Algoritmos y Programación I
Tema 10: Análisis asintótico de algoritmos

22 de julio de 2009

1. Eficiencia de algoritmos

Una vez que se disponga de un algoritmo que funciona correctamente, es necesario establecer criterios para medir su rendimiento, con el objetivo de compararlo con otros algoritmos que resuelvan el mismo problema. Estos criterios se centran principalmente en su simplicidad y en el uso eficiente de los recursos.

La sencillez o simplicidad de un algoritmo es una característica muy importante, puesto que facilita su verificación, el estudio de su eficiencia y su mantenimiento. No existe un método sistemático para determinar la simplicidad de un algoritmo, sino que simplemente se determina cuán fácil se entiende o transmite un algoritmo al ser humano.

Respecto al uso eficiente de los recursos, éste suele medirse en función de dos parámetros: el tiempo que tarda en ejecutarse y el espacio de memoria que requiere durante su ejecución, siendo el tiempo de ejecución el recurso más crítico en la mayoría de los casos.

2. Eficiencia temporal

El tiempo de ejecución de un algoritmo va a depender de diversos factores como lo son: los datos de entrada que le suministremos, la calidad del código objeto creado por el compilador, la naturaleza y rapidez de las instrucciones máquina del procesador concreto que ejecute el programa, y la complejidad intrínseca del algoritmo.

Existen dos maneras posibles de estudiar el tiempo de un algoritmo:

1. **Análisis asintótico:** proporciona una medida teórica (a priori), que consiste en obtener una función que acote (por arriba o por abajo) el tiempo de ejecución del algoritmo para unos valores de entrada dados.

2. **Benchmarking:** ofrece una medida real (a posteriori), consistente en medir el tiempo de ejecución del algoritmo para unos valores de entrada dados en una implementación concreta (lenguaje, compilador y máquina).

Ambas medidas son importantes puesto que, si bien la primera nos ofrece estimaciones del comportamiento de los algoritmos de forma independiente de su implementación, y sin necesidad de ejecutarlos, la segunda representa las medidas reales del comportamiento del algoritmo.

En este estudio sólo nos enfocaremos en el análisis asintótico de algoritmos.

3. Función de tiempo de un algoritmo

Para el análisis asintótico se desea obtener una función de acotamiento cuyo argumento sea una medida cuantitativa de la entrada del algoritmo y cuyo resultado represente el tiempo de ejecución del mismo. A continuación veremos qué asumiremos como representación de la entrada y del tiempo.

En cuanto a la entrada, observamos que su característica más influyente en el tiempo de ejecución es su tamaño, es decir, algún entero que indique el número de componentes que contiene la entrada. Por ejemplo, si la entrada es un vector, probablemente el tiempo dependerá principalmente del número de componentes en el vector a evaluar; o en un grafo, dependerá del número de vértices, de aristas o de ambos. La segunda característica de la entrada que influye en el tiempo de ejecución de un algoritmo es su valor o valores. Es decir, que para dos entradas del mismo tamaño, un algoritmo puede incluso tardarse tiempos diferentes en finalizar su ejecución.

En este sentido, para cada algoritmo identificamos algún número natural (a veces más de uno) que represente el tamaño de la entrada. En casos en los que la entrada siempre sea del mismo tamaño, o su tamaño no implique variabilidad significativa en el tiempo de ejecución, se adopta como entrada a la función de tiempo algún valor de la entrada del algoritmo que haga variar principalmente su tiempo de ejecución.

Para el resto de las características de la entrada del algoritmo que afecten el tiempo de ejecución (naturaleza de los datos, tamaños de subestructuras) se adopta una de tres suposiciones:

Peor caso: corresponde a la traza (secuencia de instrucciones) del algoritmo que realiza más instrucciones.

Mejor caso: corresponde a la traza del algoritmo que realiza menos instrucciones.

Caso medio: corresponde a la traza del algoritmo que realiza un número de instrucciones igual a la esperanza matemática de la variable aleatoria definida por todas las posibles trazas del algoritmo para un tamaño de la entrada dado, con las probabilidades de que éstas ocurran para esa entrada.

Aquí sólo estudiaremos funciones que acoten **superiormente** el tiempo de ejecución en el **peor caso**, que es generalmente el criterio más adecuado.

