

UNIVERSIDAD NACIONAL DE INGENIERÍA

Análisis y Diseño de Algoritmos

Recurrencias y mergesort

Del código al costo, y contar inversiones

Gerson Garrido

Semana 2

Hoy

1. Del código a la recurrencia, y el árbol de recursión
2. El teorema maestro de tres casos
3. Cuándo no aplica, y cómo comprobar por inducción
4. El costo de copiar
5. Mergesort demostrado: invariante e inducción
6. Contar inversiones durante la mezcla

> **Lectura**
>
> Apuntes de la semana 2. Apoyo: Erickson, secs. 1.4, 1.6 y 1.7; CLRS, capítulos *Divide-and-Conquer* y *Getting Started*. Código en `code/week-02`.

SECCIÓN 01

Recurrencias

01



Divide y vencerás *(divide and conquer)*

1. **Dividir:** partir la entrada en subproblemas más pequeños, del mismo tipo.
2. **Resolver:** una llamada recursiva por cada uno. Si es muy pequeño, el caso base.
3. **Combinar:** armar la respuesta con las de las partes.

Recurrencia *(recurrence)*

Ecuación que da el costo de un algoritmo en función de su costo en entradas más pequeñas:

$$T(n) = a \cdot T(n/b) + f(n)$$

- **a:** cuántas llamadas se ejecutan en una ejecución.
- **n/b:** de qué tamaño es cada subproblema.
- **f(n):** todo el trabajo fuera de las llamadas: ciclos, copias, dividir, combinar.

Leer la recurrencia en el código

from_code.h

```
1  void two_calls_linear(int n, long long&
       work) {
2      if (n <= 1) {
3          work++;
4          return;
5      }
6      for (int i = 0; i < n; i++) {
7          work++;
8      }
9      two_calls_linear(n / 2, work);
10     two_calls_linear(n / 2, work);
11 }
```

- Dos llamadas de tamaño n/2, y un ciclo de n vueltas:

$$T(n) = 2\,T(n/2) + n$$

- Tres llamadas en un `for`: $T(n) = 3\,T(n/2) + n$. a cuenta llamadas, no líneas.
- Ocho llamadas y un ciclo doble: $T(n) = 8\,T(n/2) + n^2$.

Las pruebas comprueban que el contador da lo mismo que la recurrencia.

Dos llamadas escritas, una ejecutada

pseudocódigo

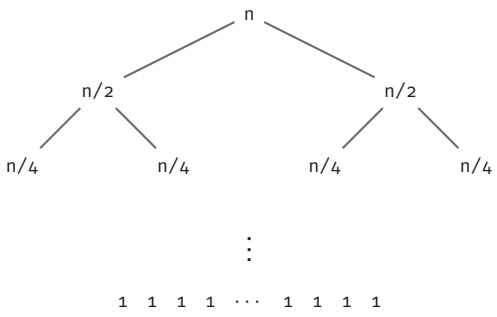
```
algoritmo buscar(a, ini, fin, x)
    si ini = fin entonces
        devolver -1
    fin si
    medio ← ini + (fin - ini) div 2
    si a[medio] = x entonces
        devolver medio
    sino si x < a[medio] entonces
        devolver buscar(a, ini, medio, x)
    sino
        devolver buscar(a, medio + 1, fin, x)
    fin si
fin algoritmo
```

- Se ejecuta una sola: **a = 1**.
- $T(n) \leq T(n/2) + 1$, y da $\Theta(\log n)$.
- Contar líneas daría a = 2, y $\Theta(n)$: incorrecto.

Pisos y techos

Mergesort es $T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + n$. Se resuelve en potencias de 2. Como T es creciente, cualquier n queda entre dos potencias: la Θ no cambia.

El árbol de recursión *(recursion tree)*: $T(n) = 2\,T(n/2) + n$



nivel 0: $1 \cdot n = n$

nivel 1: $2 \cdot n/2 = n$

nivel 2: $4 \cdot n/4 = n$

hojas: $n \cdot 1 = n$

$\log n$ niveles más las hojas, y cada uno pesa n: el total es $n \log n + n$.

La suma exacta

Con $T(n) = a\,T(n/b) + f(n)$ y $n = b^k$:

- El nivel i tiene a^i nodos, cada uno de tamaño n/b^i.
- Por lo tanto, el nivel i pesa $a^i \cdot f(n/b^i)$.
- Hay $k = \log_b n$ niveles internos.
- Hay $a^k = a^{\log_b n} = n^{\log_b a}$ hojas.

Todo el costo

$$T(n) = \sum_{i=0}^{k-1} a^i f(n/b^i) \;+\; n^{\log_b a} \cdot T(1)$$

Llamamos $c = \log_b a$: la velocidad a la que crece el número de hojas.

Quién manda: $n = 16$

8 T(n/2) + n^2

nivel 0	256.00
nivel 1	512.00
nivel 2	1024.00
nivel 3	2048.00
hojas	4096.00
total	7936.00

Crece: **pesan las hojas**.
$\Theta(n^3)$.

