Récursivité

ICC-C Cours 13

scanf

- On peut passer un nombre variable d'arguments à scanf
- Comment?

```
int scanf(const char * fmt, ...)
```

- C'est une fonction **variadique** (= nombre variable de paramètres)
- Il faut au moins un paramètre "nommé" (ici `fmt`)

Comment lit-on les arguments variadiques?

```c
#include <stdarg.h>

int sum(int first, ...)
{
    va_list args;
    int total = first;

    va_start(args, first);
    int next;
    while ((next = va_arg(args, int)) != 0)
    {
        total += next;
    }
    va_end(args);

    return total;
}
```

On utilise ce type spécial pour obtenir miraculeusement la liste des arguments

...ensemble avec cet appel

Ici on s'arrête à `0`

On obtient l'argument suivant avec la "macro" `va_arg` qui prend la liste `args` et le **type** du prochain argument à lire

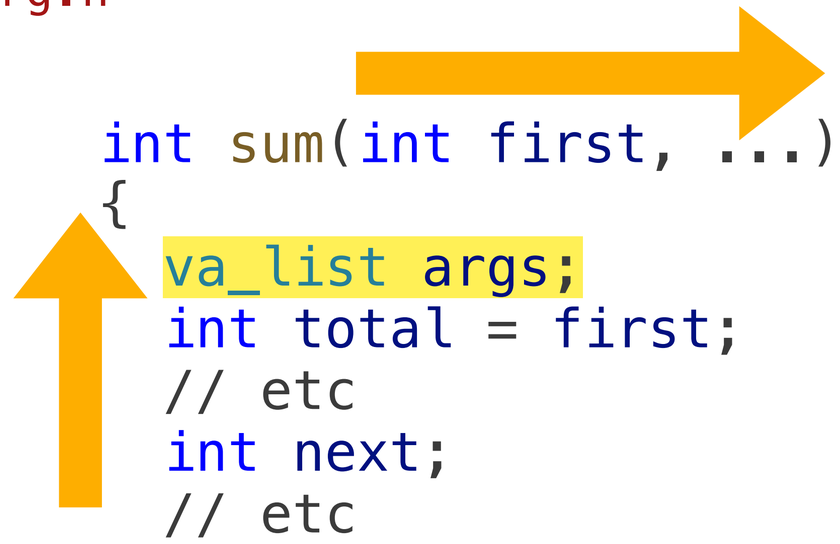
Enfin, on doit indiquer qu'on a fini de lire

Sous le capot…

Ordre (approx) dans la mémoire

- L'idée est que les paramètres se suivent en mémoire
  - L'implémentation dépend du compilateur — utiliser `stdarg.h`
  - Pas de mécanisme pour indiquer combien il y en a — il faut le prévoir
  - Dans le code on suppose un marqueur de fin `0`:

```
sum(1, 2, 3, 4, 0)
```



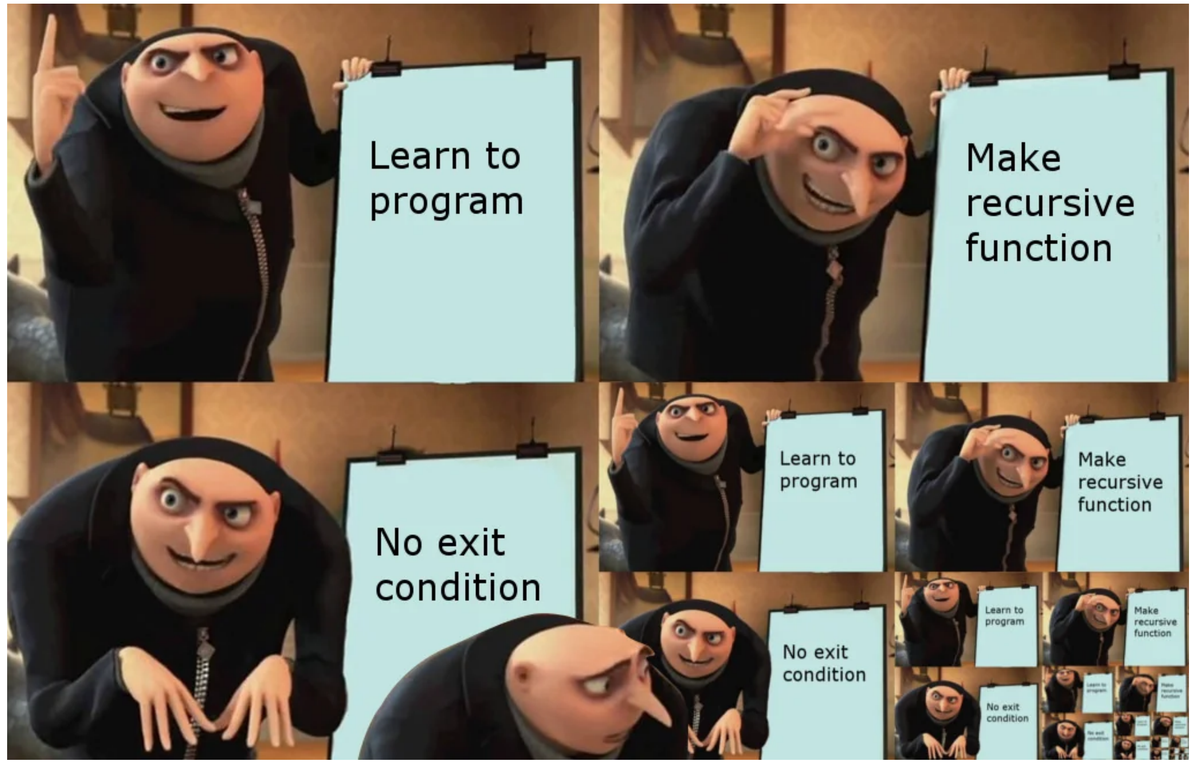
- Appel de fonction = "push" sur la pile d'exécution
- Un nouveau "**stack frame**" est crée
  = mémoire nécessaire pour stocker les arguments + variables locales

