Récursivité

1 Principe

Une fonction **récursive** est une fonction qui s'appelle elle-même : cela peut sembler étrange mais c'est tout à fait autorisé (même s'il y a des risques de boucle infinie !).

Exemple 1. La fonction **fct** donnée par l'algorithme ci-dessous est récursive ; la reconnaissez-vous ?

```
DefinitionFonction fct(Entier n):
  si n == 0:
    renvoyer 1
  sinon :
    renvoyer n * fct(n-1)
```

Remarque 1. Vous pouvez l'écrire en langage Python et faire le calcul de **fct(2)**, **fct(3)**, **fct(4)**, **fct(5)**, ... Vous avez évidemment reconnu la fonction factorielle.

Exemple 2. Voyons son utilisation en cours d'appel avec `fct(3)` :

Comme 3 ne vaut pas 0, on continue.
On doit calculer 3 * fct(2). On doit donc calculer fct(2)
    2 ne vaut pas 0, donc on continue
    On doit calculer 2*fct(1). On doit donc calculer fct(1)
        1 ne vaut pas 0, donc on continue
        On doit calculer 1*fct(0). On doit donc calculer fct(0)
            0 vaut 0, donc on renvoie fct(0) = 1
        On renvoie fct(1) = 1 * 1 = 1
    On renvoie fct(2) = 2 * 1 = 2
On renvoie fct(3) = 3 * 2 = 6

Remarque 2. Les risques indiqués plus haut sont réels : en effet, si on avait commis l'erreur d'écrire **n*fct(n)** au lieu de **n*fct(n-1)** on aurait une belle boucle infinie (et alors, merci **CTRL+I** sous Pyzo !) mais même là, on peut avoir un souci : essayer de calculer fct(5.5) ou fct(-1)...
On pourrait améliorer la fonction en ajoutant un test sur le type de **n** et sur son signe.

Exemple 3. Ecrivons maintenant de manière récursive la fonction **fibo** définie sur $\mathbb{N}$ par :

$$\textbf{fibo(0)} = \textbf{fibo(1)} = \textbf{1} \text{ et si } \textbf{n} > \textbf{1}, \textbf{fibo(n)} = \textbf{fibo(n-1)} + \textbf{fibo(n-2)}.$$

On peut assez facilement calculer **fibo(2)=2**, **fibo(3)=3**, **fibo(4)=5**.

Décrivons alors tous les appels récursifs de **fibo** effectués pour calculer **fibo(4)** :

fibo(4)
| fibo(3)
| | fibo(2)
| | | fibo(1)→ 1
| | | +
| | | fibo(0)→ 1
| | → 2
| | +
| | fibo(1)→ 1
| → 3
| +
| fibo(2)
| | fibo(1)→ 1
| | +
| | fibo(0)→ 1
| → 2

→ 5

Exemple 4. Ecrivons une fonction **somme** récursive recevant comme argument une liste et donnant la somme des termes de la liste.
Un bonne idée est de considérer la liste comme une pile...

```
DefinitionFonction somme(Liste L):
  si taille(L)==0:
    renvoyer 0
  sinon :
    L.pop() + somme(L)
```

Exemple 5. Ecrivons une fonction **pgcd** récursive recevant comme arguments deux entiers **a** et **b**, et donnant le plus grand diviseur commun de **a** et **b**.
On pourra remarquer que, si $\mathbf{b = 0}$, **pgcd(a,b)** $= \mathbf{a}$ et, si $\mathbf{b > 0}$, **pgcd(a,b)** est le pgcd de **b** et du reste de la division entière de **a** par **b**.
La preuve de l'algorithme a été faite en première année et résulte du fait que la suite des restes est une suite d'entiers strictement décroissante et donc, en un nombre fini d'itérations ou d'appels récursifs, on a un reste nul.