2 T(n/2) + n

nivel 0	16.00
nivel 1	16.00
nivel 2	16.00
nivel 3	16.00
hojas	16.00
total	80.00

Igual: **todos los niveles**.
$\Theta(n \log n)$.

2 T(n/2) + n^2

nivel 0	256.00
nivel 1	128.00
nivel 2	64.00
nivel 3	32.00
hojas	16.00
total	496.00

Decrece: **pesa la raíz**.
$\Theta(n^2)$.

Error común: "son $\log n$ niveles, así que es $n \log n$". Solo si todos pesan lo mismo.

SECCIÓN 02

Teorema maestro

02



El teorema maestro *(master theorem)*: versión de tres casos

Sean $a \geq 1$, $b > 1$ constantes, $f(n) \geq 0$, $T(n) = a\,T(n/b) + f(n)$, $T(n) = \Theta(1)$ para n pequeño, y $c = \log_b a$.

1. **Hojas.** Si $f(n) = O(n^{c-\varepsilon})$ para alguna constante $\varepsilon > 0$: $T(n) = \Theta(n^c)$.
2. **Todos los niveles.** Si $f(n) = \Theta(n^c)$: $T(n) = \Theta(n^c \log n)$.
3. **Raíz.** Si $f(n) = \Omega(n^{c+\varepsilon})$ para alguna constante $\varepsilon > 0$, y además existe $d < 1$ con $a f(n/b) \leq d f(n)$ para todo n grande: $T(n) = \Theta(f(n))$.

Condición de regularidad *(regularity condition)*

$a f(n/b) \leq d f(n)$, con $d < 1$: el nivel 1 pesa menos que el nivel 0, por un factor fijo.

Lean el caso 1 y el 3: f debe diferir de n^c por un **factor polinomial** n^ε.

Tres formas de una serie: crece, se mantiene o decrece



n = 16. Cada panel usa su propia escala: lo que importa es la forma, no comparar entre paneles.

Son los casos 1, 2 y 3, en ese orden.

Cómo se usa

Recurrencia	a, b	c	f contra n^c	Caso	Solución
$8\,T(n/2)+n^2$	8, 2	3	$n^2 = O(n^{3-1})$	1	$\Theta(n^3)$
$3\,T(n/2)+n$	3, 2	1,585	$n = O(n^{1,585-0,5})$	1	$\Theta(n^{1,585})$
$2\,T(n/2)+n$	2, 2	1	$n = \Theta(n^1)$	2	$\Theta(n \log n)$
$T(n/2)+1$	1, 2	0	$1 = \Theta(n^0)$	2	$\Theta(\log n)$
$2\,T(n/2)+n^2$	2, 2	1	$n^2 = \Omega(n^{1+1})$	3	$\Theta(n^2)$
$2\,T(n/4)+n$	2, 4	0,5	$n = \Omega(n^{0,5+0,5})$	3	$\Theta(n)$

En los casos 3, regularidad: $2(n/2)^2 = n^2/2$ y $2(n/4) = n/2$. Las dos, con $d = 1/2$.

Con $a = 1$, $c = 0$: una cadena de $\log n$ nodos con trabajo constante.

¿Para qué la regularidad?

En un polinomio se cumple sola. Si $f(n) = n^p$ con $p > c$:

$$a\,f(n/b) = \frac{a}{b^p}\,f(n), \qquad \frac{a}{b^p} < 1 \text{ porque } b^p > a.$$

Pero Ω es solo una cota por abajo, y deja saltar a f. En $T(n) = 2\,T(n/2) + f(n)$, tomemos:

$$f(2^k) = 8^k \text{ con } k \text{ impar}, \qquad f(2^k) = 4^k \text{ con } k \text{ par}.$$

- $f(n) \geq n^2$, así que $f(n) = \Omega(n^{1+1})$.
- Con k par: el nivel 1 pesa $2f(n/2) = 2(n/2)^3 = n^3/4$, y la raíz solo n^2.
- Entonces $T(n) \geq n^3/4$ con $f(n) = n^2$: $T(n)$ no es $\Theta(f(n))$.

Sin esa condición, el caso 3 sería falso.

Demostración para $n = b^k$: caso 1

Afirma: si $f(n) = O(n^{c-\varepsilon})$, entonces $T(n) = \Theta(n^c)$. **Técnica:** acotar la suma de los niveles con una serie geométrica.

1. Existe C con $f(m) \leq C\, m^{c-\varepsilon}$ para todo $m \geq 1$.
2. Como $a = b^c$, el nivel i cumple:

$$a^i f(n/b^i) \leq C\, a^i \frac{n^{c-\varepsilon}}{b^{i(c-\varepsilon)}} = C\, n^{c-\varepsilon}\, (b^\varepsilon)^i$$

3. Es una serie geométrica de razón $b^\varepsilon > 1$:

$$\sum_{i=0}^{k-1} (b^\varepsilon)^i = \frac{b^{\varepsilon k} - 1}{b^\varepsilon - 1} \leq \frac{n^\varepsilon}{b^\varepsilon - 1}$$

4. Los niveles suman $O(n^{c-\varepsilon} \cdot n^\varepsilon) = O(n^c)$.
5. Las hojas son $\Theta(n^c)$, y ellas solas ya dan la cota inferior. ■

Demostración para $n = b^k$: casos 2 y 3

Caso 2. $f(m) = \Theta(m^c)$, y $a^i/b^{ic} = 1$.

