Temas: Eficiencia. Programas eficientes. Análisis práctico.
Calculando el tiempo de ejecución de un programa.

Introducción:

Algunos programas son mejores que otros. Algunos son fáciles de usar. Algunos producen mejores o mayor cantidad de resultados. Algunos son fáciles de leer, modificar y mantener. Algunos requieren menor cantidad de tiempo o de memoria.

De todas estas medidas de calidad de un programa, en general, los programadores se preocupan por obtener un tiempo correcto a veces en perjuicio de los otros parámetros.

Cuando de eficiencia se trata es necesario saber cuál o cuáles parámetros deseamos optimizar y estos dependerán del problema en cuestión:

- Por ejemplo, en un sistema de reserva de pasajes aéreos, es importante el tiempo de respuesta para cada reservación a fin de evitar inconsistencias.
- Puede ocurrir que en algunas aplicaciones sea necesario manejar grandes volúmenes de información con lo cual, la memoria pasará a ser un recurso crítico. Por lo tanto, mi programa será eficiente, ya no por su tiempo de respuesta sino por su capacidad de manejar toda la información en forma simultánea.

Si bien es cierto que cuando hablamos de eficiencia hacemos hincapié en el tiempo de ejecución del algoritmo, estos dos conceptos no deberían tomarse como sinónimos. Eficiencia se refiere a la forma de administración de TODOS los recursos disponibles de los cuales el tiempo de CPU es uno de ellos.

Algorítmica eficiente:

La elección de algoritmos y estructuras de datos tiene un impacto fundamental en la eficiencia de un algoritmo. Sin embargo, hasta ahora, sólo los hemos estudiado como herramienta para la solución de problemas. Como forma de conocer qué algoritmos pueden alcanzar los requerimientos pedidos, es necesario analizar la eficiencia de los algoritmos que desarrollamos.

Hay dos formas de analizar algoritmos: empírica y teórica.

Para realizar un análisis empírico, es necesario introducir el algoritmo en un programa y medir los recursos consumidos. Este análisis tiene varias limitaciones: puede dar una información pobre de los recursos consumidos en caso de que el programa sea aplicado a pruebas de escritorio o si es transportado a otra computadora.

Para conducir un análisis teórico, es necesario establecer una medida intrínseca de la cantidad de trabajo realizado por el algoritmo. Esta medida nos permite comparar algoritmos y seleccionar la mejor implementación.

Tomemos un ejemplo sencillo para clarificar algunas ideas.
Considere la tarea de calcular el mínimo de tres números *a, b* y *c.* Hay cuatro formas de resolverlo:

1) Método 1:

```
m := a;
if b<m then m := b;
if c<m then m := c;
```

2) Método 2:

```
if a<=b then
        if  a<=c then
                        m := a
                else    m:= c
else
```

```
            if b<=c then
                        m := b
                  else  m := c
```

3) Método 3:

```
            if (a<=b) and (a<=c) then  m:= a;
            if (b<=a) and (b<=c) then  m:= b;
            if (c<=a) and (c<=b) then  m:= c;
```

4) Método 4:

```
            if (a<=b) and (a<=c) then
                        m:= a
                  else if  b<=c then
                              m := b
                        else  m := c;
```

Analicemos la eficiencia relativa de los cuatro métodos.

El método 1 realiza dos comparaciones y al menos una asignación. Note que si las dos comparaciones son verdaderas, se realizan tres asignaciones en lugar de una.

El método 2 utiliza también dos comparaciones y una sola asignación. Note que el trabajo de un algoritmo se mide por la cantidad de operaciones que realiza y no por la longitud del código.

El método 3 realiza seis comparaciones y una asignación. Note que aunque las primeras dos comparaciones den como resultado que a es el mínimo, las otras cuatro también se realizan.

El método 4, si bien requiere una única asignación, puede llegar a hacer tres comparaciones.

Dado que la diferencia entre 2 y 3 o 6 comparaciones no es muy importante, modifiquemos un poco el problema.

Supongamos que queremos hallar el menor (alfabéticamente) de tres strings en lugar del mínimo de tres números, donde cada uno de los strings contiene varios cientos de caracteres cada uno. Para comprender aun mejor la situación suponga que el lenguaje no provee un mecanismo para compararlos directamente sino que debe hacerlo letra por letra. Vemos entonces, que hacer dos comparaciones en lugar de tres o seis es importante.

