Análisis y Diseño de Algoritmos

Fundamentos del Análisis de la Eficiencia de los Algoritmos

Prof.: Dr. Pedro A. Rodríguez[1]

[1]Departamento de Sistemas de Información
Departamento de Ciencias de la Computación y TI
Universidad del Bio-Bio

Outline

Introducción

¿Por qué estudiar los algoritmos?

- Los algoritmos (la algoritmia) son el corazón de la ciencia de la computación.
- Diseñar nuevos algoritmos y hacer el análisis (a priori, a posteriori).
- El análisis teórico (a priori) nos indica si el algoritmo es tratable o viable.
- Elegir entre un algoritmo exacto o aproximado.
- Elegir entre un algoritmo secuencial o paralelo.
- Diferentes algoritmos para resolver un mismo problema.

- Ej. Dado un arreglo A de tamaño n, ordenar los elementos de A de menora mayor

Algorithm 1 SortBasico(A, n)

```
1: Input: A, n
2: Output: A ordenado de menor a mayor
3: for i = 1 to n − 1 do
4:     for j = i + 1 to n do
5:         if a[i] > a[j] then
6:             Intercambia(A, i, j)
7:         end if
8:     end for
9: end for
```

Algorithm 2 InsertSort(A, n)

```
1: Input: A, n
2: Output: A ordenado de menor a mayor
3: for j = 1 to n do
4:     key = a[j]
5:     i = j − 1
6:     while i ≥ 0 and a[i] > key do
7:         a[i + 1] = a[i]
8:         i = i − 1
9:     end while
10:    a[i + 1] = key
11: end for
```

Algorithm 3 HeapSort(A, n)

1: Input: A, n
2: Output: A ordenado de menor a mayor
3: BuildHeap(A)
4: **for** $i = n - 1$ to 0 **do**
5: Intercambia($A, 0, i$)
6: Heapify($A, 0, i - 1$)
7: **end for**

Algorithm 4 BuildHeap(A, n)

1: Input: A, n
2: Output: Un heap en A
3: **for** $i = n/2; i >= 0; i--$ **do**
4: Heapify($A, i, n-1$)
5: **end for**

Algorithm 5 Heapify(A, i, j)

```
1: Input: A, i, j
2: Output: Heapify
3: if (2 * i + 1) ≤ j) then
4:    if (2 * i + 2) ≤ j) then
5:       if (a[2 * i + 2] ≥ a[2 * i + 1]) then
6:          k = 2 * i + 2
7:       else
8:          k = 2 * i + 1
9:       end if
10:   else
11:      k = 2 * i + 1
12:   end if
13:   if a[i] < a[k] then
14:      Intercambia(A, i, k);
15:      Heapify(A, k, j)
16:   end if
17: end if
```

Medidas de la eficiencia

- Eficiencia temporal. ¿Qué medir?
- Eficiencia espacial. Cantidad de memoria adicional requerido además de la memoria requerida para la entrada y la salida.
- Medida del tamaño de la entrada.
  - Tamaño de n. Número de datos de entrada (ej. algoritmo de ordenamiento).
  - El número de bits de n. Si la entrada es un sólo valor (ej. calcular el factorial de n).

    $b = \lfloor log_2 n \rfloor + 1$

Medidas de la eficiencia

Unidades medidas para el tiempo de ejecución

- Factores que interfieren en el tiempo de ejecución: velocidad del computador, calidad de la implementación del algoritmo (programa), compilador, etc.
- Usar una métrica que no dependa de ninguno de esos factores.
- Posible métrica, contar el número de veces que cada operación se ejecuta. Esto es muy difícil y muchas veces innecesario.
- Mejor alternativa es identificar la operación más importante del algoritmo (operación relevante / básica).

Medidas de la eficiencia

La operación relevante

- Es aquella que contribuye con la mayor parte del tiempo total de ejecución.
- La operación relevante, en la cual se basa el análisis, de alguna forma está relacionada con el tipo de problema que se intenta resolver.
- El propósito es calcular el número de veces que se ejecuta la operación relevante.
- Generalmente la operación relevante se ubica en el loop más interno.
- El marco establecido para el análisis de la eficiencia de un algoritmo sugiere medirlo contando el número de veces que se ejecuta la operación relevante con entradas de tamaño **n**.