Appels de fonctions

- La pile d’exécution stocke les variables locales
  dans des “**stack frames**” (=trames de pile 😬)
- Les **stack frames** qui forment la pile correspondent à des appels imbriqués
- Une fonction peut s’appeler soi-même = une fonction récursive

Comment écrit-on une fonction récursive?

- **But:** résoudre un problème pour une entrée donnée = une instance de taille *N*
- La solution = une composition d'instances de plus petite taille
- Le cas de base (la plus petite instance, e.g., *N* = 1) est facile
  - C'est ainsi qu'on sort de la fonction!
- On reconstruit la solution de l'instance



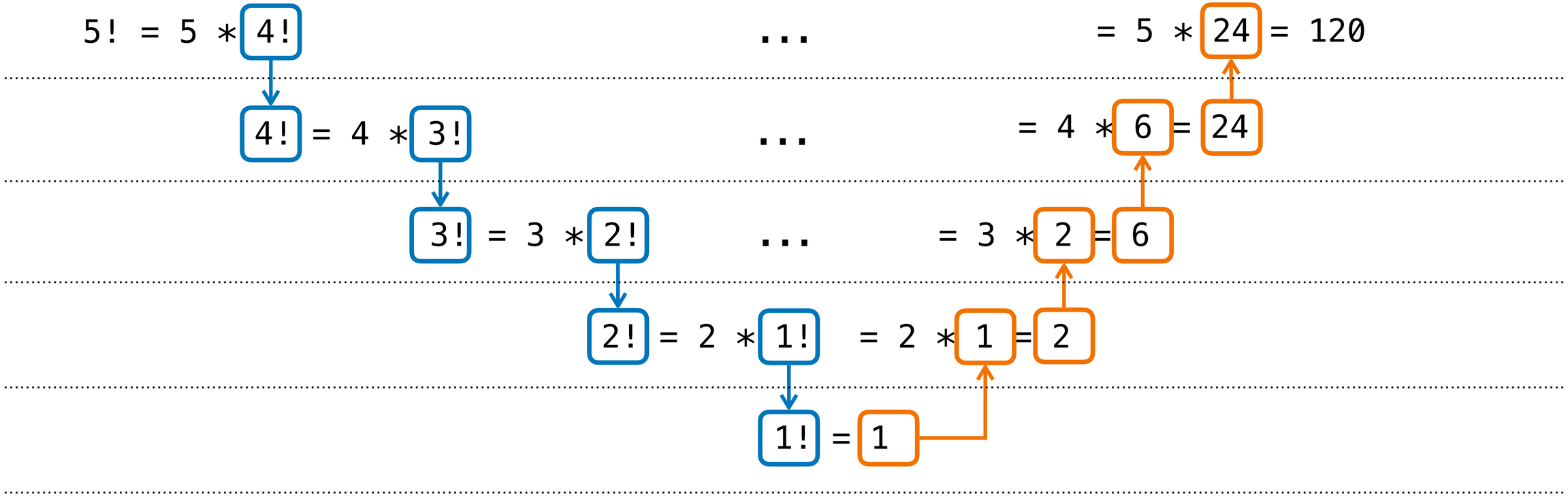
Exemple 1

Factorielle

- **Problème**
  Calculer la factorielle d'un nombre: `n!`
- **Taille**
  `n`
- **Cas de base**
  `1! = 1`
- **Relation récursive**
  `n! = n * (n-1)!`

