Récursivité

C. Charignon

Table des matières

Première partie

Cours

Principe de Hofstadter : il faut toujours plus de temps que prévu, même en tenant compte du principe de Hofstadter.

1 Premier exemple : dichotomie

Soit $(a,b) \in \mathbb{R}^2$ tel que $a < b$, et soit $f \in \mathcal{C}([a,b],\mathbb{R})$. On suppose que $f(a)$ et $f(b)$ sont de signe différent, de sorte que le théorème des valeurs intermédiaires prouve qu'il existe $c \in [a,b]$ tel que $f(c) = 0$.

Fixons un $\epsilon \in \mathbb{R}^{+*}$, et rappelons le principe de la dichotomie pour calculer une valeur approchée d'un tel c à une précision ϵ :

- Soit $m = \dfrac{a+b}{2}$, le milieu du segment $[a,b]$.
- Si $f(a)f(m) \leqslant 0$, on poursuit la recherche sur $[a,m]$.
- Sinon, on poursuit la recherche sur $[m,b]$.
- On continue ainsi jusqu'à ce que l'intervalle de recherche soit de longueur $\leqslant \epsilon$.

Ceci est bien trop vague pour être appelé un algorithme. Le « on poursuit la recherche » et le « on continue ainsi » doivent être traduits proprement.

La méthode traditionnelle pour gérer la répétition est d'utiliser une boucle. Comme nous ne savons pas à l'avance le nombre d'itérations à faire[1], nous utilisons une boucle « tant que ».

Ce qui donne :

```
def dicho(f,a,b,eps):
    """
    Précondition : f(a)*f(b) <= 0.
    Renvoie un encadrement d'un réel ∈c[a,b] tel que f(c)=0, de longueur <= eps.
    """

    deb,fin = a,b
    while fin-deb >eps:
        m=(a+b)/2
        if f(a)*f(m)<=0: fin=m
        else: deb=m

    return deb,fin
```

Cependant, il y a une autre manière de traduire le « on poursuit la recherche sur $[a,m]/\ [m,b]$ » : c'est tout simplement de relancer le programme sur le segment $[a,m]$ (ou $[m,b]$) à la place de $[a,b]$.

Ce qui donne :

```
def dicho2(f,a,b,eps):
    """
    Précondition : f(a)*f(b) <= 0.
    Renvoie un encadrement d'un réel ∈c[a,b] tel que f(c)=0, de longueur <= eps.
    """

    m=(a+b)/2
    if f(a)*f(m) <=0:
        return dicho2(f,a,m,eps)
    else:
        return dicho2(f,m,b,eps)
```

En réfléchissant un peu, on constate que ce programme ne va jamais s'arrêter : il se ré-appelle lui-même éternellement. Or nous voulons qu'il s'arrête dès que l'intervalle de recherche est de longueur inférieure à `eps`.

Pour ce, il suffit de rajouter deux lignes :

1. Bien qu'en réfléchissant un peu, on se rende compte que c'est $\left\lfloor \log_2\left(\dfrac{b-a}{\epsilon}\right) \right\rfloor$

```python
def dicho2(f,a,b,eps):
    """
    Précondition : f(a)*f(b) <= 0.
    Renvoie un encadrement d'un réel ∈c[a,b] tel que f(c)=0, de longueur <= eps.
    """

    if b-a < eps:
        return a,b
    else:
        m=(a+b)/2
        if f(a)*f(m) <=0:
            return dicho2(f,a,m,eps)
        else:
            return dicho2(f,m,b,eps)
```

Un programme comme celui-ci qui se ré-appelle lui-même s'appelle un programme « récursif ».

Voici quelques avantages d'un programme récursif. Ces avantages sont principalement liés au fait que le style récursif est plus proche des maths.

