

# CRIPTOGRAFÍA PARA INGENIER@S

## Class4crypt

© Jorgeramió 2022

Aula virtual de  
criptografía  
aplicada

Diapositivas  
utilizadas en las  
clases grabadas  
de Class4crypt

Módulo 3 Complejidad algorítmica en la criptografía

Dr. Jorge Ramió Aguirre © 2022



Attribution-NonCommercial-  
NoDerivatives 4.0 International  
(CC BY-NC-ND 4.0)

# Class4crypt

## Tu aula virtual de criptografía aplicada



<https://www.youtube.com/user/jorgeramio>

Dr. Jorge Ramío Aguirre

*El ingenio es intrínseco al ser humano,  
solo hay que darle una oportunidad  
para que se manifieste.*

<https://www.criptored.es/cvJorge/index.html>

- Módulo 1. Principios básicos de la seguridad
- Módulo 2. Matemáticas discretas en la criptografía
- ➔ Módulo 3. Complejidad algorítmica en la criptografía
- Módulo 4. Teoría de la información en la criptografía
- Módulo 5. Fundamentos de la criptografía
- Módulo 6. Algoritmos de criptografía clásica
- Módulo 7. Funciones hash
- Módulo 8. Criptografía simétrica en bloque
- Módulo 9. Criptografía simétrica en flujo
- Módulo 10. Criptografía asimétrica

# Class4crypt

## Módulo 3. Complejidad algorítmica en la criptografía

3.1. Fundamentos de complejidad algorítmica

3.2. El problema de la mochila

3.3. El problema del logaritmo discreto

3.4. El problema de la factorización entera

Lista de reproducción del módulo 3 en el canal Class4crypt

<https://www.youtube.com/playlist?list=PLq6etZPDh0ks90M46G9vnfUCCTrQyMpeE>

# Class4crypt c4c3.1

## Módulo 3. Complejidad algorítmica en la criptografía

### Lección 3.1. Fundamentos de complejidad algorítmica

3.1.1. Introducción a la teoría de la complejidad algorítmica

3.1.2. Problemas de tipo P y N

3.1.3. Introducción al problema de la mochila

3.1.4. Introducción al problema del logaritmo discreto

3.1.5. Introducción al problema de la factorización entera

Class4crypt c4c3.1 Fundamentos de complejidad algorítmica

<https://www.youtube.com/watch?v=D7oP9UbhCf4>

# Teoría de la complejidad algorítmica

- La teoría de la complejidad de los algoritmos permitirá, entre otras cosas, conocer la fortaleza de un algoritmo criptográfico y tener así una idea de su vulnerabilidad computacional
- Los algoritmos pueden clasificarse según su tiempo de ejecución, en función del tamaño de la entrada. Hablamos así de algoritmos con complejidad polinomial  $P$  (tiempo *lineal*) y de algoritmos con complejidad polinomial no determinista  $NP$  (tiempo *exponencial*)
- Los primeros darán lugar a problemas de *fácil* o rápida solución y los segundos darán lugar a problemas de *difícil* o lenta solución
- El uso de ambos será muy interesante en la criptografía

# Objetivo de esta lección

- No pretende ser una clase al uso en la que se estudien y analicen los principios y alcances de la complejidad algorítmica
- Sólo se desea presentar unos conocimientos básicos y mínimos en esta temática, para poder luego entender el uso de este tipo de problemas en la criptografía moderna, en particular:
  - El problema del logaritmo discreto, que se usa en el intercambio de clave de Diffie y Hellman, en el cifrado y firma de Elgamal, en la firma DSA y en criptografía con curvas elípticas ECC (*Elliptic Curve Cryptography*), en este último caso usando el ECDLP (*Elliptic Curve Discrete Logarithm Problem*), algo en lo que no profundizaremos en este curso
  - El problema de la factorización entera, que se usa en el algoritmo RSA



# La función $O(n)$

- En criptografía deseamos algoritmos con los que sea muy fácil y rápido cifrar pero que su ataque requiera de muchísimo cómputo
- Las operaciones que realice un algoritmo dependerán del tamaño de la entrada  $n$  y su resolución tendrá una cota superior que se define como  $O(n)$ , de forma coloquial  $O$  grande de  $n$
- Esto conlleva una complejidad que se expresa en términos del tiempo  $T$  necesario para el cálculo del algoritmo y del espacio  $S$  que su ejecución utiliza en memoria
- Esta complejidad se representará con la función  $f(n) = O(g(n))$ , donde  $n$  es el tamaño de la entrada

# Definición y complejidad de la función $f(n)$

- $f = O(n)$  ssi  $\exists c_0, n_0 / f(n) \leq c_0 * g(n)$
- Es decir, sí y solo sí existen valores  $c_0$  y  $n_0$  tales que  $f(n) \leq c_0 * g(n)$
- Por ejemplo, si  $f(n) = 4n^2 + 2n + 5$ , ¿será  $f = O(n^2)$ ?
- La pregunta que debemos hacernos es si se cumple que:
  - $c_0 * g(n) = c_0 * n^2 \geq f(n)$ , con  $n = 1, 2, 3$ , etc. para un valor  $c_0$  dado
- Ejemplo: vamos a evaluar la ecuación  $c_0 * n^2$  para  $n = 1, 2, 3, \dots$  comparar este valor con el resultado de  $f(n) = 4n^2 + 2n + 5$  y comprobar si para algún  $n$  se cumple que  $c_0 * n^2 \geq 4n^2 + 2n + 5$
- Sea en este caso  $c_0 = 6$



# Comprobando que $f = O(n^2)$

Si  $f(n) = 4n^2 + 2n + 5$  ¿se cumple que  $c_0 * g(n) = c_0 * n^2 \geq f(n)$ ?

$c_0$	$n$	$c_0 * n^2$	$f(n) = 4n^2 + 2n + 5$		$¿c_0 * n^2 \geq f(n)?$	Se cumple siempre
6	1	6	11	$11 > 6$	No	
6	2	24	25	$25 > 24$	No	
6	3	54	47	$47/54=0,87$	Sí	
6	4	96	77	$77/96=0,80$	Sí	
6	5	150	115	$115/150=0,77$	Sí	

# Tiempos de ejecución de un algoritmo

- En la expresión  $O(n)$  aparecerá el término que domina al crecer el valor de  $n$
- El tiempo de ejecución de un algoritmo  $T1$  que realiza  $2n+1$  operaciones será de tipo  $O(n)$
- El tiempo de ejecución de un algoritmo  $T2$  que realiza  $3n^2+n+3$  operaciones será de tipo  $O(n^2)$
- Sean  $a$  y  $b$  dos números enteros positivos menores o iguales que  $n$ 
  - Para la suma  $(a + b)$ :  $O(\log_2(a) + \log_2(b)) = O(\log_2(n))$
  - Para la multiplicación  $(a*b)$ :  $O(\log_2(a) * \log_2(b)) = O((\log_2(n))^2)$

# Algoritmos de complejidad polinomial

- Un algoritmo se dice que tiene tiempo de ejecución polinomial si dicho tiempo depende polinómicamente del tamaño de la entrada
- Si la entrada es de tamaño  $n$  y  $t$  es un entero, la complejidad será  $O(\log^t n)$ 
  - Si  $t = 1$ , se dice que el sistema es lineal (por ejemplo la suma de dos números)
  - Si  $t = 2$ , se dice que el sistema es cuadrático (por ejemplo el producto de dos números)
  - Si  $t = 3$ , se dice que el sistema es cúbico (por ejemplo encontrar el máximo común divisor MCD con el algoritmo Euclides)

# Ejemplo de complejidad polinomial

- **Pregunta.** El tiempo de ejecución de un algoritmo es  $O(\log^3 n)$ . Si doblamos el tamaño de la entrada, ¿en cuánto aumentará este tiempo?
- **Solución.** En el primer caso el tiempo es  $O(\log^3 n)$  y en el segundo  $O(\log^3 2n)$ . Para este sistema polinomial, el tiempo de cómputo se incrementará sólo en  $\log^3 2$
- Estos son los denominados problemas fáciles y son los que involucrarán un proceso de cifrado y descifrado (o firma digital y su posterior comprobación) por parte del o de los usuarios autorizados

# Algoritmos de mayor complejidad

- Si la entrada es de tamaño  $n$  y  $t$  es un entero, otro orden de complejidad será  $O(n^t)$
- Para  $t = 2$ , el algoritmo tendrá una complejidad cuadrática
- Para  $t = 3$ , el algoritmo tendrá una complejidad cúbica
- ¿Y si tuviésemos una complejidad  $O(a^n)$ ?... ¡incluso sería mejor!
- Algoritmos de complejidad polinomial no determinista
  - Un algoritmo se dice que tiene tiempo de ejecución polinomial no determinista si este tiempo depende exponencialmente del tamaño de la entrada

# Ejemplo de complejidad no determinista

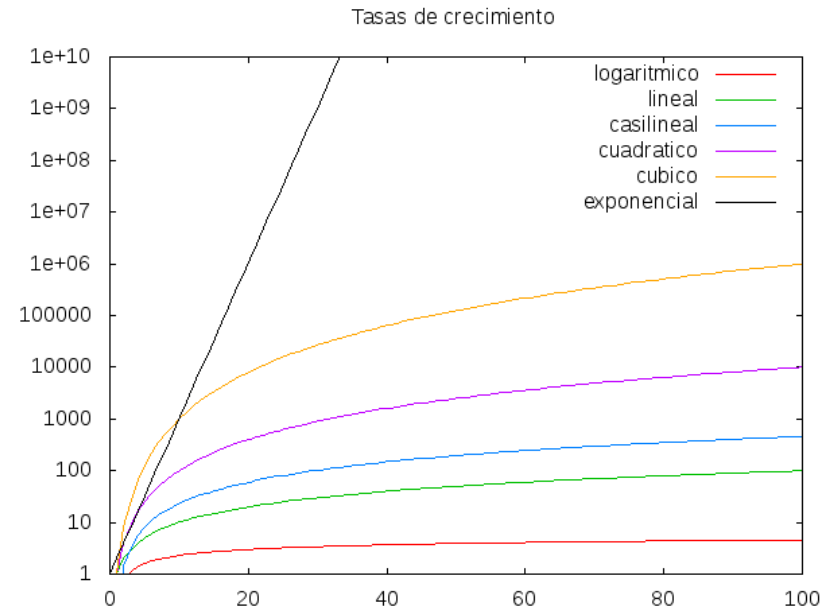
- **Pregunta.** El tiempo de ejecución de un algoritmo es  $O(2^n)$ . Si doblamos el tamaño de la entrada, ¿en cuánto aumentará este tiempo?
- **Solución.** En el primer caso el tiempo es  $O(2^n)$  y en el segundo caso  $O(2^{2n})$ . Duplicando la entrada, el tiempo de cómputo para este sistema exponencial se multiplicará por  $2^n$
- Por ejemplo, sea  $n_1 = 10$  y  $n_2 = 20$  ( $n_2 = 2 * n_1$ )
  - Si  $n_1 = 10$ , entonces  $2^{10} = 1.024$
  - Si  $n_2 = 20$ , entonces  $2^{20} = 1.048.576$
  - Sólo duplicando  $n$ , el valor inicial se ha multiplicado por 1.024 veces

# Comparativas de complejidad

- Los algoritmos polinómicos y exponenciales se comparan por sus órdenes de complejidad

- Constante  $O(1)$
- Logarítmica  $O(\log n)$
- Lineal  $O(n)$
- Casi lineal  $O(n \log n)$
- Cuadrática  $O(n^2)$
- Cúbica  $O(n^3)$
- Exponencial  $O(a^n)$

donde  $a$  es una constante



José A. Alonso Jiménez, Grupo de Lógica Computacional, Dpto. de Ciencias de la Computación e I.A., Universidad de Sevilla (ver referencia en Bibliografía)



