Análisis de Algoritmos (Eficiencia Algorítmica)

Tema IX: Complejidad en los algoritmos. Recursos que requiere un algoritmo. Operación elemental. Principio de Invarianza. Análisis del Caso Mejor, Peor y Medio. Eficiencia Algorítmica. Formas de optimización. Notación asintótica. Ecuaciones de recurrencia. NP Completitud. Clase P y NP.

El concepto de eficiencia

- Pensar en la optimización de un algoritmo requiere analizar previamente su eficiencia.

- La utilización que se hace desde el algoritmo de los recursos del sistema físico donde se ejecuta.

- Eficiencia se refiere a la forma de administración de todos los recursos disponibles en el sistema, de los cuales el tiempo de procesamiento es uno de ellos.

Definición: Un algoritmo es eficiente si realiza una administración correcta de los recursos del sistema en el cual se ejecuta

Propiedades de un Algoritmo

- Especificación precisa de la entrada
- Especificación precisa de cada instrucción
- Exactitud, corrección
- Etapas bien definidas y concretas
- Numero finitos de pasos
- Un algoritmo debe terminar
- Descripción del resultado o efecto

Una leyenda ajedrecística

Mucho tiempo atrás, el espabilado visir Sissa ben Dahir inventó el juego del ajedrez para el rey Shirham de la India. El rey ofreció a Sissa la posibilidad de elegir su propia recompensa. Sissa le dijo al rey que podía recompensarle en trigo o bien con una cantidad equivalente a la cosecha de trigo en su reino de dos años, o bien con una cantidad de trigo que se calcularía de la siguiente forma:

- un grano de trigo en la primera casilla de un tablero de ajedrez,
- más dos granos de trigo en la segunda casilla,
- más cuatro granos de trigo en la tercera casilla,
- y así sucesivamente, duplicando el número de granos en cada casilla, hasta llegar a la última casilla.

El rey pensó que la primera posibilidad era demasiado cara mientras que la segunda, medida además en simples granos de trigo, daba la impresión de serle claramente favorable.

Así que sin pensárselo dos veces pidió que trajeran un saco de trigo para hacer la cuenta sobre el tablero de ajedrez y recompensar inmediatamente al visir.

¿Es una buena elección?

El número de granos en la primera fila resultó ser:

$$2^0 + 2^1 + 2^2 + 2^3 + 2^4 + 2^5 + 2^6 + 2^7 = 255$$

La cantidad de granos en las dos primeras filas es:

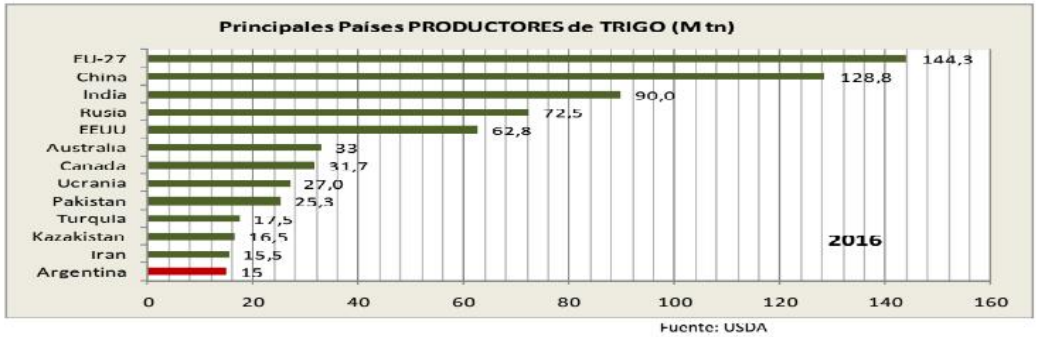
$$\sum_{i=0}^{15} 2^i = 2^{16} - 1 = 65\,535$$

Al llegar a la tercera fila el rey empezó a pensar que su elección no había sido acertada, pues para llenar las tres filas necesitaba

$$\sum_{i=0}^{23} 2^i = 2^{24} - 1 = 16\,777\,216$$

granos, que pesan alrededor de 600 kilos ...

