Récursivité

Sommaire

Sommaire

Introduction

Definition : GNU : GNU's not UNIX... (Richard Stallman)

- La récursivité est une notion intervenant en programmation : une fonction (écrite en Python, ou dans un autre langage) est dite implémentée de façon récursive lorsqu'elle s'appelle elle même.
- La récursivité d'une fonction peut découler
  - de la nature de la fonction elle-même, par exemple si elle est donnée à l'aide d'une définition par récurrence [1]
  - d'un choix délibéré du style de programmation, parfois lié au paradigme de programmation choisi ou au langage utilisé [2]
- L'écriture de fonctions récursives suppose de bien comprendre les mécanismes d'appels imbriqués de fonction, en particulier la notion de *pile mémoire* et d'*arbre d'appels*

1. il y a un lien étroit entre la notion mathématique de propriété définie par récurrence et celle de récursivité en programmation.
2. Une présentation succinte de la programmation fonctionnelle sera faite plus tard.

- D'un point de vue algorithmique, l'alternative à l'écriture récursive d'un algorithme est l'écriture *itérative*, à l'aide de boucles
- Il existe des règles permettant d'écrire une fonction récursive correcte, qui se termine.
- Dans le programme de terminale, la notion de récursivité interviendra explicitement dans les thèmes :
  - diviser pour régner ;
  - programmation dynamique.

Sommaire

Schéma d'exécution des appels imbriqués

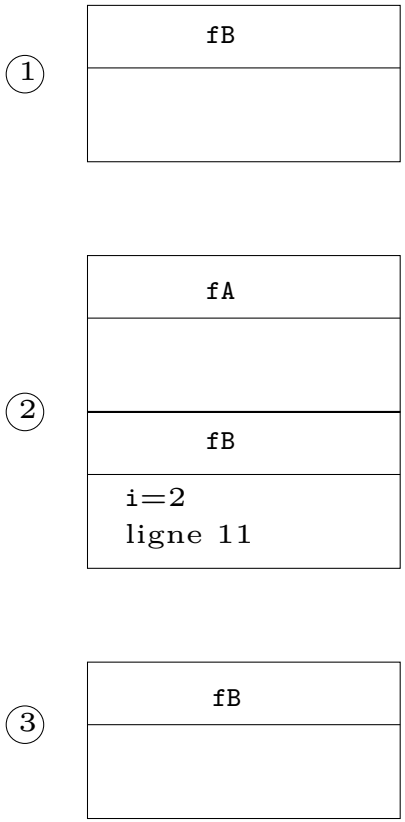
On considère le code-source Python ci-dessous

```
1   def fA():
2       print("Début fA")
3       i = 0
4       while i < 3:
5           print(f"fA {i}")
6           i = i + 1
7       print("Fin fA")
8
9   def fB():
10      print("Début fB")
11      i = 0
12      while i < 5:
13          print(f"fB {i}")
14          i = i + 1
15          if i == 2:
16              fA()
17      print("Fin fB")
18
19  fB()
```

Ex. 1 Analyser ce code puis prédire l'affichage qu'il produit. Vérifier en le testant.

- On remarque que l'exécution du code de `fA` interrompt celui de `fB`. Cependant, quand `fA` se termine, `fB` reprend son exécution au point où elle s'était arrêtée. Ceci est possible grâce au mécanisme d'*empilement de contexte*.
- au moment de l'appel de `fA`, le système empile en mémoire les informations concernant l'exécution interrompue de `fB`, en particulier l'état des variables (`i=2`) et le numéro de la ligne où l'exécution devra reprendre (ligne 12). Le contexte d'exécution de `fA` est alors empilé «au-dessus» de celui ce `fB`.
- Quand `fA` se termine, son contexte est dépilé et le contexte actif redevient celui de `fB`, dans lequel le système retrouve la valeur à donner à `i` et la ligne à exécuter.

Schéma d'exécution des appels imbriqués (suite)



1. exécution de `fB` jusqu'à son interruption par `fA`
2. empilement du contexte de `fB` au moment de son interruption et exécution de `fA`
3. dépilement du contexte de `fA` en fin d'exécution et reprise de l'exécution de `fB` à l'aide de son contexte

Schéma d'exécution des appels imbriqués (suite)

Ex. 2 En tenant compte de ce qui vient d'être dit, analyser le programme ci-dessous et prévoir le résultat qu'il produit.

