Tiempos de Ejecución

Tiempo de ejecución

```
PARA i ← 1 HASTA n-1   HACER
      PARA j ← 1 HASTA n   HACER
         S1
               FIN PARA
      FIN PARA
```

$$T_n = \sum_{i=1}^{n-1} \sum_{j=1}^{n} k + \ldots.$$

$$T_n = \sum_{i=1}^{n-1} kn = k \sum_{i=1}^{n-1} n = k\,n(n-1) = kn^2 - kn$$

Ejemplo

```
private static int ejemplo (int n)     0.
{ int i,j,k, cantA =0, cantB = 0;       1.
  for (i = 1: i <= n; i++)              2.
     { for (j = 1: j < i; j++){         3.
           contA++;                     4.
           contB++;    }                5.
       for (k = i: k < n; k++){         6.
           contA++;                     7.
           contB+= contA;    }          8.
      }                                 9.
 return contB;                         10.
}                                      11.
```

$$T_n = T_{s1} + T_{s2} + T_{10}$$

$$T_{s1} = 2 \qquad T_{s10} = 1$$

$$T_{s2} = 1 + \sum_{i=1}^{n} \left(1 + T_{s3} + T_{s6} + 2\right) + 1$$

$$T_{s2} = 2 + \sum_{i=1}^{n} \left(T_{s3} + T_{s6} + 3\right)$$

$$T_{s3} = 1 + \sum_{j=1}^{i-1} \left(1 + T_{s4} + T_{s5} + 2\right) + 1$$

$$T_{s4} = 2 \qquad T_{s5} = 2$$

$$T_{s3} = 2 + \sum_{j=1}^{i-1} \left(1 + 4 + 2\right)$$

$$T_{s3} = 2 + \sum_{j=1}^{i-1} (7) = 2 + 7(i-1) = 7i - 5$$

Ejemplo

$$T_n = T_{s1} + T_{s2} + T_{10}$$

```
0.  private static int ejemplo (int n)
1.  { int i,j,k, cantA =0, cantB = 0;
2.    for (i = 1: i <= n; i++)
3.      { for (j = 1: j < i; j++){
4.            contA++;
5.            contB++;    }
6.        for (k = i: k < n; k++){
7.            contA++;
8.            contB+= contA;    }
9.       }
10.  return contB;
11. }
```

$$T_{s1} = T_{s4} = T_{s5} = T_{s7} = T_{s8} = 2$$

$$T_{s10} = 1$$

$$T_{s2} = 2 + \sum_{i=1}^{n} \left(T_{s3} + T_{s6} + 3\right)$$

$$T_{s3} = 1 + \sum_{j=1}^{i-1} \left(1 + T_{s4} + T_{s5} + 2\right) + 1$$

$$T_{s3} = 2 + \sum_{j=1}^{i-1} (7) = 2 + 7(i-1) = 7i - 5$$

$$T_{s6} = 1 + \sum_{k=i}^{n-1} \left(1 + T_{s7} + T_{s8} + 2\right) + 1$$

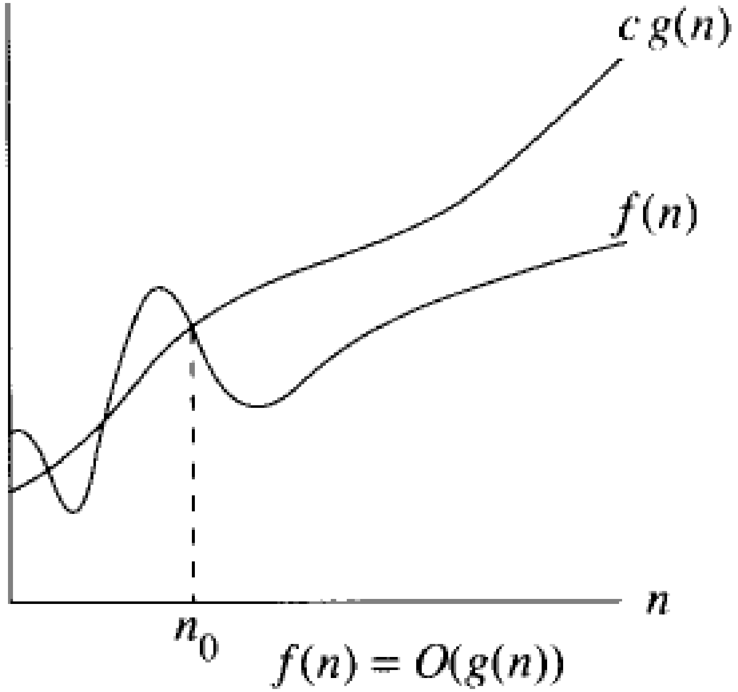
$$T_{s6} = 2 + \sum_{k=i}^{n-1} (7) = \sum_{k=i}^{n-1} (7) - \sum_{k=i}^{i-1} (7) = 2 + 7\mathrm{n} - 7\mathrm{i}$$