On rappelle que l'agorithme itératif était :

```
DefinitionFonction pgcdI(Entier a, Entier b):
  r0,r1 = a,b
  si a<0:
    r0 = -a
  si b<0:
    r1 = -b
  tant que  r1!=0:
    r0,r1 = r1,r0%r1
  retourner  r0
```

L' algorithme récursif est alors :

```
DefinitionFonction pgcd(Entier
    a, Entier b):
  si a<0 or b<0:
    retourner  pgcd(abs(a),abs
       (b))
  sinon si b ==0:
    retourner  a
  sinon:
    retourner  pgcd(b,a%b)
```

2 Avantages et inconvénients de la récursivité

L'avantage de la récursivité est avant tout un avantage pour le développeur ; en effet, certaines fonctions s'écrivent plus naturellement sous cette forme qu'en itératif.
En revanche, cela n'est pas sans problème : - le système est obligé de créer une pile des appels aux fonctions et la récursivité augmente beaucoup la taille de cette pile. On peut se retrouver face à des erreurs du type **stack overflow** ; - les appels aux fonctions ont un assez grand coÃ»t en temps de calcul ;
Passer d'un algorithme récursif à un algorithme itératif est toujours possible mais ce n'est pas toujours

si simple : on appelle cette opération la **dérécursification**. L'idée est en général de créer une pile pour gérer soi-même les appels aux calculs nécessaires... On y gagne en général du temps de calcul !

Des autres moyens d'améliorer la complexité d'un algorithme récursif sont d'essayer de se ramener à une récursivité terminale ou d'utiliser la technique de mémoïsation.

mémoïsation . Le principe est d'utiliser un tableau de valeurs que l'on remplit au fur et à mesure des différents appels récursifs. A chaque appel on regarde si la valeur à calculer est dans le tableau sinon on effectue l'appel récursif. Il faut juste déterminer à l'avance la taille du tableau... ce qui sera la limite d'utilisation de la fonction

Exemple 6.

```
tab=[None for n in range(1000)]

def fibomem(n):
  if tab[n] == None:
    if n in[1,2]:
      tab[n] = 1
    else:
      tab[n] = fibomem(n-1) +
         fibomem(n-2)
  return(tab[n])
```

Remarquons que la complexité de `fibomem(n)` est $O(n)$ (au lieu de $O(\varphi^n)$

récursivité terminale . Une fonction f est dite récursive terminale si aucun traitement n'est effectué à la remontée d'un appel récursif donc si l'appel récursif est de la forme `return(f(...))`

```
def fiboterm(n):
  def aux(k,u,v):
    if k<3:
      return(v)
    else:
      return(aux(k-1,v,u+v))
  return(aux(n,1,1))
```

Remarquons que la complexité de `fiboterm(n)` est $O(n)$ (au lieu de $O(\varphi^n)$

3 Exercices

Exercice 1. Coefficient Binomial

On rappelle la relation, pour tout couple d'entiers (n,p) vérifiant $0 < p \leqslant n$: $\binom{n}{p} = \frac{n}{p}\binom{n-1}{p-1}$

1. Ecrire une fonction récursive qui calcule $\binom{n}{p}$ pour $(n,p) \in \mathbb{Z}^2$, où on convient que la fonction retourne 0 pour $p < 0$, $n < 0$ ou $p > n$.
2. Prouver la terminaison, la correction de votre fonction. Quelle est sa complexité ? (on comptera le nombre d'appels récursifs de la fonction)

Exercice 2. : quelques fonctions récursives élémentaires

1. Ecrire une fonction récursive `produit` qui, étant donné $n \in_N n^*$, renvoie $\prod_{k=1}^{n} \frac{3k+1}{3k}$
2. Ecrire une fonction récursive `mini` qui renvoie le minimum d'une liste de flottants.

3. Ecrire une fonction récursive `decomposition` calculant la décomposition d'un entier n en base 2 (sous la forme d'une liste de 0 et de 1). Par exemple, on doit avoir ce type de réponse :

```
>>> decomposition(13)
[1, 1, 0, 1]
```

Indication : on commencera par donner une relation entre `decompostion(n)` et `decomposition(n//2)`

4. Ecrire une fonction récursive `recomposition` calculant l'entier n à partir de la liste (de 0 et de 1) formée par son écriture en base 2.
5. Ecrire une fonction récursie `palindrome` qui prend en entrée une chaine de caractères `s` et qui renvoie `True` ou `False` selon que la chaine `s` soit (ou pas) un palindrome (mot identique si on le lit à l'envers).
   Par exemple, MATHS n'est pas un palindrome, mais LAVAL et GIRAFARIG sont deux palindromes

