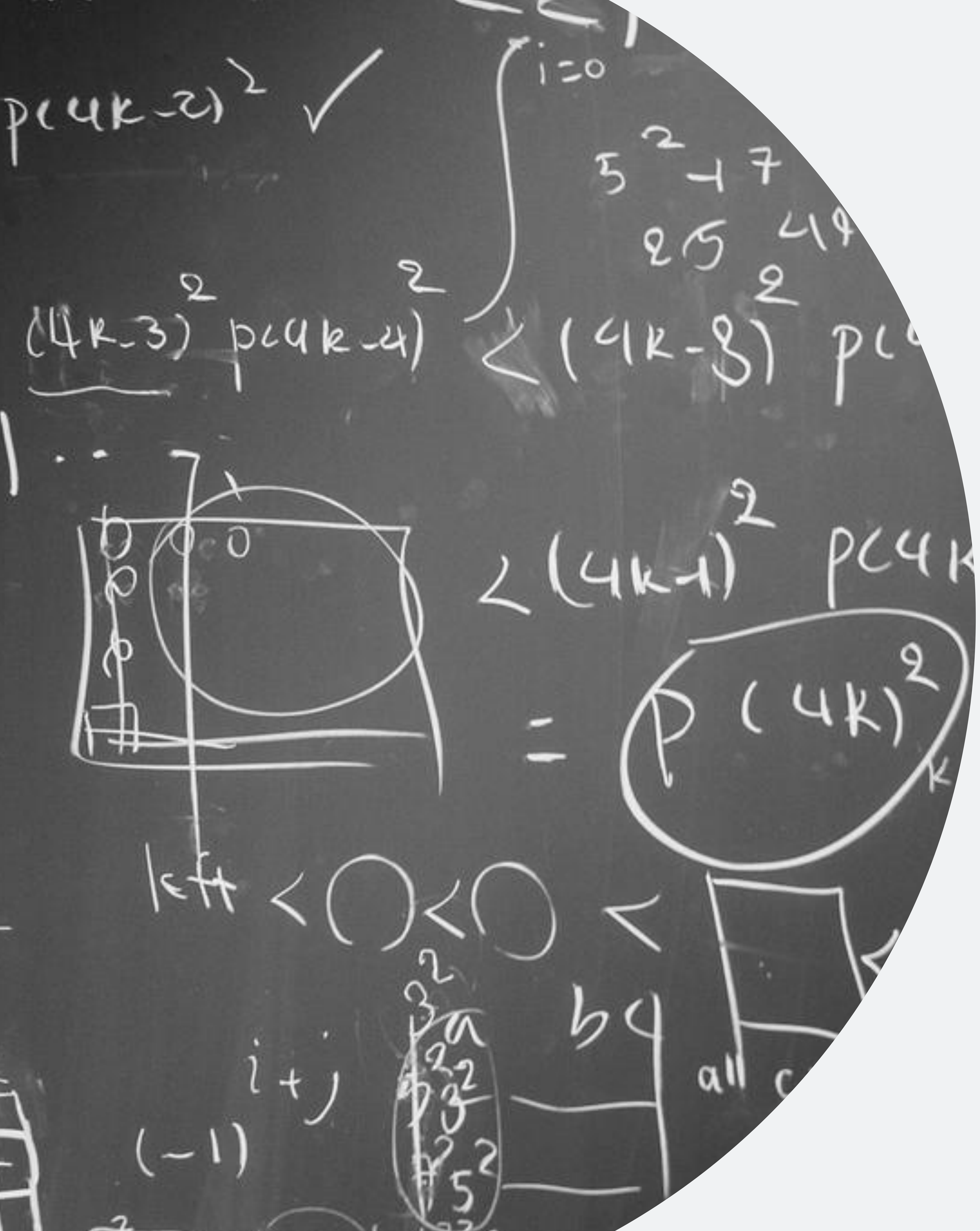


TEMA 6.

ALGORISMES RECURSIUS.

TEORIA.

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

> ***To understand recursion,***
> ***you first need to understand recursion.***

Concepte de recursivitat

- Fins ara, cada cop que hem hagut de fer càlculs repetitius, ho hem fet a través de bucles (while, do-while, for).
- Hi ha una altra manera de fer càlculs repetitius, i és amb mètodes recursius.

Concepte de recursivitat

- Fins ara, cada cop que hem hagut de fer càlculs repetitius, ho hem fet a través de bucles (while, do-while, for).
- Hi ha una altra manera de fer càlculs repetitius, i és amb mètodes recursius.

Es diu que un objecte és recursiu quan es defineix en funció de si mateix.

Concepte de recursivitat

- Fins ara, cada cop que hem hagut de fer càlculs repetitius, ho hem fet a través de bucles (while, do-while, for).
- Hi ha una altra manera de fer càlculs repetitius, i és amb mètodes recursius.

Es diu que un objecte és recursiu quan es defineix en funció de si mateix.

- Per exemple, la definició dels nombres naturals és:

(I) 1 és un nombre natural, i (II) el següent d'un nombre natural també és un nombre natural.

Aquesta definició és recursiva ja que per definir què és un nombre natural estem fent servir el concepte de nombre natural.

Interpretació: com que 1 és nombre natural i el següent d'un nombre natural també ho és, això vol dir que el 2 també ho és. I com que el 2 ho és, això vol dir que el següent, el 3, també ho és...

Concepte de recursivitat

- La funció factorial es pot definir…

No recursivament (iterativament):

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

$$4! = 4 \cdot 3 \cdot 2 \cdot 1$$

Concepte de recursivitat

- La funció factorial es pot definir…

No recursivament (iterativament):

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

$$4! = 4 \cdot 3 \cdot 2 \cdot 1$$

Recursivament:

$$n! = \begin{cases} 1 & \text{si } n = 0 \\ n \cdot (n-1)! & \text{si } n > 0 \end{cases}$$

Concepte de recursivitat

- La funció factorial es pot definir…

No recursivament (iterativament):

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

$$4! = 4 \cdot 3 \cdot 2 \cdot 1$$

Recursivament:

$$n! = \begin{cases} 1 & \text{si } n = 0 \\ n \cdot (n-1)! & \text{si } n > 0 \end{cases}$$

$$4! = 4 \cdot 3!$$

Concepte de recursivitat

- La funció factorial es pot definir...

No recursivament (iterativament):

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

$$4! = 4 \cdot 3 \cdot 2 \cdot 1$$

Recursivament:

$$n! = \begin{cases} 1 & \text{si } n = 0 \\ n \cdot (n-1)! & \text{si } n > 0 \end{cases}$$

$$4! = 4 \cdot 3!$$

$$\downarrow$$

$$3! = 3 \cdot 2!$$

Concepte de recursivitat

- La funció factorial es pot definir…

No recursivament (iterativament):

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

$$4! = 4 \cdot 3 \cdot 2 \cdot 1$$

Recursivament:

$$n! = \begin{cases} 1 & \text{si } n = 0 \\ n \cdot (n-1)! & \text{si } n > 0 \end{cases}$$

$$4! = 4 \cdot 3!$$

$$\downarrow$$

$$3! = 3 \cdot 2!$$

$$\downarrow$$

$$2! = 2 \cdot 1!$$

Concepte de recursivitat

- La funció factorial es pot definir...

No recursivament (iterativament):

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

$$4! = 4 \cdot 3 \cdot 2 \cdot 1$$

Recursivament:

$$n! = \begin{cases} 1 & \text{si } n = 0 \\ n \cdot (n-1)! & \text{si } n > 0 \end{cases}$$

$$4! = 4 \cdot 3!$$

$$3! = 3 \cdot 2!$$

$$2! = 2 \cdot 1!$$

$$1! = 1 \cdot 0!$$

Concepte de recursivitat

- La funció factorial es pot definir…

No recursivament (iterativament):

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

$$4! = 4 \cdot 3 \cdot 2 \cdot 1$$

Recursivament:

$$n! = \begin{cases} 1 & \text{si } n = 0 \\ n \cdot (n-1)! & \text{si } n > 0 \end{cases}$$

$$\begin{aligned} 4! &= 4 \cdot 3! \\ &\downarrow \\ 3! &= 3 \cdot 2! \\ &\downarrow \\ 2! &= 2 \cdot 1! \\ &\downarrow \\ 1! &= 1 \cdot 0! \\ &\downarrow \\ &1 \end{aligned}$$

Concepte de recursivitat

- La suma de dos nombres naturals també es pot definir de manera recursiva:

$$suma(a,b) = \begin{cases} a & \text{si } b = 0 \\ suma(a, b-1) + 1 & \text{si } b > 0 \end{cases}$$

Concepte de recursivitat

- La suma de dos nombres naturals també es pot definir de manera recursiva:

$$suma(a,b) = \begin{cases} a & \text{si } b = 0 \\ suma(a, b-1) + 1 & \text{si } b > 0 \end{cases}$$

$$suma(4,3) = suma(4,2) + 1$$

Concepte de recursivitat

- La suma de dos nombres naturals també es pot definir de manera recursiva:

$$suma(a, b) = \begin{cases} a & \text{si } b = 0 \\ suma(a, b - 1) + 1 & \text{si } b > 0 \end{cases}$$

$$suma(4,3) = suma(4,2) + 1$$

$$\downarrow$$

$$suma(4,2) = suma(4,1) + 1$$

Concepte de recursivitat

- La suma de dos nombres naturals també es pot definir de manera recursiva:

$$suma(a,b) = \begin{cases} a & \text{si } b = 0 \\ suma(a, b-1) + 1 & \text{si } b > 0 \end{cases}$$

$$suma(4,3) = suma(4,2) + 1$$

$$\downarrow$$

$$suma(4,2) = suma(4,1) + 1$$

$$\downarrow$$

$$suma(4,1) = suma(4,0) + 1$$

Concepte de recursivitat

- La suma de dos nombres naturals també es pot definir de manera recursiva:

$$suma(a,b) = \begin{cases} a & \text{si } b = 0 \\ suma(a, b-1) + 1 & \text{si } b > 0 \end{cases}$$

$$suma(4,3) = suma(4,2) + 1$$

$$\downarrow$$

$$suma(4,2) = suma(4,1) + 1$$

$$\downarrow$$

$$suma(4,1) = suma(4,0) + 1$$

$$\downarrow$$

$$4$$

Concepte de recursivitat

- La suma de dos nombres naturals també es pot definir de manera recursiva:

$$suma(a, b) = \begin{cases} a & \text{si } b = 0 \\ suma(a, b-1) + 1 & \text{si } b > 0 \end{cases}$$

$$suma(4,3) = suma(4,2) + 1$$

$$suma(4,2) = suma(4,1) + 1$$

$$suma(4,1) = suma(4,0) + 1 = 5$$

$$4$$

Concepte de recursivitat

- La suma de dos nombres naturals també es pot definir de manera recursiva:

$$suma(a,b) = \begin{cases} a & \text{si } b = 0 \\ suma(a, b-1) + 1 & \text{si } b > 0 \end{cases}$$

$$suma(4,3) = suma(4,2) + 1$$

$$\downarrow$$

$$suma(4,2) = suma(4,1) + 1 \quad = 6$$

$$\downarrow$$

$$suma(4,1) = suma(4,0) + 1 = 5$$

$$\downarrow$$

$$4$$

Concepte de recursivitat

- La suma de dos nombres naturals també es pot definir de manera recursiva:

$$suma(a,b) = \begin{cases} a & \text{si } b = 0 \\ suma(a, b-1) + 1 & \text{si } b > 0 \end{cases}$$

$$suma(4,3) = suma(4,2) + 1 = 7$$

$$\downarrow$$

$$suma(4,2) = suma(4,1) + 1 = 6$$

$$\downarrow$$

$$suma(4,1) = suma(4,0) + 1 = 5$$

$$\downarrow$$

$$4$$

Estructura de la recursivitat

- En tots els exemples anteriors, seguim la mateixa estructura:

$$suma(a, b) = \begin{cases} a & \text{si } b = 0 \\ suma(a, b-1) + 1 & \text{si } b > 0 \end{cases}$$

$$n! = \begin{cases} 1 & \text{si } n = 0 \\ n \cdot (n-1)! & \text{si } n > 0 \end{cases}$$

Estructura de la recursivitat

- En tots els exemples anteriors, seguim la mateixa estructura:

$$suma(a, b) = \begin{cases} a & \text{si } b = 0 \\ suma(a, b-1) + 1 & \text{si } b > 0 \end{cases}$$

$$n! = \begin{cases} 1 & \text{si } n = 0 \\ n \cdot (n-1)! & \text{si } n > 0 \end{cases}$$

Un* cas base (o cas trivial, o cas directe)
És el cas en què el resultat és immediat i no cal fer cap crida recursiva.
Actua com el punt d'aturada de la recursió, garantint que aquesta no esdevingui infinita.

**Després veurem que hi pot haver més d'un cas base i més d'un cas recursiu*

Estructura de la recursivitat

- En tots els exemples anteriors, seguim la mateixa estructura:

$$suma(a,b) = \begin{cases} a & \text{si } b = 0 \\ suma(a, b-1) + 1 & \text{si } b > 0 \end{cases}$$

$$n! = \begin{cases} 1 & \text{si } n = 0 \\ n \cdot (n-1)! & \text{si } n > 0 \end{cases}$$

Un* cas base (o cas trivial, o cas directe)
És el cas en què el resultat és immediat i no cal fer cap crida recursiva.
Actua com el punt d'aturada de la recursió, garantint que aquesta no esdevingui infinita.

Un* cas recursiu
És el cas en què el problema es resol cridant la mateixa funció amb dades més petites o simplificades.
Ha de:

- Dividir el problema: reduir el problema inicial a un conjunt més petit o més senzill
- Resoldre el cas gran a partir del petit: Usar la solució d'un subproblema més senzill per construir la solució del problema original.

**Després veurem que hi pot haver més d'un cas base i més d'un cas recursiu*

Principis dels algorismes recursius

El principi d'inducció matemàtica*:

- Permet demostrar que una propietat $P(n)$ és certa per a tots els nombres naturals $(\forall n \in \mathbb{N})$.
- Es divideix en dos passos:

Principis dels algorismes recursius

El principi d'inducció matemàtica*:

- Permet demostrar que una propietat $P(n)$ és certa per a tots els nombres naturals ($\forall n \in \mathbb{N}$).
- Es divideix en dos passos:
    - **Base d'inducció**: Demostrar que el cas inicial $P(0)$ és cert.
    - **Pas d'inducció**: Suposant que $P(n-1)$ és cert (hipòtesi d'inducció), demostrar que $P(n)$ també és cert.

Principis dels algorismes recursius

El principi d'inducció matemàtica:

- Exemple: demostrar per inducció que: $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$

Principis dels algorismes recursius

El principi d'inducció matemàtica:

- Exemple: demostrar per inducció que: $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$

Vull demostrar que:

$$P(n) = \left(\sum_{i=1}^{n} i = \frac{n(n+1)}{2} \right)$$

Principis dels algorismes recursius

El principi d'inducció matemàtica:

- Exemple: demostrar per inducció que: $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$

Vull demostrar que:

$$P(n) = \left(\sum_{i=1}^{n} i = \frac{n(n+1)}{2} \right)$$

Base d'inducció: Es compleix per al cas base 0? *

$$\sum_{i=1}^{0} i = 0, \quad \frac{0(0+1)}{2} = 0$$

$$P(0) = (0 = 0)$$

Principis dels algorismes recursius

El principi d'inducció matemàtica:

- Exemple: demostrar per inducció que: $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$

Vull demostrar que:

$$P(n) = \left(\sum_{i=1}^{n} i = \frac{n(n+1)}{2} \right)$$

Base d'inducció: Es compleix per al cas base 0? *

$$\sum_{i=1}^{0} i = 0, \quad \frac{0(0+1)}{2} = 0$$

$$P(0) = (0 = 0)$$

Hipòtesi d'inducció:

Suposo que es compleix per a (n-1)

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

Principis dels algorismes recursius

El principi d'inducció matemàtica:

- Exemple: demostrar per inducció que: $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$

Vull demostrar que:

$$P(n) = \left(\sum_{i=1}^{n} i = \frac{n(n+1)}{2} \right)$$

Base d'inducció: Es compleix per al cas base 0? *

$$\sum_{i=1}^{0} i = 0, \quad \frac{0(0+1)}{2} = 0$$

$$P(0) = (0 = 0)$$

Hipòtesi d'inducció:

Suposo que es compleix per a (n-1)

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

Pas d'inducció:

Puc demostrar que si es compleix per a $n-1$ també es compleix per a n?

Principis dels algorismes recursius

El principi d'inducció matemàtica:

- Exemple: demostrar per inducció que: $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$

Vull demostrar que:

$$P(n) = \left(\sum_{i=1}^{n} i = \frac{n(n+1)}{2} \right)$$

Base d'inducció: Es compleix per al cas base 0? *

$$\sum_{i=1}^{0} i = 0, \quad \frac{0(0+1)}{2} = 0$$

$$P(0) = (0 = 0)$$

Hipòtesi d'inducció:

Suposo que es compleix per a (n-1)

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

Pas d'inducció:

Puc demostrar que si es compleix per a $n-1$ també es compleix per a n?

$$\sum_{i=1}^{n} i = 1 + 2 + \ldots + n - 1 + n$$

Principis dels algorismes recursius

El principi d'inducció matemàtica:

- Exemple: demostrar per inducció que: $\displaystyle\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$

Vull demostrar que:

$$P(n) = \left(\sum_{i=1}^{n} i = \frac{n(n+1)}{2} \right)$$

Base d'inducció: Es compleix per al cas base 0? *

$$\sum_{i=1}^{0} i = 0, \quad \frac{0(0+1)}{2} = 0$$

$$P(0) = (0 = 0)$$

Hipòtesi d'inducció:

Suposo que es compleix per a (n-1)

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

Pas d'inducció:

Puc demostrar que si es compleix per a $n-1$ també es compleix per a n?

$$\sum_{i=1}^{n} i = \underbrace{1 + 2 + \ldots + n - 1}_{\sum_{i=1}^{n-1}} + n$$

Principis dels algorismes recursius

El principi d'inducció matemàtica:

- Exemple: demostrar per inducció que: $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$

Vull demostrar que:

$$P(n) = \left(\sum_{i=1}^{n} i = \frac{n(n+1)}{2} \right)$$

Base d'inducció: Es compleix per al cas base 0? *

$$\sum_{i=1}^{0} i = 0, \quad \frac{0(0+1)}{2} = 0$$

$$P(0) = (0 = 0)$$

Hipòtesi d'inducció:

Suposo que es compleix per a (n-1)

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

Pas d'inducció:

Puc demostrar que si es compleix per a $n - 1$ també es compleix per a n?

$$\sum_{i=1}^{n} i = \underbrace{1 + 2 + \ldots + n - 1}_{\sum_{i=1}^{n-1}} + n = \sum_{i=1}^{n-1} + n$$

Principis dels algorismes recursius

El principi d'inducció matemàtica:

- Exemple: demostrar per inducció que: $\displaystyle\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$

Vull demostrar que:

$$P(n) = \left(\sum_{i=1}^{n} i = \frac{n(n+1)}{2}\right)$$

Base d'inducció: Es compleix per al cas base 0? *

$$\sum_{i=1}^{0} i = 0, \quad \frac{0(0+1)}{2} = 0$$

$$P(0) = (0 = 0)$$

Hipòtesi d'inducció:

Suposo que es compleix per a (n-1)

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

Pas d'inducció:

Puc demostrar que si es compleix per a $n-1$ també es compleix per a n?

$$\sum_{i=1}^{n} i = \underbrace{1 + 2 + \ldots + n - 1}_{\sum_{i=1}^{n-1}} + n = \sum_{i=1}^{n-1} + n =$$

Aplicant hipòtesi d'inducció

Principis dels algorismes recursius

El principi d'inducció matemàtica:

- Exemple: demostrar per inducció que: $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$

Vull demostrar que:

$$P(n) = \left(\sum_{i=1}^{n} i = \frac{n(n+1)}{2} \right)$$

Base d'inducció: Es compleix per al cas base 0? *

$$\sum_{i=1}^{0} i = 0, \quad \frac{0(0+1)}{2} = 0$$

$$P(0) = (0 = 0)$$

Hipòtesi d'inducció:

Suposo que es compleix per a (n-1)

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

Pas d'inducció:

Puc demostrar que si es compleix per a $n-1$ també es compleix per a n?

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

Aplicant hipòtesi d'inducció

Principis dels algorismes recursius

El principi d'inducció matemàtica:

- Exemple: demostrar per inducció que: $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$

Vull demostrar que:

$$P(n) = \left(\sum_{i=1}^{n} i = \frac{n(n+1)}{2} \right)$$

Base d'inducció: Es compleix per al cas base 0? *

$$\sum_{i=1}^{0} i = 0, \quad \frac{0(0+1)}{2} = 0$$

$$P(0) = (0 = 0)$$

Hipòtesi d'inducció:

Suposo que es compleix per a (n-1)

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

Pas d'inducció:

Puc demostrar que si es compleix per a $n - 1$ també es compleix per a n?

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

Aplicant hipòtesi d'inducció

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

Principis dels algorismes recursius

** El cas base no té per què ser sempre el zero, serà el primer membre del conjunt pel qual volem demostrar que la propietat es compleix*

El principi d'inducció matemàtica:

- Exemple: demostrar per inducció que: $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$

Vull demostrar que:

$$P(n) = \left(\sum_{i=1}^{n} i = \frac{n(n+1)}{2} \right)$$

Base d'inducció: Es compleix per al cas base 0? *

$$\sum_{i=1}^{0} i = 0, \quad \frac{0(0+1)}{2} = 0$$

$$P(0) = (0 = 0)$$

Hipòtesi d'inducció:

Suposo que es compleix per a (n-1)

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

Pas d'inducció:

Puc demostrar que si es compleix per a $n-1$ també es compleix per a n?

