

# Anàlisi i disseny d'algorismes

## 0. Presentació

José Luis Verdú Mas, Jose Oncina, Mikel L. Forcada

Dept. Llenguatges i Sistemes Informàtics  
Universitat d'Alacant

28/01/2018 (Revisió: 389)



- 1 Presentació
- 2 Introducció
- 3 Eficiència
- 4 El disseny d'algorismes. Divideix i venceràs
- 5 Programació dinàmica
- 6 Algorismes voraços
- 7 Tornada arrere
- 8 Ramificació i poda



- **Titulació:** Grau en Enginyeria Informàtica
- **6 crèdits ECTS:** 3 teòrics i 3 pràctics
- **Àrea de coneixement:** Llenguatges i Sistemes Informàtics

Departament de Llenguatges i Sistemes Informàtics  
Universitat d'Alacant



## ● Teoria

- Explicacions del professor
- Discussió d'activitats en grups menuts i discussió dirigida en el grup gran.
- Resolució d'exercicis i problemes per part de l'estudiant

## ● Pràctica

- Explicacions del professor
- Quadern de pràctiques: Problemes i implementacions que l'estudiant podrà resoldre en cada sessió. Cada treball tindrà una data límit d'entrega i es presentarà a través de Moodle. No serà imprescindible presentar-lo des del laboratori.
- Exercici pràctic: Resolució d'un problema proposat i implementació de la solució en l'ordinador (durant les últimes setmanes del curs). En aquest cas, l'estudiant defensarà el seu treball davant del professor.



# Avaluació en les convocatòries ordinàries

- Mitjançant el sistema d'**avaluació continuada**

Prova	Descripció	Ponderació
Quadern de pràctiques	Resolució de problemes i implementació d'alguns algorismes proposats	20%
Cas pràctic	Durant les últimes setmanes: Resolució, implementació i defensa d'un exercici proposat	10%
Exàmens parcials	Es faran dos exàmens parcials	20%
Examen final	Comprén tots els continguts teòrics estudiats durant el curs	50%

## ATENCIÓ

- Per a optar a l'aprobat s'ha de superar el 40% de l'examen final
- Per a la resta de proves no s'hi estableix cap mínim.
- Els exercicis de pràctiques es presentaran exclusivament en la setmana que corresponga. Són treballs no recuperables.

## Nota final

$$\max \begin{cases} 0,2 \times \text{Quadern pràct.} + 0,1 \times 1^r \text{ examen parcial} + 0,1 \times 2^r \text{ examen parcial} + 0,1 \times \text{Pràctica final} + 0,5 \times \text{Examen final} \\ 0,2 \times \text{Quadern pràct.} + 0,1 \times \text{Pràctica final} + 0,7 \times \text{Examen final} \end{cases}$$

- Sempre que la nota de l'examen final siga superior a 4.
- Si no s'arriba a aquest mínim, l'element Examen final compta en la fórmula amb el valor 0,0.

- Tant la convocatòria de juliol com la de desembre tindran un examen final específic amb les mateixes condicions que en la convocatòria de juny.
- Igualment, s'aplicarà la fórmula anterior. Les notes de la resta dels elements que componen la nota final seran les obtingudes en la convocatòria de juny; és a dir, en juliol i en desembre no hi ha entrega de pràctiques.



# Anàlisi i disseny d'algorismes

## 1: Introducció

José Luis Verdú Mas, Jose Oncina, Mikel L. Forcada

Dep. Llenguatges i Sistemes Informàtics  
Universitat d'Alacant

28/01/2018 (Revisió: 389)





- 1 Presentació
- 2 **Introducció**
- 3 Eficiència
- 4 El disseny d'algorismes. Divideix i venceràs
- 5 Programació dinàmica
- 6 Algorismes voraços
- 7 Tornada arrere
- 8 Ramificació i poda



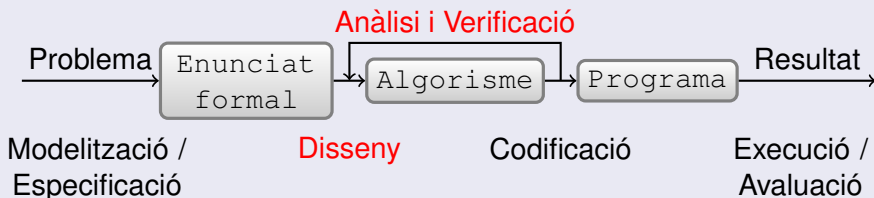
- Conèixer les diferents etapes de la resolució de problemes en programació
- Definir les etapes de disseny, anàlisi i verificació d'algorismes i conèixer-ne la importància
- Presentar les tècniques principals emprades en cadascuna de les etapes de disseny, anàlisi i verificació.



- Etapes en la resolució de problemes en programació
- L'anàlisi d'algorismes
- La verificació d'algorismes
- El disseny d'algorismes: paradigmes



## Etapes:



- Disseny i anàlisi d'algorismes.
  - Estudi de metodologies i tècniques que faciliten el disseny, anàlisi i la verificació d'algorismes

# L'anàlisi d'algorismes

- L'anàlisi d'algorismes és una disciplina en el camp de la computació que té la finalitat la de mesurar de forma quantitativa la quantitat de recursos que un algorisme necessita per a executar-se
- Recursos a analitzar:
  - Temps que un algorisme necessita per a executar-se
  - Espai (memòria) que un algorisme consumeix
- Finalitat:
  - Valoracions: l'algorisme  $A$  és “adequat”, “el més adequat”, “prohibitiu”
  - Comparacions: l'algorisme  $A$  és més adequat que el  $B$



- La verificació d'algorismes és una disciplina en el camp de la computació que té la finalitat de demostrar formalment que un algorisme funciona correctament:
  - Acaba en un temps finit
  - Retorna un resultat d'acord a la seua especificació



# Disseny d'algorismes: paradigmes

- El disseny d'algorismes estudia l'aplicació de diferents metodologies o paradigmes a la resolució de problemes en programació
- La resolució de problemes:
  - Força bruta
    - Algorismes que depenen del problema concret i no són generalitzables
    - Dificultat d'adequar canvis d'especificació
  - Paradigmes (= metodologies, esquemes, estratègies)
    - Permet la generalització i la reutilització d'algorismes
    - Cada instanciació d'un esquema dona lloc a un algorisme diferent



- Paradigmes més comuns de disseny d'algorismes
  - Divideix i venceràs
  - Programació dinàmica
  - Algorismes voraços
  - Algorismes de cerca i enumeració
    - Algorismes de tornada arrere
    - Ramificació i poda





- **“Introduction to Algorithms (Third Edition)”**  
T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein  
MIT Press, 2009
- **“Introducció a l'anàlisi i disseny d'algorismes”**; Francesc J. Ferri, Jesús V. Albert, Gregorio Martín; Universitat de València, 1998
- **“Técnicas de diseño de algoritmos”**; Rosa Guerequeta y Antonio Vallecillo; Universidad de Málaga, 1998
  - Disponible en format PDF en:  
<http://www.lcc.uma.es/~av/Libro/Libro.zip>
- **“Fundamentos de algoritmia”**; G. Brassard, P. Bratley; Prentice Hall, 1997
- **“Manual d'algorísmica: recursivitat, complexitat i disseny d'algorismes”**; Jesús Bisbal Riera; Editorial UOC, 2008



- Clases en vídeo

- <http://academicearth.org/courses/introduction-to-algorithms>
- Youtube: “Lecture \*: Data Structures and Algorithms - Richard Buckland, UNSW”

- UACloud

- Materials i anuncis
  - Apunts, transparències utilitzades pels professors, exercicis, etc.
  - Guia docent de l'assignatura
  - Anuncis i avisos a l'alumnat

- Tutories electròniques

- Tutories presencials

- consultar en <http://www.dlsi.ua.es>
- reserves: <http://www.dlsi.ua.es/alumnes/>



# Anàlisi i disseny d'algorismes

## 2: Eficiència

José Luis Verdú Mas, Jose Oncina, Mikel L. Forcada

Dep. Llenguatges i Sistemes Informàtics  
Universitat d'Alacant

28/01/2018 (Revisió: 389)



- 1 Presentació
- 2 Introducció
- 3 Eficiència**
- 4 El disseny d'algorismes. Divideix i venceràs
- 5 Programació dinàmica
- 6 Algorismes voraços
- 7 Tornada arrere
- 8 Ramificació i poda



- Proporcionar la capacitat per analitzar amb rigor l'eficiència dels algorismes
  - Distingir els conceptes d'eficiència en temps i en espai
  - Entendre i saber aplicar criteris asimptòtics als conceptes d'eficiència
  - Saber calcular la complexitat temporal o espacial d'un algorisme
  - Saber comparar, quant a l'eficiència, diferents solucions algorítmiques a un mateix problema



- Noció de complexitat
- Fites de complexitat
- Anàlisi asimptòtica
- Càlcul de complexitats
  - Algorismes iteratius
  - Algorismes recursius
- Annex



# Què és un algorisme?

## Definició: Algorisme

Un algorisme és una sèrie finita de instruccions executables i no ambigües que expressa un mètode o estratègia de resolució d'un problema

- Important:

- la **màquina** sobre la qual es defineix l'algorisme ha d'estar ben especificada
- els **recursos** (usualment temps i memòria) necessaris per a cada pas elemental han d'estar fitats
- L'algorisme ha d'**acabar** en un nombre **finit** de **passos**.



## Definició: Complexitat d'un algorisme

És una mesura de com **creixen** els **recursos** que un algorisme necessita per a executar-se quan la **grandària** del problema creix.

- Complexitat **temporal**:

**Temps** que un algorisme necessita per a executar-se en funció de la **grandària** de l'entrada

- Complexitat **espacial**:

**Memòria** que un algorisme necessita per a executar-se en funció de la **grandària** de l'entrada

Aquestes mesures ens permeten fer

- **Valoracions**: l'algorisme  $A$  és “adequat”, “el més adequat”, “prohibitiu”
- **Comparacions**: l'algorisme  $A$  és millor que el  $B$

Ens centrarem en l'estudi de la complexitat temporal





# Quina és la grandària d'un problema?

## Grandària d'un problema (instància)

Nombre de bits que es necessiten per codificar una instància

Problema	grandària
Sumar un a un enter (binari de 32 bits)	32
Dir quin és el més gran de 2 nombres	$2 \cdot 32$
Recórrer un vector de $n$ enters	$32n$
Multiplicar dues matrius d'enters de $m \times n$ i $n \times \ell$	$32(mn + n\ell)$

- Normalment es considera constant la grandària d'enters, reals, punters, etc. si s'assumeix que la seua grandària està fitada
- Quants bits es necessiten per codificar un enter  $n$  arbitràriament gran?



La complexitat pot dependre de com es codifique el problema.

## Ejemplo

Sumar un a un enter arbitràriament gran

- complexitat temporal **constant** si l'enter es codifica en base 1
- complexitat temporal **lineal** si l'enter es codifica en base 2

Normalment es prohibeixen:

- codificacions en base 1
- codificacions no compactes



- El temps d'execució d'un algorisme depén de:
  - Factors externs
    - La màquina en la qual s'executarà
    - El compilador (pot generar codi diferent).
    - Les dades d'entrada subministrats en cada execució
  - Factors interns
    - El nombre d'instruccions que executa l'algorisme i la seua durada



# Com estudiem el temps d'execució?

## Definició: anàlisi empírica o *a posteriori*

Consisteix a executar l'algorisme per a diferents valors d'entrada i **cronometrant** el temps (en segons) d'execució

- ▲ S'obté una mesura real del comportament de l'algorisme en l'entorn d'aplicació
- ▼ El resultat depén dels factors externs i interns a l'algorisme

## Definició: anàlisi teòrica o *a priori*

Consisteix a obtenir una funció que represente el temps d'execució (en operacions elementals) de l'algorisme per a qualsevol valor d'entrada.

- ▲ El resultat depén només dels factors interns
- ▲ No cal implementar i executar els algorismes
- ▼ No obté una mesura real del comportament de l'algorisme en l'entorn d'aplicació



# Temps d'execució d'un algorisme

## Definició: Operacions elementals

Considerarem elementals aquelles operacions que realitza l'ordinador en temps fitat per una constant.

## Exemple: operacions elementals

- operacions aritmètiques bàsiques
- assignacions a variables de tipus predefinit pel compilador
- els salts (crides a funcions o procediments, tornada des d'ells, etc.)
- comparacions lògiques
- accés a estructures indexades bàsiques (vectors o matrius)



# Temps d'execució d'un algorisme

Per simplificar, considerarem que el cost temporal de les operacions elementals és unitari.

## Definició: Temps d'execució d'un algorisme

Una funció ( $T(n)$ ) que mesura el nombre d'operacions elementals que realitza l'algorisme per a una grandària de problema  $n$



# Exemple

sumar els elements d'un vector

(sintaxi de la STL)

```
int suma( const vector<int> v){           // passos
    int s = 0;                           //      1
    for(int i = 0; i < v.size(); i++){    // 3+2n
        s += v[i];                       //      n
    }
    return s;                             // Total: 4+3n
}
```



# Exercicis

Transposada d'una matriu  $d \times d$  (Sintaxi de la llibreria `armadillo`)

```
int transposada( mat& A){  
    for( int i = 0; i < A.n_size; i++ )  
        for( int j = i + 1; i < A.n_size; j++ )  
            swap( A(i , j) , A(j , i) );  
}
```

Com que la complexitat del bucle interior és:  $2 + 3i$  voltes

$$T_d(d) = \underbrace{2(d-1) + 2}_{\text{bucle exterior}} + \underbrace{\sum_{i=1}^{d-1} (2 + 3i)}_{\text{interior}} = \dots = O(d^2)$$

La complexitat respecte de la grandària del problema ( $s = d^2$ ):

$$T_s(s) = T_d(d) = O(d^2) = O(s)$$





## Producte de dues matrius

(Sintaxi de la llibreria `armadillo`)

```
mat producte( mat A, mat B ){
    mat R(A.n_rows, B.n_cols);
    for( int i = 0; i < A.n_rows; i++ )
        for( int j = 0; j < B.n_cols; j++ ) {
            double acc = 0.0;
            for( int k = 0; k < A.n_cols; k++ )
                acc += A(i,k) * B(k,j);
            R(i,j) = acc;
        }
    }
    return R;
}
```

- Donat un vector d'enters  $v$  i l'enter  $z$ 
  - Retorna el primer índex  $i$  tal que  $v[i] == z$
  - Retorna  $-1$  si no troba  $z$  en el vector

```
int busca( vector<int> v, int z ){  
    for( int i = 0; i < v.size(); i++ )  
        if( v[i] == z )  
            return i;  
    return -1;  
}
```



# Problema

- No podem comptar el nombre de passos perquè per a diferents entrades d'una mateixa grandària de problema s'obtenen costos diferents
- En l'exemple de la transparència anterior:

v	z	Passos
(1, 0, 2, 4)	1	3
(1, 0, 2, 4)	0	6
(1, 0, 2, 4)	2	9
(1, 0, 2, 4)	4	12
(1, 0, 2, 4)	5	14

- Què hi podem fer?
  - Fitar el cost mitjançant dues funcions que expressen respectivament, el **cost màxim** i el **cost mínim** de l'algorisme (fites de complexitat)



- Quan apareixen diferents casos per a una mateixa talla  $n$ , s'introdueixen les mesures següents de la **complexitat**
  - Cas pitjor: **fita superior** de l'algorisme  $\rightarrow C_s(n)$
  - Cas millor: **fita inferior** de l'algorisme  $\rightarrow C_i(n)$
  - Cas mitjà: **cost mitjà**  $\rightarrow C_m(n)$
- Totes són funcions de la **grandària del problema**
- El cost mitjà és difícil d'avaluar per endavant
  - Seria necessari conèixer la **distribució de probabilitat** de l'entrada
  - **No és la mitjana de la fita inferior i de la fita superior!**



# Fites superior i inferior

```
1 int busca( vector<int> v, int z ) {  
2     for( int i = 0; i < v.size(); i++ )  
3         if( v[i] == z )  
4             return i;  
5     return -1;  
6 }
```

- En aquest cas la grandaria del problema es  $n = v.size()$

	Cas millor	cas pitjor
	$1 + 1 + 1$	$1 + 3n + 1$
Suma	3	$3n + 2$

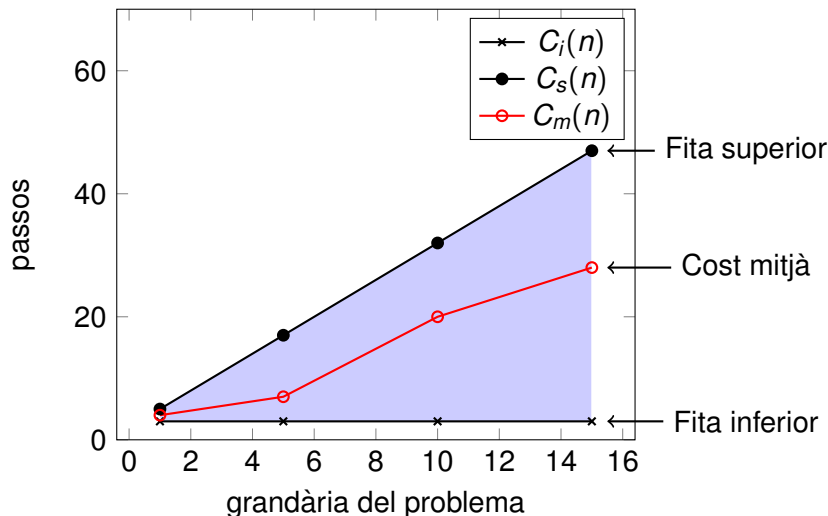
Fites:

$$C_s(n) = 3n + 2$$

$$C_i(n) = 3$$

# Fites superior i inferior

- Cost de la funció busca



- L'estudi de la complexitat resulta realment interessant **per a grandàries grans de problema** per diversos motius:
  - Les diferències “reals” en temps d'execució d'algorismes amb diferent cost per a grandàries menudes del problema no solen ser molt significatives
  - És lògic invertir temps en el desenvolupament d'un bon algorisme només si es preveu que aquest farà un gran volum d'operacions
- A l'estudi de la complexitat per a grandàries grans de problema se l'anomena **anàlisi asimptòtica**
  - Permet classificar les funcions de complexitat de manera que puguem comparar-les entre si fàcilment
  - Per a això, es defineixen classes d'equivalència que engloben les funcions que “creixen de la mateixa forma”.
- S'usa la notació asimptòtica (que definirem de seguida)



## Notació asimptòtica:

- Notació matemàtica utilitzada per representar la complexitat quan la grandària de problema ( $n$ ) creix ( $n \rightarrow \infty$ )
- Es defineixen tres tipus de notació:
  - Notació  $O$  (òmicron majúscula o *big omicron*)  $\Rightarrow$  fita superior
  - Notació  $\Omega$  (omega majúscula o *big omega*)  $\Rightarrow$  fita inferior
  - Notació  $\Theta$  (theta majúscula o *big theta*)  $\Rightarrow$  cost exacte





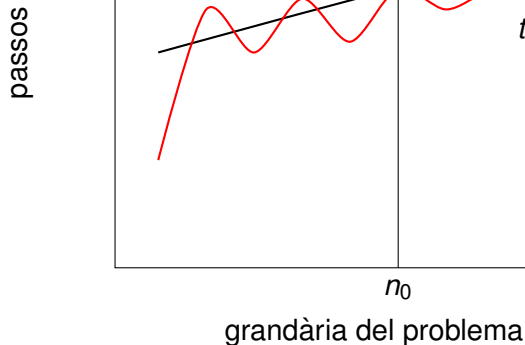
- Siga  $f : \mathbb{N} \rightarrow \mathbb{R}^+$ ; es defineix el conjunt  $O(f)$  com el conjunt de funcions fitades superiorment per un múltiple de  $f$  :

$$O(f) = \{g : \mathbb{N} \rightarrow \mathbb{R}^+ \mid \exists c \in \mathbb{R}^+ - \{0\}, \exists n_0 \in \mathbb{N}, \forall n \geq n_0, g(n) \leq cf(n)\}$$

- Donada una funció  $t : \mathbb{N} \rightarrow \mathbb{R}^+$  es diu que  $t \in O(f)$  si existeix un múltiple de  $f$  que és fita superior de  $t$  per a valors grans de  $n$



# Fita superior. Notació $O$

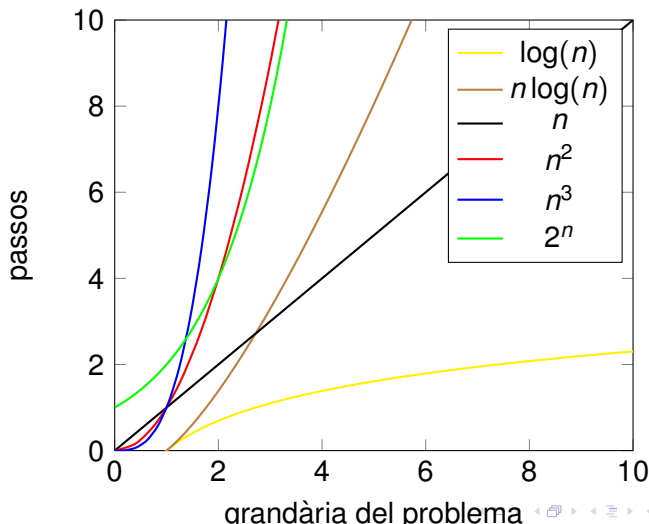


## Exemples:

- $3n + 1 \in O(n)$ ?
- $3n^2 + 1 \in O(n)$ ?
- $3n^2 + 2 \in O(n^2)$ ?

