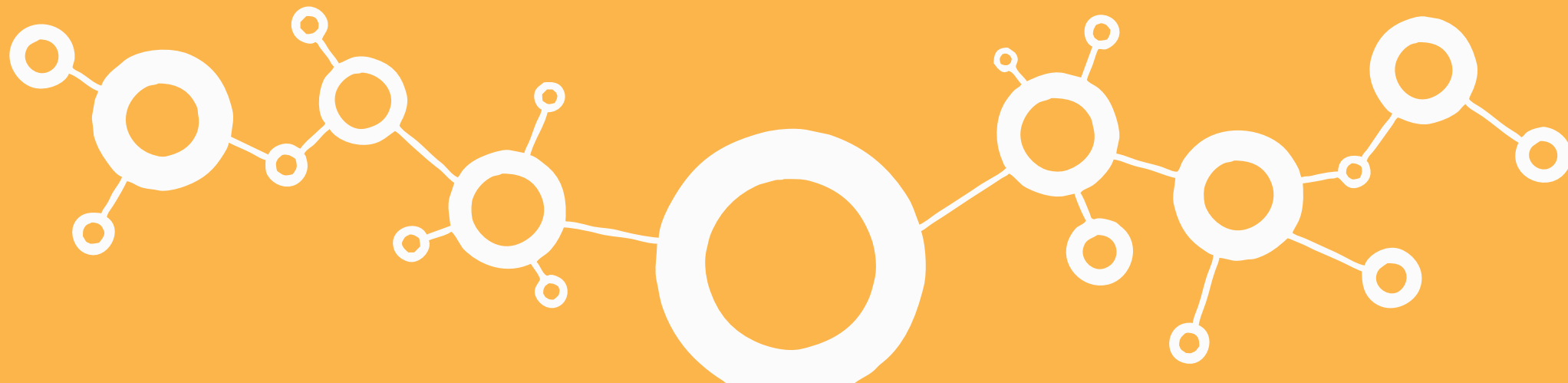




## Ch. 3 : Récursivité / Diviser pour régner



Récurtivité



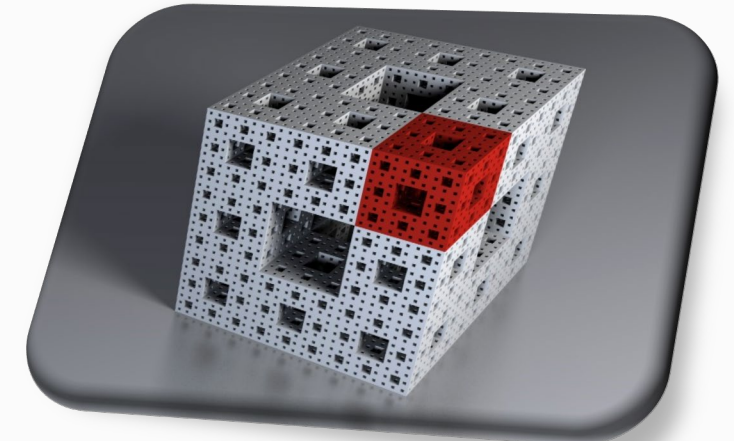
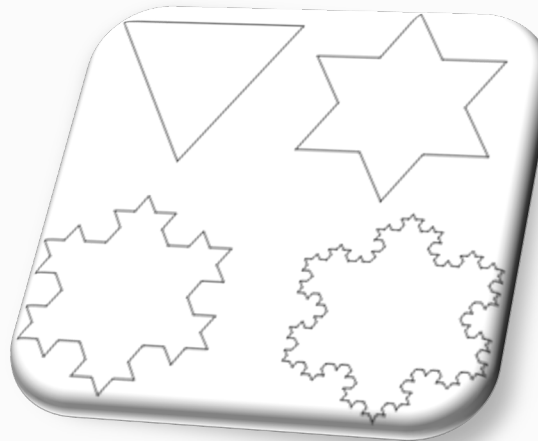
# Récurtivité

Observée fréquemment

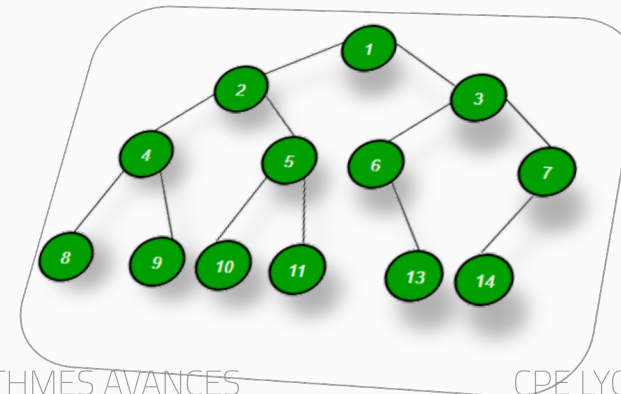
- dans la **nature**  
chou romanesco, nautilus



- en **maths**  
fractales, flocon de Koch  
éponge de Menger



- en **informatique**  
algorithmes  
structures de données arborescentes



# Fonction récursive

- Proche de la notion de *suite définie par récurrence* en maths :

$$\begin{cases} u_0 = 2 & \text{Définition du cas de base} \\ u_{n+1} = 2u_n + 3 & \text{Définition du cas général} \end{cases}$$

- Une fonction *récursive* est une fonction **qui s'appelle elle-même**, sur des entrées **plus petites**
  - ⇒ Intérêt : écrire plus facilement certaines fonctions
  - ⇒ Pourquoi sur des entrées *plus petites* ?
  - ⇒ Ne jamais oublier le cas de base / condition d'arrêt !
- **Exemple** : la fonction *factorielle* définie sur  $\mathbb{N}$  :

$$\begin{cases} 0! = 1 \\ n! = n \times (n - 1)! \end{cases}$$





# Fonction récursive

- Implémentation en Python :

```
def factorielle(n):  
    if n == 0:  
        return 1  
    else:  
        return n * factorielle(n-1)  
  
print(factorielle(3))      --> 6  
print(factorielle(5))      --> 120  
print(factorielle(1))      --> 1  
print(factorielle(0))      --> 1  
print(factorielle(-1))     --> Boucle infinie !!!
```

Comment corriger l'erreur simplement ?



# Fonction récursive

Remarque : cette version sans 'else' est aussi valide :

```
def factorielle(n):  
    if n == 0:          # condition d'arrêt / cas de base  
        return 1  
  
    return n * factorielle(n-1)    # cas général
```

⚠ La condition d'arrêt doit toujours être testée avant l'appel récursif ! Comparez avec le code suivant :