- Cada nivel con $n/b^i \geq n_0$ pesa $a^i f(n/b^i) = \Theta(n^c)$.
- Hay $\log_b n$ niveles, y solo un número constante queda bajo n_0: $T(n) = \Theta(n^c \log n)$.

Caso 3. Aplicando la regularidad i veces: $a^i f(n/b^i) \leq d^i f(n)$, mientras $n/b^{i-1} \geq n_0$.

- Esos niveles suman a lo sumo $f(n)\,(1 + d + d^2 + \cdots) = f(n)/(1 - d)$.
- Abajo quedan a lo sumo $O(a^k) = O(n^c)$ subárboles de entrada menor que una constante, $O(1)$ cada uno.
- Y $n^c = O(f(n))$, porque $f(n) = \Omega(n^{c+\varepsilon})$. Entonces $T(n) = O(f(n))$.
- La raíz sola pesa $f(n)$: $T(n) = \Omega(f(n))$. ■

Con pisos y techos el resultado es el mismo. Los detalles están en CLRS.

SECCIÓN 03

Cuándo no aplica

03

No tienen la forma

- **Tamaños distintos:** $T(n) = T(n/5) + T(7n/10) + n$. La próxima semana.
- **Por resta:** $T(n) = T(n-1) + n$. Una cadena de n nodos: $\Theta(n^2)$. Es mergesort si partiera en 1 y $n-1$.
- **Llamadas que dependen de n:** $T(n) = n\,T(n/2) + 1$.
- $a < 1$**:** no es una cantidad de llamadas.
- **Una f que depende de algo más que el subproblema:** lo veremos con las copias.

Y lo más sutil

Tienen justo la forma pedida, pero f cae **entre** los casos. Veamos dos, ambas con $a = b = 2$ y $c = 1$.

$T(n) = 2\,T(n/2) + n \log n$: ningún caso

- **No es el caso 1:** f es mayor que n.
- **No es el caso 2:** $f(n)/n = \log n$ crece sin límite.
- **No es el caso 3:** $f(n)/n^{1+\varepsilon} = \log n/n^{\varepsilon} \to 0$ para todo $\varepsilon > 0$.

El árbol no tiene problema. Con $n = 2^k$, el nivel i pesa:

$$2^i \cdot \frac{n}{2^i} \cdot \log \frac{n}{2^i} = n\,(k - i)$$

Sumando: $T(n) = n\,(k + \cdots + 1) + n = n\,\dfrac{k(k+1)}{2} + n = \Theta(n \log^2 n)$.

Una versión más general

La de CLRS, 4.ª ed.: si $f(n) = \Theta(n^c \log^m n)$ con $m \geq 0$, da $\Theta(n^c \log^{m+1} n)$. Con $m = 1$, lo mismo.

$T(n) = 2\,T(n/2) + n/\log n$: fuera incluso de esa

- **Ninguno de los tres:** $f(n)/n^{1-\varepsilon} = n^{\varepsilon}/\log n \to \infty$; $f(n)/n = 1/\log n \to 0$; f es menor que n.
- **Tampoco CLRS:** pide $m \geq 0$, y aquí sería $m = -1$. Teoremas más generales, como Akra-Bazzi, sí la cubren.

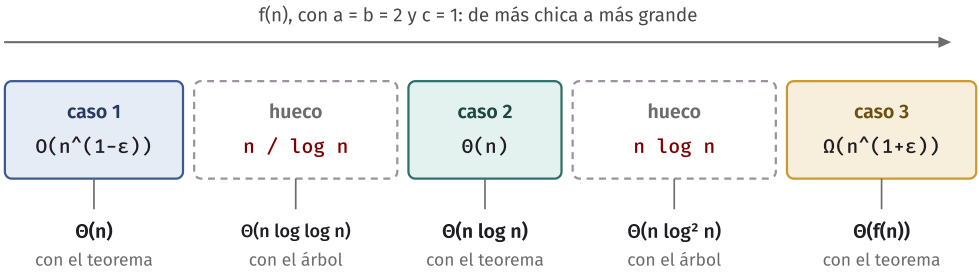
El árbol: el nivel i pesa $n/(k-i)$. Sumando,

$$T(n) = n\left(\frac{1}{k} + \frac{1}{k-1} + \cdots + 1\right) + n = n\,H_k + n$$

- $H_k = \Theta(\log k)$, agrupando: $\frac{1}{2} + \frac{1}{3} \leq 1$, $\frac{1}{4} + \cdots + \frac{1}{7} \leq 1$, y del otro lado $\frac{1}{3} + \frac{1}{4} \geq \frac{1}{2}$, $\frac{1}{5} + \cdots + \frac{1}{8} \geq \frac{1}{2}$.
- Como $k = \log n$: $T(n) = \Theta(n \log\log n)$.

