Introducción al Coste de Algoritmos

8 de abril de 2018

Coste de un Algoritmo

Analizar un algoritmo significa, en el contexto de este curso, estimar los recursos que requiere.

Aunque existen otros recursos, como el espacio de memoria, normalmente nos centraremos en analizar el **tiempo** de ejecución.

Dado que el comportamiento de un algoritmo puede variar mucho para cada posible entrada, necesitamos encontrar una forma de resumir su comportamiento de forma sencilla.

En general, el tiempo de ejecución de un algoritmo crece con el tamaño de su entrada. Por este motivo se suele representar el **tiempo** de ejecución de un algoritmo como **una función del tamaño de su entrada**.

Conceptos Básicos

El **tamaño de la entrada** $|x|$ se calcula de distintas maneras dependiendo del tipo de problema. Puede ser la longitud de un vector, el valor de un número, el número de bits necesarios para representar un número en base 2, o un par de números que reflejan dos dimensiones de los datos de un problema.

El **tiempo de ejecución** $T(x)$ de un algoritmo para una entrada particular x es el número de operaciones primitivas que realiza.

Analizaremos el tiempo de ejecución en **el caso peor**, porque es una cota superior del tiempo de ejecución para cualquier entrada.

$$T_{worst}(n) = max\{\, T(x) \mid x \in Input \wedge |x| = n\}$$

Para calcular el tiempo de ejecución **esperado** (i.e. **en el caso medio**) es preciso conocer la probabilidad de que ocurra cada entrada.

$$T_{average}(n) = \sum_{\{x \in Input \,\wedge\, |x| = n\}} Pr(x) \cdot T(x)$$

Conceptos Básicos

En lugar de calcular la función $T_{worst}(n)$ analizaremos únicamente su **orden de crecimiento**, es decir, tendremos en cuenta solamente el **término más significativo** de $T_{worst}(n)$, ignorando términos de menor orden y coeficientes constantes multiplicativos, que son relativamente insignificantes para valores grandes de n.

Notación asintótica en este curso utilizaremos las notaciones denominadas O **grande**, Ω y Θ.

La expresión $\mathbf{O(g(n))}$ denota el siguiente **conjunto de funciones**

$$O(g(n)) = \{f(n) : \mathbb{N} \to \mathbb{R}^+ \mid \text{existen } c > 0 \text{ } y \text{ } n_0 > 0 \\ \text{tales que } f(n) \leq c \cdot g(n) \text{ para todo } n \geq n_0\}$$

es decir, el conjunto de las funciones $f(n)$ para las cuales $g(n)$ es una cota asintótica superior.

Conceptos Básicos

La expresión $\mathbf{\Omega(g(n))}$ denota el siguiente **conjunto de funciones**

$$\Omega(g(n)) = \{f(n) : \mathbb{N} \to \mathbb{R}^{+} \mid \text{existen } c > 0 \text{ y } n_0 > 0 \\ \text{tales que } f(n) \geq c \cdot g(n) \ \text{ para todo } n \geq n_0\}$$

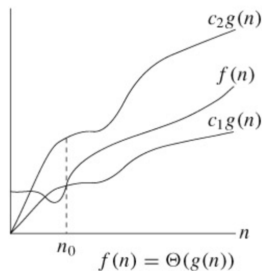
es decir, el conjunto de las funciones $f(n)$ para las cuales $g(n)$ es una cota asintótica inferior.

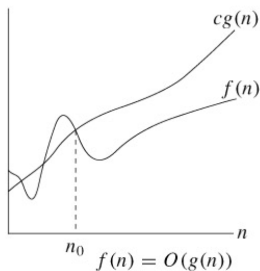
La expresión $\mathbf{\Theta(g(n))}$ denota el siguiente **conjunto de funciones**

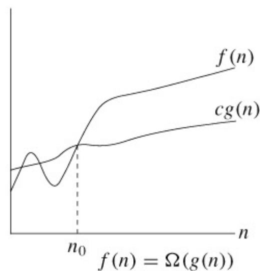
$$\Theta(g(n)) = \{f(n) : \mathbb{N} \to \mathbb{R}^{+} \mid \text{existen } c1 > 0, \ c2 > 0 \text{ y } n_0 > 0 \\ \text{tales que } c1 \cdot g(n) \leq f(n) \leq c \cdot g(n) \ \text{ para todo } n \geq n_0\}$$

es decir, el conjunto de las funciones $f(n)$ para las cuales $g(n)$ es una cota asintótica exacta.

