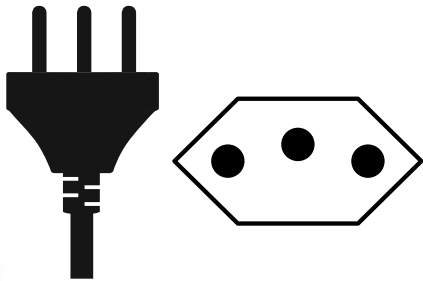


Information, Calcul et Communication

CS-119(k) ICC – Théorie Semaine 3

Rafael Pires
rafael.pires@epfl.ch

Précédemment, dans ICC-T 02



- 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

Annonces

31.03 ICC-P Cours par zoom (aussi diffusé en salle)

Séance d'exercices normale

Cours et séries, partie programmation

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|--|----|----|----|----|----|
| P | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | | 10 | 11 | 12 | 13 | 14 |
| T | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | | | 10 | 11 | 12 | 13 | 14 |

Cours et séries, partie théorique

14.03 ICC-T Changement de salle
exceptionnel : CM14

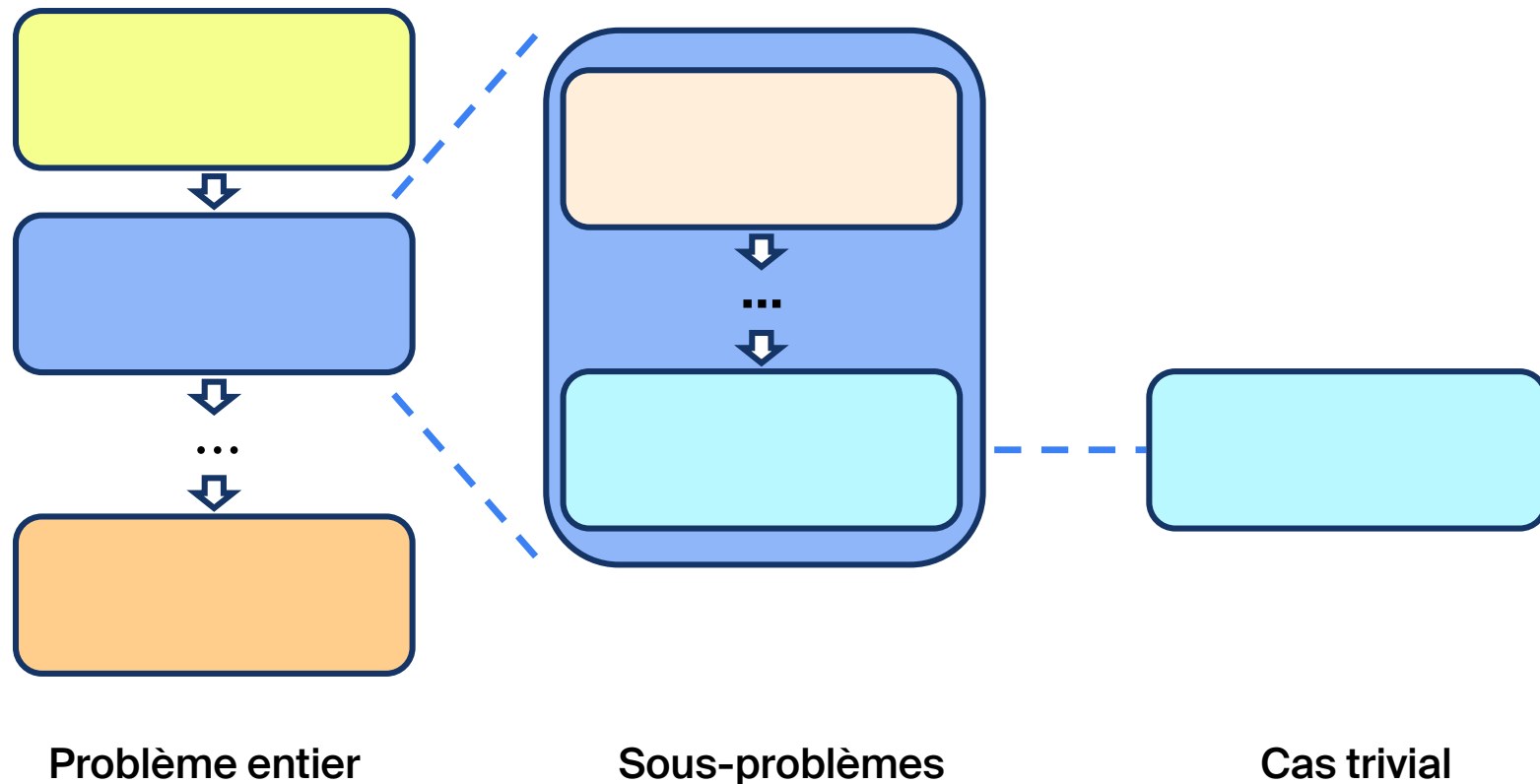
Aujourd'hui

- La récursivité
- Complexité logarithmique
- Tri par fusion

La légende des tours de Hanoï



Conception d'algorithmes : Diviser pour mieux régner



Récurtivité

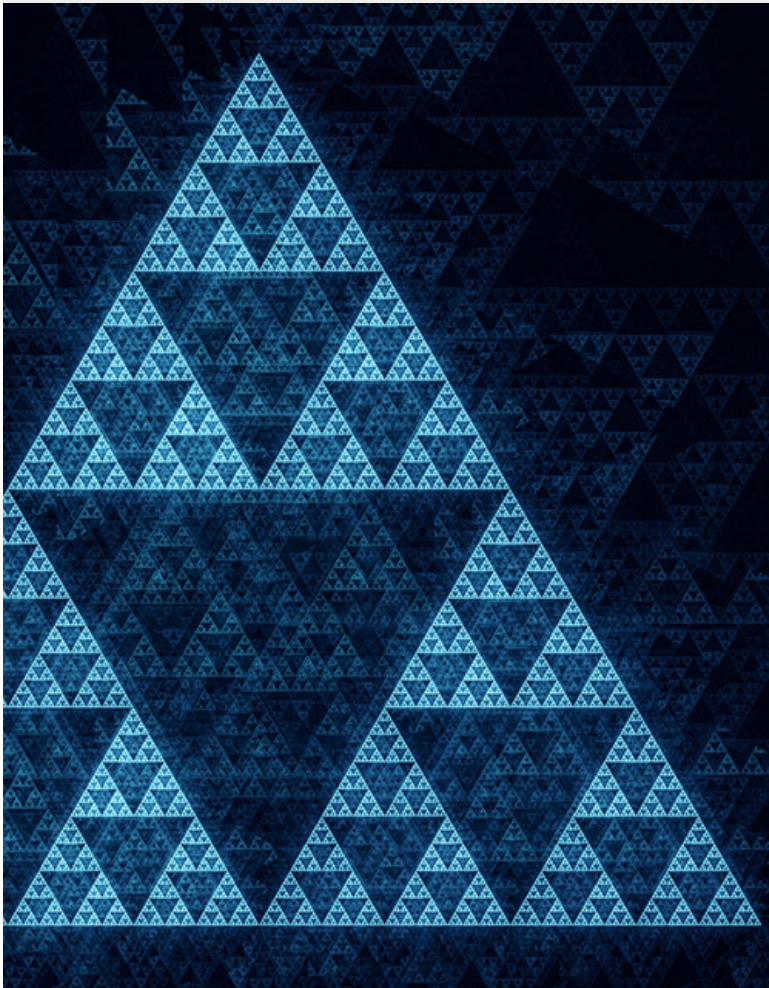


Problème entier

Sous-problèmes

Cas trivial

Réversivité 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** :

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

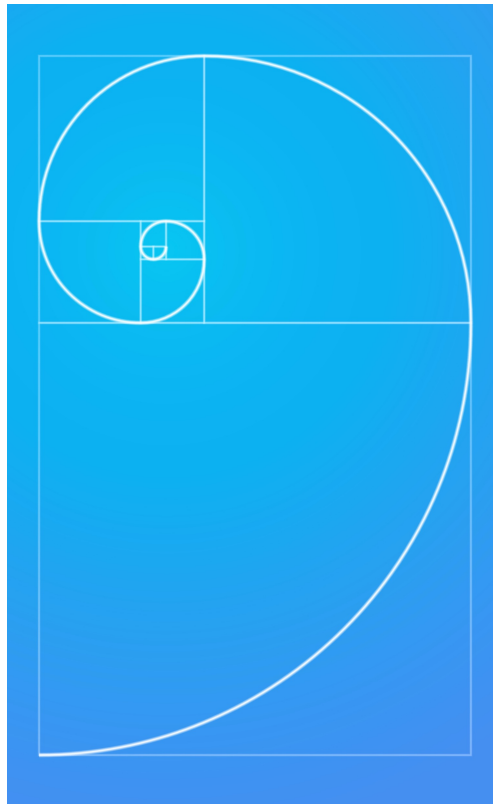
$$\binom{n}{0} = \binom{n}{n} = 1$$

Problèmes

La Factorielle



La suite de Fibonacci



Les tours de Hanoï



Problème : Factorielle



La factorielle d'un nombre entier n .

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

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

Attention



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

Problème : Factorielle



La factorielle d'un nombre entier n .

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

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

Condition de
terminaison



Si $n \leq 1$
Sortir : 1

Sinon

Sortir : $n * \text{Factorielle}(n-1)$

Recombinaison



Instance plus
simple



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 **recombinaison** les résultats d'appels récursifs pour former la solution au problème de départ.

Mon algo récursif

entrée : **X**

sortie : ...

Si terminaison(X)

Sortir : ...