$n = 16$: $16 \cdot (1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4}) + 16 = 49{,}33$, igual que el programa. Hasta $n = 1$ hay que cuidar: $\log 1 = 0$, y por eso vale desde $n \geq 2$.

Entre un caso y otro



En los huecos, f difiere de n, pero no por un factor n^ε. El caso 3 pide, además, la regularidad.

Si f cae en un hueco, el teorema de tres casos no dice nada. El árbol, sí.

Comprobar con números

$T(n)$ con la recurrencia, dividido por la solución. Es aritmética, no un algoritmo. Filas seleccionadas.

terminal

```
$ build/recurrences
log n  8T+n^2   3T+n      T+1       2T+n      2T+nlogn  2T+n/logn
       /n^3     /n^1,585  /log n    /nlogn    /nlog^2n  /nloglogn
4      1.938    2.605     1.250     1.250     0.688     1.542
16     2.000    2.997     1.062     1.062     0.535     1.095
256    2.000    3.000     1.004     1.004     0.502     0.891
1024   2.000    3.000     1.001     1.001     0.500     0.851
```

- Las cinco primeras se acercan a 2, 3, 1, 1 y 1/2: las constantes de las formas cerradas.
- La última baja a $\ln 2 \approx 0{,}693$, muy despacio: a $n = 2^{1024}$ todavía no llega.

Ojo: una tabla detecta errores y da confianza. La demostración es la suma del árbol.

Una “demostración” equivocada

Afirma: $T(n) = 2\,T(n/2) + n$ es $O(n)$. Supongamos $T(n/2) \leq c\,n/2$:

$$\begin{aligned} T(n) &= 2\,T(n/2) + n \\ &\leq 2 \cdot c\,(n/2) + n \\ &= cn + n \\ &= (c+1)\,n = O(n) \end{aligned}$$ ¡falso!

- Había que demostrar $T(n) \leq cn$, con la misma c. Salió $cn + n$, que no lo implica.
- La constante crece en cada nivel: c, $c+1$, $c+2$, … Ese es el factor $\log n$ que faltaba.

Idea clave: en una inducción se fija una constante, y se demuestra **exactamente esa cota**.

La demostración correcta

Afirma: si $T(1) = 1$ y $T(n) = 2\,T(n/2) + n$, entonces $T(n) \leq 2n \log n$ para toda potencia de 2 con $n \geq 2$. **Técnica:** inducción, con $c = 2$ fija.

Base, $n = 2$: $T(2) = 2 \cdot 1 + 2 = 4$, y $2 \cdot 2 \cdot \log 2 = 4$.

Paso, $n \geq 4$:

$$
\begin{aligned}
T(n) &= 2\,T(n/2) + n \\
&\leq 2 \cdot 2\,(n/2) \log(n/2) + n && \text{hipótesis, con } n/2 \geq 2 \\
&= 2n(\log n - 1) + n \\
&= 2n \log n - n \\
&\leq 2n \log n && \text{la misma cota}
\end{aligned}
$$

La base no puede ser $n = 1$: la cota diría $T(1) \leq 0$. Por eso el paso empieza en $n = 4$.

A veces hay que demostrar algo más fuerte

$T(n) = 2\,T(n/2) + 1$, con $T(1) = 1$. Es $O(n)$. Pero al intentar $T(n) \leq cn$:

$$T(n) \leq 2 \cdot c\,(n/2) + 1 = cn + 1$$

Se pasa por 1, con cualquier c. La afirmación es verdadera, pero la hipótesis es muy débil.

Afirma: $T(n) \leq 2n - 1$.

- **Base:** $T(1) = 1 = 2 - 1$.
- **Paso:** $T(n) = 2\,T(n/2) + 1 \leq 2\,(2 \cdot n/2 - 1) + 1 = 2n - 1$.

Idea clave

Restar un término de menor orden hace más fuerte la hipótesis. De hecho, $T(n) = 2n - 1$ exactamente.

SECCIÓN 04

El costo de copiar

04

Búsqueda binaria, tres veces

terminal

```
$ build/copy_count
búsqueda binaria
n        llamadas  entero por valor  solo la mitad  por referencia
1024     12        12288             1023           0
2048     13        26624             2047           0
4096     14        57344             4095           0
8192     15        122880            8191           0
```

Copias contadas. Peor caso: un valor menor que todos.

- **Por referencia:** $T(n) = T(n/2) + 1 = \Theta(\log n)$.
- **Copia solo la mitad:** $T(n) = T(n/2) + n = \Theta(n)$. Caso 3, con $d = 1/2$.
- **Entero por valor:** el arreglo que se copia no se achica. Con m = tamaño del rango:

$$S(m) = S(m/2) + n, \quad n \text{ fijo} \quad \Rightarrow \quad S(n) = n\,(\log n + 2) = \Theta(n \log n)$$

Peor que una búsqueda lineal.