Notación Asintótica







Conceptos Básicos

En general, diremos que un algoritmo A_1 **es más eficiente** que otro algoritmo A_2 si

- $T_{worst}^{A_1}(n) \in \Theta(g_1(n))$,
- $T_{worst}^{A_2}(n) \in \Theta(g_2(n))$ y
- $\Theta(g_1(n)) < \Theta(g_2(n))$

es decir, si el orden de crecimiento del tiempo de ejecución en el caso peor de A_1 es menor que el orden de crecimiento del tiempo de ejecución en el caso peor de A_2.

Por ejemplo, A_{merge_sort} es más eficiente que $A_{insertion_sort}$, porque

$A_{merge_sort} \in \Theta(n \log n)$,

$A_{insertion_sort} \in \Theta(n^2)$ y

$\Theta(nlog(n)) < \Theta(n^2)$.

Conceptos Básicos

Relaciones entre O, Ω y Θ

- $f_1 \in \Omega(f_2)$ si y sólo si $f_2 \in O(f_1)$
- $\Theta(f) = O(f) \bigcap \Omega(f)$

Reglas del límite

- Si $\lim_{n\to\infty} \frac{f_1}{f_2} = 0$, entonces $f_1 \in O(f_2)$.
- Si $\lim_{n\to\infty} \frac{f_1}{f_2} = \infty$, entonces $f_1 \in \Omega(f_2)$.
- Si $\lim_{n\to\infty} \frac{f_1}{f_2} = c$ donde $c \in \mathbb{R}$ es una constante $c > 0$, entonces $f_1 \in \Theta(f_2)$.

Dadas dos funciones f_1 y f_2, diremos que $\Theta(f_1) < \Theta(f_2)$ si

$$f_1 \in O(f_2) \wedge f_2 \notin O(f_1)$$

Conceptos Básicos

Sean C_1 y C_2 clases de funciones como $O(f), \Omega(f)$ o $\Theta(f)$. Definimos

- $C_1 + C_2 = \{f + g | f \in C_1 \wedge g \in C_2\}$
- $C_1 \cdot C_2 = \{f \cdot g | f \in C_1 \wedge g \in C_2\}$

Reglas de la suma

- $O(f_1) + O(f_2) = O(f_1 + f_2) = O(max(f_1, f_2))$
- $O(f_1) \cdot O(f_2) = O(f_1 \cdot f_2)$
- $\Theta(f_1) + \Theta(f_2) = \Theta(f_1 + f_2) = \Theta(max(f_1, f_2))$
- $\Theta(f_1) \cdot \Theta(f_2) = \Theta(f_1 \cdot f_2)$

Conceptos Básicos

Propiedades de O grande

- Reflexividad $f \in O(f)$
- Transitividad $f_1 \in O(f_2) \wedge f_2 \in O(f_3) \Rightarrow f_1 \in O(f_3)$
- Caracterización $f_1 \in O(f_2)$ si y sólo si $O(f_1) \subseteq O(f_2)$
- Suma $g_1 \in O(f_1) \wedge g_2 \in O(f_2) \Rightarrow g_1 + g_2 \in O(f_1 + f_2) = O(max(f_1, f_2)$
- Producto $g_1 \in O(f_1) \wedge g_2 \in O(f_2) \Rightarrow g_1 \cdot g_2 \in O(f_1 \cdot f_2)$
- Invarianza multiplicativa $O(f) = O(c \cdot f)$ para toda constante $c \in (R)^+$