- Sea C_{op} el tiempo de ejecución de la operación relevante de un algoritmo sobre un computador.
- Sea $C(n)$ el número de veces que la operación relevante se ejecuta.
- El tiempo de ejecución estimado $T(n)$ de un programa que implementa el algoritmo es:

$$T(n) \approx C_{op} * C(n)$$

- C_{op} y $C(n)$ son valores aproximados.
- Sin embargo a nosotros nos interesa conocer $C(n)$.

Valores de varias funciones importantes usadas en el análisis de algoritmos

n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
10	3.3	10^1	$3.3 \cdot 10^1$	10^2	10^3	10^3	$3.6 \cdot 10^6$
10^2	6.6	10^2	$6.6 \cdot 10^2$	10^4	10^6	$1.3 \cdot 10^{30}$	$9.3 \cdot 10^{157}$
10^3	10	10^3	$1.0 \cdot 10^4$	10^6	10^9		
10^4	13	10^4	$1.3 \cdot 10^5$	10^8	10^{12}		
10^5	17	10^5	$1.7 \cdot 10^6$	10^{10}	10^{15}		
10^6	20	10^6	$2.0 \cdot 10^7$	10^{12}	10^{18}		

- ¿Con qué algoritmos asocian a cada una de estas funciones?

Medidas de la eficiencia

Órdenes de crecimiento

- Es importante conocer el orden de crecimiento de una función para valores grandes de n.
- En el caso de las funciones 2^n y $n!$, en ambas sus valores crecen de forma astronómica a medida que el valor de n crece.
- Por ejemplo, si un computador moderno es capaz de ejecutar 10^{12} operaciones (1 billón aproximadamente) por segundo, entonces podría demorar $4*10^{10}$ años en ejecutar 2^{100} operaciones ($2^{100} \approx 1,2676506 * 10^{30}$ operaciones).
- Sin embargo, 100! toma aun mucho más tiempo que 4,5 mil millones de años (100! = $9{,}332622*10^{157}$, $\approx 2{,}959355*10^{138}$ años).

- En algunos casos el tiempo de ejecución de un algoritmo no solo depende del tamaño de su entrada sino que también de las características específicas de cada entrada en particular.

Algorithm 6 SequentialSearch(int *A*[], int k)

```
 1: Input: A: arreglo de tamaño n; k: clave a buscar.
 2: Output: retorna posición de i en A.
 3: i = 0;
 4: while (i < n) and (A[i] ≠ k) do
 5:     i = i + 1;
 6: end while
 7: if i < n then
 8:     return i;
 9: else
10:     return -1;
11: end if
```

- El peor caso se da cuando el algoritmo se ejecuta por el tiempo más largo de entre todas las posibles entradas de tamaño n.
- Aquí hablamos de la peor entrada posible para el algoritmo.
- En el ejemplo de la búsqueda secuencial, el peor caso se da cuando el elemento no existe en el arreglo o cuando éste es el último.
- En este caso el número de comparaciones es mayor. $C_{worst}(n)$ = n.
- $C_{worst}(n) = \max_{|A|=n} T(A)$; A: entrada; T: tiempo.
- Esto nos permite encontrar una cota superior para el tiempo de ejecución el cual no excederá a $C_{worst}(n)$, para cualquier entrada de tamaño n.

- El mejor caso se da cuando el algoritmo se ejecuta por el tiempo más pequeño de entre todas las posibles entradas de tamaño n.
- Aquí hablamos de la mejor entrada posible para el algoritmo.
- En el ejemplo de la búsqueda secuencial, el mejor caso se da cuando el elemento buscado es el primero en el arreglo.
- En este caso el número de comparaciones es menor. $C_{best}(n) = 1$.
- $C_{best}(n) = \min_{|A|=n} T(A)$; A: entrada; T: tiempo
- Esto nos permite encontrar una cota inferior para el tiempo de ejecución el cual no es menor a $C_{best}(n)$, para cualquier entrada de tamaño n.

