

## Tema 1: Análisis de la eficiencia de los algoritmos

---

Clara María Segura Díaz  
Fundamentos de Algoritmia, Curso 2020-21

Fíjate si es importante que lo hacemos en ED. Eficiencia es lo que buscamos siempre en ingeniería del software

Dpto. de Sistemas Informáticos y Computación  
Facultad de Informática  
Universidad Complutense de Madrid

Lo más importante es el análisis de coste

- **Diseño de programas. Formalismo y abstracción;** R. Peña. Tercera edición. Prentice Hall, 2005.  
**Capítulo 1**
- **Algoritmos correctos y eficientes: Diseño razonado ilustrado con ejercicios.** N. Martí, C. Segura, J.A. Verdejo. Ibergarceta Publicaciones, 2012.  
**Capítulo 3**
- **Fundamentos de algoritmia.** G. Brassard, P. Bratley. Prentice Hall, 1997.  
**Capítulos 2 y 4**
- **Foundations of Algorithms using C++ pseudocode;** R. Neapolitan, K. Naimipour. Segunda edición. Jones and Bartlett Publishers, 1997.  
**Capítulo 1**

# Análisis de la eficiencia de los algoritmos

- Motivación **Para que calcular el coste**
- Factores de los que depende el coste **De qué depende el coste**
- Medidas asintóticas de la eficiencia **Asintotas**
- Análisis de coste de algoritmos iterativos **Analizar el coste de algoritmos iterativos.**

## Motivación

---

## 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:

Vamos a leer  
el problema

- un grano de trigo en la primera casilla de un tablero de ajedrez,  $2^0$
- más dos granos de trigo en la segunda casilla,  $2^1$
- más cuatro granos de trigo en la tercera casilla,  $2^2$
- y así sucesivamente, duplicando el número de granos en cada casilla, hasta llegar a la última casilla.  $2^n$

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:

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

## ¿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:

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

## ¿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 llenar 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!!.**

## Endeudado hasta las cejas

En efecto, para llenar 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 llenar. Toma valores desmesurados aunque el número de casillas sea pequeño.

## Endeudado hasta las cejas

En efecto, para llenar 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 llenar. 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.

- Entendemos por **eficiencia** el rendimiento de una actividad en relación con el consumo de un cierto recurso. Es diferente de la efectividad.

**EFICIENCIA == RENDIMIENTO DE UN ALGORITMO**

- Entendemos por **eficiencia** el rendimiento de una actividad en relación con el consumo de un cierto recurso. Es diferente de la efectividad.
- Ejemplo para ver la importancia de que el coste del algoritmo sea pequeño.

Menos costoso

Num  
de  
elementos

$n$	$\log_{10} n$	$n$	$n \log_{10} n$	$n^2$	$n^3$	$2^n$
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 el rendimiento de una cosa/actividad en función de su coste. Cuanto más rdto y menor coste, más eficiente.

Cuánto menor coste tenga el algoritmo mejor. Ya que tardará menos en ejecutarse. Vamos a tomar coste como rendimiento o tiempo del algoritmo.

- Entendemos por **eficiencia** el rendimiento de una actividad en relación con el consumo de un cierto recurso. Es diferente de la efectividad.
- Ejemplo para ver la importancia de que el coste del algoritmo sea pequeño.

$n$	$\log_{10} n$	$n$	$n \log_{10} n$	$n^2$	$n^3$	$2^n$
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.

Un algoritmo  
puede llegar a  
tardar siglos.

**Factores de los que depende el coste**

## ¿Qué medimos y cómo?

- La eficiencia es mayor cuanto menor es la complejidad o el coste (consumo de recursos).

Un algoritmo de coste logarítmico es más eficiente que un algoritmo de coste lineal.

Un algoritmo de coste constante es más eficiente que un algoritmo de coste logarítmico

## ¿Qué medimos y cómo?

- La eficiencia es mayor cuanto menor es la complejidad o el coste (consumo de recursos).
- Necesitamos determinar cómo se ha de medir el coste de un algoritmo, de forma que sea posible compararlo con otros que resuelven el mismo problema y decidir cuál de todos es el más eficiente.

Cuanto menos cueste un algoritmo, cuanto MENOR SEA EL ORDEN DE UN ALGORITMO, mayor EFICIENCIA TENDRÁ el mismo.

Comparar algoritmos para llegar a poder elegir los más eficientes.

## ¿Qué medimos y cómo?

mayor eficiencia si menor coste

- La eficiencia es mayor cuanto menor es la complejidad o el coste (consumo de recursos).
- Necesitamos determinar cómo se ha de medir el coste de un algoritmo, de forma que sea posible compararlo con otros que resuelven el mismo problema y decidir cuál de todos es el más eficiente.
- Una posibilidad para medir el coste de un algoritmo es contar cuántas instrucciones de cada tipo se ejecutan, multiplicar este número por el tiempo que emplea la instrucción en ejecutarse, y realizar la suma para los diferentes tipos.

$t_a$  = tiempo de asignación

$t_c$  = tiempo de comparación

$t_i$  = tiempo de incremento

$t_v$  = tiempo de acceso a un vector

Esto NO lo hacemos porque si no pasaríamos mucho tiempo tratando de definir el coste de cada algoritmo.

En realidad lo que utilizamos son los ordenes para comparar algoritmos. Cuanto menor sea el orden menor coste. Debemos determinar si el coste es lineal, logarítmico, cuadrático...

## Ejemplo

Siempre calculamos el coste de nuestro algoritmo en el peor de los casos

La siguiente función recibe un array de  $n$  enteros y devuelve la posición del primer 0. En caso de no haber ninguno devuelve  $n$ .

```
int primerCero(const int a[ ], int n)
{
    int i=0;
    while (i<n && a[i]!=0)
        {i=i+1;}
    return i;
}
```

En el peor de los casos el bucle se ejecuta  $n$  veces. El coste es lineal.

Es un algoritmo de búsqueda que trata de buscar el primer elemento con del array que sea 0

- El tamaño de la entrada es el número  $n$  de elementos del array que vamos a explorar, cuantos más elementos mayor será el coste de este bucle.

Nosotros siempre vamos a calcular el coste en el peor de los casos

El coste de este algoritmo en el peor de los casos es lineal con respecto a  $n$ . Ya que dentro del algoritmo se hace una asignación y eso tiene coste constante.

## Ejemplo

La siguiente función recibe un array de  $n$  enteros y devuelve la posición del primer 0. En caso de no haber ninguno devuelve  $n$ .

---

```
int primerCero(const int a[ ], int n)
{
    int i=0;
    while (i<n && a[i]!=0)
        {i=i+1;}
    return i;
}
```

---

- Fijado un tamaño  $n$ , algunos arrays (ejemplares) de ese tamaño hacen trabajar más al bucle (por ejemplo, los que no tienen ningún 0) y otros menos (por ejemplo, los que tienen un 0 al principio).

Nos dijimos que no entraremos en detalles de entrar dentro de los bucles también si no hay otro bucle. Aquí por tanto decimos que el coste es lineal en el peor de los casos.

## Ejemplo

La siguiente función recibe un array de  $n$  enteros y devuelve la posición del primer 0. En caso de no haber ninguno devuelve  $n$ .

```
int primerCero(const int a[ ], int n)
{
    int i=0;
    while (i<n && a[i]!=0)
        {i=i+1;}
    return i;
}
```

- Los ejemplares que tienen un 0 en la primera posición (caso mejor) tardan:

$$t_a + 2 * t_c + t_v$$

En estos ejemplares el coste es siempre el mismo independientemente del número de elementos a explorar, es decir, es **constante**.

Porque únicamente entramos una vez al array y nos salimos. Pero en el caso peor será lineal en el número de elementos ( $n$ ) ya que recorreremos cada posición del vector.

## Ejemplo

La siguiente función recibe un array de  $n$  enteros y devuelve la posición del primer 0. En caso de no haber ninguno devuelve  $n$ .

```
int primerCero(const int a[ ], int n)
{
    int i=0;
    while (i<n && a[i]!=0)
        {i=i+1;}
    return i;
}
```

- Los ejemplares de tamaño  $n$  que no tienen ningún 0 (caso peor) tardan:

$$t_a + n * (2 * t_c + t_v + t_i) + t_c$$

En estos ejemplares, el coste depende **linealmente** del número de elementos a explorar: si tenemos el doble de elementos, cabe esperar que tardemos aproximadamente el doble de tiempo

Creo que siempre miramos en el peor de los casos. Porque queremos ver siempre cual es el coste en el caso de que se produzca el máximo número de iteraciones porque es algo que es probable que pase.

### Ordenación por selección del vector $V[1..n]$

```
int a[n];
int i, j, pmin, temp;
for (i = 0; i < n-1; i++)
    // pmin calcula la posición del mínimo de a[i.. n-1]
    {pmin = i;
        for (j = i+1; j < n; j++)
            if (a[j] < a[pmin]) pmin = j;
        // ponemos el mínimo en a[i]
        temp = a[i]; a[i] = a[pmin]; a[pmin] = temp;
    }
```

- control del primer bucle:  $t_a + (n - 1)t_i + nt_c$

A la hora de "analizar" el coste de algoritmos que poseen dos bucles, siempre debemos de analizar primero el coste del bucle más interno, en este caso el segundo for.

