Chap 2 : La récursivité

Lorsqu'on veut programmer un calcul qui se répète il y a toujours deux approches possibles :

1. Une approche **itérative** : elle utilise des boucles **for** ou **while**.
2. Une approche **récursive** que nous allons étudier dans ce chapitre.

Nous commençons par une révision de notions importantes de MPSI sur la programmation itérative : la preuve de la terminaison des boucles et les invariantes de boucles. On étudiera ensuite la manière alternative de programmer qu'est la récursivité.

Table des matières

1 La programmation itérative. Rappels de MPSI

Nous revenons sur deux points importants du cours de MPSI concernant les boucles (aussi bien **while** que **for**) :

- La condition de terminaison
- L'invariant de boucle (quand il existe)

1.1 Condition de terminaison

Pour prouver qu'un algorithme contenant une boucle est correct, il faut montrer que la boucle va nécessairement s'arrêter. Cela revient à montrer qu'une condition de terminaison de boucle qui était fausse au départ, change de nature et devient vraie : la boucle s'arrête immédiatement.

- Dans le cas d'une boucle **for**, c'est assez évident :

```
for i in range(1, 10) :
    y = 2 * i
    print("y = ", y)
```

L'itérateur i va aller de 1 à 9 et la boucle va s'arrêter...

- Pour une boucle du type **while** *Exp*, il faut vérifier que *Exp* peut passer de **True** à **False**.

Prenons l'algorithme de multiplication de de deux entiers naturels $a > 0$ et $b > 0$. Supposons que $a \leqslant b$: faire $a \times b$ revient à faire $b+b+...+b$ "a fois" (si on avait eu $b \leqslant a$, on aurait fait : $a+a+...+a$ "b fois" car l'essentiel est de faire le moins d'opérations possibles).

Précondition : 0 < a ⩽ b

```
def multiplie(a, b) :
    n = a
    prod = 0
    while n > 0 :
        prod = prod + b
        n = n - 1
    return prod
```

Question : démontrer que la boucle se termine.

1.2 Invariant de boucle

1.2.1 Définition

Dans cette section, nous introduisons un entier i qui décrit le nombre de tours de boucles effectués : i est le **compteur de boucle**. Nous supposons les conditions suivantes :

- Avant le premier passage dans la boucle : $i = 0$.
- À la *fin du premier passage dans la boucle* (c'est à dire après la dernière instruction de la boucle) : $i = 1$. De façon plus générale, à la fin du $k^{\text{ème}}$ passage : $i = k$.
- La boucle étant censée se terminer, après le dernier passage dans la boucle : $i = N$. À l'issue du processus, les instructions de la boucle auront donc été exécutées N fois.

Définition (Invariant de boucle)

On appelle **invariant de boucle** une propriété $P(i)$ qui est vraie avant l'entrée dans la boucle et qui reste vraie après chaque passage dans la boucle. Autrement dit :

1. $P(0)$ est vraie ;
2. $\forall\, i \in [\![1, N]\!]$, $P(i)$ est vraie

Un invariant de boucle est une notion d'informatique théorique qui sert très souvent à démontrer qu'une boucle effectue son travail correctement. Comme la véracité de $P(i)$ est conservée, il en résulte que, une fois que la boucle est terminée, $P(N)$ est vraie. En général, $P(N)$ représente le travail accompli par la boucle (voir exemples plus loin).

1.2.2 Premier exemple : boucle factorielle

Le premier exemple est celui d'une boucle qui calcule la factorielle d'une nombre entier $n \geqslant 0$. Cette boucle est dans une fonction FACTORIELLE(n) qui retourne $n!$

```
def FACTORIELLE(n) :
    fact = 1
    i = 0      # Compteur de boucle
    while i < n :
        i += 1
        fact = fact * i
    return fact
```

<u>Conventions de notations</u> :

Comme la valeur de **fact** change après chaque passage dans la boucle **while**, on note **fact**(i) sa valeur après les $i^{ème}$ passage et **fact**(0) sa valeur initiale, avant l'entrée dans la boucle.

