

- 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

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

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

Notació

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

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.** $T_{\text{mig}}(n) = \sum_{x \in \mathcal{A}, |x|=n} Pr(x)T(x),$
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.

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, $f = O(g)$.

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

Notació Θ ((a): fita exacta asimptòtica)

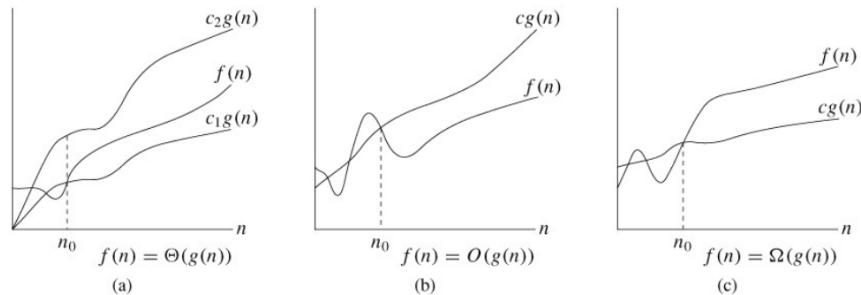
$$\Theta(g) = \{f : \mathbb{N} \rightarrow \mathbb{R}^+ \mid \exists c_1, c_2 \in \mathbb{R}^+ \exists n_0 \in \mathbb{N} \forall n \geq n_0 c_1 g(n) \leq f(n) \leq c_2 g(n)\}$$

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

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

Notació Ω ((c): fita inferior asimptòtica)

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



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

Exercicis

Siguin $k \geq 1$ 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

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

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

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

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 Θ

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'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 $\Theta(n)$.

Algorismes iteratius

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

Algorismes iteratius

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 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: $\Theta(1)$.

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

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 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

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 càlculs 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

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

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

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

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

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

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 subproblemes 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 subproblemes 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 anterior:

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

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 $\phi = 1.618033988\dots$

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 $\text{merge}(v, e, m, d)$ 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:

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

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:

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

Fita inferior dels algorismes d'ordenació

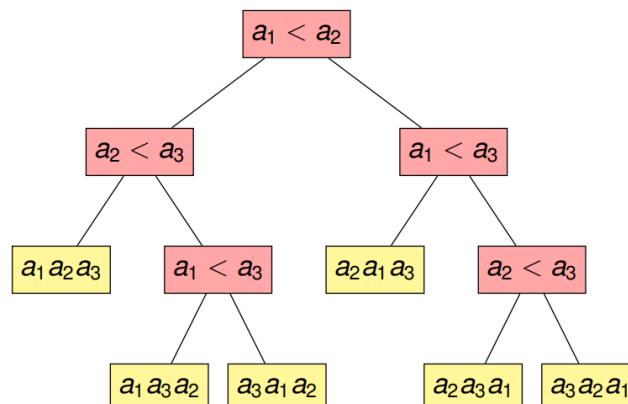
En termes de cost asymptòtic, l'algorisme d'ordenació per fusió és òptim:

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.



- 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 $\Omega(n \log n)$.

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 ≥ 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 (*Algoritmes en C++, EDA*)

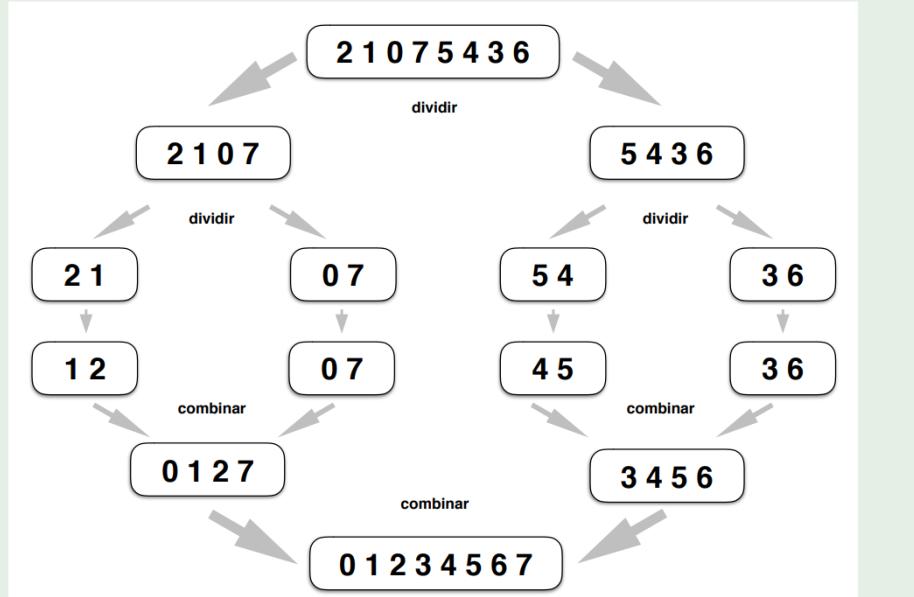
```

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);
    }
}

```

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
---	---	---	---	---	---	---

Observació

Cada comparacióafegeix 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.

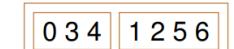
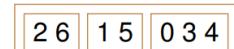
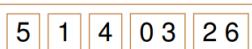
Variants

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*

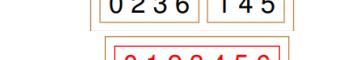
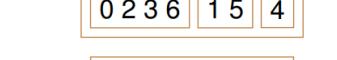
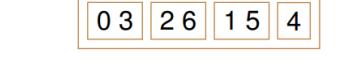
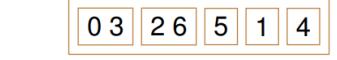
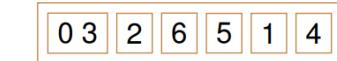
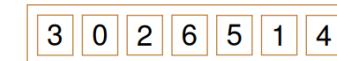
```
function mergesort_queue(a[1...n])
    Q = [] (cua buida)
    for i=1 to n:
        inject(Q, a[i])
    while |Q| > 1:
        inject(Q, merge(eject(Q), eject(Q)))
    return eject(Q)
```



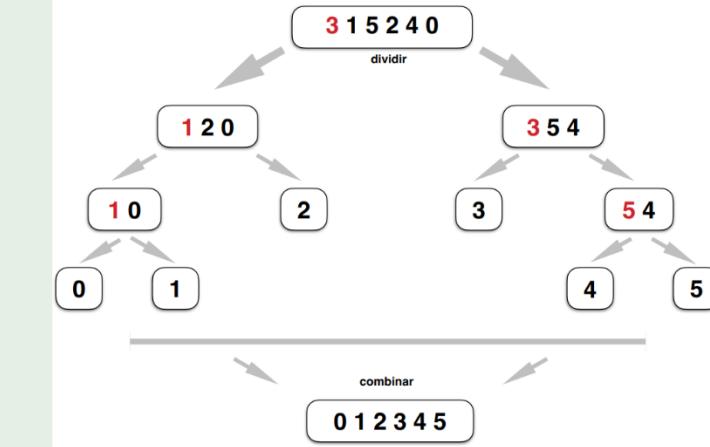
Variants

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*)

```

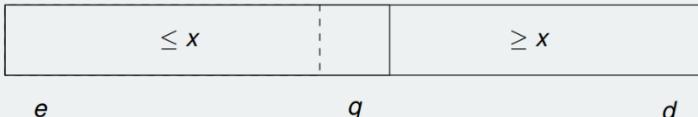
void quicksort(vector<elem>& T) {
    quicksort(T, 0, T.size() - 1);
}

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); } }

```

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

- **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.

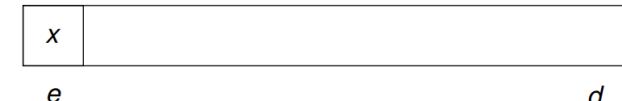
Partició de Hoare (*Algorismes en C++, EDA*)

```

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]);
    } }

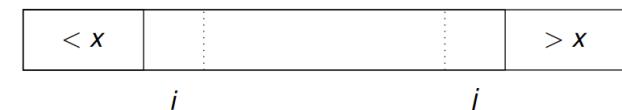
```

- Inici de la funció $\text{partition}(T, e, d)$:

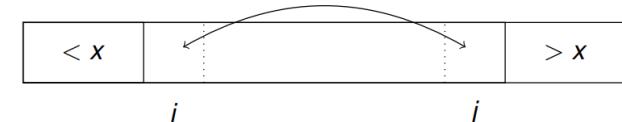


- Bucle principal:

- 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 posicio 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

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

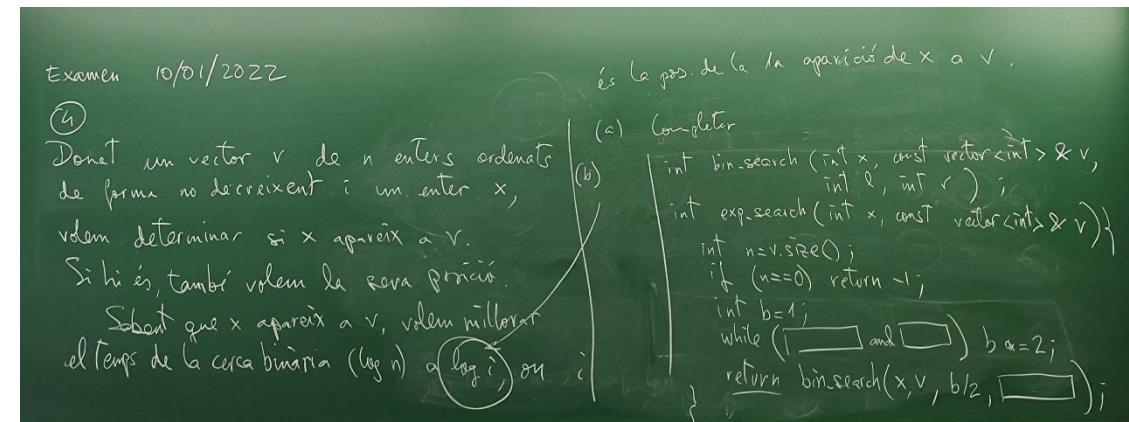
En el **cas pitjor**, el pivot és sempre l'element més petit o el més gran. Com a resultat, una crida recursiva té cost constant i l'altra $T(n-1)$.

Cas pitjor

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

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

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



Algorisme de Karatsuba

Conjectura (Kolmogorov, 1952)

Qualsevol algorisme per multiplicar dos nombres de n dígitos 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 dígitos en temps

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

Algorisme de Karatsuba

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 recurredències divisores, sabem que $T(n) \in \Theta(n^2)$.

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

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

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

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

es pot reescriure com

$$2^n \textcolor{red}{a} + 2^{\lfloor n/2 \rfloor} (\textcolor{red}{c} - \textcolor{red}{a} - \textcolor{red}{b}) + \textcolor{red}{b}$$

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

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

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

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

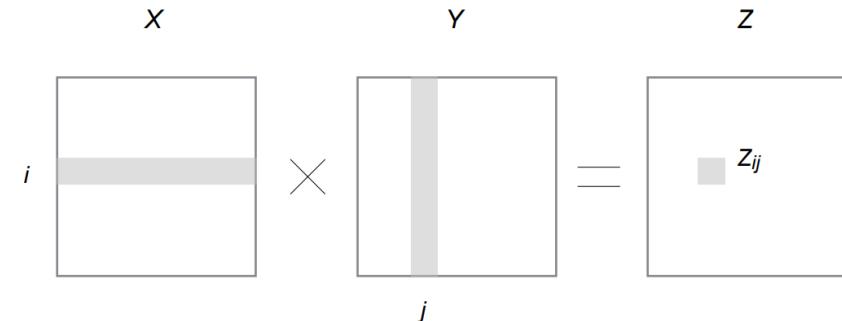
$$T(n) \in \Theta(\log n).$$

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 2×2

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

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

i l'expressem en termes de les submatrius

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

$$\left[\begin{array}{cc} \begin{bmatrix} 4 & 3 \\ 1 & 5 \end{bmatrix} \begin{bmatrix} 2 & 6 \\ 3 & 2 \end{bmatrix} + \begin{bmatrix} 1 & 6 \\ 2 & 7 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 2 & 1 \end{bmatrix} & \begin{bmatrix} 4 & 3 \\ 1 & 5 \end{bmatrix} \begin{bmatrix} 9 & 4 \\ 4 & 1 \end{bmatrix} + \begin{bmatrix} 1 & 6 \\ 2 & 7 \end{bmatrix} \begin{bmatrix} 8 & 3 \\ 3 & 1 \end{bmatrix} \\ \begin{bmatrix} 2 & 1 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} 2 & 6 \\ 3 & 2 \end{bmatrix} + \begin{bmatrix} 5 & 9 \\ 2 & 6 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 2 & 1 \end{bmatrix} & \begin{bmatrix} 2 & 1 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} 9 & 4 \\ 4 & 1 \end{bmatrix} + \begin{bmatrix} 5 & 9 \\ 2 & 6 \end{bmatrix} \begin{bmatrix} 8 & 3 \\ 3 & 1 \end{bmatrix} \end{array} \right]$$

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

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

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:

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

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	$\Theta(\log n), \Theta(\log n)$	$\Theta(\log n), \Theta(\log n)$	$\Theta(\log n), \Theta(\log n)$

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 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**.



En vectors, cal desplaçar els elements.

En estructures no ordenades, cal comprovar repetits.

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

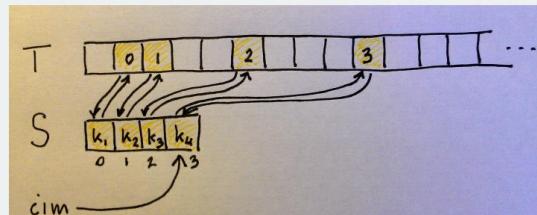
Solució: idea

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.

- 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.

Qüestió

Per què els dos últims dígits 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 dígits 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**.

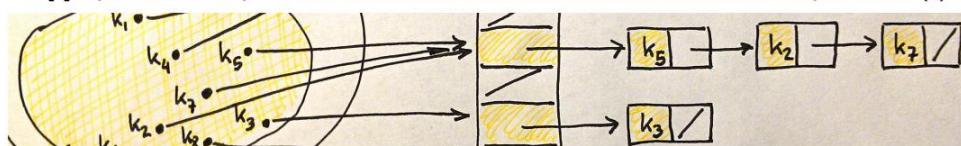
Col·lisions

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 $\Theta(1)$.

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

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·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?

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

$$67 \cdot 128^3 + 76 \cdot 128^2 + 82 \cdot 128^1 + 83 \cdot 128^0$$

$$= 141.764.947$$

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

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

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

$$h(k) = \lfloor 1024 \cdot (123 \cdot A - \lfloor 123 \cdot A \rfloor) \rfloor$$

$$= \lfloor 1024 \cdot (76,0181808 - 76) \rfloor$$

$$= \lfloor 1024 \cdot 0,0181808 \rfloor = 18.$$

Adreçament obert

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

- tots els elements s'emmagatzemem 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

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

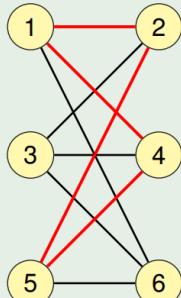
Definicions i terminologia

Definició

Un **graf** és un parell (V, E) on:

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

Exemple: graf connex i cíclic

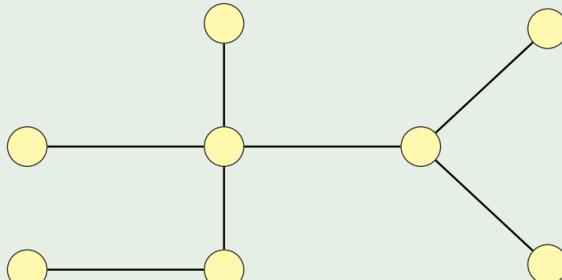


- **Connex**: hi ha un camí entre dos vèrtexs qualssevol
- **Cíclic**: hi ha cicles com $1 \rightarrow 2 \rightarrow 5 \rightarrow 4 \rightarrow 1$

Definició

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

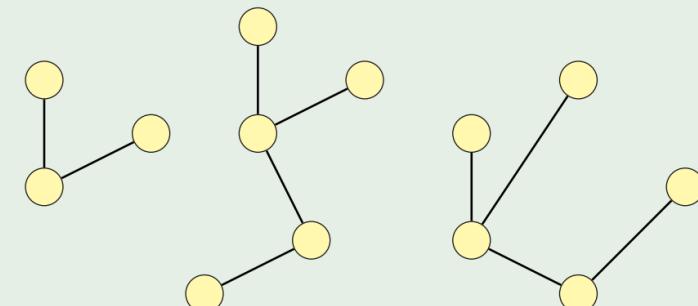
Exemple: arbre



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:

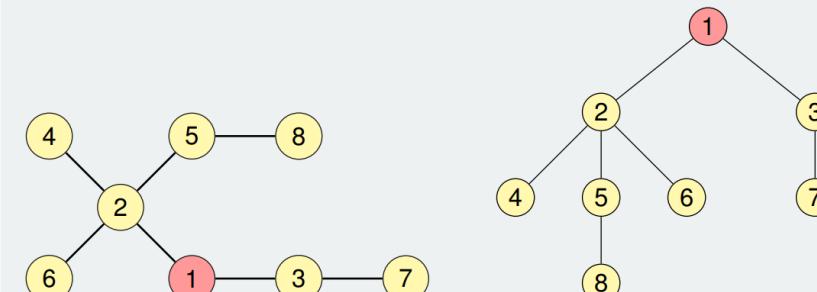
- ① G és un **arbre**
- ② Tot parell de vèrtexs de G estan units per un **camí únic**
- ③ G és connex i $m = n - 1$
- ④ G és connex però, si s'hi elimina una aresta, s'obté un graf inconnex
- ⑤ G és acíclic i $m = n - 1$
- ⑥ G és acíclic però, si s'hi afegeix una aresta, s'obté un graf cíclic

Definició

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

Representació

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

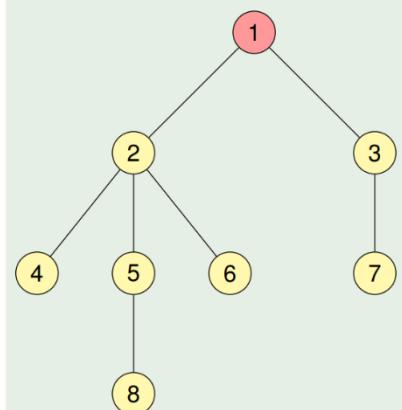


Terminologia

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

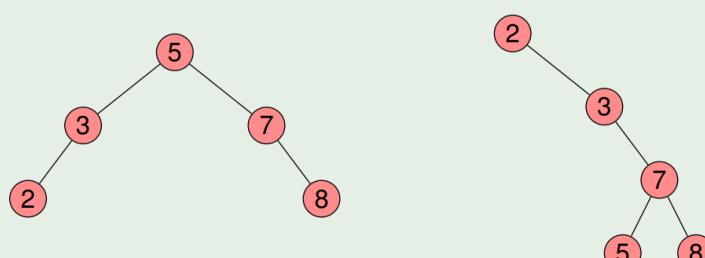
Un **arbre binari** és un arbre arrelat on cada node té un màxim de dos fills, que es diferencien com a **fill esquerre** i **fill dret**.

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

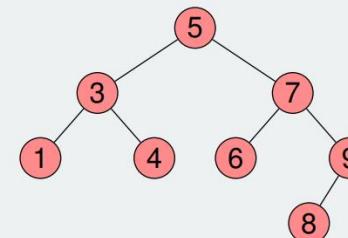
Llista ordenada dels elements d'un ABC

- 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))
  
```

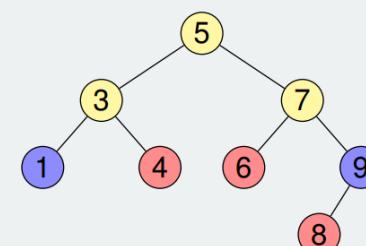
El recorregut inordre de l'ABC següent és: 1, 3, 4, 5, 6, 7, 8, 9.



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

Trobar el mínim i el màxim d'un ABC

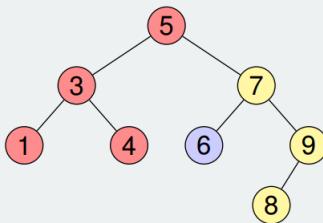
- ② **Mínim o màxim**



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)

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);
}
  
```

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

Cerca

Retornar un apuntador al node de l'arbre apuntat per p que conté $clau$ (o nullptr 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);
        }
    }
}
```

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;
        }
    }
}
```

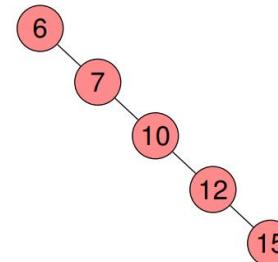
Esborrar mínim

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

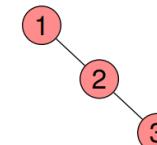
En els **arbres binaris de cerca**, hem vist que el cost de les operacions és $\Theta(h)$, on h és l'alçària. Però h pot coincidir amb el nombre de nodes:



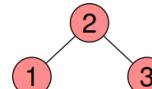
Per tant, el cost en cas pitjor és de $\Theta(n)$, on n és el nombre de nodes.

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



tindríem



Com mantenir els subarbres equilibrats?

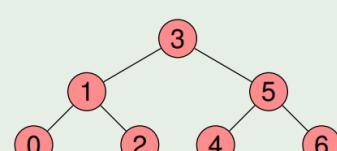
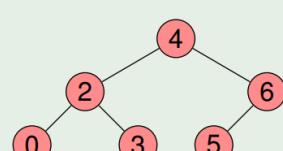
- 1 Forçant que l'ABC sigui **complet**.

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

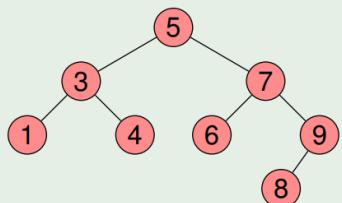
- és buit o
- la diferència d'alçàries dels seus subarbres és com a màxim d'1 i els seus subarbres també estan equilibrats

Introduction

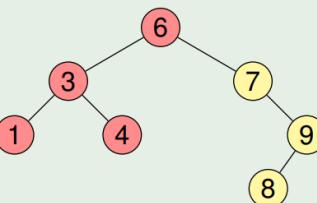
Els ABC amb la condició d'equilibri s'anomenen **arbres AVL**.

Exemple

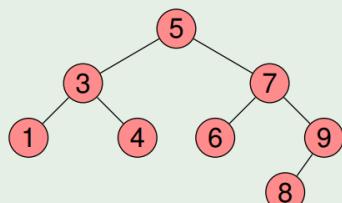
L'arbre de l'esquerra està equilibrat, però si n'esborrem l'arrel com en els ABC, deixarà d'estar equilibrat.



