Institut National Universitaire Champollion

Chapitre 6

Récursivité

Plan du chapitre

1. introduction ……..……..……..…….3

2. Exemple de fonction récursive ……..……7

3. Analyse de la récursivité ……..13

4. Une mauvaise utilisation de la récursivité ……..……..19

Introduction

Il est courant que la définition d'une fonction f appelle une autre fonction g :

Exemple 1

```
def dimensions(A) :
        n = len(A)
        p = len(A[0])
        return n,p
```

Ici, la fonction dimensions fait appel à la fonction len qui détermine la longueur d'une liste.
C'est plus surprenant, mais il est également possible qu'une fonction f appelle f (elle-même donc) lors de
son exécution.

Exemple 2

```
def test(n) :
        if n > 0 :
        print(n)
        return test(n-1)
```

Dans la définition de la fonction test, on s'aperçoit que test apparaît en 4ème ligne.

Une telle fonction s'appelant elle-même est dite récursive.

Important

Une conséquence importante de cette définition est la suivante :

Une fonction qui ne s'appelle pas elle-même n'est donc pas récursive, elle est dite itérative.

En particulier, si on vous demande d'écrire une fonction récursive nomDeLaFonction et que dans la définition de cette fonction, nomDeLaFonction n'apparaît pas, vous n'êtes pas en train d'écrire une fonction ittérative.

Vous ne respectez pas la consigne, vous êtes HORS-SUJET.

Exemple de fonction récursive

Nous allons détailler les différentes étapes permettant d'écrire une fonction récursive à travers un exemple.

Exemple : Somme des entiers

Nous cherchons à écrire une fonction Python SommeEntiers telle que SommeEntiers(n) renvoie le résultat
de l'addition 0+1+2+ … +n, pour n un entier positif.

Exemple 3

SommeEntiers(3) = 0 + 2 + 3 = 6
SommeEntiers(4) = 0 + 2 + 3 + 4 = 6
SommeEntiers(10) = 0 + 1 + 2 + 3 + 4 +...+ 10 = 55

En Python, version itérative

Vous savez très bien écrire cette fonction en Python, nous pouvons par exemple écrire :

```
def SommeEntiersIterative(n) :
        s = 0
        for i in range(n+1) :
                s += i
        return s
```

Les tests de cette fonction renvoie bien les résultats attendus. Mais cette fonction n'est pas récursive, la fonction n'est pas définie en fonction d'elle-même. Nous allons donc à présent chercher à la ré-écrire de façon récursive.

L’idée récursive

Important

La première étape de la construction algorithmique de SommeEntiers est l’idée récursive.
Il faut se poser la question : Comment ramener mon problème à un problème "plus petit" ?

Dans le cadre de notre exemple, quel est le lien entre le calcul de SommeEntiers(n-1) et SommeEntiers(n) ?

Ici, nous avons :

SommeEntiers(n) = n + SommeEntiers(n-1)

La deuxième et dernière étape de la construction algorithmique d'une fonction récursive est la recherche du (ou des) cas de base.

Nous avons vu dans la partie précédente que l'idée était de se ramener à un problème "plus petit",
mais quand ce processus s'arrête-t-il ?

Si je veux calculer SommeEntiers(4),

SommeEntiers(4) = 4 + SommeEntiers(3)
donc je dois calculer SommeEntiers(3), mais...
SommeEntiers(3) = 3 + SommeEntiers(2)
...
Quand est-ce que je m'arrête ? ? ?

Je peux choisir de m'arrêter en disant que
SommeEntiers(0) = 0

– Le choix du cas de base revient au développeur. Il se peut qu'il y ait plusieurs cas de base qui fonctionnent pour une même fonction.
– Pour une fonction, il se peut qu'il y ait plusieurs cas de base, il convient donc de prêter attention à cette étape pour ne pas en oublier !

– IMPORTANT : Nous pouvons avoir l'impression que les cas de base sont les "cas faciles" et avoir envie de commencer par la recherche des cas de base avant de chercher l'idée récursive lors de la construction algorithmique de notre fonction. C'EST UNE ERREUR !

Voyez dans cet exemple, comment on a trouvé le cas de base. Le(s) cas de base dépend(ent) de
l'idée récursive ! Il est donc fondamental de construire sa fonction dans cet ordre.

