Tema 3
ALGORITMIA

Parte I. *Introducción al Análisis de Algoritmos* .

2016/2017

Irene Martínez Masegosa

Depto. de Informática

mailto:irene@ual.es

Contents

3.1-2

1 Estudio de Algoritmos

Necesitamos resolver un problema algorítmico

Puede haber varios algoritmos correctos que resuelvan el mismo problema

↓

Debemos elegir el **mejor algoritmo**

↓

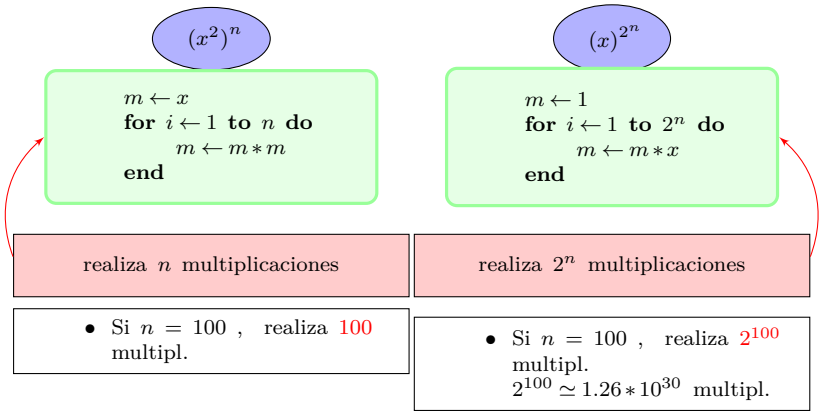
¿Cuál elegiríamos? ¿Qué criterio usaríamos para seleccionarlo?

- El más elegante?
- El más legible?
- El que use menos memoria?
- El más rápido?

3.1-3

Ejemplo:

Calcular el valor de x^{2^n}. Dos algoritmos: ¿Cuál es mejor?



Si dispongo de un ordenador que realiza un
millón de multiplicaciones por segundo,10^6:

• Tardará $100/10^6$ segundos $=$ **0.0001 segundos**	• Tardará $1.26 * 10^{24}$ seg $\simeq 4 * 10^{16}$ **AÑOS!**

Nota: 1 año = 31 556 926 segundos $\simeq 31.5 * 10^6$

3.1-4

El Análisis de Algoritmos

- El **análisis de algoritmos** permite comparar algoritmos en función de los recursos computacionales que consumen
- Considerar dos algoritmos y decir que uno es mejor que otro porque es más **eficiente** al utilizar estos recursos, o simplemente porque necesita menos

Eficiencia

Podemos definir la eficiencia de un algoritmo en base a dos factores:

- Cantidad de MEMORIA que necesita para resolver el problema,
- TIEMPO de ejecución empleado.

- No siempre coincidirán consumo de memoria óptimo y tiempo de ejecución mínimo.
- Normalmente, ambos factores compiten, y habrá que adoptar compromisos entre ambos costes.

3.1-5

Razones para Analizar Algoritmos

Predecir cómo es su ejecución

- Saber si un programa termina
- Cuánto tiempo tarda en terminar

Comparar algoritmos

- Saber qué cambios pueden hacer un algoritmo más rápido
- Cómo podemos mejorar un programa

Descubrir nuevos métodos para resolver problemas

- Permite desarrollar nueva tecnología
- Nuevas líneas de investigación

3.1-6

Algoritmo

*Un **algoritmo** es un conjunto de reglas para resolver un **problema***

- Conjunto de reglas completamente **definido**.
  La implementación en un lenguaje de programación: PROGRAMA

Conjunto de datos de entrada
(puede ser vacío)
⇓
número **finito** de pasos en un tiempo finito
⇓
Conjunto de datos de salida
(puede no haber solución)

- Es muy importante no confundir **algoritmo y problema**
- Podemos tener diferentes algoritmos para resolver el mismo problema.

3.1-7

¿Cómo realizamos el Análisis del Algoritmo?

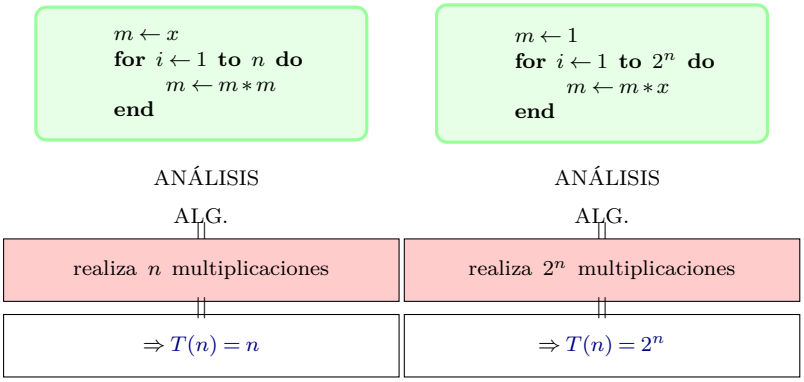
- El **análisis de algoritmos** se realiza en base al **número de elementos que el algoritmo procesa como entrada**: n.
- Ejemplos:
  - n = núm. de elementos del vector
  - n = grado de un polinomio
  - n = núm. de filas y/o columnas de una matriz
  - n = núm. de términos en una sucesión
  - n = valor de la potencia (ejemplo cálculo de x^{2^n})
- Se dice que n es el **tamaño del problema**,
  y para cada problema es necesario analizar la naturaleza de los datos para determinar qué **parámetro** define su **tamaño**.

  Por ejemplo en el cálculo de x^{2^n} el número de multiplicaciones depende del valor de n

3.1-8

Coste del Algoritmo en función de n

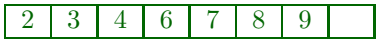
- En el análisis de un algoritmo se estima el **número de operaciones** realizadas en base a n y se obtiene una función $T(n)$ que representa el coste del algoritmo en función del tamaño de la entrada:



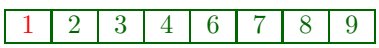
3.1-9

¿Cuántas Operaciones?

