Chapitre 1 - Récursivité

Objectifs :

- ▷ Écrire un programme récursif
- ▷ Analyser le fonctionnement d'un programme récursif.
- ▷ Connaître des exemples relevant de domaines variés utilisant la récursivité

1 Introduction

La **récursivité**, à la fois en informatique et dans la vie courante, trouve son utilité dans la résolution de problèmes de façon élégante de problèmes complexes en les décomposant en cas de base et en sous-problèmes gérables.

La récursivité trouve son application dans une variété de domaines informatiques, tels que la programmation, la conception d'algorithmes, la modélisation de données et même dans la création de structures récursives comme les arbres et les listes chaînées.

2 Définition

A retenir !

La récursivité est une méthode de résolution de problèmes qui consiste à décomposer le problème en sous-problèmes identiques de plus en plus petits jusqu'à obtenir un problème suffisamment petit pour qu'il puisse être résolu de manière triviale.

3 Exemples

Voici quelques exemples de problèmes que l'on résout facilement grâce à la récursivité :

- ▷ Calcul de factorielle
- ▷ Suite de Fibonacci
- ▷ Parcours d'arbres
- ▷ Calcul de puissance
- ▷ Tri de tableaux
- ▷ Recherche dans des structures de données
- ▷ Permutations et combinaisons
- ▷ Chemin le plus court dans un graphe
- ▷ Construction de fractales
- ▷ Évaluation d'expressions mathématiques

4 Application

4.1 Algorithme itératif

Étant donné une liste d'entiers `L = [ 2, 12, 1, 8, 5, 10, 20 ]`, on désire calculer la somme des éléments de cette liste. Comme les listes sont **itérables**, nous pouvons simplement résoudre ce problème avec un algorithme que l'on dit **itératif** comme ci-dessous :

Algorithme 1 : Fonction somme des éléments d'une liste d'entiers

Données : Une liste d'entiers *liste*
Sorties : La somme des éléments de la liste

Fonction somme_liste(*liste*) :

somme ← 0

pour chaque *chaque élément dans liste* **faire**
  somme ← *somme* + *élément*

retourner *somme*

4.2 Algorithme récursif

Supposons maintenant que nous n'ayons pas la possibilité de faire de *boucles*. On peut alors aborder le problème sous un autre angle. La somme des termes de cette liste est :

- `2 +` (la somme des termes de `[12,1,8,5,10,20]`)
- soit : `2 + (12 +` (la somme des termes de `[1,8,5,10,20]`)
- et ainsi de suite...
- jusqu'à : `2 + (12 + (1 + (8 + (5 + (10 + (` la somme des termes de `[20]))))))`
- Et enfin, il est clair que la somme des termes de `[20]` est `20`

Par conséquent, le calcul à faire est : `2 + (12 + (1 + (8 + (5 + (10 + (20)))))) = 58`.

Considérons alors une fonction `sommeliste(liste)` et qui renvoie le résultat de la somme des éléments de la liste.

L'algorithme ci-dessous que l'on dit **récursif** réalise cette seconde approche :

Algorithme 2 : Calcul récursif de la somme d'une liste d'entiers

Données : Une liste d'entiers *liste*
Sorties : La somme des éléments de la liste

Fonction somme_liste(*liste*) :
si *la longueur de la liste est égale à 1* **alors**
  retourner *liste[0]*
sinon
  retourner *liste[0] + somme_liste[1 :]*

A faire 1 : Écrire en Python cette fonction, et la tester avec plusieurs exemples.

A faire 2 : Comparer les vitesses d'exécution des fonctions itérative et récursive de l'exemple précédent.

Pour tester la vitesse d'exécution d'une fonction on utilise le module timeit, comme le montre le code ci-dessous pour une liste de 1000 entiers choisis aléatoirement entre 0 et 100 avec le module random.

```
from timeit import default_timer as timer
from random import randint

# ecrire les deux fonctions ici

L = [randint(0,100) for i in range(1000)]

debut = timer()
print(sommeliste(L))
fin = timer()
print(fin - debut)

debut = timer()
print(somme(L))
fin = timer()
print(fin - debut)
```

Question 1 Que pouvez-vous en conclure (pour cet exemple) ?

