Complejidad Algorítmica

Análisis y Diseño de Algoritmos

Dr. Jaime Osorio Ubaldo

Tiempo de ejecución o costo temporal de un algoritmo

Por ejemplo si queremos calcular la siguiente sumatoria

$$1 + 2 + 3 + \cdots + n$$

podemos usar los siguientes algoritmos:

1. Algoritmo 1.

$$suma \leftarrow \frac{n \cdot (n+1)}{2}.$$

2. Algoritmo 2.

$suma \leftarrow 0$
$i \leftarrow 1$
mientras$(i \leq n)$
    $suma \leftarrow suma + i$
    $i \leftarrow i + 1$

Costo temporal

Definiremos como una instrucción elemental a las siguientes operaciones:

1. Operación aritmética.
2. Asignación a una variable.
3. Llamada a una función.
4. Retorno de una función.
5. Comparaciones lógicas .
6. Acceso a una estructura (arreglo, matriz, lista ligada...).

El costo temporal de un algoritmo es la suma de los costos temporales de las instrucción elementales. Consideraremos que el tiempo de cualquier instrucción elemental es una unidad de tiempo.

Costo temporal

1. Algoritmo 1.

$$suma \leftarrow \frac{n \cdot (n+1)}{2}.$$

$f(n) = t_{multiplicación} + t_{división} + t_{asignación} = 1 + 1 + 1 = 3$

2. Algoritmo 2.

```
suma ← 0
i ← 1
mientras(i ≤ n)
    suma ← suma + i
    i ← i + 1
```

$$\begin{aligned} f(n) &= 2 \cdot t_{asignación} + (n+1) \cdot t_{comparación} + 2 \cdot n \cdot (t_{asignación} + t_{suma}) \\ &= 2 + n + 1 + 4n \\ &= 5n + 3 \end{aligned}$$

Costo temporal

1. Algoritmo 1. El costo temporal es constante

$$suma \leftarrow \frac{n \cdot (n+1)}{2}.$$

$f(n) = 3 \approx k$

2. Algoritmo 2. El costo temporal es lineal

$suma \leftarrow 0$
$i \leftarrow 1$
mientras$(i \leq n)$
    $suma \leftarrow suma + i$
    $i \leftarrow i + 1$

$f(n) = 5n + 3 \approx n$

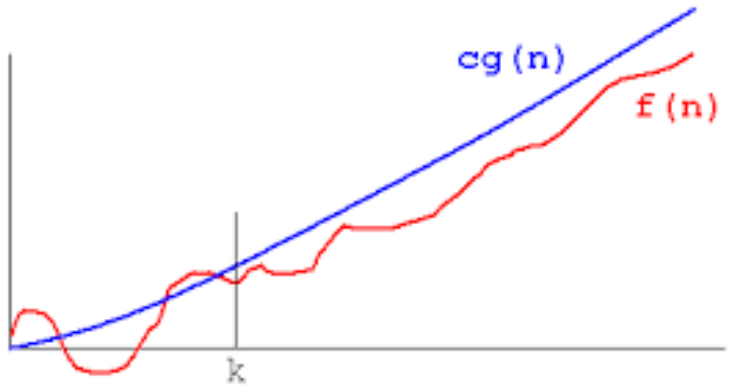
Orden de complejidad: Big $\mathcal{O}$

Definición.

Sean $f,\ g : \mathbb{N} \to \mathbb{R}^+ \cup \{0\}$. El conjunto de las funciones del orden de $g(n)$, denotado $\mathcal{O}(g(n))$, se define como sigue:

$$\mathcal{O}(g(n)) = \{f(n)\ :\ \exists c \in \mathbb{R}^+, \exists k \in \mathbb{N}, \forall n \geq k, [\ f(n) \leq c \cdot g(n)\ ]\}$$

diremos que una función f es del orden $g(n)$ cuando $f \in \mathcal{O}(g(n))$.



Ejemplos.

1. Para $f(n) = 3$ es $\mathcal{O}(1)$, ya que

$$\exists c = 3, \exists k = 2, \forall n \geq k, [\ f(n) \leq c \cdot 1\ ]$$

2. Para $f(n) = 5n + 3$ es $\mathcal{O}(n)$, ya que

$$\exists c = 6, \exists k = 3, \forall n \geq k, [\ f(n) \leq c \cdot n\ ]$$

Principales órdenes de complejidad: Big $\mathcal{O}$

1. Orden de complejidad constante $\mathcal{O}(1)$.
2. Orden complejidad $\mathcal{O}(\log n)$
3. Orden de complejidad lineal $\mathcal{O}(n)$.
4. Orden complejidad $\mathcal{O}(n \cdot \log n)$
5. Orden de complejidad cuadrática $\mathcal{O}(n^2)$.
6. Orden complejidad $\mathcal{O}(2^n)$

Orden de complejidad

1. Un instrucción crítica es la instrucción elemental que más veces se ejecuta dentro del programa en el peor caso.
2. Podemos simplificar el cálculo del costo temporal si hacemos uso de la intrucción crítica.
3. Para n suficientemente grande, el costo del resto del algoritmo perderá importancia frente al costo de repetir $f(n)$ veces la instrucción crítica.

