Facultad de Ciencias de la Computación

Introducción al análisis de algoritmos

ABRAHAM SÁNCHEZ LÓPEZ
GRUPO MOVIS
FCC-BUAP
HTTP://ASANCHEZ.CS.BUAP.MX/EVENTS.HTML

Análisis de algoritmos

- Un algoritmo es claramente un conjunto especificado de instrucciones simples que deben seguirse para resolver un problema.
- Una vez que se da un algoritmo para un problema y se decide (de alguna manera) que sea correcto, un paso importante es determinar la cantidad de recursos, como el tiempo o el espacio, que requerirá el algoritmo.
- Un algoritmo que resuelve un problema pero requiere un año no sirve para nada. Del mismo modo, un algoritmo que requiere cientos de gigabytes de memoria principal no es (actualmente) útil en la mayoría de las máquinas.
- En este seminario, discutiremos (aunque no se realizará todo!!)
  - Cómo estimar el tiempo requerido para un programa.
  - Cómo reducir el tiempo de ejecución de un programa de días o años a fracciones de segundo.
  - Los resultados del uso descuidado de la recursión.
  - Algoritmos muy eficientes para resolver diferentes problemas.

Antecedentes matemáticos, I

- El análisis requerido para estimar el uso de recursos de un algoritmo es generalmente un problema teórico y, por lo tanto, se requiere un marco formal.
- Comenzamos con algunas definiciones matemáticas.
- A lo largo del seminario utilizaremos las siguientes cuatro definiciones:
- **Definición 1**.

  $T(N) = O(f(N))$ si hay constantes positivas c y n_0 tales que $T(N) \leq cf(N)$ cuando $N \geq n_0$.
- **Definición 2.**

  $T(N) = \Omega(g(N))$ si hay constantes positivas c y n_0 tales que $T(N) \geq cg(N)$ cuando $N \geq n_0$.
- **Definición 3.**

  $T(N) = \Theta(h(N))$ sí y solo sí $T(N) = O(h(N))$ y $T(N) = \Omega(h(N))$
- **Definición 4.**

Antecedentes matemáticos, II

$T(N) = o(p(N))$ si para todas las constantes positivas c existe un n_0 tal que $T(N) < cp(N)$ cuando $N > n_0$. Menos formalmente, $T(N) = o(p(N))$ si $T(N) = O(p(N))$ y $T(N) \neq \Theta(p(N))$.

- La idea de estas definiciones es establecer un orden relativo entre las funciones.
- Dadas dos funciones, generalmente hay puntos donde una función es más pequeña que la otra, por lo que no tiene sentido 'reclamar', por ejemplo, $f(N) < g(N)$.
- Así, comparamos sus tasas de crecimiento relativas.
- Cuando apliquemos esto al análisis de algoritmos, veremos por qué esta es la medida importante.
- Si utilizamos los operadores de desigualdad tradicionales para comparar las tasas de crecimiento, entonces la primera definición dice que la tasa de crecimiento de $T(N)$ es menor o igual que ($\leq$) la de $f(N)$.

Antecedentes matemáticos, III

- La segunda definición, $T(N) = \Omega(g(N))$, dice que la tasa de crecimiento de $T(N)$ es mayor o igual que ($\geq$) la de $g(N)$.
- La tercera definición, $T(N) = \Theta(h(N))$, dice que la tasa de crecimiento de $T(N)$ es igual a (=) la tasa de crecimiento de $h(N)$.
- La última definición, $T(N) = o(p(N))$, dice que la tasa de crecimiento de $T(N)$ es menor que (<) la tasa de crecimiento de $p(N)$.
- Esto es diferente de O, porque O permite la posibilidad de que las tasas de crecimiento sean lo mismo.
- Para probar que algunas funciones $T(N) = O(f(N))$, generalmente no aplicamos estas definiciones formalmente, sino que utilizamos un repertorio de resultados conocidos.
- En general, esto significa que una prueba (o la determinación de que el supuesto es incorrecto) es un cálculo muy simple y no debe incluir cálculos, excepto en circunstancias extraordinarias (no es probable que ocurra en un análisis de algoritmo).

Antecedentes matemáticos, IV

- A continuación se presenta una lista de tasas de crecimiento típicas.

