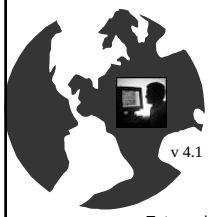
Capítulo 8
Teoría de la Complejidad Algorítmica

Seguridad Informática y Criptografía



Material Docente de
Libre Distribución

Ultima actualización del archivo: 01/03/06
Este archivo tiene: 31 diapositivas

Dr. Jorge Ramió Aguirre
Universidad Politécnica de Madrid

Introducción a la teoría de la complejidad

La teoría de la complejidad de los algoritmos permitirá, entre otras cosas, conocer la fortaleza de un algoritmo y tener así una idea de su vulnerabilidad computacional.

Complejidad Computacional

Los algoritmos pueden clasificarse según su tiempo de ejecución, en función del tamaño u orden de la entrada. Hablamos así de complejidad:

- Polinomial $\Rightarrow$ *comportamiento similar al lineal*
- Polinomial No Determinísta $\Rightarrow$ *comportamiento exponencial*

Esto dará lugar a “problemas fáciles” y “problemas difíciles” cuyo uso será muy interesante en la criptografía.

Operaciones bit en la suma

Si deseamos sumar dos números binarios n y m, ambos de k bits realizaremos k operaciones bit puesto que cada operación básica con los dígitos de una columna es una operación bit.

- Recuerde que 0+0 = 0, 0+1=1, 1+0 = 1, 1+1 = 0 con bit 1 de acarreo.
- Si un número tiene menos bits, se rellena con ceros por la izquierda.

Ejemplo: encontrar el número de operaciones bit necesarias en la suma en binario de 13+7 ⇒ 1101 + 0111 (de k = 4 bits)

```
    1 1 1 1   (bits de acarreo)
      1 1 0 1
+     0 1 1 1
  -------------
  1 0 1 0 0
```

Cada operación básica que hacemos con una columna se conoce como operación bit, luego necesitamos k = 4 operaciones bit.

Operaciones bit en la multiplicación

Para la multiplicación de un número n de k bits por un número m de h bits, el número de operaciones bit será igual a 2∗k∗h. Suponemos que k ≥ h.

- Recuerde que 0x0 = 0, 0x1=0, 1x0 = 0, 1x1 = 1.

Ejemplo: encontrar el número de operaciones bit necesarias en la multiplicación en binario 10x5 ⇒ 1010 x 101 (4 y 3 bits)

```
      1 0 1 0 x 1 0 1
      1 0 1 0
    0 0 0 0
+ 1 0 1 0       (procedemos ahora a sumar)
  1 1 0 0 1 0
```

Como cada operación básica entre dos bits es una operación bit, hemos realizado h∗k = 3∗4 multiplicaciones y luego k∗h = 4∗3 sumas, es decir en total 2∗k∗h = 24 operaciones bit.

La función O(n)

Las operaciones dependerán del tamaño de la entrada por lo que esta complejidad se expresará en términos del tiempo T necesario para el cálculo del algoritmo y del espacio S que utiliza en memoria, y se expresará mediante una función *f* (n), donde n es el tamaño de la entrada.

Esta función será una aproximación pues el resultado exacto dependerá de la velocidad del procesador.

$f(n) = O(g(n))$ Ejemplo

Y se define así: $f = O(n)$ *ssi* $\exists\ c_o, n_o / f(n) \leq c_o * g(n)$

http://www.mm.informatik.tu-darmstadt.de/courses/2002ws/ics/lectures/v14.pdf

Complejidad de una función *f(n)*

Si $f(n) = 4n^2 + 2n + 5$ ¿ $f = O(n^2)$?

¿se cumple que $c_o * g(n) = c_o * n^2 \geq f(n)$? Sea $c_o = 6$

c_o	n_o	$c_o n_o^2$	$f(n) = 4n^2 + 2n + 5$	¿$c_o * n^2 \geq f(n)$?
6	1	6	11	No
6	2	24	25	No
6	3	54	38	Sí
6	4	96	77	Sí

Se cumple siempre

Luego, la complejidad de *f* (n) es exponencial.

Tiempos de ejecución

En la expresión O(n) aparecerá el término que domina al crecer el valor de n.

- El tiempo de ejecución de un algoritmo T_1 que realiza 2n+1 operaciones es de tipo O(n); uno T_2 que realiza $3n^2+n+3$ operaciones será de tipo $O(n^2)$, etc.
- Para realizar la suma de la diapositiva anterior necesitamos O(n) = O(log n) operaciones bit y para el caso de la multiplicación, éstas serán $O(n*m) = O(\log n * \log m)$ operaciones bit.