Exemple 1

Factorielle



Implémentation

Factorielle

```
long fact(long n)
{
    if (n == 1)
    {
        return 1;
    }

    return n * fact(n-1);
}
```

Le cas de base

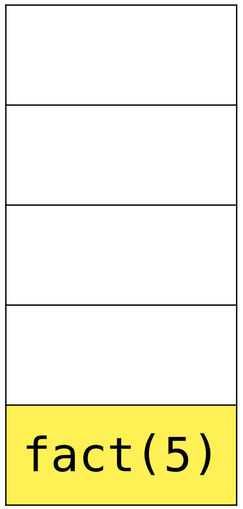
Relation de récurrence

Implémentation

Les appels

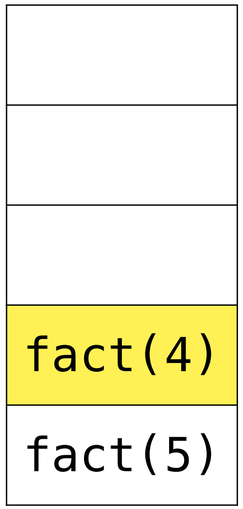
```
long fact(long n)
{
    if (n == 1)
    {
        return 1;
    }

    return n * fact(n-1);
}
```

`long n = 5`



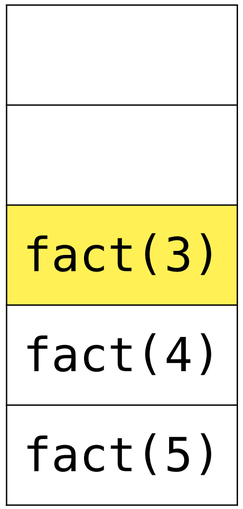
`5 * fact(4);`

`long n = 4`



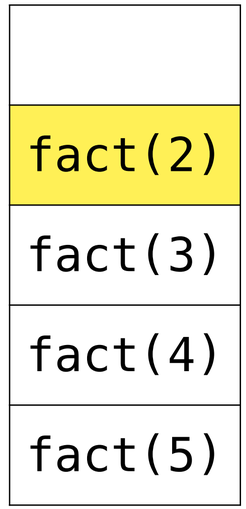
`4 * fact(3);`

`long n = 3`



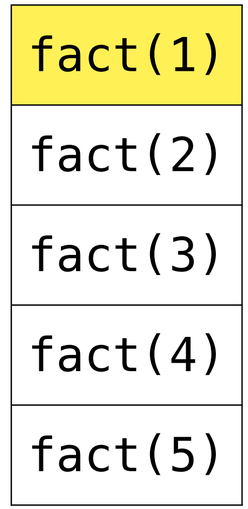
`3 * fact(2);`

`long n = 2`



`2 * fact(1);`

`long n = 1`



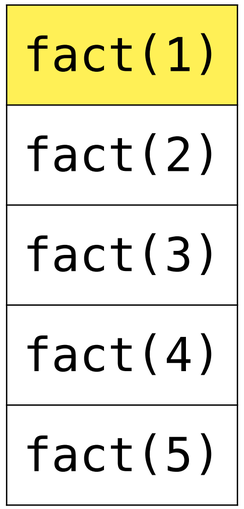
`return 1;`

Implémentation

Les retours

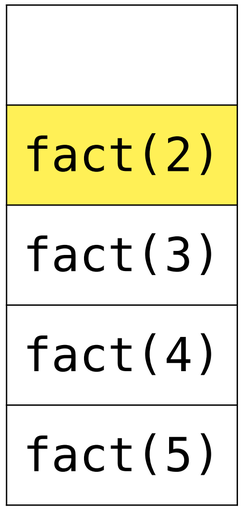
```
long fact(long n)
{
    if (n == 1)
    {
        return 1;
    }

    return n * fact(n-1);
}
```

