

LABORATORIO 5: HARD PROBLEMS IN CRYPTOGRAPHY

Torres Olivera Karla Paola

Escuela Superior de Cómputo
Instituto Politécnico Nacional, México
kpto997@gmail.com

Resumen: El presente trabajo consiste en la descripción e implementación en Python de una posible solución al problema del logaritmo discreto haciendo uso del algoritmo baby-step giant-step. Adicional a esto, se abordará el tema de encontrar los factores primos de números compuestos.

Palabras Clave: número compuesto, problema logaritmo discreto.

1 Introducción

Los algoritmos asimétricos o de clave pública utilizan dos parejas de claves, una pública conocida por todo el mundo y otra privada, que debe ser conocida únicamente por su propietario. Con este tipo de algoritmos se puede tanto cifrar información como firmarla [1].

Para el caso del cifrado, el emisor emplea la clave pública del receptor. El proceso de descifrado solo se puede realizar conociendo la clave privada que está en poder únicamente del destinatario del mensaje. Esto resuelve el problema de cómo acordar la clave secreta en algoritmos simétricos. Además, estos algoritmos ofrecen integridad y autenticidad de los mensajes [1].

Este tipo de algoritmos tiene su origen en el artículo de W. Diffie y M. E. Hellman *New directions in Cryptography* de 1976. En él proponen un protocolo para acordar una clave secreta utilizando un canal no seguro. La generación de claves se basa en la complejidad de resolver ciertos problemas matemáticos, en el caso de Diffie-Hellman utilizaron el **Problema del Logaritmo Discreto**. Los algoritmos de este tipo más relevantes son: RSA y ElGamal [1].

Tomando en cuenta lo anterior, en esta práctica se intentará resolver el problema del algoritmo discreto para números un tanto grandes con la finalidad de ver cuánto tiempo toma obtener una respuesta, por otro lado, también se resolverá el problema de obtener los factores primos de un número compuesto, debido a que también es utilizado en los cifradores de clave pública.

Contents

1	Introducción	1
2	Conceptos básicos	3
2.1	Problema del logaritmo discreto en campos primos	3
2.2	Problema del logaritmo discreto generalizado	3
2.3	Ataques contra el problema del logaritmo discreto	4
2.3.1	Algoritmos Genéricos	4
2.4	Shanks' Baby-Step Giant-Step Method	4
2.4.1	Algoritmo	4
2.5	Factores primos	5
2.5.1	Números primos	5
2.5.2	Encontrar factores primos: Tree Method	6
3	Experimentación y Resultados	7
3.1	Solución al problema del logaritmo discreto	7
3.1.1	Consideraciones del programa	7
3.1.2	Implementación del programa	7
3.1.3	Resultados obtenidos para cada uno de los casos	10
3.2	Encontrando factores primos de números compuestos	12
3.2.1	Consideraciones del programa	12
3.2.2	Implementación del programa	13
3.2.3	Resultados obtenidos para cada uno de los casos	15

2 Conceptos básicos

2.1 Problema del logaritmo discreto en campos primos

Definición Problema del logaritmo discreto (DLP) en Z_p^*
 Dado un grupo cíclico finito Z_p^* de orden $p - 1$ y un elemento primitivo $\alpha \in Z_p^*$ y otro elemento $\beta \in Z_p^*$. El problema DLP es determinar el número entero $1 \leq x \leq p - 1$ de manera que:

$$\alpha^x \equiv \beta \pmod{p}$$

Tabla 1: Definición del problema de logaritmo discreto obtenida de [2]

Tal entero x debe existir debido a que α es un elemento primitivo y cada elemento del grupo puede expresarse como una potencia de cualquier elemento primitivo. Este entero x se llama el logaritmo discreto de β a la base α , y se puede escribir formalmente:

$$x = \log_{\alpha} \beta \pmod{p}$$

Calcular logaritmos discretos módulo a primo es un problema muy difícil si los parámetros son lo suficientemente grandes [2].

2.2 Problema del logaritmo discreto generalizado

La característica que hace que el DLP sea particularmente útil en criptografía es que no está restringido al grupo multiplicativo Z_p^* , p primo, sino que puede definirse sobre cualquier grupo cíclico. Esto se llama *el problema de logaritmo discreto generalizado (GDLP)* y se puede establecer de la siguiente manera [2].

Definición Problema del logaritmo discreto generalizado
 Dado un grupo cíclico finito G con la operación grupal \circ y cardinalidad n . Consideramos un elemento primitivo $\alpha \in G$ y otro elemento $\beta \in G$. El problema del logaritmo discreto es encontrar el número entero x , donde $1 \leq x \leq n$, tal que:

$$\beta = \underbrace{\alpha \circ \alpha \circ \dots \circ \alpha}_{x \text{ times}} = \alpha^x$$

Tabla 2: Definición del problema del logaritmo discreto generalizado obtenida de [2]

Como en el caso del DLP en Z_p^* , dicho número entero x debe existir, debido a que α es un elemento primitivo y, por lo tanto, cada elemento del grupo G puede generarse mediante la aplicación repetida de la operación del grupo en α [2].