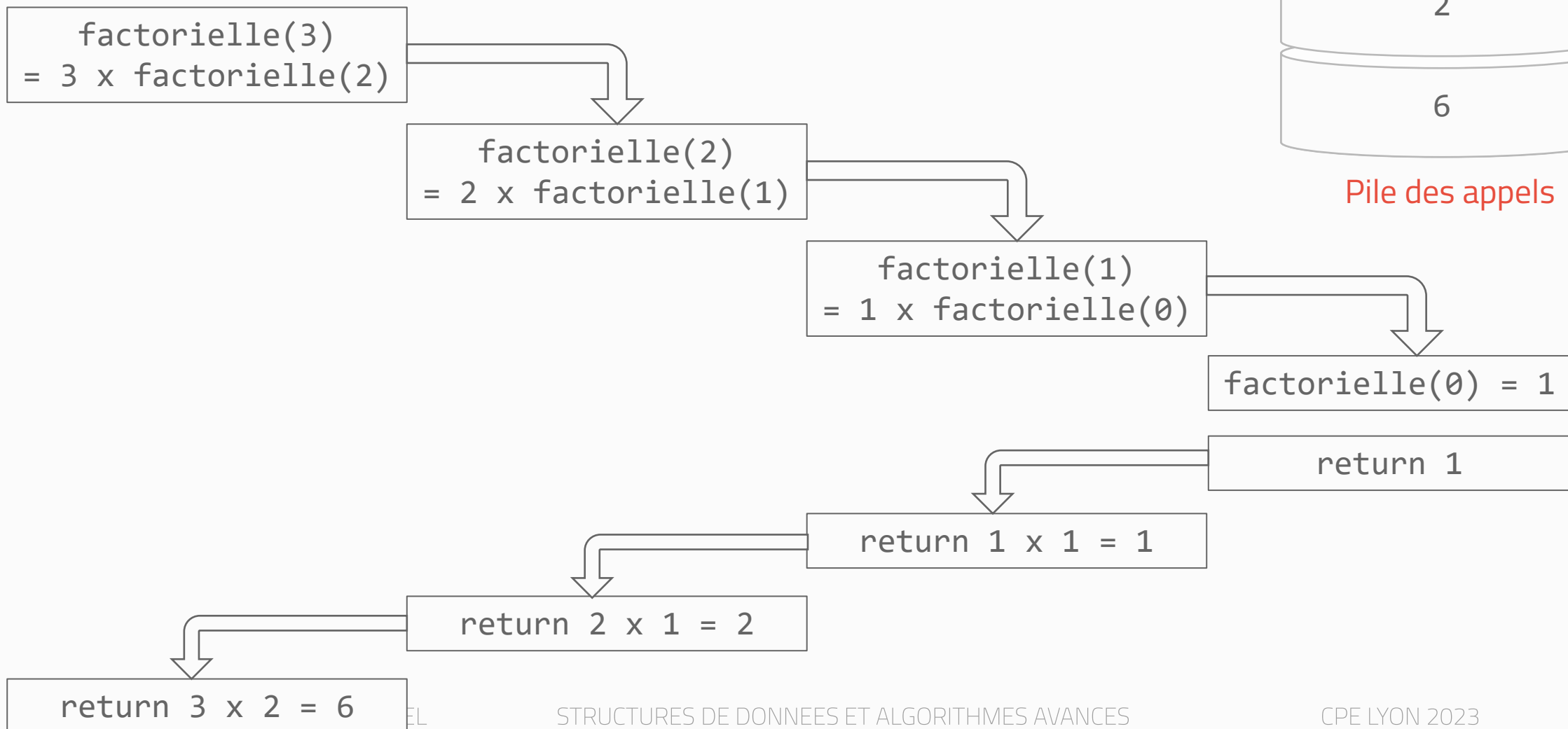
```
def factorielle(n):  
    return n * factorielle(n-1)    # cas général  
  
    if n == 0:          # condition d'arrêt / cas de base  
        return 1
```

⇒ Plus sûr de laisser le cas général dans le 'else'



# Fonction récursive

Principe : calcul de **factorielle(3)**





# Fonction récursive

Chaque appel récursif produit un nouveau **contexte d'exécution** qui lui est propre :

- L'adresse mémoire de la fonction appelante
- État de la mémoire
- Valeur des paramètres, des variables

La **pile** sert à sauvegarder temporairement les contextes d'exécution des appels précédents

- Elle est gérée automatiquement par le système d'exploitation
- Elle a une capacité limitée et peut déborder si on fait trop d'appels !

⇒ cf. erreur avec `factorielle(-1)`

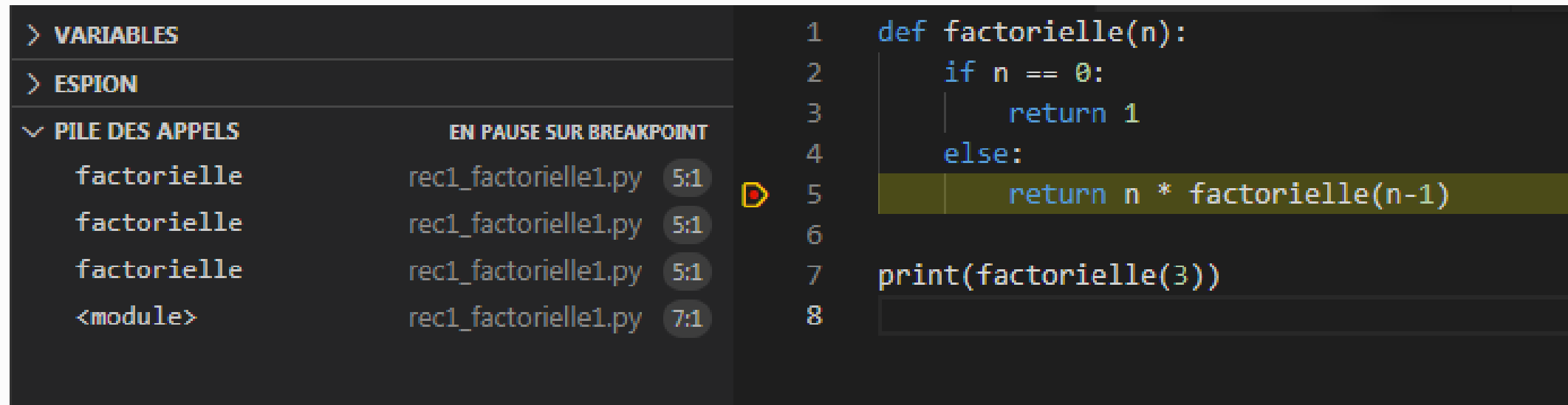


# Fonction récursive

💡 Le site [pythontutor.com](https://pythontutor.com) permet de visualiser ce fonctionnement de manière interactive

[Exemple de la factorielle](#)

💡 On peut visualiser la pile des appels avec le débogueur de VS Code (ou tout autre éditeur) :



The screenshot shows the VS Code interface with a Python file named `rec1_factorielle1.py`. The code defines a recursive function `factorielle(n)` and calls it with `print(factorielle(3))`. The call stack on the left shows four frames: three for `factorielle` and one for the `<module>`. The current frame is `factorielle` at line 5, column 1, which is highlighted in yellow. The code is as follows:

```
1 def factorielle(n):
2     if n == 0:
3         return 1
4     else:
5         return n * factorielle(n-1)
6
7 print(factorielle(3))
8
```

# Ré recursions terminale et non terminale

- Une fonction réursive est **terminale** si l'appel récuratif est la **seule** instruction dans le *return* :

```
def recursionTerminale(n):  
    // ...  
    return recursionTerminale(n - 1)
```

- Une fonction réursive est **non terminale** sinon :

```
def recursionNonTerminale(n):  
    // ...  
    return n + recursionNonTerminale(n - 1)
```

Exemple : la fonction **factorielle** précédente était **non terminale**



# Ré recursions terminale et non terminale

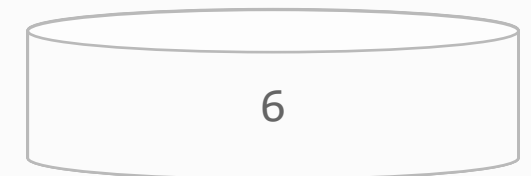
Exemple : la fonction `factorielle` précédente était **non terminale**  
⇒ on peut la transformer simplement en fonction récursive terminale :

```
def factorielle(n, resultat):  
    if n == 0:  
        return resultat  
    else:  
        return factorielle(n-1, n * resultat)  
  
factorielle(3,1)      # -> 6
```