`long n = 1`



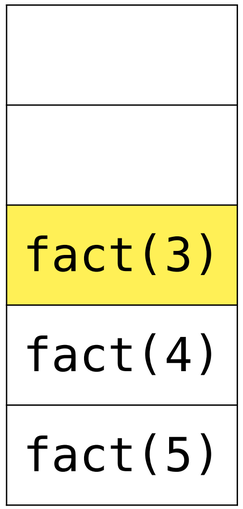
`return 1;`

`long n = 2`



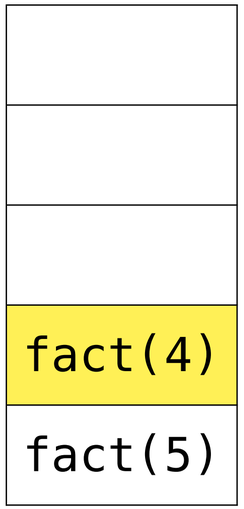
`return 2 * 1;`

`long n = 3`



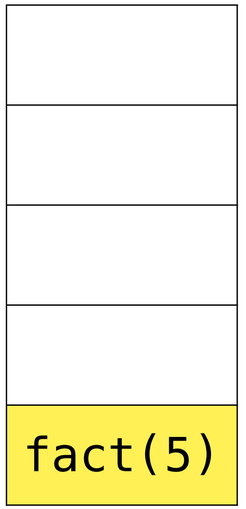
`return 3 * 2;`

`long n = 4`



`return 4 * 6;`

`long n = 5`



`return 5 * 24;`

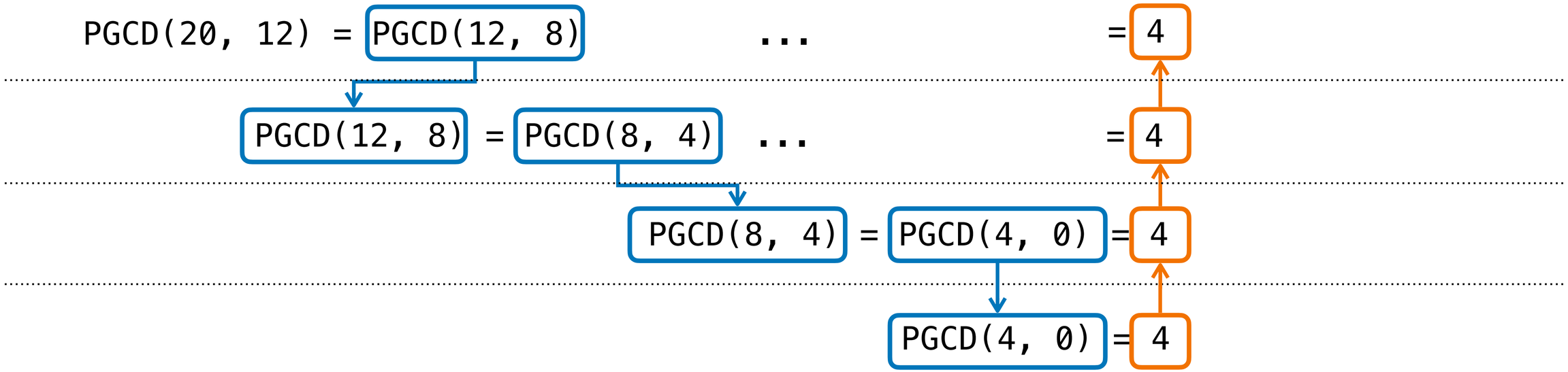
Exemple 2

Euclide

- **Problème**
  Calculer le plus grand diviseur commun `PGCD(a, b)`
- **Taille**
  `b`
- **Cas de base**
  `PGCD(a, 0) = a`
- **Relation**
  `PGCD(a, b) = PGCD(b, a % b)`