- Le code est souvent plus court et plus clair.
- La preuve et l'étude de la complexité est plus simple (voir la partie 3).
- En conséquence de quoi, il y a moins de risque d'erreur : un programme récursif est plus sûr qu'un programme impératif.
- Enfin, toute personne habituée à faire des raisonnements par récurrence en mathématiques, et à manipuler des suites définies par récurrence, trouvera les programmes récursifs parfaitement naturels.
- La conception d'un programme récursif est souvent plus simple.

Il y toutefois de nombreuses circonstances où un programme impératif est plus naturel. À commencer par les parcours simples de tableaux. Une des compétences d'un bon programmeur est de savoir choisir pour chaque problème le style de programmation qui donnera le code le plus clair et efficace.

Au rang des défauts des programmes récursifs, il faut mentionner que les ordinateurs actuels sont optimisés pour les programmes impératifs (c'est-à-dire avec des boucles), de sorte qu'un programme récursif sera souvent légèrement plus lent qu'un programme impératif.

Signalons enfin qu'il a été démontré que tout programme écrit en programmation impérative peut être traduit en programmation récursive, et réciproquement. Cependant, il y a des situations dans lesquelles l'un ou l'autre de ces deux styles sera plus simple à mettre en œuvre, et cela fait partie du travail d'un bon informaticien que de choisir le style de programmation le plus adapté à son problème.

Ceci étant, la tradition est d'enseigner les boucles avant la récursivité.

2 Autres exemples

De manière générale, lorsqu'on dispose d'une relation de récurrence et d'une condition initiale pour calculer une suite, on peut traduire ceci immédiatement en un programme récursif.

Commençons par la fonction factorielle. La définition mathématique de la suite factorielle est :

$$\begin{cases} 0! = 1 \\ \forall n \in \mathbb{N},\ (n+1)! = (n+1) \times n! \end{cases}.$$

Mais il sera plus pratique pour la traduction informatique d'effectuer un décalage d'indice : exprimer $n!$ en fonction de $(n-1)!$ plutôt que $(n+1)!$ en fonction de $n!$ comme on le fait traditionnellement en maths. Ceci devient :

$$\begin{cases} 0! = 1 \\ \forall n \in \mathbb{N}^*,\ n! = n \times (n-1)! \end{cases}$$

(Notez le $\mathbb{N}^*$ au lieu de $\mathbb{N}$.)

Ceci se traduit immédiatement en Python :

```python
def facto(n):
    if n=0:
        return 1
    else:
        return n * facto(n-1)
```

Voici encore deux petits exemples :

- $\sum_{i=0}^{n} \frac{x^i}{i!}$ (ou plus généralement n'importe quelle série) ;
- pgcd.

```python
# série expo
def expo(x,n):
    if n==0:
        return 1
    else:
        return expo(x,n-1) + x**n/factorielle(n)


# pgcd
def euclide(a,b):
    if b==0:
        return a
    else:
        q,r=divmod(a,b)
        return euclide(b,r)

# Plus court : en constatant que le quotient de la DE est inutile
def euclide(a,b):
    """ Précondition : (a,b) != (0,0) """
    if b==0:
        return a
    else:
        return euclide(b,a%b)
```

3 Complexité et preuve

Prouver qu'une fonction récursive est correcte, et calculer sa complexité, est beaucoup plus pratique à rédiger que pour une fonction contenant des boucles. Il suffit de procéder par récurrence. On peut dire qu'une fonction récursive est le pendant informatique d'une démonstration par récurrence.

Reprenons l'exemple de la factorielle : commençons par démontrer que ce programme est correct.

On définit le prédicat P suivant : $\forall n \in \mathbb{N}$, $P(n)$: « `facto(n)` termine et renvoi $n!$ ».

- *Initialisation :* Vu le premier cas du « if », on constate que `facto(0)` termine et renvoi 1. Or $0! = 1$, ainsi `facto(0)` termine et renvoi 0!, donc $P(0)$ est vrai.
- *Hérédité :* Soit $n \in \mathbb{N}$, supposons $P(n)$. Suivons l'exécution du calcul de `facto(n+1)`. Comme $n+1 \neq 0$, on arrive dans le « else ». Le programme calcule donc `facto(n)`. Or, d'après $P(n)$, `facto(n)` termine et renvoi $n!$.
  Alors le calcul de `facto(n+1)` termine et renvoi $(n+1) \times n!$, qui vaut bien $(n+1)!$.
  Donc $P(n+1)$ est vrai.

