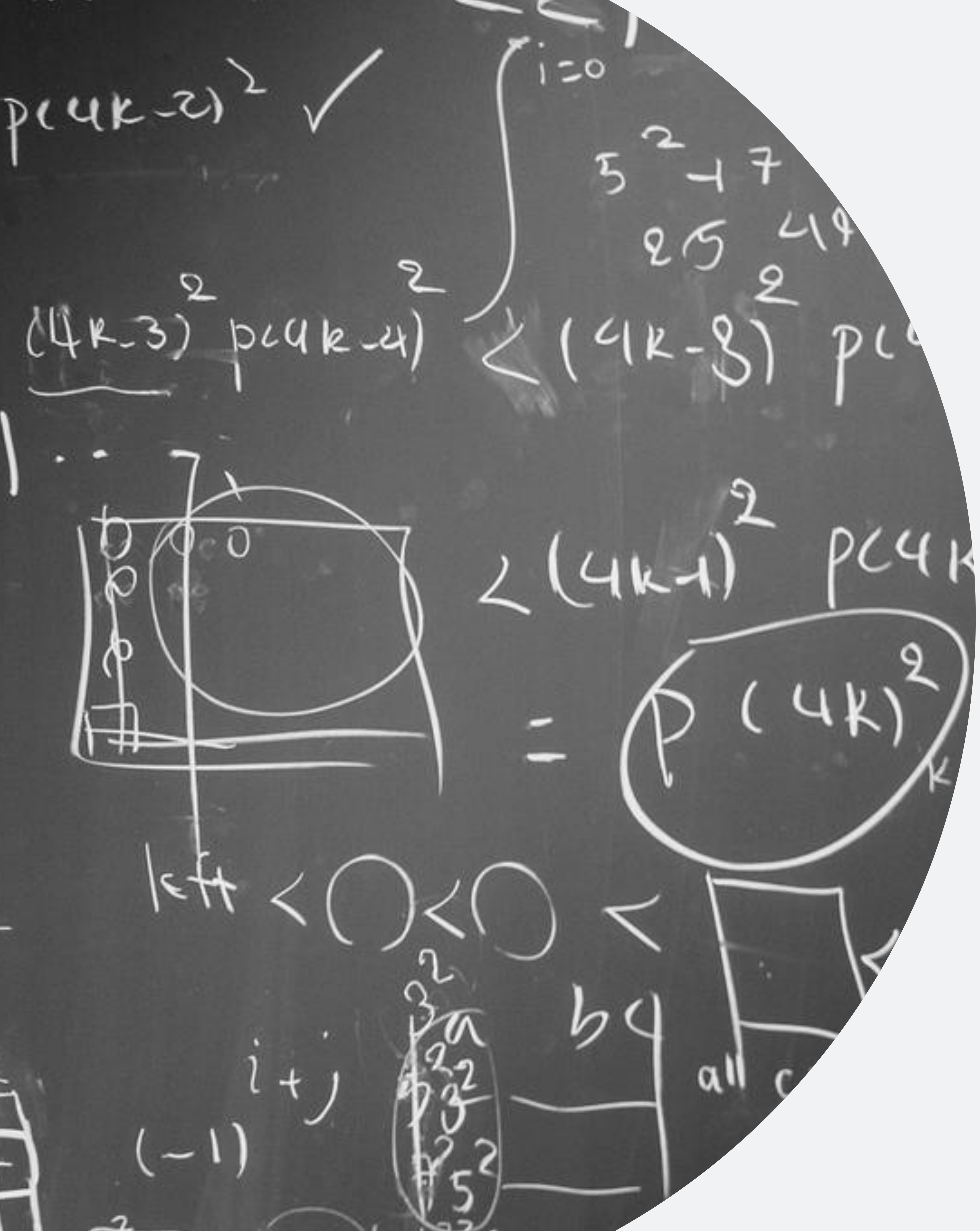
> ***Efficiency is doing things right;***
> ***effectiveness is doing the right things.***
> ***Algorithmic efficiency is both.***



TEMA 5.

COSTOS ALGORÍSMICS.

TEORIA.

FONAMENTS DE PROGRAMACIÓ II
CURS: 2024-25
UNIVERSITAT ROVIRA I VIRGILI

Eficiència dels algorismes

Per què estudiar l'eficiència dels algorismes?

- Sovint és necessari escollir entre diversos algorismes per resoldre un problema.
- Una possible estratègia seria implementar i executar tots els algorismes possibles i triar aquell que sigui més eficient. Però això té diversos problemes evidents:
  - Cost en temps i esforç d'implementació.
  - Els resultats depenen de la màquina i les dades utilitzades

- **Objectiu**: estudiar les propietats de l'algorisme a priori i implementar només aquell que considerem millor.

Eficiència dels algorismes

Concepte d'eficiència

- En termes generals, l'eficiència d'un algorisme es refereix a la quantitat de recursos computacionals usats per un algorisme durant la seva execució.

!

No confongueu "eficiència" amb "eficàcia"

- **Eficàcia**: Que té la virtut de produir l'efecte volgut.
- **Eficiència**: relació entre el treball efectuat per una màquina i els recursos que consumeix per produir aquest treball.

Eficiència dels algorismes

Concepte d'eficiència

- En termes generals, l'eficiència d'un algorisme es refereix a la quantitat de recursos computacionals usats per un algorisme durant la seva execució.

- Els recursos que considerem poden ser: **espaials** (quanta memòria consumeix l'algorisme per resoldre un problema donades unes dades)

Eficiència dels algorismes

Concepte d'eficiència

- En termes generals, l'eficiència d'un algorisme es refereix a la quantitat de recursos computacionals usats per un algorisme durant la seva execució.
- Els recursos que considerem poden ser: **espaials** (quanta memòria consumeix l'algorisme per resoldre un problema donades unes dades)

Exemple: desar una matriu d'adjacència d'una xarxa social

Opció "poc eficient" espaialment (matriu, desaprofito molts espais guardant zeros)

```
matriu =

      0     0     1     0     0
      0     0     0     1     0
      1     0     0     0     0
      0     1     0     0     1
      0     0     0     1     0
```

Eficiència dels algorismes

Concepte d'eficiència

- En termes generals, l'eficiència d'un algorisme es refereix a la quantitat de recursos computacionals usats per un algorisme durant la seva execució.
- Els recursos que considerem poden ser: **espaials** (quanta memòria consumeix l'algorisme per resoldre un problema donades unes dades)

Exemple: desar una matriu d'adjacència d'una xarxa social

Opció "poc eficient" espaialment (matriu, desaprofito molts espais guardant zeros)

```
matriu =

      0     0     1     0     0
      0     0     0     1     0
      1     0     0     0     0
      0     1     0     0     1
      0     0     0     1     0
```

Opció "més eficient" espaialment (llista, guardo només les connexions que existeixen)

```
llista =
   (3,1)        1
   (4,2)        1
   (1,3)        1
   (2,4)        1
   (5,4)        1
   (4,5)        1
```

Eficiència dels algorismes

Concepte d'eficiència

- En termes generals, l'eficiència d'un algorisme es refereix a la quantitat de recursos computacionals usats per un algorisme durant la seva execució.
- Els recursos que considerem poden ser: **espaials** (quanta memòria consumeix l'algorisme per resoldre un problema donades unes dades)
- També puc considerar els recursos **temporals**. És a dir, quant de temps triga l'algorisme a executar-se en funció de les dades que ha de tractar

Eficiència dels algorismes

Concepte d'eficiència

- En termes generals, l'eficiència d'un algorisme es refereix a la quantitat de recursos computacionals usats per un algorisme durant la seva execució.
- Els recursos que considerem poden ser: **espaials** (quanta memòria consumeix l'algorisme per resoldre un problema donades unes dades)
- També puc considerar els recursos **temporals**. És a dir, quant de temps triga l'algorisme a executar-se en funció de les dades que ha de tractar
- Normalment, hi ha un **trade-off** entre l'eficiència temporal i l'espaial: aquelles solucions que tenen un bon temps d'execució sovint requereixen l'ús de valors precalculats o tenir més dades en memòria.

Eficiència dels algorismes

Concepte d'eficiència

- En termes generals, l'eficiència d'un algorisme es refereix a la quantitat de recursos computacionals usats per un algorisme durant la seva execució.
- Els recursos que considerem poden ser: **espaials** (quanta memòria consumeix l'algorisme per resoldre un problema donades unes dades)
- També puc considerar els recursos **temporals**. És a dir, quant de temps triga l'algorisme a executar-se en funció de les dades que ha de tractar
- Normalment, hi ha un **trade-off** entre l'eficiència temporal i l'espaial: aquelles solucions que tenen un bon temps d'execució sovint requereixen l'ús de valors precalculats o tenir més dades en memòria.

Per estudiar l'eficiència d'algorismes ens centrarem en l'estudi de l'**eficiència temporal**, considerant que un algorisme que produeix un resultat correcte en un menor temps és **millor**.

Eficiència dels algorismes

Com estudiar l'eficiència d'un algorisme?

Cronometrar amb un timer

```c
#include <time.h>

     clock_t start, end;
     double cpu_time_used;

     start = clock();
     ...  /* Do the work. */
     end = clock();
     cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
```

La llibreria <time.h> ens permet cronometrar un programa

Eficiència dels algorismes

Com estudiar l'eficiència d'un algorisme?

Cronometrar amb un timer

- El temps d'execució varia entre diferents **algorismes**
- El temps d'execució varia en diferents **ordinadors**
- El temps d'execució **no es pot predir** a partir de proves que fem amb inputs petits (sabem que el temps serà diferent per mides d'input diferents, però no sabem quina és la relació entre la mida de l'input i el temps d'execució, o sigui, no podem fer una predicció).

```c
#include <time.h>

    clock_t start, end;
    double cpu_time_used;

    start = clock();
    … /* Do the work. */
    end = clock();
    cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
```

La llibreria <time.h> ens permet cronometrar un programa

Eficiència dels algorismes

Com estudiar l'eficiència d'un algorisme?

Cronometrar amb un timer

- El temps d'execució varia entre diferents **algorismes**
- El temps d'execució varia en diferents **ordinadors**
- El temps d'execució **no es pot predir** a partir de proves que fem amb inputs petits (sabem que el temps serà diferent per mides d'input diferents, però no sabem quina és la relació entre la mida de l'input i el temps d'execució, o sigui, no podem fer una predicció).

```
#include <time.h>

    clock_t start, end;
    double cpu_time_used;

    start = clock();
    ...  /* Do the work. */
    end = clock();
    cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
```

La llibreria <time.h> ens permet cronometrar un programa

Cronometrar avalua l'eficiència de l'algorisme per una implementació i una màquina concreta, no ens serveix per determinar l'eficiència general de l'algorisme de manera abstracta o independent del hardware i les dades d'entrada específiques

Eficiència dels algorismes

Com estudiar l'eficiència d'un algorisme?

- El temps d'execució d'un programa depèn de diversos factors:
  - Les dades d'entrada
  - La qualitat del codi generat pel compilador
  - La màquina on s'executa el programa
  - La complexitat del temps de l'algorisme base del programa.

Eficiència dels algorismes

Com estudiar l'eficiència d'un algorisme?

- El temps d'execució d'un programa depèn de diversos factors:
    - ~~Les dades d'entrada~~
    - ~~La qualitat del codi generat pel compilador~~
    - ~~La màquina on s'executa el programa~~
    - La complexitat del temps de l'algorisme base del programa.
- Quan dissenyem l'algorisme, no coneixem cap dels tres primers factors. Per tant, estudiar l'eficiència d'un algorisme es limita a avaluar la complexitat del temps de l'algorisme.

Eficiència dels algorismes

Com estudiar l'eficiència d'un algorisme?

- Quan estudiem la complexitat d'un algorisme, aquesta es defineix en funció de les dades d'entrada. Cal que quedi clar que l'estudi es fa en funció de **la mida de les dades d'entrada, no de les dades en si.**

Eficiència dels algorismes

Com estudiar l'eficiència d'un algorisme?

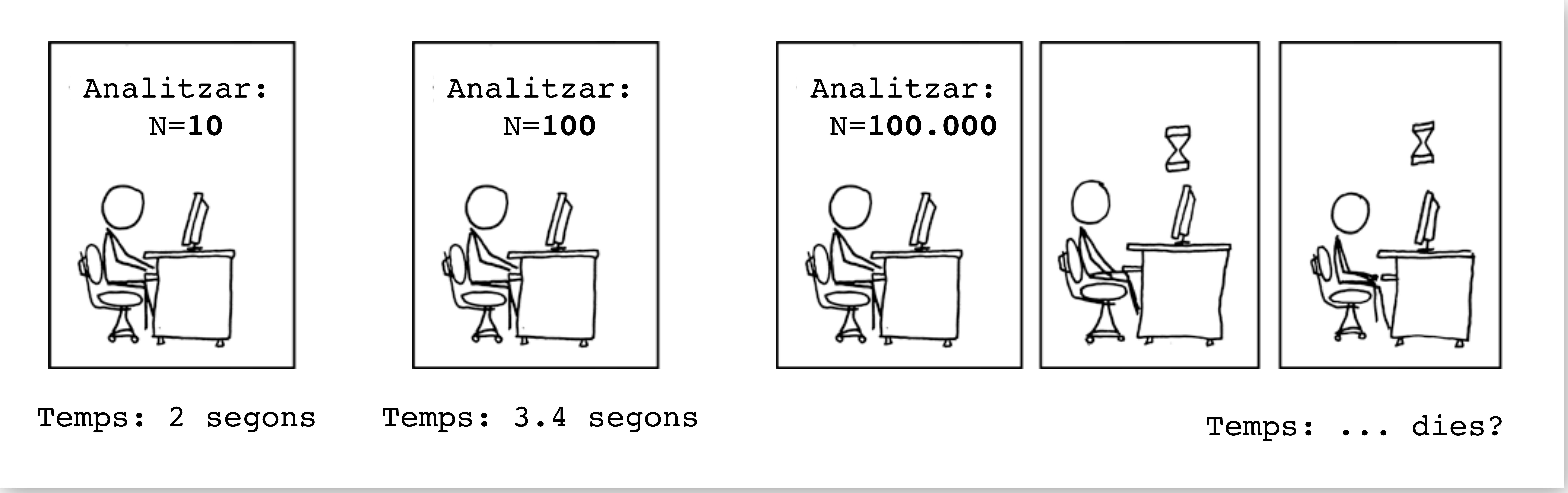
- Quan estudiem la complexitat d'un algorisme, aquesta es defineix en funció de les dades d'entrada. Cal que quedi clar que l'estudi es fa en funció de **la mida de les dades d'entrada, no de les dades en si.**
    - Per exemple, si volguéssim analitzar un algorisme d'ordenació que, donada una taula de nombres enters, ens la retorni ordenada, i volguéssim ordenar la taula: `{2, 1, 3, 1, 5, 8}`, parlaríem d'una entrada de mida **n=6**.

Eficiència dels algorismes

Volem avaluar com escala el temps d'execució d'un algorisme en funció de la mida de l'input

Com estudiar l'eficiència d'un algorisme?

- Quan estudiem la complexitat d'un algorisme, aquesta es defineix en funció de les dades d'entrada. Cal que quedi clar que l'estudi es fa en funció de **la mida de les dades d'entrada, no de les dades en si.**



Eficiència dels algorismes

Volem avaluar com escala el temps d'execució d'un algorisme en funció de la mida de l'input

Com estudiar l'eficiència d'un algorisme?

- Quan estudiem la complexitat d'un algorisme, aquesta es defineix en funció de les dades d'entrada. Cal que quedi clar que l'estudi es fa en funció de **la mida de les dades d'entrada, no de les dades en si.**
    - Per exemple, si volguéssim analitzar un algorisme d'ordenació que, donada una taula de nombres enters, ens la retorni ordenada, i volguéssim ordenar la taula: `{2, 1, 3, 1, 5, 8}`, parlaríem d'una entrada de mida **n=6**.

Denotarem per $T(n)$ la funció del temps d'execució d'un algorisme amb entrada de mida n.

$T(n)$ s'expressa sense unitats, i representa el nombre d'operacions elementals que realitza l'algorisme per obtenir la solució.

Es consideren operacions elementals les assignacions, comparacions, operacions aritmètiques...

Eficiència dels algorismes

Comparació d'algorismes

- La funció $T(n)$ ens permet comparar l'eficiència de diversos algorismes.

