Plan du chapitre

I La récursivité : une nouvelle façon de penser/coder

I.1 Un exemple introductif : le calcul de $n!$

Factorielle : point de vue itératif

$$5! = \underbrace{\left(\underbrace{\left(\underbrace{(1\times 2)}_{2}\times 3\right)}_{6}\times 4\right)}_{24}\times 5 = 120$$

Algorithme

Entrée : n

$A \leftarrow 1$

pour i allant de 2 à n **faire**

$\quad A \leftarrow A \times i$

Renvoyer : A

Programme

```
def fIt(n):
    A=1
    for i in range(2,n+1):
        A=A*i
    return A
```

Factorielle : point de vue récursif

Pour calculer $5! = 5 \times 4!$... je demande $4!$ à mon voisin et je multiplie le résultat par 5.

Programme

```
def fRec(n):
    if n<=1:  # Cas de base
        return 1
    else:
        return n*fRec(n-1) # Appel récursif
```

```
>>> fRec(5)
120
```

Moralité : il est tout à fait possible d'appeler une fonction dans son propre corps !

I.2 Définition

Définition

— On appelle **fonction récursive** toute fonction qui fait appel à elle-même, à l'intérieur de son propre corps.

— Une ligne de code où la fonction s'appelle elle-même est **un appel récursif**.

I.3 La pile d'exécution

Définition

Lors de l'exécution d'une fonction récursive, les arguments des appels récursifs, les variables locales ainsi que les adresses de retour des résultats sont stockés dans une pile, que l'on appelle la **pile d'exécution**.

Suivons pas à pas de la pile d'exécution sur un exemple

Considérons la suite $(u_n)_{n\geqslant 0}$ suivante, qui converge vers $\sqrt{2}$:

$$u_0 = 2 \qquad \text{et} \qquad \forall n \geqslant 1, u_n = \frac{u_{n-1}}{2} + \frac{1}{u_{n-1}}.$$

```
def u(n):
    if n==0:  #Cas  de base
        return 2
    else:
        A=u(n-1)  #Appel récursif
        return A/2+1/A
```

```
>>> u(2)
1.4166666666666665
```

Le suivi pas à pas de la pile d'exécution sur un exemple

```
def u(n):
    if n==0:
        return 2
    else:
        A=u(n-1)
        return A/2+1/A
```

```
>>> u(2)
1.4166666666666665
```

Limitation de la récursivité en Python

Attention! Python limite arbitrairement la hauteur de la pile à 1000 étages. Une fonction qui fait 1000 appels récursifs provoque un message d'erreur.

```
>>> u(1000)
RuntimeError: maximum recursion depth exceeded
```

I.4 Terminaison et correction d'une fonction récursive

Comment raisonner à propos d'une fonction récursive

Méthode

Pour démontrer la terminaison et la correction d'une fonction récursive, on raisonne par récurrence.

Reprenons l'exemple de la suite $(u_n)_{n\geqslant 0}$ définie par :

$$u_0 = 2 \qquad \text{et} \qquad \forall n \geqslant 1, u_n = \frac{u_{n-1}}{2} + \frac{1}{u_{n-1}}.$$

```
def u(n):
    if n==0:
        return 2
    else:
        A=u(n-1)
        return A/2+1/A
```

On veut montrer par récurrence sur $n \geqslant 0$ la propriété suivante : H_n : " l'appel **u(n)** termine et renvoie la valeur u_n. "

- ▷ La propriété H_0 est vérifiée car **u(0)** se réduit à **return 2**.
- ▷ On suppose H_{n-1} vérifié pour UN entier $n > 0$.

  Le calcul de **u(n)** commence par **A=u(n-1)**.

  Par hypothèse de récurrence, cet appel récursif se termine et renvoie u_{n-1}.

  Puis l'appel à **u(n)** renvoie la valeur de

  $$\frac{A}{2} + \frac{1}{A} = u_n,$$

  ce qui démontre H_n.

I.5 Complexité temporelle d'une fonction récursive

Complexité temporelle

Méthode

Pour déterminer la complexité temporelle, on cherche $C(n)$ le nombre d'opérations arithmétiques (addition, multiplication et division) effectuées par l'appel **u(n)**.

```
def u(n):
    if n==0:
        return 2
    else:
        A=u(n-1)
        return A/2+1/A
```

▷ $C(0) = 0$

▷ $\forall n \geqslant 1, C(n) = 3 + C(n-1)$.

La suite $(C(n))_{n\geqslant 0}$ est arithmétique :

$$\boxed{\forall n \geqslant 0, C(n) = 3n}.$$

Complexité temporelle : et si on avait écrit plus naïvement la fonction u ?

```
def u(n):
    if n==0:
        return 2
    else:
        return u(n-1)/2+1/u(n-1)
```

▷ $C(0) = 0$

▷ $\forall n \geqslant 1, C(n) = 3 + C(n-1) + C(n-1)$.

