Tema 2. Anàlisi d'algorismes. Eficiència. Ordenació

Programació (PRG)
Curs 2019/20

Departament de Sistemes Informàtics i Computació

<u>Duració:</u> 7 sessions

Continguts

1. Introducció a l'anàlisi d'algorismes. Conceptes
2. Els costs temporal i espacial dels programes
3. Complexitat asimptòtica
4. Anàlisi per casos
5. Anàlisi d’algorismes iteratius
6. Anàlisi d’algorismes recursius
7. Anàlisi d’algorismes d’ordenació: Selecció directa, Inserció directa, Bambolla unidireccional i MergeSort
8. Altres algorismes: Mescla Natural i Cerca binària

- Pràctiques relacionades:

  PL 3. Mesura empírica de la complexitat computacional (2 sessions)

Introducció a l’anàlisi d’algorismes

Secció 11.1

- És habitual disposar de més d'un programa per a resoldre un mateix problema. Per a decidir quin de tots és el millor és necessari disposar d'un criteri objectiu ⇨ l’eficiència.

- El programa més eficient és el que fa servir menys recursos (memòria RAM i el processador CPU) a la seua execució.

- L’eficiència dels programes s'expressa en termes de:

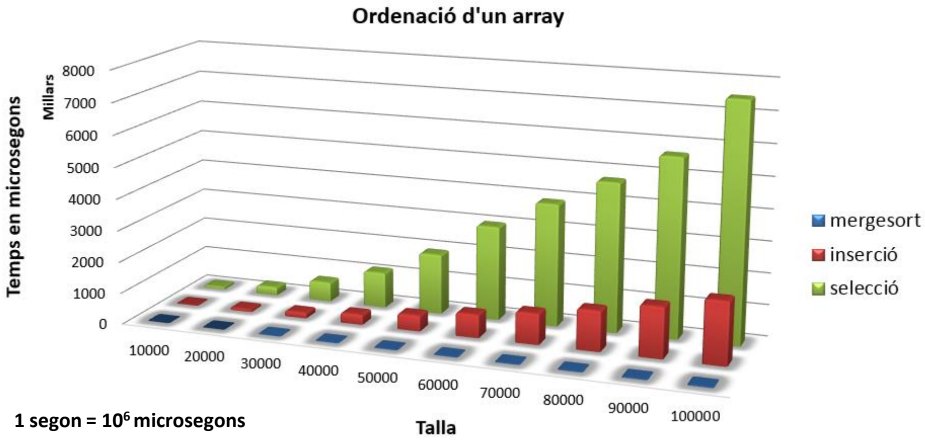
- El cost espacial. Mesura de l'espai de memòria que ocupa un programa durant la seua execució.

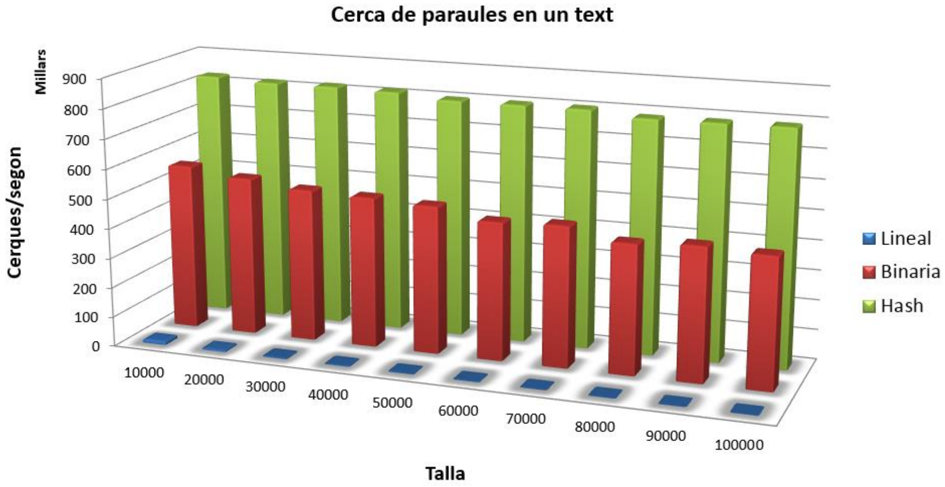
- El cost temporal. Mesura del temps que necessita un programa per a executar-se i donar un resultat a partir de les dades d'entrada.

Introducció a l'anàlisi d'algorismes

- Els costos d'un programa en concret depenen de dos tipus de factors:

- Factors propis del programa, com són l'estratègia utilitzada i els tipus de dades que empra.





- Factors aliens al programa, corresponents a l'entorn de programació on s'executa, com són el tipus de computador, el llenguatge de programació, el compilador, la càrrega del sistema, etc.

Introducció a l'anàlisi d'algorismes

- El cost d'un programa es pot estimar de dues formes:
- Anàlisi teòrica o a priori
  - El cost s'estima en funció dels factors propis del programa.
  - Es tracta d'una anàlisi independent de l'entorn de programació.
- Anàlisi experimental o a posteriori
  - El cost s'estima mesurant, p.e. en segons, el temps que tarda en executar-se un programa i, p.e. en bytes, l'espai ocupat mentre s'executa.
  - És una anàlisi en un entorn de programació particular i per a un conjunt determinat de dades d'entrada (instància).

- Es tracta de dos tipus d'anàlisi complementàries:

1. Una anàlisi experimental permet obtenir el temps d'execució en un sistema concret.
2. Però independentment de l'entorn on s'execute, un programa ha de ser eficient.
3. Una anàlisi teòrica permet preveure el comportament d'un programa abans de la seua implementació (evitant perdre el temps programant algorismes que després no es fan servir).

Els costos temporal i espacial

Secció 11.2.2

- El cost temporal (espacial) d'un algorisme es defineix com:
  - Una funció positiva, no decreixent de la quantitat de temps (espai) que necessita l'algorisme per executar-se en funció de l'amplària o grandària del problema.
- La talla, gràndaria o amplària d'un problema es defineix com:
  - El valor o conjunt de valors associats a les dades d'entrada que representen una mesura de la dificultat per a la seua resolució.
- D'ara endavant, el cost temporal d'un algorisme A s'indicarà com: T_A(talla).
- Exemples d'elecció de la talla del problema:

Problema	Grandària, amplària o talla
Cerca d'un element en un conjunt	Nombre d'elements del conjunt
Multiplicació de matrius	Dimensió de les matrius
Càlcul del factorial d'un nombre	Valor del nombre
Resolució d'un sistema d'equacions lineals	Nombre d'equacions i/o incògnites
Ordenació d'un array	Nombre d'elements de l'array

Els costos temporal i espacial

Secció 11.2.1

- Primera aproximació: Comptar operacions elementals
- El cost temporal d'un algorisme es pot mesurar prenent en consideració el temps d'execució de les seues operacions elementals.

t_a : cost d'una assignació
t_{op} : cost d'una operació aritmètica
t_c : cost d'una comparació

- Algorisme A1:

```
m = n * n;
```

$t_a + t_{op}$

- Algorisme A2:

```
m = 0;
for (int i = 0; i < n; i++) {
    m += n;
}
```

t_a
$t_a + (n+1)t_c + nt_{op}$
$nt_a + nt_{op}$

- Algorisme A3:

```
m = 0;
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        m++;
    }
}
```

t_a
$t_a + (n+1)t_c + nt_{op}$
$nt_a + n(n+1)t_c + n^2t_{op}$
n^2t_{op}

Els costos temporal i espacial

- El cost temporal d'un algorisme es defineix com la suma dels costos de les operacions elementals que aquest implica:
  - Cost de l'algorisme A1:

    $$T_{A1}(n) = t_a + t_{op}$$

  - Cost de l'algorisme A2:

    $$T_{A2}(n) = t_a + t_a + (n+1)\ t_c + n\ t_a + 2\ n\ t_{op}$$

  - Cost de l'algorisme A3:

    $$T_{A3}(n)= t_a + t_a + (n+1)\ t_c + n\ t_a + n\ (n+1)\ t_c + 2\ n^2\ t_{op} + n\ t_{op}$$

t_a : cost d'una assignació
t_{op} : cost d'una operació aritmètica
t_c : cost d'una comparació

- Comparar els costos d'aquests algorismes, amb aquest tipus d'anàlisi és prou difícil i, a més, requereix un esforç de càlcul considerable.

Els costos temporal i espacial

- El primer pas seria independitzar els costos dels temps de les operacions elementals, suposant constant el temps que necessita cada operació.
  - Cost de l'algorisme A1: (constant, no depén de n)

    $T_{A1}(n) = t_a + t_{op} \equiv k_1$

  - Cost de l'algorisme A2: (dependència lineal de n)

    $T_{A2}(n) = t_a + t_a + (n + 1)\, t_c + n\, t_a + 2\, n\, t_{op}$

    $\equiv (t_c + t_a + 2t_{op})\, n + (t_c + 2t_a) \equiv k_2\, n + k_3$

  - Cost de l'algorisme A3: (dependència quadràtica de n)

    $T_{A3}(n) = t_a + t_a + (n + 1)\, t_c + n\, t_a + n\, (n + 1)\, t_c + 2\, n^2\, t_{op} + n\, t_{op}$

    $\equiv (t_c + 2t_{op})\, n^2 + (2t_c + t_a + t_{op})\, n + (t_c + 2t_a) \equiv k_4\, n^2 + k_5\, n + k_6$

Els costos temporal i espacial

Secció 11.2.3

- Segona aproximació: Comptar passos de programa
- Pas de programa: seqüència d'operacions bàsiques significatives amb cost independent de la talla del problema.
- Un pas es pot considerar com una unitat de temps vàlida per a expressar el cost d'un algorisme. Així, l'anàlisi de costos s'independitza del temps de cada operació elemental (totes les operacions tarden una unitat de temps).
  - Cost de l'algorisme A1:

    $T_{A1}(n) = k_1$ $\quad\quad$ $T_{A1}(n) = 1$ *p.p.*
  - Cost de l'algorisme A2:

    $T_{A2}(n) = k_2 n + k_3$ $\quad\quad$ $T_{A2}(n) = n + 2$ *p.p.*
  - Cost de l'algorisme A3:

    $T_{A3}(n) = k_4 n^2 + k_5 n + k_6$ $\quad\quad$ $T_{A3}(n) = n^2 + n + 2$ *p.p.*

k_1, k_2, k_3, k_4, k_5, k_6 són desconegudes en una anàlisi a priori
El seu valor pot canviar segons la implementació

Considerem una passada del bucle com un pas de programa. En quant a la resta d'operacions (la inicialització de *m*, l'última comparació de la guarda del bucle) es pot considerar cada operació com un pas de programa o totes elles com un únic pas de programa

Els costos temporal i espacial

Secció 11.6.1

- Tercera aproximació: Comptar instruccions crítiques
- Instrucció crítica (o baròmetre):
  - Instrucció que es repeteix tantes vegades com qualsevol altra en el bucle (o bucles) de l'algorisme a estudiar.
  - Ha de complir una premissa: ser independent de les dades i del tipus de problema ⇨ normalment serà una instrucció bàsica.
- Així, es pot expressar el cost en funció de les vegades que s'executa una instrucció crítica (de cost unitari)
  - Cost de l'algorisme A1: $T_{A1}(n) = 1$ *i.c.*
  - Cost de l'algorisme A2: podem considerar com instr. crítiques:

    la guarda del bucle (`i < n`) o l'increment de i (`i++`) o l'assignació (`m += n`)

    `sup-inf+1` `sup-inf+1`

    $T_{A2}(n) = \sum_{i=0}^{n} 1 = n + 1$ *i.c.* *o* $T_{A2}(n) = \sum_{i=0}^{n-1} 1 = n$ *i.c.*
  - Cost de l'algorisme A3: podem considerar com instr. crítiques:

    la guarda del bucle (`j < n`) o l'increment de j (`j++`) o l'increment de m (`m++`)

    $T_{A3}(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n} 1 = \sum_{i=0}^{n-1} (n+1) = n^2 + n$ *i.c.* *o*

    $T_{A3}(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} 1 = \sum_{i=0}^{n-1} n = n^2$ *i.c.*

Complexitat asimptòtica

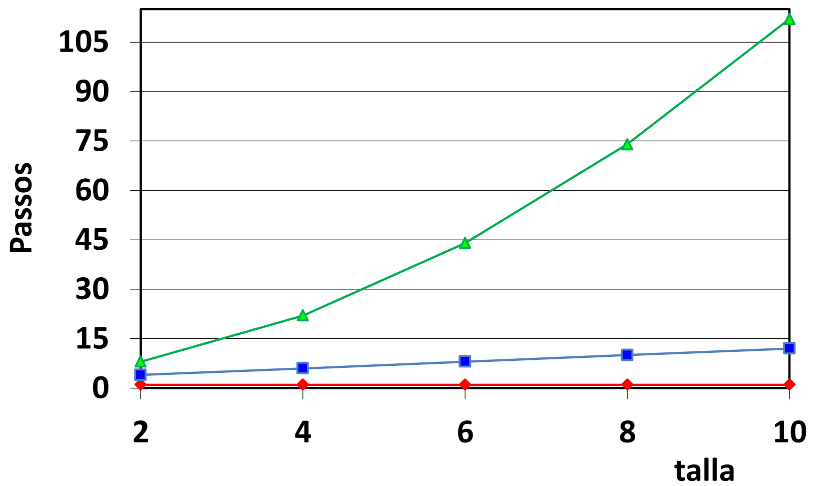
- El cost d'un algorisme s'expressa com una funció $T(n)$, que és una funció positiva no decreixent respecte a la talla del problema.

- Comparar costos d'algorismes consisteix en comparar funcions no decreixents, on el que interessa és la seua taxa de creixement.