Es importante darse cuenta de que hay grupos cíclicos en los que el DLP no es difícil. Estos no pueden usarse para un criptosistema de clave pública ya que el DLP no es una función unidireccional [2].

2.3 Ataques contra el problema del logaritmo discreto

La seguridad de muchas primitivas asimétricas se basa en la dificultad de calcular el DLP en grupos cíclicos, es decir, calcular x para un α y β dado en G [2].

Todavía no se conoce la dificultad exacta de calcular el logaritmo discreto x en un grupo real dado. Es decir, aunque se conocen algunos ataques, no se sabe si existen algoritmos mejores y más potentes para resolver el DLP. Esta situación es similar a la dificultad de la factorización de enteros, que es la función unidireccional subyacente RSA. Nadie sabe realmente cuál es el mejor método de factorización posible. Para el DLP existen algunos resultados generales interesantes con respecto a su dificultad computacional [2].

2.3.1 Algoritmos Genéricos

Los algoritmos DL genéricos son métodos que solo usan la operación de grupo. Como no explotan las propiedades especiales del grupo, trabajan en cualquier grupo cíclico. Estos algoritmos para el problema del logaritmo discreto se pueden subdividir en dos clases. La primera abarca algoritmos cuyo tiempo de ejecución depende del tamaño del grupo cíclico, como la búsqueda por *fuerza bruta*, el algoritmo *baby-step giant-step* y el método *Pollard's rho*. La segunda clase son algoritmos cuyo tiempo de ejecución depende del tamaño de los factores primos del orden del grupo, como el algoritmo *Pohlig-Hellman* [2].

2.4 Shanks' Baby-Step Giant-Step Method

Este algoritmo fue propuesto por Shanks en 1971, tiene una complejidad temporal $O(\sqrt{m} \log m)$. También es conocido como encuentro en el medio (meet-in-the-middle) porque usa la técnica de separar las tareas por la mitad [3].

2.4.1 Algoritmo

Consideremos la ecuación:

$$a^x \equiv b \pmod{m}$$

donde a y m son primos relativos.

Se tiene que $x = np - q$, donde $n = \lceil \sqrt{m} \rceil$. p se conoce como *giant-step*, debido a que aumentarlo en uno aumenta x en n . Del mismo modo, q se conoce como *baby-step* [3].

Cualquier número x en el intervalo $[0; m)$ se puede representar de esta forma, donde $p \in [1; \lceil \frac{m}{n} \rceil]$ y $q \in [0; n]$ [3].

Entonces, la ecuación se convierte en:

$$a^{np-q} \equiv b \pmod{m}$$

Usando el hecho de que a y m son primos relativos, se obtiene:

$$a^{np} \equiv ba^q \pmod{m}$$

Esta nueva ecuación se puede reescribir en una forma simplificada:

$$f_1(p) = f_2(q)$$

En [3] se muestra como se puede resolver este problema utilizando el método de encuentro en el medio:

- Calcular f_1 para todos los argumentos posibles p .
- Para todos los argumentos posibles q , calcular f_2 y buscar si se obtiene un valor común con p utilizando la búsqueda binaria.

2.5 Factores primos

2.5.1 Números primos

Decimos que un número $p > 1$ es primo si sus únicos divisores positivos son 1 y p [4].

- Si un número entero $q > 1$ no es primo, se le llama número compuesto. Por tanto, un entero q será compuesto si y sólo si existen a, b enteros positivos (menores que q) tales que $q = ab$
- Si $\text{mcd}(a, b) = 1$ se dice que a y b son primos entre sí o que son primos relativos.

Algunas propiedades importantes sobre los números primos y compuestos encontradas en [4] son:

1. Teorema Fundamental de la Aritmética: Todo número natural mayor que 1 o bien es primo, o bien se puede descomponer como el producto de números primos. Además esta descomposición es única -salvo el orden de los factores-.
2. Si un número n es compuesto, se verifica que ha de tener un divisor primo menor o igual que su raíz cuadrada.

2.5.2 Encontrar factores primos: Tree Method

Un método para encontrar los factores primos de un número compuesto es Tree Method, en el cual se dibujan “ramas de factores”. Existen diversas maneras en que se puede diseñar el árbol, sin embargo, para la elaboración del segundo programa se hizo uso del procedimiento encontrado en [5].

Ejemplo 1: Encontrar los factores primos de 189

- Paso 1: Comience dividiendo el número por el primer número primo 2 y continúe dividiendo por 2 hasta obtener un decimal o el resto. Luego divida entre 3, 5, 7, etc. hasta que los únicos números restantes sean números primos.