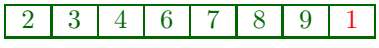
- El número de operaciones necesarias para resolver el problema dependerá no solo del **tamaño** n, sino también de **cómo se presentan los datos de entrada**, y podemos obtener desde una solución especialmente favorable hasta otra muy costosa.
- Imaginemos la inserción de un elemento en un **array de tamaño** n:



  - Si inserto al PRINCIPIO hay que desplazar TODOS los elementos a la derecha para "abrir hueco"$\Rightarrow$ n desplazamientos



  - Si inserto al FINAL $\Rightarrow$ 0 desplazamientos



3.1-10

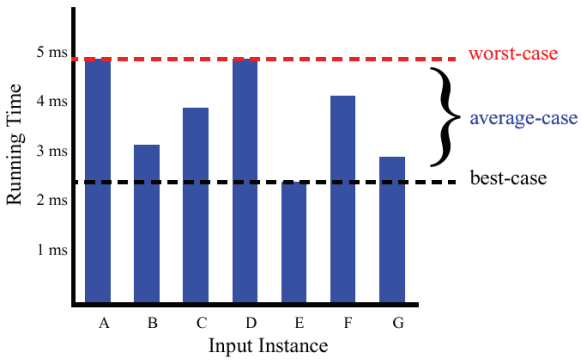
Posibilidades de estudio del Tiempo de Ejecución o *Running Time*

- Entonces, teniendo en cuenta cómo son los datos de entrada, al analizar un algoritmo podemos considerar tres posibles situaciones:
  - **Caso mejor** del algoritmo: en el que el algoritmo es más eficiente, es decir, se ejecuta con el menor número de pasos; es el tiempo empleado por los ejemplares más sencillos o concretos:
    Ej: ordenar un vector ya ordenado, multiplicar por 0,...
  - **Caso peor** del algoritmo: consideramos los ejemplares en los que el algoritmo requiera más tiempo (más pasos)
    Ej: ordenar un vector que ya está ordenado en orden inverso, insertar al principio de un array,
  - **Caso medio** del algoritmo: cualquier otro ejemplar del problema.
    Ej: insertar en una posición intermedia de un vector.
    * Requiere un conocimiento *a priori* acerca de la distribución de los casos que hay que resolver. Es difícil de obtener. Siempre estará comprendido entre los otros dos.

3.1-11

Estudio del Caso Peor: worst case

- En adelante **nos centraremos en el estudio del CASO PEOR** (*worst case running time*)
  - Es más fácil de analizar
  - Crucial en aplicaciones como juegos, finanzas, robótica,...



3.1-12

- Por tanto consideraremos que $T(n)$ **es una función que mide el número de operaciones realizadas por el algoritmo para resolver un problema de tamaño** n, en el caso peor.
- **Será una medida aproximada de la eficiencia del algoritmo**

EJEMPLOS

- Ejemplo: Buscar un elemento en un array desordenado. ¿Cuál es el caso peor? y el caso mejor?
- Ejemplo: Insertar un elemento en un array ordenado. ¿Cuál es el caso peor? y el caso mejor?

3.1-13

¿Estudio teórico o estudio empírico?

- Podemos estudiar el tiempo de ejecución de una algoritmo de **forma empírica**:
  lo **implementamos** en una lenguaje de programación y **medimos el tiempo que tarda en ejecutarse para diferentes datos de entrada**
- Por otro lado, podemos estudiar de **forma teórica** un algoritmo y **estimar**, **en base al número de operaciones que realiza**, el tiempo de ejecución.

¿Cuáles son las ventajas e inconvenientes de cada uno?

3.1-14

2 Estudio Empírico

¿Qué es el estudio experimental de un algoritmo?

Estudio Empírico (*a posteriori*) o estudio experimental

- Consiste en **implementar el algoritmo** en un lenguaje de programación y **medir el tiempo** de ejecución empleado, realizando **pruebas con diferentes tamaños** del problema.
- Podríamos decir que se realiza un benchmark del programa.

- IMPORTANTE: Los resultados de tiempo serán **diferentes** si ejecutamos el programa en **otro ordenador** o lo implementamos con **otro lenguaje de programación**.

3.1-15

¿Cómo realizamos el estudio experimental?

1. Escribir un programa que implemente el algoritmo como un método
2. Usar un método, como `currentTimeMillis()` o `nanoTime()` que devuelve la hora actual del reloj del sistema desde un momento arbitrario de inicio.
3. Medimos el tiempo justo **antes y después del método que evaluamos**, calculamos la diferencia entre los dos valores de tiempo y obtenemos el número exacto de segundos (o fracción) empleados en la ejecución.
   NOTA: NO incluimos operaciones de E/S dentro del método
4. Repetimos el paso anterior variando el tamaño de la entrada del método y su composición:
   Se recomienda ir doblando el tamaño de n
5. Analizar los valores de tiempo obtenidos mediante gráficas y/o tablas y obtener a a partir de estas la función $T(n)$

3.1-16

Ejemplo: Factorial en Java

```java
package org.lya.tema03;
import java.math.BigInteger;

public class Factorial {

    public static long factorialIterativo(int n){
        if (n == 0)
            return 1;
        long fact = 1;
        for(int i=1; i<=n; i++){
            fact = fact * i;
        }
        return fact;
    }

    public static BigInteger factorialBigInt(int n) {
        BigInteger resultado = BigInteger.ONE;
        for (int i = 1; i <= n; i++)
            resultado = resultado.multiply(new BigInteger(i + ""));
        return resultado;
    }

    public static int factorialRecursivo(int n){
        if (n == 0)
            return 1;
        else
            return n * factorialRecursivo(n - 1);
    }
```



```java
    public static void main(String[] args) {
        int N= 10;

        long startTime = System.nanoTime();
        long fac = factorialIterativo(N);
        long endTime = System.nanoTime();
        System.out.println("F.  Iterativo de " +N +"= " + fac  +"
            calculado en " + (endTime-startTime) +" nanosegundos");

        startTime = System.nanoTime();
        fac = factorialRecursivo(N);
        endTime = System.nanoTime();
        System.out.println("F.  Recursivo de " +N +"= " + fac  +"
            calculado en " + (endTime-startTime) +" nanosegundos");

        startTime = System.nanoTime();
        BigInteger facB = factorialBigInt(N);
        endTime = System.nanoTime();
        System.out.println("F. BigInteger de " +N +"= " + facB +"
            calculado en " + (endTime-startTime) +" nanosegundos");
    }
}
```

```
F. Iterativo de 10= 3628800 calculado en 3345 nanosegundos
F. Recursivo de 10= 3628800 calculado en 4148 nanosegundos
F. BigInteger de 10= 3628800 calculado en 977276 nanosegundos
```


