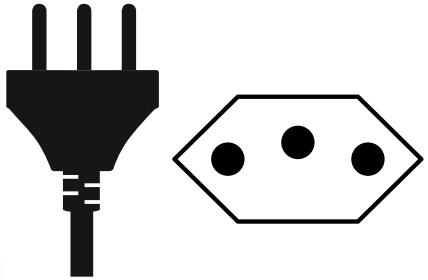
Information, Calcul et Communication

CS-119(g) ICC – Théorie
Semaine 11

Rafael Pires
rafael.pires@epfl.ch

Lausanne, 21.11.2025

Précédemment, dans ICC-T 10



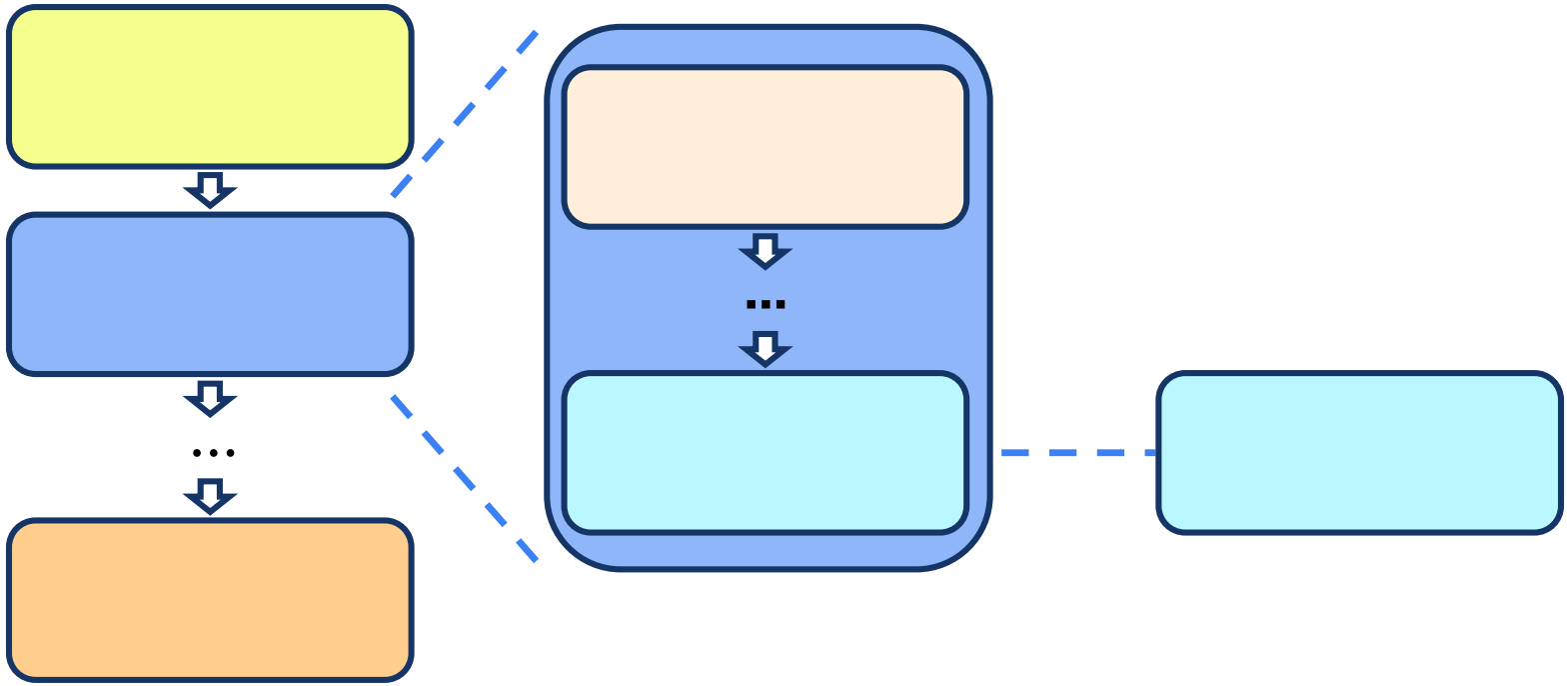
- Les sous-algorithmes
  - Problème : Tri par insertion
- La complexité temporelle
  - nombre d'opérations élémentaires dans le pire des cas
- La notation Grand Theta Θ
  - Problème : Deux font la paire

Aujourd'hui

- La récursivité
- Complexité logarithmique
- Tri par fusion

La légende des tours de Hanoï

Conception d’algorithmes : Diviser pour mieux régner



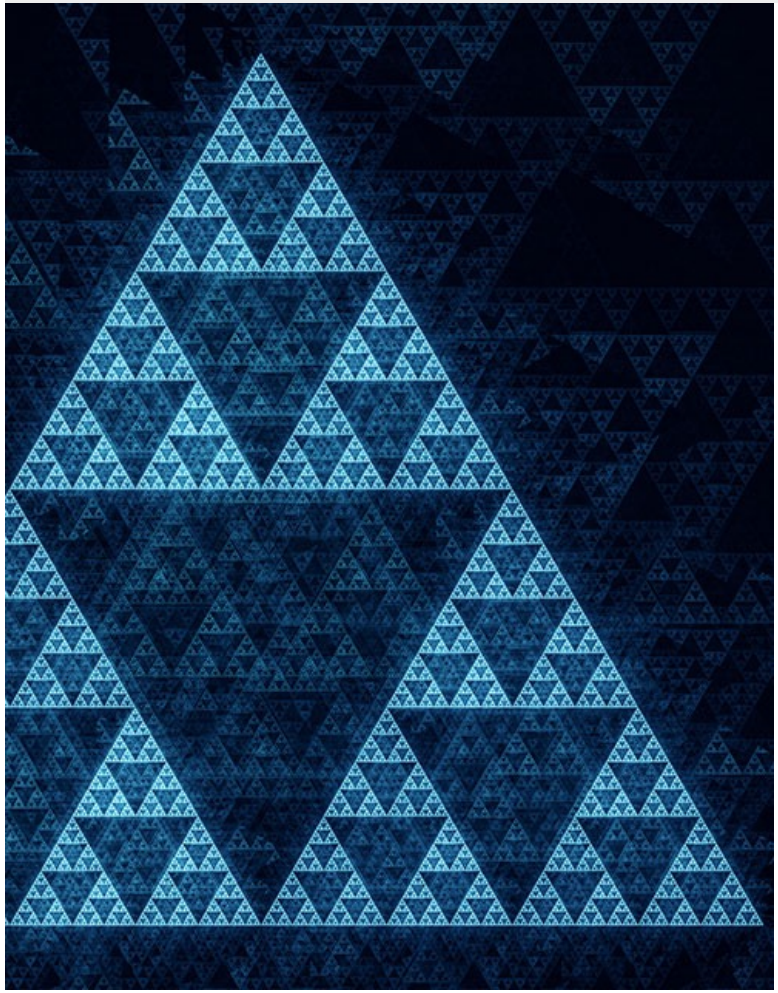
Récursivité

Problème entier

Sous-problèmes

Cas trivial

Récursivité vs. récurrence



- ***La Factorielle :***

$$n! = n * (n-1)!$$
$$0! = 1$$

- ***Suite de Fibonacci :***

$$F(n) = F(n-1) + F(n-2)$$
$$F(0) = F(1) = 1$$

- ***Les coefficients binomiaux :***

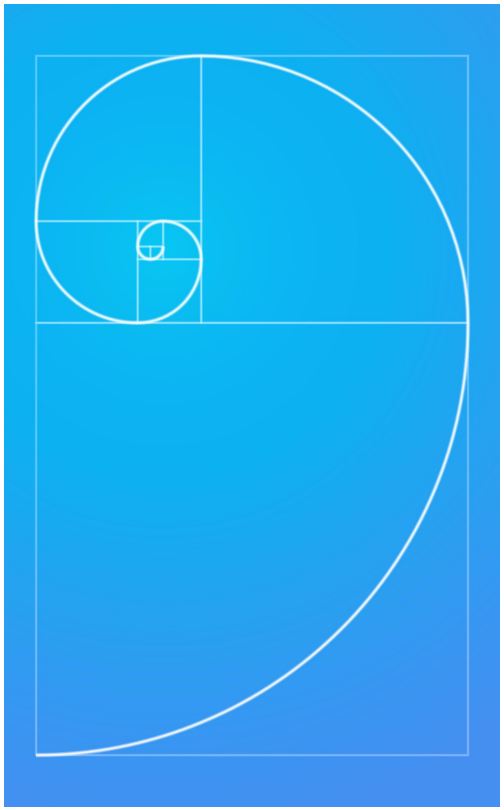
$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

$$\binom{n}{0} = \binom{n}{n} = 1$$

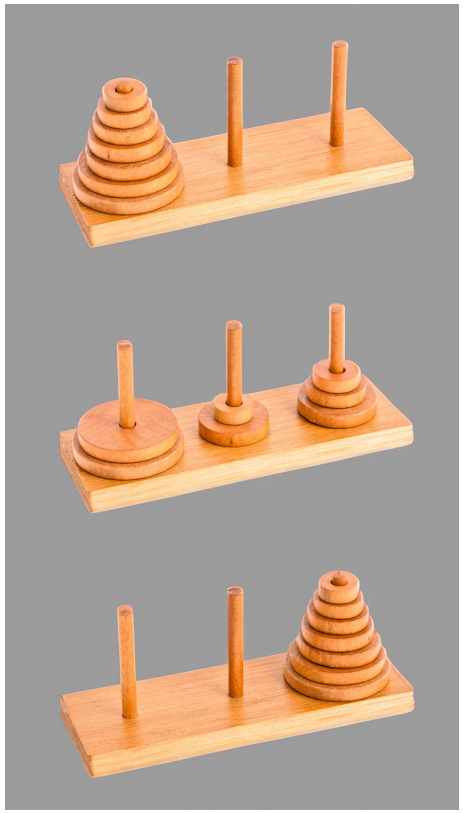
Problèmes

La Factorielle

La suite de Fibonacci



Les tours de Hanoï



Problème : Factorielle

La factorielle d'un nombre entier n.

$$n! = n * (n-1) * \cdots * 2 * 1$$

$$n! = n * (n-1)!$$

Attention

Un algorithme récursif doit toujours avoir une condition de terminaison.

Problème : Factorielle

La factorielle d'un nombre entier n.

$$n! = n * (n-1) * \cdots * 2 * 1$$

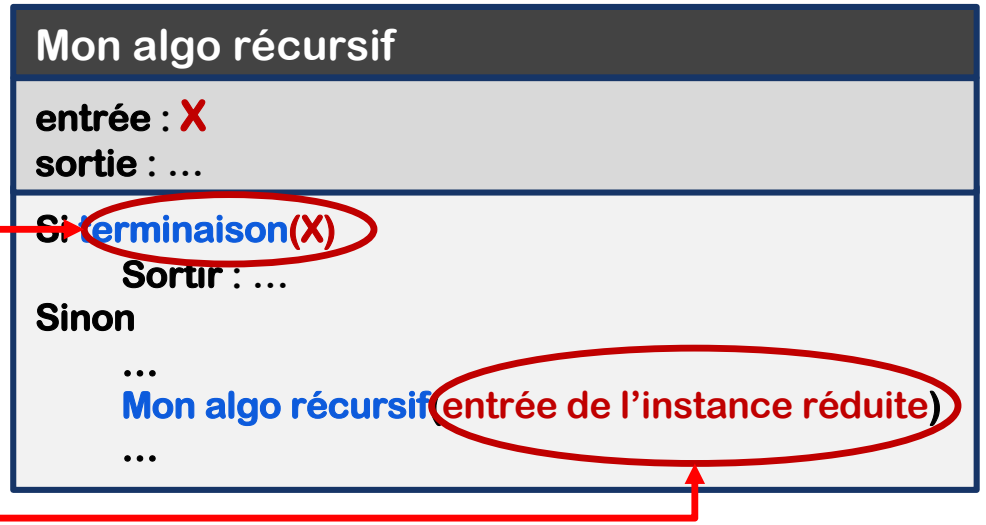
$$n! = n * (n-1)! \text{ et } 0! = 1$$

Comment implémenter la récursivité

⚠ Un algorithme récursif doit avoir une **condition de terminaison**.