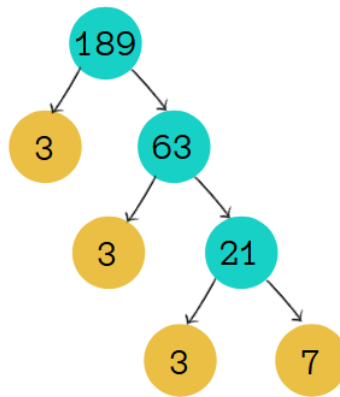


Figura 1: Árbol resultante del paso 1

- Paso 2: Escribe el número como producto de los números primos obtenidos en el paso anterior.

$$3 \times 3 \times 3 \times 7 = 3^3 \times 7$$

3 Experimentación y Resultados

3.1 Solución al problema del logaritmo discreto

Para la implementación de una posible solución al problema del logaritmo discreto se hizo uso del algoritmo *baby-step giant-step*, el cual se describió brevemente en la sección 2.4. Para este programa no fue necesario usar una librería criptográfica, debido a que mientras investigaba no encontré como tal una función que me ayudará a realizar el proceso de una manera más sencilla.

3.1.1 Consideraciones del programa

- Los valores correspondientes al generador, el módulo y el resultado se deben pasar como parámetros al momento de ejecutar el programa, respetando el orden en que se mencionaron.
- Con la finalidad de saber cuánto tiempo tarda el programa en encontrar la solución para cada uno de los casos a probar se hizo uso del módulo `timeit`, el cual proporciona una manera simple de medir los tiempos de ejecución.

```
timeit.default_timer()
```

Define un temporizador predeterminado, de manera específica dependiendo de la plataforma. En cualquier plataforma, `default_timer()` mide el tiempo del reloj de pared, no el tiempo de la CPU. Esto significa que otros procesos que se ejecutan en la misma computadora pueden inferir en este.

- Para elevar a la potencia un número y posteriormente aplicar el módulo se hizo uso de la función `pow` que ofrece Python, la cual permite realizar esta tarea en un paso.

```
pow( x, y, z )
```

La función recibe tres parámetros: x corresponde a la base, y es el exponente y z (opcional) es usado para indicar el módulo.

3.1.2 Implementación del programa

Se hizo uso de tres módulos (véase Listing 1): el primero fue utilizado para importar las funciones que permiten calcular la función techo y la raíz cuadrada de un número respectivamente; el segundo como se mencionó anteriormente se usó para calcular el tiempo de ejecución del programa; finalmente el tercero es

el que permite utilizar `sys.argv` para aceptar argumentos pasados por línea de comandos.

```
1 from math import ceil, sqrt
2 from timeit import default_timer
3 import sys
```

Listing 1: Paquetes a utilizar

Se implementó una función encargada de realizar todo el procedimiento del algoritmo *baby-step giant-step*, la cual recibe como parámetros los valores correspondientes al generador (g), el módulo (p) y el resultado (h) respectivamente. Lo primero que se hizo fue calcular el valor de $m = \lceil \sqrt{p} \rceil$, el cual es necesario para definir los intervalos de las funciones f_1 y f_2 (véase sección 2.4.1). A este valor se le sumó 1 con la finalidad de que los ciclos `for` fueran de $[0, m]$. Posteriormente, se inicializó el arreglo en el que se almacenarán los resultados obtenidos de la operación $f_1 = g^{mi} \bmod p$ donde i es el índice del primer ciclo (véase Listing 2).

```
1 def babyStepGiantStepAlgorithm( g, p, h ):
2     m = ceil( sqrt(p) ) + 1
3     giantStep = [0] * m
```

Listing 2: Declaración de función y variables a utilizar

Para calcular los datos correspondientes al *giant-step* es importante recordar los posibles valores que puede tomar el exponente: $i \in [1; \lceil \frac{p}{m} \rceil]$, debido a que m hace referencia a la raíz cuadrada de p , podemos concluir que el resultado de la división será el mismo que el de la primera variable, es decir, m . Por lo que, el primer ciclo `for` irá desde 1 hasta m .

La operación que se realiza en cada una de las iteraciones es $g^{mi} \bmod p$, los resultados obtenidos se van guardando en un arreglo, ya que como se recordará estos serán utilizados posteriormente. La implementación de este fragmento de código se muestra en el Listing 3.

```
1     for i in range( 1, m ):
2         giantStep[i] = pow( g, i * m, p )
```

Listing 3: Implementación de la primera parte del algoritmo: *giant-step*

Una vez teniendo lo anterior se procede a calcular los valores de f_2 , es decir, la segunda parte del algoritmo: *baby-step*. Recordando los intervalos vistos en la sección 2.4.1 ahora este ciclo `for` irá desde $[0, m]$ y la operación que se debe realizar es: $hg^i \bmod p$ donde i es el índice del ciclo.