Midiendo el tiempo con `currentTimeMillis()`

```java
    public static void main(String[] args) {
        int N= 10;

        long startTime = System.currentTimeMillis();
        long fac = factorialIterativo(N);
        long endTime = System.currentTimeMillis();
        System.out.println("F.  Iterativo de " +N +"= " + fac  +"
            calculado en " + (endTime-startTime) +" milisegundos");
```

```
37
38          startTime = System.currentTimeMillis();
39          fac = factorialRecursivo(N);
40          endTime = System.currentTimeMillis();
41          System.out.println("F.  Recursivo de " +N +"= " + fac  +"
                calculado en " + (endTime-startTime) +" milisegundos");
42
43          startTime = System.currentTimeMillis();
44          BigInteger facB = factorialBigInt(N);
45          endTime = System.currentTimeMillis();
46          System.out.println("F. BigInteger de " +N +"= " + facB +"
                calculado en " + (endTime-startTime)+" milisegundos");
47      }
48
49  }
```

```
F. Iterativo de 10= 3628800 calculado en 0 milisegundos
F. Recursivo de 10= 3628800 calculado en 0 milisegundos
F. BigInteger de 10= 3628800 calculado en 2 milisegundos
```


Ejemplo: Estudio Empírico de un Algoritmo

- ***Suma***(n): *suma los enteros de* 1 *a* n

```
int Suma(n)
        s ← 0
        Para i ← 1 hasta n hacer
              s ← s + i
        Devolver s
```

  - Obtenemos el tiempo ANTES y DESPUÉS del método que evaluamos:

```
SumaEnteros()
        n ←LeerEntero()

        start ← currentTimeMillis()
        Sum ← Suma(n)
        end ← currentTimeMillis()

        tiempoEmpleadoenSegundos ← (end − start)/1000
        Print Sum
        Print  tiempoEmpleadoenSegundos
```

  - Y restamos para saber el tiempo que se ha empleado.
  - Para cada valor de n deberíamos repetir la toma de tiempos varias veces y quedarnos con el **tiempo medio**.

- Salida: para n =10.000 enteros , T. Medio $\simeq$ 0.0019 segundos

```
Sum = 50005000 required 0.0018950 seconds
Sum = 50005000 required 0.0018620 seconds
Sum = 50005000 required 0.0019171 seconds
Sum = 50005000 required 0.0019162 seconds
Sum = 50005000 required 0.0019360 seconds
```

- Salida: para n = 100.000 enteros, T. Medio $\simeq$ 0.019 segundos

```
Sum = 5000050000 required 0.0199420 seconds
Sum = 5000050000 required 0.0180972 seconds
Sum = 5000050000 required 0.0194821 seconds
Sum = 5000050000 required 0.0178988 seconds
Sum = 5000050000 required 0.0188949 seconds
```

- Salida: $n = 1.000.000$ enteros, T. Medio $\simeq 0.19$ segundos

```
Sum = 500000500000 required 0.1948988 seconds
Sum = 500000500000 required 0.1850290 seconds
Sum = 500000500000 required 0.1809771 seconds
Sum = 500000500000 required 0.1729250 seconds
Sum = 500000500000 required 0.1646299 seconds
```

- CONCLUSIÓN: **El tiempo requerido aumenta conforme aumenta** n. Es decir, el tiempo depende de n. Si multiplicamos n por 10, el tiempo también se multiplica por 10.

Ejemplo 2. Estudio Empírico de un Algoritmo

- Si consideramos otro algoritmo ***Suma2(n)***: que obtiene la suma los enteros de 1 a n aplicando directamente la fórmula

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

```
int Suma2(n)
        s ← 0
        s ← n*(n+1)/2
        Devolver s
```

```
SumaEnteros()
        n ←LeerEntero()

        start ← currentTime()
        Sum ← Suma2(n)
        end ← currentTime()

        tiempoEmpleado ← (end − start)
        Print Sum
        Print  tiempoEmpleado
```

- Si lo ejecutamos 10 veces, estos serían los **tiempos medios** para n= 10.000, 100.000, 1.000.000, 10.000.000, y 100.000.000 :

```
Sum = 50005000 required 0.00000095 seconds
```

```
Sum = 5000050000 required 0.00000191 seconds
```

```
Sum = 500000500000 required 0.00000095 seconds
```

```
Sum = 50000005000000 required 0.00000095 seconds
```

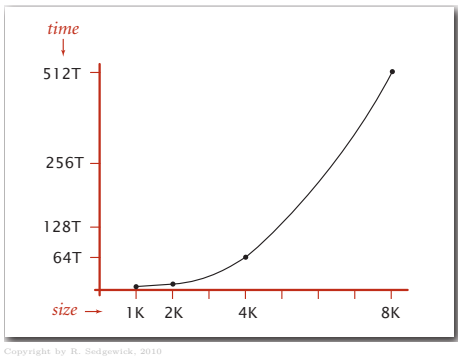
```
Sum = 5000000050000000 required 0.00000119 seconds
```

- CONCLUSIÓN: **Los tiempos son similares para todos los** n. Es decir, el tiempo NO depende de n.