Point pratique

Si une variable **a** change lors de chaque itération d'une boucle **for** ou **while** alors, dans un raisonnement d'informatique théorique, il est judicieux de la nommer **a**(i) après le $i^{ème}$ passage dans la boucle.

Considérons la propriété suivante :

$$P(i) = \{\, \mathbf{fact}(i) == i\,!\,\}$$

Question : montrer que $P(i)$ est un invariant de boucle. Que peut-on en déduire ?

1.2.3 Algorithme de la division euclidienne

L'algorithme de la division euclidienne prend comme entrée deux entiers naturels $a \geqslant 0$ et $b > 0$. Il calcule le quotient q et le reste r de la division euclidienne de a par b :

$$a = q \times b + r \quad \text{avec} \quad 0 \leqslant r < b$$

L'algorithme est écrit dans une fonction Python :

Précondition : a ⩾ 0 et b > 0

```
def divEuclide(a, b) :
    r = a
    q = 0
    while r >= b :
        r = r - b
        q += 1
    return q , r
```

Ici de même, comme les valeurs des variables **r** et **q** changent à chaque passage dans la boucle, nous seront amenés à définir un compteur de boucle i ainsi que les valeurs $\mathbf{r}(i)$ et $\mathbf{q}(i)$ après le $i^{ème}$ passage dans la boucle. Les valeurs initiales seront : $\mathbf{r}(0)$ et $\mathbf{q}(0)$.

1. Montrer que la boucle se termine
2. Soit la propriété $P(i) = \{a == q(i) \times b + r(i)\}$. Montrer que c'est un invariant de boucle. Que peut-on en déduire ?

1.2.4 Algorithme d'Euclide pour le calcul d'un pgcd

L'algorithme d'Euclide calcule le pgcd de deux entiers naturels $a \geqslant 0$ et $b > 0$. Il est écrit ci-dessous comme une fonction Python qui retourne le pgcd cherché.

Précondition : a ⩾ 0 et b > 0

```
def pgcd(a, b) :
    r1 = a
    r2 = b
    while r2 > 0 :
        temp = r2
        r2 = r1 % r2
        r1 = temp
    return r1
```

1. **Question préliminaire**. Étant donnés deux entiers naturels $a \geqslant 0$ et $b > 0$, on pose $a = q \times b + r$ avec $0 \leqslant r < b$. Montrer que pgcd(a, b) = pgcd(b, r).
2. Montrer que la boucle se termine.
3. Soient i le compteur de boucle et $P(i)$ la propriété :

$$P(i) = \{ \, \text{pgcd}(\, r1(i), r2(i) \,) == \text{pgcd}(\, a, b \,) \, \}$$

   Montrer que $P(i)$ est un invariant de boucle.
4. En déduire que cette fonction retourne bien le pgcd de a et b.

1.2.5 Conclusion

Dans l'étude des boucle **for** et **while** :

Points importants

- Prouver que la boucle finit par s'arrêter est une première étape.
- Prouver que la boucle retourne le résultat escompté est une deuxième étape. Cette deuxième étape peut souvent se faire grâce à un **invariant de boucle**. Cet invariant de boucle est une propriété qui est vraie avant l'entrée dans la boucle et qui reste vraie après chaque passage dans la boucle.

Remarque :

Lorsque $P(i)$ est donné, il est facile de prouver que c'est un invariant de boucle à l'aide d'un raisonnement par récurrence. En revanche il est parfois assez dur de trouver une propriété $P(i)$ qui soit inv. de boucle dans le cas où elle ne serait pas donnée.

2 La récursivité

2.1 Définition

Définition (fonction récursive)

Une fonction **f** est dite **récursive** si son exécution peut provoquer un ou plusieurs appels de **f** elle-même. Un langage de programmation est dit *récursif* s'il permet d'écrire des fonctions récursives. La plupart des langages actuels sont récursifs (comme Python, C, C++, etc...).