# Tiempos de ejecución según complejidad

- Supongamos un ordenador que puede realizar  $10^{10}$  instrucciones por segundo

Entrada	$O(n)$ seg	$O(n^2)$ seg	$O(n^3)$ seg	$O(2^n)$ seg
$n = 10$	$10^{-9}$	$10^{-8}$	$10^{-7}$	$10^{-7}$
$n = 10^2$	$10^{-8}$	$10^{-6}$	$10^{-4}$	$4 \cdot 10^{12}$ años
$n = 10^3$	$10^{-7}$	$10^{-4}$	$10^{-1}$	$3 \cdot 10^{293}$ años




Incrementos de un  
orden de magnitud

Computacionalmente  
imposible

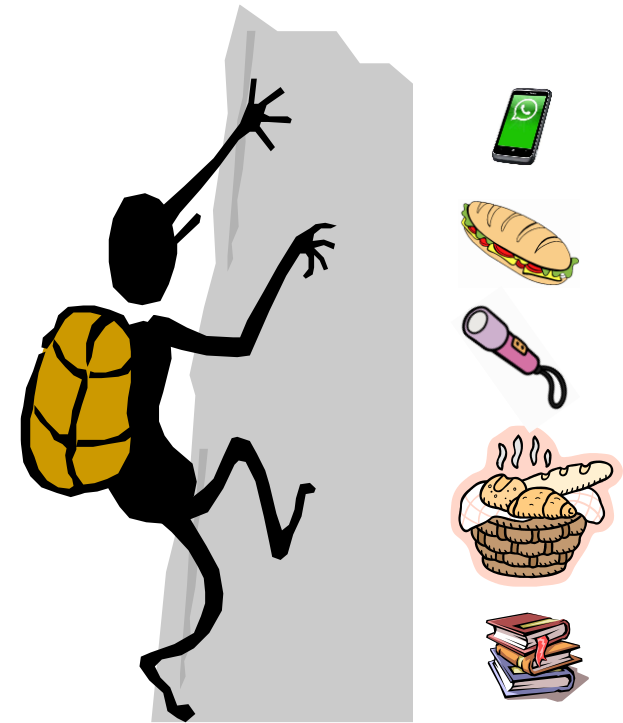
Entrada/ $10^{10}$ . Para  $n = 10^2 = 100 \Rightarrow O(n^2) = 100^2 / 10^{10} = 10^{-6}$  segundos

# Problemas tipo P y NP

- P (Polynomial time), NP (Nondeterministic Polynomial time)
- Siguiendo el documento Complejidad computacional de Daniel Gayo, de la asignatura Algorítmica y Lenguajes de Programación, Universidad de Oviedo:
  - “La clase de problemas P está formada por todos aquellos problemas de decisión para los cuales se tiene un algoritmo de solución que se ejecuta en tiempo polinomial en una máquina determinista”
  - “La clase de problemas NP está formado por todos aquellos problemas de decisión para los cuales existe un algoritmo de solución que se ejecuta en tiempo polinomial en una (hipotética) máquina no determinista. Dicho de otro modo, no se ha encontrado un algoritmo determinista que lo resuelva en tiempo polinomial”
- Algunos problemas NP interesantes en la criptografía 

# El problema de la mochila

- Dada una mochila de determinadas dimensiones de alto, ancho y fondo, y un conjunto de elementos de distintos tamaños menores que ella y de cualquier dimensión, ... ¿es posible llenar la mochila al 100% con distintos elementos de ese conjunto sin repetir ninguno de ellos?
- Es un problema de tipo NP en el que el algoritmo debe realizar en cada paso una selección iterativa entre diferentes opciones



# El problema del logaritmo discreto

- Dados los enteros  $\alpha$ ,  $x$  y un cuerpo  $p$ , resolver la operación
  - $\beta = \alpha^x \bmod p$
- Es un problema polinomial de baja complejidad algorítmica, incluso con números  $\alpha$ ,  $x$  y  $p$  grandes
- No obstante, conocidos los números  $\beta$ ,  $\alpha$  y el primo  $p$ , encontrar ahora un entero  $x$  de forma que
  - $x = \log_{\alpha} \beta \bmod p$
- Es un problema NP computacionalmente intratable si  $p$  es muy grande

# El problema de la factorización entera

- Dados dos números primos  $p$  y  $q$ , multiplicarlos para obtener el producto  $n$ 
  - $p * q = n$
- Es un problema de tipo polinomial, de fácil solución incluso con primos muy grandes
- Pero, conocido el producto  $n$ , encontrar los factores  $p$  y  $q$  de  $n$  es un problema de tipo NP, que es computacionalmente intratable si tanto  $p$  como  $q$  son muy grandes
  - $n = p * q$

# Conclusiones de la Lección 3.1

- La complejidad algorítmica nos permite diseñar algoritmos criptográficos cuya resolución sea rápida y sencilla para los intervinientes válidos, pero por el contrario se vuelva computacionalmente intratable (en tiempo y recursos) para intrusos o atacantes que no conocen una clave, trampa o certificado
- En el primer caso solucionaremos problemas en tiempo polinomial P (*lineal, cuadrático, ...*), y en el segundo caso en tiempo polinomial no determinista NP (*exponencial*)
- La complejidad de estos algoritmos se denota como  $O(n)$ , que es el término que domina al crecer el valor de  $n$ , siendo  $n$  el tamaño de entrada
- Existen varios problemas de este tipo que se usan en criptografía, como los problemas de la mochila, del logaritmo discreto, del logaritmo discreto en curvas elípticas y de la factorización entera

# Lectura recomendada (1/2)

- Análisis de la complejidad de los algoritmos, José A. Alonso Jiménez, Universidad de Sevilla
  - <https://www.cs.us.es/~jalonso/cursos/i1m-19/temas/tema-28.html>
- Criptografía y Seguridad en Computadores, Versión 5-0.1.4, José Manuel Lucena, capítulo 4, noviembre de 2019
  - <http://criptografiayseguridad.blogspot.com/p/criptografia-y-seguridad-en.html>
- Algoritmos y Complejidad, Complejidad Computacional, Pablo R. Fillottrani, Depto. Ciencias e Ingeniería de la Computación, Universidad Nacional del Sur, 2017
  - <http://www.cs.uns.edu.ar/~prf/teaching/AyC17/downloads/Teoria/Complejidad-1x1.pdf>



# Lectura recomendada (2/2)

- Algorítmica y Lenguajes de Programación, Complejidad computacional, Daniel Gayo, Universidad de Oviedo
  - <http://di002.edv.uniovi.es/~dani/asignaturas/transparencias-leccion19.PDF>
- Máquina de Turing determinista y no determinista, Wikipedia
  - [https://es.wikipedia.org/wiki/M%C3%A1quina\\_de\\_Turing#M%C3%A1quina\\_de\\_Turing\\_determinista\\_y\\_no\\_determinista](https://es.wikipedia.org/wiki/M%C3%A1quina_de_Turing#M%C3%A1quina_de_Turing_determinista_y_no_determinista)

# Class4crypt c4c3.2

## Módulo 3. Complejidad algorítmica en la criptografía

### Lección 3.2. El problema de la mochila

3.2.1. Enunciado simple del problema de la mochila

3.2.2. Resolución de un problema de mochila

3.2.3. Uso en criptografía: mochila tramposa de Merkle y Hellman

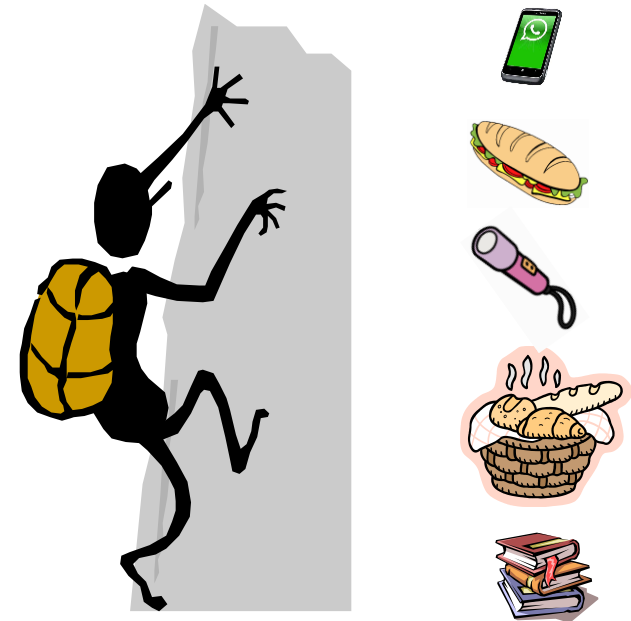
3.2.4. Ejemplo de cifrado asimétrico con mochila de Merkle y Hellman

Class4crypt c4c3.2 El problema de la mochila

[https://www.youtube.com/watch?v=M3ZG9\\_GWkzc](https://www.youtube.com/watch?v=M3ZG9_GWkzc)

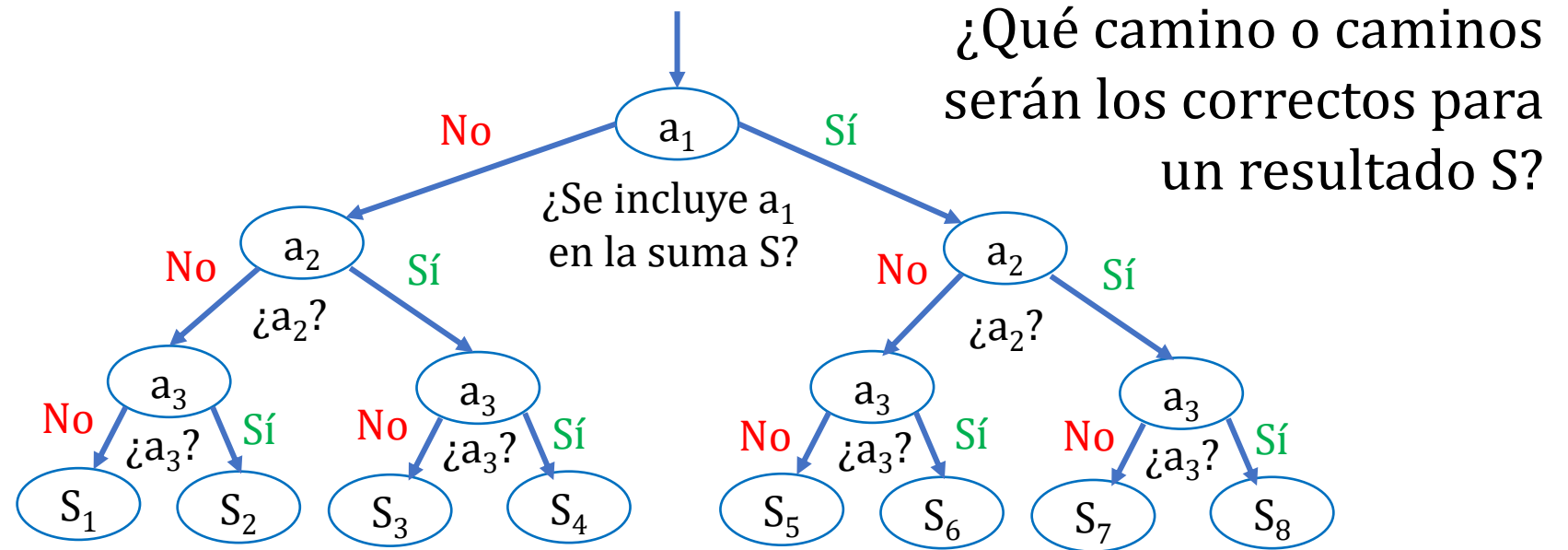
# El problema de la mochila

- Dada una mochila de dimensiones de alto, ancho y fondo conocidas, y un conjunto de elementos de distintos tamaños menores que ella y de cualquier dimensión, ... ¿es posible llenar la mochila al 100% con distintos elementos de ese conjunto?
- Dichos elementos, además de un volumen, pueden tener un peso específico o valor
- Resolución: vuelta atrás (*backtracking*), ramificación y poda (*branch and bound*)



