Algoritmos y Complejidad

Dpto. Ingeniería Industrial, Universidad de Chile

IN47B, Ingeniería de Operaciones

Contenidos

¿ Qué es un Algoritmo?

Definición

- Un algoritmo es un conjunto de pasos bien definido que toma una entrada y produce una salida.
- Un algoritmo es una secuencia de pasos que transforma la entrada en la salida deseada.

Algoritmos relacionados a problemas computacionales

Un problema computacional es una deseada relación entre una entrada y una salida.
Un algoritmo resuelve un problema computacional si logra producir la relación deseada.

El problema de ordenar

Definición

Entrada: una secuencia de números $\langle a_1, \ldots, a_n \rangle$.

Salida: Un reordenamiento $\langle a'_1, \ldots, a'_n \rangle$ tal que $a'_i \leq a'_{i+1}, i = 1, \ldots, n-1$.

Ejemplo:

Buscamos un algoritmo que dada la secuencia $\langle 31, 41, 59, 26, 41, 58 \rangle$ entregue la secuencia $\langle 26, 31, 41, 41, 58, 59 \rangle$

Una entrada particular se llama una *instancia* del problema.

Más de algoritmos

Definición

Un algoritmo se dice *correcto* si para todas las instancias termina con una salida correcta. Un algoritmo *correcto* se dice *resolver* el problema computacional asociado.

Nota

Algoritmos que son incorrectos son de interés en algunos casos, especialmente si los errores tienen asociada alguna medida de probabilidad acotada (algoritmos aleatorios).

Los límites de los algoritmos

- Un problema / función / pregunta es *computable* si existe un algoritmo que lo resuelva / compute / responda.
- No todos los problemas / funciones / preguntas son computables.
- Computabilidad depende del modelo de *máquina* que usemos.
- No todos los problemas son iguales, hay algunos mas iguales que otros (George Orwell)
  - Existen clasificaciones de dificultad entre los problemas (Teoría de la Complejidad).
  - *P* , *NP*, *NP*-completo, *NP*-duro, co-*NP*, *P*space.
  - La pregunta más famosa en complejidad: $P \neq NP$?

Eficiencia

- ¿ ... y si los computadores fueran instantáneos?
  - Aún necesitamos saber si un algoritmo es *correcto*
  - Cualquier algoritmo correcto bastaría
  - Ejemplo algoritmo universal: (Particionar + Enumerar)
- ¿ ... Y en el mundo real?
  - Tiempo y memoria (espacio) es limitada.
  - Muchos algoritmos resuelven el mismo problema, pero con distinto uso de recursos
  - *InsertionSort* ordena n elem. en approx. $c_1 n^2$ tiempo.
  - *MergeSort* ordena n elem. en approx. $c_2 n \log_2(n)$ tiempo.
  - Hardware es también un factor en la velocidad.

Eficiencia

- Consideremos máquina A de 1GHz y máquina B de 10MHz (A es 100 veces más rápida que B).
- En A corremos *InsertionSort* en 10^6 elementos con $c_1 = 2$.
- En B correnos *MergeSort* en 10^6 elementos con $c_2 = 50$.
- El resultado es $t_A = \dfrac{2 \cdot (10^6)^2}{10^9} = 2000\text{s}$,
  $t_B = \dfrac{50 \cdot 10^6 \cdot \log_2(10^6)}{10^7} = 100\text{s}$

Comparando Velocidades

- Supongamos que disponemos de un PC de 1MHz, ¿ qué tamaño de problemas podemos resolver en el tiempo?

instr.	sec.	min.	hora	día	mes	año	siglo
$\log_2(n)$	2^{10^6}	$2^{10^{7,77}}$	$2^{10^{9,55}}$	$2^{10^{10,93}}$	$2^{10^{12,42}}$	$2^{10^{13,49}}$	$2^{10^{15,49}}$
$\sqrt{n}$	$2^{10^{1,6}}$	$10^{15,55}$	$10^{19,11}$	$10^{21,87}$	$10^{24,84}$	$10^{26,99}$	$10^{30,99}$
n	10^6	$10^{7,77}$	$10^{9,55}$	$10^{10,93}$	$10^{12,42}$	$10^{13,49}$	$10^{15,49}$
$n\log_2(n)$	$10^{4,79}$	$10^{6,44}$	$10^{8,12}$	$10^{9,44}$	$10^{10,86}$	$10^{11,90}$	$10^{13,83}$
n^2	1000	$10^{3,88}$	$10^{4,77}$	$10^{5,46}$	$10^{6,21}$	$10^{6,74}$	$10^{7,74}$
n^3	100	391	1532	4420	$10^{4,14}$	$10^{4,49}$	$10^{5,16}$
2^n	19	25	31	36	41	44	51
$n!$	9	11	12	13	15	16	17

