Estructuras de Datos y Algoritmos

Tema 1: Análisis de Algoritmos

Departamento de Informática
Universidad de Valladolid

Curso 2020-21

Grado en Ingeniería Informática
Grado en Estadística

1. MEDIDA DE ALGORITMOS

Análisis de Algoritmos

- Estudio teórico de la **eficiencia** de un algorítmo
- **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

```
// tipo representa un tipo de dato cualquiera

// Devuelve el índice donde se encuentra la primera
// aparición de x en v o -1 si no existe

int busqueda(tipo[] v, tipo x) {
  int i = 0, n = v.lenght;
  while(i < n && v[i] != x) { i++; }
  if(i < n) { return i; } else { return -1; }
}
```

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.
- **Depencia 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 $\mathrm{T}(n)$, y la espacial por $\mathrm{E}(n)$.
- 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 todos las entradas de ese subconjunto. (análisis de peor y mejor caso)
  - Calcular un promedio, dada una distribución estadística de las entradas. Tipicamente 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.
  - $\mathrm{T}_{\text{peor}}(n) = \mathbf{máx}\{\ \mathrm{T}(n, \mathit{entrada})\ \}$ 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.
  - $\mathrm{T}_{\text{mejor}}(n) = \mathbf{min}\{\ \mathrm{T}(n, \mathit{entrada})\ \}$
  - No suele ser un caso relevante.

Ejemplo: Búsqueda secuencial

- **Operación elemental**: Elegimos contar comparaciónes 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.

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

- **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.

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

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

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

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

Ejemplo: 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 ➔ 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 (3)

- 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 $\mathrm{T}_{\mathrm{peor}}(n)$ y $\mathrm{T}_{\mathrm{mejor}}(n)$.
  - 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).

Inserción en vector: 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:

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

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

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

- **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

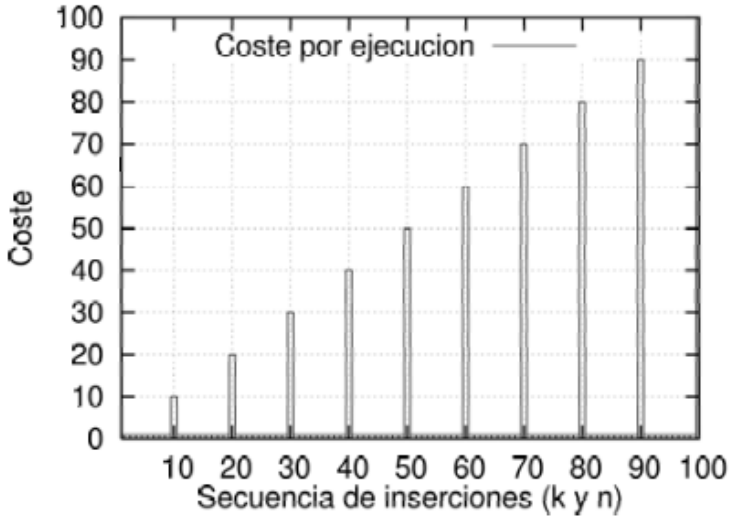
- **Primer escenario**: Si el algoritmo amplia el vector en 100 posiciones, esto garantiza que tras un inserción en un vector lleno las siguientes 99 inserciones caerán en el mejor caso:

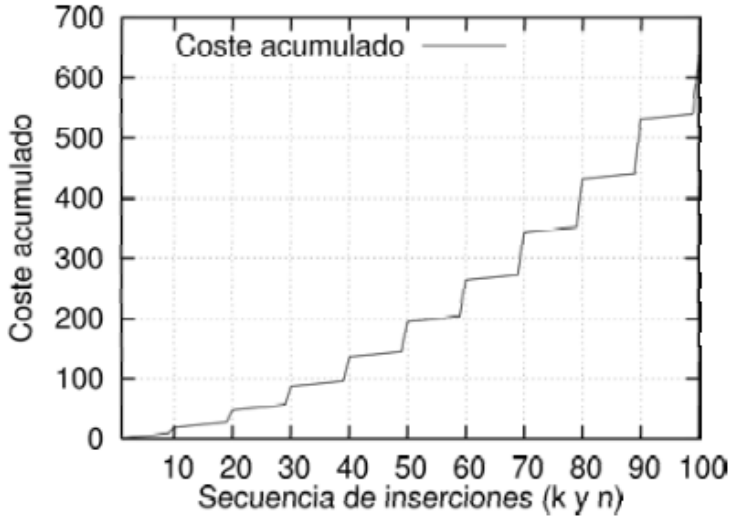
$$T_{amort}(n) = \frac{(n+1) + 99 \cdot 1}{1 + 99} = \frac{n}{100} + 1$$

