1 Problème d'introduction

1.1 Aspect récursif de l'algorithme d'Euclide

Une des premières choses que l'on apprend lorsqu'on commence à programmer est la notion de boucle : on a souvent besoin dans un algorithme de répéter une partie du programme.

Observons l'algorithme d'Euclide pour trouver le PGCD de deux entiers :

```
def pgcd(a, b):
    """ algorithme d'Euclide en version itérative """
    while b > 0:
        a, b = b, a%b
    return a
```

Prenons soin d'observer la manière de définir l'algorithme d'Euclide :
- si a est divisible par b, le pgcd de a et b vaut b,
- sinon le pgcd de a et b vaut le pgcd de b et a%b

On constate que la manière la plus simple et naturelle d'énoncer cet algorithme est de définir le pgcd par le pgcd lui même !

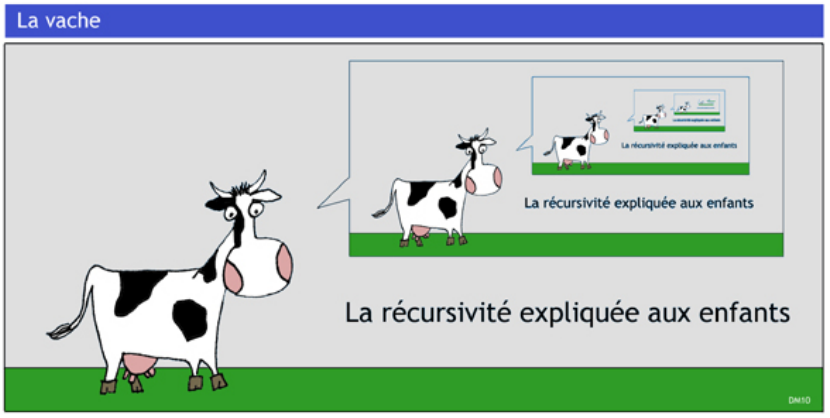
> Lorsque la définition d'un objet fait appel à l'objet lui même, on parle de définition récursive.

Complément : on peut vérifier l'arrêt de cet algorithme avec un variant de boucle, qui est tout simplement la variable **b**.
Rappel : un variant de boucle est une grandeur positive qui décroît à chaque itération.
Cet algorithme s'arrête car a%b est un entier naturel strictement inférieur à b, que l'on affecte à b dans le corps de la boucle : on a donc bien exhibé une grandeur positive décroissante donc à un moment donné ce reste sera nul (lorsque a est divisible par b).

1.2 Exemples de récursivité

On peut rencontrer cette notion de récursivité dans la vie de tous les jours :

- Voici une illustration amusante tirée du blog de Didier Muller présentant cette notion :



— On rencontre aussi énormément la récursivité dans les *fractales*, y compris dans la nature (ex : choux romanesco) :

— Dans l'univers mathématiques :
Vous avez probablement vu en mathématiques la récursivité lorsque vous avez étudié les suites définies par récurrence.
On la retrouve aussi comme un puissant moyen de démonstration avec la démonstration par récurrence.

— Dans l'univers informatique :
La récursivité est bien sûr aussi présente en informatique : les premiers langages de programmation qui ont autorisé l'emploi de la récursivité sont LISP et Algol 60. Dans LISP, elle est tellement fondamentale que ce langage ne possède pas de structures de boucles : tout se fait avec la récursivité !

Tous les langages de programmation haut niveau peuvent utiliser la récursivité, c'est à dire la faculté que possède une fonction à s'appeler elle-même.

2 Premier exemple de programmation récursive

Revenons sur notre exemple de l'algorithme d'Euclide. Une autre manière de l'implémenter est de traduire en Python mot à mot la manière dont on a décrit cet algorithme.
Pour rappel :
— si a est divisible par b, le pgcd de a et b vaut b,
— sinon le pgcd de a et b vaut le pgcd de b et a%b

Cette algorithme peut se coder en quelques lignes traduisant directement cette pensée :

```
def pgcd(a, b):
    """ algorithme d'Euclide en version récursive """
    if a % b == 0:
        return b
    else:
        return pgcd(b, a%b)
```

Remarquez à quel point la syntaxe Python est proche du langage naturel !

