Explicación TP1

Programación III - Ing. en Computación - UNLP

Cursada 2012

1. Propósito

Nos interesa analizar y comparar diferentes algoritmos para saber cuál es más rápido.

Una forma de hacerlo es escribir el código de cada algoritmo, ejecutarlos con diferentes entradas y medir el tiempo de reloj que demora cada uno.

Por ejemplo, para ordenar una secuencia de números existen diferentes algoritmos: inserción, mergesort, etc. Se puede tomar cada uno de estos algoritmos y ejecutarlos con una secuencia de números, midiendo el tiempo que tarda cada uno.

Sin embargo, existe otra forma de comparar algoritmos, que se conoce como análisis asintótico. Consiste en analizar el código y convertirlo en una expresión matemática que nos diga cuanto tiempo demora cada uno en función de la cantidad de elementos que se está procesando.

De esta forma, ya no es necesario ejecutarlos, y nos permite comparar algoritmos en forma independiente de una plataforma en particular. En el caso de los algoritmos de ordenación, la función nos dará un valor en base a la cantidad de elementos que se están ordenando.

2. Análisis Asintótico BigOh

$f(n)$ es de $\mathcal{O}(g(n))$ si y solo si $\exists c$ tal que $f(n) \leq cg(n), \forall n \geq n_0$.

La definición de BigOh dice que una función $f(n)$ es de orden $g(n)$ si la función $g(n)$ multiplicada por una constante c acota por arriba a la función $f(n)$.

La función $g(n)$ acota a $f(n)$ si cada valor de $g(n)$ es mayor o igual que $f(n)$. Por ejemplo, la función $g(n) = n + 1$ acota por arriba a la función $f(n) = n$ ya que para todo $n \geq 0, f(n) \leq g(n)$.

Nuestro objetivo es encontrar un c que multiplicado por $g(n)$ haga que se cumpla la definición. Si ese c existe, $f(n)$ será de orden $g(n)$. Caso contrario, si no es posible encontrar c, $f(n)$ no es de orden $g(n)$.

El problema entonces se reduce a encontrar el c, que algunas veces es trivial y directo, y otras no lo es. Además, el hecho de no poder encontrarlo no quiere decir que no exista, por lo que hace falta usar algun mecanismo que nos asegure que existe o no.

2.1. Por definición

Usando la definición de BigOh, una forma de descubrir si una función $f(n)$ es de orden $g(n)$ se traduce en realizar operaciones para despejar el valor de c, y descubrirlo en caso que exista, o llegar a un absurdo en caso que no exista.

2.2. Por definición, ejemplo

Ejemplo: ¿3^n es de $\mathcal{O}(2^n)$?

Aplicando la definición de BigOh, tenemos que encontrar un c que cumpla:

$$3^n \leq c2^n, \forall n \geq n_0 \tag{2.1}$$

La segunda parte de la definición, que dice para todo n mayor o igual a n_0, es tan importante como la primera, y no hay que olvidarse de escribirlo. El valor de n_0 puede ser cualquier que nos ayude a hacer verdadera la desigualdad. Por ejemplo, en este caso se puede tomar $n_0 = 1$.

n_0 no puede ser negativo. Elegir un n_0 negativo significa que estamos calculando el tiempo de ejecución con una cantidad negativa de elementos, algo que no tiene sentido.

Eso también significa que las funciones siempre deben dar un valor positivo, dado que el valor de la función de la que se quiere calcular el orden representa tiempo de ejecución de un algoritmo. El tiempo debe ser siempre positivo, y por ejemplo decir que una función demora -3 segundos en procesar 10 elementos tampoco tiene sentido.

Se asume que 3^n es de $\mathcal{O}(2^n)$, y se pasa el 2^n dividiendo a la izquierda:

$$\frac{3^n}{2^n} \leq c, \forall n \geq n_0 \tag{2.2}$$

$$\left(\frac{3}{2}\right)^n \leq c, \forall n \geq n_0 \tag{2.3}$$

Se llega a un absurdo, ya que no es posible encontrar un número c para que se cumpla la desigualdad para todo n mayor que algún n_0 fijo. Por lo tanto es falso, 3^n no es de $\mathcal{O}(2^n)$.

