M.C. Escher, *Drawing Hands*

Récursivité

1 Définition et premier exemple

Définition. Une *fonction récursive* est une fonction qui s'appelle elle-même.

Exemple. Calculons $n!$ de deux manières :

Version itérative à partir de la définition

$$n! = \prod_{k=1}^{n} k$$

```
def fact_ite(n: int) -> int:
    """Calcul itératif de n!
    avec n >= 0."""
    p = 1
    for k in range(1, n+1):
        p = p * k
    return p
```

Version récursive à partir de la propriété

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

```
def fact_rec(n: int) -> int:
    """Calcul récursif de n!
    avec n >= 0."""
    if n == 0:
        return 1
    else:
        return n * fact_rec(n-1)
```

Par exemple, l'appel `fact_rec(4)` donne la séquence d'opérations suivante :

appel de `fact_rec(4)`
appel de `fact_rec(3)`
appel de `fact_rec(2)`
appel de `fact_rec(1)`
renvoi de 1 (fin de `fact_rec(1)`)
calcul de 2*1 et renvoi de 2 (fin de `fact_rec(2)`)
calcul de 3*2 et renvoi de 6 (fin de `fact_rec(3)`)
calcul de 4*6 et renvoi de 24 (fin de `fact_rec(4)`)

Pour le visualiser, sur Python Tutor [1], écrire la fonction `fact_rec` et exécuter l'appel `fact_rec(4)`.

2 Un exemple visuel pour comprendre

Rappel : la commande `'bla' * 3` donne la chaîne de caractères qui est la concaténation de trois exemplaires de `'bla'`, *i.e.* `'blablabla'`. Cette notation fonctionne d'ailleurs également avec une liste : `[0]*5` donne `[0, 0, 0, 0, 0]`.

Pour comprendre ce qui se passe lors de l'appel d'une fonction récursive, comparons l'exécution de ces deux fonctions récursives.

1. `http://www.pythontutor.com/visualize.html#mode=edit`

```python
def affiche1(n: int) -> None:
    if n > 0:
        print('o' * n)
        affiche1(n-1)
```

```
>>> affiche1(5)
ooooo
oooo
ooo
oo
o
```

```python
def affiche2(n: int) -> None:
    if n > 0:
        affiche2(n-1)
        print('o' * n)
```

```
>>> affiche2(5)
o
oo
ooo
oooo
ooooo
```

3 Lien avec les piles

L'exécution se fait avec une pile. En effet, à chaque appel récursif qui n'est pas un cas d'arrêt (*i.e.* le cas $n = 0$ pour `fact_rec`), les valeurs des variables locales (n dans le calcul `n * fact_rec(n-1)`) doivent être mémorisées. Pour cela, on empile les appels et ces valeurs intermédiaires jusqu'à atteindre un cas d'arrêt. Une fois celui-ci atteint, on dépile successivement et on calcule les retours successifs jusqu'à retrouver une pile vide.

Remarque. Par défaut, Python limite la taille de la pile d'exécution à environ 1 000. Si on dépasse cette limite, on obtiendra une erreur du type `RecursionError: maximum recursion depth exceeded in comparison`.

4 Terminaison et correction

4.1 Terminaison

Pour démontrer la terminaison d'un algorithme récursif, il suffit de trouver une quantité à valeurs dans $\mathbb{N}$ qui décroît strictement à chaque appel récursif et qui prend donc nécessairement une valeur définissant un cas d'arrêt au bout d'un nombre fini d'étapes. Pour que l'algorithme termine, il est donc indispensable de penser au cas d'arrêt (éventuellement plusieurs).

Exemple. Démontrons la terminaison de la fonction `fact_rec`.
On remarque qu'à chaque appel de `fact_rec`, la valeur de n décroît strictement (de 1) donc on va nécessairement aboutir au cas $n = 0$ au bout d'un nombre fini d'étapes, la fonction termine donc (à condition que l'entrée soit un entier naturel).

4.2 Correction

Pour montrer qu'une fonction récursive renvoie le résultat souhaité, on procède par récurrence. L'initialisation correspond au cas d'arrêt et l'hérédité à la conservation de la propriété souhaitée lors de l'appel récursif.