- Ni el peor caso y tampoco el mejor caso entregan la información necesaria sobre el comportamiento real del algoritmo,
- El análisis del caso promedio es recomendado cuando un algoritmo tiene diferentes tiempos de ejecución para el mismo tamaño de la entrada.
- Se deben construir algunos supuestos sobre las posibles entradas de tamaño n.

- En el caso de la búsqueda secuencial tenemos dos supuestos estándar:
  - La probabilidad de una búsqueda exitosa es igual a p ($0 \leq p \leq 1$).
  - La probabilidad de éxito del primer acierto en la i-ésima posición del arreglo es la misma para todo i.
- En el caso de la búsqueda exitosa la probabilidad del primer acierto es p/n, mientras que en la búsqueda no exitosa la probabilidad es (1-p).

$$C_{avg}(n) = [1\frac{p}{n} + 2\frac{p}{n} + ... + i\frac{p}{n} + ...+ n\frac{p}{n}] + n(1-p)$$

- El número promedio de comparaciones está dado por $C_{avg}(n)$

$$C_{avg}(n) = [1\frac{p}{n} + 2\frac{p}{n} + ... + i\frac{p}{n} + ...+ n\frac{p}{n}] + n(1-p)$$

$$= \frac{p}{n}[1 + 2 + ... + i + ...+ n] + n(1-p)$$

$$= \frac{p}{n}[\sum_{i=1}^{n} i] + n(1-p)$$

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

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

- Los dos casos están dados por:

$$p = \begin{cases} 1 & \text{búsqueda exitosa, número de comparaciones: } \frac{(n+1)}{2} \\ 0 & \text{búsqueda no exitosa, número de comparaciones: } n \end{cases}$$

- El estudio del caso promedio es mucho más difícil que el peor y el mejor de los casos.

- Distribución de la probabilidad de las entradas posibles se asume como el valor esperado del conteo de la operación relevante:

$$\text{Caso promedio} = \sum_{|A| \leq n} Pr(A)T(A)$$

- El ámbito del análisis de la eficiencia se concentra en el orden de crecimiento del conteo de la operación relevante del algoritmo.
- Para comparar y “rankear” tales órdenes de crecimiento, en ciencias de la computación se usan tres notaciones:
  - O (o grande).
  - Ω (omega grande).
  - Θ (teta grande).
- Informalmente, $O(g(n))$ es el conjunto de todas las funciones con un orden de crecimiento más bajo o el mismo que g(n).
- Por ejemplo: $n \in O(n^2)$; $100n + 5 \in O(n^2)$; $n^3 \notin O(n^2)$.

- Definición: una función $f(n)$ es $O(g(n))$, si:

  $\exists\ c > 0$, $n_0 > 0$, tal que $\forall\ n \geq n_0$, $f(n) \leq cg(n)$; $[f(n) \preceq g(n)]$

- En este caso podemos decir que: $f(n)$ es $O(g(n))$; o $f(n) = O(g(n))$; o $f(n) \in O(g(n))$.

- Decimos entonces que $f(n)$ está acotado por arriba (cota superior) por una constante múltiplo de $g(n)$, $\forall$ n grande.

