TD 6 : La récursivité

PC/PC* - Lycée Thiers

Exercice 1 : Version récursive de l'algorithme d'Euclide

On rappelle l'algorithme d'Euclide permettant de retourner le *pgcd* de 2 entiers a et b :

```
Tant que b ≠ 0
    r = reste de la division de a par b
    a = b
    b = r
retourner a
```

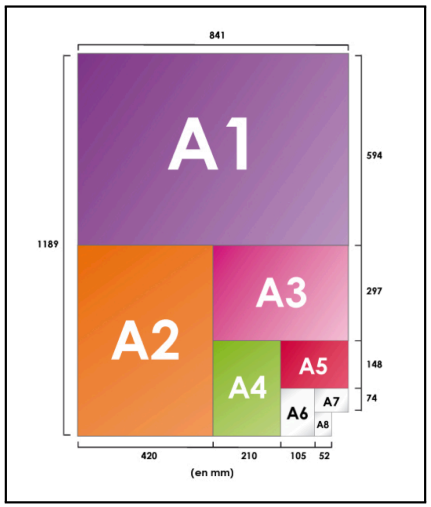
Ecrire une version récursive de l'algorithme d'Euclide.

Exercice 1 : Algorithme d'Euclide

```
# Algorithme d'Euclide, Version récursive
def pgcd(a,b):
    if b == 0:
        return a
    return pgcd(b,a%b)
```

Exercice 2 : Formats de papier

Le format de papier A0 correspond à un rectangle de largeur de 841cm et une longueur de 1189cm. le format A1 est obtenu en coupant en deux parties égales le format A0, il a donc pour longueur la largeur de A0 et pour largeur la moitié de la longueur de A0. Sur le même principe une feuille A1 contient deux feuilles A2, une feuille A2 deux feuilles A3, etc...



Ecrire une fonction récursive prenant en paramètre un entier naturel n et qui retourne longueur et largeur d'une feuille de format An.

Exercice 2 : corrigé

```
# Formats de papier
def A(n):
    if n==0:
        return (118.9, 84.1)
    Long, larg = A(n-1)
    return larg, Long/2
```

Exemple :

```
>>> A(4)
(29.725, 21.025)
```

Exercice 3 : Recherche d'une racine par dichotomie

Soit f une application continue sur une intervalle $[a, b]$ vérifiant $f(a).f(b) \leqslant 0$. D'après le théorème des valeurs intermédiaires f admet une racine sur $[a, b]$. On peut déterminer une valeur approchée de cette racine, par une recherche par dichotomie.

Ecrire une version récursive de la recherche d'une racine par dichotomie. L'appel de `dichotomie(f,a,b,e)` retournera une valeur approchée à e près d'une racine de f sur $[a, b]$.

Exercice 3 : Recherche d'une racine par dichotomie - Correction

```
def dichotomie(f,a,b,e):
    assert f(a)*f(b) <= 0
    if (b-a) <= e:
        return (a+b)/2.
    else:
        m = (a+b)/2.
        if f(a)*f(m) <= 0:
            return dichotomie(f,a,m,e)
        else:
            return dichotomie(f,m,b,e)
```

Exercice 4 : rangées de briques

Ecrire une fonction récursive retournant le nombre de façons différentes de constituer une rangée de longueur n de briques uniquement avec des briques de longueur 1 et 2 ?

Exercice 4 : correction

```
def briques(n):
    if n==0:
        return 0
    if n==1:
        return 1
    if n==2:
        return 2    # Deux façons pour une rangée de lgr 2
    return briques(n-1) + briques(n-2)
```

Deux possibilités mutuellement exclusives :

soit la dernière brique posée est de longueur 1, soit elle est de longueur 2.

Exercice 5 : Coefficients binomiaux

1. Ecrire une fonction récursive qui calcule le coefficient binomial $\binom{n}{k}$ à l'aide de la relation de Pascal :

$$\forall\,(k,n),\in\mathbb{N}^2 \quad \binom{n}{k} = \begin{cases} 0 & \text{si } k > n \\ 1 & \text{si } k = 0 \\ \binom{n-1}{k-1} + \binom{n-1}{k} & \text{si } 0 < k \leqslant n \end{cases}$$

2. L'améliorer en lui passant en paramètre le tableau de taille $(n+1)\times(k+1)$ des coefficients déjà calculés.
3. Comparer leur temps d'éxécution.

