

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

Mida de l'entrada i cost

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

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

Notació

A partir d'ara, escriurem **log** en lloc de **log₂**.

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, $f = O(g)$.

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 $n_0 = 4$.

Notació asimptòtica: definicions

Notació Θ ((a): fita exacta asimptòtica)

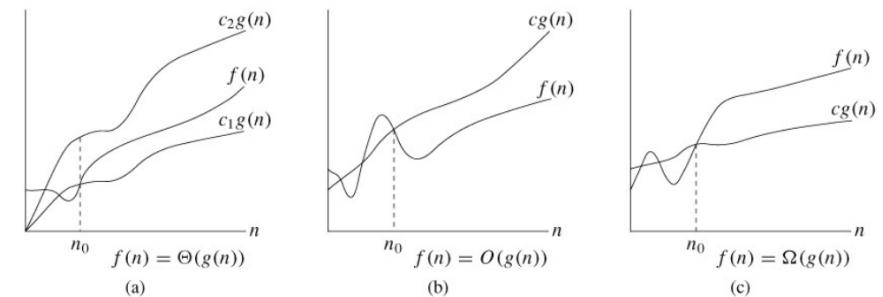
$$\Theta(g) = \{f : \mathbb{N} \rightarrow \mathbb{R}^+ \mid \exists c_1, c_2 \in \mathbb{R}^+ \exists n_0 \in \mathbb{N} \forall n \geq n_0 \quad c_1g(n) \leq f(n) \leq c_2g(n)\}$$

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 Ω ((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 , Ω i Θ

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 $0 < c < \infty \Rightarrow O(f) = O(g)$

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

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.

Propietats de Θ

Mateixes que O gran menys caracterització per :

- **Simetria.** $f \in \Theta(g) \iff g \in \Theta(f) \iff \Theta(f) = \Theta(g)$

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*)
- **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'algorisme 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

Càlcul 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
- Avaluar 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 $\Theta(n)$.

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

Càlcul 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
6 void ordena_seleccio (vector<int>& v, int n) {
7   for (int i = n-1; i > 0; --i) {
8     int k = posicio_maxim(v, i);
9     swap(v[k], v[i]);
10 }
```

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: $\Theta(1)$.

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

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 insertat.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|-----|
| 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

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

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

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

$$\begin{aligned} T(n) &= T(n-1) + \Theta(1) \text{ per a } n \geq 1, \text{ i} \\ T(0) &= \Theta(1). \end{aligned}$$

Solució

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

Recurrència per a la cerca binària

$$\begin{aligned} T(n) &= T(n/2) + \Theta(1) \text{ per a } n \geq 1, \text{ i} \\ T(0) &= \Theta(1). \end{aligned}$$

Solució

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

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:

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

Teoremes mestres

Per sistematitzar l'anàlisi del cost dels algoritmes 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 anteriors:

- **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 subtractives

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

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 $f(k) = f(k - 1) + f(k - 2)$ per a $k \geq 2$, 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 $\phi = 1.618033988\dots$

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:

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

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:

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

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:

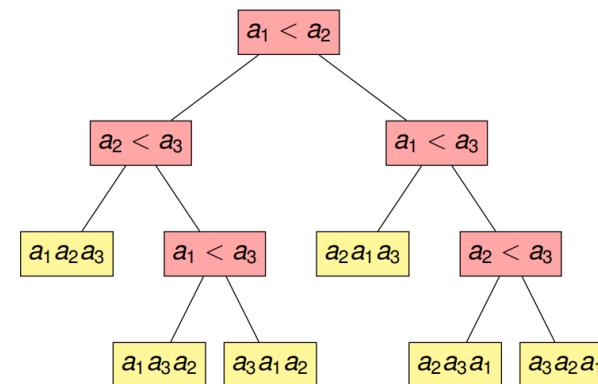
$$T(n) = T(2^m) = S(m) \in \Theta(\log m) = \Theta(\log \log n).$$

Fita inferior dels algorismes d'ordenació

Proposició

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

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.**



- 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

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

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.

Donat un vector T de talla ≥ 2 , l'algorisme consisteix a:

- Partir T en dues meitats
- Ordenar recursivament la meitats de T per separat
- 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

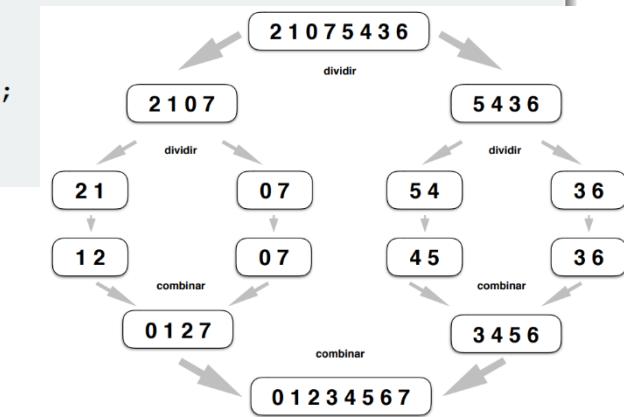
Ordenació per fusió (Algorismes en C++, EDA)