En conclusion, P est initialisé et héréditaire, donc $\forall n \in \mathbb{N}$, $P(n)$.

À présent, étudions la complexité. On commence par définir la suite donnant le nombre d'opérations : pour tout $n \in N$, je pose C_n le nombre de multiplication pour calculer `facto(n)`. On constate alors en regardant le code que :

$$\begin{cases} C_0 = 0 \\ \forall n \in \mathbb{N}^*,\ C_n = C_{n-1} + 1 \end{cases}.$$

En effet, le « C_{n-1} » vient de l'appel récursif, et le « $+1$ » de la multiplication par n.
On voit alors que C est arithmétique de raison 1, et $\forall n \in \mathbb{N}$, $C_n = n$.

4 Exemple de mauvaise fonction récursive

4.1 Méthode de Héron

Traitons l'exercice 1. Voici une réponse naïve à cet exercice :

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

On se rend compte que cette fonction ne parvient à donner une réponse en temps raisonnable lorsque $n \geqslant 30$. Étudions donc sa complexité.

Notons pour tout $n \in \mathbb{N}$, C_n le nombre de divisions (par exemple) pour calculer $u(a,n)$. En lisant l'algorithme, on constate que :

$$\begin{cases} C_0 = 0 \\ \forall n \in \mathbb{N}^*,\ C_n = 2C_n + 2 \end{cases}.$$

Ainsi, la suite C est arithmético-géométrique. On vérifie que $C+2$ est géométrique de raison 2, puis que :

$$\forall n \in \mathbb{N}, C_n = 2^{n+1} - 2$$

Ainsi la complexité est exponentielle !

Pourtant, imaginez vous en train de calculer u_n à partir de u_0 : pour passer d'un terme de la suite au suivant, vous faites deux divisions, il vous faudra donc un total de $2n$ divisions pour passer de u_0 à u_n.

Si vous pouvez le faire en $2n$ divisions, l'ordinateur devrait pouvoir lui aussi !

Pour bien comprendre le phénomène, traçons l'arbre des appels par exemple de $u(a,10)$.

On remarque alors que la fonction est appelée un grand nombre de fois pour les mêmes arguments. Il est très simple de corriger cette fonction : on va calculer une seule fois `u(a,n-1)` et l'enregistrer.

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

4.2 Fibonacci

Autre exemple : la suite de Fibonacci. Soit $F \in \mathbb{N}^{\mathbb{N}}$ telle que $\begin{cases} F_0 = 0, F_1 = 1 \\ \forall n \in \mathbb{N}, F_{n+2} = F_{n+1} + F_n \end{cases}$.

Remarque : Nombre de manières de monter un escalier par pas de 1 ou 2 marche(s).

Une programmation naïve de cette suite est :

```
def fiboNaif(n):
    if n==0 :
        return 0
    elif n==1:
        return 1
    else:
        return fiboNaif(n-1)+fiboNaif(n-2)
```

Notons pour tout $n \in \mathbb{N}$ le nombre d'additions effectuées par `fiboNaif n`. Vu le programme,

$\begin{cases} C_0 = 0 = C_1 \\ \forall n \in [\![2, \infty[\![, C_n = C_{n-1} + C_{n-2} + 1 \end{cases}$.

Sans ce $+1$, ça serait une suite récurrente double, et nous saurions calculer son terme général par le cours de math. On peut employer la même méthode que pour étudier une suite arithmético-géométrique pour se débarrasser de ce $+1$: on chercher une constante k telle que $C-k$ soit vraiment une suite à récurrence linéaire double, c'est-à-dire telle que $\forall n \in [\![2, \infty[\![, (C_n - k) = (C_{n-1} - k) + (C_{n-2} - k)$. Manifestement, $k = 1$ convient.

