

Estructuras de Datos y Algoritmos

Tema 1: Análisis de Algoritmos

Departamento de Informática
Universidad de Valladolid

Curso 2011-12

Grado en Ingeniería Informática
Grado en Ingeniería Informática de Sistemas





1. MEDIDA DE ALGORITMOS



Análisis de Algoritmos

- Estudio teórico de la **eficiencia** de un algoritmo
- **Eficiencia**: Medida del coste en el uso de **recursos** que necesita el algoritmo para llevar a cabo su tarea.
- **Recursos** más importantes:
 - **Tiempo** de ejecución ← Valor por defecto
 - **Espacio** de almacenamiento
- La medida se denomina **complejidad** del algoritmo.
- Otros aspectos (contemplados en otras asignaturas):
 - Funcionalidad
 - Robustez
 - Modularidad
 - Mantenibilidad
 - Facilidad de uso
 - Extensibilidad
 - Corrección
 - ...



Búsqueda Secuencial

```
type
  TDato = ...;
  // Array dinámico (índices 0-based)
  TVector = array of TDato;

{ Devuelve el índice donde se encuentra la primera
  aparición de x en V o -1 si no existe }
function Busqueda(const V: TVector; x: TDato): integer;
var
  n,i : integer;
begin
  i := 0; n := length(V); { N° de elementos de V }
  while (i < n) and (V[i] <> x) do i := i+1;
  if i < n then Busqueda := i else Busqueda := -1
end;
```

Problemas con la medida (tpo)



- **Dependencia con el tamaño de la entrada**: No se tarda lo mismo en buscar en un vector de 10 elementos que buscar en uno de 1.000.000
- **Dependencia de valores de la entrada**: Aunque fijemos el tamaño del vector, no se tarda lo mismo en buscar un valor que está en la primera posición que otro que no esté en el vector.
- **Dependencia del procesador**: Aunque fijemos el tamaño y los valores concretos del vector y el valor buscado, el algoritmo tardará tiempos distintos en ordenadores diferentes.



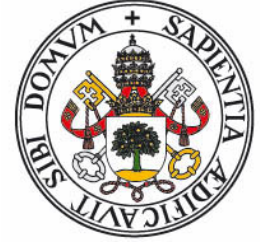
Dependencia del procesador

- **Solución:** No medir tiempo (segundos, por ejemplo), sino el número de **operaciones elementales** ejecutadas.
- **Operación Elemental:** Toda operación que tarda un **tiempo constante** en cualquier procesador razonable.
 - Tipicamente se consideran elementales las asignaciones, operaciones aritméticas y relacionales con tipos de datos de tamaño fijo, acceso a arrays.
 - En general se cuenta **sólo un tipo** de operación concreta (la más relevante para la eficiencia del algoritmo)
 - Conociendo el número de operaciones y lo que tarda cada una en procesador concreto, se puede hallar el tiempo para ese procesador.
 - Medida **independiente** del procesador.



Tamaño de la entrada

- **Definición estricta:** El mínimo **número de bits** necesario para representar la parte no precalculable de la **entrada** del algoritmo.
- **Definición útil:** Uno o más valores relacionados con los datos de entrada que sirven de **parámetros** para expresar las **funciones** que miden el uso de recursos del algoritmo.
 - En el caso de algoritmos que trabajan sobre colecciones de datos, suele ser **el número de datos** que contienen.
 - Para algoritmos de cálculo con enteros de tamaño arbitrario, se suele usar el número de bits de esos enteros (se tratan como arrays de bits).
 - El tamaño de la entrada puede indicarse por **más de un valor** (siempre enteros positivos).



Dependencia con el tamaño

- **Solución:** Expresar la complejidad no mediante un valor sino por una **función** cuyo parámetro(s) es el tamaño de la entrada.
 - El tamaño de la entrada, si es un único valor, se suele denominar n .
 - La complejidad temporal se denominará mediante la función $T(n)$, y la espacial por $E(n)$.
- De esa función nos interesa, más que su forma concreta, su **ritmo de crecimiento**.
 - Nos da la idea de como *escala* un algoritmo: Cómo crece su complejidad cuando aumenta el tamaño de la entrada.



Dependencia con los valores

- **Solución:** Dividir el análisis en **casos**.
 - Analizar subconjuntos de las entradas cuya complejidad **es la misma** para todas las entradas de ese subconjunto. (análisis de **peor** y **mejor** caso)
 - Calcular un **promedio**, dada una distribución estadística de las entradas. Típicamente se supone que todas las posibles entradas son equiprobables. (análisis de **caso promedio** y de **tiempo amortizado**)
- **Nota:** Estos análisis trabajan sobre entradas de un **tamaño fijo** (aunque no especificado).
 - **Subnota:** Aunque en el análisis de tiempo amortizado realmente las entradas **varían** de tamaño, el ignorar este hecho no suele tener consecuencias adversas.



Mejor y Peor Caso

- **Análisis en el peor caso:** Calcula la complejidad del algoritmo para las entradas (del mismo tamaño) que **máximizan** la complejidad.
 - $T_{\text{peor}}(n) = \mathbf{m\acute{a}x}\{ T(n, \text{entrada}) \}$ para toda entrada de tamaño n .
 - Suele ser el caso más relevante (hay excepciones).
- **Análisis en el mejor caso:** Calcula la complejidad del algoritmo para las entradas (del mismo tamaño) que **minimizan** la complejidad.
 - $T_{\text{mejor}}(n) = \mathbf{min}\{ T(n, \text{entrada}) \}$
 - No suele ser un caso relevante.



Búsqueda Secuencial

```
type
  TDato = ...;
  // Array dinámico (índices 0-based)
  TVector = array of TDato;

{ Devuelve el índice donde se encuentra la primera
  aparición de x en V o -1 si no existe }
function Busqueda(const V: TVector; x: TDato): integer;
var
  n,i : integer;
begin
  i := 0; n := length(V); { N° de elementos de V }
  while (i < n) and (V[i] <> x) do i := i+1;
  if i < n then Busqueda := i else Busqueda := -1
end;
```



Búsqueda secuencial

- **Operación elemental:** Elegimos contar comparaciones en las que intervenga un elemento del vector.
- **Tamaño de la entrada:** Elegimos tomar como tamaño de entrada el número de elementos del vector.
- **Peor Caso:** Para vectores de tamaño n , las entradas que hacen que el algoritmo trabaje más son aquellas en que el valor buscado no se encuentra en el vector.

$$T_{\text{peor}}(n) = n \text{ comparaciones}$$

- **Mejor Caso:** Las entradas que hacen que el algoritmo trabaje menos son aquellas en que el valor buscado está en la primera posición del vector.

$$T_{\text{mejor}}(n) = 1 \text{ comparación}$$



Caso Promedio (1)

- Calcula el **promedio** de la complejidad del algoritmo para todas las entradas posibles (del mismo tamaño).
 - Supondremos que todas las entradas son equiprobables
 - Llamaremos s a una entrada cualquiera del algoritmo.
 - Llamamos S_n al conjunto de todas las posibles entradas de tamaño n .
 - $|S_n|$ denota el número de elementos de S_n

$$T_{med}(n) = \frac{\sum_{s \in S_n} T(n,s)}{|S_n|}$$



Caso Promedio (2)

- Suele ser mas sencillo **agrupar** las entradas en conjuntos de aquellas entradas para las que el algoritmo tarda lo mismo.
- Suele pasar que las entradas para las que el algoritmo da la misma respuesta tardan lo mismo
- Llamamos $S_{n,t}$ al subconjunto de S_n formado por las entradas que tardan un tiempo t .
- Llamamos R al conjunto de todos los tiempos posibles (valores enteros) que puede tardar el algoritmo con entradas de tamaño n .

$$T_{med}(n) = \sum_{t \in R} \frac{|S_{n,t}|}{|S_n|} \cdot t$$

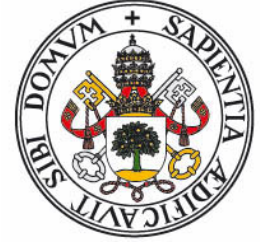
Probabilidad de que una entrada provoque t operaciones



Búsqueda secuencial

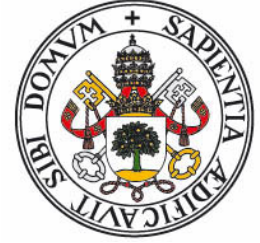
- Elegimos **restringir** el análisis a **búsquedas exitosas** (el valor buscado está en el vector).
- El conjunto R (número de posibles comparaciones que puede realizar el algoritmo) es $[1..n]$.
- El conjunto $S_{n,t}$ es el conjunto de todos los vectores posibles de tamaño n y de posibles valores de búsqueda para los que el algoritmo realiza t operaciones \rightarrow el valor buscado está en la posición $t-1$.
- Todos los subconjuntos $S_{n,t}$ tienen el mismo tamaño (es equiprobable el encontrar el valor en cualquier posición)

$$\frac{|S_{n,t}|}{|S_n|} = \frac{1}{n} \Rightarrow T_{med}(n) = \sum_{t \in [1..n]} t/n = \frac{n+1}{2}$$



Caso Promedio

- Da una idea mucho más ajustada de cómo se comporta el algoritmo para entradas cualesquiera, o para muchas ejecuciones **independientes** del mismo.
- Pero puede ser muy **difícil** de analizar.
- Sólo se indica para aquellos algoritmos en los que el caso promedio **difiere** significativamente del peor caso (ejemplo: quicksort).
- Cuando un algoritmo se ejecuta muchas veces, pero las ejecuciones no son independientes (se actualiza un conjunto de datos, por ejemplo) se suele usar el análisis de **tiempo amortizado**.



Tiempo amortizado (1)

- En vez de promediar sobre distintas entradas, se promedia sobre una **serie de ejecuciones** de un algoritmo.
 - Se utiliza cuando la entrada del algoritmo se modifica en cada ejecución (ejemplo: operaciones de inserción en un conjunto de datos).
 - Es importante cuando las entradas tienen sólo peor y mejor caso, sin casos intermedios..
 - .. y la ejecución de un peor caso garantiza que un determinado número de las siguientes ejecuciones pertenezcan al mejor caso.
 - Si se da esta situación, se promedia una serie de ejecuciones [peor caso + mejores casos garantizados].



Tiempo amortizado (2)

- Sea un algoritmo con las siguientes características:
 - La aplicación realiza una serie de ejecuciones, cada una de ellas tomando la entrada actualizada por la anterior.
 - Un peor y mejor caso $T_{peor}(n)$ y $T_{mejor}(n)$.
 - La ejecución del peor caso garantiza que las siguientes $f(n)$ ejecuciones serán del mejor caso.
 - El tiempo amortizado se calcula como:

$$T_{amort}(n) = \frac{T_{peor}(n) + f(n) \cdot T_{mejor}(n)}{1 + f(n)}$$

- **Nota:** En realidad, en un cálculo riguroso se debería sustituir $f(n) \cdot T_{mejor}(n)$ por un sumatorio que refleje el cambio de n en las subsiguientes operaciones.