Eficiència dels algorismes

Comparació d'algorismes

- La funció $T(n)$ ens permet comparar l'eficiència de diversos algorismes.

Exemple: factorial (iteratiu)

- El factorial, calculat com: $n! = \prod_{i=1}^{n} i$
- Rep per paràmetre el valor de n, i l'algorisme multiplica n per $n-1$, i el resultat per $n-2$, etc.
- Per tant, per calcular el factorial de n, necessitem fer n multiplicacions.
- A grans trets, podem dir que la funció factorial té una funció de cost $T(n) = n$, i ho interpretem com que la funció factorial té un cost ***proporcional*** a n.

Eficiència dels algorismes

Comparació d'algorismes

- La funció $T(n)$ ens permet comparar l'eficiència de diversos algorismes.

- Si ara volem calcular el cost d'un algorisme A_1, de cost $T_1(n) = n$, amb un altre algorisme A_2 que té, per exemple, cost $T_2(n) = n^2 + 2n$, direm que el primer és més eficient.

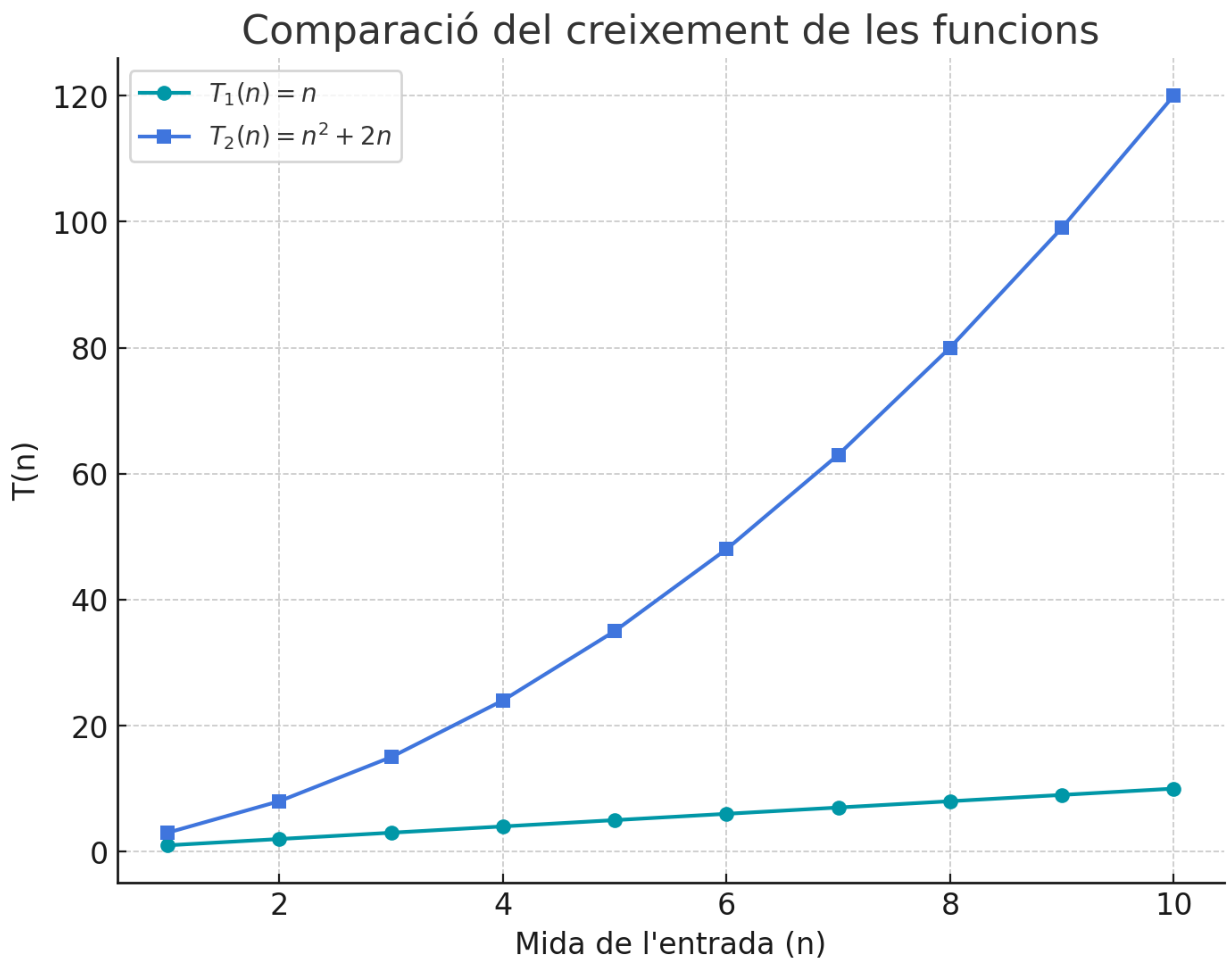
Eficiència dels algorismes

Comparació d'algorismes

- La funció $T(n)$ ens permet comparar l'eficiència de diversos algorismes.

- Si ara volem calcular el cost d'un algorisme A_1, de cost $T_1(n) = n$, amb un altre algorisme A_2 que té, per exemple, cost $T_2(n) = n^2 + 2n$, direm que el primer és més eficient.

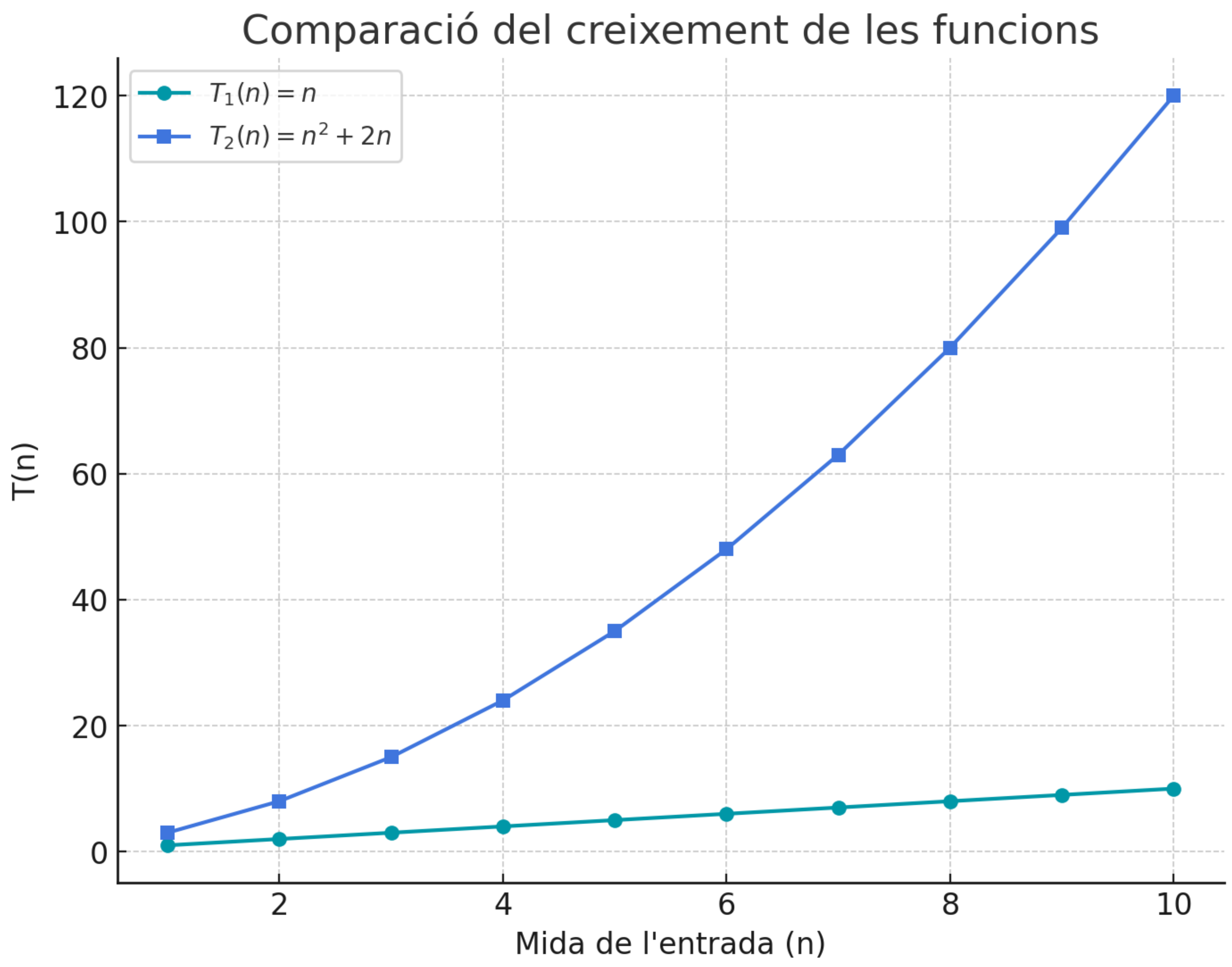
- Això és perquè a mesura que n (la mida de l'entrada) es faci més gran, el valor de T_1 creix més lentament que el valor de T_2.



Eficiència dels algorismes

Comparació d'algorismes

- La funció $T(n)$ ens permet comparar l'eficiència de diversos algorismes.
- Si ara volem calcular el cost d'un algorisme A_1, de cost $T_1(n) = n$, amb un altre algorisme A_2 que té, per exemple, cost $T_2(n) = n^2 + 2n$, direm que el primer és més eficient.
- Això és perquè a mesura que n (la mida de l'entrada) es faci més gran, el valor de T_1 creix més lentament que el valor de T_2.



El concepte d'eficiència d'un algorisme és relatiu.
No podem afirmar que un algorisme és eficient en termes absoluts, només podem dir que és més eficient o menys eficient que un altre.

Càlcul de la funció de cost $T(n)$

- Per comparar diversos algorismes, es pot calcular, de manera teòrica, la funció $T(n)$.
- Per fer-ho, assumim que coneixem el cost de cada operació. Aquest depèn de l'arquitectura de l'ordinador, el model de memòria, el conjunt d'instruccions, el compilador i possibles optimitzacions.

Càlcul de la funció de cost $T(n)$

- Per comparar diversos algorismes, es pot calcular, de manera teòrica, la funció $T(n)$.
- Per fer-ho, assumim que coneixem el cost de cada operació. Aquest depèn de l'arquitectura de l'ordinador, el model de memòria, el conjunt d'instruccions, el compilador i possibles optimitzacions.

- Per exemple, un possible cost podria ser:
  - T_C (temps de la comparació) = 1
  - T_A (temps de l'assignació) = 1
  - T_S (temps de la suma) = 1
  - T_{INC} (temps de l'increment/decrement) = 1
  - T_P (temps del producte) = 2
  - T_{DIV} (temps divisió entera) = 3

Càlcul de la funció de cost $T(n)$

- Per comparar diversos algorismes, es pot calcular, de manera teòrica, la funció $T(n)$.
- Per fer-ho, assumim que coneixem el cost de cada operació. Aquest depèn de l'arquitectura de l'ordinador, el model de memòria, el conjunt d'instruccions, el compilador i possibles optimitzacions.

- Per exemple, un possible cost podria ser:
  - T_C (temps de la comparació) = 1
  - T_A (temps de l'assignació) = 1
  - T_S (temps de la suma) = 1
  - T_{INC} (temps de l'increment/decrement) = 1
  - T_P (temps del producte) = 2
  - T_{DIV} (temps divisió entera) = 3

- Llavors podríem calcular la funció de cost $T(n)$ d'un algorisme com aquest:

```
t := 2;
a := t * 3;
b := (a * 3 + t) div 2;
```

Càlcul de la funció de cost $T(n)$

- Per comparar diversos algorismes, es pot calcular, de manera teòrica, la funció $T(n)$.
- Per fer-ho, assumim que coneixem el cost de cada operació. Aquest depèn de l'arquitectura de l'ordinador, el model de memòria, el conjunt d'instruccions, el compilador i possibles optimitzacions.

- Per exemple, un possible cost podria ser:
  - T_C (temps de la comparació) = 1
  - T_A (temps de l'assignació) = 1
  - T_S (temps de la suma) = 1
  - T_{INC} (temps de l'increment/decrement) = 1
  - T_P (temps del producte) = 2
  - T_{DIV} (temps divisió entera) = 3

- Llavors podríem calcular la funció de cost $T(n)$ d'un algorisme com aquest:

```
t := 2;
a := t * 3;
b := (a * 3 + t) div 2;
```

$T_A = 1$

Càlcul de la funció de cost $T(n)$

- Per comparar diversos algorismes, es pot calcular, de manera teòrica, la funció $T(n)$.
- Per fer-ho, assumim que coneixem el cost de cada operació. Aquest depèn de l'arquitectura de l'ordinador, el model de memòria, el conjunt d'instruccions, el compilador i possibles optimitzacions.

- Per exemple, un possible cost podria ser:
  - T_C (temps de la comparació) = 1
  - T_A (temps de l'assignació) = 1
  - T_S (temps de la suma) = 1
  - T_{INC} (temps de l'increment/decrement) = 1
  - T_P (temps del producte) = 2
  - T_{DIV} (temps divisió entera) = 3

- Llavors podríem calcular la funció de cost $T(n)$ d'un algorisme com aquest:

```
t := 2;
a := t * 3;
b := (a * 3 + t) div 2;
```

$T_A = 1$

$T_P + T_A = 2 + 1$

Càlcul de la funció de cost $T(n)$

- Per comparar diversos algorismes, es pot calcular, de manera teòrica, la funció $T(n)$.
- Per fer-ho, assumim que coneixem el cost de cada operació. Aquest depèn de l'arquitectura de l'ordinador, el model de memòria, el conjunt d'instruccions, el compilador i possibles optimitzacions.

- Per exemple, un possible cost podria ser:
  - T_C (temps de la comparació) = 1
  - T_A (temps de l'assignació) = 1
  - T_S (temps de la suma) = 1
  - T_{INC} (temps de l'increment/decrement) = 1
  - T_P (temps del producte) = 2
  - T_{DIV} (temps divisió entera) = 3

- Llavors podríem calcular la funció de cost $T(n)$ d'un algorisme com aquest:

```
t := 2;
a := t * 3;
b := (a * 3 + t) div 2;
```

$T_A = 1$

$T_P + T_A = 2 + 1$

$T_P + T_S + T_{DIV} + T_A = 2 + 1 + 3 + 1$

Càlcul de la funció de cost $T(n)$

- Per comparar diversos algorismes, es pot calcular, de manera teòrica, la funció $T(n)$.
- Per fer-ho, assumim que coneixem el cost de cada operació. Aquest depèn de l'arquitectura de l'ordinador, el model de memòria, el conjunt d'instruccions, el compilador i possibles optimitzacions.

- Per exemple, un possible cost podria ser:
  - T_C (temps de la comparació) = 1
  - T_A (temps de l'assignació) = 1
  - T_S (temps de la suma) = 1
  - T_{INC} (temps de l'increment/decrement) = 1
  - T_P (temps del producte) = 2
  - T_{DIV} (temps divisió entera) = 3

- Llavors podríem calcular la funció de cost $T(n)$ d'un algorisme com aquest:

```
t := 2;
a := t * 3;
b := (a * 3 + t) div 2;
```

$T_A = 1$

$T_P + T_A = 2 + 1$

$T_P + T_S + T_{DIV} + T_A = 2 + 1 + 3 + 1$

$$T(n) = 11$$

Càlcul de la funció de cost $T(n)$

Velocitat de creixement

- Hi ha un seguit de funcions que apareixen amb freqüència a l'anàlisi d'algorismes.
- Convé recordar quines impliquen un creixement més ràpid o més lent.