Endeudado hasta las cejas



- En efecto, para rellenar las 64 casillas del tablero hacen falta

$$\sum_{i=0}^{63} 2^i = 2^{64} - 1 = 18\,446\,744\,073\,709\,551\,615 \approx 1{,}84 * 10^{19}$$

granos, cantidad equivalente a las cosechas mundiales actuales de 1000 años!!.

- La función $2^n - 1$ (exponencial) representa el número de granos adeudados en función del número n de casillas a rellenar. Toma valores desmesurados aunque el número de casillas sea pequeño.
- El coste en tiempo de algunos algoritmos expresado en función del tamaño de los datos de entrada es también exponencial. Por ello es importante estudiar el coste de los algoritmos y ser capaces de comparar los costes de algoritmos que resuelven un mismo problema.

Producción Mundial de Arroz 2016/2017 será de 483 millones de toneladas

Los orígenes del análisis de eficiencia

- La técnica que se utilizaba en los primeros años de la programación para comparar la eficiencia de distintos algoritmos consistía en ejecutarlos para datos diferentes y medir el tiempo consumido.
- Dado que las máquinas y los lenguajes eran dispares, y que el tiempo de ejecución depende no solo del tamaño sino también del contenido de los datos, resultaba muy difícil comparar tales resultados.
- El primer estudio serio sobre la eficiencia de los algoritmos se lo debemos a Daniel Goldenberg del MIT. En 1952 realizó un análisis matemático del número de comparaciones necesarias, en el mejor y en el peor caso, de cinco algoritmos distintos de ordenación.
- La tesis doctoral de Howard B. Demuth de la Universidad de Stanford estableció en 1956 las bases de lo que hoy llamamos *análisis de la complejidad de los algoritmos*.

Ideas esenciales del análisis de complejidad actual

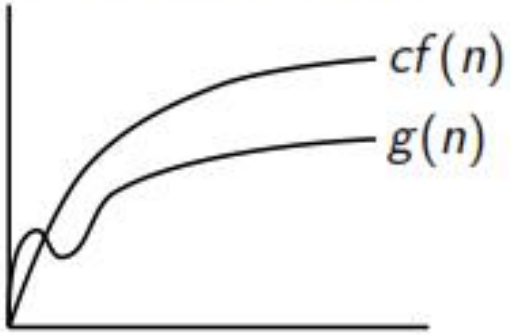
- Lo primero que se decide es si se desea un estudio del *caso mejor*, del *caso peor* o del *caso promedio* del algoritmo. El análisis más frecuente y el más sencillo es el del caso peor.
- Después, se decide cuál es el parámetro n que va a medir el tamaño del problema. Para cada problema, habrá que determinar qué significa dicha n.
- Finalmente, se estudia la función de dependencia $f(n)$ entre el tamaño n del problema y el número de pasos elementales del algoritmo.
- El objetivo es clasificar dicha función en una determinada *clase de complejidad*. Para ello, se ignoran las constantes multiplicativas y aditivas y se desprecian los sumandos de menor orden.
- La consecuencia es que funciones tales como $f(n) = 10n^2 + 32$ y $g(n) = \frac{1}{2}n^2 - 123n - 15$ se clasifican en la misma clase que $h(n) = n^2$. Dado que n^2 es la función más sencilla de dicha clase, la propia clase se denota $O(n^2)$, expresión que ha de entenderse como el conjunto de funciones del orden de n^2.

Medidas asintóticas

- El único factor determinante en el coste de un algoritmo es el tamaño n de los datos de entrada. Por eso, trabajamos con funciones $f : \mathbb{N} \longrightarrow \mathbb{R}_0^+$.
- No nos importa tanto las funciones concretas, sino la forma en la que crecen. El conjunto de las funciones del orden de $f(n)$ se define como:

$$O(f(n)) = \{g : \mathbb{N} \longrightarrow \mathbb{R}_0^+ \mid \exists c \in \mathbb{R}^+, n_0 \in \mathbb{N}. \forall n \geq n_0. g(n) \leq cf(n)\}$$

Diremos que una función g es del orden de $f(n)$ cuando $g \in O(f(n))$.

