Numérique et science informatique
Classe de Terminale

Lycée hoche

année scolaire 2021-2022

Contents

1 Exemples 2
1.1 Suite définie par récurrence 2
2 Autres exemples 3
2.1 La suite de Fibonacci 3
2.2 La somme des n premiers entiers consécutifs 4
2.3 Factorielle d'un nombre entier positif 5
3 Les limites de la récursivité 6
4 Bilan 6

Récursivité

Définition

Une fonction récursive est une fonction qui s'appelle elle-même.

1 Exemples

1.1 Suite définie par récurrence

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 (programme de terminale de la spécialité mathématiques).
Dans cet exemple, on s'intéresse à la suite (u_n) définie par son premier terme $u_0 = 2$ et la formule de passage d'un terme au suivant : $u_n = 3u_{n-1}$.On souhaite calculer le terme d'indice n de cette suite.

Version itérative

Il s'agit de la méthode que vous avez l'habitude d'utiliser pour répondre à un problème de programmation :
pour calculer u_3 par exemple, on calcule $u_1 = 3 \times u_0 = 3 \times 2 = 6$
$u_2 = 3 \times u_1 = 3 \times 6 = 18$
$u_3 = 3 \times u_2 = 3 \times 18 = 54$

Exercice 1

Ecrire une fonction qui calcule le terme d'indice n de la suite (u_n) de manière itérative sous la forme suivante:

```
def u(n):
# à compléter
return ...
assert u(3) == 54
```

Exercice 2

Ecrire une fonction qui calcule le terme d'indice n de la suite (u_n) de manière itérative sous la forme suivante:

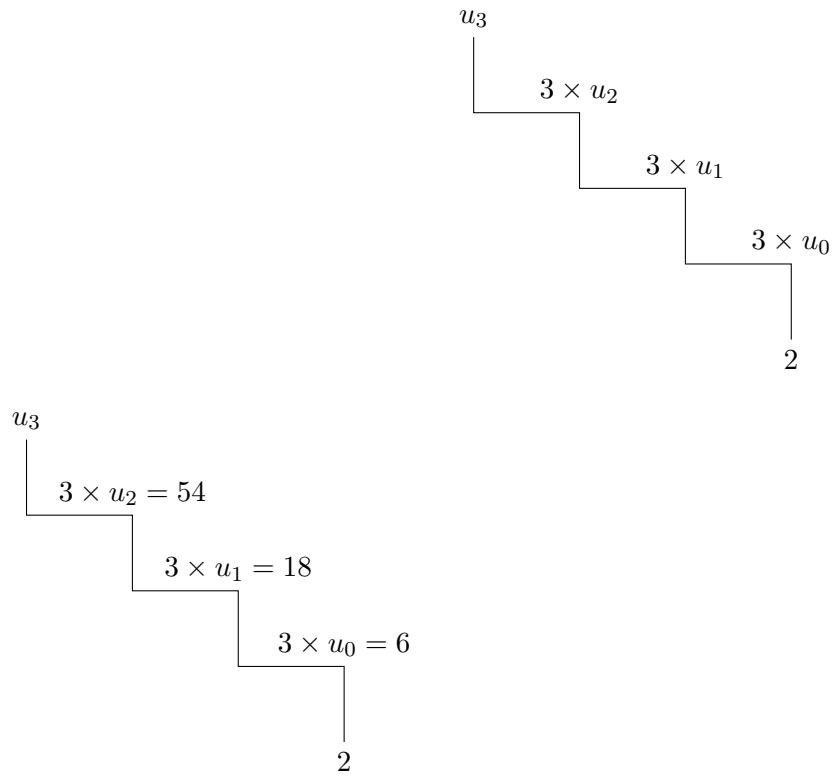
```
def u(n):
# à compléter
return ...
assert u(3) == 54
```

Version récursive

On peut également programmer le calcul des termes de cette suite en calquant sa définition par récurrence :

```
def u(n):
if n==1:
return 2
else:
    return 3 * u(n-1)
```

L'évaluation de l'appel à u(3) peut se représenter de la manière suivante.Cette manière de représenter l'exécution d'un programme en indiquant les différents appels effectués est appelée un **arbre d'appels**.