Exercice 3. Exponentiation rapide

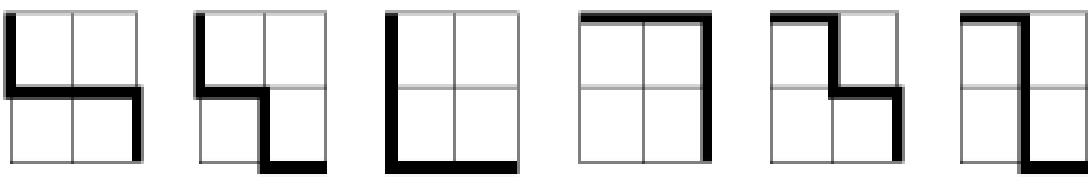
1. Ecrire une fonction récursive qui calcule `puissance` qui prend comme argument un flottant x et un entier n et qui retourne x^n (sans utiliser la fonctionnalité proposée par $**$.
2. Ecrire une fonction `exponentiationrapide` prenant les mêmes arguments et retournant le même résultat que la fonction précédente mais reposant sur la remarque suivante :
   - ✏ si n pair, $x^n = \left(x^2\right)^{n/2}$. On calcule donc $y^{n/2}$ avec $y = x^2$.
   - ✏ si n impair, $x^n = x\left(x^2\right)^{(n-1)/2}$. On calcule $y^{(n-1)/2}$ avec $y = x^2$ puis on multiplie par x
3. Calculer la complexité de ces deux algorithmes

Exercice 4. Suite de Padovan

La suite de Padovan est définie par la relation de récurrence : $\begin{cases} u_0 = u_1 = u_2 = 1 \\ \forall n \geqslant 3,\ u_n = u_{n-2} + u_{n-3} \end{cases}$

1. Programmer de façon récursive (naïve) la suite de Padovan. On écrira donc une fonction `u(n)` qui renvoie u_n.
2. Modifier votre programme pour qu'il affiche de plus le nombre d'appels récursifs effectués.
   Cette nouvelle fonction renverra un tuple dont le premier élément est u_n et dont le second élément représente le nombre d'appels récursifs effectués. Par exemple si $n = 7$, cette fonction renverra $(5, 8)$
3. Donner une version itérative du calcul du $n-$ième terme de la suite de Padovan.
4. On note a_n le nombre d'appels récursifs pour calculer u_n par l'algorithme de la question 1.
   Par exemple, $a_0 = a_1 = a_2 = 0$ car il n'y a pas d'appels récursifs pour calculer u_0, u_1 et u_2. Ecrire une relation de récurrence vérifiée par les a_n.
   On pourrait montrer que $a_n \sim C\psi^n$ avec C constante strictement positive et ψ l'unique racine réelle de $X^3 - X - 1$
5. A l'aide d'une fonction auxiliaire récursive terminale ou de la technique de mémoïsation, proposée un algorithme récursif plus efficace que l'algorithme de la question 1

Exercice 5. Calcul du nombre de chemins avec mémoïsation En commençant du coin en haut à gauche sur une grille 2×2 et en ne se déplaçant que vers la droite ou vers le bas, il y a 6 chemins pour aller du coin en haut à gauche vers le coin en bas à droite :



On cherche le nombre de chemins du coin en haut à gauche vers le coin en bas à droite dans une grille de taille 20×20.
On note $\varphi(i,j)$ le nombre de chemins permettant d'aller du coin en haut à gauche noté $(0,0)$ vers celui de coordonnées (i,j).

1. Justifier que $\varphi(i,j) = \begin{cases} 1 & \text{si } i = 0 \text{ ou } j = 0 \\ \varphi(i-1,j) + \varphi(i,j-1) & \text{sinon} \end{cases}$
2. Programmer la fonction `phi` suivant ce principe.
   Vérifier que l'on trouve $\varphi(15,10) = 3268760$ mais que le calcul ne se termine pas en un temps raisonnable pour $\varphi(20,20)$.
3. Pour améliorer ce procédé de calcul, on peut **mémoïzer** : on travaille avec un tableau de valeurs, où l'on place 0 (ou None) pour toutes les valeurs non calculées. A chaque appel à φ, on commence par chercher dans le tableau des valeurs calculées :
   - ✏ s'il y a une valeur strictement positive (ou autre chose que None), on la renvoie
   - ✏ sinon, on calcule récursivement $\varphi(i-1,j) + \varphi(i,j-1)$

   On va mettre en œuvre ce procédé pour calculer $\varphi(20,20)$.
   (a) Commencer par créer un tableau `T` de taille 21×21 tel que `T[i][j] = 1` si i ou j est nul, et `T[i][j] = 0` sinon.
   (b) Ecrire une fonction récursive qui calcule $\varphi(20,20)$ à l'aide du tableau `T`