InsertionSort

Problema:

entrada: secuencia de n números $\langle a_1, \ldots, a_n \rangle$

salida: reordenamiento $\langle a'_1, \ldots, a'_n \rangle$ tal que $a'_i \leq a'_{i+1}$.

InsertionSort

1: $[A]$ secuencia de entrada
2: **for** $j = 2, j \leq length[A]$ **do**
3: $key \leftarrow A[j]$, $i \leftarrow j - 1$.
4: /* *inserta* $A[j]$ *en la secuencia ordenada* $A[1], \ldots, A[j-1]$
5: **while** $i > 0$ and $A[i] > key$ **do**
6: $A[i+1] \leftarrow A[i]$, $i \leftarrow i - 1$.
7: **end while**
8: $A[i+1] \leftarrow key$.
9: **end for**

Convenciones en pseudo-códigos

- Identación identifica bloques de código.
- Aceptamos controles como **while**, **for**, **if-then-else**
- Texto que siga /* es considerado comentario.
- Para asignar valores usamos $a \leftarrow b$.
- Aceptamos uso de arreglos indexados por números consecutivos, i.e. $[A] = A[1], \ldots, A[n]$.
- Las expresiones booleanas son evaluadas de izquierda a derecha y*cortocircuitadas*. Por ejemplo (**True** or **A**) siempre retorna **True** sin evaluar **A**.
- Funciones reciben los parámetros por valor, y guardan una copia local.

Uso de recursos

- Analizar un algoritmo es tratar de predecir cuantos recursos usará el algoritmo.
- Recursos pueden ser memoria, disco duro, ancho de banda de comunicaciones.
- Típicamente nosotros usaremos tiempo.
- La idea es proveer de una escala de *calidad* de algoritmos.

Modelo Computacional

- Necesitamos definir un tipo de máquina a usar, que sea medianamente realista.
- Usaremos máquinas **RAM** (Random Access Machine), i.e. un PC común y corriente.
  - Instrucciones son ejecutadas secuencialmente (sin paralelismo)
  - Datos nativos son enteros y puntos flotante.
  - Control incluye condicionales, loops, llamada a sub-rutinas y retorno de rutinas.
  - Aritmética incluye $+, -, \cdot, /$ y multiplicación por 2^k.
  - Instrucciones básicas toman una unidad de tiempo.
  - Asumimos codificación binaria.
  - Números usan k bits (típicamente 32 o 64).

Buscando algo razonable

- Tiempo ejecución depende en el *Tamaño* del problema.
- Instancias del mismo tamaño pueden demorar distinto.
- ¿Cómo definimos *tamaño*?
  - Depende del problema.... usamos el tamaño natural.
  - Para ordenar, el tamaño es el largo de la entrada.
  - Para triangularizar $M \in \mathbb{R}^{n \times m}$, el tamaño es nm.
  - Para multiplicar dos números, el tamaño es el número de bits de la codificación binaria de los operandos.
- Tiempo de ejecución es el número de pasos básicos que el algoritmo realiza en una instancia dada.

Análisis de InsertionSort

InsertionSort

1: [A] secuencia de entrada. $c_1 n$
2: **for** $j = 2, j \leq length[A]$ **do**
3: $key \leftarrow A[j]$, $i \leftarrow j - 1$. $c_2(n-1)$
4: /* *inserta* $A[j]$ *en la secuencia ordenada* $A[1], \ldots, A[j-1]$
5: **while** $i > 0$ and $A[i] > key$ **do**
6: $A[i+1] \leftarrow A[i]$, $i \leftarrow i - 1$. $c_3 \sum(t_j : j = 2, \ldots, n)$
7: **end while**
8: $A[i+1] \leftarrow key$. $c_4(n-1)$
9: **end for**