💡 **Avantage** : on n'a plus besoin de stocker tous les résultats intermédiaires sur la pile



Ré cursion non terminale



Ré cursion terminale



# Itératif vs. récursif

Toute fonction récursive peut être transformée en fonction itérative (et réciproquement)

- Récursif → Itératif

Demande de gérer manuellement et explicitement une pile

- Itératif → Récursif

L'itération peut être remplacée facilement par une récursion terminale



# Itératif vs. récursif

## Transformation d'une fonction récursive en fonction itération

```
def function_non_recursive(inputs):  
    CALL, HANDLE = 0, 1  
    call_stack = [(CALL, inputs)]  
    return_stack = []  
    while call_stack:  
        action, data = call_stack.pop()  
        if action == CALL:  
            ... # 4  
            call_stack.append((HANDLE, some_data)) # 3  
            call_stack.append((CALL, some other data)) # 2  
            return_stack.append(some other data) # 1  
            call_stack.append((CALL, some_data)) # 1  
        else: # HANDLE  
            pop items from return_stack  
            use them to calculate something  
            and push that something to return_stack  
    return return_stack[-1] # return top value from the return_stack
```

Pile des appels  
Pile des valeurs de retour

L'ordre des HANDLE / CALL  
peut varier selon l'algo.

C'est ici qu'on "consomme"  
les valeurs de la pile de  
retour



Toute fonction récursive peut être transformée en fonction itérative (et réciproquement)

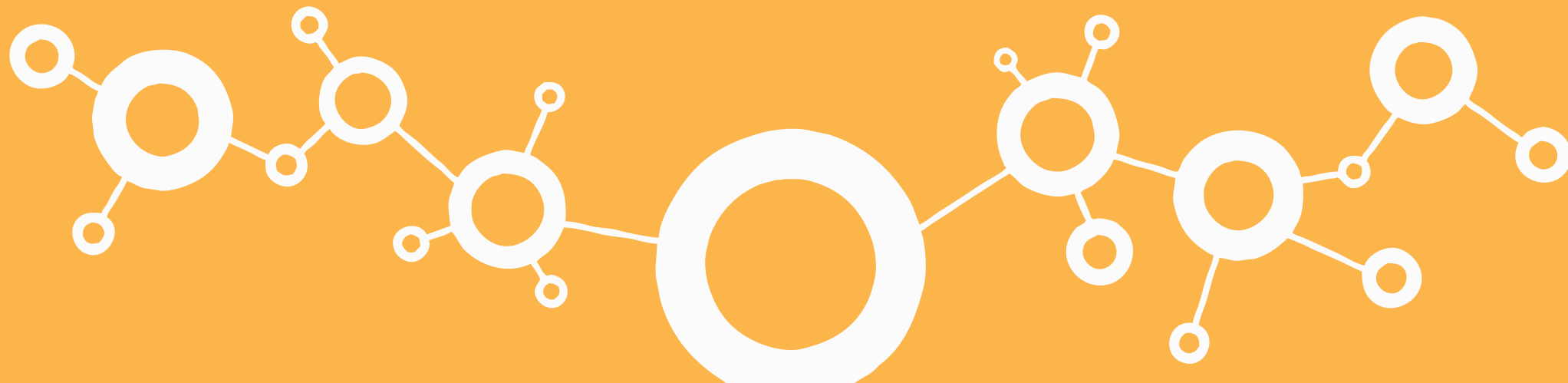
## ✓ Avantages du récursif

- Fonctions plus lisibles et plus élégantes *une fois écrites* (ex. Tours de Hanoï)
- Plus naturel dans les algorithmes qui font intervenir du *backtracking* ou du *diviser pour régner*
- Pile d'exécution gérée automatiquement

## ✗ Inconvénients du récursif

- Plus difficile à appréhender quand on n'a pas l'habitude
- Temps d'exécution plus élevé (gestion des appels de fonction et des contextes d'exécution)
- Dans certains cas, la pile système peut être trop petite





Diviser pour régner



# Diviser pour régner

Technique algorithmique consistant à :

1. **Diviser** : découper un problème initial en sous-problèmes
2. **Régner** : résoudre les sous-problèmes (récursivement, ou directement s'ils sont assez petits)
3. **Combiner** : calculer une solution au problème initial à partir des solutions des sous-problèmes

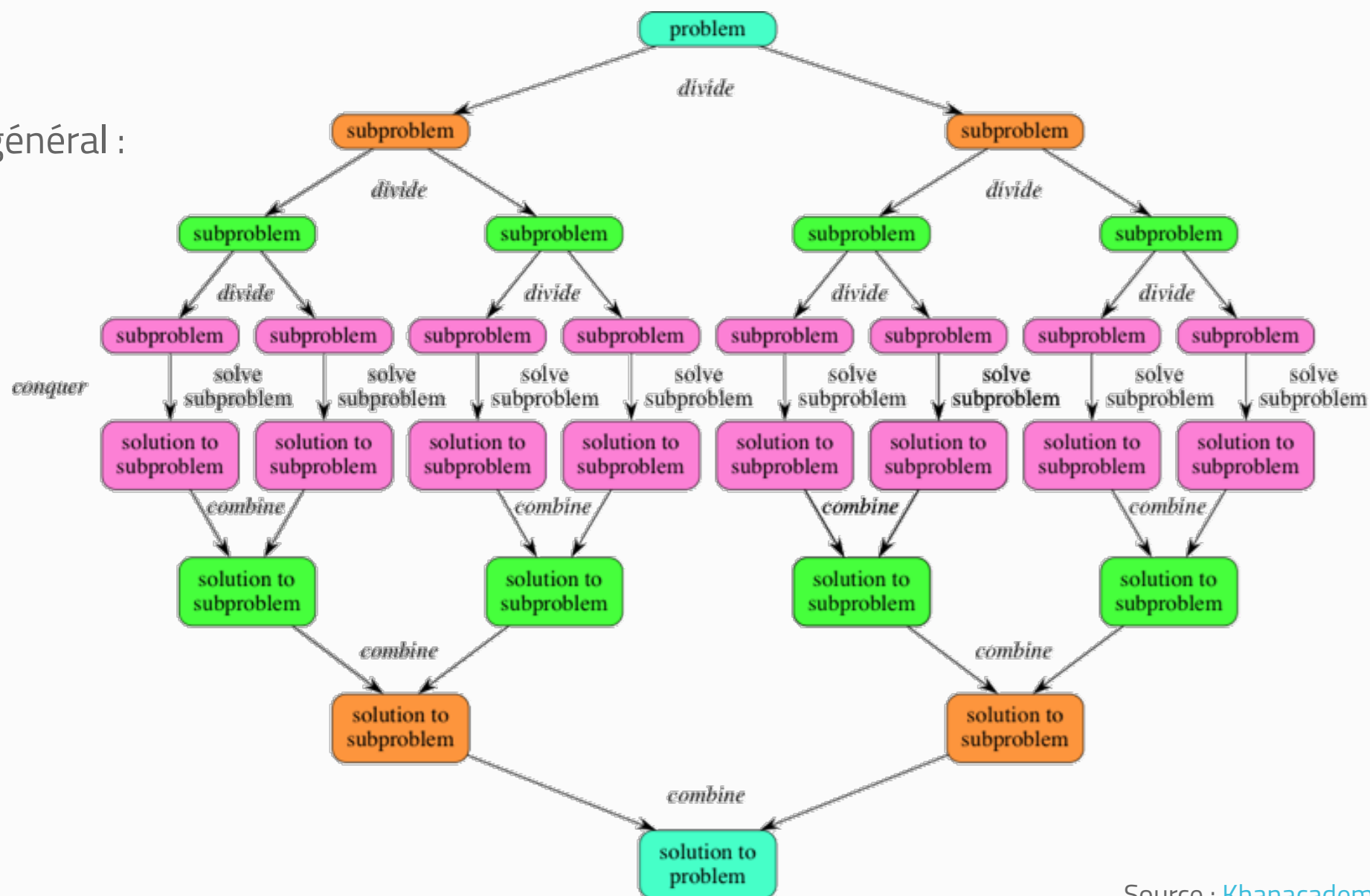
## Intérêts :

