TP d'informatique n°08 et 09
Récursivité

PCSI 2023 – 2024

I Objectifs :

- savoir ce qu'est un algorithme récursif et dans quel cas il est pertinent d'en utiliser un ;
- avoir conscience des limites des algorithmes récursifs ;
- ⚠ gérer les « cas de bases » ou « cas d'arrêt ».

II Introduction

En informatique, on dit qu'une fonction (ou un algorithme) est récursive lors que cette fonction s'appelle elle-même. On oppose généralement cela aux algorithmes dit « itératifs » dans lesquels on utilise des boucles **for** ou **while**[1].
Ce type d'algorithme peut faciliter l'écriture du code lorsque la fabrication d'une solution à un problème "de taille n" s'obtient facilement à partir de solution pour "des tailles inférieures à n". Par contre, cela cache un travail de mémorisation de résultats intermédiaires pour l'ordinateur et cela peut donc conduire à des algorithmes inefficaces si la technique est mal utilisée.
De façon générale, il y a un parallèle très fort entre les programmes récursifs et les démonstrations mathématiques par *récurrence*. S'il est facile de démontrer par récurrence qu'une méthode fourni un résultat correct, il sera probablement facile d'écrire un programme récursif.

1. Un premier exemple

Un exemple classique est la fonction factorielle : connaissant $(n-1)!$ il suffit de multiplier par n pour obtenir $n!$ sauf si $n = 0$ auquel cas on prend pour convention $0! = 1$.

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

Exercice n°1 Exécuter cet algorithme « à la main » (c'est-à-dire sur une feuille de papier, sans utiliser l'ordinateur) pour calculer `fact(5)`.

Remarque : Ce n'est pas un très bon exemple, car il est très facile d'écrire un algorithme itératif ici, mais il a le mérite d'être simple à comprendre.

2. Un exemple de problème

```
import time
def youpie():
    print("vive la physique")
    time.sleep(0.01)#fait une pause pour ralentir l'exécution
    return youpie()
youpie()
```

1. On peut bien entendu concevoir des fonctions récursives qui contiennent des boucles. Ce sera le cas dans certains exos « à faire pour la prochaine fois ».

Exercice n°2 Taper en python le programme ci-dessus et tester le. Que se passe-t'il ? Expliquer.

Remarque : ainsi, comme avec une boucle "while", une fonction récursive mal conçue peut faire partir dans une boucle infinie[2]. Il est donc important, comme dans le cas d'une boucle while, de prévoir une condition "d'arrêt" et de s'assurer que l'on atteindra un jour cette condition.
Dans le cas d'une fonction récursive, cela se fait généralement en gérant à part un ou plusieurs « cas de base » ou « cas d'arrêt » (c'est-à-dire sans appel récursif) et en s'assurant que chaque appel récursif nous « rapproche » des cas de bases. Dans le cas de la fonction `fact`, le cas de base était le cas $n = 0$. On se rapprochait du cas de base car l'argument de la fonction diminuait de 1 à chaque appel.

♡ ⚠ Lorsque l'on code une fonction récursive, il faut (1) bien réfléchir aux cas de base (2) s'assurer que chaque appel récursif nous "rapproche" de ces cas de base pour que l'algorithme se termine.

Exercice n°3 Il est possible de limiter le nombre d'appels à la fonction en mettant un argument qui sert de compteur. Lorsque la fonction fait un appel récursif, elle le fait avec le $compteur - 1$ et on prévoit un arrêt lorsque l'argument (donc le compteur vaut 0). Corriger ainsi la fonction précédente pour qu'elle s'exécute n fois.

3. Un meilleur exemple : la suite de Fibonacci

La suite de Fibonacci est une suite d'entiers définie de la façon suivante : $F_0 = 0$; $F_1 = 1$ et $\forall n \in \mathbb{N}, \quad F_{n+2} = F_n + F_{n+1}$. Vous avez codé un algorithme itératif correspondant à cette suite dans le TP 1.