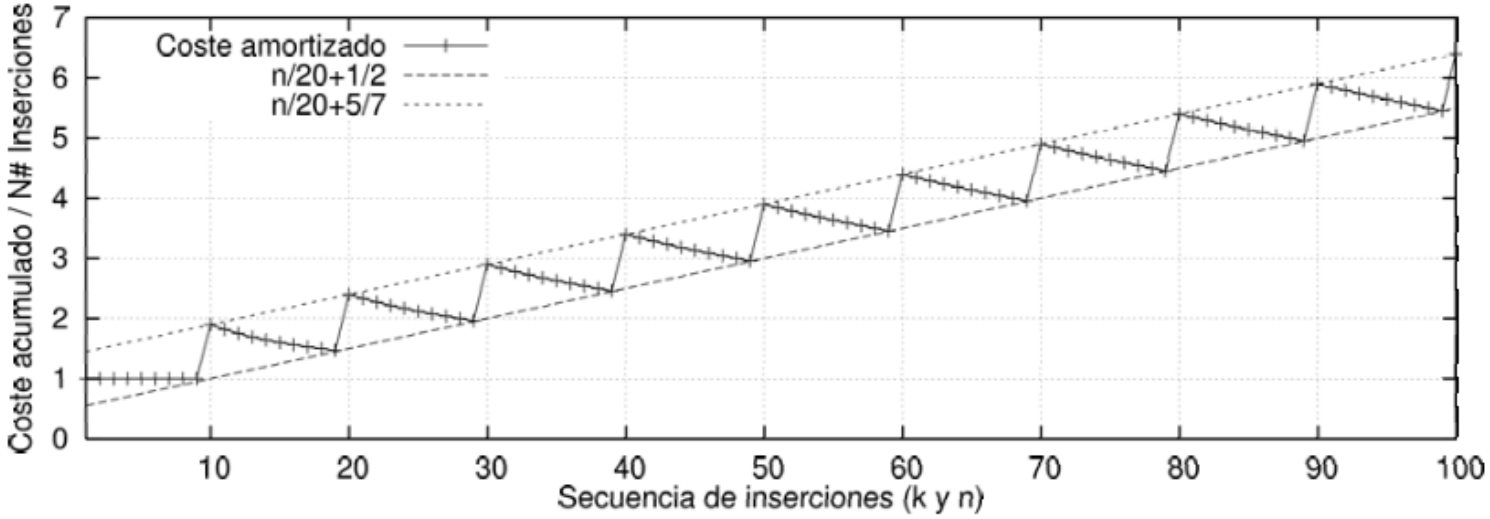
- **Segundo escenario:** Si el algoritmo duplica la capacidad del vector, esto garantiza que tras cada inserción en un vector lleno las siguientes n-1 inserciones caerán en el mejor caso:

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

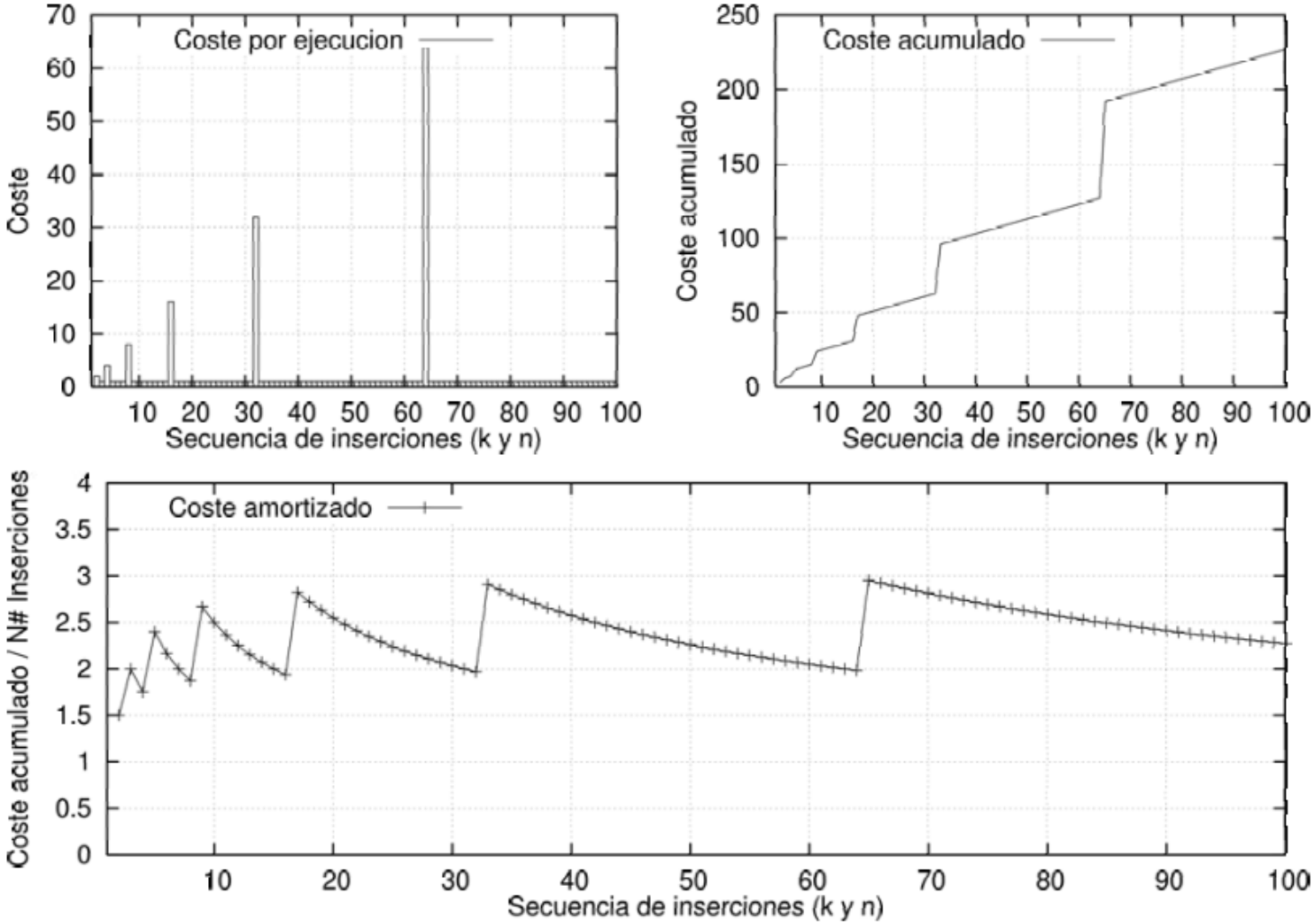
Primer escenario







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 requirir 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

