UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escola Tècnica Superior d'Enginyeria Informàtica etsinf

Tema 2. Análisis de algoritmos. Eficiencia. Ordenación

Programación (PRG)

Departamento de Sistemas Informáticos y Computación

CC BY NC SA

Contenidos

1. Introducción al análisis de algoritmos. Conceptos
2. Los costes temporal y espacial de los programas
3. Complejidad asintótica
4. Análisis por casos
5. Análisis de algoritmos iterativos
6. Análisis de algoritmos recursivos
7. Análisis de algoritmos de ordenación
8. Otros algoritmos. Mezcla natural y búsqueda binaria

Introducción al análisis de algoritmos

- Es habitual disponer de más de un programa para resolver un mismo tipo de problema.
- Para decidir cuál de todos es el mejor es necesario disponer de un criterio objetivo ⇨ la eficiencia.
- El programa más eficiente es el que utiliza menos recursos al ejecutarse.
- Los recursos básicos de un ordenador son la memoria RAM y el procesador (CPU).

Introducción al análisis de algoritmos

- La eficiencia de los programas se expresa en términos de:

- El coste espacial. Medida del espacio de memoria que ocupa un programa durante su ejecución.

- El coste temporal. Medida del tiempo que necesita un programa para ejecutarse y dar un resultado a partir de los datos de entrada.

Introducción al análisis de algoritmos

- Los costes de un programa concreto dependen de dos tipos de factores:

- Factores propios del programa, como son la estrategia utilizada y los tipos de datos que emplea.

- Factores circunstanciales del entorno de programación donde se ejecuta, como son el tipo de computador, el lenguaje de programación, el compilador, la carga del sistema, etc.

Introducción al análisis de algoritmos

- El coste de un programa se puede estimar de dos formas:
- Análisis teórico o a priori
  - El coste se estima en función de los factores propios del programa.
  - Se trata de un análisis independiente del entorno de programación.
- Análisis experimental o a posteriori
  - El coste se estima midiendo en segundos el tiempo que tarda en ejecutarse un programa, y en bytes el espacio ocupado mientras se ejecutaba.
  - Es un análisis en un entorno de programación particular y para un conjunto determinado de datos de entrada (instancia).

Introducción al análisis de algoritmos

- Detalles a tener en cuenta:

1. Los dos tipos de análisis son importantes y necesarios, es decir, son complementarios.

2. Todo programador debe saber hacer un análisis teórico de sus programas, con el objeto de evitar perder el tiempo programando algoritmos que después ha de tirar.

3. La eficiencia es un criterio que todo programador debe tener presente cuando diseña programas.

4. Independientemente del entorno donde se ejecute, un programa debe ser eficiente.

Los costes temporal y espacial

- El coste temporal de un algoritmo se mide en base al tiempo de ejecución de sus operaciones elementales.
  - Algoritmo A1:

```
m = n*n;
```

  - Algoritmo A2:

```
m=0;
for (int i=0; i<n; i++) m+=n;
```

  - Algoritmo A3:

```
m=0;
for (int i=0; i<n; i++)
      for (int j=0; j<n; j++)
            m++;
```

Los costes temporal y espacial

- El coste temporal de un algoritmo se define como la suma de los costes de las operaciones elementales que implica:
  - Coste del algoritmo A1:

$$T_{A1} = t_a + t_{op}$$

  - Coste del algoritmo A2:

$$T_{A2} = t_a + t_a + (n+1)\, t_c + n\, t_a + 2\, n\, t_{op}$$

  - Coste del algoritmo A3:

$$T_{A3} = t_a + t_a + (n+1)\, t_c + n\, t_a + n\,(n+1)\, t_c + 2\, n^2\, t_{op} + n\, t_{op}$$

    - donde t_a es el coste de la operación de asignación, t_{op} es el coste de una operación aritmética (en este caso, suma o multiplicación) y t_c es el coste de una comparación.

- Comparar los costes de estos algoritmos con este análisis resulta bastante difícil y, además, requiere un esfuerzo de conteo considerable.

Los costes temporal y espacial

- El primer paso sería independizar los costes de los tiempos de las operaciones elementales, suponiendo constante el tiempo que necesita cada operación (todas las operaciones tardan una unidad de tiempo). Por tanto:
  - Coste del algoritmo A1: (constante, no depende de n)

    $T_{A1} = t_a + t_{op} \equiv k_1$
  - Coste del algoritmo A2: (dependencia lineal de n)

    $T_{A2} = t_a + t_a + (n + 1)\, t_c + n\, t_a + 2\, n\, t_{op} \equiv k_2\, n + k_3$
  - Coste del algoritmo A3: (dependencia cuadrática de n)

    $T_{A3} = t_a + t_a + (n + 1)\, t_c + n\, t_a + n\, (n + 1)\, t_c + 2\, n^2\, t_{op} + n\, t_{op} \equiv k_4\, n^2 + k_5\, n + k_6$

Los costes temporal y espacial

- El coste temporal (espacial) de un algoritmo se define como:
  - Una función no decreciente de la cantidad de tiempo (espacio) que necesita el algoritmo para ejecutarse en función del tamaño del problema.
- El tiempo necesario casi siempre depende de la cantidad de datos a procesar.
- La talla o tamaño de un problema se define como:
  - El valor o conjunto de valores asociados a los datos de entrada que representan una medida de la dificultad para su resolución.

Los costes temporal y espacial

- Ejemplos de elección del tamaño del problema

Problema	Tamaño
Búsqueda de un elemento en un conjunto	Número de elementos en el conjunto
Multiplicación de matrices	Dimensión de las matrices
Cálculo del factorial de un número	Valor del número
Resolución de un sistema de ecuaciones lineales	Número de ecuaciones y/o incógnitas
Ordenación de un array	Número de elementos en el array

Los costes temporal y espacial

- A partir de ahora, el coste temporal de un algoritmo A se expresará como: $T_A(talla)$.

- Luego, el coste del algoritmo A2 se expresará como $T_{A2}(n)$.

- Para independizar el análisis del tiempo de las operaciones elementales y de las diferentes entradas del problema, la función del coste temporal se define por conteo de pasos:
  - Coste del algoritmo A1: $T_{A1} = 1$ *paso*
  - Coste del algoritmo A2: $T_{A2} = n + 2$ *pasos*
  - Coste del algoritmo A3: $T_{A3} = n^2 + n + 2$ *pasos*

Los costes temporal y espacial

- Paso de programa:
  - Secuencia de operaciones básicas significativas con coste independiente de la talla del problema.
  - Un paso se puede considerar como una unidad de tiempo válida para expresar el coste de un algoritmo.
  - Así, el análisis de costes se independiza del tiempo de cada operación elemental.