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

Aplicant hipòtesi d'inducció

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

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

Principis dels algorismes recursius

El principi d'inducció matemàtica*:

Resumint, quan usem el Principi d'Inducció necessitem:

(I) Un cas base o trivial de comprovar

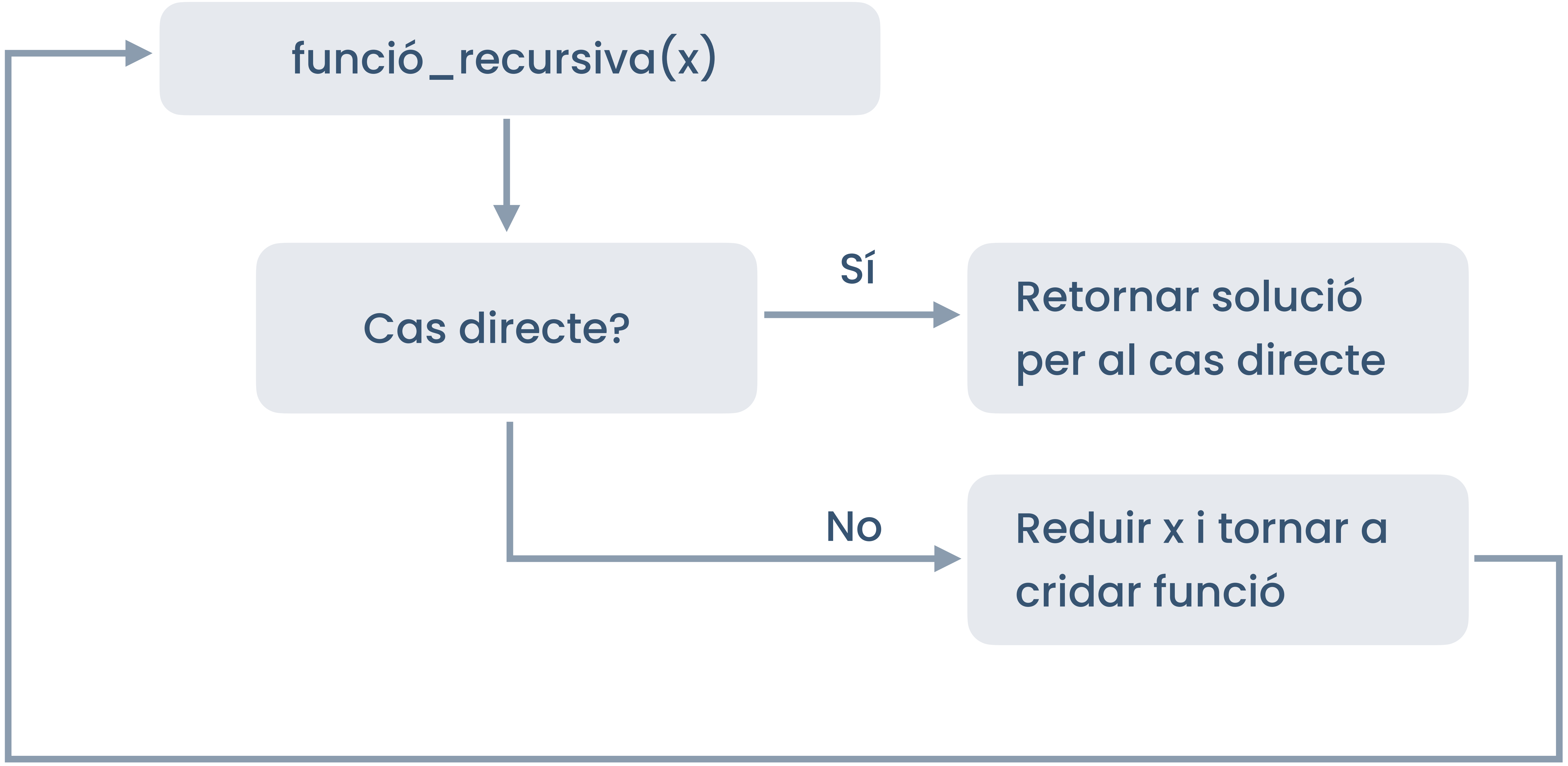
(II) Demostrar que si la propietat és certa per a un cas concret, també ho serà per al següent.

Això guarda molta similitud amb la programació de funcions recursives

Programar amb recursivitat

Funcions recursives

- Una *funció recursiva* és una funció que es crida a si mateixa.



Programar amb recursivitat

Exemple: factorial

- La funció factorial, definida com:

$$n! = \begin{cases} 1 & \text{si } n = 0 \\ n \cdot (n-1)! & \text{si } n > 0 \end{cases}$$

Ens suggereix una implementació recursiva:

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

Programar amb recursivitat

Exemple: factorial

- La funció factorial, definida com:

$$n! = \begin{cases} 1 & \text{si } n = 0 \\ n \cdot (n-1)! & \text{si } n > 0 \end{cases}$$

Ens suggereix una implementació recursiva:

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

Arbre de crides:

- Per al cas, e.g. factorial(4):

Programar amb recursivitat

Exemple: factorial

- La funció factorial, definida com:

$$n! = \begin{cases} 1 & \text{si } n = 0 \\ n \cdot (n-1)! & \text{si } n > 0 \end{cases}$$

Ens suggereix una implementació recursiva:

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

Arbre de crides:

- Per al cas, e.g. factorial(4):

```
factorial(4)
```

Programar amb recursivitat

Exemple: factorial

- La funció factorial, definida com:

$$n! = \begin{cases} 1 & \text{si } n = 0 \\ n \cdot (n-1)! & \text{si } n > 0 \end{cases}$$

Ens suggereix una implementació recursiva:

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

Arbre de crides:

- Per al cas, e.g. factorial(4):

```
factorial(4)
     |
     v
 4 * factorial(3)
```

Programar amb recursivitat

Exemple: factorial

- La funció factorial, definida com:

$$n! = \begin{cases} 1 & \text{si } n = 0 \\ n \cdot (n-1)! & \text{si } n > 0 \end{cases}$$

Ens suggereix una implementació recursiva:

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

Arbre de crides:

- Per al cas, e.g. factorial(4):

```
factorial(4)

4 * factorial(3)

3 * factorial(2)
```

Programar amb recursivitat

Exemple: factorial

- La funció factorial, definida com:

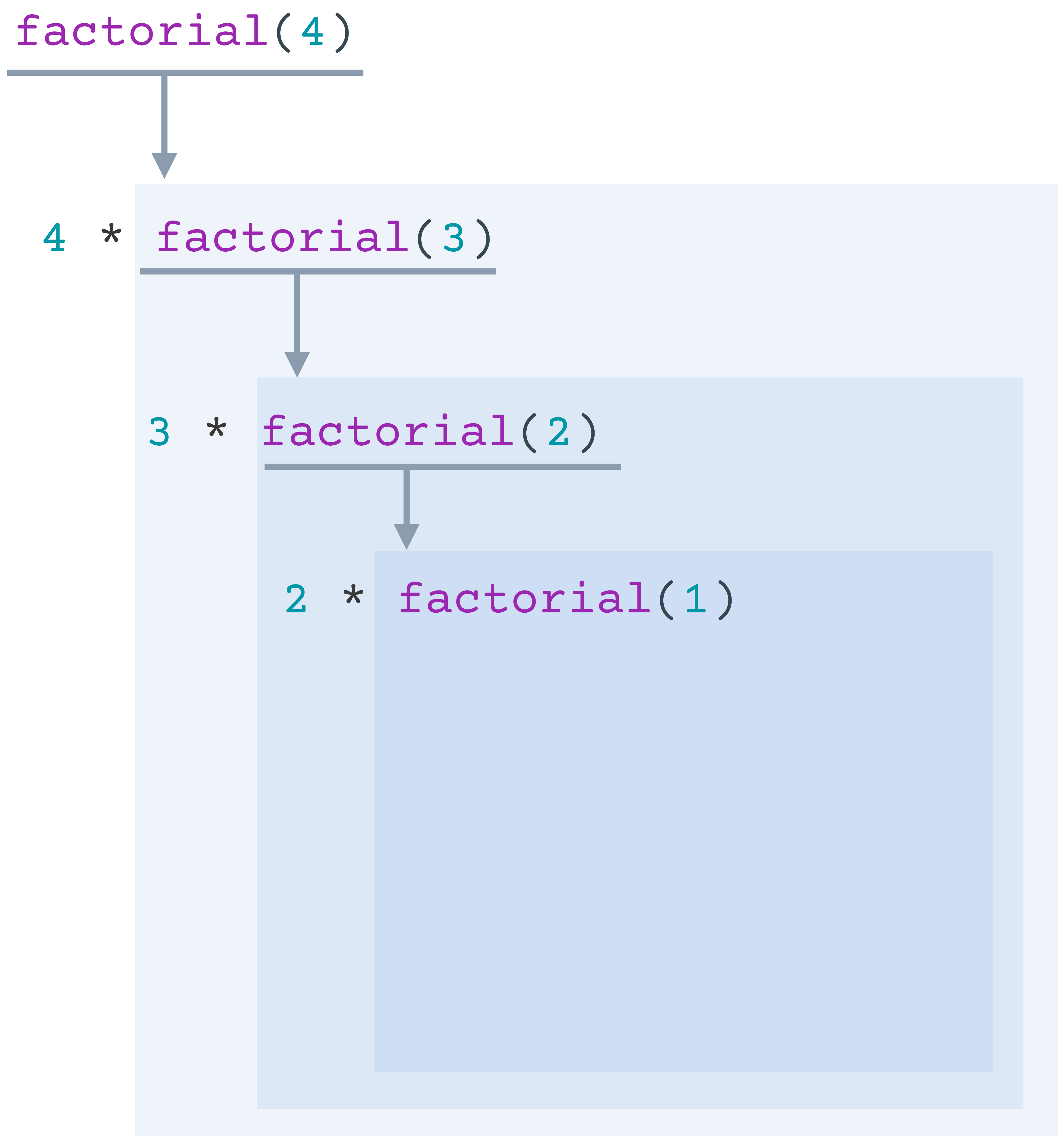
$$n! = \begin{cases} 1 & \text{si } n = 0 \\ n \cdot (n-1)! & \text{si } n > 0 \end{cases}$$

Ens suggereix una implementació recursiva:

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

Arbre de crides:

- Per al cas, e.g. factorial(4):



Programar amb recursivitat

Exemple: factorial

- La funció factorial, definida com:

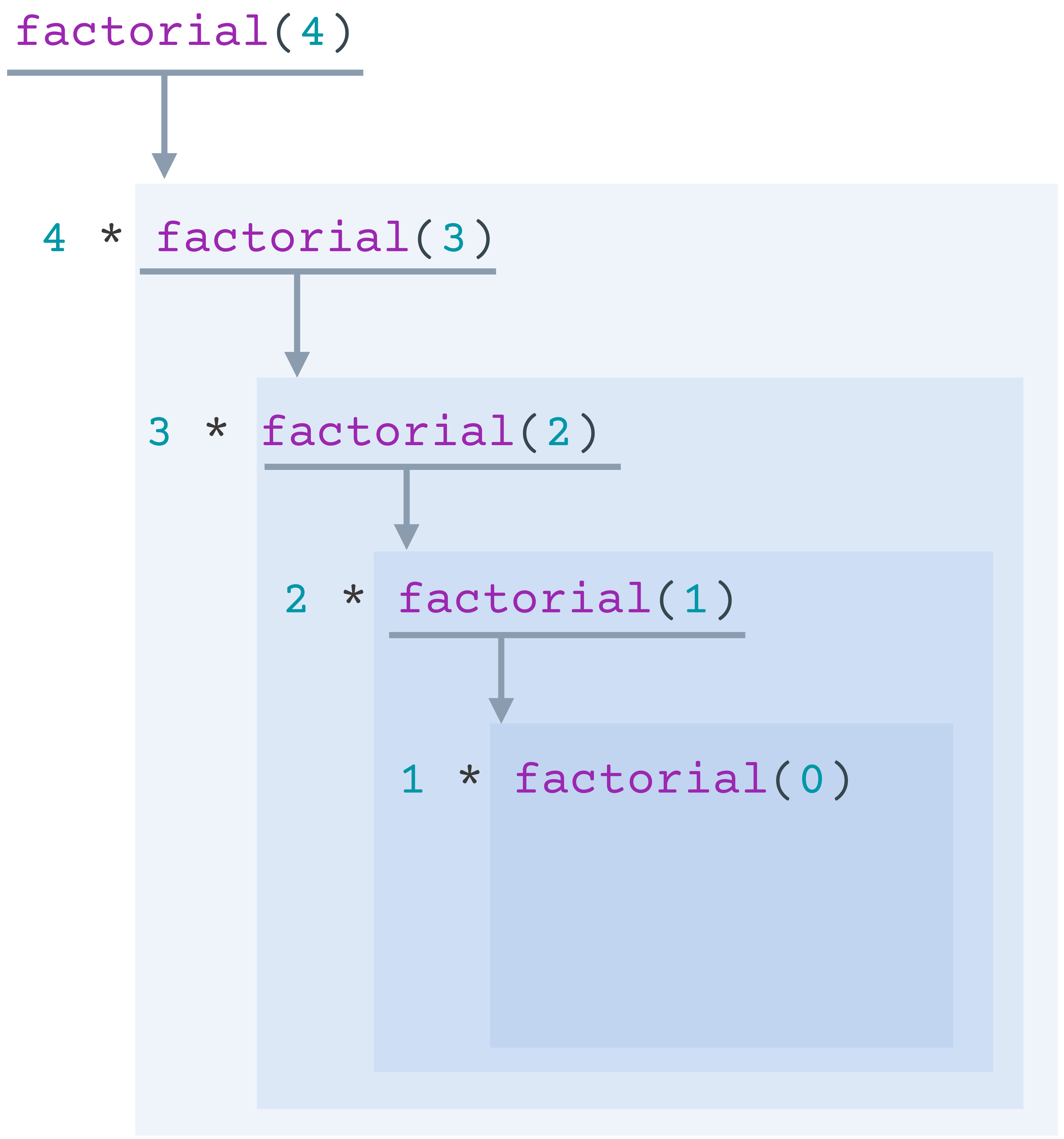
$$n! = \begin{cases} 1 & \text{si } n = 0 \\ n \cdot (n-1)! & \text{si } n > 0 \end{cases}$$

Ens suggereix una implementació recursiva:

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

Arbre de crides:

- Per al cas, e.g. factorial(4):



Programar amb recursivitat

Exemple: factorial

- La funció factorial, definida com:

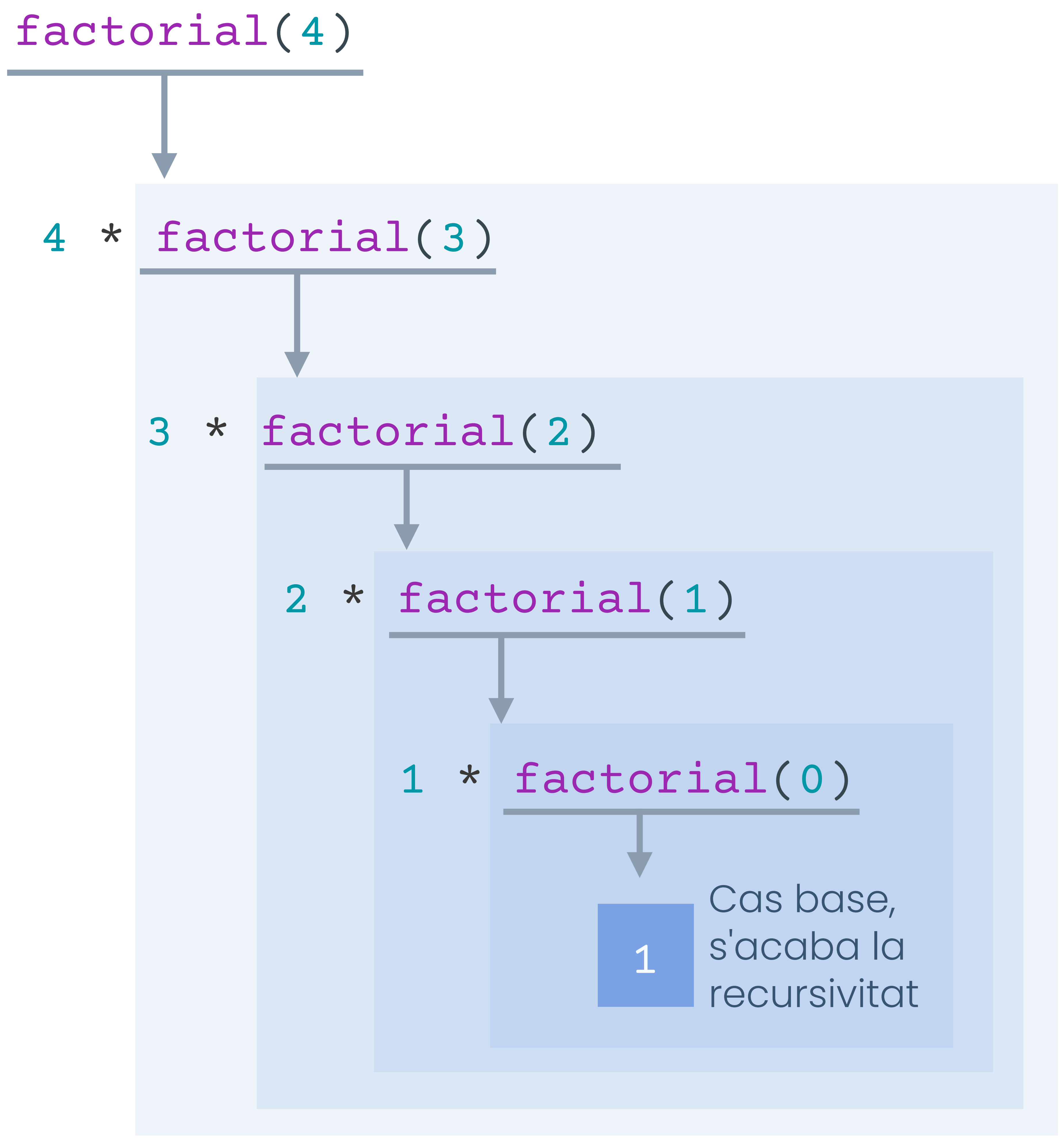
$$n! = \begin{cases} 1 & \text{si } n = 0 \\ n \cdot (n-1)! & \text{si } n > 0 \end{cases}$$

Ens suggereix una implementació recursiva:

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

Arbre de crides:

- Per al cas, e.g. factorial(4):



Programar amb recursivitat

Finalització de la seqüència de crides recursives

- Igual que el principi d'inducció, per escriure un algorisme recursiu correcte és necessari que hi hagi com a mínim un cas base en el que no hi hagi crida recursiva. Si no, faríem infinites crides recursives!

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

Programar amb recursivitat

Finalització de la seqüència de crides recursives

- Igual que el principi d'inducció, per escriure un algorisme recursiu correcte és necessari que hi hagi com a mínim un cas base en el que no hi hagi crida recursiva. Si no, faríem infinites crides recursives!
- A més, la branca recursiva s'ha d'assegurar que sempre arribarem al cas base.

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

Programar amb recursivitat

Finalització de la seqüència de crides recursives

- Igual que el principi d'inducció, per escriure un algorisme recursiu correcte és necessari que hi hagi com a mínim un cas base en el que no hi hagi crida recursiva. Si no, faríem infinites crides recursives!
- A més, la branca recursiva s'ha d'assegurar que sempre arribarem al cas base.

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

Exemple

Implementació incorrecta del factorial.

Aquest codi té un cas base correcte però la branca recursiva no ens permet mai arribar al cas base.

Aquest programa provoca crides infinites a la funció factorial:

e.g. factorial(4) = 4 * factorial(5) = 4 * 5 * factorial(6) = ...

```
int factorial(int n)
{
    int valor;
    if (n == 0){
       return 1;
    }
    else {
       return n * factorial(n+1);
    }
}
```

Programar amb recursivitat

Com pensar recursivament

Estem acostumats a resoldre la majoria de problemes de manera iterativa. Com passar a "mentalitat recursiva"?

Programar amb recursivitat

Com pensar recursivament

Estem acostumats a resoldre la majoria de problemes de manera iterativa. Com passar a "mentalitat recursiva"?

1. **Identifica un patró repetitiu al problema**

Programar amb recursivitat

Com pensar recursivament

Estem acostumats a resoldre la majoria de problemes de manera iterativa. Com passar a "mentalitat recursiva"?

1. Identifica un patró repetitiu al problema

En el cas del factorial, te n'has d'adonar que $4! = 4 \cdot 3!$, i que $3! = 3 \cdot 2!$, ...

Programar amb recursivitat

Com pensar recursivament

Estem acostumats a resoldre la majoria de problemes de manera iterativa. Com passar a "mentalitat recursiva"?

1. Identifica un patró repetitiu al problema

En el cas del factorial, te n'has d'adonar que $4! = 4 \cdot 3!$, i que $3! = 3 \cdot 2!$, ...

2. Defineix el cas en el qual aturem el procés (cas base)

Programar amb recursivitat

Com pensar recursivament

Estem acostumats a resoldre la majoria de problemes de manera iterativa. Com passar a "mentalitat recursiva"?

1. Identifica un patró repetitiu al problema

En el cas del factorial, te n'has d'adonar que $4! = 4 \cdot 3!$, i que $3! = 3 \cdot 2!$, ...

2. Defineix el cas en el qual aturem el procés (cas base)

Seguint la seqüència anterior, veiem que hi ha un moment en què ja no podem seguir amb la seqüència repetitiva, i ens n'adonem que és quan volem calcular el factorial de 0.

Programar amb recursivitat

Com pensar recursivament

Estem acostumats a resoldre la majoria de problemes de manera iterativa. Com passar a "mentalitat recursiva"?

1. Identifica un patró repetitiu al problema

En el cas del factorial, te n'has d'adonar que $4! = 4 \cdot 3!$, i que $3! = 3 \cdot 2!$, ...

2. Defineix el cas en el qual aturem el procés (cas base)

Seguint la seqüència anterior, veiem que hi ha un moment en què ja no podem seguir amb la seqüència repetitiva, i ens n'adonem que és quan volem calcular el factorial de 0.

3. Redueix el problema: pensa en com pots convertir el problema gran en un subproblema més petit

Programar amb recursivitat

Com pensar recursivament

Estem acostumats a resoldre la majoria de problemes de manera iterativa. Com passar a "mentalitat recursiva"?

1. Identifica un patró repetitiu al problema

En el cas del factorial, te n'has d'adonar que $4! = 4 \cdot 3!$, i que $3! = 3 \cdot 2!$, ...

2. Defineix el cas en el qual aturem el procés (cas base)

Seguint la seqüència anterior, veiem que hi ha un moment en què ja no podem seguir amb la seqüència repetitiva, i ens n'adonem que és quan volem calcular el factorial de 0.

3. Redueix el problema: pensa en com pots convertir el problema gran en un subproblema més petit

Al pas 1 ja hem vist que podem expressar el factorial de 4 com 4 multiplicat pel factorial d'un nombre més petit. Així és com reduïm el problema

Programar amb recursivitat

Com pensar recursivament

Estem acostumats a resoldre la majoria de problemes de manera iterativa. Com passar a "mentalitat recursiva"?

1. Identifica un patró repetitiu al problema

En el cas del factorial, te n'has d'adonar que $4! = 4 \cdot 3!$, i que $3! = 3 \cdot 2!$, ...

2. Defineix el cas en el qual aturem el procés (cas base)

Seguint la seqüència anterior, veiem que hi ha un moment en què ja no podem seguir amb la seqüència repetitiva, i ens n'adonem que és quan volem calcular el factorial de 0.