Función	Nombre
c	Constante
log N	Logarítmica
$\log^2 N$	Log cuadrada
N	Lineal
N log N	
N^2	Cuadrática
N^3	Cúbica
2^N	Exponencial

- Las cosas importantes a saber son:
- **Regla 1.**

Si $T_1(N) = O(f(N))$ y $T_2(N) = O(g(N))$, entonces

a) $T_1(N) + T_2(N) = O(f(N) + g(N))$ (intuitivamente y menos formalmente es $O(\max (f(N) + g(N)))$,

b) $T_1(N) * T_2(N) = O(f(N) * g(N))$.

Antecedentes matemáticos, V

- **Regla 2.**

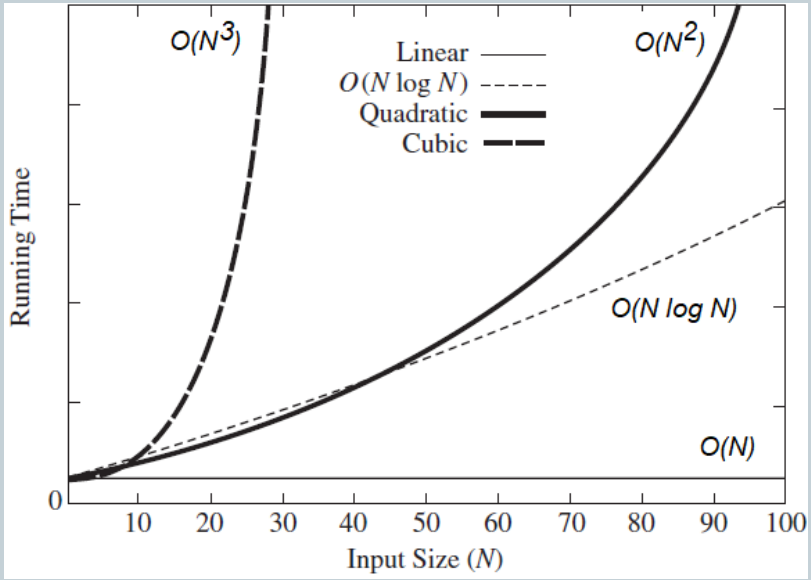
  Si $T(N)$ es un polinomio de grado k, entonces $T(N) = \Theta(N^k)$.
- **Regla 3.**

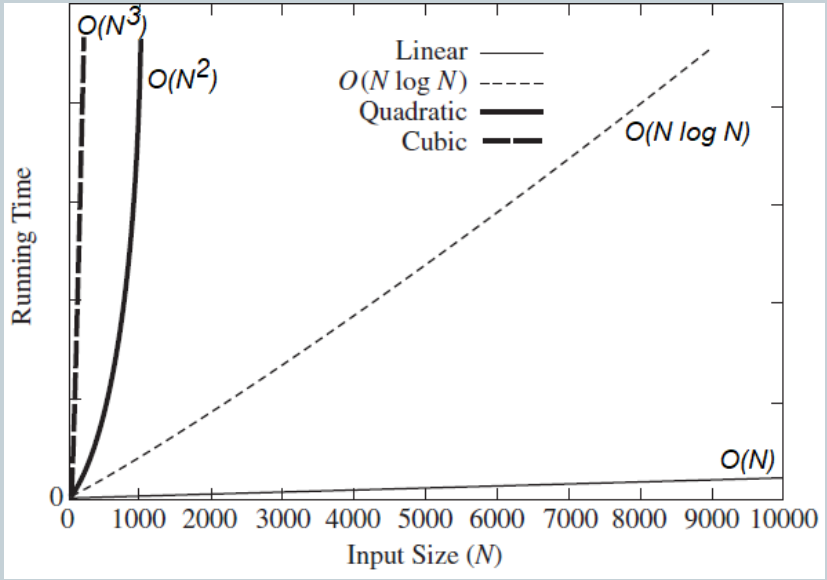
  $\log^k N = O(N)$ para cualquier constante k. Esto nos dice que los logaritmos crecen muy lentamente.
- En este seminario, no realizaremos las demostraciones de las reglas. Pero esta información es suficiente para ordenar por tasas de crecimiento la mayoría de las funciones comunes como se muestra en la tabla del acetato 6.
- Igualmente, siempre se pueden determinar las tasas de crecimiento relativo de dos funciones $f(N)$ y $g(N)$ mediante el cálculo del $\lim_{n\to\infty} f(N)/g(N)$, usando la regla de L'Hôpital si es necesario.
- El límite puede tener uno de cuatro valores:
  - El límite es 0: esto significa que $f(N) = o(g(N))$.
  - El límite es $c \neq 0$: esto significa que $f(N) = \Theta(g(N))$.