- Permet de résoudre simplement des problèmes difficiles (ex. : tours de Hanoï)
- Entraîne souvent une meilleure complexité algorithmique
- Facilement parallélisable
- Moins sujet aux problèmes d'arrondis sur les calculs



# Diviser pour régner

Schéma général :

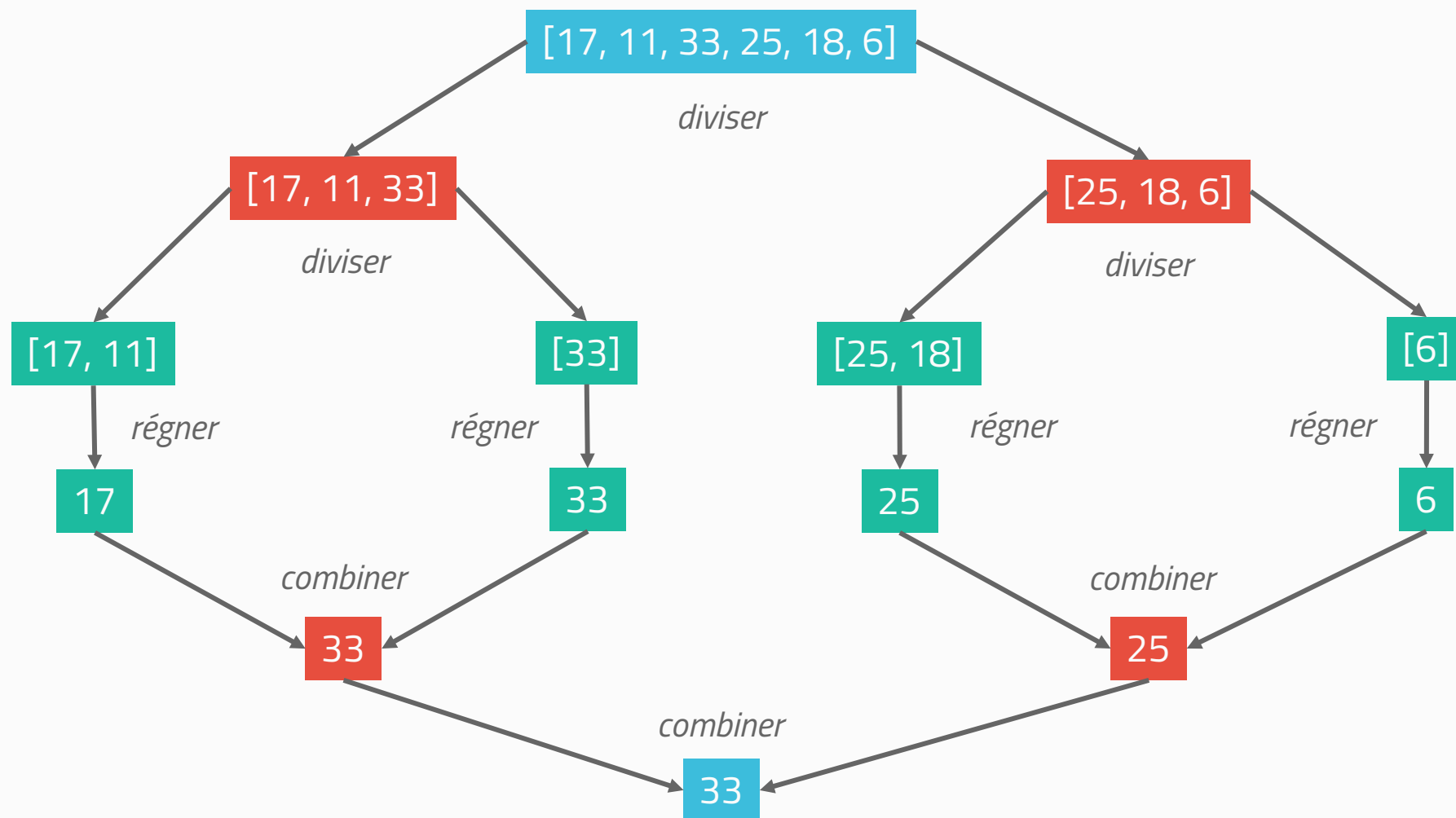


Source : [Khanacademy](https://www.khanacademy.com/)



# Diviser pour régner

Exemple : recherche du maximum dans le tableau non trié [17, 11, 33, 25, 18, 6]



**Algorithmes** basés sur le principe *Diviser pour régner* :

- Multiplication de grands entiers : [algorithme de Karatsuba](#) ( $O(n^{1,585})$  vs  $O(n^2)$ )
- Multiplication de matrices : [algorithme de Strassen](#) ( $O(n^{2,807})$  vs  $O(n^3)$ )
- Recherche des [deux points les plus proches](#) dans un ensemble de points ( $O(n \log n)$  vs  $O(n^2)$ )
- **Tri fusion** (cf. suite du cours)
- **Tri rapide** (cf. suite du cours)
- Transformée de Fourier rapide (FFT)

Comment calculer la complexité d'un algorithme récursif / Diviser pour régner ?





Complexité des algorithmes récurrents

# Un premier exemple

Reprenons le problème de la recherche du maximum dans un tableau non trié

- On a un algorithme itératif trivial, de complexité  $\Theta(n)$
- On a un algorithme récursif, décrit précédemment. Est-il meilleur que l'algorithme itératif ?

