TP 5 INFO Récursivité

Dans ce TP, nous introduisons et étudions la notion de récursivité.

Partie I Généralités

La récursivité est une méthode de programmation dans laquelle une fonction s'appelle elle-même pour résoudre un problème. Dans de nombreuses situations, l'utilisation de la récursivité permet de rédiger des solutions relativement simples à des problèmes qui paraissaient à priori complexes.

Définition (Fonction récursive) : Une fonction est dite récursive si elle s'appelle elle-même.

Exemple 1 : La fonction récursive ci-dessous permet de calculer la factorielle d'un entier.

```
def factorielle(n:int) -> int:
    if n==0:
        return(1)
    else:
        return(n*factorielle(n-1))
```

Remarque 1 : La notion de récursivité en informatique est l'analogue de la définition d'un objet par récurrence en mathématiques (comme les suites récurrentes, certaines figures fractales, etc.).

ATTENTION : Une mauvaise utilisation d'une fonction récursive peut avoir pour conséquence un programme qui ne se termine pas, comme sur l'exemple ci-dessous.

```
>>> factorielle(-1)
RecursionError: maximum recursion depth exceeded
```

Pour ce cas, on pourrait ajouter une assertion au début de notre fonction afin d'éviter ce problème.

```
assert type(n)==int and n>=0
```

Par contre, la même problématique se produit avec l'exemple ci-dessous.

```
>>> def test(n:int) -> int:
...     return(test(n-1))
>>> test(0)
RecursionError: maximum recursion depth exceeded
```

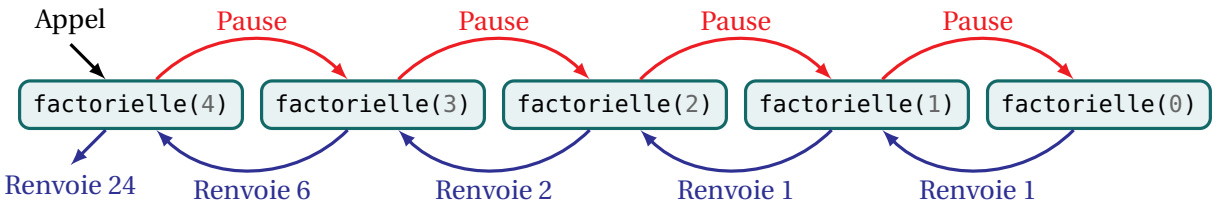
Ainsi, pour qu'une fonction récursive s'arrête, deux éléments sont indispensables :

- il est nécessaire qu'il y ait une condition d'arrêt dans le corps de la fonction ;
- l'appel de la fonction pour une entrée valide ne doit pas engendrer une suite infinie d'appels récursifs.

En pratique, la condition d'arrêt correspond au traitement des cas où l'entrée se trouve sur le « bord » du domaine de validité des entrées.

Remarque 2 : Lorsque l'ordinateur utilise une fonction récursive, dès qu'il rencontre un nouvel appel à la fonction, il va mettre en pause l'exécution de la fonction actuelle en sauvegardant (dans ce qu'on appelle la pile d'exécution) les données nécessaires pour pouvoir reprendre l'exécution où elle s'était arrêtée dès que le résultat de l'appel à la nouvelle fonction a été obtenu. Ce procédé va se répéter jusqu'à tomber sur la condition d'arrêt, ce qui engendre nécessairement un coût en mémoire (pour sauvegarder les données de toutes les fonctions en pause).

Illustration : Lorsqu'on exécute `factorielle(4)`, on peut représenter la situation avec le schéma suivant.



Lorsqu'il y a trop d'appels récursifs qui sont en attente, l'interprète Python arrête l'exécution du programme et il nous renverra l'erreur `RecursionError`. Par défaut, cette limite est fixée à 1000 appels récursifs.

Partie II Quelques fonctions récursives élémentaires

Dans cette partie, on rédige quelques fonctions récursives très simples. La récursivité a peu d'intérêt sur ces exemples, mais ils permettent de se familiariser avec cette nouvelle notion.

Question 1 : Écrire une fonction récursive `somme(L:list) -> int` qui prend comme argument une liste d'entiers `L` et qui retourne la somme des éléments de `L`.