$$T(n) = -(c_2 + c_4) + (c_1 + c_2 + c_4)n + c_3 \sum(t_j : j = 2, \ldots, n)$$

Análisis de InsertionSort

- Tiempo de ejecución varía para instancias del mismo tamaño, notemos que $1 \leq t_j \leq j$.
- Podemos decir que (mejor caso) $T(n) \geq an + b$ para a, b fijos, i.e. $T(n)$ es al menos lineal.
- Podemos decir que (peor caso) $T(n) \leq an^2 + bn + c$ para a, b, c fijos, i.e. $T(n)$ es a lo más cuadrática.
- ¿Podemos hablar de tiempo promedio?
  - Necesitamos definir distribución en la entrada.
  - En general engorroso y difícil.
  - Peor caso nos da cota superior.
  - Muchas veces el peor caso es bastante común.
  - En la mayoría de los casos, el peor caso es bastante representativo del promedio.

Órdenes de crecimiento

- Nuestro objetivo es clasificar algoritmos.
- Hemos hecho bastantes simplificaciones en el modelo computacional.
- Haremos una simplificación más, cuando comparamos tiempos de ejecución, eliminamos las *constantes*
  - Informalmente
    $\mathcal{O}(an^2 + bn + c) = \mathcal{O}(an^2 + bn) = \mathcal{O}(an^2) = \mathcal{O}(n^2)$.
  - $\mathcal{O}(n^k + \sum(a_i x^i : i = 0, \ldots, k-1)) = \mathcal{O}(n^k)$.
  - $\mathcal{O}(2^n + n^k) = \mathcal{O}(2^n)$.

Notación Asintótica

Notación Θ (asintóticamente equivalente)

Dado $f, g : \mathbb{R}_+ \to \mathbb{R}$, decimos que $f(n) = \Theta(g(n)) \Leftrightarrow \exists$ $c_1, c_2 \in \mathbb{R}_+, n_o \in \mathbb{N}$ tal que $c_1 g(n) \leq f(n) \leq c_2 g(n)$ $\forall n \geq n_o$.

Notación $\mathcal{O}$ (cota superior asintótica)

Dado $f, g : \mathbb{R}_+ \to \mathbb{R}$, decimos que $f(n) = \mathcal{O}(g(n)) \Leftrightarrow \exists$ $c \in \mathbb{R}_+, n_o \in \mathbb{N}$ tal que $f(n) \leq cg(n)$ $\forall n \geq n_o$.

Notación Ω (cota inferior asintótica)

Dado $f, g : \mathbb{R}_+ \to \mathbb{R}$, decimos que $f(n) = \Omega(g(n)) \Leftrightarrow \exists$ $c \in \mathbb{R}_+, n_o \in \mathbb{N}$ tal que $cg(n) \leq f(n)$ $\forall n \geq n_o$.

Notación Asintótica

Teorema

Dado $f, g : \mathbb{R}_+ \to \mathbb{R}$, $f(n) = \Theta(g(n)) \Leftrightarrow f(n) = \mathcal{O}(g(n))$ y $f(n) = \Omega(g(n))$.

Notación o (sobre-estimación cota superior asintótica)

Dado $f, g : \mathbb{R}_+ \to \mathbb{R}$, decimos que $f(n) = o(g(n)) \Leftrightarrow \lim(f(n)/g(n) : n \to \infty) = 0$.

Notación ω (sub-estimación cota inferior asintótica)

Dado $f, g : \mathbb{R}_+ \to \mathbb{R}$, decimos que $f(n) = \omega(g(n)) \Leftrightarrow \lim(g(n)/f(n) : n \to \infty) = 0$.

¿En Español?

- Podemos hacer una analogía entre estas notaciones y comparaciones entre números:
  - $f(n) = \Theta(g(n)) \approx f = g$.
  - $f(n) = \mathcal{O}(g(n)) \approx f \leq g$.
  - $f(n) = o(g(n)) \approx f < g$.
  - $f(n) = \Omega(g(n)) \approx f \geq g$.
  - $f(n) = \omega(g(n)) \approx f > g$.
- Algunas diferencias:
  - No todas las funciones son comparables en términos asintóticos: $n, n^{1+sin(n)}$.