Reprenons le problème du calcul de la factorielle d'un entier naturel $n \geqslant 0$ à l'aide d'une **approche récursive**, ce qui signifie qu'on écrit une **fonction** FACTORIELLE *qui peut s'appeler elle-même.*

fonction FACTORIELLE(n) :
Précondition : $n \geqslant 0$

```
1   si n == 0 ou n == 1 faire :
2     retourner 1
3   sinon :
4     x = n * FACTORIELLE(n - 1)
5     retourner x
```

Les points à contrôler **impérativement** sont :

1. Il faut vérifier que la suite des appels récursifs de FACTORIELLE va s'arrêter : critère de **terminaison**. Dans le cas contraire les appels s'enchaînent sans arrêt et le programme ne finit pas[1].

   $\Longrightarrow$ il faut placer un **cas terminal** dans la fonction : c'est une *condition qui fait cesser* les appels récursifs. Il faut alors être certain que ce cas terminal peut être atteint pour n'importe quelle valeur de départ des *paramètres.*

   La première condition ligne 1, qui retourne directement 1 si $n == 0$ ou $n == 1$ est le cas terminal de notre fonction FACTORIELLE et il est nécessairement atteint.

2. Il faut vérifier que l'algorithme va bien produire le résultat demandé, c'est à dire qu'il est **correct** ;

L'approche récursive se prête bien au calcul des nombres définis par une récurrence, comme dans le cas de certaines suites par exemple.

1. Comme chaque appel de FACTORIELLE empile $n-1$, $n-2$, etc... sur la pile gérée par l'unité centrale, il va arriver un moment où celle-ci **déborde**. Cela arrête automatiquement le programme et une erreur " débordement de pile " (stack overflow pour les anglo-saxons) est générée.

2.2 Avantage et inconvénient des fonctions récursives

- Avantages :
  — Écriture de la fonction proche du langage des mathématiques et de la récurrence. Facilité de prouver que l'algorithme est **correct** grâce à la démonstration par récurrence.
  — Programmes élégants.

- Inconvénients :
  — Chaque appel provoque l'**empilage** des paramètres de la fonction. S'il y a beaucoup de niveaux de récursion, cela peut provoquer facilement le **débordement** de la pile de la machine.
  — Les différents **empilages** (y compris pour les valeurs de retour) et **dépilages** ralentissent le programme.
  — Les appels récursifs qui s'enchainent peuvent devenir très compliqués et produire plusieurs fois le même calcul, ce qui à nouveau ralentit fortement le programme (voir exercice 3).

2.3 Exercices

1. Considérons la suite $u_n = 3\,u_{n-1} - 1$ pour $n \geqslant 1$ et $u_0 = 2$. Écrire une fonction `u_itera(n)` qui calcule u_n en utilisant une approche itérative, puis une autre fonction `u_rec(n)` faisant la même chose mais avec approche récursive.

2. Considérons la somme $S(n) = \displaystyle\sum_{k=1}^{n} \frac{1}{k(k+1)}$. Écrire une fonction `somme(n)` qui renvoie $S(n)$ en utilisant une approche itérative puis une approche récursive.

3. Considérons la suite de Fibonacci, générée par la récurrence suivante :
$$u_n = \begin{cases} u_{n-1} + u_{n-2} & \text{si } n > 1 \\ 1 & \text{si } n = 1 \\ 0 & \text{si } n = 0 \end{cases}$$
   (a) Écrire une fonction `fibo_itera(n)` qui calcule cette suite en utilisant une approche itérative,
   (b) Réécrire la la fonction `fibo_rec(n)` avec une approche récursive. La lancer pour calculer `fibo_rec(6)` puis `fibo_rec(200)`. Que constate-t-on dans ce dernier cas ?
   (c) Pour y voir plus clair, placer un compteur mouchard `cmpt` dans le programme, défini comme une variable globale et l'augmenter de 1 à chaque appel de `fibo_rec`. Que constate-t-on ?
   (d) Tracer l'arbre des appels récursifs de `fibo_rec(6)`. Cet arbre permet de visualiser l'ensemble des appels récursifs ainsi que l'ordre de ces appels.