```

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

```
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 |
|---|---|---|---|---|---|---|

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 $\Theta(1)$)

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 recurrències 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.

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)
```

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

| | | | | | |
|---|---|---|---|---|----|
| 2 | 6 | 5 | 1 | 4 | 03 |
|---|---|---|---|---|----|

| | | | | |
|---|---|---|----|----|
| 5 | 1 | 4 | 03 | 26 |
|---|---|---|----|----|

| | | | |
|---|----|----|----|
| 4 | 03 | 26 | 15 |
|---|----|----|----|

| | | |
|----|----|-----|
| 26 | 15 | 034 |
|----|----|-----|

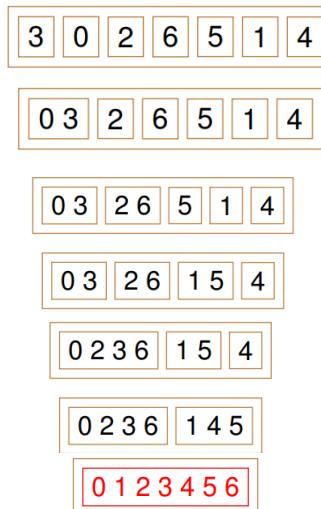
| | |
|-----|------|
| 034 | 1256 |
|-----|------|

| |
|---------|
| 0123456 |
|---------|

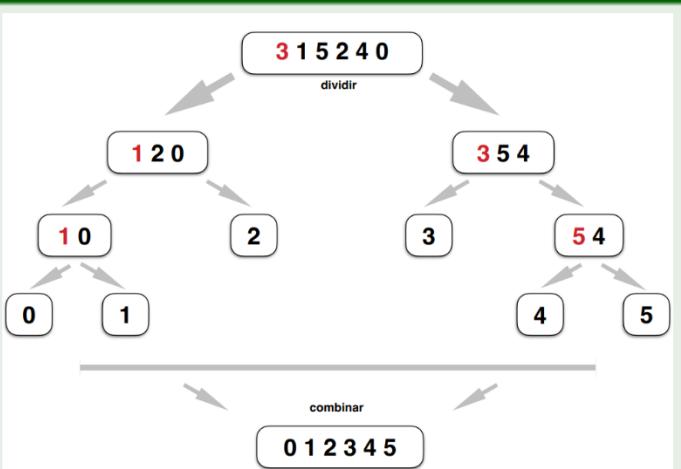
| | |
|-----|------|
| 034 | 1256 |
|-----|------|

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); } }
```

$q = \text{partition}(T, e, d)$

- Precondició: $0 \leq e \leq d \leq T.size() - 1$

- Postcondició: $\exists x \text{ (pivot)} \forall i$

- si $e \leq i \leq q$, tenim que $T[i] \leq x$
- si $q < i \leq d$, tenim que $T[i] \geq x$



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 $\text{partition}(T, e, d)$:

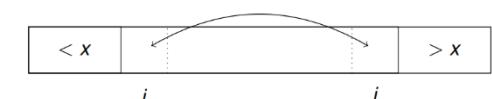


- Bucle principal:

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



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



Variants

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.

Cas pitjor

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

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

Algorisme de Karatsuba

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

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

Exemple

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

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

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

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

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

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

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

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 recurrències 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

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

on $a = x_E y_E$, $b = x_D y_D$, $c = (x_E + x_D)(y_E + y_D)$.

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

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, tindriem

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

(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

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;  
}
```

