Analisis y diseño de algoritmos

Juan Gutiérrez

September 2019

1 InsertionSort: correctitud y análisis

Resuelve el problema de ordenamiento.

Entrada: Secuencia $\langle a_1, a_2, \ldots, a_n \rangle$

Salida: Permutacion $\langle a'_1, a'_2, \ldots, a'_n \rangle$ tal que $a'_1 \leq a'_2 \leq \ldots \leq a'_n$

```
INSERTION-SORT(A)
for j = 2 to A.length
    key = A[j]
    // Insert A[j] into the sorted sequence A[1..j-1].
    i = j - 1
    while i > 0 and A[i] > key
        A[i + 1] = A[i]
        i = i - 1
    A[i + 1] = key
```

Figure 1: Tomada del libro Cormen, Introduction to Algorithms

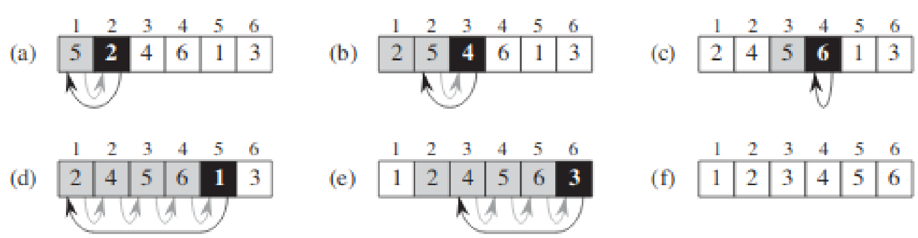

Figure 2: Tomada del libro Cormen, Introduction to Algorithms

1.1 Análisis de la correctitud

Invariante: Al inicio de cada iteracion del for de las lineas 1–8, el subarreglo $A[1..j-1]$ consiste en los elementos de $A[1..j-1]$ pero ordenados de manera no descendente.

Probaremos que la invariante es cierta.

Inicialización: La invariante es cierta antes de la primera iteración. Ya que, cuando $j = 2$, tenemos que $A[1..j-1] = A[1]$ consiste en el mismo $A[1]$ ordenado.

Manuntención: Asumimos que la invariante se cumple al inicio de la j-ésima iteración. Sabemos que $A[1..j]$ está ordenado. Debido al bucle while (lineas 5–7), el elemento $A[j+1]$ es insertado en la posición $i+1$ de manera tal que $A[1..j+1]$ queda ordenado. (Obs: una demostracion completa deberia probar que el elemento es insertado adecuadamente, con una invariante nueva en el bucle while, ver más adelante).

Terminación: Al inicio de la último iteración, tenemos que $j = n+1$. Luego $A[1..j-1] = A[1..n]$ está ordenado.

Como fue mencionado anteriormente, una demostración más completa debería probar con otra invariante que el bucle while hace lo pedido. Las invariantes que permiten demostrar esto son.

Sea A' el arreglo A al inicio del bucle while (líneas 5–7). Entonces, al inicio de cada iteración del bucle while, se cumple que

- $key \leq A[i+2..j]$
- $A[i+2..j] = A'[i+1..j-1]$
- $A[1..i] = A'[1..i]$

Queda como ejercicio al lector demostrar estas invariantes. Note que la finalización de estas invariantes implica que $A[1..i] \leq key \leq A[i+2..j]$, por lo tanto la línea 8 garantiza que el arreglo $A[1..j]$ está ordenado, como queríamos.

1.2 Análisis del tiempo de ejecución

Al hacer análisis de tiempo de ejecución, siempre supondremos que el algoritmo es ejecutado bajo el modelo RAM, es decir, las instrucciones son ejecutadas una tras otra sin paralelismo. De esta manera, cada instruccion atómica, como sumar, restar, multiplicar, asignar, etc, toma tiempo constante.

Sea $T(n)$ el tiempo de ejecución total del algoritmo. Para cada $j = 2 \ldots n$, sea t_j el número de veces que el while de la linea 5 es ejecutado. Para cada $i = 1 \ldots 8$, sea c_i el tiempo unitario que toma ejecutar la línea i. Note que este tiempo es constante.