Problema Computacional

Problema Abstracto:

Un *problema abstracto* Q es una relación binaria entre un conjunto de instancias I y un conjunto de soluciones S.

Ejemplo:

Problema: Camino mas Corto

Instancias: Es una tripleta formada por un grafo $G = (V, A)$ y dos nodos $u, v \in V$.

Soluciones: Secuencia de arcos consecutivos en G, incluyendo la secuencia vacía para representar que no existen caminos.

Problemas de Decisión y Optimización

Problema de Decisión

Un problema se dice de decisión, si el conjunto de soluciones posibles se puede reducir a $\{0, 1\}$ (equivalentemente a $\{si, no\}$).

Ejemplo:

Problema: Camino de largo fijo

Instancias: Una 4-tupla consistente de un Grafo $G = (V, A)$, un par de nodos $s, t \in V$ y un entero k.

Soluciones: 1 si existe un camino de s a t que no use mas de k arcos, 0 si no.

Problemas de Decisión y Optimización

Problema de Optimización

Un problema se dice de optimización si su objetivo es buscar el mínimo o máximo de una función sobre un conjunto definido (posiblemente vacío).

- La teoría de complejidad se centra en torno a problemas de decisión.
- En general, un problema de decisión puede re-escribirse como un problema de optimización y viceversa.
- El último punto permite usar la teoría de complejidad a problemas de optimización también.

Codificaciones:

- Necesidad de representar problemas para el computador (Ej. binaria).
- Otras codificaciones son posibles (decimal, hexadecimal, unaria).

Codificación

Una codificación (binaria) de un conjunto abstracto S es una función $e : S \to \{0,1\}^*$, donde $\{0,1\}^* = \bigcup_{i \in \mathbb{N}} \{0,1\}^i$.

Ejemplo

$e(\{1,2,3,4,\ldots) = \{1,10,11,100,\ldots\}$, así,
$e(17) = 10001$.

Problemas Concretos

Problema Concreto

Un problema se dice *concreto* si $I = \{0,1\}^*$ y si $S \subseteq \{0,1\}^*$. Problemas que un computador entiende.

Algoritmos en problemas Concretos

Un algoritmo A resuelve un problema concreto P en tiempo $\mathcal{O}(T(n))$, si dada una instancia i de tamaño $n = |i|$ provee una solución en tiempo $\mathcal{O}(T(n))$.

Funciones computables polinomialmente ($f \in \mathcal{P}$)

Una función $f : \{0,1\}^* \to \{0,1\}^*$ es computable polinomialmente $\Leftrightarrow \exists A$ algoritmo que computa f en tiempo $\mathcal{O}(n^k)$, $k \in \mathbb{N}$ fijo.

La Clase $\mathcal{P}$

Definición

$\mathcal{P}$ es el conjunto de problemas concretos de decisión para los cuales existe un algoritmo que lo resuelve en tiempo polinomial.

Codificando problemas abstractos (de decisión)

Un problema concreto $P : \{0,1\}^* \to \{0,1\}$ es una codificación de un problema abstracto $Q : S \to \{0,1\}$ si existe una función (codificación) $e : S \to \{0,1\}^*$ tal que $P(e(s)) = Q(s)$ para todo $s \in S$ y además $P(i) = 0$ para todo $i \in \{0,1\}^* \setminus e(S)$.

Complejidad de Problemas Abstractos

- Complejidad definida sobre problemas concretos.
- Complejidad problema depende de la codificación:
  - Problema de factorización de números es polinomial en codificación unaria (enumeración).
  - En codificación binaria, no se sabe.
- ¿Está todo perdido?
  - Dos codificaciones e_1, e_2 están relacionados polinomialmente $\Leftrightarrow$ existen $f_{12}, f_{21} \in \mathcal{P}$ tal que $\forall i \in \{0,1\}^*$ $f_{12}(e_1(i)) = e_2(i)$ y $f_{21}(e_2(i)) = e_1(i)$.
  - En general (sin codificaciones unarias), todas las codificaciones están relacionadas polinomialmente.
  - Implica que para codificaciones razonables, si una codificación de Q esta en $\mathcal{P}$, todas lo están.

Definiciones Básicas

Alfabeto: (Σ) Conjunto finito de elementos.