Mergesort con un & de menos

Recibir el auxiliar por valor: $2n - 1$ llamadas, con n copias cada una: $\Theta(n^2)$.

terminal

```
$ build/copy_count
mergesort con el auxiliar por valor
n        llamadas  copias
1024     2047      2096128
2048     4095      8386560
4096     8191      33550336
```

- Un árbol con n hojas y dos hijos por nodo tiene $n - 1$ nodos internos: $2n - 1$ llamadas.
- Copiar **solo el subproblema** en mergesort no cambia la Θ: se suma a una f que ya era $\Theta(n)$.

Error común: ignorar el costo de copiar. Pregúntense **de qué tamaño** es la copia. En C++: `const std::vector<int>&` o `std::span`.

Y al medirlo

terminal

```
$ build/copy_bench
n       referencia ms  razón  por valor ms  razón
2000             0.06                  0.68
4000             0.18   3.01           3.35   4.96
8000             0.45   2.47          21.98   6.55
16000            0.99   2.19         138.28   6.29
32000            2.20   2.22         833.45   6.03
```

`-O2`, menor de 3, máquina compartida. Depende de la máquina.

- Con 32000 elementos: unos 2 ms contra más de 800.
- Las copias se multiplican por 4; el tiempo, por 6. Lo que no cuadra también importa.
- Una copia no siempre cuesta lo mismo: cada una reserva memoria, y un bloque más grande puede dejar de caber en la cache. Es una hipótesis: esta medición no la comprueba.

SECCIÓN 05

Mergesort demostrado

05



El pseudocódigo: es lo que se demuestra

pseudocódigo

```
algoritmo mezclar(a, aux, ini, medio, fin)
    i ← ini
    j ← medio
    para k desde ini hasta fin - 1 hacer
        si j = fin o (i < medio y a[i] ≤ a[j]) entonces
            aux[k] ← a[i]
            i ← i + 1
        sino
            aux[k] ← a[j]
            j ← j + 1
        fin si
    fin para
    para k desde ini hasta fin - 1 hacer
        a[k] ← aux[k]
    fin para
fin algoritmo
```

pseudocódigo

```
algoritmo ordenar(a, aux, ini, fin)
    si fin - ini ≤ 1 entonces
        devolver
    fin si
    medio ← ini + (fin - ini) div 2
    ordenar(a, aux, ini, medio)
    ordenar(a, aux, medio, fin)
    mezclar(a, aux, ini, medio, fin)
fin algoritmo
```

Rangos semiabiertos: [ini, fin) no incluye a `fin`.

La condición se evalúa de izquierda a derecha y se detiene al decidirse: `a[j]` no se lee con `j = fin`.

En C++: `merge_sort.h`

merge_sort.h

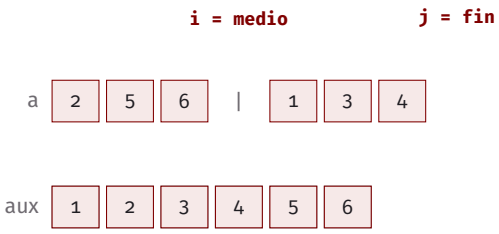
```cpp
template <typename T>
void merge_ranges(std::vector<T>& a, std::vector<T>&
     buffer,
                  std::size_t lo, std::size_t mid, std
     ::size_t hi) {
    std::size_t i = lo, j = mid;
    for (std::size_t k = lo; k < hi; k++) {
        if (j == hi || (i < mid && a[i] <= a[j])) {
            buffer[k] = a[i++];
        } else {
            buffer[k] = a[j++];
        }
    }
    for (std::size_t k = lo; k < hi; k++) a[k] =
     buffer[k];
}
```

- El auxiliar se crea una sola vez, y se pasa por referencia.
- `<=`: en un empate pasa primero el de la izquierda. Por eso es **estable**.
- `mid = lo + (hi - lo) / 2`.

En la práctica

Para ordenar, usen `std::stable_sort`. Aquí lo escribimos para demostrarlo y extenderlo.

Una mezcla, vuelta por vuelta



Lo que ya pasó son siempre los más chicos, y están ordenados. Eso es la invariante.

La invariante de la mezcla

Afirma: si $a[ini, medio)$ y $a[medio, fin)$ están ordenados, al terminar el primer ciclo $aux[ini, fin)$ tiene los mismos elementos, ordenados. **Técnica:** invariante de ciclo.

Invariante, antes de cada vuelta

1. Ya se copiaron exactamente $a[ini, i)$ y $a[medio, j)$: $(i - ini) + (j - medio) = k - ini$.
2. $aux[ini, k)$ tiene esos mismos elementos, ordenados.
3. Todo elemento de $aux[ini, k)$ es $\leq$ que todo elemento de $a[i, medio)$ y de $a[j, fin)$.

Inicio. $k = ini$, $i = ini$, $j = medio$: nada copiado, las tres valen sobre conjuntos vacíos.
Atención a (3): sin ella, (2) no se conservaría.