$$T_{s3} + T_{s6} = 7\mathrm{i} - 5 + 2 + 7\mathrm{n} - 7\mathrm{i} = 7\mathrm{n} - 3$$

$$T_{s2} = 2 + \sum_{i=1}^{n} 7\mathrm{n} = 2 + 7n^2$$

Notación asintótica

- La *notacion asintótica superior*, es una función que sirve de cota superior o techo de un conjunto de funciones y es de gran utilidad para clasificar la eficiencia de los algoritmos.



Algoritmos – notación asintótica

$O(f(n)) = \{t: N \rightarrow R+ \mid (\exists\, c \in R+, \exists\, n_0 \in N\, n_0 :$

$\forall\, n >= n_0,\ \ t(n) \leq c\, f(n)\ \}$

$$T_n = T_{s1} + T_{s2} + T_{10} = 5 + 7n^2$$

T(n) es de $O(n^2)$ → n^2 es cota superior

Orden de f(n) = conjunto de las funciones t que tomando un valor c constante donde para todo n mayor a un cierto valor t(n) es menor o igual que c por f(n).

- f(n) ES UNA FUNCIÓN QUE ACOTA SUPERIORMENTE A t(n))

Algoritmos – notación asintótica

¿Qué pasa cuando n $\rightarrow \infty$?

T(n) es de $O(n^2)$ $\rightarrow$ n^2 es cota superior

Se dice que es de **orden** $\mathbf{n^2}$

T(n) es de O($f(n)$) $\rightarrow$ $f(n)$ es cota superior

Se dice que es de **orden f(n)**

$\exists\, c$ y $n_0 : \forall\ n >= n_0,\ T(n) <= c\, f(n)$

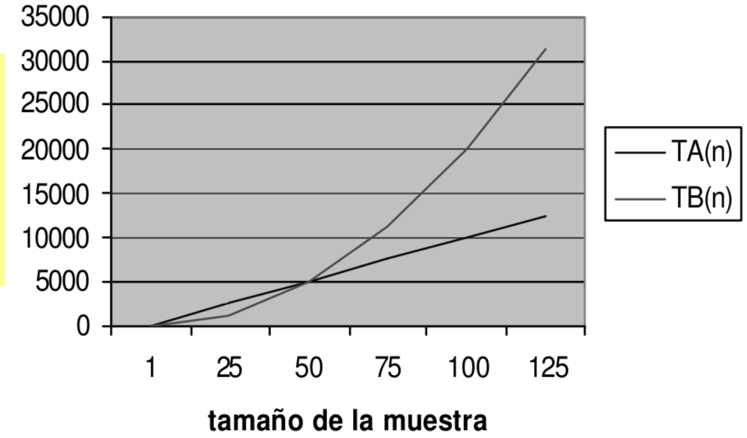
Existe un valor ***c*** y ***n_0*** tales que se cumple para todos los valores ***n*** mayores que ***n_0*** , T(n)<= ***c.f(n)***

Orden para comparar

- Se analizan el orden de los algoritmos
  - $T_A(n) = 100 * n \rightarrow$ **O(n)** ,crecimiento lineal
  - $T_B(n) = 2 * n^2 \rightarrow$ **O(n²)** ,crecimiento cuadrático

Es mejor algoritmo el que tenga menor orden de crecimiento

(menor cantidad de recursos)