Velocitat de creixement	Nom	Valors per n=1,2,...,10
k	Constant	1, 1, 1, 1, 1, 1, 1, 1
$log(n)$	Logarítmica	0.0, 0.69, 1.1, 1.39, 1.61, 1.79, 1.95, 2.08, 2.2, 2.3
n	Lineal	1, 2, 3, 4, 5, 6, 7, 8, 9, 10
$n \cdot log(n)$	Quasi-lineal	0.0, 1.39, 3.3, 5.55, 8.05, 10.75, 13.62, 16.64, 19.78, 23.03
n^2	Quadràtica	1, 4, 9, 16, 25, 36, 49, 64, 81, 100
n^k	Polinòmica	1, 8, 27, 64, 125, 216, 343, 512, 729, 1000 (per k=3)
2^n	Exponencial	2, 4, 8, 16, 32, 64, 128, 256, 512, 1024

Càlcul de la funció de cost $T(n)$

Cas millor, pitjor, i promig

- A vegades, queda clar quina és la funció de cost d'un algorisme determinat. Per exemple:

```
funció suma_vector (v: taula[] d'enter,
                    mida: enter)
                    retorna enter és
var
   i: enter;
   resultat: enter;
fvar
   resultat := 0;
   per (i: = 0; i < mida; i:=i+1) fer
      resultat := resultat + v[i];
   fper
   retorna resultat;
ffunció
```

- Aquest algorisme fa dues assignacions inicialment (`resultat:=0` i `i:=0`). En total, 2 operacions bàsiques.
- A cada iteració fa una comparació (`i<mida`), dues sumes (`resultat`, i `i`), i un accés al vector (`v[i]`). En total, 4 operacions bàsiques.

- Si assumim que totes les operacions bàsiques tenen cost 1 (per simplificar el problema):
- Com que les quatre operacions anteriors les farà per cada element del vector, la funció de cost és: $T(n) = 2 + 4n$

Càlcul de la funció de cost $T(n)$

Cas millor, pitjor, i promig

- Altres vegades no queda tant clar quantes vegades farem cada operació:

```
funció cercar_element (t: taula[] d'enter,
N: enter, elem: enter) retorna booleà és
var
  i: enter;
  trobat: booleà;
fvar
inici
  i := 0;
  trobat := fals;
  mentre (i<N) i (no(trobat)) fer
    si (t[i] = elem)
      trobat := cert;
    fsi
    i:=i+1;
  fmentre
  retorna (trobat);
ffunció
```

Càlcul de la funció de cost $T(n)$

Cas millor, pitjor, i promig

- Altres vegades no queda tant clar quantes vegades farem cada operació:

```
funció cercar_element (t: taula[] d'enter,
N: enter, elem: enter) retorna booleà és
var
  i: enter;
  trobat: booleà;
fvar
inici
  i := 0;
  trobat := fals;
  mentre (i<N) i (no(trobat)) fer
    si (t[i] = elem)
      trobat := cert;
    fsi
    i:=i+1;
  fmentre
  retorna (trobat);
ffunció
```

Millor cas

Pitjor cas

Cas average

Càlcul de la funció de cost $T(n)$

Cas millor, pitjor, i promig

- Altres vegades no queda tant clar quantes vegades farem cada operació:

```
funció cercar_element (t: taula[] d'enter,
N: enter, elem: enter) retorna booleà és
var
  i: enter;
  trobat: booleà;
fvar
inici
  i := 0;
  trobat := fals;
  mentre (i<N) i (no(trobat)) fer
    si (t[i] = elem)
      trobat := cert;
    fsi
    i:=i+1;
  fmentre
  retorna (trobat);
ffunció
```

Millor cas

Si l'element **`elem`** està a la primera posició de la taula

Només hem d'explorar la primera posició de la taula

Pitjor cas

Cas average

Càlcul de la funció de cost $T(n)$

Cas millor, pitjor, i promig

- Altres vegades no queda tant clar quantes vegades farem cada operació:

```
funció cercar_element (t: taula[] d'enter,
N: enter, elem: enter) retorna booleà és
var
  i: enter;
  trobat: booleà;
fvar
inici
  i := 0;
  trobat := fals;
  mentre (i<N) i (no(trobat)) fer
    si (t[i] = elem)
      trobat := cert;
    fsi
    i:=i+1;
  fmentre
  retorna (trobat);
ffunció
```

Millor cas

Si l'element **`elem`** està a la primera posició de la taula

Només hem d'explorar la primera posició de la taula

Pitjor cas

Si l'element **`elem`** no és a la taula

Haurem hagut de recórrer la taula sencera (N)

Cas average

Càlcul de la funció de cost $T(n)$

Cas millor, pitjor, i promig

- Altres vegades no queda tant clar quantes vegades farem cada operació:

```
funció cercar_element (t: taula[] d'enter,
N: enter, elem: enter) retorna booleà és
var
  i: enter;
  trobat: booleà;
fvar
inici
  i := 0;
  trobat := fals;
  mentre (i<N) i (no(trobat)) fer
    si (t[i] = elem)
      trobat := cert;
    fsi
    i:=i+1;
  fmentre
  retorna (trobat);
ffunció
```

Millor cas

Si l'element **`elem`** està a la primera posició de la taula

Només hem d'explorar la primera posició de la taula

Pitjor cas

Si l'element **`elem`** no és a la taula

Haurem hagut de recórrer la taula sencera (N)

Cas average

Si l'element **`elem`** està al mig

Haurem hagut de recórrer mitja taula (N/2)

Càlcul de la funció de cost $T(n)$

Cas millor, pitjor, i promig

- Altres vegades no queda tant clar quantes vegades farem cada operació:

Millor cas

Si l'element `elem` està a la primera posició de la taula

Només hem d'explorar la primera posició de la taula

Estudiar el millor cas sovint és poc útil, ja que representa una situació ideal que rarament es dona a la pràctica.

Pitjor cas

Si l'element `elem` no és a la taula

Haurem hagut de recórrer la taula sencera (N)

Cas average

Si l'element `elem` està al mig

Haurem hagut de recórrer mitja taula (N/2)

Càlcul de la funció de cost $T(n)$

Cas millor, pitjor, i promig

- Altres vegades no queda tant clar quantes vegades farem cada operació:

Millor cas

Si l'element `elem` està a la primera posició de la taula

Només hem d'explorar la primera posició de la taula

Estudiar el millor cas sovint és poc útil, ja que representa una situació ideal que rarament es dona a la pràctica.

Pitjor cas

Si l'element `elem` no és a la taula

Haurem hagut de recórrer la taula sencera (N)

Cas average

Si l'element `elem` està al mig

Haurem hagut de recórrer mitja taula (N/2)

Estudiar el cas mig requereix conèixer la distribució de probabilitats de totes les possibles entrades, cosa que pot ser complex o fins i tot impossible.

Càlcul de la funció de cost $T(n)$

Cas millor, pitjor, i promig

- Altres vegades no queda tant clar quantes vegades farem cada operació:

Millor cas

Si l'element **`elem`** està a la primera posició de la taula

Només hem d'explorar la primera posició de la taula

Estudiar el millor cas sovint és poc útil, ja que representa una situació ideal que rarament es dona a la pràctica.

Pitjor cas

Si l'element **`elem`** no és a la taula

Haurem hagut de recórrer la taula sencera (N)

Estudiar el pitjor cas és més pràctic, ja que ens proporciona una cota superior del creixement del cost de l'algorisme, garantint que no serà més lent que aquest límit en cap situació.

Cas average

Si l'element **`elem`** està al mig

Haurem hagut de recórrer mitja taula (N/2)

Estudiar el cas mig requereix conèixer la distribució de probabilitats de totes les possibles entrades, cosa que pot ser complex o fins i tot impossible.

Càlcul de l'ordre de creixement (Big-O)

La Big-O notation

- **La funció** $T(n)$ Representa la funció exacta que descriu el **nombre d'operacions elementals** que executa un algorisme en funció de la mida de l'entrada (n).
  - Aquesta notació conté molts detalls, que sovint no són importants: dues funcions poden tenir una forma diferent però tenir velocitats de creixement molt similars.
    - Per exemple, $T_1(n) = 5n + 20$ i $T_2(n) = 2n + 100$.

Càlcul de l'ordre de creixement (Big-O)

La Big-O notation

- **La funció** $T(n)$ Representa la funció exacta que descriu el **nombre d'operacions elementals** que executa un algorisme en funció de la mida de l'entrada (n).
  - Aquesta notació conté molts detalls, que sovint no són importants: dues funcions poden tenir una forma diferent però tenir velocitats de creixement molt similars.
    - Per exemple, $T_1(n) = 5n + 20$ i $T_2(n) = 2n + 100$.

- **La notació Big-O** serveix per indicar la velocitat de creixement de la funció de temps $T(n)$ d'un algorisme
  - Simplifica $T(n)$ concentrant-se només en el comportament asimptòtic (quan la mida de l'entrada n es fa molt gran). Ignora les constants multiplicatives i els termes de menor creixement.

Càlcul de l'ordre de creixement (Big-O)

La Big-O notation

- **La funció $T(n)$** Representa la funció exacta que descriu el **nombre d'operacions elementals** que executa un algorisme en funció de la mida de l'entrada (n).
  - Aquesta notació conté molts detalls, que sovint no són importants: dues funcions poden tenir una forma diferent però tenir velocitats de creixement molt similars.
    - Per exemple, $T_1(n) = 5n + 20$ i $T_2(n) = 2n + 100$.

- **La notació Big-O** serveix per indicar la velocitat de creixement de la funció de temps $T(n)$ d'un algorisme
  - Simplifica $T(n)$ concentrant-se només en el comportament asimptòtic (quan la mida de l'entrada n es fa molt gran). Ignora les constants multiplicatives i els termes de menor creixement.

> Es diu que $T(n)$ és $\mathcal{O}(f(n))$ quan la velocitat de creixement de $T(n)$ està acotada superiorment per $f(n)$. Així, $\mathcal{O}(f(n))$ representa el conjunt de totes les funcions $g(n)$ que creixen, com a molt, tant ràpidament com $f(n)$.

Càlcul de l'ordre de creixement (Big-O)

La Big-O notation

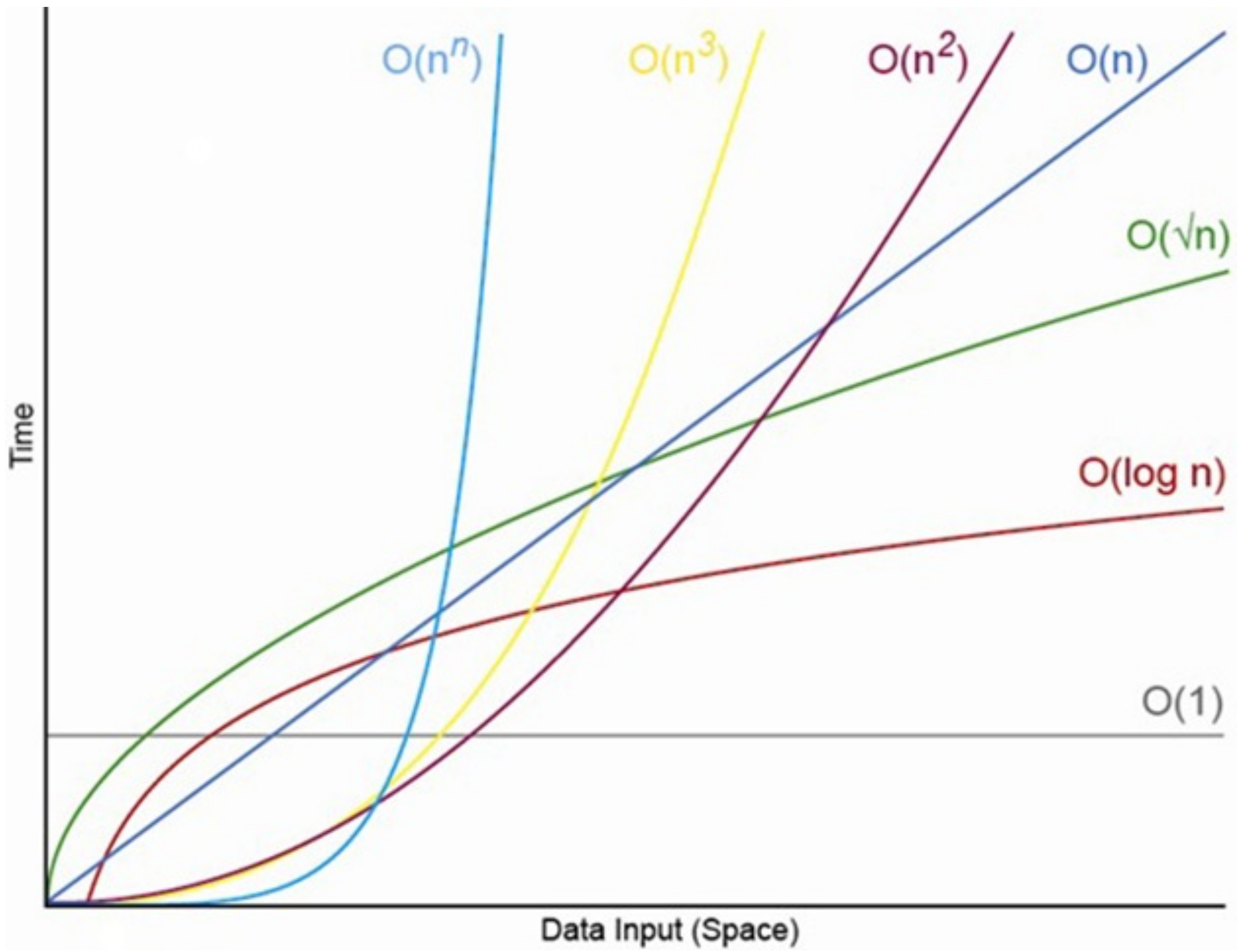
Asymptotic Notation in Seven Words

$$\textit{suppress}\ \underbrace{\textit{constant factors}}_{\textit{too system-dependent}}\ \textit{and}\ \underbrace{\textit{lower-order terms}}_{\textit{irrelevant for large inputs}}$$

SOURCE: ALGORITHMS ILLUMINATED

Càlcul de l'ordre de creixement (Big-O)

La Big-O notation



Ordres de complexitat

Constant	$\mathcal{O}(1)$
Logarítmic	$\mathcal{O}(\log_2 n)$
Lineal	$\mathcal{O}(n)$
Quasilineal	$\mathcal{O}(n\log_2 n)$
Quadràtic	$\mathcal{O}(n^2)$
Cúbic	$\mathcal{O}(n^3)$
Polinòmic	$\mathcal{O}(n^k)$, amb k conegut
Exponencial	$\mathcal{O}(k^n)$, amb k conegut
Factorial	$\mathcal{O}(n!)$
No afitat	$\mathcal{O}(\infty)$

- Comparant els ordres de creixement de diferents algorismes podem saber quins es comportaran millor quan l'input sigui molt gran.

Classe de complexitat		n = 10	n = 100	n = 1000	n = 1,000,000
Constant	$O(1)$	1	1	1	1
Logarítmic	$O(\log n)$	1	2	3	6
Lineal	$O(n)$	10	100	1000	1,000,000
Quasi-lineal	$O(n \log n)$	10	200	3000	6,000,000
Quadràtic	$O(n^2)$	100	10,000	1,000,000	1,000,000,000,000
Exponencial	$O(2^n)$	1024	126765060 022822940 149670320 5376	10715086071862673209484250490600 01810561404811705533607443750388370 35105112493612249319837881569585812 7594672917553146825187145285692314 0435984577574698574803934567774 824230985421074605062371141877954 1821530464749835819412673987675591 655439460770629145711964776865421676 604298316526243868372056680693 76	☠️

Classe de complexitat	Codi típic	Descripció	Exemple
Constant $O(1)$	`a := b + c;`	Assignació simple	Sumar dos nombres

Logarítmic
$O(\log n)$
	`mentre (n > 1) fer`
	`   ...`
	`   n := n/2;`
	`   ...`
	`fmentre`
		Dividir per la meitat	Cerca binària

Lineal
$O(n)$
	`per (i:=0; i<n; i:=i+1) fer`
	`   ...`
	`fper`
		Bucle	Trobar el màxim

Quasi-lineal $O(n \log n)$	`[Ho veurem al capítol d'ordenació]`	"Divide and conquer"	Mergesort, Quicksort...