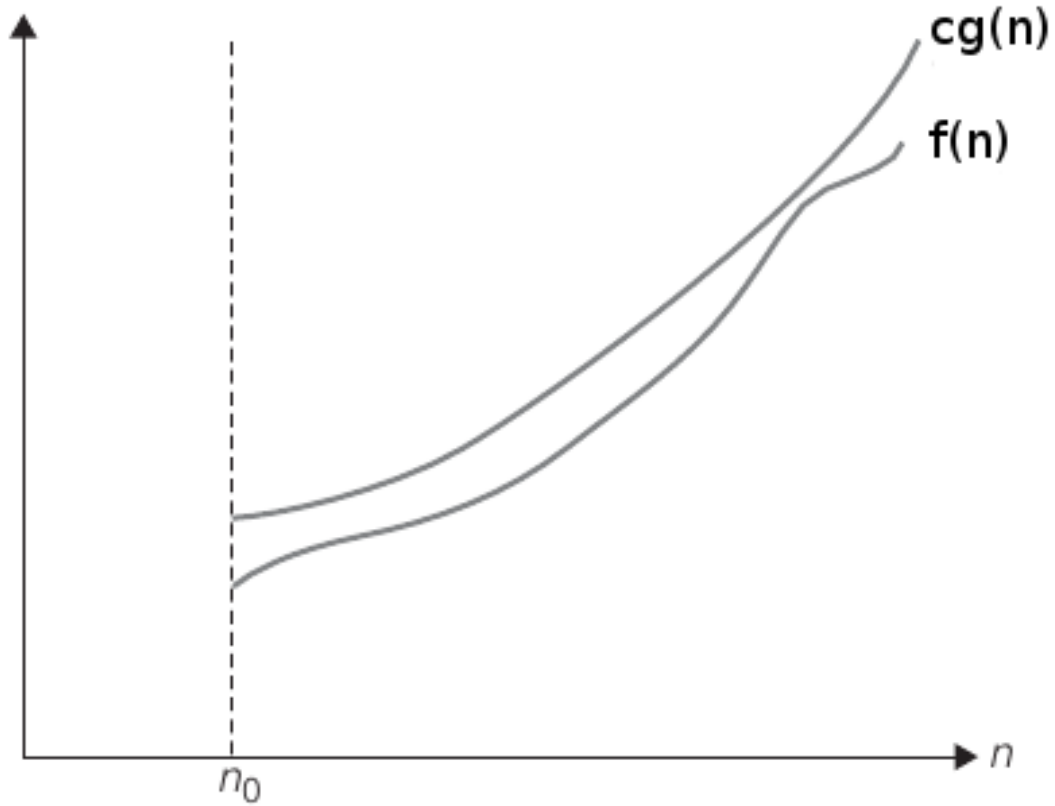
- Por ejemplo: sea f(n) = 100n + 5 $\in O(n^2)$:

  100n + 5 $\leq$ 100n + n ($\forall\ n \geq 5$)

  101n $\leq 101n^2$, entonces las constantes c y n_0 son: c = 101, n_0 = 5.

  o, 100n + 5 $\leq$ 100n + 5n, ($\forall$ n $\geq$ 1), donde c = 105, n_0 = 1.

Curvas de las funciones f(n) $\leq$ cg(n)



- Definición: una función $f(n)$ es $\Omega(g(n))$, si:

  $\exists\ c > 0$, $n_0 > 0$, tal que $\forall\ n \geq n_0$, $f(n) \geq cg(n)$; $[f(n) \succeq g(n)]$

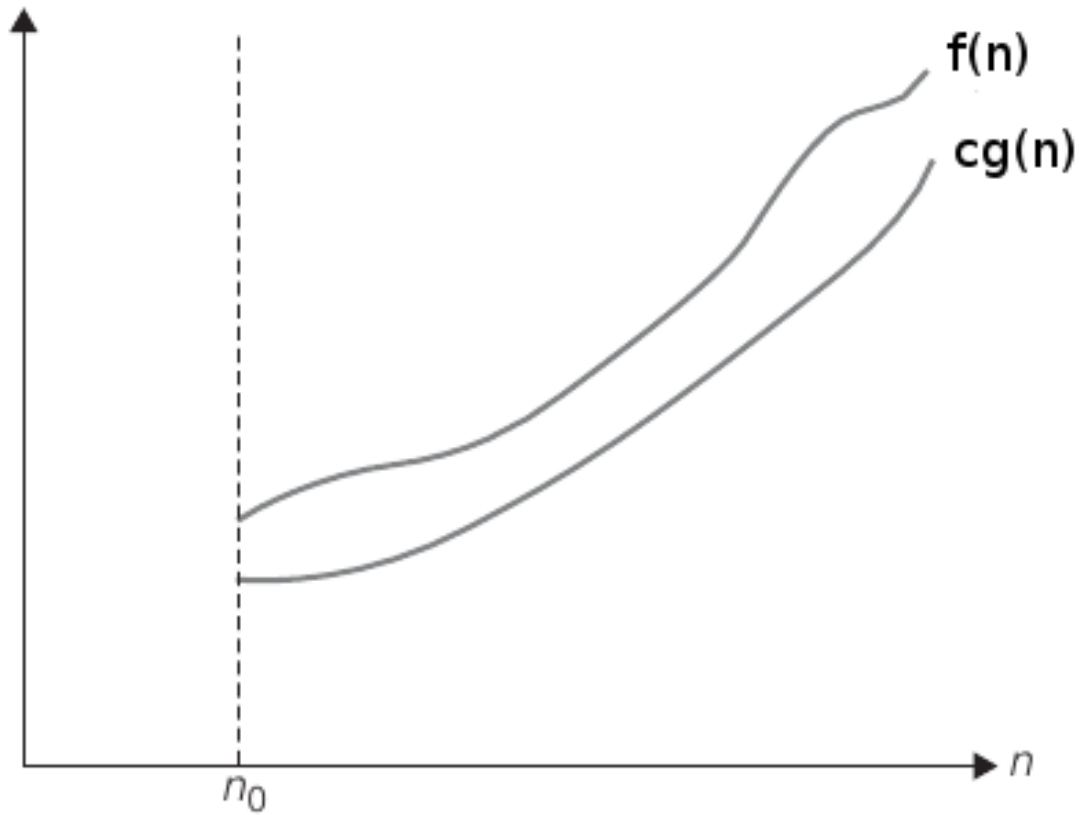
- En este caso podemos decir que: $f(n)$ es $\Omega(g(n))$; o $f(n) = \Omega(g(n))$; o $f(n) \in \Omega(g(n))$.