Enfin, la version récursive en Python :

Une fois que tout ce travail préparatoire a été fait, et que nous disposons :

– de l'idée récursive :

SommeEntiers(n) = n + SommeEntiers(n-1)

– et du cas de base :

SommeEntiers(0) = 0

La fonction s'écrit alors très simplement en Python :

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

Petite analyse de la récursivité

Nous aurons d'autres cours pour approfondir la notion de récursivité, mais donnons tout de même déjà quelques éléments d'analyse afin de mieux comprendre comment cela fonctionne.

Analyse de l'exécution

Là encore, observons un exemple pour comprendre comment l'ordinateur gère une telle fonction.

Pour calculer SommeEntiers(3) :

```
SommeEntiers(3)
                3 + SommeEntiers(2)
                            2 + SommeEntiers(1)
                                        1 + SommeEntiers(0)
                                                    0
```

La machine va lancer les différents appels récursifs en **mémorisant tous les calculs intermédiaires** qu'il n'a pas encore réussi à faire. Ces calculs "en cours" sont mémorisés dans une pile.

Une pile est une structure de données que vous étudierez dans l'UE de Licence 2 du même nom, mais visualisez la pile d'assiettes sales qui attend à côté de votre évier, et vous aurez déjà une très bonne idée de ce dont il s'agit !

Les appels récursifs à la fonction SommeEntiers s'empilent donc jusqu'à arriver sur un cas de base qui nous fournit un premier résultat (ici SommeEntiers(0) = 0).

Démarrent ensuite la phase suivant, toutes les calculs intermédiaires vont être dépilés et effectués en remontant.

Limite : hauteur de la pile

Il existe de nombreuses situations algorithmiques que vous croiserez tout au long de votre scolarité où il est efficace d'utiliser une fonction récursive.

Attention cependant, dans certains langages de programmation non fonctionnels (Python, C/C++, Java...),
la pile d'appels ne doit pas dépasser une certaine limite.

Vous pouvez tester la fonction SommeEntiers que nous avons écrit, vous observerez :

– SommeEntiers(970) fonctionne parfaitement et renvoie le résultat 470935
– SommeEntiers(971) ne fonctionne pas. Le message d'erreur *RecursionError : maximum recursion depth exceeded in comparison* est renvoyé :

à force d’empiler les calculs intermédiaires, on a dépassé la capacité de la pile, Python n’arrive plus à mémoriser tous les calculs qu’il doit faire.

Cette erreur est appelée débordement de la pile (ou stack overflow en Anglais).

Une erreur classique lorsqu'on débute l'apprentissage de la récursivité est de créer une récursivité infinie en écrivant par exemple :

```
def testInfini(n) :
        print("Python")
        if n == 0 :
                return "Récursivité"
        else :
                return testInfini(n)
```

L'erreur vient du fait que vous définissez testInfini(n) en fonction de testInfini(n) : appels récursifs infinis.

Vous obtiendrez le message d'erreur *RecursionError : maximum recursion depth exceeded in comparison :* en tentant de calculer votre exemple, Python aura bouclé jusqu'à atteindre la capacité maximale de sa pile.

à force d’empiler les calculs intermédiaires, on a dépassé la capacité de la pile, Python n’arrive plus à mémoriser tous les calculs qu’il doit faire.

Cette erreur est appelée débordement de la pile (ou stack overflow en Anglais).

Une mauvaise utilisation de la récursivité

Pour terminer, donner un exemple de mauvaise utilisation de la récursivité, en considérant la suite de Fibonacci.

La suite de Fibonacci. La suite de Fibonacci est la suite d'entiers dont les premiers termes sont les suivants :

1 1 2 3 5 8 13 21 34 55 89 144 233 377 …

Les deux premiers termes de la suite sont 1 et 1, et ensuite chaque terme est la somme des deux précédents.

Le problème est d'écrire une fonction Fibo(n) qui calcule le n-ième terme de la suite de Fibonacci.

Une implémentation récursive.

Une implémentation récursive apparaît rapidement :

– Idée récursive : Fibo(n) = Fibo(n-1)+ Fibo(n-2) puisque chaque terme est la somme des deux termes précédents.