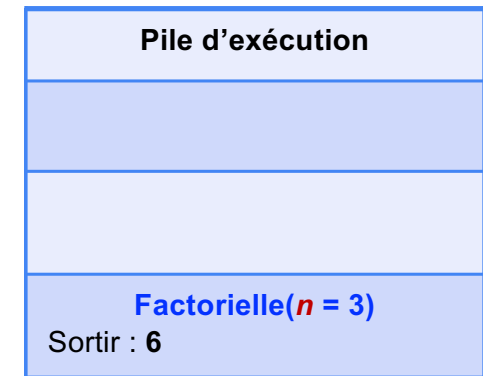
Sinon

...

Mon algo récursif (entrée de l'instance réduite)

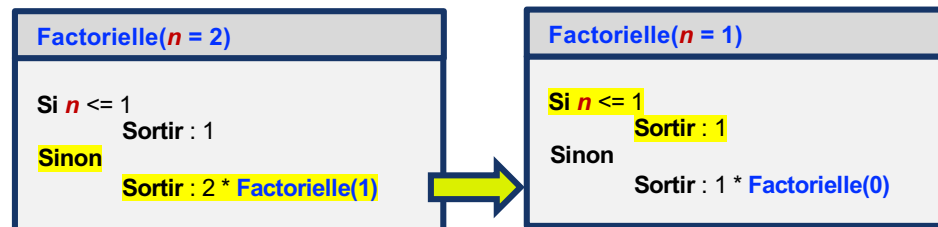
...

Exécution : Factorielle

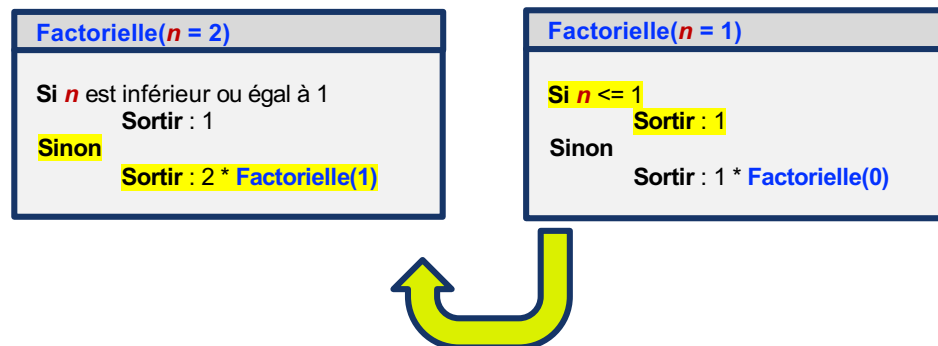


Quelques observations

- 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)

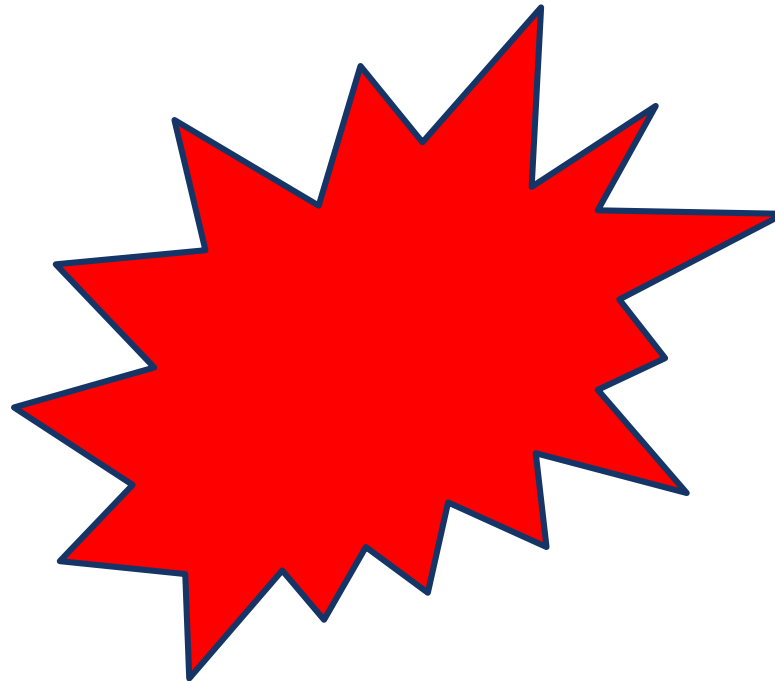


Exécution : Débordement de la pile



Factorielle(n)

Sortir : $n * \text{Factorielle}(n-1)$



Factorielle($n = -7$)
Sortir : $-7 * \text{Factorielle}(-8)$

Factorielle($n = -6$)
Sortir : $-6 * \text{Factorielle}(-7)$

Factorielle($n = -5$)
Sortir : $-5 * \text{Factorielle}(-6)$

Factorielle($n = -4$)
Sortir : $-4 * \text{Factorielle}(-5)$

Factorielle($n = -3$)
Sortir : $-3 * \text{Factorielle}(-4)$

Factorielle($n = -2$)
Sortir : $-2 * \text{Factorielle}(-3)$

Factorielle($n = -1$)
Sortir : $-1 * \text{Factorielle}(-2)$

Factorielle($n = 0$)
Sortir : $0 * \text{Factorielle}(-1)$

Factorielle($n = 1$)
Sortir : $1 * \text{Factorielle}(0)$

Factorielle($n = 2$)
Sortir : $2 * \text{Factorielle}(1)$

Factorielle($n = 3$)
Sortir : $3 * \text{Factorielle}(2)$

Attention



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

Factorielle : récursive vs. itérative



Réursive :

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 :

$\Theta(n)$

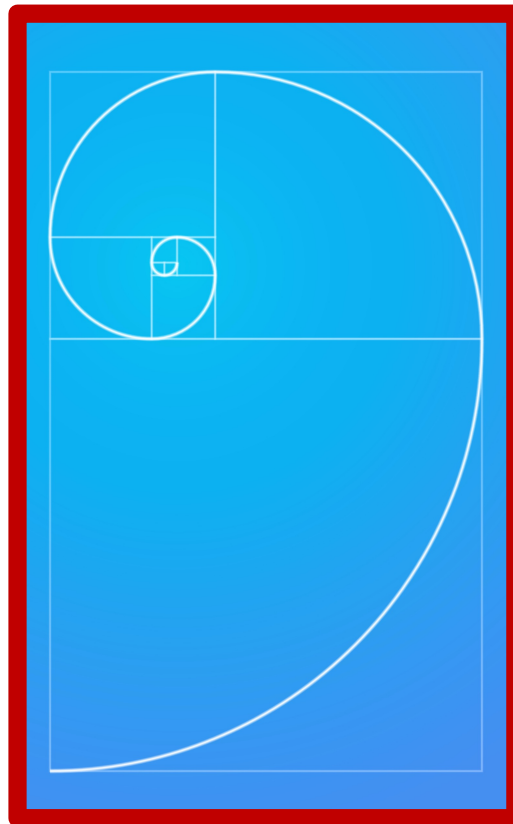
$\Theta(n)$

Problèmes

La Factorielle



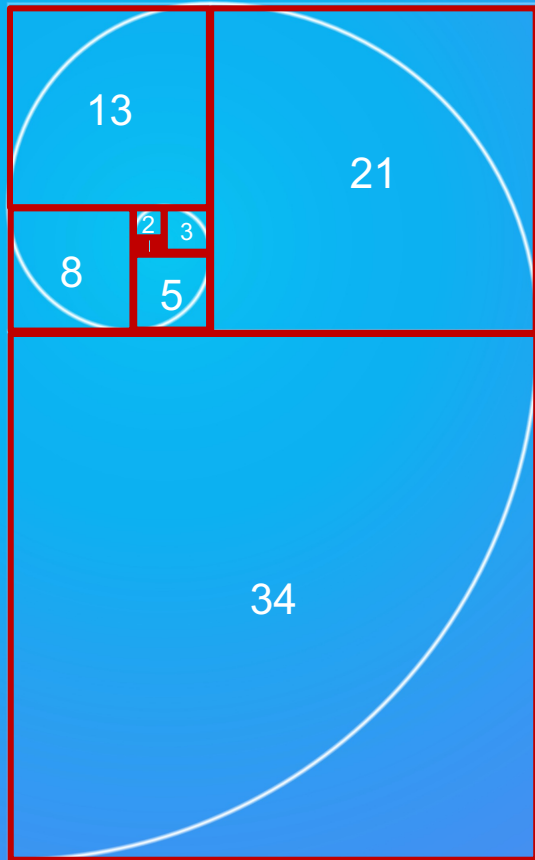
La suite de Fibonacci



Les tours de Hanoï



Problème : Suite de Fibonacci



Mois :

0

1

2

3

4

5

Couples :