Para cada uno de los resultados obtenidos es necesario revisar si se cumple que $f_1 = f_2$, es decir, buscar si ese valor también se obtuvo en *giant-step*, recorriendo el arreglo donde se almacenaron los resultados de ese paso. En Python la implementación de esto es un tanto sencillo, debido a que no se necesita hacer uso de otro ciclo `for`, basta con poner una condicional `if`, sin

embargo, internamente ambas maneras son equivalentes, ya que se hace una búsqueda lineal, es decir, se recorre todo el arreglo. Una vez que se cumpla esta condición necesitamos saber en qué posición se encuentra dentro del arreglo, esto se realizó con ayuda del método `index` pasando como parámetro el valor a buscar. Finalmente, se multiplica por m , es decir, por la raíz cuadrada de p y se le resta la posición en la que se encontró en el *baby-step*.

Un punto importante a considerar es que la respuesta debe pertenecer al grupo cíclico finito, es decir, debe ser menor que p . En caso contrario, aunque cumpla todos los requisitos anteriores no puede ser la solución final, para verificar esto se hizo uso de otra condicional `if`. Si el número candidato cumple esta última condición podemos concluir que se encontró la respuesta final, por lo que se retorna.

En caso de que no se haya encontrado ninguna coincidencia se retorna un `-1`, lo cual significa que no se encontró una solución. La implementación de *baby-step*, así como de las verificaciones se pueden observar en el Listing 4.

```

1     for i in range( 0, m ):
2         babyStep = ( pow( g, i, p ) * h ) % p
3         if babyStep in giantStep:
4             index = giantStep.index( babyStep )
5             result = index * m - i
6             if result < p:
7                 return result
8     return -1

```

Listing 4: Implementación de la segunda parte del algoritmo: *baby-step*

En el programa principal se empieza a contar el tiempo de ejecución antes de llamar a la función encargada de realizar el algoritmo *baby-step giant-step* y se finaliza una vez que se haya mostrado la solución y comprobación en consola.

Para realizar la comprobación, se elevó el generador a la solución encontrada por el algoritmo módulo p , el resultado de esta operación debe ser igual a h , en caso contrario es erróneo (véase Listing 5).

```

1 startTime = default_timer()
2 result = babyStepGiantStepAlgorithm( int(sys.argv[1]), int(sys.
   argv[2]), int(sys.argv[3]) )
3 print( " Result of Discrete logarithm problem: ", result )
4 print( " Checking result g^(x) = ", pow( int(sys.argv[1]),
   result, int(sys.argv[2]) ) )
5 endTime = default_timer()
6 print( " Execution time: ", endTime - startTime, "segundos" )

```

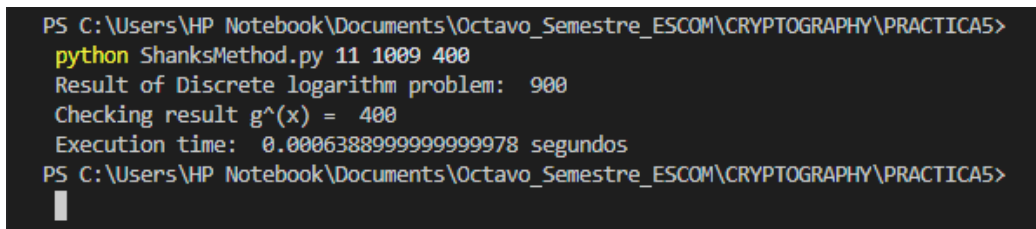
Listing 5: Programa principal

3.1.3 Resultados obtenidos para cada uno de los casos

a) $11^x \bmod 1009 = 400$

Para este inciso la respuesta obtenida por el programa fue $x = 900$ con un tiempo de ejecución de 0.00063 segundos. La comprobación dio el mismo valor de h , por lo que se puede concluir que el cálculo se realizó de manera correcta.

El programa en ejecución se puede observar en la Figura 2.



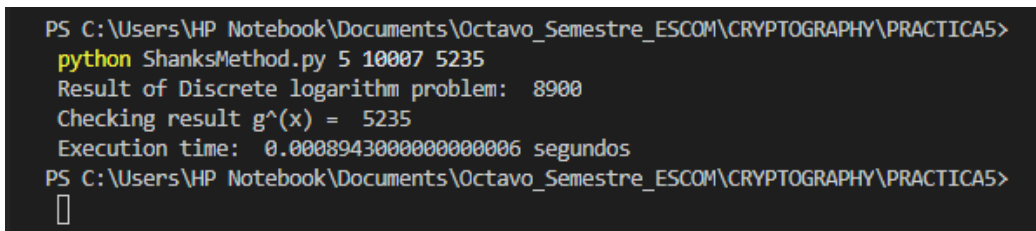
```
PS C:\Users\HP Notebook\Documents\Octavo_Semestre_ESCOM\CRYPTOGRAPHY\PRACTICA5>
python ShanksMethod.py 11 1009 400
Result of Discrete logarithm problem: 900
Checking result g^(x) = 400
Execution time: 0.0006388999999999978 segundos
PS C:\Users\HP Notebook\Documents\Octavo_Semestre_ESCOM\CRYPTOGRAPHY\PRACTICA5>
```

