

## Exemple 1: el problema de selecció

### Problema de selecció

Donada una llista de  $n$  naturals, determinar el  $k$ -èsim més gran.

### Primera solució

Ordenar els nombres en un vector de forma decreixent i retornar el  $k$ -èsim.

### Segona solució

Escriure els  $k$  primers nombres en un vector  $V[0 \dots k-1]$  i ordenar-los decreixentment. Per a cada element restant:

- si és més petit que  $V[k-1]$ , es descarta
- si no, se situa correctament en  $V$  i s'elimina el més petit

Retornar  $V[k-1]$ .

- amb un algorisme d'ordenació bàsic (bombolla, inserció):  $O(n^2)$
- amb un algorisme d'ordenació eficient:  $O(n \log n)$

$$O((k \log k) + (n - k) \cdot k)$$

- Si  $k$  és constant, és  $O(k \cdot n) = O(n)$
- Si  $k = \lceil n/2 \rceil$ , és  $O(\frac{n}{2} \cdot \frac{n}{2}) = O(n^2)$

## Exemple 2: el mur infinit

### Mur infinit

Estem davant d'un mur que s'allarga indefinidament en totes dues direccions. Volem trobar l'única porta que el travessa, però no sabem a quina distància és ni en quina direcció. Tot i que és fosc, portem una espelma que ens permet veure la porta quan hi som a prop.

### Primera solució

- Avancem 1 metre i tornem a l'origen
- Retrocedim 2 metres i tornem a l'origen
- Avancem 3 metres i tornem a l'origen
- Retrocedim 4 metres i tornem a l'origen
- ...

### Segona solució

- Avancem 1 metre i tornem a l'origen
- Retrocedim 2 metres i tornem a l'origen
- Avancem 4 metres i tornem a l'origen
- Retrocedim 8 metres i tornem a l'origen
- ...

Temps quan la porta és a distància  $n$ :

$$T(n) = 2 \sum_{i=1}^{n-1} i + n = 2 \frac{(n-1)n}{2} + n = n^2 \in O(n^2).$$

$$\left( \text{recordem que } \sum_{i=1}^{n-1} i = \frac{(n-1)n}{2} \right)$$

Si la porta és a distància  $n = 2^k$ , aleshores

$$T(n) = 2 \sum_{i=0}^{k-1} 2^i + 2^k = 2(2^k - 1) + 2^k = 3n - 2 \in O(n).$$

$$\left( \text{recordem que } \sum_{i=0}^{k-1} 2^i = 2^k - 1 \right)$$

### Mida de l'entrada i cost

Donat un algorisme  $A$  amb conjunt d'entrades  $\mathcal{E}$ , l'**eficiència** o **cost** d' $A$  es pot expressar com una funció  $T : \mathcal{E} \rightarrow \mathbb{R}^+$ .

Però calcular  $T$  per **cada entrada** pot ser complicat i de poca utilitat. És més útil agrupar les entrades amb la **mateixa mida** i estudiar el cost sobre aquestes entrades en conjunt.

#### Mida

La **mida** (o **talla**) d'una entrada  $x$  és el nombre de símbols necessari per codificar-la. Es representa amb  $|x|$ .

#### Convencions segons el tipus d'entrada

- **Nombres naturals** → codificació en binari

$$|27| = 5 \text{ perquè } \langle 27 \rangle_2 = 11011$$

- **Llistes, vectors** → nombre de components

$$|(23, 1, 7, 0, 12, 500, 2, 11)| = 8$$

## Mida de l'entrada i cost

### Exercici

Demostreu que  $|\langle x \rangle_2| = \lfloor \log_2 x \rfloor + 1$ , on  $\langle x \rangle_2$  és la codificació binària de  $x$ .

Pista: expresseu  $x$  en binari

$$\langle x \rangle_2 = b_{k-1}b_{k-2}\dots b_0$$

on  $b_{k-1} \neq 0$  i calculeu els valors mínim i màxim de  $x$  en funció de  $k$ .

### Notació

A partir d'ara, escriurem  $\log$  en lloc de  $\log_2$ .

Suposem que  $A$  és un algorisme i  $T(x)$  el cost d'executar  $A$  amb entrada  $x \in \mathcal{E}$ . Es defineixen 3 **funcions de cost** segons la mida de l'entrada:

- **Cas pitjor.**  $T_{\text{pitjor}}(n) = \max\{T(x) \mid x \in \mathcal{E} \wedge |x| = n\}$

Dona garanties sobre límits que l'algorisme no superarà.

- **Cas millor.**  $T_{\text{millor}}(n) = \min\{T(x) \mid x \in \mathcal{E} \wedge |x| = n\}$

Poc útil.

- **Cas mig.**  $T_{\text{mig}}(n) = \sum_{x \in A, |x|=n} Pr(x)T(x)$ ,  
on  $Pr(x)$  és la probabilitat de l'ocurrència de l'entrada  $x$  en  $\mathcal{E}$

Cal definir la distribució de probabilitat. Sol ser difícil de calcular.

Taula 1 (Garey/Johnson, *Computers and Intractability*)

Comparació de funcions polinòmiques i exponentials.

| cost  | 10        | 20        | 30        | 40          | 50                     |
|-------|-----------|-----------|-----------|-------------|------------------------|
| $n$   | 0.00001 s | 0.00002 s | 0.00003 s | 0.00004 s   | 0.00005 s              |
| $n^2$ | 0.0001 s  | 0.0004 s  | 0.0009 s  | 0.0016 s    | 0.0025 s               |
| $n^3$ | 0.001 s   | 0.008 s   | 0.027 s   | 0.064 s     | 0.125 s                |
| $n^5$ | 0.1 s     | 3.2 s     | 24.3 s    | 1.7 min     | 5.2 min                |
| $2^n$ | 0.001 s   | 1.0 s     | 17.9 min  | 12.7 dies   | 35.7 anys              |
| $3^n$ | 0.059 s   | 58 min    | 6.5 anys  | 3855 segles | $2 \times 10^8$ segles |

## Mida de l'entrada i cost

Taula 2 (Garey/Johnson, *Computers and Intractability*)

Efecte de les millores en la tecnologia sobre algorismes polinòmics i exponentials.

S'indiquen les mides de les entrades processades per unitat de temps

| cost  | tecnologia actual | tecnologia $\times 100$ | tecnologia $\times 1000$ |
|-------|-------------------|-------------------------|--------------------------|
| $n$   | $N_1$             | $100N_1$                | $1000N_1$                |
| $n^2$ | $N_2$             | $10N_2$                 | $31.6N_2$                |
| $n^3$ | $N_3$             | $4.64N_3$               | $10N_3$                  |
| $2^n$ | $N_4$             | $N_4 + 6.64$            | $N_4 + 9.97$             |
| $3^n$ | $N_5$             | $N_5 + 4.19$            | $N_5 + 6.29$             |

## Ordres de magnitud

Necessitem una notació que:

- permeti donar una fita superior de

$$T_{\text{pitjor}}(n) = \max\{T(x) \mid x \in \mathcal{A} \wedge |x| = n\}.$$

(sabrem que l'algorisme mai superarà la fita)

- que sigui independent dels factors constants  
(així no dependrà de la implementació)

### Notació O gran

Donada una funció  $g$ ,  $O(g)$  és la classe de funcions  $f$  que “no creixen més de pressa que  $g$ ”. Formalment,  $f \in O(g)$  si existeixen  $c > 0$  i  $n_0 \in \mathbb{N}$  tals que

$$\forall n \geq n_0 \quad f(n) \leq c \cdot g(n).$$

En lloc de  $f \in O(g)$ , s'escriu sovint “ $f$  és  $O(g)$ ” o, també,  $f = O(g)$ .

## Ordres de magnitud

### Exemple

Sigui  $f(n) = 3n^3 + 5n^2 - 7n + 41$ . Llavors, podem afirmar que  $f \in O(n^3)$ .

Per justificar-ho, només cal trobar constants  $c$  i  $n_0$  tals que

$$\forall n \geq n_0 \quad f(n) \leq cn^3.$$

Però  $3n^3 + 5n^2 - 7n + 41 \leq 8n^3 + 41$ . Triem  $c = 9$ . Llavors,

$$8n^3 + 41 \leq 9n^3 \iff 41 \leq n^3,$$

que es compleix a partir de  $n_0 = 4$ . Per tant,  $\forall n \geq 4 \quad f(n) \leq 9n^3$  i, llavors,  $f(n) = O(n^3)$  amb  $c = 9$  i  $n_0 = 4$ .

### Exercici

Trobeu una constant  $n_0$  que, juntament amb  $c = 4$ , demostri que  $f \in O(n^3)$  per a la funció  $f$  de l'exemple.

$$3n^3 + 5n^2 - 7n + 41 \leq 4n^3 \rightarrow 5n^2 - 7n + 41 \leq n^3 \rightarrow n_0 = 6 \text{ and } c = 4$$

## Notació asimptòtica: definicions

- La **notació asimptòtica** permet classificar les funcions d'acord amb la seva taxa relativa de creixement.
- Té en compte el comportament de les funcions per a entrades grans. Per exemple,  $n^2 \geq 10^6 n$  a partir d'un cert valor de  $n$ :

$$n^2 \geq 10^6 n \iff n \geq 10^6.$$

Per a  $n \geq 10^6$ , doncs,  $n^2$  creix més de pressa que  $10^6 n$ . En aquest cas, diem que la funció  $f(n) = 10^6 n$  està fitada per  $g(n) = n^2$  **asimptòticament**.

- La notació  **$O(g)$** , anomenada "O gran", representa el conjunt de funcions fitades **asimptòticament** per  $g$ .

## Notació asimptòtica: definicions

### Notació $\Theta$ ((a): fita exacta asimptòtica)

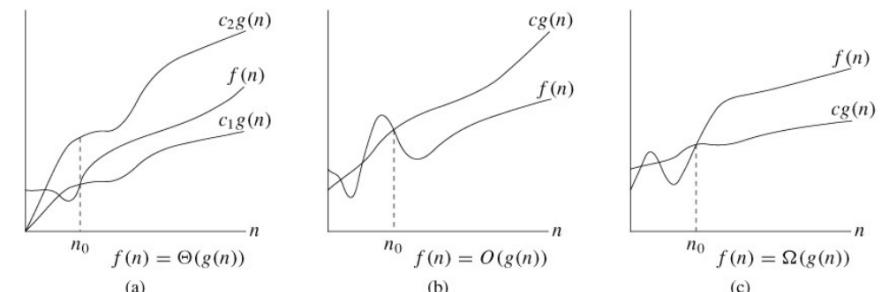
$$\Theta(g) = \{f : \mathbb{N} \rightarrow \mathbb{R}^+ \mid \exists c_1, c_2 \in \mathbb{R}^+ \exists n_0 \in \mathbb{N} \forall n \geq n_0 \quad c_1 g(n) \leq f(n) \leq c_2 g(n)\}$$

### Notació O gran ((b): fita superior asimptòtica)

$$O(g) = \{f : \mathbb{N} \rightarrow \mathbb{R}^+ \mid \exists c \in \mathbb{R}^+ \exists n_0 \in \mathbb{N} \quad \forall n \geq n_0 \quad f(n) \leq c \cdot g(n)\}$$

### Notació $\Omega$ ((c): fita inferior asimptòtica )

$$\Omega(g) = \{f : \mathbb{N} \rightarrow \mathbb{R}^+ \mid \exists c \in \mathbb{R}^+ \exists n_0 \in \mathbb{N} \quad \forall n \geq n_0 \quad f(n) \geq c \cdot g(n)\}$$



## Notació asimptòtica: propietats

### Relacions entre $O$ , $\Omega$ i $\Theta$

Donades dues funcions  $f$  i  $g$ :

- $f \in \Omega(g) \iff g \in O(f)$
- $\Theta(f) = O(f) \cap \Omega(f)$
- $O(f) = O(g) \iff \Omega(f) = \Omega(g) \iff \Theta(f) = \Theta(g)$

### Regla del límit

- $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0 \Rightarrow f \in O(g)$  però  $g \notin O(f)$
- $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \infty \Rightarrow g \in O(f)$  però  $f \notin O(g)$
- $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = c$ , on  $0 < c < \infty \Rightarrow O(f) = O(g)$

### Exercicis

Sigui  $k \geq 1$  i  $c > 1$ . Demostreu:

- ①  $n^k \in O(c^n)$  i  $n^k \notin \Omega(c^n)$
- ②  $\log^k n \in O(n)$

## Notació asimptòtica: propietats

### Propietats de l'O gran

Donades les funcions  $f, f_1, f_2, g, g_1, g_2$  i  $h$ :

- **Reflexivitat.**  $f \in O(f)$
- **Transitivitat.**  $f \in O(g) \wedge g \in O(h) \Rightarrow f \in O(h)$
- **Caracterització.**  $f \in O(g) \iff O(f) \subseteq O(g)$
- **Regla de la suma.**  $f_1 \in O(g_1) \wedge f_2 \in O(g_2) \Rightarrow f_1 + f_2 \in O(\max(g_1, g_2))$
- **Regla del producte.**  $f_1 \in O(g_1) \wedge f_2 \in O(g_2) \Rightarrow f_1 \cdot f_2 \in O(g_1 \cdot g_2)$
- **Invariança multiplicativa.** Per a tota constant  $c \in \mathbb{R}^+$ ,  $O(f) = O(c \cdot f)$

### Exercici

Feu servir la regla del límit per demostrar la transitivitat de l'O gran, és a dir, que si  $f, g, h$  són funcions, llavors  $f \in O(g) \wedge g \in O(h) \Rightarrow f \in O(h)$ .

Suposant que  $f \in O(g)$  i  $g \in O(h)$ , tenim que

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} < \infty \quad \wedge \quad \lim_{n \rightarrow \infty} \frac{g(n)}{h(n)} < \infty.$$

Aleshores,

$$\lim_{n \rightarrow \infty} \frac{f(n)}{h(n)} = \lim_{n \rightarrow \infty} \frac{f(n) \cdot g(n)}{g(n) \cdot h(n)} = \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} \cdot \lim_{n \rightarrow \infty} \frac{g(n)}{h(n)} < \infty$$

i, per tant,  $f \in O(h)$ .

### Exercici

Feu servir la regla del límit per demostrar les altres propietats de l'O gran.

### Exercici

Argumenteu per què l'affirmació  $f \in O(g)$  és equivalent a

$$\exists c \in \mathbb{R}^+ \quad \forall n \quad f(n) \leq c \cdot g(n).$$

Recordeu que, per definició,  $f \in O(g)$  si

$$\exists c \in \mathbb{R}^+ \quad \exists n_0 \in \mathbb{N} \quad \forall n \geq n_0 \quad f(n) \leq c \cdot g(n).$$

### Nota

La notació  $\forall^\infty n P(n)$  representa que  $P(n)$  es compleix per a tots els valors de  $n$  excepte per a un nombre finit.

## Notació asimptòtica: propietats

### Propietats de $\Theta$

Donades les funcions  $f, f_1, f_2, g, g_1, g_2$  i  $h$ :

- **Reflexivitat.**  $f \in \Theta(f)$
- **Transitivitat.**  $f \in \Theta(g) \wedge g \in \Theta(h) \Rightarrow f \in \Theta(h)$
- **Simetria.**  $f \in \Theta(g) \iff g \in \Theta(f) \iff \Theta(f) = \Theta(g)$
- **Regla de la suma.**  $f_1 \in \Theta(g_1) \wedge f_2 \in \Theta(g_2) \Rightarrow f_1 + f_2 \in \Theta(\max(g_1, g_2))$
- **Regla del producte.**  $f_1 \in \Theta(g_1) \wedge f_2 \in \Theta(g_2) \Rightarrow f_1 \cdot f_2 \in \Theta(g_1 \cdot g_2)$
- **Invariança multiplicativa.** Per a tota constant  $c \in \mathbb{R}^+$ ,  $\Theta(f) = \Theta(c \cdot f)$

### Notació de classes

Si  $\mathcal{F}_1$  i  $\mathcal{F}_2$  són classes de funcions (com ara  $O(f)$  o  $\Omega(f)$ ), definim:

- $\mathcal{F}_1 + \mathcal{F}_2 = \{f + g \mid f \in \mathcal{F}_1 \wedge g \in \mathcal{F}_2\}$
- $\mathcal{F}_1 \cdot \mathcal{F}_2 = \{f \cdot g \mid f \in \mathcal{F}_1 \wedge g \in \mathcal{F}_2\}$

### Regles de la suma i el producte (segona versió)

Donades dues funcions  $f$  i  $g$ :

- $O(f) + O(g) = O(f + g) = O(\max\{f, g\})$
- $O(f) \cdot O(g) = O(f \cdot g)$
- $\Theta(f) + \Theta(g) = \Theta(f + g) = \Theta(\max\{f, g\})$
- $\Theta(f) \cdot \Theta(g) = \Theta(f \cdot g)$

## Formes de creixement

### Costos freqüents

- **Constant:**  $\Theta(1)$ 
  - Decidir la paritat
  - Sumar dues variables numèriques
- **Logarítmic:**  $\Theta(\log n)$ 
  - Cerca binària
- **Lineal:**  $\Theta(n)$ 
  - Recorregut seqüencial (p. ex., calcular el màxim, el mínim, la mitjana)
- **Quasilineal:**  $\Theta(n \log n)$ 
  - Ordenacions per fusió (Mergesort) i ràpida (Quicksort)

## Formes de creixement

### Costos freqüents

- **Quadràtic:**  $\Theta(n^2)$ 
  - Suma de dues matrius quadrades
  - Ordenació per selecció i bombolla
- **Cúbic:**  $\Theta(n^3)$ 
  - Producte de dues matrius quadrades
  - Enumeració de triples
- **Polinòmic:**  $\Theta(n^k)$ , per a  $k \geq 1$  constant
  - Enumerar combinacions
  - Test de primalitat  
(amb variants de l'algorithm AKS que van de  $\Theta(n^{12})$  a  $\Theta(n^6)$ )
- **Exponencial:**  $\Theta(k^n)$ , per a  $k > 1$  constant
  - Cerca en un espai de configuracions (d'amplada  $k$  i profunditat  $n$ )
- **Altres funcions:**  $\Theta(\sqrt{n}), \Theta(n!), \Theta(n^n)$

## Formes de creixement

### Notació

Donades dues funcions  $f$  i  $g$ , escrivim  $f \prec g$  per indicar que  $f \in O(g)$  però  $g \notin O(f)$ .