Ejemplo: Inserción en vector

- Se dispone de un array *parcialmente* lleno (capacidad m , número de elementos n) y se diseñan dos estrategias de inserción:
 - En ambos casos, si $n < m$ (existe espacio libre), se inserta al final de los elementos existentes (una operación)
 - Si $n = m$ (vector lleno) se crea un nuevo vector más grande y se copian los n elementos del antiguo en él.
 - La primera estrategia crea un vector con 100 posiciones extra.
 - La segunda estrategia crea un vector con n posiciones extra (duplica la capacidad del antiguo).

Análisis



- **Peor caso:** El peor caso se da cuando se inserta en un vector lleno. En ambas estrategias, se deben copiar los n elementos existentes e insertar el antiguo, por lo que:

$$T_{\text{peor}}(n) = n+1$$

- **Mejor caso:** Cuando el vector no está lleno sólo se debe realizar una inserción:

$$T_{\text{mejor}}(n) = 1$$

- **Caso promedio:** No es aplicable, ya que los valores de las entradas no influyen en el tiempo, tan sólo si el vector está lleno o no, y a priori no se puede saber la probabilidad de esa situación para una ejecución aislada.



Análisis Tiempo Amortizado

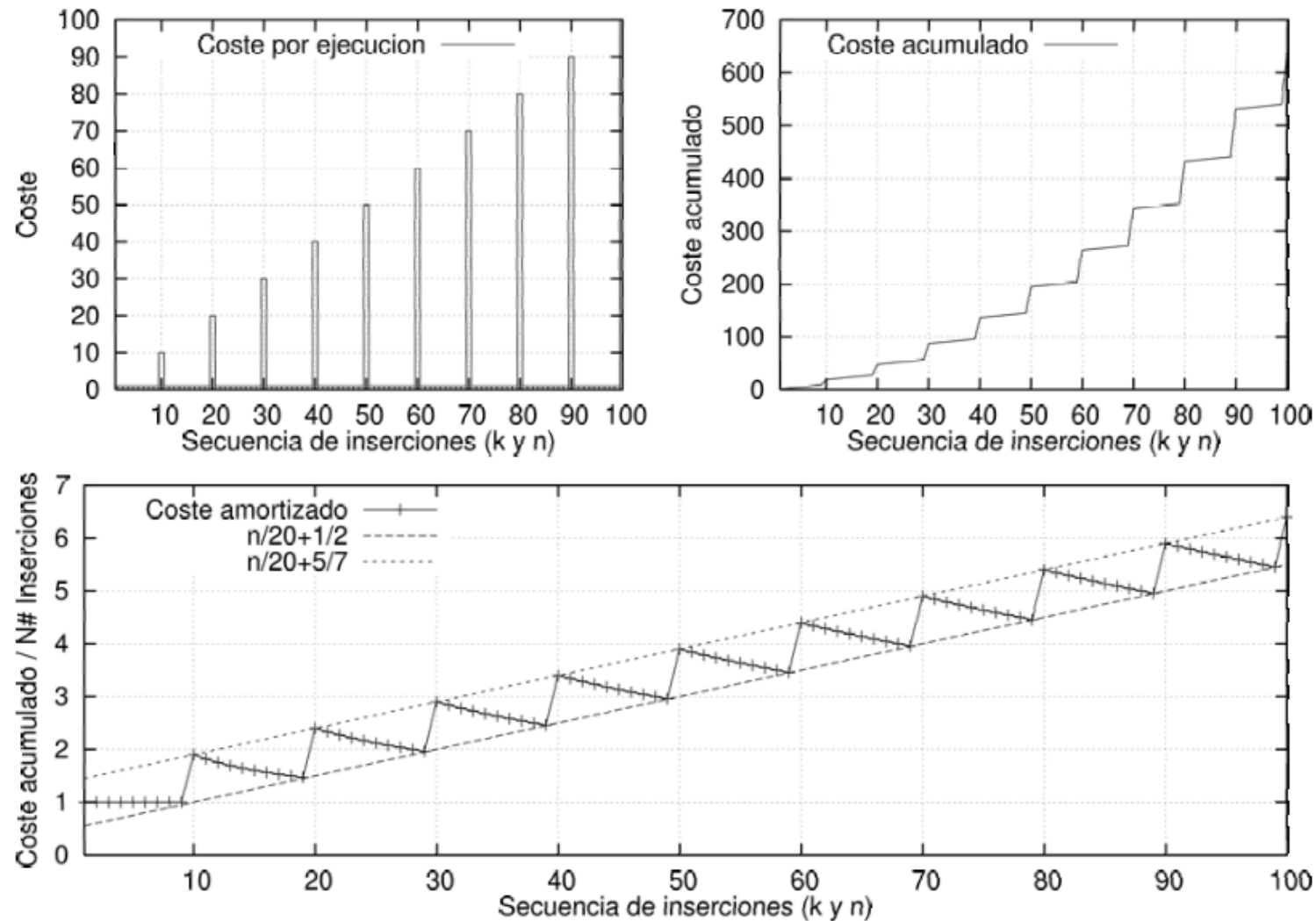
- **Primer escenario:** Si el algoritmo amplía el vector en 100 posiciones, esto garantiza que tras una inserción en un vector lleno las siguientes 99 inserciones caerán en el mejor caso:

$$T_{amort}(n) = \frac{(n + 1) + 99 \cdot 1}{1 + 99} = \frac{n}{100} + 1$$

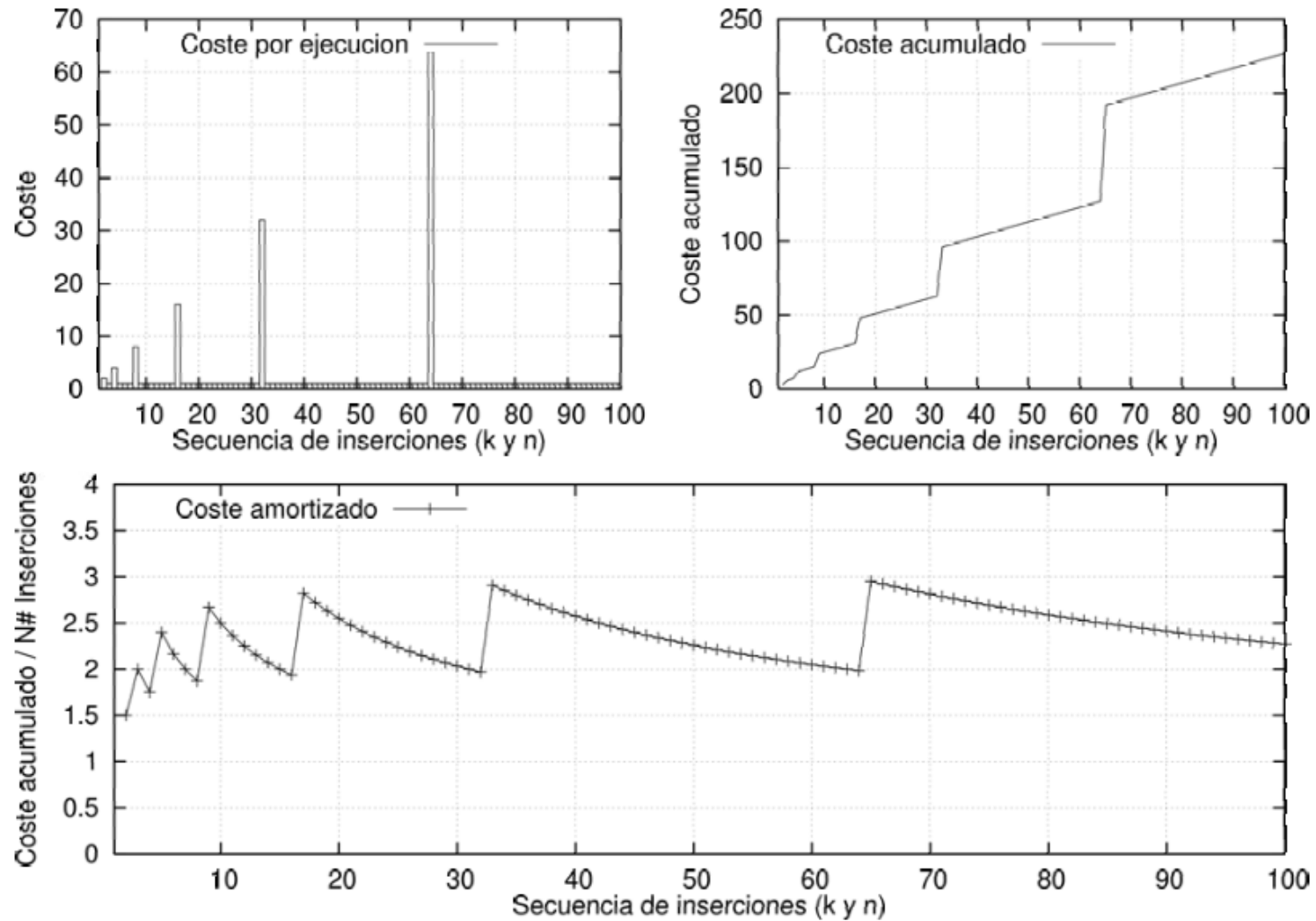
- **Segundo escenario:** Si el algoritmo duplica la capacidad del vector, esto garantiza que tras cada inserción en un vector lleno las siguientes $n-1$ inserciones caerán en el mejor caso:

$$T_{amort}(n) = \frac{(n + 1) + (n - 1) \cdot 1}{1 + (n - 1)} = 2$$

Primer escenario



Segundo escenario





Complejidad temporal

- El análisis de la complejidad temporal consiste en **contar** el número de operaciones elementales
 - El tiempo se acumula (a diferencia del espacio)
 - Una secuencia de operaciones se computa sumando los tiempos de los bloques que la forman.
 - Una bifurcación requiere un análisis por casos.
 - Las iteraciones requieren el calcular el número de repeticiones (puede no ser sencillo o requerir análisis por casos). También hay que analizar si el tiempo de cada repetición es constante o no:
 - **Si** → Multiplicar.
 - **No** → Sumatorios.