Dicho bucle se recorrerá  $n$  veces mientras que el bucle más externo se recorrerá un total de  $n-1$  veces (es decir, el orden de  $n$  veces). Por tanto, el coste de dicho algoritmo es cuadrático.

## Ejemplo

Ordenación por selección del vector  $V[1..n]$

---

```
int a[n];
int i, j, pmin, temp;
for (i = 0; i < n-1; i++)
    // pmin calcula la posición del mínimo de a[i.. n-1]
    {pmin = i;
        for (j = i+1; j < n; j++)
            if (a[j] < a[pmin]) pmin = j;
        // ponemos el mínimo en a[i]
        temp = a[i]; a[i] = a[pmin]; a[pmin] = temp;
    }
```

---

- control del primer bucle:  $t_a + (n - 1)t_i + nt_c$
- primera asignación:  $(n - 1)t_a$

## Ejemplo

Ordenación por selección del vector  $V[1..n]$

---

```
int a[n];
int i, j, pmin, temp;
for (i = 0; i < n-1; i++)
    // pmin calcula la posición del mínimo de a[i.. n-1]
    {pmin = i;
        for (j = i+1; j < n; j++)
            if (a[j] < a[pmin]) pmin = j;
        // ponemos el mínimo en a[i]
        temp = a[i]; a[i] = a[pmin]; a[pmin] = temp;
    }
```

---

- control del primer bucle:  $t_a + (n - 1)t_i + nt_c$
- primera asignación:  $(n - 1)t_a$
- control del bucle interno, para cada  $i$ :  $t_a + (n - i - 1)t_i + (n - i)t_c$

## Ejemplo

Ordenación por selección del vector  $V[1..n]$

---

```
int a[n];
int i, j, pmin, temp;
for (i = 0; i < n-1; i++)
    // pmin calcula la posición del mínimo de a[i.. n-1]
    {pmin = i;
        for (j = i+1; j < n; j++)
            if (a[j] < a[pmin]) pmin = j;
        // ponemos el mínimo en a[i]
        temp = a[i]; a[i] = a[pmin]; a[pmin] = temp;
    }
```

---

- control del primer bucle:  $t_a + (n - 1)t_i + nt_c$
- primera asignación:  $(n - 1)t_a$
- control del bucle interno, para cada  $i$ :  $t_a + (n - i - 1)t_i + (n - i)t_c$
- instrucción **if**, para cada  $i$ , tiempo mínimo:  $(n - i - 1)(2t_v + t_c)$   
tiempo máximo:  $(n - i - 1)(2t_v + t_c + t_a)$

## Ejemplo

Ordenación por selección del vector  $V[1..n]$

---

```
int a[n];
int i, j, pmin, temp;
for (i = 0; i < n-1; i++)
    // pmin calcula la posición del mínimo de a[i.. n-1]
    {pmin = i;
        for (j = i+1; j < n; j++)
            if (a[j] < a[pmin]) pmin = j;
        // ponemos el mínimo en a[i]
        temp = a[i]; a[i] = a[pmin]; a[pmin] = temp;
    }
```

---

- control del primer bucle:  $t_a + (n - 1)t_i + nt_c$
- primera asignación:  $(n - 1)t_a$
- control del bucle interno, para cada  $i$ :  $t_a + (n - i - 1)t_i + (n - i)t_c$
- instrucción **if**, para cada  $i$ , tiempo mínimo:  $(n - i - 1)(2t_v + t_c)$   
tiempo máximo:  $(n - i - 1)(2t_v + t_c + t_a)$
- intercambiar:  $(n - 1)(4t_v + 3t_a)$

## Ejemplo

El bucle interno en total en el caso peor, el más desfavorable:

$$\sum_{i=0}^{n-2} (t_a + t_c + (n - i - 1)(t_i + 2t_c + 2t_v + t_a))$$

## Ejemplo

El bucle interno en total en el caso peor, el más desfavorable:

$$\sum_{i=0}^{n-2} (t_a + t_c + (n - i - 1)(t_i + 2t_c + 2t_v + t_a))$$

El bucle interno en total en el caso mejor, el más favorable:

$$\sum_{i=0}^{n-2} (t_a + t_c + (n - i - 1)(t_i + 2t_c + 2t_v))$$

## Ejemplo

El bucle interno en total en el caso peor, el más desfavorable:

$$\sum_{i=0}^{n-2} (t_a + t_c + (n - i - 1)(t_i + 2t_c + 2t_v + t_a))$$

El bucle interno en total en el caso mejor, el más favorable:

$$\sum_{i=0}^{n-2} (t_a + t_c + (n - i - 1)(t_i + 2t_c + 2t_v))$$

La suma de todos los tiempos da lugar a dos polinomios de la forma:

$$T_{\min} = An^2 - Bn + C$$

$$T_{\max} = A'n^2 - B'n + C'$$

## Ejemplo

El bucle interno en total en el caso peor, el más desfavorable:

$$\sum_{i=0}^{n-2} (t_a + t_c + (n - i - 1)(t_i + 2t_c + 2t_v + t_a))$$

El bucle interno en total en el caso mejor, el más favorable:

$$\sum_{i=0}^{n-2} (t_a + t_c + (n - i - 1)(t_i + 2t_c + 2t_v))$$

La suma de todos los tiempos da lugar a dos polinomios de la forma:

$$T_{\min} = An^2 - Bn + C$$

$$T_{\max} = A'n^2 - B'n + C'$$

El coste depende de forma cuadrática del tamaño de los datos de entrada por lo que si se duplica el número de elementos de la entrada cabe esperar que tardemos no el doble sino cuatro veces el tiempo.

Se pondría que coste es  $O(n^2)$  coste es del orden de..

Creo que esto es bastante importante y no lo tenía tan presente cuando comencé a estudiar.

El tiempo de ejecución de un algoritmo depende en general de tres factores:

1. El **tamaño** de los datos de entrada. Por ejemplo:

- Para un vector: su longitud.
- Para un número natural: su valor o el número de dígitos.
- Para un grafo: el número de vértices y/o el número de aristas.

Ejercicios de recursión sobre un número del tema 4.

El tiempo de ejecución de un algoritmo depende en general de tres factores:

1. El **tamaño** de los datos de entrada. Por ejemplo:
  - Para un vector: su longitud.
  - Para un número natural: su valor o el número de dígitos.
  - Para un grafo: el número de vértices y/o el número de aristas.
2. El **contenido** de esos datos: para distintas entradas del mismo tamaño el coste puede variar desde el caso más favorable (dado por  $T_{\min}$ ) y el más desfavorable (dado por  $T_{\max}$ ).

# Factores

El tiempo de ejecución de un algoritmo depende en general de tres factores:

1. El **tamaño** de los datos de entrada. Por ejemplo:

- Para un vector: su longitud.
- Para un número natural: su valor o el número de dígitos.
- Para un grafo: el número de vértices y/o el número de aristas.

2. El **contenido** de esos datos: para distintas entradas del mismo tamaño el coste puede variar desde el caso más favorable (dado por  $T_{\min}$ ) y el más desfavorable (dado por  $T_{\max}$ ). Umbral del tema 5, experimental

3. El **código generado por el compilador** y el **computador** concreto utilizados, que afectan a los tiempos de las operaciones elementales.

Si hacen una pregunta teórica, que no lo creo, podrían preguntarnos de que depende el tiempo de ejecución de un algoritmo: del tamaño, del contenido y del compilador y computador.

- Para poder comparar algoritmos independientemente del valor de los datos de entrada, el segundo factor debemos eliminarlo:

- O bien midiendo solo el **caso peor**, es decir la ejecución que tarde más tiempo de todos los ejemplares de tamaño  $n$ .

Si el tiempo que tarda un algoritmo  $A$  en procesar una entrada concreta  $\bar{x}$  lo denotamos por  $t_A(\bar{x})$ , definimos la complejidad de  $A$  en el **caso peor** como

$$T_A(n) = \max\{t_A(\bar{x}) \mid \bar{x} \text{ de tamaño } n\}$$



- Para poder comparar algoritmos independientemente del valor de los datos de entrada, el segundo factor debemos eliminarlo:

- O bien midiendo solo el **caso peor**, es decir la ejecución que tarde más tiempo de todos los ejemplares de tamaño  $n$ .

Si el tiempo que tarda un algoritmo  $A$  en procesar una entrada concreta  $\bar{x}$  lo denotamos por  $t_A(\bar{x})$ , definimos la complejidad de  $A$  en el **caso peor** como

$$T_A(n) = \max\{t_A(\bar{x}) \mid \bar{x} \text{ de tamaño } n\}$$

- O bien midiendo todos los casos de tamaño  $n$  y calculando el tiempo del **caso promedio**.

