ICS - Algorithmique

Cours 7 : Introduction, correctitude, algorithmes récursifs

12.11.2025

7.1 Introduction

Qu'est-ce qu'un algorithme ?

- Un algorithme est une **procédure** pour résoudre un problème.
  - Il prend en **entrée** (**input**) une **instance** de ce problème
  - et produit la **sortie** (**output**) correspondant à cette instance.

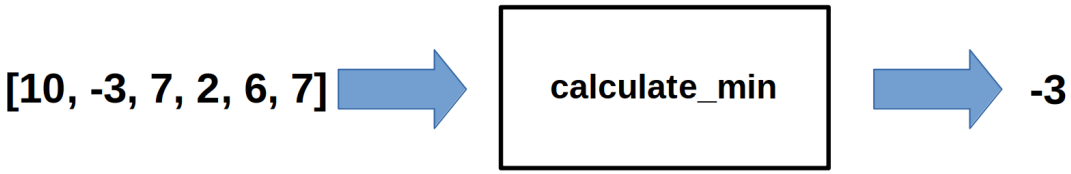
Qu'est-ce qu'un algorithme ?

Par exemple, on peut écrire un algorithme qui calcule le minimum d'une liste de nombres.

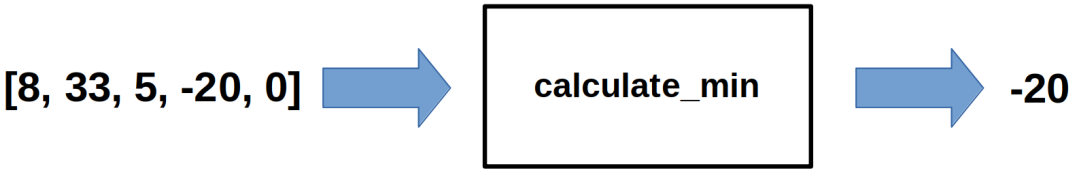
- **Entrée** : une liste de nombre réels
- **Sortie** : le minimum des nombres de la liste
- **Algorithme** (en français) :
  - mettre `le_min` = le premier élément de la liste ;
  - passer sur chaque élément de la liste. S'il est plus petit que `le_min`, mettre `le_min` = cet élément ;
  - retourner `le_min`.

Qu'est-ce qu'un algorithme ?

- Une instance de ce problème : la liste `[10, -3, 7, 2, 6, 7]`



- Une autre instance : la liste `[8, 33, 5, -20, 0]`



- Ne sont pas des instances valides de ce problème :
  - `[[0, 1], [1, 2]]`
  - `['a', 'b', 'c']` (même s'il existe un ordre sur les chaîne de caractères !)

Problème vs. algorithme

- A un problème peuvent correspondre plusieurs algorithmes qui le résolvent
  - Comment caractériser “le meilleur” ?
- Pour certains problèmes on ne sait pas s’il existe un algorithme qui les résoud ; [1] pour certains problèmes on sait qu’il n’en existe aucun. [2]
- Pour certains problèmes, on ne connaît encore aucun algorithme efficace qui les résoud.
  - Que veut dire “efficace” ?

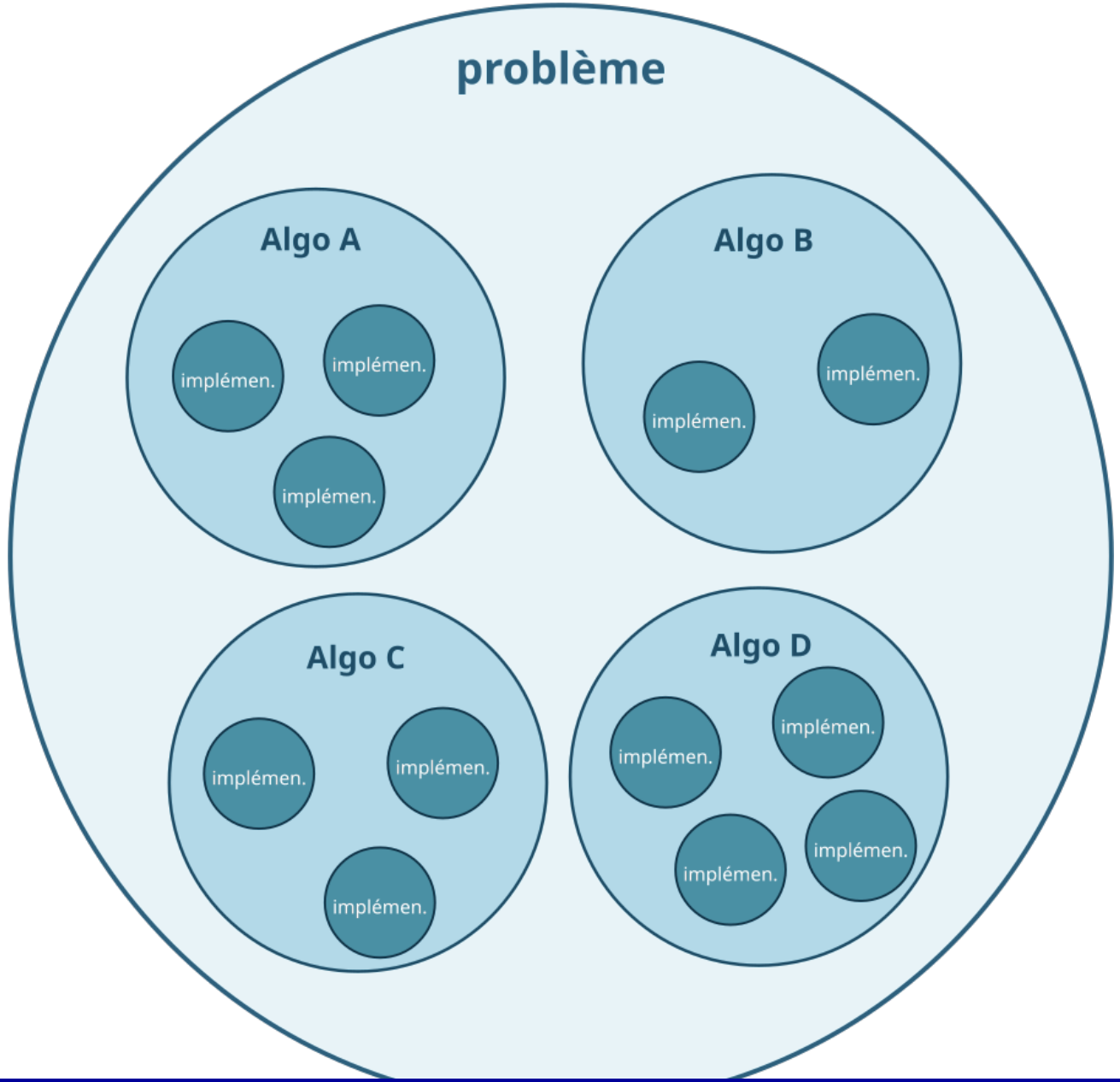
1. https://en.wikipedia.org/wiki/Collatz_conjecture
2. https://en.wikipedia.org/wiki/Halting_problem