En cuanto a la representación del tiempo, no es posible expresarlo en alguna unidad de tiempo concreta, como por ejemplo en segundos, puesto que no existe una computadora estándar a la que puedan hacer referencia todas las medidas. Sin embargo, el tiempo de ejecución de dos implementaciones diferentes de un algoritmo dependerá principalmente del tiempo de ejecución de cada una de las operaciones elementales que realiza el procesador; y dado que los tiempos de éstas en cada procesador se pueden acotar por una constante, es suficiente entonces con contar el número de operaciones elementales que ejecutaría un algoritmo.

En conclusión, la función de tiempo de un algoritmo tiene como entrada el tamaño de la entrada del algoritmo y como salida el número de operaciones elementales que éste realiza.

4. Conteo de operaciones elementales (OE)

A continuación se presenta una lista de reglas generales para el cálculo del número de OE, siempre considerando el peor caso:

1. Se considerarán operaciones elementales las siguientes: operaciones aritméticas, lógicas y relacionales, asignaciones, llamadas a acciones nominadas, retorno de acciones nominadas y acceso a un elemento de un arreglo.

2. Para secuencias de instrucciones, se suman las OE de cada una de las instrucciones.

3. Para condicionales simples, se suman las OE correspondientes a la evaluación de la condición más las OE del cuerpo del condicional.

4. Para condicionales dobles, se suman las OE correspondientes a la evaluación de la condición más el máximo entre el número de OE de las dos cláusulas del condicional.

5. Para los ciclos *mientras* y *repetir* se toma la sumatoria sobre todas las iteraciones (en el peor caso) del número de OE de la condición más el número de OE del cuerpo del ciclo.

6. Para los ciclos *para* se cuenta el número de OE correspondientes a la conversión del ciclo *para* a un ciclo *mientras*.

7. Para llamadas a acciones nominadas, se cuenta 1 (por la llamada) más el número de OE en la evaluación de los argumentos más el número de OE correspondientes a la ejecución de la acción nominada.

8. Para algoritmos recursivos se define una función de tiempo a su vez recursiva, llamada ecuación de recurrencia, la cual tiene solo un caso base y solo un caso recursivo. El caso base se toma como la mayor cantidad de OE para llegar a cualquiera de los casos base del algoritmo. El caso recursivo se toma como la mayor cantidad de OE necesarias para ejecutar los casos recursivos. La ecuación de recurrencia resultante se debe convertir a una forma no recurrente como se verá más adelante.

Por ejemplo, considérese el siguiente algoritmo:

```
proc buscar(v: arreglo[1..MAX] de Entero, n: Entero, x: Entero): Entero
var
  i: Entero
inicio
  i ← 1                                  // 1 OE
  mientras (i ≤ n ∧ v[i] < x) hacer      // 4 OE
    i ← i + 1                            // 2 OE
  fmientras
  si (i ≤ n ∧ v[i] = x) entonces         // 4 OE
    retornar i                           // 1 OE
  sino
    retornar 0                           // 1 OE
  fsi
fin
```

En este caso el tamaño de la entrada corresponde al número de elementos del vector (n). Por lo tanto, su función de tiempo es:

$$T(n) = 1 + \left(\sum_{i=1}^{n} 4 + 2\right) + (4 + máx(1,1)) = 1 + 6n + 5 = 6n + 6$$

5. Notación asintótica *O-grande*

Dado que la diferencia entre los tiempos de ejecución reales entre dos algoritmos se hace más aparente con tamaños de entrada grandes, lo que realmente importa acerca de las funciones de tiempo es cuán rápido crecen. Es por ello que se define la notación asintótica *O-grande*, así como otras notaciones similares, para establecer clases de equivalencia entre funciones de tiempo según su nivel de crecimiento, es decir, dos funciones de tiempo pertenecen a la misma clase de equivalencia si "crecen de la misma forma".

Dada una función f, queremos estudiar aquellas funciones g que a lo sumo crecen tan rápido como f. Al conjunto de tales funciones se les llama cota superior de f y lo denominamos $O(f)$ ("o de efe"). Conociendo la cota superior de un algoritmo, podemos asegurar que, en ningún caso, el tiempo empleado será de un orden superior al de la cota. Formalmente, sea $f : \mathbb{N} \to \mathbb{R}^{\geq 0}$, entonces

$$O(f) = \{g : \mathbb{N} \to \mathbb{R}^{\geq 0} | \exists c \in \mathbb{R}^+ . \exists n_0 \in \mathbb{N} . \forall n \geq n_0 . g(n) \leq cf(n)\}$$

Intuitivamente, $t \in O(f)$ indica que t está acotada superiormente por algún múltiplo de f. Normalmente estaremos interesados en la menor función f (la que crezca más lento) tal que $t \in O(f)$.