Los costes temporal y espacial

- Ejemplos de elección de la operación básica:

Problema	Operación básica
Búsqueda de un elemento en un conjunto	Comparación entre el valor y los elementos del conjunto
Multiplicación de matrices	Producto de los elementos de las matrices
Cálculo del factorial de un número	Producto
Resolución de un sistema de ecuaciones lineales	Suma
Ordenación de un array	Comparación entre valores

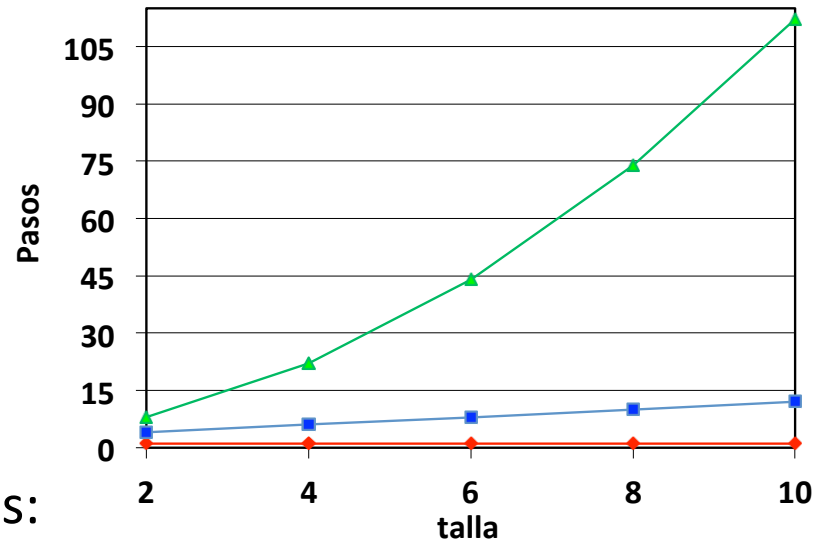
Complejidad asintótica

- El coste de un algoritmo se expresa como:
  - Una función T(n), que es una función no decreciente con respecto a la talla del problema.
- Comparar costes de algoritmos consiste en:
  - Comparar funciones no decrecientes, donde lo que interesa es su tasa de crecimiento.

$T_{A1}(n) = 1 \approx k$

$T_{A2}(n) = n + 2 \approx n$

$T_{A3}(n) = n^2 + n + 2 \approx n^2$



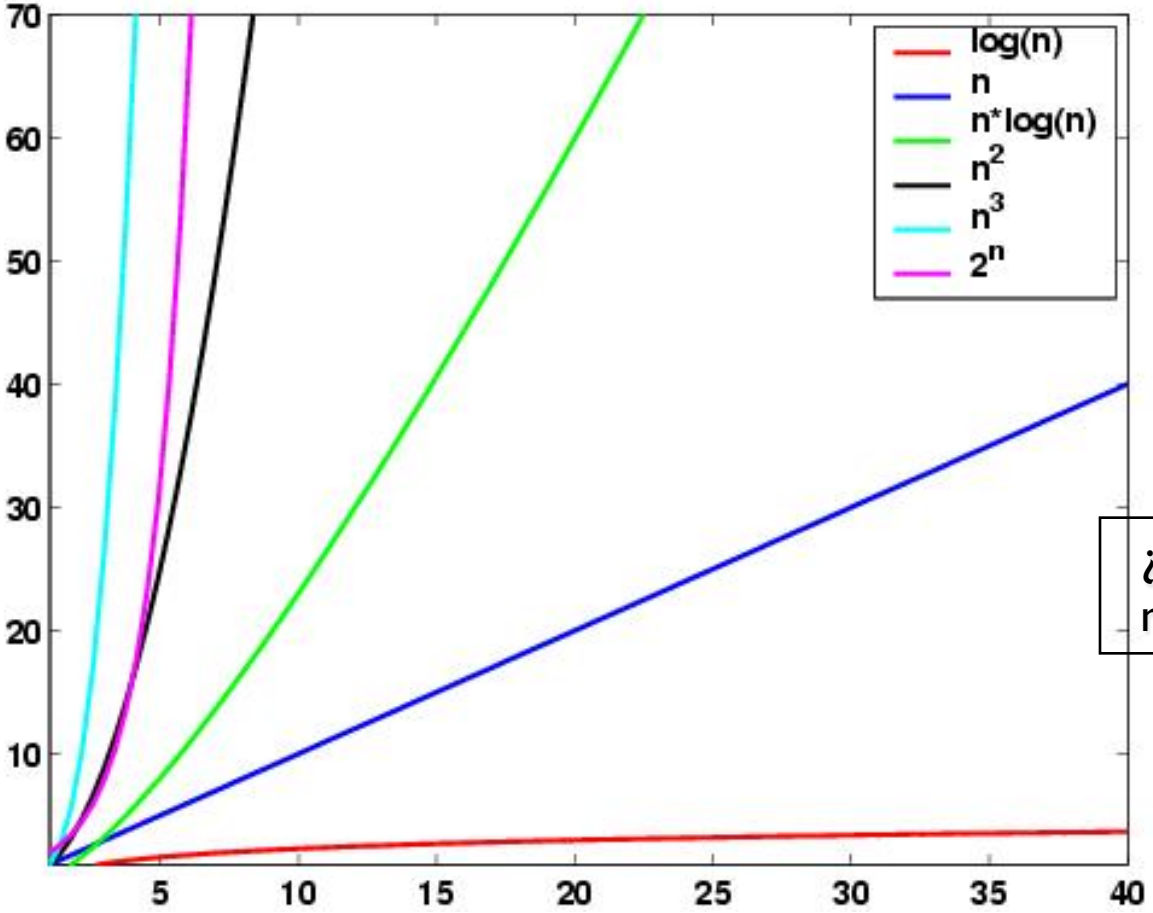
- Si T(n) es un polinomio, entonces:
  - El término de mayor grado del polinomio determina el aspecto de la curva de crecimiento.

Complejidad asintótica

- Tabla con funciones de coste típicas en orden creciente

Función	Nombre
c	constante
$\log n$	logarítmica
$\log^2 n$	logarítmica al cuadrado
n	lineal
$n \log n$	n log n
n^2	cuadrática
n^3	cúbica
2^n	exponencial