Algorithme vs. implémentation

- Un **algorithme** est une procédure générale pour résoudre un problème.
- Un **programme** est une implémentation spécifique d'un algorithme dans un langage de programmation donné, par exemple Python, et sur un système donné.
- A un même algorithme correspondent donc plusieurs implémentations. [3]
- Les algorithmes vus dans ce cours seront donnés sous forme de programmes Python.

3. Voir `http://www.rosettacode.org` pour des implémentations de divers algorithmes en plus de 800 langages de programmation.

Problème, algorithmes, implémentation



Structure de ces 6 cours

- Formalisation de la notion d'algorithme, correctitude d'un algorithme
- Algorithmes récursifs
- Complexité algorithmique (analyse mathématique du temps de parcours des algorithmes)
- Algorithmes de recherche
- Algorithmes de tri
- Algorithmes de graphes

7.2 Correctitude d'algorithmes

Evaluer un algorithme

- Correctitude
- Efficacité

Invariant de boucle

Comment prouver qu'un algorithme calcule bien ce qu'il est censé calculer ?

- Pour un algorithme sans boucles, c'est assez simple.
- Pour un algorithme contenant des boucles `while` ou `for`, c'est plus compliqué !
- On formule et on prouve un **invariant de boucle** : une propriété qui est valide pour toutes les itérations de la boucle et dont la validité à la sortie de la boucle prouve qu'elle calcule bien la bonne valeur.

Invariant de boucle

L'utilisation d'un invariant de boucle pour prouver la correctitude d'un programme se fait en trois étapes :

- **Initialisation** : Prouver que l'invariant de boucle est vrai avant la première itération de la boucle
- **Maintenance** : Prouver que si l'invariant de boucle est vrai pour une itération de la boucle, alors il est vrai pour la prochaine itération de la boucle

  (Avec ces deux étapes, on a prouvé que l'invariant reste vrai à toutes les itérations de la boucle.)
- **Terminaison** : Si l'invariant de boucle est vrai à la sortie de la boucle, on en déduit (directement ou indirectement) la correctitude du programme.

Une boucle pour calculer une série géométrique

Problème : pour un réel a et un entier $n \geq 0$ en entrée, écrire un algorithme (sous forme de fonction Python) qui retourne

$$\sum_{k=0}^{n-1} a^k = 1 + a + a^2 + \cdots + a^{n-1}.$$

Voici une solution :

```
def serie_geom(a, n):

    s = 0
    for i in range(n):
        s = s * a + 1

    return s
```

Prouvons que cet algorithme calcule bien la somme demandée !

Série géométrique : invariant de boucle

```python
def serie_geom(a, n):

    s = 0
    for i in range(n):
        s = s * a + 1

    return s
```

Invariant de boucle :
Au début de la $i$ème itération de la boucle `for`, `s` contient la valeur

$$\sum_{k=0}^{i-1} a^k = 1 + a + \cdots + a^{i-1}.$$

Série géométrique : invariant de boucle - preuve (initialisation)

```python
def serie_geom(a, n):

    s = 0
    for i in range(n):
        s = s * a + 1

    return s
```

Invariant de boucle :
Au début de la $i$ème itération de la boucle `for`, s contient la valeur

$$\sum_{k=0}^{i-1} a^k = 1 + a + \cdots + a^{i-1}.$$

- **Initialisation** : Avant le début de la boucle `for`, $s = 0$, ce qui correspond bien à la somme vide. [4]