Quadràtic
$O(n^2)$
	`per (i:=0; i<n; i:=i+1) fer`
	`   per (j:=0; j<n; j:=j+1) fer`
	`      ...`
	`   fper`
	`fper`
		Doble bucle	Recorregut d'una taula 2D

Exponencial $O(2^n)$	`[Esquemes de cerca combinatòria]`	Cerca exhaustiva	Trobar tots els grups que es poden formar amb N elements

Càlcul de l'ordre de creixement (Big-O)

Com calcular l'ordre de creixement

- Calcular la funció del temps d'execució i expressar-la en notació asimptòtica (Big-O). Per fer-ho hem d'analitzar cadascuna de les instruccions, aplicar unes regles i prendre el cost màxim (terme dominant)

Càlcul de l'ordre de creixement (Big-O)

Com calcular l'ordre de creixement

- Calcular la funció del temps d'execució i expressar-la en notació asimptòtica (Big-O). Per fer-ho hem d'analitzar cadascuna de les instruccions, aplicar unes regles i prendre el cost màxim (terme dominant)

Regla de la suma:

- Es fa servir en instruccions que s'executen seqüencialment

$$\begin{aligned} f_1 &\in \mathcal{O}(g_1), \\ f_2 &\in \mathcal{O}(g_2) \end{aligned} \quad \Rightarrow \quad f_1 + f_2 \in \mathcal{O}(\max(g_1, g_2)) \, .$$

Càlcul de l'ordre de creixement (Big-O)

Com calcular l'ordre de creixement

- Calcular la funció del temps d'execució i expressar-la en notació asimptòtica (Big-O). Per fer-ho hem d'analitzar cadascuna de les instruccions, aplicar unes regles i prendre el cost màxim (terme dominant)

Regla de la suma:

- Es fa servir en instruccions que s'executen seqüencialment

$$\begin{array}{l} f_1 \in \mathcal{O}(g_1), \\ f_2 \in \mathcal{O}(g_2) \end{array} \quad \Rightarrow \quad f_1 + f_2 \in \mathcal{O}(\max(g_1, g_2))\,.$$

```
...
per (i:=0; i<n; i++)
  escriure("a");
fper
per (i:=0; i<n*n; i++)
  escriure("b");
fper
...
```

Càlcul de l'ordre de creixement (Big-O)

Com calcular l'ordre de creixement

- Calcular la funció del temps d'execució i expressar-la en notació asimptòtica (Big-O). Per fer-ho hem d'analitzar cadascuna de les instruccions, aplicar unes regles i prendre el cost màxim (terme dominant)

Regla de la suma:

- Es fa servir en instruccions que s'executen seqüencialment

$$\begin{matrix} f_1 \in \mathcal{O}(g_1), \\ f_2 \in \mathcal{O}(g_2) \end{matrix} \quad \Rightarrow \quad f_1 + f_2 \in \mathcal{O}(\max(g_1, g_2)) \,.$$

```
...
per (i:=0; i<n; i++)
  escriure("a");          → O(n)
fper
per (i:=0; i<n*n; i++)   → O(n²)
  escriure("b");
fper
...
```

Càlcul de l'ordre de creixement (Big-O)

Com calcular l'ordre de creixement

- Calcular la funció del temps d'execució i expressar-la en notació asimptòtica (Big-O). Per fer-ho hem d'analitzar cadascuna de les instruccions, aplicar unes regles i prendre el cost màxim (terme dominant)

Regla de la suma:

- Es fa servir en instruccions que s'executen seqüencialment

$$\begin{array}{l} f_1 \in \mathcal{O}(g_1), \\ f_2 \in \mathcal{O}(g_2) \end{array} \quad \Rightarrow \quad f_1 + f_2 \in \mathcal{O}(\max(g_1, g_2))\,.$$

```
...
per (i:=0; i<n; i++)
  escriure("a");
fper
per (i:=0; i<n*n; i++)
  escriure("b");
fper
...
```

$\text{escriure("a")} \rightarrow O(n)$

$\text{per (i:=0; i<n*n; i++)} \rightarrow O(n^2)$

$O(n) ,\ O(n^2) \rightarrow O(n + n^2) = O(n^2)$

Càlcul de l'ordre de creixement (Big-O)

Com calcular l'ordre de creixement

- Calcular la funció del temps d'execució i expressar-la en notació asimptòtica (Big-O). Per fer-ho hem d'analitzar cadascuna de les instruccions, aplicar unes regles i prendre el cost màxim (terme dominant)

Regla de la suma:

- Es fa servir en instruccions que s'executen seqüencialment

$$\begin{aligned} f_1 &\in \mathcal{O}(g_1), \\ f_2 &\in \mathcal{O}(g_2) \end{aligned} \quad \Rightarrow \quad f_1 + f_2 \in \mathcal{O}(\max(g_1, g_2)) \,.$$

```
...
per (i:=0; i<n; i++)
  escriure("a");
fper
per (i:=0; i<n*n; i++)
  escriure("b");
fper
...
```

O(n)

O(n^2)

O(n + n^2) =
= O(n^2)

Regla del producte

- Es fa servir en instruccions que estan anidades

$$\begin{aligned} f_1 &\in \mathcal{O}(g_1), \\ f_2 &\in \mathcal{O}(g_2) \end{aligned} \quad \Rightarrow \quad f_1 \cdot f_2 \in \mathcal{O}(g_1 \cdot g_2) \,.$$

Càlcul de l'ordre de creixement (Big-O)

Com calcular l'ordre de creixement

- Calcular la funció del temps d'execució i expressar-la en notació asimptòtica (Big-O). Per fer-ho hem d'analitzar cadascuna de les instruccions, aplicar unes regles i prendre el cost màxim (terme dominant)

Regla de la suma:

- Es fa servir en instruccions que s'executen seqüencialment

$$\begin{array}{l} f_1 \in \mathcal{O}(g_1), \\ f_2 \in \mathcal{O}(g_2) \end{array} \quad \Rightarrow \quad f_1 + f_2 \in \mathcal{O}(\max(g_1, g_2)) \,.$$

```
...
per (i:=0; i<n; i++)
  escriure("a");
fper
per (i:=0; i<n*n; i++)
  escriure("b");
fper
...
```

$O(n)$, $O(n^2)$ → $O(n + n^2) =$
$= O(n^2)$

Regla del producte

- Es fa servir en instruccions que estan anidades

$$\begin{array}{l} f_1 \in \mathcal{O}(g_1), \\ f_2 \in \mathcal{O}(g_2) \end{array} \quad \Rightarrow \quad f_1 \cdot f_2 \in \mathcal{O}(g_1 \cdot g_2) \,.$$

```
...
per (i:=0; i<n; i++)
  per (j:=0; i<n; i++)
    escriure("Hola");
  fper
fper
...
```

Càlcul de l'ordre de creixement (Big-O)

Com calcular l'ordre de creixement

- Calcular la funció del temps d'execució i expressar-la en notació asimptòtica (Big-O). Per fer-ho hem d'analitzar cadascuna de les instruccions, aplicar unes regles i prendre el cost màxim (terme dominant)

Regla de la suma:

- Es fa servir en instruccions que s'executen seqüencialment

$$\begin{array}{l} f_1 \in \mathcal{O}(g_1), \\ f_2 \in \mathcal{O}(g_2) \end{array} \quad \Rightarrow \quad f_1 + f_2 \in \mathcal{O}(\max(g_1, g_2)) \,.$$

```
...
per (i:=0; i<n; i++)
  escriure("a");
fper
per (i:=0; i<n*n; i++)
  escriure("b");
fper
...
```

O(n)

O(n²)

O(n + n²) =
= O(n²)

Regla del producte

- Es fa servir en instruccions que estan anidades

$$\begin{array}{l} f_1 \in \mathcal{O}(g_1), \\ f_2 \in \mathcal{O}(g_2) \end{array} \quad \Rightarrow \quad f_1 \cdot f_2 \in \mathcal{O}(g_1 \cdot g_2) \,.$$

```
...
per (i:=0; i<n; i++)
  per (j:=0; i<n; i++)
    escriure("Hola");
  fper
fper
...
```

O(n)

O(n)

Càlcul de l'ordre de creixement (Big-O)

Com calcular l'ordre de creixement

- Calcular la funció del temps d'execució i expressar-la en notació asimptòtica (Big-O). Per fer-ho hem d'analitzar cadascuna de les instruccions, aplicar unes regles i prendre el cost màxim (terme dominant)

Regla de la suma:

- Es fa servir en instruccions que s'executen seqüencialment

$$\begin{array}{l} f_1 \in \mathcal{O}(g_1), \\ f_2 \in \mathcal{O}(g_2) \end{array} \quad \Rightarrow \quad f_1 + f_2 \in \mathcal{O}(\max(g_1, g_2))\,.$$

```
...
per (i:=0; i<n; i++)
  escriure("a");
fper
per (i:=0; i<n*n; i++)
  escriure("b");
fper
...
```

`per (i:=0; i<n; i++) escriure("a"); fper` → O(n)

`per (i:=0; i<n*n; i++) escriure("b"); fper` → O(n²)

→ $O(n + n^2) = O(n^2)$

Regla del producte

- Es fa servir en instruccions que estan anidades

$$\begin{array}{l} f_1 \in \mathcal{O}(g_1), \\ f_2 \in \mathcal{O}(g_2) \end{array} \quad \Rightarrow \quad f_1 \cdot f_2 \in \mathcal{O}(g_1 \cdot g_2)\,.$$

```
...
per (i:=0; i<n; i++)
  per (j:=0; i<n; i++)
    escriure("Hola");
  fper
fper
...
```

`per (i:=0; i<n; i++)` → O(n)

`per (j:=0; i<n; i++)` → O(n)

→ $O(n*n) = O(n^2)$

Càlcul de l'ordre de creixement (Big-O)

Costos de les construccions algorísmiques

Operacions elementals

- Les operacions elementals que no depenen de la mida n de l'entrada tenen cost constant: $\mathcal{O}(1)$
- Quines operacions es consideren elementals i quines no depèn de la plataforma en la que es desenvolupi l'algorisme.
- En general, considerarem operacions elementals les operacions bàsiques (primitives) d'un llenguatge imperatiu com C.
- Serien: assignacions de tipus de dades primitius (enters, caràcter…), operacions aritmètiques, lectura i escriptura a fitxer o per pantalla, accés a la posició d'un vector…

Càlcul de l'ordre de creixement (Big-O)

Costos de les construccions algorísmiques

Operacions elementals

- Les operacions elementals que no depenen de la mida n de l'entrada tenen cost constant: $\mathcal{O}(1)$
- Quines operacions es consideren elementals i quines no depèn de la plataforma en la que es desenvolupi l'algorisme.
- En general, considerarem operacions elementals les operacions bàsiques (primitives) d'un llenguatge imperatiu com C.
- Serien: assignacions de tipus de dades primitius (enters, caràcter...), operacions aritmètiques, lectura i escriptura a fitxer o per pantalla, accés a la posició d'un vector...

```
x := 10;
x := x + 1;
b := (x > 0 i y < 1);
x := v[i];
escriure("Hola!");
llegir_f(numero);
```

Totes aquestes instruccions tenen cost constant

Càlcul de l'ordre de creixement (Big-O)

Costos de les construccions algorísmiques

Operacions elementals

- Les operacions elementals que no depenen de la mida n de l'entrada tenen cost constant: $\mathcal{O}(1)$
- Quines operacions es consideren elementals i quines no depèn de la plataforma en la que es desenvolupi l'algorisme.
- En general, considerarem operacions elementals les operacions bàsiques (primitives) d'un llenguatge imperatiu com C.
- Serien: assignacions de tipus de dades primitius (enters, caràcter…), operacions aritmètiques, lectura i escriptura a fitxer o per pantalla, accés a la posició d'un vector…

```
x := 10;
x := x + 1;
b := (x > 0 i y < 1);
x := v[i];
escriure("Hola!");
llegir_f(numero);
```

Totes aquestes instruccions tenen cost constant

- Compte! Si una instrucció crida a una funció, el cost de la instrucció és el cost de la funció, que pot ser que no sigui constant.

```
trobat := cerca_element(v, elem);
```

Aquesta instrucció té cost lineal amb la mida del vector v, perquè en el pitjor cas ha de recórrer tots els elements del vector v.

Càlcul de l'ordre de creixement (Big-O)

Costos de les construccions algorísmiques

Seqüència d'un bloc d'instruccions

- Quan tenim un bloc d'instruccions que s'executen seqüencialment, apliquem la regla de la suma.
- Per tant, el cost d'un bloc de sentències serà el cost de la sentència més costosa.

Càlcul de l'ordre de creixement (Big-O)

Costos de les construccions algorísmiques

Seqüència d'un bloc d'instruccions

- Quan tenim un bloc d'instruccions que s'executen seqüencialment, apliquem la regla de la suma.
- Per tant, el cost d'un bloc de sentències serà el cost de la sentència més costosa.

```
escriure("Introdueix nombre");   → O(1)
llegir(num);                     → O(1)
num = num * 100;                 → O(1)
escriure("El nombre es", num);   → O(1)
```

$\max \{O(1), O(1), O(1), O(1)\} \rightarrow O(1)$

Càlcul de l'ordre de creixement (Big-O)

Costos de les construccions algorísmiques

Seqüència d'un bloc d'instruccions

- Quan tenim un bloc d'instruccions que s'executen seqüencialment, apliquem la regla de la suma.
- Per tant, el cost d'un bloc de sentències serà el cost de la sentència més costosa.

`escriure("Introdueix nombre");`	$O(1)$	max	$O(1)$
`llegir(num);`	$O(1)$		
`num = num * 100;`	$O(1)$		
`escriure("El nombre es", num);`	$O(1)$		

`escriure("Introdueix nombre");`	$O(1)$	max	$O(n)$
`llegir(num);`	$O(1)$		
`cercar_element(v, num);`	$O(n)$		

Càlcul de l'ordre de creixement (Big-O)

Costos de les construccions algorísmiques

Estructures condicionals

- Els condicionals estan formats per una condició, una branca que s'executa en cas que es compleixi la condició, i una branca que s'executa en cas que no es compleixi.
- El cost es calcula com la complexitat de la branca amb major complexitat + la complexitat d'avaluar la condició.
  - *On el "+" representa la regla de la suma.*

```
si condició llavors
   sentències_branca_si;
sino
   sentències_branca_sino;
fsi
```

Càlcul de l'ordre de creixement (Big-O)

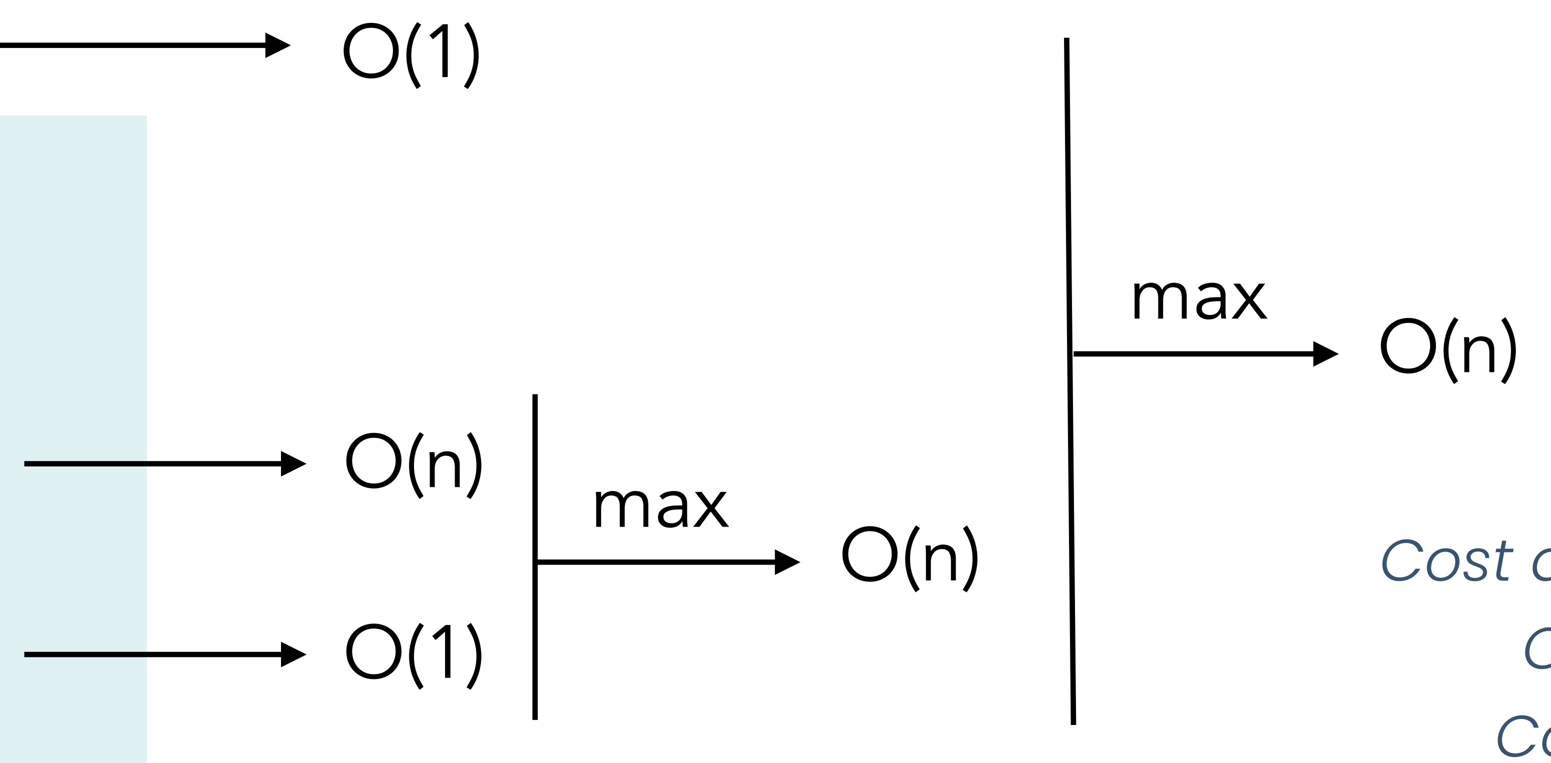
Costos de les construccions algorísmiques