On pose alors $u = C + 1$ (c'est-à-dire $\forall n \in \mathbb{N}$, $u_n = C_n + 1$). Donc $u_0 = 1, u_1 = 1$ et $\forall n \in \mathbb{N}$, $u_{n+2} = u_{n+1} + u_n$.

La suite u est une brave suite à récurrence double linéaire. Son équation caractéristique est $r^2 = r + 1$ d'inconnue $r \in \mathbb{C}$, dont les racines sont $\dfrac{1+\sqrt{5}}{2}$ et $\dfrac{1-\sqrt{5}}{2}$. Donc il existe deux constantes $(\alpha, \beta) \in \mathbb{R}^2$ telle que

$$\forall n \in \mathbb{N}, u_n = \alpha \left(\frac{1+\sqrt{5}}{2}\right)^n + \beta \left(\frac{1-\sqrt{5}}{2}\right)^n$$

Comme $\left|\dfrac{1-\sqrt{5}}{2}\right| < \left|\dfrac{1+\sqrt{5}}{2}\right|$, on a $\left|\dfrac{1-\sqrt{5}}{2}\right|^n \ll_{n\to\infty} \left|\dfrac{1+\sqrt{5}}{2}\right|^n$. D'où $\left|\dfrac{1-\sqrt{5}}{2}\right|^n \ll_{n\to\infty} \beta\left|\dfrac{1+\sqrt{5}}{2}\right|^n$.

Il est impossible que $\alpha = 0$ sans quoi on aurait $\lim_{n\to\infty} C_n = -1$, alors que C est une suite positive. Ainsi, $\alpha\left|\dfrac{1-\sqrt{5}}{2}\right|^n \ll_{n\to\infty} \beta\left|\dfrac{1+\sqrt{5}}{2}\right|^n$.

Dès lors :

$$C_n = u_n - 1 \underset{x\to\infty}{\sim} \alpha \left(\frac{1+\sqrt{5}}{2}\right)^n \underset{n\to\infty}{=} O\left(\left(\frac{1+\sqrt{5}}{2}\right)^n\right).$$

C'est une complexité exponentielle !

Remarque : On peut calculer α en utilisant les conditions initiales. Le calcul de α, n'est pas nécessaire si on s'intéresse seulement à l'ordre de grandeur. Le calcul de β lui est franchement inutile.

cf exercice : 8 pour une version efficace du calcul de cette suite.

Remarque : Plus simple pour se débarrasser de la constante : On a $\forall n \in \mathbb{N}$, $C_{n+2} \geqslant C_{n+1} + C_n$, $C_0 = 0$ et $C_1 = 1$. Notons u la suite telle que $u_1 = 1$, $u_2 = 2$ et $\forall n \in \mathbb{N}$, $u_{n+2} = u_{n+1} + u_n$, on montre alors par récurrence simple que $\forall n \in \mathbb{N}$, $C_n \geqslant u_n$.

5 Diviser pour régner

5.1 Principe

La stratégie « diviser pour régner » est une stratégie récursive qui s'applique à de nombreuses situations et permet souvent d'obtenir des algorithme efficaces. Elle se compose de trois étapes :

- *Diviser :* Diviser le problème à résoudre en deux (ou plus) problèmes plus petits ;
- Résoudre, par un simple appel récursif, les petits problèmes ;
- *Régner :* En déduire une solution du problème initial.

Et n'oubliez pas le cas d'arrêt lorsque le problème à résoudre est suffisament petit.

5.2 Exemple : calcul de puissances

Nous avons déjà rencontré un algorithme de ce type : la dichotomie. Cependant, il s'agissait là d'un cas très simple puisque une fois l'intervalle divisé en deux, la recherche se poursuit dans un seul des demi-intervalles. Il n'y a donc besoin que d'un seul appel récursif. De ce fait, il est facile de programmer l'algorithme sans récursivité, ce qui avait été fait en première année.

