

## TP (S1) 3

## Récursivité

- 1 **Suites récurrentes & Listes** .....
- 2 **Deux grands classiques** .....
- 3 **Dessins** .....

**Objectifs**

- Introduction aux fonctions récursives.
- Exemples de problèmes résolus par récursivité.

Fichier externe?

**NON** pas de fichier externe dans ce TP**1. SUITES RÉCURRENTES & LISTES****1.1. Suites récurrentes****Exercice 1 Suites récurrentes** [Sol 1]

1. Écrire une fonction récursive puis itérative  $u(n: \text{int}) \rightarrow \text{float}$ , qui calcule le  $n^{\text{ème}}$  terme de la suite définie par :

$$u_0 = 2, \quad \forall n \in \mathbb{N}, \quad u_{n+1} = \frac{1}{2} \left( u_n + \frac{3}{u_n} \right).$$

2. Même question, mais uniquement une version récursive, avec les suites  $(v_n), (w_n)$  définies par :

$$\mathbf{2.1)} \quad v_0 = 1, \quad \text{et} : \quad \forall n \in \mathbb{N}, \quad v_{n+1} = 2v_n + n,$$

$$\mathbf{2.2)} \quad w_0 = 1, \quad w_1 = 0, \quad \text{et} : \quad \forall n \in \mathbb{N}, \quad w_{n+2} = w_{n+1}^2 + 2w_n. \text{ Pour cette question, on s'autorisera une version « naïve » comme pour la suite de FIBONACCI vue en cours. Nous verrons dans un prochain TP comment améliorer (en terme de nombre d'appels récursifs) ce type de script.}$$

**Exercice 2 Exponentiation** [Sol 2]

1. **[Naïve]** Écrivez une fonction récursive  $\text{puissance}(x: \text{float}, n: \text{int}) \rightarrow \text{float}$  renvoyant  $x^n$  pour  $n$  entier naturel, en exprimant la définition sous la forme :

$$x^n = \begin{cases} 1 & \text{si } n = 0 \\ x \times x^{n-1} & \text{si } n \in \mathbb{N}^*. \end{cases}$$

Listez à la main les appels successifs engendrés par  $\text{puissance}(2, 10)$  pour cette première version.

2. **[Rapide]** Dans l'exemple précédent, comme sur celui de la factorielle, l'appel récursif se fait en diminuant la variable  $n$  de 1. Il est parfois possible de la réduire davantage (pour minimiser le nombre d'appels nécessaires), et souvent de la diviser par 2, en utilisant un principe **dichotomique**. Reprendre la fonction précédente avec la définition suivante :

$$x^n = \begin{cases} 1 & \text{si } n = 0 \\ (x^2)^{\frac{n}{2}} & \text{si } n \in \mathbb{N}^* \text{ est pair} \\ x \times (x^2)^{\frac{n-1}{2}} & \text{si } n \in \mathbb{N}^* \text{ est impair.} \end{cases}$$

On utilisera le quotient de la division euclidienne de  $n$  par 2 (qui s'obtient par  $n // 2$ ), et qui est égal à  $\frac{n}{2}$  si  $n$  est pair et à  $\frac{n-1}{2}$  si  $n$  est impair.

Listez à la main les appels successifs engendrés par  $\text{puissance}(2, 10)$  pour cette seconde version. Commentez.

**1.2. Listes****Exercice 3 Test de croissance** [Sol 3]

1. Écrire une fonction récursive  $\text{test\_croiss}(L: \text{list}) \rightarrow \text{bool}$  qui détermine si la liste fournie en paramètre est croissante ou non. Par exemple,  $\text{test\_croiss}([1, 2, 3])$  renverra **True**, alors que  $\text{test\_croiss}([2, 1, 3])$  renverra **False**. On décrètera par convention que la liste vide est croissante.
2. Même question mais avec une version impérative, utilisant une boucle **while**.