# Per a què serveix la notació asimptòtica?

- Ens permet agrupar en classes aquelles funcions amb **el mateix creixement**



$$f \in O(f)$$

$$f \in O(g) \Rightarrow O(f) \subseteq O(g)$$

$$O(f) = O(g) \Leftrightarrow f \in O(g) \wedge g \in O(f)$$

$$f \in O(g) \wedge g \in O(h) \Rightarrow f \in O(h)$$

$$f \in O(g) \wedge f \in O(h) \Rightarrow f \in O(\min\{g, h\})$$

$$f_1 \in O(g_1) \wedge f_2 \in O(g_2) \Rightarrow f_1 + f_2 \in O(\max\{g_1, g_2\})$$

$$f_1 \in O(g_1) \wedge f_2 \in O(g_2) \Rightarrow f_1 + f_2 \in O(g_1 + g_2)$$

$$f_1 \in O(g_1) \wedge f_2 \in O(g_2) \Rightarrow f_1 f_2 \in O(g_1 g_2)$$

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0 \Rightarrow f \in O(g)$$

$$f(n) = a_m n^m + \dots + a_1 n + a_0 \text{ con } a_m > 0 \Rightarrow f(n) \in O(n^m)$$

$$O(f) \subset O(g) \Rightarrow f \in O(g) \wedge g \notin O(f)$$

- Siga  $f : \mathbb{N} \rightarrow \mathbb{R}^+$ ; es defineix el conjunt  $\Omega(f)$  com el conjunt de funcions fitades inferiorment per un múltiple de  $f$ :

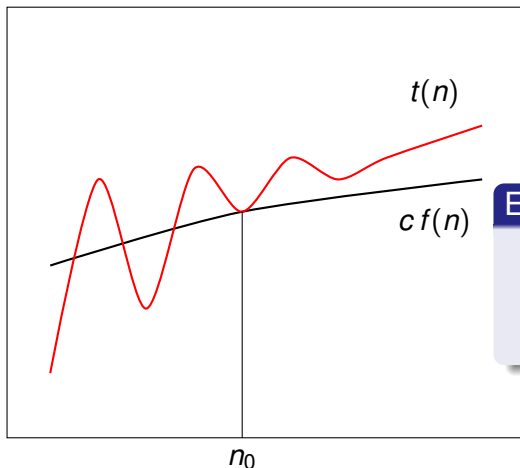
$$\Omega(f) = \{g : \mathbb{N} \rightarrow \mathbb{R}^+ \mid \exists c \in \mathbb{R}^+ - \{0\}, \exists n_0 \in \mathbb{N} : \forall n \geq n_0, g(n) \geq cf(n)\}$$

- Donada una funció  $t : \mathbb{N} \rightarrow \mathbb{R}^+$  es diu que  $t \in \Omega(f)$  si existeix un múltiple de  $f$  que és fita inferior de  $t$  per a valors grans de  $n$



# Fita inferior. Notació $\Omega$

passos



grandària del problema

## Exemples:

- $3n + 1 \in \Omega(n)$ ?
- $3n^2 + 1 \in \Omega(n)$ ?
- $3n^2 + 2 \in \Omega(n^2)$ ?



$$f \in \Omega(f)$$

$$f \in \Omega(g) \Rightarrow \Omega(f) \subseteq \Omega(g)$$

$$\Omega(f) = \Omega(g) \Leftrightarrow f \in \Omega(g) \wedge g \in \Omega(f)$$

$$f \in \Omega(g) \wedge g \in \Omega(h) \Rightarrow f \in \Omega(h)$$

$$f \in \Omega(g) \wedge f \in \Omega(h) \Rightarrow f \in \Omega(\max\{g, h\})$$

$$f_1 \in \Omega(g_1) \wedge f_2 \in \Omega(g_2) \Rightarrow f_1 + f_2 \in \Omega(\min(g_1, g_2))$$

$$f_1 \in \Omega(g_1) \wedge f_2 \in \Omega(g_2) \Rightarrow f_1 + f_2 \in \Omega(g_1 + g_2)$$

$$f_1 \in \Omega(g_1) \wedge f_2 \in \Omega(g_2) \Rightarrow f_1 f_2 \in \Omega(g_1 g_2)$$

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0 \Rightarrow g \in \Omega(f)$$

$$f(n) = a_m n^m + \cdots + a_1 n + a_0 \text{ amb } a_m > 0 \\ \Rightarrow f(n) \in \Omega(n^m)$$

- Siga  $f : \mathbb{N} \rightarrow \mathbb{R}^+$ ; es defineix el conjunt  $\Theta(f)$  com el conjunt de funcions fitades superior i inferiorment per un múltiple de  $f$ :

$$\Theta(f) = \{g : \mathbb{N} \rightarrow \mathbb{R}^+ \mid \exists c, d \in \mathbb{R}^+ - \{0\}, \exists n_0 \in \mathbb{N} : \\ \forall n \geq n_0, cf(n) \leq g(n) \leq df(n)\}$$

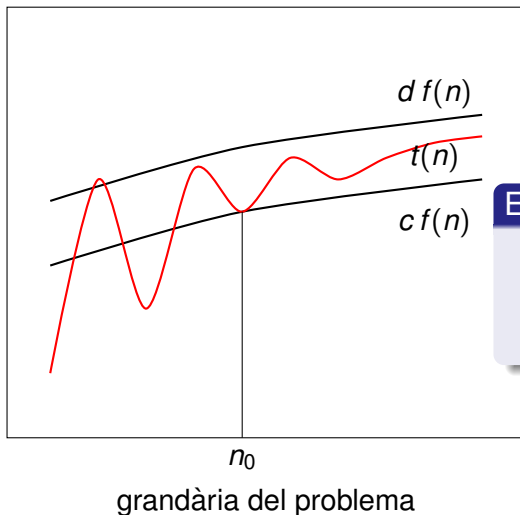
- O el que és el mateix:  $\Theta(f) = O(f) \cap \Omega(f)$
- Donada una funció  $t : \mathbb{N} \rightarrow \mathbb{R}^+$  es diu que  $t \in \Theta(f)$  si hi ha múltiples de  $f$  que son al mateix temps fita superior i fita inferior de  $t$  per a valors grans de  $n$





# Cost exacte. Notació $\Theta$

passos



## Exemples:

- $3n + 1 \in \Theta(n)$ ?
- $3n^2 + 1 \in \Theta(n)$ ?
- $3n^2 + 2 \in \Theta(n^2)$ ?



$$f \in \Theta(f)$$

$$f \in \Theta(g) \Rightarrow \Theta(f) \subseteq \Theta(g)$$

$$\Theta(f) = \Theta(g) \Leftrightarrow f \in \Theta(g) \wedge g \in \Theta(f)$$

$$f \in \Theta(g) \wedge g \in \Theta(h) \Rightarrow f \in \Theta(h)$$

$$f \in \Theta(g) \wedge f \in \Theta(h) \Rightarrow f \in \Theta(\max\{g, h\})$$

$$f_1 \in \Theta(g_1) \wedge f_2 \in \Theta(g_2) \Rightarrow f_1 + f_2 \in \Theta(g_1 + g_2)$$

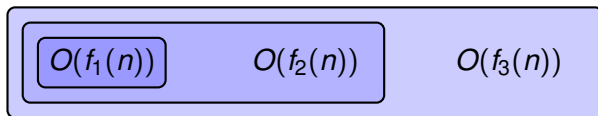
$$f_1 \in \Theta(g_1) \wedge f_2 \in \Theta(g_2) \Rightarrow f_1 f_2 \in \Theta(g_1 g_2)$$

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = k, k \neq 0, k \neq \infty \Rightarrow \Theta(f) = \Theta(g)$$

$$f(n) = a_m n^m + \dots + a_1 n + a_0 \text{ amb } a_m > 0 \\ \Rightarrow f(n) \in \Theta(n^m)$$

# Jerarquies de funcions

- Els conjunts de funcions estan inclosos uns en altres generant una ordenació de les diferents funcions. Per exemple, para  $O(\cdot)$ ,



- Les classes més utilitzades en l'expressió de complexitats són:

$$\begin{array}{ccccccc} \underbrace{O(1)}_{\text{constants}} & \subset & \underbrace{O(\log \log n)}_{\text{sublogarítmiques}} & \subset & \underbrace{O(\log n) \subset O(\log^{a(>1)} n)}_{\text{logarítmiques}} \\ & & \subset & \underbrace{O(\sqrt{n})}_{\text{sublineals}} & \subset & \underbrace{O(n)}_{\text{lineals}} & \subset & \underbrace{O(n \log n)}_{\text{lineal-logarítmiques}} \\ & & & & & & & \subset & \underbrace{O(n^2) \subset O(n^{a(>2)})}_{\text{polinòmiques}} & \subset & \underbrace{O(a(>1)^n)}_{\text{exponencials}^1} & \subset & \underbrace{O(n!) \subset O(n^n)}_{\text{superexponencials}} \end{array}$$

<sup>1</sup>Atenció:  $O(a^n) \neq O(b^n)$  per a  $a \neq b$ !

- Passos per obtenir les fites de complexitat
  - 1 Determinar la **talla** o grandària del problema
  - 2 Determinar els casos **millor** i **pitjor** (instàncies per a les quals l'algorisme tarda més o menys)
    - Per a alguns algorismes, el cas millor i el cas pitjor són el mateix ja que es comporten igualment per a qualsevol instància de la mateixa grandària
  - 3 Obtenció de les fites **per a cada cas**
    - Algorismes iteratius
    - Algorismes recursius



# Algoritmes iteratius

```
1 int suma( vector<int> v ) {  
2     int s = 0;  
3     for( int i = 0; i < v.size(); i++ )  
4         s += v[i];  
5     return s;  
6 }
```

Línea	Passos	C. Asimptòtica
2	1	$\Theta(1)$
3, 4	$n$	$\Theta(n)$
5	1	$\Theta(1)$
Suma	$n+2$	$\Theta(n)$



# Algorismes iteratius

```
1 int busca( vector<int> v, int z ) {  
2     for( int i = 0; i < v.size(); i++ )  
3         if( v[i] == z )  
4             return i;  
5     return -1;  
6 }
```

Línia	Compte Passos		C. Asimptòtica	
	Millor cas	Pitjor cas	Millor cas	Pitjor cas
2	1	$n$	$\Omega(1)$	$O(n)$
3	1	$n$	$\Omega(1)$	$O(n)$
4	1	0	$\Omega(1)$	—
5	0	1	—	$O(1)$
Suma	3	$2n + 1$	$\Omega(1)$	$O(n)$

$$C_s(n) = 2n + 1$$

$$C_i(n) = 3$$

$$C_s(n) \in O(n)$$

$$C_i(n) \in \Omega(1)$$

## Element màxim d'un vector

```
1 int maxim( vector<int> v ) {  
2     int max = v[0];  
3     for( int i = 1; i < v.size(); i++ )  
4         if( v[i] > max )  
5             max = v[i];  
6     return max;  
7 }
```



# Exemple

## Cerca en un vector ordenat

```
1 int busca( vector<int> v, int x ) {  
2     int pri = 0;  
3     int ult = v.size() - 1;  
4     while( pri < ult ) {  
5         int m = ( pri + ult ) / 2;  
6         if ( v[m] < x )      pri = m + 1;  
7         else              ult = m;  
8     }  
9     if( v[pri] == x ) return pri;  
10    else              return -1;  
11 }
```





- Donat un algorisme recursiu:

## Recerca binària

```
int busca( vector<int> v, int pri, int ult, int x){  
    if( pri == ult ) return (v[pri] == x) ? pri : -1;  
    int m = ( pri + ult ) / 2;  
    if( v[m] < x )      return busca( v, m+1, ult, x );  
    else                return busca( v, pri, m, x );  
}
```

- El cost depèn de les crides recursives, i, per tant, s'ha de definir recursivament:

$$T(n) \in \begin{cases} \Theta(1) & n = 1 \\ \Theta(1) + T(n/2) & n > 1 \end{cases} \quad (n = \text{ult} - \text{pri} + 1)$$



- Una **relació de recurrència** és una expressió que relaciona el valor d'una funció  $f$  definida per a un enter  $n$  amb un o més valors de la mateixa funció per a valors menors que  $n$

$$f(n) = \begin{cases} a f(F(n)) + P(n) & n > n_0 \\ P'(n) & n \leq n_0 \end{cases}$$

On:

- $a \in \mathbb{N}$  és una constant
- $P(n), P'(n)$  són funcions de  $n$ .
- $F(n) < n$   
(normalment  $n - b$  o  $n/b$ , amb  $b \in \mathbb{N}, b \geq 1$ )



- Les relacions de recurrència s'usen per expressar la complexitat d'un algorisme recursiu encara que també són aplicables als iteratius
- Si l'algorisme disposa de cas millor i cas pitjor, pot haver-hi una relació de recurrència per a cada cas
- La complexitat d'un algorisme s'obté en tres passos:
  - 1 Determinació de la talla del problema
  - 2 Obtenció de les relacions de recurrència de l'algorisme
  - 3 Resolució de les relacions
- Per resoldre-les, usarem el mètode de **substitució**:
  - És el mètode més senzill
  - Només per a funcions lineals (només una vegada en funció de si mateixes)
  - Consisteix a substituir cada  $f(n)$  pel seu valor en aplicar-li de nou la funció fins a obtenir un terme general



# Ordenació per selecció

- Exemple: Ordenar un vector a partir de l'element `pri`:

## Ordenació per selecció (recursiu)

```
void ordena( vector<int> &v, int pri) {  
    if( pri == v.size() )  
        return;  
    int m = pri;  
    for( int i = pri + 1; i < v.size(); i++ )  
        if( v[i] < v[m] )  
            m = i;  
    swap( v[i], v[pri]);  
    ordena(v, pri + 1);  
}
```

- Obtenir l'equació de recurrència a partir de l'algorisme:

$$T(n) \in \begin{cases} \Theta(1) & n = 1 \\ \Theta(n) + T(n-1) & n > 1 \end{cases}$$



- Resolent la recurrència per substitució

$$\begin{aligned}T(n) &= n + T(n-1) \\&= n + (n-1) + T(n-2) \\&= n + (n-1) + (n-2) + T(n-3) \\&= n + (n-1) + (n-2) + (n-3) + \cdots + 3 + 2 + T(1) \\&= \sum_{j=2}^n j + 1 \\&= \frac{n(n+1)}{2}\end{aligned}$$

Aleshores

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

# Algorisme d'ordenació per partició o *Quicksort*

- Element pivot: serveix per dividir en dues parts el vector. La seua elecció defineix variants de l'algorisme
  - A l'atzar
  - Primer element (Quicksort primer element)
  - Element central (Quicksort central)
  - Element mediana (Quicksort mediana)
- Passos:
  - Elecció del pivot
  - Es divideix el vector en dues parts:
    - part esquerra del pivot (elements menors)
    - part dreta del pivot (elements majors)
  - Es fan dues crides recursives, una amb cada part del vector



# Quicksort primer element

## Quicksort primer element

```
void quicksort( int v[], int pri, int ult ) {  
    if( ult <= pri )  
        return;  
    int p = pri;  
    int j = ult;  
    while(p < j) {  
        if( v[p+1] < v[p] ) { swap( v[p+1], v[p] );  
                               p++; }  
        else                { swap( v[p+1], v[j] );  
                               j--; }  
    }  
    quicksort(v, pri, p-1);  
    quicksort(v, p+1, ult);  
}
```

- Grandària del problema:  $n$ 
  - **Cas millor**: subproblemes  $(n/2, n/2)$

$$T(n) \in \begin{cases} \Theta(1) & n \leq 1 \\ \Theta(n) + T(\frac{n}{2}) + T(\frac{n}{2}) & n > 1 \end{cases}$$

- **Cas pitjor**: subproblemes  $(0, n-1)$  o  $(n-1, 0)$

$$T(n) \in \begin{cases} \Theta(1) & n \leq 1 \\ \Theta(n) + T(0) + T(n-1) & n > 1 \end{cases}$$





# Quicksort

- Cas millor:

$$f(n) = n + 2T\left(\frac{n}{2}\right) \quad \text{Rec. 1}$$

$$= n + 2\left(\frac{n}{2} + 2f\left(\frac{n}{4}\right)\right) = 2n + 4T\left(\frac{n}{4}\right) \quad \text{Rec. 2}$$

$$= 2n + 4\left(\frac{n}{4} + 2f\left(\frac{n}{8}\right)\right) = 3n + 8T\left(\frac{n}{8}\right) \quad \text{Rec. 3}$$

$$= in + 2^i T\left(\frac{n}{2^i}\right) \quad \text{Rec. } i$$

La recursió acaba quan  $n/2^i = 1$  per tant, hi haurà  $i = \log_2 n$  crides recursives

$$= n \log_2 n + nT(1) = n \log_2 n + n$$

Per tant,

$$T(n) \in \Omega(n \log_2 n)$$

# Quicksort

- Cas pitjor:

$$\begin{aligned}T(n) &= n + T(n-1) && \text{Rec. 1} \\&= n + (n-1) + T(n-2) && \text{Rec. 2} \\&= n + (n-1) + (n-2) + T(n-3) && \text{Rec. 3} \\&= n + (n-1) + (n-2) + \cdots + T(n-i) && \text{Rec. } i\end{aligned}$$

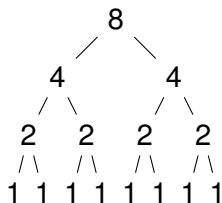
La recursió acaba quan  $n - i = 1$ ; per tant, hi haurà  $i = n - 1$  crides recursives

$$\begin{aligned}&= n + (n-1) + (n-2) + \cdots + 3 + 2 + T(1) \\&= \sum_{j=2}^n j + 1 = \frac{n(n+1)}{2}\end{aligned}$$

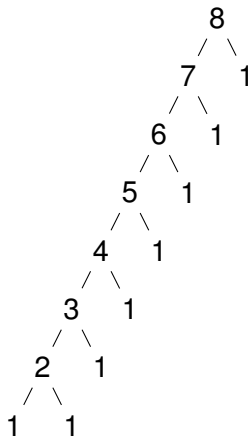
Per tant,

$$f(n) \in O(n^2)$$

# Quicksort



Cas millor  
 $\Omega(n \log_2 n)$



Cas pitjor  
 $O(n^2)$



- En la versió anterior es compleix que el cas millor és quan l'element seleccionat és la mediana (pero cal determinar-la).
- En aquest algorisme estem forçant el cas millor
- Obtenir la mediana
  - Cost menor que  $O(n \log n)$
  - S'aprofita el recorregut per reorganitzar elements i per trobar la mediana en la següent subcrida
  - La seua complexitat és per tant de  $\Theta(n \log n)$



# Anàlisi i disseny d'algorismes

## 3: El disseny d'algorismes. Divideix i venceràs

José Luis Verdú Mas, Jose Oncina, Mikel L. Forcada

Dep. Llenguatges i Sistemes Informàtics  
Universitat d'Alacant

28/01/2018 (Revisió: 389)



- 1 Presentació
- 2 Introducció
- 3 Eficiència
- 4 El disseny d'algorismes. Divideix i venceràs**
- 5 Programació dinàmica
- 6 Algorismes voraços
- 7 Tornada arrere
- 8 Ramificació i poda



# El disseny d'algorismes: objectius

- Donar a conèixer les famílies més importants de problemes algorísmics i estudiar diferents esquemes o paradigmes de disseny aplicables per a resoldre'ls.
- Aprendre a instanciar (particularitzar) un esquema genèric per a un problema concret, identificant les dades i operacions de l'esquema amb les del problema, després de comprovar que se satisfan els requisits necessaris per a la seua aplicació.
- Justificar l'elecció d'un determinat esquema quan hi ha més d'un que es pot aplicar a un mateix problema.



## DEFINICIÓ

- El disseny d'algorismes estudia l'aplicació de mètodes per resoldre problemes en programació.
- La resolució de problemes:
  - Disseny *ad hoc* (freqüentment “força bruta”)
    - Algorismes que depenen del problema concret i no són generalitzables
    - Dificultat d'adequar canvis en l'especificació
  - Esquemes:
    - Cada esquema representa un grup d'algorismes amb característiques comunes (analogia)
    - Permeten la generalització i la reutilització d'algorismes
    - Cada instanciació d'un esquema dóna lloc a un algorisme diferent





# El disseny d'algorismes: paradigmes

Esquemes algorísmics més comuns:

- Divideix i venceràs (*divide and conquer*)
- Programació dinàmica (*dynamic programming*)
- Algorismes voraços (*greedy method*)
- Algorismes de cerca i enumeració
  - Tornada arrere (*backtracking*)
  - Ramificació i poda (*branch and bound*)
- Algorismes probabilístics i heurístics<sup>2</sup>
  - Algorismes probabilístics
  - Algorismes heurístics
  - Algorismes genètics

---

<sup>2</sup>No es tractaran en l'assignatura

# Divideix i venceràs

## EXEMPLE INTRODUCTORI (1)

- Ordenar de forma ascendent un vector  $V$  de  $n$  elements.