Fórmula para las iteraciones



$$\sum_{i=c_1}^{n \pm c_2} (i \pm c_3)^a = \frac{n^{a+1}}{a+1} + O(n^a)$$



Complejidad espacial

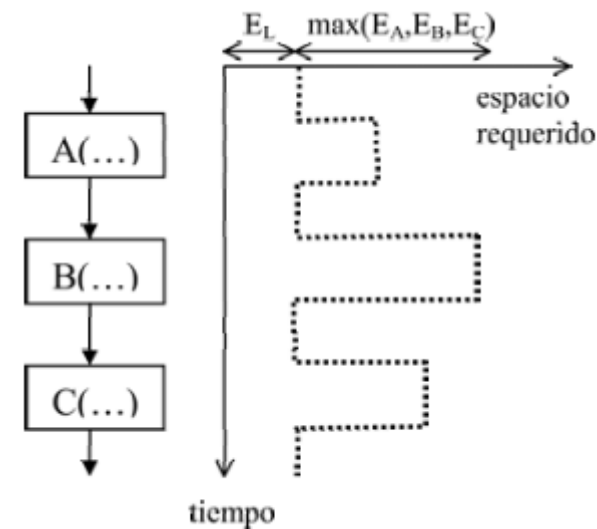
- El análisis de la complejidad espacial consiste en determinar el **máximo** espacio **extra** de almacenamiento que necesita el algoritmo.
 - No se tiene en cuenta el espacio que ocupan los datos de entrada (se supondrá siempre paso por variable).
 - Se cuenta el espacio necesario para las variables locales y los resultados.
 - También las operaciones de creación de objetos, variables dinámicas y ampliación de arrays (no suelen aparecer en los algoritmos que examinaremos)
 - Las **llamadas a subprogramas** necesitan el espacio que dicte el subprograma como necesario para su ejecución.



Llamadas a subprogramas

- Supongamos un programa que llama en secuencia a 3 subprogramas A, B y C que necesitan un espacio de almacenamiento de E_A , E_B y E_C , respectivamente.
 - Tras la ejecución de cada subprograma la memoria extra que necesita se **libera**.
 - Por lo tanto el espacio extra del algoritmo no es la suma de esos valores, sino su **máximo**.
 - E_L = Espacio vars. locales

$$\begin{aligned} E &= \max\{E_A, E_B, E_C\} + E_L \\ &= E_B + E_L \end{aligned}$$





Otros factores

- Existen otros factores que se pueden tener en cuenta a la hora de valorar la eficiencia de algoritmos:
 - **Localización:** Un algoritmo hace uso localizado de memoria si tras cada acceso existe una gran probabilidad de que el siguiente acceso sea a una posición cercana. Es importante en sistemas con memoria jerárquica (caches).
 - **Paralelización:** Posibilidad de dividir un algoritmo en tareas independientes. Importante en sistemas con múltiples procesadores.
 - **Gestión de memoria:** Secuencia que sigue un algoritmo a la hora de crear objetos. Importante en sistemas con recolección de basura (garbage collector).



2. NOTACIÓN ASINTÓTICA



Definición matemática

Dada una función $f(n)$ la notación $O(f(n))$ representa al **conjunto de funciones** con la siguiente propiedad:

$$g(n) \in O(f(n)) \Leftrightarrow \exists n_0 \in \mathbb{Z}^+, c \in \mathbb{R}^+ : \forall n > n_0 : g(n) \leq c \cdot f(n)$$

- El conjunto $O(f(n))$ se denomina *conjunto de cotas superiores generado por $f(n)$* .
- Toda función que pertenece a $O(f(n))$ se dice que está *acotada superiormente por $f(n)$* .
- Definición alternativa:

$$g(n) \in O(f(n)) \Leftrightarrow \lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} = \begin{cases} 0 \\ cte \end{cases}$$

$g(n)$ mas lenta que $f(n)$

$g(n)$ mismo crecimiento que $f(n)$

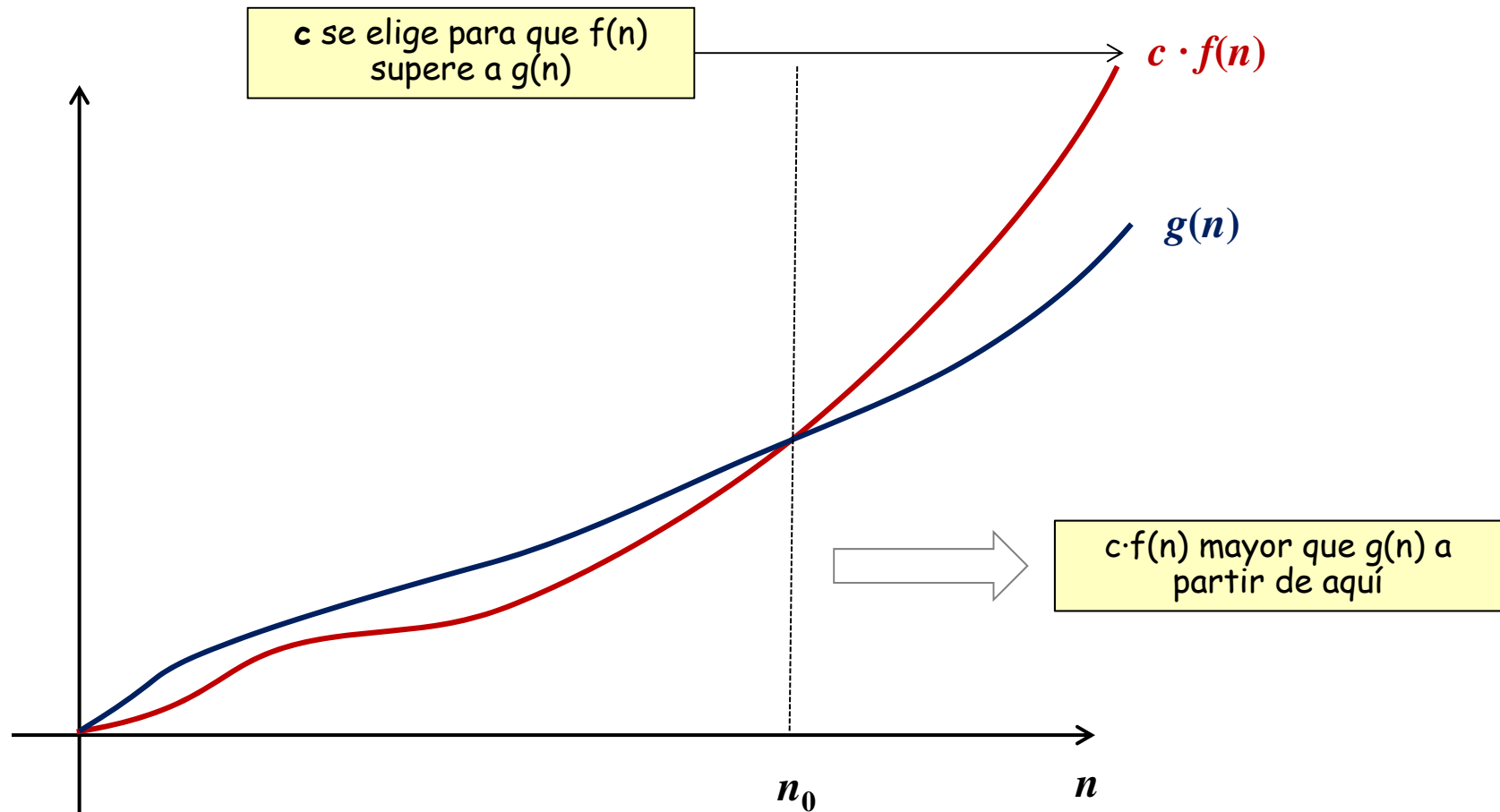
Explicación

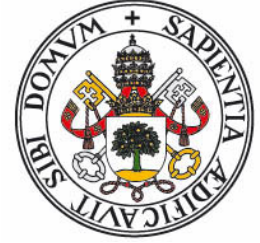


El conjunto $O(f(n))$ representa a las funciones que:

- Tienen un ritmo de crecimiento **igual o menor** que $f(n)$
- No importan las **constantes de proporcionalidad** (positivas) por las que esté multiplicada la función (podemos ajustar el valor de **c** en la definición)
- Solo importa el comportamiento para valores de n **grandes**, tendentes a infinito (podemos elegir **n_0** como queramos)
- **Nota:** Las funciones con las que tratamos son **no decrecientes**, positivas y con parámetros enteros positivos.

Representación gráfica





Ejemplos

- ¿ $3 \cdot n^2 \in O(n^2)$?

- Aplicando la definición se debe cumplir:

$$\exists n_0 \in \mathbb{Z}^+, c \in \mathbb{R}^+ : \forall n > n_0 : 3 \cdot n^2 \leq c \cdot n^2$$

- Es cierto, escogiendo $n_0 = 0$, cualquier $c > 3$

- ¿ $n/10 \in O(n^2)$?

$$\exists n_0 \in \mathbb{Z}^+, c \in \mathbb{R}^+ : \forall n > n_0 : n/10 \leq c \cdot n^2$$

- Es cierto escogiendo $n_0 = 0$, cualquier $c > 0.1$

- ¿ $0.01 \cdot n^3 \in O(n^2)$?

$$\exists n_0 \in \mathbb{Z}^+, c \in \mathbb{R}^+ : \forall n > n_0 : 0.01 \cdot n^3 \leq c \cdot n^2$$



Ejemplos

- ¿ $3 \cdot n^2 \in O(n^2)$? **Si**

- Aplicando la definición se debe cumplir:

$$\exists n_0 \in \mathbb{Z}^+, c \in \mathbb{R}^+ : \forall n > n_0 : 3 \cdot n^2 \leq c \cdot n^2$$

- Es cierto, escogiendo $n_0 = 0$, cualquier $c > 3$

- ¿ $n/10 \in O(n^2)$? **Si**

$$\exists n_0 \in \mathbb{Z}^+, c \in \mathbb{R}^+ : \forall n > n_0 : n/10 \leq c \cdot n^2$$

- Es cierto escogiendo $n_0 = 0$, cualquier $c > 0.1$

- ¿ $0.01 \cdot n^3 \in O(n^2)$? **No**

$$\exists n_0 \in \mathbb{Z}^+, c \in \mathbb{R}^+ : \forall n > n_0 : n \leq 100 \cdot c \quad \leftarrow \text{Falso!}$$

Utilidad



Sustituir en expresiones la función $g(n)$ (complicada, detallada) por $O(f(n))$ (donde $f(n)$ es más sencilla)

- En general $f(n)$ y $g(n)$ tendrán el mismo ritmo de crecimiento (**cotas ajustadas**).
- Perdemos **precisión** (constantes de proporcionalidad y términos que crecen más lentamente que el dominante)
- Ganamos **generalidad**: Muchas expresiones distintas pasan a estar representadas por una sola, usando el criterio de **crecimiento** de la función
- El **cálculo** con notación asintótica es mucho **más sencillo**
- Las **comparaciones** (basadas en ritmo de crecimiento) entre funciones **son inmediatas**.

Propiedades



- $f(n) \in O(f(n))$: Paso de expresión a notación asintótica.
- $O(k \cdot f(n)) \equiv O(f(n))$: Las constantes de proporcionalidad se omiten.
- $O(f(n) + g(n)) \equiv O(\text{máx}\{f(n), g(n)\})$: Sólo importa el término de mayor crecimiento.
- $O(f(n) - g(n)) \equiv O(f(n))$ si $g(n)$ crece más lentamente que $f(n)$, en caso contrario no se puede simplificar.
- Los productos y divisiones pueden incorporarse a la notación O-grande sin problemas.
- Dentro de la O-grande se debe usar la función **más sencilla** posible!