3. Redueix el problema: pensa en com pots convertir el problema gran en un subproblema més petit

Al pas 1 ja hem vist que podem expressar el factorial de 4 com 4 multiplicat pel factorial d'un nombre més petit. Així és com reduïm el problema

4. Confia en la recursió! Quan dissenyes una funció recursiva, has d'assumir que la funció funciona correctament per als subproblemes més petits (com a la hipòtesi d'inducció!)

Programar amb recursivitat

Exemple II: Divisió entera restant

- Seguint els passos anteriors, implementar la divisió entera del nombre ***D*** pel nombre ***d***, de manera recursiva, fent servir restes.

Programar amb recursivitat

Exemple II: Divisió entera restant

- Seguint els passos anteriors, implementar la divisió entera del nombre ***D*** pel nombre ***d***, de manera recursiva, fent servir restes.

1. Identifica un patró repetitiu al problema

Imagina que volem dividir 13 entre 4. (Resultat = 3)

Programar amb recursivitat

Exemple II: Divisió entera restant

- Seguint els passos anteriors, implementar la divisió entera del nombre ***D*** pel nombre ***d***, de manera recursiva, fent servir restes.

1. Identifica un patró repetitiu al problema

Imagina que volem dividir 13 entre 4. (Resultat = 3)

Primera resta: $13 - 4 = 9$

Programar amb recursivitat

Exemple II: Divisió entera restant

- Seguint els passos anteriors, implementar la divisió entera del nombre ***D*** pel nombre ***d***, de manera recursiva, fent servir restes.

1. Identifica un patró repetitiu al problema

Imagina que volem dividir 13 entre 4. (Resultat = 3)

Primera resta: $13 - 4 = 9$

Segona resta: $9 - 4 = 5$

Programar amb recursivitat

Exemple II: Divisió entera restant

- Seguint els passos anteriors, implementar la divisió entera del nombre ***D*** pel nombre ***d***, de manera recursiva, fent servir restes.

1. Identifica un patró repetitiu al problema

Imagina que volem dividir 13 entre 4. (Resultat = 3)

Primera resta: $13 - 4 = 9$

Segona resta: $9 - 4 = 5$

Tercera resta: $5 - 4 = 1$

Programar amb recursivitat

Exemple II: Divisió entera restant

- Seguint els passos anteriors, implementar la divisió entera del nombre ***D*** pel nombre ***d***, de manera recursiva, fent servir restes.

1. Identifica un patró repetitiu al problema

Imagina que volem dividir 13 entre 4. (Resultat = 3)

Primera resta: $13 - 4 = 9$

Segona resta: $9 - 4 = 5$

Tercera resta: $5 - 4 = 1$ Aquí me n'adono que no puc continuar, ja que el resultat és més petit que 4

Programar amb recursivitat

Exemple II: Divisió entera restant

- Seguint els passos anteriors, implementar la divisió entera del nombre ***D*** pel nombre ***d***, de manera recursiva, fent servir restes.

1. Identifica un patró repetitiu al problema

Imagina que volem dividir 13 entre 4. (Resultat = 3)

Primera resta: $13 - 4 = 9$

Segona resta: $9 - 4 = 5$

Tercera resta: $5 - 4 = 1$ Aquí me n'adono que no puc continuar, ja que el resultat és més petit que 4

2. Defineix el cas en el qual aturem el procés (cas base)

Programar amb recursivitat

Exemple II: Divisió entera restant

- Seguint els passos anteriors, implementar la divisió entera del nombre ***D*** pel nombre ***d***, de manera recursiva, fent servir restes.

1. Identifica un patró repetitiu al problema

Imagina que volem dividir 13 entre 4. (Resultat = 3)

Primera resta: $13 - 4 = 9$

Segona resta: $9 - 4 = 5$

Tercera resta: $5 - 4 = 1$ Aquí me n'adono que no puc continuar, ja que el resultat és més petit que 4

2. Defineix el cas en el qual aturem el procés (cas base)

Me n'acabo d'adonar que no puc continuar quan $D < d$.
Aquest serà el meu cas base.

Programar amb recursivitat

Exemple II: Divisió entera restant

- Seguint els passos anteriors, implementar la divisió entera del nombre ***D*** pel nombre ***d***, de manera recursiva, fent servir restes.

1. Identifica un patró repetitiu al problema

Imagina que volem dividir 13 entre 4. (Resultat = 3)

Primera resta: $13 - 4 = 9$

Segona resta: $9 - 4 = 5$

Tercera resta: $5 - 4 = 1$ Aquí me n'adono que no puc continuar, ja que el resultat és més petit que 4

2. Defineix el cas en el qual aturem el procés (cas base)

Me n'acabo d'adonar que no puc continuar quan $D < d$.
Aquest serà el meu cas base.

3. Redueix el problema

Programar amb recursivitat

Exemple II: Divisió entera restant

- Seguint els passos anteriors, implementar la divisió entera del nombre ***D*** pel nombre ***d***, de manera recursiva, fent servir restes.

1. Identifica un patró repetitiu al problema

Imagina que volem dividir 13 entre 4. (Resultat = 3)

Primera resta: $13 - 4 = 9$

Segona resta: $9 - 4 = 5$

Tercera resta: $5 - 4 = 1$ Aquí me n'adono que no puc continuar, ja que el resultat és més petit que 4

2. Defineix el cas en el qual aturem el procés (cas base)

Me n'acabo d'adonar que no puc continuar quan $D < d$.
Aquest serà el meu cas base.

3. Redueix el problema

A través del procés anterior, me n'adono que cada cop parteixo del nombre anterior menys 4. És a dir, de $D - d$.

Programar amb recursivitat

Exemple II: Divisió entera restant

- Seguint els passos anteriors, implementar la divisió entera del nombre ***D*** pel nombre ***d***, de manera recursiva, fent servir restes.

1. Identifica un patró repetitiu al problema

Imagina que volem dividir 13 entre 4. (Resultat = 3)

Primera resta: $13 - 4 = 9$

Segona resta: $9 - 4 = 5$

Tercera resta: $5 - 4 = 1$ Aquí me n'adono que no puc continuar, ja que el resultat és més petit que 4

2. Defineix el cas en el qual aturem el procés (cas base)

Me n'acabo d'adonar que no puc continuar quan $D < d$.
Aquest serà el meu cas base.

3. Redueix el problema

A través del procés anterior, me n'adono que cada cop parteixo del nombre anterior menys 4. És a dir, de $D - d$.

Per tant, la crida recursiva seria $dividir(D - d, d)$.

Programar amb recursivitat

Exemple II: Divisió entera restant

- Seguint els passos anteriors, implementar la divisió entera del nombre ***D*** pel nombre ***d***, de manera recursiva, fent servir restes.

1. Identifica un patró repetitiu al problema

Imagina que volem dividir 13 entre 4. (Resultat = 3)

Primera resta: $13 - 4 = 9$

Segona resta: $9 - 4 = 5$

Tercera resta: $5 - 4 = 1$ Aquí me n'adono que no puc continuar, ja que el resultat és més petit que 4

$\text{Quocient} = 3$

2. Defineix el cas en el qual aturem el procés (cas base)

Me n'acabo d'adonar que no puc continuar quan $D < d$.

Aquest serà el meu cas base.

3. Redueix el problema

A través del procés anterior, me n'adono que cada cop parteixo del nombre anterior menys 4. És a dir, de $D - d$.

Per tant, la crida recursiva seria $dividir(D - d, d)$.

A més, me n'adono que el resultat que busco (el quocient) en resulta de calcular el nombre de restes que faig.

Programar amb recursivitat

Exemple II: Divisió entera restant

- Seguint els passos anteriors, implementar la divisió entera del nombre ***D*** pel nombre ***d***, de manera recursiva, fent servir restes.

1. Identifica un patró repetitiu al problema

Imagina que volem dividir 13 entre 4. (Resultat = 3)

Primera resta: $13 - 4 = 9$

Segona resta: $9 - 4 = 5$

Tercera resta: $5 - 4 = 1$ Aquí me n'adono que no puc continuar, ja que el resultat és més petit que 4

$\boxed{\text{Quocient} = 3}$

2. Defineix el cas en el qual aturem el procés (cas base)

Me n'acabo d'adonar que no puc continuar quan $D < d$.
Aquest serà el meu cas base.

3. Redueix el problema

A través del procés anterior, me n'adono que cada cop parteixo del nombre anterior menys 4. És a dir, de $D - d$.

Per tant, la crida recursiva seria $dividir(D - d, d)$.

A més, me n'adono que el resultat que busco (el quocient) en resulta de calcular el nombre de restes que faig.

Aquest quocient, en una versió iterativa simplement seria un comptador que aniríem augmentant.

```
int dividir_iteratiu(int D, int d) {
    int quocient = 0;
    while (D >= d) {
        D = D - d;
        quocient++;
    }
    return quocient;
}
```

Programar amb recursivitat

Exemple II: Divisió entera restant

- Seguint els passos anteriors, implementar la divisió entera del nombre ***D*** pel nombre ***d***, de manera recursiva, fent servir restes.

1. Identifica un patró repetitiu al problema

Imagina que volem dividir 13 entre 4. (Resultat = 3)

Primera resta: $13 - 4 = 9$

Segona resta: $9 - 4 = 5$

Tercera resta: $5 - 4 = 1$ Aquí me n'adono que no puc continuar, ja que el resultat és més petit que 4

Quocient $= 3$

2. Defineix el cas en el qual aturem el procés (cas base)

Me n'acabo d'adonar que no puc continuar quan $D < d$.

Aquest serà el meu cas base.

3. Redueix el problema

A través del procés anterior, me n'adono que cada cop parteixo del nombre anterior menys 4. És a dir, de $D - d$.

Per tant, la crida recursiva seria $dividir(D - d, d)$.

A més, me n'adono que el resultat que busco (el quocient) en resulta de calcular el nombre de restes que faig.

Aquest quocient, en una versió iterativa simplement seria un comptador que aniríem augmentant.

Aquí el que hem de fer és fer que cada crida retorni un 1 (el nombre de restes que ha fet) i que aquest es vagi acumulant a cada crida. $1 + dividir(D - d, d)$

Programar amb recursivitat

Exemple II: Divisió entera restant

- Seguint els passos anteriors, implementar la divisió entera del nombre ***D*** pel nombre ***d***, de manera recursiva, fent servir restes.

1. Identifica un patró repetitiu al problema

Imagina que volem dividir 13 entre 4. (Resultat = 3)

Primera resta: $13 - 4 = 9$

Segona resta: $9 - 4 = 5$

Tercera resta: $5 - 4 = 1$ Aquí me n'adono que no puc continuar, ja que el resultat és més petit que 4

Quocient $= 3$

2. Defineix el cas en el qual aturem el procés (cas base)

Me n'acabo d'adonar que no puc continuar quan $D < d$.
Aquest serà el meu cas base.

3. Redueix el problema

A través del procés anterior, me n'adono que cada cop parteixo del nombre anterior menys 4. És a dir, de $D - d$.

Per tant, la crida recursiva seria $dividir(D - d, d)$.

A més, me n'adono que el resultat que busco (el quocient) en resulta de calcular el nombre de restes que faig.

Aquest quocient, en una versió iterativa simplement seria un comptador que aniríem augmentant.

Aquí el que hem de fer és fer que cada crida retorni un 1 (el nombre de restes que ha fet) i que aquest es vagi acumulant a cada crida. $1 + dividir(D - d, d)$

Per tant, la definició recursiva seria:

$$\text{dividir}(D, d) = \begin{cases} 0 & \text{si } D < d, \\ 1 + \text{dividir}(D - d, d) & \text{si } D \geq d\,. \end{cases}$$

Programar amb recursivitat

Exemple II: Divisió entera restant

3. Redueix el problema

A través del procés anterior, me n'adono que cada cop parteixo del nombre anterior menys 4. És a dir, de $D - d$.

Per tant, la crida recursiva seria $dividir(D - d, d)$.

A més, me n'adono que el resultat que busco (el quocient) en resulta de calcular el nombre de restes que faig.

Aquest quocient, en una versió iterativa simplement seria un comptador que aniríem augmentant.

Aquí el que hem de fer és fer que cada crida retorni un 1 (el nombre de restes que ha fet) i que aquest es vagi acumulant a cada crida. $1 + dividir(D - d, d)$

Per tant, la definició recursiva seria:

$$\text{dividir}(D, d) = \begin{cases} 0 & \text{si } D < d, \\ 1 + \text{dividir}(D - d, d) & \text{si } D \geq d\,. \end{cases}$$

4. Confia en la recursivitat!

Programar amb recursivitat

Exemple II: Divisió entera restant

3. Redueix el problema

A través del procés anterior, me n'adono que cada cop parteixo del nombre anterior menys 4. És a dir, de $D - d$.

Per tant, la crida recursiva seria $dividir(D - d, d)$.

A més, me n'adono que el resultat que busco (el quocient) en resulta de calcular el nombre de restes que faig.

Aquest quocient, en una versió iterativa simplement seria un comptador que aniríem augmentant.

Aquí el que hem de fer és fer que cada crida retorni un 1 (el nombre de restes que ha fet) i que aquest es vagi acumulant a cada crida. $1 + dividir(D - d, d)$

Per tant, la definició recursiva seria:

$$\text{dividir}(D, d) = \begin{cases} 0 & \text{si } D < d, \\ 1 + \text{dividir}(D - d, d) & \text{si } D \geq d\,. \end{cases}$$

4. Confia en la recursivitat!

```c
int dividir(int D, int d) {
    // Cas especial que afegim per
    // evitar dividir per zero
    if (d == 0) {
        printf("Error: Divisió per zero");
        return -1;
    }
    if (D < d){
        return 0;
    }
    else{
        return 1 + dividir(D - d, d);
    }
}
```

Programar amb recursivitat

Més exemples

- Calcular el màxim comú divisor amb l'algorisme d'Euclides del residu.

Programar amb recursivitat

Més exemples

- Calcular el màxim comú divisor amb l'algorisme d'Euclides del residu.

Idea bàsica

Si tenim dos nombres a i b on $a \geq b$, el $mcd(a, b)$ equival a $mcd(b, a \mod b)$

Procediment

1. Calcula el residu $(a \mod b)$
2. Substitueix a pel valor de b i b pel residu

   $a \leftarrow b$

   $b \leftarrow a \mod b$
3. Repeteix aquest procés fins que $b = 0$
4. Quan $b = 0$, el valor de a és el mcd.

Programar amb recursivitat

Més exemples

- Calcular el màxim comú divisor amb l'algorisme d'Euclides del residu.

Idea bàsica

Si tenim dos nombres a i b on $a \geq b$, el $mcd(a, b)$ equival a $mcd(b, a \mod b)$

Procediment

1. Calcula el residu $(a \mod b)$
2. Substitueix a pel valor de b i b pel residu

   $a \leftarrow b$

   $b \leftarrow a \mod b$
3. Repeteix aquest procés fins que $b = 0$
4. Quan $b = 0$, el valor de a és el mcd.

Formalització matemàtica

$$\text{mcd}(a, b) = \begin{cases} a & \text{si } b = 0, \\ \text{mcd}(b, a \mod b) & \text{si } b \neq 0. \end{cases}$$

Programar amb recursivitat

Més exemples

- Calcular el màxim comú divisor amb l'algorisme d'Euclides del residu.

Idea bàsica

Si tenim dos nombres a i b on $a \geq b$, el $mcd(a, b)$ equival a $mcd(b, a \mod b)$

Procediment

1. Calcula el residu $(a \mod b)$
2. Substitueix a pel valor de b i b pel residu

   $a \leftarrow b$

   $b \leftarrow a \mod b$
3. Repeteix aquest procés fins que $b = 0$
4. Quan $b = 0$, el valor de a és el mcd.

Formalització matemàtica

$$\text{mcd}(a, b) = \begin{cases} a & \text{si } b = 0, \\ \text{mcd}(b, a \mod b) & \text{si } b \neq 0. \end{cases}$$

Algorisme recursiu

```
int mcd (int a, int b){
  if (b == 0){
     return a;
  }
  else {
     return mcd (b, a%b);
  }
}
```

Programar amb recursivitat

Més exemples

- Calcular el màxim comú divisor amb l'algorisme d'Euclides del residu.

Idea bàsica

Si tenim dos nombres a i b on $a \geq b$, el $mcd(a, b)$ equival a $mcd(b, a \mod b)$

Procediment

1. Calcula el residu $(a \mod b)$
2. Substitueix a pel valor de b i b pel residu

   $a \leftarrow b$

   $b \leftarrow a \mod b$
3. Repeteix aquest procés fins que $b = 0$
4. Quan $b = 0$, el valor de a és el mcd.

Formalització matemàtica

$$\text{mcd}(a, b) = \begin{cases} a & \text{si } b = 0, \\ \text{mcd}(b, a \mod b) & \text{si } b \neq 0. \end{cases}$$

Algorisme recursiu

```
int mcd (int a, int b){
  if (b == 0){
     return a;
  }
  else {
     return mcd (b, a%b);
  }
}
```

Comproveu que funciona per al cas $mcd(12,8)$ *(que dóna 4)*

Programar amb recursivitat

Què fa aquest codi?

```c
void test (int n){
    if (n > 0){
        printf("%d", n);
        test(n-1);
    }
}
```

```c
#include <stdio.h>

int main()
{
    test(4);
    return 0;
}
```

Què imprimeix el codi?

Quantes crides es fan?

El cost de la recursivitat: la pila de crides

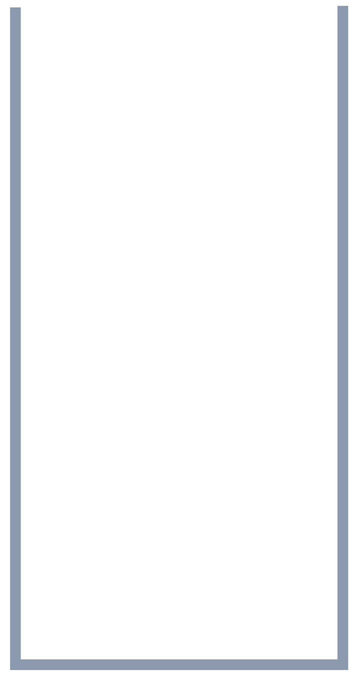
Què és una pila?

- Una pila és una estructura de dades on els elements s'hi afegeixen i s'hi eliminen per un únic extrem, de manera que només podem accedir a l'últim element que s'hi ha afegit.

El cost de la recursivitat: la pila de crides

Què és una pila?

- Una pila és una estructura de dades on els elements s'hi afegeixen i s'hi eliminen per un únic extrem, de manera que només podem accedir a l'últim element que s'hi ha afegit.

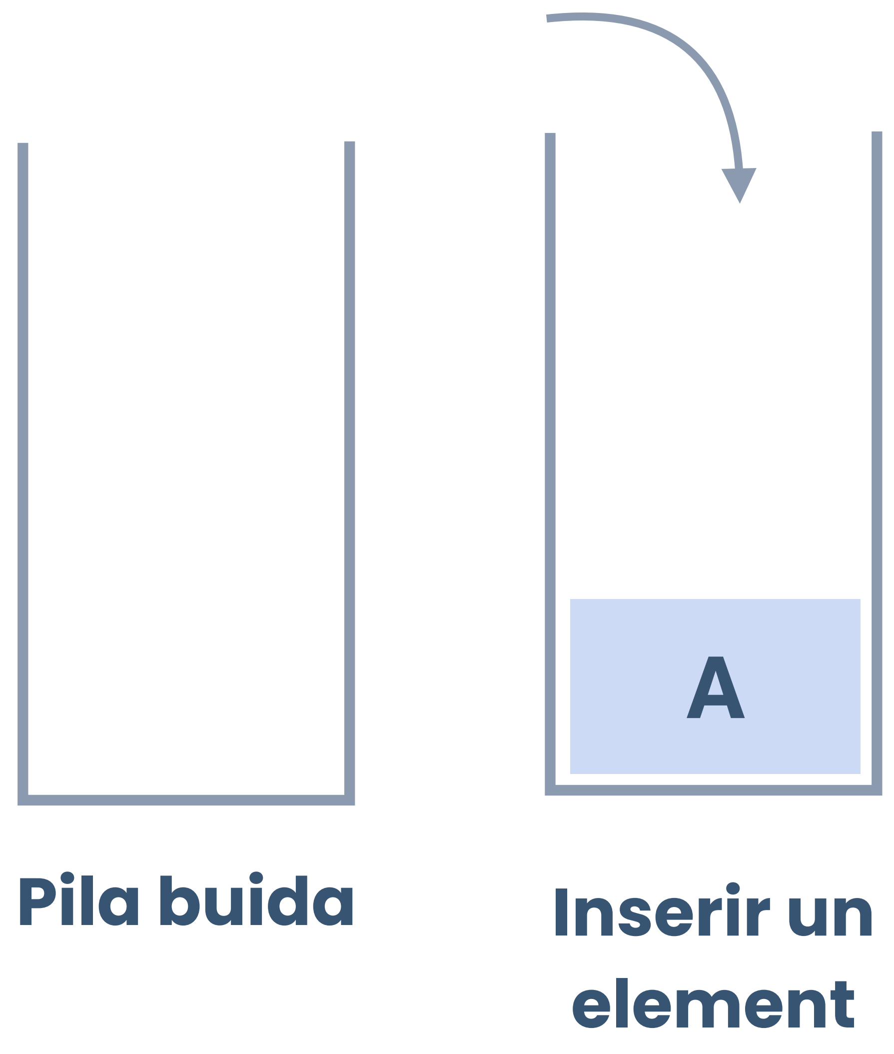


Pila buida

El cost de la recursivitat: la pila de crides

Què és una pila?

