CI7681 Teoría de Algoritmos **Mayo-Agosto 2002**

Lección 1: 29 Mayo

Profesor: Argimiro Arratia *Apuntador: Argimiro Arratia*

Presentación / Trivia Administrativa.

Notación: $\mathbb{N}$ es el conjunto de los naturales, $\mathbb{Z}$ es el conjunto de los enteros, $\mathbb{Q}$ es el conjunto de los racionales, $\mathbb{R}$ es el conjunto de los reales.

1.1 Algoritmos y Problemas

Algoritmo: es cualquier *procedimiento computacional* bien definido, que toma ciertos valores numéricos u objetos como **datos de entrada** y produce ciertos valores u objetos como **salida**.

¿Qué es un procedimiento computacional? Máquina de Turing (noción muy formal que estudiaremos más adelante), lenguaje de programación (e.g. Pascal, C).

En esta primera parte del curso, donde nos ocuparemos de estudiar diferentes técnicas para el diseño y el análisis de algoritmos, trabajaremos con una noción informal: un "pseudo Pascal".

Los algoritmos se diseñan para resolver problemas:

Problema Computacional: es un conjunto de pares donde la primera componente es el objeto de **entrada** y la segunda componente es el objeto de **salida**. Un poco más formal: si E (las entradas) y S (las salidas) son conjuntos, entonces un problema es $K \subseteq E \times S$.

Un algoritmo ρ **resuelve** un problema $K \subseteq E \times S$, si para todo $(a, b) \in K$, ρ al recibir por entrada a produce como salida b.

Ejemplo 1.1 *El problema de* ordenar números de manera ascendente*:*

$$\begin{aligned} \mathcal{O} \quad := \quad & \{(\langle a_1, \ldots, a_n\rangle, \langle b_1, \ldots, b_n\rangle) : a_i \in \mathbb{N}, \\ & b_i \in \{a_1, \ldots, a_n\}, 1 \leq i \leq n, b_1 \leq \ldots \leq b_n\} \end{aligned}$$

Observe que un algoritmo que resuelva $\mathcal{O}$ produce exactamente una salida para cada entrada. Pero no necesariamente todos los algoritmos definen funciones totales. Pueden resolver funciones parcialmente definidas o relaciones.

Tipos de Problemas

- Problemas de **Decisión**: la salida pertenece a un conjunto de sólo dos elementos ("aceptación" y "rechazo"). Por ejemplo,
$$K \subseteq E \times \{0, 1\}$$
- Problemas de **Optimización**: la salida es el **máximo** o **mínimo** valor de un conjunto de posibilidades. Por ejemplo,
$$K \subseteq E \times \{max[S], min[S]\}$$

Ejemplo 1.2 *(1) El problema de la tricoloración de un grafo:*

$$\begin{aligned} \textit{3COL} \quad := \quad & \{(G, r) : G \textit{ es un grafo y} \\ & r = 1 \textit{ si } G \textit{ es 3-colorable o } r = 0 \textit{ en caso contrario}\} \end{aligned}$$

es un problema de decisión.

(2) El problema cuya entrada es $n \in \mathbb{N}$ y salida es el divisor de n más grande y diferente de n es un problema de optimización.

1.2 Análisis de Algoritmos

Deseamos establecer una medida de cúan eficiente es un algoritmo que hallamos diseñado. Eficiencia dependerá del contexto. En general nos interesa medir eficiencia en términos del **tiempo** (y ocasionalmente del **espacio**) que consume un algoritmo en su ejecución.

El **tiempo** dependerá de: 1) tamaño de la entrada (asumimos que mientras más grande la entrada, mayor será el tiempo); 2) operaciones que realice el algoritmo.

El tamaño de la entrada dependerá del contexto del problema. Adoptaremos la postura informal de asumir que, por ejemplo, un algoritmo para un problema de grafos recibe por entrada vértices y arcos, y, por lo tanto, el tamaño de estas entradas es función del número de vértices o arcos; un algoritmo para un problema numérico recibe por entrada números naturales, y, por lo tanto, el tamaño de estas entradas es función de la cantidad de números que se desean procesar, etc.

Se nos podría exigir ser más consistentes con la realidad donde, por ejemplo, el lenguaje de una computadora es binario y, en consecuencia, considerar nuestros problemas computacionales como sucesiones de 0 y 1, i.e. palabras en $\{0, 1\}^*$. Esto además de no ayudar en el análisis de un algoritmo, no es claro que sea muy adecuado ya que la manera de codificar problemas como elementos de $\{0, 1\}^*$ no es única y depende del problema.

Establecido ya lo que entendemos por el tamaño de la entrada, convengamos en lo que será nuestra medida del tiempo de ejecución de un algoritmo.