Exemple 2

Euclide



Implémentation

Euclide

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

Le cas de base

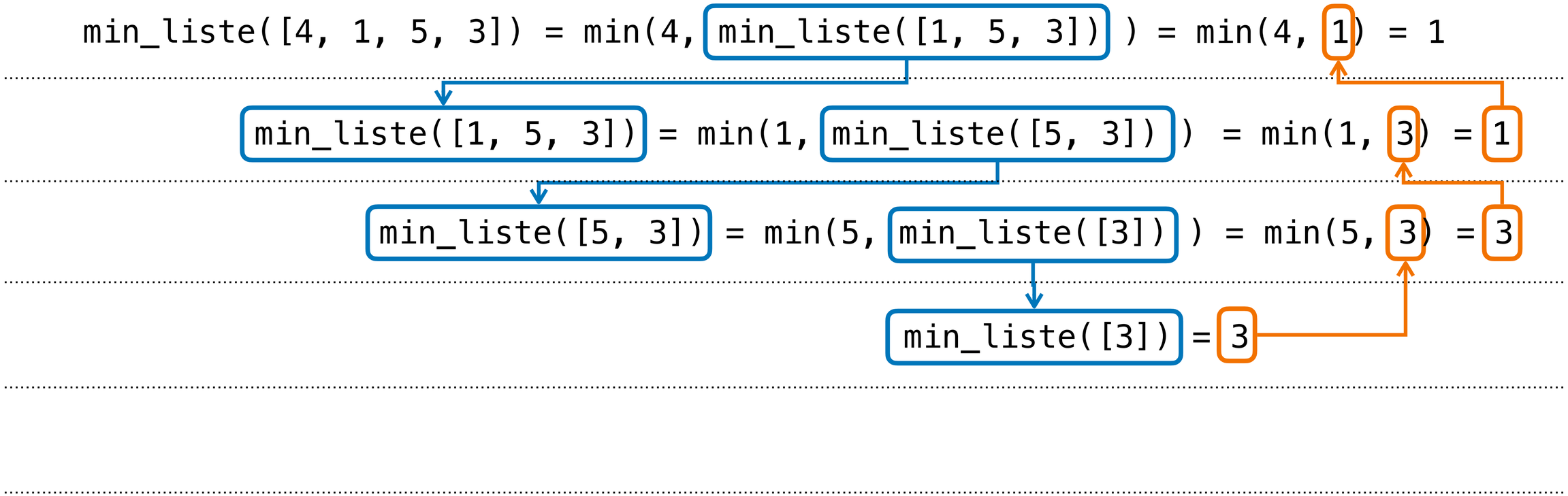
Relation de récurrence

Exemple 3
Minimum

- **Problème**
  Trouver le plus petit élément dans une liste `min_liste(L)`
- **Taille**
  La taille de la liste
- **Cas de base** (liste de taille 1)
  `L->contenu`
- **Relation**
  `min_liste(L) = min(L->contenu, min_liste(L->next))`

Exemple 3

Minimum



Implémentation

Minimum

```
int min_liste(cell_t *c)
{
    if (c->next == NULL)
    {
        return c->contenu;
    }
    return min(c->contenu, min_liste(c->next));
}
```

Le cas de base

Relation de récurrence

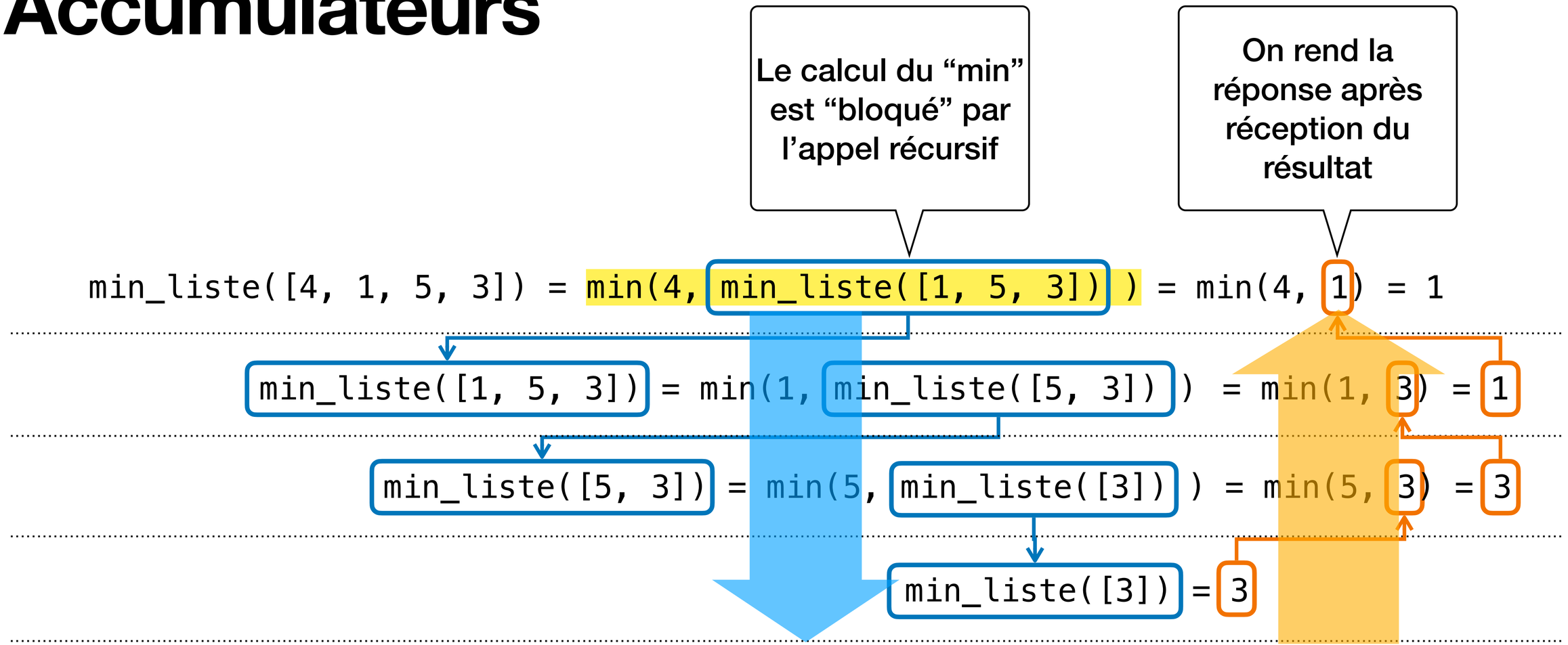
Pourquoi?