Figura 2: Programa en ejecución para el inciso a)

b) $5^x \bmod 10007 = 5235$

Para el inciso b) la respuesta obtenida por el programa fue $x = 8,900$ con un tiempo de ejecución de 0.00089 segundos. Como se puede observar este último dato no aumento mucho, sin embargo, esto se debe a que se sigue trabando con valores un tanto pequeños.

Por otro lado, la comprobación dio el mismo valor de h , concluyendo así que el cálculo se realizó de manera correcta. El programa en ejecución se puede observar en la Figura 3.



```
PS C:\Users\HP Notebook\Documents\Octavo_Semestre_ESCOM\CRYPTOGRAPHY\PRACTICA5>
python ShanksMethod.py 5 10007 5235
Result of Discrete logarithm problem: 8900
Checking result g^(x) = 5235
Execution time: 0.0008943000000000006 segundos
PS C:\Users\HP Notebook\Documents\Octavo_Semestre_ESCOM\CRYPTOGRAPHY\PRACTICA5>
```

Figura 3: Programa en ejecución para el inciso b)

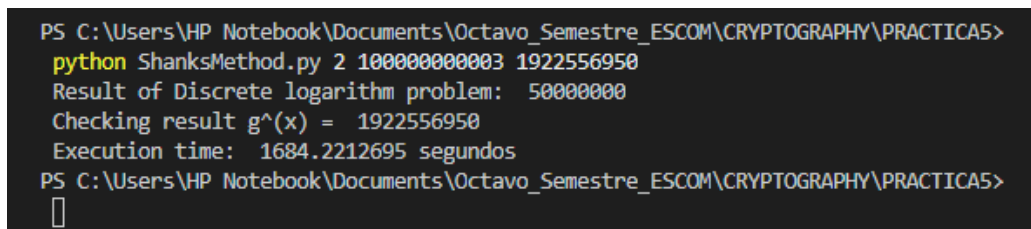
c) $2^x \bmod 100,000,000,003 = 1,922,556,950$

Para este inciso la respuesta obtenida fue $x = 50,000,000$ con un tiempo de ejecución de 1684.2212 segundos ≈ 28.07 minutos.

Al principio se me hizo extraño que el programa tardara tanto en encontrar la solución a este problema, debido a que el valor de x para este

caso es mucho menor a comparación del obtenido en el inciso d, el cual se resolvió en un menor tiempo. Revisando un poco me encontré con una posible explicación: en Internet hay diversas implementaciones de este algoritmo en las cuales almacenan de diferente manera los resultados obtenidos en *giant-step*, evitando así la necesidad de recorrer todo el arreglo en el que se almacenan los resultados, lo cual reduce la complejidad y por lo tanto el tiempo de ejecución, sin embargo, hace uso de más memoria. Debido a que mi implementación la hice de diferente manera, y la raíz cuadrada de p para este caso es mucho mayor a comparación de los demás incisos; por consiguiente el tamaño del arreglo aumenta considerablemente, tomando más tiempo recorrerlo en cada iteración del segundo ciclo **for**. Concluyendo así que mientras más grande sea el valor de p el tiempo de ejecución será mucho mayor aunque el valor de x encontrado sea pequeño.

Por otro lado, la comprobación dio el mismo valor de h , por lo que podemos decir que el cálculo se realizó de manera correcta, aunque el tiempo que tomó resolverlo es ineficiente. El programa en ejecución se puede observar en la Figura 4.



```

PS C:\Users\HP Notebook\Documents\Octavo_Semestre_ESCOM\CRYPTOGRAPHY\PRACTICA5>
python ShanksMethod.py 2 10000000003 1922556950
Result of Discrete logarithm problem: 50000000
Checking result  $g^x = 1922556950$ 
Execution time: 1684.2212695 segundos
PS C:\Users\HP Notebook\Documents\Octavo_Semestre_ESCOM\CRYPTOGRAPHY\PRACTICA5>

```

Figura 4: Programa en ejecución para el inciso c)

d) $3^x \bmod 500,000,009 = 406,870,124$

Para el inciso d) la respuesta obtenida por el programa fue $x = 400,000,000$ con un tiempo de ejecución de 4.8861 segundos. Como se puede observar este último dato es mucho menor al obtenido en el inciso anterior, debido a que el valor de p es más pequeño y, por consiguiente el tamaño del arreglo a recorrer para encontrar un valor que satisfaga la condicional $f_1 = f_2$ se reduce significativamente.