# Mochila con solución por fuerza bruta

Sea  $A = \{a_1, a_2, a_3\}$   
y el objetivo  $S$  (un  
valor resultado de  
una suma con los  
elementos de  $A$ )



Habrà  $2^3 = 8$  estados (si  $A$  tiene  $n$  números, habrá  $2^n$ , carácter exponencial)

$$S_1 = \emptyset$$

$$S_2 = a_3$$

$$S_3 = a_2$$

$$S_4 = a_2 + a_3$$

$$S_5 = a_1$$

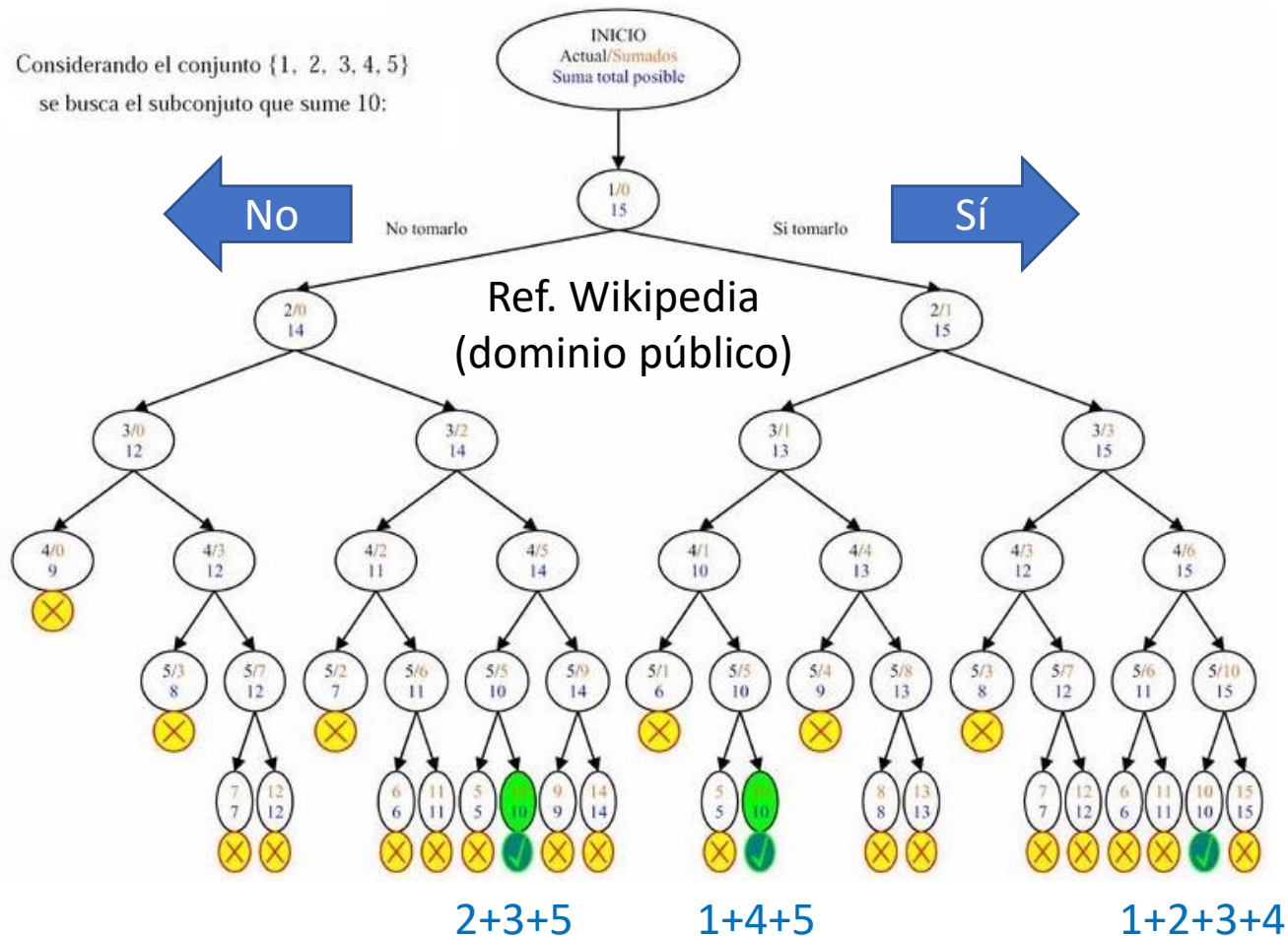
$$S_6 = a_1 + a_3$$

$$S_7 = a_1 + a_2$$

$$S_8 = a_1 + a_2 + a_3$$

$$S_1 = 000, S_2 = 001, S_3 = 010, S_4 = 011, S_5 = 100, S_6 = 101, S_7 = 110, S_8 = 111$$

# Mochila con solución *branch and bound*



- Para el objetivo suma 10, hay tres soluciones: 2+3+5, 1+4+5 y 1+2+3+4
- En vez de  $2^5 = 32$ , hay 24 ramas
- Aquí resulta muy obvio sumar y se puede hacer mentalmente
- Si son muchos los números, o bien estos tienen decimales, el problema sería más difícil
- Lo mismo si se usan muchos números enteros y muy grandes, como ocurre en los algoritmos criptográficos

# Operaciones en la solución de la mochila

- Dada la siguiente secuencia  $S = \{S_1, S_2, S_3, \dots, S_{m-2}, S_{m-1}, S_m\}$  de  $m$  números enteros positivos y un valor  $T$ , se pide encontrar un subconjunto  $S_S = \{S_a, S_b, \dots, S_j\}$ , que cumpla con ese objetivo  $T$ 
  - $T = \sum S_S = S_a + S_b + \dots + S_j$
- Si los elementos de la mochila son números grandes, no están ordenados y no siguen una distribución supercreciente (en que  $S_i$  es mayor que la suma de los anteriores  $S_j$ ) el problema es tipo NP
- Se trata de encontrar los vectores binarios  $V_i$  de forma que se cumpla la relación  $\sum (S_i * V_i) = T$

# Interés de las mochilas en la criptografía

- Aplicación en la criptografía asimétrica con dos claves, una de ellas pública y la otra privada, inversas entre sí dentro de un módulo
- Se puede usar el problema de la mochila, para permitir realizar una operación de cifra rápida y fácil y, por el contrario, su ataque resulte muy difícil, al ser un problema del tipo NP completo
- Si los elementos de esa mochila son números enteros, la solución al problema -en el caso de que exista solución- será única y fácil
- Esto se da cuando los números  $S = \{s_1, s_2, s_3, s_4, \dots, s_{n-1}, s_n\}$  forman una cadena supercreciente, es decir que el número siguiente de la lista es mayor que la suma de los todos  $s_j$  anteriores



# Cifrado con mochila de Merkle y Hellman

- En 1978 (pocos meses después que RSA) Ralph Merkle y Martin Hellman proponen un sistema de cifra de clave pública denominado mochila tramposa
- Cada usuario crea una mochila difícil (desordenada, clave pública) a partir de una mochila simple (supercreciente, clave privada)
- En el cifrado se usa la mochila difícil (pública) del receptor y en el descifrado se usa la mochila simple (privada) de ese receptor
- Para la cifra se usan los bits 1 del texto en claro para contar con el número que se encuentra en esa posición de la mochila difícil o pública del destino, y los bits ceros para no tenerlo en cuenta
- Se puede pasar fácilmente de la mochila simple a la difícil o viceversa usando una trampa, que solo posee el dueño de la clave

# Diseño mochila de Merkle y Hellman

1. Se selecciona una mochila supercreciente de  $m$  elementos  $S' = \{S'_1, S'_2, \dots, S'_m\}$
2. Se elige un entero  $\mu$  (módulo de trabajo) mayor que la suma de los elementos de la mochila  
$$\mu > \sum_{i=1}^m S'_i \quad \mu \geq 2 * S'_m$$
3. Se elige un entero  $\omega$  primo relativo con  $\mu$   
$$\omega^{-1} = \text{inv}(\omega, \mu)$$
4. Se multiplica  $\omega * S'$  mod  $\mu$ , obteniendo una mochila difícil  $S = \{S_1, S_2, \dots, S_m\}$   
$$S_i = \omega * S'_i \text{ mod } \mu$$
5. La clave pública será  $S = \{S_1, S_2, \dots, S_m\}$  y la clave privada será  $(\omega^{-1}, \mu)$
6. Cifrado:  $C = M * S$  (M en bits) --- Descifrado:  $M = \omega^{-1} * C \text{ mod } \mu$

# Ejemplo de cifrado con mochila de MH

- $S' = \{S'_1, S'_2, S'_3, S'_4, S'_5, S'_6, S'_7, S'_8\} = \{40, 75, 201, 488, 1.013, 2.001, 3.919, 8.106\}$
- Elegimos  $\mu \geq 2 * S'_m \geq 2 * 8.106 = 16.212$ , sea  $\mu = 16.250$
- Si elegimos  $\omega = 171$  [ $\text{mcd}(\omega, \mu) = 1$ ],  $\omega^{-1} = \text{inv}(\omega, \mu) = \text{inv}(171, 16.250) = 3.231$
- Calculamos nuestra clave pública:  $S_i = \omega * S'_i \bmod \mu$ 

$S_1 = 171 * 40 \bmod 16.250 = 6.840$	$S_2 = 171 * 75 \bmod 16.250 = 12.825$
$S_3 = 171 * 201 \bmod 16.250 = 1.871$	$S_4 = 171 * 488 \bmod 16.250 = 2.198$
$S_5 = 171 * 1.013 \bmod 16.250 = 10.723$	$S_6 = 171 * 2.001 \bmod 16.250 = 921$
$S_7 = 171 * 3.919 \bmod 16.250 = 3.899$	$S_8 = 171 * 8.106 \bmod 16.250 = 4.876$
- Clave pública  $S = \{S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8\}$ 
  - $S = \{6.840, 12.825, 1.871, 2.198, 10.723, 921, 3.899, 4.876\}$
- Cifrar el texto Hola = 01001000 01101111 01101100 01100001

Puede realizar estas  
operaciones con

**SAMCrypt**

# Cifrando con mochila pública de MH

- Nuestra clave pública
  - $S = \{6.840, 12.825, 1.871, 2.198, 10.723, 921, 3.899, 4.876\}$
- Nuestra clave privada
  - $\mu = 16.250, \omega^{-1} = 3.231, S' = S * \omega^{-1} \bmod \mu = \{40, 75, 201, 488, 1.013, 2.001, 3.919, 8.106\}$
- Alguien nos enviará cifrado **Hola** = 01001000 01101111 01101100 01100001
  - 01001000 Pasando por clave pública:  $12.825 + 10.723 = 23.548$
  - 01101111 Pasando por clave pública:  $12.825 + 1.871 + 10.723 + 921 + 3.899 + 4.876 = 35.115$
  - 01101100 Pasando por clave pública:  $12.825 + 1.871 + 10.723 + 921 = 26.340$
  - 01100001 Pasando por clave pública:  $12.825 + 1.871 + 4.876 = 19.572$
- $C = 23.548, 35.115, 26.340, 19.572$  (cuatro números)
  - Como los números de este ejemplo son muy pequeños y la mochila tiene muy pocos elementos, además de un tamaño 8 (byte), no sería tan difícil recuperar el mensaje usando la clave pública