Estructures condicionals

- Els condicionals estan formats per una condició, una branca que s'executa en cas que es compleixi la condició, i una branca que s'executa en cas que no es compleixi.
- El cost es calcula com la complexitat de la branca amb major complexitat + la complexitat d'avaluar la condició.
  - *On el "+" representa la regla de la suma.*

```
si condició llavors
   sentències_branca_si;
sino
   sentències_branca_sino;
fsi
```

```
si (x > 0) llavors
  // Cercar element
  trobat := cercar_element(v, x);
sino
  escriure("Error, nombre negatiu");
fsi
```



Cost d'avaluar la condició: O(1)
Cost del cas "si": O(n)
Cost del cas "sino": O(1)

Màxim de les dues branques: O(n)

Cost condició + cost branca màx = O(1) + O(n) = O(n)

Càlcul de l'ordre de creixement (Big-O)

Costos de les construccions algorísmiques

Estructures iteratives

- Si coneixem el nombre d’iteracions:
- A cada iteració s'avalua una condició, s'executa el cos del bucle, i s'actualitza la variable de control. D'aquests tres blocs ens quedem amb el de cost màxim.

```
per (i:=0; i<n; i:=i+1) fer
    sentències;
fper
```

Càlcul de l'ordre de creixement (Big-O)

Costos de les construccions algorísmiques

Estructures iteratives

- Si coneixem el nombre d'iteracions:
- A cada iteració s'avalua una condició, s'executa el cos del bucle, i s'actualitza la variable de control. D'aquests tres blocs ens quedem amb el de cost màxim.

```
per (i:=0; i<n; i:=i+1) fer
    sentències;
fper
```

- El cost serà el nombre d'iteracions multiplicat pel cost del bloc més costós.

```
Cost total = num_iteracions * (cost(condició) + cost(cos) + cost(actualització))
```

Càlcul de l'ordre de creixement (Big-O)

Costos de les construccions algorísmiques

Estructures iteratives

- Si coneixem el nombre d’iteracions:
- A cada iteració s'avalua una condició, s'executa el cos del bucle, i s'actualitza la variable de control. D'aquests tres blocs ens quedem amb el de cost màxim.

```
per (i:=0; i<n; i:=i+1) fer
    sentències;
fper
```

- El cost serà el nombre d'iteracions multiplicat pel cost del bloc més costós.

```
Cost total = num_iteracions * (cost(condició) + cost(cos) + cost(actualització))
```

```
per (i:=0; i<n; i:=i+2) fer
  escriure("i=", i);
fper
```

Càlcul de l'ordre de creixement (Big-O)

Costos de les construccions algorísmiques

Estructures iteratives

- Si coneixem el nombre d'iteracions:
- A cada iteració s'avalua una condició, s'executa el cos del bucle, i s'actualitza la variable de control. D'aquests tres blocs ens quedem amb el de cost màxim.

```
per (i:=0; i<n; i:=i+1) fer
    sentències;
fper
```

- El cost serà el nombre d'iteracions multiplicat pel cost del bloc més costós.

```
Cost total = num_iteracions * (cost(condició) + cost(cos) + cost(actualització))
```

```
per (i:=0; i<n; i:=i+2) fer
  escriure("i=", i);
fper
```

Cost de la condició: O(1)
Cost de l'actualització: O(1)
Cost del cos: O(1)

Nombre d'iteracions: n/2
Total = O(n/2) = O(n)

Càlcul de l'ordre de creixement (Big-O)

Costos de les construccions algorísmiques

Estructures iteratives

- Si coneixem el nombre d'iteracions:
- A cada iteració s'avalua una condició, s'executa el cos del bucle, i s'actualitza la variable de control. D'aquests tres blocs ens quedem amb el de cost màxim.

```
per (i:=0; i<n; i:=i+1) fer
    sentències;
fper
```

- El cost serà el nombre d'iteracions multiplicat pel cost del bloc més costós.

```
Cost total = num_iteracions * (cost(condició) + cost(cos) + cost(actualització))
```

```
per (i:=0; i<n; i:=i+2) fer
  escriure("i=", i);
fper
```

```
per (i:=1; i<n; i:=i*2) fer
  escriure("i=", i);
fper
```

Cost de la condició: O(1)
Cost de l'actualització: O(1)
Cost del cos: O(1)

Nombre d'iteracions: n/2
Total = O(n/2) = O(n)

Càlcul de l'ordre de creixement (Big-O)

Costos de les construccions algorísmiques

Estructures iteratives

- Si coneixem el nombre d’iteracions:
- A cada iteració s'avalua una condició, s'executa el cos del bucle, i s'actualitza la variable de control. D'aquests tres blocs ens quedem amb el de cost màxim.

```
per (i:=0; i<n; i:=i+1) fer
    sentències;
fper
```

- El cost serà el nombre d'iteracions multiplicat pel cost del bloc més costós.

```
Cost total = num_iteracions * (cost(condició) + cost(cos) + cost(actualització))
```

```
per (i:=0; i<n; i:=i+2) fer
  escriure("i=", i);
fper
```

Cost de la condició: O(1)
Cost de l'actualització: O(1)
Cost del cos: O(1)

Nombre d'iteracions: n/2
Total = O(n/2) = O(n)

```
per (i:=1; i<n; i:=i*2) fer
  escriure("i=", i);
fper
```

Cost de la condició: O(1)
Cost de l'actualització: O(1)
Cost del cos: O(1)

Nombre d'iteracions: $\log_2(n)$
Total = $O(\log_2(n))$

Càlcul de l'ordre de creixement (Big-O)

Costos de les construccions algorísmiques

Estructures iteratives

- Si coneixem el nombre d'iteracions:
- A cada iteració s'avalua una condició, s'executa el cos del bucle, i s'actualitza la variable de control. D'aquests tres blocs ens quedem amb el de cost màxim.

```
per (i:=0; i<n; i:=i+1) fer
    sentències;
fper
```

- El cost serà el nombre d'iteracions multiplicat pel cost del bloc més costós.

```
Cost total = num_iteracions * (cost(condició) + cost(cos) + cost(actualització))
```

```
per (i:=0; i<n; i:=i+2) fer
  escriure("i=", i);
fper
```

Cost de la condició: O(1)
Cost de l'actualització: O(1)
Cost del cos: O(1)

Nombre d'iteracions: n/2
Total = O(n/2) = O(n)

```
per (i:=1; i<n; i:=i*2) fer
  escriure("i=", i);
fper
```

Cost de la condició: O(1)
Cost de l'actualització: O(1)
Cost del cos: O(1)

Nombre d'iteracions: $log_2(n)$
Total = $O(log_2(n))$

```
per (i:=0; i<n; i:=i+1) fer
  per (j:=1; j<m; j:=j*2) fer
     escriure(i,j);
  fper
fper
```

Càlcul de l'ordre de creixement (Big-O)

Costos de les construccions algorísmiques

Estructures iteratives

- Si coneixem el nombre d'iteracions:
- A cada iteració s'avalua una condició, s'executa el cos del bucle, i s'actualitza la variable de control. D'aquests tres blocs ens quedem amb el de cost màxim.

```
per (i:=0; i<n; i:=i+1) fer
    sentències;
fper
```

- El cost serà el nombre d'iteracions multiplicat pel cost del bloc més costós.

```
Cost total = num_iteracions * (cost(condició) + cost(cos) + cost(actualització))
```

```
per (i:=0; i<n; i:=i+2) fer
  escriure("i=", i);
fper
```

Cost de la condició: O(1)
Cost de l'actualització: O(1)
Cost del cos: O(1)

Nombre d'iteracions: n/2
Total = O(n/2) = O(n)

```
per (i:=1; i<n; i:=i*2) fer
  escriure("i=", i);
fper
```

Cost de la condició: O(1)
Cost de l'actualització: O(1)
Cost del cos: O(1)

Nombre d'iteracions: $log_2(n)$
Total = O($log_2(n)$)

```
per (i:=0; i<n; i:=i+1) fer
  per (j:=1; j<m; j:=j*2) fer
     escriure(i,j);
  fper
fper
```

Cost bucle intern (j): O(n)
Cost bucle extern (i): $log_2(m)$

Total = O(n log(m))

Càlcul de l'ordre de creixement (Big-O)

Costos de les construccions algorísmiques

Estructures iteratives

- Si **no** sabem el nombre d'iteracions:
- Hem de calcular el nombre màxim d'iteracions que farem en el pitjor cas.
- Hem de calcular el màxim entre el cost de la condició i el cos del mentre.

```
mentre (condició) fer
  sentències;
fmentre
```

Càlcul de l'ordre de creixement (Big-O)

Costos de les construccions algorísmiques

Estructures iteratives

```
mentre (condició) fer
  sentències;
fmentre
```

- Si **no** sabem el nombre d'iteracions:
- Hem de calcular el nombre màxim d'iteracions que farem en el pitjor cas.
- Hem de calcular el màxim entre el cost de la condició i el cos del mentre.

- El cost total serà:

```
Cost total = max_iteracions * (cost(condició) + cost(cos))
```

Càlcul de l'ordre de creixement (Big-O)

Costos de les construccions algorísmiques

Estructures iteratives

```
mentre (condició) fer
  sentències;
fmentre
```

- Si **no** sabem el nombre d'iteracions:
- Hem de calcular el nombre màxim d'iteracions que farem en el pitjor cas.
- Hem de calcular el màxim entre el cost de la condició i el cos del mentre.

- El cost total serà:

```
Cost total = max_iteracions * (cost(condició) + cost(cos))
```

```
mentre (i < n i no(trobat)) fer
  si (array[i] = valor_buscat) llavors
        trobat := cert;
  fsi
  i:=i+1;
fmentre
```

Càlcul de l'ordre de creixement (Big-O)

Costos de les construccions algorísmiques

Estructures iteratives

```
mentre (condició) fer
  sentències;
fmentre
```

- Si **no** sabem el nombre d'iteracions:
- Hem de calcular el nombre màxim d'iteracions que farem en el pitjor cas.
- Hem de calcular el màxim entre el cost de la condició i el cos del mentre.

- El cost total serà:

```
Cost total = max_iteracions * (cost(condició) + cost(cos))
```

```
mentre (i < n i no(trobat)) fer
  si (array[i] = valor_buscat) llavors
        trobat := cert;
  fsi
  i:=i+1;
fmentre
```

Nombre maxim d'iteracions: quan no es trobi l'element: ***n iteracions***

Cost condició: O(1)
Cost cos: O(1)
Màxim dels dos: O(1)

Total *= n * O(1)* ***= O(n)***

Càlcul de l'ordre de creixement (Big-O)

Exemple

- Analitza el cost algorísmic (Big-O) de la funció producte.

Funció producte

```
funció producte(a: enter, b: enter)
retorna enter és
var
   comptador: enter;
   resultat: enter;
fvar
inici
    comptador := 1;
    resultat := 0;
    mentre (comptador <= b) fer
        resultat := suma(resultat, a);
        comptador := comptador + 1;
    fmentre
    retorna resultat;
ffunció
```

Funció suma

```
funció suma(c: enter, d: enter) retorna
enter és
var
   comptador: enter;
   resultat: enter;
fvar
inici
    comptador := 1;
    resultat := c;
    mentre (comptador <= d) fer
        resultat := resultat + 1;
        comptador := comptador + 1;
    fmentre
    retorna resultat;
ffunció
```

Càlcul de l'ordre de creixement (Big-O)

Exemple

- Analitza el cost algorísmic (Big-O) de la funció producte.

Funció producte

```
funció producte(a: enter, b: enter)
retorna enter és
var
   comptador: enter;
   resultat: enter;
fvar
inici
    comptador := 1;
    resultat := 0;
    mentre (comptador <= b) fer
        resultat := suma(resultat, a);
        comptador := comptador + 1;
    fmentre
    retorna resultat;
ffunció
```

Com que la funció producte depèn de la funció suma, comencem analitzant la funció suma

Funció suma

```
funció suma(c: enter, d: enter) retorna
enter és
var
   comptador: enter;
   resultat: enter;
fvar
inici
    comptador := 1;
    resultat := c;
    mentre (comptador <= d) fer
        resultat := resultat + 1;
        comptador := comptador + 1;
    fmentre
    retorna resultat;
ffunció
```

Càlcul de l'ordre de creixement (Big-O)

Exemple

- Analitza el cost algorísmic (Big-O) de la funció producte (i la funció suma).

Funció suma

```
funció suma(c: enter, d: enter) retorna
enter és
var
   comptador: enter;
   resultat: enter;
fvar
inici
    comptador := 1;
    resultat := c;
    mentre (comptador <= d) fer
        resultat := resultat + 1;
        comptador := comptador + 1;
    fmentre
    retorna resultat;
ffunció
```

Càlcul de l'ordre de creixement (Big-O)

Exemple

- Analitza el cost algorísmic (Big-O) de la funció producte (i la funció suma).

Funció suma

```
funció suma(c: enter, d: enter) retorna
enter és
var
   comptador: enter;
   resultat: enter;
fvar
inici
    comptador := 1;
    resultat := c;
    mentre (comptador <= d) fer
        resultat := resultat + 1;
        comptador := comptador + 1;
    fmentre
    retorna resultat;
ffunció
```

Bloc d'instruccions:

$O(1)$

$O(1)$

$\xrightarrow{\text{max}} O(1)$

Càlcul de l'ordre de creixement (Big-O)

Exemple

- Analitza el cost algorísmic (Big-O) de la funció producte (i la funció suma).

Funció suma

```
funció suma(c: enter, d: enter) retorna
enter és
var
   comptador: enter;
   resultat: enter;
fvar
inici
    comptador := 1;
    resultat := c;
    mentre (comptador <= d) fer
        resultat := resultat + 1;
        comptador := comptador + 1;
    fmentre
    retorna resultat;
ffunció
```

Bloc d'instruccions:

comptador := 1; → O(1)
resultat := c; → O(1)

$\xrightarrow{\max}$ O(1)

Bloc d'instruccions:

resultat := resultat + 1; → O(1)
comptador := comptador + 1; → O(1)

$\xrightarrow{\max}$ O(1)

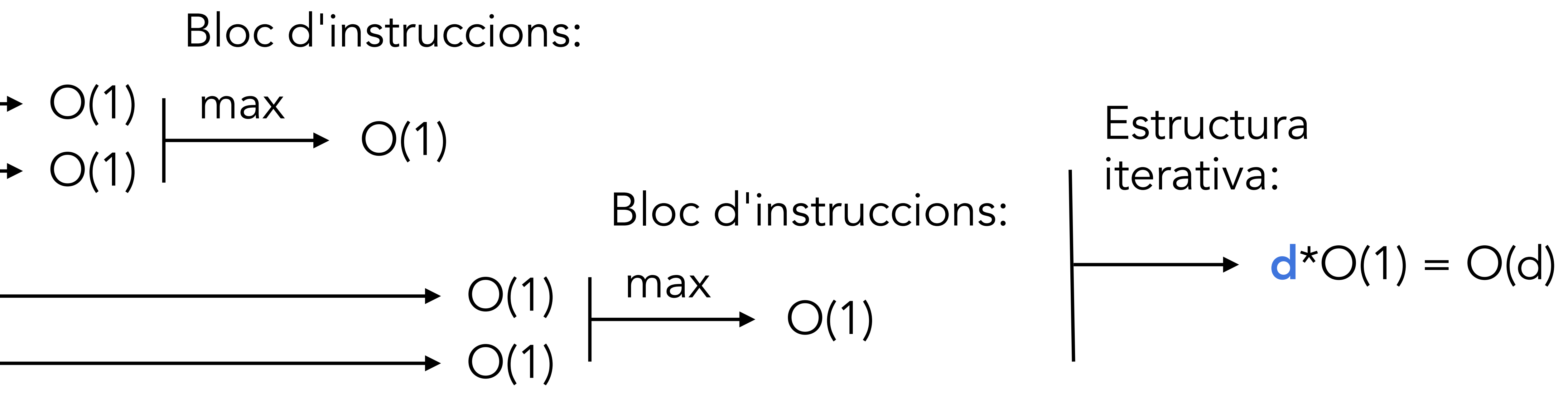
Càlcul de l'ordre de creixement (Big-O)

Exemple

- Analitza el cost algorísmic (Big-O) de la funció producte (i la funció suma).