- La définition d'une fonction récursive est plus proche de la formulation mathématique, donc plus proche du langage naturel
- La récursivité ouvre le chemin vers la programmation déclarative
  - On décrit ce que le programme doit accomplir, **sans détailler la procédure**
  - E.g., langages fonctionnels (Scala, Haskell, F#), langages logiques (Prolog)
- Le C est un langage souvent utilisé pour la programmation impérative
  - On liste chaque commande à exécuter par l'ordinateur "à l'impératif"

Désavantages

- Chaque appel récursif est stocké sur la pile d'exécution
- `min_liste` pour une liste de taille N utilise N entrées sur la pile
- Quand la pile d'exécution est remplie, l'appel de trop déclenchera une erreur "débordement de pile" = *stack overflow*
- **Remède**: simuler la pile d'exécution et traduire en algorithme itératif = algorithme sans appels récursifs
  - C'est possible, mais laborieux…

Accumulateurs



Accumulateurs

- On peut aussi calculer un résultat intermédiaire = un accumulateur
- On le stocke dans les paramètres!

```
int min_liste_tr(cell_t *c, int min_acc)
{
    if (c == NULL)
    {
        return min_acc;
    }
    // calculons le min jusqu'ici
    int new_min_acc = min(min_acc, c->contenu);

    return min_liste_tr(c->next, new_min_acc);
}
```

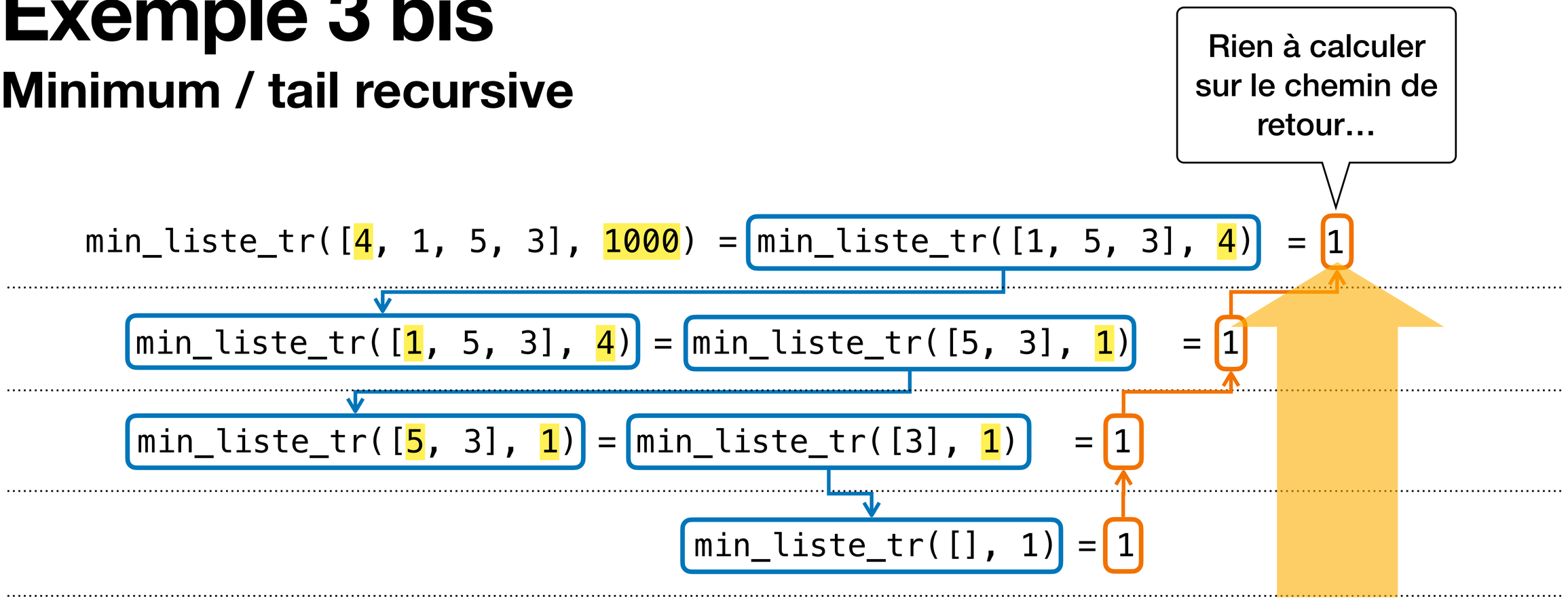
`min_acc` = accumulateur, car il accumule les résultats intermédiaires

en arrivant au cas de base on a déjà la réponse finale dans `min_acc`!