Exercice n°4 On veut écrire un algorithme récursif, quel(s) est(sont) le(s) cas de base à gérer ?
(prenez le réflexe de vous poser cette question).

♡ ***Exercice n°5*** Écrire une fonction python récursive prenant en argument un entier n et renvoyant F_n.

Exercice n°6 Testez pour quelques valeurs $n \leqslant 10$ (vous pouvez comparer les résultats avec ceux de l'algorithme itératif codé dans le TP 1).

Remarque : Cet exemple est d'un certain point de vue bien meilleur que le précédent, car en général lors du TP 1, vous avez du mal à coder cette fonction et vous en avez beaucoup moins pour la version récursive. Toutefois, il y a un problème majeur avec la fonction que vous avez probablement codé comme nous allons le mettre en évidence dans le prochain paragraphe.

4. Un problème avec la suite de Fibonacci codé naïvement

Exercice n°7 Calculer à la main (sur une feuille de papier) F_5, combien de «calculs» (additions) avez vous fait ? Combien en feriez vous pour calculer F_{10} ? F_n ?

Exercice n°8 Dans votre programme, ajoutez une variable globale qui vaut 0 (au tout début de votre programme) et qui augmente de 1 à chaque passage dans votre fonction (vous devriez avoir quelque chose qui ressemble au programme ci-dessous). Utilisez votre fonction pour calculer F_{10} puis regardez le contenu de la variable compteur après le calcul pour voir le nombre de passage dans la fonction. Faire de même pour F_{30}. Commenter et expliquer.

2. En python, il existe une protection qui interrompt l'exécution dans le cas où le nombre d'appel récursif d'une fonction est trop grand et renvoie une erreur, mais il vaut mieux ne pas compter dessus !

```
compteur = 0
def fibo(n):
    global compteur # on précise que l'on utilise une variable globale
    compteur += 1  # on incrémente (+ 1) le compteur à chaque passage
    ... # le reste de votre programme
```

Remarque : Lorsqu'un algorithme est appelé de nombreuses fois avec la même valeur en argument, il peut être pertinent de mémoriser le résultat lors du premier calcul pour le réutiliser lors des appels suivant. On parle de « mémoïsation ». Cela est à votre programme d'informatique de deuxième année.

III À vous de jouer

1. Quelques petits exemples

Exercice n°9 Coefficents binomiaux
On rappelle la formule de Pascal pour les coefficents binomiaux :

$$\binom{0}{0} = 1 \quad ; \quad \forall n,k \in \mathbb{N}^2, k \leqslant n \qquad \binom{n+1}{k+1} = \binom{n}{k+1} + \binom{n}{k} \quad ; \quad \text{si } k < 0 \text{ ou } k > n, \text{ alors } \binom{n}{k} = 0$$

Écrire une fonction `coefBinom(k:int, n:int)->int` prenant en argument deux entiers naturels k,n et renvoyant la valeur de $\binom{n}{k}$.

Exercice n°10 Recherche dichotomique dans une liste triée.
Le principe de la recherche dichotomique dans une liste triée a été vue dans le TP sur les algorithmes logarithmiques. On veut renvoyer $True$ si le nombre x appartient à la liste L et $False$ sinon. Pour cela, le principe était de regarder "au milieu" de la liste, puis en fonction du nombre vu au milieu, renvoyer $True$ ou chercher à "gauche" ou à "droite". Réaliser une fonction récursive réalisant cela. On rappelle que la syntaxe `L[d:f]` avec d et f entier permet de sélectionner[3] tous les éléments de L d'indice i compris entre d inclus et f exclus ($d \leqslant i < f$).

Exercice n°11 Exponentiation rapide.
Dans un certain nombre de protocoles informatiques sécurisés (échanges cryptés), il est utile de calculer des puissances très élevées.