**Exercice 4 Maximum, le retour** [Sol 4] Écrire une fonction récursive `maximum(L:list)->float` qui renvoie la valeur maximale des termes d'une liste fournie en paramètre (supposée non vide).

**Exercice 5 Décomposition en base d'un entier** [Sol 5] Il arrive que l'on puisse réduire une variable autrement qu'en la diminuant de 1 ou en la divisant par 2. Par exemple, écrivons une fonction `enBase(b : int, n : int) -> list` renvoyant, sous la forme d'une liste, l'écriture de l'entier naturel  $n$  en base  $b$  (l'entier  $b$  est supposé supérieur ou égal à 2). On rappelle qu'une écriture en base  $b$  s'exprime sous la forme de la liste :

$$[c_k, c_{k-1}, \dots, c_2, c_1, c_0]$$

où les chiffres  $c_0, \dots, c_k$  sont des entiers compris entre 0 et  $b - 1$ , et représente l'entier

$$n = c_0 + c_1 \times b + c_2 \times b^2 + \dots + c_{k-1} \times b^{k-1} + c_k \times b^k.$$

Par exemple la liste `[1, 0, 0, 1, 1]` est l'écriture en base 2 de l'entier

$$1 + 1 \times 2 + 0 \times 4 + 0 \times 8 + 1 \times 16 = 19.$$

Pour obtenir l'écriture en base  $b$  d'un entier naturel  $n$ , on utilise le fait que le « chiffre des unités »  $c_0$  est le reste de la division euclidienne de  $n$  par  $b$ , et que le quotient de cette division euclidienne a pour écriture `[c_k, c_{k-1}, ..., c_2, c_1]` en base  $b$ . On peut donc adopter la définition récursive suivante pour l'écriture en base  $b$  d'un entier naturel  $n$  (en distinguant le cas terminal correspondant à un nombre d'un seul chiffre en base  $b$ ) :

$$\begin{cases} [n] & \text{si } n < b \\ \text{insérer } n \% b \text{ à la fin de la liste } \text{enBase}(b, n // b) & \text{si } n \geq b \end{cases}$$

Écrivez le code de la fonction récursive `enBase`. Listez à la main les appels successifs engendrés par `enBase(2, 19)`.

**Exercice 6 Sous-listes et permutations** [Sol 6] Cet exercice est plus délicat.

1. Si  $L$  est une liste, on appelle *sous-liste* de  $L$  toute liste obtenue en enlevant un nombre quelconque d'éléments de  $L$  (et en conservant les éléments restants dans le même ordre). Par exemple, les listes suivantes sont toutes des sous-listes de la liste  $L = [1, 2, 3, 4, 5, 6]$  : `[1, 3, 5, 6]`, `[2, 6]`, `[]`, `[4, 5]`.

Écrivez une fonction récursive `sous_listes(L : list) -> list` renvoyant la liste de toutes les sous-listes de  $L$ . (Notez que le résultat sera donc une liste de listes.) Par exemple l'appel `sous_listes([1, 2, 3])` doit renvoyer (à l'ordre des sous-listes près) : `[[], [1], [2], [3], [1, 2], [1, 3], [2, 3], [1, 2, 3]]`.

*Indication :* On pourra appeler récursivement la fonction sur la liste amputée de son dernier élément, puis conserver ces sous-listes et leur ajouter toutes celles

obtenues en leur ajoutant le dernier élément de la liste initiale. Quel est le cas d'arrêt ?

2. Si  $L$  est une liste, on appelle *permutation* de  $L$  toute liste obtenue en modifiant l'ordre des éléments de  $L$ . Par exemple, les listes suivantes sont toutes les permutations possibles de la liste  $L = [1, 2, 3]$  :

$$[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1].$$

Écrivez une fonction récursive `permutations(L : list) -> list` renvoyant la liste de toutes les permutations de  $L$ .