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)

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

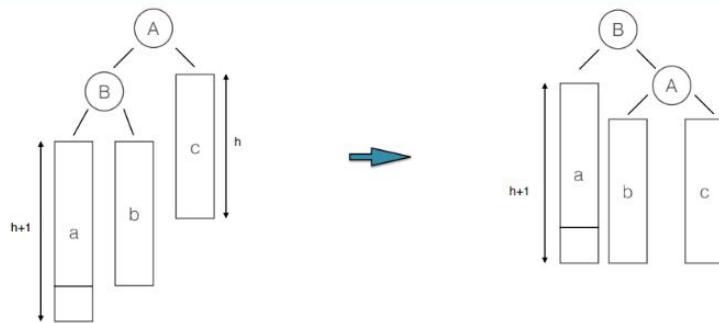
```
static int alcaria (Node* p) {
    return p ? p->alc : -1;
}

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.

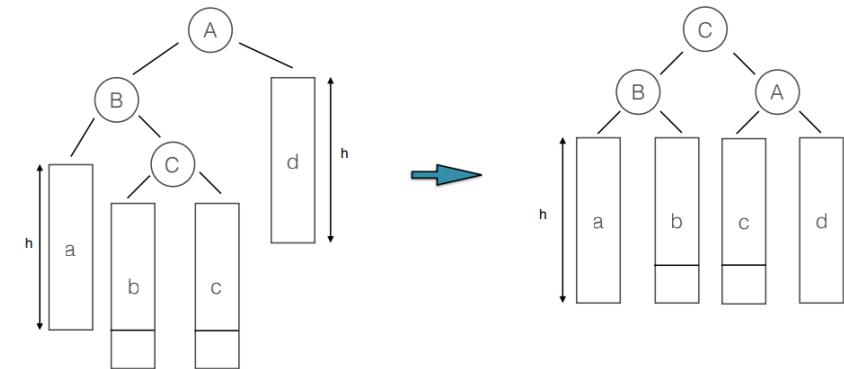
Totes quatre tenen cost $\Theta(1)$.

Implementació (AVL)



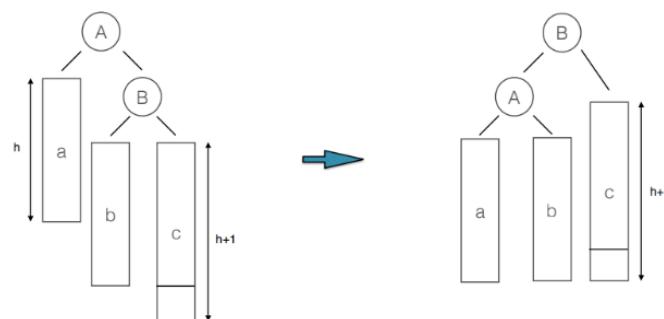
Rotació EE (LL, en anglès)

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



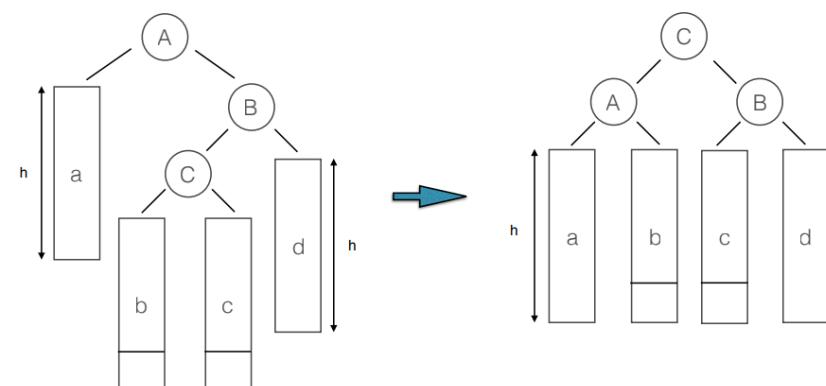
Rotació ED (LR, en anglès)

```
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 $\Theta(\log n)$.

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:

- $n_0 = 1$
- $n_1 = 2$
- $n_k = \underbrace{n_{k-1}}_{\text{cal un subarbre}} + \underbrace{n_{k-2}}_{\text{l'alçària } k-2 \text{ dona}} + 1$
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 Prentent 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 $k > 1$

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

balance factor

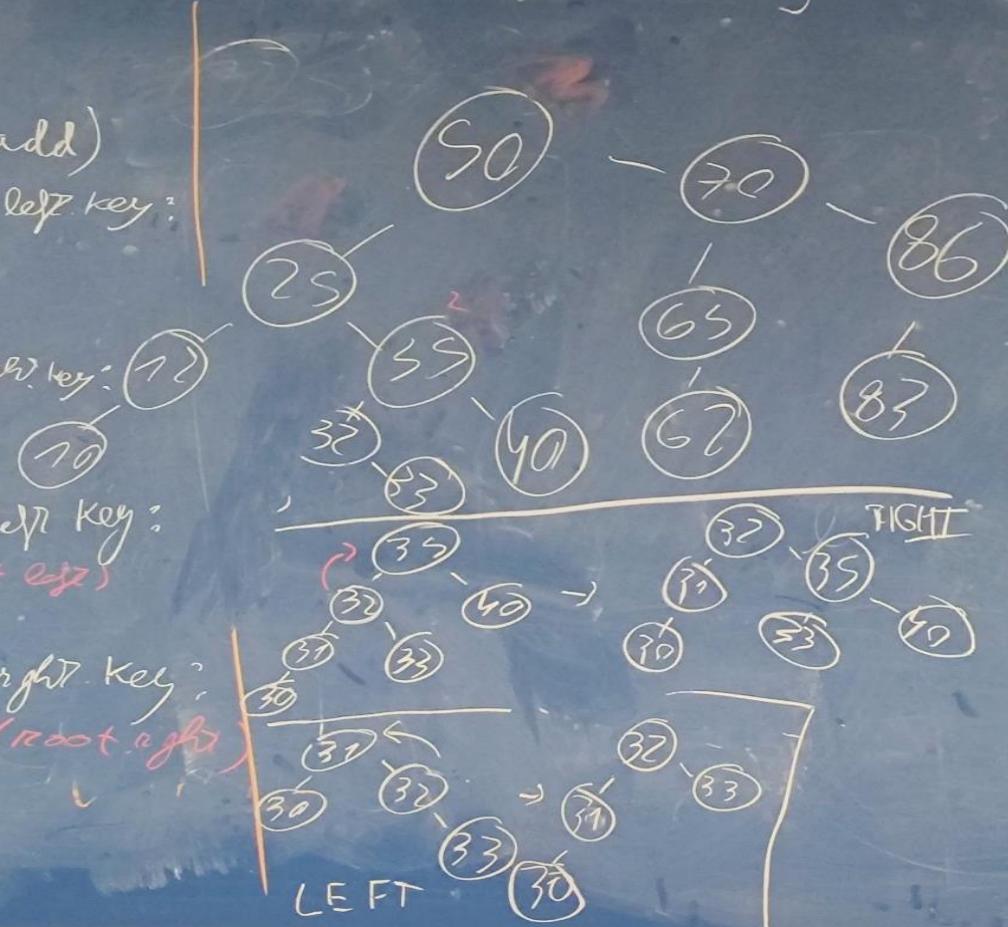
(bf of unbalanced node, key we add)

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

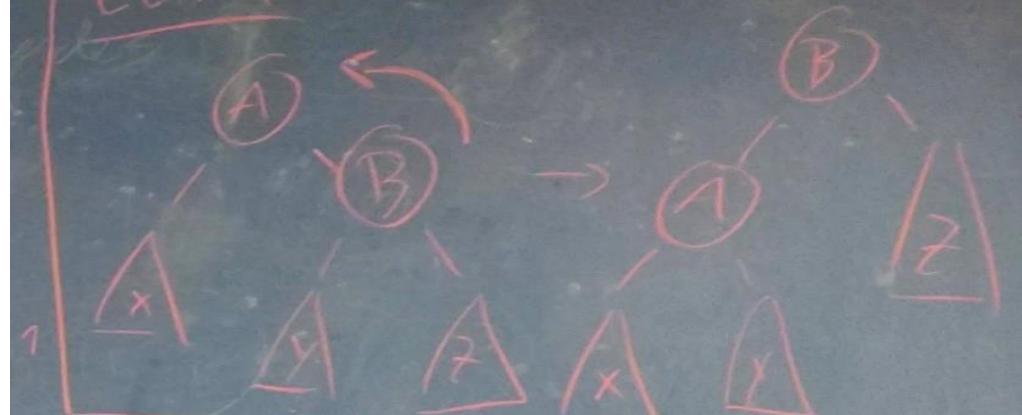
2. if $bf < -1$ and $\text{key} > \text{root.right.key}$:
LEFT(root)

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

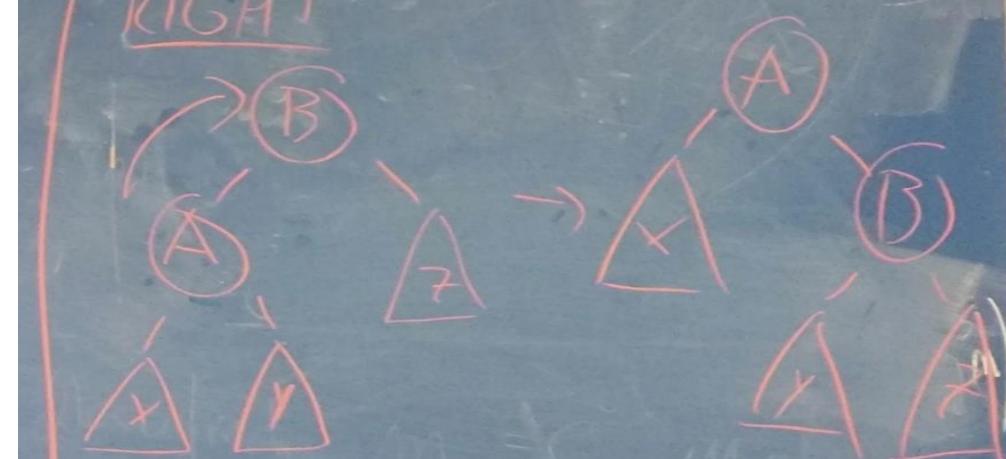
4. if $bf < -1$ and $\text{key} < \text{root.right.key}$:
 $\text{root.right} = \text{RIGHT}(\text{root.right})$
LEFT(root)



LEFT:



RIGHT

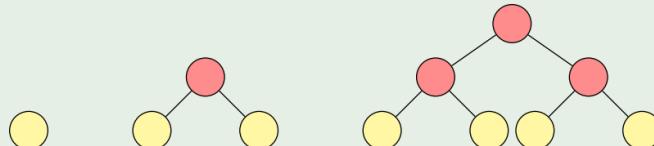


Arbres binaris perfectes

Definicions

- El **nivell** d'un node en un arbre és la distància de l'arrel al node.
- Un **arbre binari** és **perfecte** si totes les fulles són al mateix nivell (per tant, tots els nodes interns tenen dos fills).

Exemples



Definicions

- L'**alçària** d'un node és la distància màxima del node a una fulla.
- L'**alçària** d'un arbre és l'alçària de l'arrel (o el nivell màxim dels nodes).

Proposició

Un arbre binari perfecte d'alçària h té $2^{h+1} - 1$ nodes.

Demostració

Fem inducció en l'alçària. Sigui T un arbre binari perfecte d'alçària h .

- **Base d'inducció:** $h = 0$.
L'arbre té un sol node, i $1 = 2^{0+1} - 1$.
- **Pas d'inducció:** $h > 0$.
Els subarbres esquerre i dret tenen alçària $h - 1$ i, per hipòtesi d'inducció, cadascun té $2^h - 1$ nodes. El nombre de nodes de T és la suma d'aquests nodes més un (l'arrel):

$$\text{nodes de } T = 2(2^h - 1) + 1 = 2^{h+1} - 2 + 1 = 2^{h+1} - 1.$$

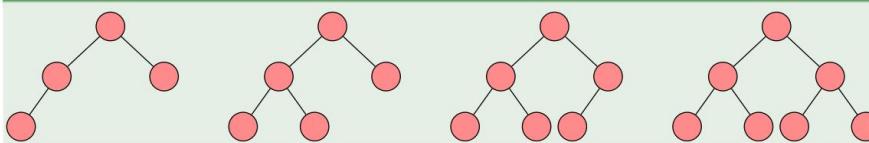
Arbres binaris complets

Definició

Un **arbre binari** d'alçària h és **complet** si

- 1 hi ha tots els nodes possibles amb nivells $0 \dots h - 1$
- 2 tots els nodes de nivell h són el màxim a l'esquerra

Exemple: arbres binaris complets d'alçària 2



Proposició

Un arbre binari complet d'alçària h té entre 2^h i $2^{h+1} - 1$ nodes.

Demostració

Sigui T un arbre binari complet d'alçària h :

- El **mínim** nombre de nodes de T es produeix quan té un sol node a nivell h . Com que fins a nivell $h - 1$, T té $2^h - 1$ nodes, sumant l'únic node a nivell h , s'obtenen 2^h nodes.
- El **màxim** nombre de nodes de T correspon a un arbre perfecte d'alçària h , que té $2^{h+1} - 1$ nodes.

Corol·lari

L'alçària d'un arbre binari complet de n nodes és $\lfloor \log n \rfloor \in \Theta(\log n)$.

Demostració

Per la proposició anterior, un arbre binari complet d'alçària h i n nodes compleix:

$$2^h \leq n < 2^{h+1}.$$

Si prenem logaritmes en base 2, tenim

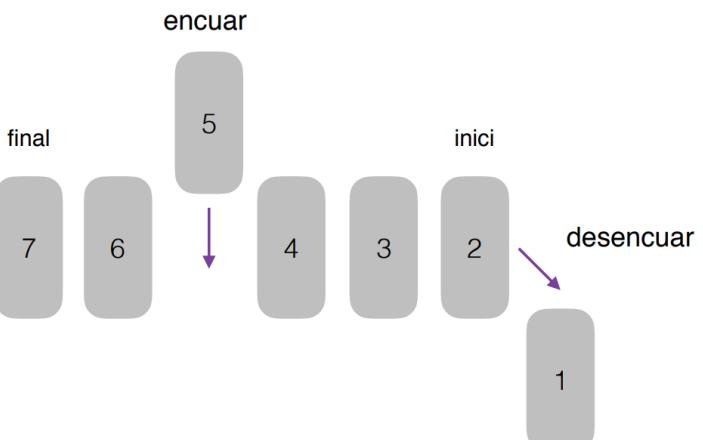
$$h \leq \log n < h + 1.$$

I prenent la part baixa del logaritme,

$$h = \lfloor \log n \rfloor.$$

Per tant, $h \in \Theta(\log n)$.

Cua amb prioritat



Operacions

Definició

Una **cua amb prioritat** és una estructura de dades que disposa de dues operacions bàsiques:

- **afegir**: inserir un element (clau més informació)
- **treure_min (treure_max)**: esborrar i retornar l'element amb la clau més petita (més gran)

Cues amb prioritat de l'STL

Descripció

- La implementació fa servir *heaps*
- Per defecte, *max-heaps* (clau més alta disponible amb cost $\Theta(1)$)
- Mètodes: push, pop, top, empty, size

Exemple: *max-heap*

```
#include <queue>
int main() {
    priority_queue<int> Q;
    Q.push(5);
    Q.push(3);
    cout << Q.top();
    Q.pop();
}
```

S'obté 5 al canal de sortida.

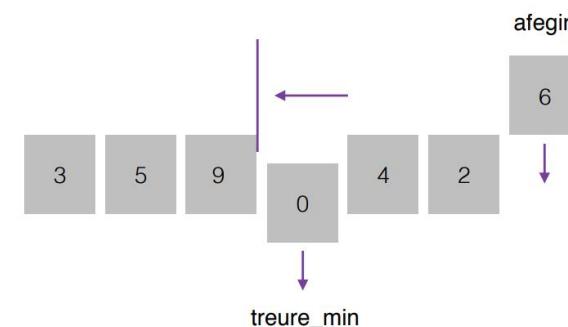
Exemple: *min-heap*

```
#include <queue>
int main() {
    priority_queue<int, vector<int>, greater<int> > Q;
    Q.push(5);
    Q.push(3);
    cout << Q.top();
    Q.pop();
}
```

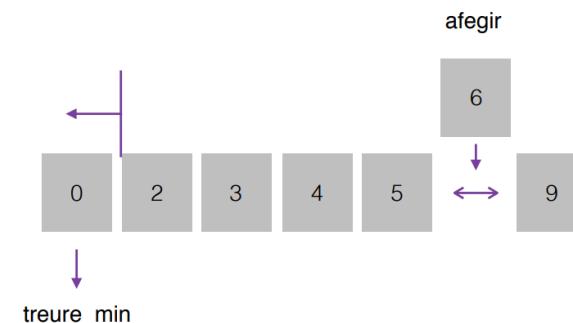
S'obté 3 al canal de sortida.

Implementacions senzilles

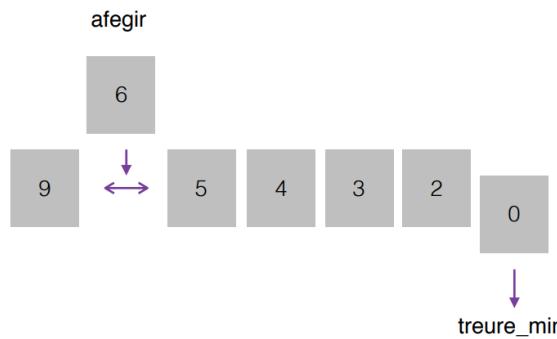
implementacions	afegir	treure_min
▷ vector desordenat	$\Theta(1)$	$\Theta(n)$
vector ordenat	$\Theta(n)$	$\Theta(n)$
vector ordenat (decreixent)	$\Theta(n)$	$\Theta(1)$
vector circular ordenat	$\Theta(n)$	$\Theta(1)$
heaps	$\Theta(\log n)$	$\Theta(\log n)$



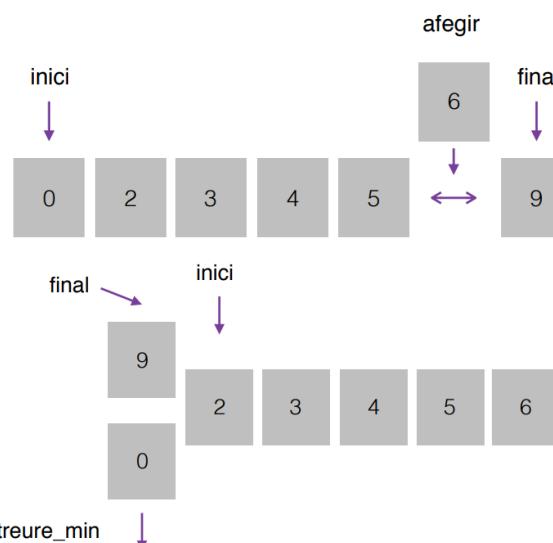
implementacions	afegir	treure_min
▷ vector desordenat	$\Theta(1)$	$\Theta(n)$
vector ordenat	$\Theta(n)$	$\Theta(n)$
vector ordenat (decreixent)	$\Theta(n)$	$\Theta(1)$
vector circular ordenat	$\Theta(n)$	$\Theta(1)$
heaps	$\Theta(\log n)$	$\Theta(\log n)$



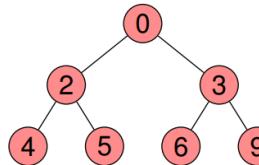
implementacions	afegir	treure_min
vector desordenat	$\Theta(1)$	$\Theta(n)$
vector ordenat	$\Theta(n)$	$\Theta(n)$
▷ vector ordenat (decreixent)	$\Theta(n)$	$\Theta(1)$
vector circular ordenat	$\Theta(n)$	$\Theta(1)$
heaps	$\Theta(\log n)$	$\Theta(\log n)$



implementacions	afegir	treure_min
vector desordenat	$\Theta(1)$	$\Theta(n)$
vector ordenat	$\Theta(n)$	$\Theta(n)$
vector ordenat (decreixent)	$\Theta(n)$	$\Theta(1)$
▷ vector circular ordenat	$\Theta(n)$	$\Theta(1)$
heaps	$\Theta(\log n)$	$\Theta(\log n)$



implementacions	afegir	treure_min
vector desordenat	$\Theta(1)$	$\Theta(n)$
vector ordenat	$\Theta(n)$	$\Theta(n)$
vector ordenat (decreixent)	$\Theta(n)$	$\Theta(1)$
▷ heaps	$\Theta(\log n)$	$\Theta(\log n)$



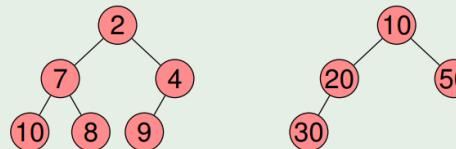
Definició

Un *min-heap* és un arbre binari complet on la clau d'un node és sempre més petita que les claus dels seus fills.

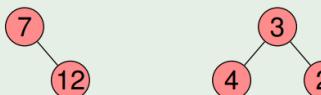
Exemples

- Quan parlem de *heaps* sense especificar res més, ens referirem als *min-heaps*

Són *min-heaps*:



No són *min-heaps*:

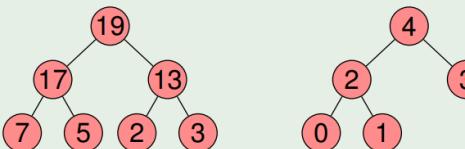


Definició

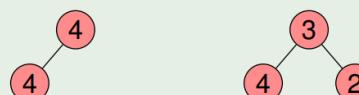
Un *max-heap* és un arbre binari complet on la clau d'un node és sempre més gran que les claus dels seus fills.

Exemples

Són *max-heaps*:



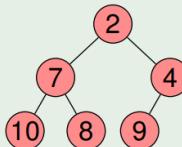
No són *max-heaps*:



Els *heaps* es representen de manera compacta mitjançant vectors.

Representació d'un *heap* mitjançant vectors

El *heap*



es representa amb el vector

	2	7	4	10	8	9		
0	1	2	3	4	5	6	7	8

No calen apuntadors perquè per a un node en posició i :

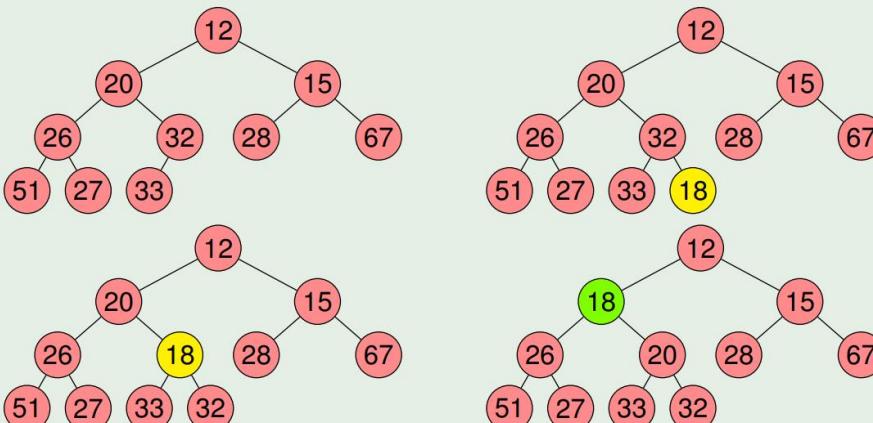
- el **pare** és a la posició $\lfloor i/2 \rfloor$
- el **fill esquerre** és a la posició $2i$
- el **fill dret** és a la posició $2i + 1$

Operacions bàsiques

Operació afegir

S'afegeix l'element en la **següent posició lliure** del vector i **es fa ascendir** fins la posició en què es torna a complir la propietat del *heap*.