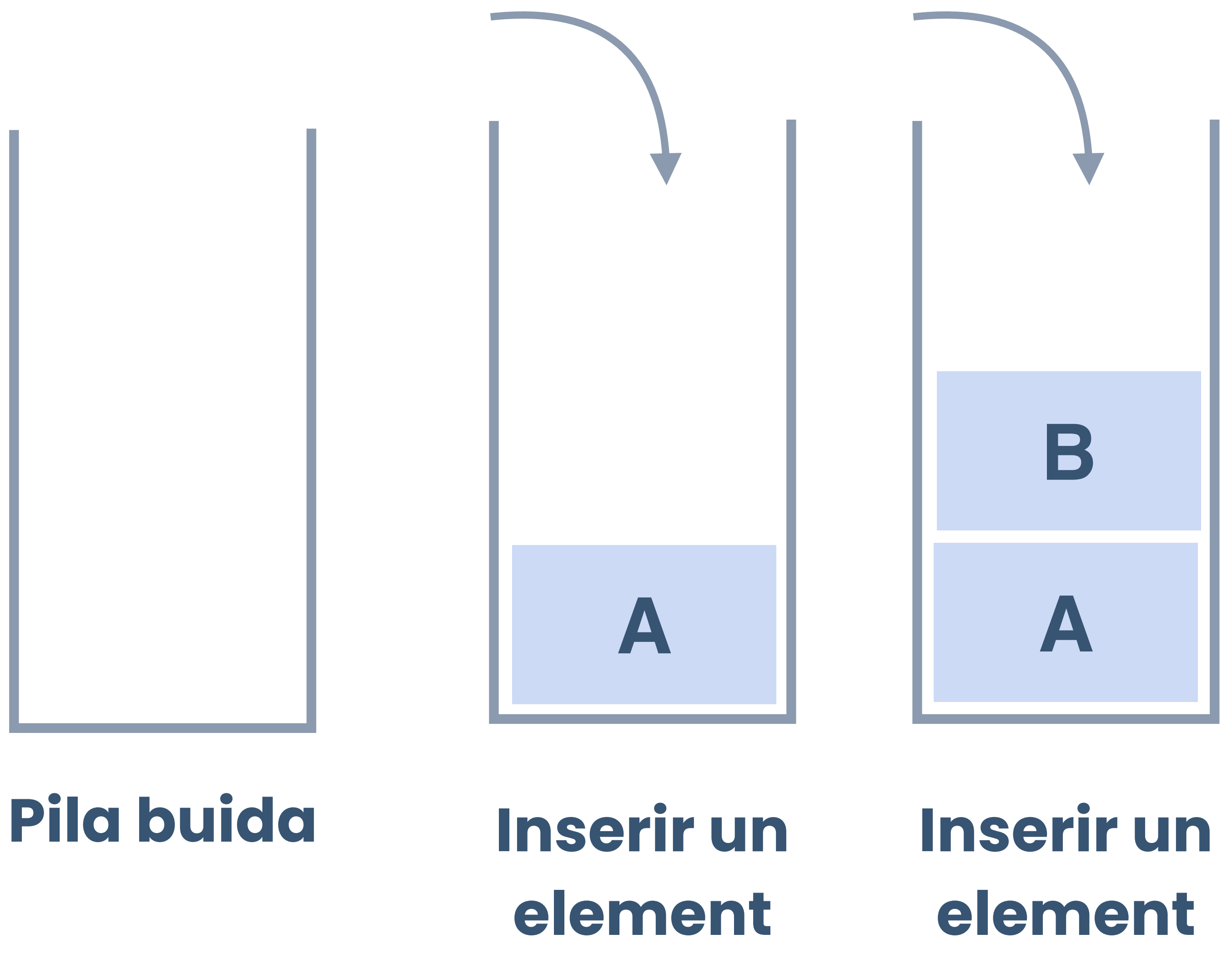
- Una pila és una estructura de dades on els elements s'hi afegeixen i s'hi eliminen per un únic extrem, de manera que només podem accedir a l'últim element que s'hi ha afegit.



El cost de la recursivitat: la pila de crides

Què és una pila?

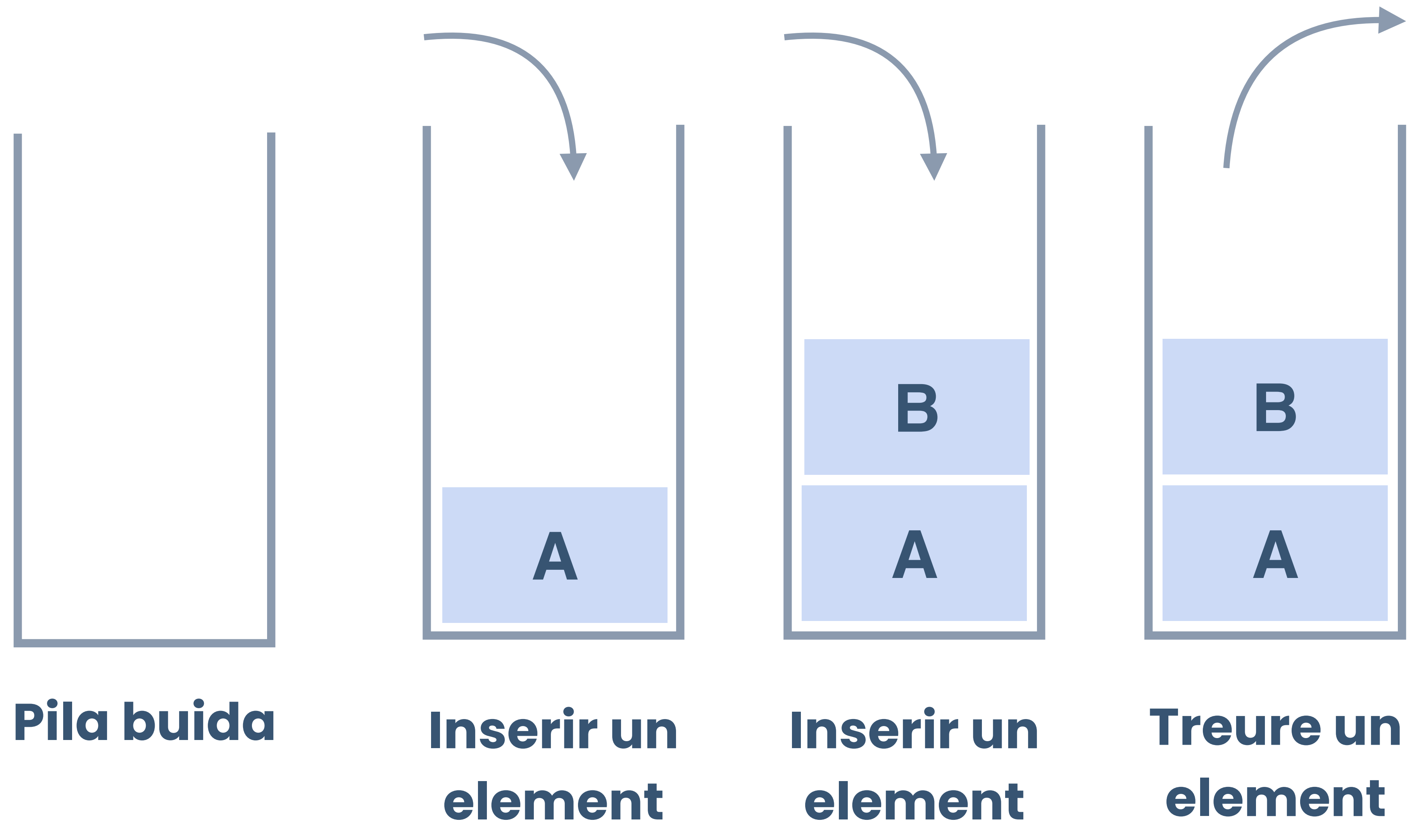
- Una pila és una estructura de dades on els elements s'hi afegeixen i s'hi eliminen per un únic extrem, de manera que només podem accedir a l'últim element que s'hi ha afegit.



El cost de la recursivitat: la pila de crides

Què és una pila?

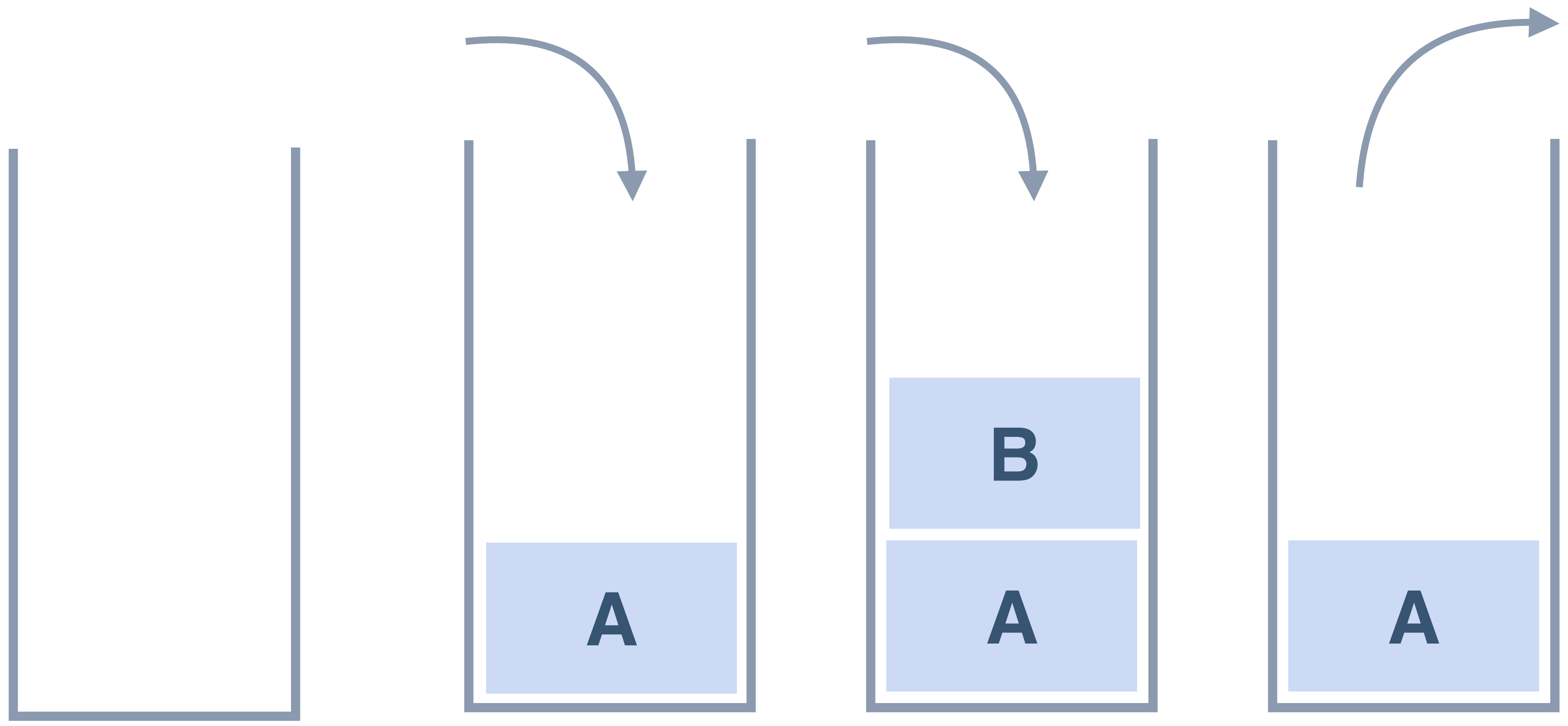
- Una pila és una estructura de dades on els elements s'hi afegeixen i s'hi eliminen per un únic extrem, de manera que només podem accedir a l'últim element que s'hi ha afegit.



El cost de la recursivitat: la pila de crides

Què és una pila?

- Una pila és una estructura de dades on els elements s'hi afegeixen i s'hi eliminen per un únic extrem, de manera que només podem accedir a l'últim element que s'hi ha afegit.

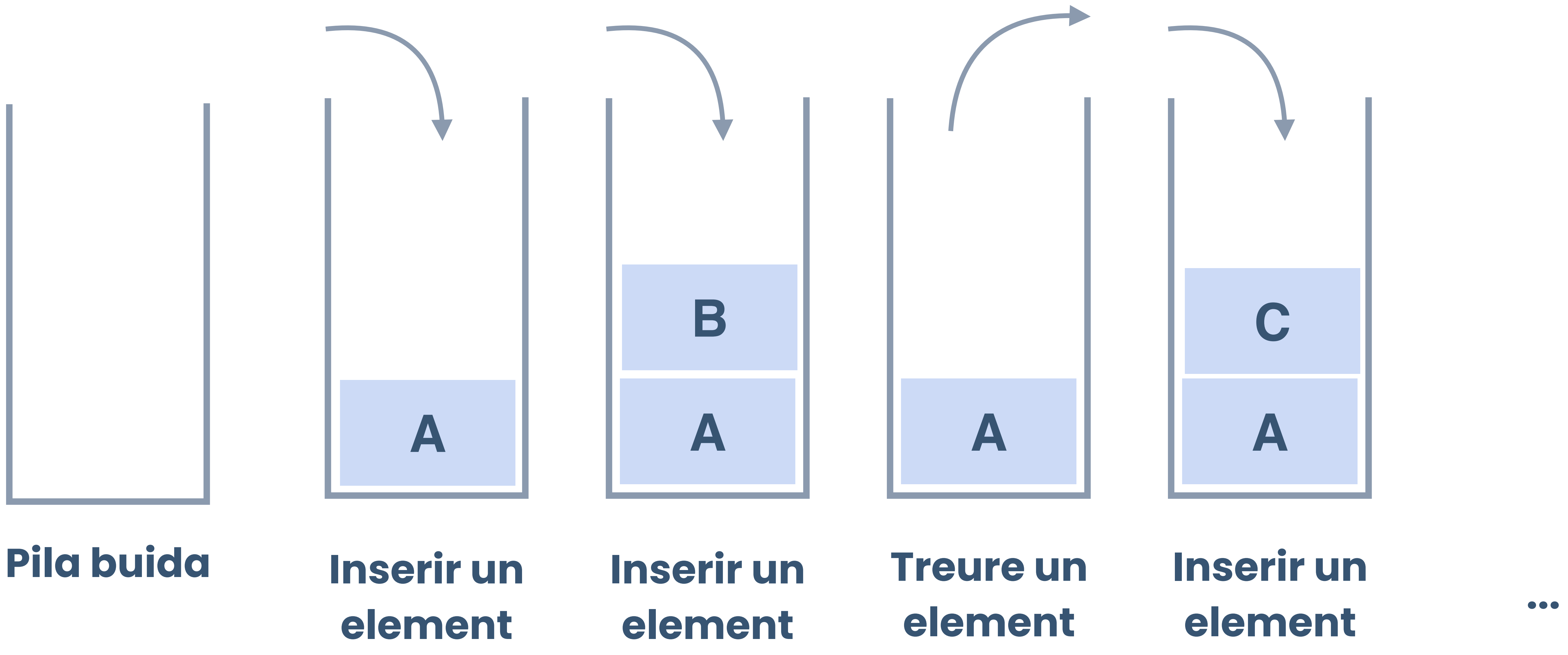


Pila buida | **Inserir un element** | **Inserir un element** | **Treure un element**

El cost de la recursivitat: la pila de crides

Què és una pila?

- Una pila és una estructura de dades on els elements s'hi afegeixen i s'hi eliminen per un únic extrem, de manera que només podem accedir a l'últim element que s'hi ha afegit.



El cost de la recursivitat: la pila de crides

La pila de crides

- Quan es crida una funció, el sistema reserva espai a la memòria perquè aquesta funció pugui fer la seva feina.
  - Sovint anomenem aquestes porcions de memòria *marcs de pila* o *marcs de funció* (**stack frames**).

El cost de la recursivitat: la pila de crides

La pila de crides

- Quan es crida una funció, el sistema reserva espai a la memòria perquè aquesta funció pugui fer la seva feina.
  - Sovint anomenem aquestes porcions de memòria *marcs de pila* o *marcs de funció* (**stack frames**).

- Cada stack frame conté la informació necessària per executar la funció: els paràmetres que ha rebut, les seves variables locals, i l'adreça on cal tornar quan la funció acabi.

```
int main() {
    int x = 7;
    int y = 10;
    printf("%d\n", suma(x,y));
    return 0;
}
```

Stack frame de main()
Variables locals: x = 7, y = 10 **Adreça de retorn:** (Al sistema operatiu)

El cost de la recursivitat: la pila de crides

La pila de crides

- Pot haver-hi més d'un marc de funció a la memòria al mateix temps.
  - Exemple: Si `main()` crida `printf()`, i aquesta al seu torn crida `suma()`, les tres funcions tenen marcs oberts.

```
int suma(int a, int b) {
     int resultat;
     resultat = a + b;
     return resultat;
}
```

```
int main() {
     int x = 7;
     int y = 10;
     printf("%d\n", suma(x,y));
     return 0;
}
```

Stack frame de main()

Variables locals
Adreça de retorn
...

Stack frame de printf()

Variables locals
Adreça de retorn
...

Stack frame de suma()

Variables locals
Adreça de retorn
...

El cost de la recursivitat: la pila de crides

La pila de crides

- Aquests marcs s'organitzen com una pila:
  - Cada nova crida afegeix un marc al capdamunt
  - Cada retorn de funció elimina el marc superior.

```
int suma(int a, int b) {
    int resultat;
    resultat = a + b;
    return resultat;
}
```

```
int main() {
    int x = 7;
    int y = 10;
    printf("%d\n", suma(x,y));
    return 0;
}
```

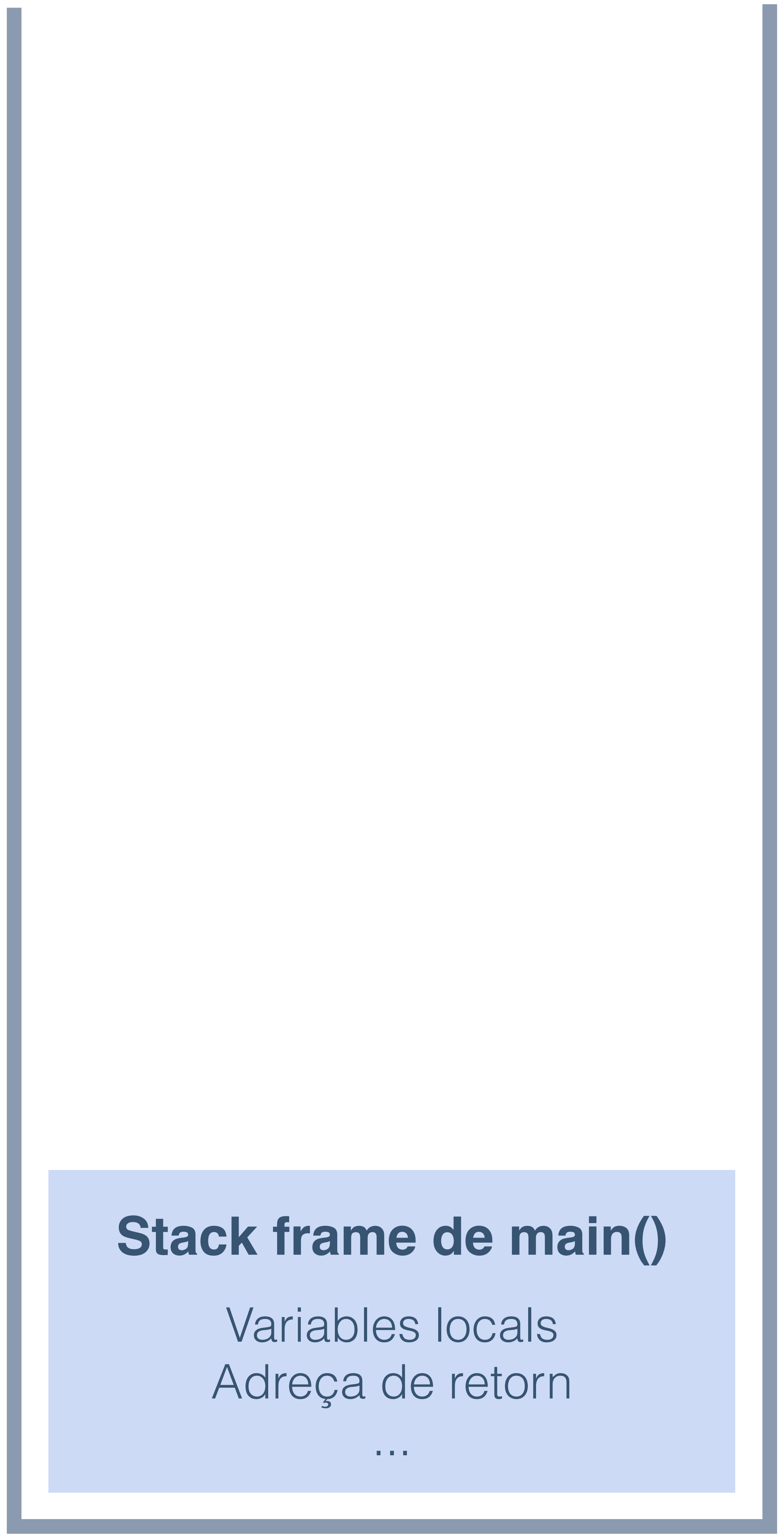
El cost de la recursivitat: la pila de crides

La pila de crides

- Aquests marcs s'organitzen com una pila:
  - Cada nova crida afegeix un marc al capdamunt
  - Cada retorn de funció elimina el marc superior.

```
int suma(int a, int b) {
    int resultat;
    resultat = a + b;
    return resultat;
}
```

```
int main() {
    int x = 7;
    int y = 10;
    printf("%d\n", suma(x,y));
    return 0;
}
```



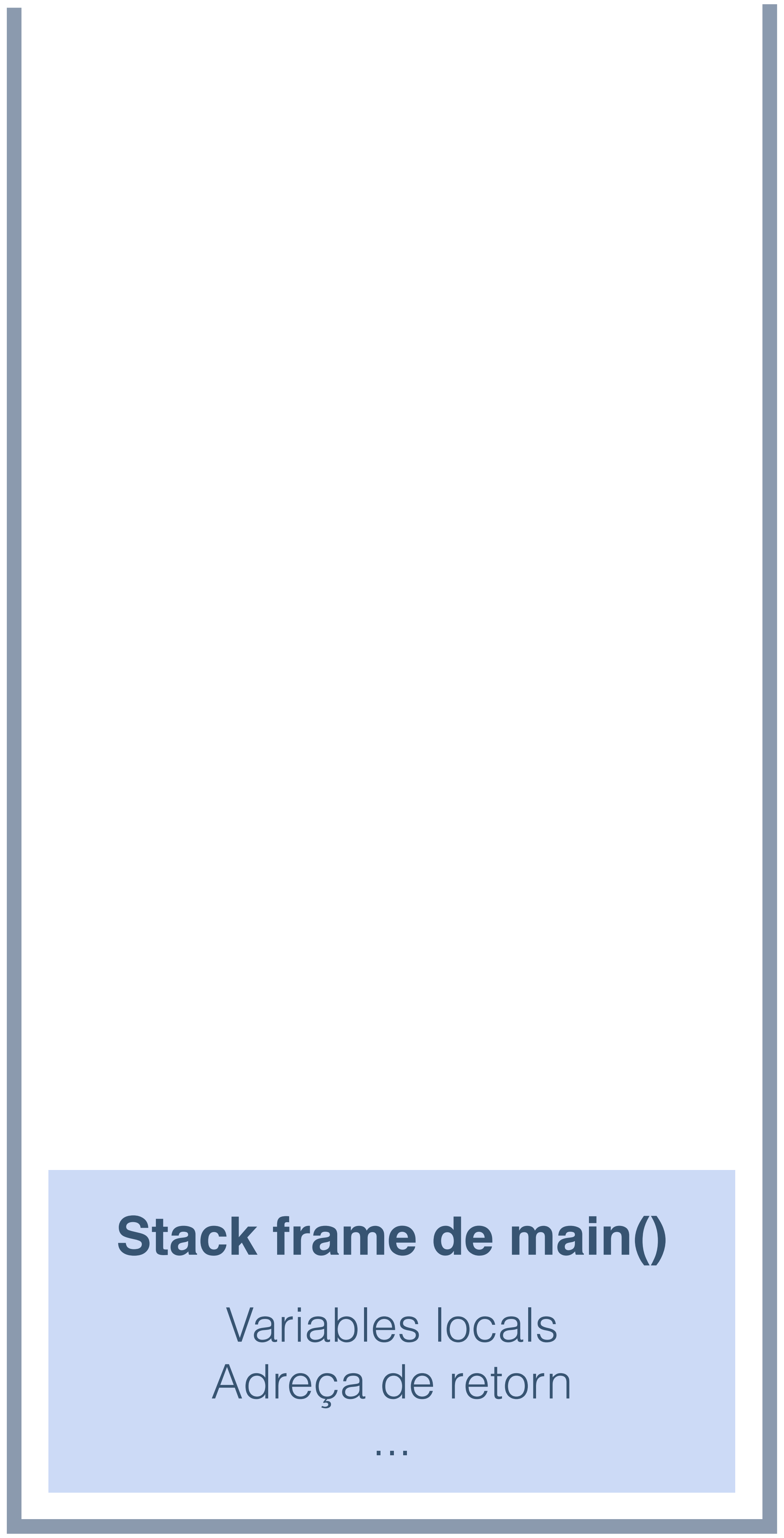
El cost de la recursivitat: la pila de crides