Por otro lado, la comprobación dio el mismo valor de h , concluyendo así que el cálculo se realizó de manera correcta. El programa en ejecución se puede observar en la Figura 5.

```
PS C:\Users\HP Notebook\Documents\Octavo_Semestre_ESCOM\CRYPTOGRAPHY\PRACTICA5>
python ShanksMethod.py 3 500000009 406870124
Result of Discrete logarithm problem: 400000000
Checking result  $g^x = 406870124$ 
Execution time: 4.8861939 segundos
PS C:\Users\HP Notebook\Documents\Octavo_Semestre_ESCOM\CRYPTOGRAPHY\PRACTICA5>
```

Figura 5: Programa en ejecución para el inciso d)

e) $3^x \bmod 500,000,009 = 187,776,257$

Para este inciso la respuesta obtenida fue $x = 500,000,000$ con un tiempo de ejecución de 6.1580 segundos. La base y el módulo son los mismos que el del inciso anterior. A pesar de que el valor de h para este caso es menor, el valor de x que cumplía las condiciones es mayor, por lo que tardó un poco más de tiempo en resolver el problema. Sin embargo, nos podemos dar cuenta que la diferencia no es mucha.

Por otro lado, la comprobación dio el mismo valor de h , concluyendo así que el cálculo se realizó de manera correcta. El programa en ejecución se puede observar en la Figura 6.

```
PS C:\Users\HP Notebook\Documents\Octavo_Semestre_ESCOM\CRYPTOGRAPHY\PRACTICA5>
python ShanksMethod.py 3 500000009 187776257
Result of Discrete logarithm problem: 500000000
Checking result  $g^x = 187776257$ 
Execution time: 6.1580534 segundos
PS C:\Users\HP Notebook\Documents\Octavo_Semestre_ESCOM\CRYPTOGRAPHY\PRACTICA5>
```

Figura 6: Programa en ejecución para el inciso e)

3.2 Encontrando factores primos de números compuestos

Para la implementación de este programa, se tomaron en consideración los aspectos vistos en la parte teórica, sin embargo, existen más métodos. Al igual que en el programa anterior no fue necesario usar una librería criptográfica, debido a que no encontré funciones que me ayudarían a realizar el procedimiento de una manera más sencilla.

3.2.1 Consideraciones del programa

- El número compuesto se debe pasar como parámetro al momento de ejecutar el programa, es decir, por línea de comando.

- Se hizo uso de un diccionario para almacenar los valores que se van obteniendo en forma de parejas {factor : potencia de ese factor}, una vez teniendo todos los factores se elevaron a su potencia correspondiente.

3.2.2 Implementación del programa

Nuevamente se hizo uso de los tres módulos explicados en la sección 3.1.2. (véase Listing 6): el primero permite importar las funciones encargadas de calcular la función techo y la raíz cuadrada de un número respectivamente; el segundo se utilizó para calcular el tiempo de ejecución del programa; mientras que el tercero permite aceptar argumentos pasados por línea de comando haciendo uso de `sys.argv`.

```
1 from math import ceil, sqrt
2 from timeit import default_timer
3 import sys
```

Listing 6: Paquetes a utilizar

Se diseñaron dos funciones: la primera realiza el algoritmo encargado de encontrar los factores primos del número compuesto y los almacena en el diccionario en forma de parejas. Posteriormente, la segunda es la encargada de elevar cada uno de los factores a su respectiva potencia.

Para la implementación de la primera función se partió del siguiente hecho, el cual también se vio en la parte teórica: “Si un número n es compuesto, se verifica que ha de tener un divisor primo menor o igual que su raíz cuadrada”. Por lo que, lo primero que se realizó fue calcular la raíz cuadrada del número, posteriormente se inicializó el diccionario, así como un contador que servirá para definir la potencia de cada uno de los factores (véase Listing 7).

```
1 def findPrimeFactors( number ):
2     rango = ceil( sqrt( number ) )
3     numbers = {}
4     count = 0
```

Listing 7: Declaración de la función e inicialización de variables a utilizar

Una vez teniendo lo anterior, se verifica si el número compuesto ingresado por el usuario es divisible entre dos, debido a que es el primer número primo posible. En caso de que lo sea, se reduce lo más que se pueda por este factor y se almacena en el diccionario con su respectiva potencia. Para esto, se hizo uso de un ciclo `while`, el cual se repite hasta que el número ya no sea múltiplo de 2, aumentando la potencia con ayuda del contador y modificando su valor dentro del diccionario haciendo uso del método `update` (véase Listing 8).

```
1 while number % 2 == 0 :
2     number = number / 2
3     count += 1
```

```
4 numbers.update({ 2: count })
```

Listing 8: Reducción del número compuesto por el factor 2

Como resultado de la operación anterior se obtiene un valor el cual ya no se puede reducir por un número par, lo cual es de gran ayuda para definir el siguiente ciclo.