5 Comment fonctionne un programme récursif ?

Un programme est **une suite d'instructions**, son exécution peut être représentée par un parcours de chemin ayant une origine et une extrémité.

Lors de **l'appel d'une fonction**, cette suite est interrompue le temps de cette fonction, avant de reprendre à l'endroit où le programme s'est arrêté.

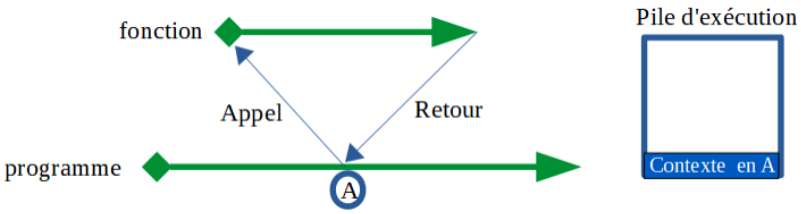

FIGURE 1 – Appel de fonction

Au moment où débute cette bifurcation, le processeur sauvegarde un certain nombre d'informations :

- adresse de retour
- état des variables, ..., etc.

Toutes ces données forment ce qu'on appelle le **contexte du programme**, et elles sont stockées dans ce qu'on appelle la **pile d'exécution**. À la fin de l'exécution de la fonction, le contexte est sorti pour permettre la poursuite de l'exécution du programme.

Lors de l'exécution d'une fonction récursive, chaque appel récursif conduit au moment où il se produit à un empilement du contexte dans la pile d'exécution.

Lorsqu'au bout de n appels se produit la condition d'arrêt, les différents contextes sont progressivement dépilés pour poursuivre l'exécution de la fonction.

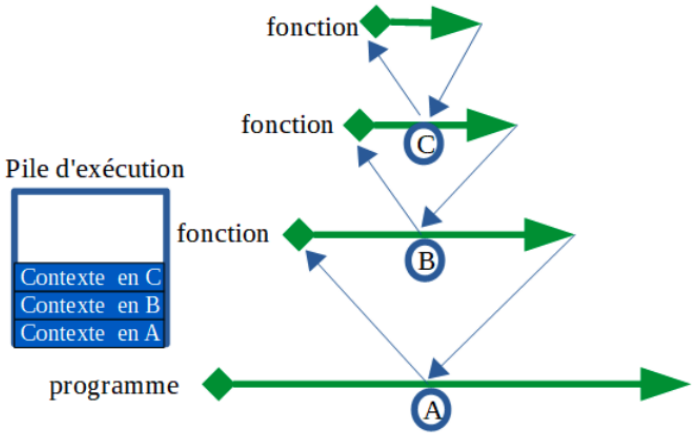

FIGURE 2 – Contextes

Il est important de prendre conscience qu'une fonction récursive s'accompagne d'une **complexité** qui va croître avec le nombre d'appels récursifs (en général linéairement, mais ce n'est pas une règle, cela dépend du contenu du contexte).

6 Les trois règles

Tout comme les robots dans le roman de science-fiction de 1942 d'*Isaac Asimov*, les algorithmes récursifs doivent obéir à trois règles :

A retenir !

- **Un algorithme récursif doit avoir un "état trivial"** , cela permet d'avoir une condition d'arrêt. Dans notre exemple : *si la liste est de longueur 1 alors on renvoie le seul élément de la liste.*
- **Un algorithme récursif doit conduire vers cet "état d'arrêt"**, cela permet de ne pas faire une infinité d'appels récursifs. Dans notre exemple, à chaque appel récursif, la liste est diminuée d'un élément donc nécessairement elle finira par n'en n'avoir plus qu'un.
- **Un algorithme récursif s'appelle lui même**.