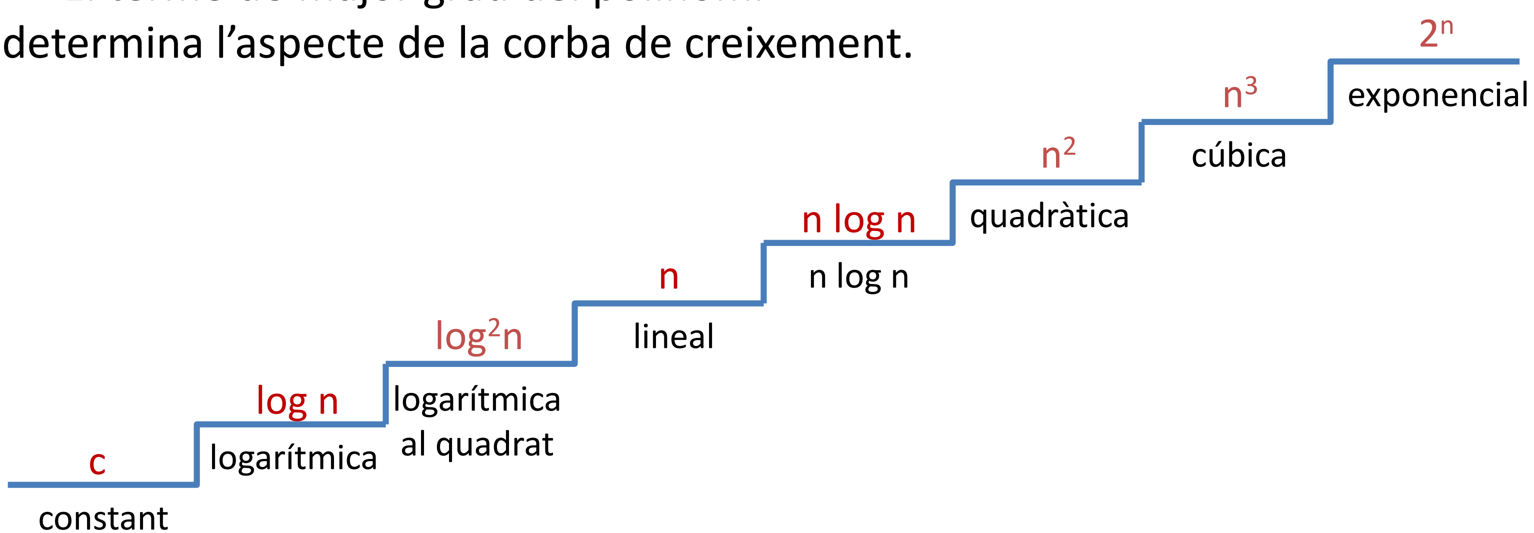
$T_{A1}(n) = 1 \qquad \approx k$

$T_{A2}(n) = n + 2 \qquad \approx n$

$T_{A3}(n) = n^2 + n + 2 \qquad \approx n^2$

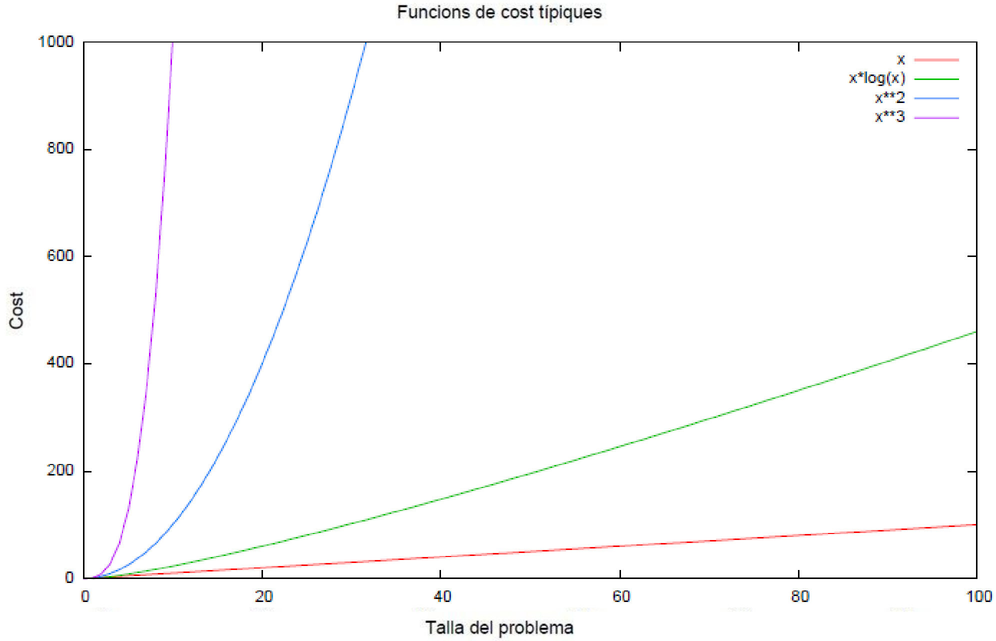


- Si $T(n)$ és un polinomi, llavors:
  - El terme de major grau del polinomi determina l'aspecte de la corba de creixement.

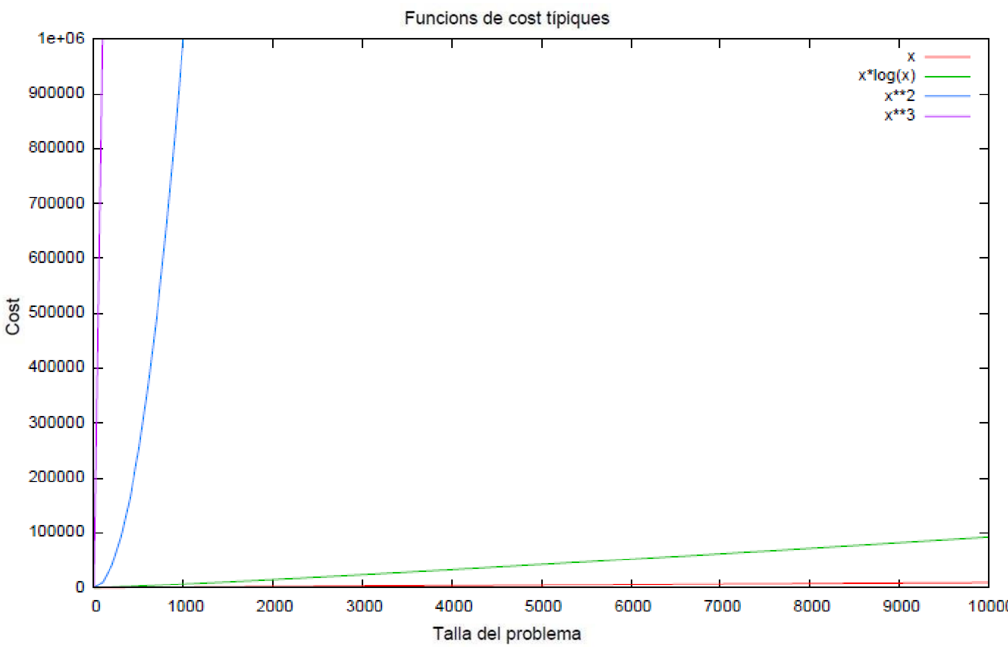


Complexitat asimptòtica

Secció 11.3.1



Les funcions no lineals duen a temps majors que les funcions lineals.
Els valors menuts de n generalment no són importants. Per a n=20, tots els algorismes acaben abans de 5 segons. La diferència entre el millor i el pitjor algorisme és molt reduïda.



Per a valors de n suficientment grans, el valor de la funció està completament determinat pel terme dominant. Per exemple, en la funció $10n^3+n^2+40n+80$, per a n=1000, el valor és 10.001.040.080, del qual 10.000.000.000 és degut al terme $10n^3$.

Complexitat asimptòtica

- Un algorisme lineal es caracteritza perquè el temps d'execució és proporcional a l'entrada del problema.
  - $T_{A1}(n) = 1$ *p.p.* Cost constant per a qualsevol talla
  - $T_{A2}(n) = n + 2$ *p.p.* Cost lineal proporcional a la talla:

    $T_{A2}(10n_0) \cong 10T_{A2}(n_0)$

    $T_{A2}(100n_0) \cong 100T_{A2}(n_0)$

    $T_{A2}(1000n_0) \cong 1000T_{A2}(n_0)$

    …
  - $T_{A3}(n) = n^2 + n + 2$ *p.p.* Cost quadràtic amb la talla:

    $T_{A2}(10n_0) \cong 100T_{A2}(n_0)$

    $T_{A2}(100n_0) \cong 10000T_{A2}(n_0)$

    $T_{A2}(1000n_0) \cong 10^6T_{A2}(n_0)$

    …
- El cost lineal és un bon cost, el cost logarítmic és millor, i el cost n log n també és bo. La resta de costos ja limiten considerablement la talla dels problemes que podem resoldre.

Complexitat asimptòtica

- Quan s'analitza el cost d'un algorisme, allò que interessa és el tipus de creixement que té.
- Si s'identifiquen els tipus de creixement o funcions típiques, es poden comparar algorismes, i així triar entre ells.
- Per tant, l'anàlisi a *priori* dels algorismes consistirà en mesurar la taxa de creixement de les funcions de cost i expressar-ho en notació asimptòtica.

- Ordre de magnitud o taxa de creixement (AMA, UT4)
- Donades f(n), g(n), n $\in \mathbb{N}$, positives, no decreixents,
  - g(n) és de major ordre que f(n) sii $\lim_{\mathrm{n}\to\infty}\left(\frac{\mathrm{g(n)}}{\mathrm{f(n)}}\right) = \infty$
  - g(n) és de menor ordre que f(n) sii $\lim_{\mathrm{n}\to\infty}\left(\frac{\mathrm{g(n)}}{\mathrm{f(n)}}\right) = 0$
  - g(n) i f(n) són del mateix ordre sii $\lim_{n\to\infty}\left(\frac{\mathrm{g(n)}}{\mathrm{f(n)}}\right) = k$

Complexitat asimptòtica

- Notació asimptòtica
  - $f(n) \in O(g(n))$ sii $g(n)$ és de major o igual ordre que $f(n)$
    - ⇨ $g(n)$ és asimptòticament una fita superior de $f(n)$.
  - $f(n) \in \Omega(g(n))$ sii $g(n)$ és de menor o igual ordre que $f(n)$
    - ⇨ $g(n)$ és asimptòticament una fita inferior de $f(n)$.
  - $f(n) \in \Theta(g(n))$ sii $g(n)$ i $f(n)$ són del mateix ordre
    - ⇨ $g(n)$ és asimptòticament una fita superior i inferior de $f(n)$.

- Propietats
  - $f(n) \in O(g(n)) \leftrightarrow g(n) \in \Omega(f(n))$
  - $f(n) \in \Theta(g(n)) \leftrightarrow f(n) \in \Omega(g(n)),\ f(n) \in O\,(g(n))$

    $f(n) \in O(g(n)) \cap \Omega(g(n))$
  - $f(n) \in O(g(n)) \leftrightarrow f(n) + g(n) \in \Theta(g(n))$

Complexitat asimptòtica

- Les raons que recolzen aquest tipus d'anàlisi són:
  - Per a valors de n grans, el valor de la funció de cost està determinat pel terme dominant.
  - El valor exacte del coeficient del terme dominant no es conserva en canviar d'entorn de programació.
  - L'ús de la notació asimptòtica estableix un ordre relatiu entre les funcions de cost que permet comparar algorismes comparant els termes dominants de la seua funció de cost.

- Taula de funcions típiques emprant la notació asimptòtica:

Funció	Nom	Notació asimptòtica
c	constant	$\Theta(1)$
log n	logarítmica	$\Theta(\log n)$
$\log^2 n$	logarítmica al quadrat	$\Theta(\log^2 n)$
n	lineal	$\Theta(n)$
n log n	n - logarítmica	$\Theta(n \log n)$
n^2	quadràtica	$\Theta(n^2)$
n^3	cúbica	$\Theta(n^3)$
2^n	exponencial	$\Theta(2^n)$

$$O(1) \subset O(\log n) \subset O(\log^2 n) \subset O(n) \subset O(n \log n) \subset O(n^2) \subset O(n^3) \subset \ldots \subset O(2^n) \subset O(n!)$$

Complexitat asimptòtica

- Exemple:

Overview Package **Class** Use Tree Deprecated Index Help

Java™ Platform Standard Ed. 8

Prev Class **Next Class** Frames No Frames

Summary: Nested | Field | Constr | Method Detail: Field | Constr | Method

java.util

Class PriorityQueue<E>

java.lang.Object
    java.util.AbstractCollection<E>
        java.util.AbstractQueue<E>
            java.util.PriorityQueue<E>

Type Parameters:

`E` - the type of elements held in this collection

All Implemented Interfaces:

Serializable, Iterable<E>, Collection<E>, Queue<E>

```
public class PriorityQueue<E>
extends AbstractQueue<E>
implements Serializable
```

An unbounded priority queue based on a priority heap. The elements of the priority queue are ordered according to their natural ordering, or by a `Comparator` provided at queue construction time, depending on which constructor is used. A priority queue does not permit `null` elements. A priority queue relying on natural ordering also does not permit insertion of non-comparable objects (doing so may result in `ClassCastException`).

...

Implementation note: this implementation provides O(log(n)) time for the enqueing and dequeing methods (`offer`, `poll`, `remove()` and `add`); linear time for the `remove(Object)` and `contains(Object)` methods; and constant time for the retrieval methods (`peek`, `element`, and `size`).

Complexitat asimptòtica

- A mesura que els ordinadors es tornen més i més ràpids pot paréixer que tot just val la pena invertir el temps en dissenyar algorismes més eficients. I si esperem a la següent generació d'ordinadors?, per què cercar l'eficiència?
- Suposem que per resoldre un problema concret es disposa d'un algorisme exponencial i d'un ordinador que executa aquest algorisme per a grandària n en $10^{-4} \times 2^n$ segons.

n	temps
10	$10^{-4} \times 2^{10}$ s., aprox. 1 dècima de segon
20	$10^{-4} \times 2^{20}$ s., aprox. 2 minuts
30	$10^{-4} \times 2^{30}$ s., més d'1 dia de càlcul
¡38! ⇦	quasi 1 any

- Suposem que es compra un ordinador nou, 100 vegades més ràpid que l'anterior. Ara es pot resoldre el mateix algorisme per a grandària n en $10^{-6} \times 2^n$ segons.
  En 1 any ⇨ n < 45!