⚠ Un algorithme récursif fait appel à lui-même avec une **instance plus simple** du problème original.

- Parfois, il faut **recombiner** les résultats d'appels récursifs pour former la solution au problème de départ.



Exécution : Factorielle

Factorielle

entrée : entier naturel **n**
sortie : n!

```
Si n <= 1
        Sortir : 1
Sinon
        Sortir : n * Factorielle(n-1)
```

Pile d'exécution

Factorielle(*n* = 3)
Sortir : **6**

Factorielle(*n* = 3)

```
Si n <= 1
        Sortir : 1
Sinon
        Sortir : 3 * 2
```

Factorielle(*n* = 2)

```
Si n <= 1
        Sortir : 1
Sinon
        Sortir : 2 * 1
```

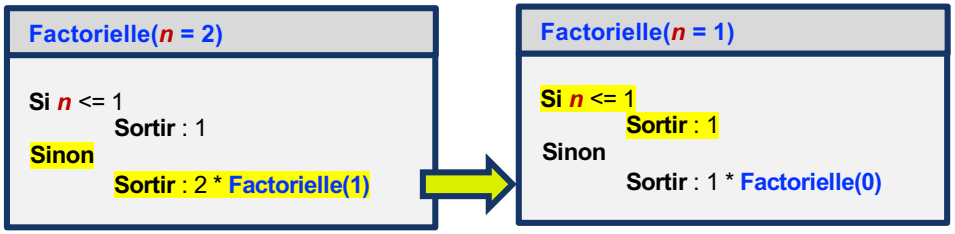
Factorielle(*n* = 1)

```
Si n <= 1
        Sortir : 1
Sinon
        Sortir : 1 * Factorielle(0)
```

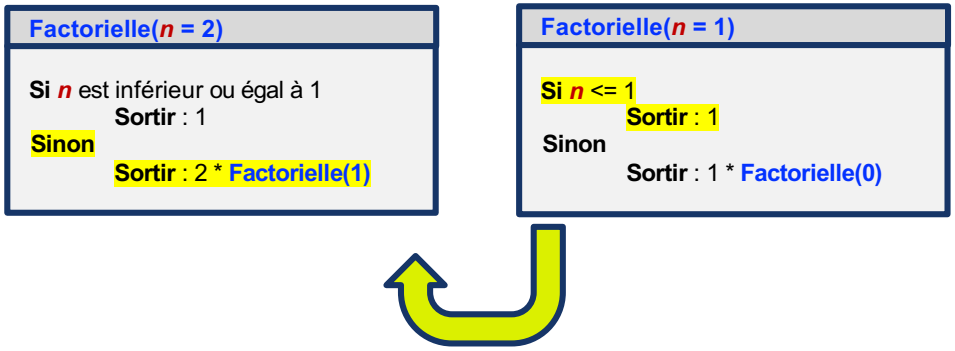
Réponse : 6

Quelques observations

- chaque appel récursif à une fonction crée **son propre contexte**



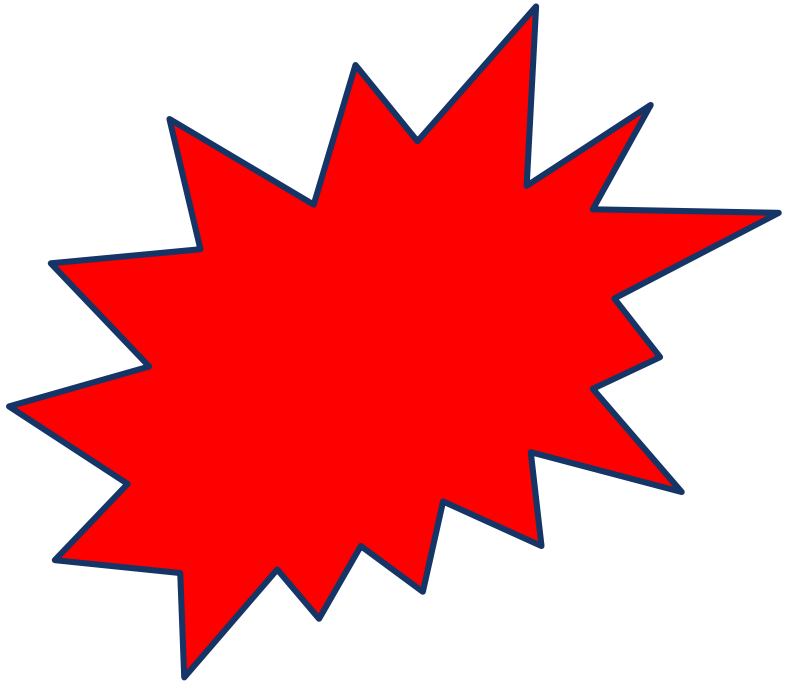
- le flux de contrôle revient au **contexte précédent** une fois que l'appel de fonction se termine (et éventuellement renvoie une valeur)



Exécution : Débordement de la pile

Factorielle(*n*)

Sortir : *n* * Factorielle(*n*-1)



Factorielle(*n* = -7)
Sortir : -7 * Factorielle(-8)

Factorielle(*n* = -6)
Sortir : -6 * Factorielle(-7)

Factorielle(*n* = -5)
Sortir : -5 * Factorielle(-6)

Factorielle(*n* = -4)
Sortir : -4 * Factorielle(-5)

Factorielle(*n* = -3)
Sortir : -3 * Factorielle(-4)

Factorielle(*n* = -2)
Sortir : -2 * Factorielle(-3)

Factorielle(*n* = -1)
Sortir : -1 * Factorielle(-2)

Factorielle(*n* = 0)
Sortir : 0 * Factorielle(-1)

Factorielle(*n* = 1)
Sortir : 1 * Factorielle(0)

Factorielle(*n* = 2)
Sortir : 2 * Factorielle(1)

Factorielle(*n* = 3)
Sortir : 3 * Factorielle(2)

Attention

Un algorithme récursif doit toujours avoir une condition de terminaison.

Factorielle : récursive vs. itérative

Récursive :

Factorielle(*n*)

Si ***n*** est inférieur ou égal à 1
    Sortir : 1
Sinon
    Sortir : ***n*** * **Factorielle(*n*-1)**

Itérative :

Factorielle(*n*)

résultat ← 1
Tant que ***n*** > 1
    résultat ← ***résultat*** * ***n***
    n ← ***n*** -1

Sortir : ***résultat***

Complexité :