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

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

Dividim X i Y en quatre quadrants cadascuna:

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

Llavors, es pot veure que

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

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 2×2

$$\begin{aligned} A &= \begin{bmatrix} 4 & 3 \\ 1 & 5 \end{bmatrix}, B = \begin{bmatrix} 1 & 6 \\ 2 & 7 \end{bmatrix}, E = \begin{bmatrix} 2 & 6 \\ 3 & 2 \end{bmatrix}, F = \begin{bmatrix} 9 & 4 \\ 4 & 1 \end{bmatrix}, \\ C &= \begin{bmatrix} 2 & 1 \\ 3 & 4 \end{bmatrix}, D = \begin{bmatrix} 5 & 9 \\ 2 & 6 \end{bmatrix}, G = \begin{bmatrix} 1 & 1 \\ 2 & 1 \end{bmatrix}, H = \begin{bmatrix} 8 & 3 \\ 3 & 1 \end{bmatrix}. \end{aligned}$$

i l'expressem en termes de les submatrius

$$\begin{aligned} XY &= \begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} E & F \\ G & H \end{bmatrix} = \begin{bmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{bmatrix} = \\ &= \left[\begin{bmatrix} 4 & 3 \\ 1 & 5 \end{bmatrix} \begin{bmatrix} 2 & 6 \\ 3 & 2 \end{bmatrix} + \begin{bmatrix} 1 & 6 \\ 2 & 7 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 2 & 1 \end{bmatrix} \quad \begin{bmatrix} 4 & 3 \\ 1 & 5 \end{bmatrix} \begin{bmatrix} 9 & 4 \\ 4 & 1 \end{bmatrix} + \begin{bmatrix} 1 & 6 \\ 2 & 7 \end{bmatrix} \begin{bmatrix} 8 & 3 \\ 3 & 1 \end{bmatrix} \right] \\ &\quad \left[\begin{bmatrix} 2 & 1 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} 2 & 6 \\ 3 & 2 \end{bmatrix} + \begin{bmatrix} 5 & 9 \\ 2 & 6 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 2 & 1 \end{bmatrix} \quad \begin{bmatrix} 2 & 1 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} 9 & 4 \\ 4 & 1 \end{bmatrix} + \begin{bmatrix} 5 & 9 \\ 2 & 6 \end{bmatrix} \begin{bmatrix} 8 & 3 \\ 3 & 1 \end{bmatrix} \right] \end{aligned}$$

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

$$T(n) = 7T(n/2) + \Theta(n^2) \quad T(n) \in \Theta(n^{\log_2 7}) \approx \Theta(n^{2.81}).$$

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

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

Llavors,

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

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, 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 $\Theta(n \log n)$

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

\Downarrow

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

Però es fa més feina de la necessària:

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

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

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:

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

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)

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.

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 sobreescrivia 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**.

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

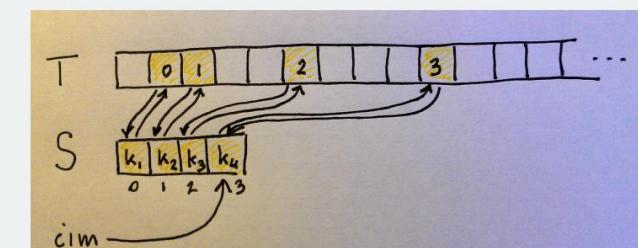
Solució: idea

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;
```

- **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.

Quan cal abandonar les taules d'accés directe.

- 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,

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

Qüestió

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

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

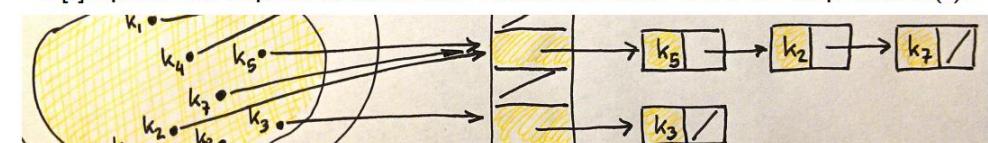
Resolució de col·lisions per **encadenament**. Cada entrada $T[j]$ conté una **llista encadenada** amb les claus amb valor de dispersió j .

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, independent de les crides anteriors i
 - el cost de **calcular h** és $\Theta(1)$.