Jerarquía de funciones

Clasificación de funciones según su crecimiento:

- $T(n) \in O(1) \Rightarrow T(n) = \text{cte}$: Funciones **constantes**, que no dependen del tamaño de la entrada.
- $T(n) \in O(n^{1/a})$, $a \rightarrow \infty$: Funciones **subpolinómicas**, crecen más lentamente que cualquier polinomio.
- $T(n) \in O(n^a)$: Funciones **polinómicas**, existe un polinomio que las acota.
- $T(n) \notin O(n^a)$, $a \rightarrow \infty$: Funciones **no polinómicas**, crecen más rápido que cualquier polinomio.



Comparación (I)

Un algoritmo resuelve un problema con $n = 100$ en 1 seg.
Cambiamos el ordenador por uno 10 veces más rápido.
¿Cuál es ahora el tamaño para el que tardará 1 seg.?:

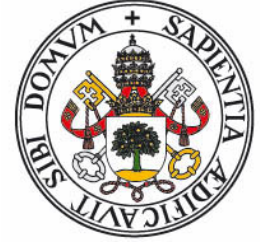
Tipo	Orden	Tamaño	Ejemplo
Constante	$O(1)$	Ilimitado	Acceso array
Subpolinómico	$O(\log n)$	100 trillones	Búsqueda binaria
	$O(\log^2 n)$	2.111.136	Planos ocultos (Doom)
Polinómico	$O(n^{1/2})$	10.000	Test primalidad
	$O(n)$	1.000	Búsqueda secuencial
	$O(n \cdot \log n)$	703	Ordenación, FFT
	$O(n^2)$	316	Ordenación (básica)
	$O(n^3)$	215	Multiplicación matrices
No polinómico	$O(2^n)$	103	Problema SAT
	$O(n!)$	101	Problema del viajante

Comparación (II)



Un algoritmo resuelve un problema con $n = 1.000$ en **1 seg.**
¿Cuanto tardará en resolver el problema para un tamaño 10 veces mayor ($n = 10.000$)?:

Tipo	Orden	Tiempo	Ejemplo
Constante	$O(1)$	1 seg.	Acceso array
Subpolinómico	$O(\log n)$	1,3 seg.	Búsqueda binaria
	$O(\log^2 n)$	1,8 seg.	Planos ocultos (Doom)
Polinómico	$O(n^{1/2})$	3,2 seg.	Test primalidad
	$O(n)$	10 seg.	Búsqueda secuencial
	$O(n \cdot \log n)$	13,3 seg.	Ordenación, FFT
	$O(n^2)$	100 seg.	Ordenación (básica)
	$O(n^3)$	17 min.	Multiplicación matrices
No polinómico	$O(2^n)$	Infinito	Problema SAT
	$O(n!)$	Infinito	Problema del viajante



Uso habitual

La notación asintótica se usa para trabajar con una expresión con el **nivel de detalle** adecuado:

- Sea $T(n) = 3n^2 - 5n + 34$. Se puede convertir en:
- $T(n) = 3n^2 - 5n + O(1)$
- $T(n) = 3n^2 + O(n)$ ← **Atención:** La O -grande siempre sumando
- $T(n) = O(n^2)$
- $T(n) = O(n^5)$ ← **Atención:** Correcto, pero cota **no ajustada**
- ~~$T(n) = 34 + O(n^2)$~~ ← **Mal:** Los términos exactos deben tener mayor crecimiento que la O -grande
- **Nota:** Se debería usar \in en vez de $=$, pero la convención matemática permite este abuso de lenguaje.



Cotas ajustadas

Cuando sustituimos $g(n)$ por $O(f(n))$, $f(n)$ debe ser una función de **igual** o **mayor** crecimiento que $g(n)$

- Salvo que se diga lo contrario, se supone que el crecimiento es **igual**.
- En ese caso se dice que la cota es **ajustada**.
- Puede haber situaciones (para simplificar cálculos, sobre todo) en que la cota obtenida no tiene garantías de estar ajustada: En ese caso debe indicarse explícitamente.
- También es posible que la cota no esté ajustada porque se refiera al **peor caso** de un algoritmo.



Otras funciones de cota

- $\Omega(f(n))$: **Cota inferior**. Funciones que crecen igual o más rápido que $f(n)$.
- $\Theta(f(n))$: **Cota estricta**. Funciones que crecen al mismo ritmo que $f(n)$.
- $\omega(f(n))$: **Cota inferior exclusiva**. Funciones que crecen más rápido (no igual) que $f(n)$.
- $o(f(n))$: **Cota superior exclusiva (o-pequeña)**. Funciones que crecen más lentamente (no igual) que $f(n)$.

Formulae



$$\sum_{i=c_1}^{n-c_2} (i + c_3)^a = \frac{n^{a+1}}{a+1} + O(n^a)$$

$$\sum_{i=c_1}^{n-c_2} O(i^a) = O(n^{a+1})$$

$$\sum_{i=a}^b 1 = b - a + 1$$

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

$$\log_b n = \frac{\log_c n}{\log_c b} = cte \cdot \log_c n \equiv O(\log n)$$



Ejemplo: Ord. por Inserción

```
type TVector = array of Tdato;  
procedure OrdIns(var V: TVector);  
var n,i,j : integer;  
    tmp : Tdato;  
begin  
    n := length(V); { N° de elementos de V }  
    for i := 1 to n-2 do  
        begin  
            j := i-1; tmp := V[i];  
            while (j > 0) and (V[j] > tmp) do  
                begin  
                    V[j+1] := V[j]; j := j-1  
                end;  
            V[j+1] := tmp  
        end  
    end;  
end;
```



Ejemplo: Ord. por Inserción

```
type TVector = array of Tdato;  
procedure OrdIns(var V: TVector);  
var n,i,j : integer;  
    tmp : Tdato;  
begin  
    n := length(V); { N° de elementos de V }  
    for i := 1 to n-1 do  
        begin  
            j := i-1; tmp := V[i];  
            while (j > 0) and (V[j] > tmp) do  
                begin  
                    V[j+1] := V[j]; j := j-1  
                end;  
            V[j+1] := tmp  
        end  
    end;  
end;
```

Peor caso:

- i comp
- $i+2$ movs

Mejor caso:

- 1 comp
- 2 movs



Ejemplo: Ord. por Inserción

- **Movimientos, peor caso, exacto:**

$$T_{peor}^{movs}(n) = \sum_{i=1}^{n-1} (i + 2) = \frac{n^2}{2} + \frac{3n}{2} - 3$$

- **Movimientos, peor caso, notación asintótica:**

$$T_{peor}^{movs}(n) = \sum_{i=1}^{n-1} i + \sum_{i=1}^{n-1} O(1) = \frac{n^2}{2} + O(n) + \cancel{O(n)} = \frac{n^2}{2} + O(n)$$

$$T_{peor}^{movs}(n) = \sum_{i=1}^{n-1} O(i) = O(n^2)$$



3. RELACIONES DE RECURRENCIA



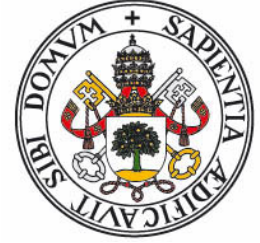
Llamadas a subrutinas

```
function EsPrimo(n: integer): boolean;  
var d : integer;  
begin  
  d := 2;  
  while (d*d <= n) and (n mod d <> 0) do d := d+1;  
  EsPrimo := d*d > n;  
end;
```

$O(n^{1/2})$

```
function NumPrimos(n: integer): integer;  
var sum,i : integer;  
begin  
  sum := 0;  
  for i :=1 to n do  
    if EsPrimo(i) then sum := sum+1;  
  NumPrimos := sum;  
end;
```

$$\sum_{i=1}^n O(\sqrt{i}) = O(n^{3/2})$$



Llamadas recursivas

Queremos contar el número de productos en función del valor de n :


```
function Fact(n: integer): integer;  
begin  
  if n <= 1 then  
    Fact := 1  
  else  
    Fact := n * Fact(n-1);  
end;
```

↓ ↓
1 + ¿?




Arbol de llamadas

```
function Fact(4): integer;  
begin  
  if 4 <= 1 then Fact := 1 else Fact := 4 * Fact(3);  
end;
```



```
if 3 <= 1 then Fact := 1 else Fact := 3 * Fact(2);
```



```
if 2 <= 1 then Fact := 1 else Fact := 2 * Fact(1);
```



```
if 1 <= 1 then Fact := 1 else Fact := 2 * Fact(1);
```



Arbol de llamadas

```
function Fact(4): integer;  
begin  
  if 4 <= 1 then Fact := 1 else Fact := 4 * 6;  
end;
```

```
if 3 <= 1 then Fact := 1 else Fact := 3 * 2;
```

```
if 2 <= 1 then Fact := 1 else Fact := 2 * 1;
```

```
if 1 <= 1 then Fact := 1 else Fact := 2 * Fact(1);
```



Evaluación con Recursividad

- La solución pasa por establecer una ecuación donde la incognita sea la **función** de medida del tiempo o espacio: $T(n)$ o $E(n)$.
- Las ecuaciones donde las incognitas son funciones, no variables, se denominan **relaciones de recurrencia**.
- El tiempo o espacio de las llamadas recursivas se representan por $T(n)$ o $E(n)$ con el parámetro adecuado.
- Las resolveremos por **sustitución** o aplicación del **teorema maestro**.