  365 x 24 x 60 x 60 segons/any = $10^{-6} \times 2^n \rightarrow 31536000 \times 10^6 = 2^n \rightarrow n = 44{,}84$

- En general, si abans es resolia un exemplar de grandària n en un temps donat, la nova màquina resoldrà grandàries de com a màxim $n + \log_2 100$, aprox. $n + 7$, en el mateix temps.

$$10^{-4} \times 2^n = 10^{-6} \times 2^{(n+x)} \rightarrow 100 = 2^x \rightarrow x = \log_2 100 \approx 7$$

Complexitat asimptòtica

- Suposem que s'inverteix en algorísmia i, per la mateixa quantitat de diners, es troba un algorisme cúbic que en la màquina original resol un exemplar de grandària n en $10^{-2} \times n^3$ s.

n		temps
10		$10^{-2} \times 10^3$ = 10 segons
20		$10^{-2} \times 20^3$ = entre 1 i 2 minuts
30		$10^{-2} \times 30^3$ = 4,5 minuts
>200		1 dia
¡1500!	⇦	1 any

- El nou algorisme no sols ofereix una millora prou major que l'adquisició del nou equip, sinó que a més, suposant que puga un permetre's amb dues coses, farà que esta adquisició siga més rendible.

- En general, les millores en un programa per canvi de llenguatge, de computador, estalvi de variables i/o d'instruccions, etc. són menys importants que les millores en l'estratègia que impliquen una taxa de creixement inferior.

poli[formaT] PRG > EXÀMENS 1B - T2. Qüestionari: eficiència

Anàlisi per casos

Secció 11.4

- El cost d'un algorisme és una funció no decreixent de la talla del problema.
- Per a una grandària en particular, el cost de l'algorisme també pot dependre d'altres característiques de les dades d'entrada (instàncies del problema).
- Una instància d'un problema representa totes les configuracions diferents de l'entrada, per a una talla determinada, per a les quals el cost de l'algorisme és semblant, és a dir, el comportament de l'algorisme és el mateix quant a costos.
- Si en l'anàlisi dels costos d'un algorisme es detecten instàncies que causen variacions en el comportament, llavors es tipifiquen els casos següents:
  - Cost de l'algorisme en el pitjor cas: És la complexitat del mateix per a la instància del problema que presente el cost major. Es denota per $T^p(n)$
  - Cost de l'algorisme en el millor cas: És la complexitat del mateix per a la instància del problema que presente el cost menor. Es denota per $T^m(n)$
  - Cost de l'algorisme en terme mitjà: És la mitjana dels costos de totes les instàncies del problema. Es denota per $T^\mu(n)$
- A l'hora d'analitzar un algorisme, interessen els seus comportaments en els casos pitjor, millor i terme mitjà.

Anàlisi per casos

- Algorismes de recorregut i cerca amb arrays:
- El problema de recorregut d'un array només presenta una instància, per tant, no es pot distingir entre casos millor i pitjor.
- El problema de cerca sí que presenta vàries instàncies:
  - l'element a cercar es troba en la posició 0
  - l'element a cercar es troba en la posició 1
  - . . .
  - l'element a cercar es troba en la posició n-1
  - l'element a cercar no es troba en cap posició.

Amb èxit

Sense èxit

n + 1 instàncies

- Si l'algorisme realitza la cerca seqüencial des del començament (posició 0 de l'array):
  - millor cas: l'element a cercar es troba en la primera posició.
  - pitjor cas: l'element no es troba.

Anàlisi per casos

- Regla general per a calcular T(n):
- Determinar la *talla* del problema, és a dir, estudiar de quins paràmetres depén el cost.
- Analitzar si, per a una talla concreta, hi ha *instàncies* significatives en quant al cost. És a dir, si s'observen comportaments diferents de l'algorisme.
- Obtindre la funció de cost:
  - Si no hi ha instàncies significatives, s'utilitza la notació Θ.
  - Si hi ha instàncies significatives, l'estudi es farà per als casos pitjor i millor. L'estudi sobre el cas pitjor dóna la fita superior del cost de l'algorisme i sobre el cas millor dóna la fita inferior del cost de l'algorisme.
  - Per a la fita superior s'utilitza la notació O.
  - Per a la fita inferior s'utilitza la notació Ω.

Anàlisi per casos

- Eficiència d'un algorisme de recorregut:
- Siga v un array de n elements que es pot recórrer mitjançant un algorisme com el següent:

```
for (int i = 0; i < n; i++) {
    tractar(v[i]);
}
```

- Es suposa que **tractar(v[i])** és una operació de cost constant sobre l'element **v[i]**.
- La *talla* del problema és **n** , el nombre d'elements de l'array.
- Com instruccions crítiques (i.c.) es poden considerar:

tractar(v[i])   **i++**   **i < n**

- L'algorisme no presenta instàncies significatives, el cost s'aproxima comptant el nombre de vegades que es repeteix la instrucció **tractar(v[i])** que, en qualsevol cas, s'executa 1 vegada menys que la guarda del bucle (lo que és despreciable a efectes de cost asimptòtic).
- El cost de l'algorisme és:

$$T(n) = \sum_{i=0}^{n-1} 1 = \underbrace{(n-1-0+1)}_{\texttt{sup-inf+1}} = n \; i.c. \in \Theta(n)$$

Anàlisi per casos

- Eficiència d'un algorisme de cerca seqüencial:
- Siga v un array de n elements on es cerca la posició d'un element que compleix una certa propietat mitjançant el següent algorisme:

```
int i = 0;
while (i < n && !propietat(v[i])) { i++; }
if (i < n) { return i; }
else { return -1; }
```

- **`propietat(v[i])`** és la propietat que ha de complir l'element que cerquem i que té un cost constant.
- La *talla* del problema és **`n`** , el nombre d'elements de l'array.
- El cost de l'algorisme depèn de la instància del problema. Per a aquest algorisme hi ha n + 1 instàncies significatives.
- Es considera com instrucció crítica la guarda del bucle:

```
i < n && !propietat(v[i])
```

  - Cost del cas pitjor: $T^p(n) = n + 1$ i.c. $\in \Theta(n)$ ⇨ $T(n) \in O(n)$ lineal
  - Cost del cas millor: $T^m(n) = 1$ i.c. $\in \Theta(1)$ ⇨ $T(n) \in \Omega(1)$ constant

Anàlisi per casos

Secció 11.6.4

- Eficiència d'un algorisme de cerca seqüencial
- Per tal d'estimar el cost en terme mitjà es necessita conèixer la distribució de probabilitat de les diferents instàncies.

- Primera situació considerada:
- La cerca sempre té èxit, i la probabilitat de que l'element a cercar es trobe en qualsevol posició de l'array és la mateixa.
- Hi ha n instàncies possibles, amb probabilitat 1/n cadascuna.
- Cost de cada instància: Nombre de vegades que s'execute el bucle, que coincideix amb la posició on es troba l'element i.

$$T^{\mu}(n) = \sum_{i=1}^{n} \frac{1}{n} i = \frac{n(n+1)}{2n} \in \Theta(n)$$

- Segona situació considerada:
- És equiprobable que l'element es trobe o no a l'array i, en cas de trobar-se, totes les posicions són equiprobables.
- Hi ha n + 1 instàncies possibles:
  - Que l'element no es trobe, amb una probabilitat 1/2.
  - Que l'element es trobe en qualsevol posició de l'array, amb probabilitat 1/2n.
- El cost de la primera instància és n+1, el cost de les restants instàncies és i.

$$T^{\mu}(n) = \frac{n+1}{2} + \sum_{i=1}^{n} \frac{1}{2n} i = \frac{n+1}{2} + \frac{n(n+1)}{4n} \in \Theta(n)$$

Anàlisi d'algorismes iteratius

- Identificar la *talla* i les *instàncies* del problema.
- Calcular T(n) per a una instància donada:
  - Comptar el número d'iteracions ⇨ dependran de com es redueix la talla en cada passada.
  - Usar el mètode de la instrucció crítica (o comptar passos de programa) per a cada iteració ⇨ no totes les passades tindran necessàriament el mateix cost (bucles niuats, crides a mètodes, …).

Anàlisi d'algorismes iteratius

Problema. Càlcul de a^n, enters $a > 0$ i $n \geq 0$

```
aⁿ = a * a * … * a    n > 0
aⁿ = 1                n = 0
```

```
/** Càlcul de la potencia n-èsima d'a.
 *  Precondició: a > 0 i n >= 0 */
public static long potencia(long a, long n) {
    long pot = 1;
    while (n > 0) {
        pot = pot * a;
        n--;
    }
    return pot;
}
```

Talla: El valor de l'exponent, n

Instàncies: No

n va disminuint en restar-li 1 en cada passada

⇨ n passades

reducció aritmètica

o disminució de la talla en una diferència constant

Instrucció crítica: n > 0 (de cost unitari)

$T_{potencia}(n) = n + 1$ i.c. $\in \Theta(n)$

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf Escuela Técnica Superior de Ingeniería Informática

Anàlisi d'algorismes iteratius

Problema. Càlcul de a^n, enters $a > 0$ i $n \geq 0$

```
aⁿ = (a * a)^(n/2)                       n > 0, n parell
aⁿ = (a * a)^((n-1)/2) * a = (a * a)^(n/2) * a   n > 0, n senar
aⁿ = 1                                   n = 0
```

$a^n = (a * a)^{n/2}$ — $n > 0$, n parell

$a^n = (a * a)^{(n-1)/2} * a = (a * a)^{n/2} * a$ — $n > 0$, n senar

$a^n = 1$ — $n = 0$

Talla: El valor de l'exponent, n

Instàncies: No

```java
/** Càlcul de la potencia n-èsima d'a.
 *  Precondició: a > 0 i n >= 0 */
public static long potencia(long a, long n) {
    long pot = 1;
    while (n > 0) {
        if (n % 2 == 1) { pot = pot * a; }
        a = a * a;
        n = n / 2;
    }
    return pot;
}
```

Instrucció crítica: `n > 0` (de cost unitari)

Quantes vegades s'avalua?

Talla a l'inici:	n
Després 1ª iteració:	$n/2$
Després 2ª iteració:	$n/2^2$
Després 3ª iteració:	$n/2^3$
…	
Després penúltima iter.:	$1 \leq n/2^m < 2$
Després última iter.:	$0 \leq n/2^{m+1} < 1$

$m = \lfloor \log_2 n \rfloor$

n va disminuint dividint-se per 2 en cada passada

⇨ m + 1 passades

reducció geomètrica

o divisió de la talla per un factor constant

$\boxed{m + 1 = \lfloor \log_2 n \rfloor + 1}$ `true` $\quad + 1$ `false`

$T_{potencia}(n) = m + 2$ i.c. $\in \Theta(\log n)$

Anàlisi d'algorismes iteratius – Bucles niuats

Problema. Donat un enter n ≥ 1, escriu en l'eixida estàndard n línies, cadascuna de n '*'.
Per exemple, si n = 4:

```
****
****
****
****
```

```
/** Precondició: n >= 1 */
public static void escriu(int n) {
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= n; j++) {
            System.out.print('*');
        }
        System.out.println();
    }
}
```

Talla: número de línies a escriure, n
Instàncies: No
Instrucció crítica: `System.out.print('*')` (de cost unitari)

$T_{escriu}(n) = \sum_{i=1}^{n} \sum_{j=1}^{n} 1 = \sum_{i=1}^{n} n = n^2$ i.c. $\in \Theta(n^2)$

Anàlisi d'algorismes iteratius – Bucles niuats

Problema. Donat un enter n ≥ 1, escriu en l'eixida estàndard n línies de n '*', de longitud creixent 1, 2, ..., n. Per exemple, si n = 4:

```
*
**
***
****
```

```
/** Precondició: n >= 1 */
public static void escriuCreixent(int n) {
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= i; j++) {
            System.out.print('*');
        }
        System.out.println();
    }
}
```

Talla: número de línies a escriure, n
Instàncies: No
Instrucció crítica: `System.out.print('*')` (de cost unitari)

$$T_{escriuCreixent}(n) = \sum_{i=1}^{n} \sum_{j=1}^{i} 1 = \sum_{i=1}^{n} i = \frac{n(n+1)}{2} \text{ i.c.} \in \Theta(n^2)$$

Anàlisi d'algorismes iteratius – Bucles niuats

Problema. Donat un enter n ≥ 0, calcula $\sum_{i=0}^{n} i!$

```
/** Càlcul de la suma de i!, i en [0..n]
 *  Precondició: n >= 0 */
public static int sumaFact(int n) {
    int res = 1;
    for (int i = 1; i <= n; i++) {
        res = res + factorial(i);
    }
    return res;
}
```

Explícitament no hi ha un niuament d'iteracions, però en cada passada s'invoca a un mètode el cost del qual varia segons la passada que s'està fent

Talla: el paràmetre del mètode, n
Instàncies: No
Instrucció crítica: `res = res + factorial(i);` de cost $\Theta(i)$

$$T_{sumaFact}(n) = \sum_{i=1}^{n} i = \frac{n(n+1)}{2} \text{ i.c.} \quad \in \Theta(n^2)$$

```
/** Càlcul de n!
 *  Precondició: n >= 0 */
public static int factorial(int n) {
    int f = 1;
    for (int i = 1; i <= n; i++) { f = f * i; }
    return f;
}
```

$T_{factorial}(n) \in \Theta(n)$

Anàlisi d'algorismes iteratius

Problema. Donat un array d'enters v (`v.length > 2`), comprova que el valor de totes les posicions a partir de la 3ª posició (índex 2 de l'array) és igual a la suma dels valors de les dues posicions precedents.

```
/** Precondició: v.length > 2 */
public static boolean compleixCondicio(int[] v) {
    int i = 2; boolean compleix = true;
    while (i < v.length && compleix) {
        compleix = v[i] == v[i - 1] + v[i - 2];
        i++;
    }
    return compleix;
}
```

Talla: Nombre d'elements de l'array v, donat per l'expressió `v.length` que anomenarem n
Instàncies: Sí
Instrucció crítica: `i++` (de cost unitari)