```

class Dictionary {
    private:

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

    vector<List> t; // Taula de dispersio
    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;
        }
    }

    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";
    }

    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 $\Theta(1 + n/M)$.

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·lari

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

Exemple

Volem emmagatzemar la informació de matrícula dels alumnes:

- Si un nom té ≤ 20 caràcters, l'espai de possibles claus és de $\approx 27^{20}$
- 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 ≈ 1 i les operacions, cost $\Theta(1)$.

Però com triem la funció de dispersió?

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:

```

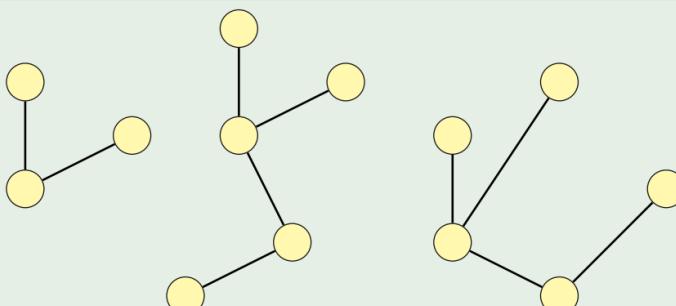
static iter find (const Key& key, list<Pair>& L) {
    iter p = L.begin();
    while (p != L.end() and p->first != key)
        ++p;
    return p;
}

```


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:

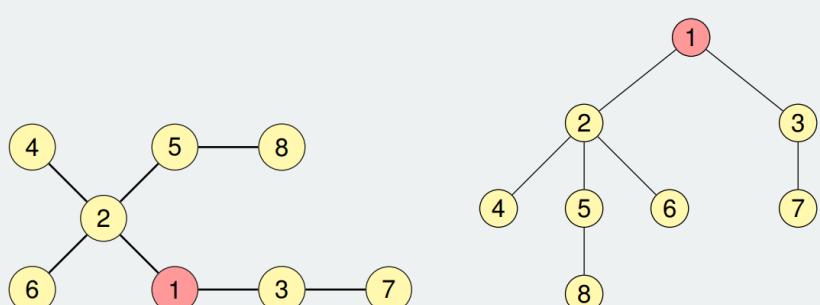
- ① G és un **arbre**
- ② Tot parell de vèrtexos de G estan units per un **camí únic**
- ③ G és connex i $m = n - 1$
- ④ G és connex però, si s'hi elimina una aresta, s'obté un graf inconnex
- ⑤ G és acíclic i $m = n - 1$
- ⑥ G és acíclic però, si s'hi afegeix una aresta, s'obté un graf cíclic

Definició

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

Representació

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

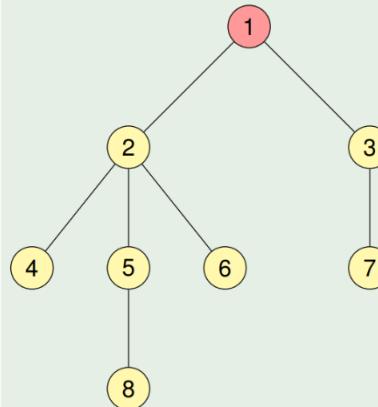


Terminologia

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

Definicions i terminologia

Exemple



- l'**arrel** és 1
- 3 és un **node intern**
- 7 és una **fulla**
- 2 és **pare** de 5
- 8 és **nét** de 2
- els **predecessors** de 8 són 5, 2 i 1
- els **successors** de 2 són 4, 5, 6 i 8
- 4, 5, 6 són **germans**
- l'**arbre** té **alçària** 3

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

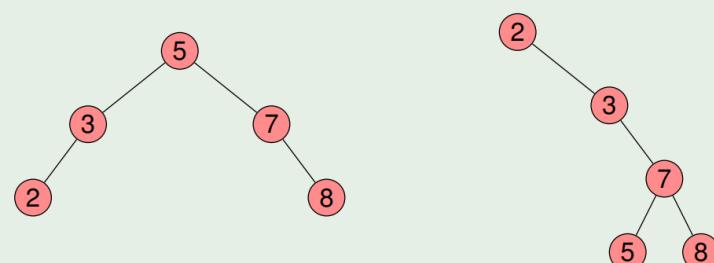
Un **arbre binari** és un arbre arrelat on cada node té un màxim de dos fills, que es diferencien com a **fill esquerre** i **fill dret**.