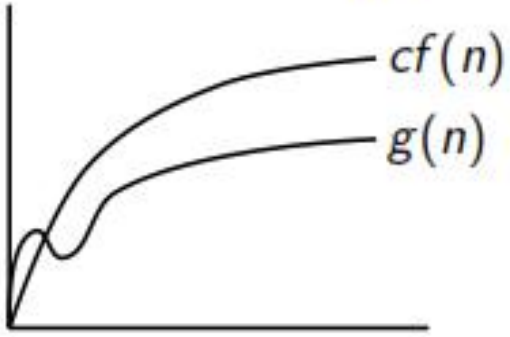


Medidas asintóticas

- El único factor determinante en el coste de un algoritmo es el tamaño n de los datos de entrada. Por eso, trabajamos con funciones $f : \mathbb{N} \longrightarrow \mathbb{R}_0^+$.
- No nos importa tanto las funciones concretas, sino la forma en la que crecen. El conjunto de las funciones del orden de $f(n)$ se define como:

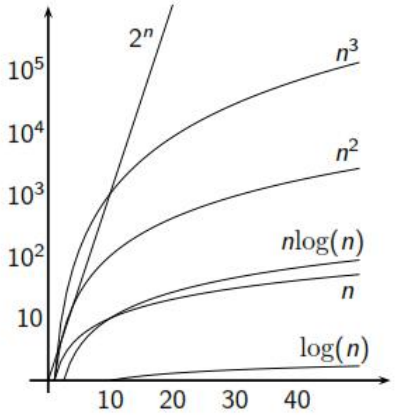
$$O(f(n)) = \{g : \mathbb{N} \longrightarrow \mathbb{R}_0^+ \mid \exists c \in \mathbb{R}^+, n_0 \in \mathbb{N}. \forall n \geq n_0 . g(n) \leq cf(n)\}$$

Diremos que una función g es del orden de $f(n)$ cuando $g \in O(f(n))$.



- Decimos que el conjunto $O(f(n))$ define un orden de complejidad.
- Si el tiempo de un algoritmo está descrito por una función $T(n) \in O(f(n))$ diremos que el tiempo de ejecución del algoritmo es del orden de $f(n)$.

Ordenes de complejidad en escala semi-logarímica



Jerarquía de órdenes de complejidad

$$\underbrace{\underbrace{O(1) \subset O(\log n) \subset O(n) \subset O(n \log n) \subset O(n^2)}_{\text{razonables en la práctica}} \subset \ldots \subset O(n^k) \subset \ldots \subset O(2^n)}_{\text{tratables}} \subset \underbrace{O(n!)}_{\text{intratables}}$$

La notación $O(f)$ nos da una cota superior del tiempo de ejecución $T(n)$ de un algoritmo. Normalmente estaremos interesados en la menor función $f(n)$ tal que

$$T(n) \in O(f(n)).$$

Ejemplos: cotas superiores hay muchas, pero algunas son poco informativas

$$3n \in O(n)$$
$$3n \in O(n^2)$$
$$3n \in O(2^n)$$

Comparación de diversos órdenes de complejidad

- Sabiendo que el tiempo transcurrido desde el Big-Bang es de $1,4 \times 10^8$ siglos, la siguiente tabla ilustra la importancia de que el coste del algoritmo sea pequeño (ms = milisegundos, s = segundos, m = minutos, h = horas, d = dias, a = años, sig = siglos).

n	$\log_{10} n$	n	$n \log_{10} n$	n^2	n^3	2^n
10	1 *ms*	10 *ms*	10 *ms*	0,1 *s*	1 *s*	1,02 *s*
10^2	2 *ms*	0,1 *s*	0,2 *s*	10 *s*	16,67 *m*	$4,02 * 10^{20}$ *sig*
10^3	3 *ms*	1 *s*	3 *s*	16,67 *m*	11,57 *d*	$3,4 * 10^{291}$ *sig*
10^4	4 *ms*	10 *s*	40 *s*	1,16 *d*	31,71 *a*	$6,3 * 10^{3000}$ *sig*
10^5	5 *ms*	1,67 *m*	8,33 *m*	115,74 *d*	317,1 *sig*	$3,16 * 10^{30093}$ *sig*
10^6	6 *ms*	16,67 *m*	1,67 *h*	31,71 *a*	317 097,9 *sig*	$3,1 * 10^{301020}$ *sig*

- Es un error pensar que basta esperar algunos años para que algoritmos tan costosos se puedan ejecutar con un coste en tiempo razonable.