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

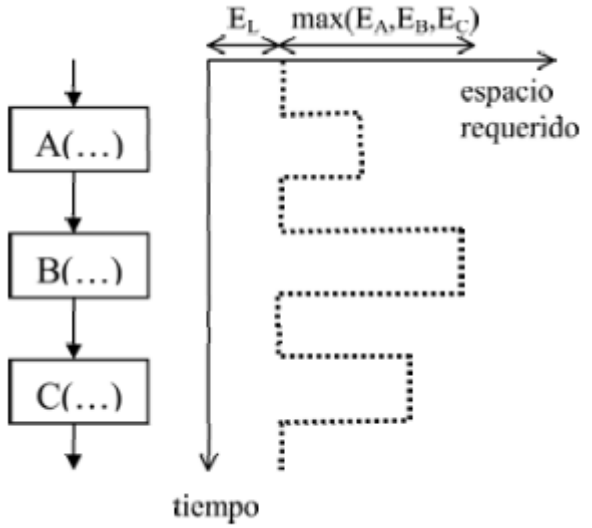
Complejidad espacial

- **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.
  - Tambien 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

$$E = max\{E_A, E_B, E_C\} + E_L$$
$$= E_B + E_L$$



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:

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

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

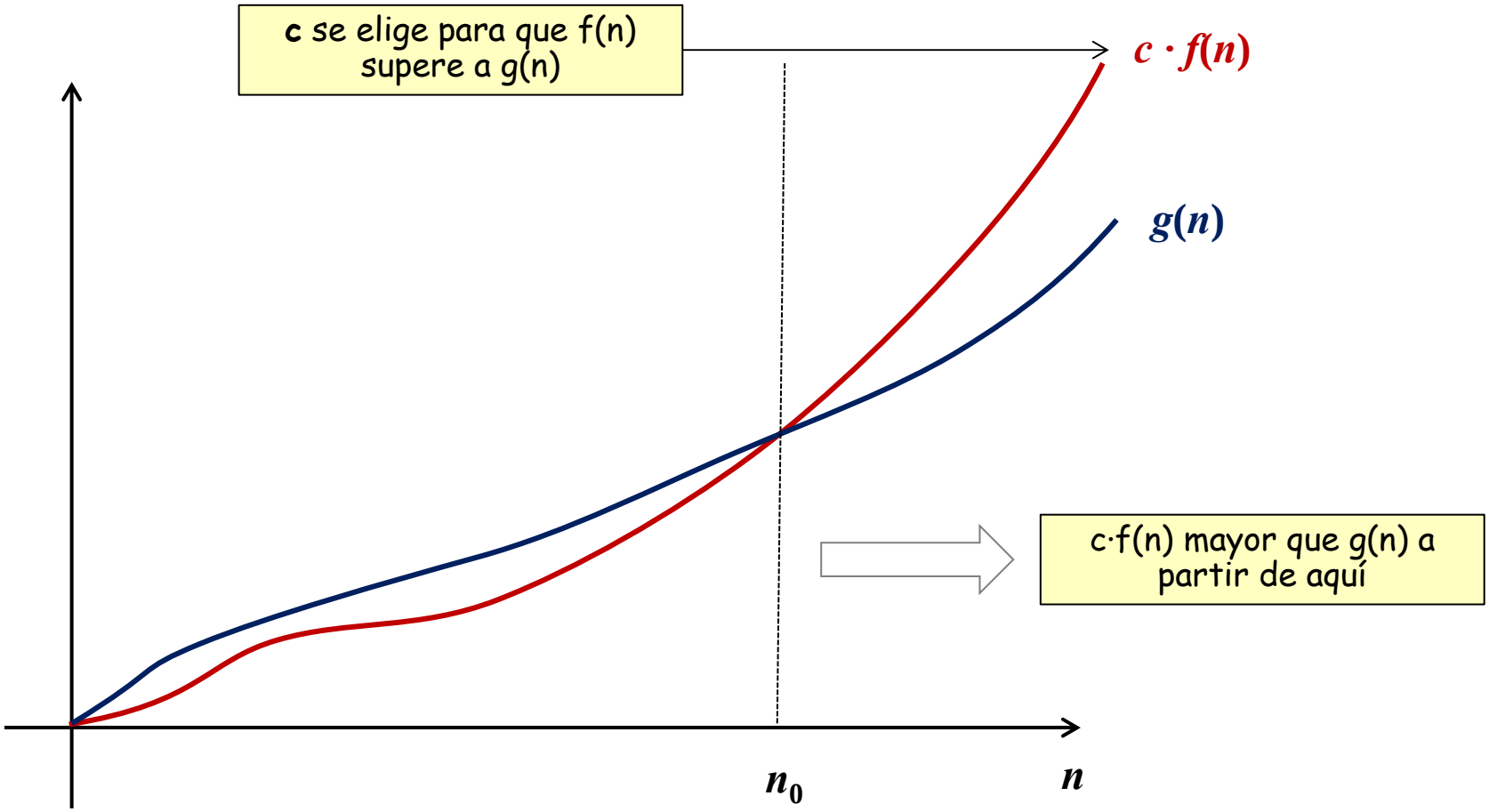
g(n) **mismo crecimiento** que f(n)

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, con 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, con 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, con 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, con cualquier c > 0.1
- **¿$0.01 \cdot n^3 \in O(n^2)$? No**

$$\exists n_0 \in \mathbb{Z}^+, c \in \mathbb{R}^+ : \forall n > n_0 : \boxed{n \leq 100 \cdot c}$$

⇦ Falso!