Secuencia de Fibonacci

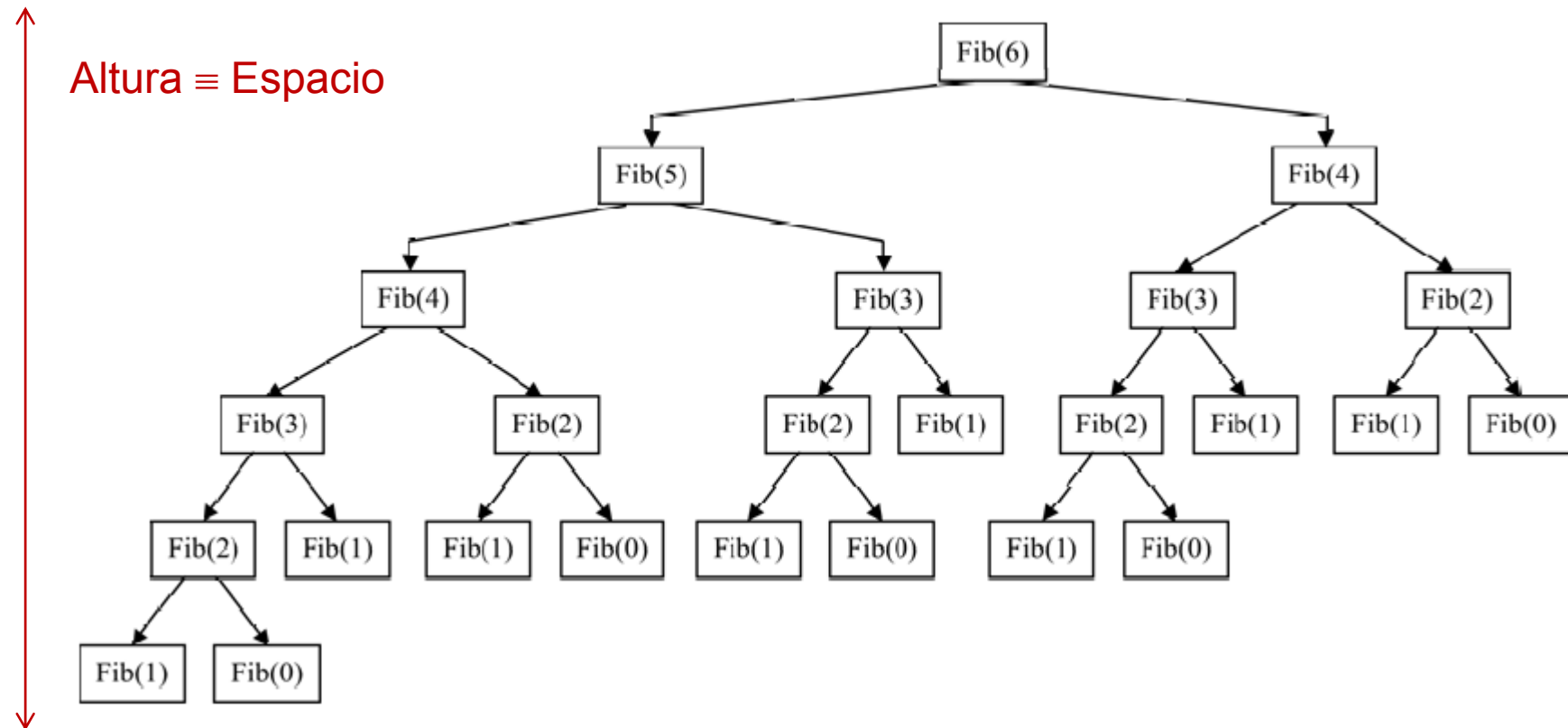
```
function Fib(n: integer): integer;  
begin  
  if n = 1 then Fib := 1 else  
  if n = 2 then Fib := 1 else  
    Fib := Fib(n-1) + Fib(n-2);  
end;
```

$$T(n) = T(n-1) + T(n-2) + O(1)$$

$$\begin{aligned} E(n) &= \max\{E(n-1), E(n-2)\} + O(1) \\ &= E(n-1) + O(1) \end{aligned}$$



Arbol de llamadas (Fibonacci)

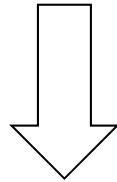


Tiempo \equiv N° de nodos

Teorema maestro

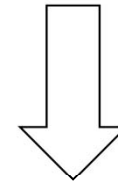


$$T(n) = a \cdot T(n - b) + O(n^k)$$

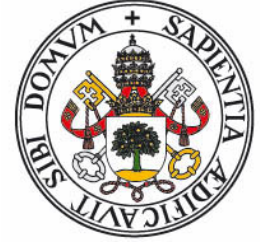


$$\begin{cases} T(n) \in O(n^{k+1}) & \text{si } a = 1 \\ T(n) \in O(a^{n/b}) & \text{si } a > 1 \end{cases}$$

$$T(n) = a \cdot T(n/b) + O(n^k)$$



$$\begin{cases} T(n) \in O(n^k) & \text{si } a < b^k \\ T(n) \in O(n^k \log n) & \text{si } a = b^k \\ T(n) \in O(n^{\log_b a}) & \text{si } a > b^k \end{cases}$$



Cotas superiores e inferiores

- Existen problemas a cuya relación de recurrencia no se puede aplicar el teorema maestro (llamadas con parámetros distintos). Por ejemplo, la función Fibonacci:

$$T(n) = T(n-1) + T(n-2) + O(1)$$

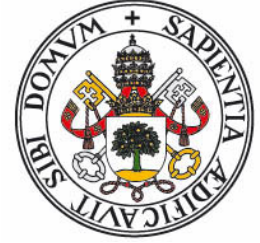
- En estos casos, lo que podemos es resolver versiones de **mayor** y **menor** crecimiento de la relación de recurrencia.

$$H(n) = 2 \cdot H(n-1) + O(1) \Rightarrow H(n) \in O(2^n)$$

$$L(n) = 2 \cdot L(n-2) + O(1) \Rightarrow L(n) \in O(2^{n/2})$$

- En este caso **H(n)** es mayor que T(n) por lo que actúa como **cota superior** suya. **L(n)** es menor que T(n), y actúa como **cota inferior**. Lo podemos expresar así:

$$T(n) \in O(2^n), T(n) \notin O(2^{\frac{n}{2}})$$



4. ALGORITMOS DE ORDENACIÓN

Taxonomía



- **Algoritmos Directos:**

- Inserción
- Selección
- Intercambio (burbuja)

$$\left. \begin{array}{l} T(n) \in O(n^2) \\ E(n) \in O(1) \end{array} \right\}$$

- **Algoritmos Avanzados:**

- Ordenación por fusión (mergesort)
- Ordenación rápida (quicksort)*
- Ordenación por montículos (heapsort)

$$\left. \begin{array}{l} T(n) \in O(n \log n) \end{array} \right\}$$

→ Caso promedio

- **Algoritmos No Universales:**

- Residuos-Recuento
- Radix sort

$$\bullet T(n, p, q) \in O(p \cdot n + p \cdot q)$$

$$\bullet T(n, m) \in O(n \log m)$$



Características

- **Universal / Específico:** Un método es universal si está basado en **comparaciones**. Es específico si se basa en propiedades especiales de los datos.
- **Tipo de acceso:** Existen algoritmos que requieren **acceso indexado** en $O(1)$ a los elementos (**arrays**). Si tan sólo se requiere **acceso secuencial**, el algoritmo se puede aplicar a otras estructuras de datos (listas enlazadas, etc.)
- **Sobre el propio vector:** Algunos algoritmos actúan sobre el propio vector y otros requieren que el resultado se proporcione sobre **otro vector** distinto del original.
- **Estabilidad:** Una estrategia de ordenación es **estable** si mantiene el **orden original** de los elementos con **claves iguales** y ello no supone una pérdida de eficiencia.

Estabilidad



Ordenación de un vector por el campo clave “Apellido”:

Apellido	Nombre
Rodriguez	Angel
Sánchez	Beatriz
Sánchez	Javier
Pérez	Laura
Rodríguez	Miguel
Pérez	Nacho
Pérez	Roberto

Vector original

Apellido	Nombre
Pérez	Laura
Pérez	Nacho
Pérez	Roberto
Rodriguez	Angel
Rodríguez	Miguel
Sánchez	Beatriz
Sánchez	Javier

Ordenación estable

Apellido	Nombre
Pérez	Nacho
Pérez	Roberto
Pérez	Laura
Rodriguez	Angel
Rodríguez	Miguel
Sánchez	Javier
Sánchez	Beatriz

Ordenación no estable



Teorema de la ordenación

No puede existir ningún algoritmo de ordenación basado en **comparaciones** cuyo tiempo de ejecución tenga una cota menor que $O(n \log n)$

- El espacio de posibles soluciones tiene un tamaño $n!$ (posibles permutaciones de un vector de n elementos)
- De ese espacio el algoritmo debe encontrar el único vector resultado (aquel que está ordenado)
- Cada comparación proporciona un bit de información
- Un algoritmo perfecto usaría cada comparación para obtener un bit del índice al espacio de permutaciones
- Ese índice contiene $\log_2(n!)$ bits.

$$\log_2(n!) \in O(n \log n)$$



Notación

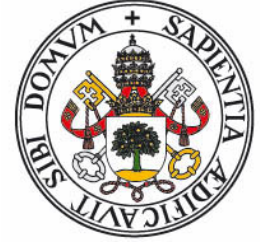
En los siguientes programas se usa la notación:

- Los arrays son dinámicos, indexados por 0.. y su número de elementos (longitud) se calcula con la función *length*.
- La operación de intercambio entre elementos del vector (`tmp := v[i]; v[i] := v[j]; v[j] := tmp`) se representa de forma abreviada como **$v[i] \Leftrightarrow v[j]$**
- Se marcan en rojo las **comparaciones** entre elementos del vector y en violeta los **movimientos o intercambios** entre elementos del vector.
- Las comparaciones se realizan entre los elementos, para simplificar la notación. En la vida real se usaran campos clave de los elementos o funciones comparadoras.



Ordenación por Inserción

```
type TVector = array of Tdato;  
procedure OrdIns(var V: TVector);  
var n,i,j : integer;  
    tmp : Tdato;  
begin  
    n := length(V); { N° de elementos de V }  
    for i := 1 to n-2 do  
        begin  
            j := i-1; tmp := V[i];  
            while (j > 0) and (V[j] > tmp) do  
                begin  
                    V[j+1] := V[j]; j := j-1  
                end;  
            V[j+1] := tmp  
        end  
    end;  
end;
```



Propiedades (Ord. Inserción)

- Eficiencia $O(n^2)$
 - Espacio $O(1)$
 - Peor caso (vector orden inverso): $n^2/2 + O(n)$
 - Mejor caso (vector ordenado): $O(n)$
 - Promedio: $n^2/4 + O(n)$
 - Las fórmulas son iguales para movimientos y comparaciones
- Método universal
- Acceso secuencial: Sólo si el acceso es bidireccional (por ejemplo listas doblemente enlazadas)
- Estable
- Adaptativo
- Sobre el propio vector



Ordenación por Selección

```
type TVector = array of Tdato;
procedure OrdSel(var V: TVector);
var
  n,i,j,jmin : integer; { Tmp : Tdato; }
begin
  n := lenght(V); { V: array[0..N-1] of Tdato }
  for i := 0 to n-2 do
  begin
    { Búsqueda del menor de zona no ordenada }
    jmin := i;
    for j := i+1 to N-1 do
      if V[j] < V[jmin] then jmin := j;
    { Intercambio menor <-> primero zona no ord. }
    V[i] ⇔ V[jmin]
  end
end;
```



Propiedades (Ord. Selección)

- Eficiencia $O(n^2)$ [$O(n^2)$ comparaciones, $O(n)$ movimientos]
 - Espacio $O(1)$
 - Siempre hace el mismo número de operaciones
 - Comparaciones: $n^2/2 + O(n)$
 - Movimientos: $3n$
- Método universal
- Acceso secuencial: Sólo si el acceso permite marcas (marcar un punto y poder volver a él)
- **No Estable** ($V[i]$ puede saltar por delante de elementos iguales)
- No adaptativo
- Sobre el propio vector



