Corrigé de la feuille d'exercices 5
Programmation récursive

Exercice 1

```
def permutations(S):
    if len(S) <= 1:              # Cas terminal
        return [S]
    result = []
    for i,e in enumerate(S):
        Se = S[:i]+S[i+1:]       # Chaine privee de e
        result.append([e+p for p in permutations(Se)])
    return result
```

Remarque. L'instruction itérative :

```
for i, e in enumerate(S):
```

est équivalente à :

```
for i in range(len(S)):
   e = S[i]
```

Exemples :

```
In [2]: permutations('123')
Out[2]: ['123', '132', '213', '231', '312', '321']

In [3]: permutations('1234')
Out[3]:
['1234', '1243', '1324', '1342', '1423', '1432', '2134',
 '2143', '2314', '2341', '2413', '2431', '3124', '3142',
 '3214', '3241', '3412', '3421', '4123', '4132', '4213',
 '4231', '4312', '4321']
```

On obtient la liste des $24 = 4!$ permutations de la chaine `'1234'`.

Exercice 2

1. Version récursive de l'algorithme de Horner.

```
def horner(P,a,i=0):
    n = len(P)-1
    if i == n:
        return P[n]
    else:
        return P[i]+a*horner(P,a,i+1)
```

qu'on appellera par l'instruction `horner(P,a)`.

2. Version récursive de l'algorithme de Horner.

```
def horner(P,a):
    n = len(P)-1
    v = P[n]
    for i in reversed(range(n)):
        v = a*v + P[i]
    return v
```

3. Dans la première version on a la relation de récurrence :

$$C(0) = \Theta(1) \quad \text{et} \quad C(n+1) = C(n) + \Theta(1)$$

où $C(n)$ désigne le nombre d'opérations élémentaires pour un paramètre n. On en déduit $C(n) = \Theta(n)$.

Dans la deuxième version on a une boucle `for` s'itérant n fois et n'effectuant que $\Theta(1)$ opérations. Ainsi encore une complexité linéaire $\Theta(n)$.

La complexité temporelle est linéaire pour les 2 versions. La complexité spatiale est linéaire pour la version récursive et bornée pour la version itérative.

Exercice 3

La somme partielle $\sum_{n=0}^{N} \frac{t_n}{2^{n+1}}$ est une valeur approchée par défaut à $2^{-(N+1)}$ près de la limite de la série. Aussi pour une valeur approchée à 2^{-n} près il suffit de calculer la somme partielle jusqu'en $n-1$.

```
from math import log

# Pre et post-traitement
def constantePTM(n):
    L = [0]
    C = constante_rec(L,n-1)     # Calcul recursif
    S = 0                        # Calcul de la somme partielle
    for k in range(n):
        S += C[k]*2**(-k-1)
    return S

# Partie recursive : calcul après N iterations
def constante_rec(L,N):
    if N == 0:
        return L
    newL = []
    for x in L:
        if x == 0:
            newL.extend([0,1])
        else:
            newL.extend([1,0])
    return constante_rec(newL,N-1)
```

Exemple : le nombre est situé entre 0,25 et 0,5. La meilleure précision que l'on puisse atteindre en flottant 64 bits est 2^{-54}. Avec 15 chiffres significatifs corrects :

```
In [2]: 2**-54
Out[2]: 5.551115123125783e-17

In [3]: constantePTM(53)
Out[3]: 0.4124540336401076
```

Exercice 4

1.a) On reprend le code donné au paragraphe 2.5.4.

```
from turtle import *

def vonkoch(longueur,n):
    if n == 1:
        forward(longueur)
    else:
        l = longueur / 3
        vonkoch(l, n - 1)
        left(60)
        vonkoch(l, n - 1)
        right(120)
        vonkoch(l, n - 1)
        left(60)
        vonkoch(l, n - 1)
```

1.b) Il suffit de tracer des courbes de Von Koch sur un carré, pour la première :

```
def floconVonKochCarre(longueur, n):
    pen(speed = 0)
    hideturtle()
    up()
    goto(-longueur/2, longueur/2)
    down()
```

```
    for i in range(4):
        vonkoch(longueur,n)
        right(90)
```

et sur un hexagone, pour la deuxième :

```
def floconVonKochHexa(longueur, n):
    pen(speed = 0)
    hideturtle()
    up()
    goto(-longueur/2, longueur/2)
    down()
    for i in range(6):
        vonkoch(longueur,n)
        right(60)
```

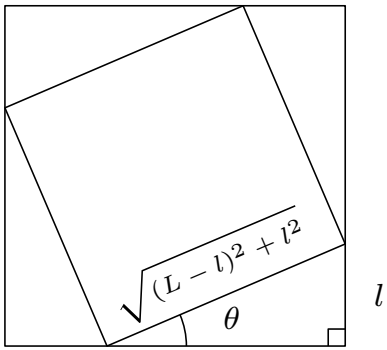
2.a) Fonction `carres(L,n)` : elle trace le carré de longueur de côté L ; puis avance de $L/2$, tourne de 45°, et appelle récursivement `carres(.,.)` avec pour paramètres $L/\sqrt{2}$ et $n-1$.