Es tracta d'un problema de cerca on la propietat a cercar és: `v[i] != v[i - 1] + v[i - 2]`. Aquesta propietat pot complir-se en qualsevol posició a partir de la 3ª (índex 2 de l'array), detenint-se la cerca. Per tant, sí hi ha instàncies significatives, que són:

- Cas millor: a la 3ª posició es compleix la propietat a cercar: `v[2] != v[1] + v[0]`.

$$T^m(n) = 1 \text{ i.c.} \in \Theta(1) \qquad ☞ \; T(n) \in \Omega(1)$$

- Cas pitjor: no es compleix la propietat per a cap posició, és a dir, per a totes les posicions a partir de la 3ª posició (índex 2 de l'array) el valor és igual a la suma dels valors de les dues posicions precedents. Açò és, $\forall$`i`,`2`$\leq$`i`<`v.length, v[i] == v[i - 1] + v[i - 2]`.

$$T^p(n) = \sum_{i=2}^{n-1} 1 = n - 2 \text{ i.c.} \in \Theta(n) \qquad ☞ \; T(n) \in O(n)$$

Anàlisi d'algorismes iteratius

Problema. Donat un array d'enters a i un enter x, torna la posició de x a l'array si el troba, o torna -1 si no el troba.

```
public static int trobarX(int[] a, int x) {
    int trobat = -1, i = 0, j = a.length-1;
    while (i <= j && trobat == -1) {
        if (a[i] == x) { trobat = i; }
        else if (a[j] == x) { trobat = j; }
        i++; j--;
    }
    return trobat;
}
```

Talla: Nombre d'elements de l'array a, donat per l'expressió `a.length` que anomenarem n
Instàncies: Sí
Instrucció crítica: `a[i] == x` (de cost unitari)

- Cas millor: x està en la posició 0 (`a[0] == x`) o en la posició `a.length-1` (`a[a.length-1] == x`). La instrucció crítica s'executa 1 vegada:

$$T_{trobarX}^{m}(n) = 1 \text{ i.c.} \in \Theta(1) \quad ☞ \quad T_{trobarX}(n) \in \Omega(1)$$

- Cas pitjor: x no està a l'array, o també quan x està en la posició central de l'array (és a dir, `a[a.length/2] == x`, en un array amb un número senar d'elements i `a[a.length/2 - 1] == x` o `a[a.length/2] == x`, en un array amb un número parell d'elements).

En la primera passada del bucle la cerca es fa sobre un array de talla n, i en cada passada la talla es decrementa en 2. Després de la k-èsima passada del bucle, la talla ve donada per n−2k. Després de l'última passada, la talla del problema s'ha reduït a un subarray de talla 0. Aleshores, n−2k = 0. S'han fet k = n/2 passades, és a dir, la instrucció crítica s'ha executat n/2 vegades en el cas pitjor, i

$$T_{trobarX}^{p}(n) = n/2 \text{ i.c.} \in \Theta(n) \quad ☞ \quad T_{trobarX}(n) \in O(n)$$

Anàlisi d'algorismes recursius

Secció 11.7

- En la complexitat temporal d'un algorisme recursiu influeixen:
  - El nombre de crides recursives que genera cada crida al mètode.
  - La forma en que es redueix la talla n del problema en cada crida. Normalment, la reducció és de la forma:
    - n – c (éssent c una constant tal que c ≥1) o
    - n/c (éssent c una constant tal que c >1).
  - El cost de la resta d'operacions que realitza l'algorisme exclosa(es) la(es) crida(es) recursiva(es).

- Per tal d'analitzar la complexitat dels algorismes recursius s'usen les equacions de recurrència.
- Les equacions de recurrència expressen el temps d'execució per als distints casos d'un algorisme recursiu (casos base i recursiu).
- Una vegada es disposa de l'equació de recurrència és possible calcular l'ordre del temps d'execució de diverses formes.
- Una forma de resoldre aquestes equacions és la tècnica de desplegament de recurrències o substitució.

Anàlisi d'algorismes recursius

- Exemple: Càlcul del factorial d'un enter n≥0

```
/** n >= 0 */
public static int factorial(int n) {
    if (n == 0) { return 1; }
    else { return n * factorial(n - 1); }
}
```

  - *Talla*: donada per la variable **n**.
  - *Instàncies*: No.
  - Cas base (**n == 0**): T(n) = 1+1 (comparació i retorn tenen un temps d'execució constant, simplificant: cost unitari, 1).
  - Cas recursiu (**n > 0**): T(n) = 4+T(n-1) (considerant 4 operacions de cost constant: la comparació, l'expressió **n-1**, el producte i el retorn; més el temps de càlcul de **factorial(n-1)**).
  - L'equació de recurrència:

$$T(n) = \begin{cases} k' & n = 0 \\ k + T(n-1) & n > 0 \end{cases}$$

sent k i k' constants positives en alguna unitat de temps

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

expressat en passos de programa (de cost unitari)

Anàlisi d'algorismes recursius

- La tècnica de desplegament de recurrències consisteix en:
  - Substituir les aparicions de T dins de l'equació recursiva fins trobar una forma general que dependixca del nombre d'invocacions recursives, i.

- En el cas del càlcul del `factorial`:

reducció aritmètica de la talla

$$T(n) = \begin{cases} 1 & n = 0 \\ 1 + \mathrm{T}(n-1) & n > 0 \end{cases}$$

T(n) = 1 + T(n-1) = 1 + (1 + T(n-2)) = 2 + (1 + T(n-3)) = ... = i + T(n-i)

on i és el nombre d'invocacions recursives (nombre de substitucions).

- A més, s'ha de verificar quantes crides duen al cas base:

n-i = 0 ⇔ n = i, T(n) = n + T(n-n) = n + T(0) = n + 1

- Si T(n) = n + 1, la complexitat és exactament de l'ordre de n, Θ(n).

Problema 12.

```
/** v.length > 0, 0 <= i < v.length */
public static int maxim(int[] v, int i) {
    if (i == 0) { return v[i]; }
    else {
        int m = maxim(v, i - 1);
        if (m > v[i]) { return m; }
        else { return v[i]; }
    }
}
```

RECORREGUT

Crida inicial: `int max = maxim(v, v.length - 1);`

Talla: nombre d'elements de l'array `v` en consideració; donada per l'expressió `i + 1`. Per facilitar el raonament posterior, anomenem a aquest número n. Açò és, `n = i + 1`.

Instàncies: Es tracta d'un problema de recorregut (recursiu) pel que, per a una mateixa talla del problema, **no** hi ha instàncies significatives.

$$T(n) = \begin{cases} T(n-1) + k & \text{si } n > 1 \\ k' & \text{si } n = 1 \end{cases}$$

sent k i k' constants positives en alguna unitat de temps. Resolent per substitució: $T(n) = T(n-1)+k = T(n-2)+2k = T(n-3)+3k = \dots = T(n-i)+i\,k$. En arribar al cas base, per a talla 1, $n-i = 1 \rightarrow i = n-1$. Amb el que $T(n) = k'+(n-1)k$

$T(n) \in \Theta(n)$, és a dir, el cost temporal depèn linealment de la talla del problema.

```java
/** 0 <= ini <= fi <= v.length - 1 o ini > fi */
public static boolean capicua(String[] v, int ini, int fi) {
    if (ini >= fi) { return true; }
    else if (v[ini].equals(v[fi])) {
        return capicua(v, ini + 1, fi - 1);
    }
    else { return false; }
}
```

CERCA

Crida inicial: `boolean b = capicua(v, 0, v.length - 1);`

Talla: nombre d'elements de l'array `v` en consideració, donada per l'expressió `fi-ini+1`. Per facilitar el raonament posterior, anomenem a aquest número n. Açò és, `n = fi-ini+1`.

Instàncies: Es tracta d'un problema de cerca (recursiva) on la propietat a cercar és `!v[ini].equals(v[fi])`. Per tant, **sí** que hi ha instàncies significatives, que són:

- cas millor: `v` no és capicúa perque `v[ini].equals(v[fi]) == false` per a `ini = 0` i `fi = v.length - 1`.
- cas pitjor: `v` és capicúa.

En el cas millor, per a qualsevol talla, es té: $T^m(n) = c$ sent **c** una constant positiva en alguna unitat de temps.

En el cas pitjor, plantegem l'equació de recurrència:

$$T^m(n) \in \Theta(1) \Rightarrow T(n) \in \Omega(1)$$

$$T^p(n) \in \Theta(n) \Rightarrow T(n) \in O(n)$$

$$T^p(n) = \begin{cases} T^p(n-2) + k & \text{si } n > 1 \\ k' & \text{si } 0 <= n <= 1 \end{cases}$$

sent k i k' constants positives en alguna unitat de temps. Resolent per substitució: $T^p(n) = T^p(n-2) + k = T^p(n-4)+2k = T^p(n-6)+3k = \ldots = T^p(n-2i) + ik$. En arribar al cas base, per a talla 0, $n-2i = 0 \rightarrow n = 2i \rightarrow i = n/2$. Amb el que $T^p(n) = k' + k*n/2$

```java
/** v.length > 0, 0 <= i <= j <= v.length - 1 */
public static int maxim(int[] v, int i, int j) {
    if (i == j) { return v[i]; }
    else {
        int m = (i + j) / 2;
        int maxEsq = maxim(v, i, m);
        int maxDre = maxim(v, m + 1, j);
        if (maxEsq > maxDre) { return maxEsq; }
        else { return maxDre; }
    }
}
```

RECORREGUT

Crida inicial: `int max = maxim(v, 0, v.length - 1);`

Talla: Nombre d'elements de v en consideració, donat per $j-i+1 = n$

Instàncies: No

$$T(n) = \begin{cases} 2T(n/2) + k & \text{si } n > 1 \\ k' & \text{si } n = 1 \end{cases}$$

sent k i k' constants positives en alguna unitat de temps

$$T(n) = 2T(n/2) + k = 2^2T(n/2^2) + 3k = 2^3T(n/2^3) + 7k =$$

$$= \ldots = 2^iT(n/2^i) + (2^i-1)k = \ldots = nk' + (n-1)k \in \Theta(n)$$

$$n/2^i = 1 \rightarrow n = 2^i$$

Problema 14.

```
/** 0 < a i 0 <= b */
public static int potencia(int a, int b) {
    if (b == 0) { return 1; }
    else if (b == 1) { return a; }
    else if (b % 2 == 0) { return potencia(a, b / 2) * potencia(a, b / 2); }
    else { return potencia(a, b / 2) * potencia(a, b / 2) * a; }
}
```

Talla: El valor de l'exponent, donat per b
Instàncies: No

$$T_{potencia}(b) = \begin{cases} 2T_{potencia}(b/2) + k & \text{si } b > 1 \\ k' & \text{si } b = 1 \text{ o } b = 0 \end{cases}$$

sent k i k' constants positives en alguna unitat de temps

$$T(b) = 2T(b/2) + k = 2^2T(b/2^2) + 3k = 2^3T(b/2^3) + 7k =$$
$$= \ldots = 2^iT(b/2^i) + (2^i-1)k = \ldots = bk' + (b-1)k \in \Theta(b)$$

$$b/2^i = 1 \rightarrow b = 2^i$$

Problema 14.

```
/** 0 < a i 0 <= b */
public static int potencia(int a, int b) {
    if (b == 0) { return 1; }
    else if (b == 1) { return a; }
    else {
        int p = potencia(a, b / 2);
        if (b % 2 == 0) { return p * p; }
        else { return p * p * a; }
    }
}
```

Talla: El valor de l'exponent, b

Instàncies: No

reducció geomètrica de la talla

$$T_{potencia}(b) = \begin{cases} T_{potencia}(b/2) + k & \text{si } b > 1 \\ k' & \text{si } b = 1 \text{ o } b = 0 \end{cases}$$

sent k i k' constants positives en alguna unitat de temps

$$T(b) = T(b/2) + k = T(b/2^2) + 2k = T(b/2^3) + 3k =$$
$$= \ldots = T(b/2^i) + ik = \ldots = k' + (\log_2 b)k \in \Theta(\log b)$$

$$b/2^i = 1 \rightarrow b = 2^i \rightarrow i = \log_2 b$$

```
/** n >= 1 */
public static void dibuixaTri(int n) {
    if (n == 1) { System.out.println('*'); }
    else {
        for (int i = 1; i <= n; i++) {
            System.out.print('*');
        }
        System.out.println();
        dibuixaTri(n - 1);
    }
}
```

Per exemple, si n = 4:

```
****
***
**
*
```

Talla: El número de línies, donat per n

Instàncies: No

$$T_{dibuixaTri}(n) = \begin{cases} T_{dibuixaTri}(n-1) + n + k & \text{si } n > 1 \\ k' & \text{si } n = 1 \end{cases}$$

sent k i k' constants positives en alguna unitat de temps

$$T(n) = T(n-1) + n + k = T(n-2) + (n-1) + n + 2k = T(n-3) + (n-2) + (n-1) + n + 3k =$$

$$= \ldots = T(n-i) + (n-(i-1)) + \ldots + (n-2) + (n-1) + n + ik = \ldots =$$

$$\boxed{n-i = 1 \rightarrow i = n-1}$$

$$= k' + \boxed{1 + 2 + \ldots + (n-2) + (n-1) + n} + (n-1)k$$

$$\sum_{j=1..n} j = n(n+1)/2$$

$$T_{dibuixaTri}(n) \in \Theta(n^2)$$

poli [formaT] PRG > EXÀMENS 1B - T2. Qüestionari: cost recursiu

Solució des del 01/03/2020 a les 15:00

Anàlisi d'algorismes d'ordenació

Capítol 12

http://www.sorting-algorithms.com/

http://math.hws.edu/eck/js/sorting/sortlab-info.html

- Basats en comparacions
  - Algorismes directes
    - Inserció directa
    - Selecció directa
    - Intercanvi directe o "de la bambolla"
  - Algorismes ràpids
    - Mescla ("MergeSort")
    - Partició ("QuickSort")
    - Monticle ("HeapSort")
- No basats en comparacions
  - Compte de freqüències ("CountingSort")
  - Residus ("RadixSort")
  - Cubetes ("BucketSort")

- **Descarrega** (del Tema 2 de PoliformaT) el fitxer ***exercicisT2.jar*** en una carpeta ***PRG/Tema 2*** dins del teu ***disc W***
- Des de l'opció **Projecte** de ***BlueJ***, usa l'opció **Open ZIP/JAR...** per tal d'obrir aquest com un projecte ***BlueJ*** i prepara't per usar-lo

Algorisme de selecció directa

Secció 12.1

- L'algorisme d'ordenació per selecció consisteix en:
  - Seleccionar el mínim element de l'array i intercanviar-lo amb el primer.
  - Seleccionar el mínim de la resta de l'array i intercanviar-lo amb el segon.
  - I així successivament...
- Algorisme de selecció directa (*Selection Sort*):

http://www.youtube.com/watch?v=boOwArDShLU

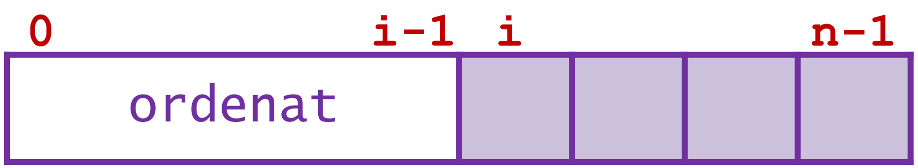
Introduction to Computer Science using Java. Part 14. Sorting and Searching
[Chapter 90. Selection Sort with Integers]()

5 3 4 1 2

https://algorithms.tutorialhorizon.com/selection-sort-java-implementation/selection-sort-gif/

Algorisme de selecció directa

- L'estratègia d'ordenació per selecció directa es pot descriure com:
  - Per a tot `i` des de `0` fins `n–2` fer:
  1. Trobar la posició del valor mínim al subarray `v[i..n-1]`
  2. Intercanviar el mínim amb `v[i]`
  - Els elements del subarray `v[0..i-1]` estan ordenats i els seus valors són inferiors als del subarray `v[i..n-1]`.