# Descifrando con mochila privada de MH (1)

- Nuestra clave privada
  - $\mu = 16.250$
  - $\omega^{-1} = 3.231$
  - $S' = S * \omega^{-1} \bmod \mu = \{S'_1, S'_2, S'_3, S'_4, S'_5, S'_6, S'_7, S'_8\}$
  - $S' = \{40, 75, 201, 488, 1.013, 2.001, 3.919, 8.106\}$
- Criptograma
  - $C = 23.548, 35.115, 26.340, 19.572$
- Calculamos  $M_i = C_i * \omega^{-1} \bmod \mu$  y se recorre una sola vez la mochila simple de derecha a la izquierda. Si el elemento  $S'_i$  está en la suma, siempre lo estará
- Se descuenta ese número de  $M_i$  y se continúa con el proceso hasta llegar al valor 0
- La solución de  $M_i$  será única

# Descifrando con mochila privada de MH (2)

- $\mu = 16.250$ ,  $\omega^{-1} = 3.231$ ,  $S' \{40, 75, 201, 488, 1.013, 2.001, 3.919, 8.106\}$
- Criptograma:  $C = 23.548, 35.115, 26.340, 19.572$ 
  - $M_1 = C_1 * \omega^{-1} \bmod \mu = 23.548 * 3.231 \bmod 16.250 = 1.088$ 
    - $M_1 = 1.013 + 75 = 01001000 = H$
  - $M_2 = C_2 * \omega^{-1} \bmod \mu = 35.115 * 3.231 \bmod 16.250 = 15.315$ 
    - $M_2 = 8.106 + 3.919 + 2.001 + 1.103 + 201 + 75 = 01101111 = o$
  - $M_3 = C_3 * \omega^{-1} \bmod \mu = 26.340 * 3.231 \bmod 16.250 = 3.290$ 
    - $M_3 = 2.001 + 1.013 + 201 + 75 = 01101100 = l$
  - $M_4 = C_4 * \omega^{-1} \bmod \mu = 19.572 * 3.231 \bmod 16.250 = 8.382$ 
    - $M_4 = 8.106 + 201 + 75 = 01100001 = a$
- $M = \text{Hola}$  (se ha recuperado el texto en claro, con confidencialidad)

# Otras mochilas en la criptografía

- El cifrado con mochila de Merkle y Hellman fue roto por Adi Shamir y Richard Zippel en 1982
  - Mayor información sobre este ataque en la bibliografía: Libro Electrónico de Seguridad Informática y Criptografía
- Además del algoritmo de cifrado con mochila de Merkle y Hellman, hubo otros sistemas propuestos, entre ellos Graham-Shamir, Morii-Kasahara , Chor-Rivest y Goodman-McAuley
  - Mayor información en documento Universidad de California que se incluye en la bibliografía



# Conclusiones de la Lección 3.2

- El problema de la mochila es un ejemplo de problema matemático con tiempo de ejecución NP, es decir polinomial no determinista
- Para su resolución, pueden usarse técnicas de vuelta atrás (backtracking) y ramificación y poda (branch and bound), que significa recorrer varios caminos diferentes, donde solo alguno o algunos de ellos entregan la solución
- La mochila de Merkle y Hellman de 1978, es un sistema de cifra asimétrica propuesto unos meses después de RSA
- Permite cifrar un mensaje conociendo la clave pública del destinatario (una mochila difícil) permitiendo confidencialidad en el envío
- El destino descifra el criptograma con su mochila simple supercreciente, que es inversa de la difícil y que sólo conoce su dueño gracias a una trampa

# Lectura recomendada (1/2)

- Knapsack problema / Vuelta atrás, Wikipedia
  - [https://en.wikipedia.org/wiki/Knapsack\\_problem](https://en.wikipedia.org/wiki/Knapsack_problem)
  - [https://es.wikipedia.org/wiki/Vuelta\\_atr%C3%A1s](https://es.wikipedia.org/wiki/Vuelta_atr%C3%A1s)
- Teoría de Algoritmos, Curso de Análisis y Diseño de Algoritmos (Backtracking, vuelta atrás), Fernando Berzal Galiano, Univ. de Granada
  - <http://elvex.ugr.es/decsai/algorithms/>
- Capítulo 13 Cifrado asimétrico con mochilas, Libro Electrónico de Seguridad Informática y Criptografía versión 4.1, Jorge Ramió, marzo de 2006
  - [https://www.criptored.es/guiateoria/gt\\_m001a.htm](https://www.criptored.es/guiateoria/gt_m001a.htm)

# Lectura recomendada (2/2)

- RapidTables
  - <https://www.rapidtables.com/convert/number/ascii-to-binary.html>
- SAMCript: Software de Aritmética Modular para Criptografía, María Nieto Díaz, dirección Jorge Ramió, 2018
  - [https://www.criptored.es/software/sw\\_m001t.htm](https://www.criptored.es/software/sw_m001t.htm)
- Merkle-Hellman Knapsack Encryption
  - <https://asecuritysite.com/encryption/knapcode>
- Knapsack Cryptosystems: The Past and the Future, Dept. of Information and Computer Science, University of California, Ming Kin Lai, 2001
  - <http://ljk.imag.fr/membres/Jean-Guillaume.Dumas/Enseignements/ProjetsCrypto/sac-a-dos-LLL/knapsack.html>

# Class4crypt c4c3.3

## Módulo 3. Complejidad algorítmica en la criptografía

### Lección 3.3. El problema del logaritmo discreto

3.3.1. Enunciado del problema del logaritmo discreto

3.3.2. Ejemplos y resolución de operaciones de exponenciación modular

3.3.3. Ejemplos y resolución de operaciones de logaritmo discreto

3.3.4. Usos en la criptografía: intercambio de clave de Diffie y Hellman, cifra y firma de Elgamal, algoritmo de firma digital DSA

Class4crypt c4c3.3 El problema del logaritmo discreto  
<https://www.youtube.com/watch?v=Fg2Y5utA-uc>

# Potencia modular y logaritmo discreto

- Dados los enteros  $\alpha$ ,  $x$  y un cuerpo  $p$ , resolver la operación
  - $\beta = \alpha^x \bmod p$  (Potencia o exponenciación modular)
  - Es un problema polinomial de baja complejidad algorítmica, incluso con números  $\alpha$ ,  $x$  y  $p$  grandes
- No obstante, conocidos los números  $\beta$ ,  $\alpha$  y el primo  $p$ , encontrar ahora un entero  $x$  de forma que
  - $x = \log_{\alpha} \beta \bmod p$  (Problema del Logaritmo Discreto PLD)
  - Se convierte en un problema polinomial no determinista NP, que es computacionalmente intratable si  $p$  es muy grande y  $x$  no es un número demasiado pequeño de forma que pueda encontrarse ese valor por fuerza bruta con poco esfuerzo

# Exponenciación modular

- Algunos cálculos con números muy pequeños pueden hacerse mentalmente
  - $3^4 \bmod 50 = 81 \bmod 50 = 31$ ;  $5^3 \bmod 120 = 125 \bmod 120 = 5$
- Con números más grandes, usando calculadoras
  - Ejercicios prácticos con Calc Windows
  - 4 dígitos<sup>4 dígitos</sup> mod 6 dígitos:  $9.715^{9.041} \bmod 587.653 = 429.145$
  - ¿Podemos resolver con esta calculadora  $12.345^{12.345}$ ? (son 167.789 bits)
- Y con números muy grandes, usando el software adecuado
  - Ejercicios prácticos con SAMCrypt (no modular)
  - $12.345^{12.345} = 28.678.652.250.036 \dots 291.259.765.625$
  - Un número de 50.510 dígitos (167.789 bits... 47 páginas en Word)
  - ¿Qué sucedería si aumentamos la base? ¿Y si aumentamos el exponente?



# Exponenciación con SAMCrypt



Puedes usar este software

- Exponenciación no modular

- $12.345^{12.345} = 50.510$  dígitos (2 segundos)
- Doblando cantidad de dígitos en la base de 5 a 10:
- $1.234.567.890^{12.345} = 112.235$  dígitos (6 segundos) ←
- Aumentando solamente un dígito el exponente:
- $12.345^{123.456} = 505.120$  dígitos (118 segundos)

¿Por qué pasa esto?

- Exponenciación modular

- 1 mil dígitos <sup>1 mil dígitos</sup> mod 1 mil dígitos  $t < 1$  segundo
- 2 mil dígitos <sup>2 mil dígitos</sup> mod 2 mil dígitos  $t = 1$  segundo
- 4 mil dígitos <sup>4 mil dígitos</sup> mod 4 mil dígitos  $t = 2,5$  segundos ←
- 8 mil dígitos <sup>8 mil dígitos</sup> mod 8 mil dígitos  $t = 20$  segundos
- Aumentamos muchísimo la base y el exponente: la complejidad es baja

# Logaritmo discreto con fuerza bruta

- Encontrar  $x = \log_{\alpha} \beta \bmod p$ , en donde  $\alpha$ ,  $\beta$  y  $p$  son conocidos
- Si el valor de  $x$  fuese muy pequeño, se podría atacar por fuerza bruta buscando todos los valores de  $x$  que hagan  $\alpha^x \bmod p = \beta$
- Por ejemplo si  $p = 1.999$ ,  $\alpha = 33$  y se sabe que  $\beta = 1.343$ 
  - $33^2 \bmod 1.999 = 1.089$ ;  $33^3 \bmod 1.999 = 1.954$ ;  $33^4 \bmod 1.999 = 514$ ;
  - continuando...  $33^{46} \bmod 1.999 = 283$ ;  $33^{47} \bmod 1.999 = 1.343$  ( $x = 47$ )
  - Como  $\alpha = 33$  es una raíz primitiva de  $p = 1.999$ , la solución es única
  - Efectivamente,  $\log_{33} 1.343 \bmod 1.999 = 47$  (p.ej. con SAMCrypt)
- Si el exponente  $x$  es un número grande, este ataque no es viable



# Logaritmo discreto con $\alpha$ raíz primitiva

- Como en el cuerpo  $p = 13$  el resto 2 es una raíz primitiva

• $2^0 \bmod 13 = 1$	$2^1 \bmod 13 = 2$	$2^2 \bmod 13 = 4$
• $2^3 \bmod 13 = 8$	$2^4 \bmod 13 = 3$	$2^5 \bmod 13 = 6$
• $2^6 \bmod 13 = 12$	$2^7 \bmod 13 = 11$	$2^8 \bmod 13 = 9$
• $2^9 \bmod 13 = 5$	$2^{10} \bmod 13 = 10$	$2^{11} \bmod 13 = 7$

- Entonces

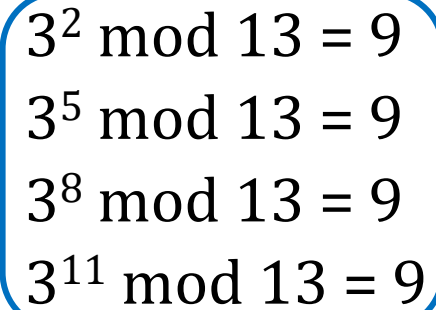
• $\log_2 1 \bmod 13 = 0$	$\log_2 2 \bmod 13 = 1$	$\log_2 3 \bmod 13 = 4$
• $\log_2 4 \bmod 13 = 2$	$\log_2 5 \bmod 13 = 9$	$\log_2 6 \bmod 13 = 5$
• $\log_2 7 \bmod 13 = 11$	$\log_2 8 \bmod 13 = 3$	$\log_2 9 \bmod 13 = 8$
• $\log_2 10 \bmod 13 = 10$	$\log_2 11 \bmod 13 = 7$	$\log_2 12 \bmod 13 = 6$

# Logaritmo discreto con $\alpha$ no raíz primitiva

- Como en el cuerpo  $p = 13$  el resto 3 no es una raíz primitiva

• $3^0 \bmod 13 = 1$	$3^1 \bmod 13 = 3$
• $3^3 \bmod 13 = 1$	$3^4 \bmod 13 = 3$
• $3^6 \bmod 13 = 1$	$3^7 \bmod 13 = 3$
• $3^9 \bmod 13 = 1$	$3^{10} \bmod 13 = 3$