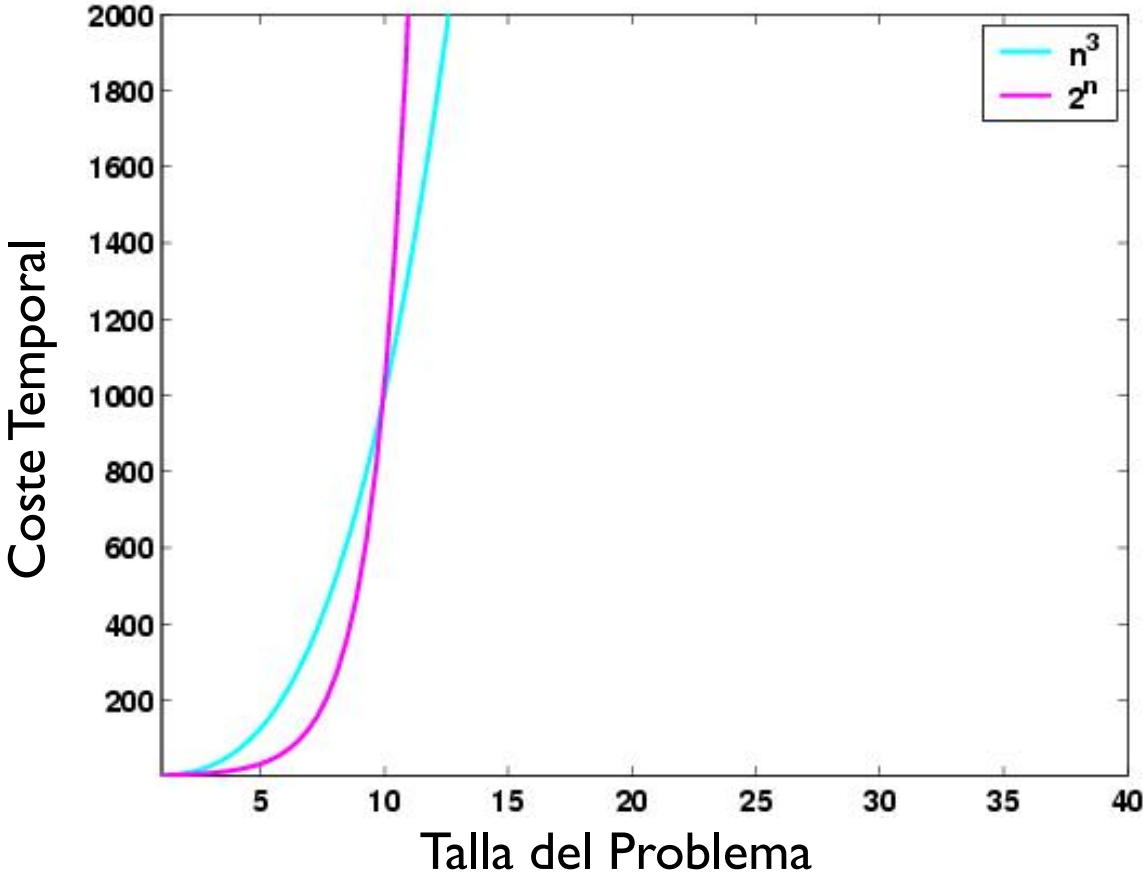
Complejidad asintótica



¿Qué función crece más rápida 2^n o n^3?

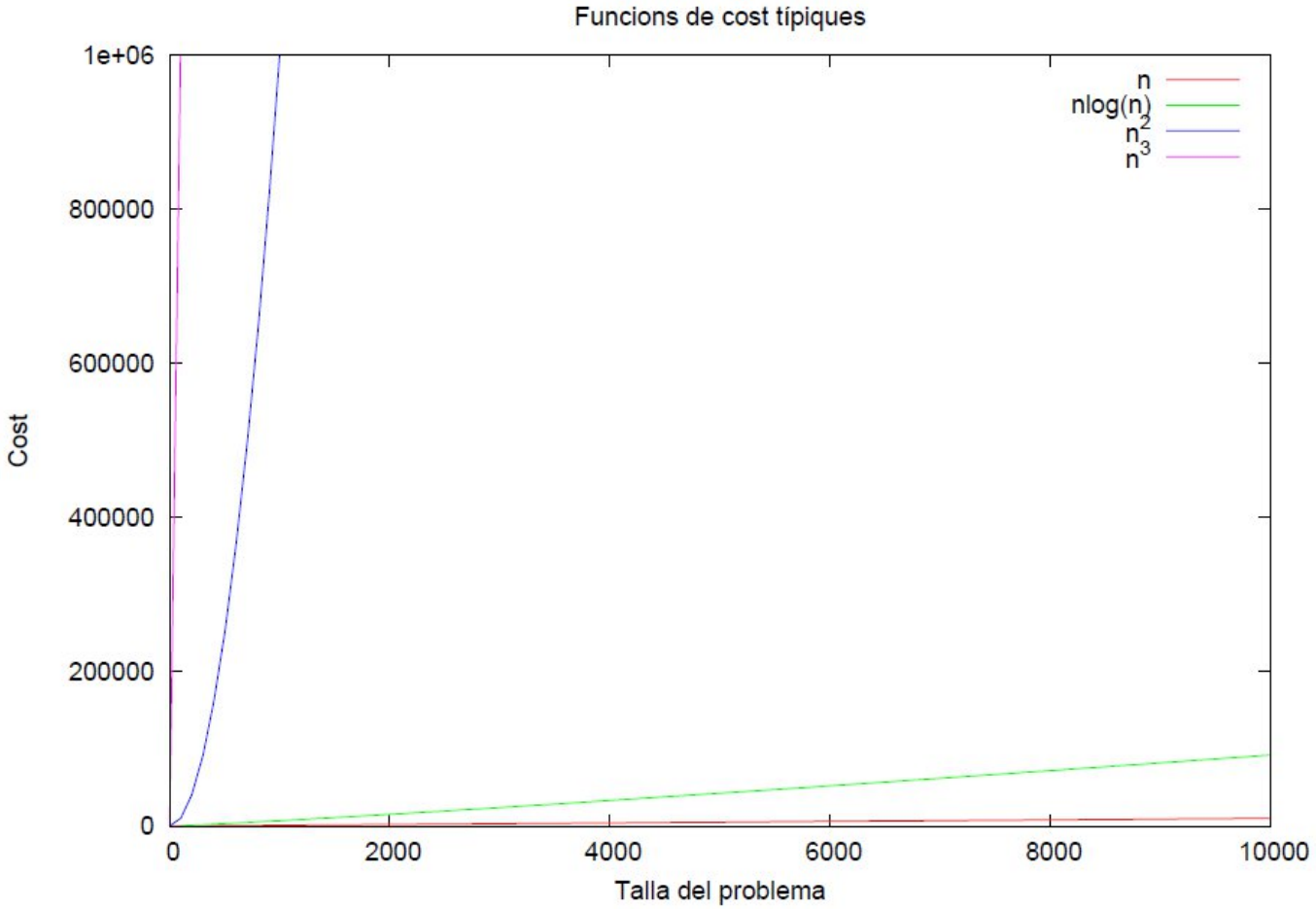
- Las funciones no lineales conducen a tiempos mayores que las funciones lineales.
- Los valores pequeños de n generalmente no son importantes. Para n=20, todos los algoritmos terminan antes de 5 seg. La diferencia entre el mejor y el peor algoritmo es menor que un parpadeo de ojos.

Cúbico vs Exponencial



- Para tamaños de problema suficientemente elevados es más costoso el exponencial que el cúbico.

Complejidad asintótica



- Para valores de n suficientemente grandes, el valor de la función está completamente determinado por el término dominante. Por ejemplo, en la función $10n^3+n^2+40n+80$, para n=1000, el valor es 10.001.040.080, del cual 10.000.000.000 es debido al término $10n^3$.

Complejidad asintótica

- **Ejemplo:** Supongamos que se está descargando un fichero de internet, de forma que hay un retraso inicial de 2 seg. para establecer la conexión y después la descarga se realiza a razón de 1.6 kbytes/seg. Si el tamaño del fichero es de N kbytes, el tiempo de descarga viene dado por: T(N)=N/1.6+2.
- La descarga de un fichero de 80 kbytes requerirá 52 seg., la descarga de un fichero el doble de grande requiere del orden de 102 seg., casi el doble.
- Un algoritmo lineal se caracteriza porque el tiempo de ejecución es proporcional a la entrada del problema.
- El coste lineal es un buen coste, el coste logarítmico es el mejor, y el coste n log n también es bueno.
- El resto de costes ya limitan considerablemente la talla de los problemas que podemos resolver.

Complejidad asintótica

- Uso de la notación asintótica

- Cuando se analiza el coste de un algoritmo, lo que nos interesa es el tipo de crecimiento que tiene.

- Si se identifican los tipos de crecimiento o funciones típicas, se puede comparar algoritmos, y así escoger.

- Por tanto, el análisis a priori de los algoritmos consistirá en medir el índice de crecimiento de las funciones de coste y expresarlo en notación asintótica.

Complejidad asintótica

- Sea $f : \mathbb{N} \rightarrow \mathbb{R}^{\geq 0}$. El conjunto de las funciones del orden de f(n), denotado por O(f(n)), se define como:

$$O(f(n)) = \{ g : \mathbb{N} \rightarrow \mathbb{R}^{\geq 0} \mid \exists\, c \in \mathbb{R}^{>0}, n_0 \in \mathbb{N} \text{ tales que, } \forall n \geq n_0, g(n) \leq c \cdot f(n) \}$$

- $g(n) \in O(f(n))$ ⇨ se dice que "g(n) es del orden de f(n)" y que "f(n) es asintóticamente una cota superior de g(n)".