- L'operació de trobar el mínim al subarray es pot plantejar com un recorregut.

Algorisme de selecció directa

- **Exemple:** Ordenació de l'array {16, 54, 7, 98, 2, 66, 30, 14}

{16, 54, 7, 98, 2, 66, 30, 14}

selecciona 2 i l'intercanvia amb 16

{**2**, 54, 7, 98, 16, 66, 30, 14}

selecciona 7 i l'intercanvia amb 54

{**2**, **7**, 54, 98, 16, 66, 30, 14}

selecciona 14 i l'intercanvia amb 54

{**2**, **7**, **14**, 98, 16, 66, 30, 54}

selecciona 16 i l'intercanvia amb 98

{**2**, **7**, **14**, **16**, 98, 66, 30, 54}

selecciona 30 i l'intercanvia amb 98

{**2**, **7**, **14**, **16**, **30**, 66, 98, 54}

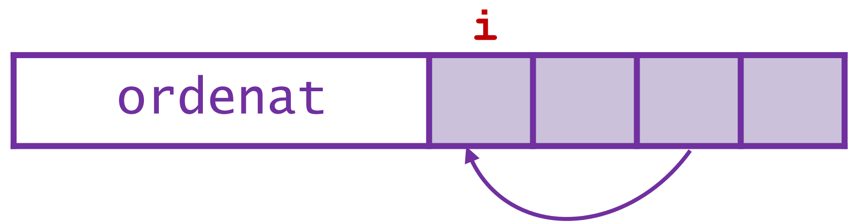
selecciona 54 i l'intercanvia amb 66

{**2**, **7**, **14**, **16**, **30**, **54**, 98, 66}

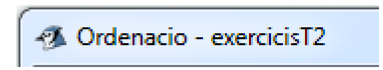
selecciona 66 i l'intercanvia amb 98

{**2**, **7**, **14**, **16**, **30**, **54**, **66**, **98**}

Algorisme de selecció directa



- Implementació:




```java
public static void selDirecta(int[] v) {
    for (int i = 0; i < v.length - 1; i++) {
        // calcular la posició del mínim de v[i..v.length-1]
        int pMin = i;
        for (int j = i + 1; j < v.length; j++) {
            if (v[j] < v[pMin]) { pMin = j; }
            // en pMin està la posició del mínim de v[i..j]
        }
        // en pMin està la posició del mínim
        // de v[i..v.length-1]
        // intercanviar v[i] amb v[pMin]
        int aux = v[pMin];
        v[pMin] = v[i];
        v[i] = aux;    // v[0..i] ordenat
    }
    // v[0..v.length-1] ordenat
}
```

Algorisme de selecció directa

- Anàlisi del cost.

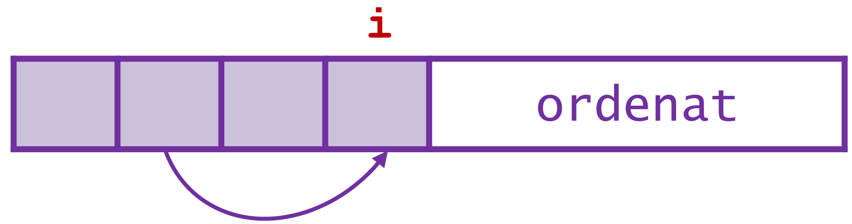
- La *talla* del problema és el nombre d'elements a ordenar, `v.length = n`.
- L'estratègia es basa en dos recorreguts niuats i *no* presenta diferents *instàncies*.
- Considerant com *instrucció crítica* `v[j] < v[pMin]` (de cost unitari)

$$T(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} (n-i-1) = \frac{n(n-1)}{2} \ i.c. \in \Theta(n^2)$$

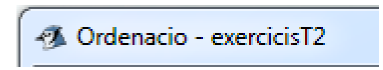
$$\sum_{i=0}^{n-2} (n-i-1) = n(n-1) - \boxed{\sum_{i=0}^{n-2} i} - (n-1) =$$

$$(n^2 - 2n + 1) - \left[\boxed{\frac{n(n+1)}{2} - (n-1) - n} \right] = n^2 - \frac{n(n+1)}{2} = \frac{n^2 - n}{2}$$

Algorisme de selecció directa



- Implementació




```
public static void selDirecta(int[] v) {
    for (int i = v.length - 1; i > 0; i--) {
        // calcular la posició del màxim de v[0..i]
        int pMax = 0;
        for (int j = 1; j <= i; j++) {
            if (v[j] > v[pMax]) { pMax = j; }
            // en pMax està la posició del màxim de v[0..j]
        }
        // en pMax està la posició del màxim de v[0..i]
        // intercanviar v[i] amb v[pMax]
        int aux = v[pMax];
        v[pMax] = v[i];
        v[i] = aux;      // v[i..v.length-1] ordenat
    }
    // v[0..v.length-1] ordenat
}
```

Algorisme d'inserció directa

Secció 12.2

- L'algorisme divideix l'array en una part ordenada i un altra no ordenada:
  - Inicialment, la part ordenada consta d'un únic element (el que ocupa la primera posició).
  - Els elements s'insereixen un a un des de la part no ordenada a l'ordenada.
    - Seleccionar l'element `v[1]` de l'array i situar-lo de manera ordenada amb el que ocupa la posició `0`.
    - Seleccionar l'element `v[2]` de l'array i situar-lo de manera ordenada entre els que ocupen les posicions `0` i `1`.
    - Repetir situant l'element `v[i]` de manera ordenada al subarray comprès entre les posicions `0` i `i-1`.
  - Finalment, la part ordenada abarca tot l'array.

6 5 3 1 8 7 2 4

https://commons.wikimedia.org/wiki/File:Insertion-sort-example-300px.gif

- Algorisme d'inserció directa (*Insertion Sort*)

http://www.youtube.com/watch?v=gTxFxgvZmQs&feature=related

Introduction to Computer Science using Java. Part 14. Sorting and Searching
[Chapter 91. Insertion Sort with Integers]()

Algorisme d'inserció directa

- L'estratègia d'ordenació per inserció directa es pot descriure com:
  - Per a tot `i` des d' `1` fins `n-1` fer:

    Inserir l'element `v[i]` de manera ordenada en `v[0..i-1]`

- L'operació d'inserir de manera ordenada `v[i]` en el subarray `v[0..i-1]` es pot realitzar:
  1. Començant en `i-1`, es cerca seqüencialment el primer element menor o igual a `v[i]`. Siga `j` la posició d'aquest element (`-1` si no hi ha cap).
  2. Es desplacen una posició cap a la dreta tots els elements des de `j+1` fins `i-1`.
  3. S' assigna a `v[j+1]` el que hi ha en `v[i]`.

Els passos 1 i 2 poden fer-se de manera combinada.

Algorisme d'inserció directa

- **Exemple:** Ordenació de l'array {16, 7, 54, 98, 2, 66, 30, 14}

{<u>16,</u> 7, 54, 98, 2, 66, 30, 14} ← inicialment, [0..0] està ordenat

{<u>7,</u> <u>16,</u> 54, 98, 2, 66, 30, 14} ← ordenats [0..1], s'ha inserit 7

{<u>7,</u> <u>16,</u> <u>54,</u> 98, 2, 66, 30, 14} ← ordenats [0..2], sense insercions

{<u>7,</u> <u>16,</u> <u>54,</u> <u>98,</u> 2, 66, 30, 14} ← ordenats[0..3], sense insercions

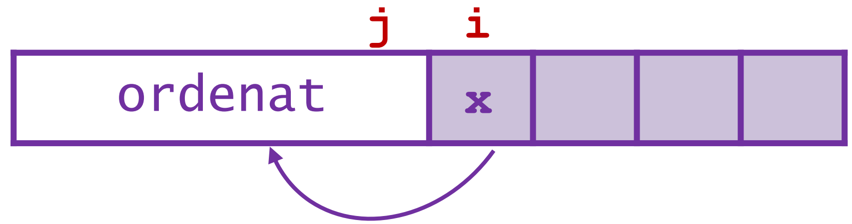
{<u>2,</u> <u>7,</u> <u>16,</u> <u>54,</u> <u>98,</u> 66, 30, 14} ← ordenats [0..4], s'ha inserit 2

{<u>2,</u> <u>7,</u> <u>16,</u> <u>54,</u> <u>66,</u> <u>98,</u> 30, 14} ← ordenats [0..5], s'ha inserit 66

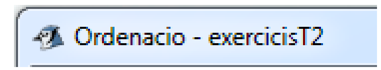
{<u>2,</u> <u>7,</u> <u>16,</u> <u>30,</u> <u>54,</u> <u>66,</u> <u>98,</u> 14} ← ordenats [0..6], s'ha inserit 30

{<u>2,</u> <u>7,</u> <u>14,</u> <u>16,</u> <u>30,</u> <u>54,</u> <u>66,</u> <u>98</u>} ← ordenats [0..7], s'ha inserit 14

Algorisme d'inserció directa



- Implementació




```
public static void insDirecta(int[] v) {
    for (int i = 1; i <= v.length - 1; i++) {
        int x = v[i]; // element a inserir
        int j = i - 1;  // fi de la part ordenada
        // cercar en la part ordenada,
        // desplaçant cap a la dreta els elements majors que x
        while (j >= 0 && v[j] > x) {
            v[j + 1] = v[j];
            j--;
        }
        v[j + 1] = x; // assignar x en la part ordenada
    }
}
```

Algorisme d'inserció directa

- Anàlisi del cost.
- La *talla* del problema és el nombre d'elements a ordenar, `v.length = n`.
- L'estructura de l'algorisme és un recorregut combinat amb una cerca niuada.
- El bucle intern de cerca no sempre es repeteix completament (es deté en trobar la posició correcta). És a dir, s'estalvien alguns passos.
- En aquest cas, es pot escollir com *instrucció crítica* (de cost unitari) la guarda del bucle intern: `j >= 0 && v[j] > x`
- Cas millor: Quan l'array està ordenat el bucle intern no itera cap vegada, la comparació sempre s'avalua a fals. En aquest cas, l'algorisme executa la instrucció crítica tantes vegades com passades fa el bucle extern.

$$T^m(n) = \sum_{i=1}^{n-1} 1 = n-1 \; i.c. \in \Theta(n) \Rightarrow T(n) \in \Omega(n)$$

- Cas pitjor: El nombre de vegades que s'executa el bucle intern és el màxim possible, és a dir, `i` vegades en cada iteració del bucle extern. Es tracta del cas en el que l'array està ordenat en sentit invers.

$$T^p(n) = \sum_{i=1}^{n-1}\left(\sum_{j=0}^{i-1} 1 + 1\right) = \sum_{i=1}^{n-1}(i+1) = \frac{n(n+1)}{2} - 1 \; i.c. \in \Theta(n^2) \Rightarrow T(n) \in O(n^2)$$

Algorisme d'intercanvi directe o de la bambolla

Secció 12.3

- Algorisme d'ordenació senzill i ineficient. Consisteix en:
  - Recórrer l'array comparant parells d'elements consecutius.
  - Intercanviar les seues posicions si no estan en l'ordre correcte.
- Algorisme d'intercanvi directe (*Bubble Sort*)

  http://www.youtube.com/watch?v=1JvYAXT_064&feature=related

  Barack Obama - Computer Science Question:
  https://www.youtube.com/watch?v=k4RRi_ntQc8

6 5 3 1 8 7 2 4

https://commons.wikimedia.org/wiki/File:Bubble-sort-example-300px.gif

Algorisme d'intercanvi directe o de la bambolla

- Implementació 1: recorregut que inclou un recorregut descendent

Ordenacio - exercicisT2

```
public static void bambolla(int[] v) {
    for (int i = 0; i < v.length - 1; i++) {
        for (int j = v.length - 1; j > i; j--) {
            if (v[j - 1] > v[j]) {
                int x = v[j - 1];
                v[j - 1] = v[j];
                v[j] = x;
            }
        }
    }
}
```

En la iteració **i**, el **i+1**-èsim **mínim** se situa en la posició **i** de l'array