INSERTION-SORT(A)	*cost*	*times*
for $j = 2$ **to** $A.length$	c_1	n
$key = A[j]$	c_2	$n-1$
// Insert $A[j]$ into the sorted sequence $A[1 \,..\, j-1]$.	0	$n-1$
$i = j - 1$	c_4	$n-1$
while $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
$A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
$i = i - 1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
$A[i+1] = key$	c_8	$n-1$

Figure 3: Tomada del libro Cormen, Introduction to Algorithms

Luego,

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1) \\ &= (c_5 + c_6 + c_7) \sum_{j=2}^{n} t_j + (c_1 + c_2 + c_4 + c_8 - c_6 - c_7)n + (c_7 + c_6 - c_2 - c_4 - c_8) \quad (1) \end{aligned}$$

Note que $t_j \leq j$. Por lo tanto, por (1),

$$\begin{aligned} T(n) &\leq (c_5 + c_6 + c_7) \sum_{j=2}^{n} j + (c_1 + c_2 + c_4 + c_8 - c_6 - c_7)n + (c_7 + c_6 - c_2 - c_4 - c_8) \\ &= (c_5 + c_6 + c_7)(\frac{n(n-1)}{2} - 1) + (c_1 + c_2 + c_4 + c_8 - c_6 - c_7)n + (c_7 + c_6 - c_2 - c_4 - c_8) \\ &= \frac{1}{2} n^2 (c_5 + c_6 + c_7) + (c_1 + c_2 + c_4 + c_8 - c_5/2 - 3c_6/2 - 3c_7/2)n + (c_7 - c_5 - c_2 - c_4 - c_8). \end{aligned}$$

Sea $a = (c_5 + c_6 + c_7)/2, b = c_1 + c_2 + c_4 + c_8 - c_5/2 - 3c_6/2 - 3c_7/2$ y $c = c_7 - c_5 - c_2 - c_4 - c_8$. La ecuación anterior prueba el siguiente teorema.

Teorema 1.1. *Para cualquier $n \geq 1$, $T(n) \leq an^2 + bn + c$, para algunas constantes a, b y c, con $a > 0$.*

Note que $t_j \geq 1$. Por lo tanto, por (1),

$$\begin{aligned} T(n) &\geq (c_5 + c_6 + c_7) \sum_{j=2}^{n} 1 + (c_1 + c_2 + c_4 + c_8 - c_6 - c_7)n + (c_7 + c_6 - c_2 - c_4 - c_8) \\ &= (c_1 + c_2 + c_4 + c_8 + c_5)n + (-c_5 - c_2 - c_4 - c_8) \end{aligned}$$

Sea $a = (c_1 + c_2 + c_4 + c_8 + c_5)/2$ y $b = (-c_5 - c_2 - c_4 - c_8)$. La ecuación anterior prueba el siguiente teorema.

Teorema 1.2. *Para cualquier $n \geq 1$, $T(n) \geq an + b$, para algunas constantes a y b con $a > 0$.*

Note que si $A[1..n]$ ya está ordenado de manera creciente, entonces t_j=1 para todo j. También, si $A[1..n]$ está ordenado de manera decreciente, entonces $t_j = j$. Podemos entonces deducir que. existen entradas que hacen que las cotas de los teoremas anteriores sean justas.

1.3 Análisis de peor caso y caso medio

Nos interesa el peor caso: el mayor tiempo de ejecución para cualquier entrada de tamaño n. Con eso garantizamos que el algoritmo no va a tomar más tiempo. Muchas veces el caso medio es igual de malo (en el insertion sort tenemos que $t_j = j/2$ en el caso medio). Veremos mas adelante el concepto de tiempo de ejecución esperado en el análisis del quicksort.