+ Operación binaria: n+m (de k bits cada uno)
∗ Operación binaria: n∗m (de k y h bits respectivamente)

Algoritmos de complejidad polinomial

- Un algoritmo se dice que tiene tiempo de ejecución polinomial (no confundir con lineal) si éste depende polinómicamente del tamaño de la entrada.
- Si la entrada es de tamaño n y t es un entero, el número de operaciones bit será $O(\log^t n)$.

Ejemplos

Si t = 1, el sistema es lineal	Suma
Si t = 2, el sistema es cuadrático	Producto
Si t = 3, el sistema es cúbico	Máximo Común Divisor (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 se incrementará sólo en $\log^3 2$ operaciones bit.

Estos son los denominados problemas fáciles y son los que involucrarán un proceso de cifra y descifrado (o firma) por parte del o de los usuarios autorizados.

Algoritmos de complejidad no determinista

- Un algoritmo se dice que tiene tiempo de ejecución polinomial no determinista (en este caso exponencial) si éste depende exponencialmente del tamaño de la entrada.
- Si la entrada es de tamaño n y t es un entero, el número de operaciones bit será $O(n^t)$.

Ejemplo

Para t = 2, será exponencial de orden 2 → n!

Para t = 3, será exponencial de orden 3

Ejemplo de complejidad no determinista

Pregunta: El tiempo de ejecución de un algoritmo es $O(n^3)$. Si doblamos el tamaño de la entrada, ¿en cuánto aumentará este tiempo?

Solución: En el primer caso el tiempo es $O(n^3)$ y en el segundo $O((2n)^3) = O(8n^3)$. El tiempo para este sistema exponencial, se incrementará en 8 operaciones bit.

Estos son los denominados problemas difíciles y son a los que deberá enfrentarse un criptoanalista o atacante que desea romper una cifra o la clave de un usuario.

Comparativas de complejidad

- Los algoritmos polinómicos y exponenciales se comparan por su complejidad $O(n^t)$.
  - Polinómico constante $\Rightarrow$ $O(1)$
  - Polinómico lineal $\Rightarrow$ $O(n)$
  - Polinómico cuadrático $\Rightarrow$ $O(n^2)$
  - Polinómico cúbico $\Rightarrow$ $O(n^3)$ ... etc.
  - Exponencial $\Rightarrow$ $O(d^{h(n)})$

    donde d es una constante y h(n) un polinomio

Si suponemos un ordenador capaz de realizar 10^9 instrucciones por segundo obtenemos este cuadro:

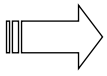


Tabla comparativa de tiempos

Entrada	O(n)	$O(n^2)$	$O(n^3)$	$O(2^n)$
n = 10	10^{-8} seg	10^{-7} seg	10^{-6} seg	10^{-6} seg
$n = 10^2$	10^{-7} seg	10^{-5} seg	10^{-3} seg	$4*10^{13}$ años
$n = 10^3$	10^{-6} seg	10^{-3} seg	1 seg	Muy grande

Incrementos de un orden de magnitud

Computacionalmente imposible

Entrada/10^9: Para $n = 100 \Rightarrow O(n^2) = 100^2/10^9 = 10^{-5}$ seg

Problemas de tipo NP

En criptografía nos interesan las funciones f(x) de un solo sentido, es decir:

- Fácil calcular f(x) pero muy difícil calcular $f^{-1}(x)$ salvo que conozcamos un secreto o trampa.

Porque dan lugar a problemas de tipo NP, polinomiales no deterministas, computacionalmente difíciles de tratar:

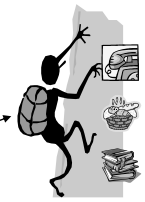
- Problema de la mochila
- Problema de la factorización
- Problema del logaritmo discreto

Definición del problema y ejemplos

- Problema logaritmo discreto en curvas elípticas
- Otros

El problema de la mochila

- Es un problema de tipo NP en el que el algoritmo debe realizar en cada paso una selección iterativa entre diferentes opciones.