4. Une somme de la forme $\sum_i^j k$ avec $j < i$ est dite vide et vaut 0.

Série géométrique : invariant de boucle - preuve : maintenance

```python
def serie_geom(a, n):

    s = 0
    for i in range(n):
        s = s * a + 1

    return s
```

Invariant de boucle :
Au début de la $i$ème itération de la boucle `for`, s contient la valeur

$$\sum_{k=0}^{i-1} a^k = 1 + a + \cdots + a^{i-1}.$$

- **Maintenance** : On suppose qu'au début de la $i$ème itération, on a $s = \sum_{k=0}^{i-1} a^k$. Or la $i$ème itération consiste à exécuter l'instruction `s = s * a + 1`. La valeur de s devient donc

$$(1 + a + \cdots + a^{i-1})a + 1 = (a + a^2 + \cdots + a^i) + 1 = \sum_{k=0}^{i} a^k$$

à la fin de la $i$ème itération (et au début de la $i+1$ème).

Série géométrique : invariant de boucle - terminaison

```python
def serie_geom(a, n):

    s = 0
    for i in range(n):
        s = s * a + 1

    return s
```

Invariant de boucle :
Au début de la $i$ème itération de la boucle `for`, s contient la valeur

$$\sum_{k=0}^{i-1} a^k = 1 + a + \cdots + a^{i-1}.$$

- **Terminaison** : A la sortie de la boucle (càd au début de la "n"ème itération, qui n'aura pas lieu), on a $s = \sum_0^{n-1} a^k$, ce qui est exactement ce que la boucle prétend calculer.

Une boucle pour calculer un maximum

```python
def max_liste(L):
    '''
    Entree: liste L de nombres de taille >= 1
    Sortie: maximum de L
    '''
    max_L = L[0]

    for i in range(1, len(L)):
        if L[i] > max_L:
            max_L = L[i]

    return max_L
```

Cet algorithme retourne le maximum de la liste L.

Maximum : invariant de boucle

```python
def max_liste(L):
    '''
    Entree: liste L de nombres de taille >= 1
    Sortie: maximum de L
    '''
    max_L = L[0]

    for i in range(1, len(L)):
        if L[i] > max_L:
            max_L = L[i]

    return max_L
```

Invariant de boucle : Au début de la *i*ème itération de la boucle `for`, `max_L` a la valeur du maximum de `L[0:i]`.

Maximum : invariant de boucle - preuve : initialisation

```python
def max_liste(L):

    max_L = L[0]

    for i in range(1, len(L)):
        if L[i] > max_L:
            max_L = L[i]

    return max_L
```

Invariant de boucle :
Au début de la *i*ème itération de la boucle `for`, `max_L` a la valeur du maximum de `L[0:i]`.

- **Initialisation** : Avant le début de la boucle `for`, donc au début de l'itération `i = 1`, `max_L` est égal à `L[0]` qui est bien le maximum (et l'unique élément) de `L[0:1]`.

Maximum : invariant de boucle - preuve : maintenance

```python
def max_liste(L):

    max_L = L[0]

    for i in range(1, len(L)):
        if L[i] > max_L:
            max_L = L[i]

    return max_L
```

Invariant de boucle :
Au début de la *i*ème itération de la boucle, `max_L` a la valeur du maximum de `L[0:i]`.

- **Maintenance** : On suppose qu'au début de la *i*ème itération, `max_L` est égal au maximum de `L[0:i]`. Durant la *i*ème itération, on affecte à `max_L` la valeur

  $$\texttt{max(max\_L, L[i])} = \texttt{max(L[0:i+1])}$$

  et donc au début de la prochaine itération, `max_L` est bien égal au maximum de `L[0:i+1]`.

Maximum : invariant de boucle - terminaison

```python
def max_liste(L):

    max_L = L[0]

    for i in range(1, len(L)):
        if L[i] > max_L:
            max_L = L[i]

    return max_L
```

Invariant de boucle :
Au début de la *i*ème itération de la boucle, `max_L` a la valeur du maximum de `L[0:i]`.

- **Terminaison** : A la sortie de la boucle (c'est-à-dire au début de la `len(L)`-ième itération, qui n'aura pas lieu), `max_L` est égal au maximum de `L[0:len(L)]`, donc au maximum de la liste entière.

Condition de terminaison

- Si l'algorithme contient une boucle `while`, il faut prouver que l'algorithme termine.
- Pour cela, on définit en général une expression (un **variant de boucle**)
  - qui croît ou décroît strictement en fonction du nombre d'itérations
  - telle que la boucle termine lorsque l'expression atteint une certaine valeur.

Algorithme d'Euclide

- Etant donné deux entiers a et b strictement positifs, on veut calculer leur plus grand commun diviseur (**pgcd**).
  - $\text{pgcd}(3060, 924) = \text{pgcd}(2^2 \cdot 3^2 \cdot 5 \cdot 17\,,\, 2^2 \cdot 3 \cdot 7 \cdot 11) = 2^2 \cdot 3 = 12.$
- Trouver la factorisation en nombres premiers d'un très grand nombre est un problème qu'on conjecture (et qu'on espère) **très difficile**.
- **L'algorithme d'Euclide**, beaucoup plus simple, se base sur une propriété importante :

$$\text{pgcd}(a, b) = \text{pgcd}(b, a \bmod b). \qquad (*)$$

$\text{pgcd}(a, b) = \text{pgcd}(b, a \bmod b)$

Preuve :

- On écrit la division entière de a par b comme

$$a = bq + r, \qquad 0 \leq r < b,$$
$$a = bq + (a \bmod b).$$

- Cette équation montre immédiatement qu'un diviseur de b et r divise aussi a (et b).
- En réorganisant cette équation en

$$r = a - bq$$

on voit aussi qu'un diviseur de a et b divise également r (et b).