Antecedentes matemáticos, VI

- El límite es ∞: esto significa que $g(N) = o(f(N))$.
- El límite oscila: no hay ninguna relación (no se detalla en este seminario).

- Graficas de varios algoritmos de la suma de la subsecuencia máxima. Existen muchos algoritmos para resolver este problema y su rendimiento varía drásticamente.





Modelo

- Para analizar algoritmos en un marco formal, se necesita de un modelo de computación.
- Nuestro modelo es básicamente una computadora normal, en la cual las instrucciones sencillas como suma, multiplicación, comparación y asignación (pero a diferencia de una real), esta tarda exactamente una unidad de tiempo.
- Es obvio que este modelo tiene algunas debilidades.
- En la vida real, por supuesto, no todas las operaciones tardan exactamente el mismo tiempo.
- En general, el recurso más importante a analizar es el tiempo de ejecución.
- Hay varios factores que afectan el tiempo de ejecución de un programa (el compilador, la computadora utilizada).
- Otros factores relevantes son el algoritmo usado y su entrada.
- El tamaño de la entrada, suele ser la consideración principal.

Qué analizar?

- Definimos dos funciones, $T_{\text{prom}}(N)$ y $T_{\text{peor}}(N)$, como el tiempo de ejecución promedio y el del peor caso, respectivamente, empleados por un algoritmo para una entrada de tamaño N.
- Claro está, $T_{\text{prom}}(N) \leq T_{\text{peor}}(N)$.
- Si hay más de una entrada, esas funciones pueden tener más de un argumento.
- Cabe señalar que en general la cantidad requerida es el tiempo del peor caso, a menos que se especifique otra cosa.
- Una razón para esto es que se da una cota para todas las entradas, incluyendo entradas particularmente malas, que un análisis del caso promedio no puede ofrecer.
- La otra razón es que la cota del caso promedio suele ser mucho más difícil de calcular.

Un ejemplo sencillo, I

- Acá tenemos un fragmento de un programa sencillo para calcular $\sum_{i=1}^{n} i^3$:

```
        public static int sum( int n )
        {
            int partialSum;
1           partialSum = 0;
2           for( int i = 1; i <= n; i++ )
3               partialSum += i * i * i;
4           return partialSum;
        }
```

- El análisis de este programa sencillo. Las declaraciones no cuentan en el tiempo.
- Las líneas (1) y (4) cuentan por una unidad de tiempo cada una. La línea (3) cuenta por cuatro unidades cada vez que se ejecuta (dos multiplicaciones, una adición y una asignación) y se ejecuta *n* veces, para un total de 4*n* unidades.
- La línea (2) tiene el costo oculto de la iniciación de *i*, la comprobación de $i \leq n$ y el incremento de *i*.

Un ejemplo sencillo, II

- El costo total de todo ello es 1 para la iniciación, $n + 1$ para todas las comprobaciones y n para todos los incrementos, lo cual da $2n + 2$.
- Se ignoran los costos de llamar y retornar de la función, para un total de $6n + 4$. Así, se dice que la función es de orden $O(n)$.

