Chapitre 1.7 : Récursivité

I Fonctions récursives

Une fonction **récursive** est une fonction qui s'appelle elle-même. Un des exemples les plus simple est le calcul de la factorielle d'un entier positif :

```
def fact(n:int) -> int:
    '''Renvoie n!
    '''
    if n == 0:
        return 1
    else:
        return n*fact(n-1)
```

Si on croit la spécification de cette fonction, `fact(n-1)` renvoie $(n-1)!$ ce qui donne bien $n \times (n-1)! = n!$. C'est en quelque sorte une fonction définie par récurrence, et qui se prête bien à la résolution de problèmes définis par récurrence (notamment le calcul des termes d'une suite).

Une fonction récursive se présente généralement sous la forme :

```
def f(...):
    if ...: #Condition d'arrêt, cas de base
        #Pas d'appel récursif
    else:
        #Un ou plusieurs appels récursifs
```

Le cas de base est très important et doit être cohérent avec les appels récursifs. Par exemple, l'appel de la fonction

```
def fact(n):
    return n*fact(n-1)
```

avec n'importe quel `n` donnera une erreur de dépassement de pile (voir plus loin).

De même, si on essaye de calculer $n!$ avec $n! = \dfrac{(n+1)!}{(n+1)}$:

```
def fact_inv(n):
    if n == 0:
        return 1
    else:
        return fact_inv(n+1)/(n+1)
```

II Principe de fonctionnement : les piles

Une **pile** (ou **stack**) est une structure de donnée **linéaire** (éléments rangés sur une ligne) qui permet de stocker des éléments en utilisant le principe **LIFO** : Last In First Out. On ne peut accéder qu'au dernier élément ajouté, appelé **sommet**.

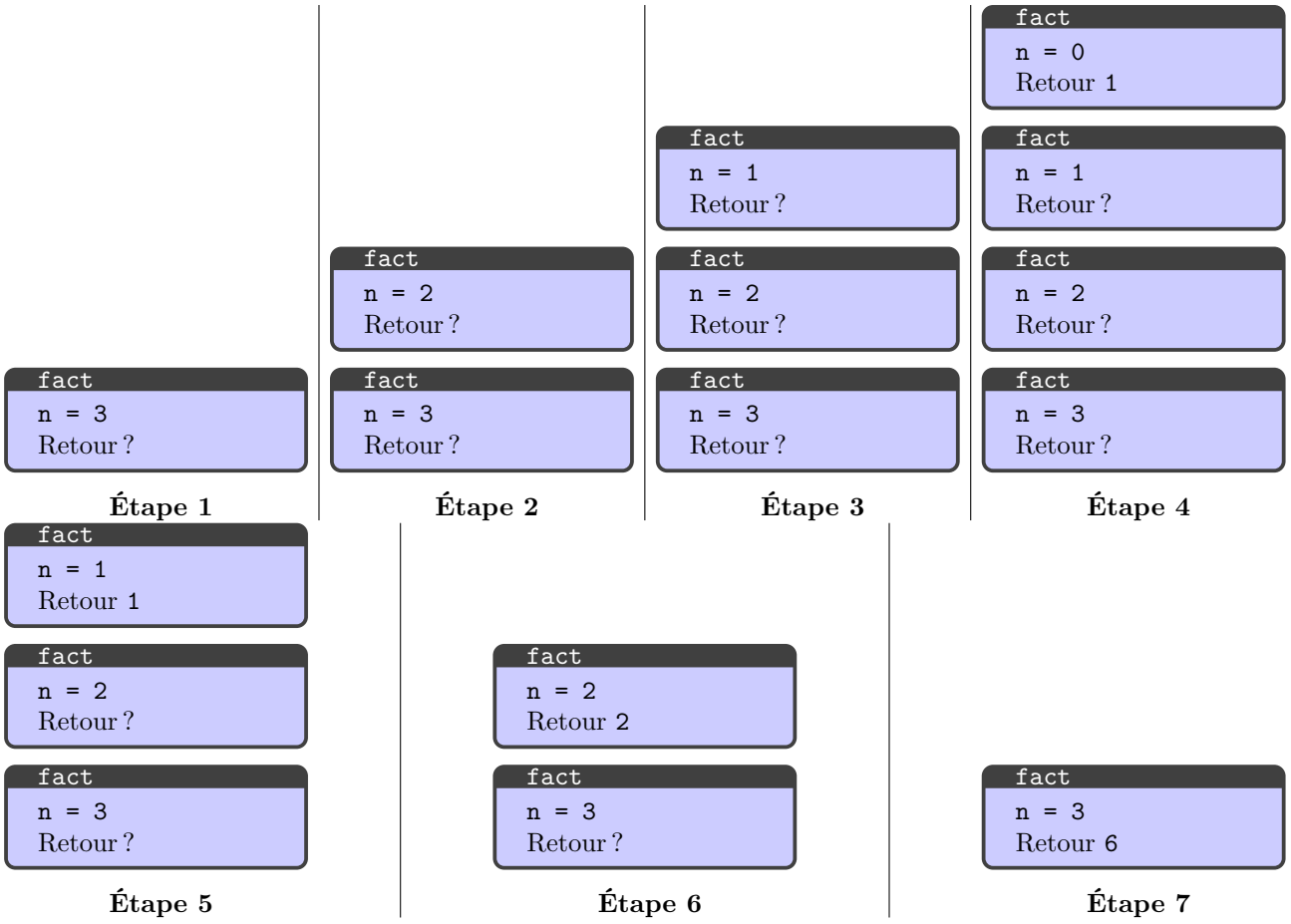
On dispose de trois opérations élémentaires sur les piles :

- **créer** une pile ;
- **empiler** (push) un élément sur une pile ;
- **dépiler** (pop) un élément d'une pile.