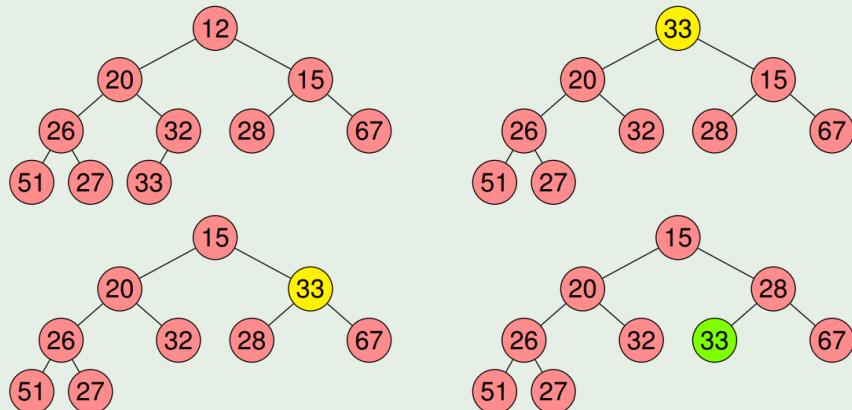
Exemple: afegir la clau 18



Operació treure-min

L'element en l'**última posició** del vector **es trasllada a la primera** i **es fa descendir** fins que troba la seva posició. Es retorna l'antiga arrel.

Exemple: esborrar el mínim



Costos de les operacions en *heaps*

operacions	cas pitjor	cas mitjà
afegir	$\Theta(\log n)$	$\Theta(1)$
treure_min	$\Theta(\log n)$	$\Theta(\log n)$

Implementació recursiva

Definició de la classe CuaPrio

El *heap* es forma en la taula `t`. La posició 0 no s'utilitza.

```
template <typename Elemt>
class CuaPrio {
```

```
private:
vector<Elemt> t;
```

Constructura

Crea una cua amb prioritat buida. Cost: $\Theta(1)$.

```
CuaPrio () {
    t.push_back(Elemt());
}
```

Consultar la talla

Retorna la talla de la cua amb prioritat. Cost: $\Theta(1)$.

```
int talla () {
    return t.size()-1;
}
```

Consultar si és buida

Indica si la cua amb prioritat és buida. Cost: $\Theta(1)$.

```
bool buida () {
    return t.talla() == 0;
}
```

Retornar element mínim

Retorna un element amb prioritat mínima. Cost: $\Theta(1)$.

```
Elem minim () {
    if (buida()) throw "CuaPrio_buida";
    return t[1];
}
```

afegir

Afegeix un nou element. Cost: $\Theta(\log n)$.

```
void afegir (Elem& x) {
    t.push_back(x);
    surar(talla());
}
```

treure_min

Treu i retorna l'element mínim. Cost: $\Theta(\log n)$.

```
Elem treure_min () {
    if (buida()) throw "CuaPrio_buida";
    Elem x = t[1];
    t[1] = t.back();
    t.pop_back();
    enfonsar(1);
    return x;
}
```

Implementació recursiva: funcions privades

surar

Fer ascendir un element fins que ocupa una posició compatible amb la condició d'ordenació del *heap*. Cost: $\Theta(\log n)$.

```
void surar (int i) {
    if (i != 1 and t[i/2] > t[i]) {
        swap(t[i],t[i/2]);
        surar(i/2);
    }
}
```

enfonsar

Fer descendir un element fins que ocupa una posició compatible amb la condició d'ordenació del *heap*. Cost: $\Theta(\log n)$.

```
void enfonsar (int i) {
    int n = talla();
    int c = 2*i;
    if (c <= n) {
        if (c+1 <= n and t[c+1] < t[c]) c++;
        if (t[i] > t[c]) {
            swap(t[i],t[c]);
            enfonsar(c);
        }
    }
}
```

Les operacions que canviem són **afigir** i **treure_min**, on les antigues **surar** i **enfonsar** estan optimitzades.

Els costos asymptòtics no canviem: $\Theta(\log n)$.

afegir

```
void afegir (Elem& x) {
    t.push_back(x);
    int i = talla();
    while (i != 1 and t[i/2] > x) {
        t[i] = t[i/2];
        i = i/2;
    }
    t[i] = x;
}
```

treure_min

```
Elem treure_min () {
    if (buida()) throw "CuaDePrio_buida";
    int n = talla();
    Elem e = t[1], x = t[n];
    t.pop_back(); --n;
    int i = 1; c = 2*i;
    while (c <= n) {
        if (c+1 <= n and t[c+1] < t[c]) ++c;
        if (x <= t[c]) break;
        t[i] = t[c];
        i = c;
        c = 2*i;
    }
    t[i] = x;
    return e;
}
```

Algorisme bàsic

Heapsort és un algorisme d'ordenació basat en les cues amb prioritat.

Donat un vector de n elements,

① afegeix els n elements a un *heap*: $\Theta(n \log n)$

② fa n operacions **treure_min** per construir un vector ordenat: $\Theta(n \log n)$

El temps total és $\Theta(n \log n)$, el mínim asimptòtic per a un algorisme d'ordenació.

Heapsort

Amb vectors separats per al *heap* i l'entrada/sortida.

Temps: $\Theta(n \log n)$.

Espai: $\approx 2n$.

```
template <typename elem>
void heapsort (vector<elem>& v) {
    n = v.size();
    CuaPrio<elem> h;
    for (int i = 0; i < n; ++i)
        h.afegeix(v[i]);
    for (int i = 0; i < n; ++i)
        v[i] = h.treure_min();
}
```

Exemple

Suposem que partim del vector:

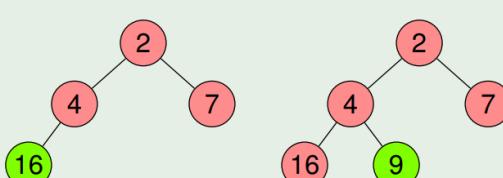
4	2	7	16	9	3	1	5
---	---	---	----	---	---	---	---

i afegim els elements a un *heap*, un per un.

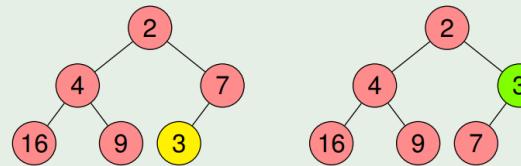
+4, +2, +7:



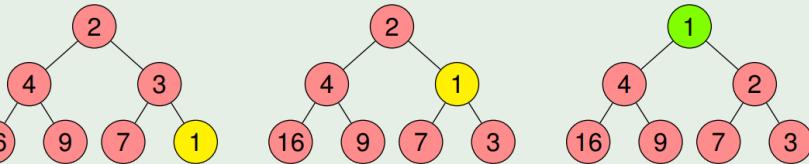
+16, +9:



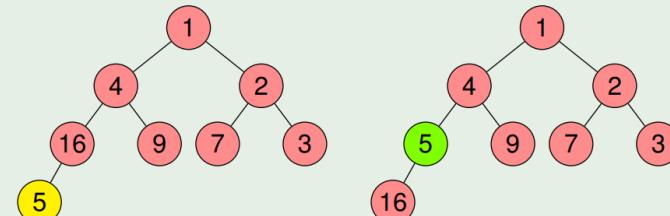
+3:



+1:



+5:

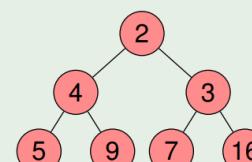
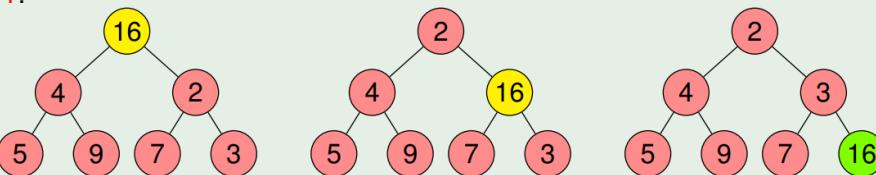


El *heap* resultant s'emmagatzema en el vector:

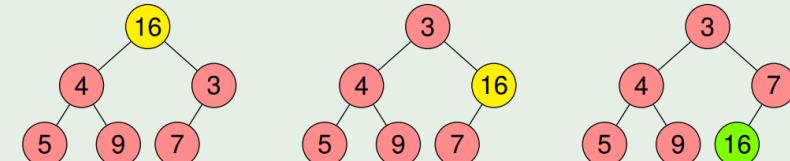
1	4	2	5	9	7	3	16
1	2	3	4	5	6	7	8

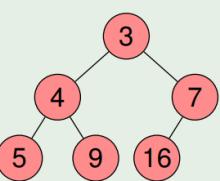
Algorisme bàsic

-1:

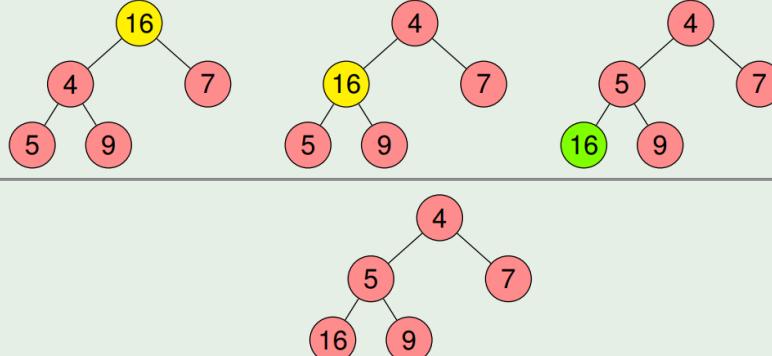


-2:

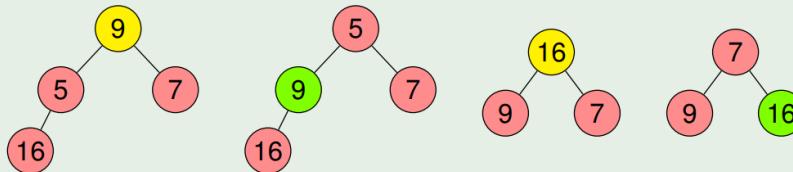




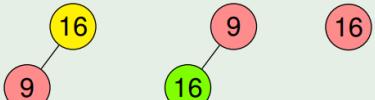
-3:



-4, -5:



-7, -9, -16:



Exemple: evolució dels vectors (operació **afegir**)

entrada/sortida

						3	1	5
1	2	3	4	5	6	7	8	

2	4	7	16	9			
---	---	---	----	---	--	--	--

heap

Exemple: evolució dels vectors (operació **afegir**)

entrada/sortida

							1	5
1	2	3	4	5	6	7	8	

2	4	3	16	9	7		
---	---	---	----	---	---	--	--

heap

Exemple: evolució dels vectors (operació **afegir**)

entrada/sortida

	2	7	16	9	3	1	5
1	2	3	4	5	6	7	8

heap

Exemple: evolució dels vectors (operació **treure_min**)

entrada/sortida

1	2	3	4	5	7	9	16
1	2	3	4	5	6	7	8

--	--	--	--	--	--	--	--

heap

Idea general

Implementar l'algorithm sobre un únic vector fent una divisió en:

- una part esquerra per mantenir el *heap*
- una part dreta per a l'entrada/sortida

Cada cop que es fa una operació de **treure_min**, s'escriu el mínim com a primer element de la part dreta. Els elements queden ordenats de manera **descendent**.

Si es volen en ordre ascendent, es pot fer servir un *max-heap*.

Funció buildHeap

Construir el *heap* en temps $\Theta(n)$ en cas pitjor en lloc de $\Theta(n \log n)$:

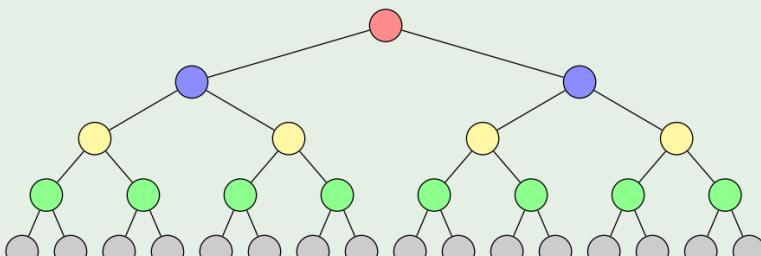
- ① Introduir els elements en el *heap* en qualsevol ordre (i temps lineal)
- ② Si el *heap* té h nivells, per a $i = h - 1, h - 2, \dots, 1$:
 - **enfonsar** tots els elements del nivell i

El fet que la majoria de *subheaps* tractats siguin petits fa que el nombre d'intercanvis fets per **enfonsar** sigui lineal.

Exemple

Per a un *heap* de 31 nodes, hi ha

- 8 *heaps* de mida 3 (arrel en verd)
- 4 *heaps* de mida 7 (arrel en groc)
- 2 *heaps* de mida 15 (arrel en blau)
- 1 *heap* de mida 31 (arrel en vermell)



Constructor que pren els elements d'un vector com a entrada.

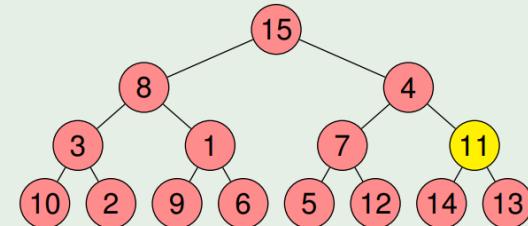
```
explicit PrioQueue (const vector<Elem>& v)
    : t(v.size() + 1) {
    for (int i = 0; i < v.size(); ++i)
        t[i + 1] = v[i];
    buildHeap();
}
```

Establir propietat d'ordre del heap a partir d'una ordenació arbitrària d'ítems.

```
void buildHeap () {
    for (int i = size() / 2; i > 0; --i)
        enfonsar(i);
}
```

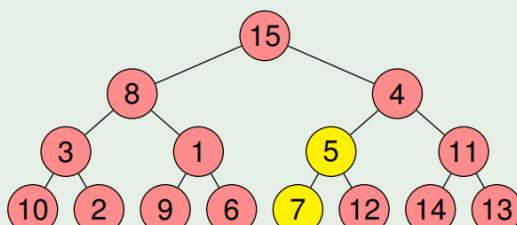
Exemple de buildHeap

Després de **enfonsar(7)**



Exemple de buildHeap

Després de **enfonsar(6)**



El temps de càlcul de **buildHeap** està fitat per la suma de les alçàries de tots els nodes.

Volem demostrar que aquesta suma és $O(n)$.

Teorema

Per a l'arbre binari perfecte d'alçària h i $2^{h+1} - 1$ nodes, la suma de les alçàries dels seus nodes és $2^{h+1} - 1 - (h + 1)$.

Demostració

Com que hi ha 2^i nodes d'alçària $h - i$, la suma de totes les alçàries és:

$$\begin{aligned} S &= \sum_{i=0}^h 2^i(h-i) \\ &= h + 2(h-1) + 4(h-2) + 8(h-3) + 16(h-4) + \dots + 2^{h-1}(1) \end{aligned}$$

Multiplicant per 2, tenim

$$2S = 2h + 4(h-1) + 8(h-2) + 16(h-3) + \dots + 2^h(1)$$

calculem $2S - S$ i obtenim

$$\begin{aligned} S &= -h + 2 + 4 + 8 + \dots + 2^{h-1} + 2^h \\ &= -h + (2^{h+1} - 1) - 1 \\ &= 2^{h+1} - 1 - (h + 1) \end{aligned}$$

Donat un arbre binari complet T de n nodes i alçària h , hem vist que $2^h \leq n$.

Per tant, $2^{h+1} \leq 2n$.

La suma d'alçàries de T és com a màxim la de l'arbre binari perfecte d'alçària h que, pel teorema, és:

$$\begin{aligned} 2^{h+1} - 1 - (h + 1) &< 2^{h+1} \\ &\leq 2n \in O(n). \end{aligned}$$

Corol·lari

La suma d'alçàries d'un arbre binari complet de n nodes és $O(n)$.

Problema de selecció

Donada una llista S de naturals i un $k \in \mathbb{N}$, determinar el k -èsim element més petit de S .

Fent servir *heaps*, podem trobar un nou algorisme:

- ➊ Construir un *min-heap* a partir de $S \rightarrow \Theta(n)$
- ➋ Efectuar k operacions **treure_min** del *min-heap* $\rightarrow \Theta(k \log n)$
- ➌ Retornar l'últim element extret $\rightarrow \Theta(1)$

Cost total: $\Theta(n + k \log n)$.

La **mediana** correspon a $k = n/2$. Cost: $\Theta(n \log n)$.

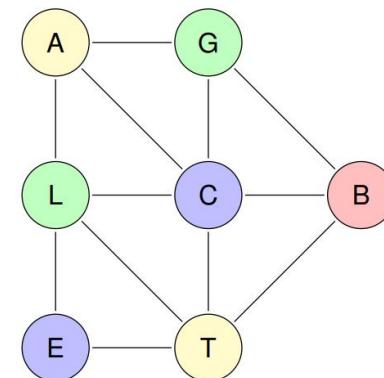
En el cas $k = \frac{n}{\log n}$, el cost és $\Theta(n)$.

Exemple: acolorir un mapa amb el mínim nombre de colors

Quin és el **mínim nombre de colors** necessari per acolorir un mapa de manera que els països veïns tinguin colors diferents?
Però tot mapa el podem representar com un graf:

- Un **vèrtex** correspon a un país
- Una **aresta** correspon a una frontera

De fet, el graf és **planar**: es pot dibuixar sense que les arestes es tallin.



Fent servir grafs, podem utilitzar el teorema següent.

Teorema dels quatre colors (Appel/Haken, 1976)

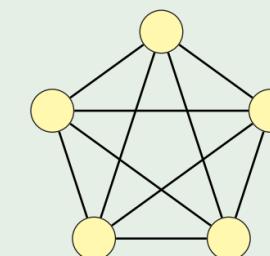
Tot graf planar es pot acolorir amb 4 colors.

Per tant, tot mapa es pot acolorir amb 4 colors
(amb algunes simplificacions, com ara que les fronteres sempre són de dimensió 1 o que els països són connexos).

Exemple: connectar cinc objectes en el pla

No es poden connectar 5 objectes sobre el pla sense creuar connexions.

En teoria de grafs, és el mateix que dir que el graf K_5



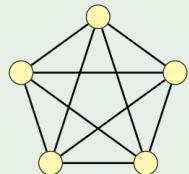
no es pot dibuixar en el pla sense creuar arestes (no és **planar**).

Definició

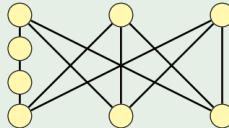
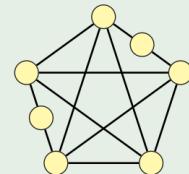
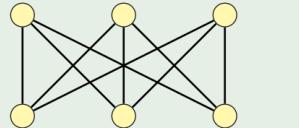
Una **subdivisió** d'un graf és un graf format subdividint les seves arestes en camins d'una o més arestes.

Exemple: subdivisions de K_5 i $K_{3,3}$

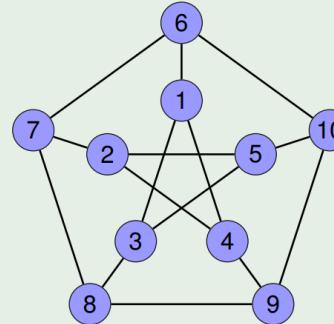
K_5 :



$K_{3,3}$:



Exemple: graf de Petersen



Formalment, és $GP(5, 2) = (V, E)$ on

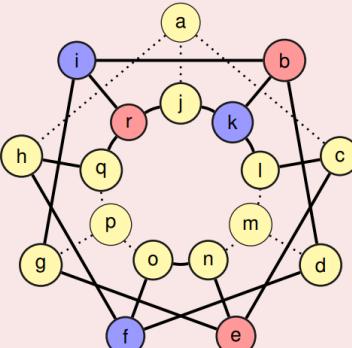
- $V = \{1, \dots, 10\}$
- $E = \{\{1, 3\}, \{1, 4\}, \{1, 6\}, \{2, 4\}, \{2, 5\}, \{2, 7\}, \{3, 5\}, \{3, 8\}, \{4, 9\}, \{5, 10\}, \{6, 7\}, \{6, 10\}, \{7, 8\}, \{8, 9\}, \{9, 10\}\}$

Teorema (Kuratowski, 1922)

Un graf és planar si i només si no conté cap subgraf que sigui una subdivisió de K_5 o $K_{3,3}$.

Solució

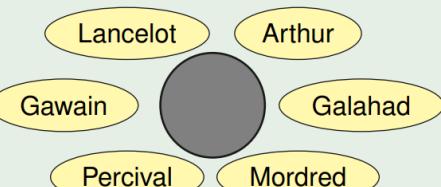
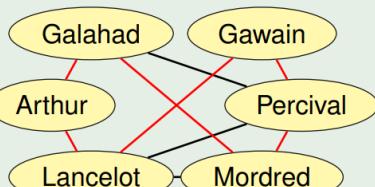
El subgraf destacat és una subdivisió de $K_{3,3}$ (en blau i vermell).



Exemple: el rei Artur i la taula rodona

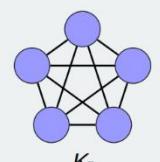
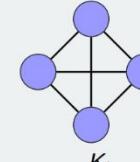
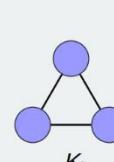
El rei Artur vol asseure els seus cavallers a la taula rodona però ha d'evitar que alguns d'ells seguin colze contra colze.

S'introduceix el graf en un algorisme que troba un cicle que passa per tots els vèrtexos (cicle Hamiltonià): aquest serà l'ordre en què hauran de seure.

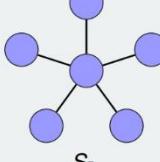
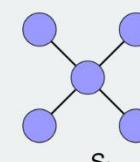
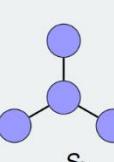


Tipus de grafs

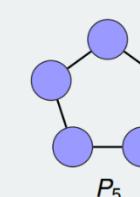
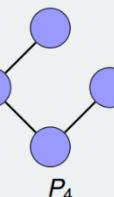
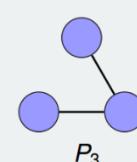
- **Complets:** K_i és el graf complet de i vèrtexs.



