ANALISIS DE ALGORITMOS

Ing. Edgar C. Ruiz Lizama

Resumen

Presenta los fundamentos matemáticos para el análisis de algoritmos a fin de determinar su eficiencia, con el cuál se puede decidir por uno entre varios algoritmos en pugna.

Abstract

This article presents the mathematical foundations for the analysis of algorithms in order to determine its efficiency, which one can be chosen among several algorithms in conflict.

Introducción

Se dice que puede existir más de un camino para llegar a la solución de un problema; de ello se desprende que podrá existir más de un algoritmo para resolver un problema, luego el punto es; cual de estos algoritmos que conducen a la solución es el más eficiente. Este artículo intenta dar luces en un tópico muy interesante en la ciencia de la computación; el cual es, el *análisis de los algoritmos*.

Para realizar el análisis de algoritmos se tienen ciertos supuestos:

1. No se tiene en cuenta una máquina o tipo especifico en particular.
2. Se asumen unidades de tiempo constantes.

Esto es necesario puesto que, si una de las características de todo algoritmo es ser universal; entonces, es independiente del lenguaje y de la máquina donde se implementa.

El mejor algoritmo es aquel que consume menos cantidad de recursos sean estos memoria principal o secundaria (espacio) y tiempo de ejecución (velocidad).

Definición

Un algoritmo es un conjunto de instrucciones sencillas, claramente especificadas, que se debe seguir para resolver un problema.

Soporte matemático

El análisis requerido para estimar el uso de recursos de un algoritmo es una cuestión teórica y por lo tanto, necesita un marco formal. Para ello se presentan cuatro definiciones importantes.

- **Definición 1:** $T(n) = O(f(n))$ si existen constantes c y n_o tales $T(n) \leq cf(n)$ cuando $n \geq n_o$.
- **Definición 2:** $T(n) = \Omega(g(n))$ si existen constantes c y n_o tales $T(n) \geq cg(n)$ cuando $n \geq n_0$
- **Definición 3:** $T(n) = \theta(h(n))$ si y solo si $T(n) = O(h(n))$ y $T(n) = \Omega(h(n))$.
- **Definición 4:** $T(n) = o(p(n))$ si $T(n) = O(p(n))$ y $T(n) \neq \theta(p(n))$.

El objetivo de estas definiciones es establecer un orden relativo entre funciones. En el análisis de algoritmos se comparan sus *"tasas de crecimiento relativas"*. Al respecto ver la tabla Nº 1.

Ejemplo 1: Sean las funciones $1000n$ y n^2: Aunque $1000n$ es mayor que n^2 para valores pequeños de n, n^2 crece con una tasa mayor, y así; n^2 finalmente será la función mayor. El punto de cambio es en este caso, n = 1000.

La primera definición dice que finalmente existe un punto n_o pasado el cual $c.f(n)$ es siempre al menos tan grande como $T(n)$, de tal modo que si se ignoran los factores constantes, $f(n)$ es al menos tan grande como $T(n)$. En este caso se tiene $T(n) = 1000n$, $f(n) = n^2$, $n_0 = 1000$ y $c = 1$. Se podría usar también, $n_0 = 10$ y $c = 100$.

Así se puede decir que $1000n = O(n^2)$; es decir del orden n cuadrada. Esta notación se conoce como O grande (Big-Oh). Con frecuencia en nuestro idioma se suele decir "*O grande*".

Expresión Matemática	*Tasas Relativas de Crecimiento*
$T(n) = O(f(n))$	Crecimiento de $T(n)$ es $\leq$ crecimiento de $f(n)$
$T(n) = \Omega(f(n))$	Crecimiento de $T(n)$ es $\geq$ crecimiento de $f(n)$
$T(n) = \theta(f(n))$	Crecimiento de $T(n)$ es $=$ crecimiento de $f(n)$
$T(n) = o(f(n))$	Crecimiento de $T(n)$ es $<$ crecimiento de $f(n)$

Tabla Nº 1: Significado de las cuatro funciones de crecimiento.