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

fonct(3)
```

On présente maintenant une fonction pouvant être implémentée récursivement et répondant à un problème concret.

Soit la fonction `somme(n)` définie pour tout entier $n \geqslant 0$ et qui renvoie le nombre

$$S_n = 0 + 1 + 2 + \cdots + n = \sum_{k=0}^{n} k$$

Sommaire

Exemples d'implémentations récursives

Un solution classique et itérative est :

```
def somme1(n):
    S = 0
    for k in range(n+1):  # itération
        S = S + k
    return S
```

Dans cette version, le lien entre la définition mathématique de S_n et le code de la fonction `somme1` n'est pas immédiat.

On peut aussi donner une définition de S_n par récurrence :

$$\begin{cases} S_0 = 0 \\ S_{n+1} = S_n + (n+1), \quad \text{pour tout entier } n \geqslant 0 \end{cases}$$

On en déduit la solution suivante, récursive :

```
def somme2(n):
    if n == 0:
        return 0
    else:
        return n + somme2(n-1) # appel récursif
```

Exemples d'implémentations récursives (suite)

Ex. 3

1. Implémenter puis tester ces deux fonctions afin de vérifier qu'elles produisent les mêmes résultats.
2. Rappeler l'expression de S_n en fonction de n (cf. cours de Première sur les suites arithmétiques). En déduire un jeu d'essais pour les fonctions `somme1` et `somme2`.

- Dans le code de `somme2`, on note la présence d'une instruction conditionnelle qui gère plusieurs *cas* :
  - on appelle *cas de base* les cas d'appels où on peut facilement trouver le résultat (ici il y en a un seul, pour $n = 0$)
  - les *cas récursifs* sont ceux où on fait un appel de la fonction (donc ici si $n \neq 0$)
- on remarque que l'implémentation récursive est très proche de la définition par récurrence de S_n

Exemples d'implémentations récursives (suite)

Il existe plusieurs formes de récursivité, correspondant à plusieurs formes de définitions par récurrence, par exemple selon le nombre de cas de base ou la forme des appels récursifs. Nous en présentons maintenant quelques uns (d'autres seront proposés en exercice).

La suite de Fibonacci (u_n) permet de modéliser d'assez nombreux phénomènes naturels[3]. Elle est définie sur $\mathbb{N}$ par

$$\begin{cases} u_0 = 0, u_1 = 1 \\ u_{n+2} = u_{n+1} + u_n \quad \text{pour tout } n \in \mathbb{N} \end{cases}$$

Une implémentation récursive de (u_n) est

```
def fibo(n):
    if n == 0 or n == 1:  # deux cas de base
        return n
    else:
        return fibo(n-1) + fibo(n-2)  # récursion «double»
```

3. en particulier certaines propriétés liées à la croissance de certains végétaux. Voir par exemple ce lien et celui-là.

Exemples d'implémentations récursives (suite)

Un calcul approché[4] du nombre π, repose sur deux suites (a_n) et (b_n), convergeant vers π, et définies récursivement comme suit :

$$\begin{cases} a_0 = 2\sqrt{3},\ b_0 = 3 \\ a_{n+1} = \dfrac{2a_n b_n}{a_n + b_n},\quad b_{n+1} = \sqrt{a_{n+1} b_n} \quad \text{pour tout } n \in \mathbb{N} \end{cases}$$

On peut implémenter récursivement le calcul de (a_n) comme ci-dessous

```
from math import sqrt

def a(n):
    if n == 0:  # cas de base
        return 2*sqrt(3)
    else:
        return (2*a(n-1)*b(n-1))/(a(n-1)+b(n-1)) # récursion mutuelle
def b(n):  # cas de base
    if n == 0:
        return 3
    else:
        return sqrt(a(n)*b(n-1)) # récursion mutuelle
```

4. celui qu'Archimède (287-212 avant J.C.) a initialement imaginé à l'aide d'une suite de polygones inscrits et exinscrits à un cercle.

Sommaire

Règles d'écriture de fonctions récursives

Pour qu'une fonction, définie sur un certain domaine de définition et implémentée récursivement, soit correcte il faut s'assurer :

1. que l'on finit par tomber sur un cas de base
2. que les valeurs d'appel de la fonction sont toujours dans son domaine
3. que la fonction est bien définie pour toutes les valeurs de son domaine

Ex. 4 On donne ci-dessous 3 fonctions écrites récursivement, supposées définies sur $\mathbb{N}$. Chacune ne respecte pas une des 3 règles précédentes. Quelle fonction ne respecte pas quelle règle ?

```
def f1(n):
  if n == 0:
    return 1
  else:
    return n + f1(n-2)
```

```
def f2(n):
  if n == 0:
    return 1
  else:
    return n + f2(n+1)
```

```
def f3(n):
  if n == 0:
    return 1
  elif n > 1:
    return n + f3(n-1)
```

Règles d'écriture de fonctions récursives (suite)

Le mécanisme d'exécution d'une fonction récursive par empilement des contextes en mémoire peut vite nécessiter une taille mémoire excédant celle disponible sur l'ordinateur exécutant le programme. On a dans ce cas, à l'exécution, une erreur de dépassement de capacité mémoire.

Ex. 5 Tester le programme suivant et traduire le message qui est renvoyé par le système.