Ceci est un cas particulier de récursivité = récursivité terminale = *tail recursion*

Exemple 3 bis

Minimum / tail recursive



Un compilateur intelligent pourrait reformuler le code en boucle itérative!

Conversion tail-recursive vers itératif

```c
int min_liste_tr(cell_t *c,
                 int min_acc)
{
  if (c == NULL)
  {
    return min_acc;
  }
  // calculons le min jusqu'ici
  int new_min_acc = min(min_acc,
                        c->contenu);

  return min_liste_tr(c->next,
                      new_min_acc);
}
```

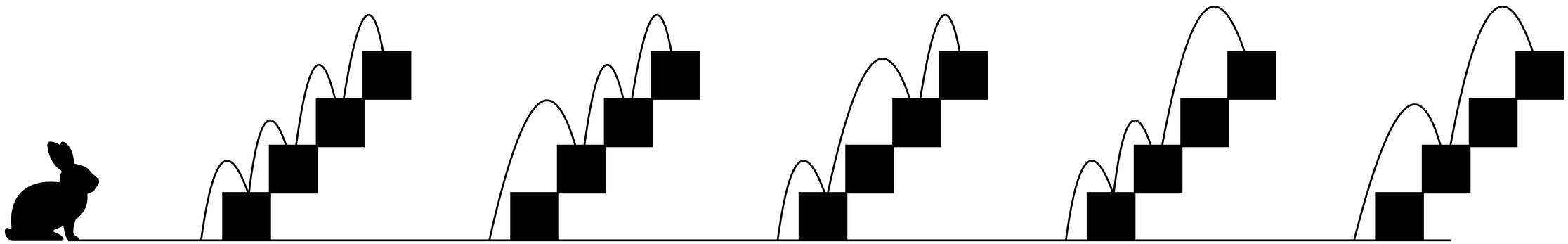
```c
int min_liste_it(cell_t *c)
{
  int min_acc;
  while (1)
  {
    if (c == NULL)
    {
      return min_acc;
    }
    // calculons le min jusqu'ici
    int new_min_acc = min(min_acc,
                          c->contenu);

    c = c->next;
    min_acc = new_min_acc;
  }
}
```

Simuler le passage des paramètres

Exemple 4

- Un lapin monte un escalier
- Il peut sauter une marche ou deux marches
- Combien de possibilités pour arriver en haut de l’escalier à N marches?

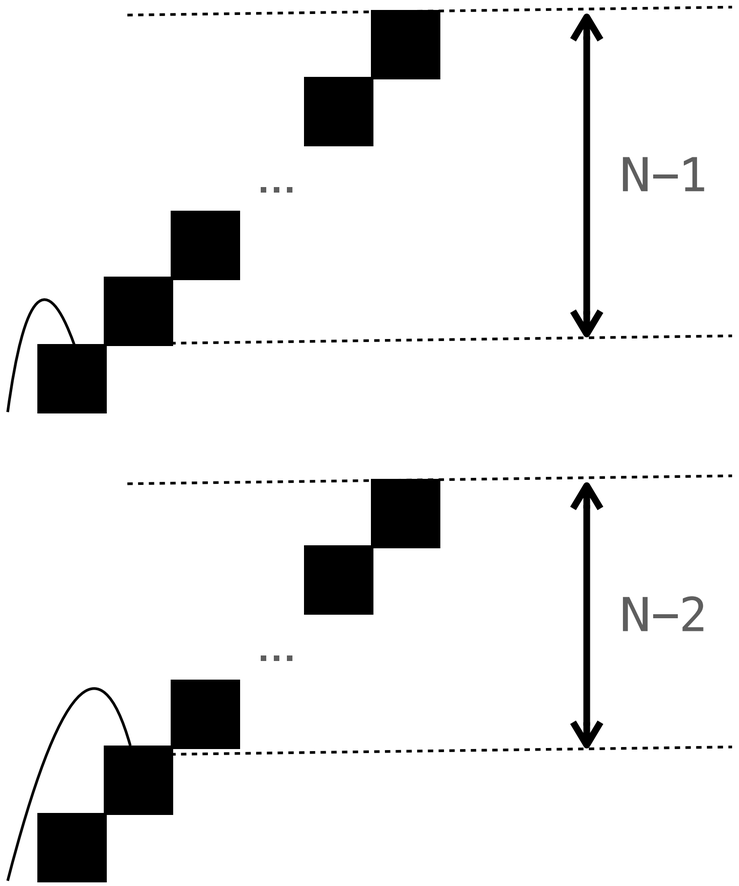
N = 4 marches



Solution

- On note `L(N)` le nombre de possibilités pour arriver en haut d'un escalier à N marches
- Soit le lapin saute une marche et se retrouve avec `L(N-1)` possibilités
- Soit le lapin saute deux marches et se retrouve avec `L(N-2)` possibilités
- `L(N) = L(N-1) + L(N-2)`
- `L(1) = 1, L(2) = 2`