1. Écrire (ou retrouver dans un tp précédent) une fonction itérative « naïve[4] » `puissance(k:int, n:int)->int` prenant deux entiers naturels k,n et renvoyant la valeur de k^n. On s'interdira bien sûr d'utiliser l'opération $**$ et on se limitera à l'opération $*$. Combien de multiplications réalise votre programme ?
   Comme dans le TP sur les algorithme dichotomique, on remarque que :
   — si n est pair, $k^n = (k^{n/2})^2$ (avec $n/2$ un nombre entier, on utilisera donc en python $//$) ;
   — si n est impair, $k^n = k \times k^{n-1}$.
2. Écrire une fonction récursive `puissanceRapide(k:int, n:int)->int` prenant deux entiers naturels k,n et renvoyant la valeur de k^n en utilisant les astuces ci-dessus. N'oublier pas de tester votre fonction.

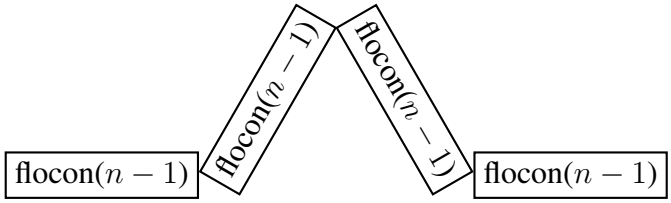
3. On crée en fait une copie de la liste, ce qui détériore fortement les performances de l'algorithme. Pour faire un meilleur algorithme, on aurait intérêt à passer en argument L, d, f, mais l'algorithme n'est alors pas plus simple que l'algorithme itératif.
4. C'est-à-dire sans chercher particulièrement à optimiser le temps de calcul.

3. Ajouter un compteur et regarder le nombre d'appels récursifs pour calculer 2^{100}.
4. S'il faut x appels récursifs pour calculer k^n, combien en faut-il pour calculer k^{2n} ? k^{2n+1} ? Et dans le cas de l'algorithme itératif précédent ?

2. Flocon de Koch

On défini un côté du flocon de Koch de rang $n \in \mathbb{N}$ de la façon suivante :

- si n = 0, on trace simplement un trait horizontal, de la gauche vers la droite, d'une longueur fixée,
- si $n > 0$, on trace un côté de flocon de rang $n-1$, puis on tourne de $60°$ dans le sens trigonométrique, on trace un autre côté flocon de rang $n-1$ (sans avoir levé le stylo, donc à la suite), on tourne de $-120°$ (donc $-60°$ par rapport à l'horizontal) on tracé à nouveau un côté de rang $n-1$, puis on tourne de $+60°$ (on revient à l'horizontal) et on tracé un dernier côté de rang $n-1$.



La figure ci-dessus à gauche illustre le principe.

Exercice n°12 À quel n correspond la figure ci-dessus à droite ?
Récupérer le fichier affichageTPrecursivite.py mis à votre disposition et mettez le dans le même dossier que votre programme. On dispose maintenant d'un module affichageTPrecursivite avec quelques fonctionnalitées :

```
import affichageTPrecursivite as affiche # nécessite matplotlib et
   numpy
d = affiche.dessin() # crée un nouveau "crayon" prêt à dessiner
d.avance() #fait avancer le crayon d'une distance fixée vers la droite
d.tourne(90)#tourne de 90° dans le sens trigo la direction dans
   laquelle le crayon va avancer par la suite
d.avance() # le crayon avance (dans la direction +90° maintenant)
d.trace() # affiche le dessin fait par le crayon
```

Exercice n°13 À l'aide du module $affichageTPrecursivite$, dessiner un carré (pour s'entrainer à utiliser le module)

Exercice n°14 Écrire une fonction `KochCote(n,d)` qui prend en argument un entier n et un "crayon" déjà initialisé et qui lui fait dessiner un côté du flocon de Koch de rang n.

On ne fera pas appel à trace() dans la fonction pour le moment.
Testez votre fonction avec le code ci-contre.

```
d = affiche.dessin()
KochCote(3,d)
d.trace()
```

Pour tracer un flocon complet, on suit les étapes : (1) tracer un côté (2) tourner de -120° (3) tracer un côté (4) tourner de -120° (5) tracer un coté.