```
def f():
    print("Hello")
    f()
f()
```

Il est possible en Python, de modifier le nombre maximal (par défaut de 1000) d'appels récursifs :

```
import sys
sys.recursionlimit(2000)
```

Cependant, on peut surtout modifier la définition récursive pour la rendre plus efficace (voir les exercices qui suivent). Il faut aussi savoir que certains langages (fonctionnels, comme par exemple scheme), sont conçus pour palier à ces problèmes de débordement de pile.

Sommaire

Exercices

- Ex. 6 Donner une définition par récurrence de la fonction factorielle $n!$, définie pour tout $n \in \mathbb{N}$ par $n! = 1 \times 2 \times \cdots \times n$ si $n > 0$ et $0! = 1$. En déduire une implémentation récursive de cette fonction et la tester.
- Ex. 7 Voici un programme qui dessine une figure à l'aide du module `turtle`

```
from turtle import *
def dessin(taille):
    up()
    goto(-taille//2, -taille)
    down()
    forward(taille)
    up()
    goto(-taille//2, taille)
    down()
    forward(taille)
def trace(taille):
    if taille > 0:
        dessin(taille)
        trace(taille-20)

trace(100)
input("Taper sur 'Entrée' pour
↪  continuer")
```

1. Aller sur le WikiBook du module `turtle` pour se familiariser (si nécessaire) avec ses commandes.
2. En déduire, sans l'exécuter et en l'analysant, ce que dessine le programme ci-contre. Vérifier en le testant.

Exercices (suite)

Ex. 8 Soit la suite (u_n) définie par récurrence sur $\mathbb{N}$ par

$$\begin{cases} u_0 = 0 & \text{si } n = 0, \\ u_{n+1} = 3u_n + 2 & \text{si } n \in \mathbb{N}. \end{cases}$$

Écrire une fonction `u(n)` qui reçoit un entier naturel `n` et qui calcule récursivement et renvoie le n$^{\text{e}}$ terme de la suite (u_n).

Ex. 9 Soit (u_n) la suite d'entiers définie par

$$u_{n+1} = \begin{cases} u_n/2 & \text{si } u_n \text{ est pair}, \\ 3 \times u_n + 1 & \text{sinon}. \end{cases}$$

Le terme u_0 est un entier quelconque supérieur ou égal à 1.

1. Calculer à la main les 10 premiers termes de la suite pour $u_0 = 2$, $u_0 = 3$ et $u_0 = 4$.

Exercices (suite)