Ordenación burbuja

```
type TVector = array of TData;  
  
procedure OrdBur(var V: TVector);  
var  
    n,i,j : integer; { Tmp : TData; }  
begin  
    n := Lenght(V); { V: array[0..N-1] of TData }  
    for i := 0 to n-2 do  
        for j := n-1 downto i+1 do  
            if V[j] > V[j+1] then  
                V[j]  $\leftrightarrow$  V[j+1]  
            end;  
        end;  
    end;
```



Propiedades (Ord. Burbuja)

- Eficiencia $O(n^2)$
 - Espacio $O(1)$
 - Comparaciones: $n^2/2 + O(n)$
 - Movimientos:
 - 0 mejor caso
 - $3n^2/4 + O(n)$ promedio
 - $3n^2/2 + O(n)$ peor caso
- Método universal
- Acceso secuencial: Sólo si el acceso es bidireccional
- Estable
- No adaptativo
- Sobre el propio vector



Ordenación por Fusión

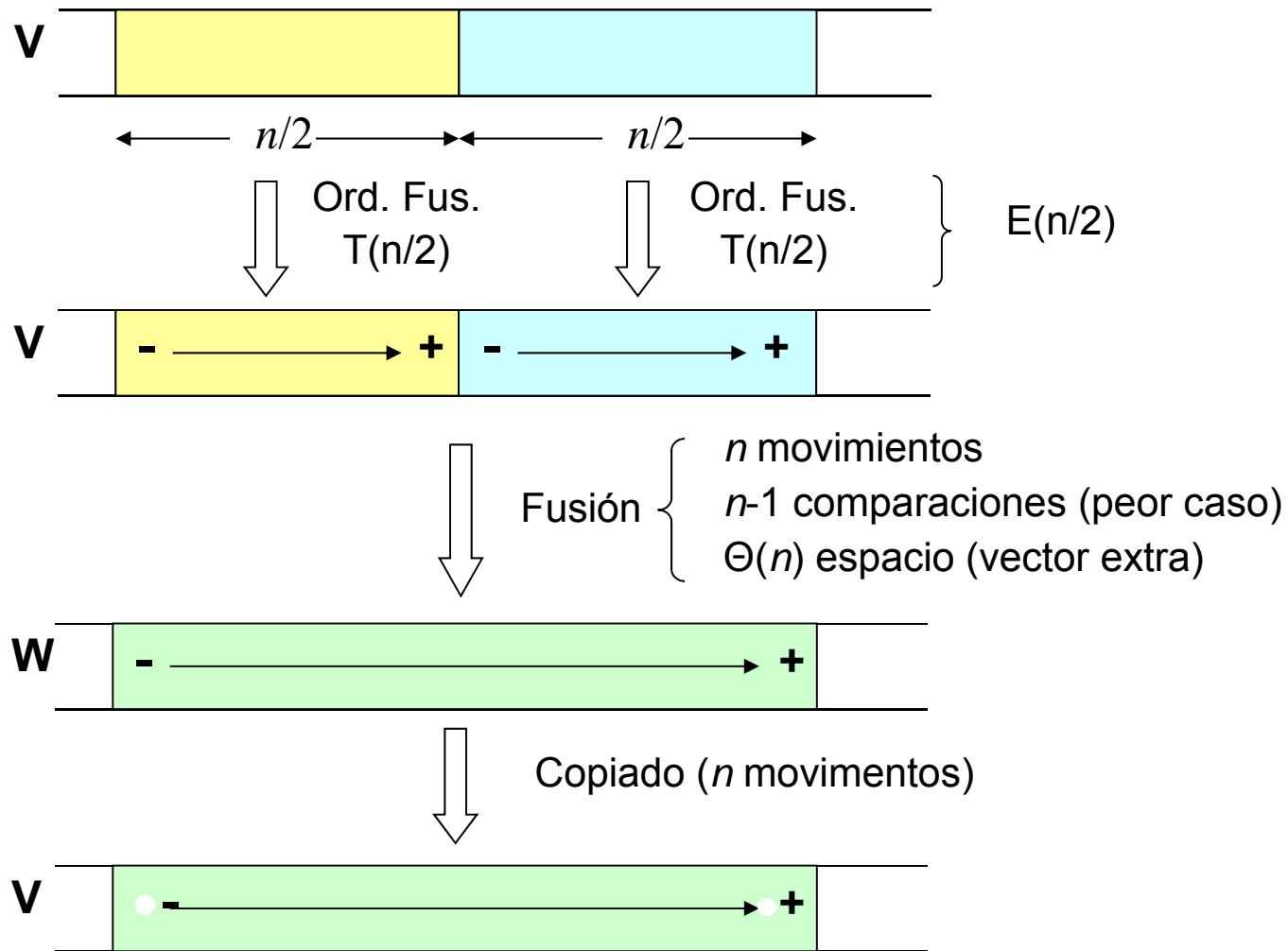
```
procedure OrdFus(var V: TVector);  
var W : TVector; { Vector temporal }  
begin  
    SetLength(W,Length(V));          { Crear vector extra }  
    OrdFusRec(V,W,0,Length(V)-1)    { Ordenar recurs. todo el vector }  
end;  
  
procedure OrdFusRec(var V,W: TVector; Ini,Fin: integer);  
{ Ordena V[Ini..Fin] }  
var Med,I : integer;  
begin  
    if Ini < Fin then  
    begin  
        Med := (Ini+Fin) div 2;  
        OrdFusRec(V,W,Ini,Med);      { Ordenación primera mitad }  
        OrdFusRec(V,W,Med+1,Fin);    { Ordenación segunda mitad }  
        Fusion(V,W,Ini,Med,Fin);     { Fusionar mitades en W }  
        for I := Ini to Fin do V[I] := W[I] { Copiar de W a V }  
    end { else caso_base }  
end;
```



Algoritmo de Fusión

```
procedure Fusion(var V,W: TVector; Ini,Med,Fin: integer);
{ Fusiona V[Ini..Med] y V[Med+1..Fin] en W[Ini..Fin] }
var Ia,Ib,Ic : integer;
begin
  { "Extraer" mínimos y llevarlos a W }
  Ia := Ini; Ib := Med+1; Ic := Ini;
  while (Ia <= Med) and (Ib <= Fin) do
    if V[Ia] < V[Ib] then
      begin
        W[Ic] := V[Ia]; Inc(Ia); Inc(Ic)
      end else begin
        W[Ic] := V[Ib]; Inc(Ib); Inc(Ic)
      end;
  { Copiar zona no vacía a W }
  while Ia <= Med do
    begin
      W[Ic] := V[Ia]; Inc(Ia); Inc(Ic)
    end;
  while Ib <= Fin do
    begin
      W[Ic] := V[Ib]; Inc(Ib); Inc(Ic)
    end;
end;
```


Análisis (Ord. Fusión)





Propiedades (Ord. Fusión)

- Eficiencia $O(n \log n)$
 - $2n \log_2 n$ movimientos
 - $n \log_2 n - n$ comparaciones (peor caso \approx caso promedio)
 - n datos (vector extra) + $O(\log n)$ espacio adicional
- Método universal
- Fácilmente adaptable a acceso secuencial (separando ambas mitades en estructuras distintas)
- Estable
- No adaptativo
- Sobre el propio vector (no se puede usar el vector extra para devolver el resultado)



Ordenación rápida

```
procedure OrdRap(var V: TVector);
var I,J : integer;
begin
    { Desordenar vector (Knuth shuffle algorithm) }
    for I := 0 to Length(V)-1 do
        V[I]  $\leftrightarrow$  V[Random(Length(V)-I)]
    { Ordenación recursiva sobre todo el vector }
    OrdRapRec(V,0,Length(V)-1)
end;
```

} Opcional

```
procedure OrdRapRec(var V: TVector; Ini,Fin: integer);
{ Ordena V[Ini..Fin] }
var Fin_Men,Ini_May : integer;
begin
    if Ini < Fin then
    begin
        Particion(V,Ini,Fin,Fin_Men,Ini_May); { Redistribuir elems. }
        OrdRapRec(V,Ini,Fin_Men); { Ordena parte de menores }
        OrdRapRec(V,Ini_May,Fin); { Ordena parte de mayores }
    end { else caso_base }
end;
```

Partición (1er método)