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!

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.

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.

Jerarquía de funciones

Clasificación de funciones según su crecimiento:

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

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

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

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

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

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

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

Ejemplo: Ordenación por Inserción

```
public static void ord_ins(double[] v)
{
    int n = v.length;
    for(int i = 1; i < n; i++) {
        double tmp = v[i];
        int j = i-1;
        while(j > -1 && v[j] > tmp) {
            v[j+1] = v[j];
            j--;
        }
        v[j+1] = tmp;
    }
}
```

Ejemplo: Ordenación por Inserción

```
public static void ord_ins(double[] v)
{
    int n = v.length;
    for(int i = 1; i < n; i++) {
        double tmp = v[i];
        int j = i-1;
        while(j > -1 && v[j] > tmp) {
            v[j+1] = v[j];
            j--;
        }
        v[j+1] = tmp;
    }
}
```

Existen i elementos en v[0..i-1]

Peor caso:

- i comparaciones
- i+2 movimientos

Mejor caso:

- 1 comparaciones
- 2 movimientos

Ejemplo: Ord. por Inserción

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

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

- **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

Medida de llamadas a subrutinas

```
// Comprueba si n es primo
public static boolean es_primo(int n) {
    int d = 2;
    while(d*d < n && n % d != 0) { d++; }
    return d*d > n;
}
```

$O(\sqrt{n})$

```
// Nº de primos entre 1 y n
public static int num_primos(int n) {
    int num = 0;
    for(int i = 1; i <= n; i++) {
        if(es_primo(n)) { num++; }
    }
    return num;
}
```

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

Llamadas recursivas

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

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

1 + ¿?

Arbol de llamadas (call stack)

```
public static int fact(4) {
  if(4 <= 1) { return 1; } else { return 4*fact(3); }
}
```

```
public static int fact(3) {
  if(3 <= 1) { return 1; } else { return 3*fact(2); }
}
```

```
public static int fact(2) {
  if(2 <= 1) { return 1; } else { return 2*fact(1); }
}
```

```
public static int fact(1) {
  if(1 <= 1) { return 1; } else { return 1*fact(0); }
}
```

Arbol de llamadas (call stack)

```
public static int fact(4) {
  if(4 <= 1) { return 1; } else { return 4*6; }
}
```

```
public static int fact(3) {
  if(3 <= 1) { return 1; } else { return 3*2; }
}
```

```
public static int fact(2) {
  if(2 <= 1) { return 1; } else { return 2*1; }
}
```

```
public static int fact(1) {
  if(1 <= 1) { return 1; } else { return 1*fact(0); }
}
```

Medida de algoritmos recursivos

- 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**.

Solución por sustitución

Llamamos $T(n)$ al número de productos que se realizan en una llamada a `fact(n)`:

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

$T(1) = 0$ $\qquad$ $T(n) = 1 + T(n\text{-}1)$

$$
\begin{aligned}
T(n) &= 1 + T(n - 1) \\
&= 1 + 1 + T(n - 2) \\
&= 1 + 1 + 1 + T(n - 3) \\
&= k + T(n - k) \\
&= n - 1 + T(n - (n - 1)) = n - 1 + T(1) = n - 1
\end{aligned}
$$