- Es importante plantear reglas generales para una mejor comprensión:
  - Regla 1 – ciclos FOR
  - Regla 2 – ciclos FOR anidados
  - Regla 3 – Proposiciones consecutivas
  - Regla 4 – IF/ELSE

Inducción matemática, I

- Demostrar por inducción sobre $n \geq 0$, las siguientes igualdades.

a) $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$

b) $\sum_{i=1}^{n} 2^i i = (n-1)2^{n+1} + 2$

$a) \sum_{i=1}^{n} i = \frac{n(n+1)}{2}$

Caso básico. Si $n = 0, \sum_{i=1}^{n} i = 0 = \frac{n(n+1)}{2}$

Paso de inducción. Suponiendo que la igualdad es cierta para n (hipótesis de inducción), tenemos para $n+1$:

$$\sum_{i=1}^{n+1} i = \sum_{i=1}^{n} i + (n+1) \overset{\text{h.i}}{=} \frac{n(n+1)}{2} + (n+1) = \frac{(n+1)(n+2)}{2}$$

Inducción matemática, II

$$b)\sum_{i=1}^{n} 2^i i = (n-1)2^{n+1} + 2$$

Caso básico. Si $n = 0, \sum_{i=1}^{n} 2^i i = 0 = -2 + 2 = (n-1)2^{n+1} + 2$

Paso de inducción. Suponiendo que la igualdad es cierta para n (hipótesis de inducción), tenemos para $n+1$:

$$\sum_{i=1}^{n+1} 2^i i = \sum_{i=1}^{n} 2^i i + 2^{n+1}(n+1) = (n-1)2^{n+1} + 2 + 2^{n+1}(n+1) =$$

$$= 2^{n+1}2n + 2 = n2^{n+2} + 2.$$

Asintóticas, I

- Justificar de forma razonada si las siguientes afirmaciones son ciertas o falsas, utilizando la definición de orden de complejidad O.

a) Si $f(n) \in O(n^2)$ y $g(n) \in O(n^3)$, entonces $f(n) + g(n) \in O(n^2)$.

b) $O(2^n+3^n) = O(3^n)$.

Solución:

- a) Esta afirmación es falsa, como se muestra en el siguiente contraejemplo. Si tomamos $f(n) = n^2 \in O(n^2)$ y $g(n) = n^3 \in O(n^3)$, pero $n^2 + n^3 \notin O(n^2)$.

  Demostramos esta última afirmación por reducción al absurdo. Supongamos que existen $n_0 \in \mathrm{N}$ y $c \in \mathrm{R}^+$ tales que para todo $n \geq n_0$

$$n^2 + n^3 \leq cn^2 \Leftrightarrow n^3 \leq (c-1)n^2$$

$$\Leftrightarrow \frac{n^3}{n^2} \leq c-1$$

$$\Leftrightarrow n \leq c-1$$

Asintóticas, II

Como la última desigualdad es falsa porque no puede ocurrir que todo $n \geq n_0$ sea menor o igual que la constante $c - 1$, hemos llegado a una contradicción.

- b) La afirmación es cierta. Como se trata de una igualdad entre conjuntos, para demostrarla hay que probar que cualquier función perteneciente a uno de los conjuntos también pertenece al otro.

  $\subseteq$ Si $h \in O(2^n+3^n)$, entonces existen $n_0 \in \mathrm{N}$ y $c \in \mathrm{R}^+$ tales que para todo $n \geq n_0$

$$h(n) \leq c_0(2^n+3^n) \leq c_0(3^n+3^n) = 2c_0 3^n,$$

  y por lo tanto $h \in O(3^n)$.

  $\supseteq$ Si $h \in O(3^n)$, entonces existen $n_1 \in \mathrm{N}$ y $c_1 \in \mathrm{R}^+$ tales que para todo $n \geq n_1$

$$h(n) \leq c_1 3^n \leq c_1(2^n+3^n),$$

  por lo que $h \in O(2^n+3^n)$.

Algoritmo secuencial, I

- Qué valor devuelve la siguiente función? Proporcionar la respuesta como función en términos de *n* y calcular su complejidad.

```
function valor(n:nat) return r:nat
var i,j,k:nat
  r=0
  for i=1 to n-1
    for j=i+1 to n
      for k=1 to j
        r=r+1
      fin
    fin
  fin
fin
```

Solución:

- Si escribimos las sumas sucesivas que realizan los tres ciclos anidados como fórmula matemática, se obtiene

$$\sum_{i=1}^{n-1}\sum_{j=i+1}^{n}\sum_{k=1}^{j}1$$

Algoritmo secuencial, II

- Para conocer el valor explícito de estas sumas en función de n, vamos a proceder de dentro hacia fuera, simplificando paso a paso cada una de estas tres sumas.