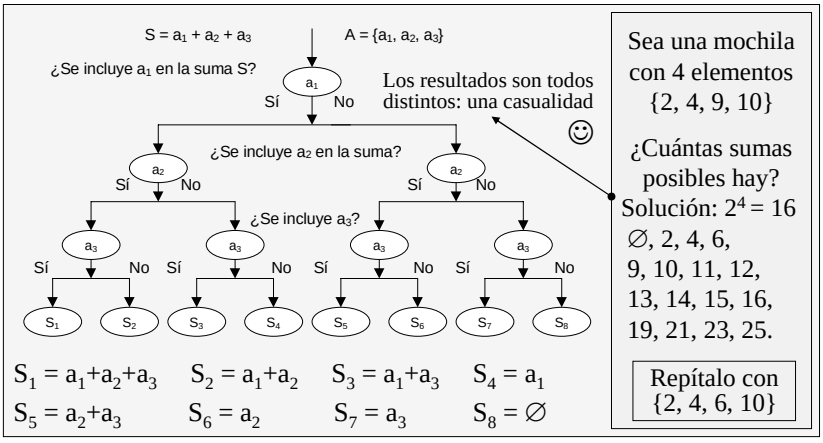


Enunciado:

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 (completa) con distintos elementos de ese conjunto sin repetir ninguno de ellos?

http://en.wikipedia.org/wiki/Knapsack_problem

Ejemplo del problema de la mochila



Sea una mochila con 4 elementos {2, 4, 9, 10}

¿Cuántas sumas posibles hay?

Solución: $2^4 = 16$

∅, 2, 4, 6, 9, 10, 11, 12, 13, 14, 15, 16, 19, 21, 23, 25.

Repítalo con {2, 4, 6, 10}

Hemos tenido que evaluar $2^3 = 8$ valores $\Rightarrow$ (carácter exponencial)

Interés de las mochilas en criptografía

¿Por qué tiene interés este problema en criptografía?

a) Es de tipo NP completo: su resolución por lo general implica una complejidad exponencial. Luego, será difícil de atacar o criptoanalizar.

b) Existe un caso en el que su resolución es lineal y, si la solución existe, ésta será única. Este caso se dará cuando $A = \{a_1, a_2, a_3, .., a_n\}$ está ordenado de menor a mayor de forma que a_i es mayor que la suma de los a_j que le preceden: $a_2 > a_1$; $a_3 > a_1 + a_2$; $a_4 > a_1 + a_2 + a_3$; etc.

Esto dará lugar a los criptosistemas de mochila tramposa que veremos en un próximo capítulo.

Problemas usados en criptografía asimétrica

Los problemas más usados en la criptografía asimétrica o de clave pública actualmente son:

- El problema de la factorización de números grandes PFNG
- El problema del logaritmo discreto PLD

En estos casos, cuando los números son del orden de mil bits (unos 310 dígitos) su cálculo se vuelve computacionalmente imposible debido al tiempo que deberíamos emplear.

Si lo desea, puede comprobar los ejemplos de las siguientes diapositivas usando el software de prácticas Fortaleza de libre distribución y que puede descargarlo desde esta dirección.

http://www.criptored.upm.es/software/sw_m001e.htm 

El problema de la factorización PFNG

Dado un número n que es el resultado del producto de dos o más primos, se pide encontrar estos factores.

Por ejemplo, cuando el valor n = p∗q es muy grande, el Problema de la Factorización de Números Grandes PFNG se vuelve computacionalmente intratable.

No obstante, el caso inverso, dado dos números primos p y q, encontrar el resultado p∗q = n, se trata de un problema de tipo polinomial.

Este problema se usará en la generación del par de claves del sistema de cifra con clave pública RSA.

http://home.netcom.com/~jrhowell/math/factor.htm

Compruebe lo que significa el PFNG

➲ Cálculo fácil o polinomial (función directa)

Calcule “a mano” los siguientes productos de dos primos y tome el tiempo aproximado que tarda en la operación:

a) 13∗31 b) 113∗131 c) 1.013∗1.031 *calcule...*☺

No vale usar calculadora...

¿Qué puede concluir de estos cálculos?

➲ Cálculo difícil o no polinomial (función inversa)

Usando la criba de Eratóstenes, factorice en dos primos los siguientes números y vuelva a tomar el tiempo empleado:

a) 629 b) 17.399 c) 1.052.627 *calcule...*☹

En el caso a) son primos de 2 dígitos, en b) de 3 y en c) de 4.

Solución al ejemplo anterior

➲ Dificultad polinomial (rápido)

a) 13∗31 = 403 b) 113∗131 = 14.803 c) 1013∗1031 = 1.044.403

A medida que aumenta el tamaño de la entrada, el tiempo de cálculo aumenta proporcionalmente con el número de dígitos.

➲ Dificultad no determinista (lento)

a) 629 b) 17.399 c) 1.052.627

Paciencia, un computador va a sufrir lo mismo ...