- $T_C(n) = 3 * n^3 + 2 * n^2 \rightarrow$ **O(n³),**crecimiento cúbico

- $T_D(n) = 3^n \rightarrow$ **? O(n³)**

No, es exponencial, pero si considero que es así debo llegar a un absurdo

Probar el orden

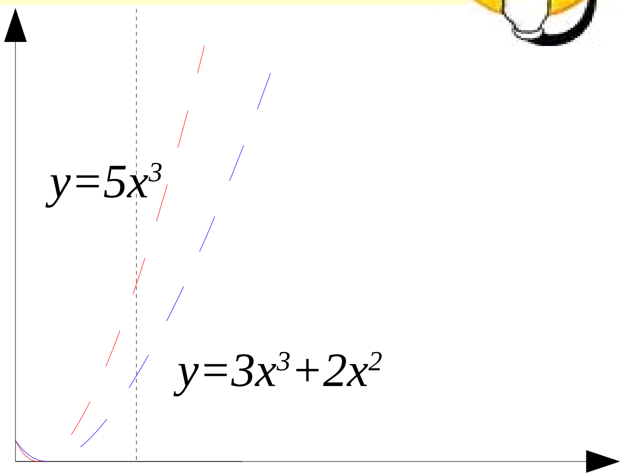
- $T_C(n) = 3 * n^3 + 2 * n^2 \rightarrow O(n^3)$, se prueba siguiendo la definición