Ejemplo:

$suma \leftarrow 0$
$i \leftarrow 1$
mientras($i \leq n$)
    $suma \leftarrow suma + i$
    $i \leftarrow i + 1$

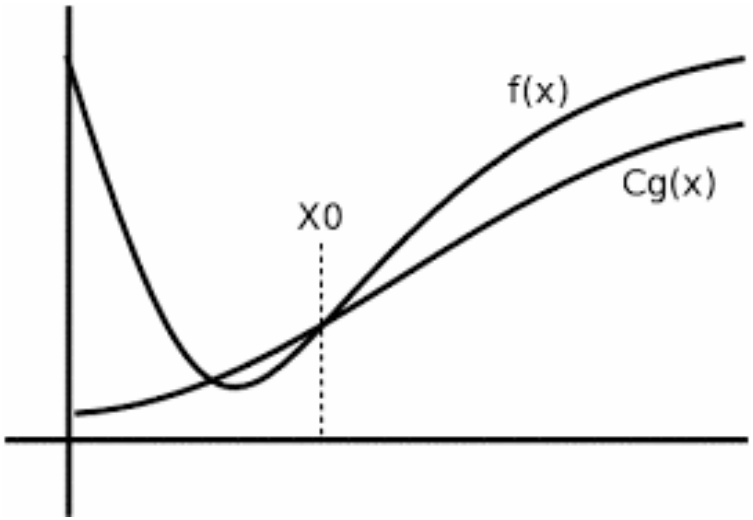
Aquí podemos elegir como instrucción crítica a la asignación que está dentro del bucle, como esto se repite n veces se tiene que $f(n) = n$. Por lo tanto el orden de complejidad es $\mathcal{O}(1.n) = \mathcal{O}(n)$.

Orden de Complejidad

Definición.

Sean $f,\ g : \mathbb{N} \to \mathbb{R}^+ \cup \{0\}$. El conjunto $\Omega(g(n))$, se lee omega de $g(n)$, se define como sigue:

$$\Omega(g(n)) = \{f(n)\ :\ \exists c \in \mathbb{R}^+, \exists x_o \in \mathbb{N}, \forall n \geq x_0, [\ f(n) \geq c \cdot g(n)\ ]\}$$



Orden de Complejidad

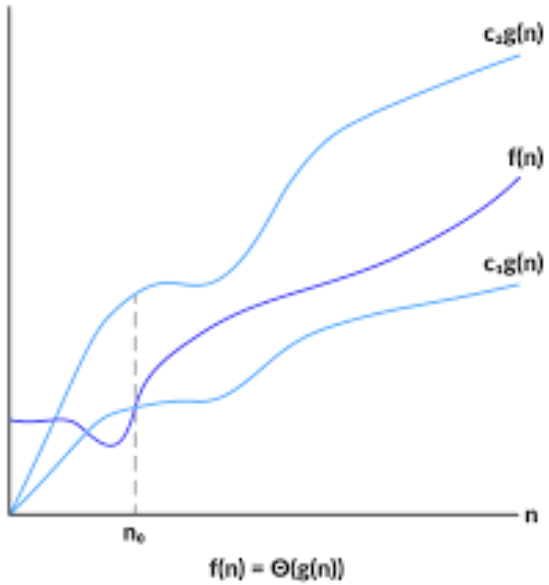
Definición.

El conjunto de funciones $\theta(g(n))$, se lee orden exacto de $g(n)$, se define como

$$\theta(g(n)) = \mathcal{O}(g(n)) \cap \Omega(g(n))$$

es decir

$$\theta(g(n)) = \{f(n) \ : \ \exists c_1, c_2 \in \mathbb{R}^+, \exists n_o \in \mathbb{N}, \forall n \geq n_0, [\ c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)\ ]\}$$



Orden de complejidad

Efecto de duplicar el tamaño del problema.

f(n)	n=100	n=200
$k_1 \log n$	1h	1.15h
$k_2 n$	1h	2h
$k_3 n \log n$	1h	2.3h
$k_4 n^2$	1h	4h
$k_5 n^3$	1h	8h
$k_6 2^n$	1h	$1.27 \times 10^{30} h$