Gráficas en el Estudio Empírico

A partir de los diferentes tamaños de entrada y de los correspondientes tiempos de ejecución que se han obtenido, podemos representar gráficamente los resultados de este modo:

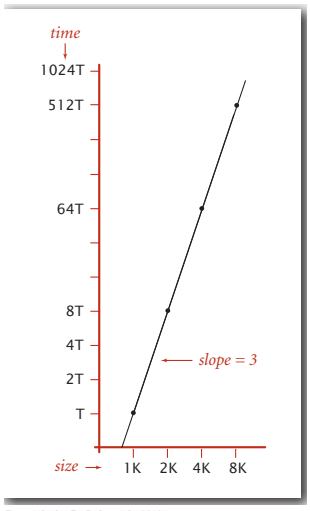
eje X: **tamaño de la entrada**(n) — eje Y: **tiempo de ejecución**



- La gráfica nos muestra la **función** que modeliza cómo crece el tiempo de ejecución en función del tamaño de la entrada n y nos permite obtener una expresión para $T(n)$.

A partir de la gráfica anterior, si tomamos **logaritmos** en ambos ejes, obtenemos esta recta:

eje X: log**(tamaño de la entrada)** — eje Y: log**(tiempo de ejecución)**



- Si $T(n)$ es de la forma: $T(n) = n^k$, entonces la pendiente de esta recta vale k
- En este ejemplo $k = 3$. Luego el tiempo de ejecución crece como el **cubo** del tamaño de la entrada: $T(n) = cn^3$
- c: *machine-dependent constant factor*

- Los tiempos de ejecución de un mismo algoritmo en diferentes ordenadores **deberían dar la misma función a diferencia de un factor constante**.

- Ese factor es una **constante asociada a la implementación** que llamaremos c.

Limitaciones del Estudio Empírico

- Es necesario implementar el agoritmo: puede ser difícil
- Los resultados no reflejan el comportamiento del algoritmo con OTRAS entradas nos probadas.
- Para comparar dos algoritmos debemos usar el mismo hardware y software.

- Necesitamos un modo mejor para caracterizar los algoritmos respecto al tiempo de ejecución. El estudio experimental computa el tiempo de ejecución actual. No proporciona una medida útil porque depende de la máquina, compilador, lenguaje de programación, etc.
- Deberíamos buscar una **modelización que sea independiente del programa o computador utilizado**, lo que nos permitiría comparar los algoritmos independientemente de la implementación concreta.

3.1-23

3 Estudio Teórico

¿Cómo realizamos el estudio estudio teórico?

- Usamos pseudocódigo del algoritmo en vez de implementarlo con un programa.
- Hay que identificar las **operaciones elementales** del algoritmo
- Tenemos que estimar **cuántas operaciones elementales se ejecutan** y cómo cambia este valor a medida que aumenta n, es decir:

  buscamos una función $f(n)$ que "describa" cómo se comporta el algoritmo a medida que crece el tamaño del problema.

3.1-24

¿Qué es una operación elemental?

Operación elemental
Su tiempo de ejecución se puede acotar superiormente por una **constante** ya que sólo depende de la implementación.

Es decir, siempre tardan igual y no dependen del valor de n.

- ¿Qué operaciones son elementales en un algoritmo?:
  - Op.: suma, resta, multiplicación, división, módulo, booleana,
  - comparaciones, **asignaciones**, evaluar una expresión lógica o aritmética, indexar un array
  - asignar valores a una matriz, vector,..
  - Llamada a un método, retorno de un método.
- **No** nos interesa el tiempo exacto de una operación elemental
  ➜ le asignaremos un **coste unitario**

3.1-25

¿Y las operaciones NO elementales?

- *Operaciones que **NO** son elementales*:
  - evaluación de un factorial,
  - comprobar divisibilidad de dos números,
  - comprobar primalidad, etc
- Operaciones elementales en las que se produce un desbordamiento (overflow) en los datos:

  ***Ejemplo**: Suma los enteros de 1 a n*

```
int Suma(n)
        sum ← 0
        Para i ← 1 hasta n hacer
                sum ← sum + i   ⇐
        devolver sum
```

- Nota: La clase `BigInteger` de Java representa enteros que no caben en un tipo de dato `int`. Se almacena cada dígito del entero en un posición de un array. Las operaciones de suma, resta, etc, con este tipo de dato NO se consideran elementales. Y tampoco las de `BigDecimal`.

http://www.tutorialspoint.com/java/math/java_math_biginteger.htm

Ejemplo: Calculo de $T(n)$ contando operaciones elementales

- Examinamos el pseudocódigo del algoritmo, y determinamos el **número de operaciones elementales** que ejecuta **como función del tamaño de la entrada**.

Obtener el elemento de valor máximo de un array

	Operaciones
funcion MaxArray(A,n)	
$maxActual \leftarrow A[0]$	2
Para $i \leftarrow 1$ hasta $n-1$ hacer	n
IF $A[i] > maxActual$	$2(n-1)$
$maxActual \leftarrow A[i]$	$2(n-1)$
Incrementar i	$2(n-1)$
devolver $maxActual$	1
	$7(n-1)$

- El algoritmo MaxArray ejecuta en total $7(n-1)$ operaciones elementales en el **peor caso**
- Si definimos:
  - a : tiempo que gasta la operación elemental más rápida
  - b : tiempo que gasta la operación elemental más lenta
- Entonces:

$$a7(n-1) \leq T(n) \leq b7(n-1)$$

- Por lo que podemos decir que:

  el tiempo de ejecución $T(n)$ está acotado por dos funciones lineales
- y por tanto:
  el algoritmo MaxArray tiene una tasa de crecimiento lineal ya que $T(n) \leq cte * n$

Tasa de crecimiento del tiempo de ejecución

- *¿Qué ocurre con $T(n)$ si cambiamos el entorno hardware/software?*

  - $T(n)$ se ve afectado por un **factor constante** , pero
  - No se altera la tasa de crecimiento de $T(n)$

- La **tasa de crecimiento lineal** del tiempo de ejecución $T(n)$ es una propiedad intrínseca del algoritmo MaxArray.

3.1-28

- **El estudio teórico nos permite comparar algoritmos independientemente del entorno hardware/software**
- **Tiene en cuenta TODAS las posibles entradas**

- El siguiente paso en el estudio teórico es estudiar el comportamiento del algoritmo para **valores muy grandes de** n:
  Notación asintótica.