Conservación: las dos ramas

Como $k < \mathit{fin}$, por (1) falta copiar al menos un elemento.

El `si` se cumple.

- $a[i]$ **existe:** si fue por $i < \mathit{medio}$, es inmediato. Si fue por $j = \mathit{fin}$, la derecha ya se copió, y como falta algo, $i < \mathit{medio}$.
- $a[i]$ **es** $\leq$ **que todo lo que falta:** en la izquierda es el primero de un tramo ordenado. En la derecha, si $j = \mathit{fin}$ no hay nadie; si no, $a[i] \leq a[j]$, y $a[j]$ es el menor de su tramo.
- **Se conservan:** (1) porque pasó de lo que falta a lo copiado. (2) porque, por (3), $a[i]$ es $\geq$ que todo lo copiado. (3) porque lo que falta ahora es un subconjunto, y el nuevo es $\leq$ que todo eso.

El `si` no se cumple.

- $j < \mathit{fin}$, así que $a[j]$ existe. Y $i = \mathit{medio}$ o $a[i] > a[j]$.
- Entonces $a[j]$ es $\leq$ que toda la izquierda que queda, y primero de la derecha ordenada. El resto, igual.

Salida. $k = \mathit{fin}$. Por (1) se copió todo; por (2), está ordenado. El segundo ciclo lo devuelve a a. ■

Mergesort es correcto: inducción sobre n

Afirma: `ordenar(a, aux, ini, fin)` termina, deja ordenado $a[ini, fin)$ con los mismos elementos, y no toca el resto. **Técnica:** inducción fuerte sobre $n = fin - ini$.

Base, $n \leq 1$**.** No hace nada, y un rango así ya está ordenado.

Paso, $n \geq 2$**.**

1. Las mitades tienen $\lfloor n/2 \rfloor$ y $\lceil n/2 \rceil$ elementos, ambas menores que n, porque $n \geq 2$.
2. Por hipótesis, cada llamada termina y ordena su mitad, sin tocar nada más.
3. Se cumple lo que `mezclar` necesita: queda ordenado. ■

Error común

Un caso base que no incluye $n = 1$. Con $n = 1$, $\lfloor 1/2 \rfloor = 0$, y la segunda llamada recibe el mismo rango: recursión infinita. La demostración lo muestra: el paso usó que las mitades son menores que n.

Cuánto cuesta

$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + \Theta(n) = \Theta(n \log n)$. Comparaciones, peor caso: $C(n) = C(\lfloor n/2 \rfloor) + C(\lceil n/2 \rceil) + n - 1$; para $n = 2^k$, $n \log n - n + 1$.

terminal

```
$ build/comparisons
n         ordenado  invertido al azar  peor      fórmula
16        32        32        48       49        49
17        33        37        50       54        54
1000      4932      5044      8710     8977      8977
1024      5120      5120      8935     9217      9217
65536     524288    524288    965653   983041    983041
1000000   9884992   10066432  18673910 18951425  18951425
```

Una mezcla hace al menos $\lfloor n/2 \rfloor$ comparaciones: $\Theta(n \log n)$ con **cualquier** entrada. Con datos al azar queda muy cerca del peor caso.

Medir con n y 2n

terminal

```
$ build/merge_bench
n         tiempo ms  razón  predice 2(k+1)/k
131072         10.3
262144         21.7   2.11     2.12
524288         48.2   2.22     2.11
1048576        98.2   2.04     2.11
2097152       204.6   2.08     2.10
4194304       425.8   2.08     2.10
8388608       893.3   2.10     2.09
```

`-O2`, menor de 3, máquina compartida. Depende de la máquina.

- Las razones oscilan alrededor de lo que predice la recurrencia.
- Con números tan cercanos, la tabla es compatible con $n \log n$, pero no la distingue de funciones parecidas. Lo que la sostiene es la demostración.

SECCIÓN 06

Contar inversiones

06

Inversiones *(inversions)*

Inversión

Un par de posiciones (i, j) con $i < j$ y $a[i] > a[j]$: dos elementos en el orden contrario.

terminal

```
$ build/inversions
3 1 4 2: 3 inversiones: (3,1) (3,2) (4,2)
1 2 3 4: 0 inversiones:
4 3 2 1: 6 inversiones: (4,3) (4,2) (4,1) (3,2) (3,1) (2,1)
```

Cuántas pueden ser

- **Mínimo, 0:** el arreglo ordenado. Uno no ordenado tiene dos vecinos al revés.
- **Máximo,** $n(n-1)/2$**:** todos los pares. Con elementos distintos, solo el decreciente.
- No cabe en `int`: $n = 65\,536$ da 2 147 450 880, y sí cabe. $n = 65\,537$ da 2 147 516 416, y ya no.

Error común

Un `int`. Es comportamiento indefinido: sin sanitizers, lo habitual es un número equivocado, a veces creíble. Usen `long long`.