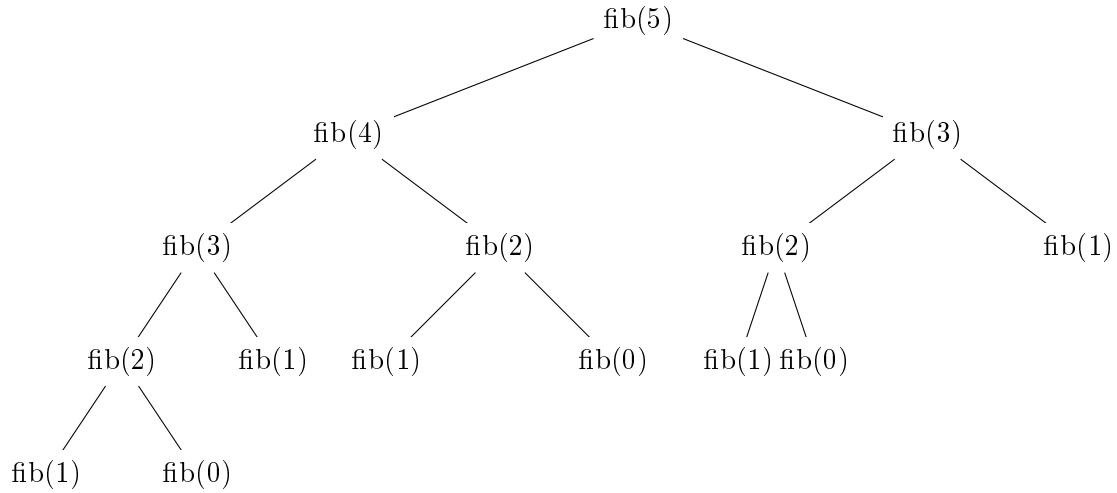
Ainsi, pour calculer la valeur renvoyée par u(3), il faut tout d'abord appeler u(2). Cet appel va lui-même déclencher un appel à u(1), qui à son tour nécessite un appel à u(0). Ce dernier appel se termine directement en renvoyant la valeur 2.

2 Autres exemples

2.1 La suite de Fibonacci

Cosidérons la suite u_n , définie par:
$u_0 = 1$, $u_1 = 1$ et $u_n = u_{n-1} + u_{n-2}$.

Dessinons l'arbre d'appel de la fonction:



Les instructions fib(1) et fib(0) sont appelées les **cas de base** de ce programme récursif :
il s'agit des cas pour lesquels on peut obtenir le résultat sans avoir recours à l'appel de la fonction.

Les cas de base sont indispensables à toute fonction récursive : ils permettent au programme de se terminer.

En effet, comme une fonction récursive s'appelle elle-même, alors, sans ces cas de base, la fonction s'appellerait à l'infini. C'est pour cette raison que les cas de base sont également appelés conditions d'arrêt.Pour concevoir un programme récursif, il faut toujours se poser la question :
« quel(s) est(sont) le(s) cas de base ? »

Exercice 3

A faire : Ecrire la version récursive de la fonction fib

```
def fib(n):
# à compléter
return ...

assert fib(3) == 3
assert fib(7) == 21
```

De la même manière:

Exercice 4

À faire : programmer des deux façons précédentes (version itérative et version récursive) le calcul du terme un d'une suite définie par ses deux premiers termes u0=5 et u1=7 et la formule de récurrence suivante :

$$u_n = 2 \times u_{n-2} + 3 \times u_{n-1}$$

2.2 La somme des n premiers entiers consécutifs

Pour définir la somme des n premiers entiers, on a l'habitude d'écrire la formule suivante :

$$0 + 1 + 2 + 3 + 4 + ... + n$$

Une solution pour calculer cette somme consiste à utiliser une boucle for pour parcourir tous les entiers i entre 0 et n, en s'aidant d'une variable intermédiaire S pour accumuler la somme des entiers de 0 à n.