- Par conséquent, a et b ont exactement les mêmes diviseurs communs que b et r. Par conséquent, leur pgcd est bien sûr le même. ■

Algorithme d'Euclide - terminaison

On peut utiliser ce fait pour créer un algorithme.

```
def pgcd(a, b):
    '''
    Entree: a, b entiers strictement positifs
    Sortie: plus grand diviseur commun de a et b
    '''
    i, k = a, b

    while k > 0:
        i, k = k, i % k

    return i
```

- Pourquoi cet algorithme termine-t-il ?
  - k décroît d'au moins 1 à chaque itération de la boucle `while` ($i \bmod k < k$ et ce sont des entiers).
  - La boucle se termine lorsque k (initialement > 0) atteint une valeur ≤ 0.

Algorithme d'Euclide - correctitude

```
def pgcd(a, b):
    i, k = a, b
    while k > 0:
        i, k = k, i % k
    return i
```

Pourquoi cet algorithme rend-il une valeur correcte ?

- Au moment où l'algorithme retourne la valeur i,
  - k vaut 0 (k étant le reste d'une division entière, il n'est jamais négatif)
  - La valeur rendue $i = \text{pgcd}(i, 0) = \text{pgcd}(i, k)$.
- Or l'égalité $\text{pgcd}(a, b) = \text{pgcd}(b, a \bmod b)$ (avec la terminaison de la boucle `while`) implique l'invariant de boucle :

  A chaque itération de la boucle, $\text{pgcd}(i, k) = \text{pgcd}(a, b)$.
- Pour a, b entiers strictement positifs en entrée, l'algorithme retourne donc bien $\text{pgcd}(a, b)$.

Algorithme d'Euclide - Exemples

$$\begin{aligned}
\text{pgcd}(3060, 924) &= \text{pgcd}(924, 288) \\
&= \text{pgcd}(288, 60) \\
&= \text{pgcd}(60, 48) \\
&= \text{pgcd}(48, 12) \\
&= \text{pgcd}(12, 0) = 12.
\end{aligned}$$

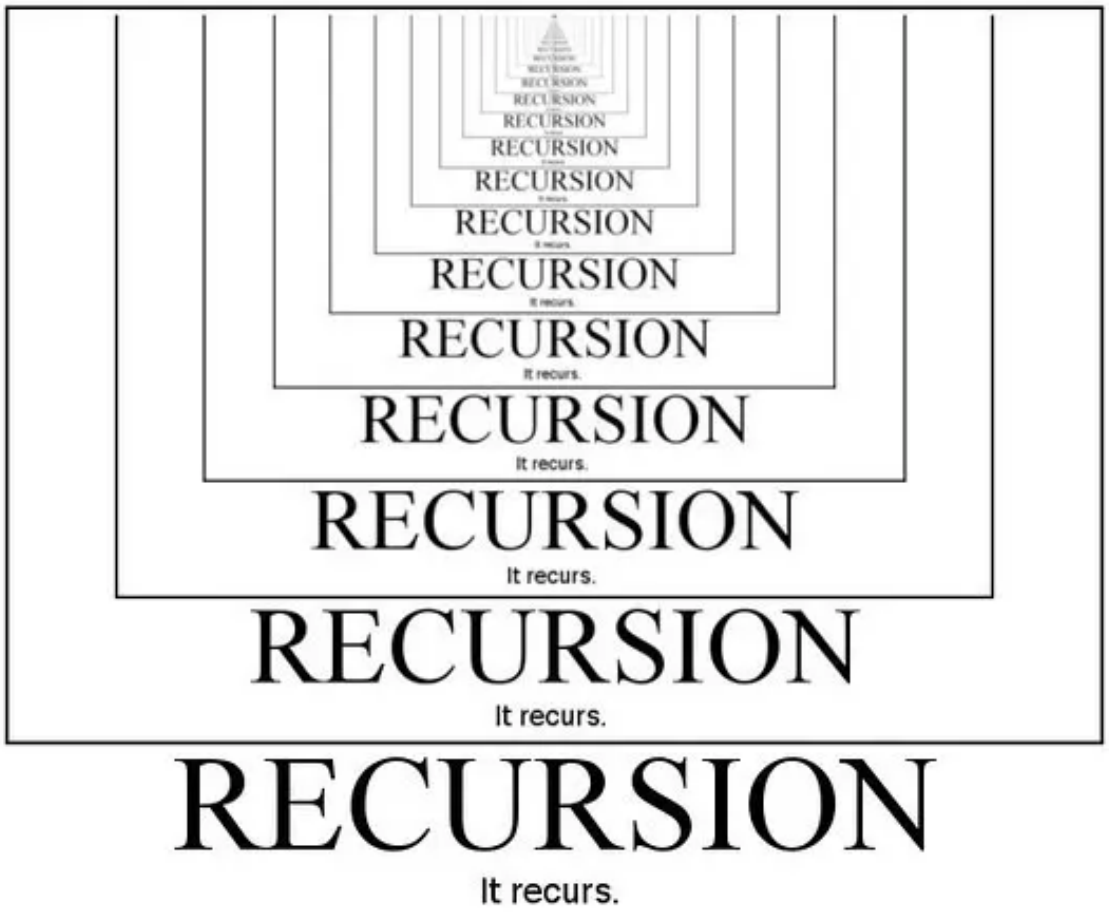
$$\begin{aligned}
\text{pgcd}(2437, 181) &= \text{pgcd}(181, 84) \\
&= \text{pgcd}(84, 13) \\
&= \text{pgcd}(13, 6) \\
&= \text{pgcd}(6, 1) \\
&= \text{pgcd}(1, 0) = 1.
\end{aligned}$$