### Solució **natural**: ordenació per selecció directa

```
1 void selection_sort (tipusElem V[], index n) {  
2     index i, j, m; tipusElem t;  
3     for (i = 0; i < n; i++) {  
4         for (j = i, m = i; j < n; j++) {  
5             if (V[j] < V[m]) m = j;  
6         }  
7         t = V[i]; V[i] = V[m]; V[m] = t;  
8     }  
9 }
```

- El bucle de la línia 3 s'executa  $n$  vegades i el bucle de la línia 4 s'executa  $n - i$  vegades amb  $i \in [0, n - 1]$ .
- Per tant la línia 5 s'executa  $\sum_{i=0}^{n-1} (n - i) = \frac{1}{2}n(n - 1)$  vegades.
- La línia 7 s'executa  $n$  vegades.

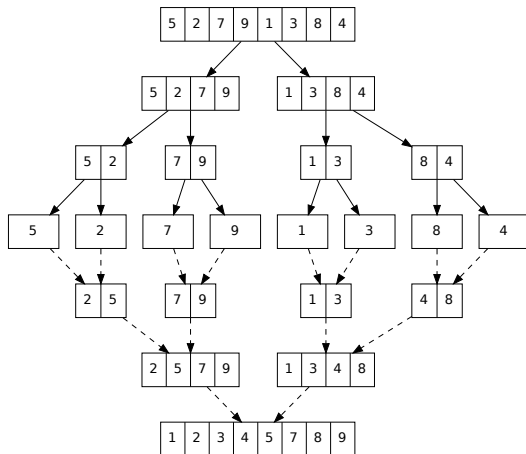
Complexitat:  $f(n) \in \Theta(n^2)$



# Divideix i venceràs

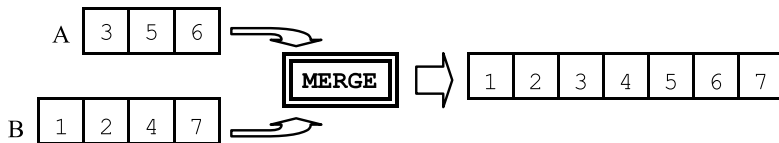
## EXEMPLE INTRODUCTORI (2)

- Ordenar de forma ascendent un vector  $V$  de  $n$  elements.
- Solució usant l'esquema “divideix i venceràs”:



## EXEMPLE INTRODUCTORI (3)

- L'algorisme `mergeSort` utilitza la funció `merge` que obté un vector ordenat com a fusió de dos vectors també ordenats



## Mergesort

```
1 void mergeSort(tipusElem V[], index pf, pi)
2 {
3     index m;
4     void merge(tipusElement* , index, index, index);
5     if (pi < pf) { /* pi=pf vol dir 1 element */
6         m = (pi+pf)/2;
7         mergeSort(V, pi, m); mergeSort(V, m+1, pf);
8         merge(V, pi, m, pf);
9     }
10    return;
11 }
```

$\text{merge}(A, pi, m, pf) \in \Theta(n)$  on  $n = pf - pi + 1$ .



## EXEMPLE INTRODUCTORI (4)

- Algorisme mergeSort
- Talla ( $n$ ) de l'algorisme ( $n = pf - pi + 1$  : nombre d'elements del vector)
- Equació de recurrència (cost exacte):<sup>3</sup>

$$T(n) = \begin{cases} 1 & n \leq 1 \\ n + 2T(\frac{n}{2}) & n > 1 \end{cases}$$

- Complexitat temporal:  $f(n) \in \Theta(n \log n)$
- Quina és la complexitat espacial?
  - Quina és la complexitat espacial de merge?
  - Atenció: es pot reutilitzar la memòria.

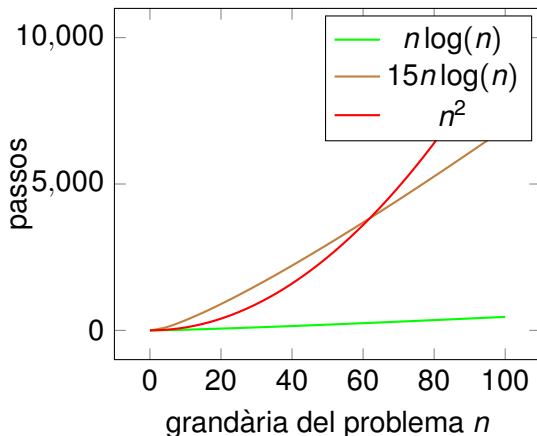
---

<sup>3</sup> $n$ , el cost de merge, és realment una expressió de l'estil de  $a + bn$ , però això no afecta al resultat final.

# Divideix i venceràs

## EXEMPLE INTRODUCTORI (5)

Comparació d'algorismes:



Només si el coeficient de  $n \log n$  és major que el pot passar que per a certs valors petits de  $n$ ,  $n \log n$  siga menys avantatjós que  $n^2$ .

## DEFINICIÓ I ÀMBIT D'APLICACIÓ

- Tècnica de disseny d'algorismes que consisteix a:
  - descompondre el problema en subproblemes de menor grandària que l'original
  - resoldre cada subproblema de forma individual i independent
  - combinar les solucions dels subproblemes per obtenir la solució del problema original
- Consideracions:
  - No sempre un problema de talla menor és més fàcil de resoldre
  - La solució dels subproblemes no implica necessàriament que la solució del problema original es pugui obtenir fàcilment
- Aplicable si trobem:
  - Forma de descompondre un problema en subproblemes de talla menor
  - Forma directa de resoldre problemes menors que una grandària determinada
  - Forma de combinar les solucions dels subproblemes que permeti obtenir la solució del problema original



# Divideix i venceràs

## EXPRESSIÓ DE L'ESQUEMA

### Esquema divideix i venceràs

```
1 tOut DV(tIn x) {  
2     tIn q[N]; tOut s[N];  
3     tIn* descompon(tIn*, int, tIn);  
4     tOut combina(tOut*, int);  
5     tOut trivial(tIn);  
6     if (menut(x)) return trivial(x);  
7     else {  
8         descompon(x, N, q);  
9         for (i=0; i<N; i++) s[i]=DV(q[i]);  
10        return combina(s, N);  
11    }  
12 }
```

Particularització (**instanciació**) de l'esquema general per al cas de Mergesort:

- **descompon**:  $m = (p_i + p_f)/2$
- **trivial**: tornada sense fer res si **menut** ( $p_i = p_f$ )
- **combina**: `merge(...)`



## Recordem **quicksort**:

```
1 void quicksort( tipusElem V[], index pri, ult ) {  
2     if( ult <= pri ) return;  
3     index p = pri; index j = ult;  
4     while(p < j) {  
5         if (V[p+1] < V[p]) { swap( V[p+1], V[p] ); p++;}  
6         else                { swap( V[p+1], V[j] ); j--;}  
7     }  
8     quicksort(V, pri, p-1);  
9     quicksort(V, p+1, ult);  
10 }
```



Particularització (**instanciació**) de l'esquema general per al cas de quickSort:

- **descompon**: càlcul de la posició  $p$  de l'element pivot (línies 3 a 7)
- **trivial**: tornada sense fer res si **menut** ( $pri = ult$ )
- **combina**: no és necessari perquè les crides recursives treballen sobre el mateix vector



## ANÀLISI D'EFICIÈNCIA (1)

- Eficiència: costos de logarítmics a exponencials. Depén de:
  - nombre de subproblemes ( $h$ )
  - Grandària dels subproblemes
  - Grau d'intersecció entre els subproblemes
- Equació de recurrència:
  - $g(n)$  = temps de **descompon** i **combina** per a grandària  $n$  (sense crides recursives)
  - $b$  = constant de divisió de grandària de problema

$$f(n) = hf\left(\frac{n}{b}\right) + g(n)$$

- Solució general: suposant l'existència d'un enter  $k$  tal que:  
 $g(n) \in \Theta(n^k)$

$$f(n) \in \begin{cases} \Theta(n^k) & \text{si } h < b^k \\ \Theta(n^k \log_b n) & \text{si } h = b^k \\ \Theta(n^{\log_b h}) & \text{si } h > b^k \end{cases}$$



## ANÀLISI D'EFICIÈNCIA (2)

- **Teorema de reducció:** els millors resultats quant a cost s'aconsegueixen quan els subproblemes són aproximadament de la mateixa grandària (i òbviament no contenen subproblemes comuns).<sup>4</sup>
- Si es compleix la condició del teorema de reducció ( $b = h = a$ )

$$f(n) = af\left(\frac{n}{a}\right) + g(n) \quad g(n) \in \Theta(n^k)$$

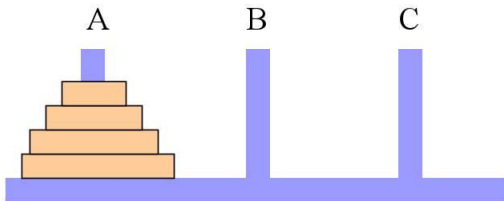
$$f(n) = \begin{cases} \Theta(n^k) & k > 1 \\ \Theta(n \log n) & k = 1 \\ \Theta(n) & k < 1 \end{cases}$$

---

<sup>4</sup>Si contenen subproblemes comuns, no hem dividit i per tant no vencerem

## • Les torres d'Hanoi: el problema

- Col·locar els discos de la torre A en la C emprant com a ajuda la torre B
- Els discos han de moure's un a un i sense col·locar mai un disc sobre un altre més petit



- Com s'abordaria el problema?
- Quina seria la complexitat de l'algorisme resultant?

## Les torres d'Hanoi: solució

- $\text{hanoi}(n, A \xrightarrow{B} C)$  és la solució del problema: moure els  $n$  discos superiors del pivot  $A$  al pivot  $C$ .
- Suposem que sabem moure  $n - 1$  discos: sabem com resoldre  $\text{hanoi}(n - 1, X \xrightarrow{Y} Z)$ .
- També sabem com moure 1 disc del pivot  $X$  al  $Z$ :  
 $\text{hanoi}(1, X \xrightarrow{Y} Z)$ , que és el cas trivial. L'anomenarem  $\text{mou}(X \rightarrow Z)$ .
- Resoldre  $\text{hanoi}(n, A \xrightarrow{B} C)$  equival a executar:
  - $\text{hanoi}(n - 1, A \xrightarrow{C} B)$
  - $\text{mou}(A \rightarrow C)$
  - $\text{hanoi}(n - 1, B \xrightarrow{A} C)$





## Les torres d'Hanoi: complexitat/1

Fixeu-vos que ací la talla dels subproblemes no és  $n/a$  sinó  $n - 1$ :

- No es poden aplicar les fórmules generals de les transparències anteriors
- Divideix i venceràs és ací més una estratègia de solució (un esquema algorísmic) que una manera d'aconseguir una solució amb menor complexitat
  - El problema té una complexitat intrínseca pitjor que les descrites en les transparències anteriors



## Les torres d'Hanoi: complexitat/2

- Equació de recurrència per al cost exacte (assumint cost 1 per a totes les operacions d'1 disc):

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

- Solució:

$$T(n) \stackrel{1}{=} 1 + 2T(n-1)$$

$$\stackrel{2}{=} 1 + 2 + 4T(n-2)$$

$$\stackrel{3}{=} 1 + 2 + 4 + 8T(n-3)$$

...

$$\stackrel{k}{=} \sum_{i=1}^k 2^{i-1} + 2^k T(n-k) = 2^k - 1 + 2^k T(n-k)$$

- Si  $k = n - 1$ ,  $T(n - k) = 1$ , i per tant,

$$T(n) = 2^n - 1 \in \Theta(2^n)$$

## EXERCICIS

- 1 Selecció del  $k$ -ésim mínim
  - Donat un vector  $A$  de  $n$  nombres enters diferents, dissenyar un algorisme que trobe el  $k$ -ésim valor mínim.<sup>5</sup>
- 2 Cerca binària o **dicotòmica**
  - Donat un vector  $X$  de  $n$  elements ordenat de forma ascendent i un element  $e$ , dissenyar un algorisme que retorne la posició de l'element  $e$  en el vector  $X$ .
- 3 Càlcul recursiu de la potència enésima.

---

<sup>5</sup>En el cas que  $k = n/2$ , aquest algorisme calcula la mediana.

# Divideix i venceràs

**Selecció del  $k$ -ésim mínim:** Donat un vector  $A$  de  $n$  nombres enters diferents, dissenyar un algorisme que hi trobe el  $k$ -ésim valor mínim.

## $k$ -ésim mínim: versió 1 (basat en QuickSort)

```
1 index quickselect (tipusElem v[], index pri, ult, k){
2   index p, j;
3   if( ult == pri ) /* mai no hauria de ser menys de 1, o siga, ult < pri */
4     if (pri==k) return v[pri];
5     else return ERROR;
6   else {
7     p = pri;
8     j = ult;
9     while(p < j) {
10      if( v[p+1] < v[p] ) { swap( v[p+1], v[p] ); p++; }
11      else { swap( v[p+1], v[j] ); j--; }
12    }
13    if (k==p) return v[p];
14    else if (k<p) return quickselect (v, pri, p-1,k);
15    else if (k>p) return quickselect (v, p+1, ult,k);
16  }
```

# Divideix i venceràs

## EXERCICIS

**Cerca binària o dicotòmica:** Donat un vector  $A$  de  $X$  elements ordenat de forma ascendent i un element  $e$ , dissenyar un algorisme que retorne la posició de l'element  $e$  en el vector  $X$ .

### Cerca binària

```
1 int cercaBinaria(tipusElem X[], index pX,uX; tipusElem e) {  
2     index m;  
3     if (pX<=uX) /* Hi ha almenys 1 element */ {  
4         m=(uX+pX)/2;  
5         if (X[m]==e) return m;  
6         else if (X[m]<e) return(cercaBinaria(X,pX,m-1,e));  
7             else return(cercaBinaria(X,m+1,uX,e));  
8     }  
9     else return -1; /* No trobat */  
10 }
```

- Reconeixeu en l'algorisme els components de l'esquema *divideix i venceràs*?



- Aquesta solució es pot veure com un *divideix i venceràs* en el qual
  - L'operació **descompon** ve representada per  $m = (uX + pX)/2$ ;
  - El problema **menut** correspon al cas en què no es compleix  $pX \leq uX$
  - Només s'ataca un dels dos subproblemes
  - No hi és necessària la combinació



# Divideix i venceràs

- Equació de recurrència **per al cas pitjor** (assumint cost 1 per a totes les operacions excepte la pròpia crida a `busquedaBinaria`):

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

(agrupem  $n = 0$  en  $n = 1$  perquè no es produeix divisió).

- Solució:<sup>6</sup>

$$\begin{aligned} T(n) &\stackrel{1}{=} 1 + T(n/2) \\ &\stackrel{2}{=} 2 + T(n/2^2) \\ &\dots \\ &\stackrel{k}{=} k + T(n/2^k) \end{aligned}$$

- Si  $2^k = n$ ,  $T(n/2^k) = T(1) = 1$ , i, per tant, com que  $k = \log(n)$ ,

$$O(n) = \log(n)$$

<sup>6</sup>S'obtidria el mateix resultat usant les fórmules generals?

## EXERCICIS

**Càlcul de la potència enèsima:** si assumim que multiplicar dos elements d'un determinat tipus té un cost constant, és possible calcular l'enèsima potència  $X^n$  d'un element  $X$  d'aquest tipus en temps sublineal usant la següent recursió:

$$X^n = \begin{cases} X & n = 1 \\ X^{\frac{n}{2}} X^{\frac{n}{2}} & n = 2k, k > 1 \\ X^{\frac{n-1}{2}} X^{\frac{n-1}{2}} X & n = 2k + 1, k > 1 \end{cases}$$

Escriuiu un algorisme per calcular eficientment  $X^n$ .

- Es pot evitar repetir operacions?
- Quin és el cost asimptòtic de l'algorisme resultant?





# Anàlisi i disseny d'algorismes

## 4. Programació dinàmica

José Luis Verdú Mas, Jose Oncina, Mikel L. Forcada

Dep. Llenguatges i Sistemes Informàtics  
Universitat d'Alacant

28/01/2018 (Revisió: 389)



- 1 Presentació
- 2 Introducció
- 3 Eficiència
- 4 El disseny d'algorismes. Divideix i venceràs
- 5 Programació dinàmica**
- 6 Algorismes voraços
- 7 Tornada arrere
- 8 Ramificació i poda



# Exemple introductori. Un problema amb complexitat polinòmica

- **Obtenir el valor del coeficient binomial**  $\binom{n}{r}$

Identitat de Pascal:<sup>7</sup>  $\binom{n}{r} = \binom{n-1}{r-1} + \binom{n-1}{r}$ ;  $\binom{n}{0} = \binom{n}{n} = 1$

Una solució recursiva ineficient

precondició:  $\{ n \geq r, n \in \mathbb{N}, r \in \mathbb{N} \}$

```
unsigned long Binomial( unsigned int n, unsigned int r){  
    unsigned long valor;  
    if (r==0 || r==n) valor = 1;  
    else                valor = Binomial(n-1, r-1) + Binomial(n-1, r);  
    return valor;  
}
```

- **Complexitat temporal** (expressada mitjançant una relació de recurrència múltiple)

$$f(n, r) = \begin{cases} 1 & \text{si } r = 0 \text{ o } r = n \\ 1 + f(n-1, r-1) + f(n-1, r) & \text{en altre cas} \end{cases}$$

LS#

# La solució recursiva es ineficient

- Aproximant a una relació de recurrència lineal:
- Si suposem que:

$$T(n-1, r) \geq T(n-1, r-1)$$

$$T(n, r) \sim g(n, r) = \begin{cases} 1 & n = r \\ 1 + 2g(n-1, r) & \text{en altre cas} \end{cases}$$

$$g(n, r) = 2^k - 1 + 2^k g(n-k, r) \quad \forall k = 1 \dots (n-r)$$

- Per tant:

$$T(n, r) \sim g(n, r) \in O(2^{n-r})$$



# La solució recursiva és ineficient

- Si suposem:

$$T(n-1, r) \leq T(n-1, r-1)$$

$$T(n, r) \sim g(n, r) = \begin{cases} 1 & r = 0 \\ 1 + 2g(n-1, r-1) & \text{en altre cas} \end{cases}$$

$$g(n, r) = 2^k - 1 + 2^k g(n-k, r-k) \quad \forall k = 1 \dots r$$

- Per això:

$$T(n, r) \sim g(n, r) \in O(2^r)$$

- Combinant les dues possibilitats:

$$T(n, r) \sim g(n, r) \in O(2^{\min(r, n-r)})$$

Aquesta solució recursiva no és acceptable!



# Exemple introductori. Una solució recursiva ineficient

$(n, r) = \binom{n}{r}$	Passos	$(n, r) = \binom{n}{r}$	Passos
(40, 0)	1	(2, 1)	3
(40, 1)	79	(4, 2)	11
(40, 2)	1559	(6, 3)	39
(40, 3)	19759	(8, 4)	139
(40, 4)	182779	(10, 5)	503
(40, 5)	$1,3 \times 10^{06}$	(12, 6)	1847
(40, 7)	$3,7 \times 10^{07}$	(14, 7)	6863
(40, 9)	$5,4 \times 10^{08}$	(16, 8)	25739
(40, 11)	$4,6 \times 10^{09}$	(18, 9)	97239
(40, 15)	$8,0 \times 10^{10}$	(20, 10)	369511
(40, 17)	$1,8 \times 10^{11}$	(22, 11)	1410863
(40, 20)	$2,8 \times 10^{11}$	(24, 12)	5408311

Pitjor cas:  $n = 2r$  (segona columna)<sup>8</sup>; creixement és aprox.  $2^n$ .