1

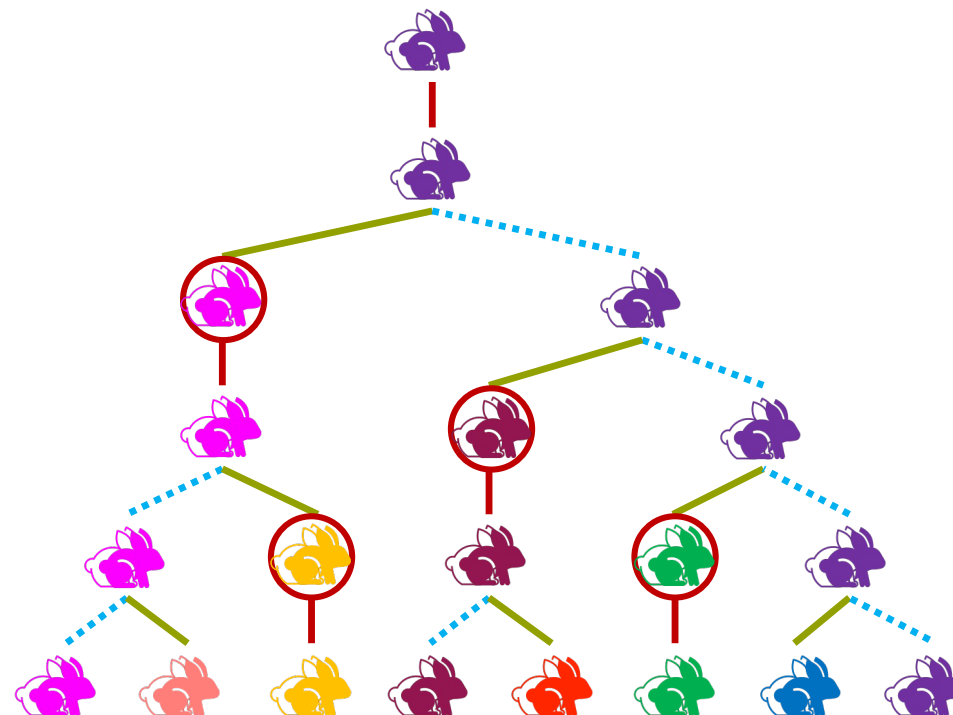
1

2

3

5

8

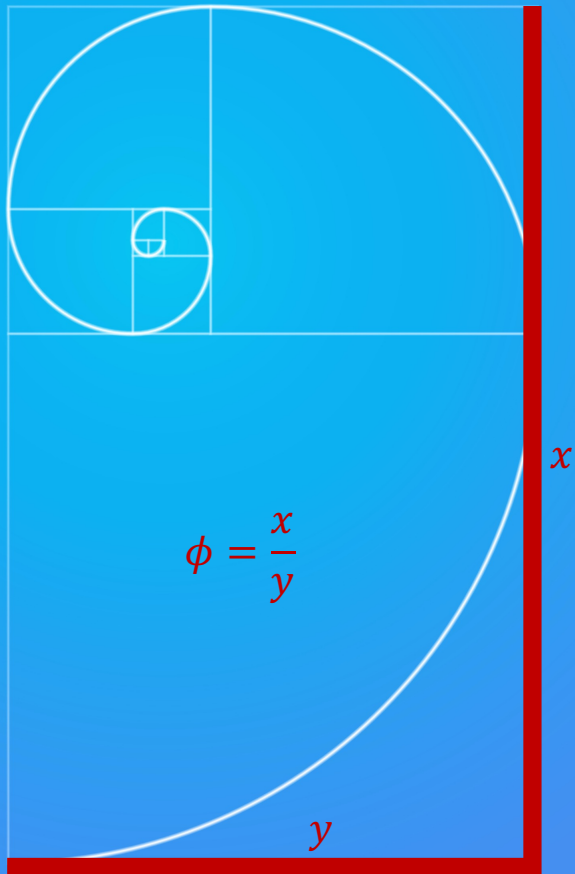


$$F(n) = F(n - 1) + F(n - 2)$$

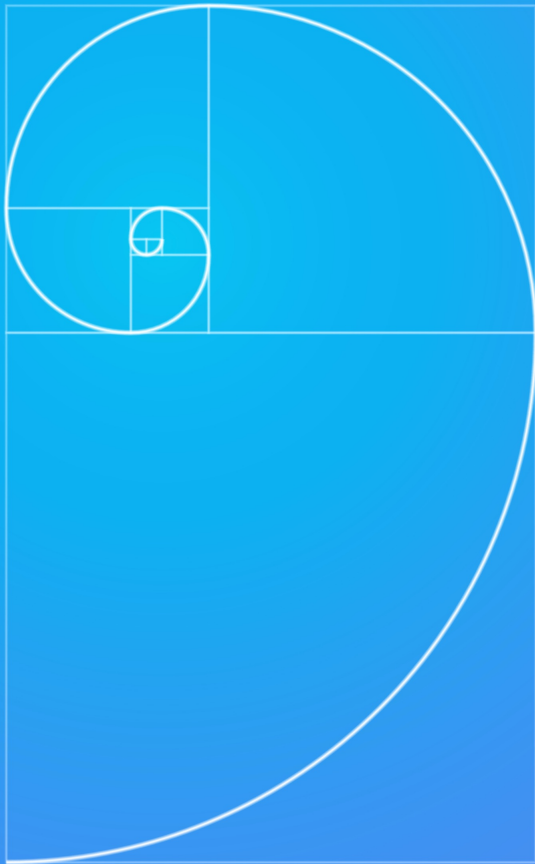
$F(n - 1)$: les couples existants du mois précédent, qui sont toujours là.

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

Nombre d'or : ϕ



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

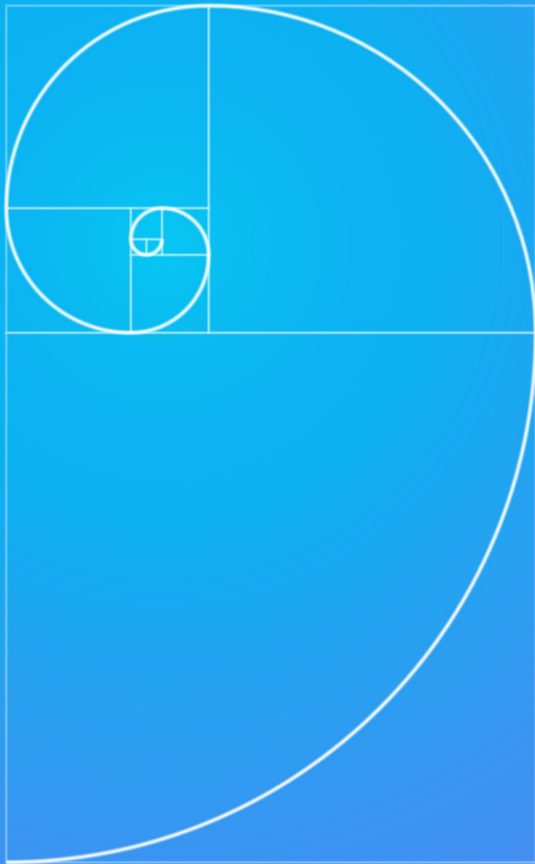
Si $n \leq 1$

Sortir : 1

Sinon

Sortir : $\text{Fibonacci}(n - 1) + \text{Fibonacci}(n - 2)$

Suite de Fibonacci : récursive vs. itérative



Réursive :

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**

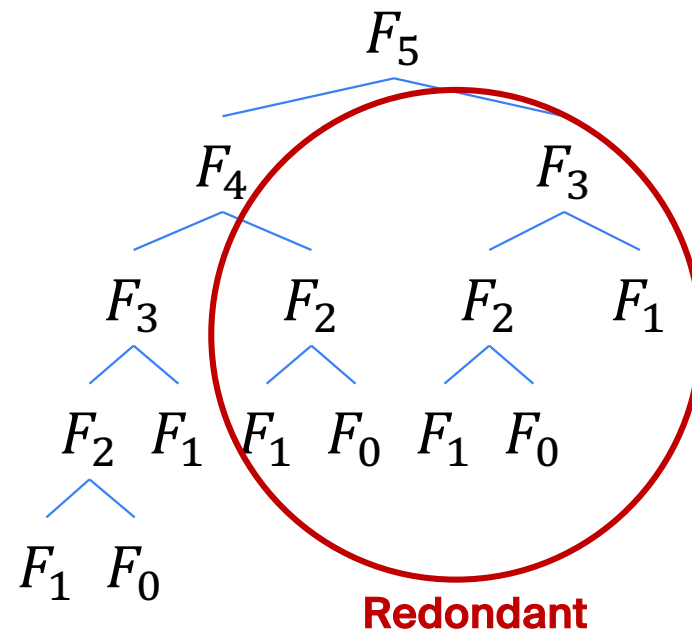
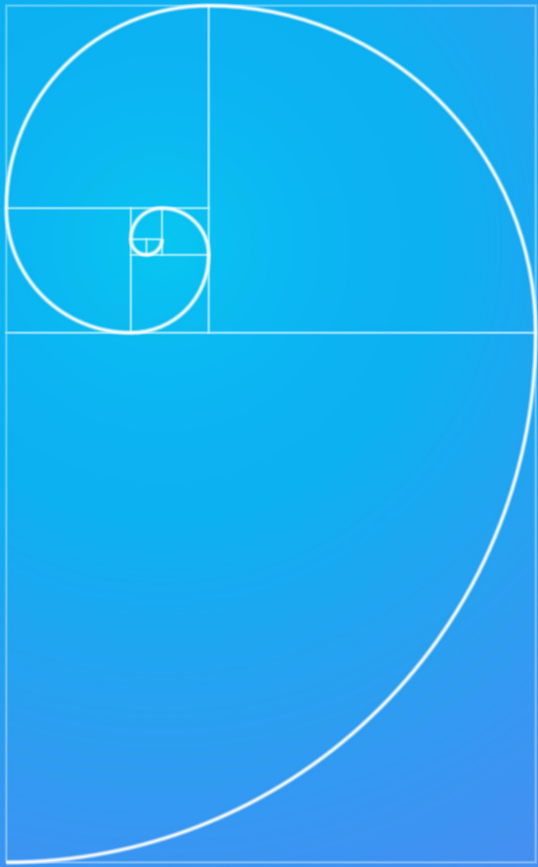
dernier ← **tmp**