L'interpréteur Python utilise une **pile d'appel** pour stocker les appels de fonctions lors de l'exécution d'un script :

- l'interpréteur exécute la fonction en haut de la pile ;
- à chaque fois que le script appelle une fonction, l'interpréteur créé un bloc fonction (qui contient des informations sur la fonction : les arguments, de la place pour les variables locales, de la place pour la valeur de retour, ...) et le place sur la pile et commence à l'exécuter.

Par exemple, lors de l'exécution de `fact(3)` :



Sous Python, la pile d'appel est limitée à 1000 appels. Autrement, on obtient :

```
RuntimeError : maximum recursion depth exceeded
```

III Terminaison et complexité

Pour justifier la terminaison d'une fonction récursive, on n'a pas de variant, mais on procède en suivant les mêmes idées que pour les boucles.

■ **Exemple 1** La fonction récursive :

```
def fact(n : int) -> int:
    ''' Fonction récursive qui renvoie
    '''
    assert
    if n == 0:
        return 1
    else:
        return n*fact(n-1)
```

termine car ■

Pour déterminer la complexité temporelle d'une fonction récursive, on établit en général une relation de récurrence entre les coûts pour différentes tailles d'entrée.

■ **Exemple 2** On calcule les termes de la suite (u_n) définie par $u_0 = 0$ et $u_{n+1} = u_n + 1$ si $u_n < 1$ et $u_{n+1} = \dfrac{u_n}{2}$ sinon.

```
def u(n : int) -> float:
    if n == 0:
        return 0
```

```
    else:
        if u(n-1) < 1:
            return u(n-1) + 1
        else:
            return u(n-1)/2
```

Notons c_n la complexité pour l'appel de `u(n)`. Alors, $c_0 = O(1)$ et pour tout $n \geq 1$, $c_n = 2c_{n-1} + 2$. On a une suite arithmético-géométrique et $c_n = O(2^n)$.
On peut améliorer énormément cette complexité :

■

Les fonctions récursives fournissent en général un code concis et élégant, mais qui peut cacher des problèmes de dépassement de pile et de complexité. Par exemple, considérons la suite de Fibonacci :

$$\begin{cases} F_0 = 1 \\ F_1 = 1 \\ F_n = F_{n-1} + F_{n-2} \quad \text{si } n \geq 2 \end{cases}$$

On obtient naturellement la fonction récursive suivante :

```
def fibo(n):
```

Celle-ci pose un problème important : lors de l'exécution de `fibo(5)`, la fonction `fibo(2)` est exécutée 3 fois, et il y a 15 appels en tout ! La complexité est exponentielle.
On peut dérécursifier la fonction précédente pour obtenir une complexité linéaire, mais une fonction moins facilement lisible :

```
def fibo_iter(n):
```

On peut aussi résoudre le problème soulevé par la fonction récursive en utilisant une technique appelée mémoïsation : l'idée est de sauvegarder au fur et à mesure les valeurs de la suite pour éviter d'avoir à les recalculer.
Pour cela, on utilise un dictionnaire `d` dont les clés seront les entiers n et les valeurs les F_n qu'on a déjà calculé. A chaque appel, on commence par vérifier si n est dans `d`, auquel cas, on renvoie la valeur associée. Sinon, on appelle récursivement la fonction pour calculer F_n et on stocke cette valeur dans le dictionnaire avant de la renvoyer. On obtient par exemple :

```python
def fibo_memo(n, d={0:1, 1:1}):
    if n in d:
        return d[n]
    else:
        F = fibo_memo(n-2, d) + fibo_memo(n-1, d)
        d[n] = F
        return F
```

On utilise ici une astuce pythonesque : lorsqu'on appelle la fonction `fibo_memo(5)` (sans préciser `d`) le paramètre `d` est automatiquement crée avec valeur `{0:1, 1:1}`.

IV Exercices

Exercice 1 — Mystère. On considère les deux fonctions suivantes :

```python
def mys(n):
    if n == 0:
        return True
    else:
        return tere(n-1)

def tere(n):
    if n == 0:
        return False
    else:
        return mys(n-1)
```

1. Exécuter à la main les instructions mys(4) et tere(4).
2. Que font ces fonctions ? Le démontrer par récurrence.
3. Recopier et commenter ces fonctions.
4. Comment modifier la fonction mys pour qu'elle renvoie la même chose mais sans utiliser la fonction tere ?

■

Exercice 2 On considère la fonction suivante :

```python
def decompte(n):
    if n >= 0:
        print(n)
        decompte(n-1)
```

1. Exécuter à la main cette fonction avec de petites valeurs de n. Commenter cette fonction.
2. Écrire une fonction récursive compte(n) qui affiche les entiers de 0 à n.

■

Exercice 3 Écrire une fonction récursive `dichotomie_rec(L, e, g, d)` qui prend en paramètres une liste triée par ordre croissant, un nombre `e` et deux indices `g` et `d` et qui renvoie `True` si `e` est dans `L[g:d]` en utilisant le principe de dichotomie. ■

Exercice 4 — Méthode de Hörner. Soit $P = \sum_{k=0}^{n} a_k X^k$ un polynôme. L'algorithme de Hörner se base sur l'égalité suivante :

$$P = ((\ldots((a_n X + a_{n-1})X + a_{n-2})X + \ldots)X + a_1)X + a_0$$

pour calculer $P(b)$ pour un flottant b. Le polyôme P est représenté par la liste $[a_n, \ldots, a_0]$.
Écrire une fonction récursive `hornerrec(P, b)` qui prend un polynôme et un flottant et qui renvoie la valeur de `P(b)` en utilisant l'algorithme de Hörner. Cette fonction ne doit pas modifier `P`. ■