La pila de crides

- Aquests marcs s'organitzen com una pila:
  - Cada nova crida afegeix un marc al capdamunt
  - Cada retorn de funció elimina el marc superior.

```c
int suma(int a, int b) {
    int resultat;
    resultat = a + b;
    return resultat;
}
```

```c
int main() {
    int x = 7;
    int y = 10;
    printf("%d\n", suma(x,y));
    return 0;
}
```



El cost de la recursivitat: la pila de crides

La pila de crides

- Aquests marcs s’organitzen com una pila:
  - Cada nova crida afegeix un marc al capdamunt
  - Cada retorn de funció elimina el marc superior.

```
int suma(int a, int b) {
    int resultat;
    resultat = a + b;
    return resultat;
}
```

```
int main() {
    int x = 7;
    int y = 10;
    printf("%d\n", suma(x,y));
    return 0;
}
```

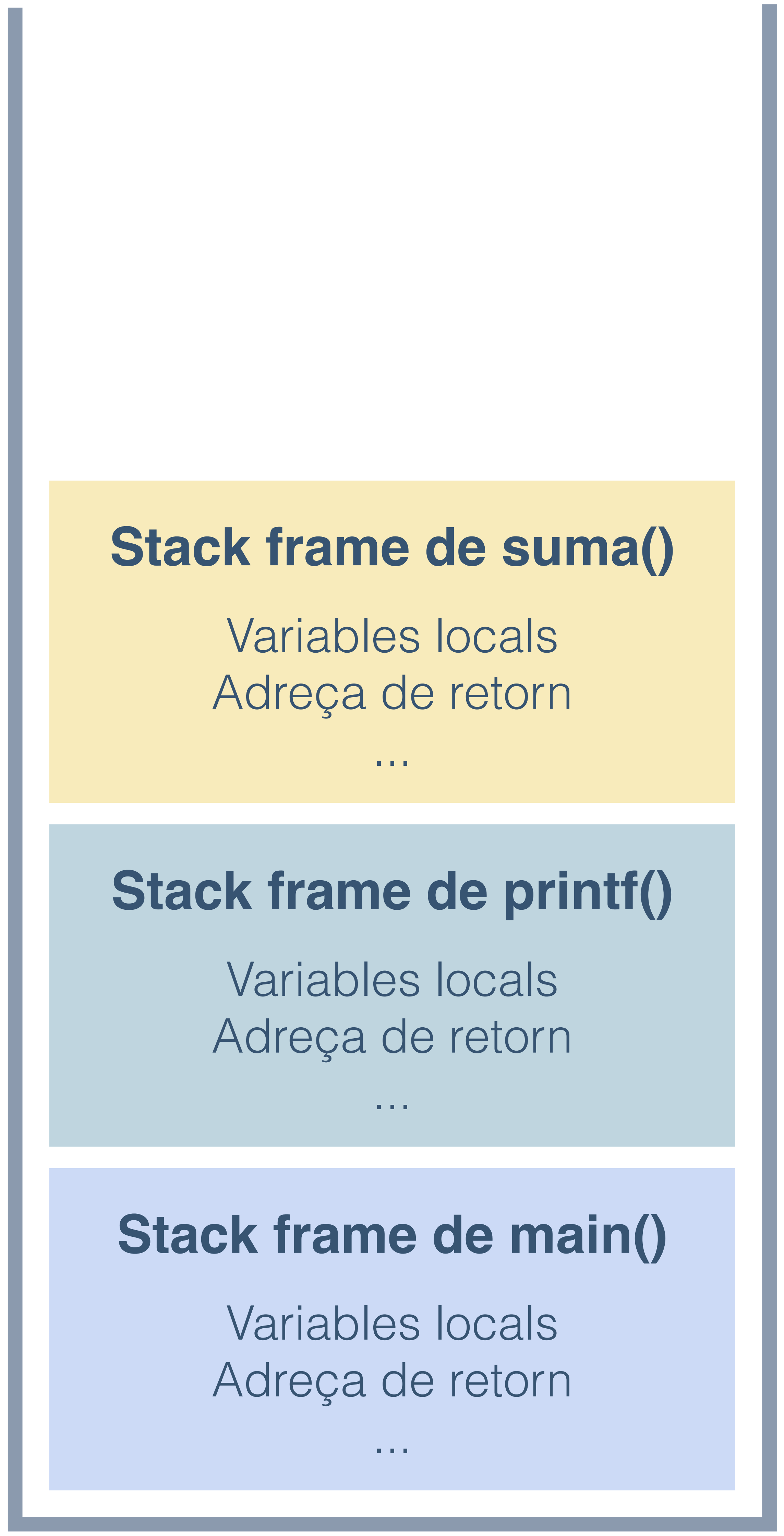
El cost de la recursivitat: la pila de crides

La pila de crides

- Aquests marcs s'organitzen com una pila:
  - Cada nova crida afegeix un marc al capdamunt
  - Cada retorn de funció elimina el marc superior.

```c
int suma(int a, int b) {
    int resultat;
    resultat = a + b;
    return resultat;
}
```

```c
int main() {
    int x = 7;
    int y = 10;
    printf("%d\n", suma(x,y));
    return 0;
}
```



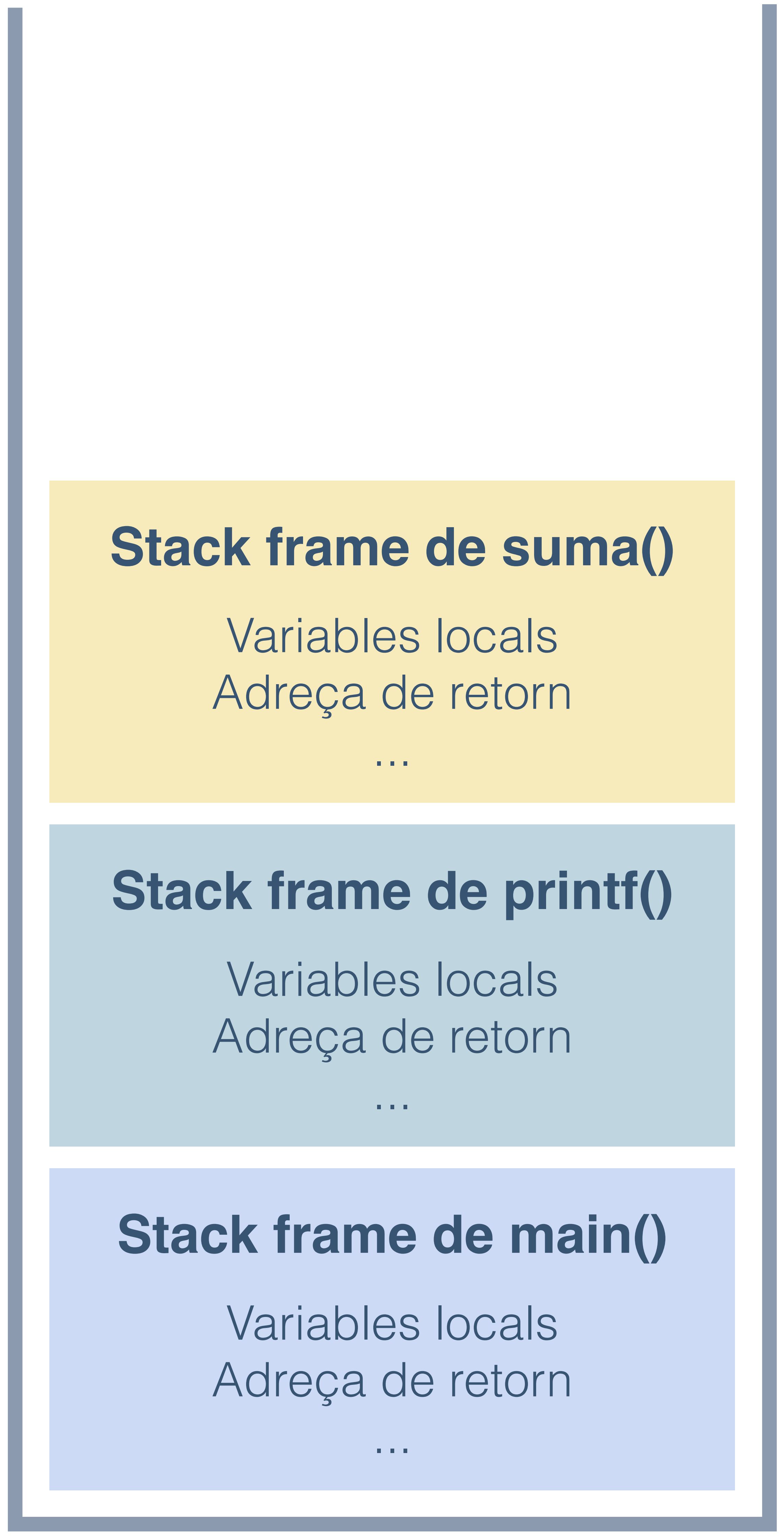
El cost de la recursivitat: la pila de crides

La pila de crides

- Aquests marcs s’organitzen com una pila:
  - Cada nova crida afegeix un marc al capdamunt
  - Cada retorn de funció elimina el marc superior.

```
int suma(int a, int b) {
    int resultat;
    resultat = a + b;
    return resultat;
}
```

```
int main() {
    int x = 7;
    int y = 10;
    printf("%d\n", suma(x,y));
    return 0;
}
```



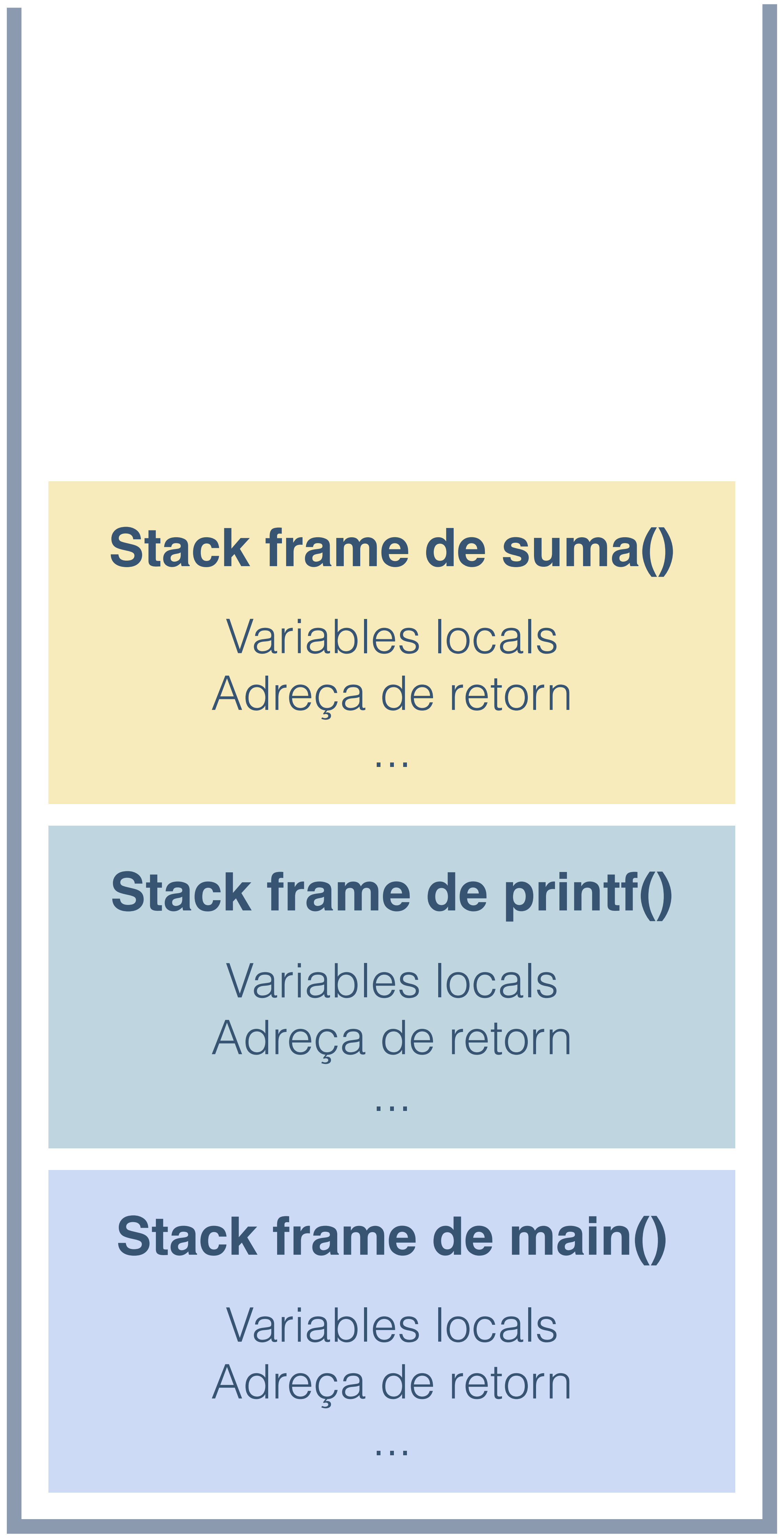
El cost de la recursivitat: la pila de crides

La pila de crides

- Aquests marcs s'organitzen com una pila:
  - Cada nova crida afegeix un marc al capdamunt
  - Cada retorn de funció elimina el marc superior.

```c
int suma(int a, int b) {
    int resultat;
    resultat = a + b;
    return resultat;
}
```

```c
int main() {
    int x = 7;
    int y = 10;
    printf("%d\n", suma(x,y));
    return 0;
}
```



El cost de la recursivitat: la pila de crides

La pila de crides

- Aquests marcs s'organitzen com una pila:
  - Cada nova crida afegeix un marc al capdamunt
  - Cada retorn de funció elimina el marc superior.

```c
int suma(int a, int b) {
    int resultat;
    resultat = a + b;
    return resultat;
}
```

```c
int main() {
    int x = 7;
    int y = 10;
    printf("%d\n", suma(x,y));
    return 0;
}
```

El cost de la recursivitat: la pila de crides

La pila de crides

- Aquests marcs s'organitzen com una pila:
  - Cada nova crida afegeix un marc al capdamunt
  - Cada retorn de funció elimina el marc superior.

```
int suma(int a, int b) {
    int resultat;
    resultat = a + b;
    return resultat;
}
```

```
int main() {
    int x = 7;
    int y = 10;
    printf("%d\n", suma(x,y));
    return 0;
}
```

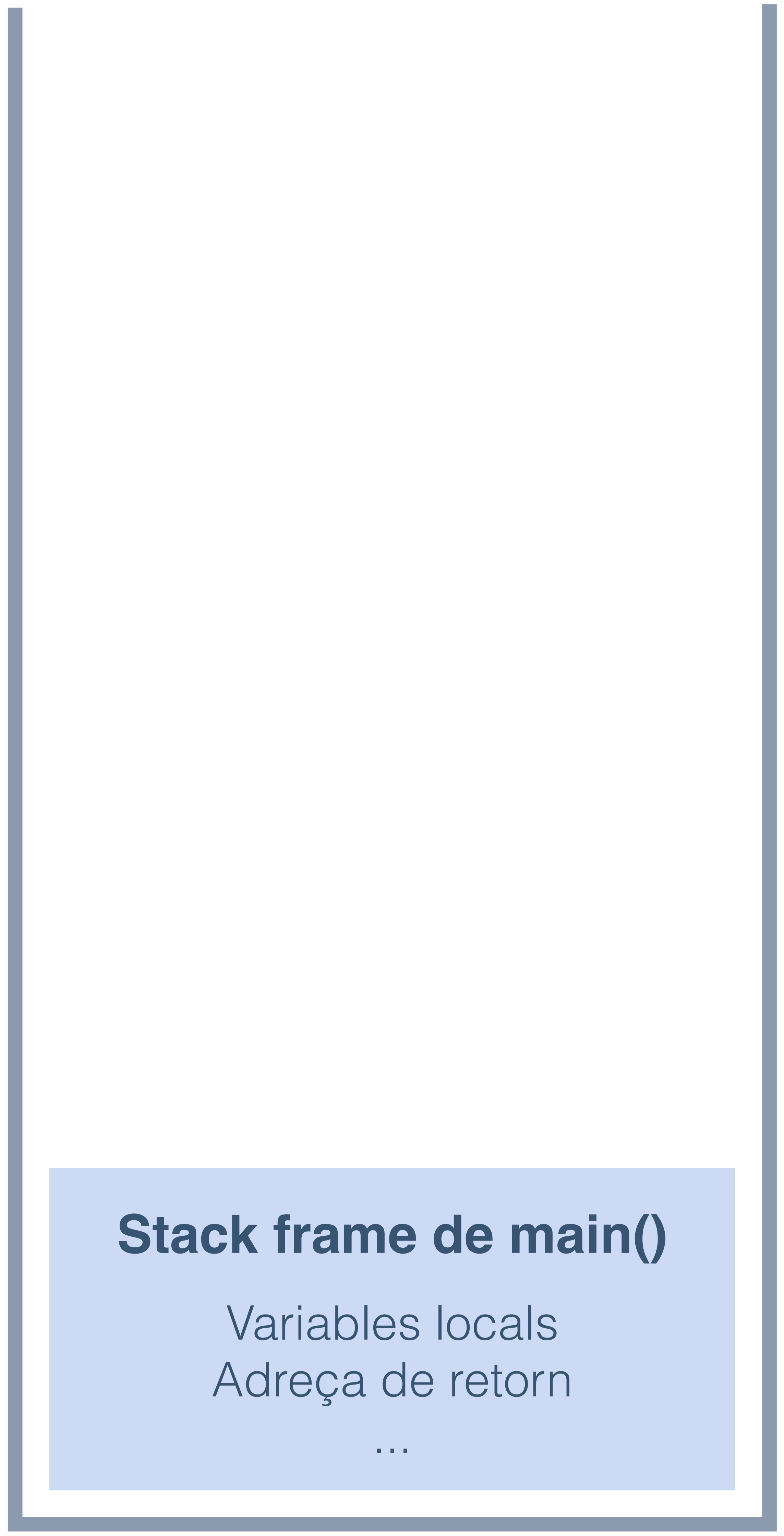
El cost de la recursivitat: la pila de crides

La pila de crides

- Aquests marcs s'organitzen com una pila:
  - Cada nova crida afegeix un marc al capdamunt
  - Cada retorn de funció elimina el marc superior.

```
int suma(int a, int b) {
    int resultat;
    resultat = a + b;
    return resultat;
}
```

```
int main() {
    int x = 7;
    int y = 10;
    printf("%d\n", suma(x,y));
    return 0;
}
```



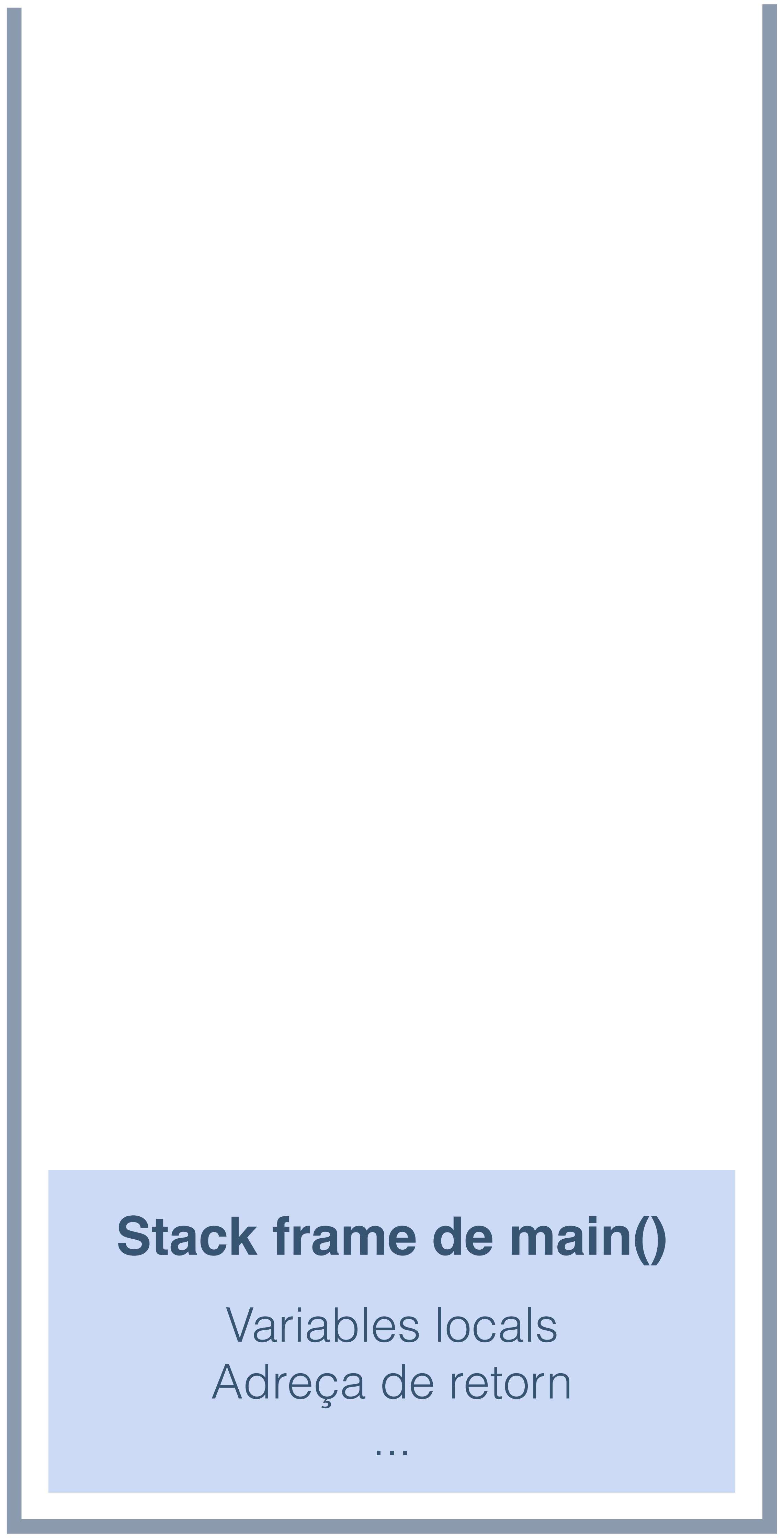
El cost de la recursivitat: la pila de crides