7.3 Algorithmes récursifs

Algorithmes récursifs

- Un algorithme récursif résout une instance d'un problème en résolvant une ou plusieurs instances *du même problème* de taille plus petite et combinant les solutions obtenues pour obtenir la solution à l'instance initiale du problème.
- On parle du paradigme "**diviser-pour-régner**".
- La correctitude de l'algorithme pour une instance d'une certaine taille découlera en général de la correctitude pour les instances plus petites.
- Pour implémenter des algorithmes récursifs, on écrit des fonctions qui s'appellent elles-mêmes. 😱

Algorithmes récursifs



Factorielle

- Pour $n \in \mathbb{N}$, la factorielle de n est définie comme

$$n! = n \cdot (n-1) \cdots 2 \cdot 1 = \begin{cases} n \cdot (n-1)! & , n \geq 1 \\ 1 & , n = 0 \end{cases}$$

- On peut calculer la valeur de $n!$ à partir de celle de $(n-1)!$, c'est-à-dire ramener la résolution du problème pour une instance de taille n à la résolution du problème pour une instance de taille $n-1$.

Factorielle

Cela suggère l'algorithme suivant :

```
def fact(n):
    '''
    Entree: entier naturel n
    Sortie: n!
    '''
    if n == 0:
        return 1
    f = fact(n-1)
    return n*f
```

cas de base

appel récursif

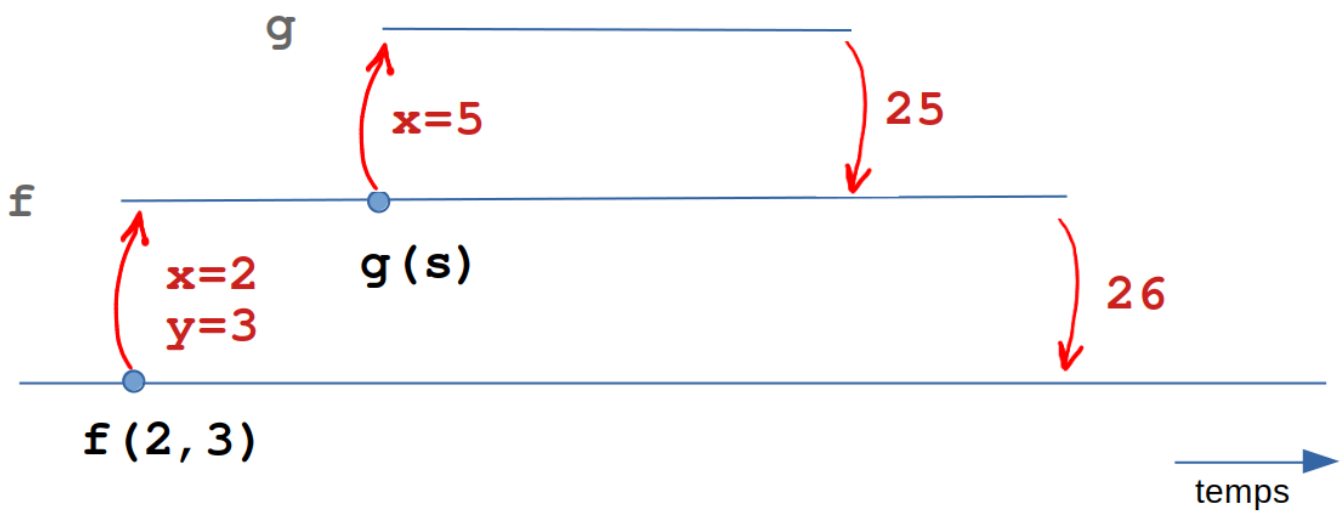
- Pour le **cas de base**, la valeur de la fonction est définie directement, sans appels récursifs.
  - On peut définir un ou plusieurs cas de base.
- On peut effectuer un ou plusieurs appels récursifs.
- `fact` calcule la factorielle de n avec n appels récursifs.