Comparación de diversos órdenes de complejidad (2)

- Supongamos que tenemos 6 algoritmos diferentes para resolver el mismo problema tales que su menor cota superior está en $O(\log n)$, $O(n)$, $O(n \log n)$, $O(n^2)$, $O(n^3)$ y $O(2^n)$.
- Supongamos que para un tamaño $n = 100$ todos tardan 1 hora en ejecutarse.
- ¿Qué ocurre si duplicamos el tamaño de los datos?

$T(n)$	$n = 100$	$n = 200$
$k_1 \cdot \log n$	$1h.$	$1,15h.$
$k_2 \cdot n$	$1h.$	$2h.$
$k_3 \cdot n \log n$	$1h.$	$2,3h.$
$k_4 \cdot n^2$	$1h.$	$4h.$
$k_5 \cdot n^3$	$1h.$	$8h.$
$k_6 \cdot 2^n$	$1h.$	$1,27 \cdot 10^{30}h.$

(tiempo desde el Big-Bang $\approx 10^{14}$ horas)

Comparación de diversos órdenes de complejidad (y 3)

¿Qué ocurre si duplicamos la velocidad del computador? O lo que es lo mismo, ¿qué ocurre si duplicamos el tiempo disponible?

$T(n)$	$t = 1h.$	$t = 2h.$
$k_1 \cdot \log n$	$n = 100$	$n = 10000$
$k_2 \cdot n$	$n = 100$	$n = 200$
$k_3 \cdot n \log n$	$n = 100$	$n = 178$
$k_4 \cdot n^2$	$n = 100$	$n = 141$
$k_5 \cdot n^3$	$n = 100$	$n = 126$
$k_6 \cdot 2^n$	$n = 100$	$n = 101$

La algoritmia se ocupa de una parte de los algoritmos

- Para comparar dos algoritmos que resuelven el mismo problema basta calcular sus respectivas clases de complejidad: si coinciden, diremos que los algoritmos son igualmente buenos; en caso contrario, será mejor el que pertenezca a la clase más baja en la jerarquía.
- La algoritmia se ocupa fundamentalmente de encontrar algoritmos polinomiales para los problemas computables. Dentro de los polinomiales, las clases más codiciadas son, por ese orden, $O(1)$, $O(\log n)$, $O(n)$ y $O(n \log n)$.
- La primera no es posible para la gran mayoría de los problemas porque, dada su naturaleza (e.g. búsquedas, recorridos, etc.), el tiempo crece necesariamente con el número n de objetos considerados.
- La segunda incluye algoritmos excepcionalmente buenos (e.g. búsquedas en estructuras de datos) cuyo tiempo crece muy despacio.
- Las otras dos se comportan como un usuario no informático intuitivamente espera de los ordenadores: e.g. si el tamaño del fichero a comprimir se hace doble o triple, espera que el tiempo de compresión se duplique o triplique.
- En la práctica, son muy infrecuentes los algoritmos polinomiales con complejidad superior a $O(n^5)$.

Eficiencia y Exactitud

El diseño de un algoritmo para ser implementado por un programa de computadora debe tener dos características:

- Que sea fácil de entender, codificar y depurar
- Que consiga la mayor eficiencia a los recursos de la computadora

Entonces …

La Eficiencia de un algoritmo es la propiedad mediante la cual un algoritmo debe alcanzar la solución al problema en el tiempo mas corto posible y/o utilizando la cantidad mas pequeña posible de recursos físicos, y que sea compatible con su exactitud y corrección.

A partir del Análisis del Algoritmo medimos su eficiencia (permite medir la dificultad inherente de un problema).

Algoritmia o Algorítmica es el estudio sistemático de las técnicas fundamentales utilizadas para diseñar y analizar algoritmos eficientes. (Brassard y Bratley – 1988; 1997).