Cuando se dice que $T(n) = O(f(n))$, se está garantizando que la función $T(n)$ crece a una velocidad no mayor que $f(n)$; así $f(n)$ es una *cota superior* de $T(n)$; como esto implica que $f(n) = \Omega(T(n))$, se dice $T(n)$ es una *cota inferior* de $f(n)$.

Algunas Reglas Importantes

Respecto a las cuatro definiciones dadas, se presentan tres reglas importantes a tener en cuenta:

Regla 1:
Si $T_1(n)=O(f(n))$ y $T_2(n)=O(g(n))$, entonces:
(a) $T_1(n) + T_2(n) = O(max((f(n)), (g(n)))$
(b) $T_1(n) * T_2(n) = O(f(n) * g(n))$

Regla 2:
Si $T(x)$ es un polinomio de grado n, entonces $T(x) = \theta(x^n)$

Regla 3:
$\log^k n = O(n)$ para cualquier k constante. Esto indica que los logaritmos crecen muy lentamente.

Prueba de la regla 1(a)
Por definición existen cuatro constantes c_1, c_2, n_1, n_2, tales que $T_1(n) \leq c_1 f(n)$ para $n \geq n_1$ y $T_2(n) \leq c_2 g(n)$ para $n \geq n_2$.

Sea $n_0=max(n_1, n_2)$. Entonces, para $n\geq n_0$, $T_1(n) \leq c_1 f(n)$ y $T_2(n) \leq c_2 g(n)$, de modo que $T_1(n) + T_2(n) <= c_1 f(n) + c_2 g(n)$. Sea $c_3 = max(c_1, c_2)$. Entonces:

$$\begin{aligned} T_1(n) + T_2(n) &\leq c_3 f(n) + c_3 g(n) \\ &\leq c_3(f(n) + g(n)) \\ &\leq 2c_3\, max(f(n),g(n)) \\ &\leq c.max(f(n),g(n)) \end{aligned}$$

para $c = 2c_3$ y $n \geq n_0$

$\therefore T_1(n) + T_2(n) = O(max((f(n)), (g(n)))$

La figura Nº 1 muestra los tiempos de ejecución para algunos algoritmos[1].

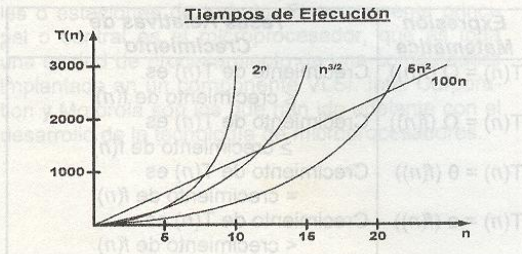

Figura Nº 1: Tiempos de ejecución para algunos algoritmos.

[1] Tomada de la referencia bibliográfica número 1.

La tabla Nº 2 muestra las tasas de crecimiento características en los tiempos de ejecución.

Función	*Nombre*
c	Constante
log n	Logarítmica
$\log^2 n$	Logarítmica cuadrada
n	Lineal
n log n	Lineal – logarítmica
n^2	Cuadrática
n^3	Cúbica
2^n	Exponencial

Tabla Nº 2: Tasas de crecimiento características

Reglas para el cálculo de los tiempos de ejecución

A continuación se presentan las reglas a tener en cuenta en el cálculo de los tiempos de ejecución.

Regla 1: Ciclos for

El tiempo de ejecución de un ciclo **for** es a lo más el tiempo de ejecución de las instrucciones que están en el interior del ciclo **for** (incluyendo las condiciones) por el número de iteraciones.

Regla 2: ciclos for anidados

Analizarlos de adentro hacia fuera. El tiempo de ejecución total de una proposición dentro del grupo de ciclos **for** anidados es el tiempo de ejecución de la proposición multiplicado por el producto de los tamaños de todos los ciclos **for**.