Exercice n°15 Écrire une fonction `KochComplet(n)` qui initialise un crayon, lui fait dessiner un flocon de rang n complet puis l'affiche.

3. La courbe du dragon

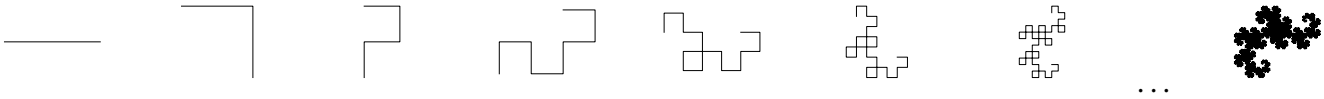
On défini la courbe du dragon de rang $n \in \mathbb{N}$ de la façon suivante :

- si $n = 0$, on trace un trait horizontal
- sinon, on dessine la courbe du dragon de rang $n-1$, on dessine une deuxième fois la courbe du dragon de rang $n-1$ en la tournant de $90°$ et on met les deux courbes bout à bout en collant la fin de la première courbe à la fin de la deuxième courbe.

Le module affichageTPrecursivite permet de réaliser ces opérations :

```
# si d = affiche.dessin() et que l'on a fait plusieurs tracé, alors
d.tourneDessin90() # fait une rotation de +90° de TOUT le dessin
#...si d1 et d2 sont deux dessins, alors
d = d1 + d2 # d = d1 + d2 est le dessin en collant les bouts ensemble
```

Exercice n°16 Écrire une fonction permetant de tracer la courbe du dragon de rang n.

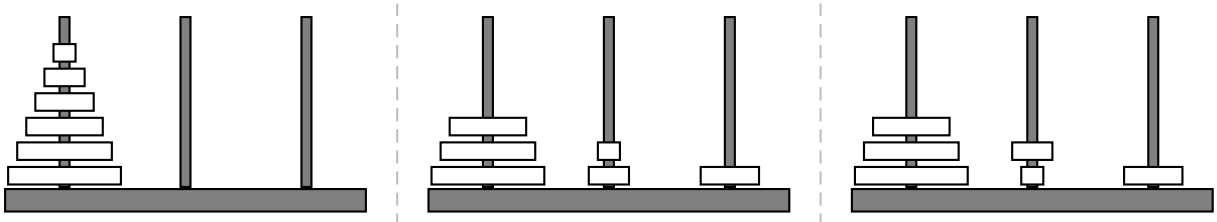


4. Les tours de Hanoï (Exemple <u>important</u> !)

On considère le problème des tours de Hanoï : il s'agit d'un jeu consistant à déplacer des disques de diamètres différents depuis une tour de départ à une tour d'arrivée en respectant quelques règles simples. Les règles sont les suivantes :

1. on ne peut déplacer qu'un disque à la fois ;
2. un disque de diamètre plus grand ne peut pas reposer sur un disque de diamètre plus faible ;
3. trois positions (tours) sont à notre disposition : une de départ, une intermédiaire et une d'arrivée ;
4. les disques sont initialement tous empilés sur la position de départ et sont donc par ordre décroissant de diamètre si on les regarde de bas en haut ;
5. à la fin, les disques doivent être tous empilés à droite (et donc par ordre décroissant aussi).

Si on dispose de n disques, alors chaque disque sera représenté par un entier entre 0 (le plus petit) et $n-1$ (le plus grand). On représentera une configuration par **une** liste de **trois** liste d'entier, la première sous-liste étant (de bas en haut) les disques à la première position, la 2e liste ceux à la 2e position et de même pour la 3e. La figure ci-dessous présente des exemples pour $n = 6$.