2. Quelle conjecture peut-on faire ? Cette conjecture, appelée conjecture de Syracuse[5], défie toujours les mathématiciens.
3. Écrire une fonction récursive `syracuse(u_0)` qui affiche les valeurs successives de la suite (u_n) définie par la valeur initiale u_0, tant que u_n est différent de 1.
4. Question bonus : écrire une version itérative de la même fonction `syracuse`

Ex. 10 Écrire une fonction récursive `pgcd(a,b)` qui renvoie le PGCD de deux entiers `a` et `b`. Rappel : pour tout entiers a et b avec $a \geqslant b$ on peut utiliser pour construire récursivement $\mathrm{PGCD}(a, b)$, au choix, l'une des deux relations suivantes :

$$\mathrm{PGCD}(a, b) = \mathrm{PGCD}(a - b, b)$$

$\mathrm{PGCD}(a, b) = \mathrm{PGCD}(b, r)$ où r est le reste de la division euclidienne de a par b.

5. du nom d'une université aux États-Unis, où elle a été beaucoup étudiée

Exercices (suite)

- Ex. 11 Écrire une fonction récursive `nombre_de_chiffres(n)` qui prend un entier positif ou nul `n` en argument et renvoie son nombre de chiffres. Par exemple `nombre_de_chiffres(34126)` doit renvoyer 5. Bonus : écrire une version itérative de la même fonction `nombre_de_chiffres`.

- Ex. 12 On s'intéresse à une écriture récursive de la fonction puissance, qui à x associe x^n, où x est un réel non nul et n un entier naturel. On rappelle que, par définition, pour $n \geqslant 1$ on a

$$x^n = \underbrace{x \times x \times \cdots \times x}_{n \text{ facteurs}}$$

et, pour $n = 0$, $x^0 = 1$. Une écriture récursive du cas $n \geqslant 1$ est

$$x^n = x \times \underbrace{x \times x \times \cdots \times x}_{(n-1) \text{ facteurs}} = x \times x^{n-1}.$$

Si on note $\text{puissance1}(x, n)$ cette première implémentation récursive de la fonction puissance, alors la définition précédente s'écrit mathématiquement, pour $n \geqslant 1$,

$$\text{puissance1}(x, n) = x \times \text{puissance1}(x, n-1)$$

Exercices (suite)

1. Déduire de ce qui précède une écriture récursive en Python de la fonction $\mathrm{puissance1}(x, n)$. Tester cette fonction sur quelques exemples.