La pila de crides

- Aquests marcs s'organitzen com una pila:
  - Cada nova crida afegeix un marc al capdamunt
  - Cada retorn de funció elimina el marc superior.

```
int suma(int a, int b) {
    int resultat;
    resultat = a + b;
    return resultat;
}
```

```
int main() {
    int x = 7;
    int y = 10;
    printf("%d\n", suma(x,y));
    return 0;
}
```



El cost de la recursivitat: la pila de crides

La pila de crides

- Aquests marcs s'organitzen com una pila:
  - Cada nova crida afegeix un marc al capdamunt
  - Cada retorn de funció elimina el marc superior.

```
int suma(int a, int b) {
    int resultat;
    resultat = a + b;
    return resultat;
}
```

```
int main() {
    int x = 7;
    int y = 10;
    printf("%d\n", suma(x,y));
    return 0;
}
```

El cost de la recursivitat: la pila de crides

```
int main() {
  factorial(4);
  return 0;
}
```

```
int factorial(int n) {
    int valor = 1;
    for(int i = 2; i <= n; i++){
      valor = valor * i;
    }
    return valor;
}
```

Funció iterativa

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

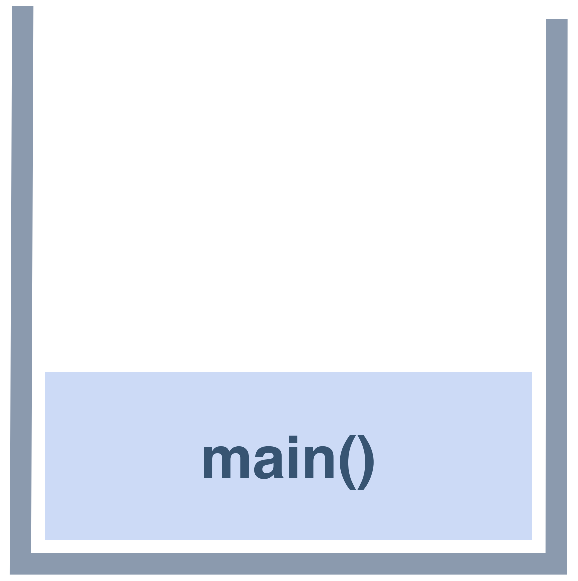
Funció recursiva

El cost de la recursivitat: la pila de crides

```
int main() {
  factorial(4);
  return 0;
}
```

```
int factorial(int n) {
    int valor = 1;
    for(int i = 2; i <= n; i++){
      valor = valor * i;
    }
    return valor;
}
```

Funció iterativa



Pila de crides d'una funció **iterativa**

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

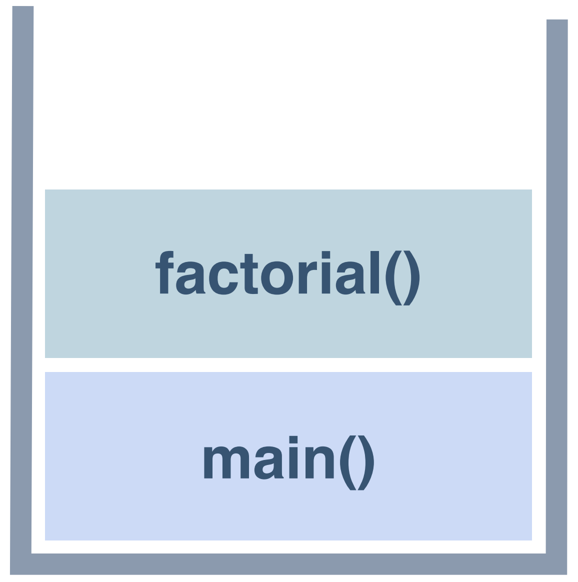
Funció recursiva

El cost de la recursivitat: la pila de crides

```
int main() {
  factorial(4);
  return 0;
}
```

```
int factorial(int n) {
    int valor = 1;
    for(int i = 2; i <= n; i++){
      valor = valor * i;
    }
    return valor;
}
```

Funció iterativa



Pila de crides
d'una funció
iterativa

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

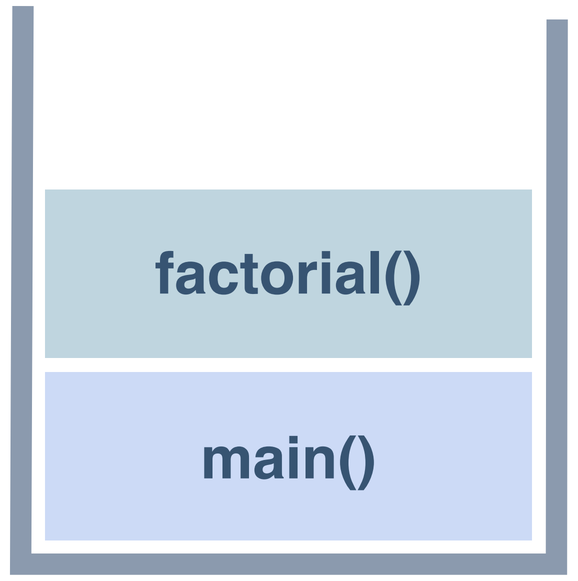
Funció recursiva

El cost de la recursivitat: la pila de crides

```c
int main() {
  factorial(4);
  return 0;
}
```

```c
int factorial(int n) {
    int valor = 1;
    for(int i = 2; i <= n; i++){
      valor = valor * i;
    }
    return valor;
}
```

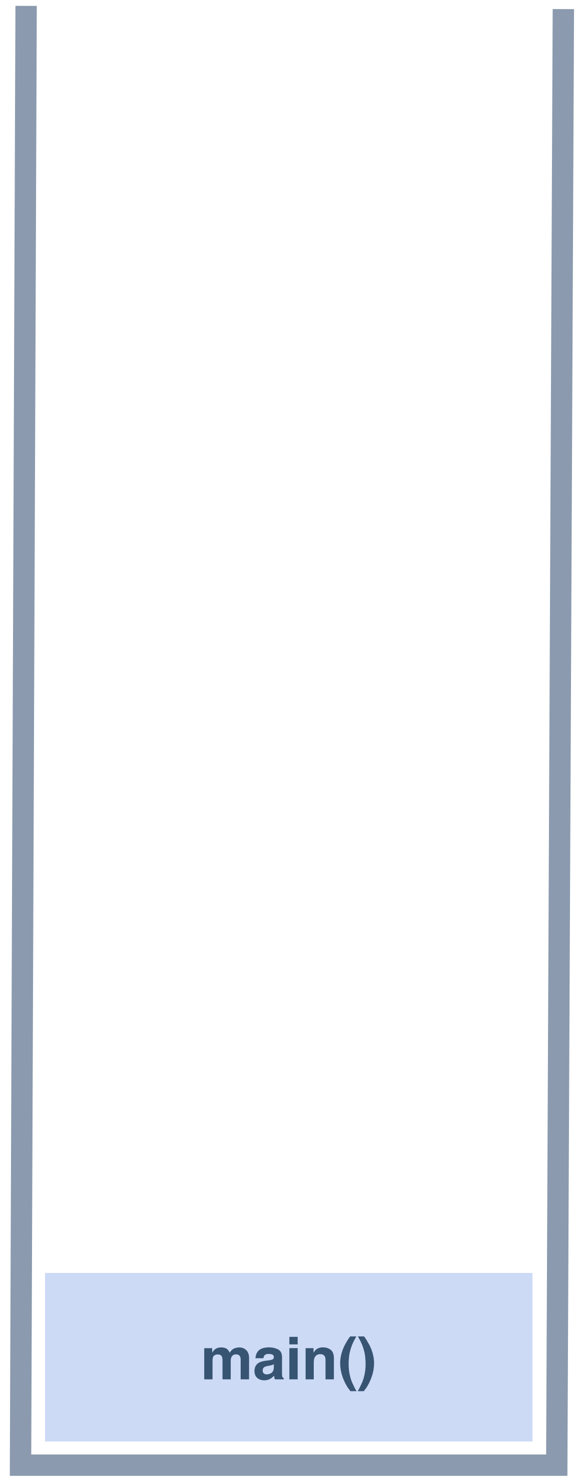
Funció iterativa



Pila de crides d'una funció **iterativa**

```c
int factorial(int n) {
   if (n == 0)
       return 1;
   else
       return n * factorial(n-1);
}
```

Funció recursiva



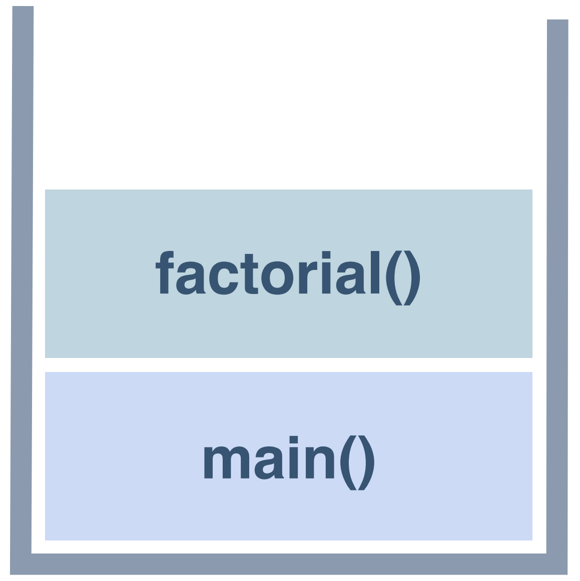
Pila de crides d'una funció **recursiva**

El cost de la recursivitat: la pila de crides

```
int main() {
  factorial(4);
  return 0;
}
```

```
int factorial(int n) {
    int valor = 1;
    for(int i = 2; i <= n; i++){
      valor = valor * i;
    }
    return valor;
}
```

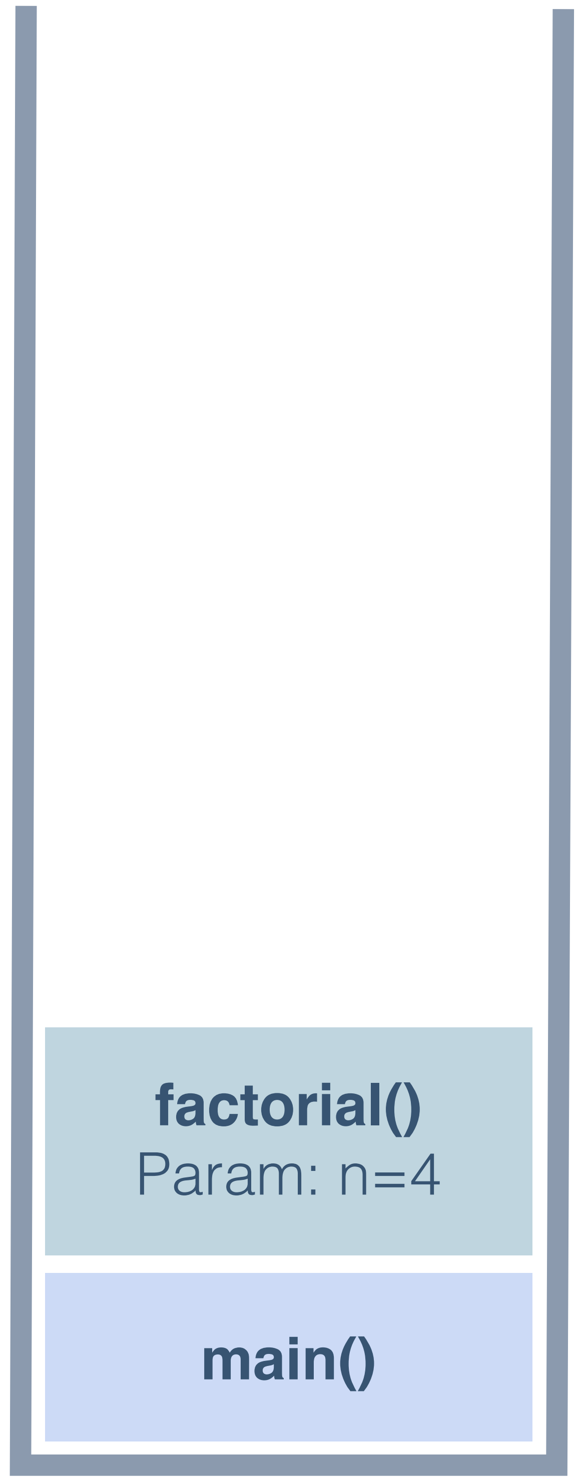
Funció iterativa



Pila de crides
d'una funció
iterativa

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

Funció recursiva



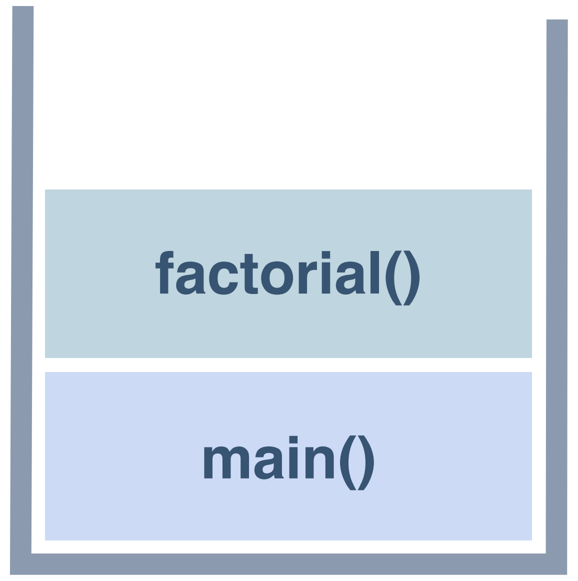
Pila de crides
d'una funció
recursiva

El cost de la recursivitat: la pila de crides

```
int main() {
  factorial(4);
  return 0;
}
```

```
int factorial(int n) {
    int valor = 1;
    for(int i = 2; i <= n; i++){
      valor = valor * i;
    }
    return valor;
}
```

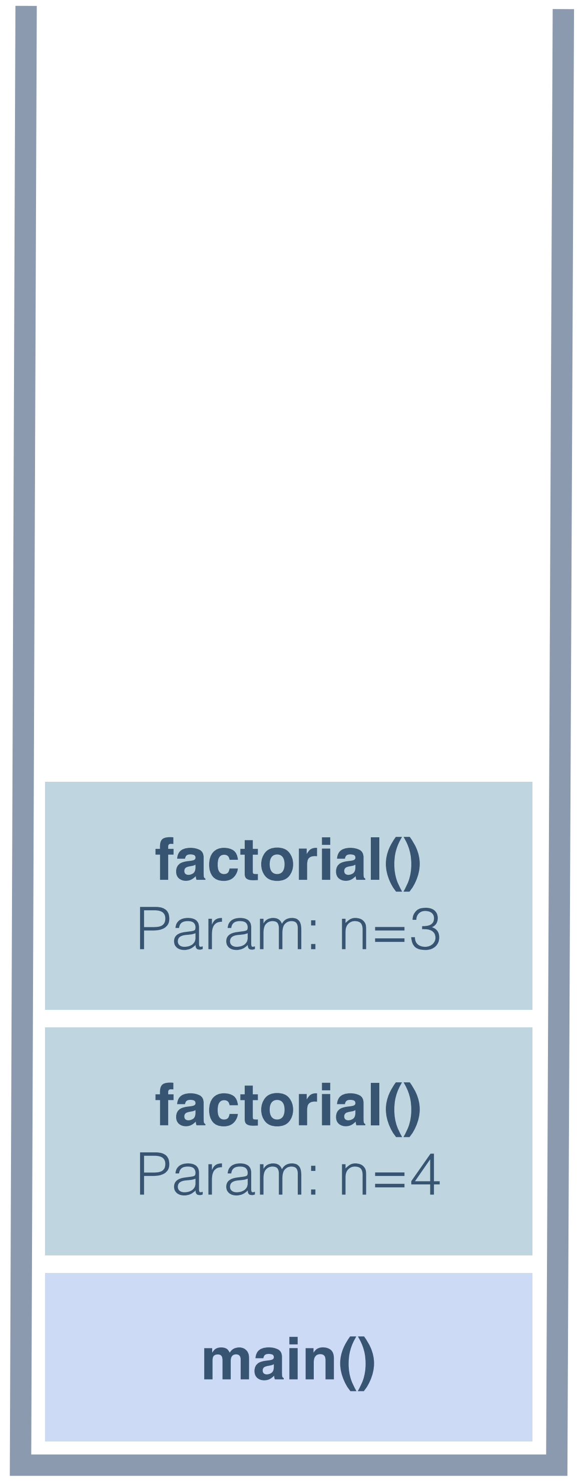
Funció iterativa



Pila de crides d'una funció **iterativa**

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

Funció recursiva



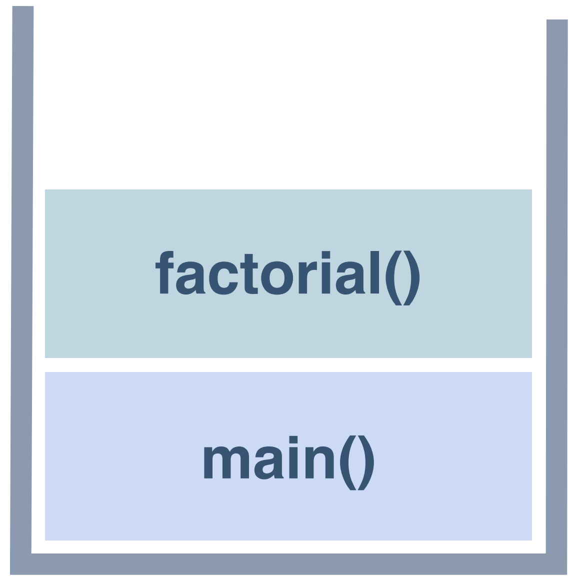
Pila de crides d'una funció **recursiva**

El cost de la recursivitat: la pila de crides

```
int main() {
  factorial(4);
  return 0;
}
```

```
int factorial(int n) {
    int valor = 1;
    for(int i = 2; i <= n; i++){
      valor = valor * i;
    }
    return valor;
}
```

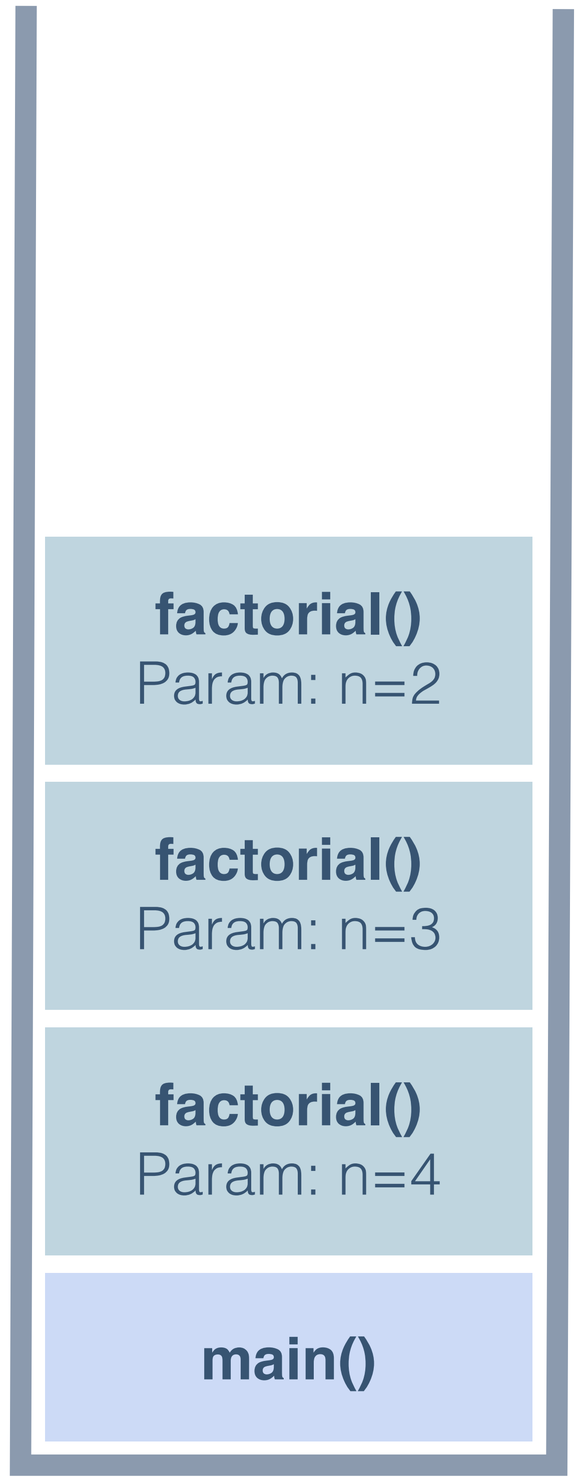
Funció iterativa



Pila de crides d'una funció **iterativa**

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

Funció recursiva



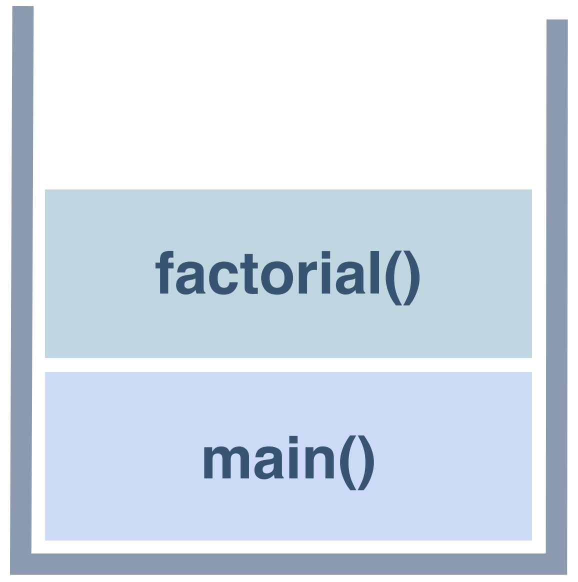
Pila de crides d'una funció **recursiva**