6. Propiedades de la notación *O-grande*

Sean $f, g, h : \mathbb{N} \to \mathbb{R}^{\geq 0}$ funciones arbitrarias. La notación *O-grande* verifica las siguientes propiedades:

1. **Reflexividad:** $f \in O(f)$.
2. **Transitividad:** $f \in O(g) \wedge g \in O(h) \Rightarrow f \in O(h)$.
3. **Regla de eliminación de constantes:** $\forall c \in \mathbb{R}^+ . O(cf) = O(f)$.
4. **Regla de la suma:** $f \in O(f') \wedge g \in O(g') \Rightarrow f + g \in O(máx(f', g')$. Por reflexividad y transitividad se tiene que: $O(f + g) = O(máx(f, g)$.
5. **Regla de los logaritmos:** $\forall a, b \in \mathbb{N}^+ . O(log_a(n)) = O(log_b(n))$.
6. $f \in O(g) \Rightarrow O(f) \subseteq O(g)$.

7. Jerarquía de órdenes de complejidad

La representación del tiempo de ejecución de los algoritmos utilizando la notación asintótica da lugar a un conjunto de clases de equivalencia, llamadas órdenes de complejidad. Estas clases se pueden clasificar según su crecimiento dando lugar a una jerarquía de órdenes de complejidad. Los órdenes de complejidad de la mayoría de los algoritmos están dentro de la siguiente jerarquía:

1. $O(1)$: complejidad constante.
2. $O(log(n))$: complejidad logarítmica.
3. $O(n)$: complejidad lineal.
4. $O(n\ log(n))$: complejidad cuasi-lineal.
5. $O(n^2)$: complejidad cuadrática.
6. $O(n^3)$: complejidad cúbica.
7. $O(n^k), (k > 3)$: complejidad polinomial.
8. $O(k^n), (k > 1)$: complejidad exponencial.
9. $O(n!)$: complejidad factorial.

8. Ejemplos de complejidades temporales de algoritmos iterativos

8.1. Ejemplo 1: Factorial

```
func factorial(n : Entero): Entero
var
  fac, i: Entero
inicio
  fac ← 1
  para i ← 2 hasta n hacer
    fac ← fac * i
  fpara
  retornar fac
fin
```

$$
\begin{aligned}
& T(n) = 1 + 1 + \left(\sum_{i=2}^{n} 1 + 2 \right) + 1 = 3n + 3 \\
\Rightarrow \quad & T(n) \in O(3n + 3) \text{ (por reflexividad)} \\
\Rightarrow \quad & T(n) \in O(3n) \text{ (por regla de la suma)} \\
\Rightarrow \quad & T(n) \in O(n) \text{ (por regla de eliminación de constantes)}
\end{aligned}
$$

8.2. Ejemplo 2: Ordenamiento burbuja

```
proc burbuja(var v: arreglo[1..MAX] de Entero, n: Entero)
var
  i, j: Entero
  temp: Entero
inicio
  para i ← 1 hasta n-1 hacer
    para j ← i+1 hasta n hacer
      si (v[j] < v[i]) entonces
        temp ← v[j]
        v[j] ← v[i]
        v[i] ← temp
      fsi
    fpara
  fpara
fin
```

$$
\begin{aligned}
& T(n) = 1 + \left(\sum_{i=1}^{n-1} 2 + \left(2 + \sum_{j=i+1}^{n} 1 + 3 + 7 \right) \right) \\
\Rightarrow \quad & T(n) = 1 + \left(\sum_{i=1}^{n-1} 2 + \left(2 + \sum_{j=1}^{n-i} 11 \right) \right) \\
\Rightarrow \quad & T(n) = 1 + \left(\sum_{i=1}^{n-1} 4 + 11(n-i) \right) \\
\Rightarrow \quad & T(n) = 1 + \left(\sum_{i=1}^{n-1} 4 \right) + 11 \left(\sum_{i=1}^{n-1} n \right) - 11 \left(\sum_{i=1}^{n-1} i \right) \\
\Rightarrow \quad & T(n) = 1 + 4(n-1) + 11n(n-1) - 11\frac{n(n-1)}{2} \\
\Rightarrow \quad & T(n) = 1 + 4n - 4 + 11n^2 - 11n - \frac{11}{2}n^2 + \frac{11}{2}n \\
\Rightarrow \quad & T(n) \in O((11 - 11/2)n^2 + (4 - 11 + 11/2)n - 3) \\
\Rightarrow \quad & T(n) \in O((11 - 11/2)n^2) \\
\Rightarrow \quad & T(n) \in O(n^2)
\end{aligned}
$$