Definimos la complejidad de un algoritmo  $A$  en el caso promedio como

$$TM_A(n) = \sum_{\bar{x} \text{ de tamaño } n} p(\bar{x}) t_A(\bar{x})$$

siendo  $p(\bar{x}) \in [0..1]$  la probabilidad de que la entrada sea  $\bar{x}$ .

Esto último es mucho más laborioso que calcular el caso peor. Así que nos quedamos con el caso peor, el que está encuadrado.

# Factores

## SÓLAMENTE NOS FIJAMOS EN EL CASO PEOR.

- Para poder comparar algoritmos independientemente del valor de los datos de entrada, el segundo factor debemos eliminarlo:

- O bien midiendo solo el **caso peor**, es decir la ejecución que tarde más tiempo de todos los ejemplares de tamaño  $n$ .

Si el tiempo que tarda un algoritmo  $A$  en procesar una entrada concreta  $\bar{x}$  lo denotamos por  $t_A(\bar{x})$ , definimos la complejidad de  $A$  en el **caso peor** como

$$T_A(n) = \max\{t_A(\bar{x}) \mid \bar{x} \text{ de tamaño } n\}$$

- O bien midiendo todos los casos de tamaño  $n$  y calculando el tiempo del **caso promedio**.

Definimos la complejidad de un algoritmo  $A$  en el caso promedio como

$$TM_A(n) = \sum_{\bar{x} \text{ de tamaño } n} p(\bar{x}) t_A(\bar{x})$$

siendo  $p(\bar{x}) \in [0..1]$  la probabilidad de que la entrada sea  $\bar{x}$ .

- Nos concentraremos en el **caso peor** por dos razones:

- Establece una cota superior fiable para **todos** los casos del mismo tamaño.
- Es más fácil de calcular.

Nos centramos en el caso peor porque en el caso mejor nos estaríamos saltando algunos de los casos y en el caso promedio hay que calcular muchas cosas. Lo más sencillo es el caso peor.

## Medidas asintóticas de la eficiencia

Durante este punto del tema hablaremos de costes y sus asintotas, hablaremos de ordenes, de omegas y de orden exacto.

En nuestros problemas normalmente hablamos del orden y del orden exacto.

## Medidas asintóticas de la eficiencia

- **El criterio asintótico** para medir la eficiencia de los algoritmos tiene como objetivo comparar algoritmos **independientemente** de los lenguajes en que están escritos, de las máquinas en que se ejecutan **y del valor concreto de los datos** que reciben como entrada.

## Medidas asintóticas de la eficiencia

- El criterio asintótico para medir la eficiencia de los algoritmos tiene como objetivo comparar algoritmos independientemente de los lenguajes en que están escritos, de las máquinas en que se ejecutan y del valor concreto de los datos que reciben como entrada.
- Tan solo considera importante el tamaño de dichos datos.
- Para cada problema habrá que definir qué se entiende por tamaño del mismo.

- El criterio asintótico para medir la eficiencia de los algoritmos tiene como objetivo comparar algoritmos independientemente de los lenguajes en que están escritos, de las máquinas en que se ejecutan y del valor concreto de los datos que reciben como entrada.
- Tan solo considera importante el tamaño de dichos datos.
- Para cada problema habrá que definir qué se entiende por tamaño del mismo.
- Se basa en tres principios:
  1. El coste o eficiencia es una función que solo depende del tamaño de la entrada, e.g.  $f(n) = n^2$ . En este caso sería de coste cuadrático.
  2. Las constantes multiplicativas o aditivas no se tienen en cuenta, e.g.  $f(n) = n^2$  y  $g(n) = 3n^2 + 27$  se consideran costes equivalentes. El coste en ese caso sigue siendo  $n^2$ .
  3. La comparación entre funciones de coste se hará para valores de  $n$  suficientemente grandes, es decir los costes para tamaños pequeños se consideran irrelevantes.

Creo que el coste del algoritmo vendrá definido por el tamaño de los datos.

Nosotros en este tema 1 lo importante es que determinemos el tamaño del problema. Cuanto mayor tamaño mayor coste, menos eficiente. Eso si, no siempre se puede el menor tamaño en los problemas.

**Definición** El conjunto de las funciones **del orden de  $f(n)$** , denotado  $O(f(n))$ , se define como

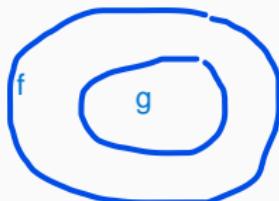
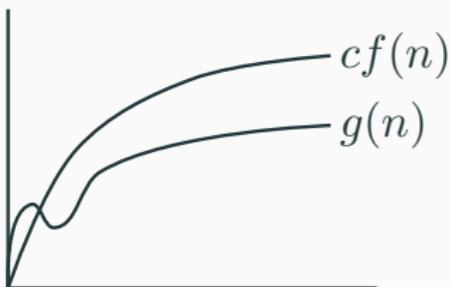
Lo de abajo explica que significa que una función  $g(n)$  pertenezca al orden de  $f(n)$

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

Asimismo, 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  $f(n)$* .

## Ejemplos

- $\log n \in O(n)$

Tenemos que demostrar que  $\log(n)$  pertenece al orden de  $n$ .

Encontramos que, según el enunciado anterior de arriba  $n_0 = 1$  y  $c = 1$ . Para todo  $n \geq 1$   $\log n \leq n$ . Ya que  $0 < 1$ .

Ahora deberíamos de demostrar que, además de para 1 se cumple para el resto de casos. Por ello aplicamos INDUCCIÓN.

BASE:  $n=1 \rightarrow 0 < 1$  Se cumple para la base.

## Ejemplos

- $\log n \in O(n)$

Encontramos  $n_0 = 1$  y  $c = 1$  tal que  $\forall n \geq 1. \log n \leq n$   
 $0 <= 1$

Para el caso base se cumple.

## Ejemplos

- $\log n \in O(n)$

Encontramos  $n_0 = 1$  y  $c = 1$  tal que  $\forall n \geq 1. \log n \leq n$

Lo demostramos por inducción.

Base:  $n = 1, \log 1 = 0 \leq 1$  ✓

## Ejemplos

- $\log n \in O(n)$

Encontramos  $n_0 = 1$  y  $c = 1$  tal que  $\forall n \geq 1. \log n \leq n$

Lo demostramos por inducción.

Base:  $n = 1, \log 1 = 0 \leq 1$

Paso inductivo: h.i.  $\log n \leq n$

$$\log(n+1) \leq \log 2n \leq \log 2 + \log n = 1 + \log n \leq^{h.i.} n+1$$

## Ejemplos

- $\log n \in O(n)$

Encontramos  $n_0 = 1$  y  $c = 1$  tal que  $\forall n \geq 1. \log n \leq n$

Lo demostramos por inducción.

Por tanto, se cumple por la hipótesis inductiva que  $\log n + 1 \leq n + 1$

Base:  $n = 1, \log 1 = 0 \leq 1$

Paso inductivo: h.i.  $\log n \leq n$

$$\log(n+1) \leq \log 2n \leq \log 2 + \log n = 1 + \log n \leq^{h.i.} n + 1$$

- $(n+1)^2 \in O(n^2)$

## Ejemplos

- $\log n \in O(n)$

Encontramos  $n_0 = 1$  y  $c = 1$  tal que  $\forall n \geq 1. \log n \leq n$

Lo demostramos por inducción.

Base:  $n = 1, \log 1 = 0 \leq 1$

Paso inductivo: h.i.  $\log n \leq n$

$$\log(n+1) \leq \log 2n \leq \log 2 + \log n = 1 + \log n \leq^{h.i.} n+1$$

- $(n+1)^2 \in O(n^2)$

Demostramos por inducción que  $\forall n \geq 1. (n+1)^2 \leq 4n^2$

Para el caso base 1:  $2^2 \leq 4(1)^2$

$$4 = 4$$

Para el caso base se cumple

## Ejemplos

- $\log n \in O(n)$

Encontramos  $n_0 = 1$  y  $c = 1$  tal que  $\forall n \geq 1. \log n \leq n$

Lo demostramos por inducción.

Base:  $n = 1, \log 1 = 0 \leq 1$

Paso inductivo: h.i.  $\log n \leq n$

$$\log(n+1) \leq \log 2n \leq \log 2 + \log n = 1 + \log n \leq^{h.i.} n+1$$

- $(n+1)^2 \in O(n^2)$

Demostramos por inducción que  $\forall n \geq 1. (n+1)^2 \leq 4n^2$

Base:  $n = 1, (1+1)^2 \leq 4 \cdot 1^2$

## Ejemplos

- $\log n \in O(n)$

Encontramos  $n_0 = 1$  y  $c = 1$  tal que  $\forall n \geq 1. \log n \leq n$

Lo demostramos por inducción.