Da igual que el algoritmo sea éste muy elemental u otro más eficaz; aquí resulta evidente que el tiempo de cálculo aumenta mucho al incrementar en un dígito los números en cuestión. Es no lineal.

Solución: Los resultados a), b) y c) son el producto de los números primos inmediatamente superiores a los que se usaron en el cálculo polinomial es decir 17*37; 127*137; 1019*1033.

El problema del logaritmo discreto PLD

Dado un par de enteros α y β que pertenecen al Campo de Galois GF(p), se pide encontrar un entero x de forma tal que $x = \log_{\alpha} \beta \bmod p$.

Si el valor p es muy grande, el Problema del Logaritmo Discreto PLD es computacionalmente intratable.

No obstante, el caso inverso, dado dos números α y x, encontrar $\beta = \alpha^x \bmod p$ es un problema polinomial.

Este problema se usará, entre otros, en la creación de las claves del sistema de cifra con clave pública ElGamal y en el protocolo de intercambio de clave de Diffie y Hellman.

http://en.wikipedia.org/wiki/Discrete_logarithm

El PLD en su función directa o fácil

➲ Cálculo fácil o polinomial (función directa)

Calcule "a mano" las siguientes exponenciaciones mod p y tome el tiempo aproximado que tarda en la operación:

a) $5^4 \bmod 7$ b) $8^{17} \bmod 41$ c) $92^{11} \bmod 251$

$5^4 = 625$
$8^{17} = 2.251.799.813.685.248$
$92^{11} = 3.996.373.778.857.415.671.808$

Haciendo uso de la propiedad de reducibilidad vista en el apartado de matemáticas discretas, podrá bajar significativamente el tiempo de cálculo. Este tiempo será de tipo polinomial según el tamaño de la entrada.

Solución:
$5^4 \bmod 7 = 2$
$8^{17} \bmod 41 = 39$
$92^{11} \bmod 251 = 217$

El PLD y su función inversa o difícil

➲ Cálculo difícil o no determinista (función inversa)

Aunque existen varios algoritmos para este tipo de cálculos (al igual que para la factorización) use la fuerza bruta que se explica a continuación para encontrar los siguientes valores y vuelva a tomar el tiempo empleado:

a) $\log_5 2 \bmod 7$ b) $\log_8 39 \bmod 41$ c) $\log_{92} 217 \bmod 251$

Aplicando fuerza bruta en el 1er caso (la base elevada a todos los restos de p) al final se obtiene que $\log_5 2 \bmod 7 = 4$.

$5^1 \bmod 7 = 5$ $5^2 \bmod 7 = 4$ $5^3 \bmod 7 = 6$
$5^4 \bmod 7 = 2$ $5^5 \bmod 7 = 3$ $5^6 \bmod 7 = 1$

En término medio deberá recorrer la mitad del espacio de valores para encontrarlo ... ☹

Solución:
$\log_5 2 \bmod 7 = 4$
$\log_8 39 \bmod 41 = 17$
$\log_{92} 217 \bmod 251 = 11$

Logaritmo discreto con α generador

En el cuerpo p = 13, el 2 es un generador, luego:

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

Es decir

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

Se cumplirá siempre que $a^0 \bmod p = a^{p-1} \bmod p = 1$.

Logaritmo discreto con α no generador

En p=13 el 2 era generador, pero no así el número 3...

Luego

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

$\log_3 1 \bmod 13 = 0$	$\log_3 2 \bmod 13 = NE$	$\log_3 3 \bmod 13 = 1$
$\log_3 4 \bmod 13 = NE$	$\log_3 5 \bmod 13 = NE$	$\log_3 6 \bmod 13 = NE$
$\log_3 7 \bmod 13 = NE$	$\log_3 8 \bmod 13 = NE$	$\log_3 9 \bmod 13 = 2$
$\log_3 10 \bmod 13 = NE$	$\log_3 11 \bmod 13 = NE$	$\log_3 12 \bmod 13 = NE$

NE: no existe el logaritmo discreto en ese cuerpo

¿Hay más funciones NP?

Existen otros muchos problemas matemáticos que dan lugar a problemas del tipo NP, algunos de ellos basados en estas funciones unidireccionales *one-way functions* que tanto interesan en criptografía.

Las dos últimas funciones vistas, la factorización de números grandes y el logaritmo discreto, son las que más uso tienen de momento en la criptografía actual.