<sup>8</sup>Recurrència aproximada:  $f(n) = 1 + 2f(n-1)$ ;  $f(1)=1$

# Exemple introductori. La innecessària repetició de càlculs

- Per què és ineficient aquesta solució descendent (*top-down*)?
  - Els problemes es divideixen en subproblemes de grandària similar, en aquest cas  $n - 1$ .
    - Això és el pitjor que pot ocórrer,
    - i, tot i això, és freqüent en molts problemes importants.
  - Un problema es divideix en dos subproblemes i cadascun d'aquests en altres dos, i així successivament.
  - Ambdues circumstàncies porten a complexitats prohibitives (p.ex., exponencials)
    - Encara que de vegades es pot remeiar
- **No obstant això, el total de subproblemes diferents no és tan gran!**
  - Al cap i a la fi només hi ha  $nr$  possibilitats diferents

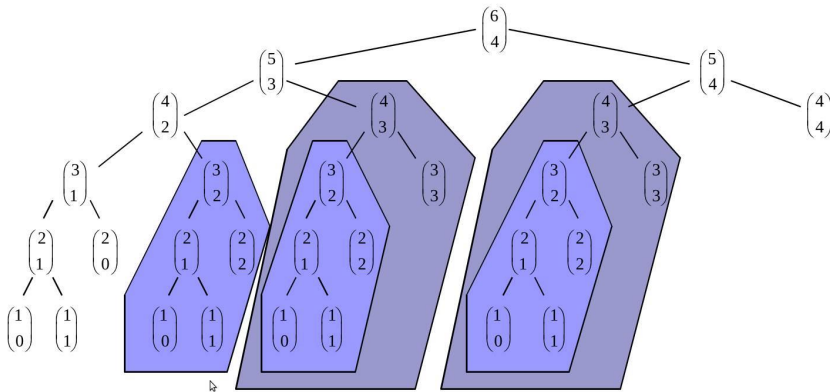
**La solució recursiva està proposant i resolent el mateix problema moltes vegades!**

- Però **atenció!** la ineficiència no és deguda a la recursivitat



# Exemple introductori. La innecessària repetició de càlculs

- Solució recursiva: exemple per a  $n = 6$  i  $r = 4$



- **INCONVENIENT:** subproblemes repetits.
  - Però només hi ha  $nr$  subproblemes diferents: el problema es pot resoldre usant magatzems intermedis





# Exemple introductori. Com evitar la repetició de càlculs?

- Es pot evitar la repetició de càlculs? En aquest cas, sí.
- Una primera solució consisteix a emmagatzemar els valors ja calculats per no haver de calcular-los de nou una segona vegada:

## Una solució recursiva millorada

$\{n \geq r, n \in \mathbb{N}, r \in \mathbb{N}\}$

```
void buida(){ /* Buidar la matriu */
    for (unsigned int i=0; i<MAX; i++)
        for (unsigned int j=0; j<MAX; j++) memo[i][j]=0; }

unsigned long Binomial( unsigned int n, unsigned int r){
    unsigned long valor;
    if (memo[n][r]) return memo[n][r];
    if (r==0 || r==n) valor = 1;
    else
        valor = Binomial(n-1, r-1) + Binomial(n-1, r);
    total++;
    memo[n][r]=valor;
    return valor; }
```

- Aquesta tècnica se sol anomenar *memoïtzació* (no *memorització*). 

# Exemple introductori. Millora important!

- Els resultats milloren moltíssim quan s'afeg un magatzem a l'algorisme recursiu:

$(n, r) = \binom{n}{r}$	Ingenu	Mem.	$(n, r) = \binom{n}{r}$	Ingenu	Mem.
(40, 0)	1	1	(2, 1)	3	3
(40, 1)	79	79	(4, 2)	11	8
(40, 2)	1559	116	(6, 3)	20	15
(40, 3)	19759	151	(8, 4)	139	24
(40, 4)	182779	184	(10, 5)	503	35
(40, 5)	$1,3 \times 10^{06}$	215	(12, 6)	1847	48
(40, 7)	$3,7 \times 10^{07}$	271	(14, 7)	6863	64
(40, 9)	$5,4 \times 10^{08}$	319	(16, 8)	25739	89
(40, 11)	$4,6 \times 10^{09}$	359	(18, 9)	97239	99
(40, 15)	$8,0 \times 10^{10}$	415	(20, 10)	369511	120
(40, 17)	$1,8 \times 10^{11}$	432	(22, 11)	1410863	143
(40, 20)	$2,8 \times 10^{11}$	440	(24, 12)	5408311	168

En el cas  $n = 2r$  (segona columna), es veu clarament un creixement del tipus  $(n/2)^2 - 1 \in \Theta(n^2)$ .

# Exemple introductori. Com evitar la repetició de càlculs?

- Hi ha més maneres d'evitar la repetició de càlculs? Sí.
  - Resoldre els subproblemes de més menut a més gran (recorregut ascendent o *bottom-up*)
  - **Emmagatzemar** les solucions en una taula  $T[0 \dots n][0 \dots r]$  on

$$T[i][j] = \binom{i}{j}$$

- L'emmagatzematge de resultats parcials permet evitar repeticions.
- La taula s'inicialitza amb la solució als subproblemes trivials (o suficientment petits):

$$T[i][0] = 1 \quad \forall i = 1 \dots (n - r)$$

$$T[i][i] = 1 \quad \forall i = 1 \dots r$$

ja que

$$\binom{m}{0} = \binom{m}{m} = 1, \quad \forall m \in \mathbb{N}$$

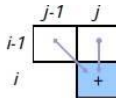


# Exemple introductori. Recorregut dels subproblemes

- Resolent els subproblemes en un ordre determinat (p.ex., en sentit ascendent) i emmagatzemant-ne les solucions:<sup>9</sup>

$$T[i][j] = T[i-1][j-1] + T[i-1][j]$$

$$\forall (i,j) : (1 \leq j \leq r, j+1 \leq i \leq n-r+j)$$



	0	1	2	3	4	$j(r)$
0	1					
1	1	1				
2	1		1			
3				1		
4					1	
5						
6						





$i(n)$

<sup>9</sup>Noteu que  $\binom{n}{r} = \binom{n}{\min(r, n-r)}$ .

# Exemple introductori. Una solució polinòmica (millorable)

- Exemple: Siga  $n = 6$  i  $r = 4$

	0	1	2	3	4
0	1				
1	1	1			
2	1	2	1		
3		3	3	1	
4			6	4	1
5				10	5
6					15

-  Cel·les sense utilitzar desaprofitament de memòria!
-  Instàncies del cas base: perfil o contorn de la matriu
-  Solucions dels subproblemes. Obtinguts, en aquest cas, de dalt cap avall i d'esquerra a dreta
-  Solució del problema inicial.  $T[6][4] = \binom{6}{4}$

## Solució trivial de programació dinàmica

$\{n \geq r, n \in \mathbb{N}, r \in \mathbb{N}\}$

```
unsigned long binomial_PD_Matriu(unsigned n, unsigned r){
    unsigned long T[n+1][r+1];

    for (unsigned i=0; i <= n-r; i++) T[i][0]= 1;
    for (unsigned i=1; i <= r; i++) T[i][i]= 1;

    for (unsigned j=1; j<=r; j++)
        for (unsigned i=j+1; i<=n-r+j; i++)
            T[i][j]= T[i-1][j-1] + T[i-1][j];

    return T[n][r];
}
```

# Exemple introductori. Una solució polinòmica (millorable)

## Solució trivial de programació dinàmica

$\{n \geq r, n \in \mathbb{N}, r \in \mathbb{N}\}$

```
unsigned long binomial_PD_Matriu(unsigned n, unsigned r){
    unsigned long T[n+1][r+1];

    for (unsigned i=0; i <= n-r; i++) T[i][0]= 1;
    for (unsigned i=1; i <= r; i++) T[i][i]= 1;

    for (unsigned j=1; j<=r; j++)
        for (unsigned i=j+1; i<=n-r+j; i++)
            T[i][j]= T[i-1][j-1] + T[i-1][j];

    return T[n][r];
}
```

- **Cost temporal exacte:**

$$c_e(n, r) = 1 + \sum_{i=0}^{n-r} 1 + \sum_{i=1}^r 1 + \sum_{j=1}^r \sum_{i=j+1}^{n-r+j} 1 = rn + n - r^2 + 1 \in \Theta(rn)$$

- Idèntic al del descendent amb memoïtzació (magatzem).



## Solució trivial de programació dinàmica

$\{n \geq r, n \in \mathbb{N}, r \in \mathbb{N}\}$

```
unsigned long binomial_PD_Matriu(unsigned n, unsigned r){
    unsigned long T[n+1][r+1];

    for (unsigned i=0; i <= n-r; i++) T[i][0]= 1;
    for (unsigned i=1; i <= r; i++) T[i][i]= 1;

    for (unsigned j=1; j<=r; j++)
        for (unsigned i=j+1; i<=n-r+j; i++)
            T[i][j]= T[i-1][j-1] + T[i-1][j];
    return T[n][r];
}
```

- **Cost espacial:**  $\Theta(rn)$  Es pot millorar?
- Podeu veure'n una de més senzilla en <http://v.gd/binCoeff>



# Exemple introductori. Per millorar la complexitat espacial

- Exercicis proposats: reducció de la complexitat espacial:
  - Modificar la funció anterior de manera que el magatzem no siga més que dos vectors de grandària  $1 + \min(r, n - r)$
  - Modificar la funció anterior de manera que el magatzem siga un únic vector de grandària  $1 + \min(r, n - r)$
  - Amb aquestes modificacions, queda afectada d'alguna manera la complexitat temporal?





# Un altre exemple introductori: Tallant tubs

- Una empresa compra tubs llargs de longitud  $n$  i els talla en tubs més curts, que ven.
- Fer-hi talls els ix gratis.
- El preu de venda d'un tub de longitud  $i$  ( $i = 1, 2, \dots, n$ ) polzades és  $p_i$ . Per exemple,

longitud $i$	1	2	3	4	5	6	7	8	9	10
preu $p_i$	1	5	8	9	10	17	17	20	24	30

- Quina és la manera òptima de tallar un tub de longitud  $n$  que maximitze el preu total?<sup>10</sup>
- Provar totes les maneres de tallar resulta prohibitiu (n'hi ha  $2^{n-1}$  maneres!). Proveu-ho per a  $n = 7$

---

<sup>10</sup>CLRS p. 360

# Un altre exemple introductori: Tallant tubs

- Busquem una descomposició  $n = i_1 + i_2 + \dots + i_k$  que es vengui pel preu màxim.
- El preu és  $r_n = p_{i_1} + p_{i_2} + \dots + p_{i_k}$ .
- Una manera de resoldre el problema recursivament és:
  - Tallar el tub de longitud  $n$  de les  $n$  formes possibles,
  - i buscar el tall que maximitza la suma del preu del tros tallat  $p_i$  i de la resta  $r_{n-i}$ ,
  - suposant que la resta del tub ha estat tallada de manera òptima:

$$r_n = \max_{1 \leq i \leq n} (p_i + r_{n-i})$$



# Un altre exemple introductori: Tallant tubs

- Solució recursiva descendent (*top-down*):

## Tallant tubs: solució recursiva descendent

```
preu talla_tub (preu p[], longitud n) {  
    preu q;  
    if (n==0) return 0;  
    q=-1;  
    for (index i=1; i<=n; i++)  
        q=max(q, p[i]+talla_tub (p, n-i));  
    return q;  
}
```

- És ineficient perquè (com en el problema anterior) es repeteixen crides amb els mateixos paràmetres.



- Complexitat de la solució recursiva:

$$T(n) = \begin{cases} 1 & \text{si } n = 0 \\ n + \sum_{j=0}^{n-1} T(j) & \text{en altre cas} \end{cases}$$

- Es fàcil<sup>11</sup> demostrar que

$$T(n) = 2^n$$

- Cap sorpresa! Hi ha  $2^{n-1}$  maneres de tallar el tub (l'arbre de recursió té  $2^{n-1}$  fulles).

---

<sup>11</sup>Adonant-se que  $T(n) - T(n-1) = 1 + T(n-1)$

## Un altre exemple introductori: Tallant tubs

- Solució recursiva descendent (*top-down*) amb magatzem (amb *memoïtzació*):

### Tallant tubs: solució recursiva descendent amb memoïtzació

```
void emplena(preu r[]) {  
  for (index i=0; i<=n; i++) r[i]=-1; }  
  
preu talla_tub_memo(preu p[], r[], longitud n) {  
  preu q;  
  if (r[n]>=0) return r[n];  
  if (n==0) q=0;  
  else {q=-1;  
    for (index i=1; i<=n; i++)  
      q=max(q, p[i]+talla_tub_memo(p, r, n-i));  
  }  
  r[n]=q;  
  return q;}
```

## Un altre exemple introductori: Tallant tubs

- Solució iterativa ascendent (*bottom-up*) amb magatzem (amb *memoïtzació*):

### Tallant tubs: solució iterativa ascendent

```
preu talla_tub_itera(preu p[], longitud n) {  
    preu r[], preu q;  
    r[0]=0;  
    for (index j=1;j<=n;j++) {  
        q=-1;  
        for (index i=1;i<=j;i++)  
            q=max(q,p[i]+r[j-i]);  
        r[j]=q;  
    }  
    return r[n];  
}
```

# Un altre exemple introductori: Tallant tubs

- Les dues solucions amb magatzem (recursiva descendent i iterativa ascendent) tenen el mateix cost temporal asimptòtic
- En l'iteratiu es veu clar que aquest cost és  $\Theta(n^2)$
- El cost espacial és  $\Theta(n)$  (vectors  $p$  i  $r$ ).



# Què hem après amb aquests exemples introductoris?

Hi ha problemes

- amb solucions recursives formalment elegants, compactes i intuïtives,
- però prohibitivament lentes perquè
  - resolen repetidament els mateixos problemes i
  - els problemes són de naturalesa similar.

Hem après a:

- **Evitar repeticions guardant resultats** (*memoïtzació*): usar un magatzem (atenció: complexitat espacial!) per evitar aquests càlculs repetits millora instantàniament el cost de les solucions descendents.
- Aprofitar-ne la **subestructura òptima**: quan la solució global a un problema incorpora solucions a problemes parcials més menuts que podem resoldre independentment, podem escriure una solució molt eficient.





## Definició: subestructura òptima

Un problema té **subestructura òptima** quan la seua solució (òptima) es pot construir eficientment a partir de les solucions (òptimes) de subproblemes seus.

- En el cas de problemes d'optimització, el concepte està relacionat amb el *principi d'“optimalitat”* de Bellman (1957).
- És una condició necessària per a trobar una solució de programació dinàmica.



# Què hem après amb aquests exemples introductoris?

Si la solució d'un problema, especialment si és d'optimització,

- és a dir, quan hi ha nombroses alternatives
- i es busca l'òptima prenent decisions

es pot obtenir

- evitant càlculs repetits, i
- traient profit de la subestructura òptima

pot convenir aplicar-hi mètodes anàlegs als vistos.

D'això se'n diu **programació dinàmica**.



# Conversió de DiV a programació dinàmica

## Esquema DiV

```
Solucio DiV( Problema p ) {  
    if( es_menusut(p) ) return trivial(p);  
    list<Solucio> s;  
    for( Problema q : descompon(p) ) s.push_back( DiV(q) );  
    return combina(s); }
```

## Esquema Programació dinàmica (recursiva)

```
Solucio PD( Problema p ) {  
    if( ja_resolt(p) ) return magatzem[p];  
    if( es_menusut(p) ) return trivial(p);  
    list<Solucio> s;  
    for( Problema q : descompon(p) ) s.push_back( PD(q) );  
    magatzem[p] = combina(s);  
    return magatzem[p]; }
```

## Esquema Programació dinàmica (iterativa)

```
Solucio PD( Problema P) {  
    vector<Solucio> magatzem;  
    Enumeracio e(P);  
    while( !empty(e) ) {  
        Problema p = pop_next(e);  
        if( es_menusut(p) )  
            magatzem[p] = trivial(p);  
        else {  
            list<Solucio> s;  
            for(Problema q : descompon(p)) s.push_back(A[q]);  
            magatzem[p] = combina(s);  
        }  
    }  
    return magatzem[P]; }
```

L'enumeració ha de complir:

- tot problema en descompon(p) apareix abans que p
- el problema P és l'últim de l'enumeració.



- Identificació:
  - Dissenyar una solució recursiva al problema (top-down)
  - Anàlisi: la complexitat temporal és prohibitiva (p.ex., exponencial)
    - Subproblemes superposats
    - Repartiment no equitatiu de les talles dels subproblemes
  - Si el problema és un problema d'optimització, verificar que s'hi pot establir una subestructura òptima.



## Transformació de recursiu a iteratiu:

- El següent pas és la construcció de la funció iterativa (bottom-up) a partir de la recursiva (top-down)
  - Les crides recursives se substitueixen per accessos a la taula-magatzem
    - Es podria dir, en termes col·loquials, que en el llenguatge de programació se substitueixen parèntesis per claudàtors.
  - Substituir l'ordre que retorna el valor en la funció recursiva per un emmagatzematge en la taula
  - Utilitzar els casos base de la solució recursiva per pre-emplenar el contorn de la taula
  - A partir del cas general en la funció recursiva, dissenyar l'estratègia que permeti crear els bucles que completen la taula a partir dels subproblemes resolts (recorregut ascendent o bottom-up)
    - **És habitual que complexitats exponencials es transformen en polinòmiques**



- La programació dinàmica recursiva consisteix a fer ús del magatzem en la versió recursiva
  - La versió recursiva pot ser més eficient que la iterativa:
    - els càlculs innecessaris poden ser més fàcils d'evitar en la recursiva que en la iterativa
- Per tant, la programació dinàmica no implica necessàriament una transformació a iteratiu
  - En els seus orígens la transformació a iteratiu era un valor afegit perquè els llenguatges de programació no admetien recursivitat



# Exemples d'aplicació

- Problemes clàssics per als quals resulta eficaç la programació dinàmica
  - El problema de la motxilla 0-1 (que veurem més endavant)
  - Càlcul dels nombres de Fibonacci.
  - Problemes amb cadenes:
    - La subseqüència comuna màxima (*longest common subsequence*) de dues cadenes.
    - La distància d'edició (*edit distance*) entre dues cadenes.
  - Problemes sobre grafs:
    - El viatjant de comerç (*travelling salesman problem*)
    - Camins més curts en un graf entre un vèrtex i tots els restants (alg. de Dijkstra)
    - Existència de camí entre qualsevol parell de vèrtexs (alg. de Warshall)
    - Camins més curts en un graf entre qualsevol parell de vèrtexs (alg. de Floyd)





## Definicions:

- Siga un graf dirigit  $G = (V, E, p)$  on
  - $V = \{1, 2, \dots, |V|\}$  són els vèrtexs,
  - $E \subseteq V \times V$  és el conjunt d'arestes de la forma  $(i, j)$ , i
  - $p : E \rightarrow \mathbb{R}$  és una funció que assigna un pes  $p(i, j)$  a cada aresta  $(i, j)$ .
- Un *camí* és una seqüència de vèrtexs  $i_1, i_2, \dots, i_N$  on cada vèrtex està unit a l'anterior per una aresta:  
 $\forall n : 1 \leq n \leq N - 1, (i_n, i_{n+1}) \in E$
- Anomenarem *pes d'un camí* a la suma dels pesos de les arestes del camí.



## Exercicis amb grafs/2

Hem de construir un algorisme que trobe el camí de menor pes entre qualssevol dos vèrtexs del graf. El cost temporal de la solució ha de ser polinòmic, de manera que pot convenir la programació dinàmica.

- 1 Per a un graf sense cicles, dissenyar una solució que comence per considerar els camins sense vèrtexs intermedis ( $m = 0$  i les vaja substituint per camins de menor pes amb un vèrtex, amb dos, etc.
- 2 (Algorisme de Floyd) Per a un graf que no té cicles de cost negatiu, dissenyar una solució que comence per considerar els camins que no usen cap vèrtex addicional  $k = 0$  i els vaja substituint per camins de menor pes que usen vèrtexs en un subconjunt  $\{1, 2, \dots, k\} \in V$  on  $k$  va augmentant fins que és tot  $V$ .



# El problema de la motxilla (Knapsack problem)



- Siguen  $n$  objectes amb valors ( $v_i \in \mathbb{R}$ ) i pesos ( $p_i \in \mathbb{R}^+ - \{0\}$ ) coneguts. Siga una motxilla amb capacitat màxima de càrrega  $P$ .
  - Quin és el valor màxim que pot transportar la motxilla sense sobrepassar la seua capacitat?
- 
- Un cas particular: **La motxilla 0/1 amb pesos discrets**
    - Els objectes **no es poden fraccionar** (motxilla 0/1 o motxilla discreta)
      - La variant més difícil
    - Els pesos són quantitats discretes o discretizables
      - Ens serviran com a índexs en indexar una taula
      - Es tracta d'una versió menys general que en suavitza la dificultat.