Base:  $n = 1, \log 1 = 0 \leq 1$

Paso inductivo: h.i.  $\log n \leq n$

$$\log(n+1) \leq \log 2n \leq \log 2 + \log n = 1 + \log n \stackrel{h.i.}{\leq} n+1$$

- $(n+1)^2 \in O(n^2)$

Demostramos por inducción que  $\forall n \geq 1. (n+1)^2 \leq 4n^2$

Base:  $n = 1, (1+1)^2 \leq 4 \cdot 1^2$  ✓ Sustituimos n por n+1

Paso inductivo: h.i.  $(n+1)^2 \leq 4n^2$

$$\begin{aligned}(n+1+1)^2 &\leq 4(n+1)^2 \\ (n+1)^2 + 1 + 2(n+1) &\leq 4n^2 + 4 + 8n \\ (n+1)^2 &\leq 4n^2 + \underbrace{6n+1}_{\geq 0}\end{aligned}$$

## Ejemplos

RECORDAR QUE NO IMPORTAN LAS CONSTANTES MULTIPLICATIVAS O ADITIVAS; PERO SI QUE IMPORTA LA BASE DE UNA POTENCIA.

$2^n \in O(3^n)$ ----> La base de las potencias importan

- $3^n \notin O(2^n)$

Aquí podríamos explicar que la base de una potencia importa y que  $3^n$  pertenece al OMEGA (que lo veremos ahora más adelante) de  $2^n$  y que  $2^n$  pertenece al orden de  $3^n$

Si nos lo piden demostrar (en el tipo test no lo piden pero en el examen escrito es probable que si), debemos de poner un contraejemplo. Si no nos acordamos decir que la base de una potencia tiene consecuencias sobre el orden de un algoritmo.

## Ejemplos

- $3^n \notin O(2^n)$

Si perteneciera, tendríamos  $c \in \mathbb{R}^+$ ,  $n_0 \in \mathbb{N}$  tal que  $3^n \leq c \cdot 2^n$  para todo  $n \geq n_0$ .

## Ejemplos

- $3^n \notin O(2^n)$

Si perteneciera, tendríamos  $c \in \mathbb{R}^+$ ,  $n_0 \in \mathbb{N}$  tal que  $3^n \leq c \cdot 2^n$  para todo  $n \geq n_0$ .

Esto implica que  $(\frac{3}{2})^n \leq c$  para todo  $n \geq n_0$ .

## Ejemplos

- $3^n \notin O(2^n)$

Si perteneciera, tendríamos  $c \in \mathbb{R}^+$ ,  $n_0 \in \mathbb{N}$  tal que  $3^n \leq c \cdot 2^n$  para todo  $n \geq n_0$ .

Esto implica que  $(\frac{3}{2})^n \leq c$  para todo  $n \geq n_0$ .

Pero esto es falso porque dado  $c$  cualquiera, basta tomar  $n = \log_{1,5} c$  para que  $(\frac{3}{2})^n > c$ , es decir, no se puede acotar superiormente.

## Propiedades

TAMPOCO IMPORTA LA BASE DEL LOGARITMO, PERO SI LA DE LA POTENCIA.

- $O(a \cdot f(n)) = O(f(n))$  con  $a \in R^+$

Lo que hemos dicho de las constantes multiplicativas.

## Propiedades

- $O(a \cdot f(n)) = O(f(n))$  con  $a \in \mathbb{R}^+$  No se si esto es por las ctes multiplicativas.
- ( $\subseteq$ )  $g \in O(a \cdot f(n)) \Leftrightarrow \exists c \in \mathbb{R}^+, n_0 \in \mathbb{N}$  tal que
- $$\forall n \geq n_0. g(n) \leq c \cdot a \cdot f(n)$$

## Propiedades

- $O(a \cdot f(n)) = O(f(n))$  con  $a \in \mathbb{R}^+$

( $\subseteq$ )  $g \in O(a \cdot f(n)) \Leftrightarrow \exists c \in \mathbb{R}^+, n_0 \in \mathbb{N}$  tal que

$$\forall n \geq n_0. g(n) \leq c \cdot a \cdot f(n)$$

Tomando  $c' = c \cdot a$  se cumple que  $\forall n \geq n_0. g(n) \leq c' \cdot f(n)$ , luego  
 $g \in O(f(n))$

## Propiedades

- $O(a \cdot f(n)) = O(f(n))$  con  $a \in \mathbb{R}^+$

( $\subseteq$ )  $g \in O(a \cdot f(n)) \Leftrightarrow \exists c \in \mathbb{R}^+, n_0 \in \mathbb{N}$  tal que

$$\forall n \geq n_0. g(n) \leq c \cdot a \cdot f(n)$$

Tomando  $c' = c \cdot a$  se cumple que  $\forall n \geq n_0. g(n) \leq c' \cdot f(n)$ , luego  
 $g \in O(f(n))$

( $\supseteq$ )  $g \in O(f(n)) \Leftrightarrow \exists c \in \mathbb{R}^+, n_0 \in \mathbb{N}$  tal que  $\forall n \geq n_0. g(n) \leq c \cdot f(n)$

## Propiedades

- $O(a \cdot f(n)) = O(f(n))$  con  $a \in \mathbb{R}^+$

( $\subseteq$ )  $g \in O(a \cdot f(n)) \Leftrightarrow \exists c \in \mathbb{R}^+, n_0 \in \mathbb{N}$  tal que

$$\forall n \geq n_0. g(n) \leq c \cdot a \cdot f(n)$$

Tomando  $c' = c \cdot a$  se cumple que  $\forall n \geq n_0. g(n) \leq c' \cdot f(n)$ , luego  
 $g \in O(f(n))$

( $\supseteq$ )  $g \in O(f(n)) \Leftrightarrow \exists c \in \mathbb{R}^+, n_0 \in \mathbb{N}$  tal que  $\forall n \geq n_0. g(n) \leq c \cdot f(n)$

Entonces tomando  $c' = \frac{c}{a}$  se cumple que  $\forall n \geq n_0. g(n) \leq c' \cdot a \cdot f(n)$ ,  
luego  $g \in O(a \cdot f(n))$

## Propiedades

ESTA PRIMERA PROPIEDAD EXPLICA QUE LAS CONSTANTES MULTIPLICATIVAS NO IMPORTAN A LA HORA DE SABER EL ORDEN DE LA FUNCIÓN .

- $O(a \cdot f(n)) = O(f(n))$  con  $a \in \mathbb{R}^+$

( $\subseteq$ )  $g \in O(a \cdot f(n)) \Leftrightarrow \exists c \in \mathbb{R}^+, n_0 \in \mathbb{N}$  tal que

$$\forall n \geq n_0. g(n) \leq c \cdot a \cdot f(n)$$

Tomando  $c' = c \cdot a$  se cumple que  $\forall n \geq n_0. g(n) \leq c' \cdot f(n)$ , luego  
 $g \in O(f(n))$

( $\supseteq$ )  $g \in O(f(n)) \Leftrightarrow \exists c \in \mathbb{R}^+, n_0 \in \mathbb{N}$  tal que  $\forall n \geq n_0. g(n) \leq c \cdot f(n)$

Entonces tomando  $c' = \frac{c}{a}$  se cumple que  $\forall n \geq n_0. g(n) \leq c' \cdot a \cdot f(n)$ ,  
luego  $g \in O(a \cdot f(n))$

- La base del logaritmo no importa:  $O(\log_a n) = O(\log_b n)$ , con  $a, b > 1$ .

$$\log_b n = \frac{\log_a n}{\log_a b}$$

Y ESTA SEGUNDA PROPIEDAD EXPLICA QUE DA IGUAL QUE SEA EN BASE 10, QUE EN BASE 22 QUE EN BASE 14 QUE EL COSTE DE DOS ALGORITMOS O QUE EL ORDEN DE DOS ALGORITMOS LOGARÍTMICOS ES EL MISMO.

## Propiedades

NO IMPORTA LA BASE DEL ALGORITMO. SI QUE IMPORTA LA BASE DE LAS POTENCIAS SI ELEVAMOS A N

- Si  $f \in O(g)$  y  $g \in O(h)$ , entonces  $f \in O(h)$ .

Si una función f pertenece al orden de una función "g" y esa misma función g pertenece al orden de otra función "h" esto implica por la propiedad transitiva que la función f pertenece al orden de la función h.

## Propiedades

- Si  $f \in O(g)$  y  $g \in O(h)$ , entonces  $f \in O(h)$ .

$f \in O(g) \Rightarrow \exists c_1 \in \mathbb{R}^+, n_1 \in \mathbb{N}$  tal que  $\forall n \geq n_1. f(n) \leq c_1 \cdot g(n)$   
 $g \in O(h) \Rightarrow \exists c_2 \in \mathbb{R}^+, n_2 \in \mathbb{N}$  tal que  $\forall n \geq n_2. g(n) \leq c_2 \cdot h(n)$

TRANSITIVIDAD

# Propiedades

CREO QUE ES LA PROPIEDAD TRANSITIVA.