- Decimos entonces que $f(n)$ está acotado por abajo (cota inferior) por una constante múltiplo de $g(n)$, $\forall$ n grande.

- Por ejemplo: $n^3 \in \Omega(n^2)$, $n^3 \geq \Omega(n^2)$, $\forall$ n ≥ 0, con c = 1, $n_0 = 0$.

- Ω también corresponde a la complejidad del problema (ej. el problema de ordenamiento es $\Omega(nlogn)$).

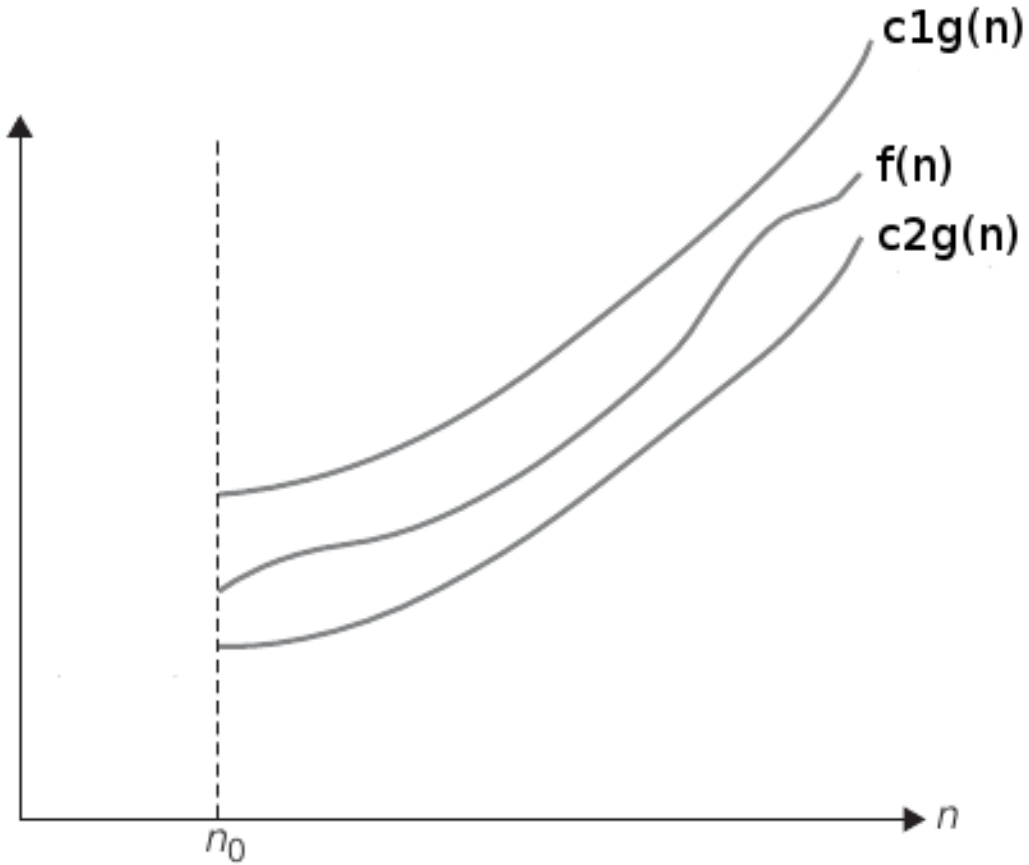
Curvas de las funciones f(n) $\geq$ cg(n)



