Ayudantía 7 - Estructura de Datos y Algoritmos ELO320

Complejidad de algoritmos

Werner Creixell, Felipe Vera

16 de mayo del 2013

Aparte de codificar programas para resolver problemas, evaluar el tiempo en el que se ejecutan es crucial a la hora de diseñar una solución para un problema. Para eso se estudia la **Complejidad de algoritmos**, para tener un modo de evaluar el tiempo en el que se ejecutan las instrucciones.

1. Tiempo como unidad

Una manera básica de medir esto es con un cronómetro lo suficientemente rápido como para encontrar cuánto se demora en ejecutar cierto algoritmo. Dependiendo de la cantidad de operaciones y la velocidad de procesamiento de la máquina, esto puede variar entre microsegundos y horas.

Sin embargo este no es un criterio muy bueno para probar la complejidad de un algoritmo, porque:

- Puede variar por la máquina en la que está implementada (Arquitectura 32, 64 bits, ARM, MSP, etc.)
- Por eso mismo ciertas instrucciones máquina básicas tardan más que otras.
- Depende de las condiciones en el sistema operativo, si se ejecutan otros programas...

2. Notación asintótica

Para lidiar con este problema, una solución es la **notación asintótica**. Este se basa en contar la cantidad de instrucciones ejecutadas por cada elemento de un arreglo entrante, y redondear este resultado como se mostrará a continuación.

Supongamos que se tiene un algoritmo como el siguiente:

```
int strlen (char *cadena)
{
    int i = 0;
    while(cadena[i] != '\0')
        i++;

    return i;
}
```

Si n es la cantidad de caracteres que tiene la cadena hasta el caracter nulo, entonces la cantidad de instrucciones que ejecuta este algoritmo será

$$T(n) = 2n + 2$$

Por que al observar bien este algoritmo:

- Hay dos instrucciones que se ejecutarán independientemente de la cantidad de elementos (hacer i = 0 y regresar i)
- Hay otras dos instrucciones que se ejecutarán una vez por cada elemento de la cadena (comparación en while e incrementar i)

Entonces la notación algorítmica, en el caso de que $T(n)$ sea un polinomio, se calcula fácilmente.

- Se eliminan los elementos de orden menor. *Para un n grande, la constante + 2 se desprecia.*
- Se eliminan las constantes que multipliquen al término de orden mayor.

Entonces, en el ejemplo anterior el $T(n) = 2n + 2$ se transforma en $T(n) = \Theta(n)$.

2.1. Distintos tipos de notación asintótica

Hay tres tipos de notación asintótica.

- **Notación Ω (Cota inferior):** Un algoritmo con complejidad $f(n)$ será $\Omega(g(n))$ si existe alguna constante k tal que $k \cdot g(n)$ sea menor o igual a $f(n)$ cuando n sea mayor o igual a algún n_0 (desde donde se cumple esta propiedad).
- **Notación O (Cota superior):** Un algoritmo con complejidad $f(n)$ será $O(g(n))$ si existe alguna constante k tal que $k \cdot g(n)$ sea mayor o igual a $f(n)$ cuando $n \leq n_0$.
- **Notación Θ (Acotado por arriba y abajo):** Un algoritmo con complejidad $f(n)$ será $\Theta(g(n))$ si existen dos constantes k_1 y k_2 tal que $k_1 \cdot g(n)$ sea mayor, y $k_2 \cdot g(n)$ sea menor a $f(n)$ cuando $n \leq n_0$. Es decir, esta intersecta a Ω y O.

En el ejemplo anterior es posible demostrar que $T(n)$ es $\Theta(n)$ porque existen ambas constantes k_1 y k_2 que pueden cumplir que, cuando n es muy grande:

$$k_1 n \leq 2n + 2 \leq k_2 n$$

Cumple, si $k_1 = 1$ y $k_2 = 3$ por dar un ejemplo.

2.2. Otra forma de ver la notación asintótica

Existe otra forma de ver estas notaciones: usando límites. Como la notación asintótica analiza cómo aumenta el tiempo de ejecución de un algoritmo con un N lo más grande posible ($n \to \infty$), y se tiene una función de complejidad $T(n)$ tal que:

$$T(n) = f(n) \qquad \text{Cualquier función polinómica}$$

Se tiene los siguientes casos.