- Si  $f \in O(g)$  y  $g \in O(h)$ , entonces  $f \in O(h)$ .

$$f \in O(g) \Rightarrow \exists c_1 \in \mathbb{R}^+, n_1 \in \mathbb{N} \text{ tal que } \forall n \geq n_1. f(n) \leq c_1 \cdot g(n)$$
$$g \in O(h) \Rightarrow \exists c_2 \in \mathbb{R}^+, n_2 \in \mathbb{N} \text{ tal que } \forall n \geq n_2. g(n) \leq c_2 \cdot h(n)$$

Tomando  $n_0 = \max(n_1, n_2)$  y  $c = c_1 \cdot c_2$ , se cumple

$$\forall n \geq n_0. f(n) \leq c_1 \cdot g(n) \leq c_1 \cdot c_2 \cdot h(n)$$

Y por tanto  $f \in O(h)$ .

Tengo que aprender a leer la expresión de arriba porque me ayudaría a saber explicar si una función pertenece al orden de otra función.

Dice que una función  $f$  pertenece al orden de una función  $g$  si existe una cota perteneciente al conjunto de los reales y un natural tal que para toda la base ( $n$ ) si sustituimos  $n_1$  por  $n$  en la función es menor o igual que sustituir la cota en  $g$  y así con el otro enunciado.

## Propiedades

- Regla de la suma:  $O(f + g) = O(\max(f, g))$ .

Por ejemplo un bucle que recorre un vector de n elementos y otro bucle que recorre un vector de m elementos. Su coste va a ser el máximo entre recorrer ambos bucles.

## Propiedades

- Regla de la suma:  $O(f + g) = O(\max(f, g))$ .  
 $(\subseteq)$   $h \in O(f + g) \Rightarrow \exists c \in \mathbb{R}^+, n_0 \in \mathbb{N}. \forall n \geq n_0. h(n) \leq c \cdot (f(n) + g(n))$

## Propiedades

- Regla de la suma:  $O(f + g) = O(\max(f, g))$ .

( $\subseteq$ )  $h \in O(f + g) \Rightarrow \exists c \in \mathbb{R}^+, n_0 \in \mathbb{N}. \forall n \geq n_0. h(n) \leq c \cdot (f(n) + g(n))$

Pero  $f \leq \max(f, g)$  y  $g \leq \max(f, g)$ , luego

$$h(n) \leq c \cdot (\max(f(n), g(n)) + \max(f(n), g(n))) = 2 \cdot c \cdot \max(f(n), g(n))$$

Luego tomando  $c' = 2 \cdot c$  se cumple que

$\forall n \geq n_0. h(n) \leq c' \cdot \max(f(n), g(n))$  y por tanto  $h \in O(\max(f, g))$ .

## Propiedades

- Regla de la suma:  $O(f + g) = O(\max(f, g))$ .

( $\subseteq$ )  $h \in O(f + g) \Rightarrow \exists c \in \mathbb{R}^+, n_0 \in \mathbb{N}. \forall n \geq n_0. h(n) \leq c \cdot (f(n) + g(n))$

Pero  $f \leq \max(f, g)$  y  $g \leq \max(f, g)$ , luego

$$h(n) \leq c \cdot (\max(f(n), g(n)) + \max(f(n), g(n))) = 2 \cdot c \cdot \max(f(n), g(n))$$

Luego tomando  $c' = 2 \cdot c$  se cumple que

$\forall n \geq n_0. h(n) \leq c' \cdot \max(f(n), g(n))$  y por tanto  $h \in O(\max(f, g))$ .

( $\supseteq$ )  $h \in O(\max(f, g)) \Rightarrow$

$$\exists c \in \mathbb{R}^+, n_0 \in \mathbb{N}. \forall n \geq n_0. h(n) \leq c \cdot \max(f(n), g(n))$$

## Propiedades

- Regla de la suma:  $O(f + g) = O(\max(f, g))$ .

( $\subseteq$ )  $h \in O(f + g) \Rightarrow \exists c \in \mathbb{R}^+, n_0 \in \mathbb{N}. \forall n \geq n_0. h(n) \leq c \cdot (f(n) + g(n))$

Pero  $f \leq \max(f, g)$  y  $g \leq \max(f, g)$ , luego

$$h(n) \leq c \cdot (\max(f(n), g(n)) + \max(f(n), g(n))) = 2 \cdot c \cdot \max(f(n), g(n))$$

Luego tomando  $c' = 2 \cdot c$  se cumple que

$\forall n \geq n_0. h(n) \leq c' \cdot \max(f(n), g(n))$  y por tanto  $h \in O(\max(f, g))$ .

( $\supseteq$ )  $h \in O(\max(f, g)) \Rightarrow$

$\exists c \in \mathbb{R}^+, n_0 \in \mathbb{N}. \forall n \geq n_0. h(n) \leq c \cdot \max(f(n), g(n))$

Pero  $\max(f, g) \leq f + g$ , luego  $h \in O(f + g)$  trivialmente.

## Propiedades

El orden de la suma de dos funciones es el orden maximo entre las dos funciones.

- Regla de la suma:  $O(f + g) = O(\max(f, g))$ .

$$(\subseteq) \quad h \in O(f + g) \Rightarrow \exists c \in \mathbb{R}^+, n_0 \in \mathbb{N}. \forall n \geq n_0. h(n) \leq c \cdot (f(n) + g(n))$$

Pero  $f \leq \max(f, g)$  y  $g \leq \max(f, g)$ , luego

$$h(n) \leq c \cdot (\max(f(n), g(n)) + \max(f(n), g(n))) = 2 \cdot c \cdot \max(f(n), g(n))$$

Luego tomando  $c' = 2 \cdot c$  se cumple que

$$\forall n \geq n_0. h(n) \leq c' \cdot \max(f(n), g(n)) \text{ y por tanto } h \in O(\max(f, g)).$$

$$(\supseteq) \quad h \in O(\max(f, g)) \Rightarrow$$

$$\exists c \in \mathbb{R}^+, n_0 \in \mathbb{N}. \forall n \geq n_0. h(n) \leq c \cdot \max(f(n), g(n))$$

Pero  $\max(f, g) \leq f + g$ , luego  $h \in O(f + g)$  trivialmente.

- Regla del producto: Si  $g_1 \in O(f_1)$  y  $g_2 \in O(f_2)$ , entonces  
 $g_1 \cdot g_2 \in O(f_1 \cdot f_2)$ .

El orden del producto es el orden del producto de las dos funciones

## Teorema del límite

### IMPORTANTE TEOREMA DEL LÍMITE

- $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = m \in \mathbb{R}^+ \Rightarrow f \in O(g) \text{ y } g \in O(f)$

Si es un número ambas pertenecen al orden de la una y de la otra.

Para demostrar si una función pertenece a otra haremos el teorema del límite, el cual consiste en hallar el límite cuando  $n$  tiende a infinito del cociente entre las dos funciones. Necesitaríamos aplicar L'Hôpital (es decir la derivada de lo de arriba y la derivada de lo de abajo) hasta conseguir un resultado.

Tenemos 3 posibles resultados:

- Si el resultado del límite es un número entonces ambas funciones pertenecen al orden de la otra. Esto implica que el orden de ambas funciones es el mismo.

- Si el resultado es 0 entonces la función de arriba pertenece al orden de la función de abajo. Esto implica que el orden de la función de arriba está contenido en el orden de la función de abajo.

- Si el resultado es infinito, entonces la función de abajo pertenece al orden de la función de arriba. Esto implica que el orden de la función de abajo está contenido en el orden de la función de arriba.

Esto es para si en el examen nos piden demostrar si una es del orden de la otra o viceversa.

## Teorema del límite

- $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = m \in \mathbb{R}^+ \Rightarrow f \in O(g) \text{ y } g \in O(f) \Leftrightarrow O(f) = O(g)$

Lo podemos razonar. Si una es del orden de  $3n$  y otra es del orden de  $2n$  entonces el resultado sería  $3/2$  ya que se van las  $n$ 's. Luego ambas son del mismo orden.

Si por ejemplo tenemos una de orden  $n^2$  y otra de orden  $n$ , al hacer el límite se van las  $n$ s y quedaría que el orden de la de abajo pertenece al orden de la de arriba. Por eso también el resultado es infinito.

## Teorema del límite

Y esto no se si implica que son del orden exacto.

- $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = m \in \mathbb{R}^+ \Rightarrow f \in O(g) \text{ y } g \in O(f) \Leftrightarrow O(f) = O(g)$
- $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0 \Rightarrow f \in O(g) \text{ y } g \notin O(f) \Leftrightarrow O(f) \subset O(g)$

Si el resultado de hacer el límite y aplicar L' Hôpital de dos funciones es 0 implica que la función de arriba es del orden de la función de abajo. Por consiguiente, la función de abajo es del omega de la de arriba (el omega juega con las cotas inferiores a contraposición de las del orden). Es como si fueran lo opuesto.