Un ejemplo más difícil

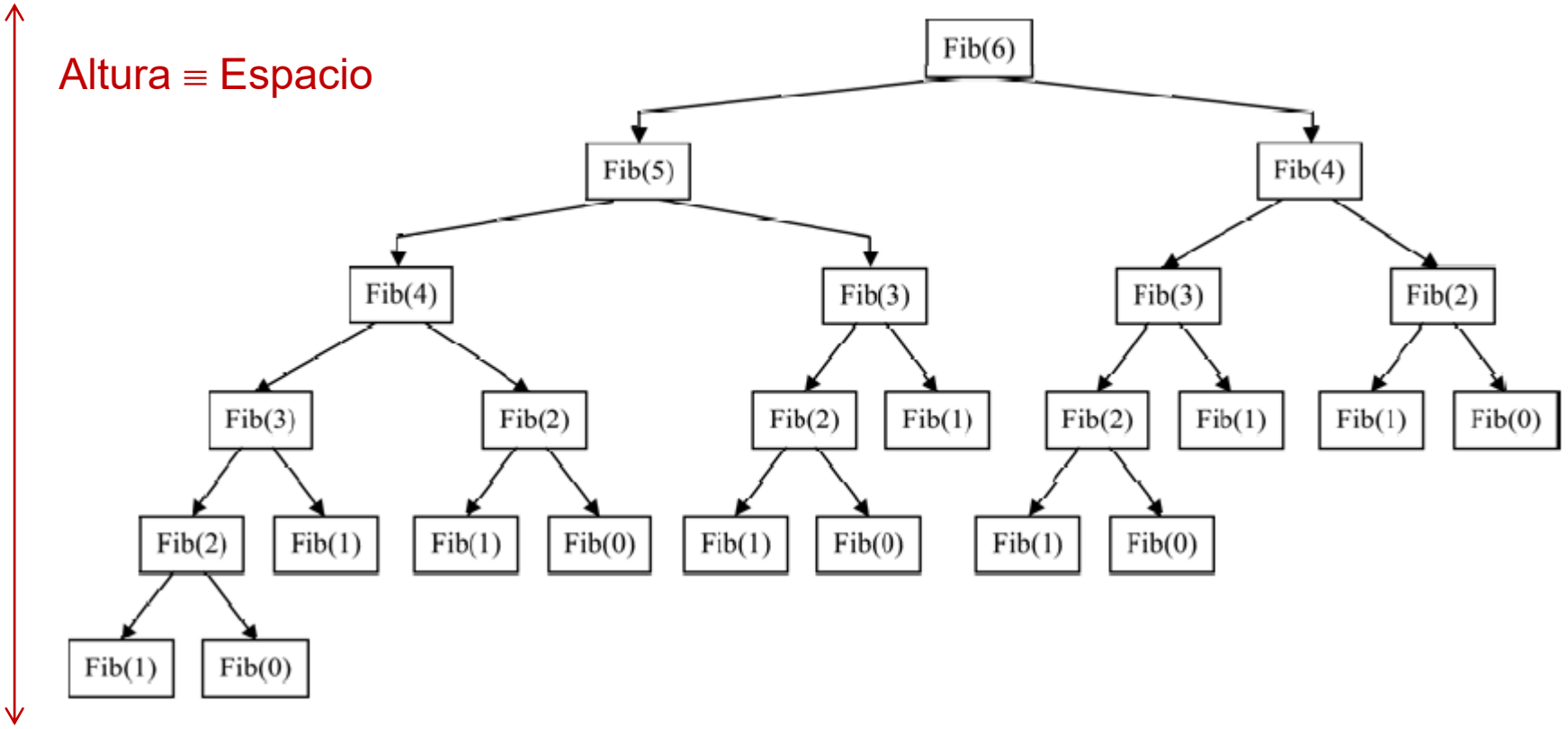
Secuencia de Fibonacci: Calcular el *n-ésimo término*:

```
public static int fib(int n) {
    if(n == 1) { return 1; } else
    if(n == 2) { return 2; } else
    { return fib(n-1) + fib(n-2); }
}
```

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

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

Arbol de llamadas (Fibonacci)



Tiempo ≡ Nº de nodos

Teorema Maestro

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

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

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

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

Teorema Maestro - Factorial

Para factorial la ecuación era: $T(n) = T(n - 1) + 1$

$$T(n) = 1 \cdot T(n - 1) + O(n^0)$$

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

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

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

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

Tenemos $a = 1, b = 1, k = 0$ y la solución es

$$\mathrm{T}(n) = O(n^{k+1}) = O(n^1) = O(n)$$

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 actua como cota superior suya. **L(n)** es menor que T(n), y actua 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)

$T(n) \in O(n^2)$
$E(n) \in O(1)$

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

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

Caso promedio

- **Algoritmos específicos**:
  - Radix sort

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

Nº bits clave

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**: La mayoría de algoritmos requieren **acceso indexado** $O(1)$ 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 actuan 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

La estabilidad puede ser esencial

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

911
099
905
009
901
910
115
190

Unidades →

91**0**
19**0**
91**1**
90**1**
11**5**
90**5**
09**9**
00**9**

Decenas →

9**0**1
9**0**5
0**0**9
9**1**0
9**1**1
1**1**5
1**9**0
0**9**9

Centenas →

009
099
115
190
901
905
910
911

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 trozos de código siguientes se usan las convenciones:

- 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] ⇔ v[j]`**
- Se marcarán en rojo las operaciones de **comparación** entre elementos del vector y en azul los **movimientos o intercambios** entre elementos del vector.
- Los ejemplos de algoritmos de ordenación clásica y avanzada mostrados a continuación trabajan sobre arrays de números reales, de esa forma las comparaciones se pueden realizar directamente entre los propios elementos. En la vida real típicamente los arrays almacenaran tipos de datos complejos (objetos) y las comparaciones se realizarán mediante campos clave de los elementos o funciones comparadoras.

Ordenación por Inserción

```
static void ord_ins(double[] v)
{
    int n = v.length;
    for(int i = 1; i < n; i++) {
        double tmp = v[i];
        int j = i-1;
        // Búsqueda del punto de inserción (con desplazamiento)
        while(j > -1 && v[j] > tmp) {
            v[j+1] = v[j];
            j--;
        }
        // Inserción del elemento
        v[j+1] = tmp
    }
}
```

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

```
static void ord_sel(double[] v) {
    int n = v.length;
    for(int i = 0; i < n-1; i++) {
        int jmin = i;
        // Búsqueda del menor de la zona no ordenada
        for(int j = i+1; j < n; j++) {
            if(v[j] < v[jmin]) { jmin = j; }
        }
        // Intercambio menor <-> primero zona no ordenada
        v[i] ⇔ v[jmin];
    }
}
```