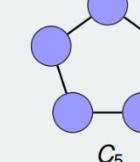
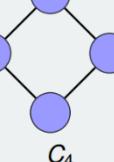
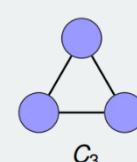
- **Estrelles:** S_i és l'estrella amb $i + 1$ vèrtexs.



- **Camins:** P_i és el camí de i vèrtexs.

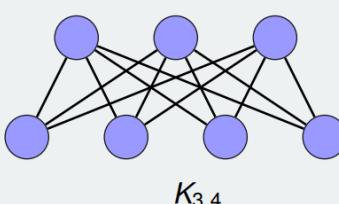
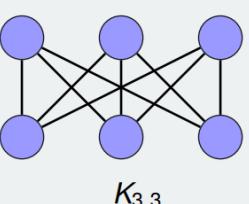
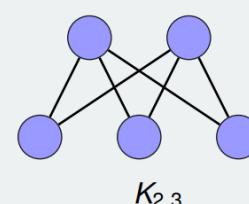


- **Cicles:** C_i és el cicle de i vèrtexs.



Tipus de grafs

- **Bipartits complets:** $K_{i,j}$ és el bipartit complet amb i vèrtexs connectats a j vèrtexs.



Propietat

Tot graf de n vèrtexs té un màxim de $\frac{n(n-1)}{2}$ arestes.

Demostració

Cada vèrtex pot tenir una aresta amb $n - 1$ vèrtexs més (però no amb ell mateix). Com que cada aresta està comptada dos cops, s'obtenen

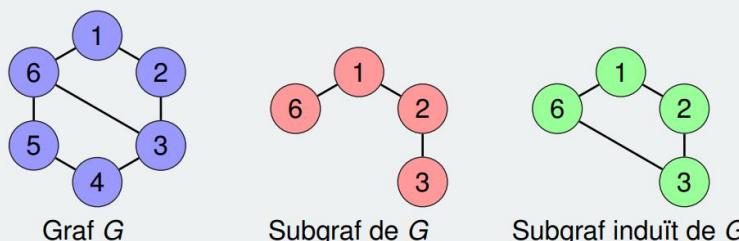
$$\frac{n(n-1)}{2} = \binom{n}{2}$$

arestes diferents.

(Són les combinacions de n elements triats de 2 en 2)

Adjacència i subgrafs

- dos vèrtexs u, v són **adjacents** si $\{u, v\}$ és una aresta
- una aresta $\{u, v\}$ es diu que és **incident** a u i v
- grau** d'un vèrtex u : nombre d'arestes incidents a u
- un graf $H = (V', E')$ és **subgraf** del graf $G = (V, E)$ si $V' \subseteq V$ i $E' \subseteq E$
- un graf H és **subgraf induït** d'un graf G si H és subgraf de G i conté totes les arestes que G té entre els vèrtexs de H



Pel **príncipi de les caselles**, G té dos vèrtexs amb el mateix grau.

Camins i cicles

Sigui $G = (V, E)$ un graf.

- Un **camí** en G és una seqüència de vèrtexs (x_0, x_1, \dots, x_n) on $x_i \in V$ per a tot $i \leq n$, $\{x_i, x_{i+1}\} \in E$ per a tot $i < n$ i tots els x_i són diferents
- Un **cicle** en G és una seqüència $(x_0, x_1, \dots, x_n, x_0)$ tal que (x_0, x_1, \dots, x_n) és un camí en G i $\{x_0, x_n\} \in E$
- Un graf és **cíclic** si conté algun cicle

Connectivitat

- Un graf és **connex** si existeix un camí entre tot parell de vèrtexs
- Un **component connex** d'un graf és un subgraf induït connex maximal (no hi ha cap vèrtex extern adjacent)

Distància

- La **distància** entre dos vèrtexs és el nombre mínim d'arestes d'un camí que els uneix
- El **diàmetre** d'un graf és la màxima distància entre qualsevol parell de vèrtexs del graf

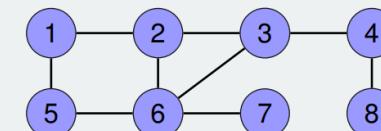


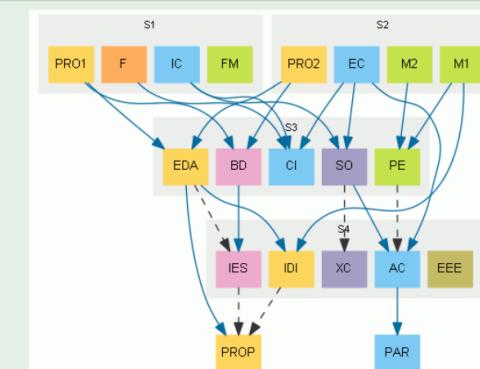
Figura: La distància entre 4 i 5 és 3. El diàmetre del graf és 4.

Grafs dirigits i etiquetats

Definició

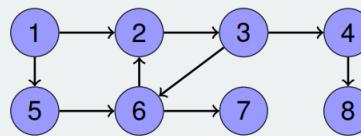
- Un **graf dirigit** o **digraf** és un parell (V, E) , on
 - V és un conjunt finit (**vèrtexs**)
 - E és un conjunt de parells ordenats de vèrtexs (**arcs**)

Graf dirigit d'assignacions obligatòries del grau



Digrafs: graus i distàncies

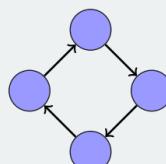
- Es distingeix entre **grau d'entrada** i **grau de sortida**
- En un **camí** (o **camí dirigit**), tots els arcs van en la mateixa direcció
- La **distància** entre dos vèrtexos es refereix als camins dirigits



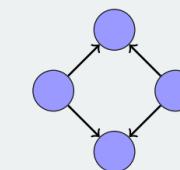
El vèrtex 2 té grau d'entrada 2 i grau de sortida 1.
La distància de 5 a 4 és 4. La de 4 a 5, ∞ .

Digrafs: connectivitat

- Un digraf és **feblement connex** (o **connex**) si el graf obtingut substituint els arcs per arestes no dirigides és connex
- Un digraf és **fortament connex** si existeix un camí dirigit entre qualsevol parell de vèrtexos
- Si un digraf és fortament connex llavors també és feblement connex
- A la inversa no és cert:



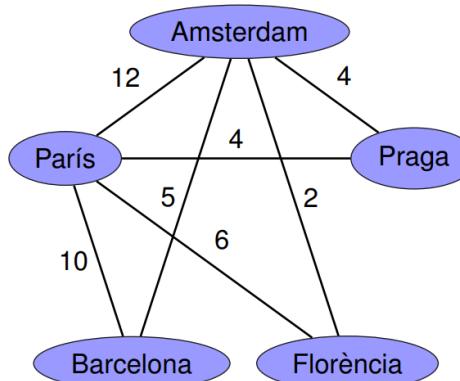
fortament connex



feblement connex

Definició

Un **graf etiquetat** (dirigit o no dirigit) és un graf en el qual les arestes tenen etiquetes associades. També se'n diu **ponderat** o **graf amb pesos**.



Es fan servir les 4 combinacions:

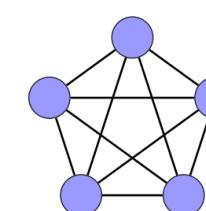
- Grafs no dirigits no etiquetats
- Grafs no dirigits etiquetats
- Grafs dirigits no etiquetats
- Grafs dirigits etiquetats

Densitat

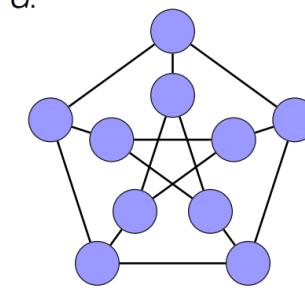
La **densitat** d'un graf G de n vèrtexs i m arestes és

$$D(G) = \frac{2m}{n(n-1)}$$

Hem vist que el nombre màxim d'arestes d'un graf G de n vèrtexs és $\frac{n(n-1)}{2}$. Per tant, $0 \leq D(G) \leq 1$ per a tot graf G .



$$D(K_5) = 1$$



$$D(GP(5,2)) = 1/3$$

Densitat

Un graf G de n vèrtexs i m arestes és **dens** si $m \approx n^2/2$ (si $D(G)$ és proper a 1). Altrament, se'n diu **espars**.

El concepte és més formal quan es consideren famílies de grafs.
Per exemple:

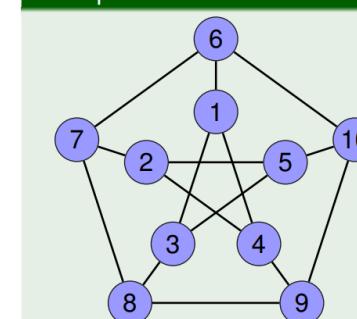
- **Grafs complet**: $\lim_{n \rightarrow \infty} D(K_n) = 1$
- **Bipartits complet** $K_{n,n}$: $\lim_{n \rightarrow \infty} D(K_{n,n}) = \frac{1}{2}$
- **Cicles**: $\lim_{n \rightarrow \infty} D(C_n) = 0$

Matriu d'adjacència / grafs no dirigits

La **matriu d'adjacència** d'un graf no dirigit $G = (V, E)$ és una matriu M de $n \times n$ valors booleanos tal que

$$M_{ij} = \begin{cases} 1, & \text{si } \{i, j\} \in E \\ 0, & \text{si } \{i, j\} \notin E \end{cases}$$

Exemple



1	0	0	1	1	0	1	0	0	0	0
2	0	0	0	1	1	0	1	0	0	0
3	1	0	0	1	0	0	0	1	0	0
4	1	1	0	0	0	0	0	0	1	0
5	0	1	1	0	0	0	0	0	0	1
6	1	0	0	0	0	0	1	0	0	1
7	0	1	0	0	0	1	0	1	0	0
8	0	0	1	0	0	0	1	0	1	0
9	0	0	0	1	0	0	0	1	0	1
10	0	0	0	0	1	1	0	0	1	0

Matriu d'adjacència: estructura de dades

Les matrius d'adjacència es poden implementar amb un vector de vectors.

```
typedef vector< vector<bool> > graph;
```

Si g és de tipus `graph` (n vèrtexs):

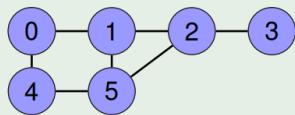
- El cost en espai és $\Theta(n^2)$
- La inicialització de g és $\Theta(n^2)$
- El vector $g[i]$ conté la informació sobre els veïns de i
- Processar els vèrtexs adjacents a i serà $\Theta(n)$
- Si el graf g no és dirigit, llavors $g[i][j] == g[j][i]$ (la informació està duplicada)
- Recórrer totes les arestes és $\Theta(n^2)$
- La matriu d'adjacència és adequada per a **grafs densos**

Llistes d'adjacència

Cada vèrtex apunta a la llista dels adjacents (grafs) o successors (digrafs).

```
typedef vector< vector<int> > graph;
```

Exemple



0	1 4
1	0 5 2
2	3 1 5
3	2
4	5 0
5	1 2 4

Si g és de tipus `graph` (n vèrtexs i m arestes):

- El cost en espai és $\Theta(n + m)$
- La inicialització és $\Theta(n)$
- Els adjacents a i estan encadenats sense un ordre especial
- Processar els vèrtexs adjacents a i serà $O(m)$
- Si g no és dirigit, la informació està duplicada
- Recórrer totes les arestes és $\Theta(n + m)$
- Les llistes d'adjacència són adequades per a **grafs esparsos**

Cost de les operacions

n : nombre de vèrtexs / m : nombre d'arestes

operacions	matriu d'adjacència	llistes d'adjacència
espai	$\Theta(n^2)$	$\Theta(n + m)$
crear	$\Theta(n^2)$	$\Theta(n)$
afegir vèrtex	$\Theta(n)$	$\Theta(1)$
afegir aresta	$\Theta(1)$	$O(n)$
esborrar aresta	$\Theta(1)$	$O(n)$
consultar vèrtex	$\Theta(1)$	$\Theta(1)$
consultar aresta	$\Theta(1)$	$O(n)$
és v aïllat?	$\Theta(n)$	$\Theta(1)$
successors*	$\Theta(n)$	$O(n)$
predecessors*	$\Theta(n)$	$\Theta(n + m)$
adjacents ⁺	$\Theta(n)$	$O(n)$

* només en grafs dirigits

+ només en grafs no dirigits

Cerca en profunditat

La **cerca** (o recorregut) **en profunditat** (en anglès: **DFS**, de *Depth-First Search*) resol la pregunta:

A quins vèrtexs es pot accedir des d'un vèrtex donat?

Un algorisme només pot comprovar les adjacències: si és possible anar d'un vèrtex a un altre. La situació és semblant a **l'exploració d'un laberint**.

Per explorar un laberint, cal tenir guix i corda:

- El **guix** evita anar en cercles (saber què hem visitat)
- La **corda** permet anar enrere i veure passadissos encara no visitats

Quins són els anàlegs informàtics del guix i la corda?

- el guix és un **vector de booleans**
- la corda és una **pila**

Cerca en profunditat recursiva (des d'un vèrtex donat)

Utilitzem la representació de llistes d'adjacència.

```
typedef vector< vector<int> > graph;
```

Recorregut dels vèrtexs accessibles a partir d'un vèrtex donat u . La pila és implícita en la recursió.

```
void DFS_rec (const graph& G, int u,
              vector<boolean>& vis, list<int>& L) {
    if (not vis[u]) {
        vis[u] = true; L.push_back(u);
        for (int v : G[u])
            DFS_rec(G, v, vis, L);
    }
}
```

Cost de la cerca des d'un vèrtex

- Es fa una quantitat constant de treball (2 primeres línies): $\Theta(1)$
- Es fan crides recursives als veïns. En total,
 - cada vèrtex del component connex es marca un cop
 - cada aresta $\{i,j\}$ es visita dos cops: des de i i des de j

Per tant, $O(n + m)$

Cost total: $O(n + m)$

Cerca en profunditat recursiva

Recorregut en profunditat de tot el graf, encara que no sigui connex.

```
list<int> DFS_rec (const graph& G) {
    int n = G.size();
    list<int> L;
    vector<boolean> vis(n, false);
    for (int u = 0; u < n; ++u)
        DFS_rec(G, u, vis, L);
    return L;
}
```

Cost de la cerca en profunditat

Si el graf G té

- k components connexos (c.c.)
- $n = \sum_{i=1}^k n_i$ vèrtexs (el c.c. i té n_i vèrtexs)
- $m = \sum_{i=1}^k m_i$ arestes (el c.c. i té m_i arestes)

llavors el cost és

$$\sum_{i=1}^k \Theta(n_i + m_i) = \Theta(\sum_{i=1}^k n_i + \sum_{i=1}^k m_i) = \Theta(n + m).$$

Cerca en profunditat iterativa

```
list<int> DFS_ite (const graph& G) {
    int n = G.size();
    list<int> L;
    stack<int> S;
    vector<bool> vis(n, false);

    for (int u = 0; u < n; ++u)
        S.push(u);
    while (not S.empty()) {
        int v = S.top(); S.pop();
        if (not vis[v]) {
            vis[v] = true; L.push_back(v);
            for (int w : G[v])
                S.push(w);
        }
    }
    return L;
}
```

Cerca en amplada

La **cerca** (o recorregut) **en amplada** (en anglès: **BFS**, de *Breadth-First Search*) **avança localment** des d'un vèrtex inicial s visitant els vèrtexs a distància $k + 1$ de s després d'haver visitat els vèrtexs a distància k de s .

- L'algorisme de cerca en amplada, a partir d'un vèrtex s , calcula
 - un **recorregut** en amplada a partir de s
 - les **distàncies** mínimes de s a tots els vèrtexs
 - els **camins** mínims de s fins tots els vèrtexs
- La cerca en amplada funciona amb grafs
 - **dirigits i no dirigits**
 - **sense pesos** (etiquetes numèriques) en les arestes
- És dels algorismes de grafs més senzills i model d'altres:
 - **algorisme de Dijkstra** per trobar camins més curts en grafs amb pesos
 - **algorisme de Prim** per trobar l'arbre d'expansió mínim

Cerca en amplada

Entrada: graf $G = (V, E)$ dirigit o no dirigit i vèrtex $s \in V$

S sortida: per a tots els vèrtexos u accessibles des de s ,

$\text{dist}(u)$ contindrà la distància des de s fins a u

el camí més curt (revessat) de s a u és $(u, \text{prev}(u), \text{prev}(\text{prev}(u)), \dots, s)$

BFS(G, s)

per a tot $u \in V$

$\text{dist}(u) = \infty$

$\text{prev}(u) = \text{indefinit}$

$\text{dist}(s) = 0$

$Q = [s]$ (cua que només conté s)

mentre Q no sigui buida

$u = \text{desencuar}(Q)$

per a tota aresta $(u, v) \in E$

si $\text{dist}(v) = \infty$ llavors

$\text{encuar}(Q, v)$

$\text{dist}(v) = \text{dist}(u) + 1$

$\text{prev}(v) = u$

Per a cada $d = 0, 1, 2, \dots$ hi ha un moment en el qual:

- la cua conté els nodes a distància d
- els vèrtexos a distància $\leq d$ de s tenen la distància correcta
- els vèrtexos a distància $> d$ de s tenen la distància ∞

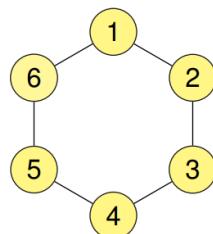
Cost amb un graf de n vèrtexos i m arestes:

- Cada vèrtex es posa un cop a la cua: $\Theta(n)$ operacions de cua
- Cada aresta es visita un cop (dirigits) o dos (no dirigits): $\Theta(m)$

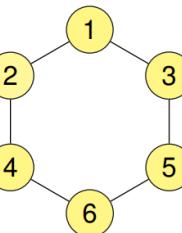
Cost total: $\Theta(n + m)$ (amb llistes d'adjacència).

Ara podem comparar DFS i BFS:

- En DFS, la cerca torna només quan ja no queden vèrtexos per visitar (es pot arribar a fer una gran volta per visitar un veí)
- En BFS, es visiten els vèrtexos per ordre de distància creixent



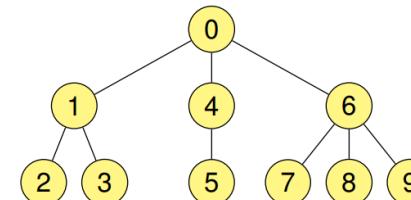
DFS



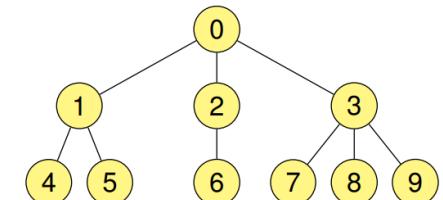
BFS

En arbres,

- DFS correspon al recorregut preordre i és la base del *backtracking*
- BFS recorre els vèrtexos per nivells



DFS



BFS

Els codis de DFS i BFS són molt semblants.

- La **diferència** fonamental és l'ús
 - d'una **pila** en DFS
 - d'una **cua** en BFS
- Com en DFS, en BFS només **s'explora el component connex del vèrtex inicial**. Per explorar la resta del graf, podem recomençar la cerca des dels altres vèrtexs amb l'ajut d'un bucle.

BFS

Fem servir llistes d'adjacència. Càlcul del recorregut, no de les distàncies.

```
list<int> BFS (const graph& G) {
    int n = G.size();
    list<int> L;
    queue<int> Q;
    vector<bool> enc(n, false);
    for (int u = 0; u < n; ++u) {
        if (not enc[u]) {
            Q.push(u); enc[u] = true;
            while (not Q.empty()) {
                int v = Q.front(); Q.pop();
                L.push_back(v);
                for (w : G[v])
                    if (not enc[w])
                        Q.push(w); enc[w] = true;
            }
        }
    }
    return L;
}
```

Cost de la cerca en amplada (com en DFS)

Si el graf G té

- k components connexos (c.c.)
- $n = \sum_{i=1}^k n_i$ vèrtexs (el c.c. i té n_i vèrtexs)
- $m = \sum_{i=1}^k m_i$ arestes (el c.c. i té m_i arestes)

llavors el cost és

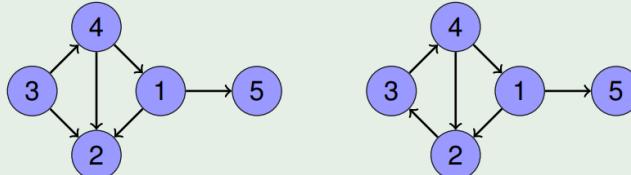
$$\sum_{i=1}^k \Theta(n_i + m_i) = \Theta(\sum_{i=1}^k n_i + \sum_{i=1}^k m_i) = \Theta(n + m).$$

Ordenació topològica

Definició

Un dag és un **graf dirigit acíclic**.

Exemple



El digraf de l'esquerra és un dag; el de la dreta, no.

Definició

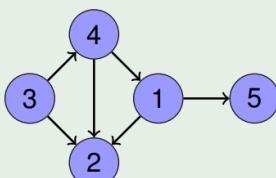
Una **ordenació topològica** d'un dag $G = (V, E)$ és una seqüència

$$v_1, v_2, v_3, \dots, v_n$$

tal que $V = \{v_1, \dots, v_n\}$ i si $(v_i, v_j) \in E$, llavors $i < j$.

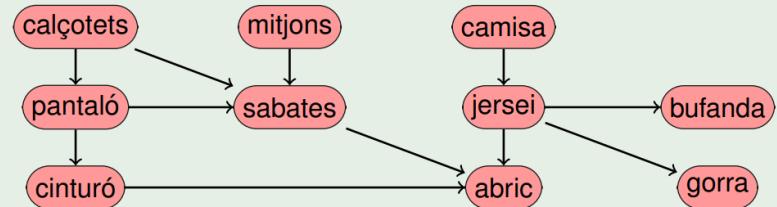
Exemple

Un dag pot tenir més d'una ordenació topològica.



Tant 3,4,1,2,5 com 3,4,1,5,2 en són ordenacions topològiques.

Exemple: roba masculina d'hivern



Possibles ordenacions:

- calcotets, mitjons, pantaló, camisa, cinturó, jersey, sabates, abric, bufanda, gorra
- mitjons, camisa, calcotets, jersey, pantaló, cinturó, bufanda, gorra, sabates, abric

Problema de l'ordenació topològica

Donat un dag, trobar una ordenació topològica.

Algorisme (esquema voràc)

Repetir fins que no quedin vèrtexs:

- ➊ Trobar un vèrtex u amb grau d'entrada 0
- ➋ Escriure u i esborrar-lo del graf

Cost

Si el graf té n vèrtexs,

- cercar un vèrtex amb grau d'entrada 0 costa $\Theta(n)$
- el pas anterior es repeteix n vegades

Cost total: $\Theta(n^2)$.