Description de l'algorithme récursif :

- cas de base (le tableau contient 1 seul ou 2 éléments) : on retourne le maximum
- cas général : on découpe récursivement le problème en deux sous-problèmes de taille identique
- combinaison des résultats : recherche du maximum de deux éléments

En résumé, si on note  $T(n)$  la complexité en temps de cet algorithme :

$$T(n) = \begin{cases} \Theta(1) & \text{si } n \leq 2 \\ 2T(n/2) + \Theta(1) & \text{si } n > 2 \end{cases}$$

$\Theta(1)$  : coût de la comparaison de deux éléments  
 $\Theta(1)$  : coût du découpage + coût de la combinaison

En toute rigueur, dans le cas général,  $T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(1)$



Pour de nombreux algorithmes de type Divide and Conquer, le nombre d'opérations effectuées s'écrit selon une équation de récurrence du type  $T(n) = aT(n/b) + f(n)$  où :

- $a \geq 1$  : nombre de sous-problèmes dans lesquels le problème est divisé à chaque itération
- $n/b$  (avec  $b > 1$ ) : taille de chaque sous-problème
- $f(n)$  est une fonction *positive* : nombre d'opérations pour subdiviser et recombinaison des solutions des sous-problèmes

💡 Mais il existe également des algorithmes de type Divide and Conquer dont la complexité s'écrit sous une forme différente



# Forme générale

## Exemples :

- un algorithme peut découper un problème en **sous-problèmes de tailles différentes**, par exemple 2/3 vs. 1/3 ; si les étapes de division / recombinaison prennent un temps linéaire, la complexité d'un tel algorithme est  $T(n) = T(2n/3) + T(n/3) + \Theta(n)$
- les sous-problèmes ne sont **pas nécessairement une fraction constante** de la taille du problème original : une version récursive de la *recherche séquentielle* pourrait créer un sous-problème contenant systématiquement un élément de moins que le problème précédent : la complexité de cet algorithme est donc  $T(n) = T(n - 1) + \Theta(1)$  (**exercice**)
- On pourrait imaginer un algorithme de complexité  $T(n) = 2^n T(n/2) + (2 - \cos n)$

Comment résoudre ces relations de récurrence pour obtenir une expression asymptotique  $O(\dots)$  ?





# 1<sup>ère</sup> méthode : itération

Exemple : Recherche du maximum dans un tableau non trié :  $T(n) = 2T(n/2) + \Theta(1)$

D'après cette relation, on a :  $T(n) = 2T(n/2) + k$  ( $k$  est une constante)

$$= 2(2(T(n/4) + k)) + k$$

$$= 4T(n/4) + 3k$$

$$= 8T(n/8) + 7k$$

$$= \dots$$

$$= 2^i T(n/2^i) + (2^i - 1) \times k \quad (\text{où } i = \log_2 n)$$

$$= 2^{\log_2 n} T(n/2^{\log_2 n}) + (2^{\log_2 n} - 1) \times k$$

$$= n \times T(1) + (n - 1) \times k$$

Or  $T(1) = T(2) = k'$  (constante). Donc :

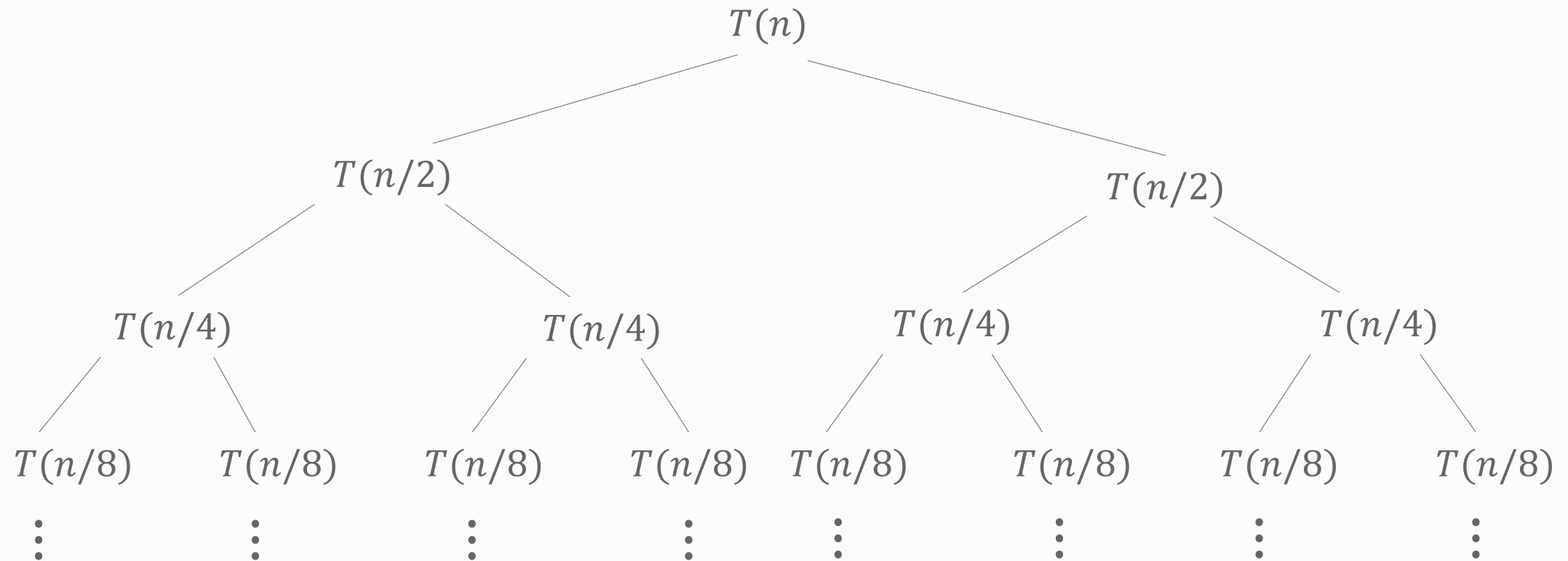
$$T(n) = n \times k' + (n - 1) \times k = \Theta(n)$$



## 2<sup>ème</sup> méthode : arbre de récursion

Exemple : Recherche du maximum dans un tableau non trié :  $T(n) = 2T(n/2) + k$

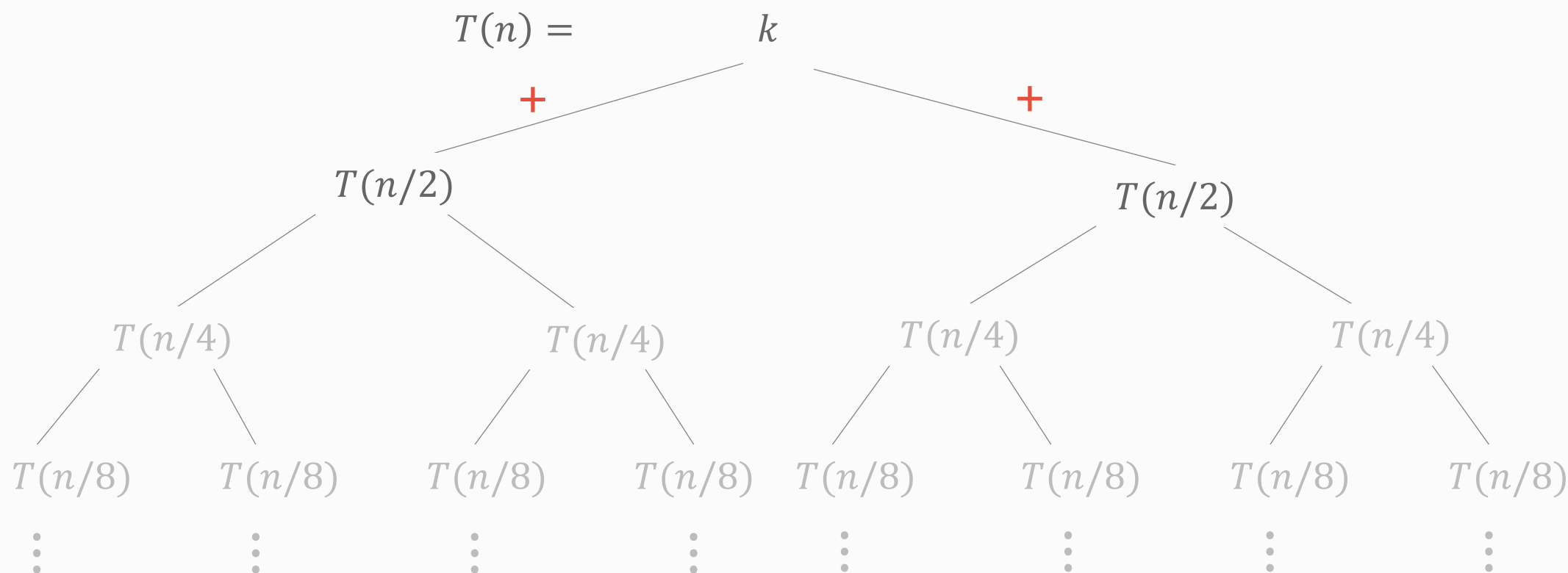
Chaque problème est séparé ici en deux problèmes de taille identique :