### Exercici

Trobeu dos costos  $f, g$  de l'escala anterior per als quals  $f \prec \sqrt{n} \prec g$ .

### Solució

Triem  $f(n) = \log n$  i  $g(n) = n$  i apliquem la regla del límit:

①  $\log n \prec \sqrt{n}$ . Per la regla de L'Hôpital,

$$\lim_{n \rightarrow \infty} \frac{\log n}{\sqrt{n}} = \lim_{n \rightarrow \infty} \frac{1/(\ln 2 \cdot n)}{1/2 \cdot n^{-1/2}} = \frac{2}{\ln 2} \cdot \lim_{n \rightarrow \infty} \frac{n^{1/2}}{n} = \frac{2}{\ln 2} \cdot \lim_{n \rightarrow \infty} \frac{1}{n^{1/2}} = 0.$$

②  $\sqrt{n} \prec n$ . Trivialment,  $\lim_{n \rightarrow \infty} \frac{\sqrt{n}}{n} = \lim_{n \rightarrow \infty} \frac{1}{\sqrt{n}} = 0$ .

## Algorismes iteratius

### Càcul del cost:

- El cost d'una **operació elemental** és  $\Theta(1)$ . Això inclou:
  - una assignació entre tipus bàsics (`int`, `bool`, `double`, ...)
  - una lectura o escriptura d'un tipus bàsic
  - una comparació
  - una operació aritmètica
  - l'accés a un component d'un vector
  - el pas d'un paràmetre per referència
- Avaluat una **expressió** té cost igual a la suma dels costos de les operacions que s'hi fan (incloses les crides a les funcions, si n'hi ha).
- El cost de **construir o copiar un vector** de mida  $n$  (assignació, pas per valor, return) és  $\Theta(n)$ .

## Algorismes iteratius

### Càcul del cost:

- Si el cost d'un fragment  $F_1$  és  $C_1$  i el d'un fragment  $F_2$  és  $C_2$ , llavors el cost de la **composició seqüencial**

$F_1; F_2;$

és  $C_1 + C_2$ .

En general, si  $N$  és constant i el fragment  $F_k$  té cost  $C_k$ , el cost de la composició seqüencial

$F_1; F_2; \dots; F_k;$

és  $C_1 + C_2 + \dots + C_N$ .

## Algorismes iteratius

### Càcul del cost:

- Si el cost de  $F$  durant la  $k$ -èsima iteració és  $C_k$ , el d'avaluar  $B$  és  $D_k$  i el nombre d'iteracions és  $N$ , llavors el cost de la **composició iterativa**

`while (B) F;`

és  $(\sum_{k=1}^N C_k + D_k) + D_{N+1}$ .

# Algorismes iteratius

Càcul del cost:

- Si el cost d'un fragment  $F$  és  $C$  i el cost d'avaluar  $B$  és  $D$ , llavors el cost de la **composició alternativa d'una branca**

`if (B) F;`

és  $\leq D + C$ .

- Si el cost d'un fragment  $F_1$  és  $C_1$ , el d'un fragment  $F_2$  és  $C_2$  i el d'avaluar  $B$  és  $D$ , llavors el cost de la **composició alternativa de dues branques**

`if (B) F1; else F2;`

és  $\leq D + \max(C_1, C_2)$ .

## Exemple d'ordenació per selecció

Passos per ordenar la seqüència 5, 6, 1, 2, 0, 7, 4, 3 segons l'algorisme de selecció. En vermell, els elements ja ordenats. En blau, els elements intercanviats pel màxim.

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 5 | 6 | 1 | 2 | 0 | 7 | 4 | 3 |
| 5 | 6 | 1 | 2 | 0 | 3 | 4 | 7 |
| 5 | 4 | 1 | 2 | 0 | 3 | 6 | 7 |
| 3 | 4 | 1 | 2 | 0 | 5 | 6 | 7 |
| 3 | 0 | 1 | 2 | 4 | 5 | 6 | 7 |
| 2 | 0 | 1 | 3 | 4 | 5 | 6 | 7 |
| 1 | 0 | 2 | 3 | 4 | 5 | 6 | 7 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

## Ordenació per selecció

```
0 int posicio_maxim(const vector<int>& v, int m) {
1   int k = 0;
2   for (int i = 1; i <= m; ++i)
3     if (v[i] > v[k]) k = i;
4   return k; }

5 void ordena_seleccio (vector<int>& v, int n) {
6   for (int i = n-1; i > 0; --i) {
7     int k = posicio_maxim(v,i);
8     swap(v[k],v[i]); }}
```

2, 6 Iteracions bucles:  $m - 1 + 1 = m \in \Theta(m)$ ,  $(n - 1) - 1 + 1 = n - 1 \in \Theta(n)$ .

7 Cost  $\Theta(i)$ .

altres Instruccions de cost constant:  $\Theta(1)$ .

$$t_{sel}(n) = \Theta(1) + \sum_{i=1}^{n-1} (\Theta(i) + \Theta(1)) = \Theta(\sum_{i=1}^{n-1} i) = \Theta\left(\frac{(n-1)n}{2}\right) = \Theta(n^2)$$

## Exemple d'ordenació per inserció

Passos per ordenar la seqüència 5, 6, 1, 2, 0, 7, 4, 3 segons l'algorisme d'inserció. En vermell, els elements ja ordenats. En blau, el nombre de posicions que s'ha desplaçat l'element inserit.

|   |   |   |   |   |   |   |   |     |
|---|---|---|---|---|---|---|---|-----|
| 5 | 6 | 1 | 2 | 0 | 7 | 4 | 3 | (0) |
| 5 | 6 | 1 | 2 | 0 | 7 | 4 | 3 | (0) |
| 1 | 5 | 6 | 2 | 0 | 7 | 4 | 3 | (2) |
| 1 | 2 | 5 | 6 | 0 | 7 | 4 | 3 | (2) |
| 0 | 1 | 2 | 5 | 6 | 7 | 4 | 3 | (4) |
| 0 | 1 | 2 | 5 | 6 | 7 | 4 | 3 | (0) |
| 0 | 1 | 2 | 4 | 5 | 6 | 7 | 3 | (3) |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | (4) |

## Ordenació per inserció

```
0 void ordena_insercio(vector<int>& v, int n) {
1   for (int k = 1; k <= n-1; ++k) {
2     int t = k-1;
3     while (t >= 0 and v[t+1] < v[t]) {
4       swap(v[t], v[t+1]);
5       --t; }}
```

0 Pas de paràmetres:  $\Theta(1)$ .

1 Iteracions bucle:  $(n - 1) - 1 + 1 = n - 1 \in \Theta(n)$ .

1,2 Condició d'iteració i línia 2:  $\Theta(1)$ .

3 Iteracions bucle: entre 0  $\in \Theta(1)$  i  $k - 1 - 0 + 1 = k \in \Theta(k)$ .

4,5 Assignacions amb cost  $\Theta(1)$ .

$$\Theta(1) + (\Theta(n) \times \Theta(1)) \leq t_{ins}(n) \leq \Theta(1) + \sum_{k=1}^{n-1} \Theta(k)$$

Però sabem que

$$\sum_{k=1}^{n-1} k = 1 + 2 + \dots + (n - 1) = \frac{(n - 1)n}{2}.$$

Aleshores,

$$\sum_{k=1}^{n-1} \Theta(k) = \Theta\left(\sum_{k=1}^{n-1} k\right) = \Theta\left(\frac{(n - 1)n}{2}\right) = \Theta(n^2)$$

i, per tant,

$$\Theta(n) \leq t_{ins}(n) \leq \Theta(n^2).$$

## Algorismes recursius

El cost d'un algorisme recursiu s'expressa sovint en forma de recurrència.

### Definició

Una **recurrència** és una equació o una desigualtat que descriu una funció expressada en termes del seu valor per a entrades més petites.

### Exemple

$$C(n) = \begin{cases} 1, & \text{si } n = 1 \\ C(n-1) + n, & \text{si } n \geq 2 \end{cases}$$

Resoldre una recurrència vol dir donar-ne una forma tancada o, almenys, fites  $\Theta$  o  $O$  de la seva solució.

## Algorismes recursius

### Exemple

$$C(n) = \begin{cases} 1, & \text{si } n = 1 \\ C(n-1) + n, & \text{si } n \geq 2 \end{cases}$$

### Idea

Podem observar que

- $C(1) = 1$
- $C(2) = 1 + 2 = 3$
- $C(3) = 3 + 3 = 6$
- $C(n) = C(n-1) + n = C(n-2) + (n-1) + n = \dots$

### Solució

$$\begin{aligned} C(n) &= C(n-1) + n \\ &= C(n-2) + (n-1) + n \\ &= C(n-3) + (n-2) + (n-1) + n \\ &\vdots \\ &= C(1) + 2 + \dots + (n-2) + (n-1) + n \\ &= 1 + 2 + \dots + n \\ &= \sum_{i=1}^n i = \frac{n(n+1)}{2} \in \Theta(n^2). \end{aligned}$$

## Algorismes recursius

Per descriure una recurrència que expressi el cost d'un algorisme recursiu, n'hi ha prou a determinar:

- el **paràmetre de recursió**  $n$ ,
- el **cost del cas base** ( $n = 0, n = 1, \dots$ )
- el **cost del cas inductiu**
  - nombre de crides recursives
  - valor del paràmetre recursiu en les crides
  - cost dels càculs addicionals no recursius

### Cerca lineal recursiva

Comprovar si un nombre  $x$  apareix en un vector  $v$  entre les posicions 0 i  $n-1$  comparant-lo amb  $v[0], v[1], \dots, v[n-1]$ .

Si es troba  $x$ , retornar la seva posició en  $v$ . Altrament, retornar -1.

```
int cerca_lineal(const vector<int>& v, int n, int x) {
    if (n == 0) return -1;
    else if (v[n-1] == x) return n-1;
    else return cerca_lineal(v, n-1, x);
}
```

El paràmetre de recursió és  $n$ , la mida del vector. Definim la recurrència  $T(n)$  que representa el cost (en cas pitjor) de l'algorisme:

$$T(n) = T(n-1) + \Theta(1)$$

### Recurrència per a la cerca lineal

$$T(n) = T(n-1) + \Theta(1) \text{ per a } n \geq 1, \text{ i}$$

$$T(0) = \Theta(1).$$

### Solució

$$\begin{aligned} T(n) &= T(n-1) + \Theta(1) \\
&= T(n-2) + 2 \cdot \Theta(1) \\
&= T(n-3) + 3 \cdot \Theta(1) \\
&\vdots \\
&= T(0) + n \cdot \Theta(1) \\
&= (n+1) \cdot \Theta(1) \\
&= \Theta(n+1) = \Theta(n). \end{aligned}$$

## Algorismes recursius

### Cerca binària recursiva

Comprovar si un nombre  $x$  apareix en un vector ordenat  $v$  entre les posicions  $i$  i  $j$  per cerca binària.

Si es troba  $x$ , retornar la seva posició en  $v$ . Altrament, retornar  $-1$ .

```
int cerca_binaria(const vector<int>& v, int i, int j, int x)
{ if (i <= j) {
    int k = (i + j) / 2;
    if (x == v[k])
        return k;
    else if (x < v[k])
        return cerca_binaria(v, i, k-1, x);
    else
        return cerca_binaria(v, k+1, j, x);
}
else return -1;
}
```

El paràmetre de recursió és  $n = j - i$ , la mida de l'interval a explorar.

Definim la recurrència  $T(n)$ , el cost (en cas pitjor) de l'algorisme:

$$T(n) = T(n/2) + \Theta(1)$$

### Recurrència per a la cerca binària

$$T(n) = T(n/2) + \Theta(1) \text{ per a } n \geq 1, \quad i$$

$$T(0) = \Theta(1).$$

### Solució

$$\begin{aligned} T(n) &= T(n/2) + \Theta(1) \\ &= T(n/4) + 2 \cdot \Theta(1) \\ &= T(n/8) + 3 \cdot \Theta(1) \\ &\vdots \\ &= T(n/2^{\log n}) + \log n \cdot \Theta(1) \\ &= T(1) + \log n \cdot \Theta(1) \\ &= T(0) + (\log n + 1) \cdot \Theta(1) \\ &= (\log n + 2) \cdot \Theta(1) = \Theta(\log n + 2) = \Theta(\log n). \end{aligned}$$

## Teoremes mestres

Per sistematitzar l'anàlisi del cost dels algorismes recursius, els classifiquem en dos grups en funció de com divideixen el problema d'entrada en subproblems en les crides recursives.

Sigui  $A$  un algorisme que, amb una entrada de mida  $n$ , fa  $a$  crides recursives i una feina addicional no recursiva de cost  $g(n)$ . Llavors, si en les crides recursives els subproblems tenen mida

- $n - c$ , el cost d' $A$  ve descrit per la recurrència

$$T(n) = a \cdot T(n - c) + g(n)$$

- $n/b$ , el cost d' $A$  ve descrit per la recurrència

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

Les dues menes de recurrències anteriori:

- **substractives**:  $T(n) = a \cdot T(n - c) + g(n)$
- **divisores**:  $T(n) = a \cdot T(n/b) + g(n)$

es poden resoldre amb els **teoremes mestres** que veurem a continuació.

### Teorema mestre de recurrències substractives

$$\text{Sigui } T(n) = \begin{cases} f(n), & \text{si } 0 \leq n < n_0 \\ a \cdot T(n - c) + g(n), & \text{si } n \geq n_0 \end{cases}$$

on  $n_0 \in \mathbb{N}$ ,  $c \geq 1$ ,  $f$  és una funció arbitrària i  $g \in \Theta(n^k)$  per a  $k \geq 0$ .

Aleshores

$$T(n) \in \begin{cases} \Theta(n^k), & \text{si } a < 1 \\ \Theta(n^{k+1}), & \text{si } a = 1 \\ \Theta(a^{n/c}), & \text{si } a > 1 \end{cases}$$

### Exemple 1

Hem vist que el cost de l'algorisme recursiu de **cerca lineal** es pot descriure amb la recurrència  $T(n) = T(n - 1) + \Theta(1)$  per a  $n \geq 1$ , i  $T(0) = \Theta(1)$ .

Per tant,  $n_0 = 1$ ,  $a = 1$ ,  $c = 1$ ,  $k = 0$ . Llavors,  $T(n)$  pertany al segon cas:

$$T(n) \in \Theta(n^{k+1}) = \Theta(n).$$

## Exemple 2

En la recurrència  $T(n) = T(n - 1) + \Theta(n)$ , tenim els valors

$$a = 1, c = 1, k = 1.$$

Llavors,  $T(n)$  pertany al segon cas:

$$T(n) \in \Theta(n^{k+1}) = \Theta(n^2).$$

## Exemple 3

En la recurrència  $T(n) = 2 \cdot T(n - 1) + \Theta(n)$ , tenim els valors

$$a = 2, c = 1, k = 1.$$

Llavors,  $T(n)$  pertany al tercer cas:

$$T(n) \in \Theta(2^n).$$

## Exemple 4

Els nombres de Fibonacci estan definits per la recurrència

$$f(k) = f(k - 1) + f(k - 2) \text{ per a } k \geq 2, \text{ amb } f(0) = f(1) = 1.$$

La solució recursiva és evident.

```
int fibonacci (int k) {
    if (k <= 1) return 1;
    else return fibonacci(k-1) + fibonacci(k-2);
}
```

- El cost segueix la recurrència  $T(k) = T(k - 1) + T(k - 2) + \Theta(1)$
- No podem aplicar directament el teorema mestre per resoldre-la!

Podem aplicar el teorema mestre a dues fites de  $T(k)$ :

- $T(k) = T(k - 1) + T(k - 2) + \Theta(1) \leq 2T(k - 1) + \Theta(1)$  dona  $T(k) \in O(2^k)$
- $T(k) = T(k - 1) + T(k - 2) + \Theta(1) \geq 2T(k - 2) + \Theta(1)$  dona  $T(k) \in \Omega(2^{k/2}) = \Omega(\sqrt{2}^k)$

Es pot demostrar que  $T(k) = \Theta(\phi^k)$ , on  $\phi = \frac{1+\sqrt{5}}{2}$  (*nombre d'or*).

Noteu que  $\sqrt{2} = 1.414213562\dots$  i  $\phi = 1.618033988\dots$

## Exemple 2

Hem vist que el cost de l'**ordenació per fusió** es pot descriure amb la recurrència  $T(n) = 2T(n/2) + \Theta(n)$  per a  $n \geq 2$  i  $T(1) = \Theta(1)$ .

Per tant,  $n_0 = 2$ ,  $a = 2$ ,  $b = 2$ ,  $k = 1$ ,  $\alpha = 1$ . Llavors,  $T(n)$  pertany al 2n cas:

$$T(n) \in \Theta(n^k \log n) = \Theta(n \log n).$$

Examen final 16/01/2023

①(a)

```
void f(int x) {
    if (x != 0) {
        f(x/2);
        cout << x%2;
    }
}
```

$$\begin{aligned} C(n) &= C(n-1) + \Theta(1) \\ T. \text{ mestre} &\Rightarrow C(n) \in \Theta(n) \end{aligned}$$

$\xrightarrow{\quad /2 \quad} \quad \xrightarrow{\quad \%2 \quad}$

$10110 \xrightarrow{\quad \%2 \quad} 0$

Segui  $x \in \mathbb{N}$  i  $n = |x|$   
Quin és el cost de  $f$  en funció de  $n$ ?

Teorema mestre de recurrències divisores

$$\text{Sigui } T(n) = \begin{cases} f(n), & \text{si } 0 \leq n < n_0 \\ a \cdot T(n/b) + g(n), & \text{si } n \geq n_0 \end{cases}$$

on  $n_0 \in \mathbb{N}$ ,  $b > 1$ ,  $f$  és una funció arbitrària i  $g \in \Theta(n^k)$  per a  $k \geq 0$ .

Sigui  $\alpha = \log_b(a)$ . Aleshores,

$$T(n) \in \begin{cases} \Theta(n^k), & \text{si } \alpha < k \\ \Theta(n^k \log n), & \text{si } \alpha = k \\ \Theta(n^\alpha), & \text{si } \alpha > k \end{cases}$$

## Exemple 1

Hem vist que el cost de l'algorisme recursiu de **cerca binària** es pot descriure amb  $T(n) = T(n/2) + \Theta(1)$ ,  $n \geq 1$ , i  $T(0) = \Theta(1)$ .