Un análisis un poco más cercano nos muestra que la funcion $(\frac{3}{2})^n$ es siempre creciente hasta el infinito, donde el límite:

$$\lim_{n\to\infty} \left(\frac{3}{2}\right)^n \tag{2.4}$$

toma el valor infinito. Por ejemplo: para cualquier constante c fija, para algunos valores de n se cumple, pero para otros no. Con $c = 100$,

$$\left(\frac{3}{2}\right)^n \leq 100, \forall n \geq n_0 \tag{2.5}$$

Despejando el valor de n,

$$\log_{\frac{3}{2}} \left(\frac{3}{2}\right)^n \leq \log_{\frac{3}{2}} 100, \forall n \geq n_0 \tag{2.6}$$

$$\log_{\frac{3}{2}} \left(\frac{3}{2}\right)^n = n; \log_{\frac{3}{2}} 100 \approx 11{,}35 \tag{2.7}$$

$$n \leq 11{,}35, \forall n \geq n_0 \tag{2.8}$$

No es posible que n sea menor a 11,35 y a la vez n sea mayor a algún n_0, ya que n debe ser mayor que n_0 siempre.

2.3. Por regla de los polinomios

En las teorías se vieron algunas reglas para simplificar el trabajo:

Dado un polinomio $P(n)$ de grado k, sabemos que el orden del polinomio es $\mathcal{O}(n^k)$. Ejemplo: $P(n) = 2n^2 + 3n + 5$. Como el grado del polinomio es 2, el orden de $P(n)$ es $\mathcal{O}(n^2)$.

Esta regla solo se aplica a polinomios. Cualquier función que no lo sea habrá que desarrollarla por algún otro método. Ejemplo: $T(n) = n^{\frac{1}{2}} = \sqrt{n}$ no es un polinomio, y no puede aplicarse esta regla.

2.4. Por regla de la suma

Si $T_1(n) = \mathcal{O}(f(n))$ y $T_2(n) = \mathcal{O}(g(n))$ entonces:

$$T_1(n) + T_2(n) = max(\mathcal{O}(f(n)), \mathcal{O}(g(n))) \tag{2.9}$$

2.5. Otras reglas

$T(n) = \log^k n \rightarrow$ es $\mathcal{O}(n)$.

$T(n) = cte$, donde cte es una expresión constante que no depende del valor de $n \rightarrow$ es $\mathcal{O}(1)$.

$T(n) = cte \times f(n) \rightarrow$ es $\mathcal{O}(f(n))$.

3. Identidades de sumatorias y logaritmos

$$\sum_{i=1}^{n} c = nc \tag{3.1}$$

$$\sum_{i=k}^{n} c = (n-k+1)c \tag{3.2}$$

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} \tag{3.3}$$

$$\sum_{i=0}^{n} i^2 = \frac{n(n+1)(2n+1)}{6} \tag{3.4}$$

$$\sum_{i=0}^{n} i^3 = \left(\frac{n(n+1)}{2}\right)^2 \tag{3.5}$$

$$\sum_{i=0}^{n} i^4 = \frac{n(n+1)(2n+1)(3n^2+3n-1)}{30} \tag{3.6}$$

$$\sum_{i=k}^{n} f(i) = \sum_{i=1}^{n} f(i) - \sum_{i=1}^{k-1} f(i) \tag{3.7}$$

$$\sum_{i=0}^{n} 2^i = 2^{n+1} - 1 \tag{3.8}$$

$$\log_b a = \frac{log_c a}{\log_c b} \tag{3.9}$$

$$\log \frac{a}{b} = \log a - \log b \tag{3.10}$$

$$\log (a \times b) = \log a + \log b \tag{3.11}$$

4. Análisis de Algoritmos y Recurrencias

El análisis de algoritmos consiste en convertir un bloque de código o función escrita en algún lenguaje de programación a su equivalente versión que nos permita calcular su función de tiempo de ejecución $T(n)$.

Eso es, el objetivo es encontrar cual es el tiempo de ejecución a partir de la cantidad de elementos.

El objetivo de resolver una recurrencia es convertir la función en su forma explícita equivalente sin contener sumatorias ni llamadas recursivas. Para ello hace falta aplicar las definiciones previas de sumatorias, y el mecanismo de resolución de recurrencias.

4.1. Expresión constante

Cualquier expresión que no dependa de la cantidad de elementos a procesar se marcará como constante, $c_1, ..., c_k$.

4.2. Grupo de constantes

Cualquier grupo de constantes $c_1, ..., c_k$ se pueden agrupar en una única constante c, $c = (c_1 + ... + c_k)$.

4.3. Secuencias

Dadas dos o más sentencias una después de la otra sus tiempos se suman.

4.4. Condicionales

Dado un condicional *if*, el tiempo es el peor caso entre todos los caminos posibles de ejecución.