Question 2 : Écrire une fonction récursive `renverser(chaine:str) -> str` qui prend en argument une chaîne de caractères `chaine` et renvoie une version renversée de la chaîne de caractère. Par exemple, l'exécution de `renverser("abcd")` doit renvoyer `"dcba"`.

Question 3 : Écrire une fonction récursive `somme_decimales(n:int) -> int` qui prend comme argument un entier naturel `n` et qui retourne la somme de ses chiffres.

Partie III Générer des figures alphanumériques

Dans cette sous-partie, on écrit quelques fonctions récursives permettant d'afficher à l'écran des figures composées de caractères.

Question 4 : Écrire une fonction récursive `triangle1(n:int) -> None` qui prend en argument un entier naturel `n` et affiche un triangle isocèle « croissant » de hauteur `n` composé de `#`. Voici un exemple ci-dessous.

```
>>> triangle1(4)
#
##
###
####
```

Question 5 : Écrire une fonction récursive `triangle2(n:int) -> None` qui prend en argument un entier naturel `n` et affiche un triangle isocèle « décroissant » de hauteur `n` composé de `#`. Voici un exemple ci-dessous.

```
>>> triangle2(4)
####
###
##
#
```

Partie IV Récursivité et algorithmes dichotomiques

IV.A - Algorithme de recherche dichotomique dans un tableau trié

On souhaite écrire une version récursive de l'algorithme de recherche dichotomique dans un tableau trié qui permet de savoir si un élément `x` appartient à un tableau `t` d'entiers triés dans l'ordre croissant. Rappelons succinctement le principe de cet algorithme. On note c un indice pointant (approximativement) sur un élément au milieu du tableau, puis on distingue deux cas :

- si `x < t[c]`, alors on poursuit la recherche dans la première partie du tableau ;
- sinon on poursuit la recherche dans la seconde partie du tableau.

La recherche s'arrêtera et pourra renvoyer une réponse lorsqu'on tombera sur un tableau ne contenant qu'un élément.

Question 6 : Écrire une fonction récursive `recherche_dicho(t:tuple, x:int) -> bool` prenant en argument un tuple d'entiers rangés dans l'ordre croissant et implémentant l'algorithme décrit ci-dessus.

Remarques 3 :

a) L'algorithme décrit ci-dessus utilise la méthode **diviser pour régner** qui se divise généralement en les trois étapes ci-dessous.
   1) **Diviser :** étant donné le problème à résoudre, on découpe l'entrée en deux ou plusieurs morceaux.
   2) **Résoudre :** à l'aide d'appels récursifs, on résout le problème sur chacun des morceaux.
   3) **Combiner :** à partir des résultats des appels récursifs, on construit la solution du problème de départ.

   Nous verrons d'autres algorithmes s'appuyant sur cette méthode dans l'année.

b) On peut vérifier que la complexité de l'algorithme d'exponentiation rapide est $O\big(\log(n)\big)$.

IV.B - Algorithme d'exponentiation rapide

L'objectif de cette partie est de rédiger l'algorithme d'exponentiation rapide qui permet de calculer efficacement a^n pour $a \in \mathbb{Z}$ et $n \in \mathbb{N}^*$. L'idée de base est d'exploiter la relation

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

Pour répondre à la question ci-dessous, il n'est donc pas permis d'utiliser l'opérateur `**` de Python.

Question 7 : Écrire une fonction récursive `expo_rapide(a:int, n:int) -> int` prenant en argument un entier `a` et un entier naturel non nul `n` et qui renvoie a^n.

Remarques 4 :

a) L'algorithme d'exponentiation rapide s'appuie également sur la méthode diviser pour régner.

b) On peut vérifier que la complexité de l'algorithme d'exponentiation rapide est $O\big(\log(n)\big)$.