El cost de la recursivitat: la pila de crides

```
int main() {
  factorial(4);
  return 0;
}
```

```
int factorial(int n) {
    int valor = 1;
    for(int i = 2; i <= n; i++){
      valor = valor * i;
    }
    return valor;
}
```

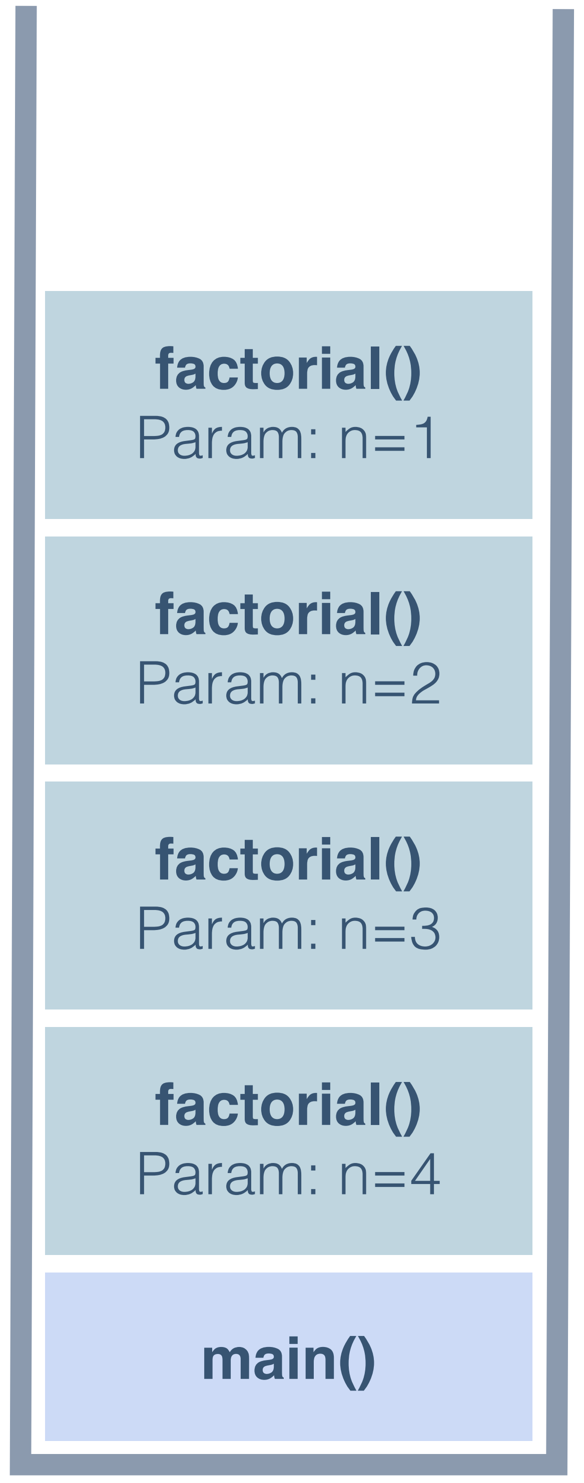
Funció iterativa



Pila de crides d'una funció **iterativa**

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

Funció recursiva



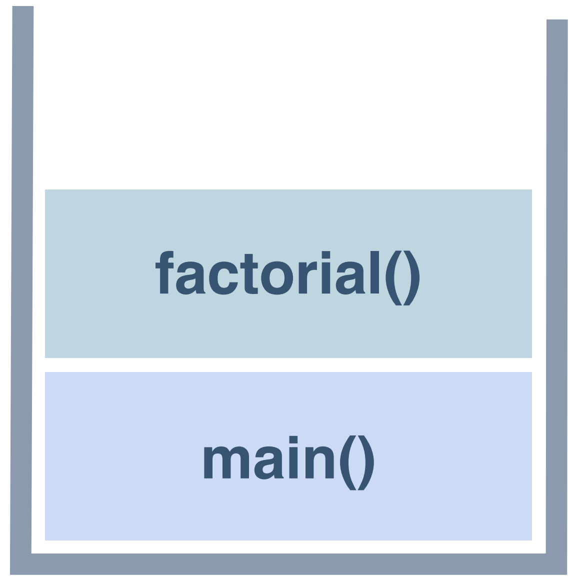
Pila de crides d'una funció **recursiva**

El cost de la recursivitat: la pila de crides

```c
int main() {
  factorial(4);
  return 0;
}
```

```c
int factorial(int n) {
    int valor = 1;
    for(int i = 2; i <= n; i++){
      valor = valor * i;
    }
    return valor;
}
```

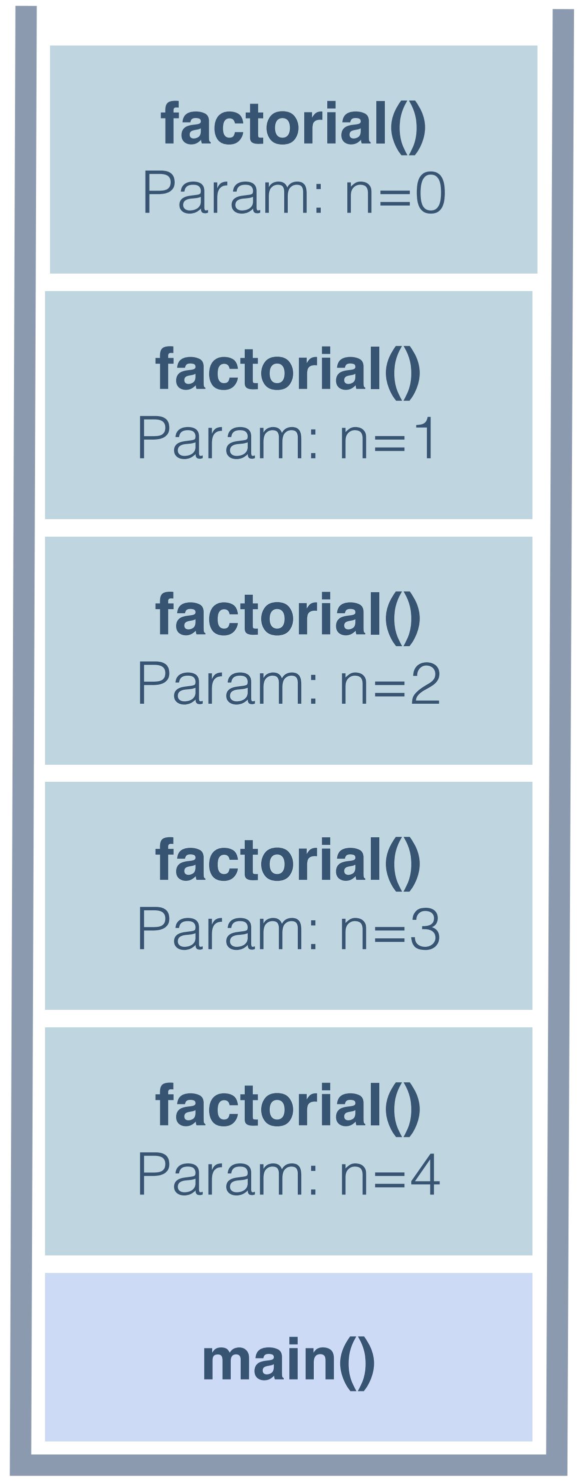
Funció iterativa



Pila de crides d'una funció **iterativa**

```c
int factorial(int n) {
   if (n == 0)
       return 1;
   else
       return n * factorial(n-1);
}
```

Funció recursiva



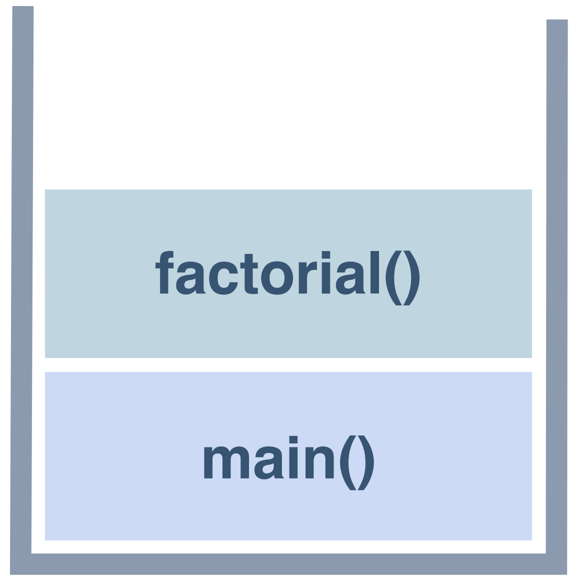
Pila de crides d'una funció **recursiva**

El cost de la recursivitat: la pila de crides

```
int main() {
  factorial(4);
  return 0;
}
```

```
int factorial(int n) {
    int valor = 1;
    for(int i = 2; i <= n; i++){
      valor = valor * i;
    }
    return valor;
}
```

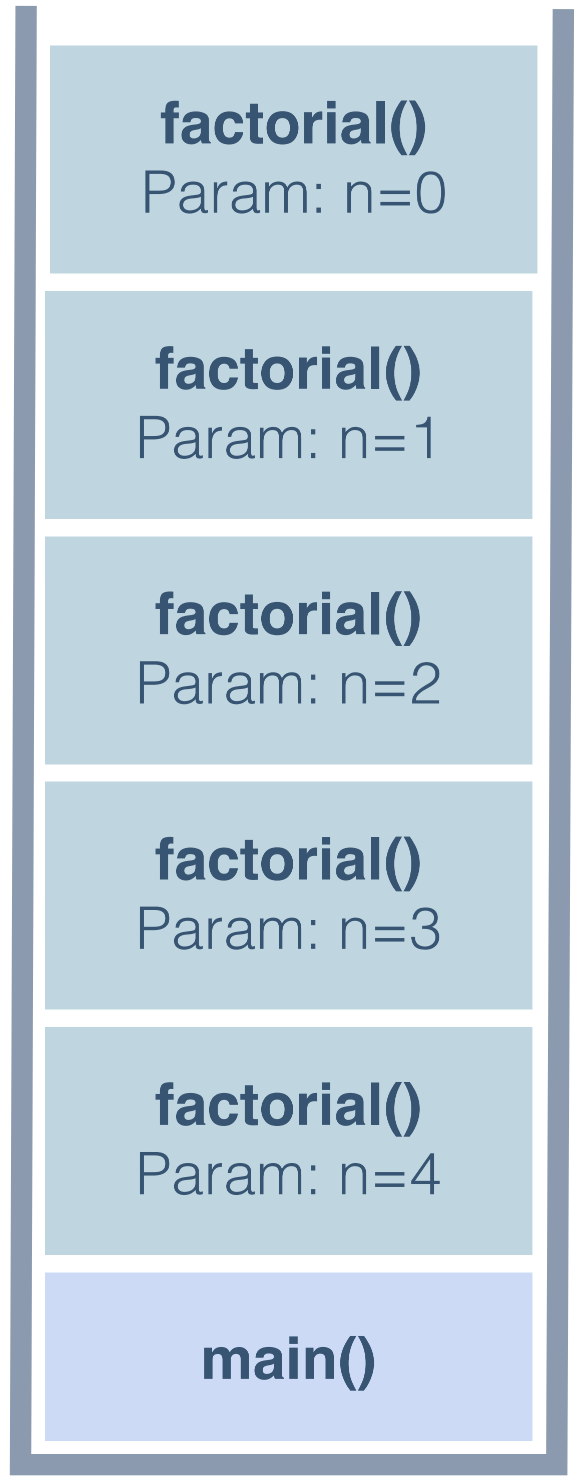
Funció iterativa



Pila de crides d'una funció **iterativa**

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

Funció recursiva



Pila de crides d'una funció **recursiva**

Tipus de recursivitat

Segons el nombre de crides recursives: simple vs. múltiple

- Es diu que una funció és recursiva simple si hi ha una sola crida recursiva. (e.g. el factorial)
- En canvi, és recursiva múltiple si hi ha més d'una crida recursiva. Per exemple, el càlcul del terme n-èssim de la successió de Fibonacci.

Tipus de recursivitat

Segons el nombre de crides recursives: simple vs. múltiple

- Es diu que una funció és recursiva simple si hi ha una sola crida recursiva. (e.g. el factorial)
- En canvi, és recursiva múltiple si hi ha més d'una crida recursiva. Per exemple, el càlcul del terme n-èssim de la successió de Fibonacci.

La successió de Fibonacci comença amb dos nombres naturals, el 0, i l'1, i cada terme successiu es calcula com la suma dels dos anteriors.

$$F(0) = 0$$

$$F(1) = 1$$

$$F(n) = F(n-1) + F(n-2)$$

Tipus de recursivitat

Segons el nombre de crides recursives: simple vs. múltiple

- Es diu que una funció és recursiva simple si hi ha una sola crida recursiva. (e.g. el factorial)
- En canvi, és recursiva múltiple si hi ha més d'una crida recursiva. Per exemple, el càlcul del terme n-èssim de la successió de Fibonacci.

La successió de Fibonacci comença amb dos nombres naturals, el 0, i l'1, i cada terme successiu es calcula com la suma dels dos anteriors.

$F(0) = 0$

$F(1) = 1$

$F(n) = F(n-1) + F(n-2)$

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

Tipus de recursivitat

Segons el nombre de crides recursives: simple vs. múltiple

- Es diu que una funció és recursiva simple si hi ha una sola crida recursiva. (e.g. el factorial)
- En canvi, és recursiva múltiple si hi ha més d'una crida recursiva. Per exemple, el càlcul del terme n-èssim de la successió de Fibonacci.

La successió de Fibonacci comença amb dos nombres naturals, el 0, i l'1, i cada terme successiu es calcula com la suma dels dos anteriors.

$F(0) = 0$

$F(1) = 1$

$F(n) = F(n-1) + F(n-2)$

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

```
int fib(int n)
{
    if (n == 0){
       return 0;
    }
    else if (n == 1){
       return 1;
    }
    else {
      return fib(n-1)+fib(n-2);
    }
}
```

Hi ha dues crides recursives (recursivitat múltiple)

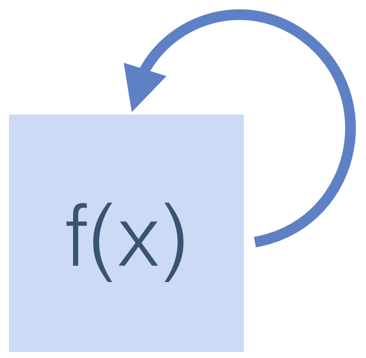
Tipus de recursivitat

Segons la naturalesa de la crida: directa vs. indirecta

Tipus de recursivitat

Segons la naturalesa de la crida: directa vs. indirecta

- Una funció és recursiva directa si es crida a ella mateixa directament.



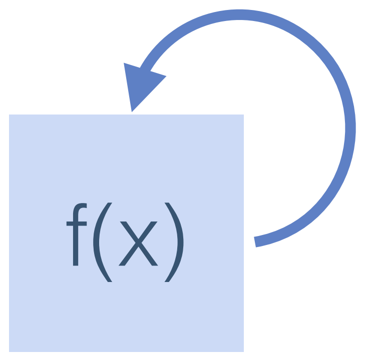

```
void directa() {
    printf("Recursivitat directa\n");
    directa(); // La funció es crida
directament a ella mateixa
}
```

⚠️ *Compte que aquest codi genera infinites crides recursives!*

Tipus de recursivitat

Segons la naturalesa de la crida: directa vs. indirecta

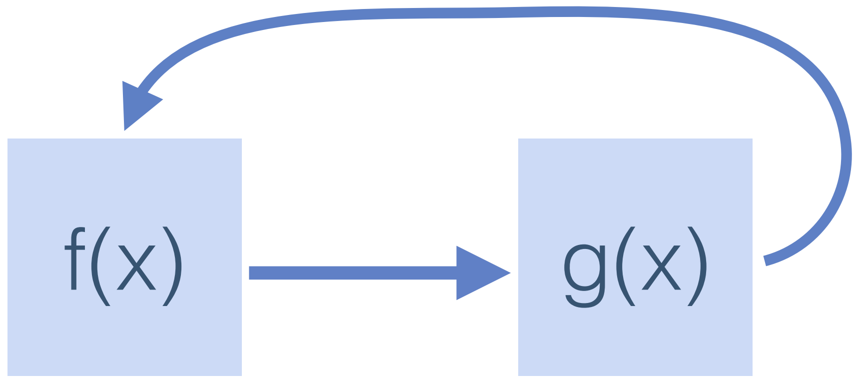
- Una funció és recursiva directa si es crida a ella mateixa directament.




```c
void directa() {
    printf("Recursivitat directa\n");
    directa(); // La funció es crida directament a ella mateixa
}
```

⚠️ *Compte que aquest codi genera infinites crides recursives!*

- En canvi, és recursiva indirecta si es crida de manera transitiva a través d'una o més funcions.




```c
void funcA() {
    printf("Recursivitat indirecta\n");
    funcB(); // Es crida a funcB
}

void funcB() {
    funcA(); // Es torna a cridar funcA
}
```

⚠️ *Aquest també!*

Tipus de recursivitat

Segons la posició de la crida: final vs. no-final

- Si la crida recursiva no és l'última operació, és a dir, hi ha càlculs pendents després de la crida recursiva, parlem de recursivitat no final.

  *En aquest exemple, quan tornem de cridar factorialNoFinal encara hem de multiplicar per **n**. Per això és no final.*

```
int factorialNoFinal(int n) {
    if (n == 0) {
        return 1;
    }
    return n * factorialNoFinal(n - 1);
}
```

Tipus de recursivitat

Segons la posició de la crida: final vs. no-final

- Si la crida recursiva no és l'última operació, és a dir, hi ha càlculs pendents després de la crida recursiva, parlem de recursivitat no final.

  En aquest exemple, quan tornem de cridar factorialNoFinal encara hem de multiplicar per ***n****. Per això és no final.*

```
int factorialNoFinal(int n) {
    if (n == 0) {
        return 1;
    }
    return n * factorialNoFinal(n - 1);
}
```

- En canvi, si la crida recursiva es l'última operació que fa la funció abans de retornar un resultat, parlem de recursivitat final (tail recursion).

Tipus de recursivitat

Segons la posició de la crida: final vs. no-final

- Si la crida recursiva no és l'última operació, és a dir, hi ha càlculs pendents després de la crida recursiva, parlem de recursivitat no final.

  *En aquest exemple, quan tornem de cridar factorialNoFinal encara hem de multiplicar per **n**. Per això és no final.*

```
int factorialNoFinal(int n) {
    if (n == 0) {
        return 1;
    }
    return n * factorialNoFinal(n - 1);
}
```

- En canvi, si la crida recursiva es l'última operació que fa la funció abans de retornar un resultat, parlem de recursivitat final (tail recursion).

```
int factorialFinal(int n, int acumulador) {
    if (n == 0) {
       // Cas base: tornem el resultat acumulat
        return acumulador;
    }
    return factorialFinal(n - 1, n * acumulador);
}
```

Tipus de recursivitat

Segons la posició de la crida: final vs. no-final

- Si la crida recursiva no és l'última operació, és a dir, hi ha càlculs pendents després de la crida recursiva, parlem de recursivitat no final.

  *En aquest exemple, quan tornem de cridar factorialNoFinal encara hem de multiplicar per **n**. Per això és no final.*

```
int factorialNoFinal(int n) {
    if (n == 0) {
        return 1;
    }
    return n * factorialNoFinal(n - 1);
}
```

- En canvi, si la crida recursiva es l'última operació que fa la funció abans de retornar un resultat, parlem de recursivitat final (tail recursion).

La distinció és important quan parlem d'eficiència. La recursivitat final permet una transformació a iteratiu gairebé automàtica. Això ens permet fer optimitzacions del codi que el facin més eficient. Per altra banda, la recursivitat no-final implica que el resultat de la crida recursiva s'ha de guardar a la pila fins que es resolguin els càlculs pendents.

```
int factorialFinal(int n, int acumulador) {
    if (n == 0) {
        // Cas base: tornem el resultat acumulat
        return acumulador;
    }
    return factorialFinal(n - 1, n * acumulador);
}
```

Tipus de recursivitat

Segons la posició de la crida: final vs. no-final

```
int factorialNoFinal(int n) {
    if (n == 0) {
        return 1;
    }
    return n * factorialNoFinal(n - 1);
}
```

Tipus de recursivitat

Segons la posició de la crida: final vs. no-final

```
int factorialNoFinal(int n) {
    if (n == 0) {
        return 1;
    }
    return n * factorialNoFinal(n - 1);
}
```

L'arbre de crides és ramificat amb càlculs pendents:

```
factorialNoFinal(4)
↳ 4 * factorialNoFinal(3)
      ↳ 3 * factorialNoFinal(2)
            ↳ 2 * factorialNoFinal(1)
                  ↳ 1 * factorialNoFinal(0)
                        ↳ Retorna 1
```

Tipus de recursivitat

Segons la posició de la crida: final vs. no-final

```
int factorialNoFinal(int n) {
    if (n == 0) {
        return 1;
    }
    return n * factorialNoFinal(n - 1);
}
```

L'arbre de crides és ramificat amb càlculs pendents:

```
factorialNoFinal(4)
 ↳ 4 * factorialNoFinal(3)
        ↳ 3 * factorialNoFinal(2)
               ↳ 2 * factorialNoFinal(1)
                      ↳ 1 * factorialNoFinal(0)
                             ↳ Retorna 1
```

Cada nivell ha de guardar l'operació pendent fins que totes les crides recursives s'acabin, i després torna a calcular de tornada.

```
              factorialNoFinal(4)
   -> 4 * (3 * (2 * (1 * 1))) -> retorna 24
```

Tipus de recursivitat

Segons la posició de la crida: final vs. no-final

```
int factorialNoFinal(int n) {
    if (n == 0) {
        return 1;
    }
    return n * factorialNoFinal(n - 1);
}
```

L'arbre de crides és ramificat amb càlculs pendents:

```
factorialNoFinal(4) =24
 ↳ 4 * factorialNoFinal(3) =12
       ↳ 3 * factorialNoFinal(2)  =6
             ↳ 2 * factorialNoFinal(1)  =2
                   ↳ 1 * factorialNoFinal(0)  =1
                         ↳ Retorna 1
```

Cada nivell ha de guardar l'operació pendent fins que totes les crides recursives s'acabin, i després torna a calcular de tornada.

```
            factorialNoFinal(4)
   -> 4 * (3 * (2 * (1 * 1))) -> retorna 24
```

Tipus de recursivitat

Segons la posició de la crida: final vs. no-final

```
int factorialNoFinal(int n) {
    if (n == 0) {
        return 1;
    }
    return n * factorialNoFinal(n - 1);
}
```

L'arbre de crides és ramificat amb càlculs pendents:

```
factorialNoFinal(4) =24
↳ 4 * factorialNoFinal(3) =12
      ↳ 3 * factorialNoFinal(2) =6
            ↳ 2 * factorialNoFinal(1) =2
                  ↳ 1 * factorialNoFinal(0) =1
                        ↳ Retorna 1
```

Cada nivell ha de guardar l'operació pendent fins que totes les crides recursives s'acabin, i després torna a calcular de tornada.

```
          factorialNoFinal(4)
-> 4 * (3 * (2 * (1 * 1))) -> retorna 24
```

Pila durant les crides:

factorialNoFinal(0)	n = 0
factorialNoFinal(1)	n = 1
factorialNoFinal(2)	n = 2
factorialNoFinal(3)	n = 3
factorialNoFinal(4)	n = 4
main()	

Pila de crides

Tipus de recursivitat

Segons la posició de la crida: final vs. no-final

```
int factorialNoFinal(int n) {
    if (n == 0) {
        return 1;
    }
    return n * factorialNoFinal(n - 1);
}
```

L'arbre de crides és ramificat amb càlculs pendents:

```
factorialNoFinal(4) =24
↳ 4 * factorialNoFinal(3) =12
    ↳ 3 * factorialNoFinal(2) =6
        ↳ 2 * factorialNoFinal(1) =2
            ↳ 1 * factorialNoFinal(0) =1
                ↳ Retorna 1
```

Cada nivell ha de guardar l'operació pendent fins que totes les crides recursives s'acabin, i després torna a calcular de tornada.

```
            factorialNoFinal(4)
-> 4 * (3 * (2 * (1 * 1))) -> retorna 24
```

Pila durant les crides:

factorialNoFinal(0) n = 0
factorialNoFinal(1) n = 1
factorialNoFinal(2) n = 2
factorialNoFinal(3) n = 3
factorialNoFinal(4) n = 4
main()

Pila de crides

A la pila hi ha tants entorns com crides hem fet a la funció `factorialNoFinal`. No es desapilen fins que no fem el camí de tornada. Per això la recursió no-final pot causar un desbordament de pila.