$3^2 \bmod 13 = 9$   
 $3^5 \bmod 13 = 9$   
 $3^8 \bmod 13 = 9$   
 $3^{11} \bmod 13 = 9$

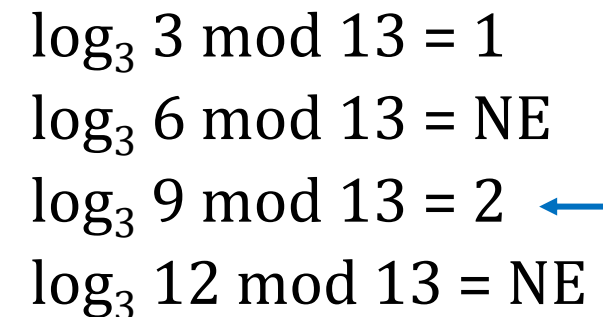


Solución:  $\text{exp} = 2 + 3k$

- Entonces

• $\log_3 1 \bmod 13 = 0$	$\log_3 2 \bmod 13 = \text{NE}$
• $\log_3 4 \bmod 13 = \text{NE}$	$\log_3 5 \bmod 13 = \text{NE}$
• $\log_3 7 \bmod 13 = \text{NE}$	$\log_3 8 \bmod 13 = \text{NE}$
• $\log_3 10 \bmod 13 = \text{NE}$	$\log_3 11 \bmod 13 = \text{NE}$

$\log_3 3 \bmod 13 = 1$
$\log_3 6 \bmod 13 = \text{NE}$
$\log_3 9 \bmod 13 = 2$
$\log_3 12 \bmod 13 = \text{NE}$



# Logaritmo discreto con SAMCript

- Sea  $p$  un primo de 40 a 50 bits,  $\alpha$  una raíz primitiva,  $x = 125$
- 40 bits  $x = \log_2 455.724.407.932 \bmod 729.731.628.413$  (1 seg)
- 42 bits  $x = \log_3 1.568.712.172.137 \bmod 2.266.333.380.079$  (2 seg)
- 44 bits  $x = \log_3 1.400.611.805.375 \bmod 17.568.365.469.881$  (3 seg)
- 46 bits  $x = \log_3 31.408.415.753.007 \bmod 49.604.255.233.937$  (22 seg)
- 48 bits  $x = \log_2 68.719.082.765.791 \bmod 268.885.439.840.939$  (36 seg)
- 50 bits  $x = \log_5 312.028.532.499.463 \bmod 663.206.602.279.213$  (43 seg)
- 50 bits  $x = \log_7 161.390.513.288.025 \bmod 967.081.312.992.721$  (9 seg) ←
- 50 bits  $x = \log_{14} 801.969.266.079.988 \bmod 967.081.312.992.721$  (54 seg)
- 50 bits  $x = \log_{37} 739.873.271.748.987 \bmod 967.081.312.992.721$  (168 seg)
- 50 bits  $x = \log_{38} 434.320.563.871.606 \bmod 967.081.312.992.721$  (68 seg)
- Observa lo que sucede con 50 bits y cambiando la base  $\alpha$  para el mismo primo

# Ejercicios del logaritmo discreto en CLCrypt

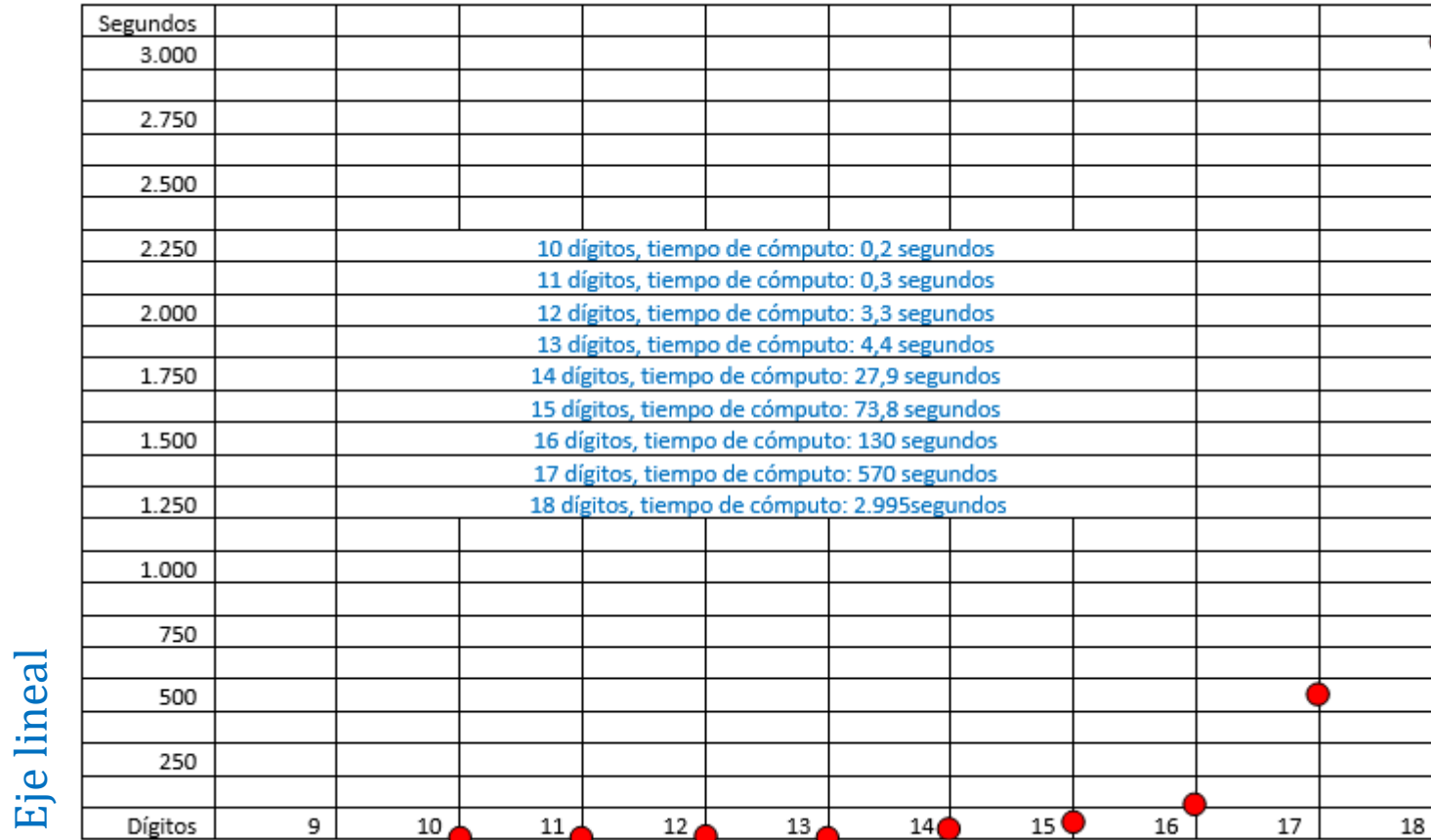


Figura 30. PLD: curva del tiempo de cómputo versus tamaño de  $n$  desde 10 a 110 bits.

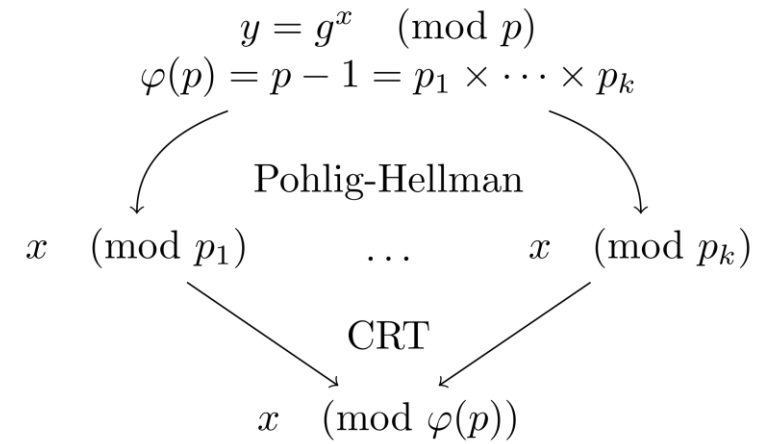
- Comportamiento típico del tiempo de ejecución versus el tamaño del primo en el PLD (realizado con otra máquina)
- En esa curva pueden existir discontinuidades
- Se recomienda leer el apartado del logaritmo discreto en la práctica CLCrypt 09 Matemáticas discretas en criptografía con SAMCrypt, que encontrarás en la bibliografía

# PLD con software web Alpertron

- Primos de 25 a 39 dígitos,  $\alpha$  la raíz más pequeña y  $x = 0x7D1$  ( $2.001_{10}$ )
- $25 \times = \text{Log}_{13} 57D92D85C2CD760A90C2E \bmod 5D50F43FBD07357BC53D1$  (0 segundos)
- $26 \times = \text{Log}_6 2D8D65AACFCA213AADF6BA \bmod 2DEDDC642E83B2BE8A61C5$  (0 segundos)
- $27 \times = \text{Log}_2 21A9E9D35465971F59D0F51 \bmod 2D5FBA7E894AEC3A2DDA045$  (> 10 minutos)
- $28 \times = \text{Log}_6 16038264D13A580D45ECD1F9 \bmod 1CB4B7DDDDAB6FD5D1927259$  (0 segundos)
- $29 \times = \text{Log}_2 7BBE39E5D82FBF348112B118 \bmod C8CA0B64D0B449EE2FEB6A0D$  (> 10 minutos)
- $30 \times = \text{Log}_D 204501199AA9295019893E8B0 \bmod 24D24C653651D39D24D7C89EB$  (12 segundos)
- $31 \times = \text{Log}_A 6207D138FCC8BCD6608A1CB75F \bmod 6B8976BDB8832EA914F6F06A7B$  (4 segundos) ←
- $32 \times = \text{Log}_5 25C4D0C2A58B716030A1F6BCE40 \bmod 375CE61F621C94A761AB1760C31$  (0 segundos)
- $33 \times = \text{Log}_B DA66B4C88F2CC465F670CEE9165 \bmod F0787C1FF0F0F47C1F00787C1E1$  (0 segundos)
- $34 \times = \text{Log}_5 6A2344989CCACCD729861C95BDB8 \bmod F5A6B315159AFDD33C14321357C9$  (> 10 minutos)
- $35 \times = \text{Log}_3 2EDF9FAF9601C9F0DA2D2137341D5 \bmod C099EC8C775710571F08D04D8F149$  (> 10 minutos)
- $36 \times = \text{Log}_5 45C62D2BE7A0DCCA88EBCAC668DBEA \bmod 5EEC038059ACAEB9A2E967723675DF$  (85 segundos)
- $37 \times = \text{Log}_2 177A0FC2D7EE4DBE4D6DACC6B5E58AA \bmod 49FCA8F44D33D81723F9FEE569FC1E5$  (> 10 minutos)
- $38 \times = \text{Log}_2 7FEB160425F215CC2B5CD84FAA80FF \bmod 14BFF2802A7EC7C3DB9927A90EF8E165$  (> 10 minutos)
- $39 \times = \text{Log}_2 ABC8168D2DB73F228B12A92B638A508 \bmod AB6601FAAE93909308F2D729565B64DB$  (1 segundo)
- Observa las discontinuidades en el tiempo de cómputo

# Algoritmos para el PLD y su complejidad

- No se conoce un algoritmo eficiente que solucione el PLD
- Algoritmos para afrontar el PLD
  - Fuerza bruta
  - Paso enano - paso gigante  $O(\sqrt{p^e})$
  - Criba en el campo
  - Cálculo de índices
  - Pohlig-Hellman  $O(e\sqrt{p})$
  - Pollard rho ( $\rho$ )
  - Y otros más
    - Ver documento Advanced Cryptology de la Universidad de Oxford que se incluye en la bibliografía
  - Todos con un crecimiento exponencial en función del tamaño del cuerpo  $p$