9. Resolución de ecuaciones de recurrencia

El conteo de operaciones elementales en los algoritmos recursivos da lugar a ecuaciones de recurrencia, las cuales deben ser convertidas a formas no recurrentes. Esta tarea puede no ser fácil dependiendo del caso. A continuación se presenta una serie de técnicas de resolución de recurrencias.

9.1. Método de expansión

Consiste en sustituir recurrencias por su igualdad hasta percibir un patrón particular a partir del cual sea posible inferir la solución con un $T(n_0)$ conocido. Este método generalmente sólo funciona para ecuaciones de recurrencia con una sola llamada recursiva.

9.1.1. Ejemplo 1: factorial

```
func factorial(n: Entero): Entero
inicio
  si (n ≤ 1) entonces
    retornar 1
  sino
    retornar n*factorial(n-1)
  fsi
fin
```

$$
T(n) = \begin{cases} 2 & \text{si } n \leq 1 \\ 5 + T(n-1) & \text{si } n > 1 \end{cases}
$$

Luego, para $a = 2$ y $b = 5$:

$$\begin{aligned} T(n) &= b + T(n-1) = b + (b + T(n-2)) \\ &= 2b + T(n-2) = 2b + (b + T(n-3)) \\ &= 3b + T(n-3) = 3b + (b + T(n-4)) \\ &= 4b + T(n-4) \\ &\vdots \\ &= kb + T(n-k) \end{aligned}$$

La sucesión termina cuando $n - k = 1$, por lo tanto $k = n - 1$. Sustituyendo:

$$\begin{aligned} & T(n) = b(n-1) + T(1) \\ \Rightarrow \quad & T(n) \in O(bn - b + a) \\ \Rightarrow \quad & T(n) \in O(bn) \\ \Rightarrow \quad & T(n) \in O(n) \end{aligned}$$

9.1.2. Ejemplo 2

```
func rec1(n: Entero): Entero
inicio
  si (n = 0) entonces
    retornar 5
  sino
    retornar rec1(n-1) + rec1(n-1)
  fsi
fin
```

$$T(n) = \begin{cases} a & \text{si } n = 0 \\ b + 2T(n-1) & \text{si } n > 0 \end{cases}$$

$$\begin{aligned} T(n) &= b + 2T(n-1) = b + 2(b + 2T(n-2)) \\ &= 3b + 4T(n-2) = 3b + 4(b + 2T(n-3)) \\ &= 7b + 8T(n-3) = 7b + 8(b + 2T(n-4)) \\ &= 15b + 16T(n-4) \\ &\vdots \\ &= (2^k - 1)b + 2^k T(n-k) \end{aligned}$$

Para $n-k=0$, se tiene que $k=n$. Sustituyendo:

$$\begin{aligned} & T(n) = (2^n - 1)b + 2^n T(0) \\ \Rightarrow \quad & T(n) \in O(b2^n - b + a2^n) \\ \Rightarrow \quad & T(n) \in O(ab2^n - b) \\ \Rightarrow \quad & T(n) \in O(ab2^n) \\ \Rightarrow \quad & T(n) \in O(2^n) \end{aligned}$$

9.1.3. Ejemplo 3

```
func rec2(n: Entero): Entero
inicio
  si (n ≤ 1) entonces
    retornar 2
  sino
    retornar 2*rec2(n div 2)
  fsi
fin
```

$$T(n) = \begin{cases} a & \text{si } n \leq 1 \\ b + T(n/2) & \text{si } n > 1 \end{cases}$$

$$\begin{aligned} T(n) &= b + T(n/2) = b + (b + T(n/4)) \\ &= 2b + T(n/4) = 2b + (b + T(n/8)) \\ &= 3b + T(n/8) \\ &\vdots \\ &= kb + T(n/2^k) \end{aligned}$$