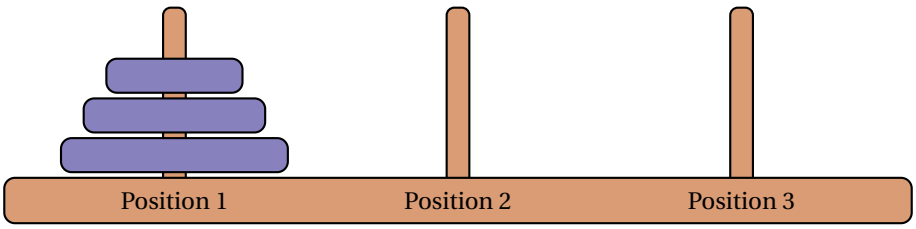
Partie V Les tours de Hanoï

Les tours de Hanoï est un jeu de réflexion consistant à déplacer n disques de diamètres différents d'une position de départ à une position d'arrivée en passant par une position intermédiaire en respectant les règles suivantes :

- on ne peut déplacer plus d'un disque à la fois ;
- on ne peut placer un disque que sur un autre disque plus grand que lui ou sur un emplacement vide.

On suppose que cette dernière règle est également respectée dans la configuration de départ.

Illustration : Le schéma ci-dessous représente le jeu des tours de Hanoï avec $n = 3$ disques. L'objectif du jeu de déplacer les disques de la position 1 à la position 3 en respectant les règles décrites ci-dessus.



Pour décrire un mouvement d'un disque au sommet d'une tour à la position `i` vers la tour à la position `j`, nous utiliserons la notation `i->j`. Avec ces notations, une solution du jeu pour $n = 3$ est donnée par la succession de mouvements `1->3, 1->2, 3->2, 1->3, 2->1, 2->3, 1->3`.

Pour résoudre le jeu pour un nombre quelconque $n \in \mathbb{N}^*$ de disques, on peut construire la solution récursivement de la manière suivante.

1) On déplace les $n-1$ disques au-dessus de la tour de la position de départ vers la position intermédiaire en utilisant la récursivité.
2) On déplace le plus grand disque de la position de départ vers la position d'arrivée.
3) On déplace les $n-1$ disques de la position intermédiaire vers la position d'arrivée en utilisant la récursivité.

Question 8 : Écrire une fonction récursive `hanoi(n:int, pos_dep:int, pos_arr:int) -> list` prenant en argument le nombre de disques `n`, leur position au départ `pos_dep` et leur position d'arrivée `pos_arr` et qui renvoie la liste des mouvements à effectuer pour résoudre le jeu. Par exemple, on aura le résultat suivant.

```
>>> hanoi(3, 1, 3)
['1->3', '1->2', '3->2', '1->3', '2->1', '2->3', '1->3']
```

Partie VI Générer des fractales

Pour traiter cette partie, commencer par recopier le code ci-dessous dans votre fichier Python. Il nous permettra dans la suite de dessiner des figures géométriques dans une fenêtre graphique.

```
import matplotlib.pyplot as plt
fig, ax = plt.subplots(1)
plt.axis("equal") # Repère orthonormée

# Pour régler l'emplacement de la fenêtre d'affichage.
x_min, x_max = -2, 4
y_min, y_max = -2, 4
ax.set_xlim(x_min, x_max)
ax.set_ylim(y_min, y_max)
```

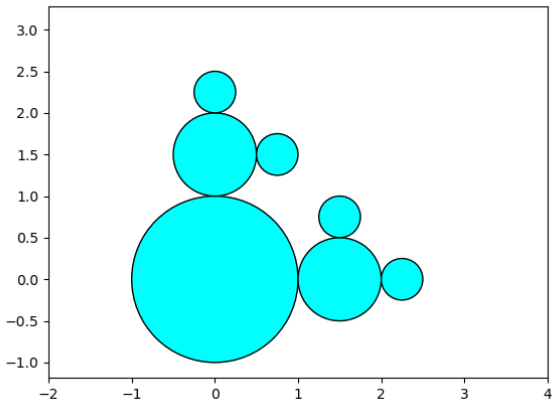
Rappelons qu'une fois le dessin tracé, il faut utiliser la commande `plt.show()` pour afficher la fenêtre graphique.