La condition terminale survient pour $n = 0$, où l'appel renvoie `None`.

```
from turtle import *

def carres(L,n):
    hideturtle()
    down()
    if n==0 :
        return
    for i in range(4):
        forward(L)
        right(90)
    forward(L/2.)
    right(45)
    carres(L/(2**0.5),n-1)
```

2.b) Déterminons géométriquement les opérations à effectuer avant chaque appel récursif.
Soit l la distance qui sépare un sommet du grand carré à l'un des sommets du carré suivant.



D'après le théorème de Pythagore, le carré suivant a pour côté $\sqrt{(L-l)^2+l^2}$.
L'angle Θ est l'angle au sommet d'un triangle rectangle ; il vérifie : $\tan(\Theta) = l/(L-l)$,
donc $\theta = \arctan(l/(L-l))$. Entre chaque appel récursif il faut donc :

- avancer de l ;
- tourner de l'angle $\arctan(l/(L-l))$;
- changer L en $\sqrt{(L-l)^2+l^2}$;
- changer l en $l \times \sqrt{(L-l)^2+l^2}/L$.

On obtient alors le code pour le tracé.

```
from math import atan, pi

def rad2deg(a):
    return a*180/pi

def Carres(L,l,n):
    hideturtle()
    down()
    if n==0 :
        return
```

```
    for i in range(4):
        forward(L)
        right(90)
    forward(l)
    angle = rad2deg(atan(l/(L-l)))
    L1 = ((L-l)**2+l**2)**0.5
    right(angle)
    L, l = L1, l*L1/L
    Carres(L,l,n-1)
```

2.c) La première fonction est un cas particulier de la seconde avec $l = L/2$. Leur complexité sont donc les mêmes. Il n'y a qu'un seul appel récursif et $\Theta(1)$ opérations élémentaires. On a donc, en notant $C(n)$ le nombre d'opérations élémentaires pour un paramètre n, la relation de récurrence, $C(n+1) = C(n) + \Theta(1)$, d'où une complexité temporelle linéaire, $\Theta(n)$.

La complexité est due à la pile d'exécution ; elle est d'ordre $\Theta(n)$.

Les complexités sont linéaires en temps et en espace.

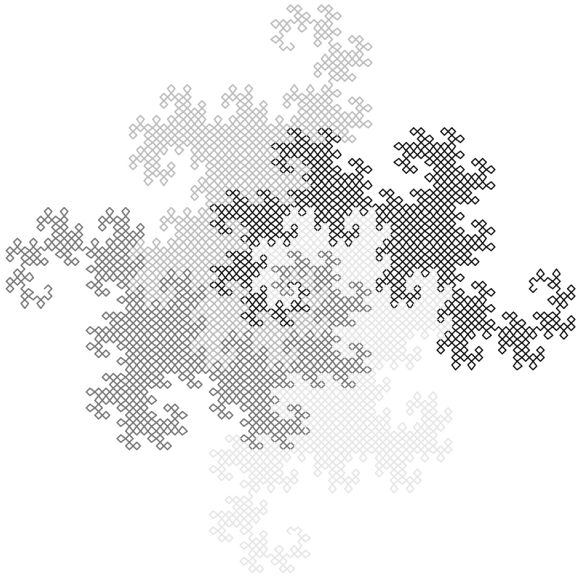
3.a) Tracé de la fractale du dragon.

```
from turtle import *
from math import sqrt

# Fonction dragon_gauche()
def dragon_gauche(L,n):
    pen(speed=0)
    hideturtle()
    if n==0:
        forward(L)
    else:
        left(45)
        dragon_gauche(L/sqrt(2),n-1)
        right(90)
        dragon_droit(L/sqrt(2),n-1)
        left(45)

# Fonction dragon_droit()
def dragon_droit(L,n):
    pen(speed=0)
    hideturtle()
    if n==0:
        forward(L)
    else:
        right(45)
        dragon_gauche(L/sqrt(2),n-1)
        left(90)
        dragon_droit(L/sqrt(2),n-1)
        right(45)
```

3.b) Tracé de l'élément de pavage avec 4 courbes du dragon.



Code :

```
L = 200
n = 11
up(); goto(0,0); down()
color("black")
dragon_gauche(L,n)
up(); goto(0,0); down()
left(90)
color("cyan")
dragon_gauche(L,n)
up(); goto(0,0); down()
left(90)
color("magenta")
dragon_gauche(L,n)
up(); goto(0,0); down()
left(90)
color("yellow")
dragon_gauche(L,n)
```

3.c) Tracé de l'élément de pavage avec 2 courbes du dragon.

Code :

```
L = 200
n = 11
up(); goto(0,0); down()
color("cyan")
dragon_gauche(L,n)
up(); goto(L,0); down()
left(180)
color("magenta")
dragon_gauche(L,n)
```