Conceptos Básicos

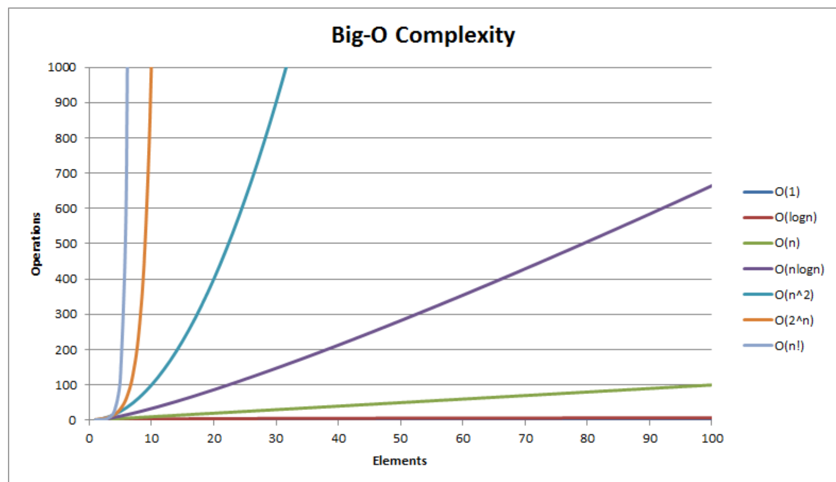
Propiedades de Θ

- Reflexividad $f \in \Theta(f)$
- Transitividad $f_1 \in \Theta(f_2) \wedge f_2 \in \Theta(f_3) \Rightarrow f_1 \in \Theta(f_3)$
- Simetría $f_1 \in \Theta(f_2)$ si y sólo si $f_2 \in \Theta(f_1)$ si y sólo si $\Theta(f_1) = \Theta(f_2)$
- Suma $g_1 \in \Theta(f_1) \wedge g_2 \in \Theta(f_2) \Rightarrow g_1 + g_2 \in \Theta(f_1 + f_2) = \Theta(max(f_1, f_2)$
- Producto $g_1 \in \Theta(f_1) \wedge g_2 \in \Theta(f_2) \Rightarrow g_1 \cdot g_2 \in \Theta(f_1 \cdot f_2)$
- Invarianza multiplicativa $\Theta(f) = \Theta(c \cdot f)$ para toda constante $c \in (R)^+$

Ordenes de Crecimiento Frequentes

- **Constante** $\Theta(1)$: Decidir si un número es par o impar.
- **Logarítmico** $\Theta(\log n)$: Búsqueda binaria.
- **Radical** $\Theta(\sqrt{n})$: Algoritmo sencillo para comprobar si un número es primo.
- **Lineal** $\Theta(n)$: Búsqueda secuencial en un vector.
- **Cuasilineal** $\Theta(n \log n)$: Ordenación eficiente de un vector.
- **Cuadrático** $\Theta(n^2)$: Suma de dos matrices cuadradas de tamaño $n \times n$.
- **Cúbico** $\Theta(n^3)$: Producto de dos matrices cuadradas de tamaño $n \times n$.
- **Polinómico** $\Theta(n^k)$ para $k \geq 1$ constante: Enumerar las combinaciones de n elementos tomados de k en k.
- **Exponencial** $\Theta(k^n)$ para $k > 1$ constante: búsqueda en un espacio de configuraciones de anchura k y profundidad n.
- **Otros** $\Theta(n!), \Theta(n^n)$

Ordenes de Crecimiento Frecuentes



Cálculo del Coste Temporal en Algoritmos No Recursivos

- El coste de una **operación elemental** es $O(1)$, por ejemplo
  - asignación entre tipos atómicos (`int`, `bool`, `double`, `char`)
  - incremento o decremento de una variable de tipo atómico
  - operación aritmética
  - lectura o escritura de tipo atómico
  - comparación en tipos atómicos
  - acceso al elemento i-ésimo de un vector
- Evaluar una expresión tiene coste igual a la suma de los costes de las operaciones que contiene (incluidas las llamadas a funciones).
- El coste de una instrucción de tipo `return exp;` es la suma del coste de la evaluación de la expresión `exp` más el coste de asignar el resultado.
- El coste del paso de parámetros por referencia es $O(1)$.
- El coste de construir o copiar un vector de tamaño n (por ejemplo, declaración, paso por valor, retorno de resultados por valor) es $O(n)\times$ Coste de construir o copiar cada elemento.

Cálculo del Coste Temporal en Algoritmos No Recursivos

- Si el coste de un fragmento de un programa F_1 es C_1 y el de otro fragmento F_2 es C_2, el coste de la **composición secuencial**

$$F_1; F_2$$

es $C_1 + C_2$

- Si el coste de un fragmento de un programa F_1 es C_1, el de otro fragmento F_2 es C_2, i el de evaluar B es D, entonces el coste de la **composición alternativa**

```
if (B) F_1;
else F_2;
```

es $D + C_1$ si se cumple B, y $D + C_2$ en otro caso. En cualquier caso, el coste es menor o igual que $D + \max(C_1, C_2)$.

Cálculo del Coste Temporal en Algoritmos No Recursivos

- Si el coste de evaluar un fragmento de un programa F durante la k-ésima iteración es C_k, el coste de evaluar la expresión booleana B durante la k-ésima iteración es D_k, y el número de iteraciones es N, entonces el coste de evaluar de la **composición iterativa**

```
while (B) {
    F;
}
```

es $D_{N+1} + \sum_{k=1}^{N}(C_k + D_k)$.