Elijo un c y n_0, entonces tengo que probar $3n^3+2n^2 <= cn^3$

Sea $c = 5$ y $n_0 = 1$

$=> \forall \; n_0 >= 1$,

$3 \; n^3 + 2 \; n^2 <= 5 \; n^3$



es de orden $O(n^3)$

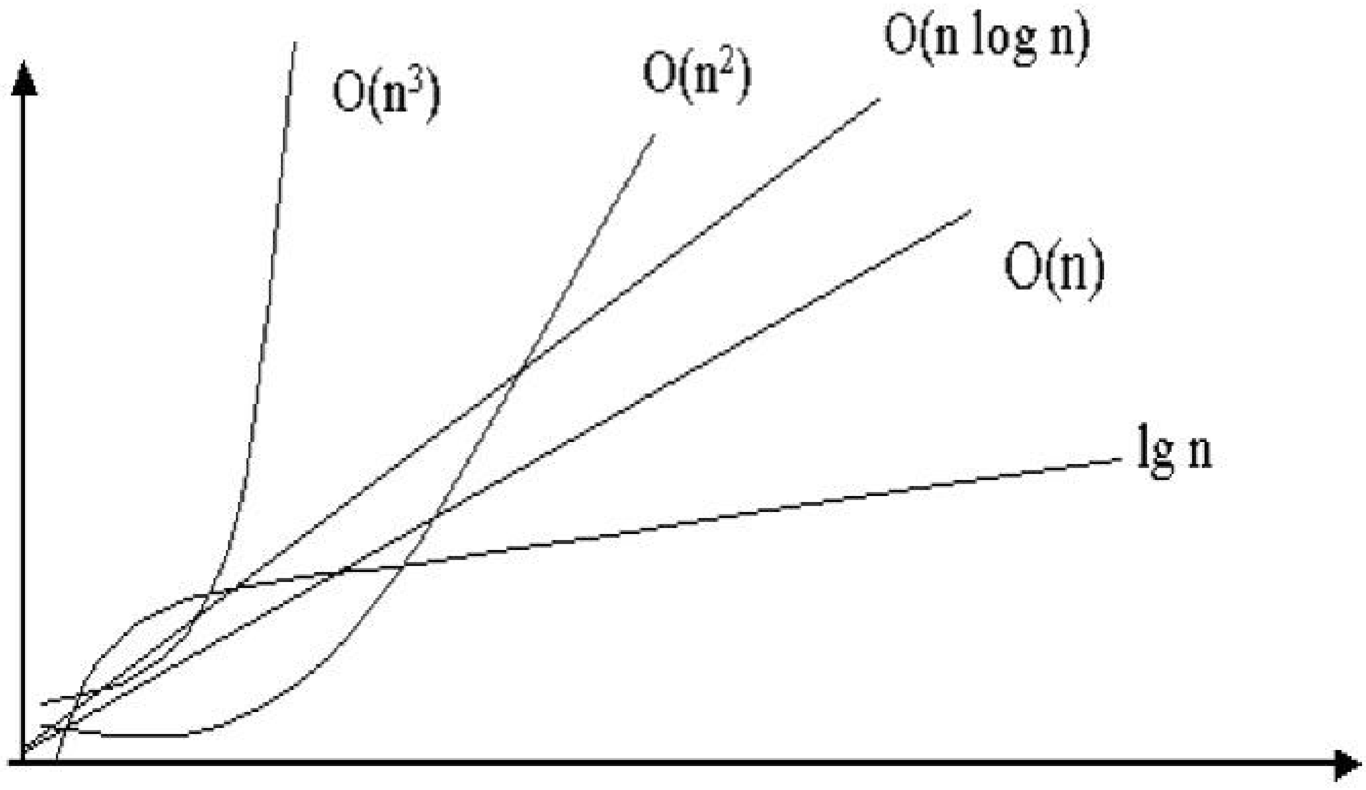
Funciones de complejidad

MEJOR

O(1)	Complejidad constante
O(log n)	Complejidad logarítmica
O(n)	Complejidad lineal
O(n log n)	
O(n^2)	Complejidad cuadrática
O(n^3)	Complejidad cúbica
O(n^k)	Complejidad polinómica (k >3)
O(k^n)	Complejidad exponencial

PEOR

Funciones de O



Análisis de eficiencia

- Los factores constantes no son importantes, se eliminan

  $c*O(f(n))$ es $O(f(n))$

- Los términos de orden inferior se descartan

  si $T(n) = a_k n^k + a_k n^{k-1} + \ldots.. a_0$

  $T(n)$ es $O(n^k)$

- La base de los logaritmos no es importante

  $O(\log n)$ no se especifica la base $(\log_b n = \log_c n * \log_b c)$

Reglas generales

Regla de la suma

- Se toma el maximo de los ordenes de ejecucion. Sean p, q dos fragmentos de programa que se ejecutan en forma consecutiva:

$T_p(n) \Rightarrow O(f(n))$

$T_q(n) \Rightarrow O(g(n))$

$T_p(n) + T_q(n) \Rightarrow O(max(f(n), g(n))$

SI cond HACER

S1

FIN SI

S2

$O(f(n))$ — `Ej. O(n²)`

$O(g(n))$ — `Ej. O(n³)`

$O(max(f(n), g(n))$ — `Ej. O(n³)`

Reglas generales

Regla del producto

- Se toma el producto de los órdenes de ejecucion. Sean p, q dos fragmentos de programa tales que:

$$T_p(n) \rightarrow O(f(n))$$

$$T_q(n) \rightarrow O(g(n))$$

$$T_p(n) * T_q(n) \rightarrow O(f(n) * g(n))$$

```
MIENTRAS cond HACER
      MIENTRAS cond HACER
            S1
      FIN MIENTRAS
FIN MIENTRAS
```

$T_p(n)$

$T_q(n)$

$T_p(n) * T_q(n)$

Uso de reglas

```
PARA i= 1 HASTA n HACER
        a[i] ← 0
FIN PARA
PARA i= 1 HASTA n HACER
        PARA j= 1 HASTA n HACER
                a[i] ← a[i] + a[j] + i + j
        FIN PARA
FIN PARA
```

$$\left.\begin{array}{l}\text{PARA i= 1 HASTA n HACER}\\ \quad a[i] \leftarrow 0 \quad O(1)\\ \text{FIN PARA}\end{array}\right\} O(n)$$

$$\left.\begin{array}{l}\text{PARA i= 1 HASTA n HACER}\\ \left.\begin{array}{l}\quad\text{PARA j= 1 HASTA n HACER}\\ \quad\quad a[i] \leftarrow a[i] + a[j] + i + j\\ \quad\text{FIN PARA}\end{array}\right\} O(n)\\ \text{FIN PARA}\end{array}\right\} O(n^2)$$

O(f(n) g(n))=*

= max(n, n2) =

= max(O(n), O(n²))

es de orden cuadrado

O(n²)

Buscando en un arreglo desordenado

- Si el arreglo está desordenado, la única opción es *Búsqueda Secuencial* :
  - Comienza desde el principio del arreglo
  - Avanza de a uno hasta que
    - el valor sea encontrado,
    - o el final del arreglo sea alcanzado
- También puede comenzar desde el final y retroceder hacia el principio
- Si el arreglo está parcialmente completo, la búsqueda termina de acuerdo al criterio utilizado (variable contador o último valor significativo)