Per tant,  $n_0 = 1$ ,  $a = 1$ ,  $b = 2$ ,  $k = 0$ ,  $\alpha = 0$ . Llavors,  $T(n)$  pertany al 2n cas:

$$T(n) \in \Theta(n^k \log n) = \Theta(\log n).$$

## Exemple 2

Funció principal de l'ordenació per fusió (*mergesort*)

```
template <typename elem>
void mergesort(vector<elem>& v, int e, int d) {
    if (e < d) {
        int m = (e + d) / 2;
        mergesort(v, e, m);
        mergesort(v, m + 1, d);
        merge(v, e, m, d);
    }
}
```

Tenint en compte que el cost de la crida `merge(v, e, m, d)` és  $\Theta(n)$  (on  $n = d - e + 1$ ), el cost total es pot expressar amb la recurrència:

$$T(n) = 2T(n/2) + \Theta(n) \text{ per a } n \geq 2, \text{ i } T(1) = \Theta(1).$$

## Teoremes mestres

### Exercici 1

Resoleu la recurrència  $T(n) = T(\sqrt{n}) + 1$ .

### Solució

Fem el canvi de variable  $m = \log n$ . Aleshores,

$$T(n) = T(2^m) = T(2^{m/2}) + 1.$$

Definim  $S(m) = T(2^m)$ , que compleix

$$S(m) = S(m/2) + 1.$$

Pel segon teorema mestre, tenim que  $S(m) \in \Theta(\log m)$  i, per tant:

$$T(n) = T(2^m) = S(m) \in \Theta(\log m) = \Theta(\log \log n).$$

### Exercici 2

Resoleu la recurrència  $T(n) = 2T(\sqrt{n}) + \log n$ .

## Fita inferior dels algorismes d'ordenació

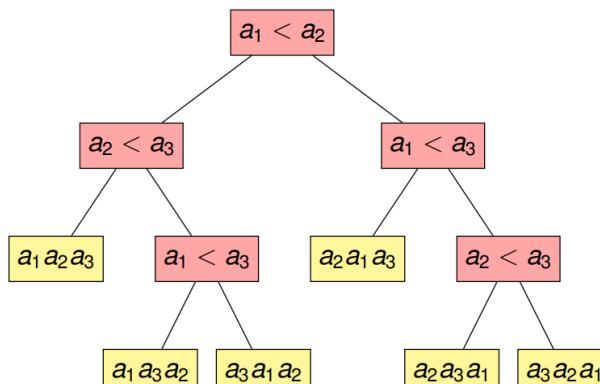
En termes de cost asimptòtic, l'algorisme d'ordenació per fusió és òptim:

### Proposició

Tot algorisme d'ordenació basat en comparacions té cost  $\Omega(n \log n)$ .

Es pot argumentar fent servir arbres per representar els algorismes d'ordenació basats en comparacions.

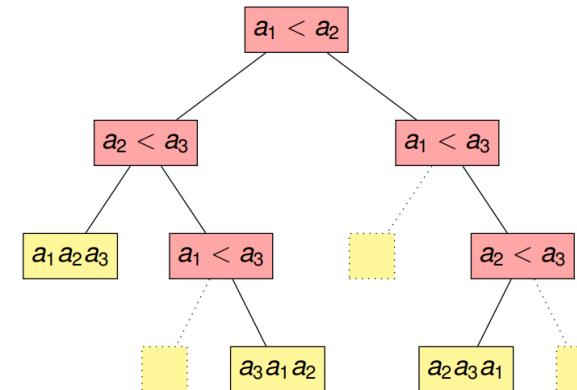
Suposem que volem ordenar  $a_1$ ,  $a_2$  i  $a_3$ . Si  $a_1 < a_2$ , seguim per la branca esquerra; si no, per la dreta. Els rectangles grocs representen les ordenacions trobades. **L'alçària de l'arbre és el cost en cas pitjor.**



## Fita inferior dels algorismes d'ordenació

Considerem un arbre que ordena  $n$  elements:

- cada fulla correspon a una permutació de  $\{1, 2, \dots, n\}$
- cada permutació de  $\{1, 2, \dots, n\}$  ha d'aparèixer en alguna fulla  
(si una no hi fos, què passaria si es donés com a entrada?)



- com que hi ha  $n!$  permutacions de  $n$  elements, l'arbre té  $\geq n!$  fulles
- tot arbre binari amb  $\geq k$  fulles té alçària  $\geq \log k$
- per tant, **l'alçària del nostre arbre és almenys de  $\log n!$**

El cost de l'algorisme representat per l'arbre és, per tant,  $\Omega(\log n!)$ . Com que

$$n! \geq n \cdot (n-1) \cdot (n-2) \cdot \dots \cdot \lfloor n/2 \rfloor \geq (n/2)^{(n/2)}$$

tenim que

$$\log n! \geq \log(n/2)^{(n/2)} = \frac{n}{2} \log(n/2) \in \Omega(n \log n).$$

### Proposició

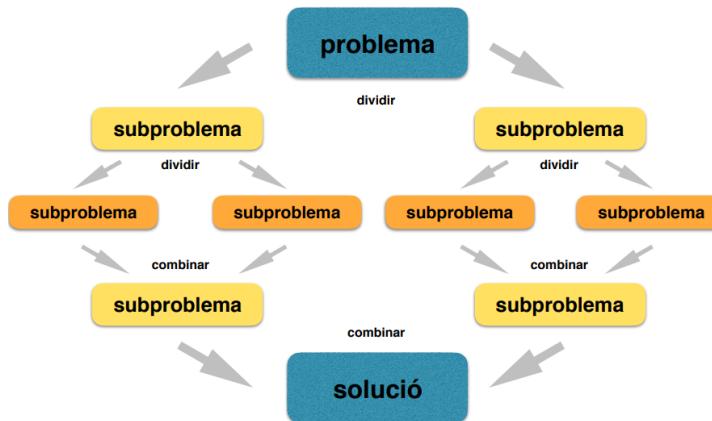
Tot algorisme d'ordenació basat en comparacions té cost  $\Omega(n \log n)$ .

## Dividir i vèncer

L'estratègia **dividir i vèncer** resol un problema en tres passos:

- ➊ **dividint-lo** en *subproblems*, exemples més petits del mateix problema
- ➋ **resolent** els subproblems recursivament
- ➌ **combinant** les solucions de manera adequada

## Dividir i vèncer



La feina es fa, per tant, en tres parts: (1) en la divisió en subproblemes, (2) al final de la recursió i (3) en la combinació de les solucions.

Els algorismes de [dividir i vèncer](#) segueixen sovint una mateixa estratègia. Ataquen un problema de mida  $n$

- dividint-lo en  $a$  subproblemes de mida  $n/b$ ,
- resolent els subproblemes recursivament i
- combinant les respostes,

on  $a \geq 1$ ,  $b > 1$ , i el cost de dividir en subproblemes i combinar respostes és  $\Theta(n^k)$  per a  $k \geq 0$ .

Per tant, el cost de l'algorisme es pot descriure mitjançant la recurrència

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

que es pot resoldre aplicant el [teorema mestre de recurrències divisores](#).

### Teorema mestre de recurrències divisores

$$\text{Sigui } T(n) = \begin{cases} f(n), & \text{si } 0 \leq n < n_0 \\ a \cdot T(n/b) + g(n), & \text{si } n \geq n_0 \end{cases}$$

on  $n_0 \in \mathbb{N}$ ,  $b > 1$ ,  $f$  és una funció arbitrària i  $g \in \Theta(n^k)$  per a  $k \geq 0$ .

Sigui  $\alpha = \log_b(a)$ . Aleshores,

$$T(n) \in \begin{cases} \Theta(n^k), & \text{si } \alpha < k \\ \Theta(n^k \log n), & \text{si } \alpha = k \\ \Theta(n^\alpha), & \text{si } \alpha > k \end{cases}$$

## Algorisme de fusió bàsic

L'[ordenació per fusió](#), o *mergesort*, és un bon exemple de l'esquema [dividir i vèncer](#) que fa servir un nombre de comparacions gairebé òptim.

És un algorisme estable en un doble sentit:

- preserva l'ordre entre valors iguals
- es comporta de manera semblant amb independència del grau d'ordenació de l'entrada

Donat un vector  $T$  de talla  $\geq 2$ , l'algorisme consisteix a:

- ➊ Partir  $T$  en dues meitats
- ➋ Ordenar recursivament la meitats de  $T$  per separat
- ➌ Retornar la fusió de les dues meitats

L'operació clau (punt 3) consisteix a [fusionar](#) dos vectors ordenats en un.

### Exemple de fusió

| entrada           | E | X | E | M | P | L | E | F | U | S | I | O |
|-------------------|---|---|---|---|---|---|---|---|---|---|---|---|
| ordenar 1a meitat | E | E | L | M | P | X | E | F | U | S | I | O |
| ordenar 2a meitat | E | E | L | M | P | X | E | F | I | O | S | U |
| resultat fusió    | E | E | E | F | I | L | M | O | P | S | U | X |

## Algorisme de fusió bàsic

### Ordenació per fusió (*Algorismes en C++, EDA*)

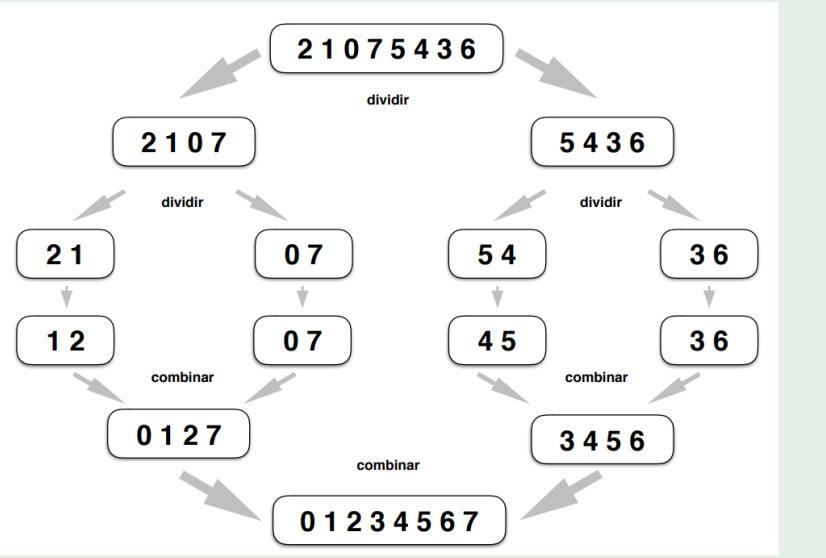
```

template <typename elem>
void mergesort (vector<elem>& T) {
    mergesort(T, 0, T.size() - 1);
}

template <typename elem>
void mergesort (vector<elem>& T, int e, int d) {
    if (e < d) {
        int m = (e + d) / 2;
        mergesort(T, e, m);
        mergesort(T, m + 1, d);
        merge(T, e, m, d);
    }
}
  
```

## Algorisme de fusió bàsic

Exemple (aquí, la recursió acaba per a talla 2, en l'algorisme és per a 1)



## Algorisme de fusió bàsic

```
template <typename elem>
void merge (vector<elem>& T, int e, int m, int d) {
    vector<elem> B(d-e+1);
    int i = e, j = m + 1, k = 0;
    while (i <= m and j <= d) {
        if (T[i] <= T[j]) B[k++] = T[i++];
        else B[k++] = T[j++];
    }
    while (i <= m) B[k++] = T[i++];
    while (j <= d) B[k++] = T[j++];
    for (k = 0; k <= d-e; ++k) T[e+k] = B[k];
}
```

## Exemple

|               |               |
|---------------|---------------|
| 0   1   2   7 | 3   4   5   6 |
| 0             |               |

## Exemple

|                           |               |
|---------------------------|---------------|
| 0   1   2   7             | 3   4   5   6 |
| 0   1   2   3   4   5   6 |               |

Donat que el procediment `merge` és lineal, el cost de l'ordenació per fusió es pot expressar fàcilment amb la recurrència

$$T(n) = \begin{cases} \Theta(1), & \text{si } n = 1 \\ 2T(n/2) + \Theta(n), & \text{si } n > 1 \end{cases}$$

i, aplicant el teorema mestre de recurredades divisores, tenim que

$$T(n) \in \Theta(n \log n).$$

## Variants

Ordenació per fusió amb inserció per a vectors petits (*Alg. en C++, EDA*)

```
template <typename elem>
void mergesort (vector<elem>& T, int e, int d) {
    const int talla_critica = 50;
    if (d - e < talla_critica)
        ordena_insercio(T, e, d);
    else {
        int m = (e + d) / 2;
        mergesort(T, e, m);
        mergesort(T, m + 1, d);
        merge(T, e, m, d);
    }
}
```

Les dues **versions iteratives** que veurem parteixen del fet que les fusions només comencen al final de la recursió, de manera que:

- comencen directament pels elements a ordenar i
- arriben al vector ordenat mitjançant fusions.

## Observació

Cada comparació afegeix un element a la taula *B* excepte l'última, que n'afegeix almenys dos.

- Per tant, el nombre de **comparacions** de tipus `elem` és  $< n = d - e + 1$
- El nombre d'**assignacions** de tipus `elem` és  $2n$
- El cost és **lineal** (assumint que assignar un `elem` és  $\Theta(1)$ )

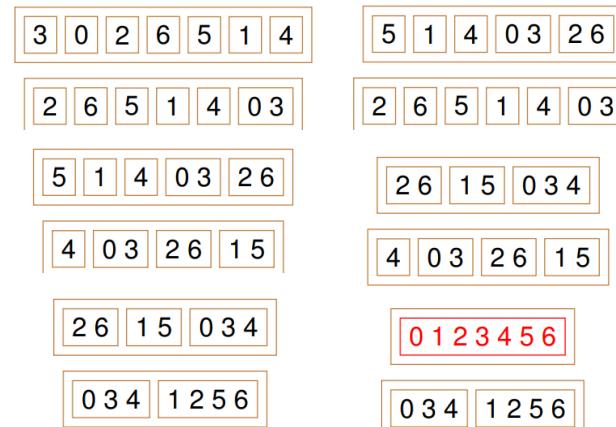
## Variants

### Ordenació per fusió iterativa 1

Versió del llibre *Algorithms* de Dasgupta/Papadimitriou/Vazirani en pseudocodi (pàg. 51). Es fa servir el TAD cua amb operacions

- `inject(Q, e)`: afegir l'element `e` a la cua `Q` i
- `eject(Q)`: funció que extreu i retorna l'últim element de `Q`

```
function mergesort_queue(a[1...n])
    Q = [] (cua buida)
    for i=1 to n:
        inject(Q, a[i])
    while |Q| > 1:
        inject(Q, merge(eject(Q), eject(Q)))
    return eject(Q)
```



## Algorisme general

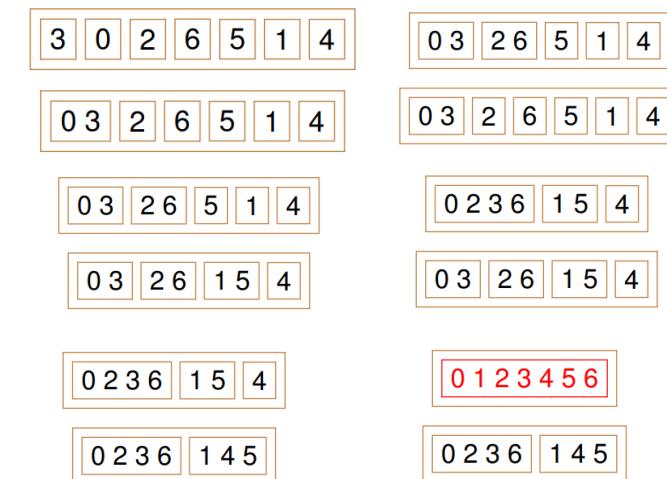
Donat un vector  $T$  d'almenys 2 elements, l'**algorisme bàsic** fa el següent:

- ➊ Tria un element  $x$  de  $T$
- ➋ Dividir  $T - \{x\}$  en dos grups disjunts:
  - $T_1$ , que conté elements  $\leq x$  de  $T$
  - $T_2$ , que conté elements  $\geq x$  de  $T$
- ➌ Ordenar  $T_1$  i  $T_2$  recursivament
- ➍ Retornar  $T_1$  seguit de  $T_2$

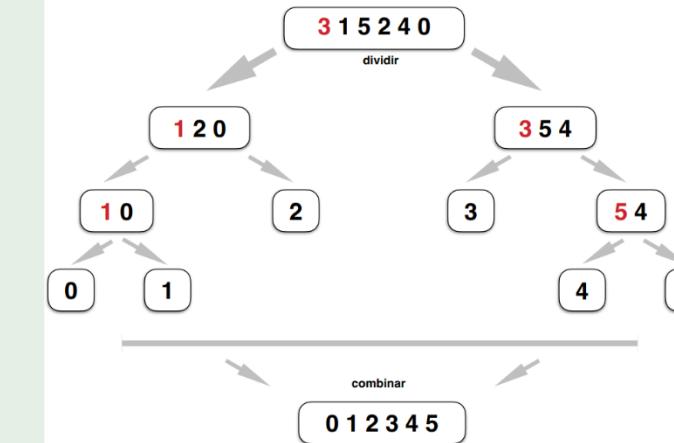
## Variants

### Ordenació per fusió iterativa 2 (Algorismes en C++, EDA)

```
template <typename elem>
void mergesort_bottom_up (vector<elem>& T) {
    int n = T.size();
    for (int m = 1; m < n; m *= 2) {
        for (int i = 0; i < n-m; i += 2*m) {
            merge(T, i, i+m-1, min(i+2*m-1, n-1));
        }
    }
}
```



### Exemple



## Algorisme general

Ordenació ràpida (*Algorismes en C++, EDA*)

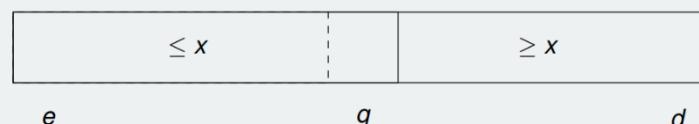
```

void quicksort(vector<elem>& T) {
    quicksort(T, 0, T.size() - 1); }

template <typename elem>
void quicksort(vector<elem>& T, int e, int d) {
    if (e < d) {
        int q = partition(T, e, d);
        quicksort(T, e, q);
        quicksort(T, q + 1, d); } }
```

**q = partition(T, e, d)**

- **Precondició:**  $0 \leq e \leq d \leq T.size() - 1$
- **Postcondició:**  $\exists x$  (pivot)  $\forall i$ 
  - si  $e \leq i \leq q$ , tenim que  $T[i] \leq x$
  - si  $q < i \leq d$ , tenim que  $T[i] \geq x$



## Comparació entre ordenació per fusió i ràpida

- **Paral·lelismes** amb l'ordenació per fusió:

- resol dos subproblemes
- fa un treball addicional lineal

- **Estratègies oposades:**

- **Ordenació per fusió:**

- divisió en subproblemes trivial
- combinació dels vectors feta amb cura

- **Ordenació ràpida:**

- divisió en subproblemes feta amb cura
- combinació de vectors trivial

## Partició de Hoare

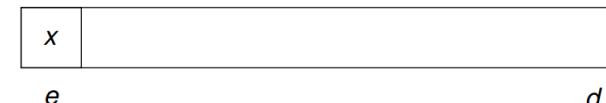
Partició original de Hoare amb el primer element com a pivot.