Qué miden

Cuánto se parecen dos rankings. Dos personas ordenan las mismas n películas. Un par está en **desacuerdo** si una pone X antes que Y, y la otra, al revés.

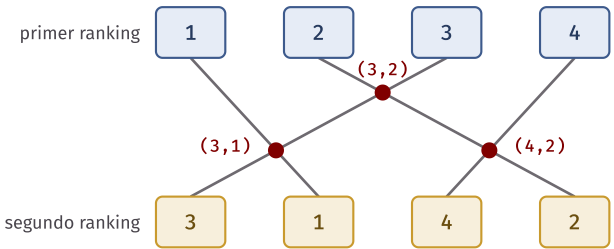
- 0 si son iguales, $n(n-1)/2$ si uno es el otro al revés. Se conoce como distancia de Kendall tau.
- Si el primero es $1, 2, \ldots, n$, los pares en desacuerdo son las inversiones del segundo. El caso general: primer paso del laboratorio.

Cuánto cuesta ordenar con pasos pequeños. Con elementos distintos, el mínimo de intercambios de vecinos para ordenar es igual a las inversiones.

- Intercambiar vecinos cambia las inversiones en **exactamente 1**: los demás pares conservan su orden.
- **Se necesitan esos:** hay que llegar a 0, y cada intercambio baja a lo sumo 1.
- **Alcanzan:** si no está ordenado, hay vecinos al revés, e intercambiarlos baja 1. ■

Inserción y burbuja las quitan de una en una: por eso son $\Theta(n^2)$ en el peor caso.

Dos rankings: cada cruce es un desacuerdo



3 cruces

3 pares en desacuerdo

3 inversiones de 3 1 4 2

Dos líneas se cruzan exactamente cuando los rankings ordenan ese par al revés.

Fuerza bruta: $\Theta(n^2)$

pseudocódigo

```
algoritmo inversiones_fuerza_bruta(a, n)
    cuenta ← 0
    para i desde 0 hasta n - 1 hacer
        para j desde i + 1 hasta n - 1 hacer
            si a[i] > a[j] entonces
                cuenta ← cuenta + 1
            fin si
        fin para
    fin para
    devolver cuenta
fin algoritmo
```

terminal

```
$ build/brute_bench
n       inversiones  tiempo ms  razón
10000   24961152          17.7
20000   99565270          71.5   4.04
40000   402372200        282.0   3.95
80000   1605128162      1128.9   4.00
160000  6407002153      4525.7   4.01
```

-O2, menor de 3, máquina compartida. Depende de la máquina.

- A este ritmo, $n = 10^6$ tomaría unos 3 minutos. Mergesort ordena un millón en una décima de segundo.
- Hay $\Theta(n^2)$ pares. Pero no hace falta visitarlos: basta **contarlos**, muchos de un solo golpe.

Dentro de cada mitad, y entre las dos

$$\text{inversiones}(a) = \text{inversiones}(\text{izq}) + \text{inversiones}(\text{der}) + \text{cruzadas}(\text{izq}, \text{der})$$

5 2 6 1 4 3, por fuerza bruta

```
[0,6)  izquierda 1  derecha 1  cruzadas 7  total 9
[0,3)  izquierda 0  derecha 0  cruzadas 1  total 1
[1,3)  izquierda 0  derecha 0  cruzadas 0  total 0
[3,6)  izquierda 0  derecha 1  cruzadas 0  total 1
[4,6)  izquierda 0  derecha 0  cruzadas 1  total 1
```

- Contar las cruzadas revisando pares: $T(n) = 2\,T(n/2) + \Theta(n^2) = \Theta(n^2)$. No se gana nada.
- Si fuera $\Theta(n)$, la recurrencia sería la de mergesort.
- **Observación clave:** ordenar cada mitad no cambia las cruzadas. Solo dependen de qué valor está en cada lado, no de dónde.
- Y mergesort deja las mitades justo ordenadas.

Contar durante la mezcla: 2 5 6 | 1 3 4

k	i	j	pasa	de	quedan a la izquierda	suma
0	0	3	1	derecha	3 (2, 5, 6)	3
1	0	4	2	izquierda	–	3
2	1	4	3	derecha	2 (5, 6)	5
3	1	5	4	derecha	2 (5, 6)	7
4	1	6	5	izquierda	–	7
5	2	6	6	izquierda	–	**7**

Cuando el 1 pasa antes que todos, es menor que **todos** los que quedan a la izquierda: 3 inversiones de un golpe.

Son 7, igual que las cruzadas que contó la fuerza bruta. **Error común:** contar 1 en vez de todos los que quedan: aquí daría 3, y con 2 | 1 acertaría de casualidad.

Las tres vueltas que suman



k = 0: pasa el 1, de la derecha

`medio - i = 3 - 0 = 3`

`(2,1) (5,1) (6,1)`

suma

3



k = 2: pasa el 3, de la derecha

`medio - i = 3 - 1 = 2`

`(5,3) (6,3)`

suma

5



k = 3: pasa el 4, de la derecha

`medio - i = 3 - 1 = 2`

`(5,4) (6,4)`

suma

7

En las otras tres vueltas pasa uno de la izquierda (el 2, el 5, el 6) y no se suma nada.