La eficiencia como factor espacio – tiempo debe estar estrechamente relacionada con la buena calidad, el funcionamiento optimo y la facilidad de mantenimiento del programa.

Análisis de la eficiencia de un algoritmo

- Tiempo de ejecución

  Desde este punto se consideran mas eficientes aquellos algoritmos que cumplan con la especificación del problema en el menor tiempo posible. En este caso el recurso a optimizar es el tiempo de procesamiento. (Ej.: reserva de pasaje; monitoreo de señales en tiempo real, el control de alarmas, etc.).

- Uso de la memoria

  Serán eficientes aquellos algoritmos que utilicen estructuras de datos adecuadas de manera de minimizar la memoria ocupada. Se pone énfasis en el volumen de la información a manejar en la memoria (Ej.: manejo de base de datos, procesamiento de imágenes en memoria, reconocimiento de patrones, etc.).

Análisis de algoritmos según tiempo de ejecución

- Para poder medir la eficiencia de un algoritmo, desde el punto de vista de su tiempo de ejecución, es fundamental contar con una medida del trabajo que realiza. Esta medida permitirá comparar los algoritmos y seleccionar, de todas las posibles, la mejor implementación.

- Básicamente en los algoritmos hay dos operaciones elementales: las comparaciones de valores y las asignaciones.

- Según el ordenador que se utilice se tendrá en cuenta el tiempo de cada clase de operaciones.

Se considera la tarea de calcular el mínimo de tres números a, b y c y se analizan cuatros formas de resolverlo:

Método 1

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

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
```

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;
```

Método 4

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

Análisis de Algoritmos

Método 1

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

Realiza dos comparaciones y, al menos, una asignación. Si las dos comparaciones son verdaderas, se realizan tres asignaciones en lugar de una

Análisis de Algoritmos

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
```

- Utiliza también dos comparaciones y una sola asignación.
- El trabajo de un algoritmo se mide por la cantidad de operaciones que realiza y no por la longitud del código.

Análisis de Algoritmos

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;
```

- Realiza seis comparaciones y una asignación, ya que aunque las primeras dos comparaciones den como resultado que a es el mínimo, las otras cuatro también se realizan.

Método 4

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

- Requiere una sola asignación. Pero puede llegar a hacer tres comparaciones.

Análisis de Algoritmos

- La diferencia entre 2, 3 o 6 comparaciones puede parecer poco importante.

- Hagamos una modificación al problema: Hallar el menor (alfabéticamente) de tres strings o cadenas en lugar del mínimo de tres números. Donde cada uno de los sgtrings contenga 200 caracteres. Suponemos que el lenguaje no provee un mecanismo para compararlos directamente, sino que debe hacerse letra por letra.

Análisis de Algoritmos

- Se puede ver:

- Las comparaciones se pueden convertir en 400, 600 o 1200 (incremento potencial = importante en el tiempo de ejecución).

Análisis de Algoritmos

Si en lugar de hallar el menor de 3 strings, se requiere hallar el mínimo de n strings, con n grande.

- El método 3 implica comparar cada numero con los restantes, lo cual lleva a realizar n(n-1) comparaciones, mientras que el método 1 solo compara el número buscado una vez con cada uno de los n elementos y haciendo de esta forma (n-1) comparaciones.

- El método 3 requiere de mucho mas tiempo que el método 1 ya que realiza n comparaciones por cada elemento.

- El método 1 aparece como el más fácil de implementar y generalizar.