7 Itératif ou récursif ?

Il n'existe pas de réponse définitive à la question de savoir si un algorithme récursif est préférable à un algorithme itératif ou le contraire.

Il a été prouvé que ces deux **paradigmes de programmation** sont équivalents.

On peut donc affirmer que tout algorithme itératif possède une version récursive, et réciproquement.

Le choix du langage peut aussi avoir son importance : un langage fonctionnel tel que *Caml* est conçu pour exploiter la récursivité et le programmeur est naturellement amené à choisir la version récursive de l'algorithme qu'il souhaite écrire.

À l'inverse, *Python*, même s'il l'autorise, ne favorise pas l'écriture récursive (limitation basse du nombre d'appels récursifs ; 1000 par défaut).

Enfin, le choix d'écrire une fonction récursive ou itérative peut dépendre du problème à résoudre. Certains problèmes se résolvent particulièrement simplement sous forme récursive.

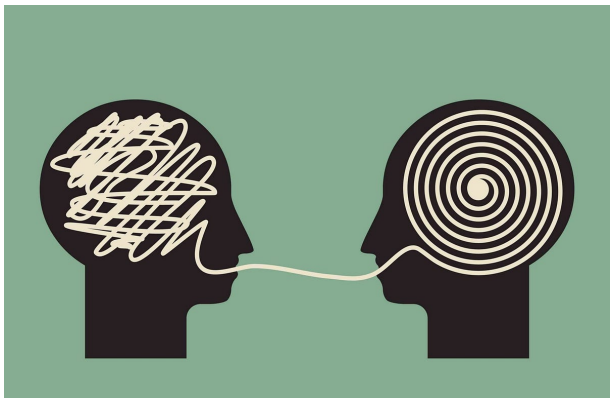

FIGURE 3 – Itératif vs récursif

8 Exercices

Exercice 1 : *Mystère*
On considère la fonction `myst(L)` qui prend une liste `L` en entrée et définie récursivement par :

```
def myst (L):
    if L == []:
        return 0
    else:
        return 1 + myst (L[1:])
```

Que fait cette fonction ?

Exercice 2 : *Maximum*
Écrire les versions *itérative* et *récursive* de fonctions renvoyant le maximum d'une liste d'entiers.

Exercice 3 : *Puissance*
Écrire les versions *itérative* et *récursive* de fonctions calculant x^noù n est un entier et x est un réel.

Exercice 4 : *Factorielle*
Écrire les versions *itérative* et *récursive* de fonctions calculant $n!$ (appelée *factorielle* n) où n est un entier positif sachant que : $n! = n \cdot (n-1) \cdot (n-2) \cdots 2 \cdot 1$

Rappel : Par définition, $0\,! = 1$.

Exercice 5 : *Décimal $\rightarrowtail$ Binaire*
Le but de cet exercice est de convertir un nombre décimal en binaire. Pour cela, on utilisera la succession de divisions euclidiennes par 2.

Exemple :
Considérons le nombre 29 en base 10.
On effectue les divisions successives par 2 jusqu'à obtenir un quotient nul.

29	2
1	14

14	2
0	7

7	2
1	3

3	2
1	1

1	2
1	0

En lisant de droite à gauche les restes obtenus, on obtient l'écriture en base 2 de l'entier.

Ici, on obtient donc $(292)_{10} = (11101)_2$.

Écrire une fonction récursive `dec_vers_bin(n)` qui prend un entier naturel comme argument et renvoie son écriture binaire sous la forme d'une chaîne de caractères.

Exemple :

```
>>> dec\_vers\_bin (38)
'100110'
```

Exercice 6 : *Binaire $\rightarrowtail$ Décimal*
On souhaite cette fois convertir un binaire représenté par une chaîne de caractères en un décimal, codé par un entier.

Si n est un entier naturel et a_0, a_1,...,a_p sont les chiffres de son écriture en base 2, on a :

$$n = 2^p a_p + 2^{p-1} a_{p-1} a_0$$

En remarquant que $n = 2 \times (2^{p-1} a_{p-1} + ... + a_1) + a_0$ si $n \leq 2$, écrire une fonction récursive `bin_vers_dec` qui répond au problème.