Ejemplo 2:

```
for i ← 1 to n do
    for j ← 1 to n do
        k ← k + 1
```

Claramente se ve que el tiempo de ejecución será $O(n^2)$

Regla 3: Proposiciones consecutivas

Simplemente se suman; lo cual significa que el máximo es el único que cuenta (Regla 1(a)).

Ejemplo 3:
El siguiente fragmento, tiene trabajo $O(n)$, seguido de trabajo $O(n^2)$ por lo que finalmente es $O(n^2)$.

```
for i ← 1 to n do
    a[i] ← 4*i
for i ← 1 to n do
    for j ← 1 to n do
```

$$a[i] \leftarrow a[i] + a[j] + i + j$$

Regla 4: if – else

En una proposición **if-then-else** el tiempo de ejecución nunca es más grande que el tiempo de ejecución de la condición más el mayor de los tiempos de ejecución entre la parte **then** y la parte **else**.

```
if <cond> then
        S1
else
        S2
```

Regla 5: Llamadas a funciones

Si hay llamadas a funciones, estas deben analizarse primero.

Ejemplo 4:
Sea el siguiente el algoritmo de ordenación por selección simple el cual utiliza la recursividad.

```
      seleccionsimple (A,n)
      begin
(1)      if n > 1 then
(2)         max = encontrar (A,n)
(3)         temp = A[n]
(4)         A[n] = A[max]
(5)         A[max] = temp
(6)         seleccionsimpl(A, n-1)
      end selecionsimple
```

donde el algoritmo para la función encontrar es:

```
        encontrar(A, n)
        begin
(2.1)   i = n
(2.2)   for j = 1 to n-1
(2.3)      if (A[j] > A[i]) then
(2.4)         i = j
           retornar i
        end encontrar
```

Se observa que el costo de *encontrar*(A,n) viene dado por el tamaño de n (línea (2.1) y (2.2)) por una constante a, más una constante b_1 (líneas 2.3 y 2.4).

Luego el proceso recursivo tiene un costo T(n), tal como sigue:

if $n > 1$ then
costo $an + b_1$ // línea (2)
costo b_2 // línea 3,4 y 5
costo $T(n-1)$ // línea (6)

totalizando estos costos se tiene:

$$\begin{aligned} T(n) &= an + b_1 + b_2 + T(n-1) \\ &= an + (b_1 + b_2) + T(n-1) \\ &= an + b + T(n-1), \quad n>1 \end{aligned}$$

Obsérvese que $b = b_1 + b_2$. Como T(1) es una constante, su costo es c; entonces la relación de recurrencia resultante es:

$$T(n) = \begin{cases} c, & n = 1 \\ an + b + T(n-1), & n > 1 \end{cases}$$

cuya solución es:

$$T(n) = \left(\frac{a}{2}\right)n^2 + \left(\frac{a}{2} + b\right)n + (c - a - b)$$

Luego aplicando la Regla 1(a) se concluye que:

$$T(n) = O(n^2)$$

Ejemplo 5:
El algoritmo para encontrar el factorial de un número puede plantearse de manera iterativa o de manera recursiva. Veamos el pseudocódigo para el planteamiento recursivo.

```
      int factorial (int n)
      begin
(1)        if n <= 1 then
(2)           retornar 1
           else
(3)           retornar n * factorial (n-1)
      end factorial
```

A priori podemos inferir que el tamaño apropiado para esta función es el valor de n. Sea T(n) el tiempo de ejecución para factorial(n). El tiempo de ejecución para las líneas (1) y (2) es O(1); mientras que, para la línea (3) es O(1) + T(n-1), luego para ciertas constantes c y d, la relación de recurrencia será:

$$T(n) = \begin{cases} c + T(n-1) & si \quad n > 1 \\ d & si \quad n \leq 1 \end{cases}$$