Le nombre d'appels récursifs de la fonction $\mathrm{puissance1}$ dépend linéairement de son argument n (plus précisément : l'appel de $\mathrm{puissance1}(x, n)$ déclenche $n + 1$ appels récursifs). On peut, pour réduire ce nombre d'appels, utiliser une définition alternative de la fonction puissance :

$$x^n = \begin{cases} 1 & \text{si } n = 0, \\ \left(x^{n/2}\right)^2 & \text{si } n \text{ est pair}, \\ x \times \left(x^{(n-1)/2}\right)^2 & \text{si } n \text{ est impair}. \end{cases}$$

Par exemple, le calcul de x^{10} est maintenant réalisé comme suit :

$$x^{10} = \left(x^5\right)^2 = \left(x \times \left((x \times (x^0))^2\right)^2\right)^2.$$

Exercices (suite)

En notant $\text{puissance2}(x, n)$ l'implémentation récursive de la fonction puissance associée à cette nouvelle définition, on a, pour $n \geqslant 1$, les égalités suivantes :

- Si n est pair : $\text{puissance2}(x, n) = (\text{puissance2}(x, n/2))^2$
- Si n est impair : $\text{puissance2}(x, n) = x \times (\text{puissance2}(x, (n-1)/2))^2$

2. Combien d'appels récursifs sont générés par $\text{puissance2}(2, 10)$?
3. Implémenter en Python la fonction récursive $\text{puissance2}(x, n)$. Vérifier son bon fonctionnement sur un jeu d'essais.
4. Combien d'appels récursifs sont générés par $\text{puissance2}(2, 1000)$? Bonus : quelle formule donne ce nombre d'appels en fonction de n ?

Exercices (suite)

Ex. 13 Écrire une fonction récursive `somme` qui prend en paramètre une liste de nombres et renvoie la somme des termes de cette liste. Par exemple `somme([4, 7, 2])` doit renvoyer `13`.

Ex. 14 Expliquer quel est le résultat renvoyé par la fonction `mystere` dont le code est donné ci-dessous :

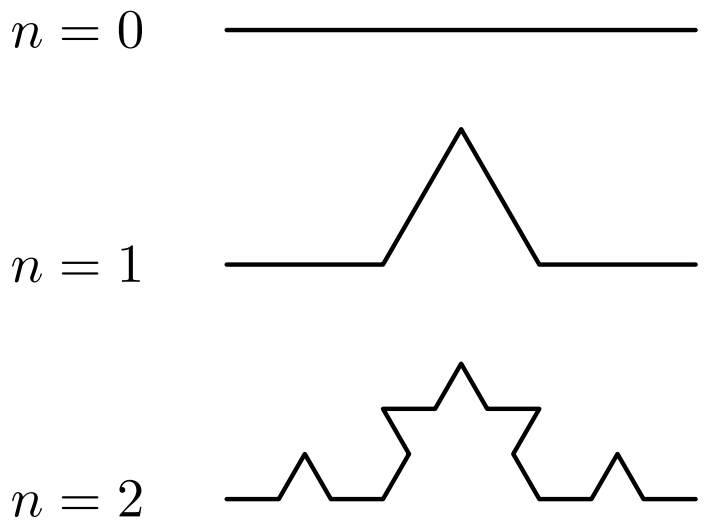
```
def mystere(n):
    if n < 2:
        return str(n)
    else:
        return mystere(n//2) + str(n%2)
```

Ex. 15 Écrire une fonction récursive `inverse(ch)` qui prend en paramètre une chaîne de caractères `ch` et renvoie la chaîne obtenue en inversant l'ordre des caractères. Par exemple, `inverse("azerty")` a pour valeur `"ytreza"`.

Exercices (suite)

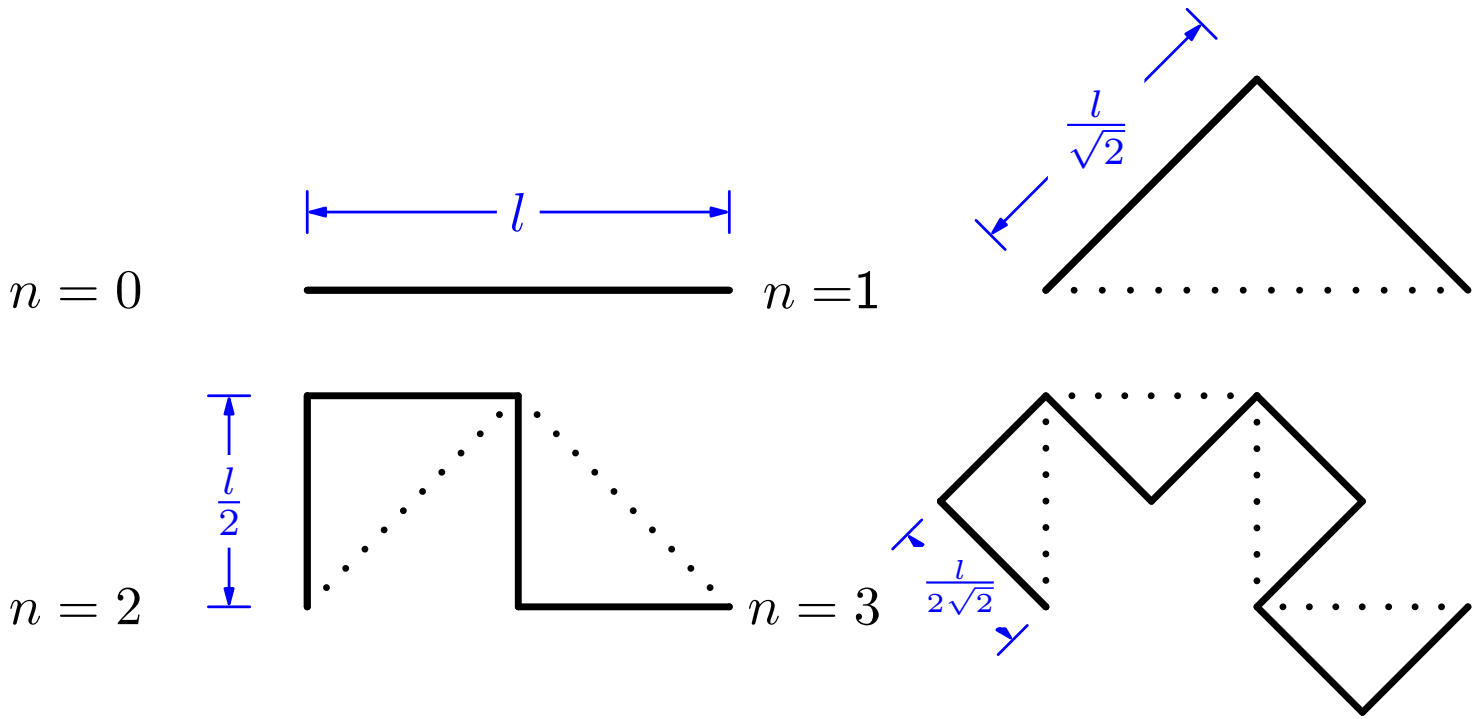