- Sea $f : \mathbb{N} \rightarrow \mathbb{R}^{\geq 0}$. El conjunto de las funciones al menos del orden de f(n), denotado por $\Omega(f(n))$, se define como:

$$\Omega(f(n)) = \{ g : \mathbb{N} \rightarrow \mathbb{R}^{\geq 0} \mid \exists\, c \in \mathbb{R}^{>0}, n_0 \in \mathbb{N} \text{ tales que, } \forall n \geq n_0, g(n) \geq c \cdot f(n) \}$$

- $g(n) \in \Omega(f(n))$ ⇨ se dice que "g(n) es al menos del orden de f(n)" y que "f(n) es asintóticamente una cota inferior de g(n)".

Complejidad asintótica

- Sea $f: \mathbb{N} \rightarrow \mathbb{R}^{\geq 0}$. El conjunto de las funciones exactamente del orden de f(n), denotado por $\Theta(f(n))$, se define como:

$$\Theta(f(n)) = \{ g : \mathbb{N} \rightarrow \mathbb{R}^{\geq 0} \mid \exists\, c_1, c_2 \in \mathbb{R}^{>0} \text{ y } n_0 \in \mathbb{N} \text{ tales que, } \forall n \geq n_0,$$
$$0 \leq c_1 \cdot f(n) \leq g(n) \leq c_2 \cdot f(n) \}$$

- El conjunto de la funciones exactamente del orden de f(n) está formado por aquellas funciones que tienen como cota superior e inferior a la función f(n).

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

- $g(n) \in \Theta(f(n))$ ⇨ se dice que "g(n) es del orden exacto de f(n)" y que "f(n) es asintóticamente una cota superior e inferior de g(n)"

Complejidad asintótica

- Las razones que apoyan este tipo de análisis son:

1. Para valores de n grandes, el valor de la función está determinado por el término dominante.

2. El valor exacto del coeficiente del término dominante no se conserva al cambiar de entorno de programación.

3. El uso de la notación asintótica establece un orden relativo entre las funciones de coste comparando los términos dominantes:

$$O(\log n) \subset O(n) \subset O(n \log n) \subset O(n^2) \subset O(n^3) \subset O(2^n)$$

Complejidad asintótica

- Tabla de funciones típicas usando la notación asintótica:

Función	Nombre	Notación Asintótica
c	constante	$\Theta(1)$
log n	logarítmica	$\Theta(\log n)$
$\log^2 n$	logarítmica al cuadrado	$\Theta(\log^2 n)$
n	lineal	$\Theta(n)$
n log n	n log n	$\Theta(n \log n)$
n^2	cuadrática	$\Theta(n^2)$
n^3	cúbica	$\Theta(n^3)$
2^n	exponencial	$\Theta(2^n)$

Complejidad asintótica

- A medida que los ordenadores se vuelven más y más rápidos puede parecer que apenas merece la pena invertir nuestro tiempo en diseñar algoritmos más eficientes. Y ¿si esperamos a la siguiente generación de ordenadores?, ¿por qué hay que buscar la eficiencia?.

- Supongamos que para resolver un problema concreto se dispone de un algoritmo exponencial y de un ordenador que ejecuta dicho algoritmo para tamaño n en $10^{-4}x2^n$ segundos.

n	tiempo
10	$10^{-4}x2^{10}$ s., aprox. 1 décima de seg.
20	$10^{-4}x2^{20}$ s., aprox. 2 minutos
30	$10^{-4}x2^{30}$ s., más de 1 día de cálculo
¡38! ⇦	1 año

- Supongamos que se compra un ordenador nuevo cien veces más rápido que el anterior. Ahora se puede resolver el mismo problema para tamaño n en $10^{-6}x2^n$ segundos. En 1 año ⇨ n < ¡45!

- En general, si antes se resolvía un ejemplar de tamaño n en un tiempo dado, la nueva máquina resolverá tamaños como mucho de n+log 100, aprox. n+7, en el mismo tiempo.

Complejidad asintótica

- Supongamos que se invierte en algoritmia y, por la misma cantidad de dinero, se encuentra un algoritmo cúbico que en la máquina original resuelve un ejemplar de tamaño n en $10^{-2}xn^3$ s.

n		tiempo
10		$10^{-2}x10^3$ = 10 seg.
20		$10^{-2}x20^3$ = entre 1 y 2 minutos
30		$10^{-2}x30^3$ = 4 minutos y medio
>200		1 día
¡1500!	⇦	1 año

- El nuevo algoritmo no sólo ofrece una mejora mucho mayor que la adquisición del nuevo hardware, sino que además, suponiendo que pueda uno permitirse ambas cosas, hará que esta adquisición sea más rentable.

- En general, las mejoras en un programa por cambio de lenguaje, de computador, ahorro de variables y/o de instrucciones, etc. son menos importantes que las mejoras en la estrategia que impliquen una tasa de crecimiento inferior.

Análisis por casos

- El coste de un algoritmo es una función no decreciente de la talla del problema.

- Para un tamaño en particular, el coste del algoritmo también depende de los datos de entrada (una instancia del problema).

- Una instancia de un problema representa todas las configuraciones diferentes de la entrada, de una talla determinada, para las cuales el coste del algoritmo es similar, es decir, el comportamiento del algoritmo es el mismo en cuanto a costes.

Análisis por casos

- Si en el análisis de los costes de un algoritmo se detectan instancias que causan variaciones en el comportamiento, entonces se tipifican los siguientes casos:
  - Coste del algoritmo en el peor caso: es la complejidad del mismo para la instancia del problema que presente el coste mayor. Se denota por $T^p(n)$
  - Coste del algoritmo en el mejor caso: es la complejidad del mismo para la instancia del problema que presente el coste menor. Se denota por $T^m(n)$
  - Coste del algoritmo en término medio: es la media de los costes de todas las instancias del problema. Se denota por $T^\mu(n)$
- A la hora de analizar un algoritmo, interesan sus comportamientos en los casos peor y término medio.

Análisis por casos

- Algoritmos de recorrido y búsqueda con arrays
- El problema de recorrido de un array sólo presenta una instancia, por tanto, no se puede distinguir entre casos mejor y peor.
- El problema de búsqueda sí presenta varias instancias:
  - el elemento a buscar se encuentra en la posición 0
  - el elemento a buscar se encuentra en la posición 1
  - . . .
  - el elemento a buscar se encuentra en la posición n-1
  - el elemento a buscar no se encuentra

  Con éxito

  Sin éxito

  n+1 instancias
- Si el algoritmo realiza la búsqueda secuencial desde el principio (posición 0 del array):
  - mejor caso: el elemento a buscar se encuentra en la primera posición.
  - peor caso: el elemento no se encuentra o es el último.

Análisis por casos

- Regla general:

1. Determinar la talla del problema, es decir, estudiar de qué parámetros depende el coste.
2. Analizar si, para una talla concreta, existen instancias significativas en cuanto al coste. Es decir, se observan comportamientos diferentes del algoritmo.
3. Obtener la función de coste.
   - Si no hay instancias significativas, se utiliza la notación Θ.
   - Si existen instancias significativas, el estudio se hará para los casos peor y mejor. El estudio sobre el caso peor da la cota superior del coste del algoritmo y sobre el caso mejor da la cota inferior del coste del algoritmo.
   - Para la cota superior se utiliza la notación O.
   - Para la cota inferior se utiliza la notación Ω.