Traça per a **i = 0**, des de **j = v.length - 1** fins **1**

16	54	7	98	2	66	30	14
16	54	7	98	2	66	14	30
16	54	7	98	2	14	66	30
16	54	7	98	2	14	66	30
16	54	7	2	98	14	66	30
16	54	2	7	98	14	66	30
16	2	54	7	98	14	66	30
2	16	54	7	98	14	66	30

- Anàlisi del cost.
- La *talla* del problema és el nombre d'elements a ordenar, **v.length = n**.
- Considerant com *instrucció crítica* **v[j-1] > v[j]** (de cost unitari):

$$T(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} (n-i-1) = n(n-1) - \left[\frac{n(n+1)}{2} - (n-1) - n \right] - (n-1) = \frac{n(n-1)}{2} \; i.c. \in \Theta(n^2)$$

Algorisme d'intercanvi directe o de la bambolla

Traça per a `i = 0`,
des de `j = v.length - 1` fins 1

16	54	7	98	2	66	30	14
16	54	7	98	2	66	14	30
16	54	7	98	2	14	66	30
16	54	7	98	2	14	66	30
16	54	7	2	98	14	66	30
16	54	2	7	98	14	66	30
16	2	54	7	98	14	66	30
2	16	54	7	98	14	66	30

Traça per a `i = 1`,
des de `j = v.length - 1` fins 2

2	16	54	7	98	14	66	30
2	16	54	7	98	14	30	66
2	16	54	7	98	14	30	66
2	16	54	7	14	98	30	66
2	16	54	7	14	98	30	66
2	16	7	54	14	98	30	66
2	7	16	54	14	98	30	66

Traça per a `i = 2`,
des de `v.length - 1` fins 3

2	7	16	54	14	98	30	66
2	7	16	54	14	98	30	66
2	7	16	54	14	30	98	66
2	7	16	54	14	30	98	66
2	7	16	14	54	30	98	66
2	7	14	16	54	30	98	66

Traça per a `i = 3`,
des de `v.length - 1` fins 4

2	7	14	16	54	30	98	66
2	7	14	16	54	30	66	98
2	7	14	16	54	30	66	98
2	7	14	16	30	54	66	98
2	7	14	16	30	54	66	98

Traça per a `i = 4`,
des de `v.length - 1` fins 5

2	7	14	16	30	54	66	98
2	7	14	16	30	54	66	98
2	7	14	16	30	54	66	98
2	7	14	16	30	54	66	98
2	7	14	16	30	54	66	98

Traça per a `i = 5`,
des de `v.length - 1` fins 6

2	7	14	16	30	54	66	98
2	7	14	16	30	54	66	98
2	7	14	16	30	54	66	98

Traça per a `i = 6`,
des de `v.length - 1` fins 7

2	7	14	16	30	54	66	98
2	7	14	16	30	54	66	98

Algorisme d'intercanvi directe o de la bambolla

Ordenacio - exercicisT2

- Implementació 2: recorregut que inclou un recorregut ascendent

```java
public static void bambolla(int[] v) {
    for (int i = 0; i < v.length - 1; i++) {
        for (int j = 0; j < v.length - 1 - i; j++) {
            // comparar parells d'elements consecutius
            if (v[j] > v[j + 1]) {
                // si parell desordenat,
                // aleshores intercanvi
                int x = v[j];
                v[j] = v[j + 1];
                v[j + 1] = x;
            }
        }
    }
}
```

En la iteració **i**, el **i+1**-èsim **màxim** se situa en la posició **v.length−1−i** de l'array

Traça per a **i = 0**, des de **j = 0** fins **6**

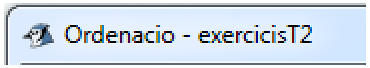
16	54	7	98	2	66	30	14
16	54	7	98	2	66	30	14
16	7	54	98	2	66	30	14
16	7	54	98	2	66	30	14
16	7	54	2	98	66	30	14
16	7	54	2	66	98	30	14
16	7	54	2	66	30	98	14
16	7	54	2	66	30	14	98

- Anàlisi del cost.
- La *talla* del problema és el nombre d'elements a ordenar, **v.length = n**.
- Considerant com *instrucció crítica* **v[j] > v[j+1]** (de cost unitari):

$$T(n) = \sum_{i=1}^{n-2} \sum_{j=0}^{n-2-i} 1 = \sum_{i=1}^{n-2} (n-i-1) = n(n-2) - \left[\frac{n(n+1)}{2} - n - (n-1)\right] - (n-2)\ i.c. \in \Theta(n^2)$$

Algorisme d'intercanvi directe o de la bambolla

- Una millora consisteix en afegir una bandera o "flag" que indique si s'ha produït algun intercanvi durant el recorregut:
  - Si no se n'ha produït cap, l'array està ordenat i es pot acabar.
  - Amb aquesta millora, $T^m(n) \in \Theta(n)$
  - Però en promedi el seu cost continua éssent quadràtic.




```java
public static void bambollaAmbFlag(int[] v) {
    boolean ordenat = false;
    for (int i = 0; i < v.length - 1 && !ordenat; i++) {
        int cont = 0;
        for (int j = v.length - 1; j > i; j--) {
            if (v[j - 1] > v[j]) {
                int x = v[j - 1]; v[j - 1] = v[j]; v[j] = x;
                cont++;
            }
        }
        if (cont == 0) { ordenat = true; }
    }
}
```

```java
public static void selDirecta(int[] v) {
    for (int i = 0; i < v.length - 1; i++) {
        int pMin = i;
        for (int j = i + 1; j < v.length; j++) {
            if (v[j] < v[pMin]) { pMin = j; }
        }
        int aux = v[pMin];
        v[pMin] = v[i];
        v[i] = aux;
    }
}
```

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

```java
public static void insDirecta(int[] v) {
    for (int i = 1; i <= v.length - 1; i++) {
        int x = v[i];
        int j = i - 1;
        while (j >= 0 && v[j] > x) {
            v[j + 1] = v[j];
            j--;
        }
        v[j + 1] = x;
    }
```

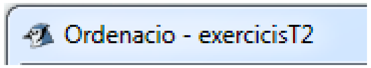
$$T(n) \in \Omega(n)$$

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

```java
public static void bambolla(int[] v) {
    for (int i = 0; i < v.length - 1; i++) {
        for (int j = v.length - 1; j > i; j--) {
            if (v[j - 1] > v[j]) {
                int x = v[j - 1];
                v[j - 1] = v[j];
                v[j] = x;
            }
        }
    }
}
```

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

Problema. L'algorisme d'ordenació que segueix és una versió del de la bambolla que només ordena els **m** elements més menuts ($1 \leq$ **m** $\leq$ `v.length`) d'un array d'enters.




```
public static void bambollaM(int[] v, int m) {
    for (int i = 0; i < m; i++) {
        for (int j = v.length - 1; j > i; j--) {
            if (v[j - 1] > v[j]) {
                int x = v[j - 1];
                v[j - 1] = v[j];
                v[j] = x;
            }
        }
    }
}
```

Talla: el nombre d'elements de l'array `v` i el nombre d'elements que es volen ordenar `m`. L'expressió que la defineix és (`v.length`, `m`). Per facilitar el raonament posterior, `n` = `v.length`. Amb lo qual la talla serà `(n, m)`.

Instàncies: Es tracta d'un problema de recorregut (dos recorreguts niuats) pel que per a una mateixa talla del problema no hi ha instàncies significatives.

Instrucció crítica: Triem la condició `v[j - 1] > v[j]` que, en qualsevol cas, s'executa 1 vegada menys que la guarda del bucle (lo que és despreciable a efectes de cost asimptòtic). Considerem aquesta instrucció de cost unitari.

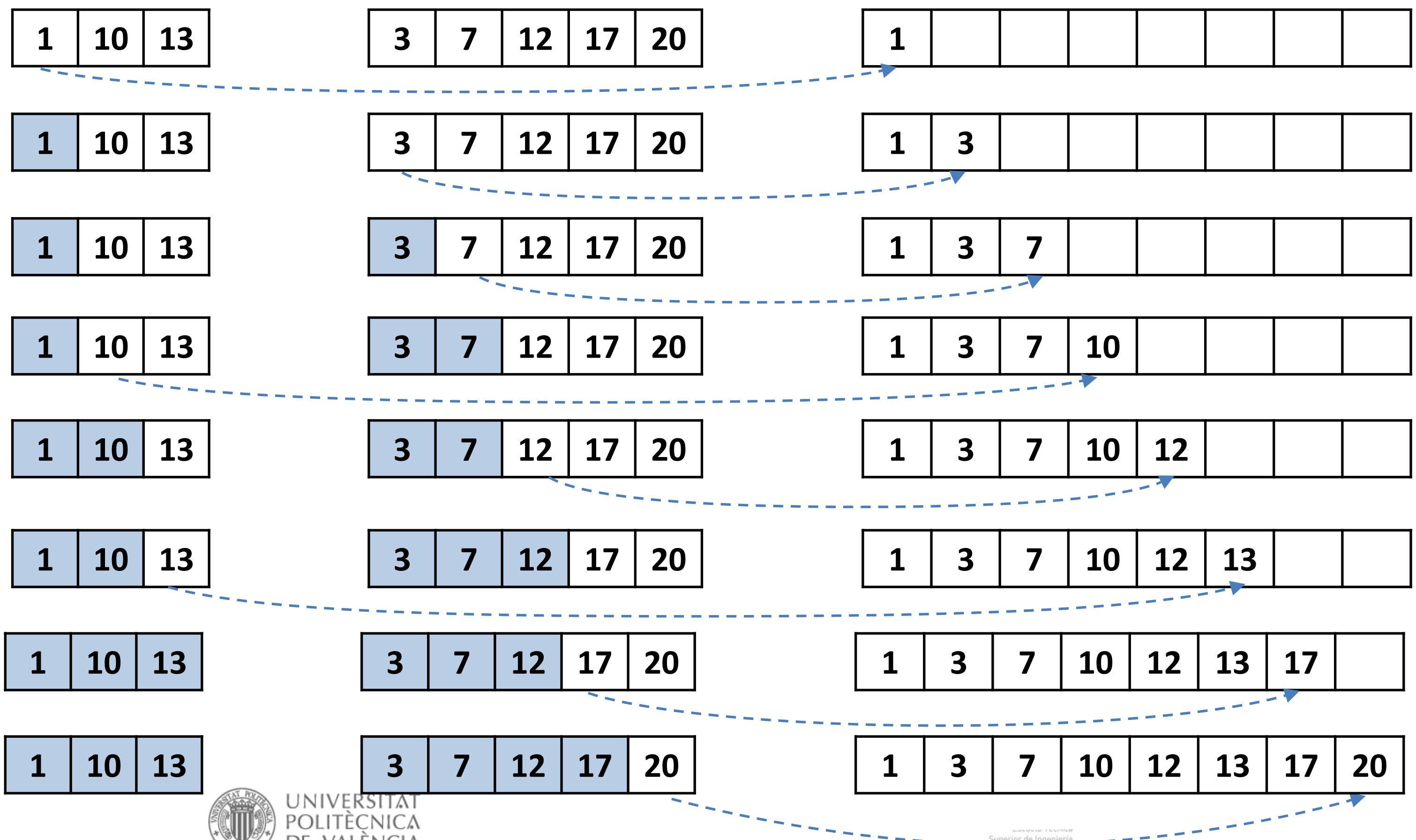
$T(n, m) = \sum_{i=0}^{m-1} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{m-1} (n - i - 1) = n * m - \sum_{i=0}^{m-1} i - m = n * m - \frac{m(m+1)}{2}$ i.c.

Podem observar que, en funció del valor de m, el cost temporal variarà des d'un creixement lineal (només s'ordena 1 element, m = 1 ⇨ $\Theta(n)$) a un creixement quadràtic (s'ordena tot l'array, m = n ⇨ $\Theta(n^2)$) amb el nombre d'elements de l'array. $T(n, m) \in \Theta(n * m)$

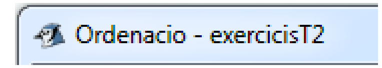
Algorisme de mescla natural

Secció 12.5.1

- Donats dos arrays a i b ordenats de longituds no necessàriament iguals, s'han de fusionar en un nou array destí c de tal forma que quede ordenat.
- El següent algorisme resol el problema en dues fases:
  1. Un bucle compara els elements dels dos arrays i els copia de manera ordenada en l'array destí. Aquest bucle acaba en arribar al final d'un dels arrays.
  2. Un segon bucle copia els restants elements de l'altre array al final de l'array destí.



Algorisme de mescla natural



- Implementació

```
/** a i b estan ordenats, c.length és a.length + b.length */
public static void mesclaNatural(int[] a, int[] b, int[] c) {
    int i = 0, l = a.length, j = 0, m = b.length, k = 0;
    while (i < l && j < m) {
        if (a[i] < b[j]) { c[k] = a[i]; i++; }
        else { c[k] = b[j]; j++; }
        k++;
    }
    for (int r = i; r < l; r++)  { c[k] = a[r]; k++; }
    for (int r = j; r < m; r++)  { c[k] = b[r]; k++; }
}
```

- La *talla* n del problema ve determinada per la suma de les llargàries dels arrays a mesclar (**`a.length`** + **`b.length`**).
- Per a comptar els passos de l'algorisme, es poden comptar les voltes que s'executa **`k++`**, l'increment de l'índex d'accès a l'array destí.
- Llavors, s'executa **`l + m`** vegades, on **`l`** és **`a.length`** i **`m`** és **`b.length`**.