El cost es pot millorar si fem més eficient la cerca d'un vèrtex de grau 0. Necessitarem:

- Un vector per emmagatzemar el grau d'entrada de cada vèrtex
- Una estructura (pila, cua, etc.) que contingui els vèrtexs de grau 0

Algorisme

Inicialitzem una pila amb els vèrtexs de grau 0. Mentre no sigui buida:

- ➊ Desempilem un vèrtex, l'escrivim i reajustem els graus
- ➋ Empilem els nous vèrtexs de grau 0

Ordenació topològica

Preparació del vector i de la pila d'un graf de n vèrtexs i m arestes.

Representació amb llistes d'adjacència. Cost $\Theta(n + m)$.

```
list<int> ordenacio_topologica (graph& G) {
    int n = G.size();
    vector<int> ge(n, 0);
    for (int u = 0; u < n; ++u)
        for (int v : G[u])
            ++ge[v];

    stack<int> S;
    for (int u = 0; u < n; ++u)
        if (ge[u] == 0)
            S.push(u);
}
```

Bucle principal.

```
list<int> L;
while (not S.empty()) {
    int u = S.top(); S.pop();
    L.push_back(u);
    for (int v : G[u])
        if (--ge[v] == 0)
            S.push(v);
}
return L;
```

Es visita

- un cop cada vèrtex
- un cop cada aresta

Per tant, el cost és $\Theta(n + m)$.

Algorisme de Dijkstra

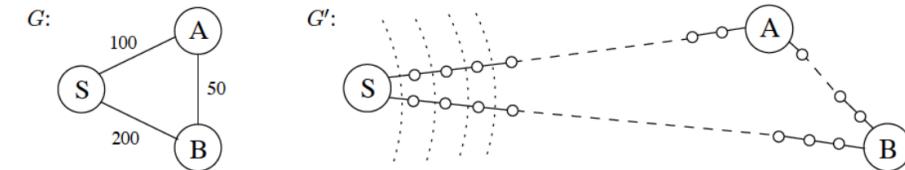
Truc per utilitzar BFS en grafs amb distàncies

Trencar les arestes del graf d'entrada en arestes de "mida unitària" introduint vèrtexs de farciment.



Cada aresta $e = (u, v)$ amb etiqueta $d(e)$ se substitueix per $d(e)$ arestes amb etiqueta 1 afegint-hi $d(e) - 1$ vèrtexs nous entre u i v .

El truc anterior resultaria massa ineficient amb distàncies grans. Però podem imaginar que posem **alarms** que avisen de quan arribem d'un vèrtex (dels originals) a un altre.



Per exemple,

- Posem una alarma per a A i $T = 100$ i una altra per a B i $T = 200$
- Quan arribem a A , el temps de B s'ajusta a $T = 150$

Algorisme "de les alarmes"

L'algorisme següent simula BFS sobre G' sense arribar a crear vèrtexs addicionals.

Donat el graf G amb distàncies d i un vèrtex inicial S :

- ① Posar l'alarma del vèrtex S a 0.
- ② Repetir fins que no quedin alarmes:

- Trobar l'alarma següent: vèrtex u i temps T
- Per a cada veí v de u en G
 - si v no té cap alarma o és $> T + d(u, v)$, llavors reajustar l'alarma per a $T' = T + d(u, v)$

L'algorisme anterior és un exemple d'**algorisme voràç** (*greedy*).

Esquema voràç

Un **algorisme voràç** resol un problema per etapes fent, a cada etapa, allò que sembla millor. Habitualment, consisteixen en una estratègia simple que es va repetint.

Algorisme de Dijkstra

L'algorisme de les alarmes calcula distàncies en qualsevol graf que tingui pesos positius i enters en les arestes.

Per implementar el sistema d'alarmes, triem una **cua amb prioritat** amb les operacions següents:

- **crear-cua**: construir cua amb prioritat amb els elements i claus disponibles
- **afegir**: inserir un nou element a la cua
- **treure-min**: retornar l'element amb clau més baixa i treure'l de la cua
- **decrementar-clau**: actualitzar el decrement en la clau d'un element
- **buida**: retorna un booleà que indica si la cua és buida

Algorisme de Dijkstra dels camins mínims

Entrada: Graf $G = (V, E)$, dirigit o no dirigit; distàncies positives en les arestes $\{d(u, v) \mid u, v \in V\}$; vèrtex inicial s ;

Sortida: Per a tot vèrtex u accessible des de s , $dist(u)$ = distància de s a u .

Dijkstra (G, d, s)

```

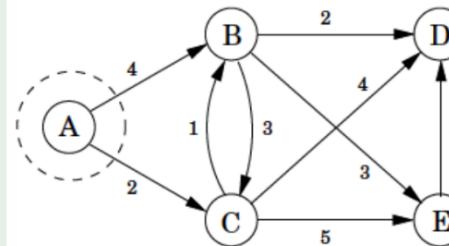
per a tot vèrtex  $u \in V$ 
     $dist(u) = \infty$ 
     $prev(u) = nil$ 
     $dist(s) = 0$ 
 $H = \text{crear-cua}(V)$  (fent servir  $dist$  per les claus)
mentre no buida( $H$ )
     $u = \text{esborrar-min}(H)$ 
    per a tota aresta  $(u, v) \in E$ 
        si  $dist(v) > dist(u) + d(u, v)$ 
             $dist(v) = dist(u) + d(u, v)$ 
             $prev(v) = u$ 
            decrementar-clau( $H, v$ )

```

Cost de l'algorisme de Dijkstra

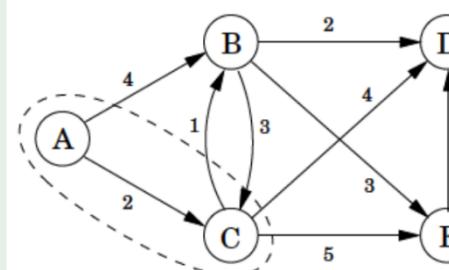
L'estrucció és la mateixa que la de BFS, però cal tenir en compte els costos de les operacions de la cua amb prioritat. Amb un heap, el cost per a grafs de n vèrtexs i m arestes és: $\Theta((n + m) \log n)$.

Exemple (Algorithms, Dasgupta et al.)



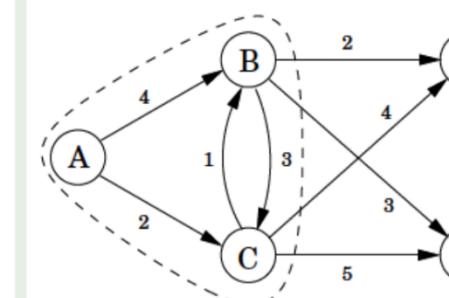
A: 0	D: ∞
B: 4	E: ∞
C: 2	

Exemple (Algorithms, Dasgupta et al.)



A: 0	D: 6
B: 3	E: 7
C: 2	

Exemple (Algorithms, Dasgupta et al.)



A: 0	D: 5
B: 3	E: 6
C: 2	

Quina cua amb prioritat és millor?

Implementació	esborrar-min	afegir/ decrementar-clau	$n \times$ esborrar-min + $(n + m) \times$ afegir
vector	$O(n)$	$O(1)$	$O(n^2)$
heap binari	$O(\log n)$	$O(\log n)$	$O((n + m) \log n)$
heap d -ari*	$O(\frac{d \log n}{\log d})$	$O(\frac{\log n}{\log d})$	$O((nd + m) \frac{\log n}{\log d})$
heap Fibonacci	$O(\log n)$	$O(1)^+$	$O(n \log n + m)$

* Tria òptima per a d : $d = m/n$.

+ Cost amortitzat ($O(1)$ en mitjana al llarg de tot l'algorisme).

Algorisme de Dijkstra

En lloc de decrementar les claus, s'acumulen en la cua amb prioritat però un vector \mathbf{S} evitara que un vèrtex es tracti dos cops.

```

typedef pair<double, int> ArcP;           // arc amb pes
typedef vector<vector<ArcP>> Grafp;    // graf amb pesos

void dijkstra(const Grafp& G, int s, vector<double>& d,
              vector<int>& p) {
    int n = G.size();
    d = vector<double>(n, infinit); d[s] = 0;
    p = vector<int>(n, -1);
    vector<bool> S(n, false);
    priority_queue<ArcP, vector<ArcP>, greater<ArcP>> Q;
    Q.push(ArcP(0, s));

    while (not Q.empty()) {
        int u = Q.top().second; Q.pop();
        if (not S[u]) {
            S[u] = true;
            for (ArcP a : G[u]) {
                int v = a.second;
                double c = a.first;
                if (d[v] > d[u] + c) {
                    d[v] = d[u] + c;
                    p[v] = u;
                    Q.push(ArcP(d[v], v));
                }
            }
        }
    }
}

```

Arbres d'expansió mínims

Sigui $G = (V, E)$ un graf amb pesos, connex i no dirigit, i $\omega : E \rightarrow \mathbb{R}$.

Definition

Un **arbre d'expansió** (*spanning tree*, en anglès) de G és un subgraf $T = (V, A)$ de G , amb $A \subseteq E$, que és connex i acíclic.

Observem que T és un arbre i conté tots els vèrtexs de G

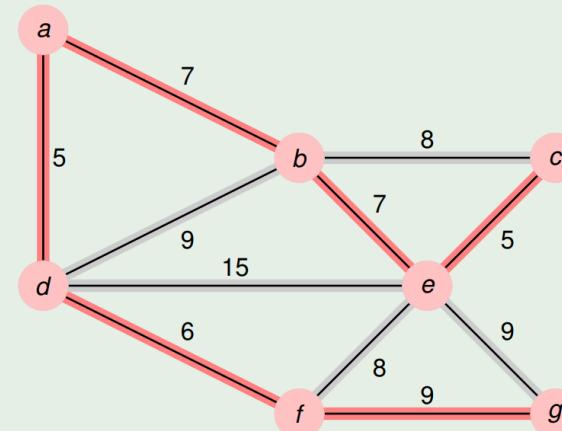
Definició

Un **arbre d'expansió mínim** (MST, de l'anglès *minimum spanning tree*) de G és un arbre d'expansió $T = (V, A)$ de G el pes total del qual

$$\omega(T) = \sum_{e \in A} \omega(e)$$

és mínim entre tots els arbres d'expansió de G .

Dos arbres d'expansió mínims



Hi ha molts algoritmes diferents per calcular MST. Tots segueixen un **esquema voràç**, és a dir, repeteixen una acció que sembla òptima a cada moment.

```

A = ∅;
Cand = E;
while (|A| ≠ |V| - 1) {
    triar e ∈ Cand tal que  $T = (V, A \cup \{e\})$  no tingui cicles
    A = A ∪ {e}
    Cand = Cand - {e}
}

```

Definition

Un subconjunt d'arestes $A \subseteq E$ és **prometedor** si A és un subconjunt de les arestes d'un MST de G .

Definition

Un **tall** d'un graf $G = (V, E)$ és una partició de V , és a dir, un parell (C, C') tal que

- $C \cup C' = V$
- $C \cap C' = \emptyset$

Definition

Una aresta e **respecta** un tall (C, C') si ambdós extrems de e pertanyen a C o ambdós pertanyen a C' ; altresment, diem que e **travessa** el tall.

Un conjunt d'arestes A **respecta** un tall si totes les arestes de A el respecten.

Arbres d'expansió mínims

Teorema

Sigui A un conjunt prometedor d'arestes que respecta el tall (C, C') de G .
 Sigui e una aresta amb pes mínim entre les que travessen el tall (C, C') .
 Aleshores, $A \cup \{e\}$ és prometedor.

Aquest teorema dona un mètode per dissenyar algorismes per a un MST:

- ➊ començar amb un conjunt d'arestes buit A
- ➋ definir quin és el tall inicial de G
- ➌ mentre el tall no sigui trivial
 - triar una aresta e de pes mínim entre les que travessen el tall
 - afegir e a A i passar a C l'extrem de e que pertanyia a C'
(com que e travessava el tall, no es pot crear un cicle)

Demostració

Sigui A' el conjunt d'arestes d'un MST T tal que $A \subseteq A'$.

(T existeix perquè assumim que A és prometedor)

Considerem dos casos:

- ➊ $e \in A'$
Llavors, $A \cup \{e\}$ és prometedor.
- ➋ $e \notin A'$

Com que A respecta el tall, hi ha una aresta $e' \in A' - A$ que travessa el tall (altrament, T no seria connex).

El subgraf $T' = (V, A' - \{e'\} \cup \{e\})$ és un arbre d'expansió i, per tant,

$$\omega(T) \leq \omega(T') = \omega(T) - \omega(e') + \omega(e).$$

Llavors, $\omega(e') \leq \omega(e)$.

Però, com que e tenia pes mínim, es compleix que $\omega(e) \leq \omega(e')$.

Per tant, $\omega(T) = \omega(T')$ i T' és un MST.

Aleshores, $A \cup \{e\}$ és prometedor.

Algorisme de Prim

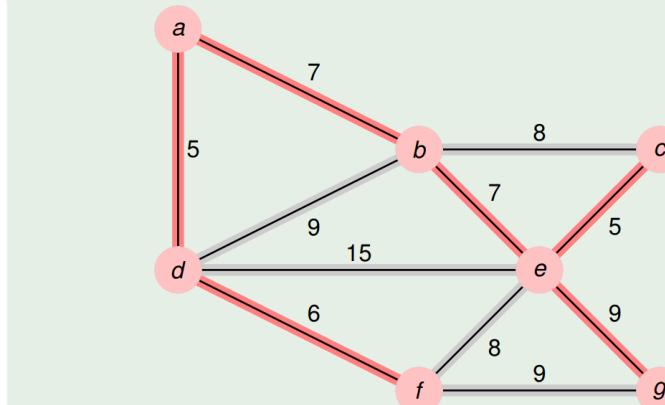
En l'**algorisme de Prim** (coneugut també com **algorisme Prim-Jarník**), mantenim un subconjunt de vèrtexs visitats.

- El conjunt de vèrtexs es divideix entre visitats i no visitats
- Cada iteració de l'algorisme tria l'aresta de pes mínim entre les que uneixen vèrtexs visitats amb no visitats
- Pel teorema, l'algorisme és correcte

Algorisme de Prim

El graf es representa amb llistes d'adjacència. Els parells són (cost, vertex).

```
typedef pair<int,int> P;
int mst(const vector<vector<P>>& g) { // Ret. el cost d'un MST
    vector<bool> vis(n, false);
    vis[0] = true;
    priority_queue<P, vector<P>, greater<P> > pq;
    for (P x : g[0]) pq.push(x);
    int sz = 1;
    int sum = 0;
    while (sz < n) {
        int c = pq.top().first;
        int x = pq.top().second;
        pq.pop();
        if (not vis[x]) {
            vis[x] = true;
            for (P y : g[x]) pq.push(y);
            sum += c;
            ++sz; } }
    return sum; }
```



Anàlisi del cost

El cost del bucle, amb $O(m)$ iteracions, domina el cost de l'algorisme.

Comptem per separat la selecció d'aresta i la visita de nous candidats dins de cada iteració:

- ➊ **Selecció d'aresta.** La selecció d'aresta en cada iteració és $O(\log m)$.
- ➋ **Visita de nous candidats.** Cada vèrtex es marca exactament un cop.
Quan es visita un vèrtex x , afegir nous candidats costa $O(\deg(x) \log m)$.

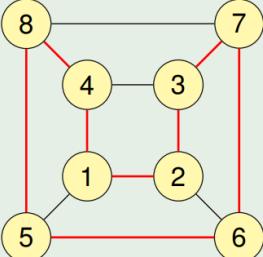
El cost total és, doncs,

$$O(m \log m) + \sum_{x \in V} O(\deg(x) \log m) = O(m \log m) = O(m \log n).$$

De fet, es pot veure que el cost en cas pitjor és $\Theta(m \log n)$.

Algorismes de força bruta

Exemple: Cicle hamiltonià



Les possibilitats són totes les permutacions de vèrtexs:

$(12345678), (12345687), \dots, (\textcolor{red}{12376584}), \dots$

Suposem que volem processar totes les cadenes de zeros i uns de mida n .

Disposem d'un procediment PROCESSAR(C) que fa el tractament desitjat del vector C .

BINARI(n)

(processar totes les cadenes binàries de mida n)

```

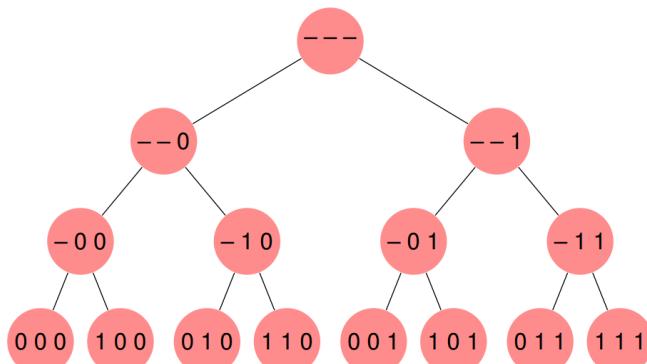
si  $n = 0$  llavors
    PROCESSAR( $C$ )
si no
     $C[n] \leftarrow 0$ ; BINARI( $n - 1$ )
     $C[n] \leftarrow 1$ ; BINARI( $n - 1$ )

```

El cost ve donat per $T(n) = 2T(n - 1) + \Theta(1)$. Pel teorema mestre I:

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

Per a $n = 3$, s'obté l'arbre de recursió:



Podem considerar un algorisme genèric iteratiu de cerca exhaustiva que fa servir els procediments:

- PRIMER(x): genera un primer candidat
- SEGÜENT(x, c): genera el candidat següent a c
- VÀLID(x, c): comprova si el candidat c és solució
- NUL(c): diu si c és un candidat "nul"

Algorisme genèric de força bruta

```

FORÇA BRUTA( $x$ )
 $c \leftarrow \text{PRIMER}(x)$ 
mentre no ( $\text{NUL}(c)$ )
    si  $\text{VALID}(x, c)$  llavors
        PROCESSAR( $c$ )
     $c \leftarrow \text{SEGÜENT}(x, c)$ 

```

Quin cost té la cerca exhaustiva?

Per exemple, les cerques en profunditat (DFS) i amplada (BFS) en grafs també són cerques exhaustives:

- si el graf ve donat a l'entrada, són cerques polinòmiques
- si hi ha un arbre o graf implícit, normalment són exponencials

Exemple: nombres primers

```

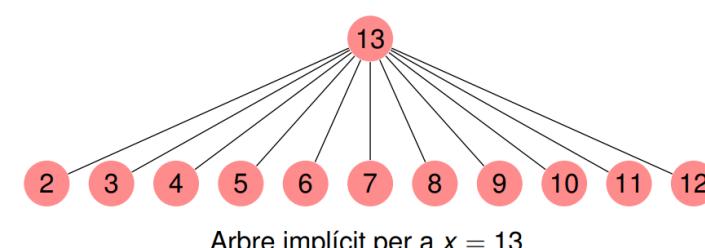
bool es_primer (int x) {
    if (x <= 1) return false;
    for (int i = 2; i < x; ++i)
        if (x % i == 0) return false;
    return true;
}

```

Nombre màxim d'iteracions: $(x - 1) - 2 + 1 = x - 2$.

Cost en funció de x : $\Theta(x)$.

Cost en funció de $n = |x|$: $\Theta(2^n)$.



Cerca amb retrocés

Un algorisme de **cerca amb retrocés** (o **tornada enrere**) funciona com una cerca exhaustiva, però s'atura quan troba una solució parcial que no es pot estendre a una solució.

L'esquema de cerca amb retrocés:

- es pot veure com una implementació intel·ligent de la cerca exhaustiva amb un cost millorat, però sovint encara exponencial
- en anglès, s'anomena **backtracking**

Exemple: moblar un pis

- **Estratègia de força bruta:** provar totes les configuracions dels mobles en tots els espais
- **L'estratègia de cerca amb retrocés** aprofita que:
 - cada moble acostuma a anar a un espai concret (*no posarem el sofà a la cuina*)
 - hi ha mobles que van junts (*cadires i taula, llit i tauletes*)
 - si una subdistribució no és satisfactòria, no considerarem la distribució que la conté
(si no ens agrada posar un moble davant d'una finestra, ja no explorarem a partir d'aquí)

Eliminar un gran grup de possibilitats en un pas es coneix com a **poda**.

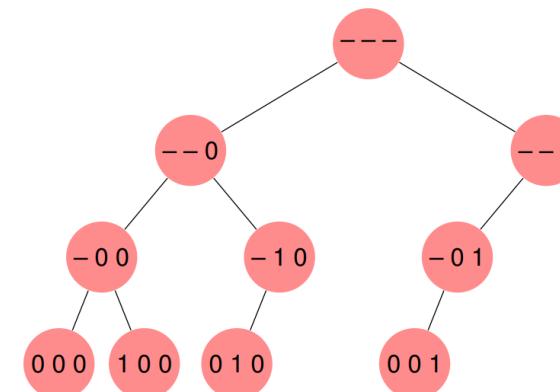
Suposem que volem totes les **cadenes de zeros i uns de mida n amb un màxim de k uns**. Podem modificar l'algorisme vist abans de manera que eviti la recursió dels subarbres que contenen més de k uns.

Crida inicial: $\text{BINARI}(n, 0)$.

$\text{BINARI}(n, i)$
(processar totes les cadenes binàries de mida n amb $\leq k - i$ uns)

```
si  $n = 0$  llavors
    PROCESSAR( $C$ )
si no
     $C[n] \leftarrow 0$ ;  $\text{BINARI}(n - 1, i)$ 
    si  $i < k$  llavors
         $C[n] \leftarrow 1$ ;  $\text{BINARI}(n - 1, i + 1)$ 
```

Per a $n = 3$ i $k = 1$, s'obté l'arbre de recursió:



Algorisme genèric

Es pot definir un algorisme genèric de tornada enrere:

- L'espai de solucions d'un problema s'acostuma a organitzar en forma d'**arbre de configuracions**
- Cada node o configuració de l'arbre es representa amb un vector

$$A = (a_1, a_2, \dots, a_k)$$

que conté les tries ja fetes

- El vector A s'amplia en la fase **avançar** triant un a_{k+1} d'un **conjunt de candidats S_{k+1}** (explorar en profunditat)
- A es redueix en la fase **retrocedir** (backtrack)

```
TORNADA ENRERE( $x$ )
    calcular  $S_1$  // conjunt de candidats per a  $a_1$ 
     $k \leftarrow 1$ 
    mentre  $k > 0$ 
        mentre  $S_k \neq \emptyset$ 
             $a_k \leftarrow$  un element de  $S_k$ 
             $S_k \leftarrow S_k - \{a_k\}$ 
             $A \leftarrow (a_1, a_2, \dots, a_k)$ 
            si SOLUCIÓ( $A$ ) llavors
                PROCESSAR( $A$ )
             $k \leftarrow k + 1$  // avançar
            calcular  $S_k$  // candidats per a  $a_k$ 
             $k \leftarrow k - 1$  // retrocedir
```

- Cost en **temps**: mida de l'arbre (normalment **exponencial**)
- Cost en **espai**: profunditat de l'arbre (normalment **polinòmic**)

Algorisme genèric

Exemple: permutacions de n elements

Quines són les permutacions dels naturals $\{1, \dots, n\}$?