Análisis de algoritmos iterativos

- El primer paso para obtener la función de coste de un algoritmo iterativo consiste en detectar una o más instrucciones críticas.
- Instrucción crítica (o barómetro):
  - Instrucción que se repite tantas veces como cualquier otra en el bucle o los bucles del algoritmo a estudiar.
  - Debe cumplir una premisa: ser independiente de los datos y del tipo de problema ⇨ normalmente será una instrucción básica.
- Así, se puede expresar el coste en función de las veces que se ejecuta una instrucción crítica ⇨ notación asintótica.

Análisis de algoritmos iterativos

- Eficiencia de un algoritmo de recorrido
- Sea v un array de n elementos que se puede recorrer mediante un algoritmo como el siguiente:

```
for (int i=0; i<n; i++)
    tratar(v[i]);
```

- Se supone que `tratar(e)` es una operación de coste constante sobre el elemento `e`.
- Se puede decir que el coste del algoritmo está en función del número de elementos del array T(n).
- El algoritmo no presenta instancias significativas, el coste se aproxima contando el número de veces que se repite la instrucción `tratar(v[i])`.
- Como instrucciones críticas se pueden considerar:

`tratar(v[i])` `i++` `i<n`

- El coste del algoritmo es: $T(n) = n \in \Theta(n)$

Análisis de algoritmos iterativos

- Eficiencia de un algoritmo de búsqueda secuencial
- Sea v un array de n elementos en el que se busca la posición de un elemento que cumpla una determinada propiedad mediante el siguiente algoritmo:

```
int i=0;
while ( (i<n) && !(PROPIEDAD(v[i])) ) i++;
if ( i<n ) return i;
else return -1;
```

- **`PROPIEDAD(e)`** es la propiedad que debe cumplir el elemento que buscamos y tiene un coste constante.
- El coste del algoritmo es función del número de elementos del array: T(n).

Análisis de algoritmos iterativos

- Eficiencia de un algoritmo de búsqueda secuencial
- El coste del algoritmo depende de la instancia del problema.
- Para este algoritmo hay n + 1 instancias significativas.
- Consideramos como instrucción crítica la guarda del bucle:

```
(i<n) && !(PROPIEDAD(v[i]))
```

  - Caso peor: $T^p(n) = n + 1 \in \Theta(n)$ LINEAL
  - Caso mejor: $T^m(n) = 1 \in \Theta(1)$ CONSTANTE

  - Cota superior: $T(n) \in O(n)$
  - Cota inferior: $T(n) \in \Omega(1)$

Análisis de algoritmos iterativos

- Eficiencia de un algoritmo de búsqueda secuencial
- Para poder estimar el coste en término medio se necesita conocer:
  - La distribución de probabilidad de las diferentes instancias.
- Se consideran dos situaciones:
  1. La búsqueda siempre tiene éxito, y la probabilidad de que el elemento a buscar se encuentre en cualquier posición del array es la misma.
  2. Es equiprobable que el elemento se encuentre o no en el array y, en caso de encontrarse, todas las posiciones son equiprobables.

Análisis de algoritmos iterativos

- Primer supuesto
- Hay n instancias posibles, con probabilidad 1/n cada una.
- Coste de cada instancia: número de veces que se ejecute el bucle, que coincide con la posición donde se encuentra el elemento i.

$$T^{\mu}(n) = \sum_{i=1}^{n} \frac{1}{n} i = \frac{n(n+1)}{2n} \in \Theta(n)$$

Análisis de algoritmos iterativos

- Segundo supuesto
- Hay n+1 instancias posibles:
  - Que el elemento no esté, con probabilidad 1/2.
  - Que el elemento se encuentre en cualquier posición del array, con probabilidad 1/2n.
- El coste de la primera instancia es n+1, el coste de las restantes instancias es i.

$$T^{\mu}(n) = \frac{n+1}{2} + \sum_{i=1}^{n} \frac{1}{2n} i = \frac{n+1}{2} + \frac{n(n+1)}{4n} \in \Theta(n)$$

Análisis de algoritmos recursivos

- En la complejidad temporal de un algoritmo recursivo influyen:
  - El número de llamadas recursivas que genera cada llamada al método.
  - La forma en que se reduce el tamaño del problema n en cada llamada. Normalmente, la reducción es de la forma:
    - n – c (siendo c una constante tal que c ≥ 1) o
    - n/c (siendo c una constante tal que c > 1).
  - El coste del resto de operaciones que realiza el algoritmo excluida(s) la(s) llamada(s) recursiva(s).

Análisis de algoritmos recursivos

- Para analizar la complejidad de los algoritmos recursivos se usan las ecuaciones de recurrencia.
- Las ecuaciones de recurrencia permiten indicar el tiempo de ejecución para los distintos casos de un algoritmo recursivo (casos base y regla general).
- Una vez se dispone de la ecuación de recurrencia es posible calcular el orden del tiempo de ejecución de diversas formas.
- Un modo de resolver estas ecuaciones: la técnica de despliegue de recurrencias o sustitución.

Análisis de algoritmos recursivos

- **Ejemplo**: Cálculo del factorial de un número entero

```
/** n>=0 */
int factorial (int n) {
   if ( n==0 ) return 1;
   else return n * factorial(n-1);
}
```

  - Tamaño de los datos: dado por la variable n.
  - Caso base (n=0). Tiempo de ejecución: T(n) = 1+1 (comparación y retorno tienen un tiempo de ejecución constante, simplificando: coste unitario, 1).
  - Caso recursivo (n>0). Tiempo de ejecución: T(n) = 4+T(n-1) (considerando 4 operaciones de coste constante: la comparación, la expresión `n-1`, el producto y el retorno; más el tiempo de cálculo de `factorial(n-1)`).
  - La ecuación de recurrencia:

$$T(n) = \begin{cases} 2 & n = 0 \\ 4 + T(n-1) & n > 0 \end{cases}$$

Análisis de algoritmos recursivos

- La técnica de despliegue de recurrencias consiste en:
  - Sustituir las apariciones de T dentro de la ecuación recursiva hasta encontrar una forma general que dependa del número de invocaciones recursivas, k.
- En el caso del cálculo del **`factorial`**: $$T(n) = \begin{cases} 2 & n = 0 \\ 4 + T(n-1) & n > 0 \end{cases}$$

$T(n) = 4 + T(n-1) = 4 + (4 + T(n-2)) = 4 + (4 + (4 + T(n-3))) = \ldots = 4k + T(n-k)$

donde k es el número de invocaciones recursivas.

- Además, se debe verificar cómo se cumple para el caso base:

  $n-k=0 \Leftrightarrow n=k,\ T(n) = 4n + T(0) = 4n + 2$

- Si $T(n) = 4n+2$, la complejidad es exactamente del orden $\Theta(n)$.

Análisis de algoritmos de ordenación

- Basados en comparaciones
  - Algoritmos directos
    - Inserción Directa
    - Selección Directa
    - Intercambio/Burbuja
  - Algoritmos rápidos
    - Mezclas ("MergeSort")
    - Partición ("QuickSort")
    - Montículos ("HeapSort")
- No basados en comparaciones
  - Conteo o apartados ("CountingSort")
  - Residuos ("RadixSort")
  - Cubetas ("BucketSort")

http://www.sorting-algorithms.com/

Algoritmo de selección directa

- El algoritmo de ordenación por selección consiste en:
  - Seleccionar el mínimo elemento del array e intercambiarlo con el primero.
  - Seleccionar el mínimo en el resto del array e intercambiarlo con el segundo.
  - Y así sucesivamente...