4. * Réécrire la fonction `pgcd(a, b)` en version récursive. On utilisera la propriété : $\text{pgcd}(a, b) = \text{pgcd}(b, r)$ où r est le reste de la division euclidienne de a par b : $r = a \,\%\, b$.

5. * Soit x un flottant et n un entier naturel. Il est possible d'écrire une fonction récursive `expo(x, n)` qui renvoie x^n à l'aide de la remarque suivante :
$$x^n = \begin{cases} \left(x^{\frac{n}{2}}\right)^2 & \text{si } n \text{ est pair} \\ \\ x\,\left(x^{\frac{n-1}{2}}\right)^2 & \text{si } n \text{ est impair} \end{cases}$$
   Écrire cette fonction en Python.

6. **** Les tours de Hanoi**

   On dispose de trois piquets avec socle, numérotés 0, 1 et 2, et de n disques troués qui sont deux à deux de tailles différentes. Au départ, les n disques sont empilés par ordre croissant de taille sur le piquet n°0 (Figure 1).

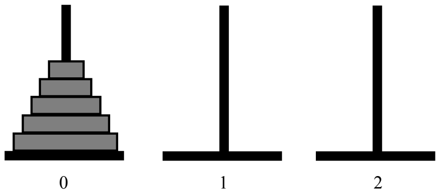

Figure 1

Le but du jeu est de déplacer ces n disques du piquet n°0 sur le piquet n°2, en respectant les règles suivantes :

— On ne déplace qu'un seul disque à la fois et le disque déplacé doit être sur l'un des deux autres piquets ; c'est ce que l'on appelle un déplacement.
— Un disque ne doit jamais être placé au-dessus d'un disque plus petit que lui.

Réalisation : les trois piquets sont représentés par une liste de trois listes : piquets = [[...], [...], [...]] où chaque sous-liste représente un des 3 piquets. Les disques sont des entiers dont la valeur représente la taille du disque. Par exemple, l'état initial de la Figure 1 où tous 5 disques sont sur le piquet 0 est représenté par : `piquets == [ [5, 4, 3, 2, 1], [ ], [ ] ]`.

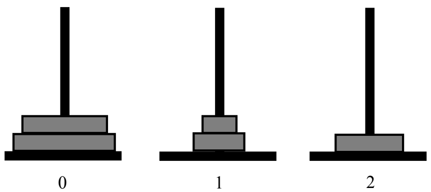

Figure 2

L'état intermédiaire décrit par la Figure 2 est représenté par `piquets == [ [5, 4], [2, 1], [3] ]`.

(a) On se donne un entier n (nombre de disques). Générer la liste `L = [ n, n-1, ..., 1]`, puis la liste `piquets` dans l'état initial, c'est à dire `[ L, [ ], [ ] ]`.

(b) Par la suite, les trois piquets seront numérotés p, q et r avec $(p, q, r) \in \{0, 1, 2\}^3$ mais tous différents deux à deux. Écrire une fonction `deplace(p, q)` qui déplace le dernier disque du piquet n°p vers le piquet n°q, en l'insérant au sommet de la pile de disques.

(c) On souhaite écrire une fonction `Hanoi(n, p, q, r)` qui déplace toute une pile de n disques initialement située sur le piquet n°p vers le piquet n°r, en utilisant le piquet n°q. Par exemple, avec l'état initial de la Figure 1, l'appel de `Hanoi(5, 0, 2, 1)` produit le résultat ci-dessous :

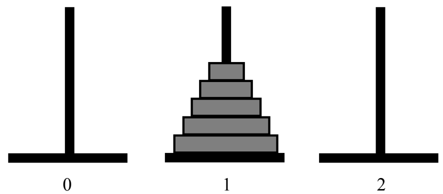

Figure 3

Écrire cette fonction de façon totalement récursive, la récursivité portant sur le nombre n de disques de la pile.