- Definición: una función $f(n)$ es $\Theta(g(n))$, si es $O(g(n))$ y $\Omega(g(n))$.
- La función f(n) está acotada por arriba y por abajo por alguna constante positiva múltiplo de g(n), $\forall$ n grande.
- Es decir, si existen algunas constantes positivas c_1, c_2 y n_0, tal que:

  $c_2 g(n) \leq f(n) \leq c_1 g(n), \forall\ n \geq n_0$.
- Por ejemplo, el algoritmo de sorting MergeSort es $\Theta(nlogn)$.
- Probemos que f(n) = $\frac{1}{2} n(n-1)$ es $\Theta(n^2)$.

Curvas de las funciones $c_2 g(n) \leq f(n) \leq c_1 g(n)$,



- Probar que $f(n) = \frac{1}{2}n(n-1)$ es $\Theta(n^2)$.

$f(n) = \frac{1}{2}n^2 - \frac{1}{2}n \leq \frac{1}{2}n^2$
$\forall\ n \geq 0$
$c_1 = \frac{1}{2}$

$f(n) = \frac{1}{2}n^2 - \frac{1}{2}n \geq \frac{1}{2}n^2 - \frac{1}{4}n^2$
$f(n) \geq \frac{1}{4}n^2$
$\forall\ n \geq 2$
$c_2 = \frac{1}{4}$

$c_1 = \frac{1}{2}$, $c_2 = \frac{1}{4}$, $n_0 = 2$

- Regla del máximo: Sean f, g: $\mathbb{N} \rightarrow \mathbb{R}$ dos funciones arbitrarias de los números naturales en los reales.
- La regla del máximo dice que si T(n) = $t_1(n)$ + $t_2(n)$, siendo t_1 = O(f(n)) y

  t_2 = O(g(n)), T(n) = O(f(n) + g(n)) = O(max(f(n), g(n)))
- Por ejemplo: O(n^2 + n^3 + $nlogn$) = O(max(n^2, n^3, $nlogn$)) = $O(n^3)$

Notaciones Asintóticas y Clases de Eficiencia Básicas

o chica y ω (omega chica)

- Definición: una función $f(n)$ es $o(g(n))$ (o chica de g(n)), si:

  $f(n) < cg(n)$; $[f(n) \prec g(n)]$, $f(n)$ es menor estricto de g(n).

- Definición: una función $f(n)$ es $\omega(g(n))$ (omega chica de g(n)), si:

  $f(n) > cg(n)$; $[f(n) \succ g(n)]$, $f(n)$ es mayor estricto de g(n).

- ¿Cómo comparar el orden de crecimiento entre dos funciones?
- Calculando el límite de la razón de ambas funciones. Sean f, g: $\mathbb{N} \to \mathbb{R}$, entonces podemos distinguir tres casos importantes:
- Regla de L'Hopital: $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{f'(n)}{g'(n)}$

(1) $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \Rightarrow$ f(n) = $o(g(n))$, y $f(n) \notin \Theta(g(n))$.

(2) $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty \Rightarrow$ f(n) = $\omega(g(n))$, y $f(n) \notin \Theta(g(n))$.

(3) $\lim_{n\to\infty} \frac{f(n)}{g(n)} =$ c, donde c $\in \mathbb{R} \Rightarrow$ f(n) = $\Theta(g(n))$