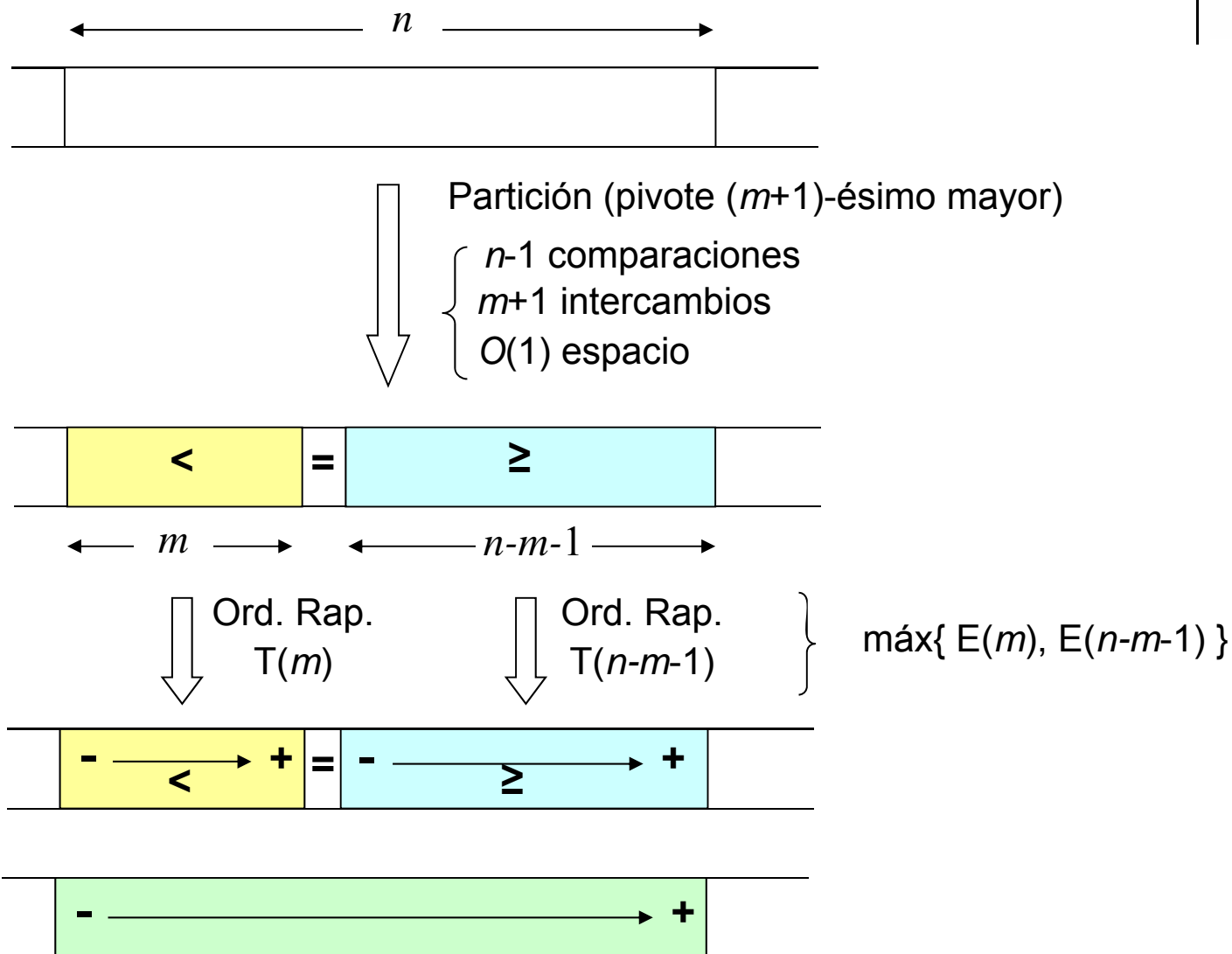
```
procedure Partición(var V: TVector; Ini,Fin: integer;
                    var Fin_Men, Ini_May: integer);
{ Reorganiza V[Ini..Fin] de manera que termine organizado en tres zonas:
  · V[Ini..Fin_Men] contiene elementos menores o iguales al pivote.
  · V[Fin_Men+1..Ini_May-1] contiene elementos iguales al pivote.
  · V[Ini_May..Fin] contiene elementos mayores o iguales al pivote.
  Ninguna zona se extiende en todo V[Ini..Fin] }
var
  Izda,Dcha : integer; Pivote : Tdato;
begin
  Pivote := V[Ini]; { Hay otras alternativas a elección de pivote }
  Izda := Ini; Dcha := Fin;
  while Izda <= Dcha do
  begin { Invariante: V[Ini..Izda-1] <= Pivote, V[Dcha+1..Fin] >= Pivote }
    while V[Izda] < Pivote do Inc(Izda);
    while V[Dcha] > Pivote do Dec(Dcha);
    if Izda <= Dcha then
    begin
      V[Izda] ⇔ V[Dcha];
      Inc(Izda); Dec(Dcha)
    end
  end;
  Fin_Men := Dcha; Ini_May := Izda
end;
```

Partición (2º método)



```
procedure Partición(var V: TVector; Ini,Fin: integer;
                    var Fin_Men, Ini_May: integer);
{ Reorganiza V[Ini..Fin] de manera que termine organizado en tres zonas:
  · V[Ini..Fin_Men] contiene elementos menores que el pivote.
  · V[Fin_Men+1] contiene el pivote.
  · V[Ini_May..Fin] contiene elementos mayores o iguales al pivote.
  Ini_May = Fin_Men+2 }
var
  I,Lim : integer;
begin
  { Se toma como pivote el 1º elemento, V[Ini] (hay otras alternativas) }
  Lim := Ini+1;
  for I := Ini+1 to Fin do
    { Invariante: V[Ini+1..Lim-1] < Pivote, V[Lim..I-1] >= Pivote }
    if V[I] < V[Ini] then
      begin
        V[I] ⇔ V[Lim];
        Inc(Lim)
      end; { if-for }
    V[I] ⇔ V[Lim-1]; { Pivote entre menores y mayores/iguales }
  Fin_Men := Lim-2; Ini_May := Lim
end;
```

Análisis (Ord. Rápida, 2º mét.)





Mejor caso

- Se da cuando las particiones son equilibradas ($m \approx n/2$)
- $T(n) = 2T(n/2) + O(n)$, $E(n) = E(n/2) + O(1)$
- Comparaciones: $n \log_2 n + O(n)$ [igual a ord. fusión]
- Movimientos: $1.5 n \log_2 n + O(n)$ [mejor que ord. fusión]
- Espacio: $O(\log n)$ [**mucho** mejor que ord. fusión]



Peor caso

- Se da cuando las particiones son desequilibradas: Una de las zonas está vacía ($m = 0$ ó $m = n-1$)
- $T(n) = T(n-1) + O(n)$, $E(n) = E(n-1) + O(1)$
- Comparaciones: $n^2/2 + O(n)$
- Movimientos: $O(n)$ si $m = 0$, $1.5 n^2 + O(n)$ si $m = n-1$
- Espacio: $O(n)$
- La ordenación rápida es un algoritmo $O(n^2)$
- Tipos de vectores que provocan el peor caso:
 - Pivote primero o último: Vector ordenado o en orden inverso
 - Pivote elem. medio: Vector con elementos en orden creciente hasta la mitad y decreciente a partir de entonces, o al revés.
 - Pivote al azar: La probabilidad de caer en el peor caso decrece exponencialmente.



Caso Promedio (I)

- Relación de recurrencia general (2º método partición)

$$T(n,m) = T(m,\cdot) + T(n-m-1,\cdot) + f(n,m)$$

- Donde m es el número de elementos menores que el pivote
- $f(n,m)$ son las operaciones no recursivas (partición):
 - $n-1$ comparaciones
 - $m+2$ intercambios
- Si el pivote es un elemento cualquiera de la zona, entonces cualquier valor de $m \in [0..n-1]$ es **equiprobable**.
- Número de operaciones promedio:

$$\hat{T}(n) = \frac{1}{n} \sum_{m=0}^{n-1} T(n,m)$$



Caso Promedio (II)

- Aplicandolo a la relación de recurrencia (comparaciones):

$$\hat{T}(n) = \frac{1}{n} \sum_{m=0}^{n-1} (\hat{T}(m) + \hat{T}(n-m-1) + n-1)$$

- Operando:

$$\hat{T}(n) = \frac{2}{n} \sum_{m=0}^{n-1} \hat{T}(m) + n-1$$

$$n \cdot \hat{T}(n) = 2 \sum_{m=0}^{n-1} \hat{T}(m) + n \cdot (n-1) \quad (\text{eq. A})$$

$$(n-1) \cdot \hat{T}(n-1) = 2 \sum_{m=0}^{n-2} \hat{T}(m) + (n-1) \cdot (n-2) \quad (\text{eq. B})$$



Caso Promedio (III)

- Restando las ecuaciones A y B:

$$n \cdot \hat{T}(n) - (n-1) \cdot \hat{T}(n-1) = 2 \cdot \hat{T}(n-1) + 2(n-1)$$

$$n \cdot \hat{T}(n) = (n+1) \cdot \hat{T}(n-1) + 2(n-1)$$

$$\frac{\hat{T}(n)}{n+1} = \frac{\hat{T}(n-1)}{n} + \frac{2(n-1)}{n \cdot (n+1)}$$

$$\frac{\hat{T}(n)}{n+1} = \frac{\hat{T}(n-1)}{n} + \frac{4}{n} - \frac{2}{n+1}$$

$$\hat{T}(n) = n \cdot \ln n + O(n) = 1.44 \cdot n \cdot \log_2 n$$



Propiedades (Ord. Rápida)

- Eficiencia $O(n^2)$
 - Peor caso: $O(n^2)$ tiempo, $O(n)$ espacio.
 - Mejor caso: $O(n \log n)$ tiempo, $O(\log n)$ espacio
 - Promedio: $O(n \log n)$ tiempo, $O(\log n)$ espacio
 - El tiempo promedio sólo es un 40% mayor que el mejor
- Método universal
- Acceso secuencial: No.
- No Estable
- No adaptativo → Antiadaptativo
- Sobre el propio vector

Ordenación por montículos



```
procedure Monticulos(var V: TVector);
var Lim,I,P,H : integer;
begin
  Lim := Length(V)-1;
  for I := (Lim-1) div 2 downto 0 do
  begin
    P := I; H := 2*I+1;
    if (H < Lim) and (V[H] < V[H+1]) then Inc(H);
    while (H <= Lim) and (V[P] < V[H]) do
    begin
      V[P]  $\leftrightarrow$  V[H]; P := H; H := 2*P+1;
      if (H < Lim) and (V[H] < V[H+1]) then Inc(H);
    end
  end;
  for I := Lim-1 downto 0 do
  begin
    V[0]  $\leftrightarrow$  V[I+1]; P := 0; H := 1;
    if (H < I) and (V[H] < V[H+1]) then Inc(H);
    while (H <= I) and (V[P] < V[H]) do
    begin
      V[P]  $\leftrightarrow$  V[H]; P := H; H := 2*P+1;
      if (H < I) and (V[H] < V[H+1]) then Inc(H);
    end
  end
end;
end;
```



Propiedades (Ord. Montículos)

- Eficiencia $O(n \log n)$
 - Espacio $O(1)$
 - No recursiva
 - Entre 2 y 5 veces más lenta que la ordenación rápida
- Método universal
- Acceso secuencial: No.
- No Estable
- No adaptativo
- Sobre el propio vector



Algoritmos no universales

- Se basan en los **valores** de las claves, no en comparar unas claves con otras.
- Supondremos que los elementos tienen un **campo clave**:
 - Dividido en p **subclaves** (p puede ser 1)
 - Organizadas en **orden lexicográfico**
 - Cada una de las cuales almacena (o se puede convertir en) un **entero positivo** en el rango $0..q-1$
- N° de posibles valores distintos de cada clave: $m = q^p$

type

TData = record

clave: array[0..P-1] of 0..Q-1;

...

end;

+ → -
Significativa

→ Constantes



Ordenación por recuento

```
procedure Recuento(const V: TVector; var W: TVector; k: integer);
{ Devuelve en W una copia del vector V ordenado por subclave k }
var
  c,i: integer;
  { Tabla de recuento. T[i] = Nº de elementos en V con clave[k] = i }
  T : array[0..Q-1] of integer;
begin
  n := length(V);
  { Inicializar la tabla de recuento }
  for c := 0 to Q-1 do T[c] := 0;
  { Contar nº de elementos con cada posible valor de subclave k }
  for i := 0 to n-1 do Inc(T[V[i].clave[K]]);
  { Convertir recuento en posición del último elemento con ese valor de
    subclave en el vector ordenado }
  T[0] := T[0]-1;
  for c := 1 to Q-1 do T[c] := T[c-1] + T[c];
  { Rellenar vector ordenado W a partir de V }
  for i := n-1 downto 0 do
  begin
    c := V[i].clave[K];
    W[T[c]] := V[i];
    T[c] := T[c]-1
  end
end;
```