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:

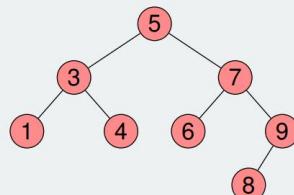
- Les **operacions bàsiques dels diccionaris** (**consultar, assignar, esborrar**) es poden fer en temps proporcional a l'alçària
- L'**alçària esperada** d'un ABC de n nodes és de $\Theta(\log n)$
- Per tant, les operacions bàsiques tenen un **cost mitjà** de $\Theta(\log n)$
- L'**alçària màxima**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

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:

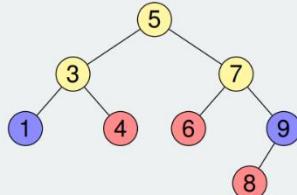
```
RECORREGUT-INORDRE(x)
  if x ≠ NUL llavors
    RECORREGUT-INORDRE(esquerre(x))
    escriure clau(x)
    RECORREGUT-INORDRE(dret(x))
```

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 $\Theta(n)$.

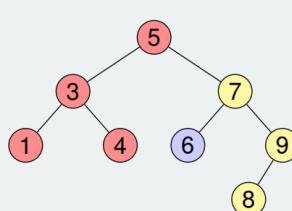
- 2 Mínim o màxim



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)

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
```

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);
}
```

Destructor: $\Theta(n)$.

```
~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->fesq), copia(p->fdre) )
                         : nullptr;
}
```

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);
        ++
    }
}
```

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);
        }
    }
}
```

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;
        }
    }
}
```

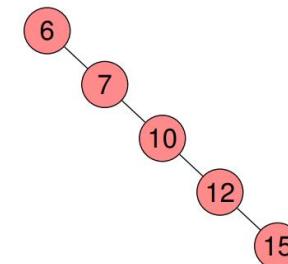
Esborrar mínim

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

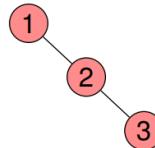
En els **arbres binaris de cerca**, hem vist que el cost de les operacions és $\Theta(h)$, on h és l'alçària. Però h pot coincidir amb el nombre de nodes:



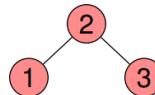
Per tant, el cost en cas pitjor és de $\Theta(n)$, on n és el nombre de nodes.

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 esborrats de manera que l'alçària es mantingui en $\approx \log n$. En comptes de tenir



tindriem



Com mantenir els subarbres equilibrats?

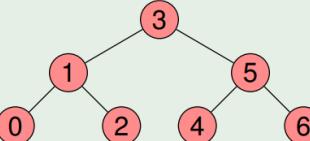
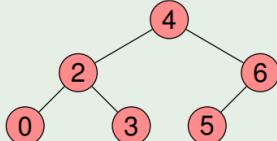
- 1 Forçant que l'ABC sigui **complet**.

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?

- 3 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

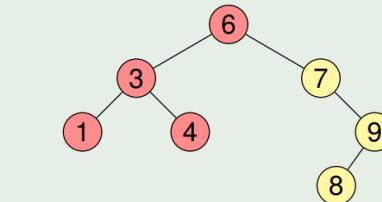
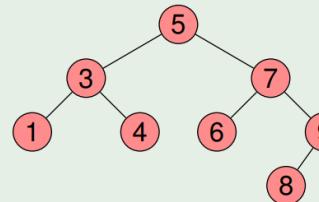
- és buit o
- la diferència d'alçàries dels seus subarbres és com a màxim d'1 i els seus subarbres també estan equilibrats

Introduction

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

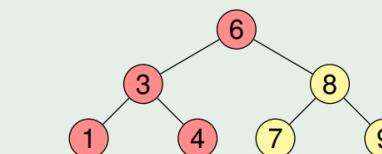
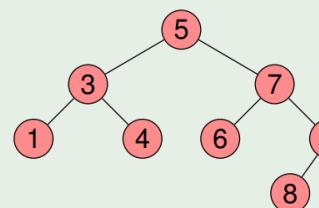
Exemple

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)

```
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 $\Theta(1)$.