Partició de Hoare (*Algorismes en C++, EDA*)

```

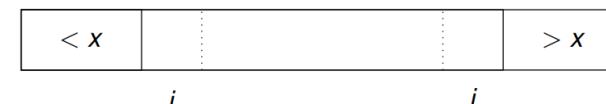
template <typename elem>
int partition (vector<elem>& T, int e, int d) {
    elem x = T[e];
    int i = e - 1;
    int j = d + 1;
    for (;;) {
        while (x < T[--j]);
        while (T[++i] < x);
        if (i >= j) return j;
        swap(T[i], T[j]);
    } }
```

- Inici de la funció `partition(T, e, d)`:

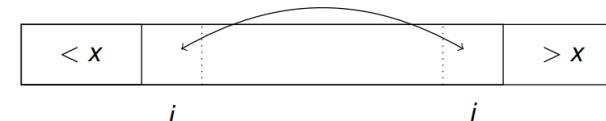


- Bucle principal:

- es troben els valors de  $i, j$  més centrals tals que



- s'intercanvien els continguts de les posicions  $i, j$



## Variants

Altres diferències amb l'ordenació per fusió:

- Els subproblemes no sempre tenen la mateixa mida
- La mida dels subproblemes depèn de la tria del **pivot** (element inicial)

Considerem tres estratègies per a l'elecció del pivot:

### 1 Tria el primer element:

- és acceptable si l'entrada és aleatòria
- si l'entrada està ordenada (en ordre creixent o decreixent), l'algorisme pren temps  $\Theta(n^2)$  per no fer res

### 2 Tria un element aleatori

- en mitjana divideix el problema en subproblemes semblants
- no sempre fa l'algorisme més ràpid  
(generar nombres aleatoris pot ser costós)

### 3 Tria la mediana de tres elements

- la millor opció seria triar la mediana del vector, però és massa cara
- una bona estimació és fer la mediana de 3, normalment el 1r, l'element del mig i l'últim

### Ordenació ràpida amb la mediana de tres com a pivot

```
template <typename elem>
void quicksort (vector<elem>& T, int e, int d) {
    if (e < d) {
        int centre = (e + d) / 2;
        if (T[e] < T[centre]) swap(T[centre], T[e]);
        if (T[d] < T[centre]) swap(T[centre], T[d]);
        if (T[d] < T[e]) swap(T[e], T[d]);
        // el pivot es a la posició e
        int q = partition(T, e, d);
        quicksort(T, e, q);
        quicksort(T, q + 1, d);
    }
}
```

Després de les línies 1, 2 i 3, tenim

$$T[\text{centre}] \leq T[e], T[\text{centre}] \leq T[d], T[e] \leq T[d].$$

Per tant,  $T[\text{centre}] \leq T[e] \leq T[d]$  i la mediana és  $T[e]$ .

Per a vectors molt petits, l'ordenació per **inserció** es comporta millor que **quicksort**. Una bona solució, doncs, és tallar la recursió quan el vector és més petit que una certa mida (normalment, entre 5 i 20).

### Ordenació ràpida amb inserció per a vectors petits

```
template <typename elem>
void quicksort (vector<elem>& T, int e, int d) {
    const int talla_critica = 20;
    if (d - e < talla_critica)
        ordena_insercio(T, e, d);
    else {
        int q = partition(T, e, d);
        quicksort(T, e, q);
        quicksort(T, q + 1, d);
    }
}
```

### Recurrència

Sigui  $T(n)$  el cost de l'algorisme d'ordenació ràpida amb  $n$  elements.

Llavors,

$$T(n) = \begin{cases} \Theta(1), & \text{si } n \leq 1 \\ T(i) + T(n-i) + \Theta(n), & \text{si } n > 1 \end{cases}$$

on  $i$  és el nombre d'elements de la primera meitat i  $n - i$  de la segona.

### Anàlisi: cas pitjor

#### Cas general

$$T(n) = T(i) + T(n-i) + \Theta(n)$$

En el **cas pitjor**, el pivot és sempre l'element més petit o el més gran. Com a resultat, una crida recursiva té cost constant i l'altra  $T(n-1)$ .

#### Cas pitjor

$$T(n) = T(n-1) + \Theta(n)$$

Resolent la recurrència directament (o aplicant el primer teorema mestre), s'obté:

$$T(n) \in \Theta(n^2).$$

## Anàlisi: cas millor

### Cas general

$$T(n) = T(i) + T(n-i) + \Theta(n)$$

En el **cas millor**, el pivot és la mediana del vector  $i$ , per tant, els dos sub-vectors tenen la mateixa mida

### Cas millor

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

És la mateixa recurrència de l'ordenació per fusió que, pel segon teorema mestre, dona el cost:

$$T(n) \in \Theta(n \log n).$$

## Anàlisi: cas mitjà

### Cas general

$$T(n) = T(i) + T(n-i) + \Theta(n)$$

Per al **cas mitjà**, suposarem que la partició és aleatòria.

Hi ha una probabilitat  $\frac{1}{n-1}$  de tenir una meitat de mida  $i$  per a  $1 \leq i \leq n-1$ .

El valor mitjà de  $T(i)$  i de  $T(n-i)$  serà de  $\frac{1}{n} \sum_{j=1}^{n-1} T(j)$ .

### Cas mitjà

Per a alguna constant  $c$ ,

$$T(n) = \frac{2}{n-1} \left[ \sum_{j=1}^{n-1} T(j) \right] + cn \quad (1)$$

## Anàlisi: cas mitjà

Multipliquem l'equació 1 per  $n-1$ :

$$(n-1)T(n) = 2 \left[ \sum_{j=1}^{n-1} T(j) \right] + cn(n-1) \quad (2)$$

Per eliminar el sumatori, escrivim el cas  $n+1$ :

$$nT(n+1) = 2 \left[ \sum_{j=1}^n T(j) \right] + c(n+1)n \quad (3)$$

i restem l'equació (2) de la (3):

$$nT(n+1) - (n-1)T(n) = 2T(n) + 2cn. \quad (4)$$

Reordenant els termes,

$$nT(n+1) = (n+1)T(n) + 2cn. \quad (5)$$

Dividim l'equació 5 per  $n(n+1)$ :

$$\frac{T(n+1)}{n+1} = \frac{T(n)}{n} + \frac{2c}{n+1}. \quad (6)$$

Substituïm  $n$  per tots els valors des de  $n-1$  fins a 1:

$$\frac{T(n)}{n} = \frac{T(n-1)}{n-1} + \frac{2c}{n} \quad (7)$$

$$\frac{T(n-1)}{n-1} = \frac{T(n-2)}{n-2} + \frac{2c}{n-1} \quad (8)$$

$$\vdots$$

$$\frac{T(2)}{2} = \frac{T(1)}{1} + \frac{2c}{2}. \quad (9)$$

La suma de totes les equacions (7)–(9) dona

$$\frac{T(n)}{n} = \frac{T(1)}{1} + 2c \sum_{i=2}^n \frac{1}{i}. \quad (10)$$

D'altra banda, se sap que

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

on  $\gamma \approx 0.577$  és la constant d'Euler.

Substituint aquest valor en l'equació (10) obtinguda abans

$$\frac{T(n)}{n} = \frac{T(1)}{1} + 2c \sum_{i=2}^n \frac{1}{i} \quad \text{es dedueix que}$$

i, per tant,

$$\frac{T(n)}{n} \in \Theta(\log n)$$

$$T(n) \in \Theta(n \log n).$$

Examen 10/01/2022

4)

Donat un vector  $v$  de  $n$  enters ordenats de forma no decreixent i un enter  $x$ , volem determinar si  $x$  apareix a  $v$ .

Si hi és, també volem la seva posició.

Sabent que  $x$  apareix a  $v$ , volem millorar el temps de la cerca binària ( $\log n$ ) a  $\log \log n$  on  $i$

és la pos de la 1a aparició de  $x$  a  $v$ .

(a) Completar

```
int bin-search (int x, const vector<int> &v,  
                int l, int r);  
  
int exp-search (int x, const vector<int> &v)  
{  
    int n=v.size();  
    if (n==0) return -1;  
    int b=1;  
    while ((   and   ) b*=2;  
    return bin-search(x,v,b/2,   );  
}
```

## Algorisme de Karatsuba

El matemàtic rus A. Kolmogorov va conjecturar que l'algorisme escolar de multiplicació és òptim.

Conjectura (Kolmogorov, 1952)

Qualsevol algorisme per multiplicar dos nombres de  $n$  díigits té cost  $\Omega(n^2)$ .

Un estudiant de 23 anys de Kolmogorov, Anatolii Alexeevitch Karatsuba, va trobar un algorisme de cost  $\Theta(n^{1.585})$ .

Refutació (Karatsuba, 1960)

Hi ha un algorisme que multiplica dos nombres de  $n$  díigits en temps

$$\Theta(n^{\log_2 3}) \approx \Theta(n^{1.585}).$$

### Observació

El matemàtic Carl Friedrich Gauss (1777-1855) va observar que malgrat que el producte de dos complexos

$$(a+bi)(c+di) = ac - bd + (bc + ad)i$$

sembla requerir 4 productes de reals, es pot obtenir només amb 3.

La raó és que

$$bc + ad = (a+b)(c+d) - ac - bd.$$

## Algorisme de Karatsuba

Avancem ara al s. XX.

Suposem que  $x$  i  $y$  són dos naturals de  $n$  bits. Expressem  $x$ ,  $y$  en dues parts:

$$x = \boxed{x_E} \boxed{x_D} = 2^{\lfloor n/2 \rfloor} x_E + x_D$$

$$y = \boxed{y_E} \boxed{y_D} = 2^{\lfloor n/2 \rfloor} y_E + y_D$$

### Exemple

Si  $x = 10010111_2$  i  $y = 11001010_2$  (el subíndex 2 vol dir "en binari"), llavors

$$x = \boxed{x_E} \boxed{x_D} = \boxed{1001}_2 \boxed{0111}_2$$

$$y = \boxed{y_E} \boxed{y_D} = \boxed{1100}_2 \boxed{1010}_2$$

Ara, el producte

$$xy = (2^{\lfloor n/2 \rfloor} x_E + x_D)(2^{\lfloor n/2 \rfloor} y_E + y_D)$$

quan  $n$  és parell es pot reescriure com

$$xy = 2^n x_E y_E + 2^{\lfloor n/2 \rfloor} (x_E y_D + x_D y_E) + x_D y_D \quad (11)$$

i quan  $n$  és senar com

$$xy = 2^{n-1} x_E y_E + 2^{\lfloor n/2 \rfloor} (x_E y_D + x_D y_E) + x_D y_D.$$

Un algorisme basat en aquesta expressió tindria cost

$$T(n) = 4T(n/2) + \Theta(n).$$

Pel teorema mestre de recurrències divisores, sabem que  $T(n) \in \Theta(n^2)$ .

Però si apliquem el **truc de Gauss**, podem obtenir el producte  $xy$  fent servir 3 subproductes en lloc de 4 i, així, rebaixar el cost quadràtic.

## Algorisme de Karatsuba

Com abans, observem que

$$x_E y_D + x_D y_E = (x_E + x_D)(y_E + y_D) - x_E y_E - x_D y_D.$$

Si ara anomenem

$$\mathbf{a} = x_E y_E, \quad \mathbf{b} = x_D y_D, \quad \mathbf{c} = (x_E + x_D)(y_E + y_D),$$

llavors el producte per a  $n$  parell (l'equació 11)

$$xy = 2^n x_E y_E + 2^{n/2}(x_E y_D + x_D y_E) + x_D y_D$$

es pot reescriure com

$$2^n \mathbf{a} + 2^{n/2}(\mathbf{c} - \mathbf{a} - \mathbf{b}) + \mathbf{b}$$

que només depèn de 3 subproductes (com el cas de  $n$  senar).

La nova expressió dona lloc a un algorisme de cost

$$T(n) = 3T(n/2) + \Theta(n)$$

i, pel teorema mestre de recurredències divisores, sabem que

$$T(n) \in \Theta(n^{\log_2 3}) \approx \Theta(n^{1.585}).$$

## Algorisme de Karatsuba

### Solució de Karatsuba (cas parell, base 10)

Donat  $n$  parell,  $x = (10^{n/2}x_E + x_D)$  i  $y = (10^{n/2}y_E + y_D)$ , tenim que

$$xy = 10^n \mathbf{a} + 10^{n/2}(\mathbf{c} - \mathbf{a} - \mathbf{b}) + \mathbf{b}$$

on  $\mathbf{a} = x_E y_E$ ,  $\mathbf{b} = x_D y_D$ ,  $\mathbf{c} = (x_E + x_D)(y_E + y_D)$ .

### Exemple: calcular $1234 * 4321$

- Problema: calcular  $x * y$  per a  $x = 1234$ ,  $y = 4321$

#### • Subproblemes:

- 1 Calcular  $a = 12 * 43$
- 2 Calcular  $b = 34 * 21$
- 3 Calcular  $c = (12 + 34) * (43 + 21) = 46 * 64$

### Subproblema 1: calcular $a = 12 * 43$

#### • Subproblemes:

- 1 Calcular  $a_1 = 1 * 4 = 4$
- 2 Calcular  $b_1 = 2 * 3 = 6$
- 3 Calcular  $c_1 = (1 + 2) * (4 + 3) = 21$

- Solució:  $a = 10^2 * 4 + 10 * (21 - 4 - 6) + 6 = 516$

### Subproblema 2: calcular $b = 34 * 21$

#### • Subproblemes:

- 1 Calcular  $a_2 = 3 * 2 = 6$
- 2 Calcular  $b_2 = 4 * 1 = 4$
- 3 Calcular  $c_2 = (3 + 4) * (2 + 1) = 21$

- Solució:  $b = 10^2 * 6 + 10 * (21 - 6 - 4) + 4 = 714$

### Subproblema 3: calcular $c = 46 * 64$

#### • Subproblemes:

- 1 Calcular  $a_3 = 4 * 6 = 24$
- 2 Calcular  $b_3 = 6 * 4 = 24$
- 3 Calcular  $c_3 = (4 + 6) * (6 + 4) = 100$

- Solució:  $c = 10^2 * 24 + 10 * (100 - 24 - 24) + 24 = 2944$

### Resultat

- Problema: calcular  $x * y$  per a  $x = 1234$ ,  $y = 4321$

#### • Subproblemes:

- 1  $a = 12 * 43 = 516$
- 2  $b = 34 * 21 = 714$
- 3  $c = 46 * 64 = 2944$

- Solució:  $10^4 * 516 + 10^2 * (2944 - 516 - 714) + 714 = 5332114$

## Exponenciació ràpida

- L'algorisme iteratiu evident per calcular  $x^n$  fa servir la descomposició

$$x^n = \underbrace{x \cdot x \cdot \dots \cdot x}_{n \text{ factors}}$$

que requereix  $\Theta(n - 1) = \Theta(n)$  multiplicacions.

- Però amb un enfocament recursiu, tenim

$$x^n = x^{n/2} \cdot x^{n/2}$$

quan  $n$  és parell i

$$x^n = x^{(n-1)/2} \cdot x^{(n-1)/2} \cdot x$$

quan  $n$  és senar.

Definició recursiva de  $x^n$

$$x^n = \begin{cases} 1, & \text{si } n = 0 \\ x^{\lfloor n/2 \rfloor} \cdot x^{\lfloor n/2 \rfloor}, & \text{si } n > 0 \text{ i parell} \\ x^{\lfloor n/2 \rfloor} \cdot x^{\lfloor n/2 \rfloor} \cdot x, & \text{si } n \text{ és senar} \end{cases}$$

## Exemple

Amb un algorisme basat en la definició anterior, tindríem

$$\begin{aligned} x^{62} &= (\cancel{x^3})^2, \quad x^{31} = (\cancel{x^15})^2 \cdot x, \quad x^{15} = (\cancel{x^7})^2 \cdot x \\ x^7 &= (\cancel{x^3})^2 \cdot x, \quad x^3 = (\cancel{x^1})^2 \cdot x, \quad x^1 = (\cancel{x^0})^2 \cdot x, \quad x^0 = 1 \end{aligned}$$

(en blau, els valors calculats recursivament)

## Exponenciació ràpida

```
double potencia (double x, int n) {
    if (n == 0) {
        return 1;
    } else {
        double y = potencia (x, n / 2);
        if (n % 2 == 0) return y * y;
        else return y * y * x;
    }
}
```

El càlcul del cost és directe i ve donat per la recurrència

$$T(n) = T(n/2) + \Theta(1)$$

que, segons el teorema mestre de recurrències divisores, implica

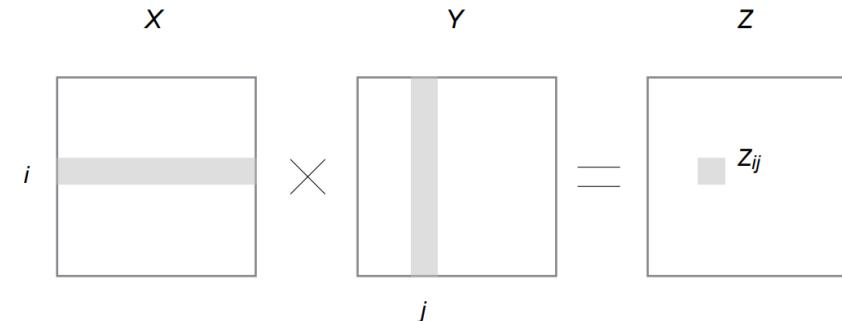
$$T(n) \in \Theta(\log n).$$

## Algorisme de Strassen

El producte de dues matrius  $X$  i  $Y$  de mida  $n \times n$  és una matriu  $Z$  de mida  $n \times n$  tal que

$$Z_{ij} = \sum_{k=1}^n X_{ik} Y_{kj}.$$

$Z_{ij}$  és el producte de la fila  $i$ -èsima de  $X$  per la columna  $j$ -èsima de  $Y$ :



## Producte estàndard de matrius

Algorisme  $\Theta(n^3)$ , adaptat de *Data Structures and Alg. Analysis in C++, M.A. Weiss*.

```
matrix<int> producte_matrius
    (const matrix<int>& X, const matrix<int>& Y)
{
    int n = X.numrows();
    matrix<int> Z(n, n, 0);
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            for (int k = 0; k < n; k++)
                Z[i][j] += X[i][k] * Y[k][j];
    return Z;
}
```

Una primera idea és que el producte de matrius es pot fer *per blocs*.