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

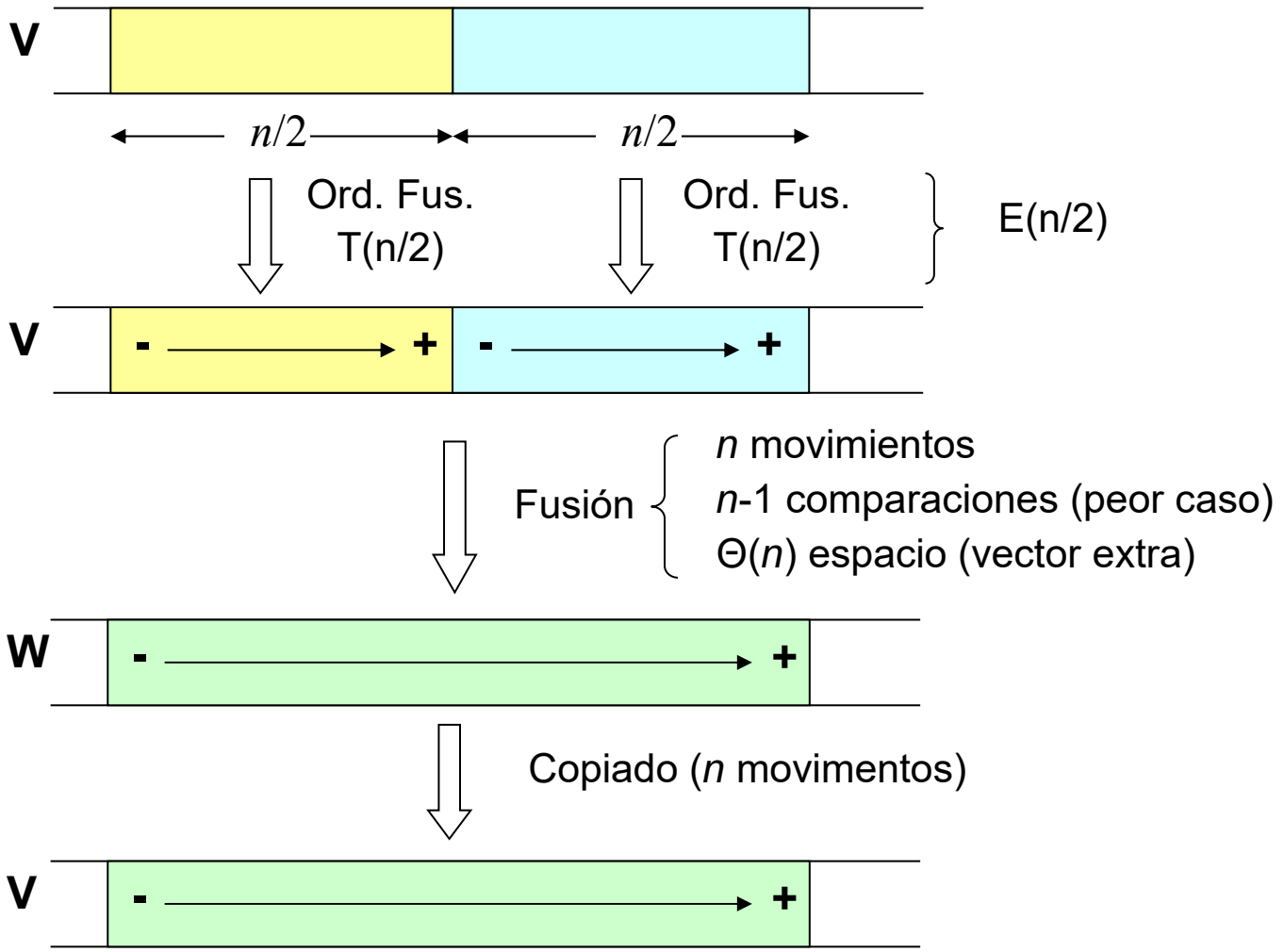
Ordenación burbuja

```
static void ord_bur(double[] v) {
    int n = v.length;
    for(int i = 0; i < n-1; i++) {
        // Intercambio de parejas en zona no ordenada [i..n-1]
        for(int j = n-1; j > i; j--) {
            if(v[j-1] > v[j]) { v[j-1] ⇔ v[j]; }
        }
    }
}
```

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



Algoritmo de Fusión (no es la ordenación, ojo!)

```
// Fusiona v[ini..med] y v[med+1..fin] en w[ini..fin]
static void fusion(double[] v, double[] w,
                   int ini, int med, int fin) {
    int i1 = ini;   // Indice a primera mitad, v[ini..med]
    int i2 = med+1; // Indice a segunda mitad, v[med+1..fin]
    for(int i = ini; i <= fin; i++) {
        if(i1 > med) { // Se ha acabado la primera mitad
            w[i] = v[i2++];
        } else if(i2 > fin) { // Se ha acabado la segunda mitad
            w[i] = v[i1++];
        } else if(v[i1] < v[i2]) { // Muevo el menor
            w[i] = v[i1++];
        } else {
            w[i] = v[i2++];
        }
    } }
```