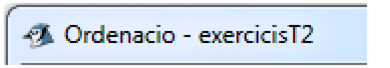
$$T(\mathbf{n}) = \mathbf{l} + \mathbf{m} \in \Theta(\mathbf{l} + \mathbf{m}) \text{ , és a dir, } T(\mathbf{n}) \in \Theta(\mathbf{n})$$

Algorisme de mescla natural

- Versió de mescla natural especialitzada: mescla en `v[ini..fi]` de `v[ini..meitat]` i `v[meitat + 1..fi]`
  - Mescla de les dues meitats ordenades d'un array (`v`) fent ús d'un array auxiliar (`aux`).
  - Els paràmetres del mètode són, a més de l'array, els índexs que delimiten les dues meitats (`ini`, `meitat` i `fi`).
  - Si es vol ordenar tot l'array, en la crida inicial `ini = 0`, `fi = v.length-1` i `meitat = (fi + ini)/2 = (v.length-1)/2`.

```
public static void mesclaNatural2(int[] v, int ini, int meitat, int fi)
```

Algorisme de mescla natural




```
/** Versió de mescla natural especialitzada:
/*  mescla en v[ini..fi] de v[ini..meitat] i v[meitat+1..fi] */
public static void mesclaNatural2(int[] v,int ini,int meitat,int fi) {
    int[] aux = new int[fi - ini + 1];
    int i = ini, j = meitat + 1, k = 0;
// bucle que mescla les 2 meitats ordenades de v en aux, fins arribar al final d'una de les 2
    while (i <= meitat && j <= fi) {
        if (v[i] < v[j]) { aux[k] = v[i]; i++; }
        else { aux[k] = v[j]; j++; }
        k++;
    }
    // bucle que copia els elements restants de la primera meitat al final d'aux
    for (int r = i; r <= meitat; r++) { aux[k] = v[r]; k++; }
    // bucle que copia els elements restants de la segona meitat al final d'aux
    for (int r = j; r <= fi; r++) { aux[k] = v[r]; k++; }
    // bucle que copia tots els elements d'aux en v
    int s = 0;
    for (i = ini; i <= fi; i++) { v[i] = aux[s]; s++; }
}
```

- La *talla* n del problema ve determinada pel nombre d'elements de l'array a mesclar (**fi-ini**+1).

$T_{mesclaNatural2}(n) = \sum_{i=ini}^{meitat} 1 + \sum_{j=meitat+1}^{fi} 1 + \sum_{i=ini}^{fi} 1 =$

$(meitat - ini + 1) + (fi - (meitat + 1) + 1) + (fi - ini + 1) =$

$(fi - ini + 1) + (fi - ini + 1) = n + n$ i.c.

$$\boxed{T_{mesclaNatural2}(n) = n + n \in \Theta(n)}$$

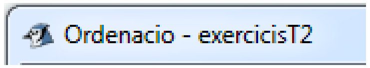
Algorisme MergeSort

Secció 12.4

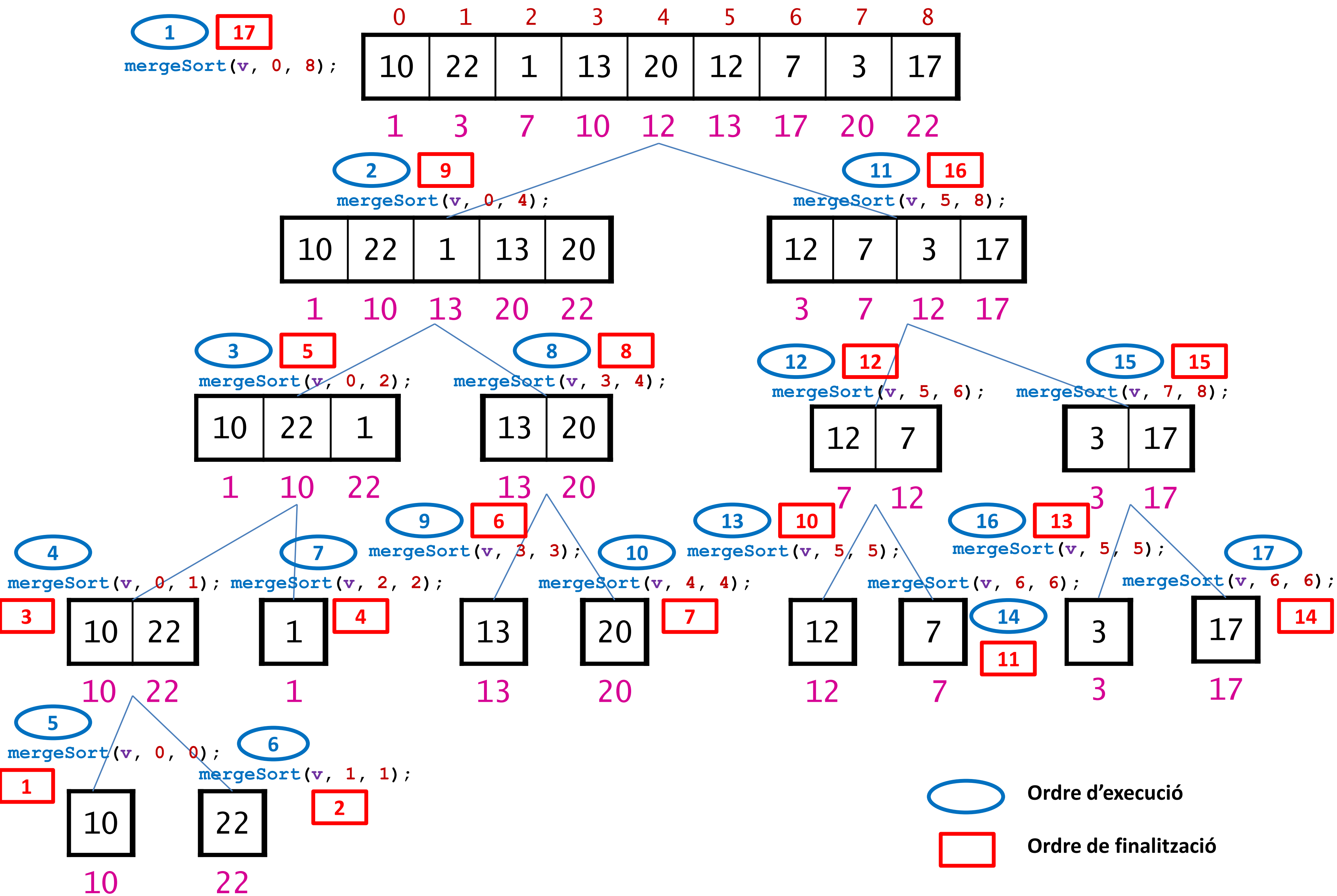
- Algorisme d'ordenació que consisteix en:
  - Dividir l'array en dues parts de la mateixa talla.
  - Ordenar per separat cadascuna de les parts (mitjançant crides recursives).
  - Mesclar ambdues parts mantenint l'ordenació.
- Algorisme de mescla directa (MergeSort)

  http://www.youtube.com/watch?v=HA6ghMlYuO4

- Implementació




```
public static void mergesort(int[] v, int ini, int fi) {
    if (ini < fi) {
        int meitat = (fi + ini) / 2;
        mergesort(v, ini, meitat);
        mergesort(v, meitat + 1, fi);
        // versió de mescla natural especialitzada:
        // mescla en v[ini..fi] de
        // v[ini..meitat] i v[meitat + 1..fi]
        mesclaNatural2(v, ini, meitat, fi);
    }
}
```


1
17
mergeSort(v, 0, 8);
0 1 2 3 4 5 6 7 8
10 22 1 13 20 12 7 3 17
1 3 7 10 12 13 17 20 22
2
9
mergeSort(v, 0, 4);
10 22 1 13 20
1 10 13 20 22
11
16
mergeSort(v, 5, 8);
12 7 3 17
3 7 12 17
3
5
mergeSort(v, 0, 2);
10 22 1
1 10 22
8
8
mergeSort(v, 3, 4);
13 20
13 20
12
12
mergeSort(v, 5, 6);
12 7
7 12
15
15
mergeSort(v, 7, 8);
3 17
3 17
4
3
mergeSort(v, 0, 1);
10 22
10 22
7
4
mergeSort(v, 2, 2);
1
1
9
6
mergeSort(v, 3, 3);
13
13
10
7
mergeSort(v, 4, 4);
20
20
13
10
mergeSort(v, 5, 5);
12
12
14
11
mergeSort(v, 6, 6);
7
7
16
13
mergeSort(v, 5, 5);
3
3
17
14
mergeSort(v, 6, 6);
17
17
5
1
mergeSort(v, 0, 0);
10
10
6
2
mergeSort(v, 1, 1);
22
22
Ordre d'execució
Ordre de finalització

Algorisme MergeSort

- Anàlisi del cost
- *Talla* del problema: nombre d'elements a ordenar, donat per l'expressió fi-ini+1 = n

- Cost de `mesclaNatural2`: $T_{mesclaNatural2}(n) = n + n \in \Theta(n)$

- Cost de `mergesort`:

$$T(n) = \begin{cases} c_1 & 0 \le n \le 1 \\ 2T(n/2) + c_2 n & n > 1 \end{cases}$$

sent c_1 i c_2 constants positives en alguna unitat de temps

$$T(n) = 2T\left(\frac{n}{2}\right) + c_2 n = 2\left[2T\left(\frac{n}{4}\right) + c_2\frac{n}{2}\right] + c_2 n = 4T\left(\frac{n}{4}\right) + 2c_2 n =$$

$$4\left[2T\left(\frac{n}{8}\right) + c_2\frac{n}{4}\right] + 2c_2 n = 8T\left(\frac{n}{8}\right) + 3c_2 n = \ldots = 2^i T\left(\frac{n}{2^i}\right) + ic_2 n$$

$$\frac{n}{2^i} = 1 \to n = 2^i \to i = \log_2 n$$

$$T(n) = c_1 n + c_2 n \log_2 n \Rightarrow T(n) \in \Theta(n \log n)$$

Altres algorismes. Cerca binària

Secció 12.5.2

- Hi ha moltes situacions en les que és necessari fer repetides cerques d'elements dins d'una col.lecció.

- Si la col.lecció de dades no està ordenada, s'ha de realitzar una cerca exhaustiva (és a dir, començar des d'un extrem cap a l'altre, fins que es trobe o s'arribe al final sense trobar-lo): cerca lineal o seqüencial ⇨ Cost lineal

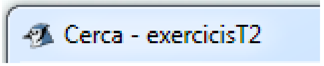
- Si la col.lecció de dades està ordenada, es disposa d'una estratègia molt eficient: la cerca dicotòmica o binària ⇨ Cost logarítmic

- Enunciat del problema. Obtenir la posició d'un enter `x` en un array d'enters `a[ini..fi]` ordenat ascendentment, sent `0 <= ini <= fi < a.length`.

- Estratègia de l'algorisme: examinar la posició central, `meitat = (ini + fi) / 2`, i decidir segons els tres casos possibles:
  - `a[meitat] == x`, aleshores la cerca acaba amb èxit i torna `meitat`.
  - `a[meitat] > x`, aleshores la cerca continua en `a[ini..meitat-1]`.
  - `a[meitat] < x`, aleshores la cerca continua en `a[meitat+1..fi]`.

Si `x` no es troba, torna `-1`.

Altres algorismes. Cerca binària



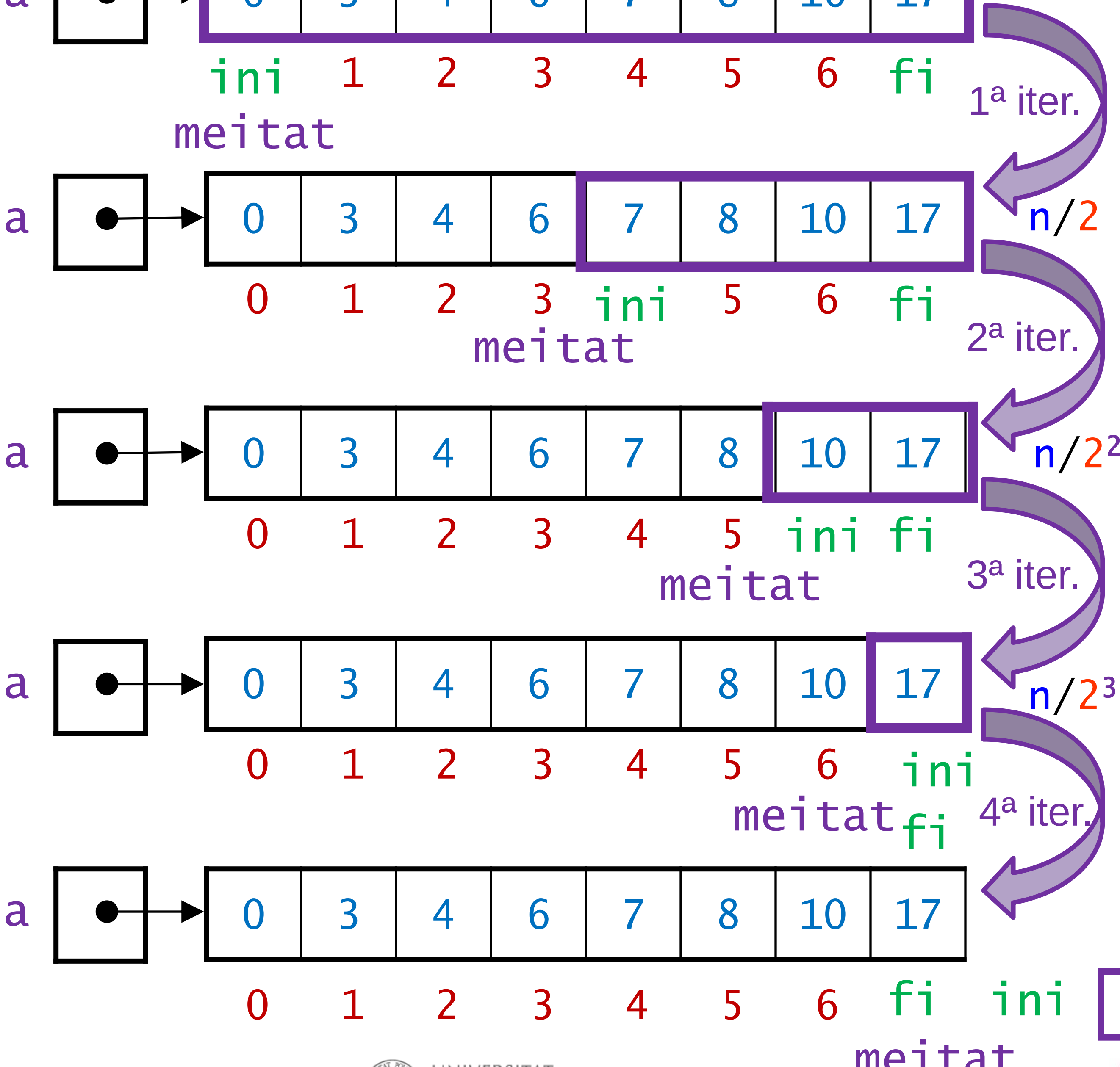
- Implementació cerca binària iterativa.

```
/** Torna la posició de x en a[ini..fi] o -1 si no es troba.
 *  PRECONDICIÓ: a ordenat ascendentment i
 *               0 <= ini <= a.length i -1 <= fi < a.length */
public static int cercaBinIter(int[] a, int x, int ini, int fi) {
    int meitat = 0;
    boolean trobat = false;
    while (ini <= fi && !trobat) {
        meitat = (ini + fi) / 2;
        if (x == a[meitat]) { trobat = true; }
        else if (x < a[meitat]) { fi = meitat - 1; }
        else { ini = meitat + 1; }
    }
    if (trobat) { return meitat; }
    else { return -1; }
}
```