## 2. DEUX GRANDS CLASSIQUES

### 2.1. L'algorithme de HÖRNER

Soit  $n \in \mathbb{N}$  et soit  $p$  une fonction polynomiale de degré  $n$ ,  $p : x \mapsto a_n x^n + \dots + a_1 x + a_0$ . Une telle fonction sera représentée en Python par la liste  $P$  de taille  $n + 1$  de coefficients rangés par degrés décroissants (les coefficients d'un polynôme étant uniques) :  $P = [a_n, a_1, a_0]$ . Par exemple :

- $x \mapsto 2x^2 + x - 1$  sera représentée par la liste `[2, 1, -1]`,
- $x \mapsto 2x^3 + x^2 - x$  sera représentée par la liste `[2, 1, -1, 0]`.

Le but de l'exercice suivant et de comparer le nombre de multiplications effectuées par différentes fonctions permettant l'évaluation de  $p$  en un réel  $x$  (c'est-à-dire le calcul de  $p(x)$ ).

**Exercice 7** [Sol 7]

1. On propose les deux fonctions suivantes :

```
def eval_pol1(P:list,x:float)->float:
    n, res = len(P)-1, 0
    for k in range(n+1):
        res = res + P[n-k]*x**k
    return res

def eval_pol2(P:list,x:float)->float:
    n, res = len(P)-1, 0
    X = 1
    for k in range(n+1):
        res = res + P[n-k]*X
```

```
X = X*x
return res
```

Vérifiez que ces deux fonctions renvoient le résultat attendu puis explicitez  $u_n$ , resp.  $v_n$ , le nombre de multiplications effectuée par `eval_pol1`, resp. `eval_pol2`, pour évaluer une fonction polynomiale de degré  $n$ .

2. L'algorithme de HORNER repose sur le principe suivant :

- si  $p$  est constante, le calcul de  $p(x)$  est immédiat,
- sinon, il suffit de remarquer que :

$$p(x) = ((a_n x^{n-1} + \dots + a_2) \times x + a_1) \times x + a_0.$$

- 2.1) Écrire une fonction récursive `eval_polH_rec(P:list,x:float)->float` permettant le calcul de  $p(x)$  via l'algorithme de HORNER.
- 2.2) Explicitez le nombre  $w_n$  de multiplications effectuées par `eval_polH_rec` pour évaluer une fonction polynomiale de degré  $n$  et comparez aux fonctions précédentes.
- 2.3) Proposez une version non récursive de l'algorithme de HORNER.

## 2.2. Les tours de HANOÏ

Il y a des problèmes pour lesquels il peut s'avérer utile (et même indispensable) de raisonner récursivement pour leur résolution. En voici un, connu sous le nom des *tours de HANOÏ*.

On dispose de trois tours et d'une pile de  $n$  disques placée par exemple sur la Tour 1. Le but du jeu est de déplacer cette pile sur une des deux autres tours en respectant les règles suivantes :

- On ne peut déplacer qu'un disque à la fois.
- On peut poser un disque sur une tour vide, ou sur un disque de diamètre supérieur uniquement.

L'image animée *Hanoi.gif* dans le dossier partagé permet de visualiser l'animation.

On voudrait écrire une fonction :

```
Hanoi(n:int, Tdepart:int, Tarrivee:int, Taux:int)->None
```

qui affiche à l'écran (avec `print`) les déplacements à effectuer sous la forme  $i \rightarrow j$  où  $i$  et  $j$  sont deux numéros de tours, la pile de  $n$  disques étant initialement sur `Tdepart`, et devant se retrouver sur `Tarrivee` à la fin.

**Exercice 8 Tours de HANOÏ : penser récursif!** [Sol 8]

1. Compléter l'analyse suivante en s'assurant à chaque étape que les règles sont bien respectées :

- Si  $n = 1$  : il y a juste à afficher .....
- Si  $n > 1$  :
  1. pour pouvoir déplacer le plus grand disque, celui-ci doit être libéré, pour cela il faut déplacer la pile des  $n - 1$  .....de .....vers .....
  2. on peut alors déplacer le plus grand disque de .....vers ..... (les règles sont respectées car .....),
  3. il faut déplacer la pile des  $n - 1$  .....de .....vers ..... (en respectant les règles, ce qui est possible car .....).