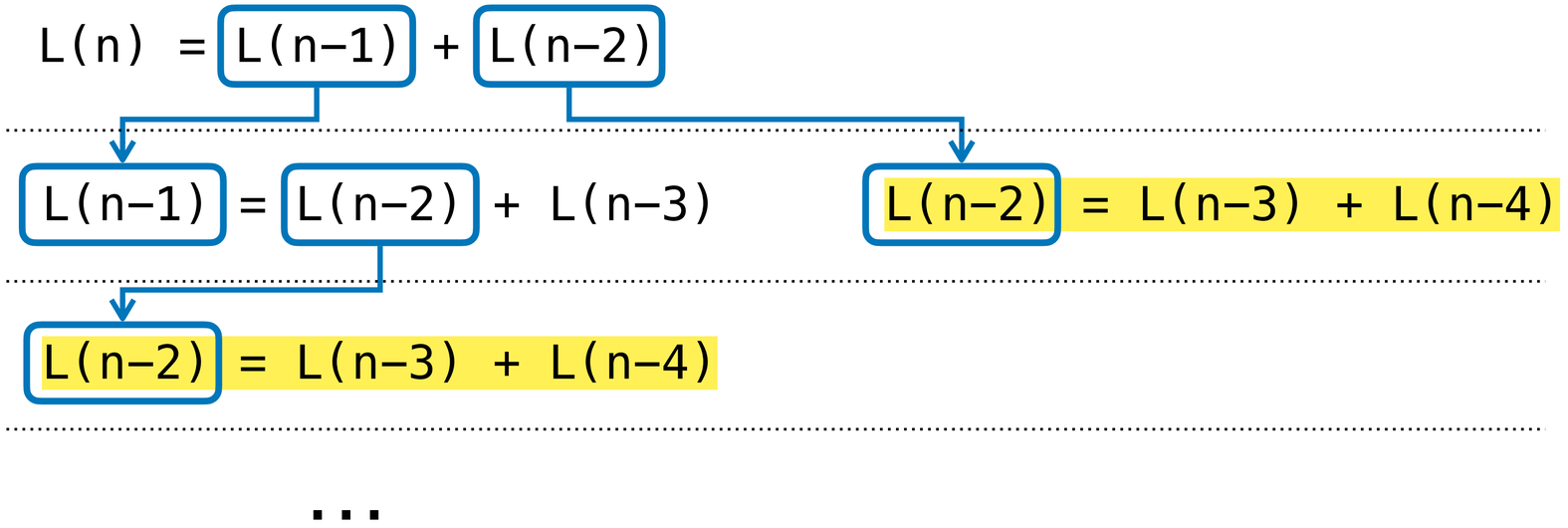
(~suite de Fibonacci)



Implémentation

… naïve

```
long L(int n)
{
    if (n == 1)
    {
        return 1;
    }
    if (n == 2)
    {
        return 2;
    }

    return L(n-1) + L(n-2);
}
```



- L’appel pour calculer `L(n-2)` est effectué **2 fois**…
- Est-ce tellement mauvais?
- **Complexité exponentielle** `O(2^n)` 😬 🐌

Intuition

- $T(n)$ = Nombre d'opérations (additions) pour calculer L(n)
- On calcule L(n−1), L(n−2), et une addition, donc
  $T(n) = T(n-1) + T(n-2) + 1$
- Donc une borne inférieure du nombre d'opérations est
  $T(n) \geq 2 \cdot T(n-2)$
- $T(n) \geq 2 \cdot T(n-2) \geq 4 \cdot T(n-4) \geq \ldots \geq 2^{\lfloor \frac{n}{2} \rfloor} T(n - 2\lfloor n/2 \rfloor)$
- Donc $T(n) \geq 2^{\frac{n}{2}}$ ce qui constitue une croissance exponentielle du nombre d'opérations avec n

Mémoïsation

= sauvegarder les résultats intermédiaires

```
long mem[100]; // initialisé à zéro avant utilisation
long L_mem(int n)
{
    if (n == 1)
    {
        return 1;
    }
    if (n == 2)
    {
        return 2;
    }

    if (mem[n] == 0)
    {
        mem[n] = L_mem(n - 1) + L_mem(n - 2);
    }
    return mem[n];
}
```

mem[n] est zéro ssi L(n) pas encore calculé

- On sauvegarde la valeur de L(n) dans le tableau mem à l'indice n
- Retrouver L(n)
  - La première fois mem[n] est 0, donc on lance le calcul récursif
  - La prochaine fois qu'on veut L(n), sa valeur (non-nulle) sera dans mem[n] et on la retourne
  - Fini les calculs redondants!