VI.A - Un arbre de disques

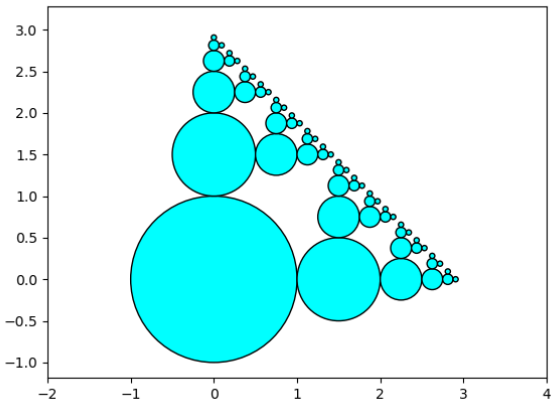
Dans cette sous-partie, nous aurons également besoin de la fonction ci-dessous qui permet de dessiner un disque dont les coordonnées du centre et le rayon sont passés en paramètre.

```
# Fonction dessinant un cercle de centre (x, y) et de rayon r
def circle(x:float, y:float, r:float):
    ax.add_artist(plt.Circle((x,y), r, ec='k', fc='cyan'))
```

Question 9 : Écrire une fonction `arbre(n:int, x:float, y:float, r:float) -> None` prenant en argument les coordonnées du centre `(x, y)` et le rayon `r` du plus grand disque et qui génère les graphiques ci-dessous.



Résultat de `arbre(3, 0, 0, 1)`



Résultat de `arbre(6, 0, 0, 1)`

VI.B - Les triangles de Sierpiński

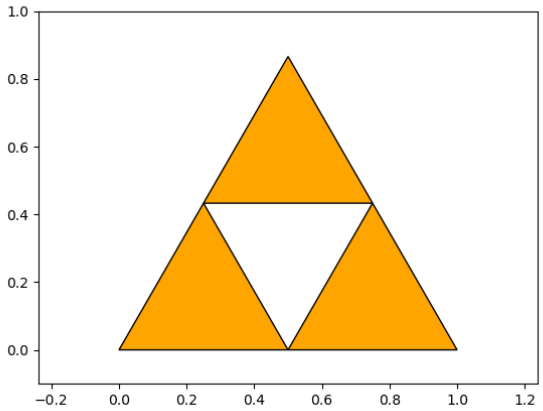
Dans cette sous-partie, nous aurons également besoin de la fonction ci-dessous qui permet de dessiner un triangle dont les coordonnées des sommets sont passées en paramètre.

```
from matplotlib.patches import Polygon
# Fonction dessinant un triangle plein à partir des coordonnées des sommets
def triangle(x1:float, y1:float, x2:float, y2:float, x3:float, y3:float) -> None:
    lst = [[x1, y1], [x2, y2], [x3, y3]]
    ax.add_patch(Polygon(lst, closed = True, ec='k', fc='orange'))
```

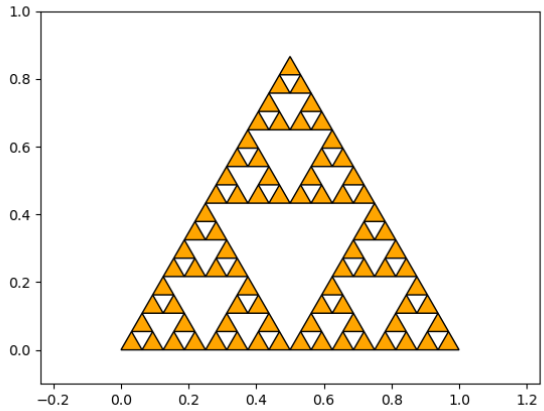
Question 10 : Écrire une fonction

```
sierpinski(n:int, x:float, y:float, c:float) -> None
```

prenant en argument les coordonnées (x, y) d'un sommet d'un triangle et le côté c d'un triangle et qui génère les graphiques ci-dessous.



Résultat de `sierpinski(1,0,0,1)`



Résultat de `sierpinski(4,0,0,1)`

Partie VII Problèmes d'énumération

VII.A - Construction des arrangements avec répétition d'une chaîne de caractères

Un k-arrangement avec répétition d'une chaîne de caractères `chaine` est une chaîne de caractères de longueur k que l'on peut construire en prenant des lettres, avec éventuellement des répétitions, dans `chaine`.

Exemple 2 : Les 2-arrangement avec répétition de `abc` sont `aa, ab, ac, ba, bb, bc, ca, cb, cc`.