n ← **n** - 1

Sortir : **dernier** + **avant**

$\Theta(n)$

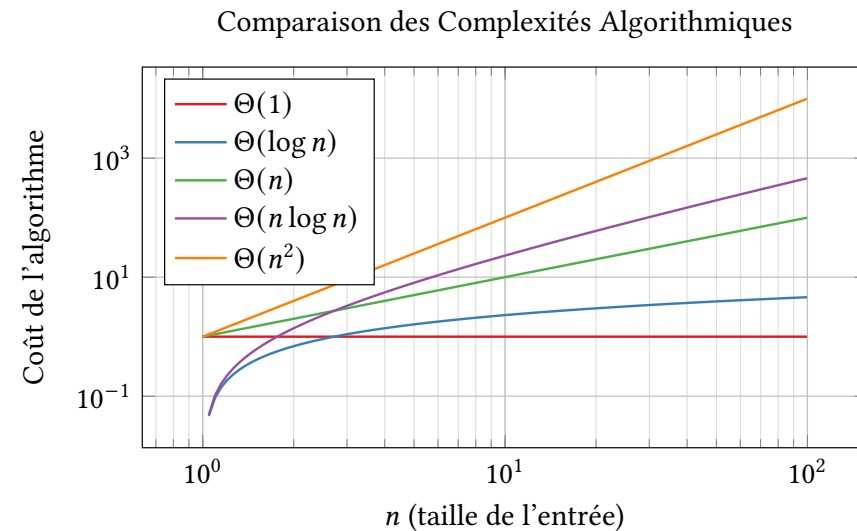
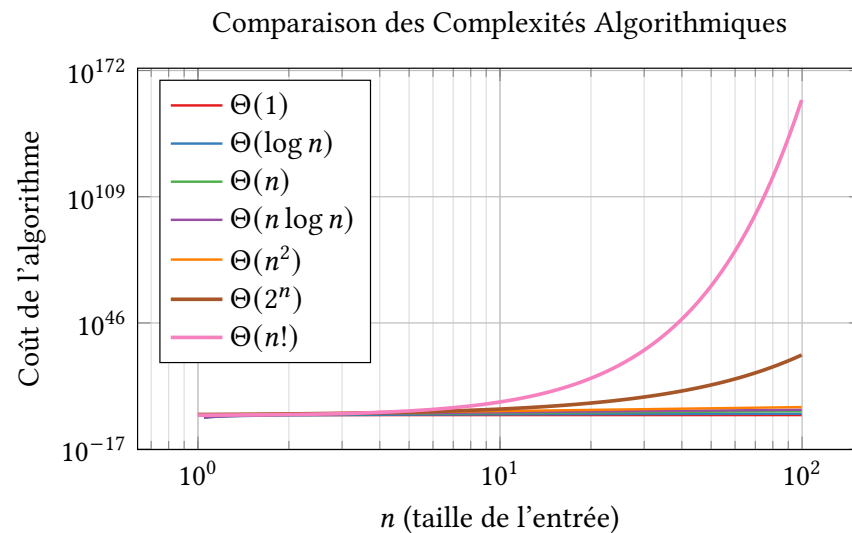
Problème : Fibonacci



Complexité :

$$\Theta(2^n)$$

ICC-T 02 : Notation $\Theta(\cdot)$: Ordres de grandeur



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

Plus lents, mais souvent acceptés : $\Theta(n^2) \dots \Theta(n^k)$, $\Theta(n \cdot \log(n))$

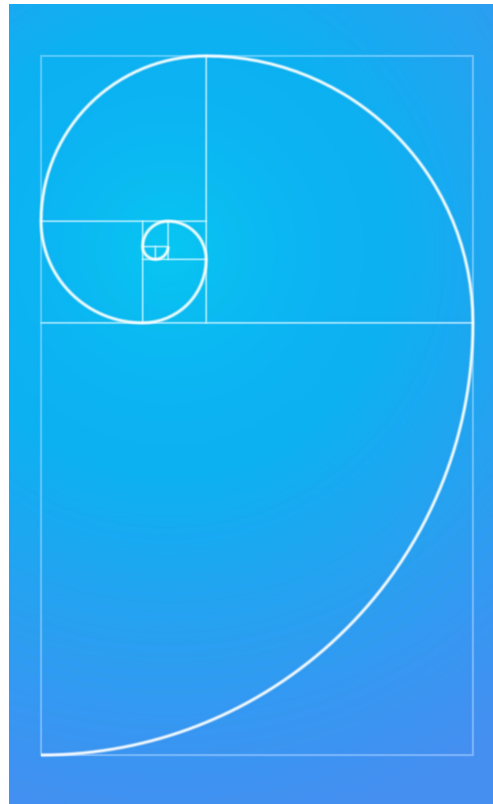
Rapides : $\Theta(1)$, $\Theta(\log n)$, $\Theta(n)$

Problèmes

La Factorielle



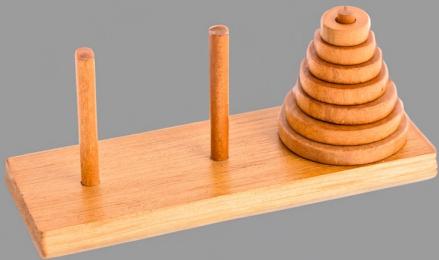
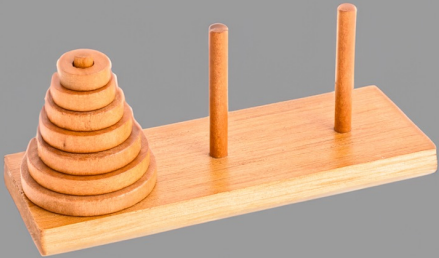
La suite de Fibonacci



Les tours de Hanoï



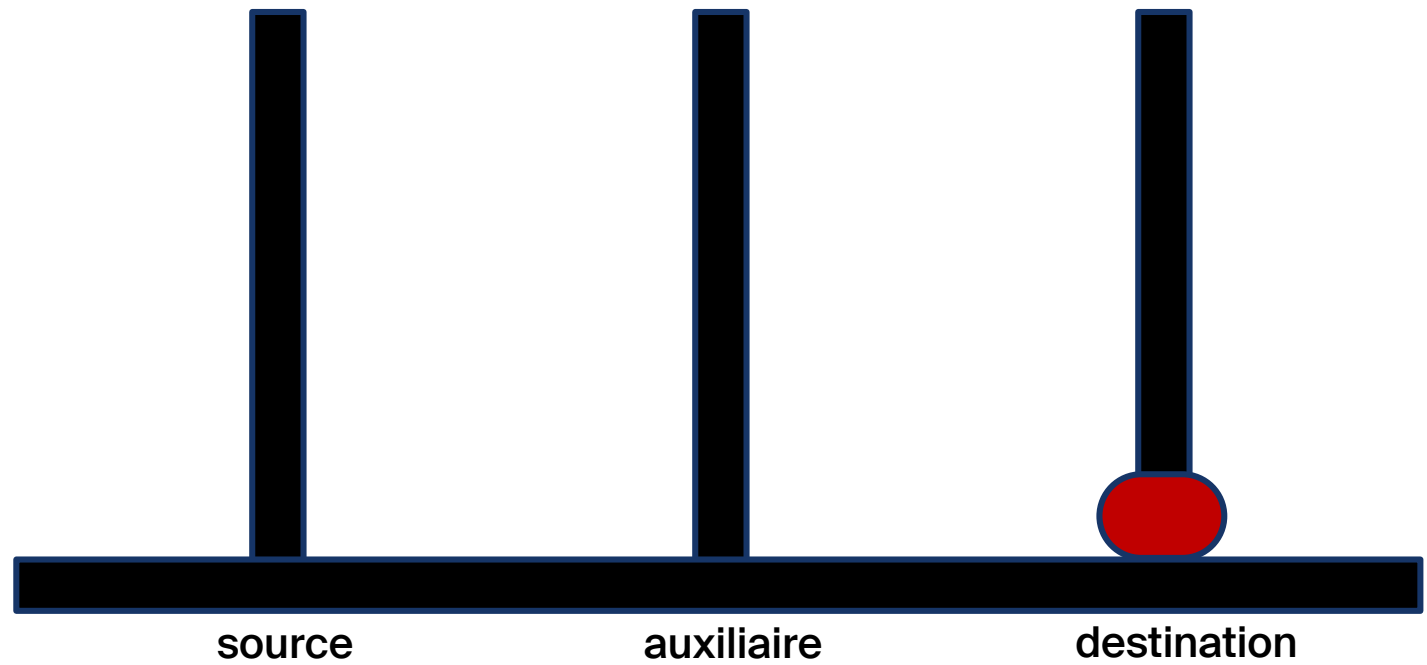
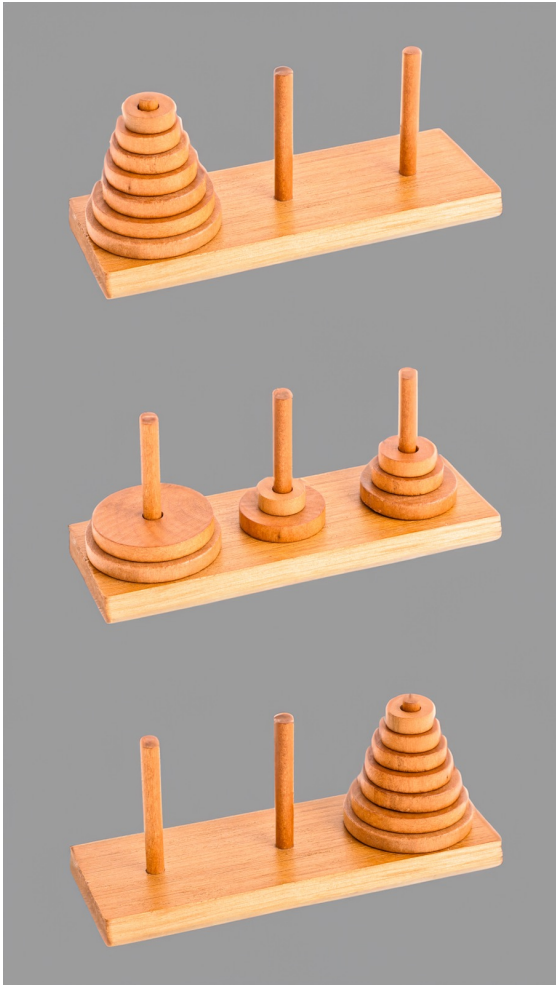
Problème : Tours de Hanoi