Dividim  $X$  i  $Y$  en quatre quadrants cadascuna:

$$X = \begin{bmatrix} A & B \\ C & D \end{bmatrix}, \quad Y = \begin{bmatrix} E & F \\ G & H \end{bmatrix}.$$

Llavors, es pot veure que

$$XY = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} E & F \\ G & H \end{bmatrix} = \begin{bmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{bmatrix}.$$

Els subproductes es poden calcular aleshores recursivament.

## Algorisme de Strassen

### Exemple

Per fer el producte de  $X$  i  $Y$

$$X = \begin{bmatrix} 4 & 3 & 1 & 6 \\ 1 & 5 & 2 & 7 \\ 2 & 1 & 5 & 9 \\ 3 & 4 & 2 & 6 \end{bmatrix}, Y = \begin{bmatrix} 2 & 6 & 9 & 4 \\ 3 & 2 & 4 & 1 \\ 1 & 1 & 8 & 3 \\ 2 & 1 & 3 & 1 \end{bmatrix},$$

definim les vuit matrius  $2 \times 2$

$$A = \begin{bmatrix} 4 & 3 \\ 1 & 5 \end{bmatrix}, B = \begin{bmatrix} 1 & 6 \\ 2 & 7 \end{bmatrix}, E = \begin{bmatrix} 2 & 6 \\ 3 & 2 \end{bmatrix}, F = \begin{bmatrix} 9 & 4 \\ 4 & 1 \end{bmatrix},$$

$$C = \begin{bmatrix} 2 & 1 \\ 3 & 4 \end{bmatrix}, D = \begin{bmatrix} 5 & 9 \\ 2 & 6 \end{bmatrix}, G = \begin{bmatrix} 1 & 1 \\ 2 & 1 \end{bmatrix}, H = \begin{bmatrix} 8 & 3 \\ 3 & 1 \end{bmatrix}.$$

i l'expressem en termes de les submatrius

$$XY = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} E & F \\ G & H \end{bmatrix} = \begin{bmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{bmatrix} =$$

$$\begin{bmatrix} \begin{bmatrix} 4 & 3 \\ 1 & 5 \end{bmatrix} \begin{bmatrix} 2 & 6 \\ 3 & 2 \end{bmatrix} + \begin{bmatrix} 1 & 6 \\ 2 & 7 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 2 & 1 \end{bmatrix} & \begin{bmatrix} 4 & 3 \\ 1 & 5 \end{bmatrix} \begin{bmatrix} 9 & 4 \\ 4 & 1 \end{bmatrix} + \begin{bmatrix} 1 & 6 \\ 2 & 7 \end{bmatrix} \begin{bmatrix} 8 & 3 \\ 3 & 1 \end{bmatrix} \\ \begin{bmatrix} 2 & 1 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} 2 & 6 \\ 3 & 2 \end{bmatrix} + \begin{bmatrix} 5 & 9 \\ 2 & 6 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 2 & 1 \end{bmatrix} & \begin{bmatrix} 2 & 1 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} 9 & 4 \\ 4 & 1 \end{bmatrix} + \begin{bmatrix} 5 & 9 \\ 2 & 6 \end{bmatrix} \begin{bmatrix} 8 & 3 \\ 3 & 1 \end{bmatrix} \end{bmatrix}$$

## Algorisme de Strassen

Quin cost té el nou algorisme? Recordem que es basa en el producte

$$XY = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} E & F \\ G & H \end{bmatrix} = \begin{bmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{bmatrix}.$$

El cost  $T(n)$  és la suma de fer:

- vuit productes de matrius de mida  $n/2$ :  $8T(n/2)$
- quatre sumes de matrius de mida  $n/2$ :  $\Theta(n^2)$

Per tant, tenim la recurrència

$$T(n) = 8T(n/2) + \Theta(n^2)$$

que, pel teorema mestre de recurrències divisores, dona

$$T(n) \in \Theta(n^{\log_2 8}) = \Theta(n^3).$$

## Algorisme de Strassen

Però el nombre de productes es pot reduir a 7.

Volker Strassen (1969)

$$XY = \begin{bmatrix} P_5 + P_4 - P_2 + P_6 & P_1 + P_2 \\ P_3 + P_4 & P_1 + P_5 - P_3 - P_7 \end{bmatrix}$$

on

$$P_1 = A(F - H), P_4 = D(G - E)$$

$$P_2 = (A + B)H, P_5 = (A + D)(E + H)$$

$$P_3 = (C + D)E, P_6 = (B - D)(G + H)$$

$$P_7 = (A - C)(E + F)$$

Fent servir la descomposició de Strassen, obtenim un algorisme amb cost

$$T(n) = 7T(n/2) + \Theta(n^2)$$

que, pel teorema mestre de recurrències divisores, dona

$$T(n) \in \Theta(n^{\log_2 7}) \approx \Theta(n^{2.81}).$$

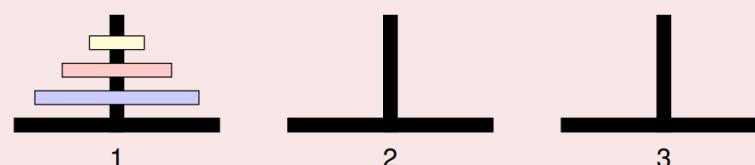
## Torres de Hanoi

### Solució 1

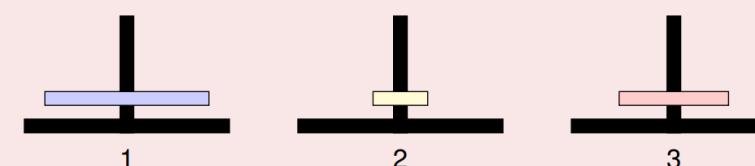
Imaginem les varetes disposades en forma de triangle. Ara, en els moviments:

- **senars**: movem el disc més petit una vareta en sentit horari
- **parells**: fem l'únic moviment possible que no involuci el disc més petit

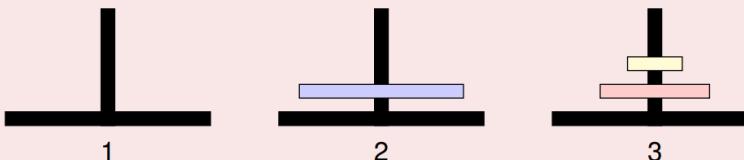
Moviment senar – Moviment parell



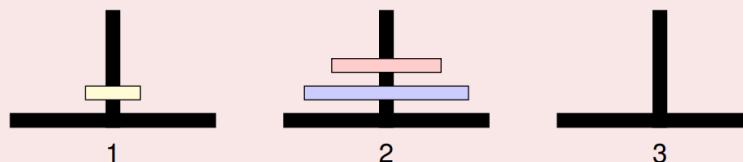
Moviment senar – Moviment parell



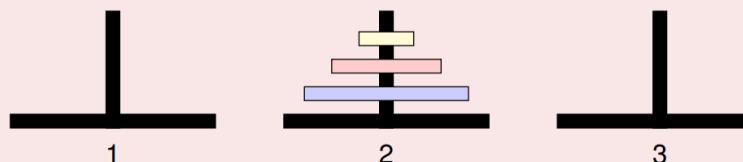
## Moviment senar – Moviment parell



## Moviment senar – Moviment parell



## Moviment senar – Moviment parell



## Torres de Hanoi

### Algorisme

```
void hanoi(int n, int a, int b, int c){
    // descriu els moviments de n discs des d'a fins a b
    // fent servir c com a auxiliar
    if (n > 0) {
        hanoi(n-1, a, c, b);
        cout << a, b;
        hanoi(n-1, c, b, a);
    }
}
```

### Cost

El cost creix com el nombre de moviments. Definim

$$T(n) = \text{nombre de moviments que fa } \text{hanoi}(n, 1, 2, 3).$$

Llavors,

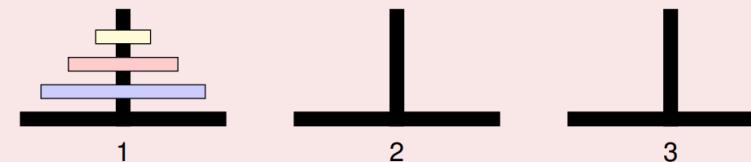
$$T(n) = \begin{cases} 0, & \text{si } n = 0 \\ 2T(n-1) + 1, & \text{si } n > 0 \end{cases}$$

## Torres de Hanoi

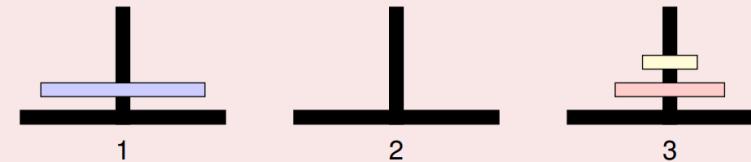
### Solució 2

Per traslladar  $n$  discs de la vareta  $a$  a la  $b$  (on  $\{a, b, c\} = \{1, 2, 3\}$ ):

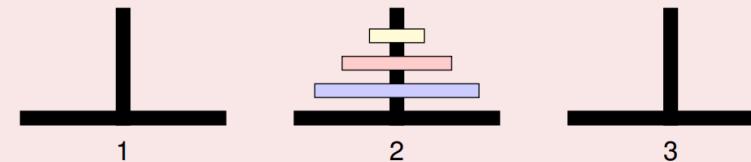
- si  $n = 1$ , moure l'únic disc de  $a$  a  $b$
- si  $n > 1$ ,
  - ➊ traslladar  $n - 1$  discs de  $a$  a  $c$
  - ➋ moure el disc restant (el més gros) de  $a$  a  $b$
  - ➌ traslladar  $n - 1$  discs de  $c$  a  $b$



### Traslladar 2 discs de 1 a 3



### Traslladar 2 discs de 3 a 2



# Torres de Hanoi

## Recurrència

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

## Solució asimptòtica

Aplicant el teorema mestre de recurrències subtractives, obtenim  $T \in \Theta(2^n)$ .

## Solució exacta

Definim  $S(n) = T(n) + 1$  i l'escrivim sense dependre de  $T(n)$ :

- $S(0) = 1$
- $S(n) = 2T(n-1) + 2 = 2(S(n-1) - 1) + 2 = 2S(n-1), \text{ si } n > 0$

Ara,  $S(n)$  es resol directament i dona:  $S(n) = 2^n$  per a tot  $n \geq 0$ .

Per tant,

$$T(n) = 2^n - 1 \text{ per a tot } n \geq 0.$$

## Mediana

### Definició

La **mediana** d'una llista  $S$  de  $n$  nombres és l'element  $\lfloor n/2 \rfloor$ -èsim de  $\text{SORT}(S)$ , on  $\text{SORT}(S)$  és la llista dels elements de  $S$  ordenada creixentment.

### Exemple

La mediana de  $[15, 3, 34, 5, 10]$  és 10 perquè quan s'escriuen en ordre, és el nombre que queda al mig:

$$3 \ 5 \ 10 \ 15 \ 34$$

La mediana de  $[15, 3, 12, 34, 5, 10]$  també és 10 perquè la llista és

$$3 \ 5 \ 10 \ 12 \ 15 \ 34$$

### Característiques de la mediana:

- Sempre és **un dels valors** del conjunt de dades
- És **menys sensible a les observacions atípiques (outliers)**. Per exemple:
  - ① Donats els nombres 1, 1, 1, 1, 1, 1, 1, 1, 1, 100 (10 vegades 1 i una vegada 100),
    - la mitjana és 10
    - la mediana és 1
  - ② Donats 2, 4, 6, 8, 10.000,
    - la mitjana és 2004
    - la mediana és 6

Per **calcular** la mediana, n'hi ha prou a ordenar els elements.

Es pot fer en temps  $\Theta(n \log n)$

$$[8, 0, 3, 10, 5, 7, 12, 3, 7, 2, 9, 1, 6]$$



$$[0, 1, 2, 3, 3, 5, 6, 7, 8, 9, 10, 12]$$

Però es fa més feina de la necessària:

només volem l'element del mig i no caldria ordenar la resta

$$\{0, 3, 5, 3, 2, 1\}, 6, \{8, 10, 7, 12, 7, 9\}$$

## Mediana

Sovint és més fàcil treballar amb una versió més general d'un problema. En aquest cas, triem **el problema de selecció**.

### Definició

Si  $S$  és una llista i  $k$  tal que  $1 \leq k \leq |S|$ , anomenem

$$\text{SEL}(S, k)$$

al  $k$ -èsim element més petit de  $S$ .

### Problema de selecció

Donada una llista  $S$  i un natural  $k$ ,  $1 \leq k \leq |S|$ , determinar  $\text{SEL}(S, k)$ .

La mediana d'una llista  $S$  de  $n$  nombres és  $\text{SEL}(S, \lfloor n/2 \rfloor)$ .

Idea per a un algorisme

Per a cada nombre  $x$ , dividim la llista en 3 conjunts d'elements:

- els més petits que  $x$
- els que són iguals a  $x$
- els més grans que  $x$

Si tenim el vector

$$S = [ 2 \ 36 \ 5 \ 21 \ 8 \ 13 \ 11 \ 20 \ 5 \ 4 \ 1 ]$$

per a  $x = 5$  el dividim en

$$S_E = [ 2 \ 4 \ 1 ] \quad S_x = [ 5 \ 5 ] \quad S_D = [ 36 \ 21 \ 8 \ 13 \ 11 \ 20 ]$$

## Mediana

Idea per a un algorisme

$$S_E = [ 2 \ 4 \ 1 ] \quad S_x = [ 5 \ 5 ] \quad S_D = [ 36 \ 21 \ 8 \ 13 \ 11 \ 20 ]$$

Suposem ara que volem el 8è element de  $S$

$$S = [ 2 \ 36 \ 5 \ 21 \ 8 \ 13 \ 11 \ 20 \ 5 \ 4 \ 1 ]$$

Sabem que serà el 3r element de  $S_D$  perquè  $|S_E| + |S_x| = 5$ .

$$S_E = [ 2 \ 4 \ 1 ] \quad S_x = [ 5 \ 5 ] \quad S_D = [ 36 \ 21 \ 8 \ 13 \ 11 \ 20 ]$$

Podem definir l'operador  $\text{SEL}(S, k)$  de manera recursiva:

$$\text{SEL}(S, k) = \begin{cases} \text{SEL}(S_E, k), & \text{si } k \leq |S_E| \\ x, & \text{si } |S_E| < k \leq |S_E| + |S_x| \\ \text{SEL}(S_D, k - |S_E| - |S_x|), & \text{si } k > |S_E| + |S_x| \end{cases}$$

## Mediana

Esbós de l'algorisme

Algorisme:

- ① Tria un element  $x$  a l'atzar
- ② Dividir  $S$  respecte de  $x$ :  $S_E$ ,  $S_x$ ,  $S_D$
- ③ Calcular  $\text{SEL}(S, k)$  recursivament

A quin ritme convergeix?

- diem que un element triat és **bo** si és  $\geq 25\%$  i  $\leq 75\%$  del vector
- $\Rightarrow$  un element triat a l'atzar té una probabilitat del 50% de ser bo
- $\Rightarrow$  cal triar 2 elements de mitjana abans de trobar-ne un de bo

Això dona

$$T(n) = T(3n/4) + O(n)$$

que implica que  $T(n) \in O(n)$  en mitjana.

## TAD Diccionari

Diccionari

Anomenem **diccionari** (també *taula de símbols*, *associative array* o *map*) a una estructura de dades que conté un conjunt finit d'elements, cadascun dels quals amb un identificador únic anomenat **clau**.

Operacions bàsiques:

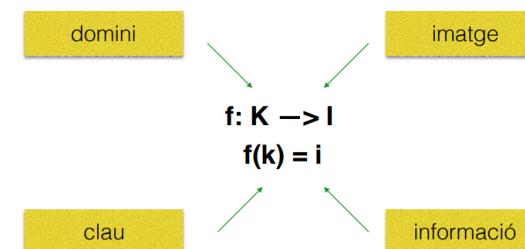
- **assignar**: incloure un element nou
- **esborrar**: eliminar un element
- **consultar**: comprovar si un element hi és

Cada element del diccionari és un parell:

element = (clau, informació)

Es fan servir diccionaris per implementar **taules de símbols** en compiladors i **taules de memòria** en sistemes operatius.

Es pot pensar en els diccionaris com generalitzacions dels **vectors** o com **funcions**:



Suposarem que:

- les funcions són **bijjectives**
- les claus formen un **conjunt totalment ordenat**

Conjunt totalment ordenat

Un conjunt  $K$  està **totalment ordenat** per una operació  $\leq$  si per a tot  $a, b, c \in K$ :

- si  $a \leq b$  i  $b \leq a$ , llavors  $a = b$  (**antisimetria**)
- si  $a \leq b$  i  $b \leq c$ , llavors  $a \leq c$  (**transitivitat**)
- $a \leq b$  o  $b \leq a$  (**totalitat** o **comparabilitat**)

# TAD Diccionari

Com a conjunt de **claus** considerarem el conjunt  $\mathbb{N}$  ordenat per la relació  $\leq$ .

Una subfamília important dels diccionaris és la dels anomenats **diccionaris ordenats** en què és possible fer un recorregut ordenat dels elements per ordre creixent de les seves claus.

En C++ s'aconsegueix mitjançant iteradors.