En la siguiente página Web encontrará una interesante lista con 88 problemas de tipo NP.

http://www.csc.liv.ac.uk/~ped/teachadmin/COMP202/annotated_np.html 

Fin del capítulo

Cuestiones y ejercicios (1 de 2)

1. Deseamos sumar de forma binaria el número 15 (1111) y el número 10 (1010), ambos de k = 4 bits. Haga la suma binaria y verifique que el número de operaciones bit desarrolladas es k = 4.
2. Si multiplicamos en binario 1010∗11, donde k = 4 bits y h = 2 bits, compruebe que el número de operaciones bit realizadas es 2∗k∗h.
3. ¿Por qué son interesantes los problemas de tipo NP en criptografía?
4. Defina el problema de la mochila y su posible utilización en un sistema de cifra.
5. ¿Es siempre única la solución de una mochila? Piense sobre el particular y su trascendencia si las utilizamos sin ningún control en sistemas de cifra.
6. Factorice mentalmente el valor n = 143. Intente hacer lo mismo para n = 1.243. ¿Qué opina ahora del problema de la factorización?

Cuestiones y ejercicios (2 de 2)

7. ¿Se le ocurre alguna forma de utilizar este problema de un solo sentido para cifrar información?
8. A partir de la ecuación $\beta = x^{\alpha} \bmod p$, defina el problema del logaritmo discreto.
9. Calcule $7^{13} \bmod 31 = C$ usando la propiedad de reducibilidad. Compruebe con Fortaleza que el exponente $13 = \log_7 C \bmod 31$.
10. ¿Qué utilidad le encuentra a este problema en criptografía?
11. ¿Qué relación existe entre generadores de un cuerpo, el conjunto completo de restos y el cálculo del logaritmo discreto?
12. ¿Qué sucede si deseamos encontrar un logaritmo discreto que no existe, por ejemplo $\log_5 19 \bmod 31$?

    Nota: si usa el software Fortaleza, deberá detener la operación al no contemplarse en su código este hecho ☹.

Use el portapapeles Prácticas del tema 8 (1/2)

Software Fortaleza: http://www.criptored.upm.es/software/sw_m001e.htm

1. Encuentre el tiempo que tarda el programa en calcular los siguientes productos de primos: 23x61; 281x829; 3001x5477; 26317x63901.
2. Encuentre cuánto tarda el programa en realizar un producto de dos números de 300 dígitos cada uno.
3. Factorice por Pollard Rho los siguientes productos y encuentre el tiempo que tarda el programa: 1403; 232949; 16436477; 1681682617.
4. Factorice por Pollard Rho los números de 16 dígitos que se indican y observe el tiempo empleado: 3745667802664727; 4044773133465121; 3907781049017851; 41710432708253. Saque conclusiones.
5. ¿Cuánto tarda el programa en demostrar la primalidad de estos números 50000000000000000059; 500000000000000000000000000009; 5000000000000000000000000000000000000021; de 20, 30 y 40 dígitos? Compárelos con los tiempos de factorización y saque conclusiones.

Use el portapapeles

Prácticas del tema 8 (2/2)

6. Encuentre el tiempo que tarda el programa en calcular las siguientes potencias modulares: $35^{1215} \bmod 3456$; $7824^{87652456} \bmod 34654783$; $891278265367^{876254356758778234002462} \bmod 76248874098100968727234 5$.
7. Si d = dígitos, ¿cuánto tiempo tarda el programa en calcular una potencia en los siguientes rangos de valores: $50d^{100d} \bmod 100d$; $50d^{100d} \bmod 150d$, $50d^{100d} \bmod 200d$, $50d^{100d} \bmod 250d$? Saque conclusiones.
8. Si d = dígitos, ¿cuánto tiempo tarda el programa en calcular una potencia en los siguientes rangos de valores: $50d^{25d} \bmod 200d$; $50d^{50d} \bmod 200d$, $50d^{75d} \bmod 200d$, $50d^{100d} \bmod 200d$? Saque conclusiones.
9. Compruebe las siguientes potencias y luego mediante los algoritmos de Búsqueda Exhaustiva, Paso Gigante - Paso Enano y Pohlig - Hellman, calcule el correspondiente logaritmo discreto. El módulo p es primo y α es un generador en p. Observe los tiempos de ejecución y saque conclusiones.
   - $401^{357} \bmod 87211 = 31211 \Rightarrow \log_{401} 31211 \bmod 87211 = 357$.
   - $246^{8924} \bmod 384773 = 67350 \Rightarrow \log_{246} 67350 \bmod 384773 = 8924$.