# El problema de la motxilla 0/1: Identificació

- És un problema d'optimització:
  - Cada solució és un conjunt d'objectes carregats, que representem com un vector booleà de decisions:  
 $(x_1, x_2 \dots x_n) : x_i \in \{0, 1\}, 1 \leq i \leq n$ 
    - En  $x_i$  s'emmagatzema la decisió sobre l'objecte  $i$ . Si és escollit, aleshores  $x_i = 1$ . En cas contrari  $x_i = 0$
  - La solució òptima està representada pel vector de decisions que maximitza l'expressió  $\sum_{i=1}^n x_i v_i$ , tot complint les restriccions:
    - $\sum_{i=1}^n x_i p_i \leq P$
    - $\forall i : 1 \leq i \leq n, x_i \in \{0, 1\}$
- Representarem mitjançant  $PM(i, j, C)$  la solució al problema de la motxilla de capacitat  $C$  amb els objectes  $i$  fins a  $j$ .
  - El problema proposat és, per tant,  $PM(1, n, P)$



# Motxilla 0/1: Subestructura òptima (I)

- Siga  $(x_1, x_2 \dots x_n)$  el vector de decisions corresponent una solució òptima al problema  $PM(1, n, P)$ . Aleshores, es compleix que:
  - Si  $x_1 = 0$  llavors  $(x_2 \dots x_n)$  representa una solució òptima per al subproblema  $PM(2, n, P)$
  - Si  $x_1 = 1$  llavors  $(x_2 \dots x_n)$  representa una solució òptima per al subproblema  $PM(2, n, P - p_1)$

## Demostració

Si existira una solució millor  $(x'_2 \dots x'_n)$  per a cadascun dels dos subproblemes aleshores la seqüència  $(x_1, x'_2 \dots x'_n)$  seria millor que  $(x_1, x_2 \dots x_n)$  per al problema original, fet que contradiria la suposició inicial que era l'òptima.<sup>a</sup>

---

<sup>a</sup>Em el llibre de CLRS aquestes demostracions s'anomenen “cut and paste”

- És fàcil demostrar que succeeix el mateix per a qualsevol subconjunt.
  - $(x_1, x_2 \dots x_i)$  és solució òptima de  $\text{PM}(1, i, \sum_{j=1}^i x_j p_j)$
  - $(x_{i+1}, x_{i+2} \dots x_n)$  és solució òptima de  $\text{PM}(i+1, n, P - \sum_{j=1}^i x_j p_j)$
- S'hi observa clarament que la solució al problema presenta una subestructura òptima.



# Motxilla 0/1: Aproximació matemàtica

- Es consideren les decisions en ordre descendent:  $x_n, x_{n-1}, \dots, x_1$ 
  - De forma anàloga l'exploració podria ser ascendent
- Davant la decisió  $x_i$  hi ha dues alternatives:
  - Rebutjar l'objecte  $i$ :  $x_i = 0$ .
    - No hi ha guany addicional però el pes de la motxilla no augmenta.
  - Seleccionar l'objecte  $i$ :  $x_i = 1$ .
    - El guany addicional és  $v_i$ , a costa d'augmentar el pes de la motxilla de  $p_i$
- Se selecciona l'alternativa que produïska el guany global més gran
  - No se sabrà mentre no es comproven totes les possibilitats

## Solució matemàtica

$$\{ P \geq 0, n > 0 \}$$

$Motxilla(n, P) = \max (Motxilla(n - 1, P), Motxilla(n - 1, P - p_n) + v_n)$   
amb

- $Motxilla(i, P) = -\infty$  si  $P < 0$
- $Motxilla(0, P) = 0, P \geq 0$

---

$Motxilla(i, P)$  representa el valor de la solució òptima a  $PM(1, i, P)$ .

# Motxilla 0/1: Una solució recursiva

- Resulta trivial a partir de la funció matemàtica:

## Una solució recursiva ineficient

```
1 vector <float> v, vector <unsigned> p;  
2 float Motxilla (int i, unsigned P){  
3     float S1, S2;  
4     if (i>0){  
5         if (p[i] <= P) //Encara hi ha capacitat en la motxilla per a l'objecte  
6             S1= v[i] + Motxilla(i-1,P-p[i]);  
7         else // S'intenta prendre l'objecte pero no hi cap  
8             S1=0;  
9         S2= Motxilla(i-1,P); //Es descarta explicitament aquest objecte  
10        return max(S1,S2); //el millor entre prendre'l o no prendre'l  
11    }  
12    return 0; // si i==0  
13 }
```





# Motxilla 0/1: Versió recursiva: Complexitat temporal


- En el millor cas, cap objecte cap en la motxilla:  $C_{\text{millor}}^T(n) \in \Omega(n)$ .
- En el pitjor cas (s'exploren tots els casos):
  - Siga  $f(n)$  el nombre de passos que realitza la funció recursiva (en el pitjor cas) davant un problema amb  $n$  objectes. S'hi té:

$$f(n) = \begin{cases} 1 & \text{si } n = 0 \\ 2f(n-1) + 1 & \text{en altre cas} \end{cases}$$

- El terme general es pot escriure com a funció d'una certa profunditat  $i \in \mathbb{N}$ ,  $1 \leq i \leq n$ :<sup>12</sup>

$$f(n) = 2^i f(n-i) + 2^i - 1$$

- Siga  $i = n$  (interessa l'últim valor de  $i$ , puix en aquest cas  $f(n-i)$  s'obté sense recursió),  $f(n) = 2^n f(0) + 2^n - 1 = 2^{n+1} - 1$
- Per tant,  $C_{\text{pitjor}}^T(n) \in O(2^n)$

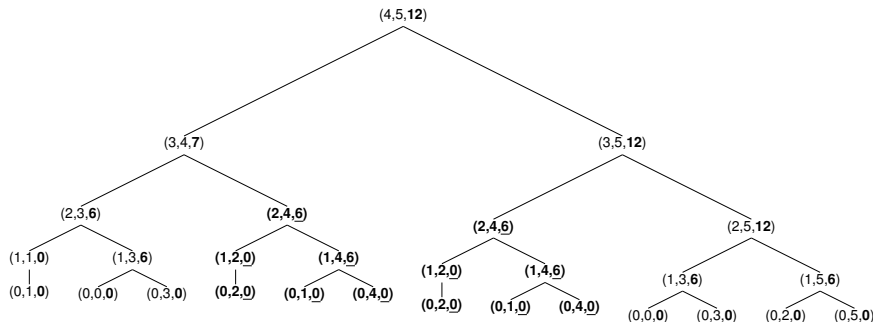
<sup>12</sup>El valor més gran que pot prendre  $i$  s'obté fàcilment en fer  $n-i=0$ , puix és el  que es compleix quan acaba la recursió (cas base).

# Motxilla 0/1: Vers. recursiva: Subproblemes repetits

$$n = 4, P = 5$$

- Exemple:  $p = (3, 2, 1, 1)$   
 $v = (6, 6, 2, 1)$

Nodes:  $(i, P, \text{Motxilla}(i, P))$ ; esquerra,  $x_i = 1$ ; dreta,  $x_i = 0$ .



# Motxilla 0/1: Magatzem de resultats parcials

- Exemple: Siguen  $n = 5$  objectes amb pesos ( $p_i$ ) i valors ( $v_i$ ) indicats en la taula. Siga  $P = 11$  el pes màxim de la motxilla.

$T[0 \dots 5][0 \dots 11]$	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
$p_1 = 2, v_1 = 1$	0	0	1	1	1	1	1	1	1	1	1	1
$p_2 = 2, v_2 = 7$	0	0	7	7	8	8	8	8	8	8	8	8
$p_3 = 5, v_3 = 18$	0	0	7	7	8	18	18	25	25	26	26	26
$p_4 = 6, v_4 = 22$	0	0	7	7	8	18	22	25	29	29	30	40
$p_5 = 7, v_5 = 28$	0	0	7	7	8	18	22	28	29	35	35	40

$T[i][j] \equiv$  Guany màxim amb els  $i$  primers objectes i amb una càrrega màxima  $j$ . Per tant, solució en  $T[5][11]$

40

Solució al problema  
Contorn o perfil

$$T[i][j] = \max(\underbrace{T[i-1][j]}_{\text{rebutjar } i}, \underbrace{T[i-1][j-p_i] + v_i}_{\text{seleccionar } i})$$

$$\begin{aligned}
 T[5][11] &= \max \left( \underline{T[4][11]}, T[4][11 - p_5] + v_5 \right) = \max(40, 36). && 5 \text{ no es pren} \\
 T[4][11] &= \max \left( T[3][11], \underline{T[3][11 - p_4] + v_4} \right) = \max(26, 40). && 4 \text{ sí que es pren} \\
 T[3][11] &= \max \left( T[2][11], \underline{T[2][11 - p_3] + v_3} \right) = \max(8, 18). && 3 \text{ sí que es pren} \\
 T[2][0] &= T[1][0] = 0. && 1 \text{ i } 2 \text{ no es prenen}
 \end{aligned}$$

# Motxilla 0/1: Una versió iterativa

## Una solució iterativa

```
1 vector <float> v; vector <unsigned> p;  
2 inline unsigned peso(unsigned i) {return p[i-1];}  
3 inline float    valor(unsigned i) {return v[i-1];}  
4  
5 float Motxilla (int n, unsigned P){  
6     float M[n+1][P+1];  
7  
8     for (unsigned i=1; i<=n;i++) M[i][0]=0; //sense espai el guany es 0  
9     for (unsigned j=0; j<=P;j++) M[0][j]=0; //sense objectes el guany es 0  
10  
11     for (unsigned i=1;i<=n;i++)  
12         for (unsigned j=1;j<=P;j++)  
13             if (peso(i)>j) // L'objecte i no cap en la seleccio actual  
14                 M[i][j]=M[i-1][j];  
15             else  
16                 if (M[i-1][j] >= M[i-1][j-peso(i)]+valor(i)  
17                     M[i][j]=M[i-1][j]; //cap, pero es considera no prendre'l  
18                 else  
19                     M[i][j]=M[i-1][j-peso(i)]+valor(i); //es considera prendre'l  
20     return M[n][P]; }
```

- Complexitat temporal

El cost temporal exacte,  $C_e^T(n, P)$  per a un problema  $(n, P)$  ve donat per:

$$C_e^T(n, P) = 1 + \sum_{i=1}^n 1 + \sum_{i=0}^P 1 + \sum_{i=1}^n \sum_{j=1}^P 1 = 1 + n + P + 1 + P(n + 1)$$

Per tant,

$$C_e^T(n, P) \in \Theta(nP)$$

- Complexitat espacial

$$C_e^E(n, P) \in \Theta(nP)$$

- Encara que es tracta d'una complexitat espacial millorable



# Motxilla 0/1: PD-recursiu

## Una solució recursiva

```
1 vector <float> v; vector <unsigned> p;  
2 inline unsigned peso(unsigned i) {return p[i-1];}  
3 inline float    valor(unsigned i) {return v[i-1];}  
4  
5 float **M;    //M[n+1][P+1]  
6 float MotxillaR_PD (unsigned i, unsigned P){  
7     float S1, S2;  
8     if (i==0) M[i][P]=0;  
9     else{  
10         if (peso(i) <= P){  
11             if (M[i-1][P-peso(i)]==kBuit)  
12                 M[i-1][P-peso(i)]= MotxillaR_PD(i-1,P-peso(i));  
13             S1= valor(i) + M[i-1][P-peso(i)];  
14         }  
15         else S1=0;  
16         if (M[i-1][P]==kBuit)  
17             M[i-1][P]==MotxillaR_PD(i-1,P);  
18         S2=M[i-1][P];  
19         M[i][P]=max(S1,S2);  
20     }  
21     return M[i][P];}
```

## Motxilla 0/1: PD-recursiu amb magatzem

- Exemple: Siguen  $n = 5$  objectes amb pesos ( $p_i$ ) i valors ( $v_i$ ) indicats en la taula. Siga  $P = 11$  el pes màxim de la motxilla.

[illegible]

- El 60% de les cel·les no s'han utilitzat per tant:

**El subproblema associat no ha estat resolt**  
**Estalvi computacional!**

40	Solució al problema
	Contorn o perfil
	Cel·les sense ús

$$\begin{array}{llll}
 T[5][11] & = \max \left( \underline{T[4][11]}, T[4][11 - p_5] + v_5 \right) & = \max(40, 36). & 5 \text{ no es pren} \\
 T[4][11] & = \max \left( T[3][11], \underline{T[3][11 - p_4] + v_4} \right) & = \max(26, 40). & 4 \text{ sí que es pren} \\
 T[3][5] & = \max \left( \underline{T[2][5]}, \underline{T[2][5 - p_3] + v_3} \right) & = \max(8, 18). & 3 \text{ sí que es pren} \\
 T[2][0] & = T[1][0] & = 0. & 1 \text{ i } 2 \text{ no es pren}
 \end{array}$$

# Motxilla 0/1: Conclusions

- La complexitat temporal de la solució obtinguda mitjançant programació dinàmica està en  $\Theta(nP)$ 
  - Un recorregut descendent a través de la taula permet obtenir també, en temps  $\Theta(n)$ , la seqüència òptima de decisions preses.
- Si  $P$  és molt gran, llavors la solució obtinguda mitjançant programació dinàmica no és bona
- Si els pesos  $p_i$  o la capacitat  $P$  pertanyen a dominis continus (p.ex. els reals) llavors aquesta solució no serveix
- La complexitat espacial de la solució obtinguda es pot reduir fins a  $\Theta(P)$
- En aquest problema, la solució PD-recursiva pot ser més eficient que la iterativa
  - Almenys, la versió que no realitza càlculs innecessaris és més fàcil d'obtenir en recursiu.





- Es pot reduir la complexitat espacial de la solució iterativa proposada?
  - Quants vectors hi farien falta i de quina grandària?
  - Sacrifica això la complexitat temporal?
- Escriu una funció per obtenir la seqüència de decisions òptima a partir de la taula completada per l'algorisme iteratiu
  - Quina complexitat temporal té aquesta funció?



# Anàlisi i disseny d'algorismes

## 5. Algorismes voraços

José Luis Verdú Mas, Jose Oncina, Mikel L. Forcada

Dept. Llenguatges i Sistemes Informàtics  
Universitat d'Alacant

28/01/2018 (Revisió: 389)



- 1 Presentació
- 2 Introducció
- 3 Eficiència
- 4 El disseny d'algorismes. Divideix i venceràs
- 5 Programació dinàmica
- 6 Algorismes voraços**
- 7 Tornada arrere
- 8 Ramificació i poda



# Algorismes voraços: exemple introductori (1)

- El problema de la motxilla (continu).<sup>13</sup> Siguen  $n$  objectes amb valors  $v_i$  i pesos  $p_i$  i una motxilla amb capacitat màxima de transport de pes  $P$ . Seleccionar un conjunt d'objectes de manera que:
  - no sobrepassi el pes  $P$  (restricció)
  - el valor transportat siga màxim (funció objectiu)
- El problema es redueix a:
  - Seleccionar un subconjunt dels objectes disponibles,
  - que complisca les restriccions, i
  - que maximitze la funció objectiu.
- Com s'ha de resoldre?
  - Selecció d'objectes a introduir en la motxilla (decisiones sobre cadascun dels objectes)
  - Es necessita un criteri que decidisca quin objecte seleccionar en cada moment (criteri de selecció)

---

<sup>13</sup> És a dir,  $x_i \in [0, 1]$

# Algorismes voraços: exemple introductori (2)



- Quin criteri seguim?

- Suposem l'exemple següent: 
$$n = 3 \begin{cases} P = 12 \\ p = (6, 5, 2) \\ v = (49, 40, 20) \end{cases}$$
- Criteris possibles:

Criteri	Solució	Pes $P$	Valor $V$
Per ordre descendent de valor	$(1, 1, \frac{1}{2})$	12	99
Per ordre ascendent de pes	$(\frac{5}{6}, 1, 1)$	12	100,8
Per ordre descendent de quoci- ent valor/pes <sup>14</sup> $(\frac{v_i}{p_i})$	$(1, \frac{4}{5}, 1)$	12	<b>101</b>

<sup>14</sup>Preu per quilo o *valor específic*

# Algorismes voraços: exemple introductori (3)



- Formalització del problema:

- Solució = vector de decisions  $X = \langle x_1, x_2, \dots, x_n \rangle$ ,  $x_i \in [0, 1]$  on  $x_i$  representa la decisió presa pel que fa a l'element  $i$ :
  - $x_i = 0$ : no se selecciona l'objecte  $i$
  - $0 < x_i < 1$ : fracció seleccionada de l'objecte  $i$
  - $x_i = 1$ : se selecciona l'objecte  $i$  complet
- Funció objectiu:  $\max \left( \sum_{i=1}^n x_i v_i \right)$  (valor transportat)
- Restricció:  $\sum_{i=1}^n x_i p_i \leq P$



# Algoritmes voraços: exemple introductori (4)

## Motxilla contínua: algorisme voraç

```
1 vector <zero_a_un> Motxilla_C (vector <float> v, vector <float> p,  
2                               float P, int n) {  
3     float P_acumulat;  
4     vector <zero_a_un> x;  
5     for (i=1;i<=n;i++) x[i]=0;  
6     ordena_elements_decreixentment_v_p(v,p);  
7     P_acumulat=0;  
8     while (i<=n && P_acumulat<P) {  
9       if (P_acumulat+p[i] <=P) { P_acumulat +=p[i]; x[i]=1; };  
10      else { x[i]=(P-P_acumulat)/p[i]; P_acumulat=P };  
11      i++; };  
12     return ;  
13 }
```

Costos: pas 5:  $O(n)$ ; pas 6:  $O(n \log(n))$ ; passos 8–11:  $O(n)$ .

Domina el pas 6:  $O(n \log(n))$ .



# Algorismes voraços: exemple introductori (5)

L'algorisme també pot escriure's recursivament:

## Motxilla contínua: algorisme voraç recursiu

```
1 vector <zero_a_un> Motxilla_CR(vector <float> v,  
2                               vector <float> p,  
3                               vector <zero_a_un> x,  
4                               float P, int n) {  
5     float P_acumulat;  
6     for (i=1;i<=n;i++) x[i]=0;  
7     ordena_elements_decreixent_v_p(v,p);  
8     return Motxilla_aux(v,p,n,P);  
9 }  
10 float Mochila_aux(vector <float> v,  
11                  vector <float> p,  
12                  vector <zero_a_un> x,  
13                  float P, int i) {  
14   x[i]=min(p[i],P)/p[i];  
15   if (x[i]==1 && i<n) return v[i]+Motxilla_aux(v,p,x,i+1,P-p[i]);  
16   else return x[i]*v[i];  
17 }
```



- Condueix sempre aquest criteri a l'obtenció de la solució òptima?
- **Teorema:** Si se seleccionen els objectes per ordre decreixent de  $v_i/p_i$  aleshores aquest algorisme troba la solució òptima.



## ● Demostració:

- Suposem els elements ordenats (decreixent  $v_i/p_i$ )
- Siga  $X = \langle x_1, x_2, \dots, x_n \rangle$  la solució obtinguda per l'algorisme
- Trivialment, si  $\forall i \in [1, n], x_i = 1$  la solució és òptima.
- En cas contrari: Suposem que  $j$  és l'índex tal que  $x_i = 1$  si  $i < j$  y  $x_i = 0$  si  $i > j$ .
- Siga  $Y = \langle y_1, y_2, \dots, y_n \rangle$  qualsevol solució factible
- Siga  $k$  la primera posició en la qual es diferencien  $X$  i  $Y$ .
- És fàcil demostrar que només podem tenir  $y_k < x_k$  i això només és possible si  $k \leq j$ .
  - Si  $k > j$  llavors  $x_k = 1$ ; per tant,  $y_k < x_k$ .
  - Si  $k = j$ , com  $x_i = 1$  para  $1 \leq i \leq k - 1$  i  $x_i = 0$  para  $k + 1 \leq i \leq n$ , només podem tenir  $y_k < x_k$  fent que alguns  $y_i > 0$  per a  $k + 1 \leq i \leq n$ .
  - Si  $k > j$ ,  $x_k = 0$  i no es pot augmentar sense augmentar el pes total.



- **Demostració** (cont.):

- D'altra banda,

$$V(X) - V(Y) = \sum_{i=k}^n (x_i - y_i) v_i = v_k(x_k - y_k) + \sum_{i=k+1}^n (x_i - y_i) v_i$$

- Quan canviem de  $X$  a  $Y$  la diferència de pes en  $k$ ,  $p_k(x_k - y_k)$  es trasllada als objectes següents.
- La millor opció, si és possible, és afegir tot aquest pes a l'element  $m = \max(k + 1, j)$ ; açò comporta fer que  $y_m$  siga  $x_m + \frac{p_k(x_k - y_k)}{\rho_m}$
- En aquest cas, clarament, el canvi de valor és  $V(X) - V(Y) = v_k(x_k - y_k) + v_l(x_m - (x_m + \frac{p_k}{\rho_m}(x_k - y_k))) = p_k(x_k - y_k)(\frac{v_k}{\rho_k} - \frac{v_m}{\rho_m}) \geq 0$ , ja que els tres factors són positius o zero.
- Podem concloure que no hi ha cap solució millor que  $X$  (intuïtivament, un repartiment del pes cap a elements de menor valor específic només pot baixar el valor total).



# Algorismes voraços: definició (1)

- Siga  $C$  un conjunt d'elements.
- **Problema d'optimització per selecció discreta:**<sup>15</sup> problema per al qual la solució consisteix a obtenir un subconjunt de  $C$  (que serà la solució al problema) tal que:
  - Satisfaga unes restriccions
  - Optimitze una certa funció objectiu
- **Solució factible:** Aquella que compleix les restriccions del problema
- **Solució òptima:** Solució factible que optimitza (maximitza o minimitza) la funció objectiu del problema.
- Els algorismes voraços implementen una **estratègia** de resolució d'aquest tipus de problemes

---

<sup>15</sup>L'algorisme de la motxilla contínua no pertany a aquest grup, ja que s'hi produeixen seleccions fraccionàries.