Supongamos ahora que en lugar de hallar el menor de 3 números buscamos hallar el mínimo de *n* números, con *n* bastante grande.

La generalización del método 3 implica comparar cada número con los restantes llevándonos a realizar *n(n-1)* comparaciones, mientras que el método 1 sólo compara el número buscado una vez con cada uno de los *n* haciendo de esta forma *(n-1)* comparaciones, con lo cual, el método 3 requiere *n* veces más de tiempo que el método 1.

Como resultado de este análisis, el método 1 aparece como el más fácil de implementar y generalizar.

Si identificamos la unidad intrínseca de trabajo realizada por cada uno de los cuatro métodos (que en nuestro caso es la comparación de dos números), podemos determinar cuales son los métodos que realizan trabajo superfluo y cuales los que realizan el trabajo mínimo necesario para llevar a cabo la tarea pedida.

El método 4 no es un buen método en particular pero nos sirve para ilustrar un punto importante relacionado con el análisis de algoritmos. Cada uno de los otros métodos resulta en un número igual de comparaciones independientemente de los valores de a, b y c. En el método 4, el número de comparaciones puede variar.

Esta diferencia, se describe diciendo que el método 4 puede realizar dos comparaciones en el *mejor caso* y tres comparaciones en el *peor caso*. En general, estamos interesados en el comportamiento del algoritmo en el peor caso. No nos preocupa cuan rápido puede resolver el problema frente a los datos correctos sino qué es lo que puede ocurrir (cuan lento puede ser) frente a datos inadecuados.

También podemos estar interesados en el comportamiento promedio del algoritmo, lo que llamamos *caso promedio*, en el sentido de poder predecir cuánto es necesario esperar para que algoritmo realice una tarea. Es más fácil estimar el *caso peor* antes que el *caso promedio*, ya que este último implica enumerar y analizar todos los casos.

Por ejemplo, si queremos encontrar el mínimo entre tres números distintos, hay seis casos distintos correspondientes a las seis formas de ordenar los tres números., En los dos casos donde *a* es el mínimo, el método 4 realiza dos comparaciones y en los otros cuatro casos, realiza tres comparaciones. De esto se deduce que si todos los casos son igualmente probables, el método 4 realiza 8/3 comparaciones en promedio.

Programas Eficientes:

Los programas eficientes están basados en la eficiencia de los algoritmos. Podemos mejorar la eficiencia de tales programas, observando algunas guías simples relacionadas con la eficiencia del código.

Un punto importante a tener en cuenta es no repetir cálculos innecesarios. Es decir, podemos escribir:

```
t := x * x * x;
y := 1/(t-1) + 1/(t-2) + 1/(t-3) + 1/(t-4)
```

en lugar de

```
y:=1/(x*x*x-1) + 1/(x*x*x-2) +1/(x*x*x-3) + 1/(x*x*x-4)
```

no solo por el ahorro del código, sino porque la expresión *x*x*x* no se recalcula cuatro veces.

Esto se ve agravado si la expresión que se recalcula está dentro de un loop.

Por ej: debe escribirse:

```
t := x*x*x*;
y:=0;
for n:=1 to 20 do y := y + 1/(t-n);
```

en lugar de

```
y:=0;
for n:=1 to 20 do y := y + 1/(x*x*x-n);
```

Ver que ahorra la evaluación de *x*x*x* veinte veces.

Es necesario salvar resultados intermedios que pueden utilizarse en cálculos posteriores. Escribir

```
t := -x*x/2;
p := 1;
y := 1;
for n:= 1 to 20 do
        begin
                p := p * t;       { p = (-x*x/2)**n }
                y := y + p;       { y = 1 + t + ... + t**n }
        end
```

requiere una única multiplicación dentro del loop.

Los ejemplos anteriores muestran que no necesariamente los códigos más compactos son los más eficientes.

Calculando el tiempo de ejecución de un algoritmo:

Cuando intentamos decir algo respecto de la eficiencia de un algoritmo, debemos preguntarnos qué es lo que queremos medir. Como vimos hasta ahora, dependiendo del problema puede interesarnos medir un recurso u otro, pero por lo general, no siempre, el tiempo de ejecución es lo más importante.

Como ya dijimos, hay dos formas de estimar el tiempo de ejecución de un programa.
Una es empíricamente. Es decir, si queremos decidir entre dos algoritmos cuál es el más eficiente, una manera es directamente, ponerlos a correr y analizar el comportamiento de cada uno respecto de los recursos consumidos.

