Complejidad Computacional

M.I. GONZÁLEZ VASCO / GRADO EN INGENIERÍA DE LA CIBERSEGURIDAD

UNIVERSIDAD REY JUAN CARLOS

Qué vamos a aprender

1. Nociones básicas de teoría de complejidad
2. Limitaciones de la teoría de complejidad (en solitario) para el análisis de herramientas criptográficas

Referencia: Capítulo 19 del Smart

1. Clases de Complejidad

Problemas de decisión

- Llamamos problema de decisión a todo problema que admite una solución binaria (si/no). Ejemplo: pertenencia a conjuntos (¿está x en el conjunto A?)
  - Dado un entero positivo, ¿es primo?
  - Dado un grafo, ¿puede colorearse con sólo k colores?
  - [problema de la mochila]

  Dados n elementos, con pesos $w_1,\ldots,w_n$ ¿es posible elegir algunos para llenar una mochila (completamente) que soporta un peso S

  ¿podemos encontrar bits $b_1,\ldots, b_n$ tales que $S=b_1w_1+\ldots+b_nw_n$?

  Este problema también tiene una versión computacional..

Problemas de búsqueda/computacionales

- Llamamos problema de búsqueda/computacional: se resuelven encontrando un conjunto de soluciones (que puede ser el conjunto vacío) a partir de ciertas premisas de partida.
  - [problema de la mochila] Dados n elementos, con pesos $w_1,\ldots,w_n$ y un umbral de capacidad S, calcular la secuencia de bits $b_1,\ldots, b_n$ tales que

$$S=b_1w_1+\ldots+b_nw_n$$

“Equivalencia” de paradigmas

- La Teoría de Complejidad se centra en problemas **de decisión,** debido a que puede demostrarse que ambos paradigmas son, en cierto sentido, equivalentes
- Una idea: Algoritmo 19.1 del Smart --- si disponemos de un oráculo para el problema de la mochila (decisión) podemos construir un algoritmo para el problema computacional (de búsqueda)

Clase P (polinomial)

- Se dice que un problema de decisión DP pertenece a la clase de complejidad P si existe un algoritmo $\mathcal{A}$ que al recibir una entrada I (codificada en n bits) para la cual la respuesta es SI, $\mathcal{A}$ responde en, a lo sumo, p(n) operaciones, siendo p un polinomio.
- Obs: si la respuesta es NO, ni siquiera le pedimos que termine en tiempo polinomial.
- La case P contiene a todos los problemas que pueden ejecutarse en tiempo *razonable*.
- Ejemplos:
  - dados tres números enteros x,y,z, ¿se cumple x=yz?
  - Dato un texto cifrado c, una clave k y un texto claro m, ¿es c un cifrado válido de m con la clave k? --sólo sirve si cifrar y descifrar es una tarea también polinomial… (ejemplo: algoritmo Square and Multiply, 15.1, pág. 249 del Smart)

Clase co-P

- Se dice que un problema de decisión DP pertenece a la clase de complejidad co-P si existe un algoritmo $\mathcal{A}$ que al recibir una entrada I (codificada en n bits) para la cual la respuesta es NO, $\mathcal{A}$ responde en, a lo sumo, p(n) operaciones, siendo p un polinomio
- Puede demostrarse que P = co-P

Idea: sea $\mathcal{A}$ un algoritmo que para una entrada I termina en tiempo n^c si la respuesta es SI. Ejecutamos $\mathcal{A}$ con I de entrada, si tras n^c+1 pasos no termina, paramos y damos NO como salida.