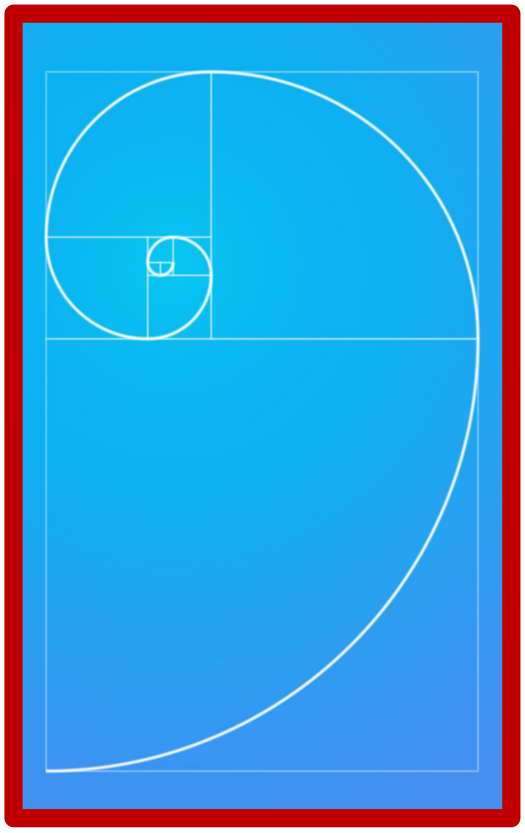
$\Theta(n)$

$\Theta(n)$

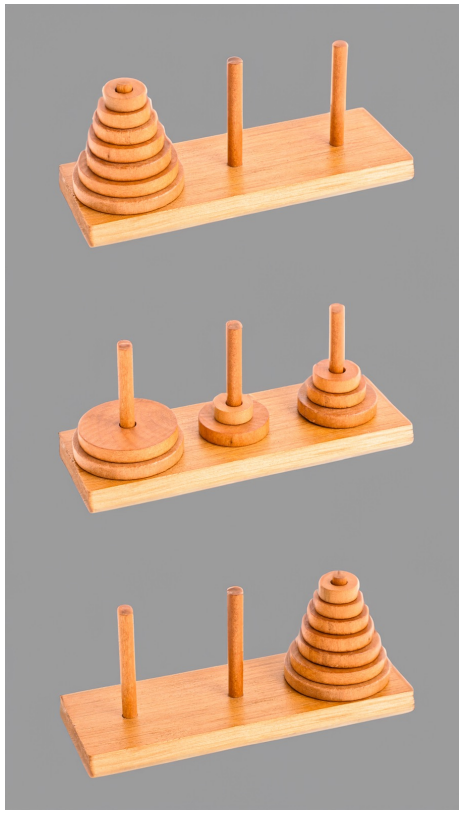
Problèmes

La Factorielle

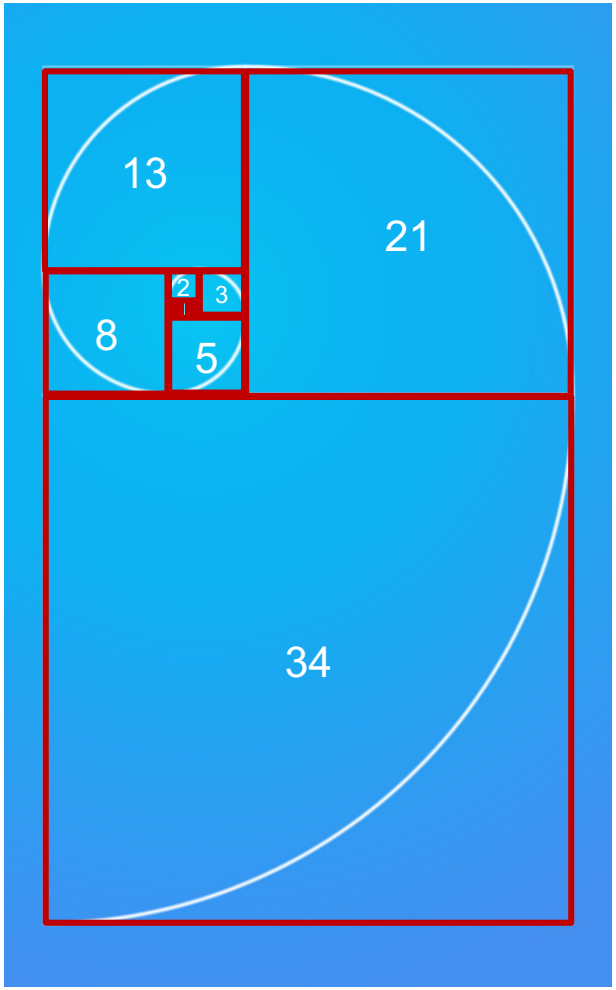
La suite de Fibonacci

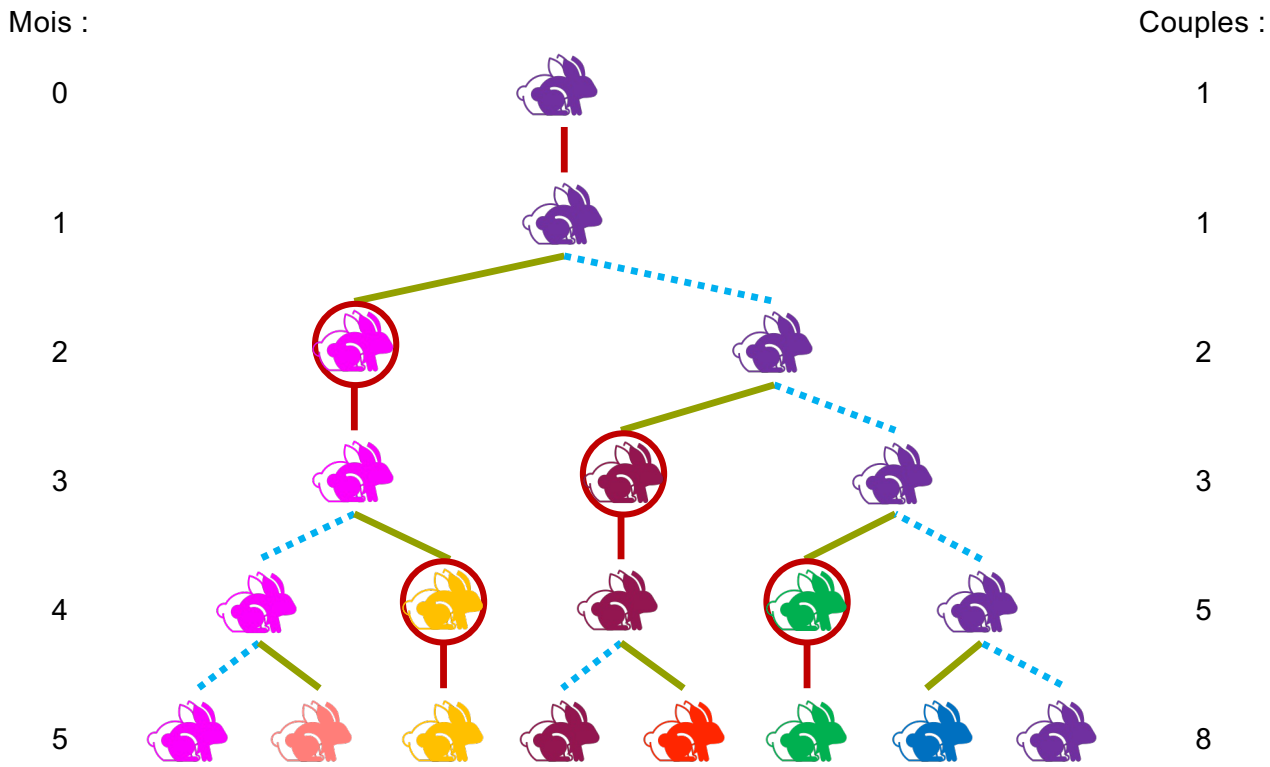


Les tours de Hanoï



Problème : Suite de Fibonacci



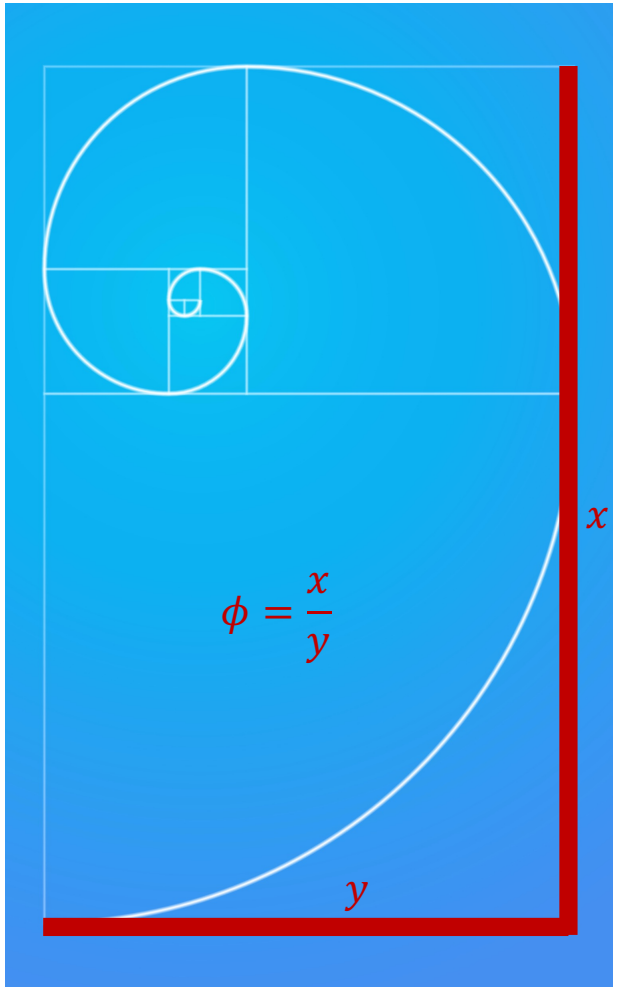


$$F(n) = F(n-1) + F(n-2)$$

$F(n-1)$: les couples existants du mois précédant, qui sont toujours là.

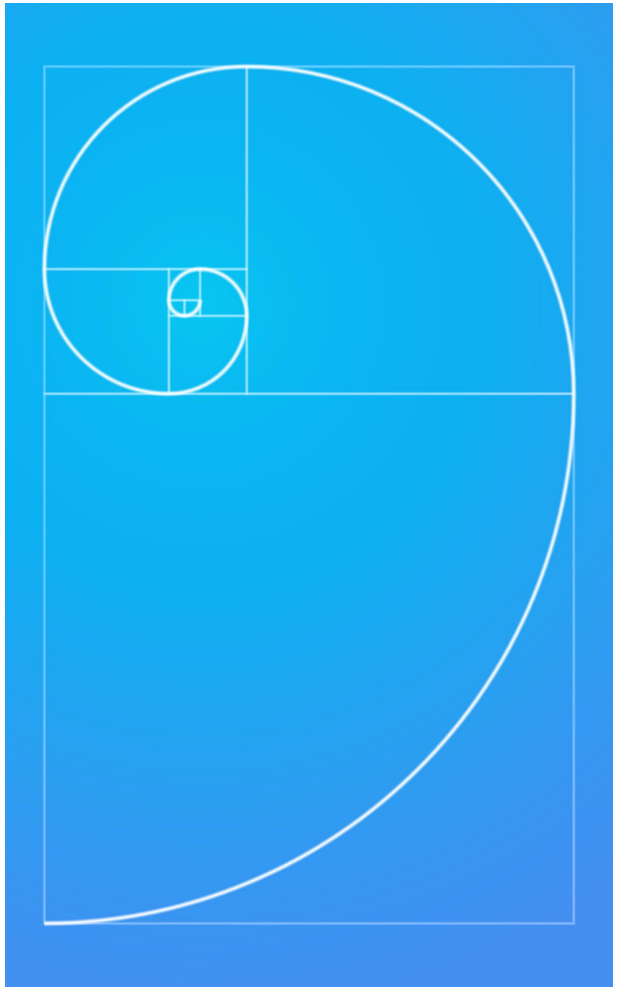
$F(n-2)$: les nouveaux couples nés ce mois-ci, descendants des couples déjà présents il y a 2 mois.

Nombre d'or : ϕ



$$\phi = \frac{1+\sqrt{5}}{2} \approx 1{,}61803\ldots$$

Problème : Suite de Fibonacci



1, 1, 2, 3, 5, 8, 13, 21, 34, …

Le nième terme de la suite de Fibonacci.

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

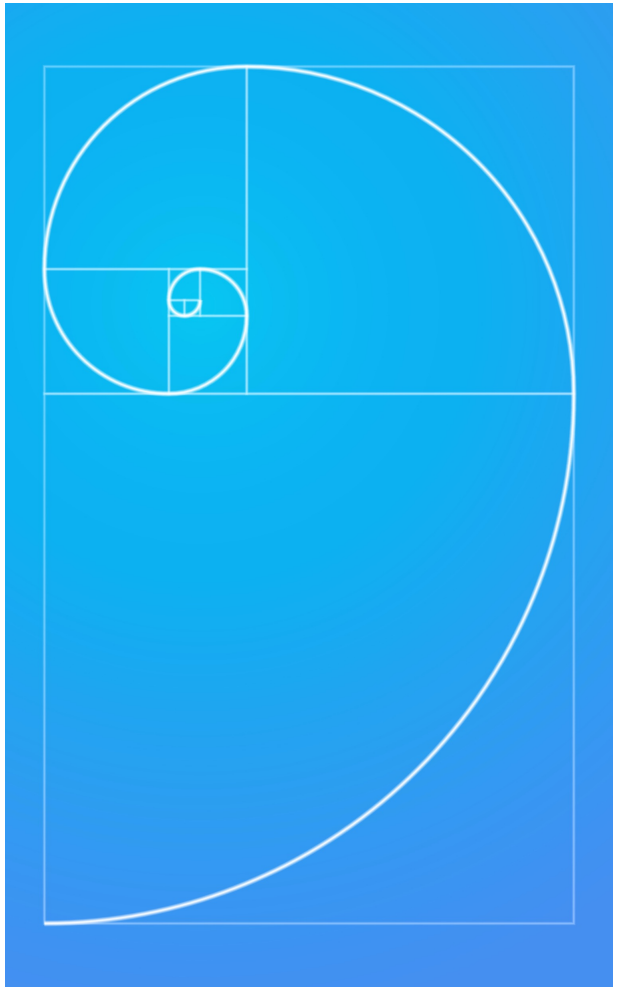
$$F(0) = F(1) = 1$$

Fibonacci

entrée : entier naturel **n**
sortie : nième nombre dans la suite de Fibonacci

```
Si n <= 1
        Sortir : 1
Sinon
        Sortir : Fibonacci(n - 1) + Fibonacci(n - 2)
```

Suite de Fibonacci : récursive vs. itérative



Récursive :

Fibonacci

entrée : entier naturel **n**
sortie : nième nombre dans la suite de Fibonacci

```
Si n <= 1
        Sortir : 1
Sinon
        Sortir : Fibonacci(n - 1) + Fibonacci(n - 2)
```

Complexité :

$\Theta(\ ?\ )$

Itérative :