Este análisis empírico no tiene en cuenta algunos factores como:

- Velocidad de la máquina. Esto es, si los corremos en computadoras distintas, no podemos tener una medida de referencia.
- Juego de datos con el que ejecutamos el algoritmo. Los datos empleados pueden ser favorables a uno de los algoritmos pero no representar el caso general, con lo cual, no nos dará una idea general del comportamiento de los procesos.

Es necesario por lo tanto, disponer de un análisis un poco más teórico que nos permita medir el tiempo de respuesta. Para poder realizar esta tarea es necesario contar con un marco matemático mínimo.

Comencemos entonces con algunas definiciones:

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

La idea de estas definiciones es establecer un orden relativo entre funciones.
Dadas dos funciones, existen puntos donde una función es menor que otra, es decir que para algún n podremos decir, f(n) < g(n). Pero lo que en realidad es importante no es ver puntos aislados sino analizar la *velocidad de crecimiento* de cada función a fin de poder obtener una relación entre ambas.
La importancia de esta medida se verá al aplicarla al análisis de algoritmos.

Aunque $1000\,n$ es mayor que n^2 para valores pequeños de n, n^2 crece más rápido, con lo cual n^2 podría ser eventualmente la función más grande. En este caso, esto ocurre para $n \geq 1000$. La primer definición dice que eventualmente hay algún punto n_0 partir del cual $c*f(n)$ es al menos tan grande como $T(n)$, e ignorando los factores constantes, $f(n)$ es como mínimo tan grande como $T(n)$.

En nuestro caso, $T(n) = 1000n$, $f(n) = n^2$, $n_0=1000$ y $c=1$. También podemos utilizar $n_0 = 10$ y $c=100$. Con lo cual podemos decir que $1000n = O(n^2)$ (orden n-cuadrado).

Si utilizamos los operadores de desigualdad para compara las velocidades de crecimiento, podemos ver que la primera dice que la velocidad de crecimiento de $T(n)$ en menor o igual que la de $f(n)$. La segunda definición, $T(n)= \Omega(g(n))$ (se pronuncia "Omega de g(n)"),dice que la velocidad de crecimiento de $T(n)$ es mayor o igual que la de $g(n)$. La tercer definición $T(n) = \theta(h(n))$ *(se pronuncia Theta de h(n))* dice que la velocidad de crecimiento de $T(n)$ es igual a la de $h(n)$. La última definición $T(n)=o(p(n))$ dice que la velocidad e crecimiento de $T(n)$ es menor que la de $p(n)$. Esta última se diferencia de $O(p(n))$ en que las velocidades de crecimiento no pueden ser iguales.

Resumiendo, cuando decimos de $T(n)=O(f(n))$, estamos garantizando que la función $T(n)$ no crece más rápido que $f(n)$, es decir que $f(n)$ es un *límite superior* para $T(n)$. Esto implica que $f(n) = \Omega(T(n))$, de donde decimos que *T(n) es un límite inferior* para $f(n)$.

Ejemplos:
1) n^3 crece más rápido que n^2 , por lo tanto, podemos afirmar que $n^2 = O(n^3)$ o que $n^3 = \Omega(n^2)$.

2) $f(n) = n^2$ y $g(n) = 2n^2$ crecen a la misma velocidad, por lo tanto ambas expresiones, $f(n)=O(g(n))$ y $g(n)=O(f(n))$ son verdaderas. Cuando dos funciones crecen a la misma velocidad, la decisión de cuando representar o no esta relación mediante $\theta()$ puede depender del contexto particular.

Intuitivamente, si $g(n) = 2n^2$, entonces $g(n)=O(n^4)$, $g(n)=O(n^3)$ y $g(n)=O(n^2)$ son todas técnicamente correctas, pero obviamente la última es la mejor respuesta.

Afirmando $g(n) = \theta(n^2)$ decimos no sólo que $g(n) = O(n^2)$ sino también que el resultado es tan bueno (cercano) como sea posible.

Volvamos entonces a nuestro problema de calcular el tiempo de ejecución de un algoritmo.

Un ejemplo sencillo

He aquí un fragmento de programa que calcula $\sum i^3$ con i=1..n

```
        function sum( n:Integer ) : Integer;
        Var j, SumaParcial : Integer;
        Begin
{1}             SumaParcial := 0;
{2}             for j := 1 to n do
{3}                     SumaParcial := SumaParcial + j*j*j;
{4}             Sum := SumaParcial;
        End;
```

El análisis de este programa es simple.