Appels de fonctions - visualisation

```
def g(x):
    return x**2

def f(x,y):
    s = x+y
    return g(s)+1

x = f(2,3)
```



Factorielle - appels récursifs

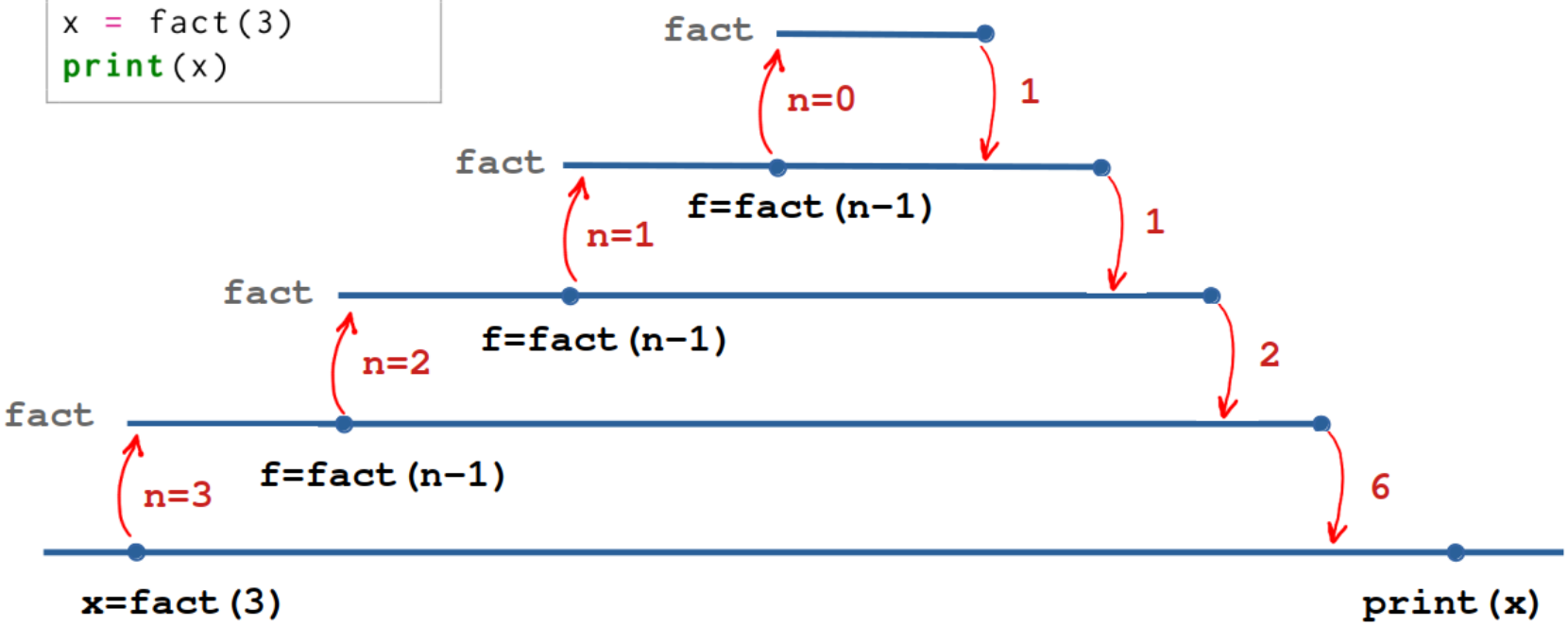
Vous pouvez suivre les appels récursifs de la fonction factorielle :

- sur Python Tutor
- sur Recursion Visualizer

Factorielle - appels récursifs

```
def fact(n):
    if n == 0:
        return 1
    f = fact(n-1)
    return n*f

x = fact(3)
print(x)
```