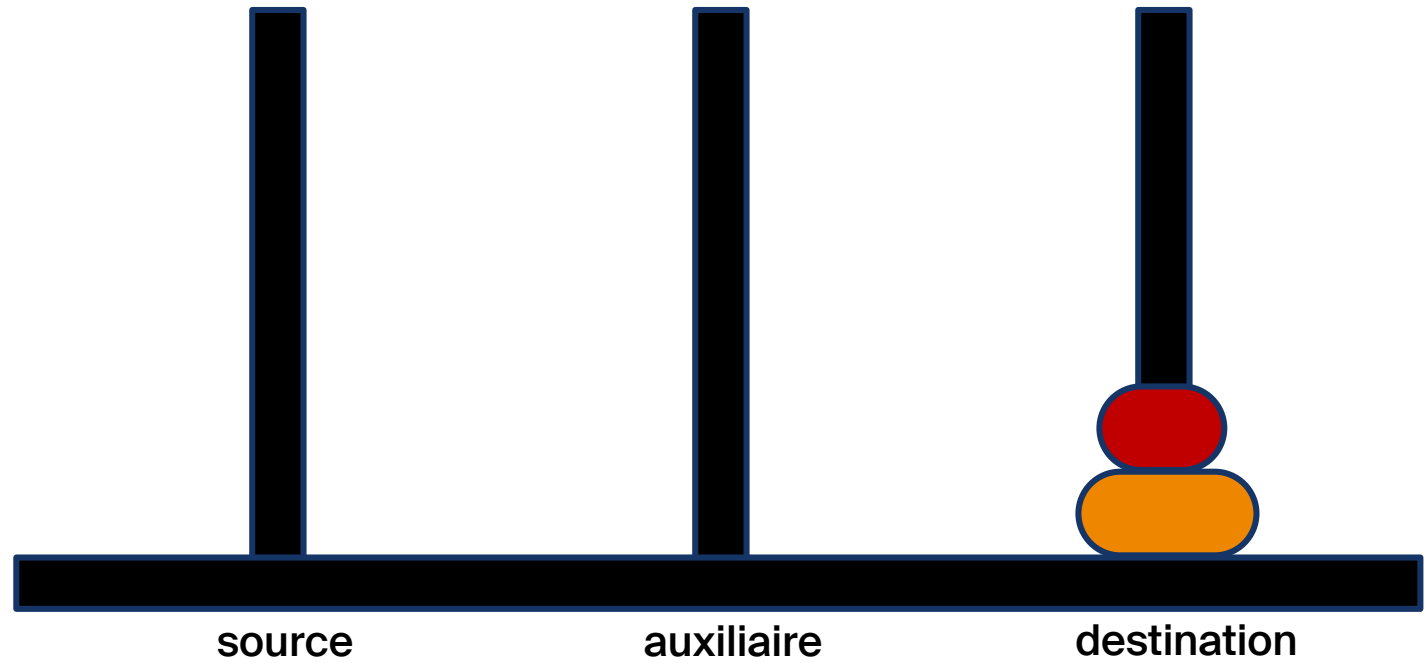
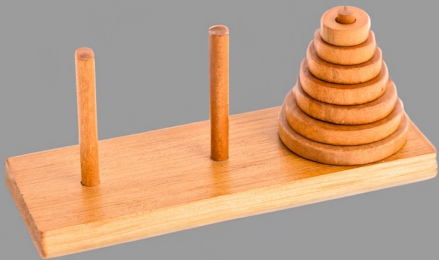
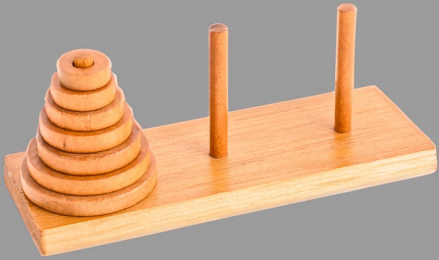
Règles :

- 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**

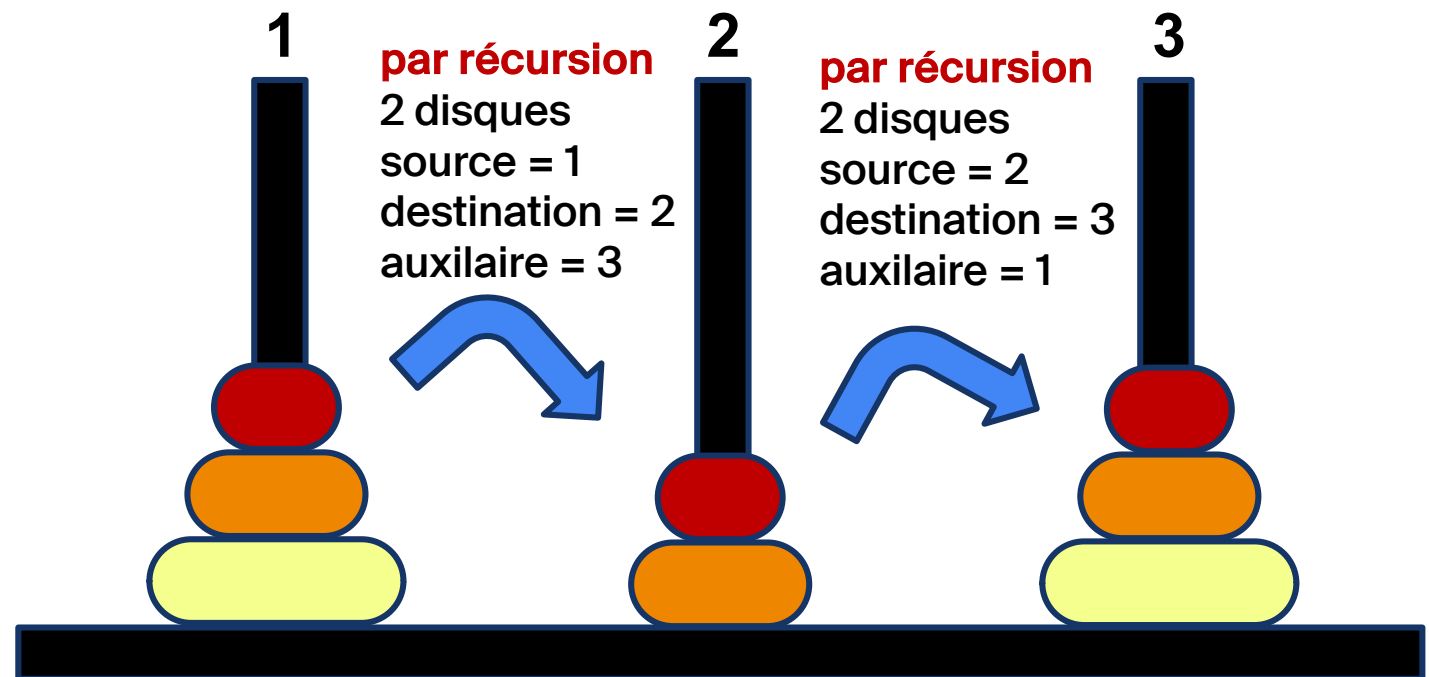
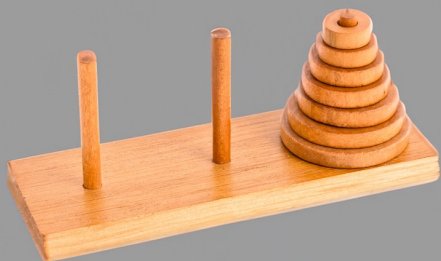
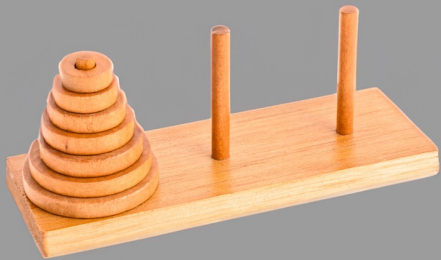
Problème : Tours de Hanoi ($n = 1$)



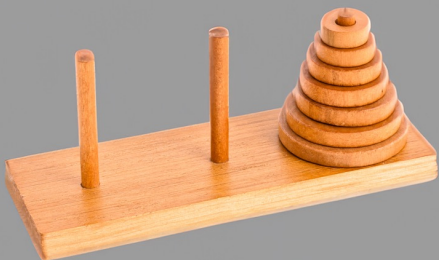
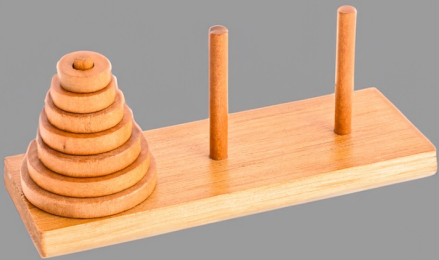
Problème : Tours de Hanoi ($n = 2$)



Problème : Tours de Hanoi ($n > 2$)