- Las declaraciones no consumen tiempo.
- Las líneas {1} y {4} implican una unidad de tiempo cada una.
- La línea {3} consume 3 unidades de tiempo cada vez (dos multiplicaciones, una suma y una asignación} y es ejecutada *n* veces dando un total de 4*n* unidades.
- La línea {2} tiene un costo oculto de inicializar j, testear si $j \leq n$ e incrementar j. El costo total es 1 por la inicialización, *n*+1 por todos los testeos y 2*n* por todos los incrementos, lo que da 3*n*+2.

Si ignoramos el costo de la invocación y retorno de la función, el total es de 7*n*+4. Por lo tanto, decimos que esta función es de *O(n)*.

Si tuviéramos que realizar todo este análisis por cada segmento de código, nuestro trabajo sería insostenible. Sin embargo, hay una serie de atajos que pueden tomarse sin afectar la respuesta final. Por ejemplo, la línea {3} es obviamente de *O(1)* no importa lo que haya en su interior, es algo que se ejecuta *n veces*. Lo mismo ocurre con la línea {1}, es insignificante respecto de *n*.

Existen, por lo tanto, ciertas reglas que podemos tener en cuenta para facilitar nuestra tarea:

Reglas Generales:

Regla 1: Para lazos incondicionales

El tiempo de ejecución de un loop incondicional es a lo sumo el tiempo de ejecución de las sentencias que están dentro del lazo (incluyendo testeos) multiplicada por la cantidad de iteraciones que se realizan.

Regla 2: Para lazos incondicionales anidados

Realizar el análisis de adentro hacia a fuera. El tiempo total de ejecución de un bloque dentro de lazos anidados es el tiempo de ejecución del bloque multiplicado por el producto de los tamaños de todos los lazos incondicionales.

Regla 3: If - Then - Else

Dado un fragmento de código de la forma

```
If condicion    Then
                        S1
                Else
```

S2

el tiempo de ejecución no puede ser superior al tiempo del testeo más el mayor tiempo de ejecución entre el tiempo de S1 y el tiempo de S2.

Regla 4: Para sentencias consecutivas

Si un fragmento de código está formado por dos bloque uno con tiempo $T_1(n)$ y otro $T_2(n)$, el tiempo total es el máximo de los dos anteriores, es decir:

Si $T_1(n) = O(f(n))$ y $T_2(n) = O(g(n))$, $T_1(n) + T_2(n) = máx(O(f(n)), O(g(n)))$

Analicemos un poco más esta última afirmación:

Vemos que por definición, $T_1(n) = O(f(n))$ y $T_2(n) = O(g(n))$, de donde existen cuatro constantes c_1, c_2, n_1 y 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 donde $T_1(n) + T_2(n) \leq c_1 f(n) + c_2 g(n)$.

Sea $c_3 = \max(c_1, c_2)$, entonces

$$
\begin{aligned}
T_1(n) + T_2(n) &\leq c_3 f(n) + c_3 g(n) \\
&\leq c_3 \, (f(n) + g(n)) \\
&\leq 2c_3 \, \max(f(n), g(n)) \\
&\leq c \, \max(f(n), g(n)) \qquad \text{para } c = 2c_3 \text{ y } n \geq n_0
\end{aligned}
$$

A modo de ejemplo, vemos el siguiente fragmento de código formado por un bloque de $O(n)$ seguido de otro de $O(n^2)$ dando como resultado un fragmento de $O(n^2)$

```
for j := 1 to n do
        { algo para hacer de O(1) };

for j:=1 to n do
    for k:=1 to n do
        { algo para hacer de O(1) };
```

Podemos ver que utilizando estas reglas el resultado final puede ser sobreestimado pero nunca subestimado.

Si se tiene un programa con módulos que no son recursivos, es posible calcular el tiempo de ejecución de los distintos procesos, uno a la vez, partiendo de aquellos que no llaman a otros.

Debe haber al menos un módulo con esa característica.

Después puede evaluarse el tiempo de ejecución de los procesos que sólo tienen módulos que no tienen llamadas, usando los tiempos de ejecución de los procesos llamados que ya hemos evaluado antes.

Se continúa la tarea evaluando el tiempo de ejecución de cada proceso después de haber evaluado los tiempos correspondientes de los módulos que llama.