Au chapitre suivant, nous étudierons deux algorithmes de type « diviser pour régner », pour lesquels il y aura besoin de deux appels récursifs, ce qui fait qu'une version impérative serait bien plus compliquée à concevoir.

En attendant, voici un autre exemple simple d'algorithme de type « diviser pour régner », permettant de calculer des puissances. C'est un classique : je vous conseille d'en connaître le principe.

Fixons $x \in \mathbb{R}$. Pour tout $n \in [\![2, \infty[\![$, pour calculer x^n nous procédons ainsi :

1. *Diviser* : On calcule $n//2$ (notation Python : c'est la division euclidienne de n par 2).
2. *Appel récursif :* un appel récursif à la fonction nous fournit $x^{n//2}$.
3. *Régner :* Pour en déduire x^n :
   - Si n est pair, $x^n = \left(x^{n//2}\right)^2$;

- Sinon, $x^n = \left(x^{n//2}\right)^2 \times x$.

En Python, ceci donne :

```
266
267 def puissance(x,n):
268     if n==0:
269         return 1
270     else:
271
272         k=n//2
273         xpk=puissance(x,k)
274         return xpk**2 * x**(n%2)
```

cf exercice : 8, question 3.

Deuxième partie

Exercices

Exercices : récursivité

Exercice 1. *! Calcul approché de racines carrées

Soit $a \in \mathbb{R}^{+*}$. Soit $u \in \mathbb{R}^{\mathbb{N}}$ la suite vérifiant $u_0 = a$ et $\forall n \in \mathbb{N}$, $u_{n+1} = \dfrac{u_n}{2} + \dfrac{a}{2u_n}$. On démontre que u converge très rapidement vers $\sqrt{a}$ (c'est la méthode de Newton !).

Écrire une fonction prenant en argument a et n et calculant u_n.

Exercice 2. * Calcul approché de racines 2 : suites récurrentes croisées

Soit $a \in \mathbb{R}^{+*}$. Soient $(u, v) \in (\mathbb{R}^{\mathbb{N}})^2$ les deux suites définies par : $\begin{cases} u_0 = 1 \text{ et } v_0 = a \\ \forall n \in \mathbb{N}, u_{n+1} = \frac{2u_n v_n}{u_n + v_n} \text{ et } v_{n+1} = \frac{u_n + v_n}{2} \end{cases}$

(On démontre facilement que ces deux suites convergent vers $\sqrt{a}$.)

Écrire des fonctions récursives u et v pour calculer u_n et v_n en fonction de n.

Comme pour l'exercice précédent, si vous n'arrivez pas à dépasser $n = 30$ c'est que vous n'avez pas été efficace...

Exercice 3. **! Coefficients binomiaux

On souhaite écrire une fonction `coeffBinomial` prenant en entrée deux entiers positifs p et n et calculant $\binom{n}{p}$.

1. On propose 3 méthodes. Les essayer, et identifier la plus efficace des trois.
   - En écrivant et utilisant une fonction `factorielle` .
   - En se basant sur la relation de Pascal.
   - En démontrant puis utilisant la formule (dite « du pion ») : $\forall (n,p) \in (\mathbb{N}^*)^2$, $\binom{n}{p} = \dfrac{n}{p}\binom{n-1}{p-1}$.

     N.B. Le résultat devra être un entier, et pas un flottant. On demande pour des raisons d'efficacité que tous les calculs soient faits en entiers.
2. Pour la deuxième méthode : étudions le calcul de $\binom{7}{3}$. Remplir un triangle de Pascal en inscrivant dans chaque case combien de fois le coefficient correspondant a été calculé. Autrement dit, pour tout (n,p), on écrira dans la case n, p le nombre d'appels faits à `CB2(n,p)`. En déduire le nombre total d'appels récursifs à la fonction qui a été effectué.
3. Montrer que de manière générale, le nombre d'additions pour calculer $\binom{n}{p}$ grâce à la relation de Pascal est supérieur à $\binom{n}{p} - 1$.