Exemple. Démontrons que la fonction `fact_rec` renvoie la valeur de $n!$ lorsqu'on lui donne l'entier naturel n comme argument.
Pour $n \in \mathbb{N}$, posons $\mathcal{P}(n)$: « `fact_rec(n)` renvoie la valeur de $n!$ ».
Initialisation : D'une part, lors de l'appel `fact_rec(0)`, le test en ligne 4 est vrai donc la valeur renvoyée est 1. D'autre part, on a par convention $0! = 1$. Ainsi $\mathcal{P}(0)$ est vraie.
Hérédité : Supposons $\mathcal{P}(n)$ vraie pour un certain entier naturel n, *i.e.* `fact_rec(n)` renvoie la valeur de $n!$. Lors de l'appel `fact_rec(n+1)`, on a `n+1 > 0` donc c'est la ligne 7 qui s'exécute, la valeur renvoyée est donc `(n+1) * fact_rec(n)`. Or, d'après $\mathcal{P}(n)$, `fact_rec(n)` renvoie la valeur de $n!$ donc `(n+1) * fact_rec(n)` renvoie $(n+1) \times n! = (n+1)!$, *i.e.* $\mathcal{P}(n+1)$ est vraie.
Conclusion : On a démontré la correction de la fonction `fact_rec`, *i.e.* pour tout $n \in \mathbb{N}$, `fact_rec(n)` renvoie la valeur de $n!$.

5 Avantages

Le principal intérêt de la récursivité est qu'elle permet souvent d'obtenir un code clair et concis (si le problème s'y prête bien sûr).

5.1 Suite de Fibonacci

Considérons la suite de Fibonacci, c'est-à-dire la suite (u_n) définie par $u_0 = u_1 = 1$ et, pour tout $n \in \mathbb{N}$, $u_{n+2} = u_{n+1} + u_n$.
Écrivons une fonction donnant la valeur du n-ème terme de cette suite, d'abord avec deux versions itératives.

```python
def fibo_ite1(n: int) -> int:
    if n == 0 or n == 1:
        return 1
    else:
        u = 1
        v = 1
        for i in range(n-1):
            u, v = v, u+v
        return v
```

```python
def fibo_ite2(n: int) -> int:
    L = [1, 1]
    for i in range(n-1):
        L.append(L[-1] + L[-2])
    return L[-1]
```

Cette seconde version est plus simple à écrire mais nécessite beaucoup d'espace mémoire car on stocke dans une liste toutes les valeurs de u_0 à u_n alors que seule la dernière nous intéresse.

Écrivons maintenant une fonction récursive répondant au même problème.

```python
def fibo(n: int):
    if n == 0 or n == 1:
        return 1
    else:
        return fibo(n-1) + fibo(n-2)
```

La simplicité de cette version récursive saute aux yeux, à la fois par la clarté de la fonction et la proximité avec le mode de définition de la suite (u_n).

5.2 Tours de Hanoï

Un exemple classique d'utilisation de la récursivité est celui de la résolution du problème des tours de Hanoï. Pour les intéressés :

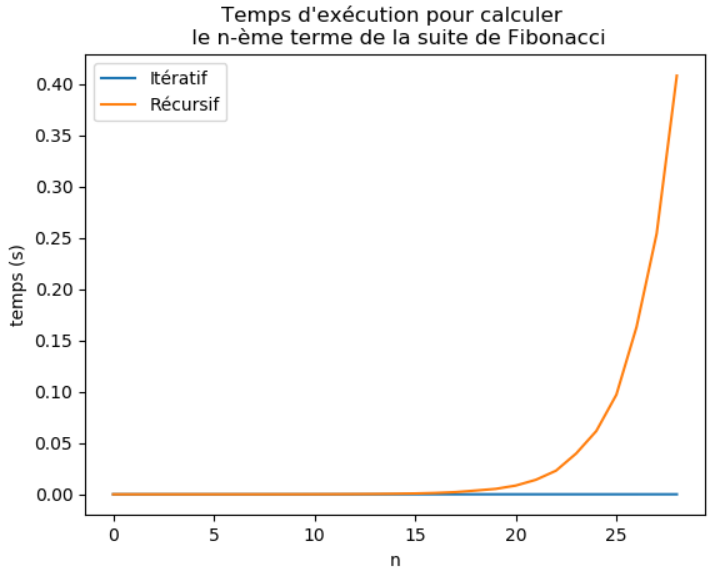
- explication du jeu sur `https://fr.wikipedia.org/wiki/Tours_de_Hano%C3%AF` avec des animations pour bien comprendre ;
- possibilité d'y jouer sur `http://championmath.free.fr/tourhanoi.htm`.

6 Inconvénients

Cependant, outre le problème de la taille limitée de la pile d'exécution, on peut se rendre compte aussi de la différence de rapidité d'exécution entre des versions récursives et itérative.

6.1 Un premier problème évitable

Comparons les temps d'exécution pour les premières valeurs de n des deux fonctions permettant de calculer les termes de la suite de Fibonacci :



Pour comprendre cette différence, déterminons la complexité temporelle de ces deux fonctions en fonction de la valeur de l'entier n donné en argument.