Pasos del algoritmo Pohlig-Hellman (David Wong)

# Usos del PLD en la criptografía

- Intercambio de claves de Diffie y Hellman
  - El usuario A calcula  $\alpha^a \bmod p = \beta_A$  y el usuario B calcula  $\alpha^b \bmod p = \beta_B$ . Son públicos  $p$ ,  $\alpha$ ,  $\beta_A$  y  $\beta_B$ , pero  $a$  y  $b$  son secretos. Para romper el algoritmo, habrá que resolver el PLD para  $a$  y para  $b$  con números grandes
- Cifra y firma de Elgamal
  - La clave pública  $\beta$  se crea utilizando un valor secreto o privado  $x$ , de forma que  $\beta = \alpha^x \bmod p$ . Para romper el algoritmo, habrá que resolver el PLD para  $x$  con números grandes
- Firma DSA Digital Signature Algorithm
  - Es una variante de la firma de Elgamal con dos primos,  $p$  y  $q$ . Para romper el algoritmo, habrá que resolver el PLD con números grandes

# Conclusiones de la lección 3.3

- La exponenciación modular  $\beta = \alpha^x \bmod p$  es una operación con complejidad del tipo P (polinomial), rápida y sencilla
- El problema del logaritmo discreto  $x = \log_{\alpha} \beta \bmod p$  es la operación inversa a la anterior, con complejidad NP (polinomial no determinista). Si los números son grandes, se convierte en un problema de muy difícil solución
- El comportamiento típico del tiempo de cómputo versus el tamaño del módulo  $p$  en bits del problema del logaritmo discreto es de tipo exponencial
- Si bien el crecimiento no siempre es continuo, ya que en función del primo  $p$  e incluso de la base  $\alpha$ , el tiempo de cómputo puede ser muy dispar
- El problema del logaritmo discreto se usa en criptografía moderna de clave pública (DH, Elgamal, DSA) para aportar fortaleza a los algoritmos



# Lectura recomendada (1/2)

- Discrete logarithm, Wikipedia
  - [https://en.wikipedia.org/wiki/Discrete\\_logarithm](https://en.wikipedia.org/wiki/Discrete_logarithm)
- On the complexity of the discrete logarithm and Diffie–Hellman problems, Blake & Garefalakis, Elsevier, 2004
  - <https://www.sciencedirect.com/science/article/pii/S0885064X04000056>
- Mathematics of Public Key Cryptography, Part III: Exponentiation, Factoring and Discrete Logarithms, Steven Galbraith, Cambridge University Press, 2012
  - <https://www.math.auckland.ac.nz/~sgal018/crypto-book/crypto-book.html>
- Advanced Cryptology (2018-2019), DLP and factoring algorithms, Christophe Petit, University of Oxford
  - [https://courses-archive.maths.ox.ac.uk/node/download\\_material/12660](https://courses-archive.maths.ox.ac.uk/node/download_material/12660)

# Lectura recomendada (2/2)

- An Improved Algorithm for Computing Logarithms over  $GF(p)$  and its Cryptographic Significance, S. Pohlig, M. Hellman, IEEE Trans on Information Theory (24): 106–110, 1978
  - <http://www-ee.stanford.edu/~hellman/publications/28.pdf>
- SAMCrypt: Software de Aritmética Modular para Criptografía, María Nieto Díaz,, 2018
  - [https://www.criptored.es/software/sw\\_m001t.htm](https://www.criptored.es/software/sw_m001t.htm)
- Big primes Generator
  - <https://bigprimes.org/>
- Matemáticas discretas en criptografía con SAMCrypt, CLCrypt, Jorge Ramió, 2019
  - [https://www.criptored.es/descarga/CLCrypt\\_entrega\\_09\\_Matematicas\\_Discretas\\_Criptografia\\_SAMCrypt.pdf](https://www.criptored.es/descarga/CLCrypt_entrega_09_Matematicas_Discretas_Criptografia_SAMCrypt.pdf)
- Calculadora de logaritmos discretos, Dario Alpern, 2020
  - <https://www.alpertron.com.ar/LOGDI.HTM>

# Class4crypt c4c3.4

## Módulo 3. Complejidad algorítmica en la criptografía

### Lección 3.4. El problema de la factorización entera

3.4.1. Enunciado del problema de la factorización entera PFE

3.4.2. Operaciones de multiplicación y su inversa la factorización entera

3.4.3. Solución del problema de la factorización entera con msieve153

3.4.4. Solución del problema de la factorización entera con web Alpertron

3.4.5. Algoritmos del PFE y complejidad asociada

3.4.6. Uso en criptografía moderna de clave pública: algoritmo RSA

Class4crypt c4c3.4 El problema de la factorización entera

[https://www.youtube.com/watch?v=of\\_5ioayJo0](https://www.youtube.com/watch?v=of_5ioayJo0)

# Multiplicación y factorización de enteros

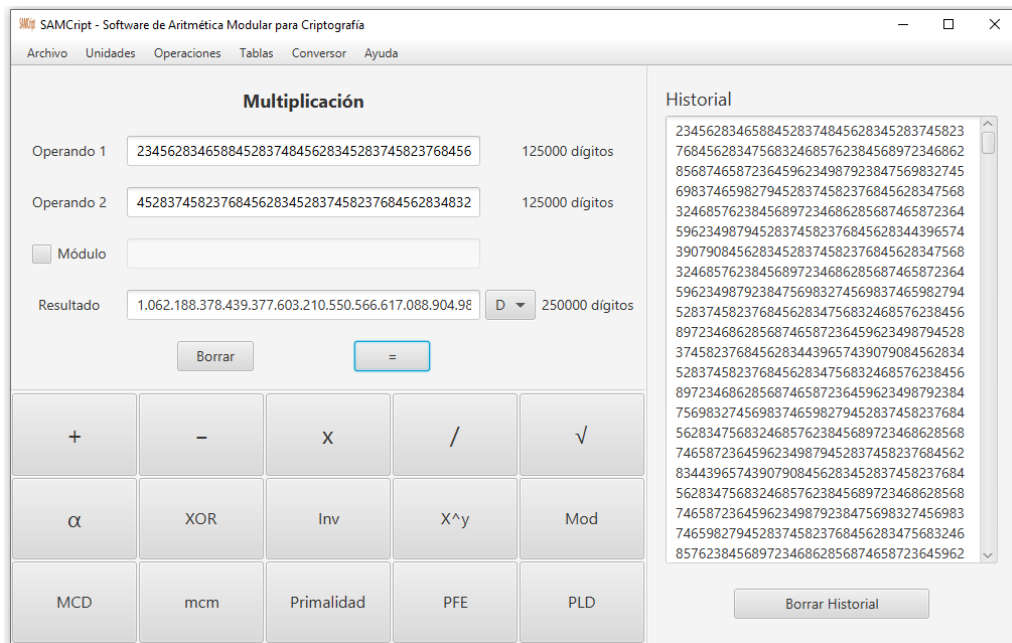
- Multiplicación

- Dados los enteros  $\{p, q\}$ , encontrar  $n = p \cdot q$ , es una operación con una complejidad de tipo P polinómica, sencilla y rápida. Si aumentamos el tamaño de los datos de entrada, el tiempo de cómputo aumenta de una forma proporcional o casi lineal

- Factorización

- No obstante, la operación inversa, encontrar esos primos  $p$  y  $q$  si sólo se conoce el producto  $n$ , es una operación con una complejidad de tipo NP, polinomial no determinista. Si los números son grandes, este cómputo se vuelve muy difícil o intratable. Si aumentamos el tamaño de los datos de entrada, el tiempo de cómputo aumenta de forma exponencial
- Esto se conoce como el problema de la factorización entera PFE

# Producto números grandes con SAMCrypt



- Multiplicación de dos números de 10.000 dígitos
  - $N_1 * N_2$  (t = 0 segundos)
- Multiplicación de dos números de 25.000 dígitos
  - $N_3 * N_4$  (t = 1 segundo)
- Multiplicación de dos números de 50.000 dígitos
  - $N_5 * N_6$  (t = 6 segundos)
- Multiplicación de dos números de 75.000 dígitos
  - $N_7 * N_8$  (t = 14 segundos)
- Multiplicación de dos números de 100.000 dígitos
  - $N_9 * N_{10}$  (t = 25 segundos)
- Multiplicación de dos números de 125.000 dígitos
  - $N_{11} * N_{12}$  (t = 40 segundos)
- Sin tener en cuenta otros aspectos, el tiempo de cómputo va creciendo proporcional a la entrada

# El problema de la factorización entera

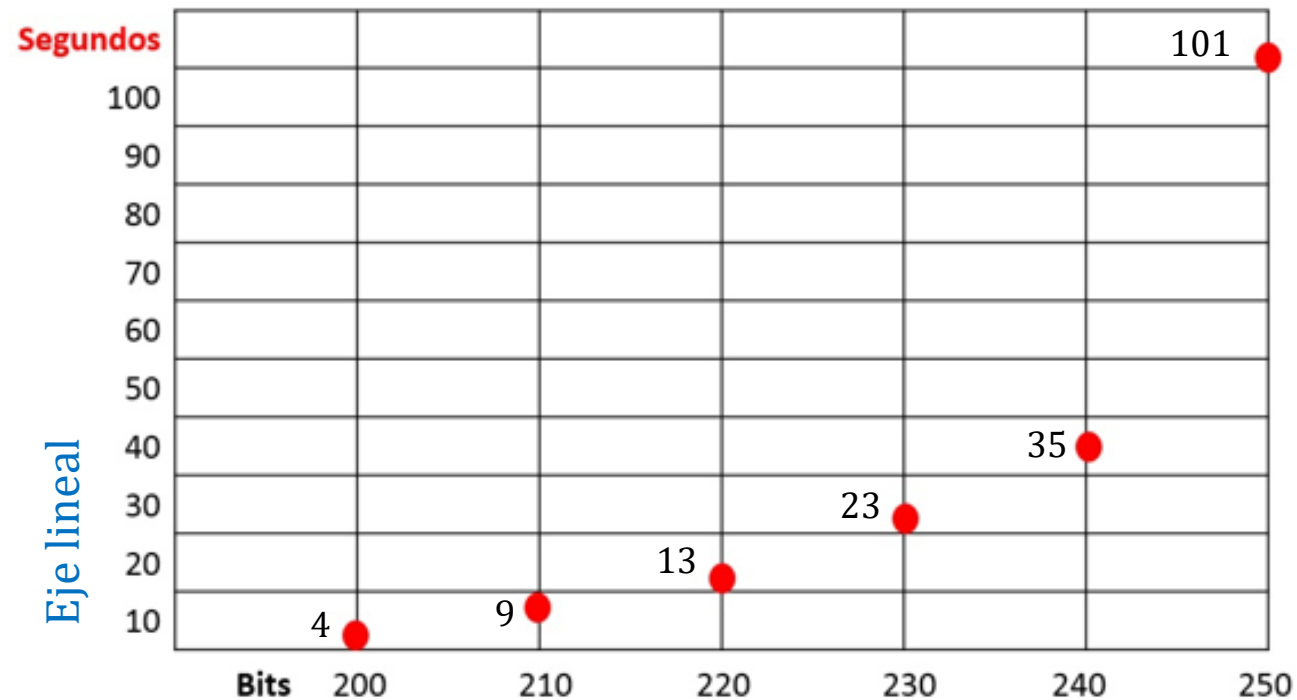


- Conocido el valor  $n$ , resultado del producto de dos primos  $p$  y  $q$  ambos de dimensiones similares, encontrar dichos primos es un problema polinomial no determinista NP, cuya solución es difícil cuando los primos son grandes
- La opción más elemental pero menos eficiente es la criba de Eratóstenes
- Si  $n = 24.146.449 = p * q$ , dividimos  $n$  entre 2, 3, 5, 7, 11, 13, ... hasta encontrar un entero como resultado de la división. Si  $p$  y  $q$  tienen un tamaño similar, como  $10^3 * 10^3 < 24.146.449 < 10^4 * 10^4$ , comenzamos la criba con primos de 4 dígitos, es decir, 1.009, 1.013, 1.019, 1.021, 1.031, 1.033, 1.039,... 3.433, **3.449**
- Como  $24.146.449 / \text{3.449} = \text{7.001}$  se concluye que  $p = 3.449$  y  $q = 7.001$
- Para primos de 1.024 bits efectivos (primer bit de los 1.024 en 1), esta opción es completamente inviable (ver cantidad de primos clase Class4crypt c4c2.1)