## 2<sup>ème</sup> méthode : arbre de récursion

Exemple : Recherche du maximum dans un tableau non trié :  $T(n) = 2T(n/2) + k$

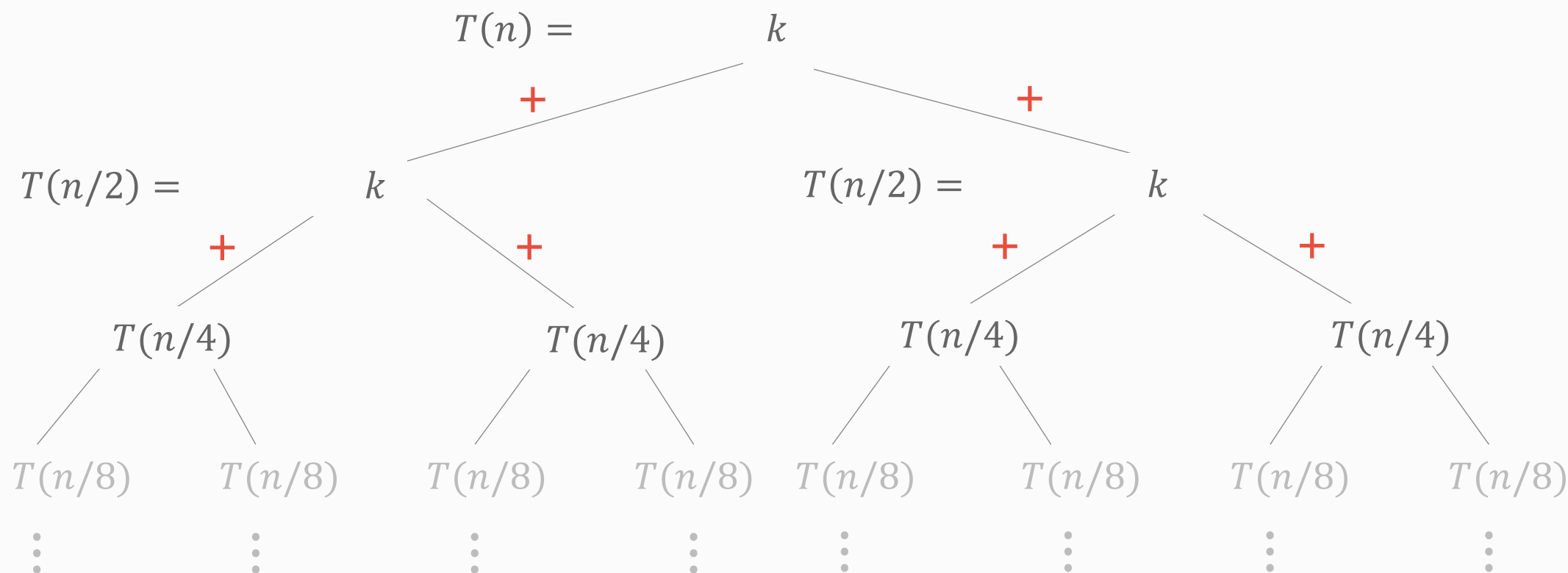
Chaque problème est séparé ici en deux problèmes de taille identique :



## 2<sup>ème</sup> méthode : arbre de récursion

**Exemple :** Recherche du maximum dans un tableau non trié :  $T(n) = 2T(n/2) + k$

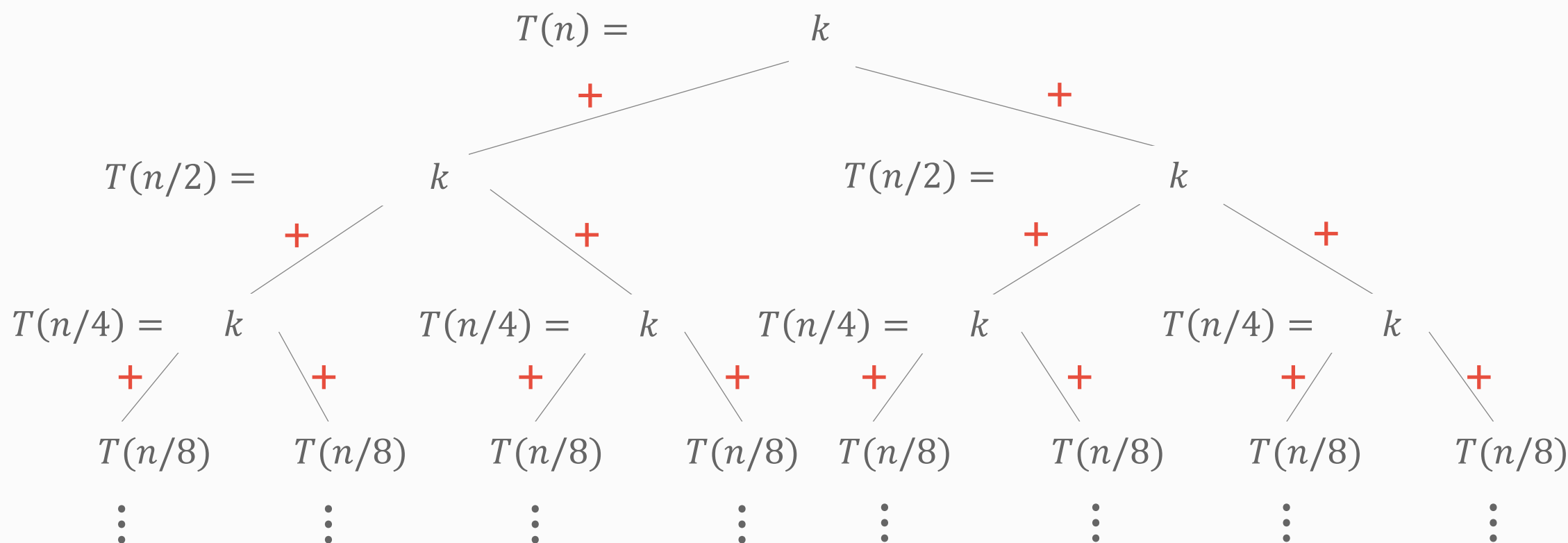
Chaque problème est séparé ici en **deux** problèmes de **taille identique** :