Ex. 16 La courbe de Von-Koch est une courbe fractale définie récursivement. Le cas de base, à l'ordre 0, et les cas à l'ordre $n = 1$ et 2 sont donnés sur la figure ci-dessous. On voit que le passage d'un ordre au suivant se fait en réappliquant toujours la même construction sur chaque segment, d'où la récursivité de la courbe. En faisant tendre n vers $+\infty$ la courbe limite obtenue est la courbe de Von-Koch.

1. Reproduire sur votre feuille les figures correspondant à l'ordre 0, 1 et 2 et construire celle associée à l'ordre 3.
2. Écrire une fonction nommée `koch(n, l)` qui dessine, à l'aide du module `turtle`, le flocon de Von-Koch à l'ordre `n` en partant d'un segment de longueur `l` pixels.



Exercices (suite)

Ex. 17 La courbe de Heighway est une courbe fractale définie récursivement en partant d'un segment. La figure ci-dessous montre les courbes obtenues aux ordres 0, 1, 2 et 3. La courbe limite, obtenue en faisant tendre n vers $+\infty$, est celle du dragon de Heighway [6]. Sur la figure, tous les segments sont de même longueur (pour un ordre n donné) et tous les angles entres deux segments consécutifs sont droits.

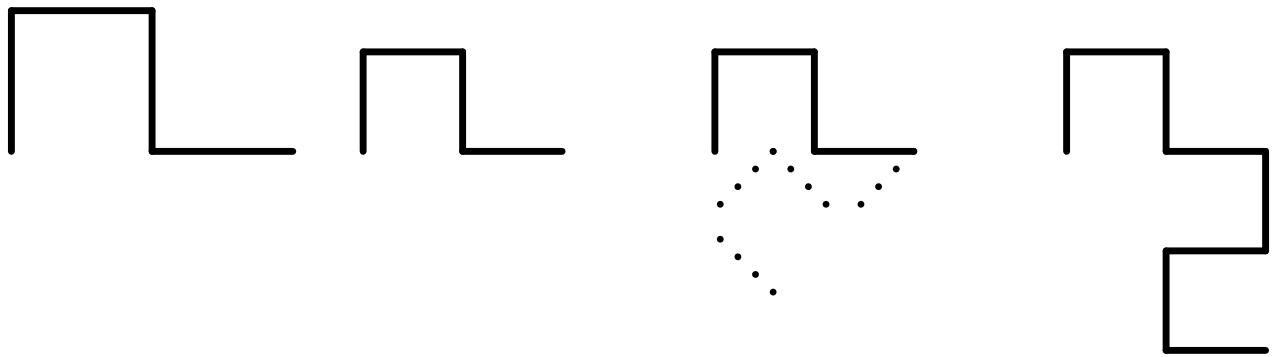


6. John Heighway est un physicien américain qui a découvert cette courbe en 1966 alors qu'il travaillait pour la NASA, cf. ce lien pour plus d'informations.

1. Reproduire sur votre feuille les courbes aux ordres 0, 1, 2 et 3 et tracer la courbe à l'ordre 4.

On remarque, comme le montre la figure ci-dessous pour le passage de $n = 2$ à $n = 3$, que chaque courbe peut se déduire de la précédente par :

- une réduction d'un facteur $\frac{1}{\sqrt{2}}$
- une duplication suivie d'une rotation de $+90$ °autour du dernier point de la courbe [7]



7. suivie d'une rotation de $+45$ °de la courbe complète obtenue autour du point initial, si on veut obtenir exactement les courbes de la figure précédente.

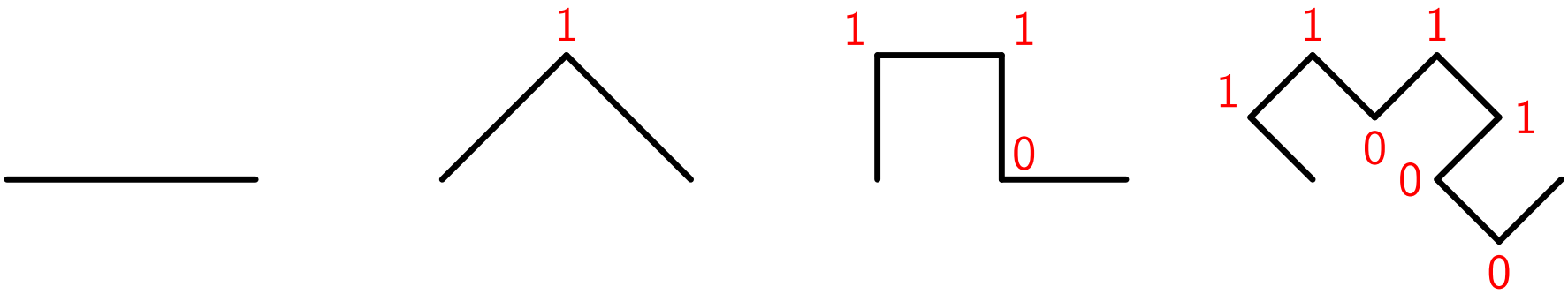
Exercices (suite)

En partant de cette définition récursive de la courbe de Heighway, on remarque qu'on peut coder les courbes en donnant pour chacune d'elles :

- la longueur l_n de chaque segment : $l_n = \frac{l}{\sqrt{2}^n}$,