4.5. for y while

Los bloques *for* y *while* se marcan como la cantidad de veces que se ejecutan, expresado como una sumatoria desde 1 a esa cantidad. En el caso del *for* se usará como variable de la sumatoria el mismo indice del for. En el caso del *while* se usará una variable que no haya sido usada.

4.6. Llamadas recursivas

Las llamadas recursivas se reemplazan por la definicion de tiempo recursiva. Las definiciones recursivas son funciones definidas por partes.

4.7. Ejemplo 1, iterativo

```
1 void int sumar(int[] datos) {
2     int acumulado = 0;
3     for (int i=0; i<datos.length; i++) {
4         acumulado = acumulado + datos[i];
5     }
6     return acumulado;
7 }
```

Linea 2: c_1.

Linea 3: $\sum_{i=1}^{n}$.

Linea 4: c_2.

Linea 6: c_3.

Todo junto, el $T(n)$ de la función sumar es:

$$T(n) = c_1 + (\sum_{i=1}^{n} c_2) + c_3 \tag{4.1}$$

El siguiente paso es convertir el $T(n)$ a su versión explícita sin sumatorias. Aplicando las identidades de la sección anterior:

$$T(n) = c_1 + nc_2 + c_3 = c_4 + nc_2, c_4 = c_1 + c_3 \tag{4.2}$$

Por lo tanto, el $T(n)$ de la función sumar es:

$$T(n) = c_4 + nc_2 \tag{4.3}$$

¿Cuál es su orden de ejecución BigOh? $T(n)$ es $\mathcal{O}(n)$.

4.8. Ejemplo 2, recursivo

```
1 int rec1(int n){
2   if (n <= 1)
3     return 3;
4   else
5     return 1 + rec1(n-1) * rec1(n-1);
6 }
```

Las funciones recursivas se definen por partes, donde cada parte se resuelve de forma similar a un codigo iterativo:

$$T(n) = \begin{cases} c_1 & n \leq 1 \\ 2T(n-1) + c_2 & n > 1 \end{cases}$$

El objetivo es encontrar una versión explícita de la recurrencias. Se comienza por escribir las primeras definiciones parciales:

$$T(n) = 2T(n-1) + c_2, n > 1 \tag{4.4}$$
$$T(n-1) = 2T(n-2) + c_2, n-1 > 1 \tag{4.5}$$
$$T(n-2) = 2T(n-3) + c_2, n-2 > 1 \tag{4.6}$$

Segundo, se expande cada término tres ó cuatro veces, las que sean necesarias para descubrir la forma en la que los diferentes términos van cambiando:

$$T(n) = 2T(n-1) + c_2 \tag{4.7}$$
$$T(n) = 2(2T(n-2) + c_2) + c_2 \tag{4.8}$$
$$T(n) = 2(2(2T(n-3) + c_2) + c_2) + c_2 \tag{4.9}$$
$$T(n) = 2(2^2T(n-3) + 2c_2 + c_2) + c_2 \tag{4.10}$$
$$T(n) = 2^3T(n-3) + 2^2c_2 + 2c_2 + c_2, \forall n-3 \geq 1 \tag{4.11}$$

La definición recursiva continúa mientras el $n > 1$, por lo que en k pasos tenemos la expresión general del paso k:

$$T(n) = 2^kT(n-k) + \sum_{i=0}^{k-1} 2^ic_2 \tag{4.12}$$

La llamada recursiva finaliza cuando $n - k = 1$, por lo tanto:

$$n - k = 1 \tag{4.13}$$
$$k = n - 1 \tag{4.14}$$
$$T(n) = 2^{n-1}T(1) + \sum_{i=0}^{n-1-1} 2^i c_2 \tag{4.15}$$
$$T(n) = 2^{n-1}c_1 + \sum_{i=0}^{n-2} 2^i c_2 \tag{4.16}$$
$$T(n) = 2^{n-1}c_1 + c_2 \sum_{i=0}^{n-2} 2^i \tag{4.17}$$
$$T(n) = 2^{n-1}c_1 + c_2(2^{n-2+1} - 1) \tag{4.18}$$
$$T(n) = 2^{n-1}c_1 + 2^{n-1}c_2 - c_2 \tag{4.19}$$
$$T(n) = 2^{n-1}(c_1 + c_2) - c_2 \tag{4.20}$$

¿Cuál es su orden de ejecución BigOh? $T(n)$ es $\mathcal{O}(2^n)$.