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);  
            }  
            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 $\Theta(1)$
 $\implies \Theta(\log n)$
- el cost de **consultar**, com en els ABC, és $\Theta(\log h)$, on h és l'alçària de l'arbre $\implies \Theta(\log n)$

Proposició

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

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

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

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:

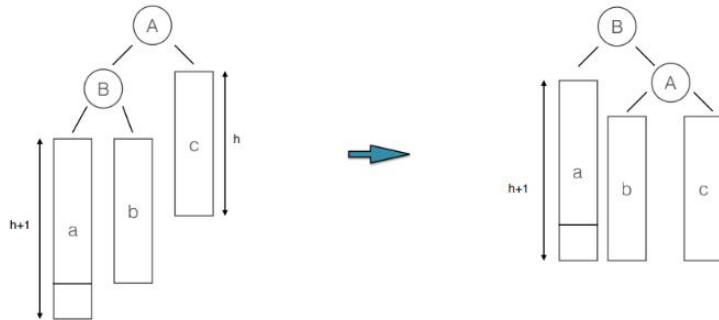
$$n \geq n_k \geq 2^{\frac{k}{2}}$$

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

La definició dels nombres de Fibonacci és:

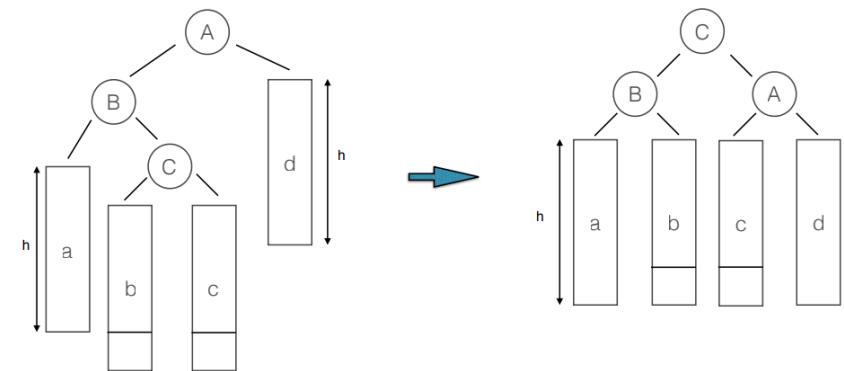
- $F_0 = 1$
- $F_1 = 1$
- $F_k = F_{k-1} + F_{k-2}$ for $k > 1$

Implementació (AVL)



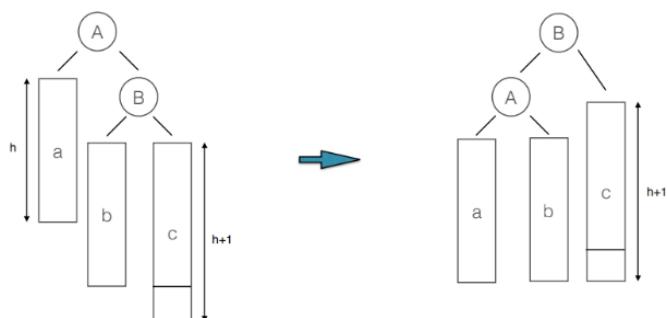
Rotació EE (LL, en anglès)

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



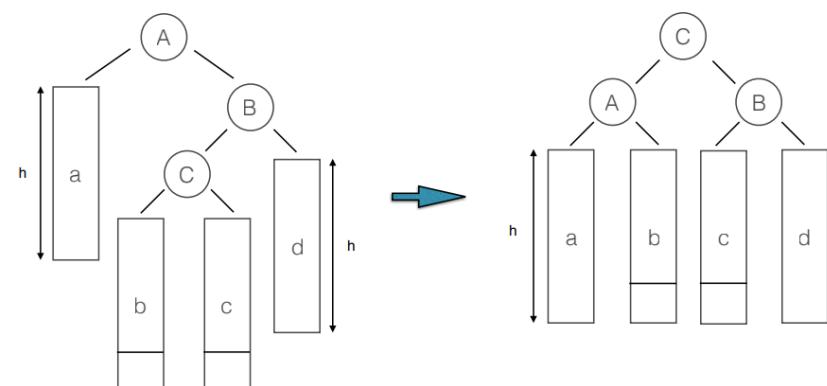
Rotació ED (LR, en anglès)

```
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);
}
```

$$bf(x) = \text{altura } x.\text{left} - \text{altura } x.\text{right} \in [-1, 0, 1]$$