- Algoritmo de selección directa (*Selection Sort*):
  http://www.youtube.com/watch?v=boOwArDShLU

Algoritmo de selección directa

- La estrategia de ordenación por selección directa se puede sintetizar en:
  - Para todo `i` desde `0` hasta `n-2` hacer:
  1. Encontrar la posición del mínimo en el subarray que va de `i` a `n-1`.
  2. Intercambiar el mínimo con `v[i]`.

  - Los elementos del subarray `v[0..i-1]` están ordenados entre sí y además tienen valores inferiores a los del subarray `v[i..n-1]`.

- La operación de encontrar el mínimo en el subarray es un recorrido.

Algoritmo de selección directa

- **Ejemplo:** ordenar el array {16, 54, 7, 98, 2, 66, 30, 14}

  {2, 54, 7, 98, 16, 66, 30, 14} ← selecciona 2 e intercambia con 16

  {2, 7, 54, 98, 16, 66, 30, 14} ← selecciona 7 e intercambia con 54

  {2, 7, 14, 98, 16, 66, 30, 54} ← selecciona 14 e intercambia con 54

  {2, 7, 14, 16, 98, 66, 30, 54} ← selecciona 16 e intercambia con 98

  {2, 7, 14, 16, 30, 66, 98, 54} ← selecciona 30 e intercambia con 98

  {2, 7, 14, 16, 30, 54, 98, 66} ← selecciona 54 e intercambia con 66

  {2, 7, 14, 16, 30, 54, 66, 98} ← selecciona 66 e intercambia con 98

Algoritmo de selección directa

- Implementación

```
static void selDirecta(int v[]){
   for (int i=0; i<v.length-1; i++) {
  // calcular la posición del mínimo de v[i:v.length-1]
      int pMin = i;
      for(int j=i+1; j<v.length; j++)
         if (v[j]<v[pMin]) pMin = j;
            // en pMin está la posición
            // del mínimo de v[i:v.length-1]
      // intercambiar v[i] con v[pMin]
      int aux = v[pMin];
      v[pMin] = v[i];
      v[i] = aux;          // desde 0 a i están ordenados
   }
          // array ordenado desde 0 hasta v.length-1
}
```

Algoritmo de selección directa

- Costes
- La talla del problema es el número de elementos a ordenar, n.
- La estructura se basa en dos recorridos anidados, y el coste no presenta variación frente a diferentes instancias del problema.

$$T(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} (n-i-1) = \frac{n(n-1)}{2} \in \Theta(n^2)$$

Algoritmo de inserción directa

- El algoritmo divide el array en una parte ordenada y otra no ordenada:
  - Inicialmente, la parte ordenada consta de un único elemento (el que ocupa la primera posición).
  - Los elementos son insertados uno a uno desde la parte no ordenada a la ordenada.
    - Seleccionar el elemento `v[1]` del array y situarlo de manera ordenada con el que ocupa la posición `0`.
    - Seleccionar el elemento `v[2]` del array y situarlo de manera ordenada entre los que ocupan las posiciones `0` y `1`.
    - Repetir situando el elemento `v[i]` de manera ordenada en el subarray comprendido entre las posiciones `0` e `i-1`.
  - Finalmente, la parte ordenada abarca todo el vector.
- Algoritmo de inserción directa (*Insertion Sort*)
  http://www.youtube.com/watch?v=gTxFxgvZmQs&feature=related

Algoritmo de inserción directa

- La estrategia de ordenación por inserción directa se puede describir como sigue:
  - Para todo `i` desde `1` hasta `n-1` hacer:

  1. Insertar el elemento `v[i]` de manera ordenada en el subarray `v[0..i-1]`:

     1. Se busca secuencialmente en `v[0..i-1]` el primer elemento menor o igual a `v[i]`. Sea `j` la posición de dicho elemento (o `-1` si no se encuentra).
     2. Se desplazan una posición hacia la derecha todos los elementos desde `j+1` hasta `i-1`.
     3. Se asigna el que había en `v[i]` a `v[j+1]`.

     - Los pasos 1 y 2 se pueden hacer de manera combinada.

Algoritmo de inserción directa

- **Ejemplo:** ordenar el array {16, 7, 54, 98, 2, 66, 30, 14}

{**16,** 7, 54, 98, 2, 66, 30, 14} ← inicialmente, [0..0] está ordenado

{**7, 16,** 54, 98, 2, 66, 30, 14} ← ordenados [0..1], inserta 7

{**7, 16, 54,** 98, 2, 66, 30, 14} ← ordenados [0..2], sin inserciones

{**7, 16, 54, 98,** 2, 66, 30, 14} ← ordenados [0..3], sin inserciones

{**2, 7, 16, 54, 98,** 66, 30, 14} ← ordenados [0..4], inserta 2

{**2, 7, 16, 54, 66, 98,** 30, 14} ← ordenados [0..5], inserta 66

{**2, 7, 16, 30, 54, 66, 98,** 14} ← ordenados [0..6], inserta 30

{**2, 7, 14, 16, 30, 54, 66, 98**} ← ordenados [0..7], inserta 14

Algoritmo de inserción directa

- Implementación

```
static void insDirecta(int v[]){
   for(int i=1; i<=v.length-1; i++) {
      int x = v[i]; // elemento a insertar
      int j = i-1;  // inicio de la parte ordenada
    // buscar en la parte ordenada,
    // desplazando a derecha los elementos mayores que x
      while( j>=0 && v[j]>x ) {
         v[j+1] = v[j];
         j--;
      }
      v[j+1] = x; // asignar x en la parte ordenada
   }
}
```

Algoritmo de inserción directa

- Costes
- La talla del problema es n, el número de elementos a ordenar.
- La estructura del algoritmo es un recorrido y una búsqueda combinadas.
- El bucle interno, el que hace la búsqueda, no siempre se repite completamente (cuando encuentra la posición correcta se detiene). Es decir, se ahorran algunos pasos.
- En este caso se puede escoger como instrucción crítica la comparación (`v[j]>x`).

Algoritmo de inserción directa

- Costes

- Caso mejor: Cuando el array está ordenado el bucle interno no itera ninguna vez, la comparación siempre se evalúa a falso. En este caso, el algoritmo ejecuta tantos pasos como el bucle externo.

$$T^m(n) = \sum_{i=1}^{n-1} 1 = n-1 \in \Theta(n) \Rightarrow T(n) \in \Omega(n)$$

- Caso peor: El número de veces que se ejecuta el bucle interno es el máximo posible, es decir, `i` veces en cada iteración del bucle externo. Se trata del caso en el que el array está ordenado al revés de como se quiere ordenar.

$$T^p(n) = \sum_{i=1}^{n-1} i \in \Theta(n^2) \Rightarrow T(n) \in \mathrm{O}(n^2)$$

Algoritmo de intercambio directo o burbuja

- Algoritmo de ordenación sencillo e ineficiente. Consistente en:
  - Recorrer el array comparando pares de elementos consecutivos.
  - Intercambiándolos si no están en el orden correcto.
- Algoritmo de intercambio directo (*Bubble Sort*)

  http://www.youtube.com/watch?v=1JvYAXT_064&feature=related

Algoritmo de intercambio directo o burbuja

- **Ejemplo**: ordenar el array {16, 54, 7, 98, 2, 66, 30, 14}
- **Primera iteración:**