## Teorema del límite

- $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = m \in \mathbb{R}^+ \Rightarrow f \in O(g) \text{ y } g \in O(f) \Leftrightarrow O(f) = O(g)$
- $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0 \Rightarrow f \in O(g) \text{ y } g \notin O(f) \Leftrightarrow O(f) \subset O(g)$
- $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = +\infty \Rightarrow f \notin O(g) \text{ y } g \in O(f) \Leftrightarrow O(f) \supset O(g)$

SI el límite es infinito en este caso positivo, la función de abajo será del orden de la función de arriba. (Cuando digo abajo o arriba me refiero al numerador y al denominador).

ENTONCES, EN EL TEOREMA DEL LÍMITE TENEMOS TRES (3) OPCIONES:

1. LÍMITE=M: IMPLICA QUE AMBAS SON DEL MISMO ORDEN
2. LÍMITE=0:IMPLICA QUE EL DE ARRIBA ES DEL ORDEN DEL DE ABAJO.
3. LÍMITE= INFINITO: IMPLICA QUE EL DE ABAJO ES DEL ORDEN DEL DE ARRIBA

## Teorema del límite

- $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = m \in \mathbb{R}^+ \Rightarrow f \in O(g) \text{ y } g \in O(f) \Leftrightarrow O(f) = O(g)$
- $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0 \Rightarrow f \in O(g) \text{ y } g \notin O(f) \Leftrightarrow O(f) \subset O(g)$
- $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = +\infty \Rightarrow f \notin O(g) \text{ y } g \in O(f) \Leftrightarrow O(f) \supset O(g)$

Por hipótesis sobre el límite sabemos que  $\forall \epsilon > 0 \exists n_0 \in \mathbb{N}$  tal que

$$\forall n \geq n_0. \left| \frac{f(n)}{g(n)} \right| < \epsilon.$$

Tomando  $\epsilon = 1$ , tenemos  $\forall n \geq n_0. f(n) < g(n) \Rightarrow f \in O(g)$

## Teorema del límite

- $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = m \in \mathbb{R}^+ \Rightarrow f \in O(g) \text{ y } g \in O(f) \Leftrightarrow O(f) = O(g)$
- $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0 \Rightarrow f \in O(g) \text{ y } g \notin O(f) \Leftrightarrow O(f) \subset O(g)$
- $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = +\infty \Rightarrow f \notin O(g) \text{ y } g \in O(f) \Leftrightarrow O(f) \supset O(g)$

Por hipótesis sobre el límite sabemos que  $\forall \epsilon > 0. \exists n_0 \in \mathbb{N}$  tal que

$$\forall n \geq n_0. \left| \frac{f(n)}{g(n)} \right| < \epsilon.$$

Tomando  $\epsilon = 1$ , tenemos  $\forall n \geq n_0. f(n) < g(n) \Rightarrow f \in O(g)$

Demostramos  $g \notin O(f)$  por reducción al absurdo.

$$\begin{aligned} g \in O(f) &\Leftrightarrow \exists n_0 \in \mathbb{N}, \exists c \in \mathbb{R}^+ \text{ tal que } \forall n \geq n_0. g(n) \leq cf(n) \\ &\Leftrightarrow \exists n_0 \in \mathbb{N}, \exists c \in \mathbb{R}^+ \text{ tal que } \forall n \geq n_0. \frac{1}{c} \leq \frac{f(n)}{g(n)} \end{aligned}$$

$$\frac{1}{c} = \lim_{n \rightarrow \infty} \frac{1}{c} \leq \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$$

$\frac{1}{c} \leq 0$  contradicción con  $c \in \mathbb{R}^+$ .

## Ejemplos

- $\log n \in O(n)$

$$\lim_{n \rightarrow \infty} \frac{n}{\log n} = \left[ \begin{array}{c} \infty \\ \infty \end{array} \right] = {}^{L'Hopital} \lim_{n \rightarrow \infty} \frac{n \ln 2}{\ln n} = \lim_{n \rightarrow \infty} \frac{\ln 2}{1/n} = \lim_{n \rightarrow \infty} n \ln 2 = \infty$$

No se si está mal derivado. Lo que yo haría sería derivar lo de arriba y lo de abajo de manera individual. El límite de lo de arriba es 1 y el límite de lo de abajo es 1/n.

Por lo tanto quedaría n que al sustituirlo por infinito seria infinito. Esto implica que  $\log n$  es del orden de  $n$  pero no viceversa.

## Ejemplos

- $\log n \in O(n)$

$$\lim_{n \rightarrow \infty} \frac{n}{\log n} = \left[ \begin{array}{c} \infty \\ \infty \end{array} \right] = {}^{L'Hopital} \lim_{n \rightarrow \infty} \frac{n \ln 2}{\ln n} = \lim_{n \rightarrow \infty} \frac{\ln 2}{1/n} = \lim_{n \rightarrow \infty} n \ln 2 = \infty$$

- $P(x) = a_k x^k + a_{k-1} x^{k-1} + \dots + a_1 x + a_0$ , con  $a_k \in \mathbb{R}^+$ ,  $P(x) \in O(x^k)$

$$\lim_{x \rightarrow \infty} \frac{P(x)}{x^k} = a_k > 0$$

## Ejemplos

- $\log n \in O(n)$

$$\lim_{n \rightarrow \infty} \frac{n}{\log n} = \left[ \begin{array}{c} \infty \\ \infty \end{array} \right] = {}^{L' Hopital} \lim_{n \rightarrow \infty} \frac{n \ln 2}{\ln n} = \lim_{n \rightarrow \infty} \frac{\ln 2}{1/n} = \lim_{n \rightarrow \infty} n \ln 2 = \infty$$

- $P(x) = a_k x^k + a_{k-1} x^{k-1} + \dots + a_1 x + a_0$ , con  $a_k \in \mathbb{R}^+$ ,  $P(x) \in O(x^k)$

$$\lim_{n \rightarrow \infty} \frac{P(x)}{x^k} = a_k > 0$$

Para que esto ocurra el límite debería de dar infinito.

- $O(n^k) \subset O(2^n)$

$$\begin{aligned} \lim_{n \rightarrow \infty} \frac{2^n}{n^k} &= \left[ \begin{array}{c} \infty \\ \infty \end{array} \right] = {}^{L' Hopital} \\ &= \lim_{n \rightarrow \infty} \frac{2^n \ln 2}{kn^{k-1}} = \lim_{n \rightarrow \infty} \frac{2^n (\ln 2)^2}{k(k-1)n^{k-2}} = (k \text{ veces}) = \\ &= \lim_{n \rightarrow \infty} \frac{2^n (\ln 2)^k}{k! n^0} = \frac{(\ln 2)^k}{k!} \lim_{n \rightarrow \infty} 2^n = \infty \end{aligned}$$

Para demostrar que el orden de una de las funciones está contenido en la otra debemos comprobar por el teorema del límite las propiedades anteriores.

## Ejemplos

- $\log n \in O(n)$

$$\lim_{n \rightarrow \infty} \frac{n}{\log n} = \left[ \begin{array}{c} \infty \\ \infty \end{array} \right] = {}^{L' Hopital} \lim_{n \rightarrow \infty} \frac{n \ln 2}{\ln n} = \lim_{n \rightarrow \infty} \frac{\ln 2}{1/n} = \lim_{n \rightarrow \infty} n \ln 2 = \infty$$

- $P(x) = a_k x^k + a_{k-1} x^{k-1} + \dots + a_1 x + a_0$ , con  $a_k \in \mathbb{R}^+$ ,  $P(x) \in O(x^k)$

$$\lim_{n \rightarrow \infty} \frac{P(x)}{x^k} = a_k > 0$$

- $O(n^k) \subset O(2^n)$

$$\begin{aligned} \lim_{n \rightarrow \infty} \frac{2^n}{n^k} &= \left[ \begin{array}{c} \infty \\ \infty \end{array} \right] = {}^{L' Hopital} \\ &= \lim_{n \rightarrow \infty} \frac{2^n \ln 2}{kn^{k-1}} = \lim_{n \rightarrow \infty} \frac{2^n (\ln 2)^2}{k(k-1)n^{k-2}} = (k \text{ veces}) = \\ &= \lim_{n \rightarrow \infty} \frac{2^n (\ln 2)^k}{k! n^0} = \frac{(\ln 2)^k}{k!} \lim_{n \rightarrow \infty} 2^n = \infty \end{aligned}$$

el de abajo es del orden del de arriba

- $O(2^n) \subset O(n!)$

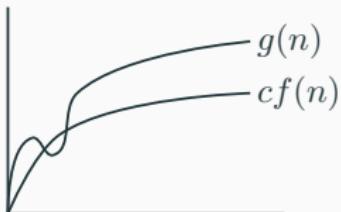
$$\lim_{n \rightarrow \infty} \frac{n!}{2^n} = \lim_{n \rightarrow \infty} \frac{n}{2} \frac{n-1}{2} \cdots \frac{4}{2} \frac{3}{2} \frac{2}{2} \frac{1}{2} \geq \lim_{n \rightarrow \infty} \frac{4}{2} \frac{4}{2} \cdots \frac{4}{2} \frac{3}{2} \frac{2}{2} \frac{1}{2} = \frac{3}{4} \lim_{n \rightarrow \infty} 2^{n-3} = \infty$$

El de abajo es del orden del de arriba.