Tipus de recursivitat

Segons la posició de la crida: final vs. no-final

```
int factorialNoFinal(int n) {
    if (n == 0) {
        return 1;
    }
    return n * factorialNoFinal(n - 1);
}
```

L'arbre de crides és ramificat amb càlculs pendents:

```
factorialNoFinal(4) =24
↳ 4 * factorialNoFinal(3) =12
      ↳ 3 * factorialNoFinal(2)  =6
            ↳ 2 * factorialNoFinal(1)  =2
                  ↳ 1 * factorialNoFinal(0)  =1
                        ↳ Retorna 1
```

Cada nivell ha de guardar l'operació pendent fins que totes les crides recursives s'acabin, i després torna a calcular de tornada.

```
              factorialNoFinal(4)
-> 4 * (3 * (2 * (1 * 1))) -> retorna 24
```

Pila durant les crides:

main()

Pila de crides

A la pila hi ha tants entorns com crides hem fet a la funció `factorialNoFinal`. No es desapilen fins que no fem el camí de tornada. Per això la recursió no-final pot causar un desbordament de pila.

Tipus de recursivitat

Segons la posició de la crida: final vs. no-final

```
int factorialNoFinal(int n) {
    if (n == 0) {
        return 1;
    }
    return n * factorialNoFinal(n - 1);
}
```

L'arbre de crides és ramificat amb càlculs pendents:

```
factorialNoFinal(4) =24
 ↳ 4 * factorialNoFinal(3) =12
       ↳ 3 * factorialNoFinal(2) =6
             ↳ 2 * factorialNoFinal(1) =2
                   ↳ 1 * factorialNoFinal(0) =1
                         ↳ Retorna 1
```

Cada nivell ha de guardar l'operació pendent fins que totes les crides recursives s'acabin, i després torna a calcular de tornada.

```
factorialNoFinal(4)
-> 4 * (3 * (2 * (1 * 1))) -> retorna 24
```

Pila durant les crides:

A la pila hi ha tants entorns com crides hem fet a la funció `factorialNoFinal`. No es desapilen fins que no fem el camí de tornada. Per això la recursió no-final pot causar un desbordament de pila.

Tipus de recursivitat

Segons la posició de la crida: final vs. no-final

```
int factorialFinal(int n, int acumulador) {
    if (n == 0) {
       // Cas base: tornem el resultat acumulat
        return acumulador;
    }
    return factorialFinal(n - 1, n * acumulador);
}
```

Tipus de recursivitat

Segons la posició de la crida: final vs. no-final

```
int factorialFinal(int n, int acumulador) {
    if (n == 0) {
       // Cas base: tornem el resultat acumulat
        return acumulador;
    }
    return factorialFinal(n - 1, n * acumulador);
}
```

A l'**arbre de crides** no hi ha càlculs pendents, ja que el resultat s'acumula progressivament a l'acumulador.

```
factorialFinal(4, 1)
 ↳ factorialFinal(3, 4)
    ↳ factorialFinal(2, 12)
       ↳ factorialFinal(1, 24)
         ↳ factorialFinal(0, 24)
          (Cas base: retorna 24)
```

factorialFinal(0, 24) retorna directament 24 (cas base). Per tant, no cal que tornem a pujar a l'arbre. El càlcul es fa a la baixada.

Tipus de recursivitat

Segons la posició de la crida: final vs. no-final

```
int factorialFinal(int n, int acumulador) {
    if (n == 0) {
       // Cas base: tornem el resultat acumulat
        return acumulador;
    }
    return factorialFinal(n - 1, n * acumulador);
}
```

A l'**arbre de crides** no hi ha càlculs pendents, ja que el resultat s'acumula progressivament a l'acumulador.

```
factorialFinal(4, 1)
 ↳ factorialFinal(3, 4)
    ↳ factorialFinal(2, 12)
        ↳ factorialFinal(1, 24)
          ↳ factorialFinal(0, 24)
           (Cas base: retorna 24)
```

factorialFinal(0, 24) retorna directament 24 (cas base). Per tant, no cal que tornem a pujar a l'arbre. El càlcul es fa a la baixada.

Pila durant les crides:

main()

Pila de crides

Tipus de recursivitat

Segons la posició de la crida: final vs. no-final

```c
int factorialFinal(int n, int acumulador) {
    if (n == 0) {
        // Cas base: tornem el resultat acumulat
        return acumulador;
    }
    return factorialFinal(n - 1, n * acumulador);
}
```

A l'**arbre de crides** no hi ha càlculs pendents, ja que el resultat s'acumula progressivament a l'acumulador.

```
factorialFinal(4, 1)
 ↳ factorialFinal(3, 4)
    ↳ factorialFinal(2, 12)
       ↳ factorialFinal(1, 24)
         ↳ factorialFinal(0, 24)
          (Cas base: retorna 24)
```

factorialFinal(0, 24) retorna directament 24 (cas base). Per tant, no cal que tornem a pujar a l'arbre.

Pila durant les crides:

factorialFinal(4) n = 4, acumulador = 1
main()

Pila de crides

Al contrari que al cas de recursivitat no-final, fent servir certes optimitzacions, cada crida es resol completament i per tant, es desapila. Això fa que no poguem tenir un desbordament de pila.

Tipus de recursivitat

Segons la posició de la crida: final vs. no-final

```
int factorialFinal(int n, int acumulador) {
    if (n == 0) {
        // Cas base: tornem el resultat acumulat
         return acumulador;
    }
    return factorialFinal(n - 1, n * acumulador);
}
```

A l'**arbre de crides** no hi ha càlculs pendents, ja que el resultat s'acumula progressivament a l'acumulador.

```
factorialFinal(4, 1)
 ↳ factorialFinal(3, 4)
    ↳ factorialFinal(2, 12)
       ↳ factorialFinal(1, 24)
         ↳ factorialFinal(0, 24)
          (Cas base: retorna 24)
```

factorialFinal(0, 24) retorna directament 24 (cas base). Per tant, no cal que tornem a pujar a l'arbre.

Pila durant les crides:

factorialFinal(3) n = 3, acumulador = 4
main()

Pila de crides

Al contrari que al cas de recursivitat no-final, fent servir certes optimitzacions, cada crida es resol completament i per tant, es desapila. Això fa que no poguem tenir un desbordament de pila.

Tipus de recursivitat

Segons la posició de la crida: final vs. no-final

```
int factorialFinal(int n, int acumulador) {
    if (n == 0) {
       // Cas base: tornem el resultat acumulat
        return acumulador;
    }
    return factorialFinal(n - 1, n * acumulador);
}
```

A l'**arbre de crides** no hi ha càlculs pendents, ja que el resultat s'acumula progressivament a l'acumulador.

```
factorialFinal(4, 1)
 ↳ factorialFinal(3, 4)
    ↳ factorialFinal(2, 12)
       ↳ factorialFinal(1, 24)
         ↳ factorialFinal(0, 24)
          (Cas base: retorna 24)
```

factorialFinal(0, 24) retorna directament 24 (cas base). Per tant, no cal que tornem a pujar a l'arbre.

Pila durant les crides:

factorialFinal(2) n = 2, acumulador = 12
main()

Pila de crides

Al contrari que al cas de recursivitat no-final, fent servir certes optimitzacions, cada crida es resol completament i per tant, es desapila. Això fa que no poguem tenir un desbordament de pila.

Tipus de recursivitat

Segons la posició de la crida: final vs. no-final

```
int factorialFinal(int n, int acumulador) {
    if (n == 0) {
        // Cas base: tornem el resultat acumulat
         return acumulador;
    }
    return factorialFinal(n - 1, n * acumulador);
}
```

A l'**arbre de crides** no hi ha càlculs pendents, ja que el resultat s'acumula progressivament a l'acumulador.

```
factorialFinal(4, 1)
 ↳ factorialFinal(3, 4)
    ↳ factorialFinal(2, 12)
        ↳ factorialFinal(1, 24)
           ↳ factorialFinal(0, 24)
            (Cas base: retorna 24)
```

factorialFinal(0, 24) retorna directament 24 (cas base). Per tant, no cal que tornem a pujar a l'arbre.

Pila durant les crides:

factorialFinal(1) n = 1, acumulador = 24
main()

Pila de crides

Al contrari que al cas de recursivitat no-final, fent servir certes optimitzacions, cada crida es resol completament i per tant, es desapila. Això fa que no puguem tenir un desbordament de pila.

Tipus de recursivitat

Segons la posició de la crida: final vs. no-final

```
int factorialFinal(int n, int acumulador) {
    if (n == 0) {
        // Cas base: tornem el resultat acumulat
         return acumulador;
    }
    return factorialFinal(n - 1, n * acumulador);
}
```

A l'**arbre de crides** no hi ha càlculs pendents, ja que el resultat s'acumula progressivament a l'acumulador.

```
factorialFinal(4, 1)
 ↳ factorialFinal(3, 4)
    ↳ factorialFinal(2, 12)
       ↳ factorialFinal(1, 24)
         ↳ factorialFinal(0, 24)
          (Cas base: retorna 24)
```

factorialFinal(0, 24) retorna directament 24 (cas base). Per tant, no cal que tornem a pujar a l'arbre.

Pila durant les crides:

factorialFinal(0) n = 0, acumulador = 24
main()

Pila de crides

Al contrari que al cas de recursivitat no-final, fent servir certes optimitzacions, cada crida es resol completament i per tant, es desapila. Això fa que no poguem tenir un desbordament de pila.

Tipus de recursivitat

Segons la posició de la crida: final vs. no-final

COMPTE!

Això que acabem de dir que s'apila i es desapila és només vàlid si el compilador aplica una tècnica d'optimització anomenada *TCO (Tail Call Optimization)*

En realitat, no podem desapilar el frame anterior perquè tècnicament encara no ha acabat d'executar-se fins que no es resolgui la crida recursiva. El que passa és que el compilador detecta que el resultat de la crida recursiva és el resultat final i que després no ha de fer res més. Per tant, en lloc de guardar un nou frame a la pila i esperar la crida recursiva per després retornar-ne el resultat, el que fa és reutilitzar el frame actual per a la nova crida. Per això no provoca desbordaments de pila.

Perquè gcc faci aquesta optimització se l'ha de cridar passant-li l'opció `-O2`

Recursivitat final

```
int factorialFinal(int n, int acumulador) {
    if (n == 0) {
        // Cas base: tornem el resultat acumulat
         return acumulador;
    }
    return factorialFinal(n - 1, n * acumulador);
}
```

Pila durant les crides:

factorialFinal(0) n = 0, acumulador = 24
main()

Pila de crides

Al contrari que al cas de recursivitat no-final, fent servir certes optimitzacions, cada crida es resol completament i per tant, es desapila. Això fa que no poguem tenir un desbordament de pila.

Tipus de recursivitat

Segons la posició de la crida: final vs. no-final

- El preu a pagar per tenir una funció més eficient (recursivitat final) és la introducció d'un segon paràmetre: l'acumulador. Aquest paràmetre, però, no té sentit per a l'usuari, que no sabrà quin valor assignar-li. "acumulador" és en el fons, un detall d'implementació.

Quin valor hauríem de donar al paràmetre acumulador a la primera crida? L'usuari de la funció té perquè saber-ho, això?

```
int factorialFinal(int n, int acumulador) {
    if (n == 0) {
       // Cas base: tornem el resultat acumulat
        return acumulador;
    }
    return factorialFinal(n - 1, n * acumulador);
}
```

Tipus de recursivitat

Segons la posició de la crida: final vs. no-final

- El preu a pagar per tenir una funció més eficient (recursivitat final) és la introducció d'un segon paràmetre: l'acumulador. Aquest paràmetre, però, no té sentit per a l'usuari, que no sabrà quin valor assignar-li. "acumulador" és en el fons, un detall d'implementació.

  Quin valor hauríem de donar al paràmetre acumulador a la primera crida? L'usuari de la funció té perquè saber-ho, això?

- Per evitar aquesta complicació, és habitual utilitzar una funció "wrapper" que amagui aquest detall d'implementació i ofereixi una interfície més clara.

```
int factorialFinal(int n, int acumulador) {
    if (n == 0) {
       // Cas base: tornem el resultat acumulat
        return acumulador;
    }
    return factorialFinal(n - 1, n * acumulador);
}
```

```
int factorial(int n) {
     // Crida a la funció auxiliar amb
     // l'acumulador inicialitzat a 1
    return factorialFinal(n, 1);
}
```

Tipus de recursivitat

Com es redueixen les dades: substractiva vs. divisora

- En recursivitat substractiva, el problema es redueix restant una quantitat fixa o dinàmica a cada pas. *Exemple típic: el factorial, on el problema es redueix en una unitat (n-1) a cada crida.*

```
int esDivisible(int a, int b) {
    if (a < b) {
        return (a == 0);
    }
    return esDivisible(a - b, b);
}
```

Tipus de recursivitat

Com es redueixen les dades: substractiva vs. divisora

- En recursivitat substractiva, el problema es redueix restant una quantitat fixa o dinàmica a cada pas. *Exemple típic: el factorial, on el problema es redueix en una unitat (n-1) a cada crida.*

```
int esDivisible(int a, int b) {
    if (a < b) {
        return (a == 0);
    }
   return esDivisible(a - b, b);
}
```

- En recursivitat divisora, el problema es redueix dividint-lo en una fracció o en parts més petites. *Exemple típic: l'algorisme de cerca binària, que divideix l'interval per la meitat a cada pas. O també l'algorisme que compta quants dígits binaris té un nombre decimal.*

```
int comptarDigitsBinaris(int n) {
    if (n == 0) {
        return 0;
    }
   return 1 + comptarDigitsBinaris(n / 2);
}
```

Tipus de recursivitat

Com es redueixen les dades: substractiva vs. divisora

- En recursivitat substractiva, el problema es redueix restant una quantitat fixa o dinàmica a cada pas. *Exemple típic: el factorial, on el problema es redueix en una unitat (n-1) a cada crida.*

```
int esDivisible(int a, int b) {
    if (a < b) {
        return (a == 0);
    }
   return esDivisible(a - b, b);
}
```

- En recursivitat divisora, el problema es redueix dividint-lo en una fracció o en parts més petites. *Exemple típic: l'algorisme de cerca binària, que divideix l'interval per la meitat a cada pas. O també l'algorisme que compta quants dígits binaris té un nombre decimal.*

```
int comptarDigitsBinaris(int n) {
    if (n == 0) {
        return 0;
    }
   return 1 + comptarDigitsBinaris(n / 2);
}
```

La importància d'aquesta distinció rau en el fet que el cost computacional dels algorismes subtractius i divisors és diferent. Els algorismes subtractius redueixen el conjunt de dades de manera lineal (una quantitat fixa en cada pas), mentre que els divisors ho fan de manera logarítmica (dividint el conjunt en parts més petites).
En quin dels dos reduirem més ràpid el conjunt de dades??? 🙄

Transformació de recursiu a iteratiu

- Tot algorisme recursiu es pot convertir a iteratiu, i viceversa.
- Alguns algorismes, s'escriuen de manera més natural recursivament i d'altres, iterativament.

Transformació de recursiu a iteratiu

- Tot algorisme recursiu es pot convertir a iteratiu, i viceversa.
- Alguns algorismes, s'escriuen de manera més natural recursivament i d'altres, iterativament.

Abans hem definit la funció factorial com a recursiva,
perquè aquesta definició així ens ho suggeria:

$$n! = \begin{cases} 1 & \text{si } n = 0 \\ n \cdot (n-1)! & \text{si } n > 0 \end{cases}$$

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

Transformació de recursiu a iteratiu

- Tot algorisme recursiu es pot convertir a iteratiu, i viceversa.
- Alguns algorismes, s'escriuen de manera més natural recursivament i d'altres, iterativament.

Abans hem definit la funció factorial com a recursiva, perquè aquesta definició així ens ho suggeria:

$$n! = \begin{cases} 1 & \text{si } n = 0 \\ n \cdot (n-1)! & \text{si } n > 0 \end{cases}$$

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

Però també hauríem pogut definir la funció factorial com a iterativa. Aquesta definició ens ho suggereix així:

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

```
int factorial(int n)
{
    int valor = 1;
    for(int i = 2; i <= n; i++){
      valor = valor * i;
    }
    return valor;
}
```

Transformació de recursiu a iteratiu

Amb els algorismes recursius hi ha certs costos associats (pila de crides), per tant és útil aprendre a transformar algorismes recursius a iteratius.

- Tot algorisme recursiu es pot convertir a iteratiu, i viceversa.
- Alguns algorismes, s'escriuen de manera més natural recursivament i d'altres, iterativament.

Abans hem definit la funció factorial com a recursiva, perquè aquesta definició així ens ho suggeria:

$$n! = \begin{cases} 1 & \text{si } n = 0 \\ n \cdot (n-1)! & \text{si } n > 0 \end{cases}$$

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

Però també hauríem pogut definir la funció factorial com a iterativa. Aquesta definició ens ho suggereix així:

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

```
int factorial(int n)
{
    int valor = 1;
    for(int i = 2; i <= n; i++){
      valor = valor * i;
    }
    return valor;
}
```

Transformació de recursiu a iteratiu

- Els algorismes amb recursivitat simple es poden transformar a iteratius de manera mecànica:

Transformació de recursiu a iteratiu

- Els algorismes amb recursivitat simple es poden transformar a iteratius de manera mecànica:

Esquema d'algorisme amb recusivitat simple

```
algorisme alg_recursiu (x) és
  si B(x) llavors
     S;
  sino
     alg_recursiu(T(x));
     S';
  fsi
falgorisme
```

Transformació de recursiu a iteratiu

- Els algorismes amb recursivitat simple es poden transformar a iteratius de manera mecànica:

Esquema d'algorisme amb recusivitat simple

```
algorisme alg_recursiu (x) és
  si B(x) llavors
     S;
  sino
     alg_recursiu(T(x));
     S';
  fsi
falgorisme
```

On:

- $B(x)$ és la condició que avalua si estem al cas base
- $T(x)$ és la transformació dels paràmetres d'entrada que redueix el problema
- S és el conjunt de sentències a executar al cas base
- S' són les sentències a executar en el cas recursiu

Transformació de recursiu a iteratiu

- Els algorismes amb recursivitat simple es poden transformar a iteratius de manera mecànica:

Esquema d'algorisme amb recusivitat simple

```
algorisme alg_recursiu (x) és
  si B(x) llavors
     S;
  sino
     alg_recursiu(T(x));
     S';
  fsi
falgorisme
```

Per exemple, en el cas del **factorial**:

```
int factorial(int n)
{           B(x)
    if (n == 0){
       return 1;  S
    }
    else {                S'       T(x)
       return n * factorial(n-1);
    }
}
```

On:

- $B(x)$ és la condició que avalua si estem al cas base
- $T(x)$ és la transformació dels paràmetres d'entrada que redueix el problema
- S és el conjunt de sentències a executar al cas base
- S' són les sentències a executar en el cas recursiu

Transformació de recursiu a iteratiu

- Els algorismes amb recursivitat simple es poden transformar a iteratius de manera mecànica:

Esquema d'algorisme amb recusivitat simple

```
algorisme alg_recursiu (x) és
  si B(x) llavors
     S;
  sino
     alg_recursiu(T(x));
     S';
  fsi
falgorisme
```

Transformació de l'esquema anterior a iteratiu:

```
algorisme alg_iteratiu (x) és
   S;
   mentre ( no B(x) ) fer
     T(x);
     S';
   fmentre
falgorisme
```

On:

- $B(x)$ és la condició que avalua si estem al cas base
- $T(x)$ és la transformació dels paràmetres d'entrada que redueix el problema
- S és el conjunt de sentències a executar al cas base
- S' són les sentències a executar en el cas recursiu

Transformació de recursiu a iteratiu

- Els algorismes amb recursivitat simple es poden transformar a iteratius de manera mecànica:

Transformació de l'esquema anterior a iteratiu:

```
algorisme alg_iteratiu (x) és
   S;
   mentre ( no B(x) ) fer
     T(x);
     S';
   fmentre
falgorisme
```

```
int factorial(int n)
{          B(x)
   if (n == 0){
      return 1;  S
   }
   else {                S'        T(x)
      return n * factorial(n-1);
   }
}
```

Aplicant aquest esquema al factorial:

```
algorisme fact_iter (n) és
   i := n;
   result := 1;
   mentre ( i != 0) fer
      result = result * i;
      i = i - 1;
   fmentre
   retorna result;
falgorisme
```

Dubtes?

Al laboratori dissenyarem i implementarem diversos algorismes recursius simples.