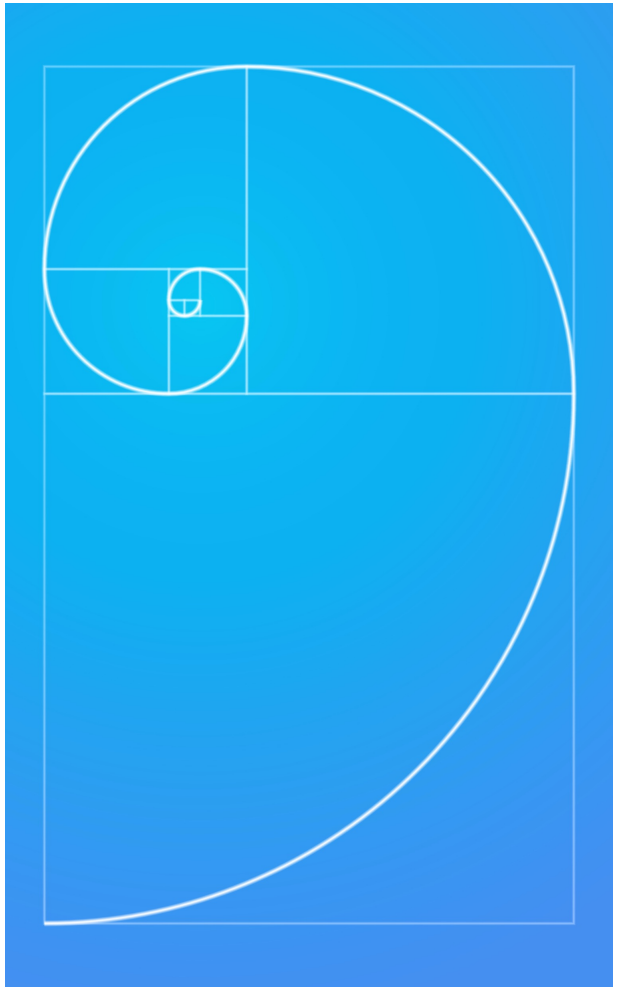
Fibonacci

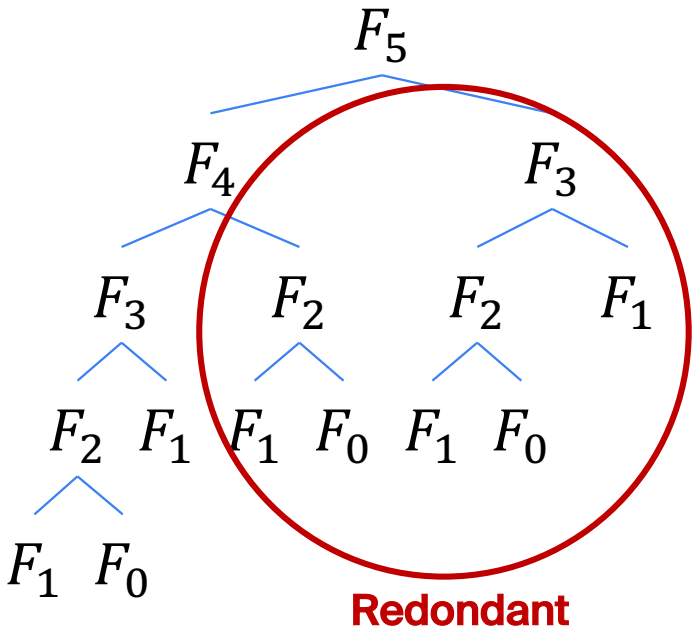
entrée : entier naturel **n**
sortie : nième nombre dans la suite de Fibonacci

```
avant ← 0
dernier ← 1
Tant que n > 1
        tmp ← avant + dernier
        avant ← dernier
        dernier ← tmp
        n ← n - 1
Sortir : dernier + avant
```

$\Theta(n)$

Problème : Fibonacci

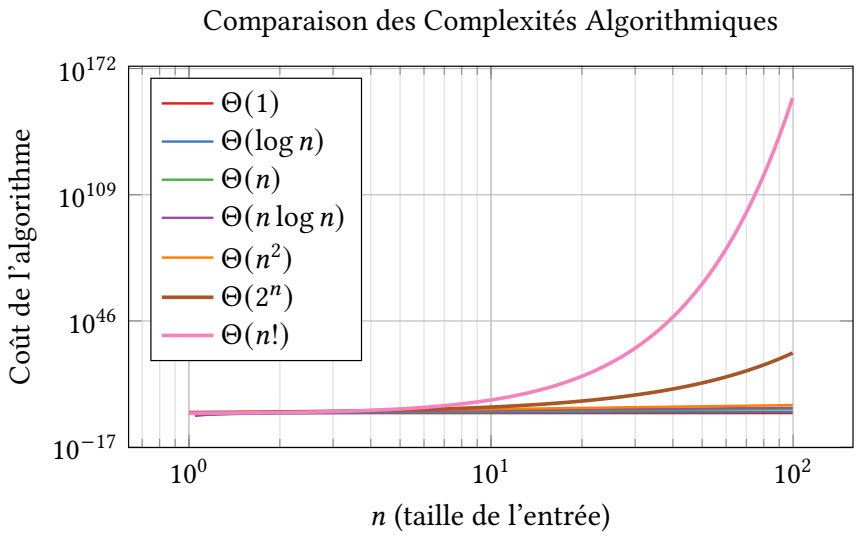


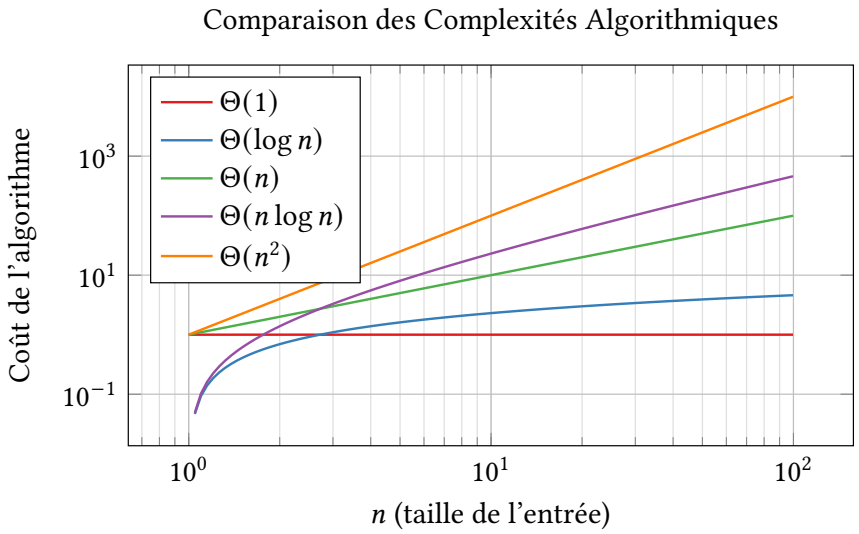


Complexité :

$$\Theta(\phi^n)$$

ICC-T 10 : Notation $\Theta(\cdot)$: Ordres de grandeur





Impraticables : $\boldsymbol{\Theta(2^n)}$, $\boldsymbol{\Theta(n!)}$

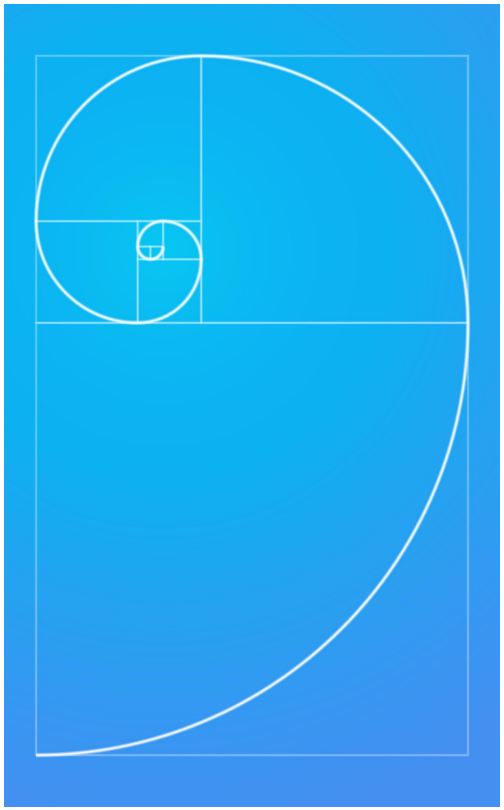
Plus lents, mais souvent acceptés : $\boldsymbol{\Theta(n^2)}$ ... $\boldsymbol{\Theta(n^k)}$, $\boldsymbol{\Theta(n.\log(n))}$

Rapides: $\boldsymbol{\Theta(1)}$, $\boldsymbol{\Theta(\log n)}$, $\boldsymbol{\Theta(n)}$