Efecto de duplicar el tiempo disponible.

f(n)	t=1h	t=2h
$k_1 \log n$	n=100	n=10 000
$k_2 n$	n=100	n=200
$k_3 n \log n$	n=100	n=178
$k_4 n^2$	n=100	n=141
$k_5 n^3$	n=100	n=126
$k_6 2^n$	n=100	n=101

Orden de complejidad

1. Los que se comportan de un modo más acorde con las expectativas del usuario no informático son los de complejidad lineal y $\mathcal{O}(n \log n)$: al duplicar el tamaño del problema se duplica aproximadamente el tiempo empleado, y al duplicar el tiempo disponible, el tamaño que es posible tratar también se duplica.
2. El algoritmo de complejidad logarítmica tiene un comportamiento excepcionalmente bueno: doblar el tiempo disponible permite tratar problemas enormes en relación con el original.
3. Los órdenes cuadráticos y cúbicos tienen un comportamiento claramente inferior al lineal: en la tabla anterior se observa que un incremento del 100 % en el tiempo disponible solo se consigue un incremento del 41 % y del 26 %, respectivamente, en el tamaño del problema.

Orden del complejidad

1. En general, dado un algoritmo del orden $\mathcal{O}(n^a)$, multiplicar el tiempo disponible (o la velocidad del computador) por un factor k multiplica el tamaño del problema que es posible tratar por un factor $\sqrt[a]{k}$. Estos cálculos llevan a la conclusión de que no es posible tratar problemas demasiado grandes con algoritmos con estas tasas de crecimiento.
2. No obstante, los algoritmos cuyas tasas de crecimiento están acotadas superiormente por n^a, se dice que es de complejidad polinomial, y los problemas que se resuelven se llaman problemas tratables.

Orden de complejidad

1. Los problemas de complejidad exponencial o mayor, reciben el nombre de intratables. Se llaman así los problemas cuyos mejores algoritmos tienen tiempos en $\Omega(2^n)$.
2. Un algoritmo de tiempo de ejecución exponencial al duplicar la velocidad del procesador apenas afecta al tamaño del problema tratado, y duplicar el tamaño del problema conduce a tiempos de ejecución del orden de varios billones de veces la edad del universo (un siglo son aproximadamente 10^6 horas).

Orden de complejidad

Ejemplos.

1. Un ejemplo típico es encontrar la ruta más corta para visitar varias ciudades (el problema del agente viajero).
2. El problema de la asignación cuando intervienen 3 o más dimensiones. Por ejemplo, la asignación de salones, horarios, profesores y asignaturas tiene 4 dimensiones.
3. El problema de la mochila.
4. La búsqueda del camino simple más largo de un grafo.
5. la verificación de la existencia de ciclos hamiltonianos en un grafo.

Hasta la fecha no se ha encontrado un algoritmo que resuelva esta clase de problemas en tiempo polinomial. Los mejores algoritmos para resolver estos problemas crecen exponencialmente con el tamaño de la entrada y por esto se les cataloga como problemas intratables. Estos algoritmos requerieren más tiempo del disponible, excepto para tamaños de entrada muy pequeños.

Orden de complejidad

Conclusiones.

1. Es una buena inversión dedicar tiempo a encontrar algoritmos con mejores tasas de crecimiento. Por ejemplo pasar de un algoritmo $\mathcal{O}(n^2)$ a otro $\mathcal{O}(n \log n)$ ha de considerarse una gran mejora.
2. Encontrar un algoritmo $\mathcal{O}(\log n)$ para problemas que se resolvían en $\mathcal{O}(n)$ es una inmensa suerte.
3. Encontrar un algoritmo polinomial para problemas cuyos mejores algoritmos conocidos sean exponenciales, es un logro merecedor del premio Turing (el equivalente en Informática al premio Nobel).
4. Desgraciadamente, existen muchos problemas interesantes que han permanecido hasta la fecha en la categoría de exponenciales pese a los esfuerzos de numerosos investigadores, por lo que se sopecha con gran convicción que tales algoritmos no existen. Habrá que buscar pues, otra manera de pasar a la posteridad.
5. Si bien el criterio asintótico es muy útil hay circunstancias especiales en el que si el tamaño del problema no es muy grande o el algoritmo solo se usa unas cuantas veces se puede optar por usar el algoritmo teóricamente menos eficiente.

Los mejores algoritmos

Lea estas publicaciones y otras similares, y comparte.
https://www.genbeta.com/desarrollo/los-10-mejores-algoritmos-de-computacion-del-siglo-xx