**2 - p[0] - q[0]) % m
    yr = (pendiente * (p[0] - xr) - p[1]) % m
    return xr, yr, 1



def exponCuadradosSuc(k, p, a, b, m):
    result = (0, 1, 0)
    while k > 0:
        if p[2] > 1:
            return p
        if k % 2 == 1:
            result = sumaEnCurvasElipticas(p, result, a, b, m)
        k = k // 2
        p = sumaEnCurvasElipticas(p, p, a, b, m)

    return result
```

```
def lenstra(n, cota):
    timer = 0
    start_timer = time.time()

    while timer < 1200:
        gcd = n
        while gcd == n:
            p = random.randint(0, n - 1), random.randint(0, n - 1),1
            a = random.randint(0, n - 1)
            b = (pow(p[1],2) - pow(p[0],3) - a * p[0]) % n
            gcd = math.gcd(4 * pow(a,3) + 27 * pow(b,2), n)

        global memory
        memory = round(psutil.Process(os.getpid()).memory_info().rss / 1024 ** 2, 2)
        if gcd > 1:
            return gcd

        k = 2
        while k < cota:
            p = exponCuadradosSuc(k, p, a, b, n)
            if p[2] > 1:
                return math.gcd(p[2], n)
            k += 1
        timer = time.time() - start_timer
```

Baby-step Giant-step

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

    alpha_invN = pow(alpha, -n, p)
    gamma = beta
    for q in range(n):
        global memory
        memory = round(psutil.Process(os.getpid()).memory_info().rss / 1024 ** 2, 2)
        if gamma in t:
            j = t[gamma]
            k = q * n + j
            return k
        gamma = (gamma * alpha_invN) % p
        if(time.time() - start_timer > 1200): return None
    return False
```

Pollard-rho Log Discreto

```
def f(x, a, b, alpha, beta, p):
    x = ((alpha**a) * (beta**b)) % p
    modulo = x % 3
    if modulo == 1:
        x_ii = (beta * x) % p
        a_ii = a
        b_ii = (b+1) % (p-1)
    elif modulo == 0:
        x_ii = (x**2) % p
        a_ii = (2*a) % (p-1)
        b_ii = (2*b) % (p-1)
    else: #modulo == 2
        x_ii = (alpha * x) % p
        a_ii = (a+1) % (p-1)
        b_ii = b

    return x_ii, a_ii, b_ii


def pollard_rho_logDisc(p, alpha, beta, o):
    start_timer = time.time()
    a = b = aa = bb = 0
    i = x = xx = 1

    while i < p:
        if(time.time() - start_timer > 2400): return None
        x,a,b = f(x, a, b, alpha, beta, p)
        xx,aa,bb = f(xx, aa, bb, alpha, beta, p)
        xx,aa,bb = f(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
        global memory
        memory = round(psutil.Process(os.getpid()).memory_info().rss / 1024 ** 2, 2)

    return False
```

Método main para generar resultados

```
if __name__ == '__main__':
    dataFact = pd.read_csv("datosFactorizacionExt.csv")
    dataFact.set_index("Tamaño", inplace=True)

    resultAnalysisCSV = pd.DataFrame()
    for index, row in dataFact.iterrows():
        start_time = time.time()
        result = fermat(int(row['n']))
        end_time = time.time() - start_time

        data = {
            'Tamaño': index,
            'Número': row['n'],
            'Solución': [str(result)],
            'Tiempo': round(end_time, 3),
            'Memoria': memory
        }
        new_data = pd.DataFrame(data)
        resultAnalysisCSV = pd.concat([resultAnalysisCSV, pd.DataFrame(new_data)], ignore_index=True)
        print(resultAnalysisCSV)
        resultAnalysisCSV.to_csv("Resultados/Fermat.csv", index=False)


    resultAnalysisCSV = pd.DataFrame()
    for index, row in dataFact.iterrows():
        start_time = time.time()
        result = pollard_rho_factorizacion(int(row['n']))
        end_time = time.time() - start_time

        data = {
            'Tamaño': index,
            'Número': row['n'],
            'Solución': [str(result)],
            'Tiempo': round(end_time, 3),
            'Memoria': memory
        }
        new_data = pd.DataFrame(data)
        resultAnalysisCSV = pd.concat([resultAnalysisCSV, pd.DataFrame(new_data)], ignore_index=True)

        print(resultAnalysisCSV)
        resultAnalysisCSV.to_csv("Resultados/Pollard_rho_factorizacion.csv", index=False)

    resultAnalysisCSV = pd.DataFrame()
    for index, row in dataFact.iterrows():
        start_time = time.time()
        result = lenstra(int(row['n']), 100)
        end_time = time.time() - start_time

        data = {
            'Tamaño': index,
            'Número': row['n'],
            'Solución': [str(result)],
            'Tiempo': round(end_time, 3),
```

```
        'Memoria': memory
    }
    new_data = pd.DataFrame(data)
    resultAnalysisCSV = pd.concat([resultAnalysisCSV, pd.DataFrame(new_data)], ignore_index=True)
    print(resultAnalysisCSV)
    resultAnalysisCSV.to_csv("Resultados/Lenstra.csv", index=False)

  resultAnalysisCSV = pd.DataFrame()
  for index, row in dataFact.iterrows():
    start_time = time.time()
    result = pollard_P_1(int(row['n']))
    end_time = time.time() - start_time

    data = {
        'Tamaño': index,
        'Número': row['n'],
        'Solución': [str(result)],
        'Tiempo': round(end_time, 3),
        'Memoria': memory
    }
    new_data = pd.DataFrame(data)
    resultAnalysisCSV = pd.concat([resultAnalysisCSV, pd.DataFrame(new_data)], ignore_index=True)
    print(resultAnalysisCSV)
    resultAnalysisCSV.to_csv("Resultados/Pollard_P_1.csv", index=False)



##########################################Logaritmo Discreto############################

  dataFact = pd.read_csv("datosLogaritmoExt.csv")
  dataFact.set_index("Tamaño", inplace=True)

  resultAnalysisCSV = pd.DataFrame()
  for index, row in dataFact.iterrows():
    start_time = time.time()
    result = babyStepGiantStep(int(row['m']), int(row['alfa']), int(row['beta']))
    end_time = time.time() - start_time

    data = {
        'Tamaño': index,
        'Número': row['m'],
        'Solución': [str(result)],
        'Tiempo': round(end_time, 3),
        'Memoria': memory
    }
    new_data = pd.DataFrame(data)
    resultAnalysisCSV = pd.concat([resultAnalysisCSV, pd.DataFrame(new_data)], ignore_index=True)

    print(resultAnalysisCSV)
    resultAnalysisCSV.to_csv("Resultados/babyStepGiantStepExt.csv", index=False)


  resultAnalysisCSV = pd.DataFrame()

  for index, row in dataFact.iterrows():
    start_time = time.time()
```

```
    result = pollard_rho_logDisc(int(row['m']), int(row['alfa']), int(row['beta']), int(row['orden']))
    end_time = time.time() - start_time

    data = {
        'Tamaño': index,
        'Número': row['m'],
        'Solución': [str(result)],
        'Tiempo': round(end_time, 3),
        'Memoria': memory
    }
    new_data = pd.DataFrame(data)
    resultAnalysisCSV = pd.concat([resultAnalysisCSV, pd.DataFrame(new_data)], ignore_index=True)

    print(resultAnalysisCSV)
    resultAnalysisCSV.to_csv("Resultados/pollard_rho_logDiscExt.csv", index=False)


resultAnalysisCSV = pd.DataFrame()
```