3.1-29

4 Análisis Asintótico

Análisis Asintótico

- **El objetivo es mostrar cómo cambia el tiempo de ejecución del algoritmo a medida que el tamaño n del problema se va haciendo muy grande.**

Ejemplo:
Supongamos un algoritmo que suma n números enteros cuyo tiempo de ejecución es $T(n) = 1 + n$.

- Para sumar $n = 100$ enteros necesita 101 pasos, para sumar $n = 1$ millón de enteros, necesita 1000001 pasos
- **A medida que n se hace grande, la constante 1 se va haciendo menos significativa para el resultado final**.
  Es decir, que para valores grandes de n la aproximación será igual de precisa si no consideramos la cte 1.
- Entonces, si buscamos una aproximación para $T(n)$ podemos "saltar" la constante 1 y decir que
  el algoritmo es de un **orden de magnitud de** n, o que $T(n) = n$.

3.1-30

Ejemplo: Término Dominante

Ejemplo:
Supongamos otro algoritmo cuyo tiempo de ejecución es
$T(n) = 5n^2 + 27n + 1005$.

- Para valores pequeños de n por ejemplo $n = 2$, el término '1005' parece ser la parte dominante de la función.
- Sin embargo, a medida que n se hace grande, el término $5n^2$ se convierte en el término más importante.
- De hecho cuando n es realmente grande, los otros dos términos tienen un papel insignificante a la hora de determinar el resultado final, incluida la contante '5'.
- Por tanto, podemos decir que la función $T(n)$ tiene un **orden de magnitud de** $f(n) = n^2$.

3.1-31

Orden de Magnitud

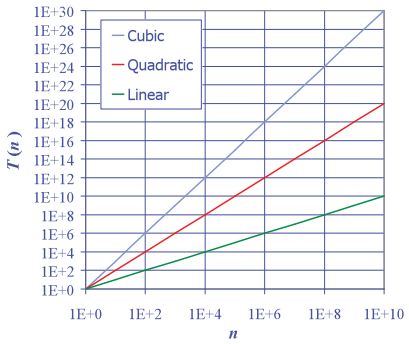
- Podemos ver que *determinar el número exacto de operaciones de un algoritmo no es tan importante como identificar la parte más dominante de la función* $T(n)$.
- Es decir, a medida que el tamaño del problema se hace más grande, una parte de $T(n)$ tiende a dominar el resto.
- Este **término dominante** es el que usaremos como referencia, y lo notaremos $f(n)$.
- Entonces cuando decimos que $T(n)$ tiene un orden de magnitud $f(n)$ se está indicando **qué parte** de $T(n)$ **crece más rápido cuando** n **se hace grande**.

3.1-32

Ejemplos

- Funciones de orden de magnitud n:
  - $T(n) = 1000n + 10^9$.
  - $T(n) = 3n + \log(n)$.
- Funciones de orden de magnitud n^2
  - $T(n) = 4n^2 + 1000n$.
  - $T(n) = 0.05n^2 + 10^6 n$.
- Funciones de orden de magnitud n^3
  - $T(n) = 3n^3 + 50n^2 + 150n$.
  - $T(n) = 10^{-4}n^3 + \log(n)$.

- Vemos como cuando n se hace grande el término de (n^3) domina:



3.1-33

¿Qué es la Notación Asintótica?

- Vemos entonces que al medir las tasas de crecimiento cuando n va tomando valores muy grandes ignoramos:
  - Todos los términos de $T(n)$ excepto el mayor
  - Cualquier constante que sume o multiplique
- Es decir que:
  - $T(n) = 3n^5 + 4n^2 + 525$ se comporta asintóticamente como n^5, y
  - $T(n) = 10000n^5 + n^4 + 3n^2 + 2n + 100$ también se comporta asintóticamente como n^5

¿Cómo formalizamos esto?

- Vamos a ver cómo "agrupar" funciones con el mismo orden de magnitud mediante **cotas asintóticas** $f(n)$, que pueden ser superiores o inferiores.

3.1-34

Cota Superior. Notación $\mathcal{O}$

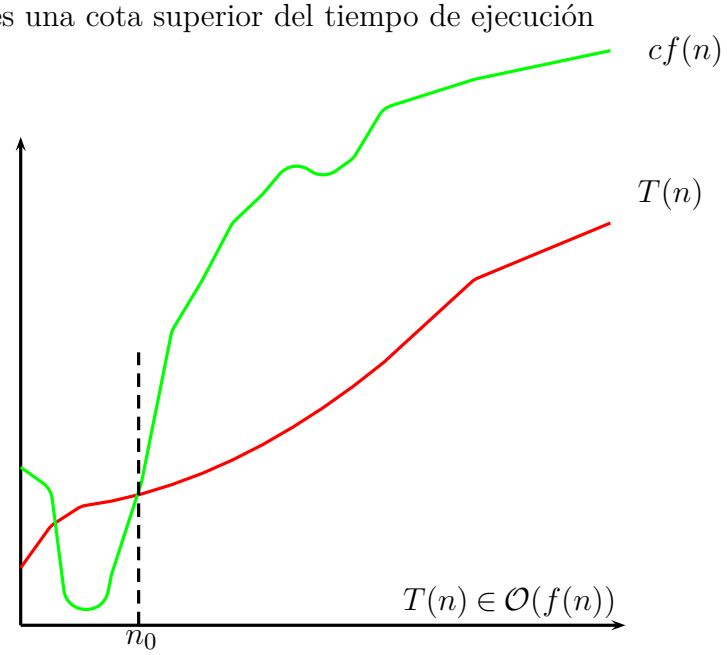
Notación $\mathcal{O}$
Un algoritmo para resolver un problema requiere un tiempo de ejecución del **orden de** $f(n)$ si

- a partir de un cierto tamaño de la entrada, que llamaremos n_0, existe una constante positiva c y una implementación del algoritmo capaz de resolver **todos los casos de tamaño** $n \geq n_0$
- en un tiempo **no superior** a $c.f(n)$

3.1-35

Notación Big-$\mathcal{O}$: Cota Superior

$f(n)$ es una cota superior del tiempo de ejecución



3.1-36

Significa que hay un punto, un valor de n al que llamaremos n_0, o umbral, a partir del cual es seguro que, para algún valor constante c (positivo):