# Ejemplos de PFE con msieve153 (1/2)

C:\msieve153>msieve153 NUM -v (200 a 250 bits)

Muy importante: siempre borrar antes msieve.dat



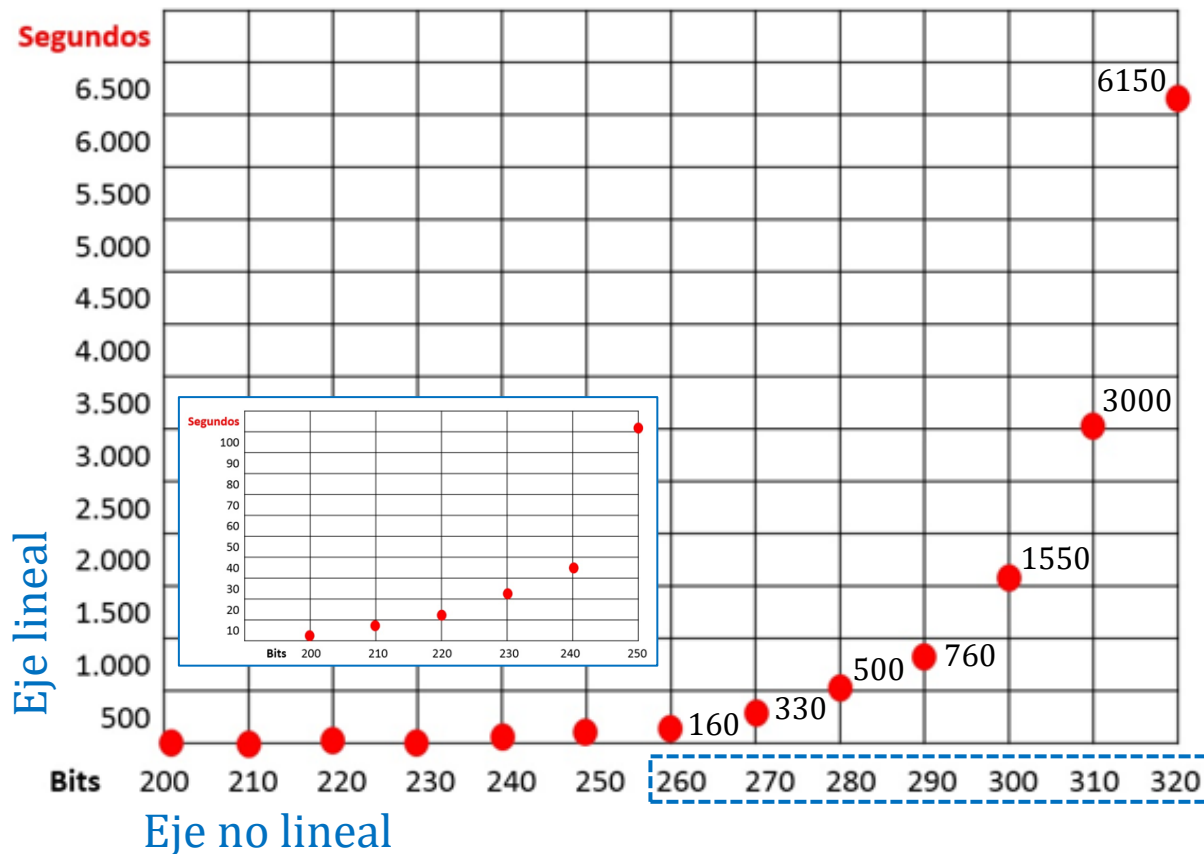
Eje no lineal

- Números  $n = p \cdot q$  de 200 a 250 bits
- NUM1 (200 bits,  $t = 4$  segundos)  
138154753916689783439712462474987283137080070  
3583102292448113
- NUM2 (210 bits,  $t = 9$  segundos)  
962845424426166508297533506045106594608403192  
127563776398625669
- NUM3 (220 bits,  $t = 13$  segundos)  
136810313000126181999112184929757638718719055  
5663030283450066543741
- NUM4 (230 bits,  $t = 23$  segundos)  
172105694454941092774029091981721726551079131  
6187439557664508355475703
- NUM5 (240 bits,  $t = 35$  segundos)  
114848743725669461628211568423976268267460017  
7631488687477584218631950269
- NUM6 (250 bits,  $t = 101$  segundos)  
100230136320705620742686359720346068189408021  
4379357460701055801194233460517

- Es interesante observar el crecimiento exponencial del tiempo por incrementos de 10 bits

# Ejemplos de PFE con msieve153 (2/2)

C:\msieve153>msieve153 NUM -v (260 a 320 bits)



- Números  $n = p \cdot q$  de 260 a 320 bits

- NUM7 (260 bits,  $t = 160$  segundos)

1760548793328664439112308465063709622270476544843296  
856540283721496322126359601

- NUM 8 (270 bits,  $t = 330$  segundos)

1120801329602221541899166082891190759317785505575040  
837210596672761060624819044011

- NUM 9 (280 bits,  $t = 500$  segundos)

1139012767901265952749262105553616610855562743863413  
746004006295729962093177017542461

- NUM 10 (290 bits,  $t = 760$  segundos)

1244386109817245735776671685906075703952171550228467  
795674293662561535962280660820543593

- NUM 11 (300 bits,  $t = 1.550$  segundos)

1347469821425774466892167818459701724451037556526230  
306195382771284234164361285823217400237

- NUM 12 (310 bits,  $t = 3.000$  segundos)

1491036249055969894744653659316386192168632359457482  
452195380700911044566526310052038976638043

- NUM 13 (320 bits,  $t = 6.150$  segundos)

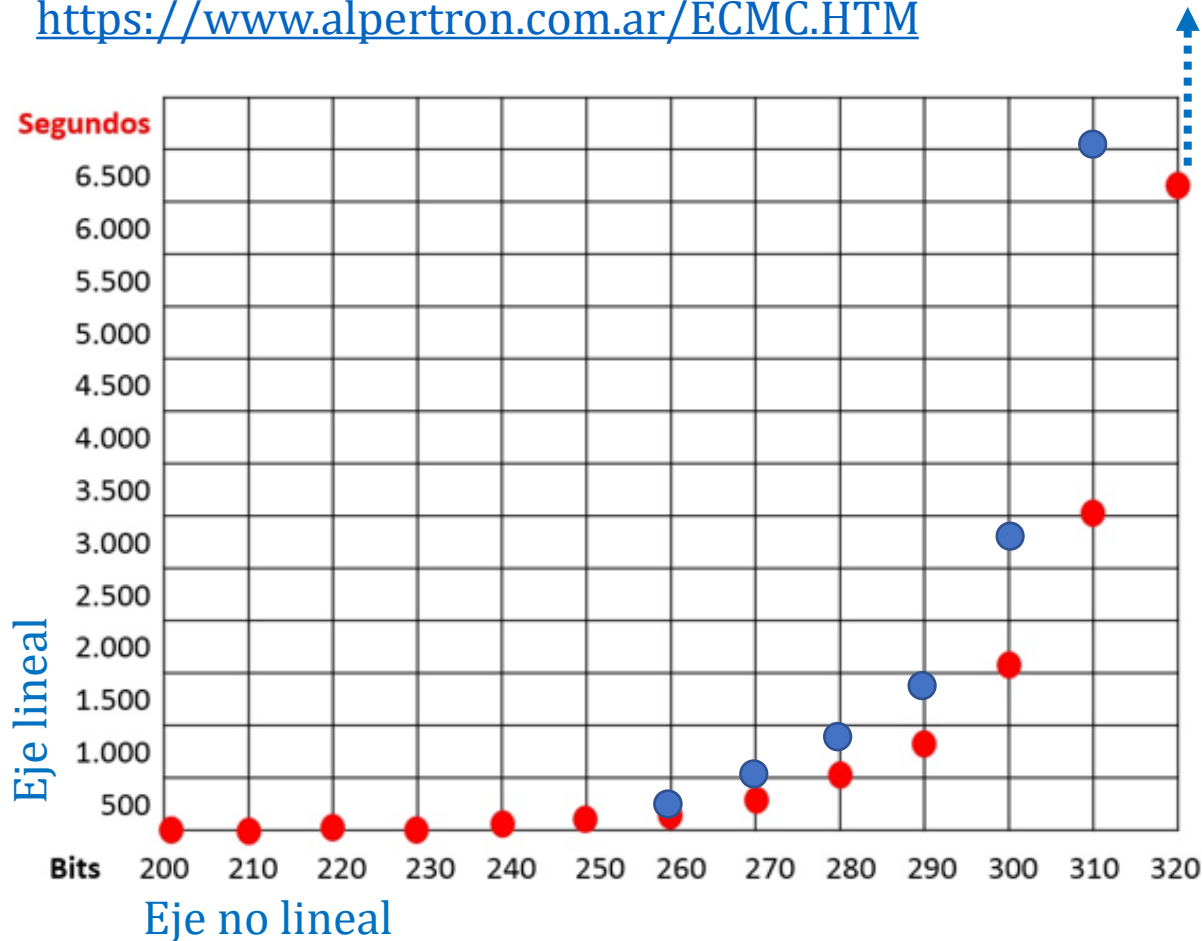
1356577227826927499322091465179931499485970498445538  
126773867510453302112259496008465712760745423

- Se mantiene la curva de tipo exponencial



# msieve153 • versus web Alpertron •

<https://www.alpertron.com.ar/ECMC.HTM>



- Se repiten los mismos últimos siete números que con msieve153, pero usando ahora Alpertron

- NUM7 (260 bits, t = 270 segundos)

1760548793328664439112308465063709622270476544843296  
856540283721496322126359601

- NUM 8 (270 bits, t = 560 segundos)

1120801329602221541899166082891190759317785505575040  
837210596672761060624819044011

- NUM 9 (280 bits, t = 810 segundos)

1139012767901265952749262105553616610855562743863413  
746004006295729962093177017542461

- NUM 10 (290 bits, t = 1.760 segundos)

1244386109817245735776671685906075703952171550228467  
795674293662561535962280660820543593

- NUM 11 (300 bits, t = 3.350 segundos)

1347469821425774466892167818459701724451037556526230  
306195382771284234164361285823217400237

- NUM 12 (310 bits, t = 6.530 segundos)

1491036249055969894744653659316386192168632359457482  
452195380700911044566526310052038976638043

- NUM 13 (320 bits: t = más de un día...)

1356577227826927499322091465179931499485970498445538  
126773867510453302112259496008465712760745423

# Factorizando 310 bits con Alpertron (1/2)

## Calculadora de factorización de números enteros

[Alpertron](#) > [Programas](#) > Calculadora de factorización de números enteros

Valor

1491036249055969894744653659316386192168632359457482452195380700911044566526310052038976638043

Una expresión numérica o ciclo por línea. Ejemplo: x=3;x=n(x);c<=100;x-1

Solo evaluar

Factorizar

Ayuda

Config

Asistente

Aprieta el botón **Ayuda** para obtener ayuda para esta aplicación. Apriétalo de nuevo para retornar a la factorización. Los usuarios con teclado pueden presionar CTRL+ENTER para comenzar la factorización. Esta es la versión WebAssembly.