> *La ejecución de una* **operación atómica** *en nuestro algoritmo, corresponde a una cantidad constante de tiempo.*

Motivación: Entendemos por **operación atómica** (o primitiva) aquella cuyo tiempo de ejecución es independiente de las variables o constantes involucradas. Por ejemplo, una instrucción de asignación:

$$a := b$$

o un test de comparación:

$$\text{¿ es } a < b \text{ ?}$$

son operaciones atómicas, ya que su ejecución es independiente de los valores de a y de b. Sin embargo, dependiendo de la arquitectura de nuestra máquina. la ejecución de cada una de estas operaciones atómicas tiene un costo (fijo). Asignamos arbitrariamente un costo c_1 de tiempo para $a := b$ y un costo c_2 de tiempo para $a < b$.

Por otra parte, la instrucción:

$$\text{para } i = 1 \text{ a } n \text{ hacer } A[i] = 0 \qquad (*)$$

donde A es un arreglo, no es una instrucción atómica ya que el tiempo de ejecución depende de n y de la longitud de A. Sin embargo, esta instrucción involucra las operaciones atómicas $i = m$ y $A[i] = 0$, a las que asignamos un costo c_1 y c_2 de tiempo, respectivamente. Asi, obtenemos como tiempo de la instrucción $(*)$:

$$c_1 n + c_2 n = (c_1 + c_2)n = (cte)n$$

ya que se ejecutan n veces.

Definición 1.3 *La* **complejidad temporal del peor caso** *de un algoritmo ρ se define como una función T con dominio en $\mathbb{N}$, tal que para cada $n \in \mathbb{N}$:*

$$\begin{aligned} T(n) \quad := \quad & max\{t(e) : \rho \text{ lee } e, \text{ una entrada de tamaño } n, \\ & \text{y consume tiempo } t(e)\} \cup \{0\} \end{aligned}$$

es decir $T(n)$ es el máximo de los tiempos que toma ρ al trabajar con todas las entradas de longitud n posibles. Se incluye el 0 para hacer T una función total, ya que puede haber un n para el que no existan entradas de esa longitud.

Observaciones:

- Esta definición asume que todos los algoritmos paran con todas las entradas posibles. Esto no es cierto en general, pero si lo es respecto a los algoritmos que estudiaremos en este curso.
- Se asume que para cada n existe un número finito de entradas de tamaño n. Esto concuerda con la realidad donde los problemas son sucesiones de 0 y 1.

Pregunta: ¿Cuántas palabras de longitud n se pueden formar con 0 y 1?

1.3 Notación asintótica

Con frecuencia resulta difícil describir con exactitud la función $T(n)$ de la Definición 1.3 y, en ese caso, nos transamos por un *estimado asintótico*, es decir, calculamos una función que acote superiormente los valores de $T(n)$, ignorando constantes y términos de menor orden.

Definición 1.4 *Sean $f(n)$ y $g(n)$ funciones con dominio en $\mathbb{N}$. Entonces,*

- $f(n) = O(g(n))$ *si existen naturales c y n_0 tal que $f(n) \leq cg(n)$ para todo $n \geq n_0$.*
- $f(n) = \Omega(g(n))$ *si $g(n) = O(f(n))$.*
- $f(n) = \Theta(g(n))$ *si $f(n) = O(g(n))$ y $g(n) = O(f(n))$*

1.4 Algoritmos para el problema de Ordenamiento ("Sorting")

Finalizamos esta lección analizando dos algoritmos para resolver el problema de ordenar una sucesión de números de manera creciente (ver Ejemplo 1.1), popularmente conocido como **sorting**.

La filosofía de estos algoritmos es *"voy resolviendo en la medida en que voy viendo"*: imagine un ente emisor que envia la sucesión de n números, de uno en uno, a un ente receptor que debe ordenarlos en forma ascendente y, para no perder tiempo, debe hacerlo en la medida en que recibe la información parcial y no esperar a tener toda la información. Este ente receptor debe entonces **insertar** el número que recibe dentro del orden de los números que ya posee. Tenemos entonces un algoritmo de **orden por inserción** (en inglés: **Insertion Sort**).

El algoritmo en la Figura 1.1 ordena por inserción.

Orden-por-Inserción(Entrada: A un arreglo de longitud n)
1. `para` $i := 2$ `hasta` n `hacer`
2. $j := i - 1$
3. $k := A[i]$
4. `mientras` $(j > 0$ & $A[j] > k)$ `hacer`
5. $A[j + 1] = A[j]$
6. $j := j - 1$
7. `% fin de mientras;`
8. $A[j + 1] := k$
9. `%fin de para`
10. `%fin`. Salida: A

Figura 1.1: Algoritmo Orden-por-Inserción (**Insertion–Sort**).