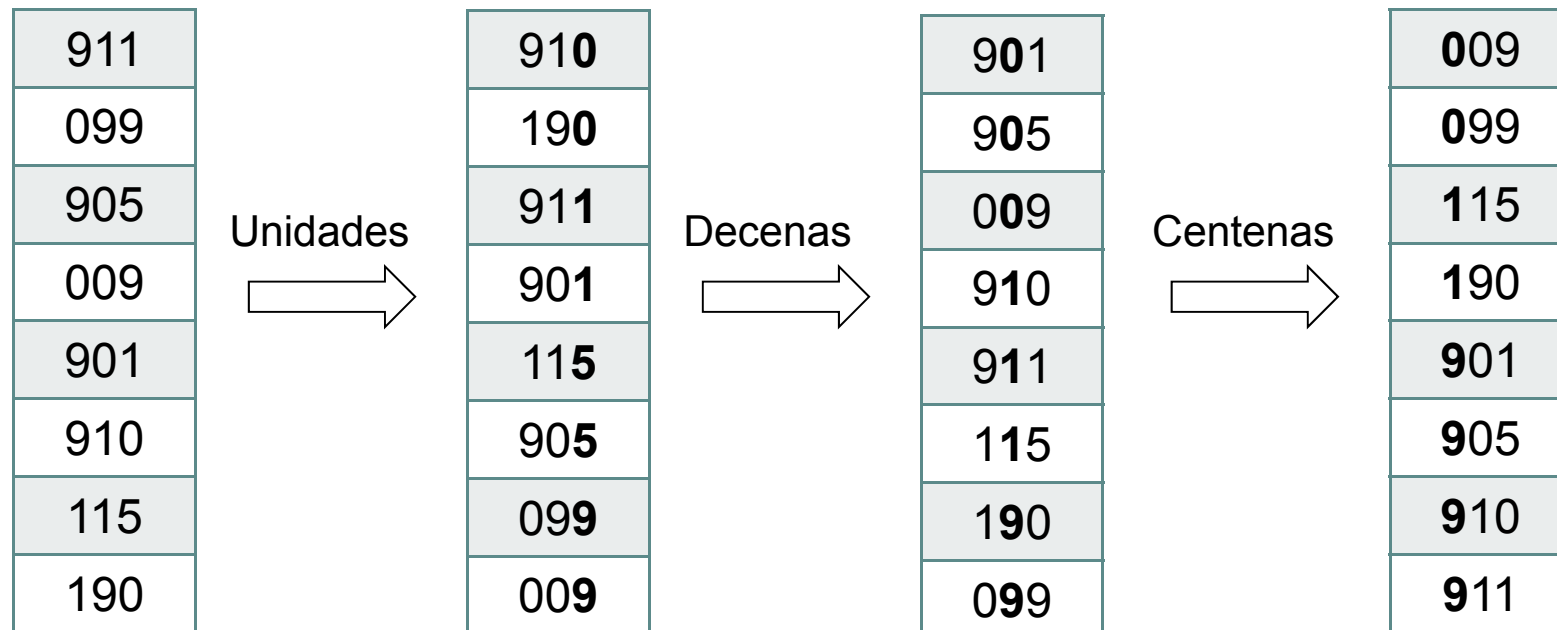
Propiedades (Ord. Recuento)

- Eficiencia (q : rango subclaves) : $O(n + q)$
 - **Espacio** $O(q)$: Tabla de recuento. No se tiene en cuenta el tamaño de los vectores de entrada y salida
 - Sólo es lineal si q es del mismo orden que n
- Método **no** universal: Las claves deben cumplir ciertos requisitos (conversión a entero)
- Acceso secuencial: No.
- **Estable** (crucial para el método residuos-recuento)
- No adaptativo
- Devuelve el resultado en **otro vector**. Si se desea ordenar sobre el propio vector hay que copiar el vector resultado sobre el original (espacio $O(q + n)$)



Estrategia de residuos

Si se dispone de una **ordenación estable** se puede ordenar por toda la clave mediante una secuencia de ordenaciones por subclaves (de - a + significativa)



Residuos - Recuento



```
procedure ResiduosRec(var V: TVector);
var
  k,i,n: integer;
  { Vector extra para las ordenaciones por subclave }
  W : TVector;
begin
  { Dimensionar vector extra }
  n := length(V);
  SetLength(W,n);
  { Ordenaciones parciales }
  for k := 0 to P-1 do
  begin
    Recuento(V,W,k);
    { Copiar W en V }
    for i := 0 to n-1 do V[i] := W[i]
  end
end;
```



Propiedades (Ord. Res-Rec)

- Eficiencia: $O(p \cdot n + p \cdot q)$
 - q : rango subclaves, p : número de subclaves
 - Espacio $O(n+q)$: Tabla de recuento y vector extra.
 - Caso habitual: Sea m el número de claves posibles, y la clave se divide en **bits**. $q = 2$ y $p = \log_2 m$

$$T(n) = O(n \log m), E(n) = O(n + \log m)$$

- Método **no** universal: Las claves deben cumplir ciertos requisitos (conversión a entero)
- Acceso secuencial: No.
- Estable
- Sobre el propio vector.



Radix-sort

- Inspirado en **quicksort** y **residuos**
- Clave **dividida en bits**
- Se realizan $p = \log_2 m$ ordenaciones por el **bit i-ésimo**
- Pero desde el bit más significativo al menos significativo (a la **inversa** que en residuos)
- No se ordena todo el vector, sino en **zonas** con el mismo valor del bit previo (recursivamente)
- Truco: Ordenar por un bit es equivalente a realizar una **partición** (sin pivote): los bits 0 a la izquierda y los bits 1 a la derecha.



Radix-sort

```
procedure Radix(var V: TVector);  
begin  
  RadixRec(V,0,length(V)-1,0)  
end;  
  
procedure RadixRec(var V: TVector; Ini,Fin,Bit: integer);  
{ Ordena V[Ini..Fin] }  
var Lim : integer;  
begin  
  if (Ini < Fin) and (Bit < P) then  
  begin  
    ParticionBit(V,Bit,Ini,Fin,Lim); { Redistribuir elems. }  
    RadixRec(V,Ini,Lim,Bit+1);      { Ordena parte de bits 0 }  
    RadixRec(V,Lim+1,Fin,Bit+1);    { Ordena parte de bits 1 }  
  end { else caso_base }  
end;
```

Partición por bits



```
procedure ParticionBit(var V: TVector; K,Ini,Fin: integer;
                      var Lim: integer);
{ Reorganiza V[Ini..Fin] en dos zonas:
  · V[Ini..Lim] contiene elementos con el bit k-ésimo = 0.
  · V[Lim+1..Fin] contiene elementos con el bit k-ésimo = 1. }
var
  Izda,Dcha : integer;
begin
  Izda := Ini; Dcha := Fin;
  while Izda <= Dcha do
  begin { Invariante: V[Ini..Izda-1] = bit 0, V[Dcha+1..Fin] = bit 1 }
    while V[Izda].clave[k] = 0 do Inc(Izda);
    while V[Dcha].clave[k] = 1 do Dec(Dcha);
    if Izda <= Dcha then
    begin
      V[Izda] ⇔ V[Dcha]; Inc(Izda); Dec(Dcha)
    end
  end;
  Lim := Dcha;
end;
```



Propiedades (Radix-sort)

- Eficiencia: $O(n \log m)$
 - m : número de claves posibles
 - Espacio $O(\log m)$: Llamadas recursivas.
- Método **no** universal: Las claves deben poderse comparar mediante su secuencia lexicográfica de bits (contraejemplo: números reales)
- Acceso secuencial: No.
- No adaptativo
- **No estable**
- Sobre el propio vector.



Resumen final

Algoritmo	Tiempo	Espacio	Estable
Inserción	$O(n^2)$	$O(1)$	Si
Selección	$O(n^2)$	$O(1)$	No
Burbuja	$O(n^2)$	$O(1)$	Si
Fusión	$O(n \log n)$	$O(n)$	Si
Rápida	$O(n^2)$ [peor] $O(n \log n)$ [prom]	$O(n)$ [peor] $O(\log n)$ [prom]	No
Montículos	$O(n \log n)$	$O(1)$	No
Residuos- Recuento	$O(p \cdot n + q \cdot n)$	$O(n + q)$	Si
Radix-Sort	$O(n \log m)$	$O(\log m)$	No



A. APÉNDICE: ALGORITMOS DE ORDENACIÓN EN HASKELL



Ordenación por Inserción

```
insertar :: Ord a => a -> [a] -> [a]
insertar x [] = [x]
insertar x (y:ys) = if x <= y
                    then x:y:ys
                    else y:(insertar x ys)
```

```
ordIns :: Ord a => [a] -> [a]
ordIns [] = []
ordIns (x:xs) = insertar x (ordIns xs)
```



Ordenación por Selección

```
minLis :: Ord a => [a] -> a
minLis [x] = x
minLis (x:xs) = if x < y then x else y
                where y = minLis xs

borraElem :: Eq a => a -> [a] -> [a]
borraElem x [] = []
borraElem x (y:ys) = if x == y
                      then xs
                      else x:(borraElem x ys)

ordSel :: Ord a => [a] -> [a]
ordSel [] = []
ordSel lis = x:(ordSel (borraElem x lis))
              where x = minLis lis
```

Ordenación por Intercambio (burbuja)



```
pasada :: Ord a => [a] -> [a]
pasada [] = []
pasada [x] = [x]
pasada x:xs = if x < y then x:y:ys else y:x:ys
               where y:ys = pasada xs

ordInt :: Ord a => [a] -> [a]
ordInt [] = []
ordInt [x] = [x]
ordInt lis = y:(ordInt ys)
               where y:ys = pasada lis
```



Ordenación por Fusión

```
fusion :: Ord a => [a] -> [a] -> [a]
fusion [] 12 = 12
fusion 11 [] = 11
fusion (x:xs) (y:ys) = if x < y
                        then x : fusion xs (y:ys)
                        else y : fusion (x:xs) ys

ordFus :: Ord a => [a] -> [a]
ordFus [] = []
ordFus [x] = [x]
ordFus lis = fusion (ordFus izda) (ordFus dcha)
            where
                mitad = (length lis) `div` 2
                izda = take mitad lis
                dcha = drop mitad lis
```



Ordenación Rápida

```
ordRap :: Ord a => [a] -> [a]
ordRap [] = []
ordRap [x] = [x]
ordRap (x:xs) = (ordRap menores) ++ [x] ++ (ordRap mayores)
               where
                 menores = [y | y <- xs, y < x]
                 mayores = [z | z <- xs, z >= x]
```



Radix-sort

```
import Data.Bits
```

```
ordRad :: Bits a => [a] -> Int -> [a]
```

```
ordRad [] _ = []
```

```
ordRad [x] _ = [x]
```

```
ordRad lis (-1) = lis
```

```
ordRad lis bit = (ordRad ceros (bit-1)) ++ (ordRad unos (bit-1))
```

```
  where
```

```
    ceros = [y | y <- lis, testBit y bit == False]
```

```
    unos  = [z | z <- lis, testBit z bit == True]
```