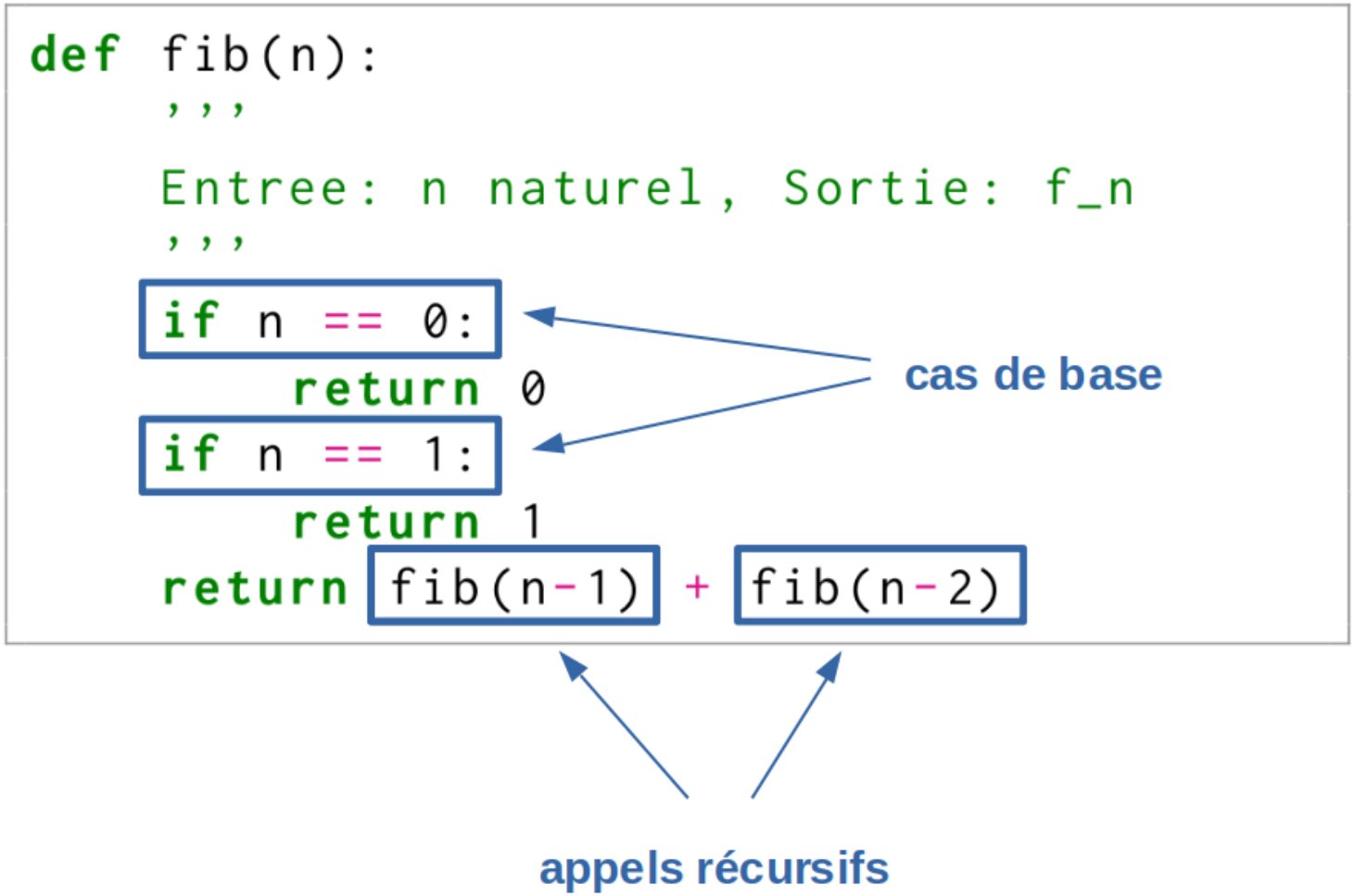
Suite de Fibonacci

- Pour $n \in \mathbb{N}$, le $n$ème nombre de Fibonacci est défini comme

$$f_n = \begin{cases} f_{n-1} + f_{n-2} & , n \geq 2 \\ 1 & , n = 1 \\ 0 & , n = 0 \end{cases}$$

- On peut ramener la résolution du problème pour une instance de valeur n à la résolution du problème pour deux instances de valeur $n - 1$ et $n - 2$.