Para obtener otros factores primos del número compuesto se hizo uso de un ciclo **for**, el cual va de $[3, \lceil \sqrt{n} \rceil]$ (por la propiedad mencionada anteriormente). Debido a que n ya no es un múltiplo de 2, se descartan los números pares, incrementando así de dos en dos nuestro ciclo, es decir, se tomarán en cuenta únicamente los impares.

La operación a realizar dentro del **for** es la misma que la del ciclo **while** explicada anteriormente, se dividirá lo más que se pueda el número compuesto para cada uno de los números impares, almacenando el factor y su potencia en caso de que i sea un divisor de n . Con este método no es necesario verificar si el índice es un número primo o no, debido a que se van descartando los que no lo son. Por ejemplo, 9 no es primo, sin embargo, cuando i sea igual a este valor y se verifique si es divisor del número compuesto el resultado será falso, debido a que anteriormente se dividió lo más que se pudo entre 3 el cual sí es primo, descartando así todos los múltiplos de este. La implementación de este fragmento de código se muestra en el Listing 9.

```
1 for i in range( 3, rango + 1, 2 ) :
2     count = 0
3     while number % i == 0 :
4         number = number / i
5         count += 1
6         numbers.update({ i: count })
```

Listing 9: Reducción del número compuesto por los demás factores posibles hasta la raíz de n

En caso de que todos los factores primos sean menores o igual a la raíz cuadrada de n , el resultado de la variable **number** al finalizar el ciclo **for** será 1, de lo contrario aún queda otro valor por encontrar. Tomando en cuenta lo anterior, con ayuda de un **if** se verificó esta condición, en caso de que no se cumpla almacena en el diccionario el factor faltante y su potencia la cual es igual a 1, fue necesario hacerle un cast a **int** debido a que en cada iteración de los ciclos anteriores se divide el número entre el factor a revisar obteniendo como resultado un **float** (véase Listing 10).

Para finalizar, retorno el diccionario que contiene todos los factores encontrados con sus respectivas potencias.

```
1 if number != 1:
2     numbers.update({ int(number): 1 })
3
```

```
4 return numbers
```

Listing 10: Verificar si existe otro factor primo

La segunda función recibe como único parámetro el diccionario obtenido anteriormente. Con ayuda de un ciclo `for` se recorre obteniendo en cada iteración el valor de la clave y su pareja, posteriormente se hace uso de la función `pow` y se imprime en consola cada uno de los factores obtenidos elevados a su potencia correspondiente (véase Listing 11).

```
1 def obtainPower( numbers ):
2     for power in numbers:
3         print( pow( power, numbers[power] ), " ", end = " " )
```

Listing 11: Función encargada de elevar cada uno de los factores a su respectiva potencia

El tiempo de ejecución se comienza a tomar antes de llamar a la función encargada de encontrar los factores primos del número compuesto y finaliza una vez que se imprimieron los valores en sus respectivas potencias, es decir, cuando termina de ejecutarse la segunda función. Con la finalidad de ver cada uno de los factores y cuántas veces se repitió, también se imprimió en consola el diccionario resultante de la primera función.

```
1 startTime = default_timer()
2 numbers = findPrimeFactors( int(sys.argv[1]) )
3 print( " Prime factors {number : power} = ", numbers )
4 print( " Factors number^(power): ", end = " " )
5 obtainPower( numbers )
6 endTime = default_timer()
7 print()
8 print( " Execution time: ", endTime - startTime, "segundos" )
```

Listing 12: Programa principal

3.2.3 Resultados obtenidos para cada uno de los casos

a) 100,160,063

Para este inciso los factores primos obtenidos fueron 10,007 y 10,009 con un tiempo de ejecución de 0.00098 segundos. A diferencia del primer programa, la comprobación la realice haciendo uso de la siguiente calculadora online: <https://www.alpertron.com.ar/ECM.HTM> la cual acepta cantidades muy grandes de números compuestos.