**Exemple: escriure equivalències en anglès de paraules en català**

L'iterador **it** apunta a un parell **<clau, valor>**, on la clau és una paraula en català i el valor, la traducció a l'anglès.

```
map<string, string> CatEng;  
...  
for (auto it : CatEng)  
    cout << it -> first << ' ' = '' << it -> second << endl;
```

## Operacions

- **assignar**: afegir un element (clau, informació) al diccionari; si existia un element amb la mateixa clau, se sobreescriva la informació
- **esborrar**: donada una clau, s'esborra l'element que té aquella clau; si no hi ha cap element amb la clau, no es fa res
- **consultar**: donada una clau, retorna una referència a la informació associada a la clau
- **talla**: retorna la talla del diccionari

## Variants de **consultar**

Considerarem variants de l'operació

- **consultar**: donada una clau, retorna una **referència a la informació** associada a la clau
- com ara
- **present**: donada una clau, retorna un **booleà** que indica si hi ha un element amb aquella clau
- **cerca**: donada una clau, retorna una **referència a l'element** amb aquella clau

En general, però, afegint operacions més complexes s'obtenen noves estructures de dades. Per exemple, afegint l'operació d'**extreure el mínim** dona lloc a les **cues amb prioritat**.

## Exemple: TAD diccionari

- **VALORS**:  $\mathcal{C} = \mathcal{P}(K \times I)$ , on  $K$  és un conjunt de claus i  $I$  d'informació

- **OPERACIONS**:

|  |   |
|--|---|
| crear: $\rightarrow \mathcal{C}$                         | assignar: $K \times I \times \mathcal{C} \rightarrow \mathcal{C}$ |
| esborrar: $K \times \mathcal{C} \rightarrow \mathcal{C}$ | consultar: $K \times \mathcal{C} \rightarrow I \cup \{\perp\}$    |

- **PROPIETATS**: suposem  $k, k_1, k_2 \in K, i, j \in I, k_1 \neq k_2$

|  |   |
|--|---|
| esborrar ( $k$ , crear) = crear  | assignar ( $k, i$ , assignar ( $k, j$ , $D$ )) = assignar ( $k, i$ , esborrar ( $k_1$ , $D$ ))          |
| esborrar ( $k$ , assignar ( $k, i$ , $D$ )) = esborrar ( $k, D$ )                              | assignar ( $k_2, i$ , assignar ( $k_1, j$ , $D$ )) = assignar ( $k_2, i$ , assignar ( $k_1, i$ , $D$ )) |
| esborrar ( $k_1$ , assignar ( $k_2, i$ , $D$ )) = assignar ( $k_2, i$ , esborrar ( $k_1, D$ )) | assignar ( $k_1, i$ , assignar ( $k_2, j$ , $D$ )) = assignar ( $k_2, j$ , assignar ( $k_1, i$ , $D$ )) |
| consultar ( $k$ , crear) = $\perp$   | consultar ( $k$ , assignar ( $k, i$ , $D$ )) = $\perp$  |
| consultar ( $k$ , esborrar ( $k, D$ )) = $\perp$   | consultar ( $k$ , assignar ( $k, i$ , $D$ )) = $i$  |
| consultar ( $k_1$ , assignar ( $k_2, i$ , $D$ )) = consultar ( $k_1, D$ )                      | consultar ( $k_1, assignar (k_2, i, D)$ ) = consultar ( $k_1, D$ )                                      |

## Nombre d'elements consultats en les operacions de diccionaris

| cas pitjor/mitjà   | assignar                          | esborrar                          | consultar                        |
|--------------------|-----------------------------------|-----------------------------------|----------------------------------|
| vector no ordenat  | $\Theta(n), \Theta(n)$            | $\Theta(n), \Theta(n)$            | $\Theta(n), \Theta(n)$           |
| vector ordenat     | $\Theta(n), \Theta(n)$            | $\Theta(n), \Theta(n)$            | $\Theta(\log n), \Theta(\log n)$ |
| llista no ordenada | $\Theta(n), \Theta(n)$            | $\Theta(n), \Theta(n)$            | $\Theta(n), \Theta(n)$           |
| llista ordenada    | $\Theta(n), \Theta(n)$            | $\Theta(n), \Theta(n)$            | $\Theta(n), \Theta(n)$           |
| taula de dispersió | $\Theta(n), \Theta(1)$            | $\Theta(n), \Theta(1)$            | $\Theta(n), \Theta(1)$           |
| AVL                | $\Theta(\log n), \Theta(\log n),$ | $\Theta(\log n), \Theta(\log n),$ | $\Theta(\log n), \Theta(\log n)$ |

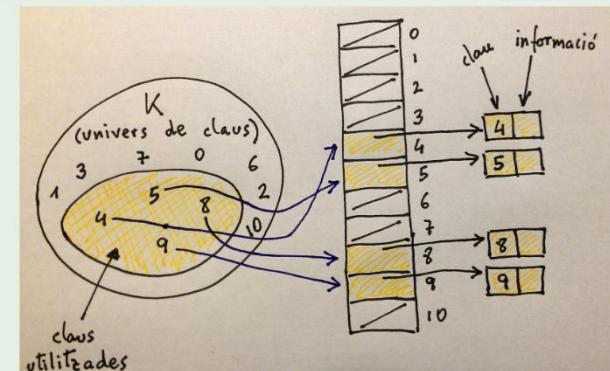
En vectors, cal desplaçar els elements.

En estructures no ordenades, cal comprovar repetits.

## Taules d'accés directe

Suposem que un club d'esports no tindrà mai més de 300 socis. Per implementar un diccionari amb el número de soci com a clau, n'hi ha prou a definir un vector  $V[0..299]$ .

## Exemple d'adreçament directe (simplificat)



# Taules d'accés directe

En l'**adreçament directe**:

- Cada posició correspon a una clau
- Les operacions seran  $\Theta(1)$  en cas pitjor
- Es pot guardar la informació directament en les posicions de la taula corresponents a la seva clau, però cal indicar si l'espai és buit

## Exercici

Volem implementar un diccionari fent servir adreçament directe en un vector *enorme*. Cal tenir en compte que

- inicialment, les entrades poden contenir deixalles
- no és pràctic inicialitzar el vector a causa de la seva talla

Descriuïu un mètode per aconseguir implementar les operacions **consultar**, **assignar** i **esborrar** en temps  $\Theta(1)$ .

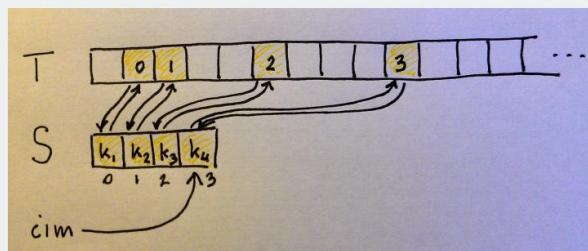
## Solució: idea

Definim un vector enorme  $T$  accessible per clau i un vector  $S$  amb tantes entrades com claus utilitzades ( $cim + 1$ ) tal que per a una clau  $k$ :

- $T[k]$  conté l'índex  $j$  d'una entrada vàlida de  $S$
- $S[j]$  conté  $k$

És a dir,  $S[T[k]] = k$  i  $T[S[j]] = j$  (en diem **cicle de validació**).

Definim també un vector  $S'$  que conté els objectes (la informació). Quan la clau  $k$  defineix un cicle de validació,  $S'[T[k]]$  conté l'objecte.



## Solució: operacions

### • inicialitzar

```
cim = -1;
```

### • consultar. Donada una clau $k$ ,

```
if (S[T[k]] == k)
    return S'[T[k]];
else
    return NULL;
```

## Solució: operacions

- **assignar.** Donat un objecte  $x$  amb clau  $k$ , si la clau no hi és,

```
++cim;
S[cim] = k;
S'[cim] = x;
T[k] = cim;
```

- **esborrar.** Donada una clau  $k$  (suposant que la clau hi és), hem d'assegurar que no queda un "forat" a  $S$ .

```
S[T[k]] = S[cim];
S'[T[k]] = S'[cim];
T[S[T[k]]] = T[k];
T[k] = 0;
--cim;
```

## Taules de dispersió

Les **taules de dispersió (hash tables)** són estructures de dades eficients per implementar els diccionaris.

- Són generalitzacions dels vectors
- El temps en cas pitjor pot ser de  $\Theta(n)$ , però amb hipòtesis raonables el temps esperat serà  $\Theta(1)$  en totes les operacions

Quan

- les claus no són nombres naturals o
- el nombre de claus utilitzades és petit en relació al nombre possible de claus o
- el nombre possible de claus és enorme,

cal abandonar les taules d'accés directe. Llavors,

En lloc de fer servir la clau com a índex per accedir a la taula, **l'índex es calcula a partir de la clau**.

Si es vol consultar un llibre a la biblioteca central de la Universitat de Karlsruhe, cal demanar el llibre amb antelació. Llavors, el personal el busca i el classifica en una habitació amb 100 prestatges d'acord amb la regla següent:

El llibre estarà col·locat en un prestatge numerat amb els dos últims dígitos del carnet de la biblioteca de qui l'ha demanat.

## Taules de dispersió

La biblioteca expedeix carnets des del 1825 amb números consecutius.

### Qüestió

Per què els dos últims díigits i no els dos primers?

### Pista

La biblioteca podria rebre la visita d'un grup d'amics amb números de carnet semblants, com ara

001523  
001525  
001531  
001570

Però no és tan probable que en algun moment molts usuaris tinguin els dos últims díigits idèntics.

Suposem que volem emmagatzemar un màxim de  $n$  claus. Declarem una taula  $T$  de  $m \leq n$  posicions.

- Si  $m = n$  i les claus són  $\{0, 1, \dots, m - 1\}$ , usem **adreçament directe**: l'element de clau  $k$  va a l'espai  $T[k]$ .
- Si  $m < n$ , usem **taules de dispersió**: l'element de clau  $k$  va a l'espai  $T[h(k)]$ , on

$$h : K \rightarrow \{0, 1, \dots, m - 1\}$$

és la **funció de dispersió** (*hash function*) i  $h(k)$ , el **valor de dispersió** de  $k$ .

La situació en què dues claus tenen el mateix valor de dispersió i, per tant, coincideixen en el mateix espai (com  $k_2$  i  $k_5$ ) se'n diu **col·lisió**.

## Col·lisions

Les col·lisions són **molt probables**.

Suposem que les nostres claus són els dies de l'any. Quanta gent ha d'haver-hi en una habitació perquè la probabilitat que almenys dues tinguin l'aniversari el mateix dia sigui del

- ① 100%? 367
- ② 50%? 23
- ③ 99.9%? 70

## Col·lisions

La tria de la funció de dispersió és fonamental per evitar col·lisions.

Algunes consideracions sobre la funció de dispersió:

- Ha de **ser determinista** però ha de **semblar aleatòria**
- Com que  $|K| > m$ , **les col·lisions no es poden evitar**
- Cal triar un mètode per resoldre les col·lisions

El mètode més simple per resoldre col·lisions és l'**encadenament** (o **encadenament separat**).

## Col·lisions

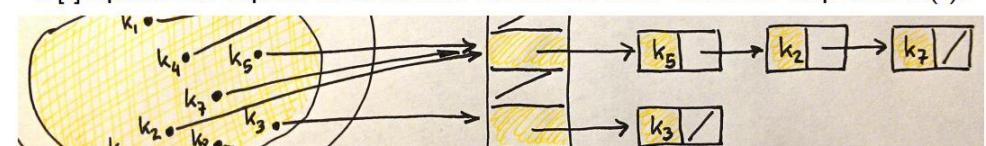
Resolució de col·lisions per **encadenament**. Cada entrada  $T[j]$  conté una **llista encadenada** amb les claus amb valor de dispersió  $j$ .

Per exemple,  $h(k_1) = h(k_4)$  i, per tant,  $T[h(k_1)]$  apunta a  $k_1$  seguit de  $k_4$ .

## Encadenament

En l'**encadenament**, els elements que tenen el mateix valor de dispersió es posen en una **llista encadenada**:

$T[i]$  apunta al cap de la llista dels elements amb valor de dispersió  $h(i)$ .



El cost de fer **una consulta** és

- $\Theta(n)$  en el **cas pitjor**. És el cas en què tots els elements tenen el mateix valor de dispersió i formen una sola llista.
- $\Theta(1)$  en el **cas mitjà** assumint que:
  - a cada crida amb una nova clau,  $h$  retorna un natural entre 0 i  $m - 1$  a l'atzar, independentment de les crides anteriors i
  - el cost de calcular  $h$  és  $\Theta(1)$ .

# Encadenament

## Classe Dictionary: implementació (*Algorithms in C++, EDA*)

```
template <typename Key, typename Info>
class Dictionary {

private:

typedef pair<Key, Info> Pair;
typedef list<Pair> List;
typedef typename List::iterator iter;

vector<List> t; // Taula de dispersió
int n;           // Nombre de claus
int M;           // Nombre de posicions
```

## Classe Dictionary: implementació

```
public:

Dictionary (int M = 1009)
: t(M), n(0), M(M) { }

void assign (const Key& key, const Info& info) {
    int h = hash(key) % M;
    iter p = find(key, t[h]);
    if (p != t[h].end())
        p->second = info;
    else {
        t[h].push_back(Pair(key, info));
        ++n;
    }
}
```

## Classe Dictionary: implementació

```
void erase (const Key& key) {
    int h = hash(key) % M;
    iter p = find(key, t[h]);
    if (p != t[h].end())
        t[h].erase(p);
    --n;
}

}
```

```
Info& query (const Key& key) {
    int h = hash(key) % M;
    iter p = find(key, t[h]);
    if (p != t[h].end())
        return p->second;
    else
        throw ''Key does not exist'';
```

## Classe Dictionary: implementació

```
bool contains (const Key& key) {
    int h = hash(key) % M;
    iter p = find(key, t[h]);
    return p != t[h].end();
}

int size () {
    return n;
}
```

El cas pitjor de les operacions assign, erase, query i contains és  $\Theta(n)$ .  
El cost mitjà és  $\Theta(1 + n/M)$ .

## Classe Dictionary: implementació

Els costos previs depenen del cost de fer una cerca, que és  $\Theta(n)$  en cas pitjor i  $\Theta(1 + n/M)$  en mitjana.

private:

```
static iter find (const Key& key, list<Pair>& L) {
    iter p = L.begin();
    while (p != L.end() and p->first != key)
        ++p;
    return p;
}
```

## Encadenament: anàlisi del cas pitjor

### Solució

Si  $K$  és el conjunt de totes les claus i  $m$  és el nombre de posicions de la taula de dispersió, quantes claus podem assegurar que col·lidiran a un mateix valor de dispersió si

- $|K| > m?$  2
- $|K| > 2m?$  3
- $|K| > 3m?$  4
- $|K| > nm?$   $n+1$

### Proposició

El cost en cas pitjor de fer una cerca (find) en l'esquema d'encadenament és  $\Theta(n)$ .

### Corol·laris

El cost en cas pitjor de fer una assignació, un esborrat o una consulta (amb cerca prèvia) en l'esquema d'encadenament és  $\Theta(n)$ .

## Encadenament: anàlisi del cas mitjà

### Definició

Si  $T$  és una taula amb  $m$  posicions que conté  $n$  elements, el **factor de càrrega** per a  $T$  és  $\alpha = n/m$  (nombre mitjà d'elements per posició).

### Proposició

El cost en cas mitjà de fer una **cerca** (`find`) en l'esquema d'encadenament és  $\Theta(1 + \alpha)$ .

### Demostració

Suposem que cerquem una clau  $k$  a  $T$ .

El cost esperat és  $\Theta(1 + E[X])$ , on  $X$  és la mida de  $T[h(k)]$ .

Definim la variable booleana  $X_e = [h(e) = h(k)]$ . Llavors,

$$E[X] = E\left[\sum_e X_e\right] = \sum_e E[X_e] = \sum_e P[X_e = 1] = n \cdot \frac{1}{m}.$$

Per tant,  $\Theta(1 + E[X]) = \Theta(1 + n/m) = \Theta(1 + \alpha)$ .

### Proposició

El cost en cas mitjà de fer una **cerca** (`find`) en l'esquema d'encadenament és  $\Theta(1 + \alpha)$ .

### Corol·lari

El cost en cas mitjà de fer una **assignació**, un **esborrat** o una **consulta** (amb cerca prèvia) en l'esquema d'encadenament és  $\Theta(1 + \alpha)$ .

## Encadenament

### Exemple

Volem emmagatzemar la informació de matrícula dels alumnes:

- Si un nom té  $\leq 20$  caràcters, l'espai de possibles claus és de  $\approx 2^{20}$
- El nombre màxim d'alumnes nous és de  $n = 300$

Definim un vector de  $m = 300$  posicions, de manera que, en mitjana, cada llista de sinònims tindrà talla  $\approx 1$  i les operacions, cost  $\Theta(1)$ .

Però com triem la funció de dispersió?

## Funcions de dispersió

Les claus seran sempre nombres naturals. Si no ho són, s'interpreten com a naturals.

### Exemple

Donada una cadena de caràcters, la interpretem com un natural.

Donada la cadena CLRS:

- valors en ASCII: C= 67, L= 76, R= 82, S= 83
- hi ha 128 caràcters en ASCII
- per tant, CLRS es transforma en el natural

$$\begin{aligned} 67 \cdot 128^3 + 76 \cdot 128^2 + 82 \cdot 128^1 + 83 \cdot 128^0 \\ = 141.764.947 \end{aligned}$$

### El mètode de la divisió

Dispersem una clau  $k$  en una de les  $m$  posicions a través de la funció

$$h(k) = k \bmod m$$

- **Exemple:** si la taula té mida  $m = 12$  i  $k = 100$ , llavors  $h(k) = 4$
- **Avantatge:** és força ràpid
- **Desavantatge:** cal evitar certs valors de  $m$  com  $2^i$
- **Bones tries per a  $m$ :** primers no gaire propers a potències de 2

### Exemple 1

Volem una taula de dispersió amb les col·lisions resoltes per encadenament, que emmagatzemi unes  $n = 2000$  cadenes de caràcters. No ens fa res examinar uns 3 elements en una cerca fallida.

Per tant, triem una taula de mida

$$m = 701.$$

Triem 701 perquè és un primer  $\approx 2000/3$  i no és proper a una potència de 2.

La funció de dispersió serà

$$h(k) = k \bmod 701.$$

# Funcions de dispersió

## Exemple 2: Factor de càrrega més petit

Volem una taula de dispersió amb encadenament per emmagatzemar  $n = 2000$  cadenes de caràcters. Ens està bé examinar uns 2 elements en una cerca fallida.