Funció suma

```
funció suma(c: enter, d: enter) retorna
enter és
var
   comptador: enter;
   resultat: enter;
fvar
inici
    comptador := 1;
    resultat := c;
    mentre (comptador <= d) fer
        resultat := resultat + 1;
        comptador := comptador + 1;
    fmentre
    retorna resultat;
ffunció
```



Càlcul de l'ordre de creixement (Big-O)

Exemple

- Analitza el cost algorísmic (Big-O) de la funció producte (i la funció suma).

Funció suma

```
funció suma(c: enter, d: enter) retorna
enter és
var
   comptador: enter;
   resultat: enter;
fvar
inici
    comptador := 1;
    resultat := c;
    mentre (comptador <= d) fer
        resultat := resultat + 1;
        comptador := comptador + 1;
    fmentre
    retorna resultat;
ffunció
```

Bloc d'instruccions:

O(1), O(1) $\xrightarrow{\text{max}}$ O(1)

Bloc d'instruccions:

O(1), O(1) $\xrightarrow{\text{max}}$ O(1)

Estructura iterativa:

$\rightarrow$ d*O(1) = O(d)

Cost total de la funció suma = O(d)

Càlcul de l'ordre de creixement (Big-O)

Exemple

- Analitza el cost algorísmic (Big-O) de la funció producte (i la funció suma).

Funció producte

```
funció producte(a: enter, b: enter)
retorna enter és
var
   comptador: enter;
   resultat: enter;
fvar
inici
    comptador := 1;
    resultat := 0;
    mentre (comptador <= b) fer
        resultat := suma(resultat, a);
        comptador := comptador + 1;
    fmentre
    retorna resultat;
ffunció
```

Càlcul de l'ordre de creixement (Big-O)

Exemple

- Analitza el cost algorísmic (Big-O) de la funció producte (i la funció suma).

Funció producte

```
funció producte(a: enter, b: enter)
retorna enter és
var
   comptador: enter;
   resultat: enter;
fvar
inici
    comptador := 1;
    resultat := 0;
    mentre (comptador <= b) fer
        resultat := suma(resultat, a);
        comptador := comptador + 1;
    fmentre
    retorna resultat;
ffunció
```

Bloc d'instruccions:

O(1)
O(1)
$\xrightarrow{\text{max}}$ O(1)

Càlcul de l'ordre de creixement (Big-O)

Exemple

- Analitza el cost algorísmic (Big-O) de la funció producte (i la funció suma).

Funció producte

```
funció producte(a: enter, b: enter)
retorna enter és
var
   comptador: enter;
   resultat: enter;
fvar
inici
    comptador := 1;
    resultat := 0;
    mentre (comptador <= b) fer
        resultat := suma(resultat, a);
        comptador := comptador + 1;
    fmentre
    retorna resultat;
ffunció
```

Bloc d'instruccions:

O(1), O(1) $\xrightarrow{\text{max}}$ O(1)

Bloc d'instruccions:

O(a), O(1) $\xrightarrow{\text{max}}$ O(a)

Càlcul de l'ordre de creixement (Big-O)

Exemple

- Analitza el cost algorísmic (Big-O) de la funció producte (i la funció suma).

Funció producte

```
funció producte(a: enter, b: enter)
retorna enter és
var
   comptador: enter;
   resultat: enter;
fvar
inici
    comptador := 1;
    resultat := 0;
    mentre (comptador <= b) fer
        resultat := suma(resultat, a);
        comptador := comptador + 1;
    fmentre
    retorna resultat;
ffunció
```

comptador := 1; → O(1)

resultat := 0; → O(1)

Bloc d'instruccions: O(1), O(1) $\xrightarrow{\text{max}}$ O(1)

resultat := suma(resultat, a); → **O(a)**

comptador := comptador + 1; → O(1)

Bloc d'instruccions: **O(a)**, O(1) $\xrightarrow{\text{max}}$ O(a)

Estructura iterativa: → **b***O(a) = O(a*b)

Càlcul de l'ordre de creixement (Big-O)

Exemple

- Analitza el cost algorísmic (Big-O) de la funció producte (i la funció suma).

Funció producte

```
funció producte(a: enter, b: enter)
retorna enter és
var
   comptador: enter;
   resultat: enter;
fvar
inici
    comptador := 1;
    resultat := 0;
    mentre (comptador <= b) fer
        resultat := suma(resultat, a);
        comptador := comptador + 1;
    fmentre
    retorna resultat;
ffunció
```

Bloc d'instruccions:

O(1), O(1) $\xrightarrow{\text{max}}$ O(1)

Bloc d'instruccions:

O(a), O(1) $\xrightarrow{\text{max}}$ O(a)

Estructura iterativa:

b*O(a) = O(a*b)

Cost total de la funció producte = O(a*b)

Més exercicis

Exercici 1. Analitza la complexitat de l'algorisme següent:

```
// Funció que cerca un enter i en retorna la posició si
l'ha trobat, o -1 si no l'ha trobat
// Compte, algorisme no correcte, només serveix per
practicar costos. n és mida del vector
funció cerca (v: taula[] d'enter, elem: enter,
n: enter) retorna enter és
var
   posició : enter;
   trobat : enter;
fvar
inici
   posició := 1;
   trobat  := -1;
   mentre (posicio < n) fer
      si (v[posicio] = elem) llavors
         trobat := posicio;
      fsi
      posicio := posicio * 2;
   fmentre
   retorna trobat;
ffunció
```

Més exercicis

Exercici 1. Analitza la complexitat de l'algorisme següent:

```
// Funció que cerca un enter i en retorna la posició si
l'ha trobat, o -1 si no l'ha trobat
// Compte, algorisme no correcte, només serveix per
practicar costos. n és mida del vector
funció cerca (v: taula[] d'enter, elem: enter,
n: enter) retorna enter és
var
   posició : enter;
   trobat : enter;
fvar
inici
   posició := 1;
   trobat  := -1;
   mentre (posicio < n) fer
      si (v[posicio] = elem) llavors
         trobat := posicio;
      fsi
      posicio := posicio * 2;
   fmentre
   retorna trobat;
ffunció
```

Solució curta

- Totes les instruccions d'aquest codi són de complexitat $O(1)$.
- L'únic factor que ens interessa és saber quantes vegades s'executa aquest bucle, en el pitjor dels casos.
- Com que la condició és `posicio < n`, i la posició es multiplica per 2 cada cop, el bucle es farà de l'ordre de $\log(n)$ vegades, on n és la mida.

Més exercicis

Exercici 1. Analitza la complexitat de l'algorisme següent:

Solució pas a pas

- Estem intentant resoldre la pregunta "*Quantes iteracions farem com a màxim en aquest bucle?*" i la resposta curta és $\log(N)$ iteracions. A continuació demostrem per què.

Més exercicis

Exercici 1. Analitza la complexitat de l'algorisme següent:

Solució pas a pas

- Estem intentant resoldre la pregunta "*Quantes iteracions farem com a màxim en aquest bucle?*" i la resposta curta és $\log(N)$ iteracions. A continuació demostrem per què.
- Primer necessitem fer un seguiment de quant val posició a cada iteració, i extreure'n un patró:

Més exercicis

Exercici 1. Analitza la complexitat de l'algorisme següent:

Solució pas a pas

- Estem intentant resoldre la pregunta "*Quantes iteracions farem com a màxim en aquest bucle?*" i la resposta curta és $\log(N)$ iteracions. A continuació demostrem per què.
- Primer necessitem fer un seguiment de quant val posició a cada iteració, i extreure'n un patró:

Iteracions	Valor posició	En funció de i
i=0	posicio = 1	2^0
i=1	posicio = 1*2 = 2	2^1
i=2	posicio = 2*2 = 4	2^2
i=3	posicio = 4*2 = 8	2^3

Més exercicis

Exercici 1. Analitza la complexitat de l'algorisme següent:

Solució pas a pas

- Estem intentant resoldre la pregunta "*Quantes iteracions farem com a màxim en aquest bucle?*" i la resposta curta és $\log(N)$ iteracions. A continuació demostrem per què.
- Primer necessitem fer un seguiment de quant val posició a cada iteració, i extreure'n un patró:

Iteracions	Valor posició	En funció de i
i=0	posicio = 1	2^0
i=1	posicio = 1*2 = 2	2^1
i=2	posicio = 2*2 = 4	2^2
i=3	posicio = 4*2 = 8	2^3

- Per tant, veiem que el valor de posició a la iteració i valdrà 2^i

Més exercicis

Exercici 1. Analitza la complexitat de l'algorisme següent:

Solució pas a pas

- Estem intentant resoldre la pregunta "*Quantes iteracions farem com a màxim en aquest bucle?*" i la resposta curta és $\log(N)$ iteracions. A continuació demostrem per què.
- Primer necessitem fer un seguiment de quant val posició a cada iteració, i extreure'n un patró:

Iteracions	Valor posició	En funció de i
i=0	posicio = 1	2^0
i=1	posicio = 1*2 = 2	2^1
i=2	posicio = 2*2 = 4	2^2
i=3	posicio = 4*2 = 8	2^3

- Per tant, veiem que el valor de posició a la iteració i valdrà 2^i
- Quan s'acabarà el bucle? Quan ja no es compleixi la condició `posicio < n`, és a dir, quan `posició = n`.

Més exercicis

Exercici 1. Analitza la complexitat de l'algorisme següent:

Solució pas a pas

- Estem intentant resoldre la pregunta "*Quantes iteracions farem com a màxim en aquest bucle?*" i la resposta curta és $\log(N)$ iteracions. A continuació demostrem per què.
- Primer necessitem fer un seguiment de quant val posició a cada iteració, i extreure'n un patró:

Iteracions	Valor posició	En funció de i
i=0	posicio = 1	2^0
i=1	posicio = 1*2 = 2	2^1
i=2	posicio = 2*2 = 4	2^2
i=3	posicio = 4*2 = 8	2^3

- Per tant, veiem que el valor de posició a la iteració i valdrà 2^i
- Quan s'acabarà el bucle? Quan ja no es compleixi la condició `posicio < n`, és a dir, quan `posició = n`.

- Com que posició podem expressar-ho com a 2^i, podem dir que: $2^i = N$

$$2^i = N$$

Més exercicis

Exercici 1. Analitza la complexitat de l'algorisme següent:

Solució pas a pas

- Estem intentant resoldre la pregunta "*Quantes iteracions farem com a màxim en aquest bucle?*" i la resposta curta és $\log(N)$ iteracions. A continuació demostrem per què.
- Primer necessitem fer un seguiment de quant val posició a cada iteració, i extreure'n un patró:

Iteracions	Valor posició	En funció de i
i=0	posicio = 1	2^0
i=1	posicio = 1*2 = 2	2^1
i=2	posicio = 2*2 = 4	2^2
i=3	posicio = 4*2 = 8	2^3

- Per tant, veiem que el valor de posició a la iteració i valdrà 2^i
- Quan s'acabarà el bucle? Quan ja no es compleixi la condició `posicio < n`, és a dir, quan `posició = n`.

- Com que posició podem expressar-ho com a 2^i, podem dir que: $2^i = N$

$$2^i = N$$

Aplicant logaritmes a banda i banda:

$$\log(2^i) = \log(N)$$

Més exercicis

Exercici 1. Analitza la complexitat de l'algorisme següent:

Solució pas a pas

- Estem intentant resoldre la pregunta "*Quantes iteracions farem com a màxim en aquest bucle?*" i la resposta curta és $\log(N)$ iteracions. A continuació demostrem per què.
- Primer necessitem fer un seguiment de quant val posició a cada iteració, i extreure'n un patró:

Iteracions	Valor posició	En funció de i
i=0	posicio = 1	2^0
i=1	posicio = 1*2 = 2	2^1
i=2	posicio = 2*2 = 4	2^2
i=3	posicio = 4*2 = 8	2^3

- Per tant, veiem que el valor de posició a la iteració i valdrà 2^i
- Quan s'acabarà el bucle? Quan ja no es compleixi la condició `posicio < n`, és a dir, quan `posició = n`.

- Com que posició podem expressar-ho com a 2^i, podem dir que: $2^i = N$

$$2^i = N$$

Aplicant logaritmes a banda i banda:

$$\log(2^i) = \log(N)$$

Aplicant propietat de potències $\log(x^k) = k \cdot \log(x)$

$$i \cdot \log(2) = \log(N)$$

Més exercicis

Exercici 1. Analitza la complexitat de l'algorisme següent:

Solució pas a pas

- Estem intentant resoldre la pregunta "*Quantes iteracions farem com a màxim en aquest bucle?*" i la resposta curta és $\log(N)$ iteracions. A continuació demostrem per què.
- Primer necessitem fer un seguiment de quant val posició a cada iteració, i extreure'n un patró:

Iteracions	Valor posició	En funció de i
i=0	posicio = 1	2^0
i=1	posicio = 1*2 = 2	2^1
i=2	posicio = 2*2 = 4	2^2
i=3	posicio = 4*2 = 8	2^3

- Per tant, veiem que el valor de posició a la iteració i valdrà 2^i
- Quan s'acabarà el bucle? Quan ja no es compleixi la condició `posicio < n`, és a dir, quan `posició = n`.

- Com que posició podem expressar-ho com a 2^i, podem dir que: $2^i = N$

$$2^i = N$$

Aplicant logaritmes a banda i banda:

$$\log(2^i) = \log(N)$$

Aplicant propietat de potències $\log(x^k) = k \cdot \log(x)$

$$i \cdot \log(2) = \log(N)$$

Com que $\log_2(2) = 1$

$$i = \log(N)$$

Més exercicis

Exercici 1. Analitza la complexitat de l'algorisme següent:

Solució pas a pas

- Estem intentant resoldre la pregunta "*Quantes iteracions farem com a màxim en aquest bucle?*" i la resposta curta és $\log(N)$ iteracions. A continuació demostrem per què.
- Primer necessitem fer un seguiment de quant val posició a cada iteració, i extreure'n un patró:

Iteracions	Valor posició	En funció de i
i=0	posicio = 1	2^0
i=1	posicio = 1*2 = 2	2^1
i=2	posicio = 2*2 = 4	2^2
i=3	posicio = 4*2 = 8	2^3

- Per tant, veiem que el valor de posició a la iteració i valdrà 2^i
- Quan s'acabarà el bucle? Quan ja no es compleixi la condició `posicio < n`, és a dir, quan `posició = n`.

- Com que posició podem expressar-ho com a 2^i, podem dir que: $2^i = N$

$$2^i = N$$

Aplicant logaritmes a banda i banda:

$$\log(2^i) = \log(N)$$

Aplicant propietat de potències $\log(x^k) = k \cdot \log(x)$

$$i \cdot \log(2) = \log(N)$$

Com que $\log_2(2) = 1$

$$i = \log(N)$$

Aquest valor de i representa el nombre d'iteracions necessàries perquè el valor de posició arribi a N. Per tant, tenim la solució a la pregunta "Quantes iteracions farem com a màxim?", la resposta és $\log(N)$ iteracions.