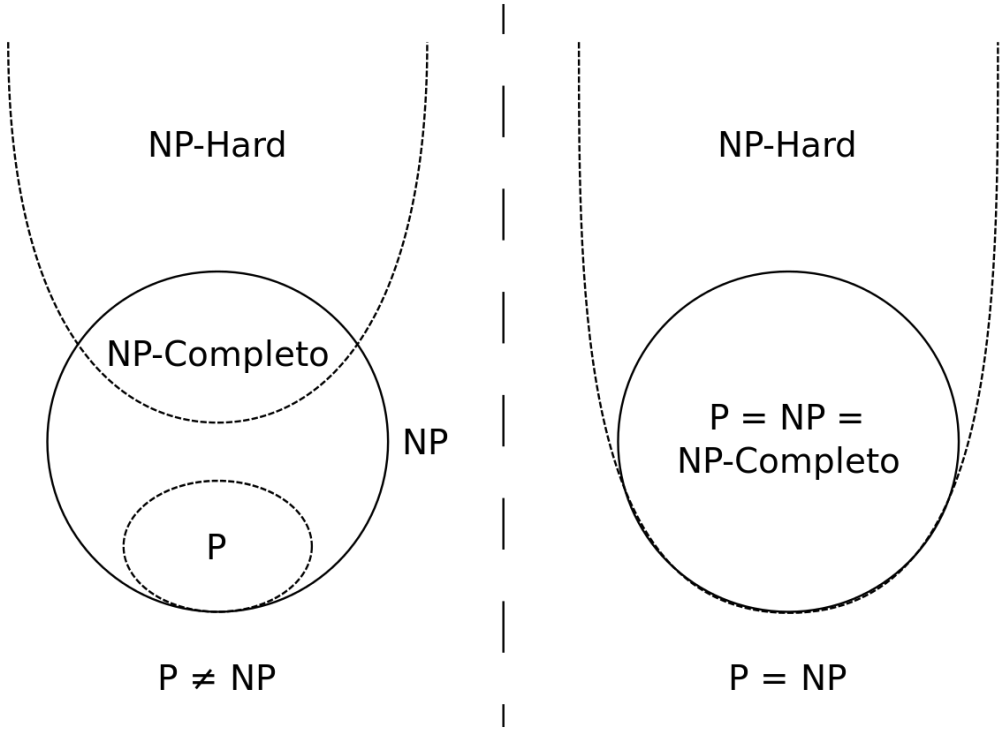
Clase NP

- Se dice que un problema de decisión DP pertenece a la clase de complejidad NP (non-deterministic polynomial time) si para toda entrada I (codificada en n bits) para la cual la respuesta es SI, existe un "*testigo*" w de esa respuesta que puede comprobarse en, a lo sumo, p(n) operaciones, siendo p un polinomio.
- Ejemplos:
  - ¿Es el número N compuesto?
  - ¿Tiene este problema de la mochila computacional solución?
- P está contenido en NP
- ¿¿¿P=NP???

Clases NP-complete, NP-hard

- Un problema DP se dice NP-complete **si está en NP** y cualquier otro problema de la clase NP se reduce a él en tiempo polinomial (es decir, si sabemos resolver DP, añadiendo esfuerzo polinomial, sabemos resolver cualquier otro problema de la clase NP)
- En cierto sentido, entendemos que en NP-complete están los problemas a los que es más difícil encontrar solución.
- Ejemplos:
  - El problema de los 3 colores en grafos
  - El problema de la mochila (general)
- Un problema se dice NP-hard si todo problema de la clase NP se reduce a él en tiempo polinomial.

¿Por qué es tan importante "P=NP"?



2. Notación asimptótica

¿Cómo medimos la eficiencia de un algoritmo?

- Contando las operaciones "elementales" que realiza.
- Cuidado: la definición de **algoritmo**: depende del modelo de computación.
  Ejemplo: algoritmo = máquina de Turing probabilística.
- Dado un algoritmo M se mide su eficiencia relativa a
  n = alguna medida del tamaño del input.
  Ejemplo:
  - input = x, una cadena de bits (formalmente $x \epsilon \{0,1\}^n$).
  - n = número de dígitos binarios (longitud) de x.