Modulo busquedaBinaria

```
MODULO busquedaBinaria ( ENTERO T[1..n], ENTERO x ) RETORNA Entero
ENTERO medio, ini, fin, ubicacion
 LOGICO encontrado
            ini ← 1
            fin ← n
            encontrado ← falso
            ubicacion ← -1
/*1*/       MIENTRAS( no encontrado AND  ini < fin ) HACER
/*2*/               medio ← ( ini + fin ) / 2 // division entera
/*3*/               SI ( x < T[medio] ) ENTONCES
/*4*/                               fin ← medio -  1
/*5*/               SINO
                              SI (x = T[medio] ) ENTONCES
                                   encontrado ← verdadero
                                   ubicacion ← medio
                              SINO
                                   ini← medio +  1
                              FIN SI
                    FIN SI
          FIN MIENTRAS
RETORNAR ubicacion
FIN MODULO
```

TE busquedaBinaria

Tiempo de Ejecución

- El rango de elementos candidatos a buscar es la mitad despues de cada comparación.

COMPARACIONES	TAMAÑO ARREGLO
0	n
1	n/2
2	n/4
3	n/8
….	….
i	$n/(2^i)$

- La búsqueda binaria corre en un tiempo *O(log n)*.

Eficiencia

- Cada comparación, divide el arreglo restante en dos mitades
  - n/2 , n/4, n/8, El proceso terminará cuando el $n = 2^k$
  - Si se aplica ***log*** a ambos términos $\log_2 n < k \log_2 2$

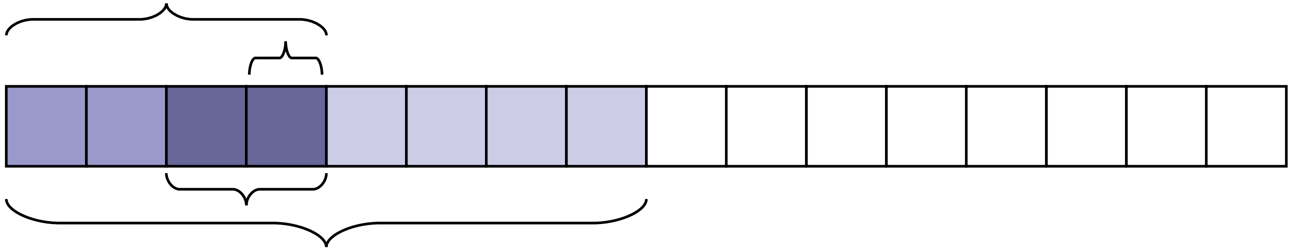
Definición de logaritmo: $\log_b x = n$ ***ssi*** $x = b^n$,
Tomemos las potencias de 2 y sus exponentes:

- $2^0 = 1$, $2^1 = 2$, $2^2 = 4$, $2^3 = 8$, *y* $2^4 = 16$ *etc.* luego
- $\log_2 1 = 0$, $\log_2 2 = 1$, $\log_2 4 = 2$, $\log_2 8 = 3$ *y* $\log_2 16 = 4$ *etc.*

- Por lo tanto la eficiencia de la búsqueda binaria es una función logarítmica $O(\log_2 n)$
- Ej. En un arreglo de 50.000 elementos la búsqueda binaria, en el peor de los casos, requiere $\log_2 50000 = 16$ comparaciones.

Búsqueda binaria

- Sólo puede usarse si el arreglo está ordenado
- Parte el arreglo en mitades y busca en la mitad que corresponde
- Evita buscar en la mitad que sabe que no lo va a encontrar
- Mucho más eficiente que la búsqueda secuencial
- Ej: en un vector de longitud 16, como máximo hace 4 comparaciones (16/2=8, 8/2=4, 4/2=2, 2/2=1)



Búsqueda Binaria

- Sea **`int v[10] = {2,4,6,8,10,12,14,16,18,20};`**
- Se desea buscar el elemento 6

0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
2	4	6	8	10	12	14	16	18	20

	inicio	**fin**	**medio**
Inicialización	0	9	
Iteración 1	0	9	(0+9)/2=4
Iteración 2	0	**3**	(0+3)/2=1
Iteración 3	**2**	3	(2+3)/2=2

Compara 6 y v[4]=10
Debe buscar en parte inferior

Compara 6 y v[1]=4
Debe buscar en parte superior

Compara 6 y v[2]=6
ÉXITO !!!