Més exercicis

Exercici 2. Analitza la complexitat de l'algorisme següent:

```
// Funció que rep dues taules "v" i "resultat" de la
mateixa mida, i guarda en resultat la suma acumulada de v

acció suma_acumulada(v: taula[] d'enter,
                     resultat: taula[] d'enter,
                     mida: enter) és
var
   i : enter;
fvar
inici
   resultat[0] := v[0];
   per (i := 0; i < mida; i := i + 1) fer
     resultat[i] := resultat[i - 1] + v[i];
   fper
   retorna resultat;
facció
```

Més exercicis

Exercici 2. Analitza la complexitat de l'algorisme següent:

```
// Funció que rep dues taules "v" i "resultat" de la
mateixa mida, i guarda en resultat la suma acumulada de v

acció suma_acumulada(v: taula[] d'enter,
                     resultat: taula[] d'enter,
                     mida: enter) és
var
   i : enter;
fvar
inici
   resultat[0] := v[0];
   per (i := 0; i < mida; i := i + 1) fer
     resultat[i] := resultat[i - 1] + v[i];
   fper
   retorna resultat;
facció
```

Si el vector "v" té N elements, quina és la complexitat de l'algorisme?

Més exercicis

Exercici 2. Analitza la complexitat de l'algorisme següent:

```
// Funció que rep dues taules "v" i "resultat" de la
mateixa mida, i guarda en resultat la suma acumulada de v

acció suma_acumulada(v: taula[] d'enter,
                     resultat: taula[] d'enter,
                     mida: enter) és
var
   i : enter;
fvar
inici
   resultat[0] := v[0];
   per (i := 0; i < mida; i := i + 1) fer
     resultat[i] := resultat[i - 1] + v[i];
   fper
   retorna resultat;
facció
```

O(1)

Nombre iteracions? "`mida`"

O(1)

Si el vector "v" té N elements, quina és la complexitat de l'algorisme?

El cost total és lineal amb la mida del vector = O(mida) = O(N)

Més exercicis

Exercici 3. Analitza la complexitat de l'algorisme següent:

```
funció suma_posicions_parells (v: taula[] d'enter,
                               mida: enter)
retorna enter és
var
   suma : enter;
fvar
inici
   suma := 0;
   per (i := 0; i < mida; i := i + 2) fer
      suma := suma + v[i];
   fper
   retorna suma;
ffunció
```

Si el vector "v" té N elements,
quina és la complexitat de l'algorisme?

Més exercicis

Exercici 3. Analitza la complexitat de l'algorisme següent:

```
funció suma_posicions_parells (v: taula[] d'enter,
                               mida: enter)
retorna enter és
var
   suma : enter;
fvar
inici
   suma := 0;
   per (i := 0; i < mida; i := i + 2) fer
      suma := suma + v[i];
   fper
   retorna suma;
ffunció
```

Si el vector "v" té N elements,
quina és la complexitat de l'algorisme?

suma := 0; → O(1)

per (i := 0; i < mida; i := i + 2) fer → Nombre iteracions? **N/2** perquè fem i:=i+2;

suma := suma + v[i]; → O(1)

El cost total és lineal amb la mida del vector
= O(N/2) = O(N)

Més exercicis

Exercici 4. Analitza la complexitat de l'algorisme següent:

```
funció suma_parelles(v: taula[] d'enter,
                     mida: enter) retorna enter és
var
   suma : enter;
fvar
inici
   suma := 0;
   per (i := 0; i < mida; i := i + 1) fer
      per (j := i + 1; j < mida; j := j + 1) fer
         suma := suma + (v[i] + v[j]);
      fper
   fper
   retorna suma;
ffunció
```

O(1)

Nombre iteracions? Aquests dos bucles no són independents l'un de l'altre. El bucle j farà més o menys iteracions depenent del valor de i.
Com calcular-ho?

O(1)

Si el vector "v" té N elements, quina és la complexitat de l'algorisme?

Més exercicis

Exercici 4. Analitza la complexitat de l'algorisme següent:

```
funció suma_p

var
   suma : ent
fvar
inici
   suma := 0;
   per (i :=
      per (j
         suma
      fper
   fper
   retorna su
ffunció
```

Per cada valor de `(i,j)`, recorre des de `i+1` fins a `mida-1`
Això vol dir que la quantitat d'iteracions de `j` depèn del valor de `i`:

- Si `i=0`, `j` pren els valors `1, 2, 3,..., mida-1` → fa `mida-1` iteracions
- Si `i=1`, `j` pren els valors `2, 3,..., mida-1` → fa `mida-2` iteracions
- Si `i=2`, `j` pren els valors `3,..., mida-1` → fa `mida-3` iteracions
- Si `i=mida-2`, `j` fa `1` iteració
- Si `i=mida-1`, `j` ja no fa cap iteració, perquè seria `j = mida` i no entra

Més exercicis

Exercici 4. Analitza la complexitat de l'algorisme següent:

```
funció suma_p

var
   suma : ent
fvar
inici
   suma := 0;
   per (i :=
      per (j
         suma
      fper
   fper
   retorna su
ffunció
```

Per cada valor de `(i,j)`, recorre des de `i+1` fins a `mida-1`
Això vol dir que la quantitat d'iteracions de `j` depèn del valor de `i`:

- Si `i=0`, `j` pren els valors `1, 2, 3,..., mida-1` → fa `mida-1` iteracions
- Si `i=1`, `j` pren els valors `2, 3,..., mida-1` → fa `mida-2` iteracions
- Si `i=2`, `j` pren els valors `3,..., mida-1` → fa `mida-3` iteracions
- Si `i=mida-2`, `j` fa `1` iteració
- Si `i=mida-1`, `j` ja no fa cap iteració, perquè seria `j = mida` i no entra

El total d'iteracions serà:

$$(mida-1) + (mida-2) + (mida-3) + \ldots + 1 + 0 = \frac{mida\ (mida - 1)}{2}$$

Més exercicis

Exercici 4. Analitza la complexitat de l'algorisme següent:

```
funció suma_p

var
   suma : ent
fvar
inici
   suma := 0;
   per (i :=
      per (j
         suma
      fper
   fper
   retorna su
ffunció
```

Per cada valor de `(i,j)`, recorre des de `i+1` fins a `mida-1`
Això vol dir que la quantitat d'iteracions de `j` depèn del valor de `i`:

- Si `i=0`, `j` pren els valors `1, 2, 3,..., mida-1` → fa `mida-1` iteracions
- Si `i=1`, `j` pren els valors `2, 3,..., mida-1` → fa `mida-2` iteracions
- Si `i=2`, `j` pren els valors `3,..., mida-1` → fa `mida-3` iteracions
- Si `i=mida-2`, `j` fa `1` iteració
- Si `i=mida-1`, `j` ja no fa cap iteració, perquè seria `j = mida` i no entra

El total d'iteracions serà:

$$(mida-1) + (mida-2) + (mida-3) + \ldots + 1 + 0 = \frac{mida\ (mida - 1)}{2}$$

Suma aritmètica

Més exercicis

Exercici 4. Analitza la complexitat de l'algorisme següent:

```
funció suma_p

var
   suma : ent
fvar
inici
   suma := 0;
   per (i :=
      per (j
         suma
      fper
   fper
   retorna su
ffunció
```

Per cada valor de `(i,j)`, recorre des de `i+1` fins a `mida-1`
Això vol dir que la quantitat d'iteracions de `j` depèn del valor de `i`:

- Si `i=0`, `j` pren els valors `1, 2, 3,..., mida-1` → fa `mida-1` iteracions
- Si `i=1`, `j` pren els valors `2, 3,..., mida-1` → fa `mida-2` iteracions
- Si `i=2`, `j` pren els valors `3,..., mida-1` → fa `mida-3` iteracions
- Si `i=mida-2`, `j` fa `1` iteració
- Si `i=mida-1`, `j` ja no fa cap iteració, perquè seria `j = mida` i no entra

El total d'iteracions serà:

$$(mida-1) + (mida-2) + (mida-3) + \ldots + 1 + 0 = \frac{mida\ (mida - 1)}{2}$$

Suma aritmètica

Anomenant N a mida, seria: O(N(N-1)/2) = **$O(N^2)$**

Més exercicis

Exercici 4. Analitza la complexitat de l'algorisme següent:

```
funció suma_parelles(v: taula[] d'enter,
                     mida: enter) retorna enter és
var
   suma : enter;
fvar
inici
   suma := 0;
   per (i := 0; i < mida; i := i + 1) fer
      per (j := i + 1; j < mida; j := j + 1) fer
         suma := suma + (v[i] + v[j]);
      fper
   fper
   retorna suma;
ffunció
```

O(1)

Nombre iteracions? **$O(N^2)$**

O(1)

Si el vector "v" té N elements, quina és la complexitat de l'algorisme?

El cost total és quadràtic amb la mida del vector = **$O(N^2)$**

Exercici

Exercici: calculeu el cost asimptòtic d'aquest algorisme

```
acció qualsevol (a: taula[] d'enter, n: enter) és
var
  i,j,k,x : enter;
fvar
inici
  per (i:=0; i<n; i++) fer
    x := a[i];
    k := i;
    per (j:=i+1; j<n; j++) fer
      si (a[j] < x) llavors
        x:=a[j];
        k:=j;
      fsi
    fper
    a[k] := a[i];
    a[i] := x;
  fper
facció
```

Exercici

Solució

Comencem per les instruccions simples més internes

```
acció qualsevol (a: taula[] d'enter, n: enter) és
var
  i,j,k,x : enter;
fvar
inici
  per (i:=0; i<n; i++) fer
    x := a[i];
    k := i;
    per (j:=i+1; j<n; j++) fer
      si (a[j] < x) llavors
        x:=a[j];
        k:=j;
      fsi
    fper
    a[k] := a[i];
    a[i] := x;
  fper
facció
```

Exercici

Solució

Comencem per les instruccions simples més internes

```
acció qualsevol (a: taula[] d'enter, n: enter) és
var
  i,j,k,x : enter;
fvar
inici
  per (i:=0; i<n; i++) fer
    x := a[i];
    k := i;
    per (j:=i+1; j<n; j++) fer
      si (a[j] < x) llavors
        x:=a[j];
        k:=j;
      fsi
    fper
    a[k] := a[i];
    a[i] := x;
  fper
facció
```

O(1)

Exercici

Solució

Comencem per les instruccions simples més internes

```
acció qualsevol (a: taula[] d'enter, n: enter) és
var
  i,j,k,x : enter;
fvar
inici
  per (i:=0; i<n; i++) fer
    x := a[i];                                   O(1)
    k := i;                                      O(1)
    per (j:=i+1; j<n; j++) fer
      si (a[j] < x) llavors
        x:=a[j];
        k:=j;
      fsi
    fper
    a[k] := a[i];
    a[i] := x;
  fper
facció
```

Exercici

Solució

Comencem per les instruccions simples més internes

```
acció qualsevol (a: taula[] d'enter, n: enter) és
var
  i,j,k,x : enter;
fvar
inici
  per (i:=0; i<n; i++) fer
    x := a[i];          O(1)
    k := i;             O(1)
    per (j:=i+1; j<n; j++) fer
      si (a[j] < x) llavors
        x:=a[j];        O(1)
        k:=j;
      fsi
    fper
    a[k] := a[i];
    a[i] := x;
  fper
facció
```

Exercici

Solució

Comencem per les instruccions simples més internes

```
acció qualsevol (a: taula[] d'enter, n: enter) és
var
  i,j,k,x : enter;
fvar
inici
  per (i:=0; i<n; i++) fer
    x := a[i];           → O(1)
    k := i;              → O(1)
    per (j:=i+1; j<n; j++) fer
      si (a[j] < x) llavors
        x:=a[j];         → O(1)
        k:=j;            → O(1)
      fsi
    fper
    a[k] := a[i];
    a[i] := x;
  fper
facció
```

Exercici

Solució

Comencem per les instruccions simples més internes

```
acció qualsevol (a: taula[] d'enter, n: enter) és
var
  i,j,k,x : enter;
fvar
inici
  per (i:=0; i<n; i++) fer
    x := a[i];                      O(1)
    k := i;                         O(1)
    per (j:=i+1; j<n; j++) fer
      si (a[j] < x) llavors
        x:=a[j];                    O(1)
        k:=j;                       O(1)
      fsi
    fper
    a[k] := a[i];                   O(1)
    a[i] := x;
  fper
facció
```

Exercici

Solució

Comencem per les instruccions simples més internes

```
acció qualsevol (a: taula[] d'enter, n: enter) és
var
  i,j,k,x : enter;
fvar
inici
  per (i:=0; i<n; i++) fer
    x := a[i];            → O(1)
    k := i;               → O(1)
    per (j:=i+1; j<n; j++) fer
      si (a[j] < x) llavors
        x:=a[j];          → O(1)
        k:=j;             → O(1)
      fsi
    fper
    a[k] := a[i];         → O(1)
    a[i] := x;            → O(1)
  fper
facció
```

Exercici

Solució

Anem pujant un nivell: aquí ens trobem un condicional.

```
acció qualsevol (a: taula[] d'enter, n: enter) és
var
  i,j,k,x : enter;
fvar
inici
  per (i:=0; i<n; i++) fer
    x := a[i];                    → O(1)
    k := i;                       → O(1)
    per (j:=i+1; j<n; j++) fer
      si (a[j] < x) llavors
        x:=a[j];                  → O(1)
        k:=j;                     → O(1)
      fsi
    fper
    a[k] := a[i];                 → O(1)
    a[i] := x;                    → O(1)
  fper
facció
```

Exercici

Solució

Anem pujant un nivell: aquí ens trobem un condicional.

```
acció qualsevol (a: taula[] d'enter, n: enter) és
var
  i,j,k,x : enter;
fvar
inici
  per (i:=0; i<n; i++) fer
    x := a[i];                  → O(1)
    k := i;                     → O(1)
    per (j:=i+1; j<n; j++) fer
      si (a[j] < x) llavors
        x:=a[j];                → O(1)
        k:=j;                   → O(1)
      fsi
    fper
    a[k] := a[i];               → O(1)
    a[i] := x;                  → O(1)
  fper
facció
```

Exercici

Solució

```
acció qualsevol (a: taula[] d'enter, n: enter) és
var
  i,j,k,x : enter;
fvar
inici
  per (i:=0; i<n; i++) fer
    x := a[i];                    O(1)
    k := i;                       O(1)
    per (j:=i+1; j<n; j++) fer
      si (a[j] < x) llavors           O(1)
        x:=a[j];                  O(1)
        k:=j;                     O(1)
      fsi
    fper
    a[k] := a[i];                 O(1)
    a[i] := x;                    O(1)
  fper
facció
```

Anem pujant un nivell: aquí ens trobem un condicional.

- La comparació és una instrucció simple d'ordre O(1)

Exercici

Solució

Anem pujant un nivell: aquí ens trobem un condicional.

- La comparació és una instrucció simple d'ordre O(1)
- El cos del "si" són dues instruccions seqüencials (r.suma)

```
acció qualsevol (a: taula[] d'enter, n: enter) és
var
  i,j,k,x : enter;
fvar
inici
  per (i:=0; i<n; i++) fer
    x := a[i];                  O(1)
    k := i;                     O(1)
    per (j:=i+1; j<n; j++) fer
      si (a[j] < x) llavors          O(1)
        x:=a[j];                O(1)
        k:=j;                   O(1)   max = O(1)
      fsi
    fper
    a[k] := a[i];               O(1)
    a[i] := x;                  O(1)
  fper
facció
```

Exercici

Solució

```
acció qualsevol (a: taula[] d'enter, n: enter) és
var
  i,j,k,x : enter;
fvar
inici
  per (i:=0; i<n; i++) fer
    x := a[i];                        → O(1)
    k := i;                           → O(1)
    per (j:=i+1; j<n; j++) fer
      si (a[j] < x) llavors           → O(1)
        x:=a[j];                      → O(1)
        k:=j;                         → O(1)      max = O(1)
      fsi
    fper
    a[k] := a[i];                     → O(1)
    a[i] := x;                        → O(1)
  fper
facció
```

Anem pujant un nivell: aquí ens trobem un condicional.

- La comparació és una instrucció simple d'ordre O(1)
- El cos del "si" són dues instruccions seqüencials (r.suma)
- Com resolem un condicional:

  `cost (condició) + max (cost(llavors), cost(sino))`

Exercici

Solució

```
acció qualsevol (a: taula[] d'enter, n: enter) és
var
  i,j,k,x : enter;
fvar
inici
  per (i:=0; i<n; i++) fer
    x := a[i];              → O(1)
    k := i;                 → O(1)
    per (j:=i+1; j<n; j++) fer
      si (a[j] < x) llavors → O(1)
        x:=a[j];            → O(1)
        k:=j;               → O(1)   max = O(1)
      fsi
    fper
    a[k] := a[i];           → O(1)
    a[i] := x;              → O(1)
  fper
facció
```

Anem pujant un nivell: aquí ens trobem un condicional.