Balance factor

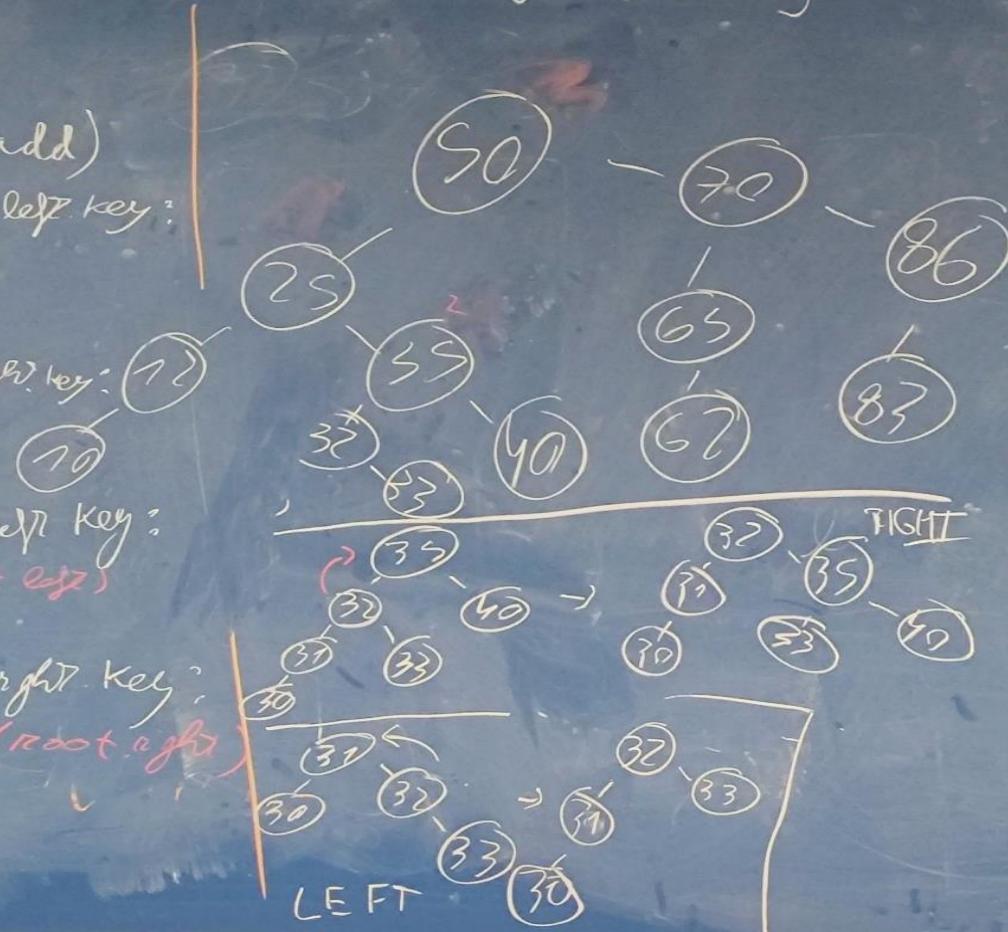
(bf of unbalanced node, key we add)

1. if $bf > 1$ and $\text{key} < \text{root.left.key}$:
RIGHT(root)

2. if $bf < -1$ and $\text{key} > \text{root.right.key}$:
LEFT(root)

3. if $bf > 1$ and $\text{key} > \text{root.left.key}$:
 $\text{root.left} = \text{LEFT}(\text{root.left})$
RIGHT(root)

4. if $bf < -1$ and $\text{key} < \text{root.right.key}$:
 $\text{root.right} = \text{RIGHT}(\text{root.right})$
LEFT(root)



LEFT:



RIGHT