Observación 1.1. *En general, el tamaño de la entrada depende del problema. Por ejemplo, en un problema de ordenación, el tamaño de la entrada es el número de elementos a ordenar. Si el problema consiste en multiplicar dos numeros, el tamaño de entrada es el numero de bits usado para representar estos números. Si el problema tiene como entrada un grafo, el tamaño de la entrada es el número de vertices y número de aristas de dicho grafo.*

2 Crecimiento de funciones

No es necesario tener una precisión exacta del tiempo de ejecución de un algoritmo. Para entradas grandes, ya no importan las constantes multiplicativas ni los términos de menor orden. Nos interesa la eficiencia asintótica de los algoritmos, osea cómo se comporta la función en el límite.

2.1 Notación O

Dada una función $g(n)$, definimos $O(g(n))$ según

$$\begin{aligned} O(g(n)) = \{f(n) : &\text{ existen constantes positivas } c, n_0 \\ &\text{tales que } 0 \leq f(n) \leq cg(n) \text{ para todo } n \geq n_0\} \end{aligned}$$

Ejemplo 2.1. *Probar que $n^2 + 10n + 2 = O(n^2)$*

(Borrador: $n^2 + 10n + 2 \leq cn^2$. Probamos con $c = 3$.)

Proof. Note que para $n \geq 10$, tenemos que $10n \leq n^2$ Luego $n^2 + 10n + 2 \leq n^2 + n^2 + n^2 = 3n^2$. Portanto, Concluimos que $n^2 + 10n + 2 = O(n^2)$. □

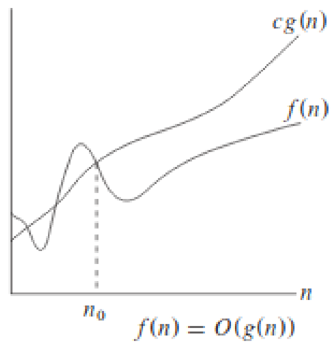

Figure 4: Tomada del libro Cormen, Introduction to Algorithms

Ejemplo 2.2. *Probar* $n^2/2 + 3n = O(n^2)$

(Borrador: $n^2/2 + 3n \leq cn^2$. Probamos con $c = 1$ tendria $3n \leq n^2/2$. Esto se cumple si n es mayor o igual a 6.)

Proof. Note que para $n \geq 6$, tenemos que $6n \leq n^2$. Luego $3n \leq n^2/2$.

Portanto,

$$0 \leq n^2/2 + 3n \leq n^2/2 + n^2/2 = n^2.$$

Concluimos que $n^2/2 + 3n = O(n^2)$ (ya que $0 \leq n^2/2 + 3n \leq cn^2$ para $n \geq n_0$, con $n_0 = 6$ y $c = 1$). □

Ejemplo 2.3. *Probar* $n/100$ *no es* $O(1)$.

Proof. Suponga por contradicción que $n/100 = O(1)$. Entonces existen constantes $n_0, c > 0$ tales que $\frac{n}{100} \leq c$ para todo $n \geq n_0$. Como $n_0 \geq n_0$ tenemos que $n_0/100 \leq c$ lo que implica que $n_0 \leq 100c$. Tome $n = 200c \geq n_0$ y note que $\frac{n}{100} = \frac{200c}{100} = 2c > c$, contradicción. □

Ejemplo 2.4. *Probar que* $an + b = O(n^2)$ *para todo* $a > 0$.

(Borrador: tomar $c = a + |b|$ y $n_0 = \max\{1, -b/a\}$.)