- el tiempo de ejecución $T(n)$ *es* $\leq cf(n)$,
- y decimos que $T(n)$ está en el orden $\mathcal{O}(f(n))$

- La constante c está **asociada a la implementación** del algoritmo
  ➜ Cambia entre diferentes implementaciones del mismo algoritmo

3.1-37

Ejemplos:

- Para los algoritmos de los ejemplos anteriores el orden de $T(n)$ sería:
  - $T(n) = 1 + n$, entonces $f(n) = n$ y por tanto $T(n)$ es $\mathcal{O}(n)$.
  - $T(n) = 5n^2 + 27n + 1005$, $f(n) = n^2$ y por tanto $T(n)$ es $\mathcal{O}(n^2)$

3.1-38

Comparamos Ordenes de Complejidad

- Ordenes de complejidad para un algoritmo que resuelve problemas de **tamaño** n:

Constante	c	$\mathcal{O}(1)$
Logarítmico	$c \log n$	$\mathcal{O}(\log n)$
Lineal	cn	$\mathcal{O}(n)$
Cuadrático	cn^2	$\mathcal{O}(n^2)$
Cúbico	cn^3	$\mathcal{O}(n^3)$
Polinómico	cn^k	$\mathcal{O}(n^k)$
Exponencial	c^n	$\mathcal{O}(k^n)$

- c es la constante asociada a la implementación, $T(n) \leq c.f(n)$
- Decimos que un algoritmo tiene un coste temporal lineal cuando es $\mathcal{O}(n)$, o cúbico cuando es $\mathcal{O}(n^3)$. De hecho decimos directamente que el algoritmo es lineal, cuadrático, cúbico, exponencial, etc..

3.1-39

¿Qué tipo de algoritmos tienen esos Ordenes de Complejidad?

T(n)	Tipo de Algoritmo	Descripción y Ejemplos de algoritmos de este tipo
1	**Constante**	No dependen del tamaño de la entrada. Ej: obtener el primer elemento de una lista
$\log n$	**Logarítmico**	Descomponen un problema grande en problemas más pequeños dividiendo el tamaño n sucesivamente. Ej: Búsqueda Binaria
n	**Lineal**	Acceden a cada uno de los n elementos. Ej: Imprimir los n elementos de una lista
$n \log n$	**Lineal logarítmico**	Descomponen problemas grandes en problemas más pequeños, los resuelven de forma independiente, y combinan estas soluciones. Ej: Mergesort
n^2	**Cuadrático**	Acceden a todos los **pares** de elementos: Ej: Buscar dos puntos más cercanos en un plano. Algoritmos de ordenación (burbuja, inserción, selección)
n^3	**Cúbico**	Varias posibilidades. Un ejemplo son algoritmos que acceden a todas las tripletas de elementos
n^K	**Polinómico**	En general, cuando se accede a todos los **subconjuntos de k elementos** que se pueden formar a partir de un conjunto de n elementos.
2^n	**Exponencial**	Normalmente en problemas en los que se busca por fuerza-bruta **subconjuntos** de elementos que satisfacen una condición. $\Rightarrow$ hay que comprobar la condición con todos los subconjuntos de tamaño 2, los de tamaño 3, etc...
$n!$	**Factorial**	Normalmente en problemas en los que se busca por fuerza-bruta sobre todas las **permutaciones** de n elementos que satisfacen una condición. Ej: Viajante de Comercio

3.1-40

¿Qué implica a efectos prácticos que un algoritmo tenga un coste lineal, logarítmico, exponencial, ...?

- Un algoritmo **constante**, ejecuta un número de instrucciones que es independiente del tamaño del problema n. El tiempo está acotado por una cte.
- Un alg. **logarítmico** crece muy lentamente a medida que crece n.
  Por ejemplo: puede resolver un problema de tamaño n=10 en 1μs, uno de n=100 en 2μs y otro de n=1000 en 3μs.
- Un alg. **cuadrático** deja de ser útil para valores grandes, pues pasar a resolver un problema el **doble** de grande implica que tarde **4** veces más.
- Un alg. **cúbico** solo es útil para problemas pequeños, pues pasar a resolver un problema el **doble** de grande implica que tarde **8** veces más.
- Un alg. **exponencial** (ej $O(2^n)$) raramente es útil. Si resuelve un problema de tamaño n=10 en un determinado tiempo, para resolver otro el **doble** de grande necesita unas **1000** veces más tiempo.

3.1-41

Ejemplo: Algoritmo exponencial

- Un algoritmo de tratamiento de textos necesita **realizar 10^n operaciones**, siendo n el número de palabras del texto tratado
- En particular, si el texto a tratar tuviera **20 palabras**, el número de operaciones a realizar sería 10^{20}.
- Si nuestro computador ejecuta 300 millones de esas operaciones por segundo:

  $\Rightarrow$ **necesita 10.000 años** para resolver este problema.
- Si tuviéramos un computador 100 millones de veces más potente, podríamos resolver el problema del texto de **20 palabras** en algo menos de una hora.
- Pero, si el texto tuviera **30 palabras**, todavía tardaríamos **un millón de años** en resolver el problema.

3.1-42

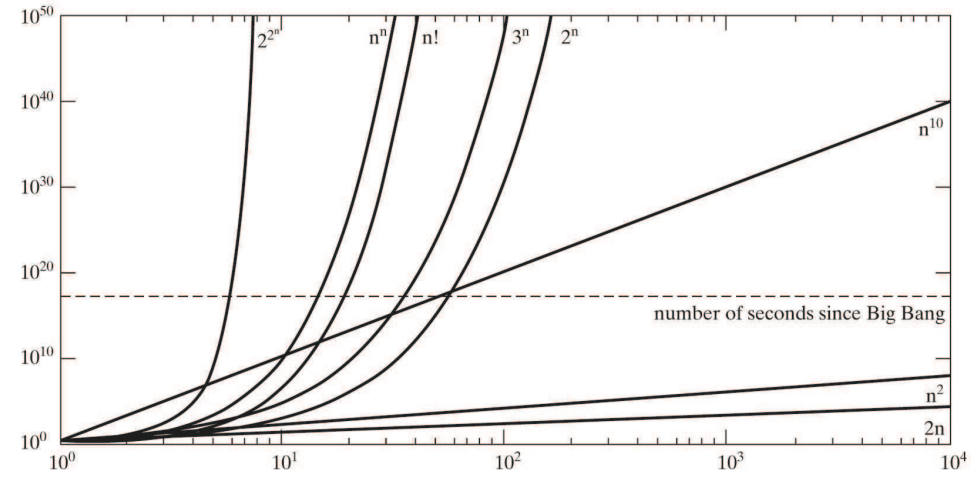
Resumen: Cómo varía el tiempo de ejecución a medida que duplicamos n