Exercice 4. ** Parcourir les fichiers d'un répertoire

Écrire une procédure qui affiche la liste de tous les fichiers d'un répertoire. Bien sûr, on affichera aussi le contenu des sous-répertoires, des sous-sous-répertoire, etc. [2]

Les commandes utiles sont présentes dans la bibliothèque « `os` ». Ainsi :

- `os.listdir(chemin)` renvoie la liste des fichiers et répertoires contenus dans `chemin` (`chemin` doit être le chemin d'accès complet à un répertoire, de type `str`).
  Notez que si `f` est le nom d'un fichier renvoyé par `os.listdir(chemin)`, alors le chemin complet vers ce fichier sera `chemin +"/"+f`. Ou plus propre, car indépendant du système d'exploitation [3], `os.path.join(chemin,f)`.
- `os.path.isdir(chemin)` indique si `chemin` est un répertoire.
- `os.path.isfile(chemin)` indique si `chemin` est un fichier.

Exercice 5. **! Tours de Hanoï

Soit $n \in \mathbb{N}^*$. On dispose de n anneaux $A_0, ..., A_{n-1}$ qu'on peut empiler sur trois tiges T_0, T_1, et T_2. Les anneaux sont de taille strictement croissante : A_0 est le plus petit et A_{n-1} le plus grand.

Au début du jeu, tous les anneaux sont sur T_0, rangé dans l'ordre décroissant (A_{n-1} en bas, A_0 en haut).

Le but du jeu est de déplacer tous les anneaux vers la tour T_1, mais en s'assurant qu'à chaque instant, chaque anneau repose sur un anneau plus gros. De plus on n'est autorisé à déplacer qu'un seul anneau à la fois.

On va écrire en pseudo-code l'algorithme pour résoudre le jeu. Pour rédiger, on écrira $i \to j$, pour dire qu'on déplace l'anneau situé au sommet de la colonne i vers le sommet de la colonne j.

2. C'est la commande `ls -R` sous unix.

3. De manière générale, le but de la bibliothèque `os` est de fournir des fonctions qui fonctionnent pour n'importe quelle système d'exploitation

1. Résoudre le problème pour $n = 2$ puis 3.
2. Supposons qu'on soit capable de déplacer le bloc des $n-1$ plus petits anneaux d'une colonne vers une autre. Comment pourrait-on en déduire une méthode pour déplacer les n anneaux ?
3. En déduire un algorithme récursif `hanoi` prenant en entrée un entier n et déplaçant n anneaux de la tige 0 vers la tige 1.
4. Calculer la complexité de l'algorithme.
5. Traduire votre algorithme en Python. On utilisera des instructions du type `print( "déplacer un anneau de la tige {} vers la tige {}".format(i,j))`. Ou pour ceux qui ont une version récente de Python, `print( f"déplacer un anneau de la tige {i} vers la tige {j}")`
6. (***) Démontrer que l'algorithme est optimal, c'est-à-dire que tout algorithme permettant de résoudre le jeu à une complexité d'au moins $2^n - 1$ anneaux déplacés.

Exercice 6. * Une fonction inutile

On considère la fonction `f` suivante :

```
def f(n):
    if n==0:
        return 0
    else:
        res=0
        for i in range(n):
            res += f(n-1)
        return res
```

1. Que renvoie f ? Le démontrer par récurrence.
2. On note pour tout $n \in \mathbb{N}$, C_n le nombre d'additions pour exécuter `f(n)`.
   (a) Donner C_0 ainsi que la relation de récurrence vérifiée par la suite C.
   (b) Démontrer que pour tout $n \in N^*$, $C_n \geqslant n!$. Commentaires sur cette fonction ?
   (c) *Bonus :* écrire une fonction plus simple pour calculer la même chose que `f`.