La notación "O grande"

Dadas dos funciones f(n) y g(n) de variable entera positiva n,

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

quiere decir que existen una constante real $c>0$ y un valor n_0 tales que

$$|f(n)| \leq c\,|g(n)|$$

para todo $n \geq n_0$.

Intuición: para valores "grandes" de n (es decir, asintóticamente), la función g (multiplicada por una constante, si es necesario) toma valores superiores a la función f.

La notación "Ω grande"

Dadas dos funciones f(n) y g(n) de variable entera positiva n,

$$f(n)=\Omega(g(n))$$

quiere decir que existen una constante real c>0 y un valor n_0 tales que

$$|f(n)| \geq c\,|g(n)|$$

para todo $n \geq n_0$.

Intuición: para valores "grandes" de n (es decir, asintóticamente), la función g (multiplicada por una constante, si es necesario) está por debajo de la función f.

La notación "Θ"

Dadas dos funciones f(n) y g(n) de variable entera positiva n,

$$f(n)=\Theta(g(n))$$

quiere decir que, simultáneamente, $f(n)=O(g(n))$ y $f(n)=\Omega(g(n))$,

es decir, existen constantes real $c,d>0$ y un valor n_0 tales que

$$c\,|g(n)| \leq |f(n)| \leq d\,|g(n)|$$

para todo $n \geq n_0$.

Intuición: para valores "grandes" de n (es decir, asintóticamente), el crecimiento de f es el mismo que el de g (salvo constantes).

Ejemplos

$f(n)=5n^3+10n^2+6n+40$

$f(n) \leq 5n^3+10n^3+6n^3+40n^3=61n^3 \rightarrow f(n)=O(n^3)$

$f(n) \geq 5n^3 \rightarrow f(n)=\Omega(n^3)$

Por tanto $f(n)=\Theta(n^3)$

Jerarquía (ejemplos)

O(1): constante

O(log(n)): logarítmica

O((log(n)^k): polilogarítmica

O(n): lineal

O(nlog(n)): cuasilineal

O(n^2): cuadrática

O(n^k): polinómica

O(a^n): exponencial

3. Determinismo Vs Aleatorización

Aleatorización

- Las clases sobre las que hablamos en 1. se refieren a algoritmos deterministas, i.e., con la misma entrada, siempre dan la misma salida (ejemplos: Capítulo 12 del Smart: ver Miller Rabin Vs Criba de Ertatóstenes "Trial Division))
- La aleatorización induce nuevas clases de complejidad, como RP, BPP y ZPP. Se cumple:

$$P \subseteq RP \subseteq NP$$

Aleatorización

- Cuando hablamos de algoritmos probabilistas/aleatorizados para resolver problemas de decisión, solemos clasificarlos en tres tipos:
  - Las Vegas (ejemplo: test de primalidad de Adelman-Huang)
    - Con probabilidad al menos ½, termina y da la respuesta correcta (en otro caso, no termina)
  - Atlantic City
    - Para cualquier salida (0/1), ésta es correcta con probabilidad al menos 2/3
  - Monte Carlo (ejemplo: Miller Rabin)
    - Si la respuesta es 0, siempre da como salida 0
    - Si la respuesta es 1, devuelve 1 con probailidad al menos ½

4. Notas para un criptólogo

Caveat: visión de la complejidad para criptografía

- La complejidad clásica utiliza el llamado paradigma "worst-case" (es complejidad en el peor de los casos)
- Muchos problemas son "fáciles" en media (o incluso en la mayoría de las entradas) pero caen en una clase de complejidad "elevada" por la existencia de (¡pocas!) entradas extremadamente difíciles de resolver (es el caso, por ejemplo, del problema de la mochila)
- En criptografía necesitamos problemas que son **difíciles** en media (o, "casi seguro"). Esto viene garantizado si son problemas para los que hay "random self-reductions", es decir, para los que cualquier entrada puede resolverse si somos capaces de resolver una entrada seleccionada al azar (e.g., problemas RSA/DDH)