Per tant, triem una taula de mida

$$m = 1009.$$

Triem 1009 perquè és un primer  $\approx 2000/2$ .

La funció de dispersió serà

$$h(k) = k \bmod 1009.$$

## Exemple 3: Mantenir el factor de càrrega petit

Ara volem emmagatzemar 18000 cadenes de caràcters addicionals. Ens està bé examinar uns 5 elements en una cerca fallida.

Per tant, triem una taula de mida

$$m = 4001.$$

Triem 4001 perquè és un primer  $\approx 20000/5$ .

Fem una redispersió de les cadenes antigues i inserim les noves amb la funció de dispersió

$$h(k) = k \bmod 4001.$$

## El mètode de la multiplicació

Per dispersar una clau  $k$  en una de les  $m$  posicions:

- ① multipliquem  $k$  per una constant  $A$  t.q.  $0 < A < 1$  i n'extraiem la part fraccional
- ② llavors, multipliquem el valor per  $m$  i prenem la part baixa

$$h(k) = \lfloor m(kA \bmod 1) \rfloor = \lfloor m(kA - \lfloor kA \rfloor) \rfloor$$

- **Avantatge:** el valor de  $m$  no és crític
- **Desavantatge:** és més lent que el mètode de divisió
- **Bones tries per a  $m$ :** potències de 2 (fan la implementació fàcil)
- **Bona tria per a  $A$ :** Knuth suggereix  $A \approx (\sqrt{5} - 1)/2 = 0,6180339887\dots$

## Exemple

En una taula de dispersió amb  $m = 1024$ ,  $k = 123$  i  $A = 0,61803399$ , tenim

$$\begin{aligned} h(k) &= \lfloor 1024 \cdot (123 \cdot A - \lfloor 123 \cdot A \rfloor) \rfloor \\ &= \lfloor 1024 \cdot (76,0181808 - 76) \rfloor \\ &= \lfloor 1024 \cdot 0,0181808 \rfloor = 18. \end{aligned}$$

## Adreçament obert

Una alternativa a l'encadenament és l'**adreçament obert**:

- tots els elements s'emmagatzemen en la mateixa taula
- quan cerquem un element, examinem les posicions de manera sistemàtica fins a trobar-lo
- la funció de dispersió té dos paràmetres: la clau i la "prova" de posició

$$h : K \times \{0, 1, \dots, m-1\} \rightarrow \{0, 1, \dots, m-1\}.$$

Per a una clau  $k$ , la seqüència de proves és

$$(h(k, 0), h(k, 1), \dots, h(k, m-1)).$$

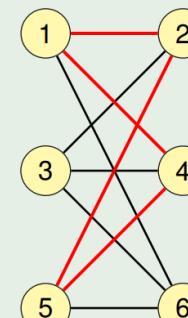
## Definicions i terminologia

### Definició

Un **graf** és un parell  $(V, E)$  on:

- $V$  és un conjunt finit (**vèrtexs**)
- $E$  és un conjunt de parells no ordenats de vèrtexs (**arestes**)

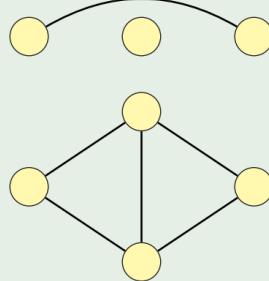
### Exemple: graf connex i cíclic



- **Connex:** hi ha un camí entre dos vèrtexs qualssevol
- **Cíclic:** hi ha cicles com  $1 \rightarrow 2 \rightarrow 5 \rightarrow 4 \rightarrow 1$

## Definicions i terminologia

Exemple: graf inconnex i cíclic

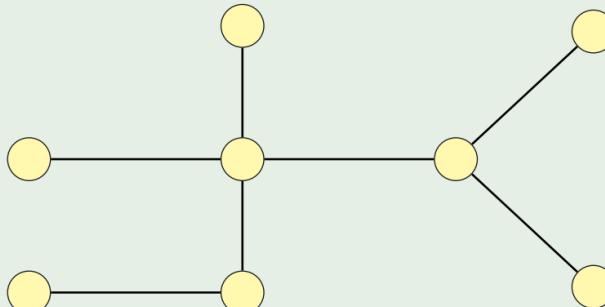


## Definicions i terminologia

### Definició

Un **arbre** és un graf connex i acíclic.

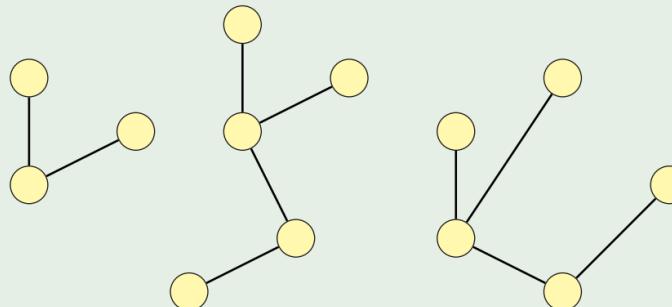
### Exemple: arbre



### Definició

Un **bosc** és un graf acíclic.

### Exemple: bosc



## Teorema

Sigui  $G = (V, E)$  un graf i siguin  $n = |V|$  i  $m = |E|$ .

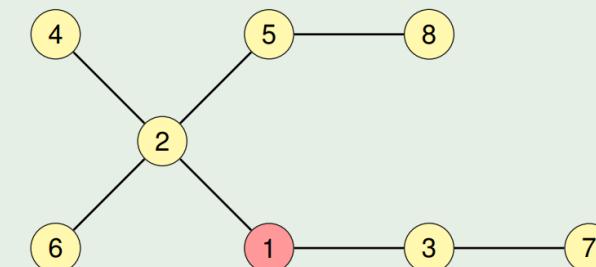
Les afirmacions següents són equivalents:

- ①  $G$  és un **arbre**
- ② Tot parell de vèrtexs de  $G$  estan units per un **camí únic**
- ③  $G$  és connex i  $m = n - 1$
- ④  $G$  és connex però, si s'hi elimina una aresta, s'obté un graf inconnex
- ⑤  $G$  és acíclic i  $m = n - 1$
- ⑥  $G$  és acíclic però, si s'hi afegeix una aresta, s'obté un graf cíclic

### Definició

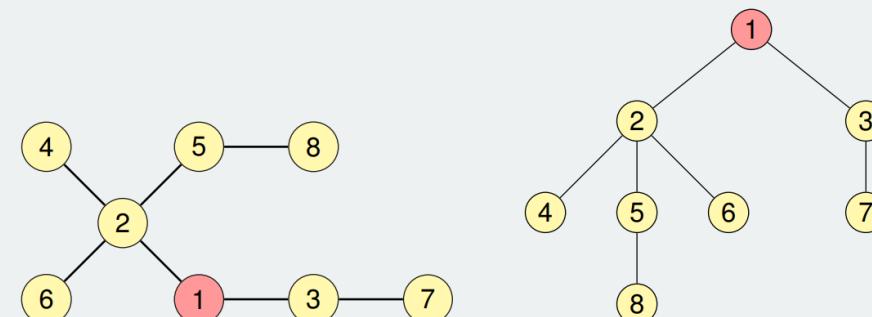
Un **arbre arrelat** és un arbre amb un vèrtex distingit (**l'arrel**).

### Exemple: arbre arrelat



### Representació

Representarem els arbres arrelats amb **l'arrel a dalt**.



### Terminologia

En aquest tema, per **arbre** entendrem **arbre arrelat**.

# Definicions i terminologia

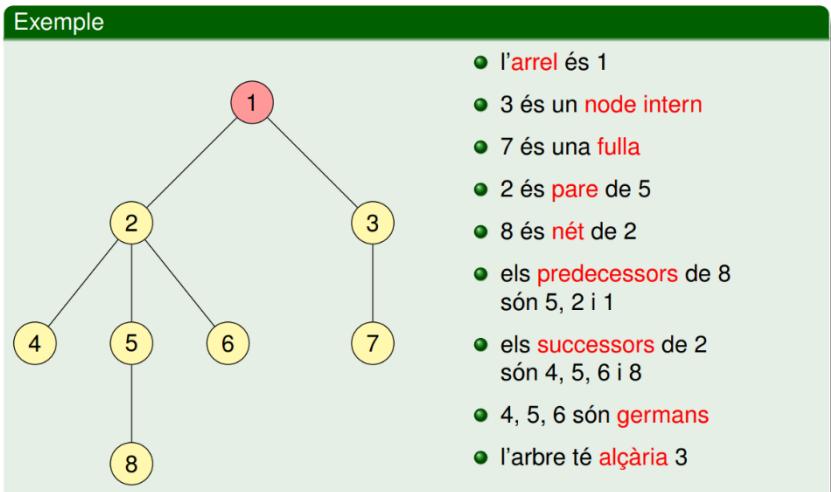
## Terminologia pròpia dels arbres

Els vèrtexs s'anomenen **nodes**. El node distingit s'anomena **arrel**.

Sigui  $x$  un node d'un arbre  $T$  amb arrel  $r$ :

- tot node  $y$  en el camí de  $r$  a  $x$  és un **predecessor** de  $x$
- si  $y$  és un predecessor de  $x$ , llavors  $x$  és un **descendent** de  $y$
- si  $y$  és un predecessor de  $x$  i existeix l'aresta  $\{y, x\}$  a  $T$ , llavors  $y$  és el **pare** de  $x$  i  $x$  és el **fill** de  $y$
- el pare del pare de  $x$  s'anomena **avi** de  $x$ ; si  $y$  és l'avi de  $x$ , llavors  $x$  és el **nét** de  $y$
- dos nodes amb el mateix pare s'anomenen **germans**
- un node sense fills es diu que és una **fulla**
- si un node no és una fulla, es diu que és un **node intern**
- l'**alçària** de  $T$  és la màxima distància (nombre d'arestes d'un camí) entre l'arrel i una fulla

## Exemple



## Introducció

La propietat dels ABC permet implementar altres operacions com

- 1 fer la **llista ordenada** dels elements en temps  $\Theta(n)$
- 2 trobar el **mínim** o el **màxim** en temps mitjà  $\Theta(\log n)$
- 3 trobar l'**anterior** o el **següent** d'un element donat en temps mitjà  $\Theta(\log n)$

## Definició

Un **arbre binari** és un arbre arrelat on cada node té un màxim de dos fills, que es diferencien com a **fill esquerre** i **fill dret**.

També es poden definir els arbres binaris de manera recursiva.

## Definició

Un **arbre binari** és una estructura formada sobre un conjunt finit de nodes que:

- no conté cap node o
- està formada per tres conjunts de nodes:
  - l'**arrel**,
  - un arbre binari anomenat **subarbre esquerre** i
  - un arbre binari anomenat **subarbre dret**.

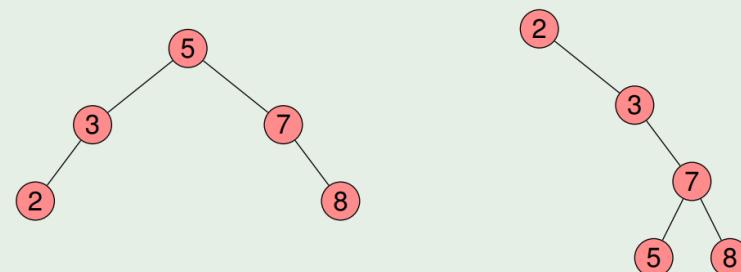
## Introducció

### Definició

Un **arbre binari de cerca** (ABC, *Binary Search Tree* o BST en anglès) és un arbre binari que té una clau associada a cada node i que compleix la propietat que la clau de cada node és

- més gran que la de tots els nodes del seu subarbre esquerre i
- més petita que la de tots els nodes del seu subarbre dret

## Exemple



Operacions dels diccionaris en els ABC:

- Les **operacions bàsiques dels diccionaris** (**consultar**, **assignar**, **esborrar**) es poden fer en temps proporcional a l'alçària
- L'**alçària esperada** d'un ABC de  $n$  nodes és de  $\Theta(\log n)$
- Per tant, les operacions bàsiques tenen un **cost mitjà** de  $\Theta(\log n)$
- L'**alçària màxima** d'un ABC de  $n$  nodes és de  $\Theta(n)$
- Per tant, les operacions bàsiques tenen un **cost**  $\Theta(n)$  en cas pitjor

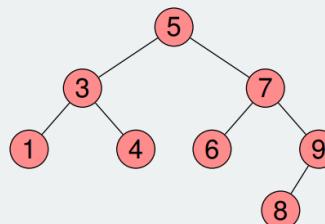
# Introducció

## Llista ordenada dels elements d'un ABC

- 1 Per fer la llista ordenada dels elements d'un ABC, n'hi ha prou a fer un recorregut inordre de l'arbre:

```
RECORREGUT-INORDRE(x)
  if x ≠ NUL llavors
    RECORREGUT-INORDRE(esquerre(x))
    escriure clau(x)
    RECORREGUT-INORDRE(dret(x))
```

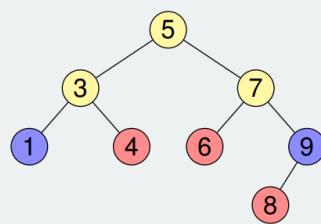
El recorregut inordre de l'ABC següent és: 1, 3, 4, 5, 6, 7, 8, 9.



Com que cada node es visita un cop, el cost és  $\Theta(n)$ .

## Trobar el mínim i el màxim d'un ABC

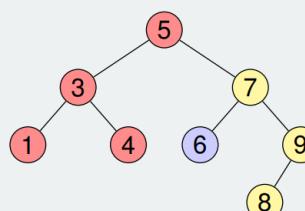
- 2 Mínim o màxim



El cost mitjà correspon a l'alçària esperada de l'arbre:  $\Theta(\log n)$ .

## Trobar l'anterior o el següent d'un element en un ABC

- 3 Següent de 5: el mínim del seu subarbre dret



Llavors, el cost mitjà és  $\Theta(\log n)$ .

# Implementació (ABC)

Implementació: *Algorismes en C++ (EDA)*, J. Petit, S. Roura, A. Atserias.

## Representació:

- Un **node** és:



- Un **diccionari** implementat amb un ABC té dos camps:

- el nombre de nodes de l'ABC
- un apuntador a l'arrel de l'ABC

## Els costos que es donen:

- es refereixen al cas pitjor
- depenen de  $n$ , que és el camp del mateix nom del diccionari (**nombre d'elements**)

## Definició de Diccionari i Node

Un **Node** emmagatzema una clau, la seva informació associada i apuntadors a dos altres nodes.

```
template <typename Clau, typename Info>
class Diccionari {
private:
    struct Node {
        Clau clau;
        Info info;
        Node* fesq; // Apuntador al fill esquerre
        Node* fdre; // Apuntador al fill dret

        Node (const Clau& c, const Info& i, Node* fe, Node* fd)
            : clau(c), info(i), fesq(fe), fdre(fd) { }
    };
    int n;           // Nombre d'elements de l'ABC
    Node* arrel;   // Apuntador a l'arrel de l'ABC
```

## Implementació (ABC): funcions públiques

### Constructores de creació i còpia / Destructora

Crear Diccionari:  $\Theta(1)$ .

```
Diccionari () {
    n = 0;
    arrel = nullptr;
}
```

Fer una còpia:  $\Theta(n)$ .

```
Diccionari (const Diccionari& d) {
    n = d.n;
    arrel = copia(d.arrel);
}
```

Destructora:  $\Theta(n)$ .

```
~Diccionari () {
    alliberar(arrel);
}
```

### Constructora d'assignació

Redefinició de l'assignació:  $\Theta(n + d.n)$ .

```
Diccionari& operator= (const Diccionari& d) {
    if (&d != this) {
        alliberar(arrel);
        n = d.n;
        arrel = copia(d.arrel);
    }
    return *this;
}
```

### Assignar i esborrar

Assignar a clau el valor info:  $\Theta(n)$ .

```
void assignar (const Clau& clau, const Info& info) {
    assignar(arrel, clau, info);
}
```

Esborrar clau i la informació associada (si no hi és, no canvia):  $\Theta(n)$ .

```
void esborrar (const Clau& clau) {
    esborrar_3(arrel, clau);
}
```

### Consultes

Donada una clau, retornar la referència a la informació associada:  $\Theta(n)$ .

```
Info& consultar (const Clau& clau) {
    if (Node* p = cerca(arrel, clau)) {
        return p->info;
    } else {
        throw "La clau no era present";
    }
}
```

Indicar si la clau hi és o no present:  $\Theta(n)$ .

```
bool present (const Clau& clau) {
    return cerca(arrel, clau) != null;
}
```

Retornar la talla del diccionari:  $\Theta(1)$ .

```
int talla () {
    return n;
}
```

## Implementació (ABC): funcions privades

Suposarem que l'arbre a tractar (apuntat per p) té

- $s$  nodes
- alçària  $h$

Els costos en cas pitjor vindran donats per  $\Theta(s)$  o  $\Theta(h)$ .

### Esborrar

Eliminar l'arbre apuntat per p:  $\Theta(s)$ .

```
static void alliberar (Node* p) {
    if (p) {
        alliberar(p->esq);
        alliberar(p->fdre);
        delete p;
    }
}
```

### Copiar

Retornar un apuntador a una còpia de l'arbre apuntat per p:  $\Theta(s)$ .

```
static Node* copia (Node* p) {
    return p ? new Node(p->clau, p->info,
                        copia(p->esq), copia(p->fdre))
              : nullptr;
}
```

## Implementació (ABC): funcions privades

### Cerca

Retornar un apuntador al node de l'arbre apuntat per  $p$  que conté  $clau$  (o null si no hi és):  $\Theta(h)$ .

```
static Node* cerca (Node* p, const Clau& clau) {
    if (p) {
        if (clau < p->clau) {
            return cerca(p->fesq, clau);
        } else if (clau > p->clau) {
            return cerca(p->fdre, clau);
        }
    }
    return p;
}
```

La cerca es fa descendint pels subarbres adequats fent ús de la propietat dels ABC.