## 2<sup>ème</sup> méthode : arbre de récursion

Exemple : Recherche du maximum dans un tableau non trié :  $T(n) = 2T(n/2) + k$

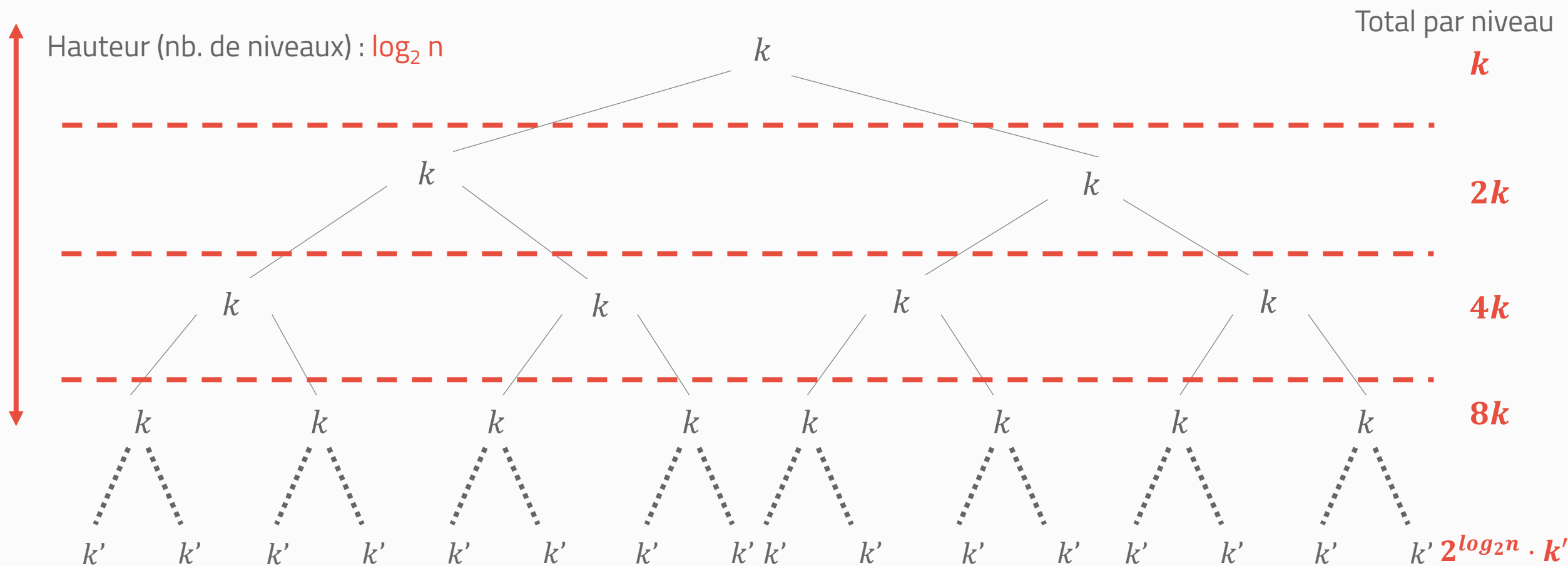
Chaque problème est séparé ici en deux problèmes de taille identique :



# 2<sup>ème</sup> méthode : arbre de récursion

Exemple : Recherche du maximum dans un tableau non trié :  $T(n) = 2T(n/2) + k$

Quelle est la somme de travail à chaque niveau ?



## 2<sup>ème</sup> méthode : arbre de récursion

**Exemple :** Recherche du maximum dans un tableau non trié :  $T(n) = 2T(n/2) + k$

Au total, on a donc :  $k + 2k + 4k + 8k + \dots + 2^{\log_2(n)-1}k + 2^{\log_2 n} \cdot k'$

$$= k(1 + 2 + 4 + 8 + \dots + 2^{\log_2(n)-1}) + n \cdot k'$$

La somme  $1 + 2 + 4 + 8 + \dots + 2^{\log_2(n)-1}$  est la somme des  $\log_2 n$  premiers termes de la suite  $(u_n)$  définie par  $u_n = 2 \times u_{n-1}$  et  $u_0 = 1$  ; il s'agit donc d'une suite *géométrique* (chaque terme est obtenu en multipliant le précédent par une constante), de *premier terme 1* et de *raison 2*.

Par conséquent, cette somme est égale à  $1 \times \frac{1-2^{\log_2 n}}{1-2} = 2^{\log_2 n} - 1 = n - 1$  (cf. fiche révisions)

On retrouve donc le résultat de la 1<sup>ère</sup> méthode :  $T(n) = (n - 1) \cdot k + n \cdot k' = \Theta(n)$



## 3<sup>ème</sup> méthode : substitution

### 2 étapes :

1. Estimer la forme de la solution (pas de recette miracle : expérience, intuition...)
2. Substituer cette solution dans la relation pour des valeurs plus petites (hypothèse de récurrence) pour trouver les constantes, et vérifier que la solution convient

**Exemple :** estimer une borne sup. du temps d'exécution pour la recherche du maximum dans un tableau non trié

- à chaque subdivision, les sous-problèmes deviennent 2x plus petits

⇒ combien de subdivisions en tout ?  $\log n$

- Au final,  $2^{\log n}$  cas de base, chacun prenant un temps constant

⇒ on peut donc faire l'hypothèse que  $T(n) = O(n)$

Rappel : on considère que les log sont toujours en base 2





## 3<sup>ème</sup> méthode : substitution

**Hypothèse :**  $T(n) = O(n)$ , i.e.  $T(n) \leq cn$  pour un  $c > 0$

**Réurrence :** On suppose l'hyp. vraie  $\forall m < n$  ; en particulier, pour  $m = n/2$ ,  $T(n/2) \leq \frac{cn}{2}$ .

Par *substitution*,  $T(n) = 2T(n/2) + k$

$$\leq \frac{2cn}{2} + k$$

$$= cn + k$$

$$= O(n)$$

Raisonnement faux !!!

Où est l'erreur ?

$\Rightarrow$  on n'a pas prouvé *exactement* notre hypothèse de départ, qui était  $T(n) \leq cn$  !



## 3<sup>ème</sup> méthode : substitution

Astuce : il faut modifier légèrement notre hypothèse !

Hypothèse :  $T(n) \leq cn - d$ , avec  $c > 0$  et  $d \geq 0$

Réurrence : On suppose l'hyp. vraie  $\forall m < n$  ; en particulier, pour  $m = n/2$ ,  $T(n/2) \leq \frac{cn}{2} - d$

Par substitution,  $T(n) = 2T(n/2) + k$

$$\leq 2 \left( \frac{cn}{2} - d \right) + k$$

$$= cn - 2d + k$$

$$\leq cn - d \quad \forall d \geq k$$

$$= O(n)$$



## 3<sup>ème</sup> méthode : substitution

⚠ Raisonement par récurrence : **ne pas oublier le cas de base !**

⇒ on doit vérifier que le/s cas de base satisfait/ont la relation trouvée

i.e. trouver le  $n_0$  t.q. la relation est satisfaite pour tout  $n \geq n_0$

Les cas de base sont ici :  $T(1) = T(2) = k'$

⇒ Il faut prouver que  $T(1) = k' \leq c - d$  et  $T(2) = k' \leq 2c - d$

⇒ Il suffit donc de prendre  $c \geq d + k'$

Nous venons donc de prouver que

$$\forall n \geq 1, \exists c > 0, T(n) \leq cn - d \text{ (où } d \text{ est une constante)}$$