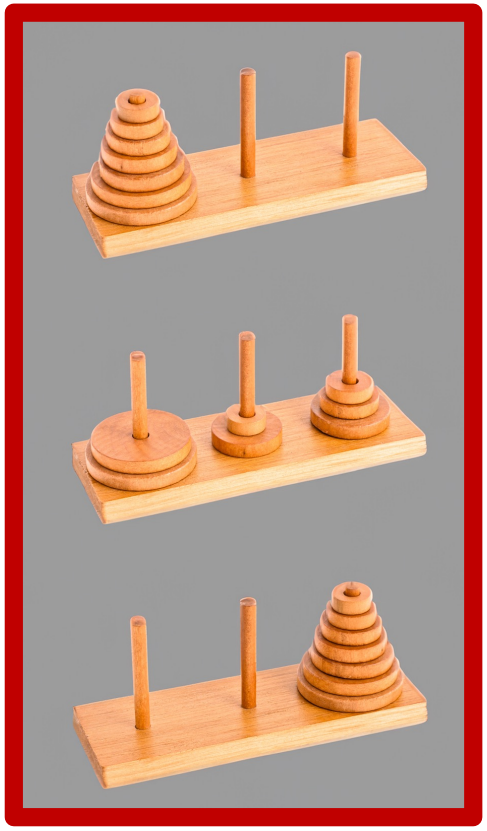
Problèmes

La Factorielle

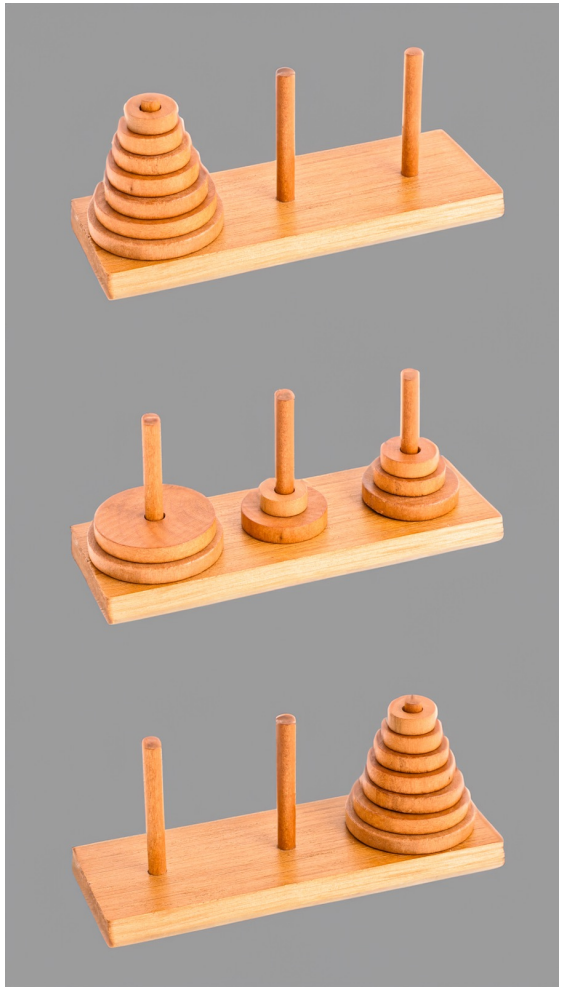
La suite de Fibonacci



Les tours de Hanoï



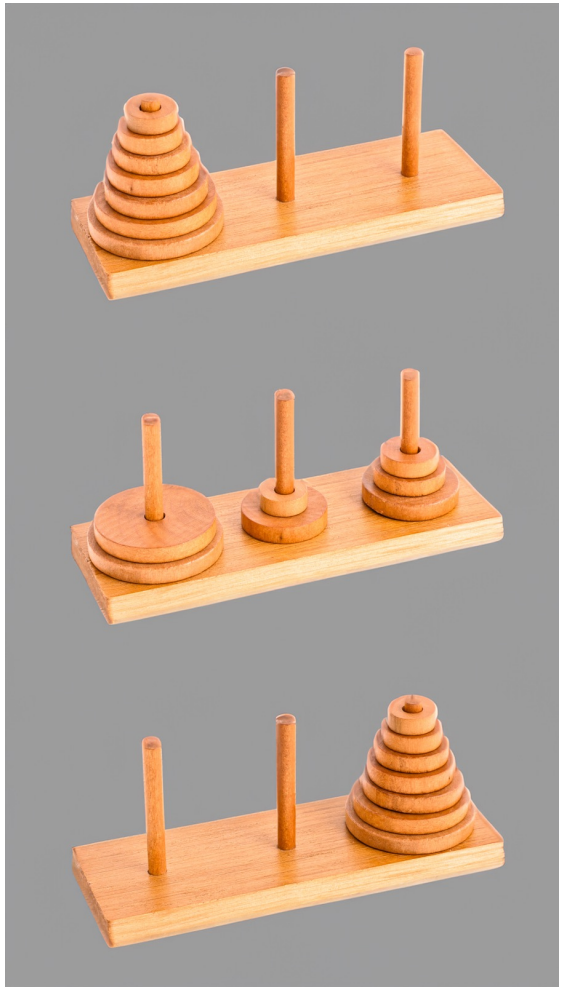
Problème : Tours de Hanoï

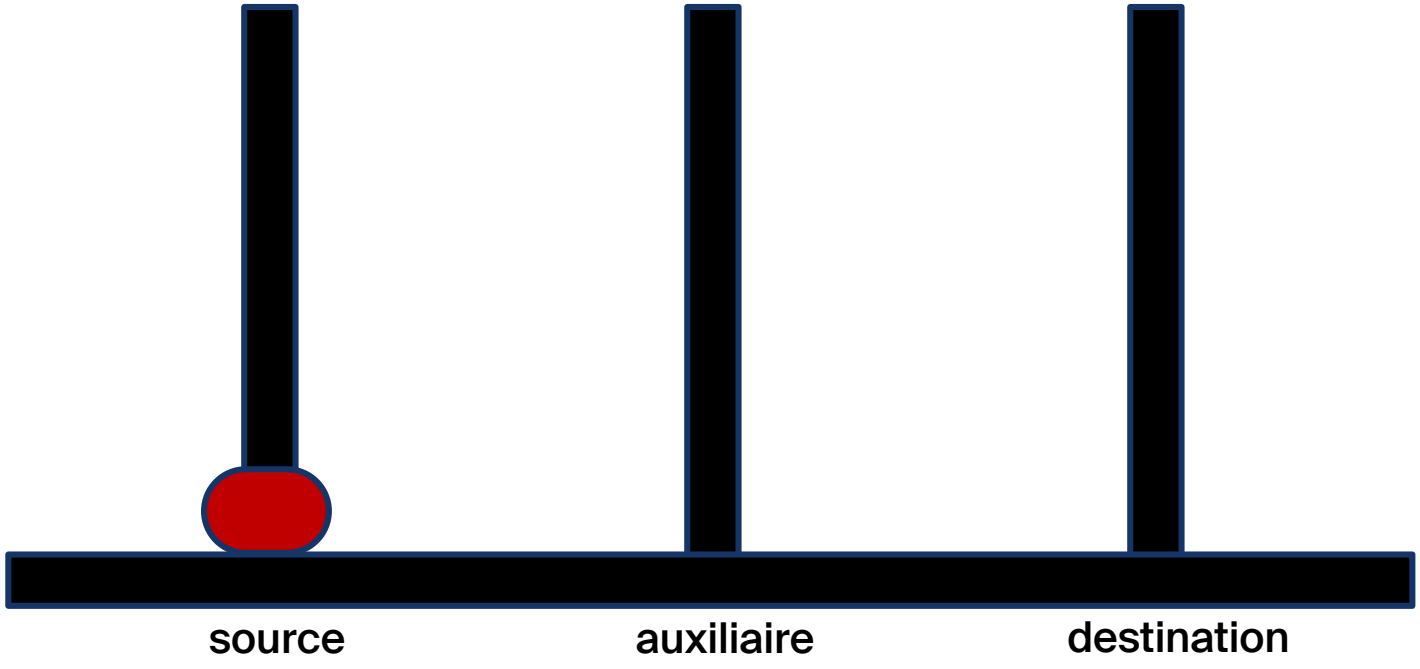


Régles :

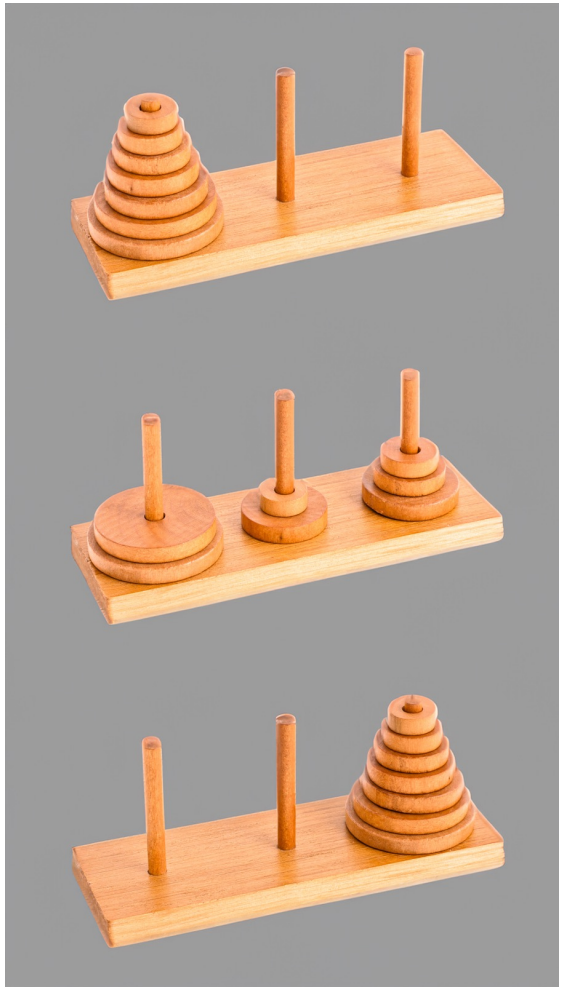
- Déplacer **tous les disques** du pilier source vers le pilier de destination en utilisant uniquement **un pilier auxiliaire**
- Déplacer **un seul disque à chaque fois**
- Ne poser un disque que sur le sol ou **sur un disque plus grand**

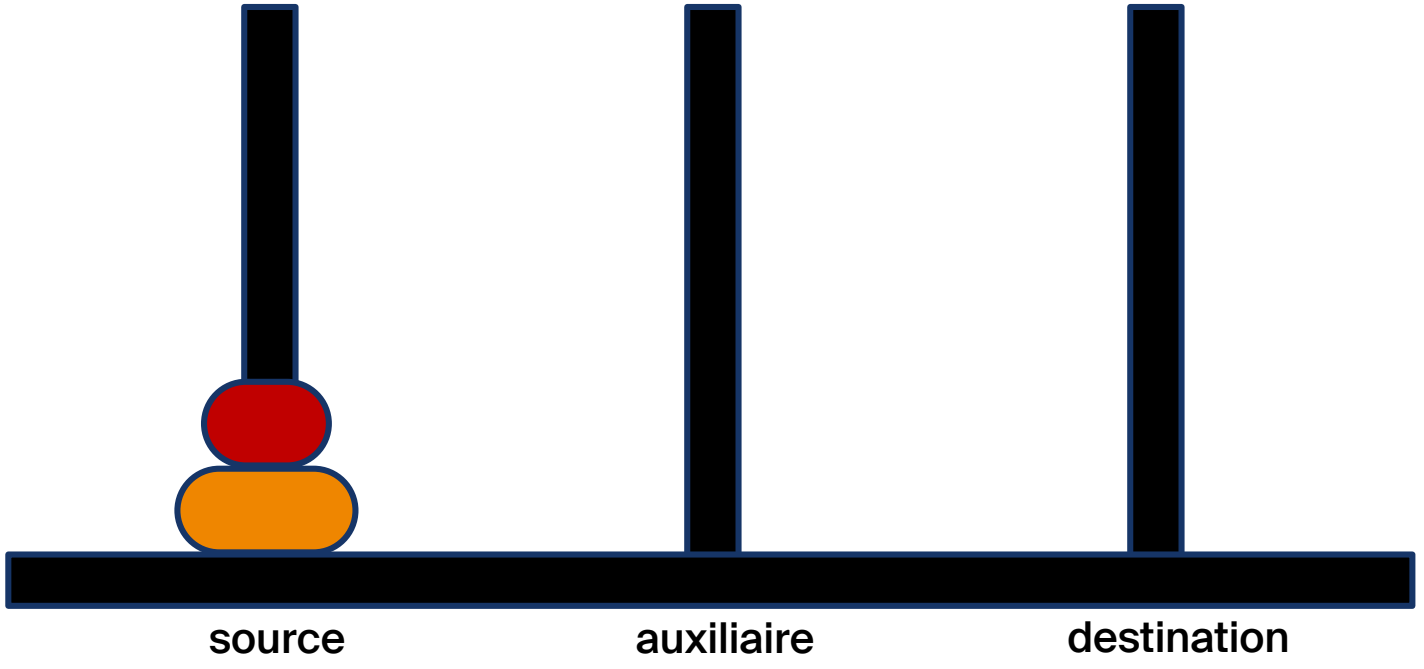
Problème : Tours de Hanoï (*n* = 1)