Exercice 7 : *Recherche d'un mot dans un texte*
On cherche à déterminer si un mot est présent dans un texte. Pour cela, on utilisera des chaînes de caractères en Python. On considérera que la chaine vide `""` est préfixe de toutes les autres chaînes.

1. Écrire une fonction `prefixe(mot,texte)` qui prend en argument deux chaînes de caractères `mot` et `texte` et détermine si le début de la chaîne `texte` est `mot` : on renverra `True` dans ce cas et `False` sinon.

   Exemple :

```
>> prefixe (" inf ," informatique ")
True
>> prefixe (" nf " ," informatique ")
False
```

2. En utilisant la fonction précédente, écrire une fonction `Est_dans_texte(mot,texte)` qui renvoie `True` si mot apparaît dans texte et `False` sinon.

Exercice 8 : *Flocon de Von Koch*
La courbe de *Von Koch* d'ordre n est défini de la manière suivante :

- si n = 0 alors c'est un segment de longueur 1.
- sinon :
  - ▷ On divise le segment de droite en trois segments de longueurs égales.
  - ▷ On construit un triangle équilatéral ayant pour base le segment médian de la première étape.
  - ▷ On supprime le segment de droite qui était la base du triangle de la deuxième étape.

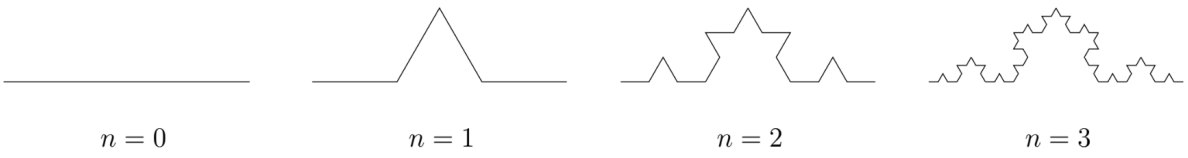

FIGURE 4 – Courbes de Von Koch d'ordre n pour $0 \leq n \leq 3$

L'objectif du problème est de dessiner cette courbe à l'aide de la bibliothèque *Turtle* grâce à une fonction récursive. Pour cela, on remarquera que la courbe de Von Koch d'ordre $n+1$ est constituée de 4 parties qui sont des courbes de Von Koch d'ordre n dont la taille a été divisée par 3.

1. Écrire une fonction récursive `courbeVK(n,l)` qui dessine cette courbe en utilisant *Turtle* avec n l'ordre de la courbe et l la longueur du segment à l'ordre 0.

2. En utilisant la fonction précédente, écrire une fonction `flocon(n,l)` dessinant le flocon de Von Koch d'ordre n.

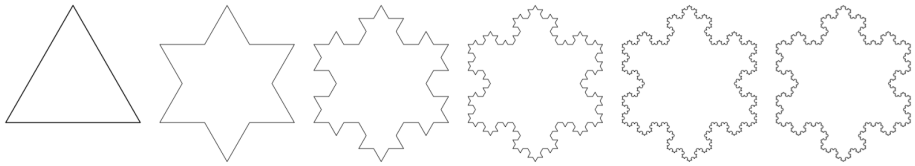

FIGURE 5 – Flocons de l'ordre 0 à l'ordre 5

On rappelle quelques commandes :

`reset()`	Efface tout
penup()	Relève le stylo
`goto(-200,125)`	Place au point de coordonnées `(-200,125)`
`setup(600,600)`	Fenêtre de `600 x 600`
`speed(0)`	Vitesse maximale
`color("blue")`	Couleur bleue
`pendown()`	Stylo en position d'écriture
`right(d)`	Tourner à droite de `d` degrés
`left(d)`	Tourner à gauche de `d` degrés
`fd(p)`	Avancer de `p` pixels