$$\text{Donc } T(n) = O(n)$$

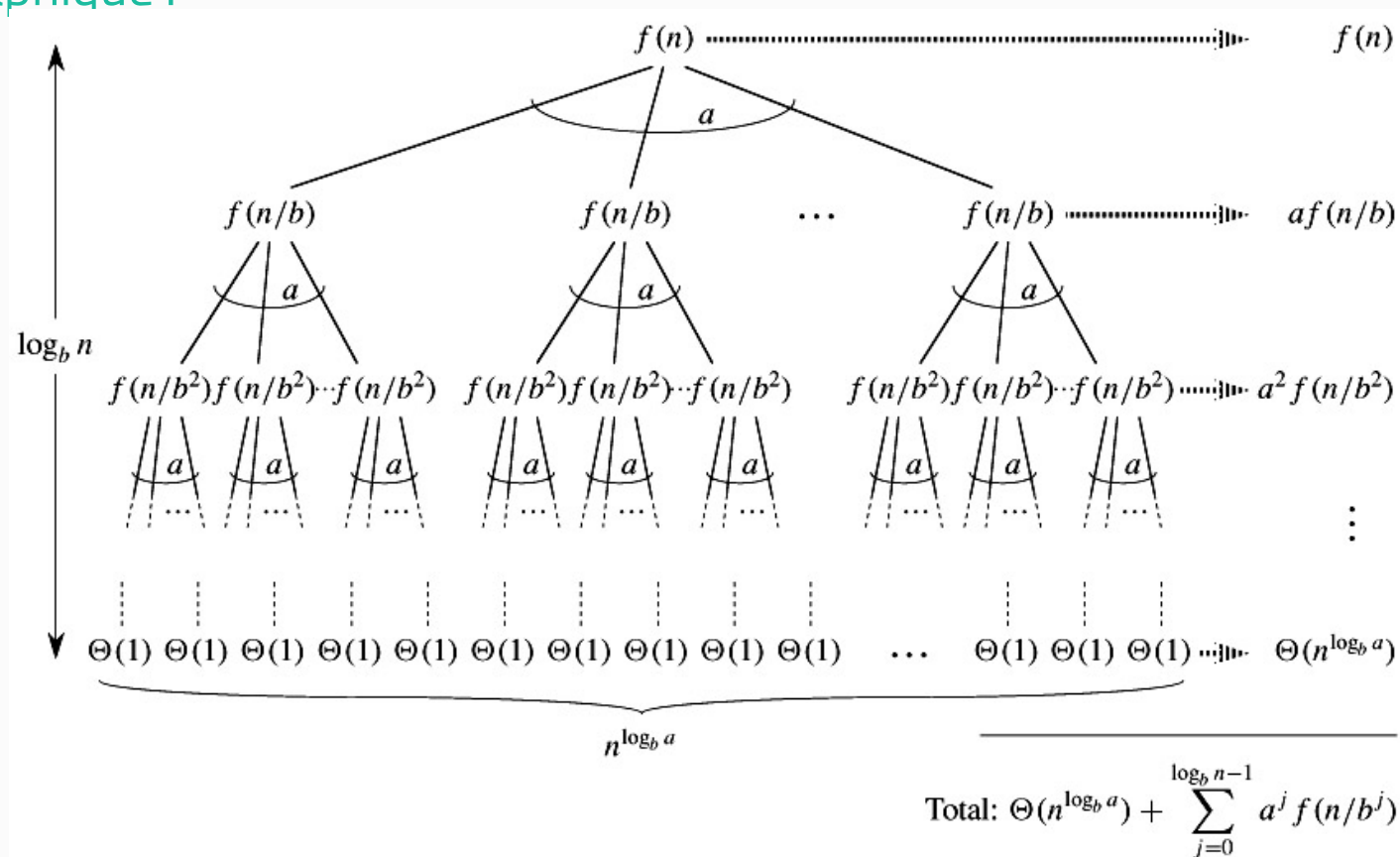
(Et non  $T(n) = \Theta(n)$ , ici !)



# 4<sup>ème</sup> méthode : Master Theorem

Méthode générale pour résoudre directement **des** récurrences de la forme  $T(n) = aT(n/b) + f(n)$   
(on suppose que pour les cas de base,  $T(n) = \Theta(1)$ )

Représentation graphique :



## 4<sup>ème</sup> méthode : Master Theorem

D'où vient le terme  $\Theta(n^{\log_b a})$  ?

💡 A chaque niveau, le nombre de branches est multiplié par  $a$ , et il existe  $\log_b n$  niveaux dans l'arbre  
 $\Rightarrow$  l'arbre possède donc  $a^{\log_b n}$  feuilles (nœuds au dernier niveau de l'arbre). Or

$$a^{\log_b n} = n^{\log_b a} \text{ (exercice)}$$

💡 Les feuilles correspondent aux cas de base de l'algorithme ; or, par hypothèse du Master Theorem,  $T(n) = \Theta(1)$  pour les cas de base. Donc le temps de calcul total au niveau des feuilles est  $\Theta(n^{\log_b a})$ .



## 4<sup>ème</sup> méthode : Master Theorem

💡 Intuitivement, le Master Theorem compare la fonction  $f(n)$  (i.e. le temps passé à subdiviser un problème en sous-problèmes puis fusionner les sous-solutions) et le nombre de feuilles de l'arbre  $n^{\log_b a}$  pour savoir quelle partie requiert le plus de temps d'exécution :

**Master Theorem** : soit une récurrence de la forme ci-dessus. Alors :

1. si  $f(n) = O(n^c)$  avec  $c < \log_b a$ , alors  $T(n) = \Theta(n^{\log_b a})$
2. si  $f(n) = \Theta(n^c)$ , avec  $c = \log_b a$ , alors  $T(n) = \Theta(n^c \log n)$
3. si  $f(n) = \Omega(n^c)$  avec  $c > \log_b a$ , **et si  $af(n/b) \leq kf(n)$  avec  $k < 1$  une constante et  $n$  suffisamment grand** (critère de « régularité »), alors  $T(n) = \Theta(f(n))$

Rem. : on peut raffiner le 2<sup>nd</sup> cas :

2. si  $f(n) = \Theta(n^c \log^k n)$ , avec  $c = \log_b a$  et  $k > -1$ , alors  $T(n) = \Theta(n^c \log^{k+1} n)$



## 4<sup>ème</sup> méthode : Master Theorem

- **Exemple 1** (recherche du maximum dans un tableau non trié) :  $T(n) = 2T(n/2) + 1$

Ici,  $a = 2, b = 2$  d'où  $n^{\log_b a} = n^{\log_2 2} = n$  ; de plus,  $f(n) = 1 = O(n^0)$

⇒ on peut appliquer le cas 1 du MT :  $T(n) = \Theta(n)$

- **Exemple 2** :  $T(n) = T(2n/3) + 1$

Ici,  $a = 1, b = 3/2$  d'où  $n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$  ; de plus,  $f(n) = 1 = \Theta(n^{\log_b a})$

⇒ on peut appliquer le cas 2 du MT :  $T(n) = \Theta(\log n)$

- **Exemple 3** :  $T(n) = 3T(n/4) + n \log n$

Ici,  $a = 3, b = 4$  d'où  $n^{\log_b a} = n^{\log_4 3} = O(n^{0.793})$  ; de plus,  $f(n) = n \log n = \Omega(n^1)$  et  $1 > 0.793$

vérifions la condition de régularité :

pour  $n$  suffisamment grand,  $af(n/b) = 3(n/4) \log(n/4) \leq 3/4 n \log n = cf(n)$  avec  $c = 3/4$

⇒ on peut appliquer le cas 3 du MT :  $T(n) = \Theta(n \log n)$

<https://www.nayuki.io/page/master-theorem-solver-javascript>