Une fonction récursive se caractérise par deux propriétés :
— une condition d'arrêt.
— la fonction contient des appels à elle-même à l'intérieur de celle-ci.

Attention à la condition d'arrêt ! : n'oubliez surtout pas la condition d'arrêt sans quoi votre algorithme tournera à l'infini et votre programme s'arrêtera faute de mémoire disponible sans jamais rendre de réponse !

La programmation récursive s'avère extrêmement pratique lorsque la résolution d'un problème se ramène à celle d'un problème plus petit similaire.

À force de réduire notre problème, on arrive à un problème trivial que l'on sait résoudre : c'est ce qu'on utilise dans notre condition d'arrêt.

Essayez de programmer la fonction puissance, prenant en paramètres deux entiers naturels a et b, renvoyant a^b.
Vous utiliserez pour cela une fonction récursive, et bien sûr pas la fonction intégrée à Python.

On rappelle la manière dont sont définis les coefficients binomiaux :

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

Par ailleurs on a les cas particuliers : $\binom{n}{1} = n$ et $\binom{n}{n} = 1$.

Cette formule est notamment utilisée pour la construction du triangle de Pascal.
Vous programmerez la fonction binom prenant en paramètres deux entiers naturels n et k, renvoyant $\binom{n}{k}$.
Cette fonction sera récursive.

3 Comment ça marche ? Pile d'exécution

La programmation récursive semble très simple dans le principe : il s'agit d'un appel de fonction à l'intérieur d'un appel de fonction donc rien de différent après tout de ce que l'on connaît déjà !

Il est temps de regarder de plus près ce qui se passe quand on fait un appel de fonction.

3.1 Appel d'une fonction dans une fonction

Imaginons le programme suivant.

```
def fonction_a():
    print("\nDébut de la fonction A")
    for i in range(5):
        print(f"A{i}", end=" ")
    print("\nFin de la fonction A")

def fonction_b():
    print("Début de la fonction B")
    for i in range(5):
        print(f"B{i}", end=" ")
        if i==2:
            fonction_a()
    print("\nFin de la fonction B")
```

L'appel de la fonction A affiche quelque chose du genre :

```
>>> fonction_a()
Début de la fonction A
A0 A1 A2 A3 A4
Fin de la fonction A
```

Rien de bien passionnant. Observons maintenant l'appel de la fonction B :

```
>>> fonction_b()
Début de la fonction B
B0 B1 B2
Début de la fonction A
A0 A1 A2 A3 A4
Fin de la fonction A
B3 B4
Fin de la fonction B
```

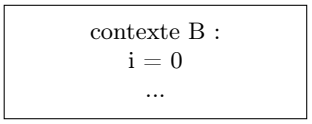
Le résultat n'a rien d'étonnant : la fonction A est invoquée au milieu de l'exécution de la fonction B. Cela peut vous sembler naturel mais pour la machine, ce mécanisme est loin d'être simple : B interrompt son exécution pour exécuter A. Lorsque A se termine, on remarque que B reprend son exécution là ou il l'avait interrompu, *y compris la variable i qui n'a pas été perturbée par l'exécution de A* et qui continue à être incrémentée comme si rien ne s'était passé.

Ce comportement est logique et naturel mais nécessite la mise en œuvre d'un mécanisme complexe dans la machine : la *pile d'exécution*.

> Lorsqu'une fonction est exécutée, les informations nécessaires à son fonctionnement (ce qu'on appelle son **contexte d'exécution**) sont sauvegardées dans une **pile d'exécution**.