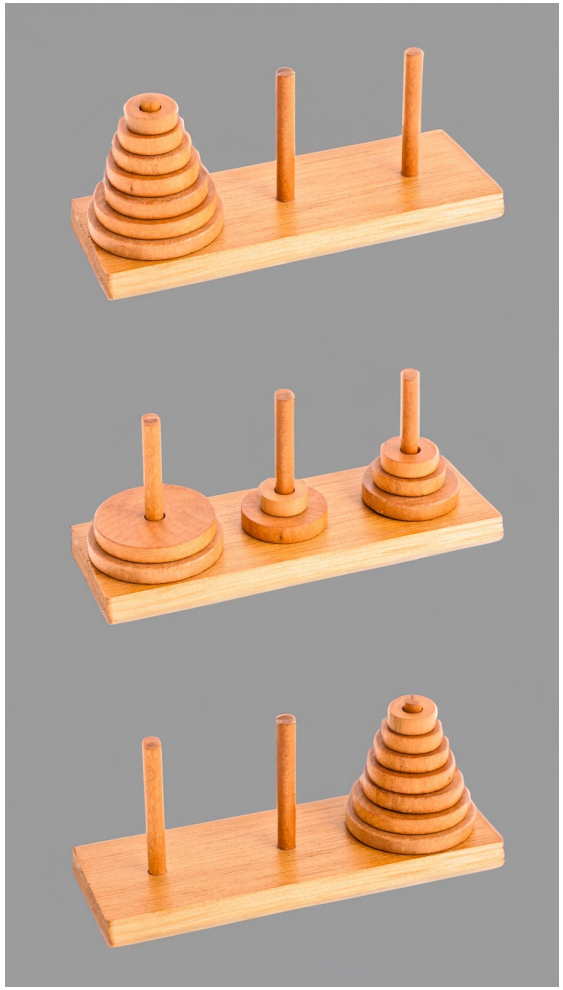


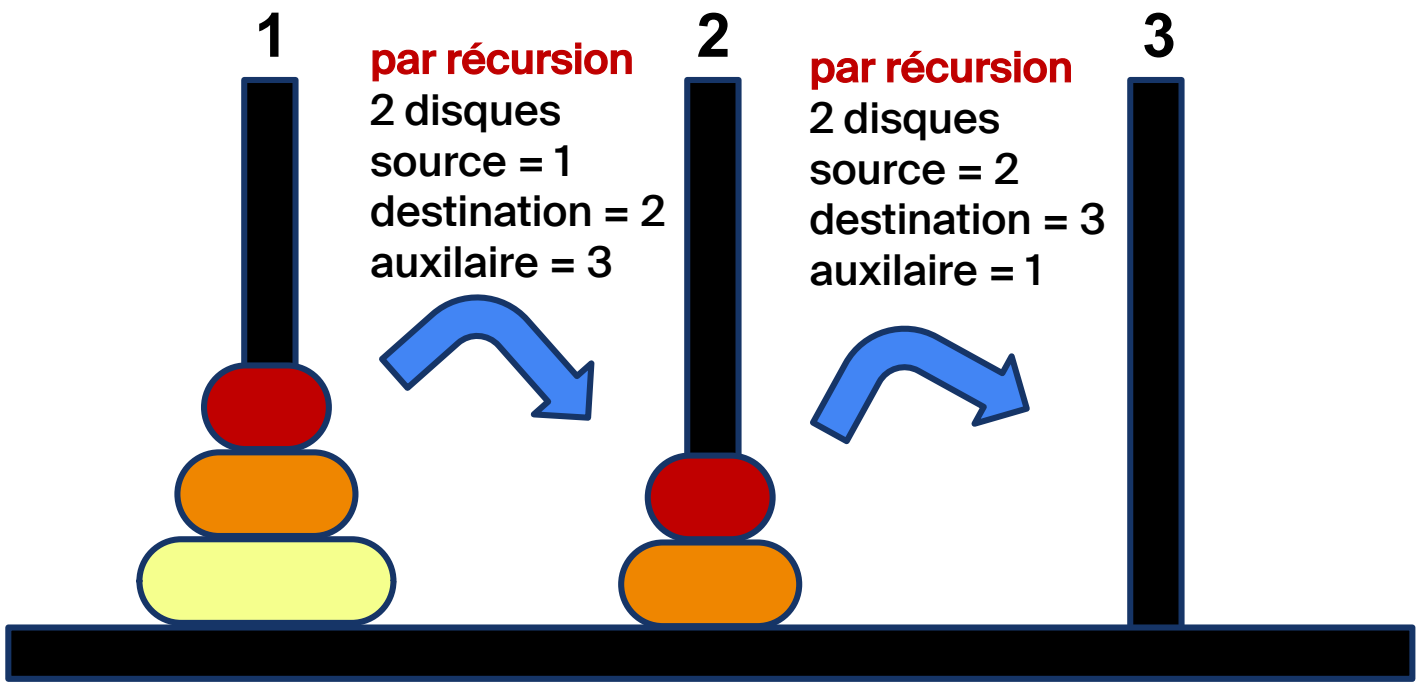
Problème : Tours de Hanoï (*n* = 2)



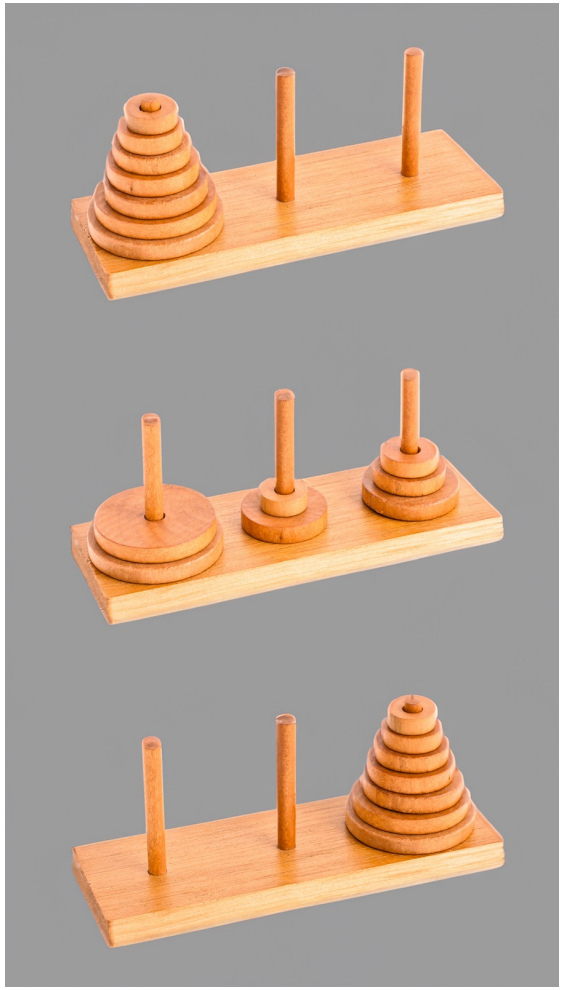


Problème : Tours de Hanoï (*n* > 2)





Tours de Hanoï : Algorithme



Tours de Hanoï

entrée : nombre de disques **n**, entier, n > 0
pilier **source**, entier, 1≤ source ≤ 3
pilier **destination**, entier, 1≤ destination ≤ 3
sortie : mouvements pour résoudre le probème des Tours de Hanoï

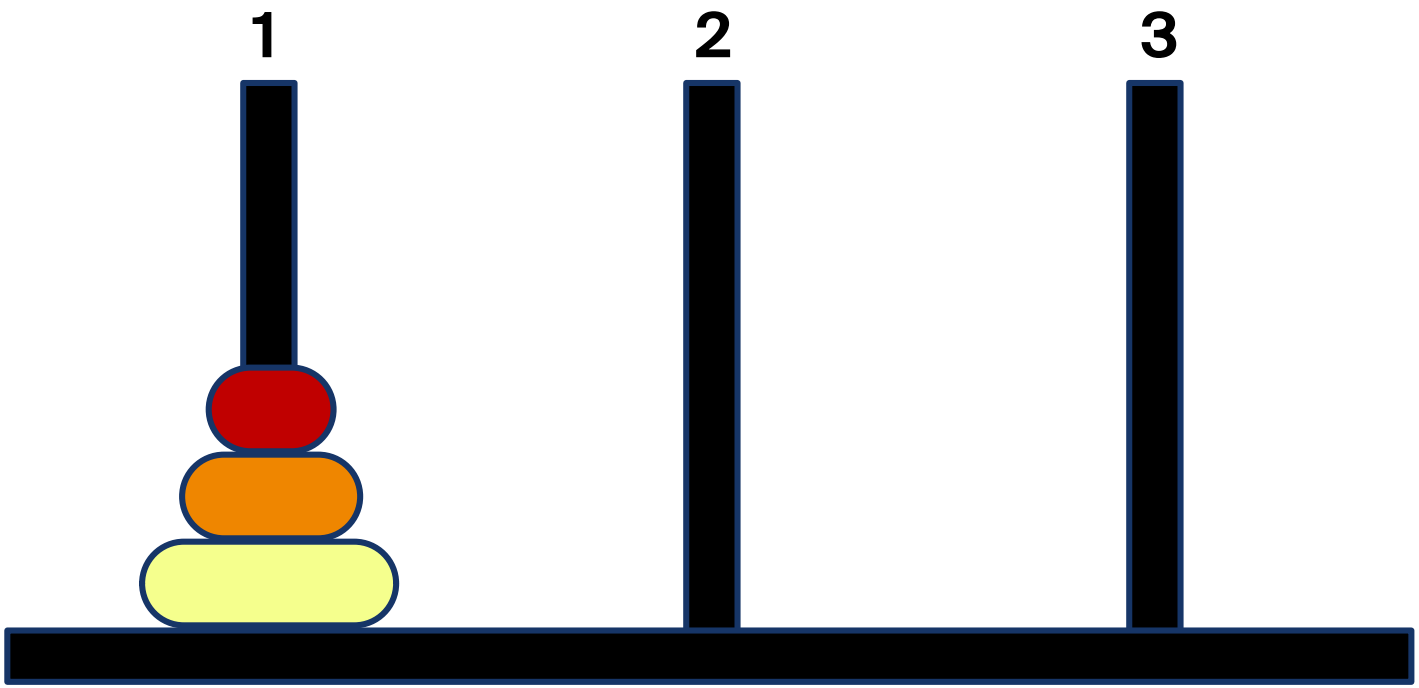
```
auxiliaire ← 6 - source - destination
Si n = 1
    Afficher : "Déplacer disque du pilier ", source, " au pilier ", destination
Sinon
    Tours de Hanoï(n - 1, source, auxiliaire)
    Tours de Hanoï(1, source, destination)
    Tours de Hanoï(n - 1, auxiliaire, destination)
```

Complexité : $\Theta(2^n)$

Si on avait 64 disques, il faudrait $2^{64}-1 \approx 10^{19}$ déplacements,
soit **des milliards d'années** même avec un supercalculateur !

Problème : Tours de Hanoï

```
1  -> 3
1  -> 2
3  -> 2
1  -> 3
2  -> 1
2  -> 3
1  -> 3
```

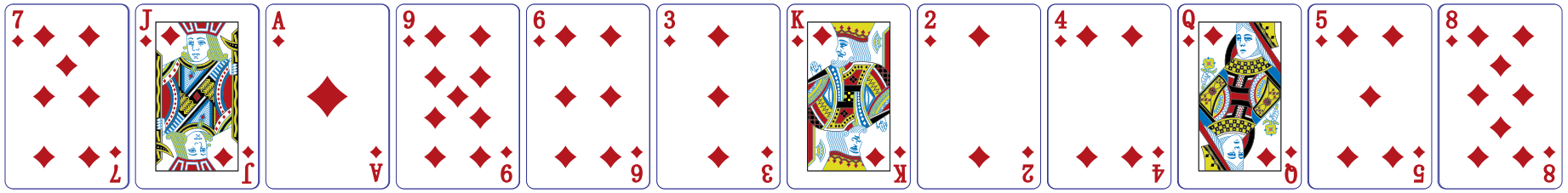


Aujourd'hui

- La récursivité
- Complexité logarithmique
- Tri par fusion

Recherche d'un élément dans une liste

- Le 10 est-il là ?



- **Liste non-ordonnée :**
  - Pas de choix, il faut parcourir toute la liste.
- **Liste ordonnée :**
  - On peut faire mieux : **recherche par dichotomie**.

Recherche dichotomique

- **Problème**
  - Identifier si un élément fait partie d'une liste ordonnée.