Exercice 7. ** Copie profonde

1. Écrire une fonction `nouvelleMatrice` qui prend en entrée deux entiers n et p et qui renvoie une matrice de format $n \times p$.
2. Écrire une fonction `copie` qui effectue une copie d'un tableau, quel que soit son niveau de profondeur. C'est donc la fonction `deepcopy` de la bibliothèque `copy` (à la différence que la fonction `deepcopy` fonctionne sur n'importe type d'objet et pas seulement les tableaux).
   Pour savoir si un objet `x` est un tableau : `isinstance(x, list)` .

Exercice 8. ** Fibonacci efficace : plusieurs méthodes

Soit $F \in \mathbb{N}^{\mathbb{N}}$ telle que $F_0 = 0, F_1 = 1$ et $\forall n \in \mathbb{N}$, $F_{n+2} = F_{n+1} + F_n$.

On a vu en cours que la programmation récursive naïve de cette suite est inefficace. On propose ci-dessous deux méthodes pour résoudre ce problème. Pour chacune on aura besoin de définir une fonction auxiliaire.

Pour chacune de ces méthodes, indiquer la complexité.

1. **Fonction auxiliaire qui renvoie une valeur supplémentaire :**
   On utilise une fonction auxiliaire `fiboAux` qui renvoie non pas seulement F_n, mais le couple (F_n, F_{n+1}).
2. **Fonction auxiliaire qui utilise un argument supplémentaire :**
   On utilise une fonction `fiboAux` qui prend en entrée deux termes consécutifs f_k et f_{k+1} et un entier n, et qui renvoie f_{k+n}.
3. **Par le calcul matriciel :** Appliquer la méthode des puissances divisées pour calculer une puissance de matrice. En déduire une nouvelle méthode pour calculer les termes de la suite de Fibonacci.
4. **Programmation dynamique :** Cette méthode n'est plus récursive. Il s'agit tout simplement de créer un tableau de n cases dans lequel on va enregistrer au fur et à mesure toutes les valeurs calculées : $f_0, f_1, \dots, f_n$.
   Programmer cette méthode et expliquer ses avantages et inconvénients.
   Remarque : La méthode consistant à remplir le triangle de Pascal utilisée dans l'exercice 3 utilise la même idée, à savoir créer un tableau pour enregistrer les valeurs précédentes.

Exercice 9. ** Une fractale : le dragon

Dans cet exercice, les points seront représentés par leur affixe. On rappelle que Python dispose d'un type `complex` pour représenter des nombres complexes. Quelques points de syntaxe :

- Le nombre i tel que $i^2 = -1$ s'obtient en tapant `1j`. Plus généralement, pour tous flottant a et b, `a+bj` représente le complexe $a + ib$.
- Les opérations classiques `* + - / ...` s'utilisent comme pour les flottants.
- On peut récupérer parties réelle et imaginaire par les attributs `real` et `imag`. Par exemple on tapera `z.real`.

1. Soient A et B deux points d'affixe z_A et z_B. Soit C tel que ABC est un triangle rectangle isocèle en C direct. Calculer l'affixe z_C de C.
2. Programmer la fonction `calcule_zc` correspondante.
3. La fonction `dragon` est alors définie par récurrence comme ceci :
   - Pour tous points A et B, `dragon(0,A,B)` est le segment $[AB]$.
   - Pour tous points A et B et tout entier $n \in \mathbb{N}^*$, soit C comme dans la question 1, alors `dragon(n,A,B)` est la concaténation de `dragon(n-1,A,C)` et de `dragon(n-1,B,C)`.