Proof. Note que, cuando $n \geq \max\{1, -b/a\}$, tenemos que $n \geq 1$ y que $n \geq -b/a$. Luego $an + b \geq a(-b/a) + b \geq 0$ y $an + b \leq an^2 + |b| \leq (a + |b|)n^2$. Por lo tanto

$$0 \leq an + b \leq (a + |b|)n^2$$

para todo $n \geq \max\{1, -b/a\}$. □

Observación 2.1. *Notación O sirve para acotar el peor caso del tiempo de ejecución de un algoritmo, y portanto también cada caso del algoritmo.*

2.2 Notación Ω

Dada una función $g(n)$, definimos $\Omega(g(n))$ según

$$\Omega(g(n)) = \{f(n) : \text{existen constantes positivas } c, n_0 \text{ tales que } 0 \leq cg(n) \leq f(n) \text{ para todo } n \geq n_0\}$$

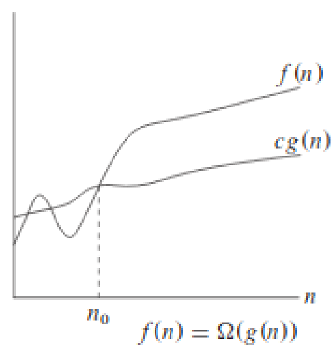

Figure 5: Tomada del libro Cormen, Introduction to Algorithms

Observación 2.2. Ω *sirve para acotar el mejor caso del algoritmo inferiormente, y por lo tanto cada caso inferiormente.*

Observación 2.3. *Si $T(n)$ es el tiempo de ejecución para el Insertion sort con una entrada de tamaño n, entonces $T(n) = O(n^2)$ y $T(n) = \Omega(n)$.*

Teorema 2.1. *$f = \Theta(g(n)$ si y solo si $f = O(g(n))$ y $f = \Omega(g(n))$*

2.3 Notación Θ

Dada una función $g(n)$, definimos $\Theta(g(n))$ según

$$\Theta(g(n)) = \{f(n) : \text{existen constantes positivas } c_1, c_2, n_0 \text{ tales que } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ para todo } n \geq n_0\}.$$

Como $\Theta(g(n))$ es un conjunto, podemos decir que $f(n) \in \Theta(g(n))$. También diremos que $f(n) = \Theta(g(n))$. Tambien se dice $f(n)$ *es* $\Theta(g(n))$.

Ejemplo 2.5. *Demostrar que $\frac{1}{2}n^2 - 3n = \Theta(n^2)$.*

(Borrador: $c_1 n^2 \leq \frac{1}{2}n^2 - 3n \leq c_2 n^2$. Entonces $c_1 \leq \frac{1}{2} - 3/n \leq c_2$. Entonces tomo $c_2 = 1/2$. Cuando $n = 7$, tengo $c_1 \leq 1/2 - 3/7 = 1/14$.)

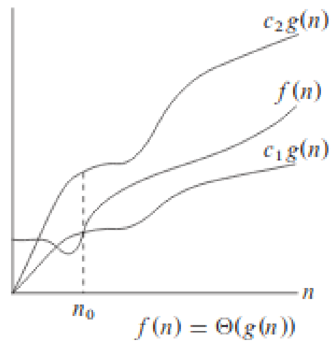

Figure 6: Tomada del libro Cormen, Introduction to Algorithms

Proof. Note que para $n \geq 7$, se cumple que

$$\frac{1}{2}n^2 - 3n = n^2(\frac{1}{2} - \frac{3}{n}) \geq \frac{n^2}{14},$$

y también se cumple que

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

Luego, para $c_1 = \frac{1}{14}, c_2 = \frac{1}{2}, n_0 = 7$, tenemos que

$$0 \leq c_1 n^2 \leq \frac{1}{2}n^2 - 3n \leq c_2 n^2$$

para todo $n \geq n_0$. Por lo tanto $\frac{1}{2}n^2 - 3n = \Theta(n^2)$. $\square$

Ejemplo 2.6. *Demostrar que* $6n^3 \neq \Theta(n^2)$.

Proof. Suponga por contradicción que $6n^3 = \Theta(n^2)$. Entonces existen constantes $c_1, c_2, n_0 > 0$ tales que $0 \leq c_1 n^2 \leq 6n^3 \leq c_2 n^2$. Luego, $6n \leq c_2$ para todo $n \geq n_0$. Contradicción pues c_2 es constante. $\square$

Ejercicio 2.1. $an^2 + bn + c = \Theta(n^2)$ *para todo* $a > 0$.