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;
```

Método 1

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

Análisis de Algoritmos

- Si se identifica la unidad intrínseca de trabajo realizada por cada uno de los cuatro métodos, es posible determinar cuales son los métodos que realizan trabajo superfluo y cuales los que realizan el trabajo mínimo necesario para llevar a cabo la tarea.

Análisis de Algoritmos

- El método 4 no es un buen método en particular, pero sirve para ilustrar un aspecto importante relacionado con el análisis de algoritmos. Cada uno de los otros métodos presenta un número igual de comparaciones independientemente de los valores de a, b y c. En el método 4, el número de comparaciones puede variar, dependiendo de los datos.
- Esta diferencia, se describe diciendo que el método 4 puede realizar dos comparaciones en el mejor caso, y tres comparaciones en el peor caso.

Método 4

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

Análisis de Algoritmos

- En general se tiene interés en el comportamiento del algoritmo en el peor caso o en el caso promedio. En la mayoría de las aplicaciones no es importante cuan rápido puede resolver el problema frente a los datos organizados favorablemente, sino ocurre frente a datos adversos (caso peor) o en el caso estadístico (caso promedio).

(Ej.: si se desea encontrar el mínimo de tres números distintos, hay seis casos diferentes correspondientes a los seis ordenes relativos en que pueden aparecer 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

- No repetir cálculos innecesarios

Ej.: puede escribirse:

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

en lugar de:

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

Si esta expresión que se recalcula está dentro de un lazo repetitivo de 1000 veces, la ejecución reiterada de 2 * x * t, significa 4000 operaciones redundantes.

Programas Eficientes

Debe escribirse

```
t:=x*x*x;
y:= 0;
for n:= 1 to 2000 do
      y := y + 1/(t – n);
```

Se evita calcular x*x*x dos mil veces.

en lugar de:

```
for n:= 1 to 2000 do
      y := y + 1/(x*x*x – n);
```

Eficiencia de bucles

En general el formato de eficiencia se puede expresar mediante una funcion:

$$F(n) = \text{eficiencia}$$

Es decir la eficiencia del algoritmo se exmina como una funcion del numero de elementos a ser procesado.

Bucles lineales

- En los bucles lineales se repiten las sentencias del cuerpo del bucle un numero determinado de veces m, que determina la eficiencia del mismo. Normalmente en los algoritmos los bucles son el termino dominante en cuanto a la eficiencia del mismo.

```
I := 1
mientras  (I <= n)
        codigo de la aplicación
        I := I + 1
fin – mientras
```

El numero de iteraciones es directamente proporcional al factor del bucle, n. Como la eficiencia es directamente proporcional al numero de iteraciones la funcion que expresa la eficiencia es:

$$F(n) = n$$

Cuantas veces se repite el cuerpo del bucle en el siguiente codigo?

```
I := 1
mientras  (I <= n)
        codigo de la aplicación
         I := I + 2
fin – mientras
```

$$F(n) = n / 2$$

Bucles Algoritmicos

```
I := 1
Mientras ( I < 1000)
    codigo de la aplicación
    I := I * 2
Fin-mientras
```