T(n)	Tipo	Si n se duplica, entonces el Tiempo de ejecución	Ejemplos
1	**Constantes**	Acotado por una constante	No dependen del tamaño de la entrada. Operaciones aritméticas. Asignaciones.
$\log n$	**Logarítmico**	Se incrementa por una cte.	Búsqueda Binaria
n	**Lineal**	Se multiplica por **2**.	Iterar sobre n elementos. Buscar elemento máximo en un array
$n \log n$	**Lineal logar.**	Un poco más del doble.	Quicksort, Mergesort, FFT
n^2	**Cuadrático**	Se multiplica por **4**.	Ordenación por Inserción, Ordenación por Selección, Ordenación Burbuja
n^3	**Cúbico**	Se multiplica por **8**.	Uso práctico en problemas pequeños.
2^n	**Exponencial**	Se eleva al **cuadrado**!	Todos los subconjuntos posibles con n elementos. Torres de Hanoi con n discos.
$n!$	**Factorial**	Si n **se incrementa en una unidad**, el t. ejec. se incrementa por un factor de n	Todas las permutaciones de n elementos. TSP

En la tercera columna vemos cómo se incrementa el tiempo de ejecución a medida que se duplica el tamaño de la entrada n, dependiendo de la tasa de crecimiento del algoritmo

3.1-43

Comparación de Tasas de Crecimiento



Comparamos tasas de crecimiento en segundos

En la tabla siguiente podemos comparar el tiempo de ejecución de diferentes algoritmos para tamaños de entrada n crecientes, en un mismo procesador que realiza **1 millón de operaciones por segundo**:

Table 2.1 The running times (rounded up) of different algorithms on inputs of increasing size, for a processor performing a million high-level instructions per second. In cases where the running time exceeds 10^{25} years, we simply record the algorithm as taking a very long time.

	n	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n = 10$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	4 sec
$n = 30$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	18 min	10^{25} years
$n = 50$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	11 min	36 years	very long
$n = 100$	< 1 sec	< 1 sec	< 1 sec	1 sec	12,892 years	10^{17} years	very long
$n = 1{,}000$	< 1 sec	< 1 sec	1 sec	18 min	very long	very long	very long
$n = 10{,}000$	< 1 sec	< 1 sec	2 min	12 days	very long	very long	very long
$n = 100{,}000$	< 1 sec	2 sec	3 hours	32 years	very long	very long	very long
$n = 1{,}000{,}000$	1 sec	20 sec	12 days	31,710 years	very long	very long	very long

- Si medimos el tiempo de ejecución en **segundos**, la conversión es:

10^2 **seg**	10^4	10^5	10^6	10^7	10^8	10^9	10^{10}	10^{11}
1.7 min	2.8 h	1.1 días	1.6 sem	3.8 mes	3.1 años	3.1 dec	3.1 **siglos**	nunca

- $2^{10} \approx 10^3$
- 2^{32} segundos ≈ 124 años

¿No sería más sencillo usar ordenadores más rápidos?

¿Puedo resolver un problema usando un ordenador más rápido?

- La respuesta es NO si se trata de algoritmos exponenciales, o factoriales.
- Un ordenador que realizase UN BILLÓN (10^{12}) de operaciones por segundo,
  ¿cuánto tardaría en resolver problemas de estos tamaños?

$T(n)$	n=20	n=40	n=50	n=60	n=70	n=80
n^5	$3.2\mu s$	$102\mu s$	$313\mu s$	$778\mu s$	1.68ms	3.28ms
2^n	1.05ms	1.1s	18.8min	13.3días	37.4años	383sig
3^n	3.5ms	4.7meses	227siglos	1.3x10^7sig	7.9x10^{11}sig	x10^{16}sig

- En las dos últimas filas tenemos tiempos de ejecución de algoritmos de orden exponencial. A partir de un determinado tamaño tardarían "demasiado tiempo" en terminar.

3.1-46

Ejemplo: Dos algoritmos para Cálculo de un determinante

Calcular determinante de una matriz M de tamaño nxn

- Método clásico: definición recursiva de determinante ➜ $\mathcal{O}(n!)$

$$det(M) = \sum_{j=1}^{n} a_{1j} det(M[1,j])$$

- Método Eliminación de Gauss-Jordan ➜ $\mathcal{O}(n^3)$
- Según la tabla de la imagen anterior, si ejecutásemos ambos algoritmos en ese ordenador, estos son los tiempos que tardarían en calcular un determinante de tamaño $n \times n$:

nxn	**Clásico**	**Gauss-Jordan**
10x10	4 seg	<1 seg
30x30	10^{25} años	<1 seg
100x100	??	1 seg

3.1-47

¿Cómo usamos la notación asintótica?

- Recordemos que nuestro objetivo es comparar algoritmos que resuelven el mismo problema para elegir el más eficiente.
- Mediante el análisis teórico del algoritmo estimamos el número de operaciones que realiza y obtenemos una expresión para $T(n)$.
- La notación asintótica nos indica **cómo se comporta el algoritmo para valores muy grandes de** n, esto es, en el límite.
- Por ello hay que tener cuidado con los cálculos que realizamos cuando n es un valor **pequeño** ya que podríamos tener resultados engañosos.
  Esto es debido a que en la notación asintótica no consideramos ni las constantes ni los otros términos de menor orden, que para valores pequeños de n sí podrían ser relevantes.
- Como por ejemplo:
  - $T(n) = n^2 + 1000n$
  - $T(n) = 0.0001n^5 + 10^6 n^2$
  - $T(n) = 10^{-4} n^5 + 10^5 n^3$

  - Y si comparamos estas dos funciones:
    $T(n) = 8n \log n$ y
    $T(n) = 0.01n^2$
    asintóticamente la primera que está en $\mathcal{O}(n \log n)$, es mejor que la segunda, $\mathcal{O}(n^2)$, pero solo se obtienen mejores valores cuando n es mayor que 10710.