## Cotas inferiores

**Definición** Sea  $f : \mathbb{N} \longrightarrow \mathbb{R}_0^+$ . El conjunto  $\Omega(f(n))$ , leído **omega de  $f(n)$** , se define como

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



Cuando decimos que el coste de un algoritmo está en  $\Omega(f(n))$  lo que estamos diciendo es que la complejidad del algoritmo *no es mejor que* la representada por la función  $f$ .

**Principio de dualidad**  $g \in \Omega(f) \Leftrightarrow f \in O(g)$

Es como lo contrario, si  $f$  pertenece al orden de  $g$  entonces  $g$  pertenece al omega de  $f$ .

$g$

$f$

## Teorema del límite

Funciona al revés que con los órdenes.

- $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = m \in R^+ \Rightarrow g \in \Omega(f) \text{ y } f \in \Omega(g)$
- $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0 \Rightarrow g \in \Omega(f) \text{ y } f \notin \Omega(g)$
- $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = +\infty \Rightarrow g \notin \Omega(f) \text{ y } f \in \Omega(g)$

Recordar que si al hacer los límites:

- El resultado es un número natural entonces ambas serán del orden y del omega de ambas.
- El resultado es 0 el numerador será del orden del denominador y por consiguiente el denominador será del omega del numerador.
- El resultado es infinito el denominador será del orden del numerador y el numerador será del omega del denominador.

## Orden exacto

**Definición** El conjunto de funciones  $\Theta(f(n))$ , leído **del orden exacto de  $f(n)$** , se define como

$$\Theta(f(n)) = O(f(n)) \cap \Omega(f(n)).$$

Intersección = y.

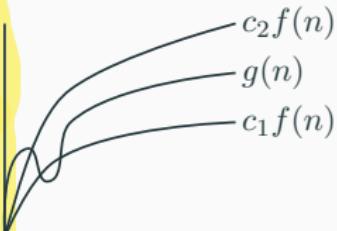
Vamos, que una función es del orden exacto de otra función si pertenece a su omega y a su orden. Lo cual, yo creo que implica que sean ambas del mismo orden.

$$\Theta(f(n)) = \{g : N \longrightarrow R_0^+ \mid \exists c_1, c_2 \in R^+, n_0 \in N. \forall n \geq n_0. c_1 f(n) \leq g(n) \leq c_2 f(n)\}$$

Orden exacto: Que esté entre las dos cotas.

Orden: que sea menor que la cota superior de la función.

Omega: que sea superior que la cota superior de la función.



Si al hacer el teorema del límite el resultado es un número distinto de infinito y de 0.

No se si el orden exacto es como si fueran del mismo orden, que son del orden exacto

## Teorema del límite

Si el resultado del límite es un número entonces serán del orden exacto, verificado en este primer ejemplo

- $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = m \in R^+ \Rightarrow g \in \Theta(f) \text{ y } f \in \Theta(g)$
- $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0 \Rightarrow f \in O(g) \text{ pero } f \notin \Theta(g)$
- $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = +\infty \Rightarrow g \in O(f) \text{ pero } g \notin \Theta(f)$

### Ejemplo:

$$P(x) = a_k x^k + a_{k-1} x^{k-1} + \dots + a_1 x + a_0, \text{ con } a_k > 0,$$

$$Q(x) = b_l x^l + b_{l-1} x^{l-1} + \dots + b_1 x + b_0, \text{ con } b_l > 0, \text{ entonces}$$

$$\lim_{n \rightarrow \infty} \frac{P(x)}{Q(x)} = 0 \text{ si } k < l \Rightarrow P(x) \in O(Q(x))$$

$$\lim_{n \rightarrow \infty} \frac{P(x)}{Q(x)} = \frac{a_k}{b_l} \text{ si } k = l \Rightarrow P(x) \in \Theta(Q(x))$$

$$\lim_{n \rightarrow \infty} \frac{P(x)}{Q(x)} = \infty \text{ si } k > l \Rightarrow P(x) \in \Omega(Q(x))$$

## Jerarquía de órdenes de complejidad

Es lo mismo que decir que 1 pertenece al orden del log, que pertenece al orden de n...

$$O(1) \subset O(\log n) \subset O(n) \subset O(n \log n) \subset O(n^2) \subset \dots \subset O(n^k) \subset \dots \subset O(2^n) \subset O(n!)$$

razonables en la práctica

tratables

intratables

Luego el log pertenece al omega de 1.

Pero hay que ser exactos.

La notación  $O(f(n))$  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

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

Ser exactos, no dar rangos muy elevados.

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

Lo que quiere en esta parte de la teoría

es que una función, en este caso  $3n$

pertenece a distintos ordenes, pero debemos ser explícitos y decir cual es el orden menor de la función, porque todas las funciones pertenecen al orden de  $n!$

pero no es exacto decir que esta función pertenece al orden de  $n!$

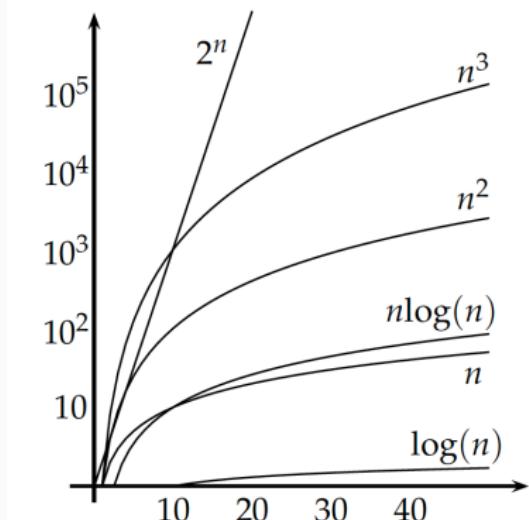
$$3n \in O(n)$$

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

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

Lo correcto y lo exacto es decir que  $3n$  pertenece al orden de  $n$  ya que el orden anterior es el de  $\log n$ , al cual no pertenece.

## Ordenes de complejidad



Aquí podemos ver que una función  $n$  es del orden de la  $n \log(n)$  ya que es inferior que la cota superior, que en este caso es la función  $n \log(n)$

Supongamos que tenemos 6 algoritmos diferentes 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.

Vemos que pasa si a 6 algoritmos distintos le duplicamos el tamaño de los datos. En el caso de la función logarítmica tarda 15 minutos más, en el caso de la lineal tarda 1 hora más (el doble), la de  $n \log(n)$  tarda 2 horas y 30 min,  $n^2$  tarda el cuádruple...

## Órdenes de complejidad

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

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

## Análisis de coste de algoritmos iterativos

---

Podemos simplificar el cálculo del coste (en el caso peor) de un algoritmo gracias a la teoría presentada.

1. Las instrucciones de asignación, entrada/salida, o expresiones aritméticas tienen un coste en  $\Theta(1)$ . Coste constante tienen las operaciones de asignación

Podemos simplificar el cálculo del coste (en el caso peor) de un algoritmo gracias a la teoría presentada.

1. Las instrucciones de asignación, entrada/salida, o expresiones aritméticas tienen un coste en  $\Theta(1)$ . **coste constante**
2. Para calcular el coste de una composición secuencial se utiliza la regla de la suma. Si el coste de  $S_1$  está en  $\Theta(f_1(n))$  y el coste de  $S_2$  está en  $\Theta(f_2(n))$ , entonces el coste de  $S_1 ; S_2$  está en  $\Theta(\max(f_1(n), f_2(n)))$ .  
Por ejemplo, en un if de comparar dos cosas, la que tenga mayor.

Podemos simplificar el cálculo del coste (en el caso peor) de un algoritmo gracias a la teoría presentada.

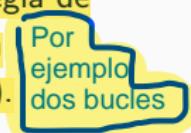
1. Las instrucciones de asignación, entrada/salida, o expresiones aritméticas tienen un coste en  $\Theta(1)$ .
2. Para calcular el coste de una composición secuencial se utiliza la regla de la suma. Si el coste de  $S_1$  está en  $\Theta(f_1(n))$  y el coste de  $S_2$  está en  $\Theta(f_2(n))$ , entonces el coste de  $S_1 ; S_2$  está en  $\Theta(\max(f_1(n), f_2(n)))$ .
3. El coste de una instrucción condicional `if B S1 else S2` está en  $\Theta(\max(f_B(n), f_1(n), f_2(n)))$ , donde  $f_B$ ,  $f_1$  y  $f_2$  representan respectivamente el coste de evaluar la condición del bucle y cada una de las dos ramas.

Podemos simplificar el cálculo del coste (en el caso peor) de un algoritmo gracias a la teoría presentada.