- $T(n) = O(g(n))$ (O es una cota superior) si es que:

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} \neq 0 \quad \text{y puede ser infinito.}$$

- $T(n) = \Omega(g(n))$ (Ω es una cota inferior) si es que:

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} \geq 0 \quad \text{y no es infinito.}$$

- $T(n) = \Theta(g(n))$ si es que:

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} > 0 \quad \text{y no es infinito.}$$

3. Analizando la complejidad de un algoritmo

3.1. Iterativo

Analizar la complejidad de un algoritmo iterativo (con *for*s y *while*s) es bastante simple. Hay que ser cuidadoso de que este *while* se ejecute una vez por cada elemento de entrada al algoritmo, porque si se ejecuta una cantidad fija de veces, no suma n elementos. El ejemplo dado arriba representa el cálculo de complejidad de un algoritmo iterativo.

3.2. Con condicionales

Se toma, por lo general, el peor caso.

3.3. Recursivo

Este caso es más complejo. Se debe escribir una expresión $T(n)$ dependiente de ella misma para analizar su complejidad.

Por ejemplo, al tener el pseudocódigo de la FFT Radix-2 (Una forma rápida para calcular la transformada de Fourier de una señal muestreada)

```
X[0,...,n − 1] <- fftradix2(x, N, s):

if N = 1 then
    X0 <- x0
else
    X[0,...,N/2 − 1] <- fftradix2(x, N/2, 2s)
    X[N/2,...,n − 1] <- fftradix2(x+s, N/2, 2s)
    for k = 0 to n/2 − 1
        t <- Xk
        Xk <- t + exp(−j * 2pi k/n) Xk+n/2
        Xk+N/2 <- t − exp(−j * 2pi k/n) Xk+n/2
    endfor
endif
```

Se analizará este algoritmo tomando el caso en el que $n > 1$, el peor caso para el if. Es posible ver que se ejecuta dos veces el mismo algoritmo con la mitad de elementos, y a continuación un bucle *for* con complejidad $\Theta(n)$. Se tiene entonces:

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + \Theta(n)$$

Entonces, para un n lo suficientemente grande, $\Theta(n)$ puede aproximarse a $k \cdot n$. Luego se tiene:

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + kn$$

Esto se puede reescribir, reemplazando n por $\frac{n}{2}$ como:

$$T(n) = 2\left[2 \cdot T\left(\frac{n}{4}\right) + k\frac{n}{2}\right] + kn$$

Eventualmente, al dividir n por 2 sucesivamente, se tendrá $n = 1$, con un algoritmo de complejidad $\Theta(1)$, debido a la otra bifurcación del condicional. Esto se debe hacer aproximadamente $\log_2(n)$ veces. Por lo tanto se tendrá:

$$kn + k\frac{n}{2} + k\frac{n}{4} + k\frac{n}{8} + ... \qquad \log_2(n) \text{ sumandos}$$

Al sumar todo esto se tiene una multiplicación:

$$T(n) \approx \log_2(n) \cdot \Theta(n) \quad = \quad \Theta(n \cdot \log(n))$$

La razón por la que no se incluye la base 2 en la medición de complejidad es debido a que el cambio de base hace que el logaritmo se divida por $\log 2$.

3.4. Método maestro

Hay una forma sistemática de analizar la complejidad de este tipo de algoritmos recursivos, y se denomina *método maestro*. Este postula lo siguiente:

Si hay un algoritmo recursivo cuya complejidad sea:

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$$

Se calcula $d = \log_b a$, y se observa qué complejidad tiene la parte no recursiva $f(n)$, el cual debería ser:

$$f(n) = \Theta(n^c)$$

Opcionalmente con un logaritmo. Comparando c y d hay tres casos.

- **Caso 1:** $c < d$: Se ignora la parte de $f(n)$, por lo que:

$$T(n) = \Theta(n^d)$$

- **Caso 2:** $c = d$: Se toma $f(n)$, y se le multiplica un $\log(n)$. *Cuenta también si $f(n)$ ya tiene un $\log(n)$, en cuyo caso se le multiplica otro*

$$T(n) = \Theta(n^d \cdot \log(n))$$

- **Caso 3:** $c > d$: Se ignora la parte recursiva, por lo que:

$$T(n) = \Theta(n^c)$$