Cada arco es una inversión cruzada: las 7 se cuentan sin recorrerlas.

El lema

Lema. Con ambas mitades ordenadas, cuando $y = a[j]$ pasa a `aux`, los elementos de la izquierda mayores que y son exactamente los de $a[i, \mathit{medio})$, y son $\mathit{medio} - i$. **Técnica:** dos inclusiones, con la invariante.

Todos los de $a[i, \mathit{medio})$ son mayores que y.

- Si $i = \mathit{medio}$, no hay ninguno.
- Si no, y pasó porque el `si` falló, y con $j \neq \mathit{fin}$ tuvo que ser porque $a[i] > a[j] = y$.
- La izquierda está ordenada: todo $a[i, \mathit{medio})$ es $\geq a[i] > y$.

Ninguno de los ya copiados es mayor que y.

- Los de $a[\mathit{ini}, i)$ están en `aux`. La condición (3) dice que son $\leq$ que todo lo que falta, y a y le faltaba copiarse. ■

Consecuencia. Cada cruzada (x, y) se cuenta una sola vez: en la vuelta en que y pasa.

El algoritmo

pseudocódigo

```
algoritmo contar_inversiones(a, aux, ini, fin)
    si fin - ini ≤ 1 entonces
        devolver 0
    fin si
    medio ← ini + (fin - ini) div 2
    cuenta ← contar_inversiones(a, aux, ini, medio)
    cuenta ← cuenta +
             contar_inversiones(a, aux, medio, fin)
    i ← ini
    j ← medio
```

pseudocódigo

```
para k desde ini hasta fin - 1 hacer
        si j = fin o (i < medio y a[i] ≤ a[j]) entonces
            aux[k] ← a[i]
            i ← i + 1
        sino
            aux[k] ← a[j]
            j ← j + 1
            cuenta ← cuenta + (medio - i)
        fin si
    fin para
    para k desde ini hasta fin - 1 hacer
        a[k] ← aux[k]
    fin para
    devolver cuenta
fin algoritmo
```

La mezcla de mergesort, con una línea más. Además de contar, **deja el rango ordenado**.

Correcto, y en cuánto

Afirma: devuelve las inversiones que tenía $a[ini, fin)$ y lo deja ordenado. **Técnica:** inducción fuerte sobre $n = fin - ini$.

Base, $n \leq 1$**:** no hay pares, devuelve 0, y ya está ordenado.

Paso, $n \geq 2$**:**

1. Por hipótesis, las llamadas cuentan dentro de cada mitad y las dejan ordenadas.
2. La mezcla deja ordenado el rango, y por el lema suma las cruzadas de las mitades ya ordenadas.
3. Esas cruzadas son las mismas de antes de ordenar. La suma es el total. ■

Costo: sumar es $O(1)$ por vuelta, así que $T(n) = 2\,T(n/2) + \Theta(n) = \Theta(n \log n)$, sin enumerar nunca los $\Theta(n^2)$ pares.

Más errores de esta sección

- **Usar < en vez de $\leq$.** Con 1, 1 no hay ninguna inversión, pero el 1 de la derecha pasaría primero y se contaría 1. El lema dependía de que en un empate pase primero el de la izquierda.
- **Sumar $i - ini$ en vez de $medio - i$.** Los que ya pasaron no son mayores: son los que faltan.
- **No copiar de vuelta a `a`.** El nivel de arriba recibiría mitades desordenadas, y el lema ya no aplicaría.

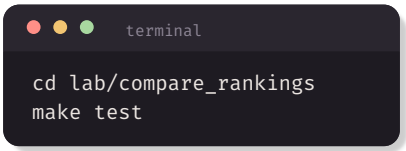
Para comprobar

Corran el programa contra la fuerza bruta en miles de arreglos chicos al azar. Un desacuerdo señala el error en un caso que se puede leer.

Laboratorio: comparar rankings

1. Con dos rankings de los mismos elementos, reducir el problema a contar inversiones en un solo arreglo.
2. Escribir primero la versión $\Theta(n^2)$: AC en los casos pequeños, TLE en el grande.
3. Escribir la versión con mergesort y pasar todos los casos.
4. Medir con n y $2n$, y comparar las razones con lo que predice la recurrencia.

Antes de escribir, calculen: ¿cuánto vale la respuesta más grande con los límites del enunciado?



Enunciado en `README.md`. El caso grande lo genera `make test`.

Para la próxima clase

- Leer los apuntes de la semana 3: selección en tiempo lineal, Karatsuba y el par más cercano.
- Laboratorio: comparar rankings.
- Resuelvan las preguntas de repaso de la guía de la semana 2.

Pregunta

$T(n) = T(n/5) + T(7n/10) + n$ tiene dos subproblemas de distinto tamaño. El teorema no aplica. ¿Qué diría el árbol de recursión, nivel por nivel?