Palabra Vacía: (ε) Representa la palabra sin letras.

Lenguaje: (L) Conjunto de concatenaciones de elementos en Σ.

Lenguaje Completo: (Σ^*) Se define como $\Sigma^* = \cup(\Sigma^k : k \in \mathbb{N})$.

- Si consideramos $\Sigma = \{0, 1\}$ entonces Σ^* es conjunto de todas las instancias concretas.
- Por cada problema de decisión concreto P podemos asociar un lenguaje L_P definido como $L_P = \{i \in \Sigma^* : P(i) = 1\}$.

Definiciones Básicas

Aceptar (Reconocer) un Lenguaje

Un algoritmo A acepta (reconoce) un lenguaje L si $\forall i \in L \Rightarrow A(i) = 1$ y $\forall i \in \{0,1\}^*,\ A(i) = 1 \Rightarrow i \in L$.

Decidir un Lenguaje

Un algoritmo A decide un lenguaje L si $\forall i \in L\ \ A(i) = 1$ y $\forall i \in \{0,1\}^* \setminus L\ \ A(i) = 0$.

Clase de Complejidad

Una Clase de Complejidad es un conjunto de lenguajes que son decididos por algoritmos que cumplen una medida de complejidad.

La Clase $\mathcal{P}$

Definición:

La Clase $\mathcal{P} = \{L \subseteq \{0,1\}^* : \ L$ es decidido por un algoritmo polinomial $\}$.

Teorema:

La Clase $\mathcal{P} = \{L \subseteq \{0,1\}^* : \ L$ es aceptado por un algoritmo polinomial $\}$.

Certificados Polinomiales

- Consideremos el problema *CAMINO* en una instancia $I = \langle G, s, t, k \rangle$, y un camino C de s a t de largo menor igual a k.
- Decimos que C es un certificado de $I \in$ *CAMINO*.
  - Claramente podemos chequear si C es un camino de s a t en un tiempo polinomial en el tamaño de G.
- Ejemplo:
  - Problema: Ciclo Hamiltoniano
    - Dado grafo G, decidir si existe un ciclo simple en G que visite todos los nodos en G.
    - Definimos el lenguaje $HAM - CYCLE := \{\langle G \rangle : G \text{ is hamiltonian graph}\}$.

Certificados Polinomiales

- Existe un algoritmo que decide $HAM - CYCLE$?
  - Enumerar todas las permutaciones de $V(G)$.
  - Tiempo ejecución?
    - Codificando G como su matriz de incidencia, $|V(G)| = \Omega(\sqrt{n})$.
    - Permutaciones serian $\Omega(\sqrt{n}!) = \Omega(2^{\sqrt{n}})$.
    - No se conoce nada mejor.
- ... Y si solo queremos un certificado?
  - Un certificado de que $\langle G \rangle \in HAM - CYCLE$ seria un ciclo C Hamiltoniano en G.
  - Chequear que C es realmente un ciclo Hamiltoniano podemos hacerlo en $\Omega(n^2)$ donde $n = |\langle G \rangle|$.

Algoritmos de Certificación

Definición:

Un algoritmo $A : \Sigma^* \times \Sigma^* \rightarrow \{0, 1\}$ se dice que certifica x si existe y tal que $A(x, y) = 1$.
El lenguaje certificado por A es
$L = \{x \in \Sigma^* : \exists y \in \Sigma^* \text{ tal que } A(x, y) = 1\}$.

...Volviendo al Ejemplo

Con la definición anterior, claramente $HAM - CYCLE$ es *certificable* en tiempo polinomial, i.e. existe un algoritmo de certificación polinomial que lo certifica.

La Clase $\mathcal{NP}$

Definición:

La clase $\mathcal{NP}$ es el conjunto de lenguajes L para los que existe un algoritmo polinomial de certificación A y una constante c tal que

$$L = \left\{ x \in \{0,1\}^* : \begin{array}{l} \exists y \in \{0,1\}^*, \\ |y| = \mathcal{O}(|x|^c), \\ \text{y tal que } A(x,y) = 1 \end{array} \right\}$$

Decimos que A certifica L en tiempo polinomial.

Una Visión Alternativa

Pensemos en un mundo *infinitamente* paralelo....