2. Écrire la fonction récursive `Hanoi(n, Tdepart, Tarrivee, Taux)` et la tester avec de petites valeurs de  $n$ . On peut vérifier ses solutions à cette adresse <http://championmath.free.fr/tourhanoi.htm>.

## 3. DESSINS

### 3.1. En utilisant print

**Exercice 9** [Sol 9]

1. Que va produire l'appel de `etoiles1(5)` pour la fonction récursive suivante? (Essayez de le deviner d'abord sans taper la fonction, puis faites-le pour contrôler.)

```
def etoiles1(n : int) -> None :
    if n > 0:
        print('*'*n)
        etoiles1(n-1)
```

2. Même question pour l'appel de `etoiles2(5)` :

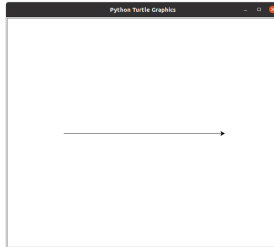
```
def etoiles2(n : int) -> None :
    if n > 0:
        etoiles2(n-1)
        print('*'*n)
```

### 3.2. En utilisant turtle pour les fractales

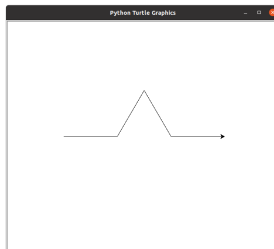
Cette partie est à regarder à la maison.

Une initiation à la récursivité ne saurait être complète sans un tracé de courbe *fractale*. Intéressons-nous à la célèbre courbe de Koch. Elle consiste :

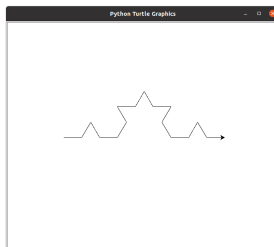
- à la profondeur 0 à tracer un simple segment d'une longueur  $\ell$  donnée (notez que le petit triangle au bout du segment est dû au module `turtle` que nous allons utiliser pour effectuer le tracé) :



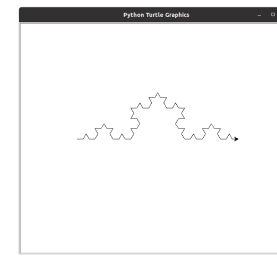
- à la profondeur 1, on remplace le segment précédent par la ligne brisée suivante (chacun des segments de cette ligne a pour longueur le tiers de celui du segment initial) :



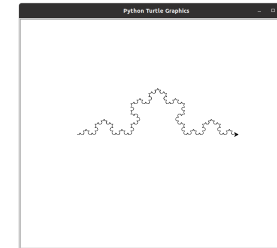
- à la profondeur 2, chacun des segments de la figure précédente est elle-même remplacée par une figure similaire à l'échelle correspondante et orientée dans sa direction :



- et ainsi de suite, à la profondeur 3 :



- puis 4 :



**Exercice 10** [Sol 10] On considère le fichier python ci-dessous (à reprendre sur votre machine par copier/coller depuis le pdf disponible en ligne).

```
from turtle import *
```

```
resetscreen() # efface l'écran
```

```
up() # arrête le tracé
```

```
setposition(-200,0) # se place à gauche de la fenêtre
```

```
down() # remets le tracé
```

```
# tracé de la courbe en profondeur 1 pour une longueur 400
```

```
l = 400
```

```
forward(l/3)
```

```
left(60)
```

```
forward(l/3)
```

```
right(120)
```

```
forward(l/3)
```

```
left(60)
```

```
forward(l/3)
```

Il contient les instructions utilisant le module `turtle` pour tracer le figure en profondeur 1 pour une longueur  $\ell$  positionnée ici à 400.

Écrire une fonction récursive `koch(l : int, p : int) -> None` effectuant le tracé pour une longueur  $\ell$  et à une profondeur  $p$ .