Exercice 5

Ecrire le programme itératif qui calcule cette somme en python.

S'il n'est pas difficile de se convaincre que la fonction somme(n) ci-dessus calcule bien la somme des n premiers entiers, on peut néanmoins remarquer que ce code Python n'est pas directement lié à la formule de départ.
En effet, il n'y a rien dans cette formule qui puisse laisser deviner qu'une variable intermédiaire S est nécessaire pour calculer cette somme...

Il existe une autre manière d'aborder ce problème : il s'agit de définir la fonction récursive somme(n) qui, pour tout entier naturel n, donne la somme des n premiers entiers de la manière suivante :

- si $n = 0$: $somme(n) = 0$
- si $n > 0$: $somme(n) = n + somme(n - 1)$

Exercice 6

1. Écrire la fonction récursive som(n) qui calcule la somme des n premiers entiers.
2. Dessiner l'arbre d'appels de cette fonction pour l'appel somme(5).
3. Parcourir cet arbre d'appels «à rebours» pour illustrer le fonctionnement de la fonction somme comme cela a été fait dans la version récursive du premier .

```
def som(n):
# à compléter
return ...

assert som(4) == 10
assert som(100) == 5050
```

2.3 Factorielle d'un nombre entier positif

La factorielle d'un entier naturel n, noté $n!$ (se lit factorielle n), est le produit des nombres entiers strictement positifs inférieurs ou égaux à n :

$n = n \times (n-1) \times ... \times 1$ avec, par convention, 0!=1.

Exercice 7

1. Écrire la fonction récursive factorielle(n) qui calcule le nombre $n!$ pour tout entier naturel n
2. Dessiner l'arbre d'appels de cette fonction pour l'appel factorielle(7).
3. Parcourir cet arbre d'appels « à rebours » pour illustrer le fonctionnement de la fonction factorielle lorsque $n = 4$.
4. Vérifier les résultats avec 5!=120 et 7!=5040.

3 Les limites de la récursivité

Considérons à nouveau la fonction récursive somme(n) calculant la somme des n premiers entiers naturels.À expérimenter sur le poste maître :

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

1. Que produit l'appel à somme(1000) ?
2. Que produit l'appel à somme(3000) ?

Explications : une partie de l'espace mémoire d'un programme est organisée sous forme d'une pile où sont stockés les contextes d'exécution de chaque appel de fonction (voir le cours sur la gestion des processus). Lorsqu'un appel récursif se termine, cela permet à la fonction appelante de retrouver son contexte propre avec ses variables dans l'état où elles étaient avant appel.Chaque appel récursif consomme donc de la mémoire dans la pile d'exécution.
Prenons l'exemple de la fonction somme.L'organisation de la mémoire au début de l'appel à somme(5) peut être représentée par la pile ci-dessous contenant un contexte d'exécution avec, entre autres, un emplacement pour l'argument n initialisé à 5, ainsi que d'autres valeurs (comme l'emplacement pour la valeur renvoyée par la fonction) qui sont simplement représentées par des dans le schéma ci-après.

4 Bilan

Méthode de programmation : il convient de bien avoir en tête la façon de concevoir une fonction (ou un programme) récursive :

```
def f(x):
    if  #cas de base:
    #traitement direct du cas de base
    #pas d'appel récursif ici
    else:
    #Traitement du cas général par appel récursif
```

N'oubliez surtout pas les cas de base qui constituent la ou les condition(s) d'arrêt, sans quoi votre algorithme tournera à l'infini et votre programme s'arrêtera faute de mémoire disponible sans jamais rendre de réponse.

Choix entre récursif et itératif : la programmation récursive est à la fois un style de programmation mais également une technique pour définir des concepts et résoudre certains problèmes qu'il n'est parfois pas facile de traiter en programmant uniquement avec des boucles. Elle s'avère extrêmement pratique lorsque la résolution d'un problème se ramène à celle d'un problème plus petit. 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.

Un programmeur doit écrire une fonction récursive quand c'est la solution la plus adaptée à son problème.