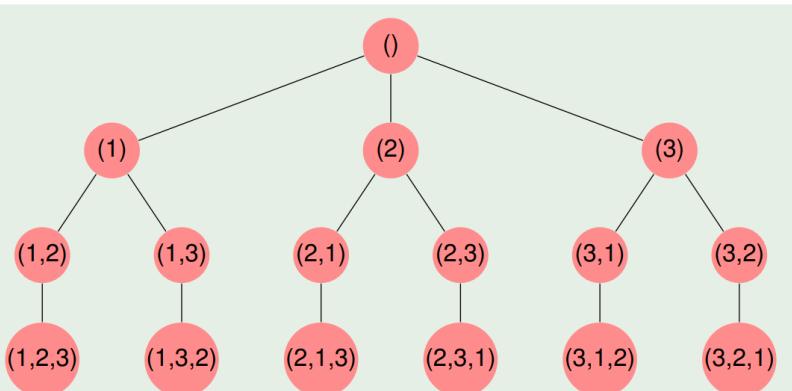
- Hi ha n possibilitats per al primer
- Fixat el primer, hi ha $n - 1$ possibilitats per al segon
- Repetint el raonament, obtenim

$$\prod_{k=1}^n k = n!$$

Adaptem l'algorisme genèric amb

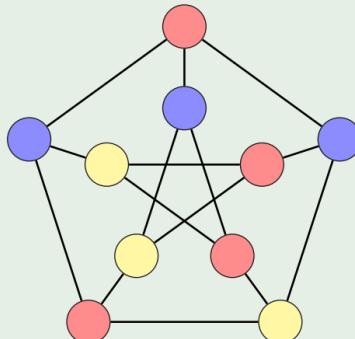
- $A = (a_1, \dots, a_k)$, tal que a_i és l' i -èsim element triat
- $S_1 = \{1, \dots, n\}$ i $S_{k+1} = \{1, \dots, n\} - A$ per a $k \geq 1$

Amb $n = 3$, s'obté l'arbre de configuracions:



Exemple: 3-Colorabilitat

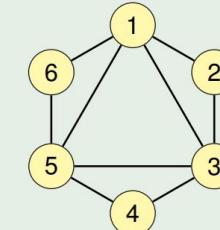
El problema de la **3-colorabilitat** consisteix a decidir si es pot assignar un color a cada vèrtex d'un total de 3 de manera que els vèrtexs adjacents tinguin colors diferents.



Una 3-coloració del graf de Petersen

3-colorabilitat

Donat el graf



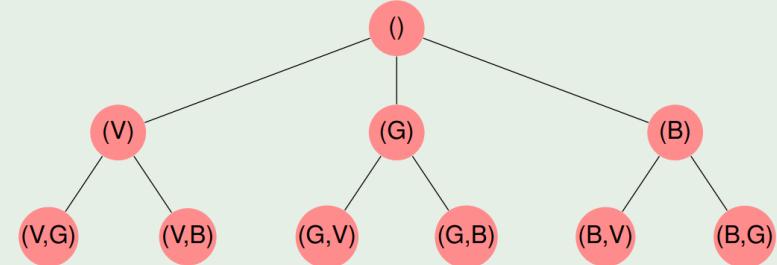
- Les **configuracions** seran assignacions parcials de colors, és a dir,

$$A = (a_1, a_2, \dots, a_k)$$

representarà el fet que el vèrtex i s'acoloreix amb el color $a_i \in \{B, G, V\}$

- El conjunt de **candidats** S_{k+1} per a a_{k+1} contindrà els colors compatibles amb els veïns que ja han estat acolorits

Els 3 primers nivells de l'arbre de configuracions serien:



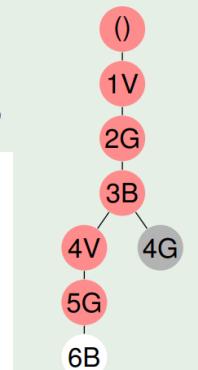
Però si el que volem és trobar només una solució,

- es pot fixar un color per al vèrtex 1
- es pot fixar un color diferent per al vèrtex 2
- qualsevol altra solució serà simètrica

Fent la tria

- $S_1 = \{V\}$
- $S_2 = \{G\}$

i definint $S_{k+1} = \{c \in \{V, G, B\} \mid \forall i \leq k \ (\{i, k+1\} \in A(G) \Rightarrow c \neq a_i\}\}$, s'obté l'arbre de configuracions



La reconstrucció Turnpike

Suposem que hi ha n punts sobre l'eix x amb coordenades

$$x_1, x_2, \dots, x_n.$$

A més, suposem que tots els valors són naturals i

$$0 = x_1 \leq x_2 \leq \dots \leq x_n$$

Els n punts determinen $n(n - 1)/2$ distàncies de la forma $|x_i - x_j|$ per a $i \neq j$ (no necessàriament diferents).

Observació

És fàcil construir el conjunt de distàncies a partir del conjunt de punts en temps $\Theta(n^2)$ (ordenades, en temps $\Theta(n^2 \log n)$).

Problema de la reconstrucció Turnpike

Donat el conjunt de distàncies, reconstruir el conjunt de punts.

Exemple amb $D = \{1, 2, 2, 2, 3, 3, 3, 4, 5, 5, 5, 6, 7, 8, 10\}$

Com que $|D| = 15 = n(n - 1)/2$, obtenim $n = 6$.

- Comencem fent $x_1 = 0$.
- Clarament, $x_6 = 10$.
- Eliminem 10 de D . Ara,

$$D = \{1, 2, 2, 2, 3, 3, 3, 4, 5, 5, 5, 6, 7, 8\}.$$

$D = \{1, 2, 2, 2, 3, 3, 3, 4, 5, 5, 5, 6, 7, 8\}$, eix $x: 0 _ _ _ 10$

- La distància més gran que queda és 8. Per tant,

$$x_2 = 2 \text{ o } x_5 = 8.$$

Si tenen solució, seran simètriques. Triem $x_5 = 8$.

- Eliminem de D les distàncies $x_6 - x_5 = 2$ i $x_5 - x_1 = 8$. Ara,

$$D = \{1, 2, 2, 3, 3, 3, 4, 5, 5, 5, 6, 7\}.$$

$D = \{1, 2, 2, 3, 3, 3, 4, 5, 5, 5, 6, 7\}$, eix $x: 0 _ _ 8 10$

- Com que 7 és el valor més alt a D , $x_4 = 7$ o $x_2 = 3$.

- Si $x_4 = 7$, les distàncies

$$x_6 - 7 = 3 \text{ i } x_5 - 7 = 1$$

han de ser a D , i hi són.

- Si $x_2 = 3$, les distàncies

$$3 - x_1 = 3 \text{ i } x_5 - 3 = 5$$

han de ser a D , i també hi són.

No tenim cap guia per triar. Provarem una opció ($x_4 = 7$) i veurem si porta a una solució. **Si no, tornarem enrere.**

- Eliminem les distàncies $x_4 - x_1 = 7$, $x_5 - x_4 = 1$ i $x_6 - x_4 = 3$. Ara,

$$D = \{2, 2, 3, 3, 4, 5, 5, 5, 6\}.$$

$D = \{2, 2, 3, 3, 4, 5, 5, 5, 6\}$, eix $x: 0 _ _ 7 8 10$

- La distància més gran és 6. Per tant, $x_3 = 6$ o $x_2 = 4$.

- Si $x_3 = 6$, llavors $x_4 - x_3 = 1$, que no pertany a D .
- Si $x_2 = 4$, llavors

$$x_2 - x_1 = 4 \text{ i } x_5 - x_2 = 4$$

i això és impossible perquè 4 només apareix un cop a D .

Aquesta línia de raonament no porta a una solució. **Tornem enrere i triem $x_2 = 3$.**

- En el conjunt de distàncies anteriors

$$D = \{1, 2, 2, 3, 3, 3, 4, 5, 5, 5, 6, 7\},$$

eliminem $x_2 - x_1 = 3$, $x_5 - x_2 = 5$ i $x_6 - x_2 = 7$. Ara,

$$D = \{1, 2, 2, 3, 3, 4, 5, 5, 6\}.$$

$D = \{1, 2, 2, 3, 3, 4, 5, 5, 6\}$, eix $x: 0 3 _ _ 8 10$

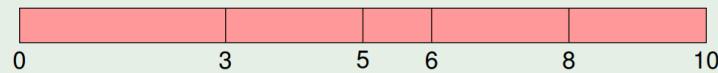
- Com que 6 és el valor més alt a D , $x_4 = 6$ o $x_3 = 4$.

- Si $x_3 = 4$, tant $x_3 - x_1$ com $x_5 - x_3$ valdrien 4, però no és possible perquè D només conté un 4.

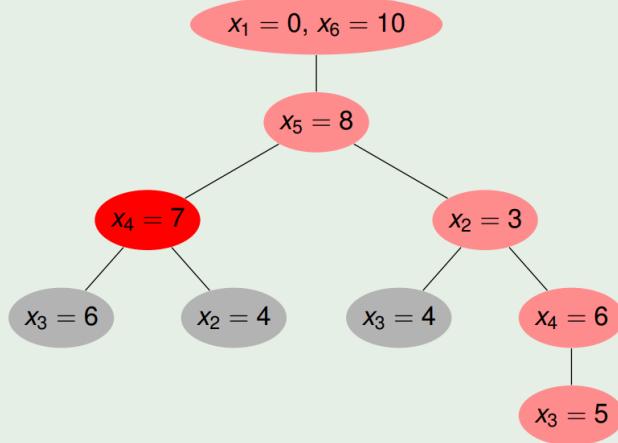
Per tant, $x_4 = 6$ i obtenim

$$D = \{1, 2, 3, 5, 5\}.$$

- Només queda triar $x_3 = 5$. Com que ens queda $D = \emptyset$, tenim una solució:



Arbre de decisió



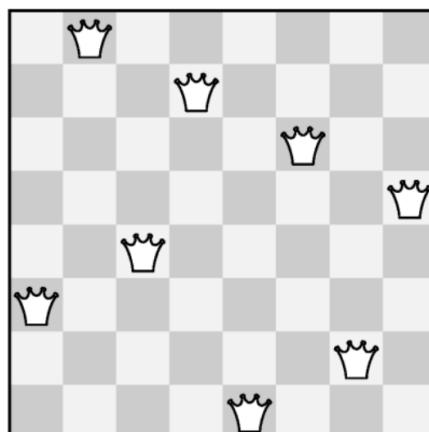
Els nodes grisos indiquen que els punts triats són inconsistents amb les dades. El node vermell no té nodes vàlids com a fills.

- Aquest mètode dona lloc a un algorisme que, si no es produeix cap tornada enrere, té cost $O(n^2 \log n)$
- Per a punts aleatoris distribuïts de manera uniforme, es produeix com a molt una tornada enrere en tot l'algorisme

Les n reines

Problema de les vuit reines

Col·locar vuit reines en un tauler d'escacs sense que cap amenaci cap altra.



Estratègies de resolució per **força bruta**:

- 1 Tria 8 **posicions diferents** del taular:

$${64 \choose 8} = 4.426.165.368 \text{ configuracions}$$

- 2 Tria 8 posicions en **files diferents**:

$$8^8 = 16.777.216 \text{ configuracions}$$

- 3 Tria 8 posicions en **files i columnes diferents**:

$$8! = 40.320 \text{ configuracions}$$

Amb **backtracking** encara es pot millorar més.

Nombre de solucions no isomorfes (per rotació o reflexió) de les n reines per a $n \in \{1, \dots, 10\}$:

n	solucions
1	1
2	0
3	0
4	1
5	2
6	1
7	6
8	12
9	46
10	92

Primera implementació:

- amb tornada enrere
- amplia la solució parcial sempre que sigui “legal” (que es pugui estendre a una solució completa)
- cost en cas pitjor: $\Theta(n^n)$

Implementarem la posició de les reines amb un vector

```
vector<int> T;
```

que indicarà que la reina de la fila i és a la columna $T[i]$.

Per saber si les reines de les files i i k comparteixen

- **columna**, comprovem si $T[i] = T[k]$
- **diagonal principal** (\searrow), comprovem si $T[i] - i = T[k] - k$
- **diagonal secundària** (\nearrow), comprovem si $T[i] + i = T[k] + k$

```

class NReines {

    int n; // nombre de reines
    vector<int> T; // configuracio actual

    void recursiu(int i) {
        if (i == n) {
            escriure();
        } else {
            for (int j = 0; j < n; ++j) {
                T[i] = j;
                if (legal(i)) {
                    recursiu(i+1);
                }
            }
        }
    }

    bool legal(int i) {
        // Indica si la config. amb les reines 0..i es legal
        // sabent que la config. amb les reines 0..i-1 ho es

        for (int k = 0; k < i; ++k) {
            if (T[k] == T[i] or
                T[i]-i == T[k]-k or T[i]+i == T[k]+k) {
                return false;
            }
        }
        return true;
    }

    void escriure() {
        for (int i = 0; i < n; ++i) {
            for (int j = 0; j < n; ++j) {
                cout << (T[i] == j ? "O" : "*");
            }
            cout << endl;
        }
        cout << endl;
    }

    public:

    NReines(int n) {
        this->n = n;
        T = vector<int>(n);
        recursiu(0);
    }
}

```

Programa principal

```

int main() {
    int n = readint();
    NReines r(n);
}

```

Segona implementació:

- amb tornada enrere
- amplia la solució parcial sempre que sigui “legal” (que es pugui estendre a una solució completa)
- amb marcatges
- cost en cas pitjor: $\Theta(n^n)$

```

class NReines {
    int n; // nombre de reines
    vector<int> T; // configuracio actual
    vector<boolean> mc; // marca de les columnes
    vector<boolean> md1; // marca de les diagonals 1
    vector<boolean> md2; // marca de les diagonals 2

    inline int diag1(int i, int j) {
        return n-j-1 + i;
    }

    inline int diag2(int i, int j) {
        return i+j;
    }

    void recursiu(int i) {
        if (i == n) {
            escriure();
        } else {
            for (int j = 0; j < n; ++j) {
                if (not mc[j] and not md1[diag1(i, j)]
                    and not md2[diag2(i, j)]) {
                    T[i] = j;
                    mc[j] = true;
                    md1[diag1(i, j)] = true;
                    md2[diag2(i, j)] = true;
                    recursiu(i+1);
                    mc[j] = false;
                    md1[diag1(i, j)] = false;
                    md2[diag2(i, j)] = false;
                }
            }
        }
    }
}

```

public:

```

NReines(int n) {
    this->n = n;
    T = vector<int>(n);
    mc = vector<boolean>(n, false);
    md1 = vector<boolean>(2*n-1, false);
    md2 = vector<boolean>(2*n-1, false);
    recursiu(0);
}
};

```

```

int main() {
    int n = readint();
    NReines r(n);
}

```

Tercera implementació:

- amb tornada enrere
- amplia la solució parcial sempre que sigui “legal” (que es pugui estendre a una solució completa)
- amb marcatges
- amb un booleà per finalitzar la cerca
- cost en cas pitjor: $\Theta(n^n)$

```
class NReines {
    int n; // nombre de reines
    vector<int> T; // configuració actual
    bool trobat; // indica si ja s'ha trobat una solució
    vector<boolean> mc; // marca de les columnes
    vector<boolean> md1; // marca de les diagonals 1
    vector<boolean> md2; // marca de les diagonals 2

    inline int diag1 (int i, int j) {
        return n-j-1 + i;
    }

    inline int diag2 (int i, int j) {
        return i+j;
    }

    void recursiu(int i) {
        if (i == n) {
            trobat = true;
            escriure();
        } else {
            for (int j = 0; j < n and not trobat; ++j) {
                if (not mc[j] and not md1[diag1(i, j)])
                    and not md2[diag2(i, j)]) {
                    T[i] = j;
                    mc[j] = true;
                    md1[diag1(i, j)] = true;
                    md2[diag2(i, j)] = true;
                    recursiu(i+1);
                    mc[j] = false;
                    md1[diag1(i, j)] = false;
                    md2[diag2(i, j)] = false;
                }
            }
        }
    }
}
```

public:

```
NReines(int n) {
    this->n = n;
    T = vector<int>(n);
    mc = vector<boolean>(n, false);
    md1 = vector<boolean>(2*n-1, false);
    md2 = vector<boolean>(2*n-1, false);
    trobat = false;
    recursiu(0);
} };
```

Programa principal

```
int main() {
    int n = readint();
    NReines r(n);
}
```

La motxilla

Problema de la motxilla (entera)

Donada una motxilla que pot carregar un pes C i n objectes amb

- pesos p_1, p_2, \dots, p_n
- i valors v_1, v_2, \dots, v_n

trobar una selecció $S \subseteq \{1, \dots, n\}$ dels objectes

- amb el màxim valor $\sum_{i \in S} v_i$
- que no superi la capacitat de la motxilla:

$$\sum_{i \in S} p_i \leq C$$

Solució amb fita superior. Aquest cop es té en compte la contribució màxima que podrien arribar a tenir tots els objectes a partir de l' $i+1$ (encara que no cùpiguem a la motxilla).

Si fins i tot agafant-los tots no es pogué superar el millor cost trobat fins ara, no cal seguir per aquell camí.

```
class Motxilla {
    int n; // nombre d'objectes
    vector<double> p; // pes de cada objecte
    vector<double> v; // valor de cada objecte
    double C; // capacitat de la motxilla
    vector<boolean> s; // solució activa
    vector<boolean> sol; // millor solució provisional
    double millor; // cost millor solució provisional
    vector<double> sv; // suma de valors per fita inferior
```

La motxilla

```
void recursiu (int i, double val, double pes) {
    // i = objecte que toca tractar
    // val = valor acumulat, pes = pes acumulat
    if (i == n) {
        if (val > millor) {
            millor = val;
            sol = s;
        }
    } else {
        // 1a possibilitat: intentar agafar l'objecte i
        if (pes+p[i] <= C and val+sv[i] > millor) {
            s[i] = true;
            recursiu(i+1, val+v[i], pes+p[i]);
        }
        // 2a possibilitat: no agafar l'objecte i
        if (val+sv[i+1] > millor) {
            s[i] = false;
            recursiu(i+1, val, pes);
        }
    }
}
```

Són com abans: solucio, cost, main.

```
public:
    Motxilla (int n, vector<double> p, vector<double> v,
              double C) {
        this->n = n;
        this->p = p;
        this->v = v;
        this->C = C;
        s = sol = vector<boolean>(n);
        millor = 0;
        sv = vector<double>(n+1);
        sv[n] = 0;
        for (int i = n-1; i >= 0; --i) {
            sv[i] = sv[i+1] + v[i];
        }
        recursiu(0, 0, 0);
    }
```

El programa principal llegeix el nombre d'objectes, s'inventa els pesos, els valors i la capacitat, crea el solucionador, l'executa i n'escriu la solució.

```
int main() {
    int n = readint();
    vector<double> p = randvector(n);
    vector<double> v = randvector(n);
    double C = 0.4*n;
    cout << v << endl << p << endl << C << endl;

    Motxilla motx(n, p, v, C);
    cout << motx.cost() << endl;
    cout << motx.solucio() << endl;
}
```

- L'**anàlisi d'algorismes** estudia la quantitat de recursos que necessita un algorisme per resoldre un problema.
- La **teoria de la complexitat** considera els algorismes possibles que resolen un mateix problema.
- Mentre l'anàlisi d'algorismes se centra en els **algorismes**, la teoria de la complexitat s'interessa pels **problemes**.

Per classificar els problemes, considerarem les seves versions decisionals.

Definició

Un **problema decisional** és un problema en el qual la sortida és **sí o no**

Equivalentment, un problema és decisional quan s'ha de determinar si l'**entrada** (també anomenada **instància**) satisfà o no una certa propietat.

Molts problemes vistos fins ara són decisionals:

- **connectivitat**: donat un graf, decidir si és connex.
- **3-colorabilitat**: donat un graf, decidir si és 3-colorable.
- **accessibilitat**: donat un graf $G = (V, E)$ i dos vèrtexs $i, j \in V$, decidir si hi ha un camí a G entre i i j .

o es poden transformar en decisionals:

- **camí curt**: donat un graf $G = (V, E)$, dos vèrtexs $i, j \in V$ i un natural k , decidir si hi ha un camí a G entre i i j de longitud màxima k .

Certes versions decisionals d'alguns problemes no tenen gaire sentit.

Problema de les n -reines decisional (1a versió)

Donat un natural n , decidir si es poden col·locar n reines en un tauler $n \times n$ sense que cap n'amenaci cap altra.

Se sap que hi ha solucions per a tot $n \neq 2, 3$.

Per tant, l'algorisme següent decideix el problema en temps $\Theta(1)$.

```
REINES(n)
    si n = 2 o n = 3 llavors
        retornar FALS
    si no
        retornar CERT
```

Problemes decisionals

El que ens interessa és trobar una solució, no saber si existeix.

Problema de les n -reines decisional (2a versió)

Donat un natural n i k valors r_1, \dots, r_k , amb $k \leq n$, decidir si es poden col·locar n reines en un tauler $n \times n$ sense que cap n'amenaci cap altra i de manera que per a tot i tal que $1 \leq i \leq k$, la reina de la fila i ocupa la columna r_i .

Aquesta versió, tot i ser decisional, permet trobar una solució amb

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

execucions de l'algorisme que el resol.

Un problema decisional es representa formalment mitjançant un conjunt: el conjunt de les entrades amb resposta **sí**

Per exemple, el conjunt que representa el problema de connectivitat és el conjunt dels grafs connexos.

I el conjunt que representa el problema de primalitat és el conjunt dels nombres primers.

En general, si T és una propietat que els elements d'un conjunt d'entrades E poden tenir o no, i ens plantegem el següent problema decisional:

Problema A

Donat $x \in E$, determinar si es compleix $T(x)$

aleshores podem descriure formalment A com el conjunt:

$$A = \{ x \in E \mid T(x) \}.$$

Les entrades dels problemes pertanyen a certs **dominis de dades** (és a dir, conjunts que podem representar en un ordinador).

Per exemple:

- els nombres naturals
- les tuples de naturals
- els grafs
- els dags amb pesos en els arcs
- les fórmules booleanes

En cada cas, considerarem una funció de **mida**.

Funció de mida

Donat un $x \in E$, on E és un domini de dades, la **mida de x** , representada amb $|x|$, és el nombre de símbols necessaris per codificar x .

Donat un problema A definit sobre un conjunt d'entrades E , distingirem entre:

- les **entrades positives**: les que pertanyen a A
- les **entrades negatives**: les que pertanyen a $E - A$

Primalitat

El problema de la primalitat el podem descriure informalment així:

Primalitat

Donat un natural x , determinar si x és primer.

O bé formalment com el conjunt de les entrades positives:

$$P = \{x \in \mathbb{N} \mid x \text{ és primer}\}.$$

Un exemple de funció de mida per als naturals és la que compta el nombre de dígits de la representació binària:

$$|x| = \text{nombre de dígits de } x \text{ en binari} = \lfloor \log_2 x \rfloor + 1.$$

Problemes decisionals

Ara que ja podem descriure els problemes com a objectes matemàtics, els podem agrupar en classes en funció de la seva complexitat.