Tours de Hanoï : Algorithme



Tours de Hanoï

entrée : nombre de disques **n**, entier, $n > 0$

 pilier **source**, entier, $1 \leq \text{source} \leq 3$

 pilier **destination**, entier, $1 \leq \text{destination} \leq 3$

sortie : mouvements pour résoudre le problème des Tours de Hanoï

auxiliaire $\leftarrow 6 - \text{source} - \text{destination}$

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 Hanoi

1 -> 3

1 -> 2

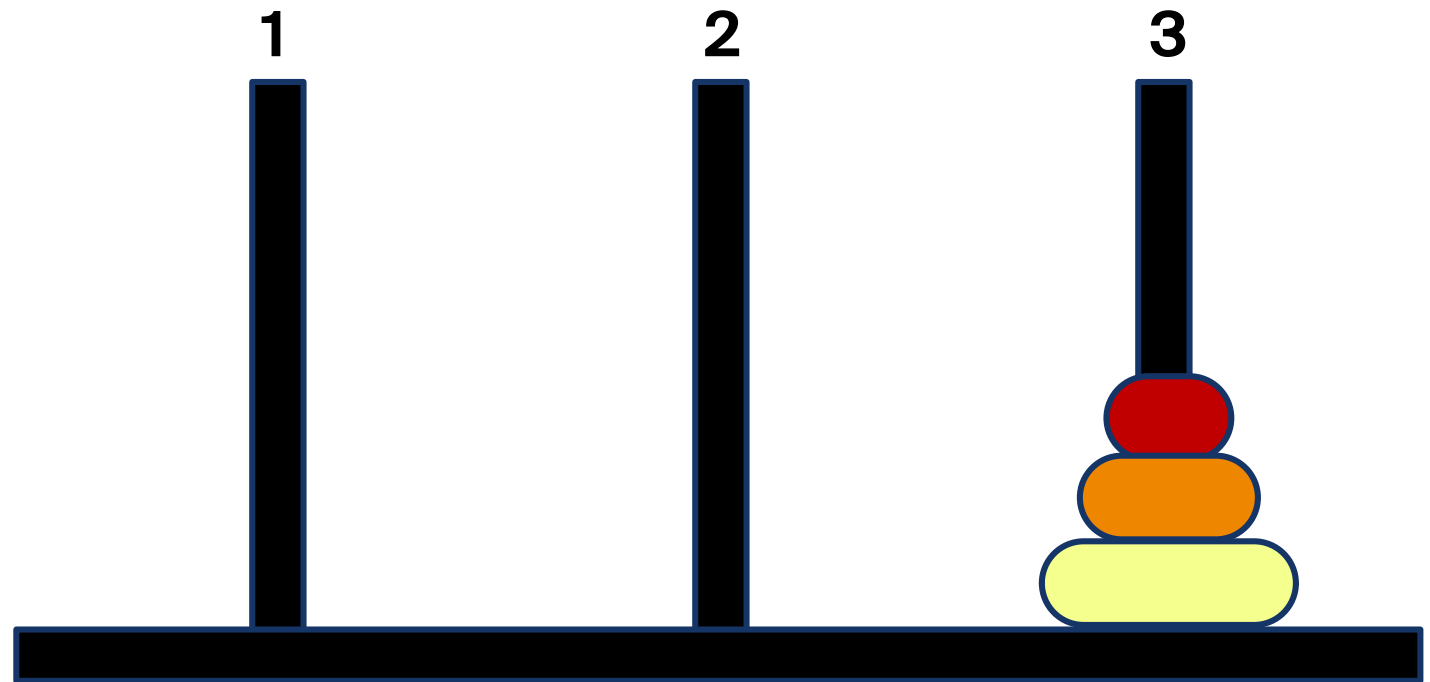
3 -> 2

1 -> 3

2 -> 1

2 -> 3

1 -> 3



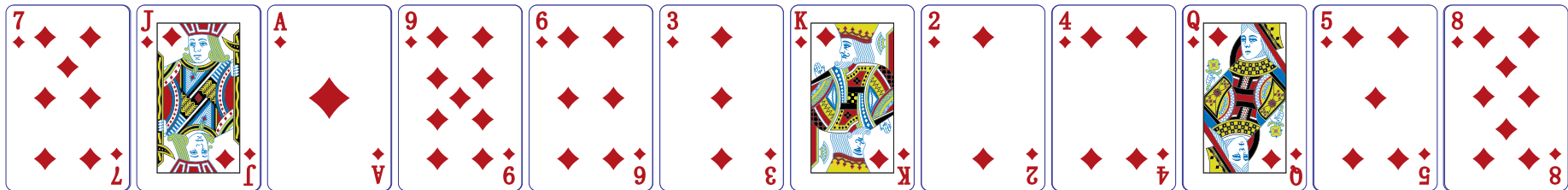
Aujourd'hui

- 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 ordonnée **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 $\leftarrow \left\lfloor \frac{n}{2} \right\rfloor$

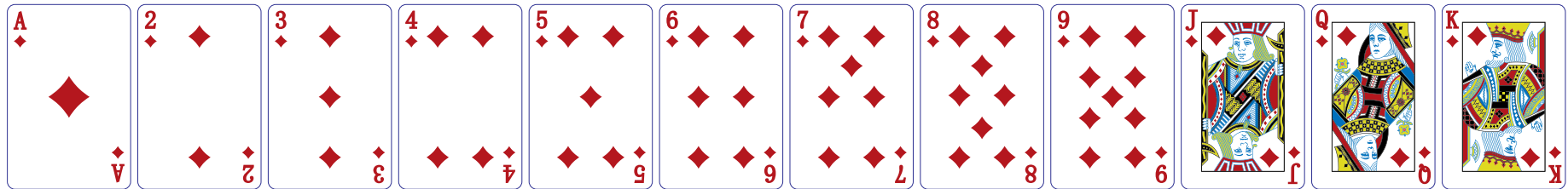
Si **x** ≤ **L**(**milieu**)

Sortir : Dichotomie(**L**(1 : **milieu**), **milieu**, **x**)

Sinon

Sortir : Dichotomie(**L**(1+**milieu** : **n**), **n** - **milieu**, **x**)

Recherche dichotomique



Dichotomie

entrée : Liste ordonnée **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)  **non**

milieu $\leftarrow \left\lfloor \frac{n}{2} \right\rfloor$

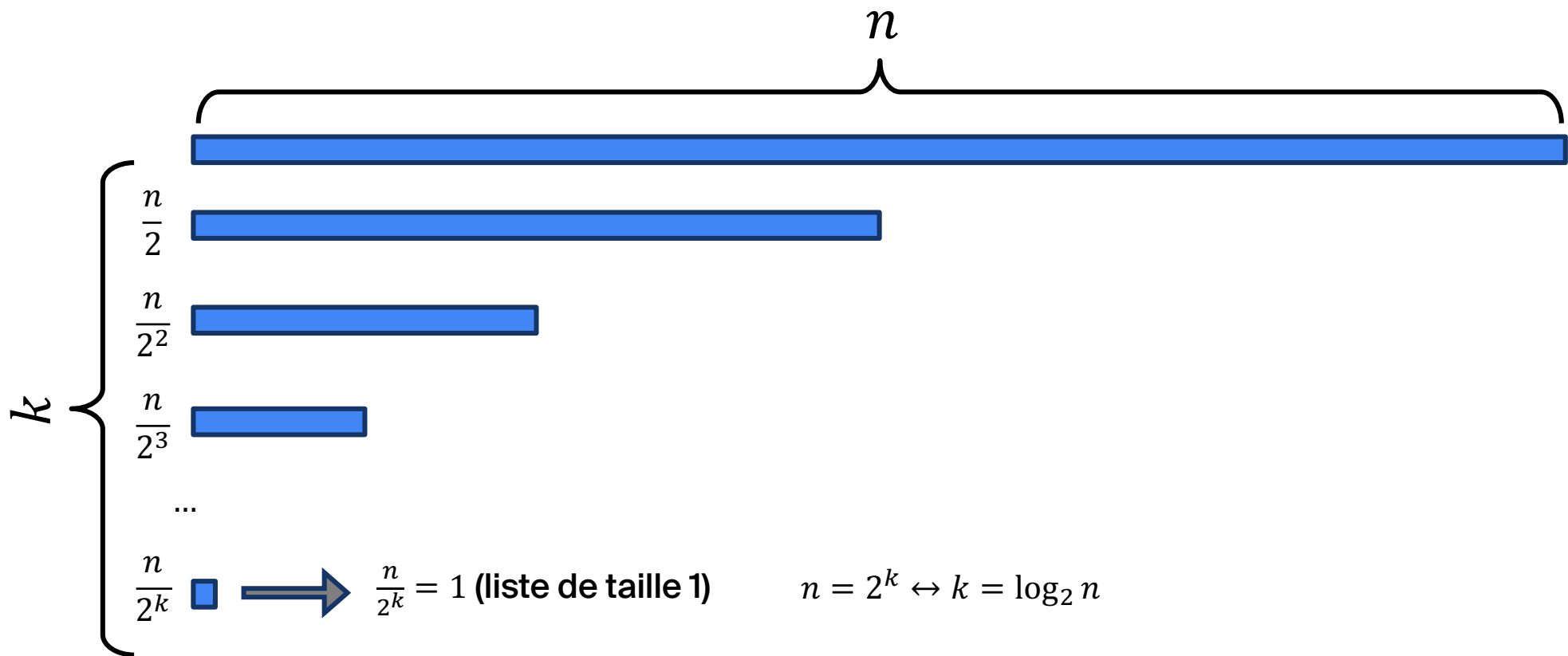
Si **x** ≤ **L**(**milieu**)

Sortir : Dichotomie(**L**(1 : **milieu**), **milieu**, **x**)

Sinon

Sortir : Dichotomie(**L**(1+**milieu** : **n**), **n** - **milieu**, **x**)