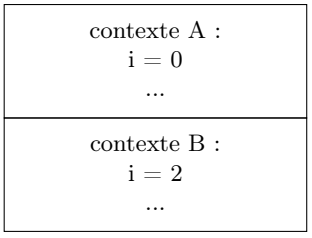
Celle pile va donc en particulier contenir la variable locale i.
L'exécution de la fonction B va se dérouler en 3 phases :

1. Début de la fonction B, on empile un nouveau contexte pour la fonction B.



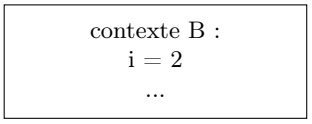
pile d'exécution

2. La fonction A est invoquée. Un nouveau contexte est empilé sur la pile.



pile d'exécution

3. La fonction A se termine. Son contexte est alors détruit et B retrouve son contexte et peut ainsi reprendre la ou il en était.



pile d'exécution

3.2 Et la récursivité ?

Un appel récursif ne se passe pas différemment : à chaque nouvel appel, un nouveau contexte d'exécution est créé afin que lorsque l'appel récursif se termine, la fonction appelante retrouve son contexte propre avec ses variables dans l'état où elles étaient avant appel.
Chaque appel récursif consomme de la mémoire dans la pile d'exécution !

Une boucle de plusieurs centaines de millions d'itérations est sans impact sur la mémoire de la machine alors que quelques centaines d'appels récursifs vont vite se révéler problématiques !

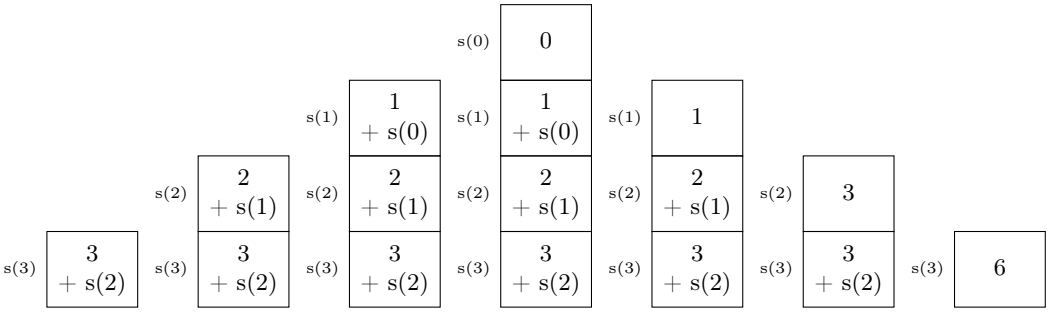
> Pour protéger l'intégrité du système, Python va surveiller le nombre d'appels récursifs afin que la pile ne déborde pas. Cela constitue une limite à ce que la récursivité permet de faire et le développeur doit clairement en être conscient.

3.3 Exemple de limite de la récursivité

Considérons cet exemple récursif simple calculant la somme des n premiers entiers naturels :

```
def somme(n):
    if n == 0:
        return 0
    return n + somme(n-1)
```

Voici la pile d'exécution pour un appel à `somme(3)` :



pile d'exécution

L'appel `somme(1000)` fonctionne encore parfaitement.

En revanche, `somme(10000)` pose problème :

```
>>> somme(10000)

---------------------------------------------------------------------------
RecursionError                            Traceback (most recent call last)
<ipython-input-46-fe6f8d324d61> in <module>
----> 1 somme(10000)

<ipython-input-42-27e72bf34ee7> in somme(n)
      2     if n==0:
      3         return 0
----> 4     return n+somme(n-1)

... last 1 frames repeated, from the frame below ...

<ipython-input-42-27e72bf34ee7> in somme(n)
      2     if n==0:
      3         return 0
----> 4     return n+somme(n-1)

RecursionError: maximum recursion depth exceeded in comparison
```

Sur les implémentation microPython de Python sur les calculatrices, la contrainte mémoire est encore plus grande.

 Sur votre calculatrice, programmez la fonction récursive `somme` et déterminez la limite à partir de laquelle une erreur va intervenir.