- la liste L_n des changements de direction (de $+$ ou $-90\,°$, codés respectivement par $+1$ et 0) à effectuer entre chaque segment.

La valeur de l_n en fonction de n étant trouvée, il reste, pour générer la courbe d'ordre n, à trouver comment générer L_n. Par exemple, voici les codages successifs des Liste L_n pour n allant de 0 à 3 :

$$L_0 = [] \quad L_1 = [1] \quad L_2 = [1, 1, 0] \quad L_3 = [1, 1, 0, 1, 1, 0, 0]$$



Exercices (suite)

En fait, on voit assez facilement que L_n peut se déduire de L_{n-1} par concaténation :

- de L_{n-1},
- de la liste $[1]$ (qui code une rotation de $+90$ °entre les deux listes, l'originale et celle dupliquée),
- de la liste à l'ordre L_{n-1} dont on doit réaliser l'«image en miroir», puisqu'elle est parcourue dans l'ordre inverse, et la complémentation (c'est à dire l'échange des 1 et des 0, les rotation de $+90$ °devenant des rotations de -90 °et réciproquement), puisque les segments sont parcourus en sens inverse.

2. En utilisant ce qui précède, trouver la liste correspondant à la courbe de Heighway à l'ordre 4.

Exercices (suite)

3. Écrire une fonction `chaine(n)`, implémentée récursivement, définie pour `n` entier naturel, qui renvoie la liste des changements de direction de la courbe de Heighway d'ordre n. Autrement dit :

   `chaine(0)` renvoie `[]`
   `chaine(1)` renvoie `[1]`
   `chaine(2)` renvoie `[110]`
   `chaine(3)` renvoie `[1101100]`

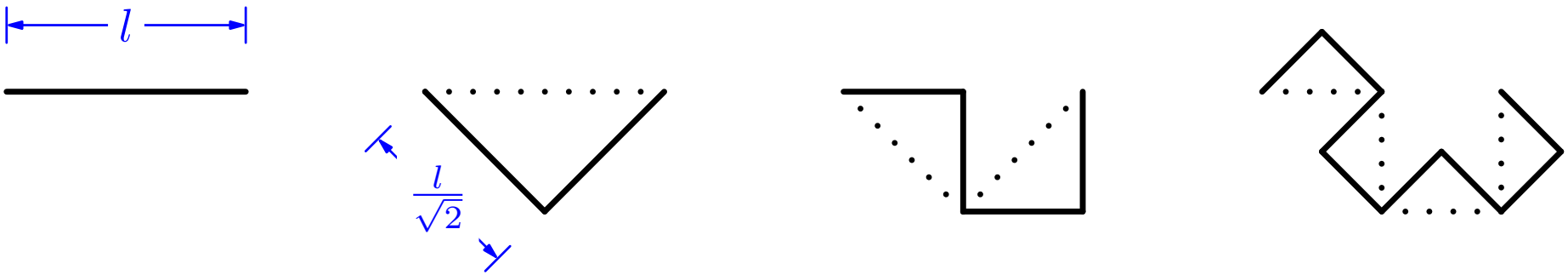
   Rq : on pourra utiliser la possibilité de renverser en Python les éléments d'une liste. Plusieurs solutions : cf. ce lien ou celui-là. De même, pour réaliser une opération élément par élément sur une une liste, on peut utiliser les listes définies en compréhension, voir ici.

4. Écrire une fonction `heighway(n, l)` qui trace la courbe de Heighway d'ordre `n` à partir d'un segment de longueur `l` (pixels). On utilisera la fonction `chaine(n)` de la question précédente puis le module `turtle` pour tracer (itérativement) la succession des segments qui composent la courbe d'ordre `n`.

   Rq : on aura intérêt à régler la vitesse d'animation de sorte que les mouvements soient les plus rapides possible, à l'aide de la commande `speed(0)`.