Complexité temporelle : Recherche dichotomique

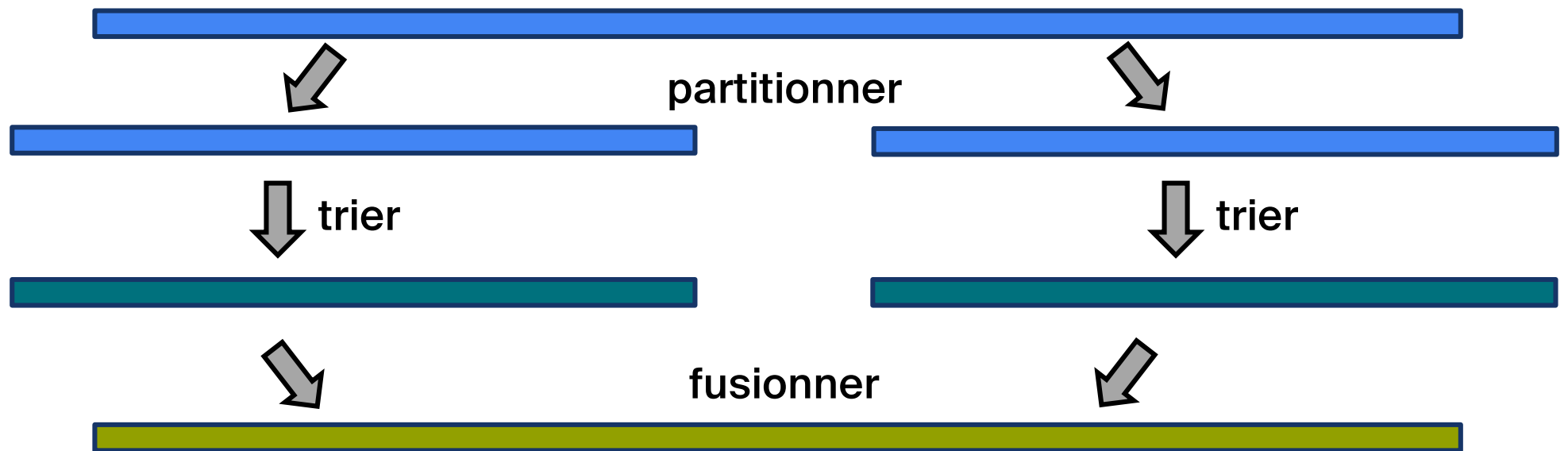


Complexité : $\Theta(\log_2 n)$

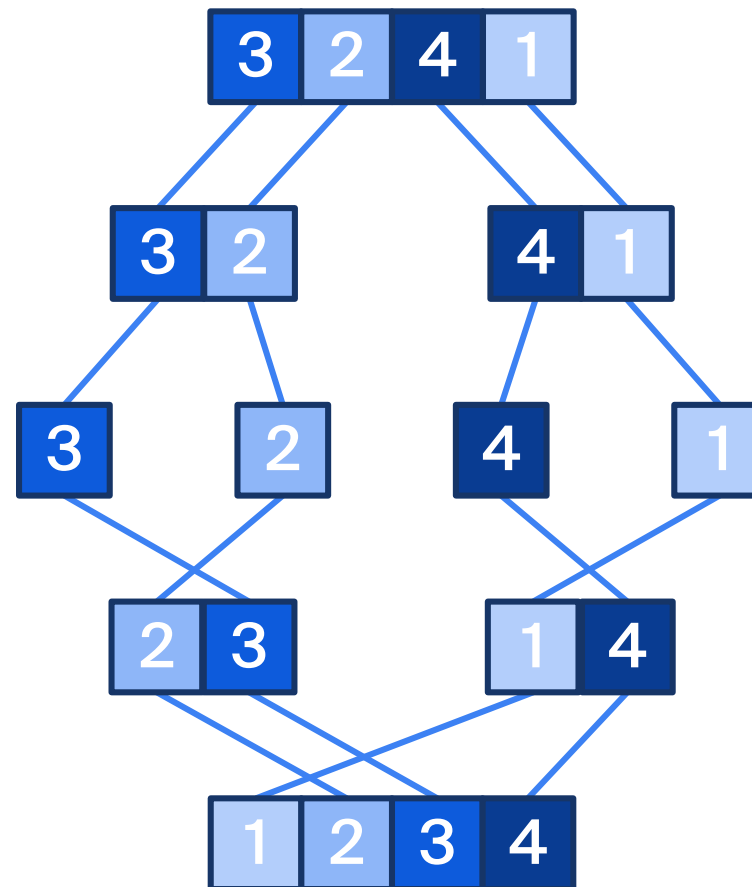
Aujourd'hui

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

Tri par fusion



Tri par fusion

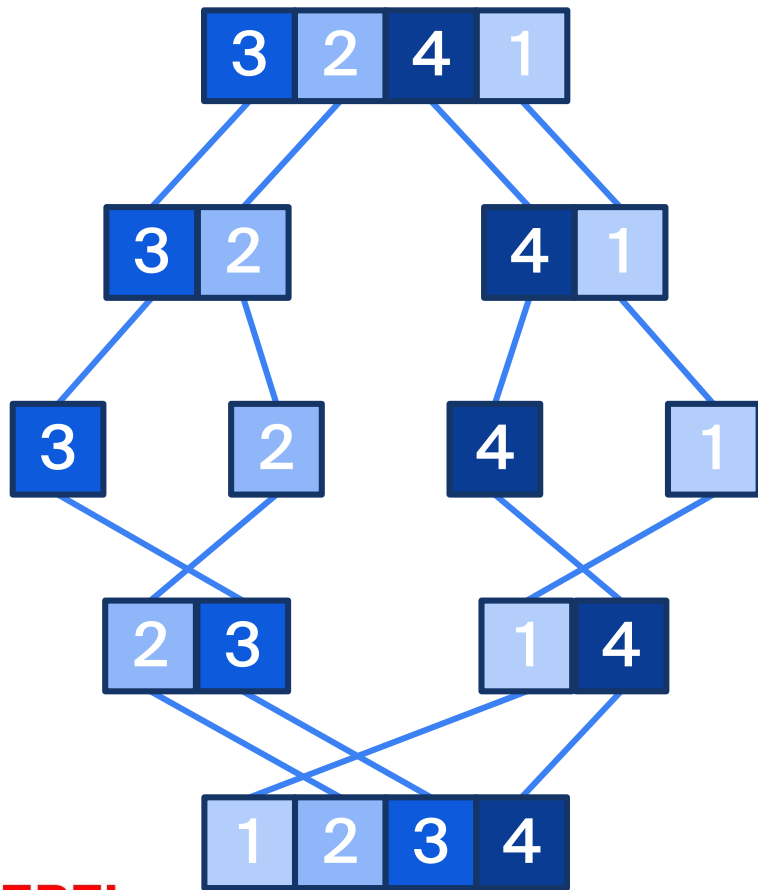


Listes de taille 1 (**triées**)

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 $\leftarrow \left\lfloor \frac{n}{2} \right\rfloor$

L₁ \leftarrow Tri par fusion(**L**(1 : **milieu**), **milieu**)

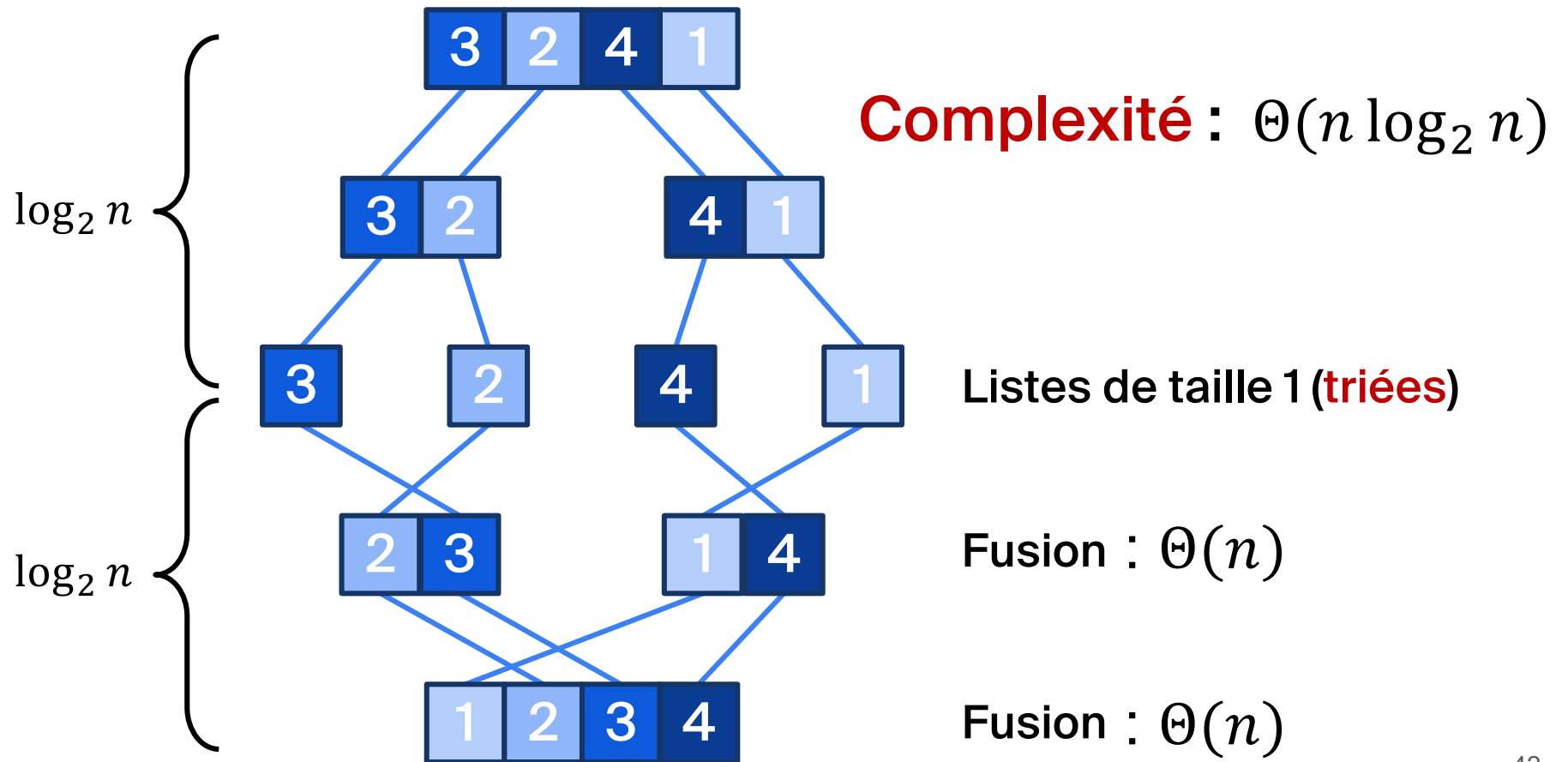
L₂ \leftarrow Tri par fusion(**L**(1+**milieu** : **n**), **n** - **milieu**)