### Assignar

Assignar  $info$  a  $clau$  si la clau és al subarbre apuntat per  $p$ ; si no hi és, afegir un nou node amb  $clau$  i  $info$ :  $\Theta(h)$ .

```
void assignar
    (Node*& p, const Clau& clau, const Info& info) {
    if (p) {
        if (clau < p->clau) {
            assignar(p->fesq, clau, info);
        } else if (clau > p->clau) {
            assignar(p->fdre, clau, info);
        } else {
            p->info = info;
        }
    } else {
        p = new Node(clau, info, nullptr, nullptr);
        ++n;
    }
}
```

### Mínim (recursiu)

Retornar un apuntador al node que conté el valor mínim en el subarbre apuntat per  $p$  (suposant que  $p$  no és nul):  $\Theta(h)$ .

```
static Node* minim (Node* p) {
    return p->fesq ? minim(p->fesq) : p;
}
```

### Màxim (iteratiu)

Retornar un apuntador al node que conté el valor màxim en el subarbre apuntat per  $p$  (suposant que  $p$  no és nul):  $\Theta(h)$ .

```
static Node* maxim (Node* p) {
    while (p->fdre) p = p->fdre;
    return p;
}
```

### Esborrar (1)

Esborrar el node que conté  $clau$  en el subarbre apuntat per  $p$ :  $\Theta(h)$ . Es penja el subarbre esquerre del mínim del subarbre dret.

```
void esborrar_1 (Node*& p, const Clau& clau) {
    if (p) {
        if (clau < p->clau) {
            esborrar_1(p->fesq, clau);
        } else if (clau > p->clau) {
            esborrar_1(p->fdre, clau);
        } else {
            Node* q = p;
            if (!p->fesq) p = p->fdre;
            else if (!p->fdre) p = p->fesq;
            else {
                Node* m = minim(p->fdre);
                m->fesq = p->fesq;
                p = p->fdre;
            }
            delete q;
            --n;
        }
    }
}
```

### Esborrar (2)

Inconvenient: es copien claus i informació, més costós que copiar apuntadors.

```
void esborrar_2 (Node*& p, const Clau& clau) {
    if (p) {
        if (clau < p->clau) {
            esborrar_2(p->fesq, clau);
        } else if (clau > p->clau) {
            esborrar_2(p->fdre, clau);
        } else if (!p->fesq) {
            Node* q = p;
            p = p->fdre;
            delete q;
            --n;
        } else if (!p->fdre) {
            Node* q = p;
            p = p->fesq;
            delete q;
            --n;
        } else {
            Node* m = minim(p->fdre);
            p->clau = m->clau;
            p->info = m->info;
            esborrar_2(p->fdre, m->clau);
        }
    }
}
```

## Implementació (ABC): funcions privades

### Esborrar (3)

Esborrar el node que conté `clau` en el subarbre apuntat per `p`:  $\Theta(h)$ .  
Es copia el mínim del subarbre dret al node esborrat i després s'esborra.

```
void esborrar_3 (Node*& p, const Clau& clau) {
    if (p) {
        if (clau < p->clau) {
            esborrar_3(p->fesq, clau);
        } else if (clau > p->clau) {
            esborrar_3(p->fdre, clau);
        } else {
            Node* q = p;
            if (!p->fesq) p = p->fdre;
            else if (!p->fdre) p = p->fesq;
            else {Node* m = esborrar_minim(p->fdre);
                   m->fesq = p->fesq; m->fdre = p->fdre;
                   p = m;}
            delete q; --n; } } }
```

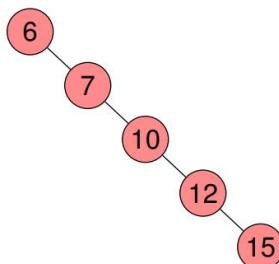
### Esborrar mínim

Esborrar i retornar el node que conté l'element mínim de `p`:  $\Theta(h)$ .

```
Node* esborrar_minim (Node*& p) {
    if (p->fesq) {
        return esborrar_minim(p->fesq);
    } else {
        Node* q = p;
        p = p->fdre;
        return q;
    } }
```

## Introducció

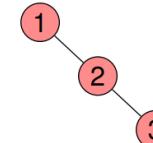
En els **arbres binaris de cerca**, hem vist que el cost de les operacions és  $\Theta(h)$ , on  $h$  és l'alçària. Però  $h$  pot coincidir amb el nombre de nodes:



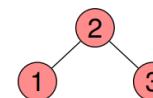
Per tant, el cost en cas pitjor és de  $\Theta(n)$ , on  $n$  és el nombre de nodes.

Per millorar el cost lineal es poden fer dues coses:

- demostrar que si en un ABC s'insereixen els elements de forma aleatòria, l'alçària és  $\approx \log n$
- fer les insercions i els esborrars de manera que l'alçària es mantingui en  $\approx \log n$ . En comptes de tenir



tindríem



Com mantenir els subarbres equilibrats?

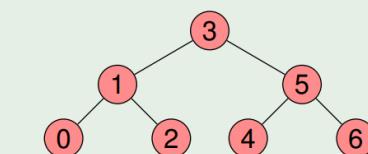
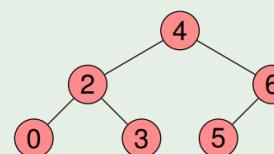
- ➊ Forçant que l'ABC sigui **complet**.

Un arbre complet és aquell que té tots els nivells plens tret, potser, de l'últim, on els nodes són el més a l'esquerra possible.

Però afegir un element pot ser massa costós.

### Exemple

Per afegir l'element 1 en el primer arbre, cal canviar-ho gairebé tot.



Com mantenir els subarbres equilibrats?

- ➋ Permetent un **petit desequilibri** en les alçàries dels subarbres esquerre i dret: una diferència màxima d'1.

Però cal fer-ho per a tots els nodes!

### Definició

Un arbre binari està **equilibrat** si

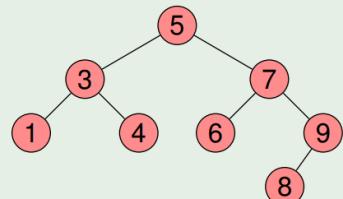
- és buit o
- la diferència d'alçàries dels seus subarbres és com a màxim d'1 i els seus subarbres també estan equilibrats

## Introduction

Els ABC amb la condició d'equilibri s'anomenen **arbres AVL**.

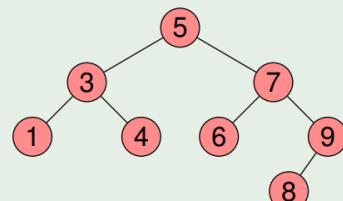
### Exemple

L'arbre de l'esquerra està equilibrat, però si n'esborrem l'arrel com en els ABC, deixarà d'estar equilibrat.

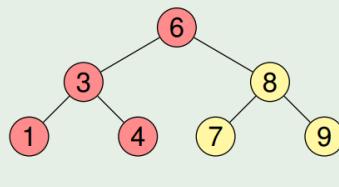


Subarbre desequilibrat en groc

L'arbre de l'esquerra està equilibrat, però si n'esborrem l'arrel com en els ABC, deixarà d'estar equilibrat. **Els AVL usen “rotacions” per solucionar-ho.**



Rotació aplicada al subarbre groc



## Implementació (AVL)

### Definició de Diccionari i Node

Com la dels ABC però afegint-hi l'alçària.

```
template <typename Clau, typename Info>
class Diccionari {
private:
    struct Node {
        Clau clau;
        Info info;
        Node* fesq; // Apuntador al fill esquerre
        Node* fdre; // Apuntador al fill dret
        int alc; // Alçària de l'arbre
        Node (const Clau& c, const Info& i,
              Node* fe, Node* fd, int a)
            : clau(c), info(i), fesq(fe), fdre(fd), alc(a) {} ;
    };
    int n; // Nombre d'elements en l'AVL
    Node* arrel; // Apuntador a l'arrel de l'AVL
```

Les funcions públiques i les privades **alliberar**, **copia** i **cerca** són iguals que en els ABC.

En els AVL necessitarem dues funcions senzilles referents a l'alçària, totes dues de cost  $\Theta(1)$ .

### Retornar i actualitzar l'alçària

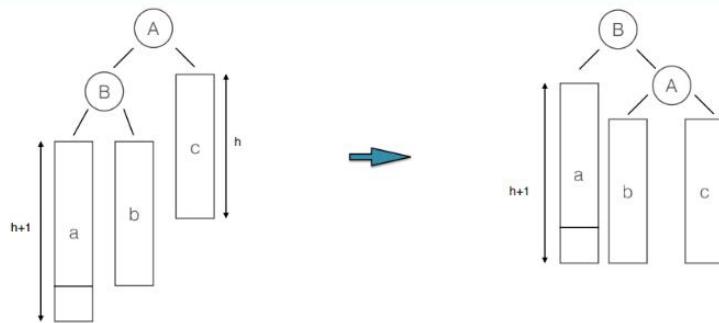
```
static int alcaria (Node* p) {
    return p ? p->alc : -1;
}

static void actualitzar_alcaria (Node* p) {
    p->alc = 1 + max(alcaria(p->fesq), alcaria(p->fdre));
}
```

Per tal de mantenir equilibrat un arbre després d'una assignació o esborrat, considerem quatre **rotacions** de l'arbre: EE, DD, ED i DE.

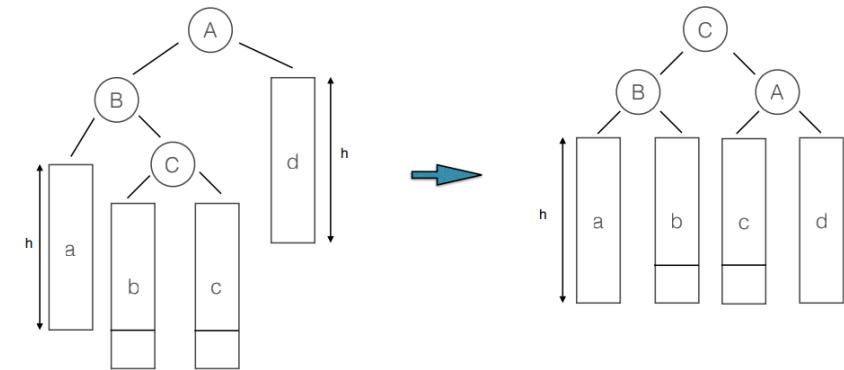
Totes quatre tenen cost  $\Theta(1)$ .

## Implementació (AVL)



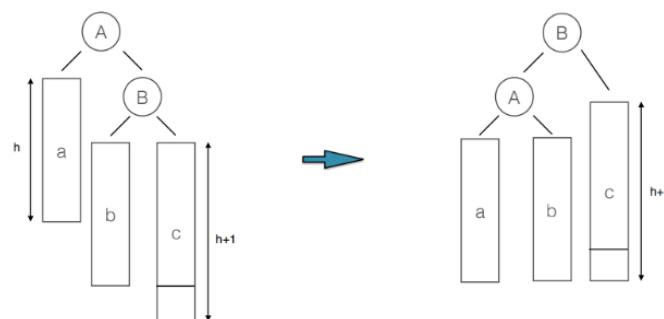
Rotació EE (LL, en anglès)

```
static void LL (Node*& p) {
    Node* q = p;
    p = p->fesq;
    q->fesq = p->fdre;
    p->fdre = q;
    actualitzar_alcaria(q);
    actualitzar_alcaria(p); }
```



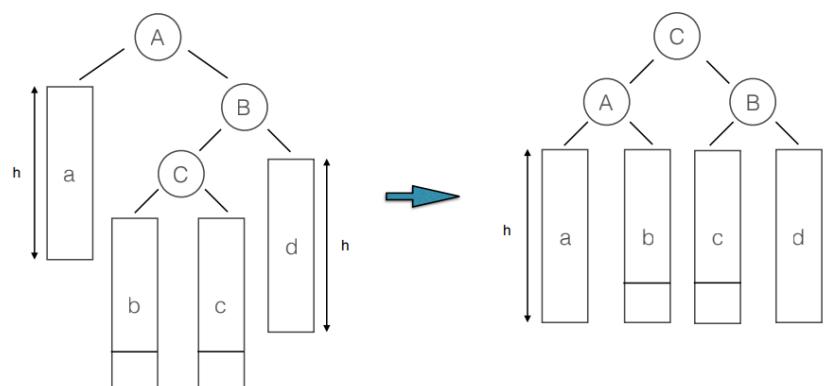
Rotació ED (LR, en anglès)

```
static void LR (Node*& p) {
    RR(p->fesq);
    LL(p);
}
```



Rotació DD (RR, en anglès)

```
static void RR (Node*& p) {
    Node* q = p;
    p = p->fdre;
    q->fdre = p->fesq;
    p->fesq = q;
    actualitzar_alcaria(q);
    actualitzar_alcaria(p); }
```



Rotació DE (RL, en anglès)

```
static void RL (Node*& p) {
    LL(p->fdre);
    RR(p);
}
```

# Implementació (AVL)

## Assignar

```

void assignar (Node*& p, const Clau& clau, const Info& info) {
    if (p) {
        if (clau < p->clau) {
            assignar(p->fesq, clau, info);
            if (alcaria(p->fesq) - alcaria(p->fdre) == 2) {
                if (clau < p->fesq->clau) LL(p);
                else LR(p);
            }
            actualitzar_alcaria(p);
        } else if (clau > p->clau) {
            assignar(p->fdre, clau, info);
            if (alcaria(p->fdre) - alcaria(p->fesq) == 2) {
                if (clau > p->fdre->clau) RR(p);
                else RL(p);
            } else p->info = info;
        } else {
            p = new Node(clau, info, nullptr, nullptr, 0);
            ++n;
        }
    }
}

```

En **mitjana**, es fa una rotació cada dues assignacions.

En el **cas pitjor**:

- **assignar i esborrar** fan  $\Theta(\log n)$  rotacions, on una rotació costa  $\Theta(1)$   
     $\Rightarrow \Theta(\log n)$
- el cost de **consultar**, com en els ABC, és  $\Theta(\log h)$ , on  $h$  és l'alçària de l'arbre  $\Rightarrow \Theta(\log n)$

## Proposició

En els AVL, les operacions **assignar**, **esborrar** i **consultar** tenen cost en cas pitjor  $\Theta(\log n)$ .

## Exercici 1

Un node en un arbre binari és un **fill únic** si té pare però no germans. El **factor de soledat** d'un arbre binari  $T$  de  $n$  nodes,  $\mathcal{FS}(T)$ , es defineix com:

$$\mathcal{FS}(T) = \frac{\text{nombre de fills únics a } T}{n}.$$

- ① Demostreu que si  $T$  és un AVL buit, llavors  $\mathcal{FS}(T) \leq 1/2$
- ② És cert que si  $T$  és un arbre binari i  $\mathcal{FS}(T) \leq 1/2$ , llavors l'alçària de  $T$  és  $O(\log n)$ ?

## Solució

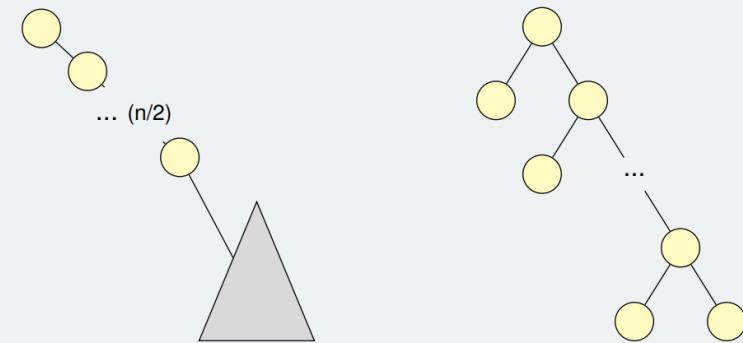
- ① Demostreu que si  $T$  és un AVL buit, llavors  $\mathcal{FS}(T) \leq 1/2$

En un AVL els fills únics només poden aparèixer a les fulles (altrament, es produiria un desequilibri de 2 o més). Com que cada fill únic té un pare diferent (que a més no és fill únic), el nombre total de fills únics ha de ser com a màxim  $\lfloor n/2 \rfloor$ . Però llavors  $\mathcal{FS}(T) \leq \frac{\lfloor n/2 \rfloor}{n} \leq \frac{n/2}{n} = \frac{1}{2}$ .

## Solució

- ② És cert que si  $T$  és un arbre binari i  $\mathcal{FS}(T) \leq 1/2$ , llavors l'alçària de  $T$  és  $O(\log n)$ ?

No, si  $\mathcal{FS}(T) \leq 1/2$  llavors hi pot haver en  $T$  fins a  $\lfloor n/2 \rfloor$  fills únics. Això permet una alçària d'almenys  $n/2$ :



## Alçària d'un AVL

Veurem el fet següent sobre AVLs.

## Teorema

L'alçària d'un AVL de  $n$  nodes és  $O(\log n)$ .

Per demostrar-lo, definim

$$n_k = \text{mínim nombre de nodes d'un AVL d'alçària } k$$

## Exercici

- ① Calculeu  $n_0$ ,  $n_1$ ,  $n_2$  i  $n_3$
- ② A partir d'aquests nombres, obtingueu una definició inductiva per a  $n_k$

## Solució

Obtenim  $n_0 = 1$ ,  $n_1 = 2$ ,  $n_2 = 4$  i  $n_3 = 7$ .

En general, podem definir  $n_k$  inductivament com:

- $n_0 = 1$
- $n_1 = 2$
- $n_k = \underbrace{n_{k-1}}_{\text{cal un subarbre}} + \underbrace{n_{k-2}}_{\text{l'alçària } k-2 \text{ dona}} + 1 \quad \begin{matrix} & & \\ & & \end{matrix}$   
d'alçària  $k-1$  mín. # de nodes

## Alçària d'un AVL

### Teorema

L'alçària d'un AVL de  $n$  nodes és  $O(\log n)$ .

### Demostració

- ① Com que  $n_m \geq n_{m-1}$  per a tot  $m > 0$ , tenim

$$n_k = n_{k-1} + n_{k-2} + 1 \geq 2n_{k-2} + 1 \geq 2n_{k-2}$$

- ② Per substitució repetida, tenim

$$n_k \geq 2n_{k-2} \geq 4n_{k-4} \geq \dots \geq \underbrace{2^{\frac{k+1}{2}}}_{k \text{ senar}} \geq \underbrace{2^{\frac{k}{2}}}_{k \text{ parell}}$$

- ③ Donat un AVL  $T$  d'alçària  $k$  i  $n$  nodes:

$$n \geq n_k \geq 2^{\frac{k}{2}}$$

- ④ Prenent logaritmes,  $k \leq 2 \log_2 n \in O(\log n)$

La definició dels nombres de Fibonacci és:

- $F_0 = 1$
- $F_1 = 1$
- $F_k = F_{k-1} + F_{k-2}$  for  $k > 1$