Para $n/2^k = 1$, se tiene que $2^k = n \Rightarrow k = log_2(n)$. Sustituyendo:

$$\begin{aligned} & T(n) = log_2(n)b + T(1) \\ \Rightarrow \quad & T(n) \in O(log_2(n)b + a) \\ \Rightarrow \quad & T(n) \in O(log_2(n)b) \\ \Rightarrow \quad & T(n) \in O(log(n)) \end{aligned}$$

9.2. Método de la ecuación característica

Una ecuación de recurrencia lineal homogénea con coeficientes constantes es una ecuación de la forma:

$$f(n) = a_1 f(n-1) + a_2 f(n-2) + \cdots + a_k f(n-k) \qquad (n \geq k)$$

A esta ecuación se le asocia la siguiente *ecuación característica*:

$$x^k - a_1 x^{k-1} - a_2 x^{k-2} - \cdots - a_k = 0$$

Si $r_1, r_2, \cdots, r_p$ son las raíces distintas de este polinomio, y sus respectivas multiplicidades son $m_1, m_2, \cdots, m_p$, entonces

$$f(n) = \sum_{i=1}^{p} r_i^n . \sum_{j=0}^{m_i - 1} c_{ij} n^j$$

donde los c_{ij} son valores constantes. Dado que $\sum_{i=1}^{p} m_i = k$, existen exactamente k constantes, las cuales se pueden determinar a partir de k valores iniciales de $f(n)$, resolviendo un sistema de k ecuaciones lineales con k incógnitas.

9.2.1. Ejemplo 1

Resolver la siguiente ecuación de recurrencia:

$$T(n) = \begin{cases} 2 & \text{si } n \leq 1 \\ 6T(n-1) + 3T(n-2) & \text{si } n > 1 \end{cases}$$

Obtenemos su ecuación característica:

$$x^2 - 6x - 3 = 0$$

Obtenemos sus raíces:

$$\begin{aligned} x &= \frac{6 \pm \sqrt{(-6)^2 - 4(1)(-3)}}{2(1)} \\ &= \frac{6 \pm \sqrt{48}}{2} \\ &= \frac{6 \pm 4\sqrt{3}}{2} \\ &= 3 \pm 2\sqrt{3} \\ \Rightarrow \quad x_1 &= 3 + 2\sqrt{3} \qquad x_2 = 3 - 2\sqrt{3} \end{aligned}$$

Como sus raíces son distintas tenemos: $p = 2, r_1 = 3 + 2\sqrt{3}, m_1 = 1, r_2 = 3 - 2\sqrt{3}, m_2 = 1$. Por lo tanto:

$$T(n) = c_1 r_1^n + c_2 r_2^n = c_1(3 + 2\sqrt{3})^n + c_2(3 - 2\sqrt{3})^n$$

Para determinar c_1 y c_2 sustituimos $n = 0$ y $n = 1$ en la fórmula:

$$\begin{aligned} T(0) &= c_1 + c_2 = 2 \\ T(1) &= c_1(3 + 2\sqrt{3}) + c_2(3 - 2\sqrt{3}) = 2 \end{aligned}$$

Tenemos entonces que:

$$
\begin{aligned}
& c_2 = 2 - c_1 \\
\Rightarrow \quad & c_1(3 + 2\sqrt{3}) + (2 - c_1)(3 - 2\sqrt{3}) = 2 \\
\Rightarrow \quad & c_1(3 + 2\sqrt{3}) + 6 - 4\sqrt{3} - 3c_1 + 2\sqrt{3}c_1 = 2 \\
\Rightarrow \quad & c_1(3 + 2\sqrt{3} - 3 + 2\sqrt{3}) = 2 - 6 + 4\sqrt{3} \\
\Rightarrow \quad & c_1 = \frac{4\sqrt{3} - 4}{4\sqrt{3}} \\
\Rightarrow \quad & c_1 = 1 - \frac{1}{\sqrt{3}} \\
\Rightarrow \quad & c_2 = 1 + \frac{1}{\sqrt{3}}
\end{aligned}
$$

Finalmente,

$$
\begin{aligned}
& T(n) = \left(1 - \frac{1}{\sqrt{3}}\right)\left(3 + 2\sqrt{3}\right)^n + \left(1 + \frac{1}{\sqrt{3}}\right)\left(3 - 2\sqrt{3}\right)^n \\
\Rightarrow \quad & T(n) \in O\left(\left(3 + 2\sqrt{3}\right)^n\right)
\end{aligned}
$$