Proof. Cuando $n \geq \max\{\frac{2|b|}{a}, 2\sqrt{\frac{|c|}{a}}\}$ se cumple que

$$\frac{|b|}{n} \leq \frac{|b|}{\max\{\frac{2|b|}{a}, 2\sqrt{\frac{|c|}{a}}\}} \leq \frac{|b|}{2|b|/a} = a/2.$$

Luego

$$\frac{-a}{2} \leq b/n \leq a/2. \tag{2}$$

Analogamente,

$$\frac{|c|}{n^2} \leq \frac{|c|}{\max\{\frac{2|b|}{a}, 2\sqrt{\frac{|c|}{a}}\}} \leq \frac{|c|}{4|c|/a} = a/4.$$

Por lo tanto,

$$\frac{-a}{4} \leq c/n^2 \leq a/2. \tag{3}$$

De (1) y (2),

$$\frac{a}{4} \leq a + \frac{b}{n} + \frac{c}{n^2} \leq \frac{7a}{4}.$$

Es decir

$$0 \leq an^2 \leq an^2 + bn + c \leq \frac{7a}{4}n^2.$$

□

2.4 Notación o

Dada una función $g(n)$, definimos $o(g(n))$ según

$$o(g(n)) = \{f(n) : \text{para cada constante } c > 0$$

$$\text{existe una constante } n_0 \text{ tal que } 0 \leq f(n) < cg(n) \text{ para todo } n \geq n_0\}$$

Ejemplo 2.7. $2n = o(n^2)$

(Borrador. Quiero $2n < cn^2$. Entonces $n > 2/c$. Tomamos $n_0 = 1 + \frac{2}{c}$.)

Proof. Sea $c > 0$ una constante arbitrária. Note que, para todo $n \geq 1 + \frac{2}{c}$, tenemos que $c + 2 \leq nc$. Como n es positivo, $nc + 2n \leq cn^2$, por lo tanto $2n < cn^2$. □

Ejemplo 2.8. $2n^2 \neq o(n^2)$

Proof. Suponga por contradicción que $2n^2 = o(n^2)$. Tome $c = 1$, tendríamos que $2n^2 < n^2$, cuando n comienza en una constante n_0, lo cual es una contradicción. □

Observación 2.4. *$f(n) = o(g(n))$ si y solo si $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$.*

2.5 Notación ω

Dada una función $g(n)$, definimos $\omega(g(n))$ según

$$\omega(g(n)) = \{f(n) : \text{para cada constante } c > 0$$

$$\text{existe una constante } n_0 \text{ tal que } 0 \leq cg(n) < f(n) \text{ para todo } n \geq n_0\}$$

Observación 2.5. *$f(n) = \omega(g(n))$ si y solo si $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$.*

2.6 Comparaciones

Podemos definir comparaciones obvias entre las distintas notaciones.

Transitividad

- $f(n) = \Theta(g(n))$, $g(n) = \Theta(h(n))$, entonces $f(n) = \Theta(h(n))$
- $f(n) = O(g(n))$, $g(n) = O(h(n))$, entonces $f(n) = O(h(n))$
- $f(n) = \Omega(g(n))$, $g(n) = \Omega(h(n))$, entonces $f(n) = \Omega(h(n))$
- $f(n) = o(g(n))$, $g(n) = o(h(n))$, entonces $f(n) = o(h(n))$
- $f(n) = \omega(g(n))$, $g(n) = \omega(h(n))$, entonces $f(n) = \omega(h(n))$

Reflexividad

- $f(n) = \Theta(f(n))$
- $f(n) = O(f(n))$
- $f(n) = \Omega(f(n))$

Simetría

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

Simetría transpuesta

- $f(n) = O(g(n))$ si y solo si $g(n) = \Omega(f(n))$
- $f(n) = o(g(n))$ entonces $g(n) = \omega(f(n))$

Observación 2.6. *Existen funciones no comparables, por ejemplo n y $n^{1+\sin n}$.*