   Programmer la fonction `dragon`. Pour gagner du temps, on fournit une fonction `traceDepuisC` qui prend en argument une liste de nombres complexes et qui trace la suite de segments correspondante.

```
import matplotlib.pyplot as plt

def traceDepuisC(l):
    """ Trace la courbe donnée par une liste d'affixe."""
    lx, ly = [], []
    for z in l:
        lx.append(z.real)
        ly.append(z.imag)
    plt.plot(lx,ly)
    plt.show()
```

Quelques indications

1 Si votre fonction n'arrive pas à dépasser u_{30} c'est que vous êtes tombé dans le piège...

2 Écrire deux fonctions mutuellement récursives est en fait inefficace. Il vaut mieux écrire une seule fonction qui calcule les deux suites en même temps.

5 3) Il va falloir écrire un algorithme un peu plus général, qui prendra en entrée également les tiges de départ et d'arrivée. En outre, il sera pratique de mettre aussi la tige intermédiaire en entrée.

6) Pour déplacer n anneaux : il va falloir déplacer le plus gros au moins une fois. Ce qui impose : personne au dessus du plus gros, et une tige vide. Donc les $n-1$ autres anneaux ont été déplacés sur la troisième tige, le coût étant $\geqslant C_{n-1}$ par récurrence.

7 Le cas de base est lorsque l'objet n'est pas un tableau.

8 Par calcul matriciel : posons $X_n = \begin{pmatrix} F_n \\ F_{n+1} \end{pmatrix}$. Trouver une matrice A telle que $\forall n \in \mathbb{N}$, $X_{n+1} = AX_n$.

9 Attention : l'ordre des points compte. `dragon(n,A,B)` est différent de `dragon(n,B,A)`.

Lorsqu'on rassemble les deux morceaux, il faudra renverser une des deux listes pour qu'ils se raboutent correctement. Utiliser la fonction `rev`.

Quelques solutions

1

2

3

4

1. Programmer cette procédure.
2. (bonus) Faire en sorte que l'affichage d'un fichier soit décaler en fonction de sa profondeur dans l'arborescence des fichiers : un fichier dans un sous-répertoire sera décalé de 4 espaces, dans un sous-sous-répertoire de 8 espaces, etc.
3. Expliquer le choix de la lettre « R » dans la commande unix `ls -R`.

5

1 Pour tout $n \in \mathbb{N}$ posons $P(n)$: « $f(n)$ termine et renvoie 0. ».

- *Initialisation :* Vu le cas de base, $f(0) = 0$.
- Soit $n \in \mathbb{N}^*$, supposons $P(n-1)$. Exécutons $f(n)$. Par hyothèse de récurrence $f(n-1)$ termine et renvoie 0. Alors la boucle consiste à sommer n fois 0. Donc `res` contient 0 à son issue. Et $f(n)$ termine et renvoie 0.

2a C_0 et pour tout $n \in \mathbb{N}^*$, $C_n = n + n \times C_{n-1}$. Le $n+$ vient des n additions dans la boucle, et le $n \times C_{n-1}$ vient des n appels récursifs à $f(n-1)$ (ce qui est parfaitement idiot au passage : il aurait fallu calculer $f(n-1)$ une seule fois et enregistrer le résultat.).

2b Récurrence évidente. Attention à l'initialiser à 1 car pour $n = 0$ la formule est fausse.

2c

```
def f_mieux(n):
    return 0
```

7

8

9

```
def calcule_zc(za,zb):
    """ Renvoie l'affixe du point C tel que ABC est directe isocèle rectangle en C."""
    # zb-zc = exp(j*pi/2)*(za - zc)
    # donc zc=(1+j)/2*(zb-jza)
    return (1+1j)/2*(zb-1j*za)

def dragon(n,za,zb):
    """ Cette fonction renvoie la liste des affixes des points formant le dragon, sauf le
        ↪ dernier. (pour éviter les doublons lors des concaténations)."""
    if n==0:
        return [za,zb]
    else:
        zc=calcule_zc(za,zb)
        partie1=dragon(n-1, za,zc)
        partie2=dragon(n-1,zb,zc)
        partie2.reverse()
        return partie1+partie2

def test(n):
    za=0
    zb=1
    traceDepuisC(dragon(n,za,zb))
```