- Si cerquem x en tot l'array a, la crida inicial serà:

```
int pos = cercaBinIter(a, x, 0, a.length - 1);
```

n = a.length   x 20




```
ini = 0,
fi = a.length - 1,
int meitat = 0;
boolean trobat = false;

while(ini<=fi && !trobat)
{ meitat = (ini+fi)/2;
  if (x==a[meitat])
      trobat = true;
  else if (x<a[meitat])
         fi = meitat-1;
  else ini = meitat+1;
}

if (trobat) {
     return meitat;
}
else { return -1; }
```

$n/2^{límit} = 1$

$límit = \log_2 n$

$\boxed{\log_2 n+1 \text{ iteracions}}$

Altres algorismes. Cerca binària

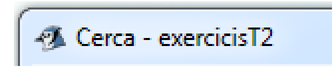
- Anàlisi del cost:
- La *talla* del problema és el nombre d'elements de l'array, `n = fi-ini+1`. Si `ini = 0` i `fi = a.length-1`, `n = a.length`.
- Es distingeixen les següents *instàncies* significatives:
  - Cas millor: L'element a cercar està on es realitza la primera comparació (en la posició central, `x == a[(ini + fi) / 2]`). En aquest cas, el cos del bucle s'executa només una vegada. El cost és constant.

    $$T^m(n) \in \Theta(1) \Rightarrow T(n) \in \Omega(1)$$

  - Cas pitjor: L'element no es troba en l'array. Com en cada passada l'interval on es fa la cerca es redueix a la meitat, el nombre de passades del bucle és $\log_2(n)+1$.

    $$T^p(n) = \log_2(n) + 1 \in \Theta(\log n) \Rightarrow T(n) \in O(\log n)$$

Altres algorismes. Cerca binària

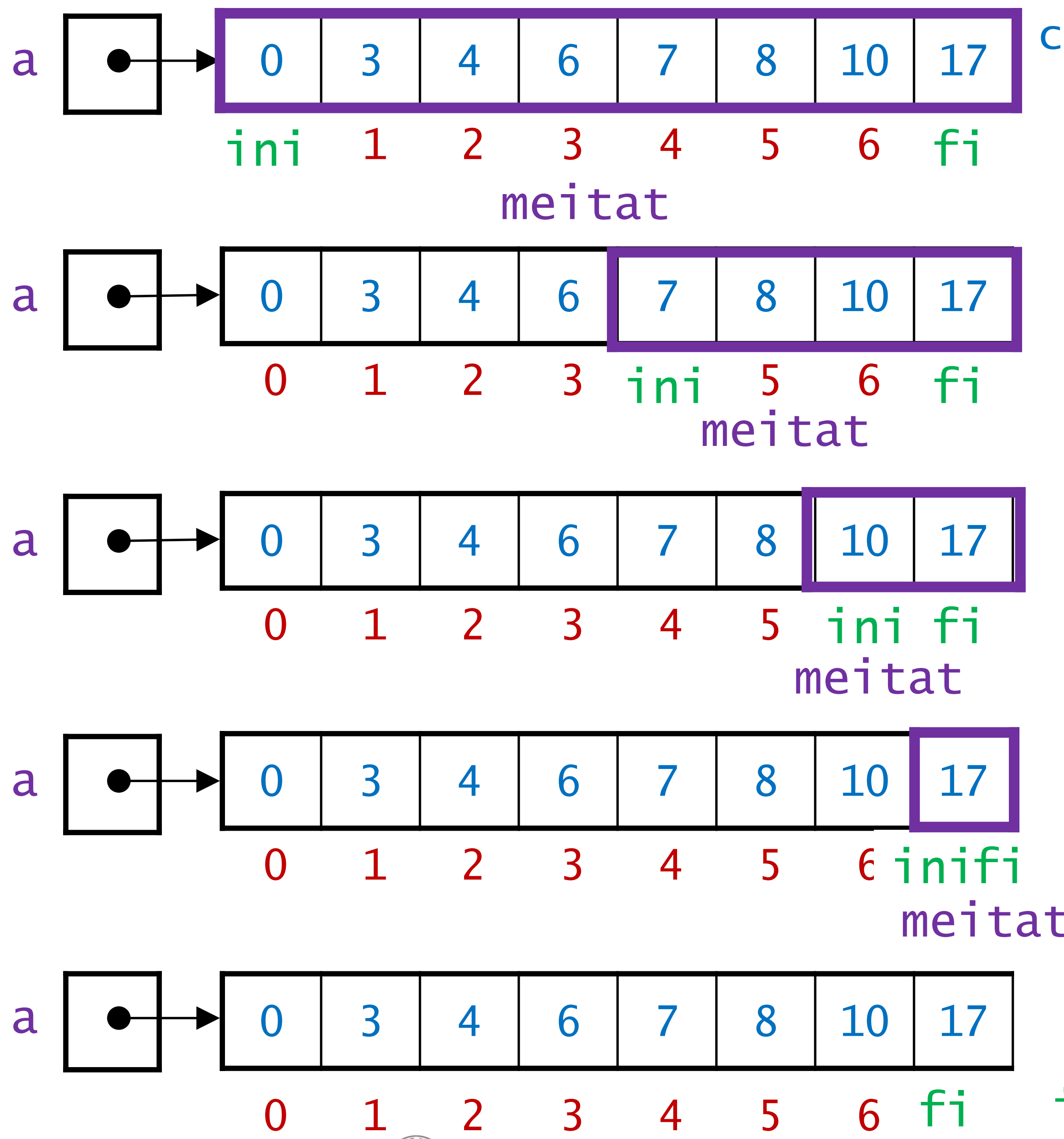


- Implementació cerca binària recursiva.

```
/** Torna la posició de x en a[ini..fi] o -1 si no es troba.
 *  PRECONDICIÓ: a ordenat ascendentment i
 *               0 <= ini <= a.length i -1 <= fi < a.length */
public static int cercaBinRec(int[] a, int x, int ini, int fi) {
    if (ini > fi) { return -1; }
    else {
        int meitat = (ini + fi) / 2;
        if (a[meitat] == x) { return meitat; }
        else if (a[meitat] > x) {
            return cercaBinRec(a, x, ini, meitat - 1);
        }
        else { return cercaBinRec(a, x, meitat + 1, fi); }
    }
}
```

- Si cerquem x en tot l'array a, la crida inicial serà:

```
int pos = cercaBinRec(a, x, 0, a.length - 1);
```



Crida inicial: x 20

`cercaBinRec(a,20,0,7);`

n = v.length

`cercaBinRec(a,20,4,7);`

$n/2$

`cercaBinRec(a,20,6,7);`

$n/2^2$

`cercaBinRec(a,20,7,7);`

$n/2^3$

$n/2^{\text{límit}} = 1$

$\text{límit} = \log_2 n$

`cercaBinRec(a,20,8,7);`

ini

$\boxed{\log_2 n+1 \text{ crides}}$

Altres algorismes. Cerca binària

- Anàlisi del cost.
- La funció de cost depèn de la grandària de l'array en consideració, açò és,

  `n = fi-ini+1`.
- Es distingeixen les següents *instàncies* significatives:
  - Cas millor: L'element a buscar està on es fa la primera comparació (`a[(ini + fi) / 2] == x`). En aquest cas, el cost és constant.

    $$T^m(n) = c \in \Theta(1) \Rightarrow T(n) \in \Omega(1)$$
  - Cas pitjor: L'element no es troba a l'array. La crida recursiva es fa amb la meitat d'elements.

$$T_{cercaBinRec}^{p}(n) = \begin{cases} T_{cercaBinRec}^{p}(n/2) + k & \text{si } n>0 \\ k' & \text{si } n=0 \end{cases}$$

sent k i k' constants positives en alguna unitat de temps

$$T(n) = T(n/2)+k = T(n/2^2)+2k = T(n/2^3)+3k = \ldots = T(n/2^i) + i\,k = \ldots \approx k'+(\log_2 n)k$$

$$n/2^i \approx 1 \rightarrow n \approx 2^i \rightarrow i \approx \log_2 n$$

$$T^p(n) \in \Theta(\log n) \Rightarrow T(n) \in O(\log n)$$

Exemple 12.8: Torres d'Hanoi. Es disposa de tres torres (origen, destí i auxiliar) i un cert nombre de discos de diferents grandàries. Inicialment, tots els discos es troben en la torre origen apilats en forma decreixent segons el seu diàmetre. El problema consisteix en desplaçar-los a la torre destí utilitzant la torre auxiliar i seguint les regles següents: els discos s'han de desplaçar un a un i en cap moment pot haver-hi un disc de major diàmetre situat sobre un més xicotet.

```
public static void hanoi(int n, String orig, String dest, String aux) {
    if (n == 1) { moureDisc(orig, dest); }
    else {
        // Moure n-1 discos de "origen" a "auxiliar"
        // torre auxiliar és "destí"
        hanoi(n - 1, orig, aux, dest);
        // Moure l'últim disc de "origen" a "destí"
        moureDisc(orig, dest);
        // Moure n-1 discos de "auxiliar" a "destí"
        // torre auxiliar és "origen"
        hanoi(n - 1, aux, dest, orig);
    }
}
```

Talla: Nombre de discos, donat per n

Instàncies: No

$$T_{hanoi}(n) = \begin{cases} 2T_{hanoi}(n-1) + k & \text{si } n > 1 \\ k' & \text{si } n = 1 \end{cases}$$

sent k i k' constants positives en alguna unitat de temps

$$T(n) = 2T(n-1) + k = 2(2T(n-2)+k) + k =$$
$$2^2T(n-2) + 3k = 2^2(2T(n-3)+k) + 3k =$$
$$2^3T(n-3) + 7k = \ldots = 2^i\,T(n-i) + (2^i-1)k = \ldots =$$

$n-i = 1 \rightarrow i = n-1$ $\quad 2^{n-1}k' + (2^{n-1}-1)k \in \Theta(2^n)$

Problema 10. El mètode següent troba el k-èssim element més menut d'un array de n enters.

n: 5 | k: 3 | v: → | 7 | 3 | 6 | 2 | 5 |

aux: → | 0 | 2 | 0 | 1 | 3 |

```
public static int k-minim(int[] v, int n, int k) {
    int[] aux = new int[n];
    int cont = 1, min, posmin;
    while (cont <= k) {
        min = Integer.MAX_VALUE;  posmin = n;
        for (int i = 0; i < n; i++) {
            if (aux[i] == 0 && v[i] < min) {
                min = v[i]; posmin = i;
            }
        }
        aux[posmin] = cont;
        cont++;
    }
    return min;
}
```

RECORREGUT

RECORREGUT

Talla: Nombre d'elements de v, donat per n, i l'índex k: (n, k)

Instàncies: No

Instrucció crítica: `aux[i] == 0 && v[i] < min` (de cost unitari)

$$T_{\text{k-minim}}(n, k) = \sum_{cont=1}^{k} \sum_{i=0}^{n-1} 1 = k * n \text{ i.c.} \quad \in \Theta(k * n)$$

Problema 9. El següent mètode obté, a partir de cert valor enter m, la seqüència dels dígits que el composen (array v) i torna el nombre de xifres que té el número (i).

```
public static int xifres(int[] v, int m) {
    int i = 0, q = m;
    while (q > 0) {
        i++;
        v[i] = q % 10;
        q = q / 10;
    }
    return i;
}
```

RECORREGUT

Talla: Valor de m

Instàncies: No

Instrucció crítica: q > 0 (de cost unitari)

En cada iteració **la talla** es redueix en un factor constant **c = 10** fins **la talla límit n_0**

$1 \le n_0 \le 9$

- $n_0 = m/c^{límit}$ ☞ $límit = \log_c(m/n_0) = \log_{10} m - \log_{10} n_0 \approx \log_{10} m$

$T_{xifres}(m) = (\log_{10} m + 1) + 1 \approx \log_{10} m$ i.c. $\in \Theta(\log_{10} m)$

nºiter = $\log_{10} m + 1$

```java
public static int xifres(int[] v, int m) {
    int i = 0, q = m;
    while (q > 0) {
        i++;
        v[i] = q % 10;
        q = q / 10;
    }
    return i;
}
```

Talla: Nombre de xifres de m, n

Instàncies: No

Instrucció crítica: `i++;` (de cost unitari)

$T_{xifres}(n) = n$ i.c. $\in \Theta(n)$

¿ $T_{xifres}(m) \in \Theta(\log_{10} m)$?

☞ El nombre de xifres d'un número m és $\boxed{\lfloor \log_{10} m \rfloor + 1 = n}$

☞ $T_{xifres}(n) \in \Theta(n) = T_{xifres}(m) \in \Theta(\log_{10} m)$

poli [formaT] PRG > EXÀMENS 1B - T2. Autoavaluació I

poli [formaT] PRG > EXÀMENS 1B - T2. Autoavaluació II

Solució des del 12/03/2020 a les 15:00