1. Las instrucciones de asignación, entrada/salida, o expresiones aritméticas tienen un coste en  $\Theta(1)$ .
2. Para calcular el coste de una composición secuencial se utiliza la regla de la suma. Si el coste de  $S_1$  está en  $\Theta(f_1(n))$  y el coste de  $S_2$  está en  $\Theta(f_2(n))$ , entonces el coste de  $S_1 ; S_2$  está en  $\Theta(\max(f_1(n), f_2(n)))$ .
3. El coste de una instrucción condicional `if B S1 else S2` está en  $\Theta(\max(f_B(n), f_1(n), f_2(n)))$ , donde  $f_B$ ,  $f_1$  y  $f_2$  representan respectivamente el coste de evaluar la condición del bucle y cada una de las dos ramas.
4. Para calcular el coste de una instrucción iterativa `while B {S}`, si el número de iteraciones está en  $\Theta(f_{iter}(n))$ :
  - Si todas las iteraciones tienen el mismo coste, utilizamos la regla del producto y el coste total del bucle está en  $\Theta(f_{B,S}(n) \cdot f_{iter}(n))$ , siendo  $f_{B,S}$  el coste de ejecutar la evaluación de la condición del bucle seguido del cuerpo del bucle.

# Análisis de algoritmos iterativos

Podemos simplificar el cálculo del coste (en el caso peor) de un algoritmo gracias a la teoría presentada.

1. Las instrucciones de asignación, entrada/salida, o expresiones aritméticas tienen un coste en  $\Theta(1)$ .
2. Para calcular el coste de una composición secuencial se utiliza la regla de la suma. Si el coste de  $S_1$  está en  $\Theta(f_1(n))$  y el coste de  $S_2$  está en  $\Theta(f_2(n))$ , entonces el coste de  $S_1 ; S_2$  está en  $\Theta(\max(f_1(n), f_2(n)))$ .  


Por ejemplo  
dos bucles
3. El coste de una instrucción condicional `if B S1 else S2` está en  $\Theta(\max(f_B(n), f_1(n), f_2(n)))$ , donde  $f_B$ ,  $f_1$  y  $f_2$  representan respectivamente el coste de evaluar la condición del bucle y cada una de las dos ramas.
4. Para calcular el coste de una instrucción iterativa `while B {S}`, si el número de iteraciones está en  $\Theta(f_{iter}(n))$ :
  - Si todas las iteraciones tienen el mismo coste, utilizamos la regla del producto y el coste total del bucle está en  $\Theta(f_{B,S}(n) \cdot f_{iter}(n))$ , siendo  $f_{B,S}$  el coste de ejecutar la evaluación de la condición del bucle seguido del cuerpo del bucle.
  - Si el coste de cada iteración es distinto, realizamos una suma desde 1 hasta  $f_{iter}(n)$  de los costes individuales de cada iteración.

Preguntarnos cuál es el coste de un programa que tiene dos bucles no anidados (si no me equivoco el coste será el de mayor coste).

## Producto de matrices cuadradas

---

```
void producto (const int A[N][N], const int B[N][N],
               int C[N][N], int n)
{
    for (int i=0;i<n;i++) e igual para este
        for (int j=0;j<n;j++) lo mismo para este
            { C[i][j]=0;
                for (int k=0;k<n;k++) El bucle interno tiene un coste lineal n
                    C[i][j]+= A[i][k]*B[k][j];
            }
}
```

---

El coste sería igual al coste del bucle más interno ( $n$ ), por el segundo bucle ( $n$ ) por el bucle más externo que es  $n$ . Al multiplicar, nos sale que el coste del algoritmo es  $n^3$ .

Como la base de una potencia SI que importa entonces no es lo mismo decir que es del orden de  $n^2$  que del orden de  $n^3$ ;  $n^2$  pertenece al orden de  $n^3$ .

## Producto de matrices cuadradas

---

```
void producto (const int A[N][N], const int B[N][N],
               int C[N][N], int n)
{
    for (int i=0;i<n;i++)
        for (int j=0;j<n;j++)
            { C[i][j]=0;
              for (int k=0;k<n;k++)
                  C[i][j]+= A[i][k]*B[k][j];
            }
}
```

---

$$T(n) \in \Theta(n^3)$$

## Ordenación por selección

ordenación por selección cambia elementos para ordenarlos.

```
void seleccion(vector<int> & v)
{
    int i, j, pmin, temp;
    for (i = 0; i < v.size()-1; i++) coste v.size-1 = aprox v.size()
    { pmin = i;
        for (j = i+1; j < v.size(); j++) este tiene coste v.size
            if (v[j] < v[pmin]) pmin = j;
        temp = v[i]; v[i] = v[pmin]; v[pmin] = temp;
    }
}
```

El coste sería cuadrático del orden exacto de  $n^2$ . El bucle más interno va desde  $j$  hasta  $v.size()$  y el más externo hasta  $v.size()-1$  lo que es lo mismo que  $v.size()$ . Por tanto  $v.size()^2$

## Ordenación por selección

---

```
void seleccion(vector<int> & v)
{
    int i, j, pmin, temp;
    for (i = 0; i < v.size()-1; i++)
    { pmin = i;
        for (j = i+1; j < v.size(); j++)
            if (v[j] < v[pmin]) pmin = j;
        temp = v[i]; v[i] = v[pmin]; v[pmin] = temp;
    }
}
```

---

Sea  $n = v.size()$ , entonces

$$T(n) \in \Theta\left(\sum_{i=0}^{n-2} (n - i - 1)\right) = \Theta(n^2)$$

## Determinar si una matriz cuadrada es simétrica

---

```
bool simetrica(vector<vector<int>>& m)
{
    bool b=true;
    int i=0,j;
    while (i<v.size() && b)
    {
        j=i+1;
        while (j<v.size() && b)
        {
            b=(m[i][j]==m[j][i]);
            j++;
        }
        i++;
    }
    return b;
}
```

---

Vamos a analizar el caso mejor y el caso peor porque se trata de dos bucles while lo cual implica que no siempre se entrará en el bucle. En el mejor de los casos se entrará una vez al bucle (en el caso de que b sea false desde la primera iteración) y por tanto el coste será constante del orden exacto de 1. En el peor de los casos es que el bucle sea siempre true y que por tanto se encargue siempre de ordenar todos los elementos. Entonces entrará  $v.size$  veces a cada bucle y el coste del algoritmo será de  $n^2$ . (Coste cuadrático)

## Determinar si una matriz cuadrada es simétrica

---

```
bool simetrica(vector<vector<int>>& m)
{
    bool b=true;
    int i=0, j;
    while (i<v.size() && b)
    {
        j=i+1;
        while (j<v.size() && b)
        {
            b=(m[i][j]==m[j][i]);
            j++;
        }
        i++;
    }
    return b;
}
```

---

Sea  $n = v.size()$ , entonces

$$T_{\min}(n) \in \Theta(1)$$

$$T_{\max}(n) \in \Theta(n^2)$$

Mejor de los casos.

Peor de los casos.

## Instrucción crítica

Se puede simplificar más el cálculo si hacemos uso del concepto de **instrucción crítica**: instrucción que más veces se ejecuta.

---

```
1 void seleccion(vector<int> & v)
2 {
3     int i, j, pmin, temp;
4     for (i = 0; i < v.size()-1; i++)
5     { pmin = i;
6         for (j = i+1; j < v.size(); j++)
7             if (v[j] < v[pmin]) pmin = j;
8         temp = v[i]; v[i] = v[pmin]; v[pmin] = temp;
9     }
10 }
```

---

Instrucción crítica :  $v[j] < v[pmin]$  se ejecuta

$$\sum_{i=0}^{n-2} (n - i - 1) = \frac{n(n-1)}{2} \in O(n^2)$$

siendo  $n = v.size()$ .

## Ordenación por inserción

---

```
void insercion(vector<int> & v)
{
    int i, j, aux;
    for (i = 1; i < v.size(); i++)
    { aux = v[i]; j = i-1;
        while (j>=0 && v[j]>aux)
        { v[j+1] = v[j];
            j--;
        }
        v[j+1]=aux;
    }
}
```

---

Instrucción crítica:  $j \geq 0$

$$\text{caso peor: } T_{\max}(n) = \sum_{i=1}^{n-1} (i + 1) = \frac{(n+2)(n-1)}{2} \in \Theta(n^2)$$

$$\text{caso mejor: } T_{\min}(n) = \sum_{i=1}^{n-1} 1 = n - 1 \in \Theta(n)$$

## Ordenación por inserción

---

```
void insercion(vector<int> & v)
{
    int i, j, aux;
    for (i = 1; i < v.size(); i++)
    { aux = v[i]; j = i-1;
        while (j>=0 && v[j]>aux)
        { v[j+1] = v[j];
            j--;
        }
        v[j+1]=aux;
    }
}
```

---

Instrucción crítica:  $j \geq 0$

$$\text{caso peor: } T_{\max}(n) = \sum_{i=1}^{n-1} (i + 1) = \frac{(n+2)(n-1)}{2} \in \Theta(n^2)$$

$$\text{caso mejor: } T_{\min}(n) = \sum_{i=1}^{n-1} 1 = n - 1 \in \Theta(n)$$

$$\text{caso promedio: } \Theta(n^2)$$