Position de départ ($n = 6$)
L = [[5, 4, 3, 2, 1, 0], [], []]

Position intermédiaire valide
L = [[5, 4, 3], [1, 0], [2]]

Position non valide
L = [[5, 4, 3], [0, 1], [2]]

Exercice n°17 Résoudre à la main le jeu pour $n = 3$

Exercice n°18 Sachant résoudre le jeu pour n quelconque, comment le résoudre pour $n' = n+1$? (Lorsque l'on sait répondre à ce genre de question, un algorithme récursif est souvent facile à écrire)

Le module `affichageTPrecursivite` fourni dispose d'une fonction `afficheHanoi(L:list)` qui prend en argument une configuration et réalise un (magnifique) affichage en ASCII. Utilisez le module pour afficher la position intermédiaire valide ci-dessus (pour s'entrainer).

Les listes possèdent une méthode `append(element)` pour rajouter un élément à la fin de la liste (revient à ajouter un disque au dessus d'une pile) et `pop()` qui renvoie le dernier élément de la liste et le supprime de la liste (revient à prendre un disque d'une pile). Vous pouvez par exemple tester le code suivant.

```
L1 = [3,2,1,0]
L2 = []
L2.append(L1.pop())
print(L1,L2)
```

Exercice n°19 Écrire une fonction `resoutHanoiInterm(L:list, depart:int, interm:int, arrive:int, nombre:int)` qui prend en argument :

- une configuration `L` des tours de Hanoï que l'on supposera valide sans le vérifier ;
- trois entier `depart, interm, arrive` de l'ensemble $\{0,1,2\}$ et distincts deux à deux, correspondant aux indices des positions de départ, position intermédiaire et position d'arrivé souhaitée ;
- un entier `nombre` qui correspond au nombre de disque que l'on veut déplacer depuis la tour d'indice `depart` jusqu'à la tour `arrive` (en supposant la tour d'indice interm vide)

La fonction devra modifier la liste de liste `L` de façon à déplacer les disques en respectant les règles et ne renvoie rien. Il est « autorisé » de faire une fonction récursive et d'afficher les position intermédiaire !

Exercice n°20 Écrire une fonction `resoutHanoi(n:int)` qui prend en argument un entier n, crée la configuration de départ puis résout le jeu des tours de Hanoï correspondant à l'entier n (en affichant les différentes étapes de la résolution).

♡

5. Un peu de mémoïsation

Exercice n°21 On reprend l'exemple des coefficients binomiaux, mais on veut améliorer les performances. Pour cela on crée un dictionnaire (variable globale) `sauveBinom = dict()` tel que la valeur associée à une clé (k,n) est $\binom{n}{k}$. Lors des appels récursifs :

- soit le résultat demandé a déjà été calculé et est dans le dictionnaire, on le renvoie directement sans le calculer ;
- soit le résultat n'est pas dans le dectionnaire et on le calcule puis on le sauve dans le dictionnaire avant de renvoyer le résultat

Remarque : plutôt que de manipuler une variable globale, on peut aussi passer en argument le dictionnaire des positions déjà calculées.

Exercice n°22 On reprend l'exemple de la suite de Fibonacci, mais on veut améliorer les performances. Pour cela on crée une liste (variable globale) `sauveFibo = [0,1]` telle que l'objet stocké en n-ième position est F_n. Lors des appels récursifs :

- soit le résultat demandé a déjà été calculé et est dans la liste, on le renvoie directement sans le calculer ;
- soit le résultat n'est pas dans la liste et on le calcule puis on le sauve dans la liste `sauveFibo` avant de renvoyer le résultat (on fera attention qu'il faudra peut-être parfois "faire grandir la liste").

Comparer les performances avec le codage précédent.
Remarque : plutôt que de manipuler une variable globale, on peut aussi passer en argument la liste des valeurs déjà calculées.

Exercice n°23 (Exercice difficile) On reprend l'exemple de la suite de Fibonacci, mais on veut encore améliorer les performances. On ne veut plus utiliser de liste puis que l'on n'a besoin normalement que de deux termes en mémoire. On crée une fonction intermédiaire récursive qui renvoie non pas F_n, mais le tuple (F_n, F_{n+1}). Puis on crée une fonction « maîtresse » qui utilise cette fonction récursive pour ne renvoyer que F_n qui est la valeur qui nous intéresse.
Pour la fonction intermédiaire, on peut aussi faire en sorte que les deux termes successifs de la suite soient en argument et les modifier au fur et à mesure.