- Considerarem classes de problemes segons els recursos necessaris per resoldre'l's.
- Cal distingir entre tres nivells d'abstracció:
 - Les **entrades**
Per exemple, les seqüències d'enters
 - Els **problems**: conjunts d'entrades
Per exemple, les seqüències d'enters ordenades
 - Les **classes**: conjunts de problemes
Per exemple, els que podem resoldre en temps lineal

Temps polinòmic i exponencial

Suposem que $t : \mathbb{N} \rightarrow \mathbb{N}$ és una funció.

Algorismes de cost t

Diem que un algorisme \mathcal{A} té cost t si el seu cost en cas pitjor pertany a $\mathcal{O}(t)$.

Problemes decidibles en temps t

Si un algorisme \mathcal{A} rep entrades d'un conjunt E i té una sortida binària, escriurem:

$$\mathcal{A} : E \rightarrow \{0, 1\}.$$

Diem que un problema decisió A és decidible en temps t

si existeix un algorisme de cost t que el decideix (el resol); és a dir, si existeix $\mathcal{A} : E \rightarrow \{0, 1\}$ de cost t tal que, per a tot $x \in E$:

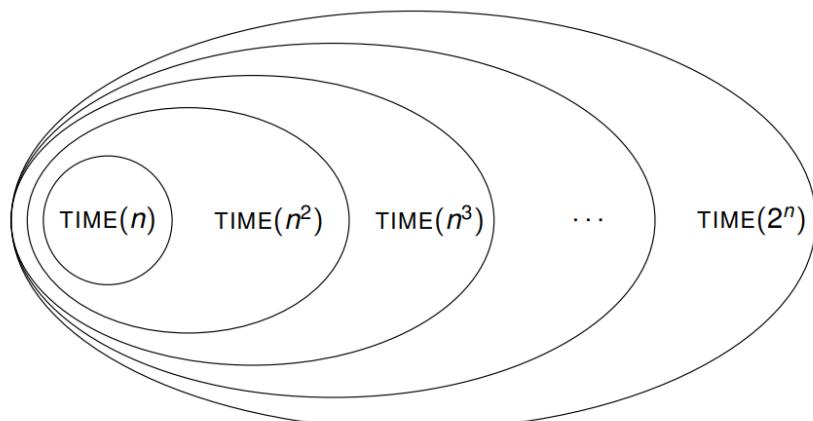
$$x \in A \Rightarrow \mathcal{A}(x) = 1$$

$$x \notin A \Rightarrow \mathcal{A}(x) = 0$$

Classe TIME(t)

Donada una funció $t : \mathbb{N} \rightarrow \mathbb{N}$, agrupem els problemes decidibles en temps t :

$$\text{TIME}(t) = \{A \mid A \text{ és decidible en temps } t\}.$$



Classe P

Definim la classe P com la unió de les classes de temps polinòmiques:

$$P = \bigcup_{k>0} \text{TIME}(n^k).$$

És a dir, un problema pertany a P si és decidible en temps n^k per a algun k .

P són els problemes que podem resoldre amb un algorisme polinòmic

Classe EXP

Definim la classe EXP com la unió de les classes de temps exponencials:

$$\text{EXP} = \bigcup_{k>0} \text{TIME}(2^{n^k}).$$

És a dir, un problema és a EXP si és decidible en temps 2^{n^k} per a algun k .

EXP són els problemes que podem resoldre amb un algorisme exponencial

Es considera que els problemes de la classe P són tractables, mentre que els de la classe EXP que no estan a P són intractables

Exemples

- CONNECTIVITAT $\in P$
- ACCESSIBILITAT $\in P$
- PRIMALITAT $\in P$
- CAMÍ CURT $\in P$
- 2-COLORABILITAT $\in P$
- 3-COLORABILITAT $\in \text{EXP}$ (no se sap si és a P)
- VIATJANT $\in \text{EXP}$ (no se sap si és a P)
- ATURADA FITADA $\in \text{EXP}$ (i se sap que no és a P)

Teorema

$$P \subsetneq \text{EXP}.$$

La inclusió estricta del teorema es pot dividir en dues parts:

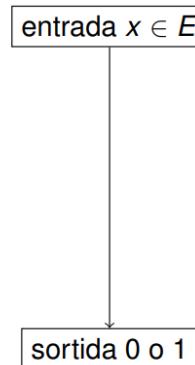
- ① $P \subseteq \text{EXP}$. Evident a partir de les definicions:

$$P = \bigcup_{k>0} \text{TIME}(n^k) \subseteq \bigcup_{k>0} \text{TIME}(2^{n^k}) = \text{EXP}$$

- ② $P \neq \text{EXP}$. La demostració està fora de l'abast de l'assignatura.

Indeterminisme

- Els algorismes vistos fins ara són **deterministes**: segueixen un únic **camí de càlcul** des de l'entrada fins al resultat.
- L'execució d'un algorisme $\mathcal{A} : E \rightarrow \{0, 1\}$ per a un conjunt de dades E es pot veure com un camí:



Un algorisme **indeterminista** pot arribar a un resultat a través de diferents camins. El seu funcionament s'assembla més a un **arbre**.

Un algorisme $\mathcal{A} : E \rightarrow \{0, 1\}$ és **indeterminista** si pot fer ús d'una nova funció

$\text{TRIAR}(x)$

que retorna un nombre y entre 0 i x .

Aleshores:

- \mathcal{A} comença el càlcul de manera determinista fins la primera instrucció **TRIAR**.
- Per a cada valor retornat per **TRIAR**, el càlcul es divideix en diferents branques amb el valor corresponent.
- Diem que \mathcal{A} retorna 1 si ho fa en **alguna** de les branques de l'arbre de càlcul.

Exemple: compostos

El problema

$$\text{COMPOSTOS} = \{x \mid \exists y \quad 1 < y < x \text{ i } y \text{ divideix } x\}$$

té un algorisme determinista trivial de temps exponencial

```
entrada x  
per a  $y = 2$  fins  $x - 1$   
  si  $y$  divideix  $x$  llavors  
    retornar 1  
  retornar 0
```

i un algorisme indeterminista de temps polinòmic

```
entrada x  
 $y \leftarrow \text{TRIAR}(x)$   
si  $1 < y < x$  i  $y$  divideix  $x$  llavors  
  retornar 1  
retornar 0
```

- En l'exemple anterior, diem que el 3 és un **testimoni** del fet que el nombre 27 és compost.
- És a dir, en el problema **COMPOSTOS** existeixen:
 - Possibles testimonis ($y < x$) del fet que un nombre x és compost. La mida dels testimonis no és més gran que la de l'entrada: $|y| \leq |x|$
 - Un algorisme polinòmic que, donat un y , **verifica** si y divideix x .
- Hi ha molts problemes per als quals hi ha testimonis curts, que es poden verificar en temps polinòmic.

Exemple: 3-colorabilitat

El problema

$$\text{3-COLORABILITAT} = \{G \mid G \text{ és 3-colorable}\}$$

també té un algorisme exhaustiu de temps exponencial

```
entrada  $G = (V, E)$   
 $n \leftarrow |V|$   
per a cada tupla  $(c_1, \dots, c_n)$  on  $\forall i \leq n \quad c_i \in \{0, 1, 2\}$   
  si  $(c_1, \dots, c_n)$  és una 3-coloració de  $G$  llavors  
    retornar 1  
  retornar 0
```

Exemple: 3-colorabilitat

i un algorisme indeterminista de temps polinòmic

```

entrada  $G = (V, E)$ 
 $n \leftarrow |V|$ 
per a  $i = 1$  fins  $n$ 
   $c_i \leftarrow \text{TRIAR}(2)$ 
si ( $c_1, \dots, c_n$ ) és una 3-coloració de  $G$  llavors
  retornar 1
si no
  retornar 0

```

La **definició formal** dels algorismes polinòmics indeterministes separa:

- el càlcul del testimoni i
- el càlcul determinista.

Decidibilitat en temps polinòmic indeterminista

Un problema decisional A definit sobre un conjunt d'entrades E es diu que és **decidible en temps polinòmic indeterminista** si existeix

- un conjunt E' de possibles testimonis
- un algorisme polinòmic $\mathcal{V} : E \times E' \rightarrow \{0, 1\}$ (anomenat **verificador**) i
- un polinomi $p(n)$

tals que per a tot $x \in E$, tenim

$$x \in A \Rightarrow \mathcal{V}(x, y) = 1 \text{ per a algun } y \in E' \text{ tal que } |y| \leq p(|x|)$$

$$x \notin A \Rightarrow \mathcal{V}(x, y) = 0 \text{ per a tot } y \in E' \text{ tal que } |y| \leq p(|x|)$$

Si $x \in A$, els y tals que $\mathcal{V}(x, y) = 1$ se'n diuen **testimonis** o **certificats**.

Per veure que un problema A és decidible en temps polinòmic indeterminista caldrà comprovar:

- que les entrades positives de A tenen testimonis de mida polinòmica i que les entrades negatives de A no tenen testimonis de mida polinòmica

(cal indicar quins són els testimonis)

- que els testimonis es poden verificar en temps polinòmic

(cal trobar un verificador)

Compostos

Considerem el problema

$$\text{COMPOSTOS} = \{x \mid \exists y \quad 1 < y < x \text{ i } y \text{ divideix } x\}$$

- Els **testimonis** per a x són tots els $y \neq 1$, x que divideixen x .
- El **polinomi** és $p(n) = n$
- El **verificador** és

```

 $\mathcal{V}(x, y)$ 
  si  $1 < y < x$  i  $y$  divideix  $x$  llavors
    retornar 1
  si no
    retornar 0

```

COMPOSTOS és decidible en temps polinòmic indeterminista perquè

$$x \in \text{COMPOSTOS} \Leftrightarrow \mathcal{V}(x, y) = 1 \text{ per a algun } y \text{ t.q. } |y| \leq p(|x|).$$

3-colorabilitat

Considerem el problema

$$\text{3-COLOR} = \{ G \mid G \text{ és 3-colorable}\}$$

- Els **testimonis** per a $G = (V, E)$ són totes les 3-coloracions C de G de la forma $C = (c_1, c_2, \dots, c_n)$, on $n = |V|$ i $c_i \in \{0, 1, 2\}$ per a tot $i \leq n$.
- El **polinomi** (amb representacions raonables de G i C) pot ser $p(n) = n$
- El **verificador** és

```

 $\mathcal{V}(G, C)$ 
   $n \leftarrow |V|$ 
  si  $C$  és una 3-coloració de  $G$  llavors
    retornar 1
  si no
    retornar 0

```

Tots els problemes decidibles en temps polinòmic indeterminista els agrupem en una classe.

Classe NP

Definim la classe NP (de *nondeterministic polynomial time*) com:

$$\text{NP} = \{A \mid A \text{ és decidible en temps polinòmic indeterminista}\}.$$

Com es relaciona NP amb P i EXP?

Indeterminisme

Diferència fonamental entre P i NP:

- els testimonis dels problemes de P es poden **trobar** en temps polinòmic
- els testimonis dels problemes de NP es poden **verificar** en temps polinòmic

Quadrats perfectes i compostos

- ① QUADRATS = $\{x \in \mathbb{N} \mid \exists y \quad 1 \leq y < x \quad \text{i} \quad x = y^2\}$
- ② COMPOSTOS = $\{x \in \mathbb{N} \mid \exists y \quad 1 < y < x \quad \text{i} \quad y \text{ divideix } x\}$

2 i 3-colorabilitat

- ① 2-COLORABILITAT = { G | G és 2-colorable }
- ② 3-COLORABILITAT = { G | G és 3-colorable }

Teorema

$$P \subseteq NP.$$

Demostració

Tot algorisme determinista també és indeterminista (però no fa ús de la instrucció TRIAR).

Vist d'una altra manera, per a tot $A \in P$, podem crear verificadors \mathcal{V} tals que

$$\mathcal{V}(x, y) = 1 \Leftrightarrow x \in A$$

independentment de y . Per tant, $A \in NP$.

Diferència entre NP i EXP:

- els problemes de NP tenen testimonis verificables en temps polinòmic
- els problemes d'EXP poden tenir testimonis exponencialment llargs

Per resoldre els problemes de NP hi ha un algorisme estàndard exponencial que cerca un testimoni i el verifica

Teorema

$$NP \subseteq EXP.$$

Demostració

Sigui $A \in NP$. Llavors, existeix un polinomi $p(n)$ i un verificador \mathcal{V} tals que

$$x \in A \Rightarrow \mathcal{V}(x, y) = 1 \text{ per a algun } y \in E \text{ tal que } |y| \leq p(|x|)$$

$$x \notin A \Rightarrow \mathcal{V}(x, y) = 0 \text{ per a tot } y \in E \text{ tal que } |y| \leq p(|x|)$$

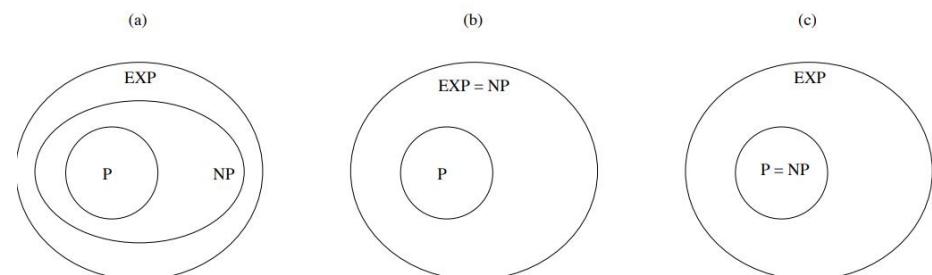
Podem considerar un algorisme exponencial per a A que cerca un testimoni:

```
entrada x
per a tot y tal que |y| ≤ p(|x|)
    si V(x, y) = 1 llavors
        retornar 1
    retornar 0
```

És fàcil veure que l'algorisme anterior és exponencial i decideix A .

Per tant, $A \in EXP$.

- No se sap si $P = NP$.
- Podem assegurar que $P \neq NP$ o $NP \neq EXP$ (perquè se sap que $P \neq EXP$).
- Per tant, hi ha tres possibilitats:



Prendrem (a) com a hipòtesi de treball.

Concepte de reducció

Reduccions

Siguin A i B dos problemes decisionals. Diem que A es redueix a B en temps polinòmic si existeix un algorisme polinòmic \mathcal{F} tal que

$$x \in A \Rightarrow \mathcal{F}(x) \in B$$

$$x \notin A \Rightarrow \mathcal{F}(x) \notin B$$

En aquest cas, escrivim $A \leq^P B$ (o $A \leq^P B$ via \mathcal{F}) i diem que \mathcal{F} és una reducció polinòmica de A a B .

Exemples de reduccions

Paritat

Considerem el llenguatge dels nombres parells

$$\text{PARELLS} = \{x \in \mathbb{N} \mid \exists y \in \mathbb{N} \quad x = 2y\}$$

i el dels senars

$$\text{SENARS} = \{x \in \mathbb{N} \mid \exists y \in \mathbb{N} \quad x = 2y + 1\}$$

Veiem que podem reduir PARELLS a SENARS ($\text{PARELLS} \leq^p \text{SENARS}$) amb un algorisme \mathcal{F} tal que $\mathcal{F}(x) = x + 1$. És evident que per a tot x :

$$x \in \text{PARELLS} \Leftrightarrow \mathcal{F}(x) \in \text{SENARS}.$$

Fixem-nos que podem reduir SENARS a PARELLS amb el mateix algorisme \mathcal{F} , és a dir, $\text{SENARS} \leq^p \text{PARELLS}$ via \mathcal{F} . En general, però, la relació \leq^p no és simètrica.

Particions

Considereu els dos problemes següents.

Partició

Donats els naturals x_1, x_2, \dots, x_n , determinar si es poden dividir en dos grups que sumin el mateix.

Motxilla

Donats els naturals x_1, x_2, \dots, x_n i una capacitat $C \in \mathbb{N}$, determinar si es pot trobar una selecció dels x_i 's que sumi exactament C .

Formalment:

$$\text{PARTICIÓ} = \{(x_1, \dots, x_n) \mid \exists I \subseteq \{1, \dots, n\} \quad \sum_{i \in I} x_i = \sum_{i \notin I} x_i\}$$

$$\text{MOTXILLA} = \{(x_1, \dots, x_n, C) \mid \exists I \subseteq \{1, \dots, n\} \quad \sum_{i \in I} x_i = C\}$$

L'algorisme

```

 $\mathcal{F}(x_1, \dots, x_n)$ 
 $S \leftarrow \sum_{i=1}^n x_i$ 
si  $S$  és senar llavors
    retornar  $(x_1, \dots, x_n, S + 1)$ 
si no
    retornar  $(x_1, \dots, x_n, S/2)$ 

```

és una reducció polinòmica de PARTICIÓ a MOTXILLA:

$$(x_1, \dots, x_n) \in \text{PARTICIÓ} \Leftrightarrow \mathcal{F}(x_1, \dots, x_n) \in \text{MOTXILLA}.$$

Definició

Un **camí hamiltonià** d'un graf G és un camí que passa per tots els vèrtexs sense repetir-ne cap.

Propietats: reflexivitat

Per a tot A , $A \leq^p A$.

N'hi ha prou a considerar l'algorisme que calcula la funció identitat:

$\mathcal{F}(x)$
retornar x

Equivalència polinòmica

Donats dos problemes decisionals A, B , escrivim $A \equiv^p B$ si $A \leq^p B$ i $B \leq^p A$.

És evident que, per a tot x

$$x \in A \Leftrightarrow \mathcal{F}(x) = x \in A.$$

Propietats: transitivitat

Per a tot A, B, C , si $A \leq^p B$ i $B \leq^p C$, llavors $A \leq^p C$.

Si

- $A \leq^p B$ via \mathcal{F} i
- $B \leq^p C$ via \mathcal{G} ,

llavors la composició $\mathcal{G} \circ \mathcal{F}$ ($\mathcal{F} | \mathcal{G}$ en notació *pipe* de UNIX) demostra que $A \leq^p C$.

Considerem que $\mathcal{G} \circ \mathcal{F}(x) = \mathcal{G}(\mathcal{F}(x))$.

Corol·lari

Les reduccions formen un preordre.

Qüestió

Observeu que, si bé les reduccions formen un preordre, en canvi no formen un ordre parcial perquè no compleixen la propietat antisimètrica:

- $\forall A, B \quad A \leq^p B \wedge B \leq^p A \Rightarrow A = B$

Tancament de P per reduccions

Per a tot A, B , si $A \leq^p B$ i $B \in P$, llavors $A \in P$.

Si

- \mathcal{B} és un algorisme polinòmic per a B i
- \mathcal{F} és un algorisme polinòmic que demostra $A \leq^p B$,

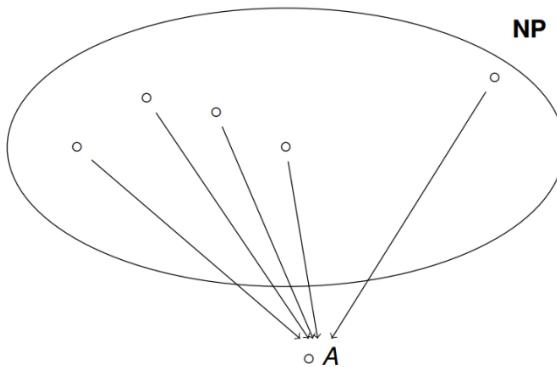
llavors la composició $\mathcal{B} \circ \mathcal{F}$ és un algorisme polinòmic per a A :

- ① $\mathcal{B} \circ \mathcal{F}$ és polinòmic perquè és composició d'algorismes polinòmics
- ② $\mathcal{B} \circ \mathcal{F}(x)$ accepta $\Leftrightarrow \mathcal{B}$ accepta $\mathcal{F}(x) \Leftrightarrow \mathcal{F}(x) \in B \Leftrightarrow x \in A$

Teoria de la NP-completesa

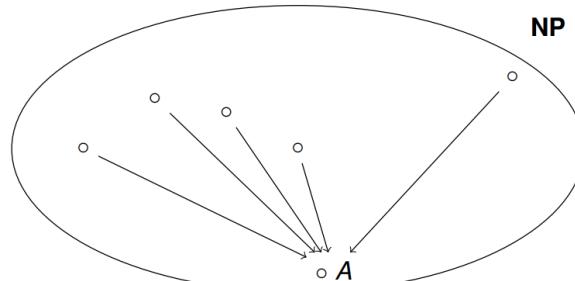
Definició

Un problema A és **NP-difícil** si per a tot problema $B \in \text{NP}$ tenim que $B \leq^P A$.



Definició

Un problema A és **NP-complet** si és NP-difícil i $A \in \text{NP}$.



Qualsevol problema NP-complet “representa” tota la classe NP respecte de P.

Més formalment...

Proposició

Sigui A un problema NP-complet. Llavors, $P = \text{NP}$ si i només si $A \in P$.

⇒ Com que A és NP-complet, $A \in \text{NP}$ i, per tant, $A \in P$.

⇐ Sigui $A \in P$.

- ① Com que A és NP-complet, sabem que per a tot $B \in \text{NP}$, $B \leq^P A$
- ② Pel tancament de P per reduccions, sabem que si $B \leq^P A$, llavors $B \in P$

Per 1 i 2, $\text{NP} \subseteq P$ i, per tant, $P = \text{NP}$.

Qualsevol parell de problemes NP-complets són equivalents.

Més formalment...

Proposició

Si A i B són NP-complets, llavors $A \equiv^P B$.

Com que A i B són NP-complets, tenim

- ① $A \in \text{NP}$ i
- ② B és NP-difícil

i llavors, $A \leq^P B$.

Simètricament, podem deduir que $B \leq^P A$. Aleshores, $A \equiv^P B$.

Fòrmules booleanes

- Una **fórmula booleana** (f.b.) és un predicat sobre variables booleanes sense quantificadors.
- Farem servir les connectives \vee (disjunció), \wedge (conjunció) i \neg (negació).

Per exemple,

$$F(x, y, z) = (x \vee y \vee \neg z) \wedge \neg(x \wedge y \wedge z)$$

és una fórmula booleana.

Forma normal conjuntiva (CNF)

- Un **literal** és una variable afirmada o negada: $x, \neg x$
- Una **clàusula** és una disjunció de literals: $(x \vee \neg y \vee z)$
- Una fórmula booleana està en **forma normal conjuntiva** (CNF) si és una conjunció de clàusules: $F(x, y, z) = (x \vee \neg y \vee z) \wedge (\neg x \vee z)$

Satisfactibilitat

Una fórmula booleana és **satisfactible** si existeix una assignació de valors de veritat a les variables per a la qual la fórmula és certa. Per exemple,

$$F(x, y, z) = (x \vee \neg y \vee z) \wedge (\neg x \vee \neg z)$$

és satisfactible amb $x = 1, y = 0, z = 0$. Escrivim $F(100) = 1$.