La Clase $\mathcal{NP}$

Preguntas Abiertas

- es $\mathcal{NP}$= $\mathcal{P}$?
- es $\mathcal{NP}$= co-$\mathcal{NP}$?
- existe algún problema en $(\mathcal{NP} \cap co - \mathcal{NP}) \setminus \mathcal{P}$?

La idea central

¿Por qué creemos que $\mathcal{P} \neq \mathcal{NP}$?

- El argumento principal es la existencia de problemas $\mathcal{NP}$-completos.
  - Los problemas $\mathcal{NP}$-completos satisfacen lo siguiente: Si alguno de ellos esta en $\mathcal{P}$, entonces *todos* los problemas están en $\mathcal{P}$; i.e. $\mathcal{P} = \mathcal{NP}$.
  - A pesar de mas de 40 años de trabajo, no se conoce ningún algoritmo polinomial para ningún problema $\mathcal{NP}$-completo.
  - $HAM - CYCLE$ es $\mathcal{NP}$-completo.
  - Si $\mathcal{NP} \setminus \mathcal{P} \neq \emptyset$, $HAM - CYCLE$ sería uno de ellos.
- En algún sentido, los problemas $\mathcal{NP}$-completos, son los más difíciles en la clase $\mathcal{NP}$.

Reducibilidad

Definición

- Un problema P puede *reducirse* a otro problema P', si cualquier instancia $p \in P$ puede *refrasearse* como un problema $p' \in P'$, cuya solución provee la respuesta al problema original.
- Ejemplo: el problema de resolver ecuaciones lineales $ax + b = 0$ puede *reducirse* al problema de resolver ecuaciones cuadráticas $0x^2 + ax + b = 0$.
- En un sentido, si P puede reducirse a P', entonces, P no es más difícil de resolver que P'.

Reducibilidad

Definición

Un lenguaje L_1 es *reducible en tiempo polinomial* a un lenguaje L_2 ($L_1 \leq_p L_2$), si existe una función f computable en tiempo polinomial satisfaciendo:

- $f : \{0,1\}^* \rightarrow \{0,1\}^*$
- $\forall x \in \{0,1\},\ x \in L_1 \Rightarrow f(x) \in L_2$.
- $\forall x \in \{0,1\},\ x \notin L_1 \Rightarrow f(x) \notin L_2$.

Decimos que f es una función de reducción; un algoritmo F que computa f se llama algoritmo de reducción.

Algunas Consecuencias:

Si $L_1 \leq_p L_2$ y $L_2 \in \mathcal{P}$, entonces $L_1 \in \mathcal{P}$.

Reducibilidad

Cual es la idea?

- El concepto de reducibilidad nos permite decir que un problema *no es mas difícil que otro* salvo un factor polinomial.
- La idea central tiene que ver con que los problemas *fáciles* son aquellos que están en $\mathcal{P}$.

$\mathcal{NP}$-completitud

Definición:

La clase de lenguajes $\mathcal{NP}$-completos son aquellos que satisfacen lo siguiente:

- $L \in \mathcal{NP}$.
- $L' \leq_p L, \quad \forall L' \in \mathcal{NP}$.

¿En español?

- La clase de lenguajes $\mathcal{NP}$-completos, son los lenguajes más difíciles dentro de $\mathcal{NP}$.
- Nótese que si $\exists L \in \mathcal{NP} - \textit{completo} \cap \mathcal{P}$ entonces $\mathcal{NP} = \mathcal{P}$.
- Así mismo, seguro que si $\mathcal{P} \neq \mathcal{NP}$ entonces $\mathcal{NP} - \textit{completo} \subseteq \mathcal{NP} \setminus \mathcal{P}$.

$\mathcal{NP}$-completitud

- Existen problemas $\mathcal{NP}$-completos?
  - Un primer problema es 3-SAT.
  - Otros problemas incluyen:
    - Problema de clique en un grafo.
    - Programación entera.
    - Vertex Cover.
    - $HAM - CYCLE$.
- ¿Cómo podemos demostrar que un problema es $\mathcal{NP}$-completo?
  - El primero es el difícil, la idea, es codificar un algoritmo, con su entrada, como una secuencia de pasos lógicos.
  - Pero después es fácil.... Como demostraríamos que IP es $\mathcal{NP}$-completo?