IV À faire pour la prochaine fois I

Exercice n°24 Triangles avec des étoiles
Écrire une fonction récursive `triangle1(n:int)` dessinant un triangle tel que ci-contre avec une base formée de n étoiles et une hauteur de n étoiles. On interdit l'utilisation de boucle for ou while et on rappelle que `print("*"*n)` affiche n étoiles et retourne à la ligne.

```
1 *****
2 ****
3 ***
4 **
5 *
```

Exercice n°25 Écrire une fonction `sous_liste(L:list)->list` qui prend en argument une liste L et qui renvoie la liste de toutes les sous-listes de L. Par exemple `sous_liste([1,2,3])` doit renvoyer $[[1,2,3],[2,3],[3],[],[2],[1,3],[1],[1,2]]$ (pas forcément dans cet ordre). Il est possible d'utiliser une fonction récursive avec une boucle for dedans ou une fonction récursive "annexe".

Exercice n°26 Quicksort (non performant ici)
Plusieurs tris performants utilisent le fait que la difficulté à trier augmente rapidement avec le nombre d'objets à trier. On peut donc avoir intérêt à couper une liste que l'on veut trier en sous-listes, trier ces sous-listes puis combiner le résultat pour obtenir notre liste triée. On se propose ici de coder[5] un exemple de tel algorithme.
Si L est une liste à trier non vide, alors :

1. on appelle x le premier élément de L ;
2. on crée trois sous-listes inf, mid, sup ;
3. dans inf, on met tous les éléments de L qui sont strictement inférieurs à x ;
4. dans mid, on met tous les éléments de L qui sont égaux à x ;
5. dans sup, on met tous les éléments de L qui sont strictement supérieurs à x ;
6. une fois inf et sup triées, il suffit de concaténer les trois listes pour obtenir le tri de L.

Quels est(sont) le(s) cas de base à gérer ? Coder ce tri et testez le.

5. D'une mauvaise façon à cause du coût en mémoire, mais on fera mieux plus tard

V À faire pour la prochaine fois II

Exercice n°27 Triangles avec des étoiles 2

Écrire une fonction récursive `triangle2(n:int)` dessinant un triangle tel que ci-contre avec une base formée de n étoiles et une hauteur de n étoiles. On interdit l'utilisation de boucle for ou while et on rappelle que `print("*"*n)` affiche n étoiles et retourne à la ligne.

```
1 *
2 **
3 ***
4 ****
5 *****
```

Exercice n°28 Écrire une fonction `permutation(L:list)->list` qui prend en argument une liste L et renvoie la liste de toutes les permutations possibles des éléments de L. Par exemple : `permutation([1,2,3])` doit renvoyer : $[[1,2,3],[2,1,3],[2,3,1],[1,3,2],[3,1,2],[3,2,1]]$ (pas forcément dans cet ordre). Il est possible d'utiliser une fonction récursive avec une boucle for dedans.

Exercice n°29 Parcours d'une arborescence de profondeur inconnue.

On veut réaliser un programme qui affiche le nom de tous les sous-dossier d'un dossier informatique. On utilisera le module `os` de python.

```
1 import os
2 liste = os.listdir() # renvoie la liste des noms des dossiers et
     fichiers dans le dossier actuel
3 os.path.isdir(chaine) # renvoie True si chaine contient le nom d'un
     dossier, False sinon
4 os.chdir(chaine) # se déplace dans le sous-dossier dont le nom est le
     contenu de chaine
5 os.chdir("..") # se déplace dans le dossier "parent", remonte d'un
     dossier
```

À l'aide des fonctions ci-dessus, réaliser un programme qui affiche le nom de tous les sous-dossiers.

Améliorer le programme en rajoutant un entier n qui limite la « profondeur » de sous-dossiers que l'on va explorer. Par exemple si $n = 2$, on ira voir les "sous-dossier" et les "sous-sous-dossiers", mais pas les "sous-sous-sous-dossiers".