# Algorismes voraços: definició (2)

- Els algorismes voraços implementen una **estratègia** concreta per obtenir la solució òptima al problema
- Passos:
  - Es construeix la solució per etapes
  - En cada etapa:
    - Es tria un element  $i$  del conjunt  $C$  per incloure'l en el subconjunt solució.
    - L'element  $i$  serà aquell que produeix un òptim local per a aquesta etapa (segons l'estat actual de la solució i sense considerar decisions futures)
    - Es comprova si la solució segueix sent factible en afegir  $i$  a la solució. Si ho és, el candidat  $i$  s'inclou en la solució. Si no ho és, es descarta per sempre.
    - La decisió és irreversible. L'element  $i$  no es torna a reconsiderar mai més.
- El procés acabarà quan hàgem trobat la solució buscada o ens quedem sense elements per considerar



## Esquema voraç

```
1 t_conjuntElements VORAC( t_conjuntElements x, t_problema dp)
2 {
3     t_conjuntElements y, solucio;
4     element decisio;
5     y=prepara(x);
6     while(noBuit(y) || !esSolucio(solucio)) {
7         decisio=selecciona(y);
8         if (esFactible(decisio,solucio))
9             solucio=insereixElement(decisio,solucio);
10        y=esborraElement(decisio,y);
11    }
12    return solucio; /* a comprovar amb essolucio() */
13 }
```

On:

- element: Element del problema;
- dp: Dades del problema;
- esSolucio:  $t\_conjuntElements \rightarrow bool$ ;
- selecciona:  $t\_conjuntElements \rightarrow element$ ;
- esFactible:  $element \times t\_conjuntElements \rightarrow bool$
- prepara:  $t\_conjuntElements \rightarrow t\_conjuntElements$

# Algorismes voraços: característiques i àmbit d'aplicació (1)

- S'hi obtenen algorismes eficients i fàcils d'implementar
  - Costos normalment polinòmics i sovint  $O(n \log(n))$
- Per què no s'usa sempre?
  - No tots els problemes admeten aquesta estratègia.
  - La cerca d'òptims locals no condueix sempre a òptims globals.
  - Per aplicar-los amb garanties a un problema s'hi ha de trobar un criteri de selecció que trobe sempre la solució òptima al problema.
  - És a dir, hem de demostrar formalment que la funció escollida aconsegueix trobar òptims globals per a qualsevol entrada de l'algorisme.



# Algorismes voraços: característiques i àmbit d'aplicació (2)

- S'apliquen molt, encara que se sàpiga que no necessàriament troben la solució òptima:
  - quan és preferible una solució aproximada a temps a una òptima massa tard, o
  - quan es busca un equilibri entre eficiència (complexitat reduïda) i eficàcia (solució aproximada que depén del criteri de selecció).
- Però **alerta!!** En aquests casos:
  - Podem obtenir una solució no òptima a un problema que en té o fins i tot podem no trobar cap solució.
  - Per a cada problema particular, cal analitzar si es poden donar aquestes circumstàncies i sobretot, valorar com afectaran a la presa de decisions derivada de l'ús de l'algorisme.





# Algorismes voraços: exemples d'aplicació

- Problema de la motxilla discreta (sense fraccionament)
- Problema del canvi



# Algorismes voraços: problema de la motxilla discreta (sense fraccionament)

- Siguen  $n$  objectes amb valors  $v_i$  i pesos  $p_i$  i una motxilla amb capacitat màxima de transport (pes)  $P$ . Seleccionar un conjunt d'objectes de manera que:
  - no sobrepassi el pes  $P$
  - el valor transportat siga màxim
- Formulació del problema:
  - Expressarem la solució mitjançant un vector  $\langle x_1, x_2, \dots, x_n \rangle$  on  $x_i$  representa la decisió presa pel que fa a l'element  $i$ .
  - Funció objectiu:  $\max \left( \sum_{i=1}^n x_i v_i \right)$  (valor transportat)
  - Restriccions
    - $\sum_{i=1}^n x_i p_i \leq P$
    - $x_i \in \{0, 1\} \begin{cases} x_i = 0 & \text{no se selecciona l'objecte } i \\ x_i = 1 & \text{sí que se selecciona l'objecte } i \end{cases}$



# Algorismes voraços: problema de la motxilla discreta (sense fraccionament)

- Quan no es permet fraccionar objectes, el mètode voraç no resol el problema.
- No hi ha cap criteri voraç (conegut) de selecció que conduïska sempre a la solució òptima. Exemple:

$$\bullet \left\{ \begin{array}{l} P = 120 \\ n = 3 \left\{ \begin{array}{l} p = (60, 60, 20) \\ v = (300, 300, 200) \end{array} \right. \end{array} \right.$$

- $v/p = (5, 5, 10)$
- solució voraç:  $\langle 0, 1, 1 \rangle \rightarrow \text{valor total} = 500$
- $X_{\text{opt}} = \langle 1, 1, 0 \rangle \rightarrow \text{valor total} = 600$
- Aquest contraexemple ens serveix per demostrar que el criteri voraç, òptim per a la motxilla amb fraccionament, no és òptim en aquest cas.



# Algorismes voraços: el problema del canvi (1)

- Consisteix a formar una suma  $M$  amb el nombre mínim de monedes preses (amb repetició) d'un conjunt  $C$ :
  - La solució voraç és una seqüència de decisions (selecció de monedes)  $S = \langle s_1, s_2, \dots, s_n \rangle$
  - La funció objectiu és  $\min |S| = \min n$
  - La restricció és  $\sum_{i=1}^n \text{valor}(s_i) = M$ .
- Obtenció de la solució: Atés que es tracta de minimitzar el nombre de monedes, se n'haurà de prendre a cada moment la de major valor possible (sempre que se satisfacen les restriccions).



# Algoritmes voraços: el problema del canvi (2)

- Consisteix a formar una suma  $M$  amb el nombre mínim de monedes preses (amb repetició) d'un conjunt  $C$ :
- Siga  $M = 65$

$C$	$S$	$n$	Solució
$\{1, 5, 25, 50\}$	$\langle 50, 5, 5, 5 \rangle$	4	òptima <sup>16</sup>
$\{1, 5, 7, 25, 50\}$	$\langle 50, 7, 7, 1 \rangle$	4	òptima
	$\langle 50, 5, 5, 5 \rangle$	4	no voraç
$\{1, 5, 11, 25, 50\}$	$\langle 50, 11, 1, 1, 1, 1 \rangle$	6	factible però no òptima
$\{5, 11, 25, 50\}$	$\langle 50, 11, ? \rangle$	???	el voraç no en troba

<sup>16</sup>No és gens difícil escriure una solució òptima recursiva amb memoïtzació:  
 $n_{\text{opt}}(M) = 1 + \min_{1 \leq i \leq |C|} n_{\text{opt}}(M - c_i); n_{\text{opt}}(0) = 0; n_{\text{opt}}(x) = \infty$  per a  $x \leq 0$ .



- 1 El fontaner diligent
- 2 Càlcul de l'arbre de recobriment de cost mínim (algorismes de Prim i Kruskal)
- 3 L'assignació de tasques



## Exercici 1: El fontaner diligent

- Un fontaner ha de fer  $n$  reparacions urgents, i sap per endavant el temps que necessitarà per a cadascuna d'elles: en la tasca  $i$ -ésima tardarà  $t_i$  minuts. Com que en la seua empresa li paguen d'acord amb la satisfacció del client, necessita decidir l'ordre en el qual atindrà els avisos per minimitzar el temps mitjà d'espera dels clients.
- En altres paraules, si anomenem  $T_i$  el temps que espera el client  $i$ -ésim fins que veu reparada la seua avaria per complet, necessita minimitzar l'expressió:

$$E_n = \sum_{i=1}^n T_i$$



## Exercici 2: Arbre de recobriment de cost mínim

- Partim d'un graf connex, ponderat i no dirigit  $G = (V, A, p)$  d'arcs no negatius, i volem trobar l'arbre de recobriment de  $G$  de cost mínim.
- Per arbre de recobriment d'un graf  $G$  entenem un subgraf sense cicles que continga tots els seus vèrtexs. En cas d'haver-hi diversos arbres de cost mínim, ens quedarem d'entre ells amb el que posseïska menys arcs.
- Hi ha almenys dos algorismes molt coneguts que resolen aquest problema, com són el de Prim i el de Kruskal. En tots dos es va construir l'arbre per etapes.
- La manera en què es realitza aquesta elecció és el que distingeix un algorisme d'altre.
- Es pot usar per determinar com utilitzar el mínim de conductor per connectar  $|V|$  punts elèctricament equivalents en un circuit integrat (el pes de cada arc és la distància entre punts).





## Exercici 3: L'assignació de tasques

- Suposem que disposem de  $n$  treballadors i  $n$  tasques. Siga  $b_{ij} > 0$  el cost d'assignar el treball  $j$  al treballador  $i$ . Una assignació de tasques pot ser expressada com una assignació dels valors 0 o 1 a les variables  $x_{ij}$ , on  $x_{ij} = 0$  significa que al treballador  $i$  no li han assignat la tasca  $j$ , i  $x_{ij} = 1$  indica que sí.
- Una assignació vàlida és aquella en què a cada treballador només li correspon una tasca i cada tasca està assignada a un treballador.
- Donada una assignació vàlida, definim el cost d'aquesta assignació com:

$$\sum_{i=1}^n \sum_{j=1}^n x_{ij} b_{ij}$$

- Direm que una assignació és òptima si és de cost mínim.



# Anàlisi i disseny d'algorismes

## 6. Tornada arrere

José Luis Verdú Mas, Jose Oncina, Mikel L. Forcada

Dept. Llenguatges i Sistemes Informàtics  
Universitat d'Alacant

28/01/2018 (Revisió: 389)



- 1 Presentació
- 2 Introducció
- 3 Eficiència
- 4 El disseny d'algorismes. Divideix i venceràs
- 5 Programació dinàmica
- 6 Algorismes voraços
- 7 Tornada arrere**
- 8 Ramificació i poda



## El problema de la motxilla (general)

Donats:

- $n$  objectes amb valors  $v_i$  i pesos  $p_i$
- una motxilla que transporta un pes màxim  $P$

Seleccionar un conjunt d'objectes de manera que:

- no se sobrepassi el pes límit  $P$  (restricció)
- el valor transportat siga màxim (funció objectiu)

### • Com obtenir la solució òptima?

- Objectes no fragmentables i pesos discrets: programació dinàmica.
- Objectes fragmentables: algorismes voraços

### • Com resolem el problema si no podem fragmentar els objectes i els pesos són valors reals?<sup>17</sup>

<sup>17</sup>A més d'usant "programació dinàmica recursiva" (memoïtzació)

# Exemple introductor

## Formalització del problema:

- Solució:  $X = (x_1, x_2, \dots, x_n)$   $x_i \in \{0, 1\}$
- Restriccions:
  - Implícites:

$$x_i \in \{0, 1\} \begin{cases} 0 & \text{no se selecciona l'objecte } i \\ 1 & \text{se selecciona l'objecte } i \end{cases}$$

- Explícites:

$$\sum_{i=1}^n x_i p_i \leq P$$

- Funció objectiu:

$$\max \sum_{i=1}^n x_i v_i$$



# Exemple introductori

- Suposem l'exemple següent :

$$P = 16 \qquad p = (7, 8, 2) \qquad v = (49, 40, 20)$$

- Combinacions possibles (espai de solucions):

**Solució pes valor**

(0, 0, 0)    0    0

(0, 0, 1)    2    20

(0, 1, 0)    8    40

(0, 1, 1)    10    60

(1, 0, 0)    7    49

(1, 0, 1)    9    69

(1, 1, 0)    15    89

(1, 1, 1)    17    109

Solucions factibles

Solució voraç

Solució òptima

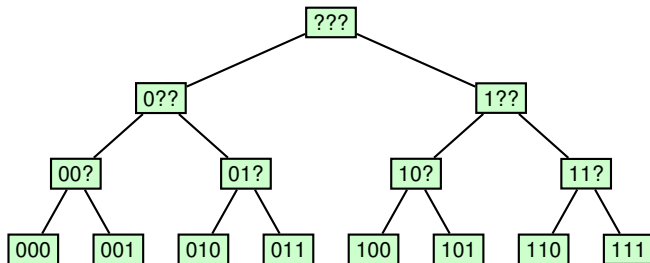
Solució NO factible



# Exemple introductorí

- Generació de totes les combinacions ( $T(n) = n2^n$ )

Exemple  $n = 3$ :  $X = (x_1, x_2, x_3)$   $x_i \in \{0, 1\}$



Totes les combinacions.

Crida inicial: combinacions(0,x)

```
void combinacions(unsigned k, vector <short> &x){  
    for (unsigned j=0; j<2; j++) {  
        x[k]=j; //Nova alternativa per a aquesta component de X  
        if (k==x.size()-1) imprimir_fulla_de_l_arbre(x);  
        else combinacions(k+1, x); //No es fulla: descendir en l'arbre  
    }  
}
```

# Exemple introductori

- **Generació de totes les solucions factibles**

és a dir,  $\{X = (x_1, x_2 \dots x_n), x_i \in \{0, 1\}, \sum_{i=1}^n x_i p_i \leq P\}$

Totes les solucions factibles.

Crida inicial: factibles(p, P, 0, x)

```
inline float pes(vector <float> &p, unsigned k, vector <short> &x){  
    float _pes=0;  
    for (unsigned i=0;i<=k;i++) _pes+=x[i]*p[i];  
    return _pes;  
}  
  
void factibles(vector <float> &p, float P, unsigned k, vector <short> &x){  
    for (unsigned j=0; j<2; j++) {  
        x[k]=j;  
        if (pes(p,k,x)<=P)  
            if (k==x.size()-1) imprimir_sol_factible(x);  
            else factibles(p,P,k+1,x);  
    }  
}
```



# Exemple introductorí

- Complexitat temporal de la funció “combinacions”
  - **Cost exacte**  $C_i^T(n)$ :  $f(n) = 2f(n-1) + 1$ ;  $f(n) \in \Theta(2^n)$
- Complexitat temporal de la funció “factibles”
  - **Fita superior**  $C_p^T(n)$ : Tots els objectes caben en la motxilla.  
 $f(n) = 2f(n-1) + n$ ;  $f(n) \in O(2^n)$
  - **Fita inferior**  $C_m^T(n)$ : Cap objecte cap en la motxilla  
( $p_i > P \forall i \in [1, n]$ ).  $f(n) = f(n-1) + n$ ;  $f(n) \in \Omega(n^2)$
  - Encara que el recorregut lineal que comprova la restricció del pes es pot evitar:

Totes les solucions factibles.

Crida inicial: float p\_a=0; factibles(p, P, 0, x, p\_a)

```
void factibles(vector<float> &p, float P, unsigned k,
               vector<short> &x, float &pes_acum){
    for (unsigned j=0; j<2; j++) {
        x[k]=j; pes_acum+=x[k]*p[k];
        if (pes_acum<=P)
            if (k==x.size()-1) imprimir_sol_factible(x);
            else factibles(p,P,k+1,x);
        pes_acum-=x[k]*p[k];
    } }
```

# Exemple introductorí

## ● Recerca d'una solució òptima

Solució òptima.

Crida inicial: float vMillor=-1; optima\_v0(v, p, P, 0, x, xMillor, vMillor)

```
inline float valor(vector <float> & v, unsigned k, vector <short> & x){  
    float _valor=0;  
    for (unsigned i=0; i<=k; i++) _valor+=x[i]*v[i];  
    return _valor;  
}
```

```
void optima_v0(vector <float> & v, vector <float> & p, float P,  
               unsigned k, vector <short> & x,  
               vector <short> & xMillor, float & vMillor){  
    cridesRecursives++; //per a comprovacions d'eficiència  
    for (unsigned j=0; j<2; j++) {  
        x[k]=j;  
        if (pes(p,k,x)<=P)  
            if (k==x.size()-1) {  
                if (valor(v,k,x)>vMillor){  
                    xMillor=x; vMillor=valor(v,k,x); }  
            }  
        else optima_v0(v,p,P,k+1,x,xMillor,vMillor);  
    } } }
```

# Exemple introductori

- **Recerca d'una solució òptima amb poda basada en la millor solució en curs**

**Es pot millorar l'eficiència de la funció anterior?**

Sí. Evitant (podant) l'exploració de vectors factibles que no van a millorar la millor solució que es té fins al moment.

- Són els anomenats *vectors no prometedors*

Per exemple, qualsevol vector incomplet  $(x_0, x_1 \cdots x_k, ?, ?, \dots, ?)$  es pot descartar si no compleix la condició:

$$\text{valor}(v, k, x) + \sum_{i=k+1}^{n-1} v_i > v_{\text{Millor}}$$

on  $\sum_{i=k+1}^{n-1} v_i$  és la màxima millora possible (carregant tots els objectes fins i tot sense complir la restricció de pes).



# Exemple introductori

- Recerca d'una solució òptima amb poda basada en la millor solució en curs

Solució òptima v1.      Crida inicial: float vMillor=-1; optima\_v1(v, p, P, 0, x, xMillor, vMillor)

```
inline float suma_valors(vector <float> & v, unsigned k){
    float _valor=0;
    for (unsigned i=k; i<v.size(); i++) _valor+=v[i];
    return _valor;

void optima_v1(vector <float> & v, vector <float> & p, float P,
               unsigned k, vector <short> & x,
               vector <short> & xMillor, float & vMillor){
    cridesRecursives++; //per a comprovacions d'eficiencia
    for (unsigned j=0; j<2; j++) {
        x[k]=j;
        if (pes(p,k,x)<=P)
            if (k==x.size()-1) {
                if (valor(v,k,x)>vMillor){
                    xMillor=x; vMillor=valor(v,k,x); }
            }
        else if (valor(v,k,x) + suma_valors(v,k+1) > vMillor)
            optima_v1(v,p,P,k+1,x,xMillor,vMillor);
    }
}
```

# Exemple introductori

**Interessa que els mecanismes de poda “actuen” com més amunt millor possible en l'arbre, és a dir, com més incomplet estiga el vector**

- En aquest sentit, la poda anterior es pot millorar mitjançant una estimació més realista (menys optimista):
- Tenint en compte que el major benefici que es pot obtenir ve donat per la solució al problema de la motxilla contínua (amb solució voraç), es descarta tot vector incomplet que no complisca:

$$\text{valor}(v, k, x) + \text{motxilla\_continua}(v', p', P', k + 1) > v_{\text{Millor}}$$

- on  $\text{motxilla\_continua}(v', p', P', k + 1)$  és la solució al problema de la motxilla amb fraccionament considerant que els  $k$  primers objectes ja han estat tractats.
- Es tracta d'una condició més restrictiva i per tant amb una capacitat de poda més gran.

# Exemple introductorí

- Recerca d'una solució òptima amb poda basada en la millor solució en curs

Solució òptima v2.      Crida inicial: float vMillor=-1; optima\_v2(v, p, P, 0, x, xMillor, vMillor)

```
float motxilla_continua(vector<float> v, vector<float> p, float P, int j);

void optima_v2(vector<float> &v, vector<float> &p, float P,
               unsigned k, vector<short> &x,
               vector<short> &xMillor, float &vMillor){
    cridesRecursives++; //per a comprovacions de eficiencia
    for (unsigned j=0; j<2; j++) {
        x[k]=j;
        if (pes(p,k,x)<=P)
            if (k==x.size()-1) {
                if (valor(v,k,x)>vMillor){
                    xMillor=x; vMillor=valor(v,k,x); }
            }
        else if (valor(v,k,x) +
                motxilla_continua(v,p,P-pes(p,k,x),k+1) > vMillor)
            optima_v2(v,p,P,k+1,x,xMillor,vMillor);
    }
}
```

# Exemple introductorí

- Recerca d'una solució òptima amb poda basada en la millor solució en curs

Sol. òptima v2. Iteratiu Llam. inicial: float vMillor=-1; optima\_v2l(v, p, P, 0, x, xMillor, vMillor)

```
void optima_v2l(vector <float> & v, vector <float> & p, float P,
               unsigned k, vector <short> & x,
               vector <short> & xMillor, float & vMillor){
    x[k]=-1;
    while (k>-1){
        while (x[k]<1){
            x[k]++;
            if (pes(p,k,x)<=P)
                if (k==x.size()-1) {
                    if (valor(v,k,x)>vMillor){ xMillor=x; vMillor=valor(v,k,x); }
                }
            else if (valor(v,k,x) +
                    motxilla_continua(v,p,P-pes(p,k,x),k+1) > vMillor){
                k++; //avancar en el vector: seg. nivell en l'arbre
                x[k]=-1;
            }
        }
        k--; //no queden mes possibilitats: retrocedir al nivell superior.
    } }
```

# Exemple introductorí

- Els mecanismes de poda basats en la millor solució en curs són molt més eficients si s'arriba amb promptitud a una “bona” solució
  - En aquest sentit, la forma en la qual es “desplega l'arbre” pot ser rellevant:
    - Per exemple, per a aquest problema: completar el vector primer amb els uns i després amb els zeros
  - Però el que, en general, augmenta dràsticament l'eficiència és partir d'una solució factible:
    - Com més s'aproxime a la solució òptima millor
    - Encara que no es tracta d'una fita optimista en el sentit estudiat anteriorment (ja que és imprescindible que siga factible)
    - Per exemple, per a aquest problema: es pot partir de la solució (subòptima) que aporta el mètode voraç a la motxilla discreta

## Solució òptima v2 partint d'un subòptim.

```
vMillor=motxilla_discreta_vorac(v,p,P,xMillor); //Heuristic vorac  
optima_v2(v,p,P,0,x,xMillor,vMillor);
```



# Exemple introductori

- Anàlisi d'eficiència: 25 mostres aleatòries de grandària  $n = 30$ .