- Es decir que **la notación asintótica es muy útil para comparar algoritmos para valores muy grandes de n pero hay que ser cuidadoso con las estimaciones para valores pequeños de n.**

3.1-48

- Para problemas pequeños, es indiferente el método que usemos porque un ordenador rápido lo ejecuta al instante

- A medida que el tamaño del problema crece, los tiempos aumentan y la elección del método condiciona que se pueda ejecutar o no
  - Los algoritmos que son "mejores asintóticamente" son buenos para entradas grandes, pero no para entradas pequeñas.

3.1-49

5 Resumen

En esta primera parte hemos visto:

1. Necesitamos poder elegir el **mejor algoritmo** que resuelva un determinado problema:

⇒ Estudiamos la **eficiencia** de cada algoritmo: seleccionamos el más eficiente.

2. ¿Cómo medimos la eficiencia? Estimamos los **recursos que consume**: tiempo de ejecución y memoria en el **peor caso**.
3. Para ello hay que tener en cuenta el tamaño del problema n y obtener una función $T(n)$ que modelice el crecimiento del tiempo de ejecución a medida que aumenta el valor de n.
4. Esta función se obtiene, teóricamente, a partir del número de operaciones elementales del algoritmo.
5. La **notación asintótica** estudia cómo se comporta el algoritmo para **valores grandes de la entrada** n. Identifica el término dominante en la definición de $T(n)$.

3.1-50

En el siguiente apartado de este tema...

1. Estudiaremos las **propiedades de las notaciones asintóticas**, y la relación entre ellas.

2. También estudiaremos **cómo obtener las cotas del tiempo de ejecución a partir del análisis de las estructuras de control** del algoritmo.

3.1-51

A Apéndice

A.1 Eficiencia en Memoria

Typical Memory Requeriments for Java Data Types

- Referencia: ocupa 16 bytes

Primitive types

Tipo	bytes
`boolean`	1
`byte`	1
`char`	2
`int`	4
`float`	4
`long`	8
`double`	8

Arrays

Tipo	bytes
`boolean[]`	$16+N$
`byte[]`	$16+N$
`char[]`	$16+2N$
`int[]`	$16+4N$
`double[]`	$16+8N$
`char[][]`	$2N^2+20N+16$
`int[][]`	$4N^2+20N+16$
`double[][]`	$8N^2+20N+16$

Ejemplo:
¿Cuánta memoria necesita un array $N \times N$ de `double`? $\approx 8N^2$ bytes

Ejemplo: Algoritmo Java
¿Cuánta memoria necesita este programa en función de N?

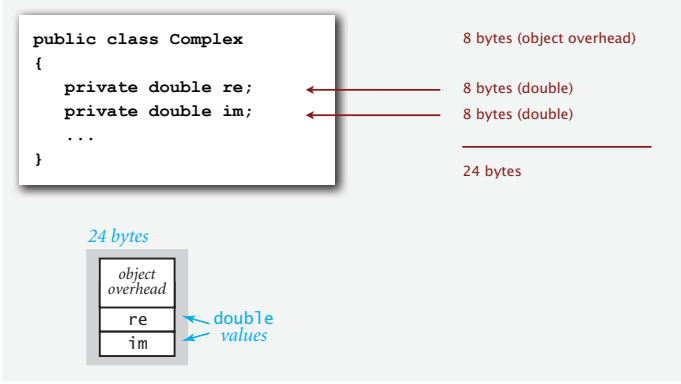
```
public class RandomWalk {
   public static void main(String[] args) {
      int N = Integer.parseInt(args[0]);
      int[][] count = new int[N][N];
      int x = N/2;
      int y = N/2;

      for (int i = 0; i < N; i++)  {
         // no new variable declared in loop
         ...
         count[x][y]++;
      }
   }
}
```


http://introcs.cs.princeton.edu

Typical Memory Requeriments for OBJECTS in Java

- Instanciar un objeto: 8 bytes + tamaño de sus campos.

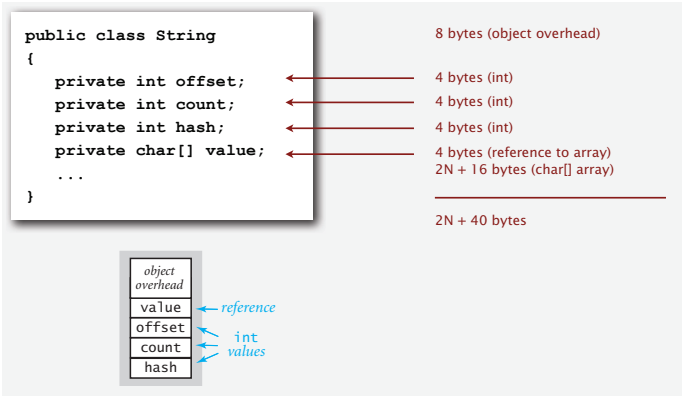
Ejemplo: objeto Complex
Una instancia del objeto Complex ocupa 24 bytes.



http://introcs.cs.princeton.edu

Ejemplo: objeto ***String***

Una instancia de `String` de longitud N consume $\approx 2N$ bytes



http://introcs.cs.princeton.edu

References

[1] [Básica] Weiss, M.A. *Estructuras de Datos en Java. 4-ed.* (capítulo 5: 5.1 - 5.3) Pearson

[2] [Básica] Brassard G., Bratley P. *Fundamentos de Algoritmia.* (capítulo 2) Prentice Hall, 1997 /2004

[3] *Algorithm Analysis.* http://cs.lmu.edu/~ray/notes/alganalysis/

[4] *Analysis of Algorithms* . http://introcs.cs.princeton.edu/java/41analysis/