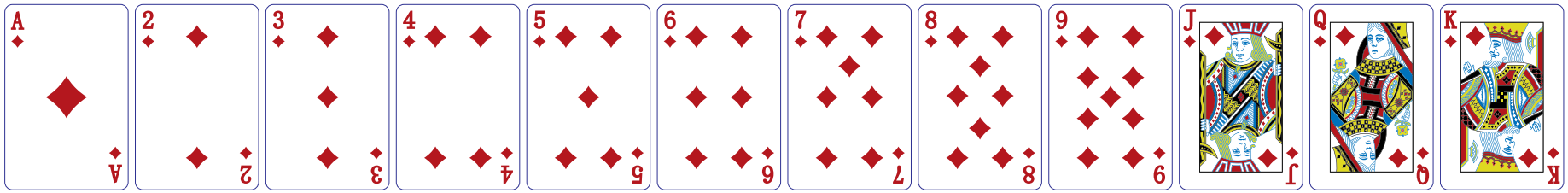
Dichotomie

entrée : Liste ordonée **L** de nombres entiers de taille **n**
objet **x** qu'on veut chercher
sortie : **oui** si x est dans L, **non** sinon

Si n = 1
    Sortir : x = L(1)

milieu $\leftarrow \left\lfloor \frac{n}{2} \right\rfloor$
Si x $\leq$ L(milieu)
    Sortir : Dichotomie(L(1 : milieu), milieu, x)
Sinon
    Sortir : Dichotomie(L(1+milieu : n), n - milieu, x)

Recherche dichotomique



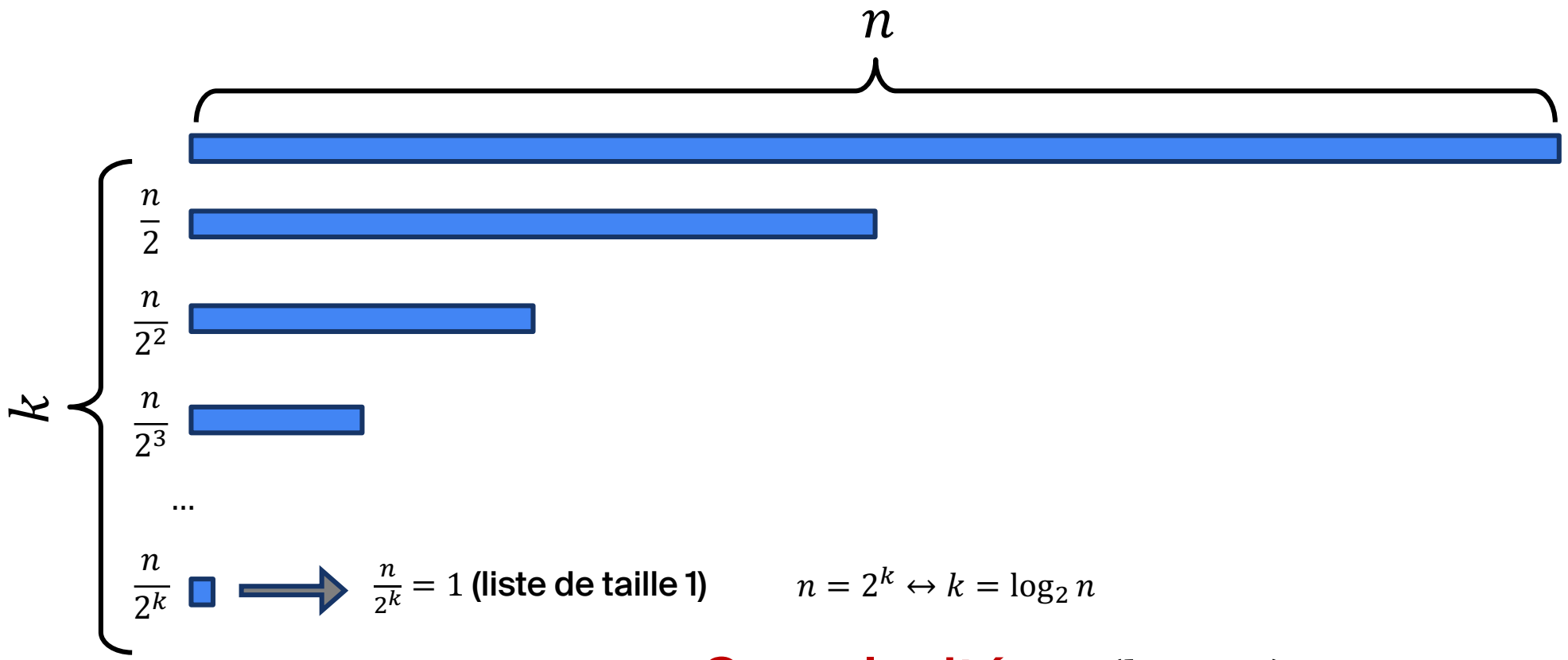
Dichotomie

entrée : Liste ordonée **L** de nombres entiers de taille **n**
objet **x** qu'on veut chercher
sortie : **oui** si x est dans L, **non** sinon

Si n = 1
    Sortir : x = L(1)

milieu $\leftarrow \left\lfloor \frac{n}{2} \right\rfloor$
Si x $\leq$ L(milieu)
    Sortir : Dichotomie(L(1 : milieu), milieu, x)
Sinon
    Sortir : Dichotomie(L(1+milieu : n), n - milieu, x)

Complexité temporelle : Recherche dichotomique

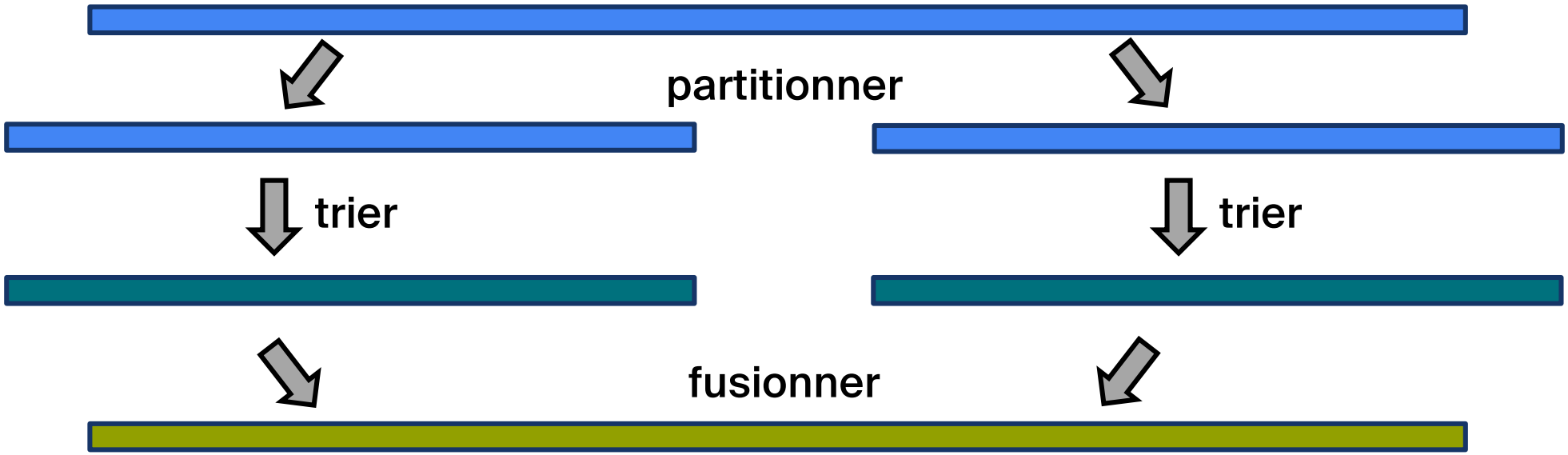


Complexité : $\Theta(\log_2 n)$

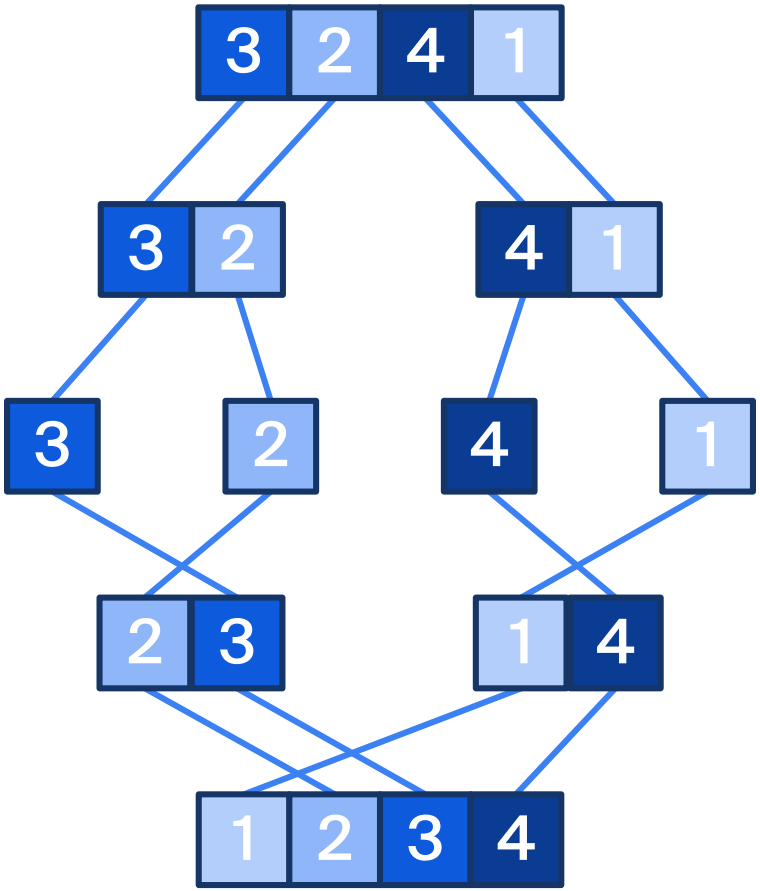
Aujourd'hui

- La récursivité
- Complexité logarithmique
- **Tri par fusion**

Tri par fusion



Tri par fusion

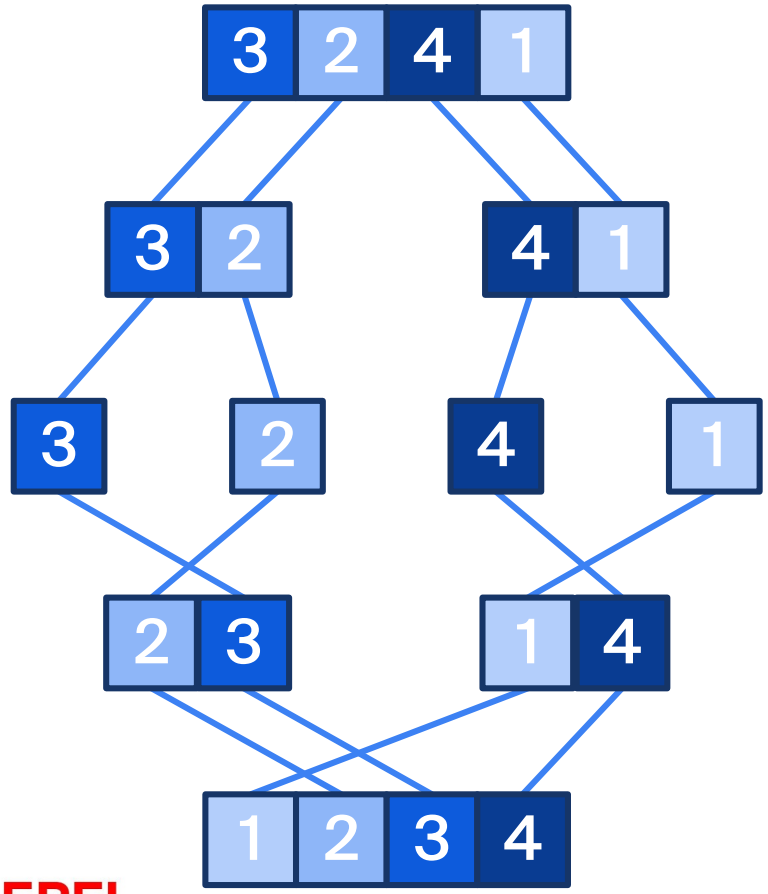


Listes de taille 1 (triées)