Question 11 : Écrire une fonction récursive `arr_avec_rep(chaine:str, k:int) -> list` prenant en argument une chaine de caractères `chaine` et un entier naturel `k` et qui renvoie une liste contenant tous les k-arrangement avec répétition de `chaine`. Par exemple, on aura le résultat suivant.

```
>>> arr_avec_rep('abc', 2)
['aa', 'ab', 'ac', 'ba', 'bb', 'bc', 'ca', 'cb', 'cc']
```

VII.B - Construction des arrangements sans répétition d'une chaîne de caractères

Un k-arrangement sans répétition d'une chaîne de caractères `chaine` est une chaîne de caractères de longueur k que l'on peut construire en prenant des lettres, sans répétition, dans `chaine`.

Exemple 3 : Les 2-arrangement sans répétition de `abc` sont `ab, ac, ba, bc, ca, cb`.

Question 12 : Écrire une fonction récursive `arr_sans_rep(chaine:str, k:int) -> list` prenant en argument une chaine de caractères `chaine` et un entier naturel `k` et qui renvoie une liste contenant tous les k-arrangement sans répétition de `chaine`. Par exemple, on aura le résultat suivant.

```
>>> arr_sans_rep('abc', 2)
['ab', 'ac', 'ba', 'bc', 'ca', 'cb']
```

VII.C - Construction des sous-séquences d'une chaîne de caractères

Une sous-séquence de longueur k d'une chaîne de caractère `chaine` est une chaîne de caractères de longueur k que l'on peut construire en prenant des lettres, sans répétition et en respectant leur ordre, dans `chaine`.

Exemple 4 : Les sous-séquences de longueur 2 de `abc` sont `ab, ac, bc`.

Question 13 : Écrire une fonction récursive `sous_seq(chaine:str, k:int) -> list` prenant en argument une chaîne de caractères `chaine` et un entier naturel `k` et qui renvoie une liste contenant toutes les sous-séquences de longueur `k` de `chaine`. Par exemple, on aura le résultat suivant.

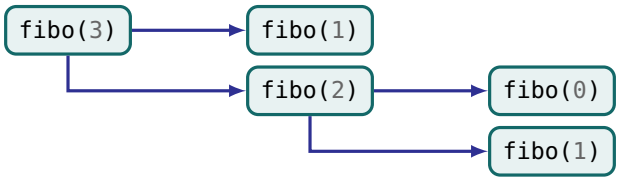
```
>>> sous_seq('abc', 2)
['ab', 'ac', 'bc']
```

Partie VIII Un premier pas vers la programmation dynamique

La simplicité de rédaction d'un algorithme récursif peut cacher une complexité non satisfaisante. Dans cette partie, on s'intéresse à la suite de Fibonacci $(F_n)_{n\in\mathbb{N}}$ définie par $F_0 = 0$, $F_1 = 1$ et $F_{n+2} = F_{n+1} + F_n$ pour tout $n \in \mathbb{N}$.

Question 14 : Écrire une fonction récursive `fibo(n:int) -> int` basée directement sur la définition ci-dessus qui prend en argument un entier naturel `n` et renvoie le nombre F_n.

Si l'on souhaite calculer le nombre F_3 avec notre fonction `fibo`, on peut représenter les appels successifs de la fonction `fibo` sous la forme d'un arbre comme ci-dessous.



Question 15 : Représenter l'arbre associé à l'exécution de `fibo(5)`.

Question 16 : Combien d'appels à la fonction `fibo` ont été effectués pour calculer F_5 ?

Si on note a_n le nombre d'appels à la fonction `fibo` pour calculer F_n pour tout $n \in \mathbb{N}$, on peut démontrer que

$$a_n \underset{n\to+\infty}{\sim} \frac{2}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^{n+1}.$$

On en déduit que la complexité temporelle de la fonction `fibo` est exponentielle.

Les arbres précédents nous permettent d'observer l'origine du problème : l'ordinateur recalcule plusieurs fois les mêmes valeurs. Pour contourner cette difficulté, nous allons utiliser une nouvelle méthode : la **mémoïsation**. Cette dernière consiste à mémoriser l'ensemble des résultats intermédiaires afin que l'ordinateur ne refasse pas plusieurs fois des calculs identiques.

Question 17 : En utilisant un dictionnaire pour mémoriser les résultats intermédiaires, écrire une fonction récursive `fibo_dico(n:int) -> int` effectuant plus efficacement la même tâche que `fibo`.

Remarque 5 : On pourra vérifier que $F_{100} = 354\,224\,848\,179\,261\,915\,075$.