- Pour `fibo_ite`, la boucle est exécutée $n-1$ fois et chaque « tour de boucle » est constitué de deux affectations donc il s'agit d'une complexité linéaire, *i.e.* $O(n)$.
- Intéressons-nous maintenant à `fibo_rec` et notons $C(n)$ la complexité de `fibo_rec(n)`. D'après la définition de la fonction, on a

$$C(0) = C(1) = 1 \text{ et pour } n \geqslant 2,\ C(n) = C(n-1) + C(n-2).$$

Autrement dit, $C(n)$ est égal au n-ème terme F_n de la suite de Fibonacci. Or, on peut montrer [2] que

$$\forall n \in \mathbb{N},\ F_n = \frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right] \underset{+\infty}{\sim} \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n.$$

Ainsi, $C(n) = O\left(\left(\frac{1+\sqrt{5}}{2}\right)^n\right)$ et en particulier la complexité est exponentielle.

Remarque. Cette complexité exponentielle est due au fait qu'on recalcule de nombreuses fois les mêmes termes. Par exemple l'appel `fibo_rec(4)` nécessite les valeurs de `fibo_rec(2)` et `fibo_rec(3)`, celle-ci nécessitant elle-même la valeur de `fibo_rec(1)` et `fibo_rec(2)`, cette dernière est donc calculée deux fois. On pourrait assez facilement pallier ce problème en conservant en mémoire, processus appelé *mémoïsation*, les valeurs obtenues au fur et à mesure et ne calculer que celles qui ne l'ont pas déjà été.

.....

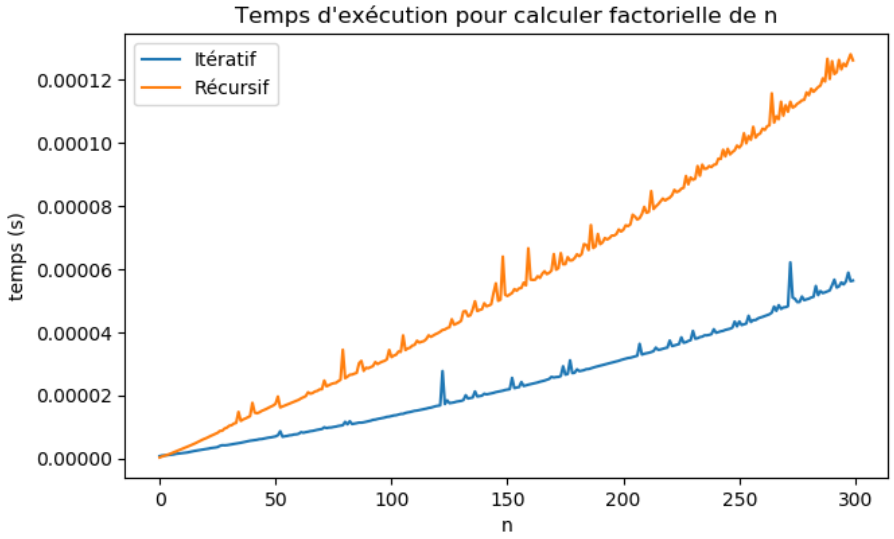
Hors programme : Pour éviter ce problème, on peut écrire d'une autre façon, appelée récursive terminale [3], la fonction qui calcule récursivement le n-ème terme de la suite de Fibonacci. Le prix à payer est une perte de simplicité dans l'écriture du code :

```
def fibobis(n: int, a: int, b: int):
    """ Calcul du n-ème terme de la suite
    de Fibonacci où u0 = a et u1 = b. """
    if n == 0:
        return a
    elif n == 1:
        return b
    else:
        return fibobis(n-1, b, a+b)
```

2. Voir cours de maths sur les suites récurrentes linéaires d'ordre 2.
3. `https://fr.wikipedia.org/wiki/R%C3%A9cursion_terminale`

6.2 Une lenteur en python

De façon plus générale, même en cas de complexité temporelle identique, un algorithme récursif en python est souvent plus lent que son analogue itératif.
Par exemple, les deux versions `fact_ite` et `fact_rec` du calcul de la factorielle sont clairement de complexité linéaire mais en pratique, il y a une différence notable de temps d'exécution :



6.3 Complexité spatiale

Enfin, la complexité spatiale d'un algorithme récursif est souvent importante à cause de la mémorisation de valeurs dans la pile.
Par exemple, `fact_ite(n)` est de complexité spatiale $O(1)$ car on stocke juste la valeur de p à chaque étape alors que celle de `fact_rec(n)` est $O(n)$ car on stocke les n appels successifs nécessaires au calcul de $n!$.