L' \leftarrow fusion(**L**₁, **L**₂)

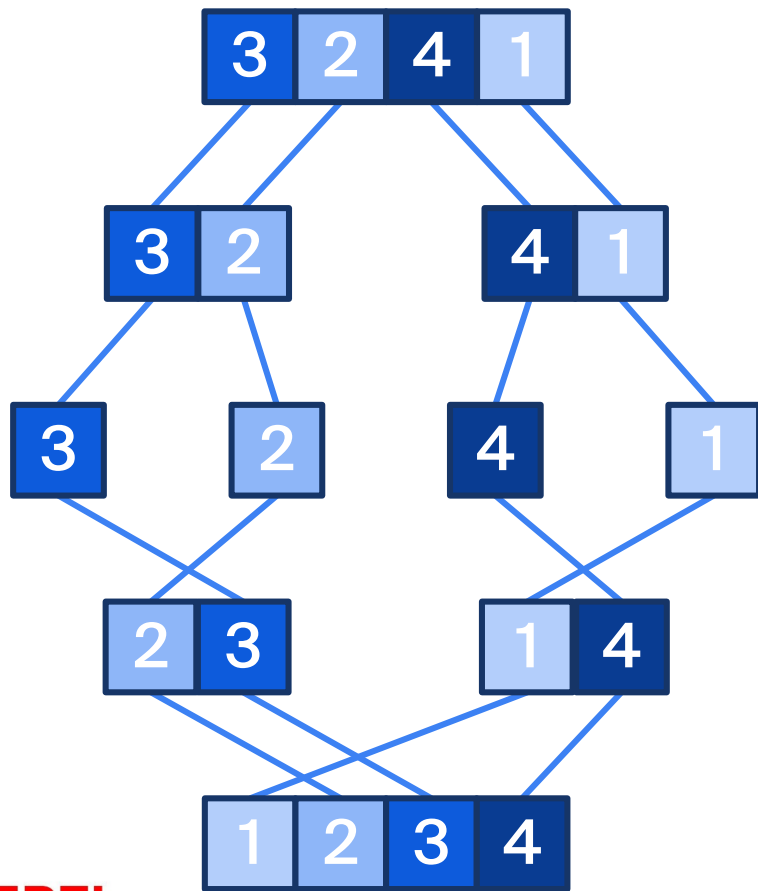
Sortir : **L'**

Quelle est la **complexité** ?

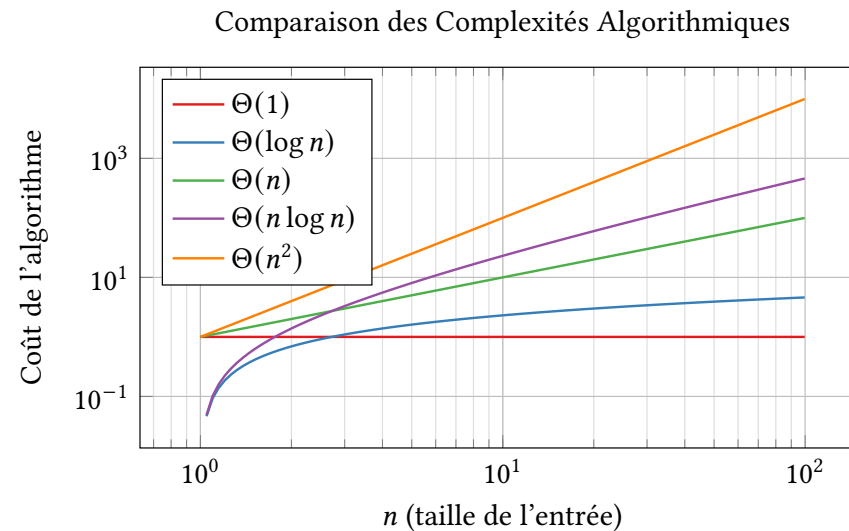
Complexité temporelle : Tri par fusion



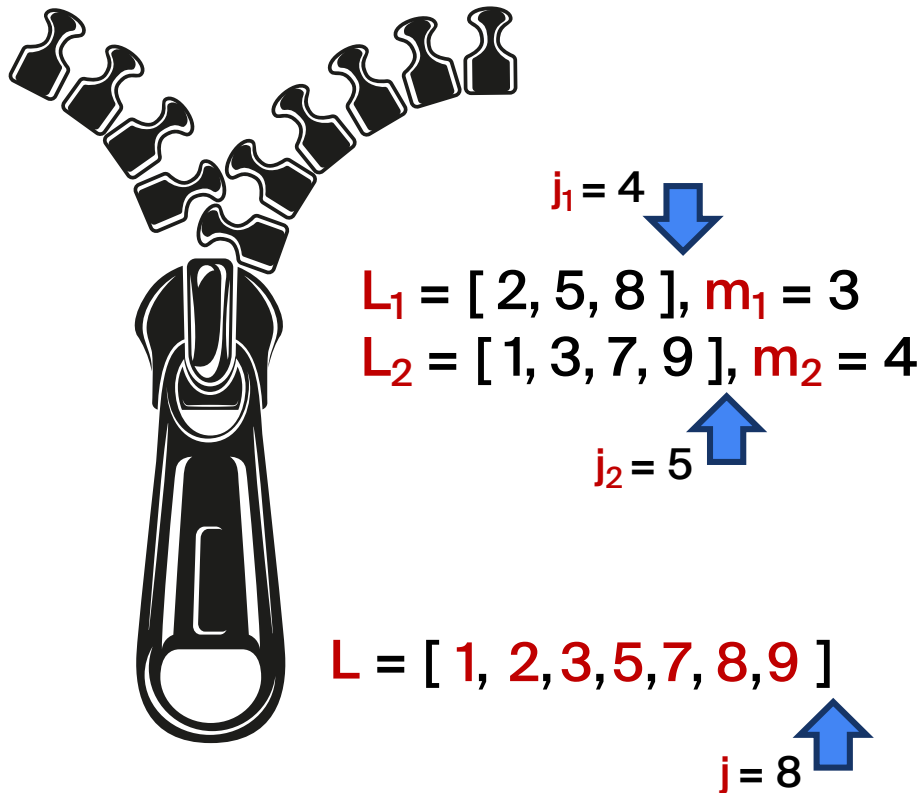
Complexité temporelle : Tri par fusion



Complexité : $\Theta(n \log_2 n)$



Tri par fusion : fermeture éclair



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

$j_1 \leftarrow 1$
 $j_2 \leftarrow 1$
 $j \leftarrow 1$

Tant que $j_1 \leq m_1$ et $j_2 \leq m_2$:

Si $L_1(j_1) \leq L_2(j_2)$:

$L(j) \leftarrow L_1(j_1)$

$j_1 \leftarrow j_1 + 1$

Sinon :

$L(j) \leftarrow L_2(j_2)$

$j_2 \leftarrow j_2 + 1$

$j \leftarrow j + 1$

Si $j_1 = m_1 + 1$:

Tant que $j_2 \leq m_2$:

$L(j) \leftarrow L_2(j_2)$

$j_2 \leftarrow j_2 + 1$

$j \leftarrow j + 1$

Sinon :

Tant que $j_1 \leq m_1$:

$L(j) \leftarrow L_1(j_1)$

$j_1 \leftarrow j_1 + 1$

$j \leftarrow j + 1$

Sortir : L (indicated by a red arrow pointing left)

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 $\leftarrow \lfloor \frac{n}{2} \rfloor$
L₁ \leftarrow Tri par fusion(**L**(1 : **milieu**), **milieu**)
L₂ \leftarrow Tri par fusion(**L**(1+**milieu** : **n**), **n** - **milieu**)
L' \leftarrow fusion(**L**₁, **L**₂)
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 \leftarrow 1
j₂ \leftarrow 1
j \leftarrow 1

Tant que **j**₁ \leq **m**₁ et **j**₂ \leq **m**₂ :
Si **L**₁(**j**₁) \leq **L**₂(**j**₂) :
 L(**j**) \leftarrow **L**₁(**j**₁)
 j₁ \leftarrow **j**₁ + 1
Sinon :
 L(**j**) \leftarrow **L**₂(**j**₂)
 j₂ \leftarrow **j**₂ + 1
j \leftarrow **j** + 1

Si **j**₁ = **m**₁ + 1 :
 Tant que **j**₂ \leq **m**₂ :
 L(**j**) \leftarrow **L**₂(**j**₂)
 j₂ \leftarrow **j**₂ + 1
 j \leftarrow **j** + 1

Sinon :
 Tant que **j**₁ \leq **m**₁ :
 L(**j**) \leftarrow **L**₁(**j**₁)
 j₁ \leftarrow **j**₁ + 1
 j \leftarrow **j** + 1

Sortir : **L**

Résumé Cours 3 – 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

EPFL



Merci