  En primer lugar, $j \geq i+1 \geq 1 + 1 = 2$ y es obvio que

$$\sum_{k=1}^{j} 1 = j$$

  En segundo lugar, observamos que como $1 \leq i \leq n-1$, el rango de j entre i+1 y n no es vacío, y si recordamos la fórmula de la sumatoria

$$\sum_{i=1}^{k} i = \frac{k(k+1)}{2}$$

  para calcular

$$\sum_{j=i+1}^{n}\sum_{k=1}^{j} 1 = \sum_{j=i+1}^{n} j = \sum_{j=1}^{n} j - \sum_{j=1}^{i} j = \frac{n(n+1)}{2} - \frac{i(i+1)}{2}$$

  En tercer lugar, recordemos otra fórmula clásica de las sumatorias

Algoritmo secuencial, III

$$\sum_{i=1}^{k} i^2 = \frac{k(k+1)(2k+1)}{6}$$

- para calcular, para n ≥ 2,

$$\sum_{i=1}^{n-1}\sum_{j=i+1}^{n}\sum_{k=1}^{j} 1 = \sum_{i=1}^{n-1}\left(\frac{n(n+1)}{2} - \frac{i(i+1)}{2}\right)$$

$$= (n-1)\frac{n(n+1)}{2} - \sum_{i=1}^{n-1}\frac{i(i+1)}{2}$$

$$= \frac{(n-1)n(n+1)}{2} - \frac{1}{2}\sum_{i=1}^{n-1}(i^2+i)$$

$$= \frac{(n-1)n(n+1)}{2} - \frac{1}{2}\sum_{i=1}^{n-1} i^2 - \frac{1}{2}\sum_{i=1}^{n-1} i$$

Algoritmo secuencial, IV

$$= \frac{(n-1)n(n+1)}{2} - \frac{1}{2}\frac{(n-1)n(2n-1)}{6} - \frac{1}{2}\frac{(n-1)n}{2}$$

$$= \frac{(n-1)n}{2}\left(n+1-\frac{2n-1}{6}-\frac{1}{2}\right)$$

$$= \frac{(n-1)n}{2}\frac{4(n+1)}{6}$$

$$= \frac{(n-1)n(n+1)}{3}$$

$$= \frac{n(n^2-1)}{3}$$

$$\in \Theta(n^3)$$

- Finalmente observamos que en los casos n=0 y n=1, la suma anterior vale 0 porque el rango de i entre 1 y n-1 es vacío. Este resultado coincide con el devuelto por la función valor (en esos casos no se entra en el ciclo al ser el rango vacío.

Otro ejemplo, I

- Dado el siguiente programa

```
i=1
j=1
while i ≤ n and j ≤ n
  if a[i, j+1] < a[i+1, j]
      j=j+1
  else
      i=i+1
  endif
endwhile
```

- Calcular su Ω, O y Θ
- Cada vez que se pasa por el ciclo while, se realiza la misma cantidad de operaciones: la comprobación del if, y la actualización de una variable (i o j).
- Por lo tanto, el tiempo de ejecución estará determinado por el número de veces que se entra en el ciclo.

Otro ejemplo, II

- Cuando la condición del if es siempre verdadera (o siempre falsa) se incrementa siempre la misma variable, con lo que se saldrá del ciclo cuando dicha variable tome el valor n+1.
- En este caso, se entrará n veces en el ciclo while y el número de instrucciones ejecutadas será de 2 de las asignaciones iniciales, n+1 comprobaciones del ciclo, 2n instrucciones de los n pasos por el interior del ciclo.
- El número total de instrucciones ejecutadas será de 3n+3 y el algoritmo tiene $\Omega(n)$.
- Para obtener el orden, como el número máximo de pasos por el ciclo se da cuando se llega a i = n y j = n, y no entra al ciclo en el siguiente pasos.
- Por lo tanto, se tendrá un número de instrucciones ejecutadas: 2 de las asignaciones iniciales, 2n+1 de las comprobaciones del ciclo, 4n de los 2n pasos por el cuerpo del ciclo.
- El número total de instrucciones es 6n+3 y el algoritmo tiene $O(n)$.
- Como $\Omega = O$, entonces el orden promedio será $\Theta(n)$.