Ordenación por Fusión

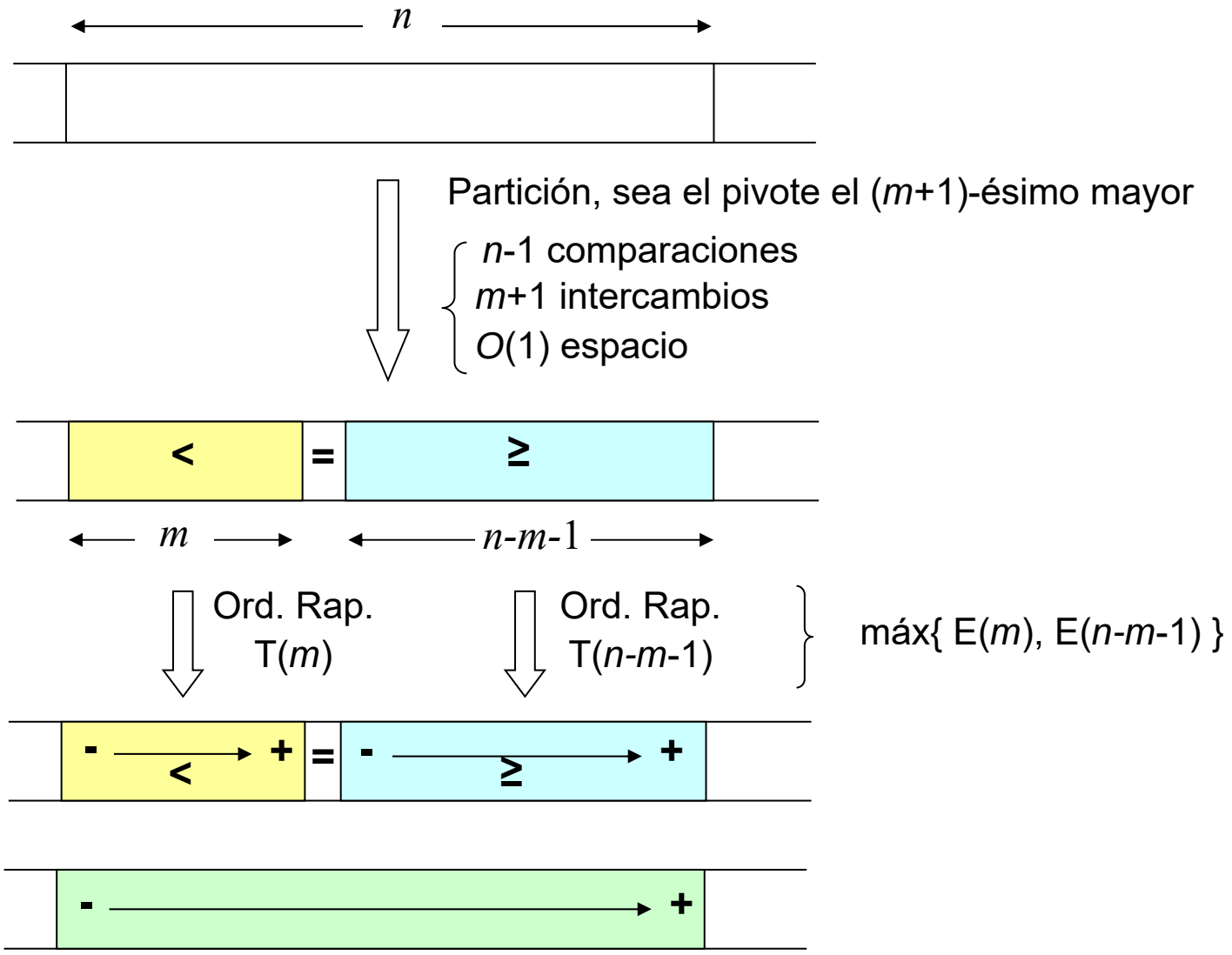
```
// Ordena el trozo v[ini..fin] del vector (usa w de vector temporal)
static void ord_fus_rec(double[] v, double[] w,
                        int ini, int fin) {
   if(ini >= fin) { return; }
   int med = (ini+fin)/2;
   ord_fus_rec(v,w,ini,med);   // Ordena la primera mitad
   ord_fus_rec(v,w,med+1,fin); // Ordena la segunda mitad
   fusion(v,w,ini,med,fin);    // Fusiona en w
   // Copiar w a v
   for(int i = ini; i <= fin; i++) { v[i] = w[i]; }
}

static void ord_fus(double[] v) {
    double[] w = new double[v.length]; // Vector temporal
    ord_fus_rec(v,w,0,v.length-1);
}
```

Propiedades (Ord. Fusión)

- Eficiencia $O(n \log n)$
  - $2n \log_2 n$ movimientos
  - $n \log_2 n - n$ comparaciones (peor caso ≈ 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



Algoritmo de Partición (el pivote es el primero)

```
// Reorganiza v[ini..fin] en 3 zonas:
// v[ini..lim-1] elementos menores que el pivote (el primero)
// v[lim+1..fin] elementos mayores o iguales al pivote
// v[lim] contiene al pivote
// La función devuelve el valor lim
static int particion(double[] v, int ini, int fin) {
    int lim = ini+1; // El pivote es v[ini]
    for(int i = ini+1; i <= fin; i++) {
        // Invariante: v[ini+1..lim-1] <, v[lim..i-1] >=
        if(v[i] < v[ini]) { v[i] ⇔ v[lim]; lim++; }
    }
    v[ini] ⇔ v[lim-1]; // Pivote entre menores y mayores-iguales
    return lim-1;
}
```

Ordenación Rápida (versión peligrosa)

```
// Ordena el trozo v[ini..fin] del vector
static void ord_rap_rec(double[] v, int ini, int fin) {
    if(ini >= fin) { return; }
    int lim = particion(v,ini,fin);
    ord_rap_rec(v,ini,lim-1); // Ordena la zona de los menores
    ord_rap_rec(v,lim+1,fin); // Ordena la zona de los mayores
}

static void ord_rap(double[] v) {
    ord_rap_rec(v,0,v.length-1);
}
```

Como se escoge como pivote el primer elemento, esta versión tiene como peor caso $O(n^2)$ a aquellos vectores que estén casi ordenados.