  {16,54,7,98,2,66,30,14} → {16,54,7,98,2,66,30,14} (sin intercambio)
  {16,54,7,98,2,66,30,14} → {16,7,54,98,2,66,30,14} (intercambio 7 - 54)
  {16,7,54,98,2,66,30,14} → {16,7,54,98,2,66,30,14} (sin intercambio)
  {16,7,54,98,2,66,30,14} → {16,7,54,2,98,66,30,14} (intercambio 2 - 98)
  {16,7,54,2,98,66,30,14} → {16,7,54,2,66,98,30,14} (intercambio 66 - 98)
  {16,7,54,2,66,98,30,14} → {16,7,54,2,66,30,98,14} (intercambio 30 - 98)
  {16,7,54,2,66,30,98,14} → {16,7,54,2,66,30,14,98} (intercambio 14 - 98)

- **Segunda iteración:**

  {16,7,54,2,66,30,14,98} → {7,16,54,2,66,30,14,98} (intercambio 7 - 16)
  {7,16,54,2,66,30,14,98} → {7,16,54,2,66,30,14,98} (sin intercambio)
  {7,16,54,2,66,30,14,98} → {7,16,2,54,66,30,14,98} (intercambio 2 - 54)
  {7,16,2,54,66,30,14,98} → {7,16,2,54,66,30,14,98} (sin intercambio)
  {7,16,2,54,66,30,14,98} → {7,16,2,54,30,66,14,98} (intercambio 30 - 66)
  {7,16,2,54,30,66,14,98} → {7,16,2,54,30,14,66,98} (intercambio 14 – 66)

Algoritmo de intercambio directo o burbuja

- Implementación

```
static void burbuja (int[] v) {
    for (int i=1; i<=v.length-1; i++)
        for (int j=0; j<v.length-i; j++)
            // comparar pares de elementos consecutivos
            if ( v[j] > v[j+1] ) {
            // si par desordenado, entonces intercambio
                int x = v[j];
                v[j] = v[j+1];
                v[j+1] = x;
            }
}
```

Algoritmo de intercambio directo o burbuja

- Costes
- Los dos bucles se ejecutan `n-1` veces:

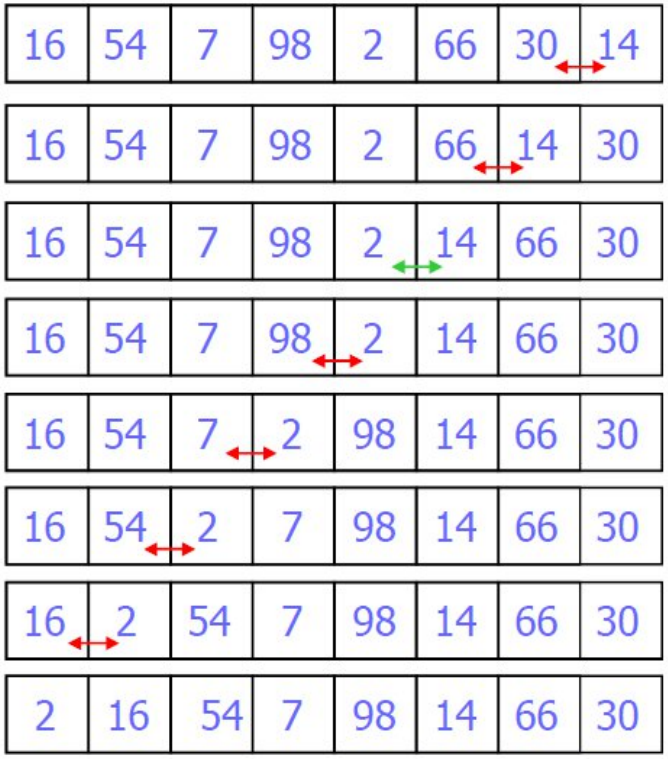
$$T(n) = \sum_{i=1}^{n-1} \sum_{j=0}^{n-i-1} 1 = \sum_{i=1}^{n-1} (n-i) = n(n-1) - [n(\frac{n+1}{2}) - n)] \in \Theta(n^2)$$

- Una mejora consiste en añadir una bandera o flag que indique si se ha producido algún intercambio durante el recorrido:
  - Si no se ha producido ninguno, el array se encuentra ordenado y se puede acabar.
  - Con esta mejora su coste sigue siendo cuadrático.

Algoritmo de intercambio directo o burbuja

- Implementación (2)

```
static void burbuja(int[] v){
   int x;
   for(int i=0;i<v.length-1;i++)
    for(int j=v.length-1;j>i;j--)
       if ( v[j-1] > v[j] ) {
          x = v[j-1];
          v[j-1] = v[j];
          v[j] = x;
      }
}
```



Algoritmo de mezcla natural

- Problema: dados 2 arrays a y b con número diferente de elementos pero ordenados, fusionarlos en un array nuevo c que quede ordenado.

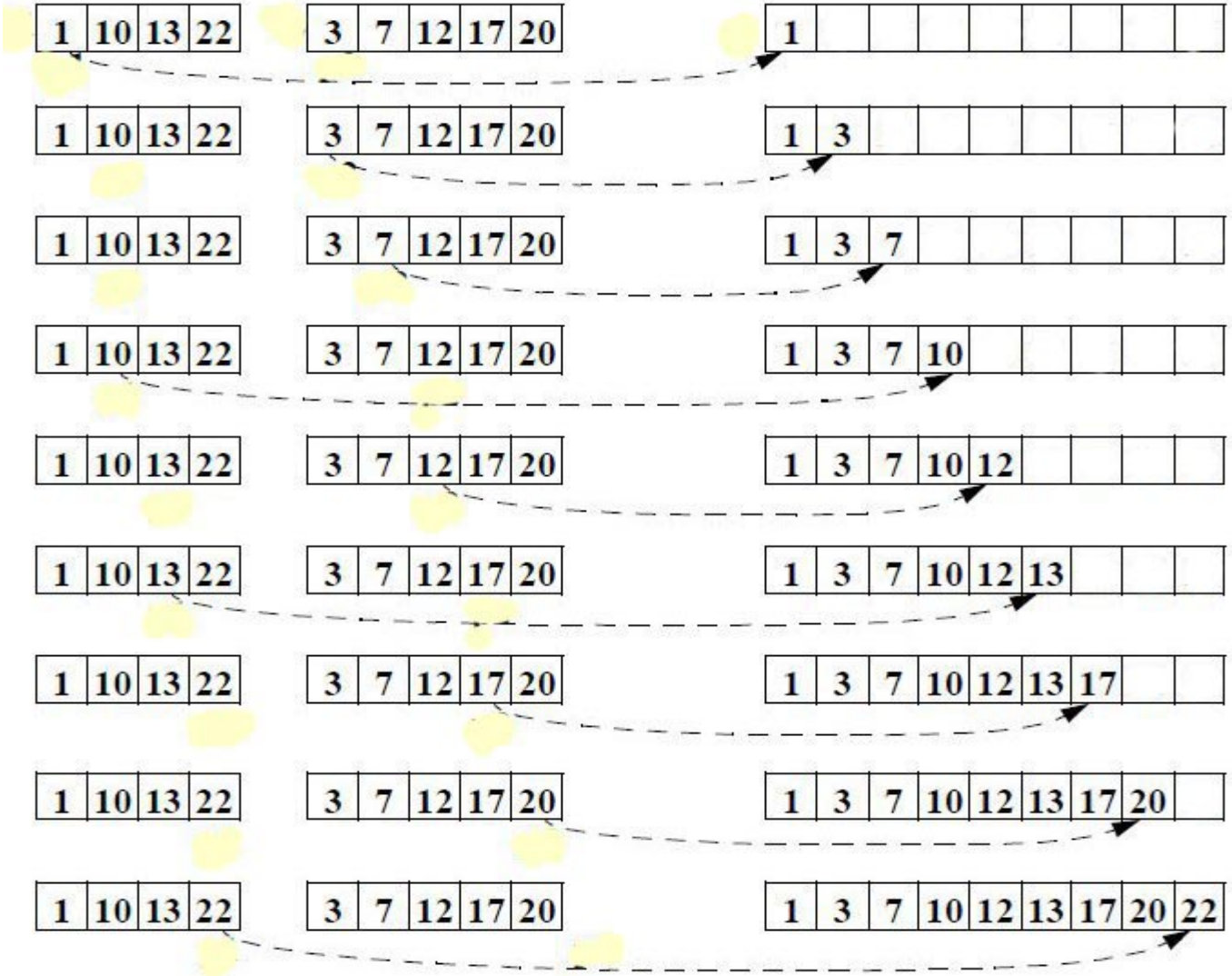
- El algoritmo que resuelve este problema en dos fases es el siguiente:

  1. Un bucle que compara los elementos de los 2 arrays y los copia de manera ordenada en el array destino. Este bucle acaba cuando se alcanza el final de uno de los arrays.

  2. Un bucle que copia, sin comparar nada, los restantes elementos de uno de los arrays al final del array destino.

Algoritmo de mezcla natural

- Ejemplo de ejecución.



Algoritmo de mezcla natural

- Implementación

```
/** a y b están ordenados
  * c.length es a.length + b.length
  */
static void mezclaNatural(int a[], int b[], int[] c){
   int i=0, l=a.length, j=0, m=b.length, k=0;

   while ( i < l && j < m ) {
        if (a[i] < b[j]) { c[k] = a[i]; i++; }
        else { c[k] = b[j]; j++; }
        k++;
   }

   for (int r=i; r < l; r++)  { c[k] = a[r]; k++; }
   for (int r=j; r < m; r++)  { c[k] = b[r]; k++; }
}
```

Algoritmo de mezcla natural

- El tamaño **`n`** del problema viene determinado por la suma de los tamaños de los arrays a mezclar.

- Para contar los pasos del algoritmo, se pueden contar las veces que se ejecuta el incremento de la variable índice de acceso al array destino.

- Entonces, se ejecuta **`l+m`** veces, donde **`l`** es **`a.length`** y **`m`** es **`b.length`**.

$$T(l,m) = l+m \in \Theta(l+m)$$

Es decir, $T(n) \in \Theta(n)$

Algoritmo MergeSort

- Algoritmo de ordenación que consiste en:
  - Dividir el array en dos partes iguales.
  - Ordenar por separado cada una de las partes (mediante llamadas recursivas).
  - Mezclar ambas partes, manteniendo la ordenación.

- Algoritmo de mezcla directa (MergeSort)

  http://www.youtube.com/watch?v=HA6ghMlYuO4

Algoritmo MergeSort

- Implementación

```
static void mergesort (int[] v, int ini, int fin) {
    if ( ini<fin ) {
        int mitad = (fin+ini)/2;
        mergesort(v, ini, mitad);
        mergesort(v, mitad+1, fin);
    // versión de mezcla natural especializada:
    // mezcla en v[ini..fin] de
    // v[ini..mitad] y v[mitad+1..fin]
        mezclaNatural2(v, ini, mitad, fin);
    }
}
```

Algoritmo MergeSort

- Costes
- Coste de **`mezclaNatural2`**: Θ(n)

- Coste de **`mergesort`**:

$$T(n) = \begin{cases} c_1 & 0 \le n \le 1 \\ 2T(n/2) + c_2 n & n > 1 \end{cases}$$

Algoritmo MergeSort

- Despliegue de recurrencias

$$T(n) = \begin{cases} c_1 & 0 \le n \le 1 \\ 2T(n/2) + c_2 n & n > 1 \end{cases}$$

$$T(n) \le 2T\left(\frac{n}{2}\right) + c_2 n \qquad T\left(\frac{n}{2}\right) \le 2T\left(\frac{n}{4}\right) + c_2 \frac{n}{2}$$

$$T(n) \le 2\left[2T\left(\frac{n}{4}\right) + c_2 \frac{n}{2}\right] + c_2 n = 4T\left(\frac{n}{4}\right) + 2c_2 n$$

$$T(n) \le 4\left[2T\left(\frac{n}{8}\right) + c_2 \frac{n}{4}\right] + 2c_2 n = 8T\left(\frac{n}{8}\right) + 3c_2 n$$

Algoritmo MergeSort

- Despliegue de recurrencias
- Si $n >= 2^i$, se tiene que:

$$T(n) \leq 2^i T\left(\frac{n}{2^i}\right) + ic_2 n$$

- Si $n = 2^k$, se obtiene $T(1)$ en la parte derecha:

$$T(n) \leq 2^k T(1) + kc_2 n$$

- Si $n = 2^k \Leftrightarrow k = \log_2 n$, y como $T(1) \leq c_1$:

$$T(n) \leq c_1 n + c_2 n \log n \Rightarrow T(n) \in \Theta(n \log n)$$

Otros algoritmos. Búsqueda binaria

- Hay muchas situaciones donde es necesario hacer repetidas búsquedas de elementos dentro de una colección.
- Si la colección de datos no está ordenada, debe realizarse una búsqueda exhaustiva (es decir, comenzar por un extremo y buscar el elemento hacia el otro extremo, bien hasta que se encuentra, o en caso de que no esté, hasta que se llega al final) ⇨ Coste lineal
- Si la colección de datos está ordenada, se dispone de una estrategia muy eficiente: la búsqueda dicotómica ⇨ Coste logarítmico

Otros algoritmos. Búsqueda binaria

- Estrategia del algoritmo:
- Se considera que se busca en un intervalo del array `v`, por ejemplo desde la posición `i` hasta la `j`, `v[i..j]`. Inicialmente, `v[0..n-1]`, donde `n` representa el número de elementos del array (`v.length`).
- Consiste en fijarse en la posición central: `m=(i+j)/2`, y decidir según los tres casos posibles:
  - `x = v[m]` ⇨ Acabar, ya que se ha encontrado `x` en `v`
  - `x < v[m]` ⇨ Buscar en el subarray `v[i..m-1]`
  - `x > v[m]` ⇨ Buscar en el subarray `v[m+1..j]`

Otros algoritmos. Búsqueda binaria

- Implementación

```
  /** El array está ordenado ascendentemente */
static int encBinIter(int[] v, int x){
      int i=0, j=v.length-1, mitad=0;
      boolean encontrado=false;
      while(i<=j && !encontrado){
            mitad = (i+j)/2;
            if (x==v[mitad]) encontrado = true;
            else if (x<v[mitad]) j = mitad-1;
                  else i = mitad+1;
      }
      if (encontrado) return mitad;
      else return -1;
}
```

Otros algoritmos. Búsqueda binaria

- Costes
- La función de coste depende del tamaño del array.
- Se distinguen las siguientes instancias significativas:
  - Caso mejor: El elemento a buscar está donde se realiza la primera comparación (en la posición central). En este caso, el bucle se ejecuta sólo una vez. El coste es constante.

$$T^m(n) \in \Theta(1) \Rightarrow T(n) \in \Omega(1)$$

  - Caso peor: El elemento no se encuentra en el array. Como cada vez se parte por la mitad el intervalo donde buscarlo, el número de pasos o veces que se repite el bucle es $\log_2(n)$.

$$T^p(n) = \log_2(n) + 1 \in \Theta(\log n) \Rightarrow T(n) \in O(\log n)$$