La suite $(C(n))_{n\geqslant 0}$ est arithmético-géométrique :

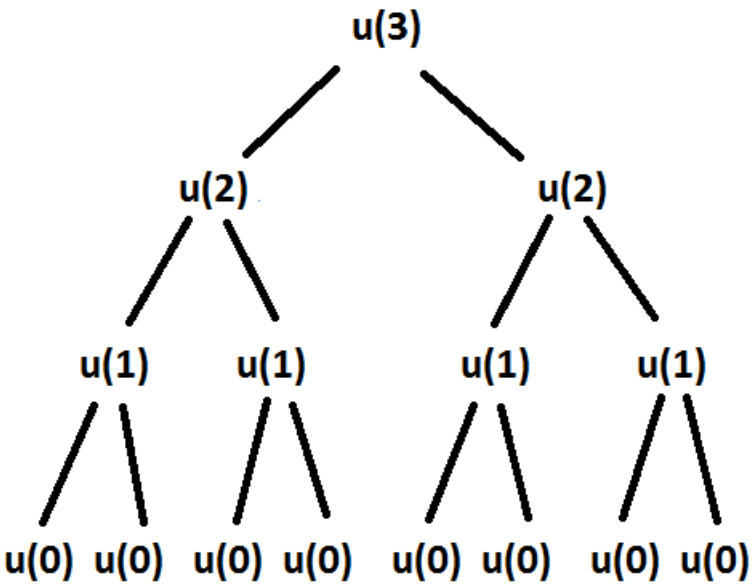
$$\boxed{\forall n \geqslant 0, C(n) = 3\left(2^n - 1\right)}.$$

Une complexité géométrique : fonction inutilisable en pratique !

Essayez en traçant les appels récursifs !

```
def u(n):
    print("Appel de u(",n,")")
    if n==0:
        return 2
    else:
        return u(n-1)/2+1/u(n-1)
```

```
>>> u(3)
Appel de u( 3 )
Appel de u( 2 )
Appel de u( 1 )
Appel de u( 0 )
Appel de u( 0 )
Appel de u( 1 )
Appel de u( 0 )
Appel de u( 0 )
Appel de u( 2 )
Appel de u( 1 )
Appel de u( 0 )
Appel de u( 0 )
Appel de u( 1 )
Appel de u( 0 )
Appel de u( 0 )
1.4142156862745097
```



II La récursivité en pratique

II.1 Comment concevoir une fonction récursive

Les règles d'or pour concevoir une fonction récursive

- On se concentre sur une formule de récursion : comment ramener notre problème à un problème analogue mais de taille plus petite?
- **Ne surtout pas oublier les cas terminaux/de base**, en général naturels. **L'écriture d'une fonction récursive commence impérativement le traitement des cas de base.**
- **Il faut qu'en enchaînant les appels récursifs, on tombe sur un cas de base.**
- On fait confiance en l'interpréteur/compilateur pour gérer la magie de l'affaire.

II.2 Quelques exemples

Entrainement 1

Calcul de π.

On peut démontrer que $\pi = 2\prod_{i=1}^{+\infty} \frac{4i^2}{4i^2-1}$.

Ecrire une fonction récursive **produit** qui prend comme paramètre un entier $n \geqslant 1$ et qui renvoie $2\prod_{i=1}^{n} \frac{4i^2}{4i^2-1}$. Evaluer la complexité temporelle.

Entrainement 2

Nombre de 0 dans une liste d'entiers

Ecrire une fonction récursive **nombreZeros** qui prend comme argument une liste d'entiers et qui renvoie le nombre d'occurences de 0 dans cette liste.

Entrainement 3

Une fonction mystere

Que fait la fonction mystère suivante, qui prend comme argument une liste **L** d'entiers.

```
def mystere(L):
    if len(L)==1:
        return L[0]
    elif L[0]<L[1]:
        L.pop(1)
    else:
        L.pop(0)
    return mystere(L)
```

Entrainement 4

Suite de Fibonacci

Ecrire une fonction récursive **fibo** prenant en entrée un entier positif n, et renvoyant la valeur f_n du terme d'indice n de la suite de Fibonacci définie par :

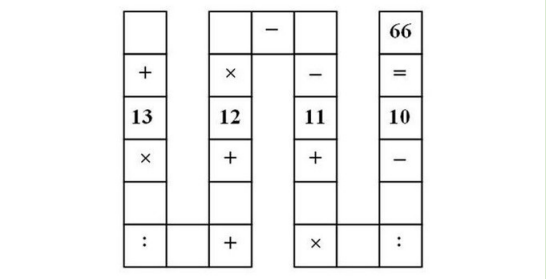
$$f_0 = 1 \qquad f_1 = 1 \qquad \forall p \in \mathbb{N}, f_{p+2} = f_{p+1} + f_p.$$

Evaluer la complexité temporelle. Comparer avec une version itérative de la fonction.

Entrainement 5

Un casse-tête donné à des élèves de CE2!

On veut remplir les cases vides de la grille suivante avec des chiffres de 1 à 9 (qu'il ne faut utiliser qu'une fois chacun) en suivant l'ordre des opérations de façon à obtenir, comme résultat final, le nombre 66.



Montrer que ce casse-tête possède 136 solutions et les déterminer toutes.

II.3 Avantages et inconvénients

Avantages

- C'est magique!
- L'écriture du code est très facile et naturelle.
- Avec une bonne méthodologie (base+propagation), le code est fiable.
- Preuves de correction en général facile (pas d'invariant à trouver).

Inconvénients

- C'est magique!
- Possibilité d'explosion de la complexité (Fibonacci).
- Possibilité d'une taille de pile trop importante.
- Complexité spatiale.