Ordenación Rápida (versión + segura)

```
import java.util.Random;

static rnd = new Random(); // Generador de números aleatorios

static void ord_rap(double[] v) {
    // Antes de ordenar, desordenamos el vector, alg. Knuth shuffle
    int n = v.length;
    for(int i = 0; i < n; i++) {
        v[i] ⇔ v[rnd.nextInt(n-i)];
    }
    ord_rap_rec(v,0,n-1);
}
```

Al tener un vector desordenado, es muy improbable que el pivote (primer elemento de cada subvector) sea siempre el menor o mayor del subvector.

Mejor caso

- Se da cuando las particiones son equilibradas ($m \approx n/2$)
- $\mathrm{T}(n) = 2\mathrm{T}(n/2) + O(n)$, $\mathrm{E}(n) = \mathrm{E}(n/2) + O(1)$
- 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 $m = n\text{-}1$)
- $T(n) = T(n\text{-}1) + O(n)$, $E(n) = E(n\text{-}1) + O(1)$
- Comparaciones: $n^2/2 + O(n)$
- Movimientos: $O(n)$ si $m = 0$, $1.5\ n^2 + O(n)$ si $m = n\text{-}1$
- 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

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

- 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 $m \in [0..n\text{-}1]$ 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)*:*

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

- Operando:

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

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

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

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

```
static void ord_mon(double[] v) {
    int m = v.length-1;
    for(int i = (m-1)/2-1; i >= 0; i--) {
        int p = i, h = 2*p+1;
        if(h < m && v[h] < v[h+1]) { h++; }
        while(h <= m && v[p] < v[h]) {
            v[p] ⇔ v[h];
            p = h; h = 2*p+1
            if(h < m && v[h] < v[h+1]) { h++; }
        }
    }
    for(int i = m-1; i >= 0; i--)  {
        v[0] ⇔ v[i+1];
        int p = 0, h = 1;
        if(h < i && v[h] < v[h+1]) { h++; }
        while(h <= i && v[p] < v[h]) {
            v[p] ⇔ v[h];
            p = h; h = 2*p+1
            if(h < i && v[h] < v[h+1]) { h++; }
        }
    } }
```

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** (el campo por el que se ordena):
  - Dividido en p **subclaves** (p puede ser 1)
  - Organizadas en **orden lexicográfico** (de más significativa a menos significativa – como el orden alfabético habitual)
  - Cada una de las cuales almacena (o se puede convertir en) un **entero positivo** en el rango $0..q\text{-}1$
- Nº de posibles valores distintos de cada clave: $m = q^p$

Ordenación Radixsort

- Inspirado en **quicksort**.
- Clave **dividida en bits** $(q = 2)$
- Existen p subclaves, el número máximo de bits que puede tener una clave $(p = \log_2 m)$
- Se realizan p ordenaciones por el **bit i-ésimo**
- Desde el bit más significativo al menos significativo.
- 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.

Algoritmo de Partición por bits

```
// Reorganiza v[ini..fin] en 2 zonas:
// v[ini..lim] elementos con el bit i-esimo = 0
// v[lim+1..fin] elementos con el bit i-esimo = 1
// La función devuelve el valor lim
static int particion_bit(tipo[] v, int ini, int fin, int i
    int izda = ini, dcha = fin;
    while(izda <= dcha) {
        while(izda <= fin && getBit(v[izda],i) == 0) { izda++; }
        while(dcha >= ini && getBit(v[dcha],i) == 1) { dcha--; }
        if(izda <= dcha) {
            v[izda] ⇔ v[dcha];
            izda++; dcha--;
        }
    }
    return dcha;
}
```

Ordenación Radixsort

```
// Ordena v[ini..fin] de acuerdo a los bits [i, i+1, ..]
static void radix_rec(tipo[] v, int ini, int fin, int i) {
    if(ini < fin && i < NUM_BITS) {
        int lim = particion_bit(v, ini, fin, i);
        radix_rec(v, ini, lim, i+1);
        radix_rec(v, lim+1, fin, i+1);
    }
}

static void radix(tipo[] v) {
    radix_rec(v, 0, v.length-1, 0);
}
```

Dado el tipo de datos de los elementos del array se debe definir una función que extraiga el bit *i*-ésimo de la clave

(0 debe ser el más significativo)

```
static int NUM_BITS = …
static int getBit(tipo n, int i) { … }
```

Propiedades (Radixsort)

- 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
Radix-Sort	O(n log m)	O(log m)	No

A. APÉNDICE: ALGORITMOS DE ORDENACIÓN EN HASKELL

Ordenación por Inserción

```haskell
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

```haskell
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)

```haskell
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

```haskell
fusion :: Ord a => [a] -> [a] -> [a]
fusion [] l2 = l2
fusion l1 [] = l1
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

```haskell
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]
```

Radixsort

```haskell
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]
```