- La comparació és una instrucció simple d'ordre O(1)
- El cos del "si" són dues instruccions seqüencials (r.suma)
- Com resolem un condicional:

`cost (condició) + max (cost(llavors), cost(sino))`

- Com que el '+' anterior denota la regla de la suma, realment el que fem és:

`max (cost (condició), cost(llavors), cost(sino))`

Exercici

Solució

```
acció qualsevol (a: taula[] d'enter, n: enter) és
var
  i,j,k,x : enter;
fvar
inici
  per (i:=0; i<n; i++) fer
    x := a[i];                          → O(1)
    k := i;                             → O(1)
    per (j:=i+1; j<n; j++) fer
      si (a[j] < x) llavors             → O(1)
        x:=a[j];                        → O(1)
        k:=j;                           → O(1)
      fsi
    fper
    a[k] := a[i];                       → O(1)
    a[i] := x;                          → O(1)
  fper
facció
```

max = **O(1)**

max = **O(1)**

Anem pujant un nivell: aquí ens trobem un condicional.

- La comparació és una instrucció simple d'ordre O(1)
- El cos del "si" són dues instruccions seqüencials (r.suma)
- Com resolem un condicional:

`cost (condició) + max (cost(llavors), cost(sino))`

- Com que el '+' anterior denota la regla de la suma, realment el que fem és:

`max (cost (condició), cost(llavors), cost(sino))`

Exercici

Solució

Al nivell següent ens trobem un “per”

```
acció qualsevol (a: taula[] d'enter, n: enter) és
var
  i,j,k,x : enter;
fvar
inici
  per (i:=0; i<n; i++) fer
    x := a[i];                    → O(1)
    k := i;                       → O(1)
    per (j:=i+1; j<n; j++) fer
      si (a[j] < x) llavors       → O(1)
        x:=a[j];                  → O(1)
        k:=j;                     → O(1)
      fsi
    fper
    a[k] := a[i];                 → O(1)
    a[i] := x;                    → O(1)
  fper
facció
```

max = **O(1)**

max = **O(1)**

Exercici

Solució

```
acció qualsevol (a: taula[] d'enter, n: enter) és
var
  i,j,k,x : enter;
fvar
inici
  per (i:=0; i<n; i++) fer
    x := a[i];
    k := i;
    per (j:=i+1; j<n; j++) fer
      si (a[j] < x) llavors
        x:=a[j];
        k:=j;
      fsi
    fper
    a[k] := a[i];
    a[i] := x;
  fper
facció
```

O(1)

O(1)

num_iteracions = n - i -1 = **O(n)**

O(1)

O(1)

O(1)

max = **O(1)**

max = **O(1)**

O(1)

O(1)

Al nivell següent ens trobem un "per"

- Quantes vegades es fa aquest bucle? n-(i+1)= n-i-1

Exercici

Solució

Al nivell següent ens trobem un "per"

- Quantes vegades es fa aquest bucle? n-(i+1)= n-i-1
- Com que estan anidats fem servir la regla del producte

```
acció qualsevol (a: taula[] d'enter, n: enter) és
var
  i,j,k,x : enter;
fvar
inici
  per (i:=0; i<n; i++) fer
    x := a[i];                 → O(1)
    k := i;                    → O(1)
    per (j:=i+1; j<n; j++) fer → num_iteracions = n - i -1 = O(n)
      si (a[j] < x) llavors    → O(1)
        x:=a[j];               → O(1)
        k:=j;                  → O(1)
      fsi
    fper
    a[k] := a[i];              → O(1)
    a[i] := x;                 → O(1)
  fper
facció
```

max = **O(1)**

max = **O(1)**

Regla del producte
O(n)*O(1) = O(n)

Exercici

Solució

Ara podem calcular el cost de tota la seqüència de dins del "per"

```
acció qualsevol (a: taula[] d'enter, n: enter) és
var
  i,j,k,x : enter;
fvar
inici
  per (i:=0; i<n; i++) fer
    x := a[i];                          → O(1)
    k := i;                             → O(1)
    per (j:=i+1; j<n; j++) fer          → num_iteracions = n - i -1 = O(n)
      si (a[j] < x) llavors             → O(1)
        x:=a[j];                        → O(1)
        k:=j;                           → O(1)
      fsi
    fper
    a[k] := a[i];                       → O(1)
    a[i] := x;                          → O(1)
  fper
facció
```

max = **O(1)**

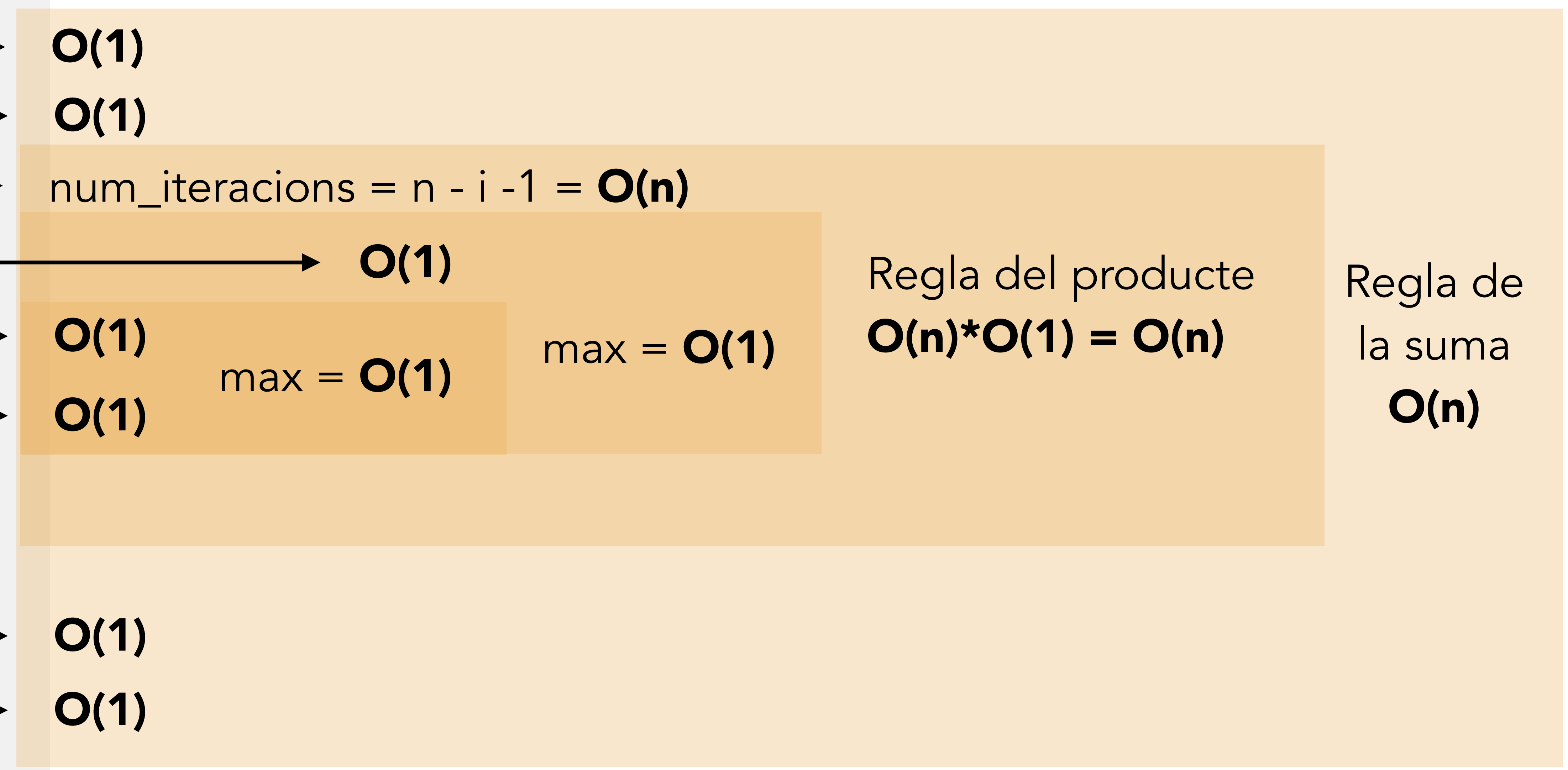
max = **O(1)**

Regla del producte
O(n)*O(1) = O(n)

Exercici

Solució

```
acció qualsevol (a: taula[] d'enter, n: enter) és
var
  i,j,k,x : enter;
fvar
inici
  per (i:=0; i<n; i++) fer
    x := a[i];
    k := i;
    per (j:=i+1; j<n; j++) fer
      si (a[j] < x) llavors
        x:=a[j];
        k:=j;
      fsi
    fper
    a[k] := a[i];
    a[i] := x;
  fper
facció
```



Ara podem calcular el cost de tota la seqüència de dins del "per"

- Com que son instruccions seqüencials, es fa servir la regla de la suma: O(1) + O(1) + O(n) + O(1) + O(1) = O(n)

Exercici

Solució

Ara ens centrem en el “per” més extern. Quantes vegades es fa? n. Per tant, O(n)

```
acció qualsevol (a: taula[] d'enter, n: enter) és
var
  i,j,k,x : enter;
fvar
inici
  per (i:=0; i<n; i++) fer
    x := a[i];                  → O(1)
    k := i;                     → O(1)
    per (j:=i+1; j<n; j++) fer  → num_iteracions = n - i -1 = O(n)
      si (a[j] < x) llavors     → O(1)
        x:=a[j];                → O(1)
        k:=j;                   → O(1)
      fsi
    fper
    a[k] := a[i];               → O(1)
    a[i] := x;                  → O(1)
  fper
facció
```

max = **O(1)**

max = **O(1)**

Regla del producte
O(n)*O(1) = O(n)

Regla de la suma
O(n)

Exercici

Solució

Ara ens centrem en el "per" més extern. Quantes vegades es fa? n. Per tant, O(n)

```
acció qualsevol (a: taula[] d'enter, n: enter) és
var
  i,j,k,x : enter;
fvar
inici
  per (i:=0; i<n; i++) fer
    x := a[i];
    k := i;
    per (j:=i+1; j<n; j++) fer
      si (a[j] < x) llavors
        x:=a[j];
        k:=j;
      fsi
    fper
    a[k] := a[i];
    a[i] := x;
  fper
facció
```

num_iteracions = n = **O(n)**

O(1)

O(1)

num_iteracions = n - i -1 = **O(n)**

O(1)

O(1)

O(1)

max = **O(1)**

max = **O(1)**

Regla del producte **O(n)*O(1) = O(n)**

Regla de la suma **O(n)**

O(1)

O(1)

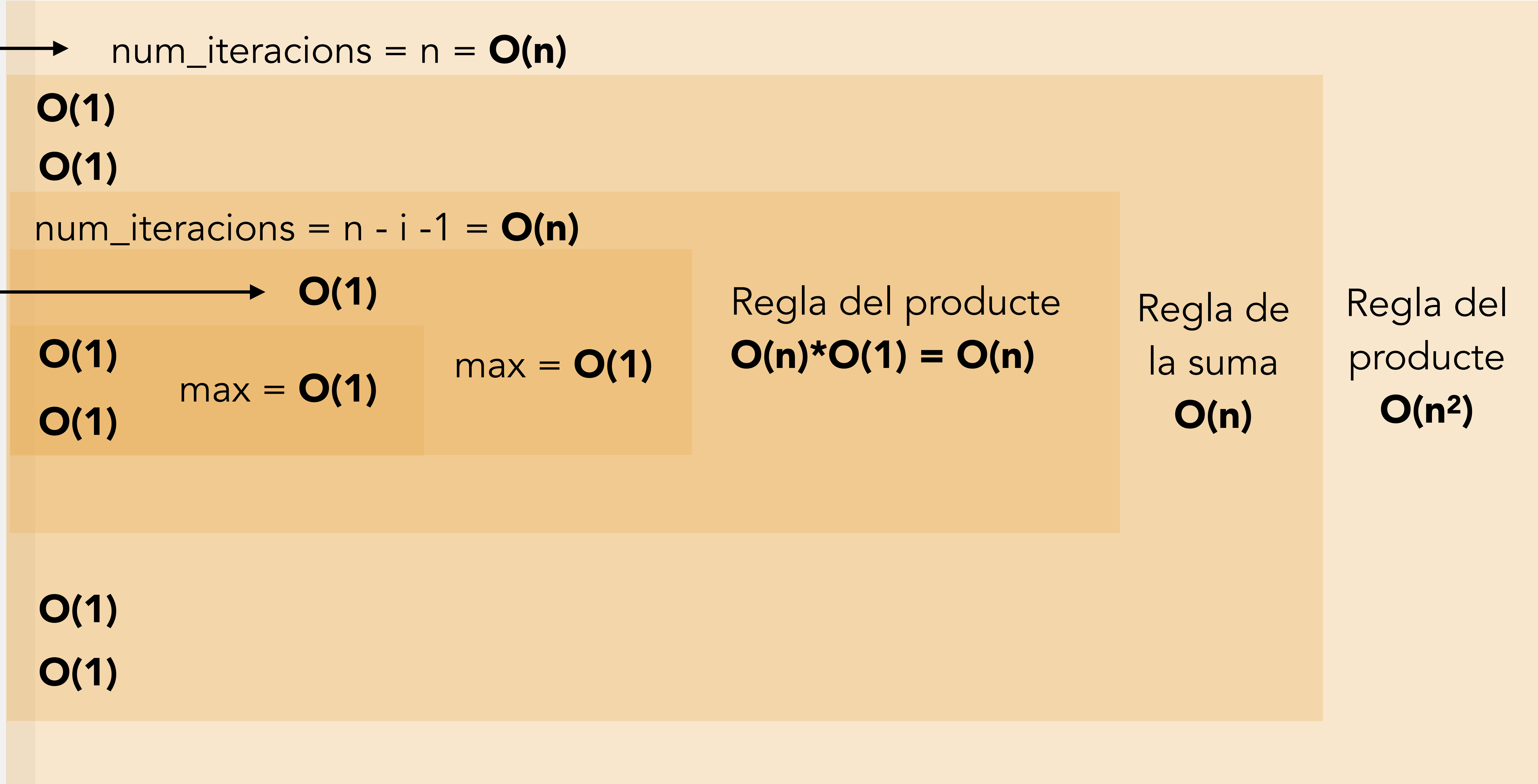
Exercici

Solució

Ara ens centrem en el "per" més extern. Quantes vegades es fa? n. Per tant, O(n)

- Com que el codi anterior que era O(n) està dins del bucle que es fa n vegades i també és O(n), aplicant la regla del producte tenim: $O(n)*O(n) = O(n^2)$

```
acció qualsevol (a: taula[] d'enter, n: enter) és
var
  i,j,k,x : enter;
fvar
inici
  per (i:=0; i<n; i++) fer
    x := a[i];
    k := i;
    per (j:=i+1; j<n; j++) fer
      si (a[j] < x) llavors
        x:=a[j];
        k:=j;
      fsi
    fper
    a[k] := a[i];
    a[i] := x;
  fper
facció
```



Resum

	Funció de cost T(n)	Ordre de creixement (Big-O)
Definició	Funció que expressa el **temps/cost exacte d'un algorisme en funció de la mida de l'entrada.**	**Representació asimptòtica del cost**, centrada en el comportament per a mides grans d'entrada.
Detall	Analitza **totes les operacions**, constants incloses.	Només considera els **termes dominants** i ignora constants i termes menors.
Precisió	Proporciona una mesura **detallada** i precisa.	Mesura general que **simplifica** el comportament global.
Usos principals	**Comparació precisa d'algoritmes** amb mides petites o mitjanes.	**Comparació de l'eficiència d'algoritmes per a mides grans d'entrada.**
Complexitat de càlcul	Pot requerir **anàlisi detallada** del codi i del nombre d'operacions.	Més simple, centrant-se en els **termes dominants.**
Exemple	$T(n) = 3n^2 + 5n + 2$	$O(n^2)$

Bibliografia addicional

- **Manual de Algorítmica**
- Jesús Bisbal Riera
- ISBN: 9788497880275
- Publicació: 12/2009
- Idiomes disponibles: Castellà i Català
- 🔗 Està disponible al CRAI de la URV, en versió paper i digital (link).