Definim

$$\text{SAT} = \{ F \mid F \text{ és una fórmula booleana satisfactible} \}$$

$$\text{CNF-SAT} = \{ F \mid F \text{ és una f.b. en CNF satisfactible} \}$$

Teorema de Cook-Levin (1971)

CNF-SAT és NP-complet.

Teoria de la NP-completesa

(1) CNF-SAT ∈ NP

- Els **testimonis** són les assignacions de les variables booleanes a $\{0, 1\}$
- En qualsevol codificació raonable d'una fórmula F de n variables, $n \leq |F|$. Com que un testimoni α consta de n bits, $|\alpha| = n \leq |F|$
- Per tant, triant $p(n) = n$, tenim que $|\alpha| \leq p(|F|)$
- Podem **verificar** si una assignació α satisfà F en temps polinòmic:
 - substituem les variables pels valors donats per α
 - avaluem les connectives de dins cap a fora

Exemple

En el cas de la fórmula booleana en CNF

$$F(x, y, z) = (x \vee \neg y \vee z) \wedge (x \vee \neg z)$$

i l'assignació $\alpha = 100$ (és a dir, $x = 1, y = 0, z = 0$), el verificador avaluaria:

- $F(\alpha) = (1 \vee \neg 0 \vee 0) \wedge (1 \vee \neg 0)$
(substituir valors)
- $F(\alpha) = (1 \vee 1 \vee 0) \wedge (1 \vee 1)$
(calcular negacions)
- $F(\alpha) = (1) \wedge (1)$
(calcular disjuncions)
- $F(\alpha) = 1$
(calcular conjuncions)

Lema

Donat un algorisme $\mathcal{A} : E \rightarrow \{0, 1\}$ amb cost d'espai polinòmic en cas pitjor, podem trobar en temps polinòmic una f.b. en CNF $F_{\mathcal{A}}$ tal que per a tot $y \in E$:

$$F_{\mathcal{A}}(y) = 1 \Leftrightarrow \mathcal{A}(y) = 1.$$

(2) CNF-SAT és NP-difícil.

Sigui $A \in \text{NP}$. Llavors, hi ha un polinomi q i un verificador \mathcal{V} t.q. per a tot x :

$$x \in A \Leftrightarrow \exists y \quad |y| = q(|x|) \wedge \mathcal{V}(x, y) = 1.$$

Sigui $\mathcal{V}_x(y)$ un nou verificador, per a x fixat, tal que

$$\mathcal{V}_x(y) = 1 \Leftrightarrow |y| = q(|x|) \wedge \mathcal{V}(x, y) = 1.$$

Llavors,

$$x \in A \Leftrightarrow \exists y \quad F_{\mathcal{V}_x}(y) \Leftrightarrow F_{\mathcal{V}_x} \in \text{CNF-SAT}.$$

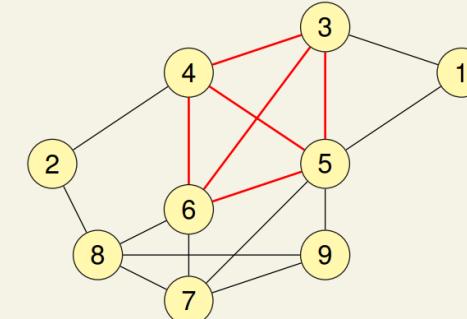
Per tant, $A \leq^p \text{CNF-SAT}$.

Problema de la clica

Diem que H és un **subgraf complet** de G si conté totes les arestes possibles entre els seus vèrtexs, és a dir, si H és isomorf a K_i per a algun i . Definim

$$\text{CLICA} = \{ (G, k) \mid G \text{ té un subgraf complet de } k \text{ vèrtexs} \}.$$

Donat el graf G



observem que $(G, 4) \in \text{CLICA}$ però $(G, 5) \notin \text{CLICA}$.

Teorema

CLICA és NP-complet

Per demostrar la NP-completesa de CLICA cal veure que:

- 1 CLICA ∈ NP
- 2 CLICA és NP-difícil

(1) CLICA ∈ NP

Sigui (G, k) una entrada de CLICA.

- Els **testimonis** són els vèrtexs dels subgrafs complets de G de k vèrtexs (en l'exemple anterior, el conjunt $C = \{3, 4, 5, 6\}$)
- El **polinomi** $p(n) = n$ és suficient perquè un testimoni C compleix $|C| \leq |(G, k)| = p(|(G, k)|)$
- Podem **verificar** en temps polinòmic si un conjunt C és un testimoni: tot parell de vèrtexs de C ha de formar una aresta en G ($\binom{|C|}{2} \leq n^2$ comprovacions)

(2) CLICA és NP-difícil

Demostrarem que $\text{CNF-SAT} \leq^p \text{CLICA}$. Aleshores,

- Com que CNF-SAT és NP-difícil, tot $S \in \text{NP}$ compleix $S \leq^p \text{CNF-SAT}$
- Per transitivitat, tot $S \in \text{NP}$ complirà $S \leq^p \text{CLICA}$
- Per tant, CLICA és NP-difícil

Problemes NP-complets

Podem expressar aquesta propietat en general.

Proposició

Sigui A un problema NP-complet i B un problema tal que $B \in \text{NP}$ i $A \leq^P B$. Llavors, B també és NP-complet.

- Com que A és NP-difícil, qualsevol $S \in \text{NP}$ satisfà $S \leq^P A$
- Per transitivitat, qualsevol $S \in \text{NP}$ satisfà $S \leq^P B$
- Per tant, B és NP-difícil

CNF-SAT \leq^P CLICA

Sigui F una fórmula booleana en CNF amb:

- clàusules C_1, \dots, C_m
- literals l_1, \dots, l_r

L'algorisme de reducció és $\mathcal{R}(F) = (G, m)$, on $G = (V, E)$ és:

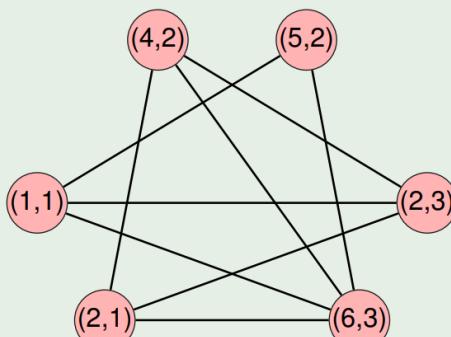
- $V = \{(i, j) \mid l_i$ apareix a $C_j\}$
(Els vèrtexs representen ocurrències de literals en clàusules)
- $E = \{\{(i, j), (k, l)\} \mid j \neq l \wedge \neg l_i \neq l_k\}$
(Les arestes representen parells de literals que poden ser certs alhora)

Exemple

$F(x_1, x_2, x_3) = C_1 \wedge C_2 \wedge C_3$, on

- $C_1 = (x_1 \vee x_2)$, $C_2 = (\neg x_1 \vee \neg x_2)$, $C_3 = (x_2 \vee \neg x_3)$
- $l_1 = x_1$, $l_2 = x_2$, $l_3 = x_3$, $l_4 = \neg x_1$, $l_5 = \neg x_2$, $l_6 = \neg x_3$

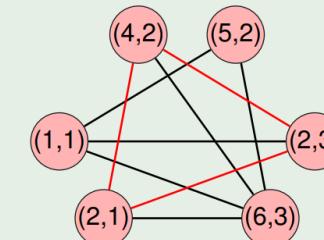
La reducció és $\mathcal{R}(F) = (G, 3)$, on G és el graf



En general, tenim que $F \in \text{CNF-SAT} \Leftrightarrow (G, m) \in \text{CLICA}$:

- ⇒ Sigui α una assignació que satisfà F . Llavors, hi ha m ocurrències de literals que α fa certes alhora i que formen un subgraf complet en G
- ⇐ Si G té un subgraf complet de m vèrtexs, cada vèrtex ha de correspondre a una clàusula diferent. Per tant, es pot fer cert un literal de cada clàusula alhora i F és satisfactible

Exemple anterior amb $l_2 = 1$, $l_4 = 1$



Definicions

- H és un **subconjunt independent** de G si consisteix en vèrtexs aïllats
- H és un **recobriment de vèrtexs** de G si té un extrem de tota aresta de G

Molts NP-complets tenen "casos particulars" que són a P.

Per exemple, en **CNF-SAT** podem fixar **el nombre de literals per clàusula** per obtenir una família infinita de problemes.

Satisfactibilitat k -fitada (k -SAT)

Donada un fórmula booleana en CNF de n variables amb $\leq k$ literals per clàusula, determinar si és satisfactible.

Satisfactibilitat 1-fitada (1-SAT)

Donada un fórmula booleana en CNF F de n variables amb 1 literal per clàusula, determinar si és satisfactible.

Per exemple,

$$F(x, y, z, t) = (x) \wedge (\neg y) \wedge (z) \wedge (\neg t).$$

1-SAT és **decidible en temps polinòmic** amb l'algorisme següent:

```
entrada  $F$ 
si  $F$  conté dos literals contradictoris llavors
    retornar 0
si no
    retornar 1
```

Problemes NP-complets

Satisfactibilitat 2-fitada (2-SAT)

Donada un fórmula booleana en CNF F de n variables amb ≤ 2 literals per clàusula, determinar si és satisfactible.

Per exemple,

$$F(x, y, z) = (x \vee y) \wedge (x \vee \neg z) \wedge (\neg x \vee y) \wedge (\neg y \vee \neg z).$$

2-SAT és **decidable en temps polinòmic**

- transformant la fórmula en un graf dirigit
- aplicant al graf un algorisme de camins

Esbós de l'algorisme

Donada una fórmula booleana en 2-CNF

$$F(x, y, z) = (x \vee y) \wedge (x \vee \neg z) \wedge (\neg x \vee y) \wedge (\neg y \vee \neg z)$$

es reescriví fent servir implicacions

$$F(x, y, z) = (\neg x \Rightarrow y) \wedge (z \Rightarrow x) \wedge (x \Rightarrow y) \wedge (y \Rightarrow \neg z)$$

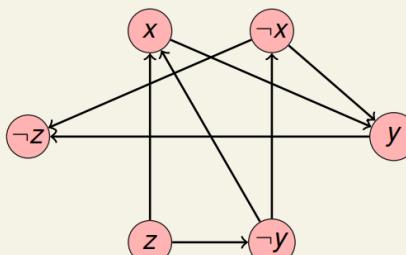
que es basen en les equivalències

- $(a \vee b) \equiv (\neg a \Rightarrow b) \equiv (\neg b \Rightarrow a)$
- $(a) \equiv (a \vee a) \equiv (\neg a \Rightarrow a) \equiv (a \Rightarrow \neg a)$

La fórmula booleana amb implicacions

$$F(x, y, z) = (\neg x \Rightarrow y) \wedge (z \Rightarrow x) \wedge (x \Rightarrow y) \wedge (y \Rightarrow \neg z)$$

es transforma en un digraf D_F i s'aplica el lema següent.



Lema

F és insatisfactible si i només per a algunes variables x , D_F té camins de x a $\neg x$ i de $\neg x$ a x .

Satisfactibilitat 3-fitada (3-SAT)

Donada un fórmula booleana en CNF F de n variables amb ≤ 3 literals per clàusula, determinar si és satisfactible.

Teorema

3-SAT és NP-complet.

Per demostrar-ho, cal provar:

- 1 3-SAT \in NP
(sembrant a CNF-SAT)
- 2 3-SAT és NP-difícil
(demonstre CNF-SAT \leq^P 3-SAT)

CNF-SAT \leq^P 3-SAT

El mètode següent transforma un fórmula booleana en CNF en una altra d'equivalent en 3-CNF.

Donada una f.b. F en CNF,

- 1 Sigui F' una f.b. buida
- 2 Per a cada clàusula $C = (a_1 \vee \dots \vee a_k)$ de F :

- si $k \leq 3$, afegir C a F'
- si $k > 3$, afegir a F' la clàusula

$$(a_1 \vee a_2 \vee z_1) \wedge (\neg z_1 \vee a_3 \vee z_2) \wedge (\neg z_2 \vee a_4 \vee z_3) \dots (\neg z_{k-3} \vee a_{k-1} \vee a_k)$$

on z_1, \dots, z_{k-3} són variables noves.

- 3 Retornar F'

Exemple

Donada una clàusula de cinc literals $C = (a_1 \vee a_2 \vee a_3 \vee a_4 \vee a_5)$, la reducció retorna

$$C' = (a_1 \vee a_2 \vee z_1) \wedge (\neg z_1 \vee a_3 \vee z_2) \wedge (\neg z_2 \vee a_4 \vee a_5).$$

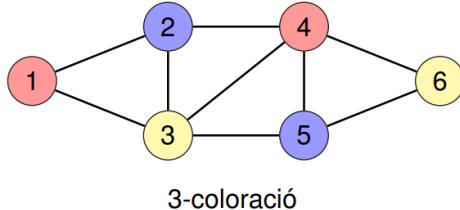
- És evident que si C és certa amb una assignació α , C' es pot satisfacer amb α i valors adequats de z_1 i z_2
- Si C' és certa amb una assignació β , algun a_i serà cert i C serà certa amb l'assignació als a_i 's de β

Definició

Un graf $G = (V, E)$ de n vèrtexs és ***k*-colorable** si existeix una funció total

$$\chi : V \rightarrow \{1, \dots, k\}$$

t.q. $\chi(u) \neq \chi(v)$ per a cada aresta $\{u, v\} \in E$. La funció χ és una ***k*-coloració**.



Amb el nombre de colors k com a paràmetre extern, podem plantejar el problema de la **colorabilitat** en funció de k .

k-Colorabilitat (***k***-COLOR)

Donat un graf G , determinar si és k -colorable.

Per als casos següents se'n coneixen algorismes polinòmics:

- 1-COLOR
- 2-COLOR

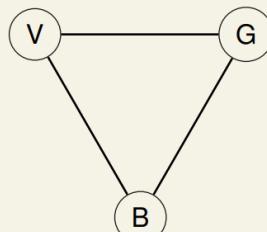
Per a 3-COLOR, demostrem la NP-completesa:

- Ja hem vist que 3-COLOR \in NP
- Ara veurem que és NP-dificil amb una reducció des de 3-CNF-SAT

CNF-SAT \leq^p 3-COLOR

Sigui F una fórmula booleana en CNF. Construirem un graf G que serà 3-colorable si i només si F és satisfactible.

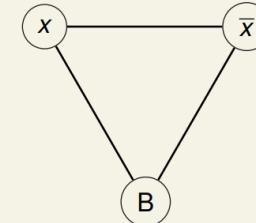
- Hi haurà 3 vèrtexs especials anomenats V , G , B .



Podem suposar que, en qualsevol coloració, tenen els colors:

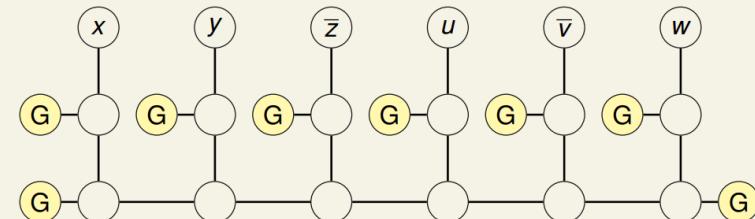
$$V \rightarrow \text{vermell}, G \rightarrow \text{groc}, B \rightarrow \text{blau}$$

- Afegim un vèrtex per cada literal i connectem cada literal i el seu complementari al vèrtex B .



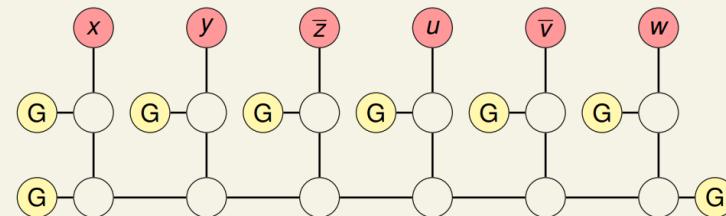
- Per cada clàusula, afegim un subgraf com el següent. En aquest cas

$$(x \vee y \vee \bar{z} \vee u \vee \bar{v} \vee w).$$



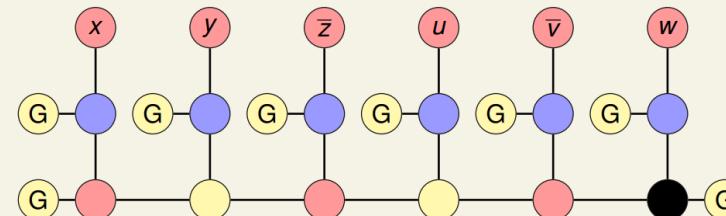
Propietat: Una coloració dels vèrtexs superiors amb vermell o groc es pot estendre a una 3-coloració global si i només si almenys un és groc.

Si tots els de dalt són vermells...



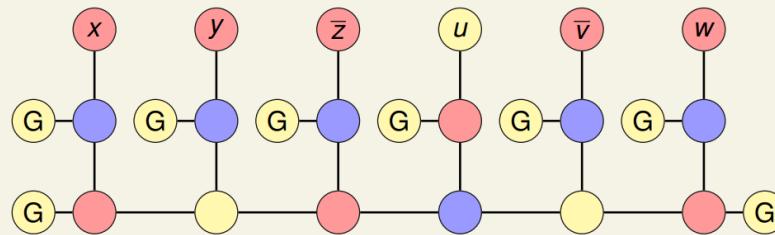
...no podem completar la 3-coloració.

Si tots els de dalt són vermells...



...no podem completar la 3-coloració.

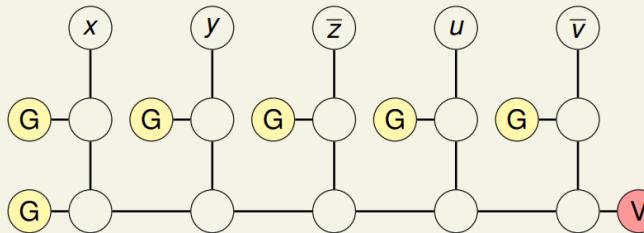
Si almenys un de dalt és groc...



...podem obtenir una 3-coloració global.

En cas que el nombre de literals sigui senar, el vèrtex de la dreta serà V.
Per exemple,

$$(x \vee y \vee \bar{z} \vee u \vee \bar{v})$$



Si G és el graf amb tots els vèrtexs i arestes definitos abans, llavors

F és satisfactible $\Leftrightarrow G$ és 3-colorable.

Com que G es pot construir en temps polinòmic, tenim que

$$\text{CNF-SAT} \leq^p \text{3-COLOR}.$$

Teorema

3-COLOR és NP-complet.

Què podem dir de la **colorabilitat de grafs planars**? Considerem la sèrie de problemes següent.

k-Colorabilitat planar (k-COLOR-PL)

Donat un graf planar G , determinar si és k -colorable.

La planaritat es pot comprovar en temps polinòmic

Per la resta de problemes k -COLOR, podem observar el següent.

Proposició

Per a tot $k > 3$, 3-COLOR $\leq^p k$ -COLOR.

La reducció consisteix, donat un graf G , a afegir-li un subgraf complet de $k - 3$ vèrtexs connectats a tots els de G .

Corol·lari

Per a tot $k > 3$, k -COLOR és NP-complet.

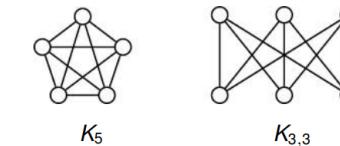
Per tant, tenim:

- k -COLOR $\in P$ per a $k \leq 2$
- k -COLOR és NP-complet per a $k \geq 3$

Definició de planaritat

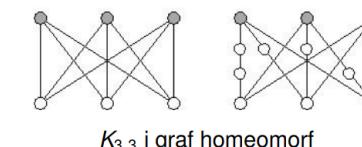
Un graf és planar si es pot dibuixar en el pla sense creuaments d'arestes.

Els grafs planars tenen **aplicacions** en disseny de circuits i gràfics.



Teorema de Kuratowski

Un graf és planar si i només si no conté cap subgraf homeomorf a K_5 o $K_{3,3}$.



$K_{3,3}$ i graf homeomorf

Teorema de Kuratowski

Un graf és planar si i només si no conté cap subgraf homeomorf a K_5 o $K_{3,3}$.

Test de planaritat

• Força bruta: $O(n^6)$

- Contreure arestes de grau 2
- Comprovar si cada subconjunt de 5 vèrtexs és un K_5
- Comprovar si cada subconjunt de 6 vèrtexs és un $K_{3,3}$

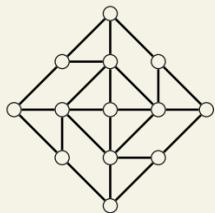
• Eficient: $O(n)$

- Aplicar DFS

Problemes NP-complets

3-COLOR \leq^P 3-COLOR-PL

Donat un graf G , considerem un dibuix de G , possiblement amb creuaments d'arestes. Cada creuament el substituïm pel giny W :

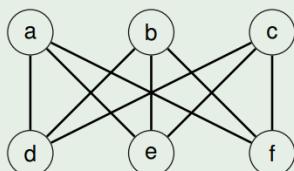


W té propietats interessants:

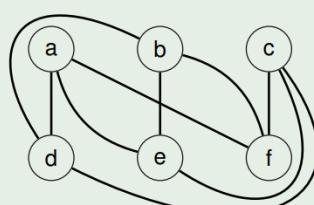
- 1 en tota 3-coloració de W , els extrems oposats tenen el mateix color
- 2 tota coloració dels extrems on els oposats tenen el mateix color es pot estendre a una 3-coloració de W

Exemple

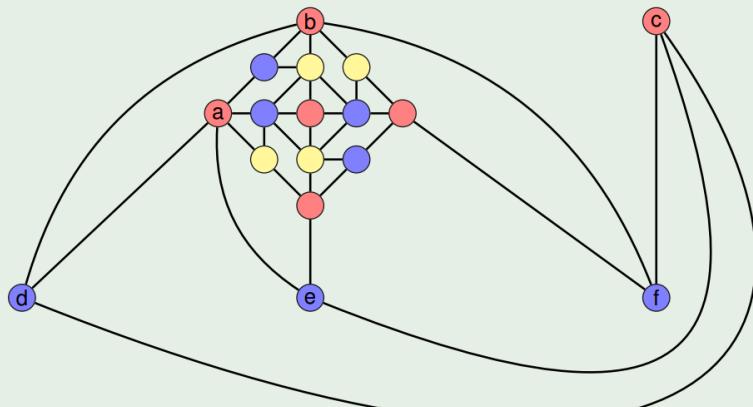
Suposem que tenim el graf $K_{3,3}$ com a entrada de 3-COLOR:



Però considerem el dibuix següent que redueix els creuaments a un:



Una 3-coloració de $K_{3,3}$ induceix una 3-coloració de (i a l'inrevés):



Corol·lari

3-COLOR-PL és NP-complet.

Per tant, tenim:

- k -COLOR-PL $\in P$ per a $k \leq 2$
- 3-COLOR-PL és NP-complet
- k -COLOR-PL $\in P$ per a $k \geq 4$
(pel teorema dels 4 colors)

Fins ara hem vist l'arbre de reduccions següent.