- 1491 036249 055969 894744 653659 316386 192168 632359 457482 452195 380700 911044 566526 310052 038976 638043 (94 dígitos) = 18435 881048 548132 379693 104172 201626 001867 800591 (47 dígitos) × 80876 864258 862876 271055 824565 588577 446118 793973 (47 dígitos)

Cantidad de divisores: 4

Suma de divisores: 1491 036249 055969 894744 653659 316386 192168 632359 556795 197502 791709 561793 495264 100255 486963 232608 (94 dígitos)

Phi de Euler: 1491 036249 055969 894744 653659 316386 192168 632359 358169 706887 969692 260295 637788 519848 590990 043480 (94 dígitos)

Möbius: 1

$n = a^2 + b^2 + c^2$

a = 38095 734894 318473 243523 565698 415480 328313 461349 (47 dígitos)

b = 6304 857803 136860 516227 679047 291233 261261 687821 (46 dígitos)

c = 101

Tiempo transcurrido: 0d 1h 48m 51.0s

# Factorizando 310 bits con Alpertron (2/2)

Tiempo transcurrido: 0d 1h 48m 51.0s

Multiplicaciones modulares:

- ECM: 130981036
- Verificación de números primos probables: 1535
- SIQS: 347519
- Suma de cuadrados: 5964

SIQS:

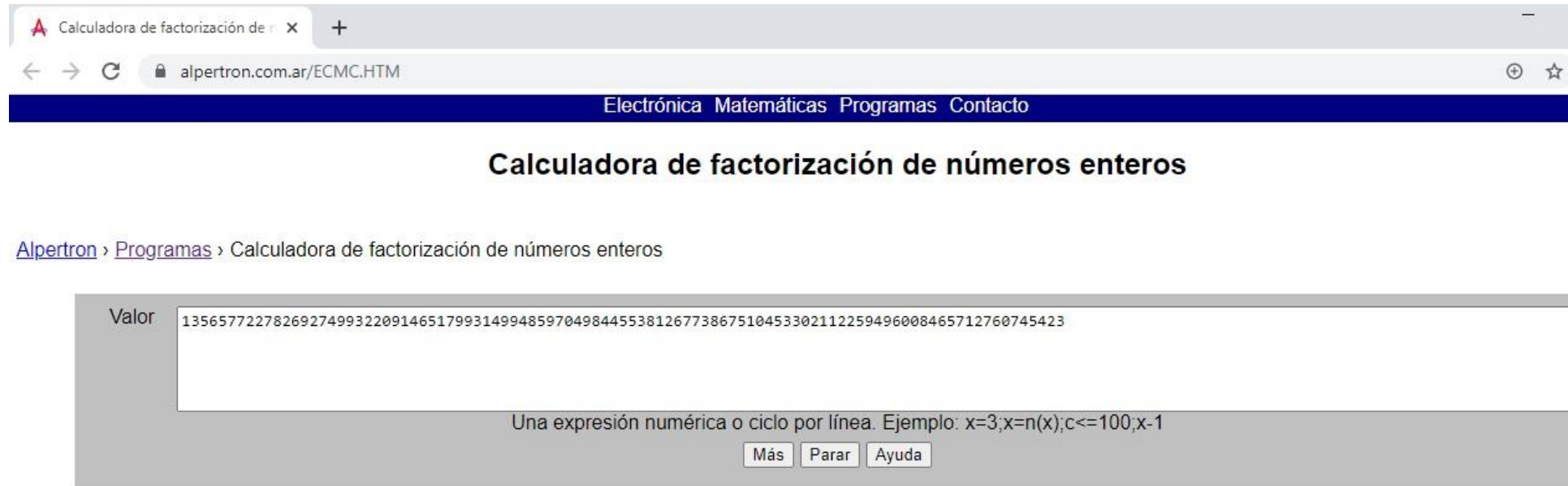
- 7046450 polinomios utilizados
- 1131274 conjuntos de divisiones de prueba
- 40178 congruencias completas (1 de cada 21531150 valores)
- 449540 congruencias parciales (1 de cada 1924363 valores)
- 42270 congruencias parciales útiles
- Tamaño de la matriz binaria:  $76214 \times 74333$

Tiempos:

- Test de primo probable de 3 números: 0d 0h 0m 0.0s
- Factorización 1 número mediante ECM 0d 0h 0m 28.4s
- Factorización 1 número mediante SIQS 0d 1h 48m 22.6s

Hecho por Darío Alpern. Actualizado el 1 de noviembre de 2020.

# Factorizando 320 bits con Alpertron



The screenshot shows a web browser window with the address bar displaying "alpertron.com.ar/ECMC.HTM". The page title is "Calculadora de factorización de números enteros". Below the title, there is a navigation bar with links: "Electrónica", "Matemáticas", "Programas", and "Contacto". The main heading is "Calculadora de factorización de números enteros". Below this, there is a breadcrumb trail: "Alpertron > Programas > Calculadora de factorización de números enteros". The input field is labeled "Valor" and contains a long number: "1356577227826927499322091465179931499485970498445538126773867510453302112259496008465712760745423". Below the input field, there is a text box with the instruction: "Una expresión numérica o ciclo por línea. Ejemplo: x=3;x=n(x);c<=100;x-1". Below the text box, there are three buttons: "Más", "Parar", and "Ayuda".

Aprieta el botón **Ayuda** para obtener ayuda para esta aplicación. Apriétalo de nuevo para retornar a la factorización. Los usuarios con teclado pueden presionar CTRL+ENTER | comenzar la factorización. Esta es la versión WebAssembly.

Factorizando 1 356577 227826 927499 322091 465179 931499 485970 498445 538126 773867 510453 302112 259496 008465 712760 745423 (97 dígitos)

Curva 3416 usando límites B1=11000000 y B2=1100000000

Transcurrió 1d 2h 6m 50s Paso 1: 52%



Se detiene el proceso transcurrido más de un día

# Algoritmos para el PFE y su complejidad

- No se conoce un algoritmo no cuántico (Shor) eficiente que solucione el PFE
- Los algoritmos de factorización pueden dividirse en dos grupos, los denominados de propósito general y los de propósito específico
- Entre los más conocidos se encuentran:
  1. Método de la criba de Eratóstenes
  2. Método de Fermat
  3. Método de Euler
  4. Método de Dixon
  5. Método de Williams  $p+1$
  6. Método de Pollar  $\rho$
  7. Método de Pollar  $p-1$
  8. Método de las fracciones continuas
  9. Método de las curvas elípticas
  10. Método de la criba numérica GNFS

General Number Field Sieve es el mejor algoritmo conocido a la fecha y tiene asociada una complejidad representada en esta expresión para un número  $n$

$$\exp\left(\left(\sqrt[3]{\frac{64}{9}} + o(1)\right) (\ln n)^{\frac{1}{3}} (\ln \ln n)^{\frac{2}{3}}\right)$$

# Usos del PFE en la criptografía

- Algoritmo RSA
  - El usuario A busca dos primos  $p_A$  y  $q_A$  muy grandes y calcula  $n_A = p_A * q_A$
  - Elige una clave pública  $e_A$  y calcula su clave privada mediante la ecuación  $d_A = \text{inv} [e_A, \phi(n_A)]$ , siendo  $\phi(n_A) = (p_A-1)(q_A-1)$
  - Para romper la clave privada  $d_A$ , habrá que conocer  $\phi(n_A)$  y para ello debe factorizar  $n_A$  y encontrar los dos primos  $p_A$  y  $q_A$ , que también son secretos
  - El último desafío RSA Challenge (1991-2007) resuelto ha sido RSA-250 (250 dígitos = 829 bits) en febrero de 2020, por Fabrice Boudot, Pierrick Gaudry, Aurore Guillevic, Nadia Heninger, Emmanuel Thomé y Paul Zimmermann, con decenas de miles de máquinas trabajando en todo el mundo en una faena que se completó en pocos meses, pero equivalente a unos 2.700 años de una máquina Intel Xeon CPU Gold 6130 de 2,1 GHz



# Conclusiones de la Lección 3.4 (1/2)

- Multiplicar dos números primos muy grandes  $p \cdot q$ , es una operación que tiene una complejidad del tipo P (polinomial), es rápida y sencilla
- Pero, al igual que sucedía con los problemas de la mochila y del logaritmo discreto, resolver el problema de la factorización entera, es decir, conocido el producto  $n$  encontrar los primos  $p$  y  $q$ , tiene una complejidad NP (polinomial no determinista) y se vuelve computacionalmente intratable si esos primos son muy grandes
- El comportamiento típico del tiempo de cómputo del PFE versus el tamaño del módulo  $n$  en bits del problema de la factorización entera es de tipo exponencial
- Su uso en la criptografía lo encontramos en el algoritmo RSA. Para romper la clave privada, el atacante deberá factorizar el módulo  $n$  en los primos  $p$  y  $q$

# Conclusiones de la Lección 3.4 (2/2)

- Existe una decena algoritmos para resolver el PFE
  - Método de la criba de Eratóstenes, de Fermat, de Euler, de Dixon, de Williams  $p+1$ , de Pollar  $\rho$ , de Pollar  $p-1$ , de las fracciones continuas, de las curvas elípticas y de la criba numérica
  - El método de la criba numérica GNFS, General Number Field Sieve, es el más eficiente en la actualidad
- RSA Factoring Challenge es un desafío abierto a la factorización de primos RSA ( $n = p \cdot q$ ), que comienza en 1991 pero que RSA deja desierto sus premios en 2007. A pesar de ello, se sigue intentando factorizar esos números, siendo el último encontrado en febrero de 2020, un módulo de 250 dígitos o 829 bits, con miles de computadores trabajando durante algunos meses en todo el mundo, un cómputo equivalente a casi 3 mil años en un PC Intel a 2,1 GHz



# Lectura recomendada (1/2)

- Integer factorization / Factorización de enteros, Wikipedia
  - [https://en.wikipedia.org/wiki/Integer\\_factorization](https://en.wikipedia.org/wiki/Integer_factorization)
  - [https://es.wikipedia.org/wiki/Factorizaci%C3%B3n de enteros](https://es.wikipedia.org/wiki/Factorizaci%C3%B3n_de_enteros)
- SAMCrypt: Software de Aritmética Modular para Criptografía, María Nieto Díaz, dirección Jorge Ramió, 2018
  - [https://www.criptored.es/software/sw\\_m001t.htm](https://www.criptored.es/software/sw_m001t.htm)
- Matemáticas discretas en criptografía con SAMCrypt, CLCrypt, Jorge Ramió, 2019
  - [https://www.criptored.es/descarga/CLCrypt\\_entrega\\_09\\_Matematicas\\_Discretas\\_Criptografia\\_SAMCrypt.pdf](https://www.criptored.es/descarga/CLCrypt_entrega_09_Matematicas_Discretas_Criptografia_SAMCrypt.pdf)
- msieve153
  - <https://sourceforge.net/projects/msieve/>

# Lectura recomendada (2/2)

- Calculadora de factorización de números enteros, Darío Alpern, 2020
  - <https://www.alpertron.com.ar/ECMC.HTM>
- RSA Factoring Challenge
  - [https://en.wikipedia.org/wiki/RSA\\_Factoring\\_Challenge](https://en.wikipedia.org/wiki/RSA_Factoring_Challenge)
- RSA-250 Factored, Schneier on Security, Bruce Schneier, 2020
  - [https://www.schneier.com/blog/archives/2020/04/rsa-250\\_factore.html](https://www.schneier.com/blog/archives/2020/04/rsa-250_factore.html)
- MOOC Crypt4you, El algoritmo RSA, Lección 8 Ataque por factorización, Jorge Ramió, 2012
  - <https://www.criptored.es/crypt4you/temas/RSA/leccion8/leccion08.html>