Los resultados concuerdan con lo que arroja la calculadora, adicional a esto hice la operación de multiplicar estos factores aparte y sí da el número compuesto, por lo que se puede concluir que el programa se ejecuto exitosamente. El programa en ejecución se puede observar en la Figura 7.

```

PS C:\Users\HP Notebook\Documents\Octavo_Semestre_ESCOM\CRYPTOGRAPHY\PRACTICA5>
python PrimeFactors.py 100160063
Prime factors {number : power} = {10007: 1, 10009: 1}
Factors number^(power): 10007 10009
Execution time: 0.000987899999999999 segundos
PS C:\Users\HP Notebook\Documents\Octavo_Semestre_ESCOM\CRYPTOGRAPHY\PRACTICA5>

```

Figura 7: Programa en ejecución para el inciso a)

b) 10,006,200,817

Para el inciso b) los factores obtenidos fueron 100,019 y 100,043 ambos con una potencia igual a uno, con un tiempo de ejecución de 0.01096 segundos. Como se puede observar este último dato no aumento mucho, sin embargo, esto se debe a que se sigue trabando con valores un tanto pequeños.

Aquí se ve la importancia de la verificación que se hizo para saber si existía otro factor primo mayor a la raíz cuadrada del número, debido a que el segundo valor obtenido cumple esta condición y en caso de no haberla puesto, no lo hubiera encontrado.

Por otro lado, la comprobación dio los mismos valores, concluyendo así que el cálculo se realizó de manera correcta. El programa en ejecución se puede observar en la Figura 8.

```

PS C:\Users\HP Notebook\Documents\Octavo_Semestre_ESCOM\CRYPTOGRAPHY\PRACTICA5>
python PrimeFactors.py 10006200817
Prime factors {number : power} = {100019: 1, 100043: 1}
Factors number^(power): 100019 100043
Execution time: 0.010967900000000003 segundos
PS C:\Users\HP Notebook\Documents\Octavo_Semestre_ESCOM\CRYPTOGRAPHY\PRACTICA5>

```

Figura 8: Programa en ejecución para el inciso b)

c) 250,035,001,189

Los factores primos obtenidos para este caso fueron 500,029 y 500,041 ambos con una potencia igual a uno, con un tiempo de ejecución de 0.0624 segundos. A pesar de que este número es mayor al del inciso anterior, el tiempo de ejecución no aumentó tanto.

Al hacer uso de la calculadora online los resultados fueron los mismos, por lo que se puede concluir que los cálculos se realizaron de manera correcta. El programa en ejecución se muestra en la Figura 9.


```

PS C:\Users\HP Notebook\Documents\Octavo_Semestre_ESCOM\CRYPTOGRAPHY\PRACTICA5>
python PrimeFactors.py 250035001189
Prime factors {number : power} = {500029: 1, 500041: 1}
Factors number^(power): 500029 500041
Execution time: 0.062440699999999995 segundos
PS C:\Users\HP Notebook\Documents\Octavo_Semestre_ESCOM\CRYPTOGRAPHY\PRACTICA5>

```

Figura 9: Programa en ejecución para el inciso c)

d) 250,000,009,000,000,081

Finalmente se tiene el inciso d) cuyo número consta de 18 cifras. El resultado obtenido para este caso fue 500,000,009 con una potencia de dos, es decir, corresponde al valor de la raíz cuadrada del número compuesto. Es por esto, que al momento de elevar el factor a su potencia correspondiente imprime el número original.

Por otro lado, el tiempo de ejecución fue de 66.2437 segundos \approx 1 minuto. Con esto podemos concluir que la implementación del programa es un tanto eficiente, debido a que el tiempo no aumentó tan drásticamente para cantidades hasta cierto punto grandes como en el caso del algoritmo *baby-step giant-step*.

La comprobación dio los mismos valores, por lo que los cálculos se realizaron de manera correcta. El programa en ejecución se muestra en la Figura 10.

```

PS C:\Users\HP Notebook\Documents\Octavo_Semestre_ESCOM\CRYPTOGRAPHY\PRACTICA5>
python PrimeFactors.py 250000009000000081
Prime factors {number : power} = {500000009: 2}
Factors number^(power): 250000009000000081
Execution time: 66.2437226 segundos
PS C:\Users\HP Notebook\Documents\Octavo_Semestre_ESCOM\CRYPTOGRAPHY\PRACTICA5>

```

Figura 10: Programa en ejecución para el inciso d)

References

- [1] J. L. B. Gómez, *UF1846 - Desarrollo de aplicaciones web distribuidas*. Arganda del Rey, Madrid: Paraninfo, 2016.
- [2] C. Paar and J. Pelzl, *Understanding Cryptography*. London New York: Springer, 2010.
- [3] “Discrete logarithm,” CP-Algorithms, accedido 06-05-20. [Online]. Available: <https://cp-algorithms.com/algebra/discrete-log.html>
- [4] “Los números primos. teorema fundamental de la aritmética,” accedido 06-05-20. [Online]. Available: http://www.dma.fi.upm.es/recursos/aplicaciones/matematica_discreta/web/aritmetica_modular/primos.html
- [5] “Examples of how to find the prime factorization of a number,” accedido 06-05-20. [Online]. Available: https://www.mesacc.edu/~scotz47781/mat120/notes/radicals/simplify/images/examples/prime_factorization.html