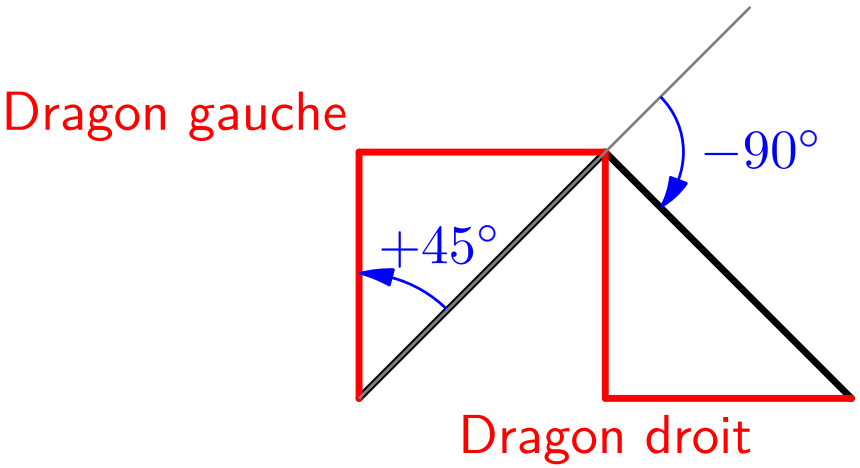
Exercices (suite)

Ex. 18 On propose dans cet exercice une autre génération récursive de la courbe du dragon de Heighway, plus géométrique. On remarque que la courbe peut être définie récursivement à partir de *deux* courbes fractales, la courbe du dragon gauche et celle du dragon droite. La première est celle qui a déjà été présentée (revoir cette figure). La courbe droite se construit comme le montre la figure suivante :

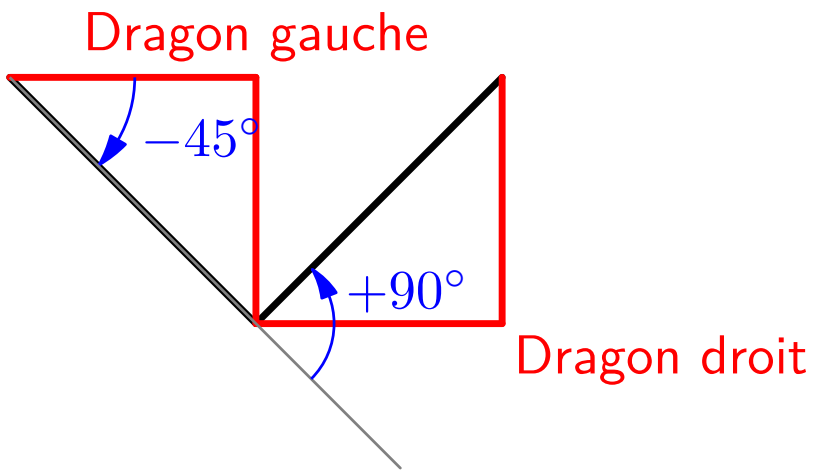


La récursion mutuelle qui définit chaque courbe, gauche et droite, est illustrée sur la figure page suivante.

Exercices (suite)



Dragon droit

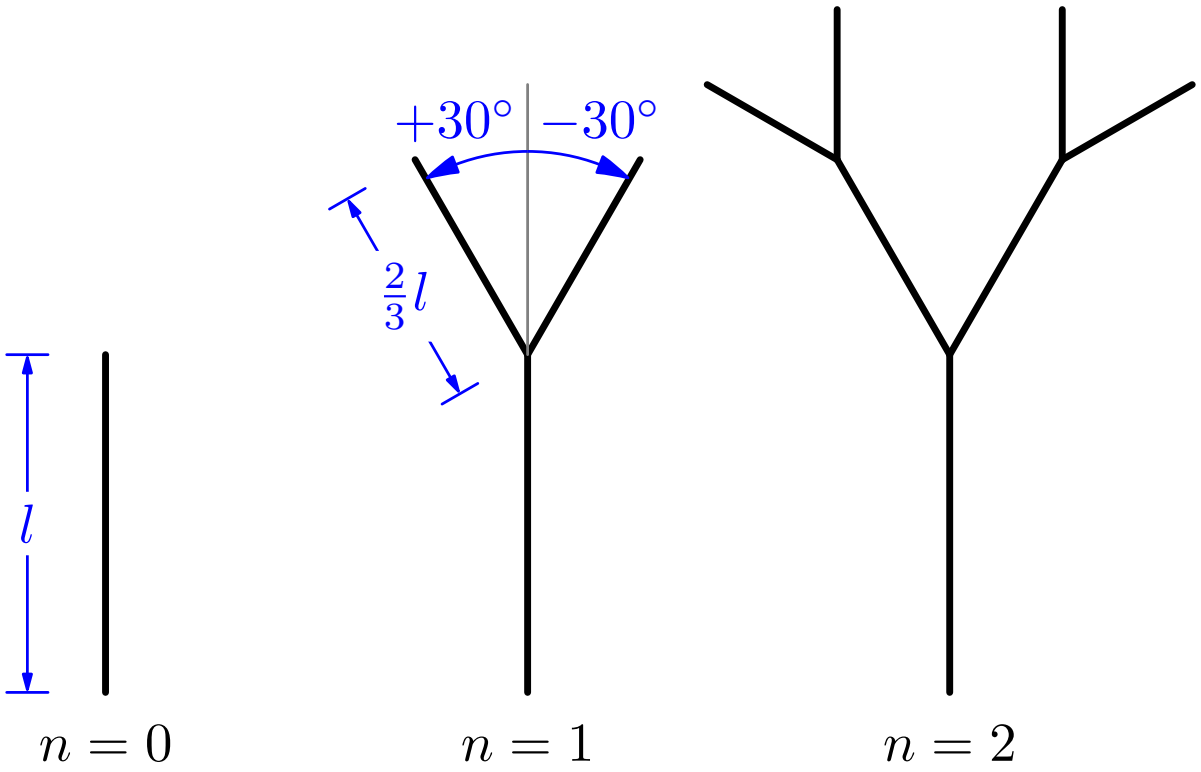


Dragon gauche

1. Déduire de ce qui précède une définition récursive en pseudo-code de 2 fonctions, `heighwayG(n, l)` et `heighwayD(n, l)`, dessinant respectivement le dragon de heighway gauche et droite. On suppose que les fonctions de dessin fournies par le module `turtle` sont accessibles à ces fonctions.

2. Implémenter puis tester la fonction heighwayG en Python à l'aide du module `turtle`.

Ex. 19 De nombreux objets présents dans la nature ont une structure qui peut être modélisée par une construction fractale. On montre ci-dessous comment construire un modèle d'arbre. Seules les trois premières étapes sont montrées. Dans le cas étudié, on voit que d'une étape à la suivante chaque nouvelle branche crée deux nouvelles branches, inclinées de part et d'autre de 30 °et de taille réduite d'un certain facteur (qui vaut ici $\frac{2}{3}$).



1. Écrire une fonction récursive, `arbre(n, l)`, qui trace, à l'aide du module turtle, une arbre fractal d'ordre n dont le tronc est de longueur `l` (en pixels). Pour rendre le tracé plus réaliste, on pourra régler la taille du trait[8] de sorte que les branches soient de plus en plus fines quand on va vers les extrémités des branches.
2. Vous pouvez créer de nombreuses variantes de l'arbre, en le rendant par exemple dissymétrique (angles et/ou coefficients de réduction différents pour les branches de gauche et droite...)

8. avec `pensize()`.