```
I := 1000
Mientras ( I >= 1)
    codigo de la aplicación
    I := I / 2
Fin-mientras
```

Bucle multiplicar

Iteracion Valor de I

Bucle de dividir

Iteracion Valor de I

Bucle de multiplicar		Bucle de dividir	
Iteracion	Valor de I	Iteracion	Valor de I
1	1	1	1000
2	2	2	500
3	4	3	250
4	8	4	125
5	16	5	62
6	32	6	31
7	64	7	15
8	128	8	7
9	256	9	3
10	512	10	1
salida	1024	Salida	0

Bucle algoritmicos

- En ambos casos se ejecutan 10 iteraciones. La razon es que en cada iteracion el valor de I se dobla en el bucle de multiplicar y se divide a la mitad en el bucle de dividir. Por consiguiente, el numero de iteraciones es una funcion del multipicador o divisor, en este caso: 2.

- Bucle de multiplicador 2 ** iteraciones < 1000
- Bucle de division 1000 / 2 ** iteraciones >= 1

Generalizando: F(n) = [log2 n]

Bucles anidados

Iteraciones = iteraciones del bucle externo * bucle interno

Existen tres tipos …

Formulas de eficiencia en bucles anidados	
LINEAL LOGARTIMICA	F(n) = [n * log2 n]
DEPENDIENTE CUADRATICA	F(n) = n *((n + 1) / 2)
CUADRATICA	F(n) = n ** 2

Tiempo de ejecución (resumen …)

- Análisis teórico: se busca obtener una medida del trabajo realizado por el algoritmo a fin de obtener una estimación teórica de su tiempo de ejecución. Básicamente se calcula el número de comparaciones y de asignaciones que requiere el algoritmo. Los análisis similares realizados sobre diferentes soluciones de un mismo problema permiten estimar cual es la solución más eficiente.

- Análisis empírico: se basa en la aplicación de juegos de datos diferentes a una implementación del algoritmo, de manera de medir sus tiempos de respuestas. La aplicación de los mismos datos a distintas soluciones del mismo problema presupone la obtención de una herramienta de comparación entre ellos. El análisis empírico tiene la ventaja de ser muy fácil de implementar, pero no tiene en cuenta algunos factores como:
  - La velocidad de la maquina, esto es, la ejecución del mismo algoritmo en computadores diferentes produce distintos resultados. (no es posible tener una medida de referencia).
  - Los datos con los que se ejecuta el algoritmo, ya que los datos empleados pueden ser favorables a uno de los algoritmos, pero pueden no representar el caso general, con lo cual las conclusiones de la evaluación pueden ser erróneas.

Conclusión: es valiosos realizar un análisis teórico que permita estimar el orden de tiempo de respuesta, que sirva como comparación relativa entre las diferentes soluciones algorítmicas, sin depender de los datos de experimentación.

Tiempo de ejecución

Definición

El tiempo de ejecución T(n) de un algoritmo se dice de orden f(n) cuando existe una función matemática f(n) que acota a T(n).

T(n) = O(f(n)) si existen constantes “c” y “n0” tales que T(n) >= c f(n) cuando n>= n0.

Tiempo de ejecución

- La definición anterior establece un orden relativo entre las funciones del tiempo de ejecución de los algoritmos T1(n) y T2(n).

- Ej. T1(n) = O(2*n) y T2(n) = O(4*n*n)

  para n > 5, resulta que T1 es mas eficiente que T2

Tiempo de ejecución

- En cuanto a la velocidad de crecimiento.
- Ej.: T(n) = 1000 n con f(n) = n**2.

Para valores pequeños de n, 1000 n es mayor que n**2. Pero n**2 crece mas rápido, con lo cual n**2 podría ser eventualmente la función mas grande.

Esto ocurre p/n>= 1000

Tiempo de ejecución

Resumiendo:

Al decir que T(n) = O(f(n)), se esta garantizando que la función T(n) no crece más rápido que f(n), es decir que f(n) es un limite superior (cota) para T(n).

Ej.: n**3 crece mas rápido que n**2, por lo tanto se puede afirmar que n**2 = O(n**3).

```
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;
```

Las declaraciones no consumen tiempo.

Las líneas {1} y {4} implican una unidad de tiempo c/u.

La línea {3} consume 4 unidades de tiempo y se ejecuta n veces. Da un total de 4n unidades.

La línea {2} tiene un costo encubierto: inicializar (1); testear si j <=n (n+1) e incrementar j (n). Lo que nos da: 2n+2 (1+ n + 1 + n).

Si se ignora el costo de la invocación de la función y el retorno, el total es de: 6n + 4, es una función de O(n).

Reglas generales

Regla 1: Para lazos incondicionales.

El tiempo de ejecución de un lazo incondicional es, a lo sumo, el tiempo de ejecución de las sentencias que están dentro del lazo, incluyendo testeos, multipicada por cantidad de iteraciones que se realizan.

Regla 2: Para lazos incondicionales anidados.

Se debe realizar el análisis desde adentro hacia afuera. El tiempo total 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 condición

then S1

else S2

El tiempo de ejecución no puede ser superior al tiempo del testeo mas el max (t1, t2) donde t1 es el tiempo de ejecución de S1 y t2 es el tiempo de ejecución de S2.

Regla 4: Para sentencias consecutivas.

Si un fragmento de codigo esta formado por dos bloques de uno con tiempo t1(n) y otro t2(n), el tiempo total es el máximo de los dos anteriores.

Bibliografia

- Análisis y diseño de Algoritmos: Un enfoque practico. Jose Ignacio Pelaez. Editorial UMA. 2003.

- Estructura de datos en C++. Luis Joyanes Aguilar y Otros. Mc Graw Hill. 2007