En general se puede suponer que $n > 2$; por lo que al desarrollar T(n-1) en la relación de recurrencia se obtiene:

$$T(n) = 2c + T(n-2) \quad si \quad n > 2$$

es decir T(n-1) = c + T(n-2), al sustituir n por $n-1$ en la relación de recurrencia. Así, es posible reemplazar T($n-1$) con c + T(n-2). De ello volviendo a usar dicha relación para obtener T(n–2) se obtiene:

$$T(n) = 3c + T(n-3) \quad si \quad n > 3$$

y así sucesivamente. En general:

$$T(n) = ic + T(n-i) \quad si \quad n > i$$

Por último, cuando i = n – 1, se obtiene:

$$T(n) = c(n-1) + T(1) = c(n-1) + d$$

En conclusión puede decirse que T(n) = O(n) para este algoritmo.

Nota: Este algoritmo recursivo para los factoriales en términos prácticos no es muy bueno puesto que para valores grandes de n el tamaño de los enteros que se van calculando exceden los límites de los enteros representables en el computador.

Ejemplo 6: El algoritmo de Euclides para hallar el máximo común divisor de dos números enteros mcd (m,n) viene dado por el siguiente pseudocódigo:

```
int mcd (int m, int n)
begin
        int resto
        while n > 0 do
        {
                resto ← m mod n
                m ← n
                n ← resto
        } // end while
        retornar m
end mcd
```

El algoritmo supone que m ≥ n. (Si n > m, la primera iteración del ciclo los intercambia).

El algoritmo funciona a base de calcular continuamente los restos hasta llegar a cero. La respuesta es el último valor distinto de cero. Así, si m = 3579 y n = 2970, la secuencia de restos es 609, 534, 75, 9, 3 y 0; por tanto mcd (3579, 2970) = 3. Como puede verse este algoritmo es muy rápido y eficiente.

A decir verdad, el tiempo de ejecución de este algoritmo depende de lo grande que sea la secuencia de residuos. Aunque *log n* parece ser una buena respuesta, no es en absoluto obvio que el valor del resto tenga que descender en un factor constante pues como se ve en la corrida el resto va de 609 a 534 entre la primera y segunda iteración. Sin embargo después de dos iteraciones, el resto es a lo más la mitad de su valor original. Esto podría demostrar que el número de iteraciones es a lo más 2*log n* = O(*log n*), y establecer así el tiempo de ejecución T(n). A fin de demostrar que esto es cierto se presenta el siguiente teorema:

Teorema 1:

Si $m > n$, entonces m mod $n < m/2$

Demostración

Existen dos casos a saber:

Caso 1:
Si $n \leq m/2$, entonces obviamente, como el resto es menor que n, el teorema se cumple.

Caso 2:
Si $n > m/2$, entonces n cabe en m una vez con un resto $m - n < m/2$, demostrando así el teorema.

Conclusiones

1. El análisis de los algoritmos es una cuestión teórica que nos permite decidir para fines prácticos de implementación si un algoritmo es más eficiente que otro.
2. Tal como se ha visto, un algoritmo es más eficiente que otro cuando consume una menor cantidad de recursos ya sea espacio(memoria) o tiempo (velocidad de ejecución).

Bibliografía

1. Aho Alfred / Hopcroft Jhon / Ullman Jefrey, " Estructura de datos y algoritmos". USA Addison – Wesley Iberoamericana S.A. 438 p (1988).
2. Brassard Glen / Bratley Paul, "Fundamentos de Algoritmia" España, Prentice Hall International. 579 p (1997).
3. Parker Alan, "Algorithms and Data Structures in C++". USA CRC Press: 257 p (1993).
4. Raffo Lecca Eduardo, "Algoritmos: Análisis y Diseño". Raffo Lecca Editores. Lima, 207p (1999).
5. Weiss Mark Allen., ¨Algoritmos y Estructuras de Datos¨ USA. Addison Wesley Iberoamericana. 489p (1995).
6. Weiss Mark Allen, "Algorithms Data Structures and Problems Solving with C++". USA. Addison – Wesley Longman, Inc. 820 p (1997).