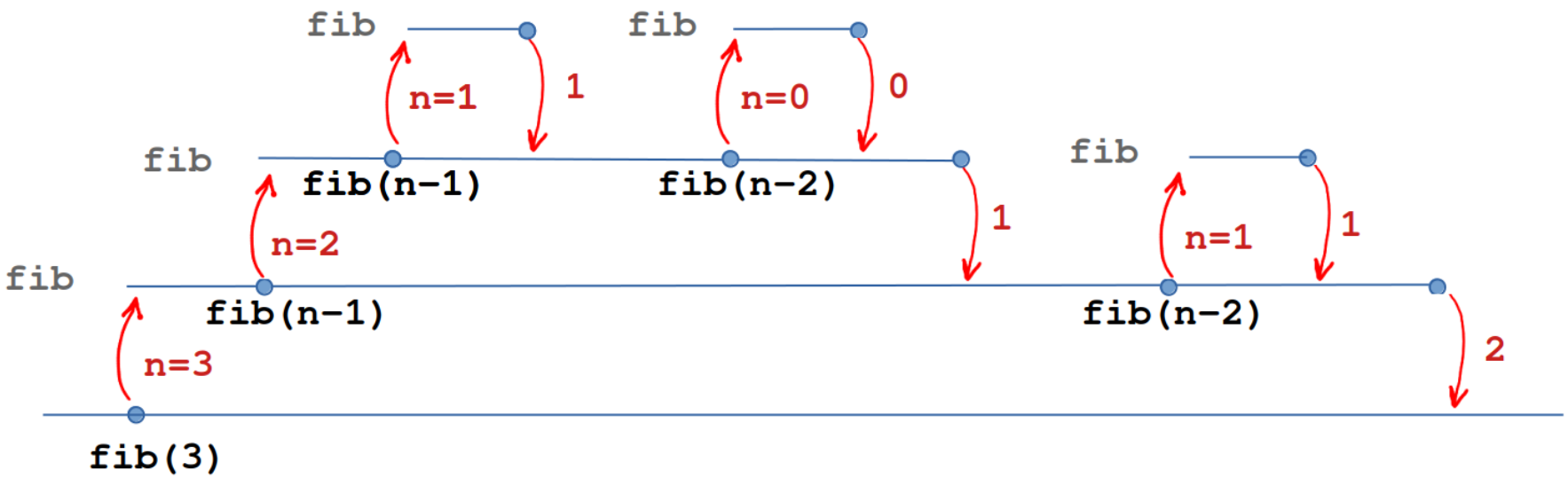
Suite de Fibonacci



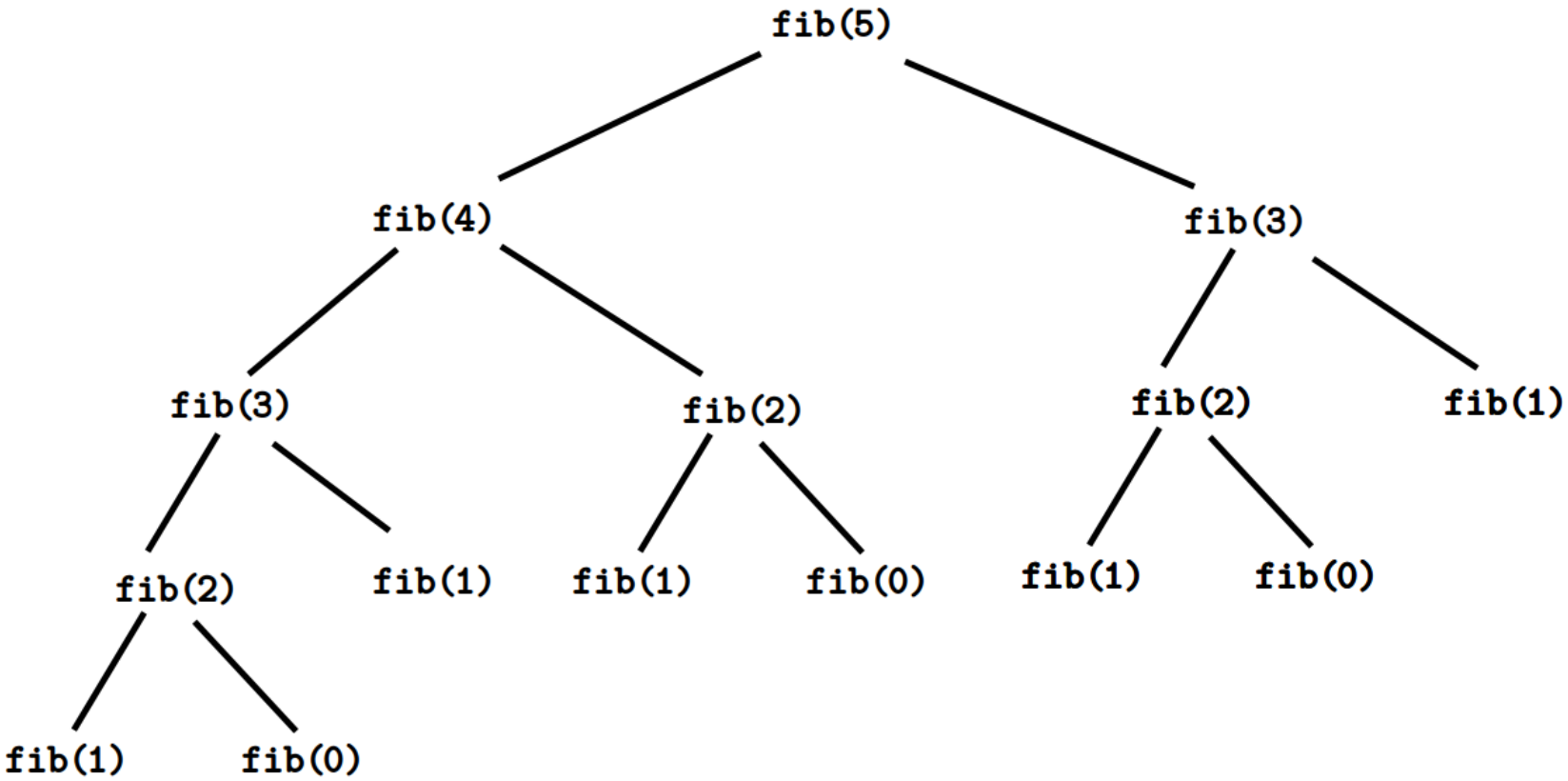
Fibonacci - appels récursifs

```
def fib(n):
    if n == 0:
        return 0
    if n == 1:
        return 1
    return fib(n-1) + fib(n-2)

print(fib(3))
```



Fibonacci - appels récursifs



On peut prouver que pour l'entrée n le nombre d'appels récursifs de `fib` et son temps de parcours sont **exponentiels** en n.

Somme récursive

On a déjà vu un algorithme **itératif** (contenant une boucle) qui prend en entrée un entier n strictement positif et calcule la somme des nombres de 1 à n.

```
def somme(n):
    s = 0
    for i in range(1, n+1):
        s += i
    return s
```

Peut-on écrire un algorithme **récursif** (sans boucle) `somme_rec` qui calcule la même somme ?

Somme récursive

- Pour n entier strictement positif,

$$\sum_{1}^{n} i = \begin{cases} 1 & , n = 1 \\ \sum_{1}^{n-1} i + n & , n > 1 \end{cases}$$

- `somme_rec(n)` doit donc retourner 1 si $n = 1$ (cas de base), et `somme_rec(n-1) + n` sinon (appel récursif).

Somme récursive

```python
def somme_rec(n):
    '''
    Entree: n entier strictement positif
    Sortie: somme des nombres entiers de 1 a n inclus
    '''
    if n == 1:
        return 1
    return somme_rec(n-1) + n
```

Algorithmes récursifs vs algorithmes itératifs

- On peut toujours remplacer un algorithme récursif par un algorithme itératif qui fait le même travail. Alors pourquoi écrire des algorithmes récursifs ?
- Certains problèmes ont naturellement une structure récursive (ex. calcul de la factorielle) et il est donc plus facile d'y réfléchir récursivement.
- Mais les algorithmes récursifs sont souvent moins efficaces...