Fusion

Fusion

Tri par fusion



Tri par fusion

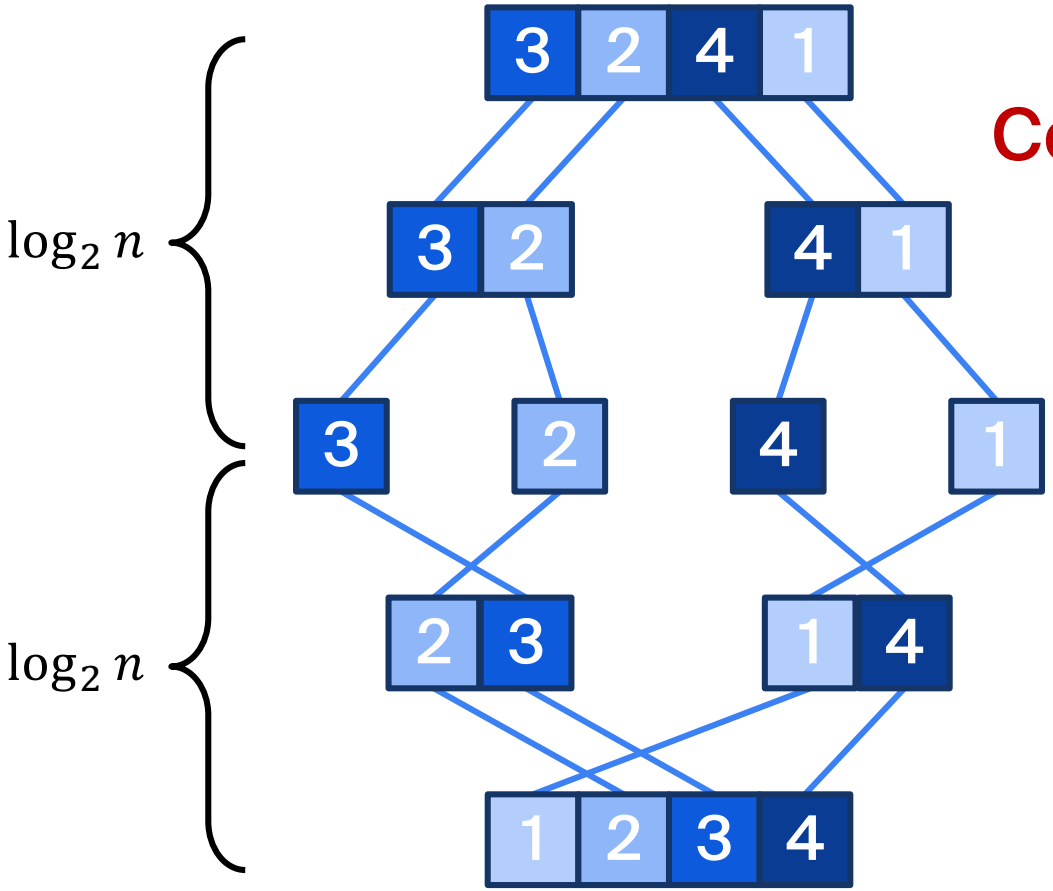
entrée : Liste **L** non triée de nombres entiers, de taille **n**
sortie : Liste **L'** triée

Si n = 1
    Sortir : L

milieu ← $\left\lfloor \frac{n}{2} \right\rfloor$
L_1 ← Tri par fusion(L(1 : milieu), milieu)
L_2 ← Tri par fusion(L(1+milieu : n), n - milieu)
L' ← fusion(L_1,L_2)
Sortir : L'

Quelle est la complexité ?

Complexité temporelle : Tri par fusion



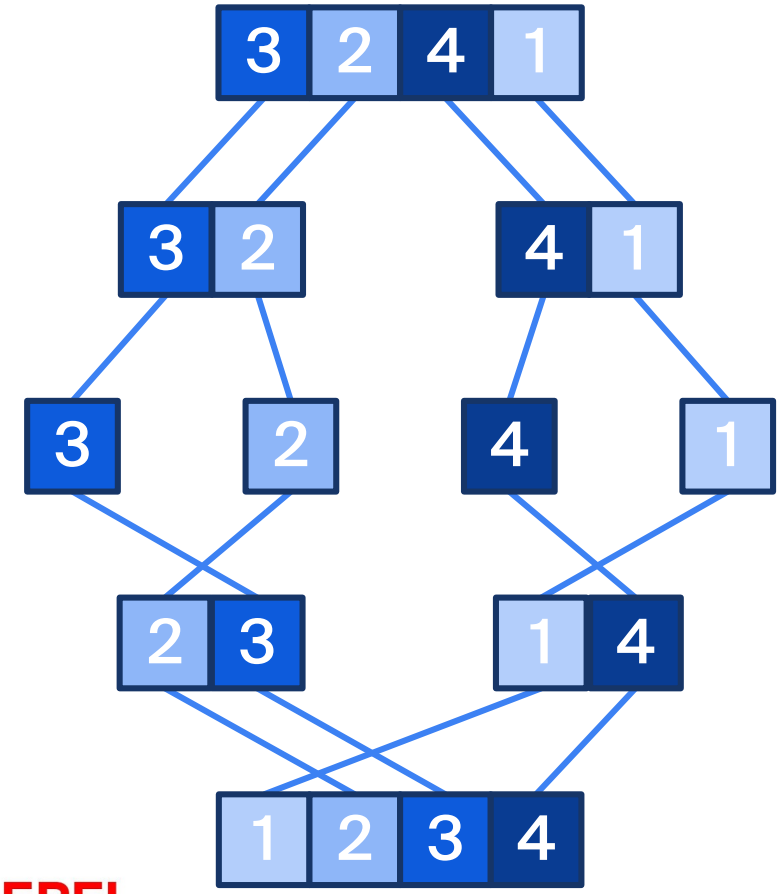
Complexité : $\Theta(n \log_2 n)$

Listes de taille 1 (triées)

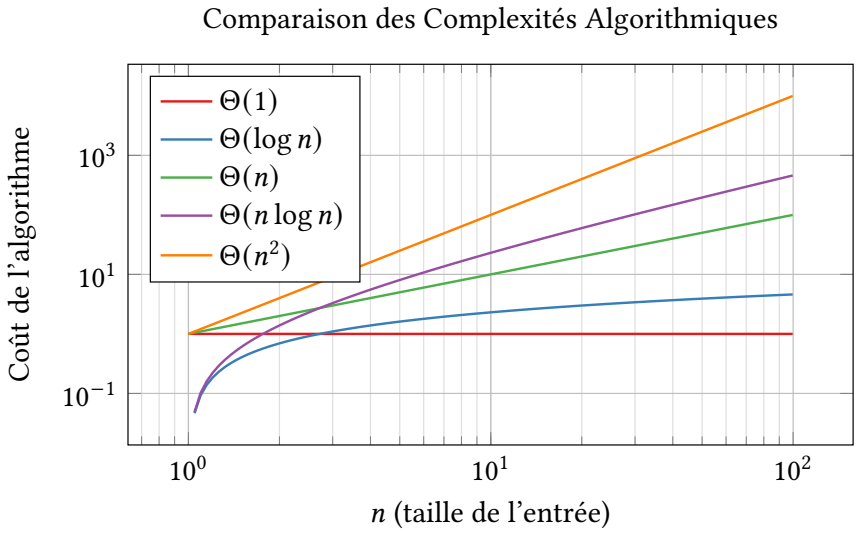
Fusion : $\Theta(n)$

Fusion : $\Theta(n)$

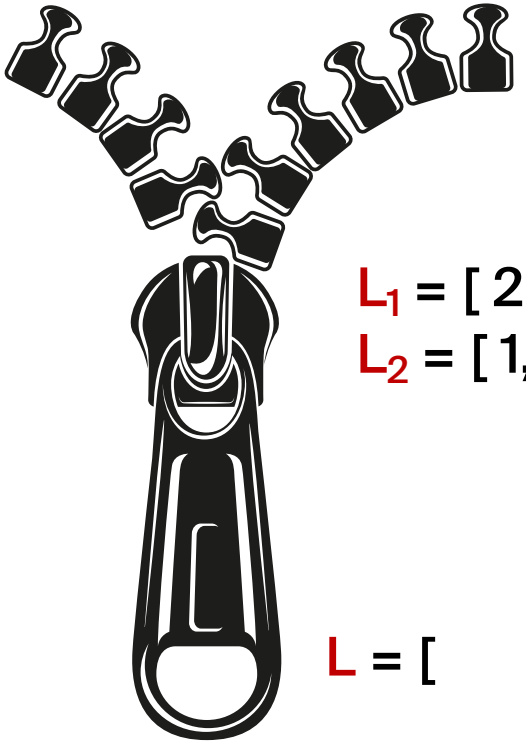
Complexité temporelle : Tri par fusion



Complexité : $\Theta(n \log_2 n)$



Tri par fusion : fermeture éclair



L_1 = [2, 5, 8], m_1 = 3
L_2 = [1, 3, 7, 9], m_2 = 4

L = [ ]

fusion

entrée : Listes ordonnées L_1, L_2 de taille m_1 et m_2 resp.
sortie : Liste L de taille $m_1 + m_2$ également ordonnée

```
j1 ← 1
j2 ← 1
j ← 1

Tant que j1 ≤ m1 et j2 ≤ m2 :
    Si L1(j1) ≤ L2(j2) :
        L(j) ← L1(j1)
        j1 ← j1 + 1
    Sinon :
        L(j) ← L2(j2)
        j2 ← j2 + 1
    j ← j + 1
```

```
Si j1 = m1 + 1 :
    Tant que j2 ≤ m2 :
        L(j) ← L2(j2)
        j2 ← j2 + 1
        j ← j + 1
Sinon :
    Tant que j1 ≤ m1 :
        L(j) ← L1(j1)
        j1 ← j1 + 1
        j ← j + 1

Sortir : L
```

Complexité : $\Theta(m_1 + m_2)$

Algorithme entier : Tri par fusion

Tri par fusion

entrée : Liste L non triée de nombres entiers, de taille n
sortie : Liste L' triée

```
Si n = 1
    Sortir : L

milieu ← ⌊n/2⌋
L1 ← Tri par fusion(L(1 : milieu), milieu)
L2 ← Tri par fusion(L(1+milieu : n), n - milieu)
L' ← fusion(L1,L2)
Sortir : L'
```

fusion

entrée : Listes ordonnées L_1, L_2 de taille m_1 et m_2 resp.
sortie : Liste L de taille $m_1 + m_2$ également ordonnée

```
j1 ← 1
j2 ← 1
j ← 1

Tant que j1 ≤ m1 et j2 ≤ m2 :
    Si L1(j1) ≤ L2(j2) :
        L(j) ← L1(j1)
        j1 ← j1 + 1
    Sinon :
        L(j) ← L2(j2)
        j2 ← j2 + 1
    j ← j + 1
```

```
Si j1 = m1 + 1 :
    Tant que j2 ≤ m2 :
        L(j) ← L2(j2)
        j2 ← j2 + 1
        j ← j + 1
Sinon :
    Tant que j1 ≤ m1 :
        L(j) ← L1(j1)
        j1 ← j1 + 1
        j ← j + 1

Sortir : L
```

Résumé Cours 11 – ICC-T

- La récursivité permet de résoudre les problèmes qui sont décomposables en sous-problèmes plus simples du même problème.
  - Factorielle
  - Suite de Fibonacci
  - Tours de Hanoï
  - Recherche par dichotomie
  - Tri par fusion
- Lorsqu'on cherche un élément dans une liste triée, la recherche par dichotomie permet une résolution plus efficace, avec une complexité de $\Theta(\log_2 n)$ comparée à la recherche exhaustive en $\Theta(n)$.
- Le tri par fusion, de complexité $\Theta(n\log_2 n)$, permet de trier une liste plus efficacement que le tri par insertion, de complexité $\Theta(n^2)$ dans le pire des cas.

rafael.pires@epfl.ch

Merci