Tipus de poda <sup>18</sup>	Partint d'un subòptim voraç <sup>19</sup>	Crides recursives realitzades	Temps mitjà (segons)
Cap (v0)	–	$1054,8 \times 10^6$	875,65
Completant amb tots els objectes restants (v1)	No	$925,5 \times 10^3$	0,112
	Sí	$389,0 \times 10^3$	0,072
Completant segons la sol. voraç motxilla contínua (v2)	No	$2,3 \times 10^3$	0,034
	Sí <sup>20</sup>	18	0,002

<sup>18</sup>Estimant el benefici que s'obtindrà amb la resta del vector (poda basada en la millor solució en curs)

<sup>19</sup>Heurístic voraç: selecció d'objectes de major valor específic. Objectes no fraccionables.

<sup>20</sup>En la pràctica, resulta elevada la probabilitat que el subòptim obtingut mitjançant l'heurístic voraç coincidisca amb l'òptim global

# Tornada arrere: definició i àmbit d'aplicació

- Alguns problemes només els podem resoldre mitjançant l'obtenció i l'estudi exhaustiu del conjunt de possibles solucions al problema.
- D'entre totes elles, se'n podrà seleccionar un subconjunt o bé, aquella que considerem la millor (la solució òptima)
- Per dur a terme l'estudi exhaustiu, la **tornada arrere** proporciona una forma sistemàtica de generar totes les possibles solucions a un problema.
- Generalment s'usa en la resolució de problemes de selecció / optimització en els quals el conjunt de solucions possibles és finit. . .
- . . . en els quals es pretén trobar una o diverses solucions que siguin:
  - Factibles: que satisfacen unes restriccions i/o
  - Òptimes: optimitzen una certa funció objectiu



# Tornada arrere: característiques I

- La solució ha de poder expressar-se mitjançant un vector de decisions:  $(x_1, x_2, \dots, x_n) \quad x_i \in D_i$ 
    - Les decisions poden pertànyer a dominis diferents però aquests dominis sempre seran discrets o discretizables.
  - És possible que s'haja d'explorar tot l'espai de solucions.
    - Els mecanismes de poda van dirigits a disminuir la probabilitat que això ocórrega.
  - L'estratègia pot proporcionar:
    - Una solució factible,
    - totes les solucions factibles,
    - la solució òptima al problema.
- ... a costa, en la majoria dels casos, de complexitats prohibitives.



# Tornada arrere: característiques II

- La generació i la recerca de la solució es realitza mitjançant un sistema de prova i error:
  - Siga  $(x_1, x_2, \dots, x_{i-1}, \dots)$  un vector per completar.
  - Decidim sobre la component  $x_i$ :
    - Si la decisió compleix les restriccions s'afegeix  $x_i$  a la solució i s'avança a la següent component  $x_{i+1}$
    - Si no compleix les restriccions es prova una altra possibilitat per a  $x_i$
    - També es prova una altra possibilitat per a  $x_i$  quan torna de  $x_{i+1}$
    - Si no hi ha més possibilitats per a  $x_i$  es retrocedeix a  $x_{i-1}$  per provar una altra possibilitat amb aquesta component (pel que el procés comença de nou).
  - Al final, i si cap mecanisme de poda ho impedeix, s'haurà explorat tot l'espai de solucions.
- Es tracta d'un recorregut en preordre sobre una estructura arbòria imaginària.



# Tornada: Esquema general recursiu

Esquema recursiu de tornada arrere.

Crida inicial: backtracking\_R(P,0,x)

```
void backtracking_R (problema P; int k; vector & x){  
  
    x[k]= preparar_recorregut_nivell_k;  
    while (existeix_germa_nivell_k){  
        x[k]=seguent_germa_nivell_k;  
        if (factible(x,k))  
            if (completada(x)) tractar(x) //la millor, totes, una qualsevol, etc.  
            else [if (prometedora(x,k)) /*sols si es busca un optim*/  
                backtracking_R(P,k+1,x)  
            ]  
    }  
}
```



# Tornada arrere: Esquema general iteratiu

## Esquema iteratiu de *backtracking*.

Crida: `bactracking_l(P,0,x)`

```
void bactracking_l (problema P; int k; vector & x){  
  
  x[k]= preparar_recorregut_nivell_k;  
  while (k>-1){  
    while (existeix_germa_nivell_k){  
      x[k]=seguent_germa_nivell_k;  
      if (factible(x,k))  
        if (completada(x)) tractar(x) //la millor, totes, la primera...  
        else [if (prometedora(x,k)) /*nomes si es busca un optim*/]{  
          k++; //avancem  
          x[k]= preparar_recorregut_nivell_k;  
        }  
      }  
    }  
    k--; //retroces  
  }  
}
```



## Voraç

- Busquem la solució òptima i existeix un criteri de decisió voraç que ens hi condueix, o bé
- busquem una aproximació a la solució òptima (un subòptim).

## Tornada arrere

- Busquem totes les solucions factibles o totes les solucions òptimes
- quan el problema no té una solució voraç.



- El viatjant de comerç (*the travelling salesman problem*)
- Permutacions dels elements d'una llista
- El problema de les 8 reines
- La funció composta mínima





# Tornada arrere. Exercicis

## El viatjant de comerç

Donat un graf ponderat  $g = (V, A)$  amb pesos no negatius, el problema consisteix a trobar un *cicle hamiltonià* de cost mínim.

- Un *cicle hamiltonià* és un recorregut en el graf que recorre tots els vèrtexs només una vegada i torna al de partida.
- El cost d'un cicle està determinat per la suma dels pesos de les arestes que el componen.



# Tornada arrere. Exercicis

## El viatjant de comerç

### Solució:

- L'expressarem mitjançant un vector  $X = (x_1, x_2, \dots, x_n)$  on  $x_i \in \{1, 2, \dots, n\}$  és el vèrtex visitat en  $i$ -èsim lloc.
  - Assumim que els vèrtexs estan numerats,  
 $V = \{1, 2, \dots, n\}$ ,  $n = |V|$
  - Fixem el vèrtex de partida (per evitar rotacions):
    - $x_1 = 1$ ;  $x_i \in \{2, 3, \dots, n\} \quad \forall i : 2 \leq i \leq n$
- Restriccions
  - No es pot visitar dues vegades el mateix vèrtex:  
 $i \neq j \rightarrow x_i \neq x_j \quad \forall i, j : 1 \leq i \leq n \quad 1 \leq j \leq n$
  - Existència d'aresta:  $\forall i : 1 \leq i < n, \text{pes}(g, x_i, x_{i+1}) \neq \infty$
  - Existència d'aresta que tanca el camí:  $\text{pes}(g, x_n, x_1) \neq \infty$
- Funció objectiu:

$$\min \sum_{i=1}^{n-1} \text{pes}(g, x_i, x_{i+1}) + \text{pes}(g, x_1, x_n)$$



# Tornada arrere. Exercicis

## El viatjant de comerç

### Viatjant de comerç - v0

```
void viatjant(graf g){
    unsigned n=g.V.size();
    vector <short> xMillor, x(n);
    x[0]=1; //el primer vertex queda fixat
    unsigned k=1; float vMillor=INT_MAX; x[k]=1;
    while (k>0){
        while (x[k]<n){
            x[k]++;
            if (hihaAresta(g,x[k-1],x[k]) && !repetit(x,k))
                if (k==n-1){
                    if (hihaAresta(g,x[k],x[0])&& pes_cami(x,k)<vMillor){
                        vMillor=pes_cami(x,k); xMillor=x; }
                }
            else{ k++; x[k]=1;}
        }
        k--;
    }
    if (vMillor==INT_MAX) cout << "No es pot resoldre" << endl;
    else {cout << "Solucio: "; mostra(xMillor);
}
```

# Tornada arrere. Exercicis

## El viatjant de comerç

### Exercici:

- La solució algorísmica proposada resulta inviable perquè té una complexitat prohibitiva:  $O(n^n)$
- Per això, i per accelerar la recerca, es demana:
  - Aplicar algun mecanisme de poda basat en la millor solució fins al moment (per exemple, començar amb la solució voraç)
  - Dissenyar algun heurístic voraç que permeti complir l'objectiu
    - Suggestiment: Utilitzar les idees dels algorismes de Prim o de Kruskal



# Tornada arrere. Exercicis

## Permutacions

Donada una llista  $L$  de  $n$  elements qualssevol, cal escriure un algorisme que mostre totes les permutacions dels seus elements.

- Solució:
  - Siga  $X = (x_1, x_2, \dots, x_n)$   $x_i \in \{1, 2, \dots, n\}$  un vector que indica quin element de  $L$ , identificat per la seua posició  $x_i$ , s'ha de mostrar en  $i$ -ésim lloc.
  - Per tant, cada permutació estarà representada per cadascun dels vectors diferents  $X$  que es puguin obtenir.
  - Restricció:  $X$  no pot tenir elements repetits:
    - $i \neq j \rightarrow x_i \neq x_j \quad \forall i, j : 1 \leq i \leq n \quad 1 \leq j \leq n$
  - No hi ha funció objectiu: es busquen totes les combinacions factibles.



# Tornada arrere. Exercicis

## Permutacions

### Permutacions (v0,iteratiu)

### Restriccions amb cost lineal

```
inline bool repetit(vector<unsigned> &x, int k){
    unsigned i=0;
    while (x[i]!=x[k]) i++;
    return i!=k; }

void permuta_v0(lista<T> &L){
    unsigned n=L.size(); vector<unsigned> x(n); int k=0;

    x[k]=-1; //assumim que el primer element en L ocupa la posicio 0
    while (k>-1){
        while (x[k]<n-1){
            x[k]++;
            if (!repetit(x,k)){
                if (k==n-1) mostrar_permutacio(L,x);
                else {k++; x[k]=-1;}
            }
        }
        k--;
    } }
```

# Tornada arrere. Exercicis

## Permutacions

- Restriccions amb cost constant

- Assumint aquest cost en l'operador [] dels vectors "x" i "jaUsada"

### Permutacions (v1, iteratiu)

### Restriccions amb cost constant









```
void permuta_v1(lista <T> &L){
    unsigned n=L.size();  vector <unsigned> x(n);  int k=0;
    vector <bool> jaUsada(n, false); //n comp. inicial. amb "false"
    x[k]=-1; //assumim que el primer element en L ocupa la posicio 0
    while (k>-1){
        while (x[k]<n-1){
            x[k]++;
            if (!jaUsada[x[k]]){
                jaUsada[x[k]]=true;
                if (k==n-1) mostra_permutacio(L,x);
                else { k++; x[k]=-1; }
                jaUsada[x[k]]=false;
            }
        }
        k--;
    }
}
```

# Tornada arrere. Exercicis

## El problema de les 8 reines

En un tauler d'escacs ( $8 \times 8$ ) obtenir totes les formes de col·locar 8 reines de manera que no s'ataquen mútuament (ni en la mateixa fila, ni columna, ni diagonal).<sup>21</sup>

- Ejemplo:

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

<sup>21</sup>De forma trivial es pot generalitzar a  $n$  reines i un tauler  $n \times n$ .



# Tornada arrere. Exercicis

## El problema de les 8 reines

### Solució:

- Com que no pot haver-hi dues reines en la mateixa fila, la reina  $i$  es col·locarà en la fila  $i$ :
  - El problema, aleshores, és determinar la columna en la qual l'hem de col·locar.
- Siga  $X = (x_1, x_2, \dots, x_n)$  on  $x_i \in \{1, 2, \dots, n\}$  representa la columna en la qual es col·loca la reina de la fila  $i$
- Restriccions:
  - 1 No pot haver-hi dues reines en la mateixa fila:
    - implícit en la forma de representar la solució.
  - 2 No pot haver-hi dues reines en la mateixa columna, és a dir,  $X$  no pot tenir elements repetits:
    - $i \neq j \rightarrow x_i \neq x_j \quad \forall i, j: 1 \leq i \leq n \quad 1 \leq j \leq n$
  - 3 No pot haver-hi dues reines en la mateixa diagonal:
    - $i \neq j \rightarrow |i - j| \neq |x_i - x_j| \quad \forall i, j: 1 \leq i \leq n \quad 1 \leq j \leq n$



# Tornada arrere. Exercicis

## El problema de les 8 reines

8\_reines (recursiu)

Crida inicial: 8\_reines(0,x)

```
inline bool factible(vector<unsigned> &x, int k){
    bool _factible=true; short i=0;
    while (i<k && _factible){
        _factible= (x[i]!=x[k] && (k-i)!= abs(x[k]-x[i]));
        i++;
    }
    return _factible;
}

void 8_reines(short k, vector<short> &x){
    x[k]=-1;
    while (x[k]<7){
        x[k]++;
        if (factible(x,k))
            if (k==7) mostrar_solucio(x);
            else 8_reines(k+1,x)
    }
}
```

# Tornada arrere. Exercicis

## La funció composta mínima

Donades dues funcions  $f(x)$  i  $g(x)$  i donats dos nombres qualssevol  $x$  i  $y$ , trobar la funció composta mínima que obté el valor  $y$  a partir de  $x$  després d'aplicacions successives i indistintes de  $f(x)$  i  $g(x)$

- Exemple: Siguen  $f(x) = 3x$ ,  $g(x) = \lfloor x/2 \rfloor$ , i siguen  $x = 3$ ,  $y = 6$ 
  - Una funció composta que transforma 3 en 6 amb operacions  $f$  i  $g$  és: (consta de 6 aplicacions)

$$(g \circ f \circ g \circ f \circ f \circ g)(3) = 6$$

- Però la mínima, formada per quatre aplicacions, és: (encara que no és l'única)

$$(f \circ g \circ g \circ f)(3) = 6$$



# Tornada arrere. Exercicis

## La funció composta mínima

Solució:

- $X = (x_1, x_2, \dots, x_k) \quad x_i \in \{0, 1\} \begin{cases} 0 \equiv \text{s'hi aplica } f(x) \\ 1 \equiv \text{s'hi aplica } g(x) \end{cases}$ 
  - $(x_1, x_2, \dots, x_k) \equiv (x_k \circ \dots \circ x_2 \circ x_1)$
  - La grandària de la seqüència (del vector) no es coneix a priori (a diferència dels exemples anteriors)
    - Es pretén minimitzar la grandària de la seqüència solució (funció objectiu)
    - Per evitar branques infinites en l'arbre de recerca, assumirem un màxim de composicions  $M$
  - Siga  $F(X, k, x)$  el resultat d'aplicar al valor  $x$  la composició representada en la tupla  $X$  fins a la seua posició  $k$ , és a dir,

$$F(X, k, x) = (x_k \circ \dots \circ x_2 \circ x_1)(x) = x_k(\dots x_2(x_1(x)) \dots)$$

- Restriccions:
  - $F(X, k, x) \neq F(X, i, x) \quad \forall i < k$ , per evitar cicles
  - $k < M$ , per evitar recerques infinites
  - $F(X, k, x) \neq 0$



# Tornada arrere. Exercicis

## La funció composta mínima

```
void composicio(unsigned x, unsigned y, unsigned M){
    vector <short> XMillor, X(M+1);
    vector <unsigned> valor(M+1); //Guarda el resultat de la composicio
    short k=0, vMillor=M+1;

    X[k]=2;
    while (k>-1){
        while (X[k]>0){
            X[k]--;
            valor[k]=F(X,k,x);
            if (valor[k]==y) { if (k<vMillor){ vMillor=k; XMillor=X; } }
            else if (k<vMillor && // Sequencia prometedora
                    k<M && valor[k]!=0 && !repetit(valor,k)){
                k++; X[k]=2;
            }
        }
        k--;
    }
    if (vMillor==M+1) cout<<"Cap solucio amb " << M <<" composicions.";
    else { cout << "Solucio trobada:" << endl;
          cout << "Nombre de composicions:" << vMillor << endl;
        }
}
```

## 1 **L'acoloriment de grafs** (l'acolorit de mapes)

Donat un graf  $G$ , trobar el nombre mínim de colors amb què es poden acolorir els seus vèrtexs de manera que no hi haja dos vèrtexs adjacents amb el mateix color.

## 2 **El recorregut del cavall d'escacs**

Trobar una seqüència de moviments “legals” d'un cavall d'escacs de manera que pugui visitar les 64 caselles del tauler d'escacs sense repetir-ne cap.

## 3 **El laberint amb quatre moviments**

Es disposa de una quadrícula  $n \times m$  de valors  $\{0, 1\}$  que representa un laberint. Un valor 0 en una casella qualsevol de la quadrícula indica una posició inaccessible; en canvi, amb el valor 1 s'indiquen les caselles accessibles. Cal trobar un camí que permeti anar de la posició  $(1, 1)$  a la posició  $(n \times m)$  amb quatre tipus de moviment (amunt, avall, dreta, esquerra).



## 4 L'assignació de tasques

- Suposem que disposem de  $n$  treballadors i  $n$  tasques. Siga  $b_{ij} > 0$  el cost de assignar el treball  $j$  al treballador  $i$ . Una assignació de tasques es pot expressar com una assignació dels valors 0 o 1 a les variables  $x_{ij}$ , on  $x_{ij} = 0$  indica que al treballador  $i$  no li han assignat la tasca  $j$ , i  $x_{ij} = 1$  indica que sí.
- Una assignació vàlida és aquella on a cada treballador només li correspon una tasca i cada tasca està assignada a un treballador.
- Donada una assignació vàlida, definim el cost d'aquesta assignació com:

$$\sum_{i=1}^n \sum_{j=1}^n x_{ij} b_{ij}$$

- Cal trobar una assignació òptima, és a dir, de cost mínim.



## 5 El repartiment de paquets

- Una empresa de transports disposa de  $M$  vehicles per a repartir  $N$  paquets, tots a la mateixa destinació. Cada paquet  $i$  té un pes  $P_i$  i ha d'estar entregat abans d'un temps  $TP_i$ . D'altra banda, cada vehicle  $j$  pot transportar una càrrega màxima  $C_j$ , tarda un temps  $TV_j$  a aplegar a la destinació i consumeix una quantitat  $L_j$  de litres de combustible, independentment de la càrrega que transporte.
- Dissenyeu un algorisme que obtinga la manera en què s'han de transportar els objectes (en quin vehicle  $j$  ha d'anar cada objecte  $i$ ) perquè el consum siga mínim.





## 6 L'empresa naviliera

- Imaginem una empresa naviliera que disposa d'una flota de  $N$  vaixells, cadascun dels quals transporta mercaderies d'un valor  $v_i$  que tarden a descarregar-se un temps  $t_i$ . Només hi ha un moll de descàrrega i el temps màxim que es pot usar és  $T$ .
- Dissenyeu un algorisme que determine l'ordre de descàrrega dels vaixells de manera que el valor descarregat siga màxim sense sobrepassar el temps de descàrrega  $T$  (si s'elegeix un vaixell per a descarregar-lo, cal que es descarregue del tot).



## 7 L'assignació de torns

- Estem al començament del curs i els alumnes han de distribuir-se en torns de pràctiques. Per a resoldre aquest problema es proposa que valoren els torns de pràctiques disponibles als quals volen assistir en funció de les seues preferències. El nombre d'alumnes és  $N$  i el de torns disponibles és  $T$ .
- Es disposa una matriu de preferències  $P$ ,  $N \times T$ , en la qual cada alumne escriu, en la fila que li correspon, un número enter (entre 0 i  $T$ ) que indica la preferència de l'alumne per cada torn (0 indica la impossibilitat d'assistir a aquest torn;  $T$  indica màxima preferència).
- D'altra banda, disposem també d'un vector  $C$  amb  $T$  elements que conté la capacitat màxima d'alumnes en cada torn.
- Es vol trobar una solució per a satisfer el nombre mínim d'alumnes segons el seu ordre de preferència, sense excedir la capacitat dels torns.



## 8 Sudoku

- El famós joc del *Sudoku* consisteix a emplenar una quadrícula de  $9 \times 9$  cel·les disposades en 9 subgrups de  $3 \times 3$  cel·les, amb números de l'1 al 9, atenent a la restricció de que no s'ha de repetir el mateix número en la mateixa fila, ni en la mateixa columna, ni en el mateix subgrup  $3 \times 3$ .
- A més, hi ha cel·les on s'ha posat ja un valor inicial, de manera que s'ha de començar a resoldre el problema a partir d'aquesta configuració sense modificar-ne cap de les cel·les inicials.