El algoritmo en la Figura 1.2 ordena por inserción de manera un poco más simplona.

Orden-por-Inserción-Ingenuo(Entrada: A un arreglo de longitud n)
1. `para` $i := 2$ `hasta` n `hacer`
2. $j := 1$
3. `mientras que` $j < i$ `hacer`
4. `si` $A[i] < A[j]$ `entonces`
5. `intercambiar los valores de` $A[j]$ y $A[i]$
6. `% fin de si`
7. $j := j + 1$
8. `% fin de mientras;`
9. `%fin de para`
10. `%fin`. Salida: A

Figura 1.2: Algoritmo Orden-por-Inserción-Ingenuo.

Observe que el primer algoritmo compara un número con el mayor de la lista que ya ordenó (procede "de derecha a izquierda"), mientras el segundo algoritmo (el ingenuo) compara un número con todos los de la lista que ya ordenó (procede "de izquierda a derecha"); por lo que podemos suponer que el segundo se comporta peor que el primero. Veamos.

1.4.1 Análisis de Orden-por-Inserción (Figura 1.1)

Sea c_i, $i = 1, 2, 3, 4, 5, 6, 8$, el costo de ejecución de las operaciones atómicas en la línea i (los comentarios tienen costo 0 porque el compilador los ignora). La línea 1 se ejecuta n veces (los $n - 1$ valores que toma i

y una última para salir del ciclo); las líneas 2, 3, y 8 se ejecutan $n-1$ veces; si para cada valor de i, t_i es el número de veces que se ejecuta el test de "mientras", entonces la línea 4 se ejecuta $\sum_{i=2}^{n} t_i$ veces, y las líneas 5 y 6 se ejecutan $\sum_{i=2}^{n}(t_i - 1)$ veces.

Por lo tanto, para cada entrada de longitud n, podemos estimar el tiempo de ejecución del algoritmo por la fórmula

$$t(n) = c_1 n + (c_2 + c_3 + c_8)(n-1) + c_4 \sum_{i=2}^{n} t_i + (c_5 + c_6) \sum_{i=2}^{n} (t_i - 1)$$

El **peor caso** ocurre cuando A está ordenado de manera decreciente: $A[1] \geq A[2] \geq \ldots \geq A[n]$. Debemos entonces comparar cada $A[i]$ con todo $A[1, 2, \ldots, i-1]$. Entonces, $t_i = i$ y $\sum_{i=2}^{n} i = \frac{n(n+1)}{2} - 1$ y $\sum_{i=2}^{n}(i-1) = \frac{n(n-1)}{2}$. Luego

$$\begin{aligned} t(n) &= O(n) + c_4 \left(\frac{n(n+1)}{2} - 1 \right) + (c_5 + c_6) \frac{n(n-1)}{2} \\ &= O(n) + O(n^2) = O(n^2) \end{aligned}$$

El **mejor caso** ocurre cuando A está ordenado de manera creciente; entonces $t_i = 1$ para todo i, y el aporte de líneas 4–6 es $O(n)$; en consecuencia $t(n) = O(n)$.

1.4.2 Análisis de Orden-por-Inserción-Ingenuo (Figura 1.2)

El análisis es similar al de Orden-por-Inserción. salvo que el test de "mientras" en la línea 3 siempre se ejecuta, para cada i, a través de $j = 1, 2, \ldots, i$. Luego $t(n) = \Theta(\sum_{i=2}^{n} i) = \Theta(n^2)$. Y he aqui el costo de la ingenuidad: este algoritmo es, para cualquier entrada, del orden de n^2.

1.5 Notas Adicionales del Apuntador

Para complementar la observación después del Ejemplo 1.1, considere el problema que tiene por entrada un natural n y como salida un par (m, k) con $m, k \neq 1$ y tal que $k \cdot m = n$. Un algoritmo que resuelva este problema producirá *alguna* salida válida correspondiente a una entrada, pero no necesariamente todas las salidas posibles: $20 = 2 \cdot 10 = 4 \cdot 5$. Por otra parte, dicho algoritmo puede quedar "colgado": al recibir por entrada un número primo no podrá producir un par apropiado.

Un grafo G es 3-colorable si es posible colorar sus vértices, usando a lo sumo 3 colores distintos, de manera que cualquier par de vértices adyacentes (i.e. conectados por un arco) tienen colores distintos.

La noción de *operación atómica* es necesariamente informal ya que no hemos definido formalmente lo que es un algoritmo. Si escogemos la máquina de Turing como modelo de algoritmo, entonces una operación atómica corresponde a un movimiento de la máquina de Turing.

Referencias

[**CLR90**] y [**AHU79**]