– Cas de base : Fibo(1) = 1e = Fibo(2), puisque les deux permiers termes sont égaux à 1 et ne peuvent être calculés à partir des termes qui précèdent. (**Un exemple où il y a deux cas de base ! !**)

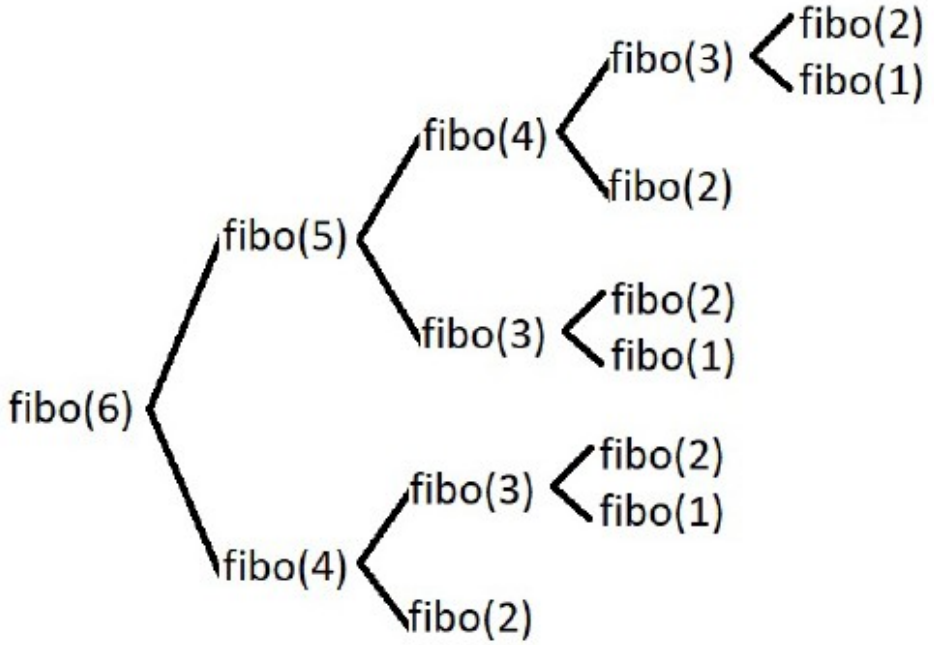
Ce qui donne

```
def fibo(n) :
        if n <= 2 :
                return 1
        else :
                return fibo(n-1) + fibo(n-2)
```

En réalité, le programme ci-dessus est extrêmement peu efficace, il répète un grand nombre de fois les mêmes calculs.

Observons l'exemple de fibo(6).
Pour calculer fibo(6), j'ai besoin de calculer fibo(5) et fibo(4). Mais pour calculer fibo(5), j'ai besoin de calculer fibo(4) et fibo(3), je vais donc calculer deux fois fibo(4).

Et ça ne s'arrête pas là ! On peut résumer tous ces calculs par l'arbre suivant :



Nous pouvons également observer ce phénomène sur Python en écrivant :

```
N = 0
def fibo(n) :
        global N
        N += 1
        if n <= 2 :
                return 1
        else :
                return fibo(n-1) + fibo(n-2)

print(fibo(30))
print("Nombre d'appels :", N)
------------------------------
832040
Nombres d'appels : 1664079
```

la variable globale N nous permet de compter le nombre d'appels récursifs qui vont être effectués.

Normalement, pour calculer fibo(30), nous avons besoin de connaître fibo(29), fibo(28), fibo(27),...., fibo(3), fibo(2), seulement 29 calculs intermédiaires sont nécessaires.

Mais on s'aperçoit que Python a effectué 1 664 079 appels ! !

Il est passé plus d'un million de fois dans la fonction fibo, il a refait plusieurs milliers de fois les mêmes calculs, quelle inefficacité !

Exercices

Exercice 1 : Somme des carrés

On suppose que n est un entier ≥ 1.
Écrire une fonction récursive sommeCarrés(n) qui calcule la somme des carrés de 1 à n, c'est à dire $1^2 + \ldots + n^2$.
Par exemple, sommeCarrés(3) = $1^2 + 2^2 + 3^2 = 14$.

Exercice 2 :

Écrire une fonction récursive factorielle(n) qui calcule n!.
Rappelons que :

$$n! = n(n-1)(n-2)...(2)(1)$$

par exemple, 3! = 3.2 .1 = 6. Par convention, 0! = 1.