# Anàlisi i disseny d'algorismes

## 7. Ramificació i poda

José Luis Verdú Mas, Jose Oncina, Mikel L. Forcada

Dept. Llenguatges i Sistemes Informàtics  
Universitat d'Alacant

28/01/2018 (Revisió: 389)



- 1 Presentació
- 2 Introducció
- 3 Eficiència
- 4 El disseny d'algorismes. Divideix i venceràs
- 5 Programació dinàmica
- 6 Algorismes voraços
- 7 Tornada arrere
- 8 Ramificació i poda**



## El problema de la motxilla (general)

Donats:

- $n$  objectes amb valors  $v_i$  i pesos  $p_i$
- una motxilla que solament aguanta un pes màxim  $P$

Seleccionar un conjunt d'objectes de manera que:

- no se sobrepassi el pes límit  $P$  (restricció)
- el valor transportat siga màxim (funció objectiu)



# Exemple introductorí

Formalització del problema:

- Solució:  $X = (x_1, x_2, \dots, x_n)$   $x_i \in \{0, 1\}$
- Restriccions:
  - Implícites:

$$x_i \in \begin{cases} 0 & \text{no se selecciona l'objecte } i \\ 1 & \text{se selecciona l'objecte } i \end{cases}$$

- Explícites:

$$\sum_{i=1}^n x_i p_i \leq P$$

- Funció objectiu:

$$\max \sum_{i=1}^n x_i v_i$$



# Exemple introductori

- Considerem l'exemple següent:

$$P = 16$$

$$p = (7, 8, 2)$$

$$v = (49, 40, 20)$$

- Espai de solucions

Solució	Pes	Valor
---------	-----	-------

(0, 0, 0)	0	0
-----------	---	---

(0, 0, 1)	2	20
-----------	---	----

(0, 1, 0)	8	40
-----------	---	----

(0, 1, 1)	10	60
-----------	----	----

(1, 0, 0)	7	49
-----------	---	----

(1, 0, 1)	9	69
-----------	---	----

(1, 1, 0)	15	89
-----------	----	----

(1, 1, 1)	17	109
-----------	----	-----

Solucions factibles

Solució voraç

Solució òptima

Solució NO factible

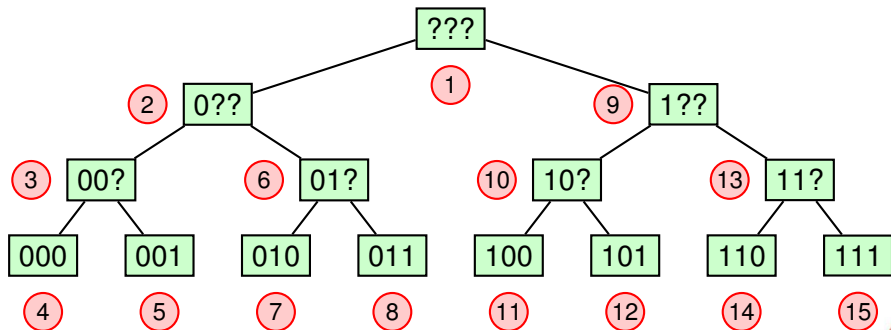




# Exemple introductori

- Combinacions possibles:

- Suposem l'exemple:  $P = 16$   $p = (7, 8, 2)$   $v = (49, 40, 20)$
- Espai de solucions
  - Generació ordenada mitjançant tornada arrere<sup>22</sup>
  - Nodes generats: 15
  - Nodes expandits: 7



<sup>22</sup>Recorregut fix, cego

# Exemple introductori

- Podríem arribar a la solució òptima amb un altre recorregut?
- Permetria això reduir el nombre de nodes generats?
- Com?
  - Usant una funció **fita optimista** que permetia fitar el valor màxim factible que es pot obtenir a partir de la solució parcial que s'està explorant.
  - No s'exploraran aquells nodes amb fites que no milloren el valor de la millor solució obtinguda fins al moment
  - S'adequarà l'ordre **d'exploració** de l'arbre de solucions segons els nostres interessos. Es prioritzaran per a l'exploració aquells nodes més prometedors (usualment aquells amb fites més elevades)



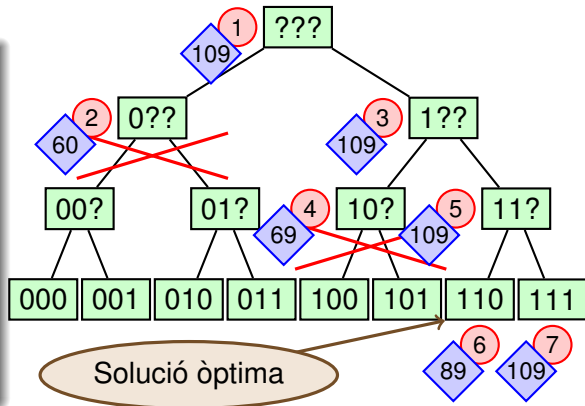
# Exemple introductorí

- Combinacions possibles:

- Suposem l'exemple:  $P = 16$   $p = (7, 8, 2)$   $v = (49, 40, 20)$

## Funció de fita

Cada node pren com a fita el valor que resultaria d'incloure en la solució aquells objectes pendents de tractar (substituir en cada node els '?' per '1'), independentment que càpien o no.



LS#

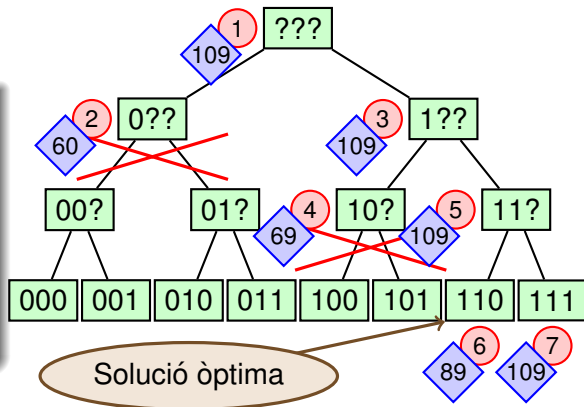
# Exemple introductor

- Combinacions possibles:

- Suposem l'exemple:  $P = 16$   $p = (7, 8, 2)$   $v = (49, 40, 20)$

## Espai de solucions

- Ordre d'expansió prioritant els nodes amb major fita
- Generats: 7
- Expandits: 3
- Reducció  $\geq 50\%$



- Variant del disseny de tornada arrere
- Realitza una enumeració parcial de l'espai de solucions mitjançant la generació d'un arbre d'expansió
- Permet l'exploració de nodes emprant diferents estratègies
  - Tornada arrere: recerca cega, LIFO (*last in, first out*)
  - FIFO (*first in, first out*), una altra recerca cega
  - Ramificació i poda: recerca dirigida



# Definició i àmbit d'aplicació

- Ús de **fites per podar** aquelles branques de l'arbre d'expansió que no condueixen a la solució òptima
- **Node viu**: aquell amb possibilitats de ser ramificat (visitat però no completament expandit)
- Els nodes vius s'emmagatzemen en estructures que faciliten el **recorregut** i l'eficiència de la recerca:
  - En amplària (estratègia FIFO)  $\Rightarrow$  cua
  - En profunditat (estratègia LIFO)  $\Rightarrow$  pila
  - Dirigida (estratègia de màxim benefici)  $\Rightarrow$  monticle (*heap*)



# Definició i àmbit d'aplicació

- Funcionament d'un algorisme de ramificació i poda
- Etapes
  - Partim del node inicial de l'arbre
  - S'hi assigna una **solució pessimista** (subòptima, solucions voraces)
  - Selecció
    - Extracció del node a expandir del conjunt de nodes vius
    - L'elecció depèn de l'estratègia emprada
    - S'actualitza la millor solució amb les noves solucions trobades
  - Ramificació
    - S'expandeix el node seleccionat en l'etapa anterior donant lloc al conjunt dels seus nodes fills
  - Poda
    - S'eliminen (poden) aquells que no condueixen a una millor solució
    - La resta de nodes s'afegeixen al conjunt de nodes vius
  - L'algorisme finalitza quan s'esgota el conjunt de nodes vius



- Procés de poda
  - Necessitem una funció **fita optimista**
  - Para cada node de l'arbre, aquesta funció estima el millor valor que podria aconseguir-se en expandir el node.
  - Si la **fita optimista** d'un node és pitjor que el valor d'una solució ja trobada, podem podar aquest node<sup>23</sup>
  - Conseqüències:
    - Per podar necessitem trobar una solució
    - Com millor siga la solució, més gran el nivell de poda que s'aconseguirà
    - Com més ajustada siga la cota optimista, més grans les podes
    - Se solen emprar solucions al problema amb les restriccions relaxades (p.ex., problema de la motxilla continu)

---

<sup>23</sup>Com en *tornada arrere*



# Esquema de ramificació i poda

```
Solution BB( Problem p ) {  
    Node init = initialNode(p);  
    Solution best = init.pessimistic_sol();  
    priority_queue<Node, vector<Node>, mycomp>  
        q.push(init);  
    while( ! q.empty() ) {  
        Node n = q.top(); q.pop();  
        if( isbetter(n.optimistic_b(), best.value()) )  
            if( n.isTerminal() )  
                best = n.sol();  
            else  
                for( Node a : n.expand() )  
                    if( a.isFeasible() )  
                        q.push(a);  
    }  
    return best;  
}
```

## ● Funcions:

- `initialNode(p)`: obté el node inicial per a l'expansió
- `pessimistic_sol()`: retorna una solució aproximada (factible però no l'òptima).<sup>24</sup>
- `n.value()` retorna el valor del node `n`
- `n.optimistic_b()`: obté una fita superior al la millor solució obtenible de l'expansió de `n`. Si `n` és terminal retornarà `n.value()`
- `n.isTerminal()`: mira si `n` és una possible solució
- `n.sol()`: extrau una solució del node
- `n.expand()`: retorna l'expansió de `n`
- `n.isFeasible()`: comprova si `n` compleix les restriccions

---

<sup>24</sup>Típicament, una voraç

- L'estratègia pot proporcionar:
  - Totes les solucions factibles
  - Una solució al problema
  - La solució òptima al problema
- Objectiu d'aquesta tècnica
  - Millorar l'eficiència en l'exploració de l'espai de solucions
- Desavantatges/necessitats
  - Trobar una bona **fita optimista** (problema relaxat)
  - Trobar una bona **fita pessimista** (estratègies voraces)
  - Trobar una bona estratègia d'exploració (millor fita optimista)
  - Necessita més memòria que els algorismes de tornada arrere
  - Les complexitats en el cas pitjor solen ser molt altes
- Avantatges
  - Són fàcils d'implementar



# Usant fites pessimistes

```
Solution BB( Problem p ) {  
    Node best, init = initialNode(p);  
    Value pb = init.pessimistic_b();  
    priority_queue<Node, vector<Node>, mycomp>  
        q.push(init);  
    while( ! q.empty() ) {  
        Node n = q.top(); q.pop();  
        if( isBetter(n.optimistic_b(), pb) ){  
            pb = best( pb, n.pessimistic_b());  
        }  
        /* max() o min() */  
        if( n.isTerminal() )  
            best = n.sol();  
        else  
            for( Node a : n.expand() )  
                if( a.isFeasible() )  
                    q.push(a);  
    }  
}
```

# Esquema de ramificació i poda

- Funcions:

- `n.pessimistic_b()`: retorna una fita inferior a la millor que s'obté de l'expansió de `n`

- L'avantatge d'usar fites pessimistes és que es pot podar abans de tenir una solució factible

- La condició:


```
n.optimistic_b() >= pb
```

es pot canviar per

```
n.optimistic_b() >= p * pb
```

(amb  $p > 1$ ) si es volen fer podes **agressives**<sup>25</sup>

---

<sup>25</sup>No garanteixen que es trobe l'òptim: només es troba si és **substancialment** millor  que una solució heurística ja disponible (pot interessar en algun cas).

- Exemples d'aplicació:
  - El problema de la motxilla
  - Seqüenciació de tasques amb penalització
- Exercici:
  - El joc del 15 (o del  $n^2 - 1$ )



# El problema de la motxilla

- Vegem com instanciar l'esquema general de ramificació i poda per implementar la resolució del problema
- Formalització del problema:
  - Solució =  $X = (x_1, x_2, \dots, x_n)$   $x_i \in \mathbb{R}$
  - Restriccions:

- Implícites:

$$x_i \in \begin{cases} 0 & \text{no se selecciona l'objecte } i \\ 1 & \text{se selecciona l'objecte } i \end{cases}$$

- Expícites:

$$\sum_{i=1}^n x_i p_i \leq P$$

- Funció objectiu:  $\max \sum_{i=1}^n x_i v_i$
- Valor de fita d'un node:
  - Valor que aconseguiria el node si hi incloem els objectes que queden pendents d'analitzar



# Estructures de dades

```
class Knapsack {  
private:  
    double P;  
    vector<double> v;  
    vector<double> w;    ...
```

[S'assumeix que  $v$  i  $w$  estan ordenats per valor específic]

```
class Node {  
private:  
    static Knapsack p;  
    double pw;  
    double pv;  
    vector<bool> s;  
public:  
    Node(): pw(0.0), pv(0.0) {};    ...
```



# Funcions

```
bool isTerminal() const {  
    return s.size() == p.v.size();  
}
```

```
bool isFeasible() const {  
    return pw <= p.P;  
}
```

```
int value() const {  
    return pv;  
}
```

```
static Node initialNode( const Knapsack& _p ) {  
    p = _p;  
    return Node();  
}
```

## Motxilla discreta voraç

```
Node pessimistic_sol() {
    Node n;
    double n.pv = pv;
    double n.pw = pw;
    for( unsigned i = s.size(); i < p.v.size(); i++ ) {
        if( n.pw + p.w[i] <= p.P ) {
            n.pv += p.v[i];
            n.pw += p.w[i];
            n.s.push_back(true);
        } else {
            n.s.push_back(false);
        }
    }
    return n;
}
```

## Motxilla contínua

```
double optimistic_b() const {  
    double value = pv;  
    double weight = pw;  
    for( unsigned i = s.size(); i < p.v.size(); i++){  
        if( weight + p.w[i] <= p.P ) {  
            value += p.v[i];  
            weight += p.w[i];  
        } else {  
            value += p.v[i] * (p.P - weight) / p.w[i];  
            break;  
        }  
    }  
    return value;  
}
```

## Motxilla voraç

```
double pessimistic_b() const {  
    double value = pv;  
    double weight = pw;  
    for( unsigned i = s.size(); i < p.v.size(); i++){  
        if( weight + p.w[i] <= p.P ) {  
            value += p.v[i];  
            weight += p.w[i];  
        }  
    }  
  
    return value;  
}
```

```
vector<Node> expand() const {  
    Node n1(*this);  
    n1.s.push_back(false);  
  
    Node n2(*this);  
    n2.pw += p.w[s.size()];  
    n2.pv += p.v[s.size()];  
    n2.s.push_back(true);  
  
    return vector<Node>({n1,n2});  
}
```



# Ús d'una cua de prioritat

```
class mycomp {  
public:  
    bool operator() (  
        const Node& n1,  
        const Node& n2  
    ) const {  
        return isBetter(n1.optimistic_b(), n2.optimistic_b());  
    }  
};  
  
typedef  
    priority_queue<Node, vector<Node>, mycomp>  
    myqueue;
```



# Seqüenciació de tasques amb penalització

- Enunciat:

- Hi ha  $N$  tasques pendents de realitzar. Cada tasca  $i$  disposa d'un temps d'execució  $t_i$ , un termini límit de finalització  $f_i$  i una penalització  $p_i$  (indemnització) que s'ha de suportar en el cas que la tasca no s'execute a temps
- Dissenyar un algorisme mitjançant ramificació i poda que determine la data de començament de cada tasca de manera que la indemnització que calga pagar en el cas que no puguem realitzar-se totes les tasques siga mínima
- Exemple:

<b>Tasca</b>	<b>1</b>	<b>2</b>	<b>3</b>	<b>4</b>
<b>Temps d'execució (dies)</b>	2	1	2	3
<b>Termini límit (dies)</b>	3	4	4	3
<b>Indemnització (milers d'euros)</b>	5	15	13	10



- Formalització del problema:
  - Nombre de tasques:  $N$
  - Representació de la solució:  $X = (x_1, x_2, \dots, x_N)$ 
    - $x_i$  representa el dia de començament de la tasca  $i$
    - Cada  $x_i$  pot prendre els valors:  $x_i \in \{0, \dots, (f_i - t_i + 1)\}$
    - El valor zero indica que la tasca no es realitza





# Seqüenciació de tasques amb penalització

- Formalització del problema:

- Representació de la solució:  $X = (x_1, x_2, \dots, x_N)$
- Cada  $x_i$  pot prendre els valors:  $x_i \in \{0, \dots, (f_i - t_i + 1)\}$

- Restriccions:

- Dues tasques no poden estar executant-se al mateix temps (Per a cada tasca  $i$  ja assignada, la nova tasca  $k$  comença després de finalitzar la tasca  $i$  o bé, la tasca  $k$  acaba abans de començar la tasca  $i$ )

$$(x_k > x_i + t_i - 1) \vee (x_k + t_k - 1 < x_i) \quad \forall i \in [1, k]$$

- Funció objectiu:

- Minimitzar el valor de la indemnització total associada a  $X$

$$\min \sum_{i=1}^N \text{ind}(x_i, p_i) \quad \text{ind}(x_i, p_i) = \begin{cases} p_i & x_i = 0 \\ 0 & x_i \neq 0 \end{cases}$$



# Estructures de dades

```
class Seq {  
private:  
    vector<int> len;  
    vector<int> due;  
    vector<int> penalty;  
    ...
```

```
class Node {  
private:  
    static Seq p;  
    vector<int> start;  
    int penalty;  
public:  
    Node(): penalty(0) {};  
    ...
```

# Funcions

```
bool isTerminal() const {  
    return start.size() == p.len.size();  
}
```

```
bool isFeasible() const {  
    return true;  
}
```

```
Node pessimistic_sol() {  
    Node n;  
    n = *this;  
    for( unsigned i=start.size(); i<p.len.size();i++){  
        n.start.push_back(0);  
        n.penalty += p.penalty[i];  
    }  
    return n;  
}
```

# Funcions

```
list<Node> expand() const {
    list<Node> vn;
    int pt = start.size();
    Node n(*this);
    n.penalty += p.penalty[pt];
    n.start.push_back(0);
    vn.push_back(n);
    for( int s = 1; s <= p.due[pt] - p.len[pt]; s++ ){
        if( fit( pt, s) ) {
            Node n(*this);
            n.start.push_back(s);
            vn.push_back(n);
        }
    }
    return vn;
}
```

# Funcions

```
int optimistic_b() const {  
    return penalty;  
}
```

```
int pessimistic_b() const {  
    int pb = penalty;  
    for( unsigned i=start.size(); i<p.len.size();i++){  
        pb += p.penalty[i];  
    }  
    return pb;  
}
```



```
bool fit( int task, int s ) const {  
    for( int prev_task=0; prev_task<task; prev_task++)  
        if( start[prev_task] > 0 &&  
            start[prev_task] < s + p.len[task] ||  
            start[prev_task] + p.len[prev_task] > s ) {  
            return false;  
        }  
    return true;  
}
```



# Ús d'una cua de prioritat

```
class mycomp {  
public:  
    bool operator () (  
        const Node& n1,  
        const Node& n2 ) const {  
        return n1.penalty > n2.penalty;  
    }  
};  
  
typedef  
    priority_queue<Node, vector<Node>, mycomp>  
    myqueue;
```



# El joc del 15 (o del $n^2 - 1$ )

- Disposem d'un tauler amb  $n^2$  caselles i de  $n^2 - 1$  peces numerades de l'1 al  $n^2 - 1$ . Donada una ordenació inicial de totes les peces en el tauler, queda només una casella buida (amb valor 0), que anomenarem *buit*.<sup>26</sup>
- L'objectiu del joc és transformar aquesta disposició inicial de les peces en una disposició final ordenada, on en la casella  $(i, j)$  es troba la peça numerada  $n(i - 1) + j$  i en la casella  $(n, n)$  es troba el buit.
- Els únics moviments permesos són els de les peces adjacents al buit (horitzontalment i verticalment), que poden ocupar-ho. En fer-ho, deixen el buit en la posició on es trobava la peça abans del moviment.
- Una altra forma d'abordar el problema és considerar que el que es mou és el buit, podent fer-ho cap amunt, a baix, esquerra o dreta. En moure's, la seua casella és ocupada per la peça que ocupava la casella on s'ha *mogut* el buit.

<sup>26</sup>Guerequeta i Vallecillo, p. 262 i ss.





# El joc del 15 (o del $n^2 - 1$ )

- Per exemple, per al cas  $n = 3$  es mostra a continuació una disposició inicial juntament amb la disposició final:

1	5	2
4	3	
7	8	6

Disposició Inicial

1	2	3
4	5	6
7	8	

Disposició final



# El joc del 15 (o del $n^2 - 1$ )

- És possible resoldre el problema mitjançant ramificació i poda utilitzant dues funcions de cost diferents:
  - La primera calcula el nombre de peces que estan en una posició diferent de la qual els correspon en la disposició final
  - La segona es basa en la suma de les distàncies de Manhattan des de la posició de cada peça a la seua posició en la disposició final
- La distància de Manhattan entre dos punts del pla de coordenades  $(x_1, y_1)$  i  $(x_2, y_2)$  ve donada per l'expressió:

$$|x_1 - x_2| + |y_1 - y_2|$$