Exercice 5 : Coefficients binomiaux - Correction

1)

```
def binom(n,k):
    if k == 0:
        return 1
    if k > n:
        return 0
    return binom(n-1,k-1) + binom(n-1,k)
```

Exercice 5 : Coefficients binomiaux - Correction

```
def binomRec(n,k,T):
    if k<0 or k>n:
        return
    if T[n][k] == None:
        if k==0:
            T[n][k] = 1
        elif k == n:
            T[n][k] = 1
        else:
            T[n][k] = binomRec(n-1,k-1,T) + binomRec(n-1,k,T)
    return T[n][k]

def Coefbinomial(n,k):
    T = [[None]*(k+1) for i in range(n+1)]
    binomRec(n,k,T)
    return T[n][k]
```

Exercice 5 : Coefficients binomiaux - Correction

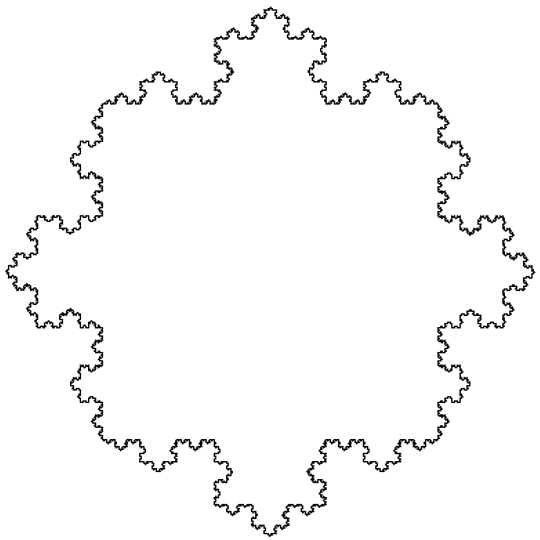
3)

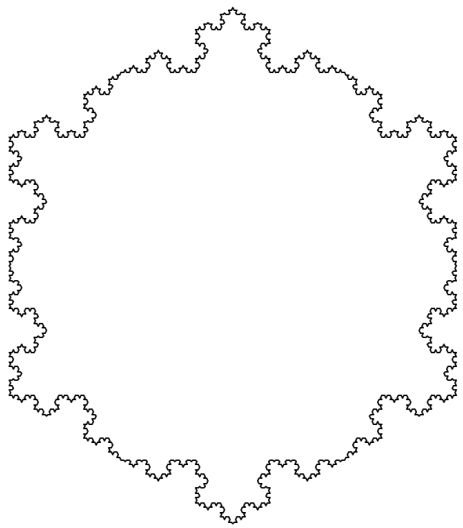
```
In [1]: %timeit binom(20,10)
10 loops, best of 3: 158 ms per loop

In [2]: %timeit Coefbinomial(20,10)
10000 loops, best of 3: 117 µs per loop
```

Par exemple pour $\binom{20}{10}$ la 2ème version est plus de 1000 fois plus rapide que la 1ère!!!

Exercice 6 : Autour du flocon de Von-Koch

1. Déterminer la complexité algorithmique du tracé du flocon de von-koch (en fonction du paramètre `n`).
2. De quel ordre de grandeur serait la durée du calcul pour une profondeur $n = 100$ avec un processeur à 10Ghz ? (on donnera la réponse (approximative) en milliards d'années).
3. En s'en inspirant, sauriez-vous tracer les graphiques suivants ?





Exercice 6 : correction

```
import turtle as tt
def vonkoch(longueur,n):
    if n == 1:
        tt.forward(longueur)
    else:
        l = longueur / 3
        vonkoch(l, n - 1); tt.left(60)
        vonkoch(l, n - 1); tt.right(120)
        vonkoch(l, n - 1); tt.left(60)
        vonkoch(l, n - 1)

def floconVonKoch(longueur, n):
    tt.pen(speed = 0) # Acceleration du mouvement
    tt.hideturtle() # Pour ne pas tracer la tortue
    tt.up()
    tt.goto(-longueur/2, longueur/3) # Depart en haut à gauche
    tt.down()
    for i in range(3):
        vonkoch(longueur,n); tt.right(120)
```

Le programme principal appelle 3 fois la partie récursive avec le paramètre `n`, et effectue en plus un nombre borné d'opérations élémentaires. Sa complexité est donc la même que la partie récursive `vonkoch(.,n)`.

Exercice 6 : correction

```
import turtle as tt
def vonkoch(longueur,n):
    if n == 1:
        tt.forward(longueur)
    else:
        l = longueur / 3
        vonkoch(l, n - 1); tt.left(60)
        vonkoch(l, n - 1); tt.right(120)
        vonkoch(l, n - 1); tt.left(60)
        vonkoch(l, n - 1)
```

Appelons $C(n)$ le nombre d'opérations effectuées par la partie récursive `vonkoch(.,n)`.

On a la relation de récurrence : $\forall k \in \mathbb{N}, \quad C(k+1) = 4.C(k) + O(1)$

2) L'ordre de cette suite est le même que celui d'une géométrique de raison 4. Donc la complexité (en temps) du tracé est exponentielle, d'ordre $\Theta(4^n)$.
(La complexité en espace est encore linéaire : pile d'exécution de taille $\leqslant n$).

Pour $n = 100$: $4^{100} \approx 1,6.10^{60}$. Soit $1,6.10^{50}s$ avec un processeur cadencé à 10Ghz : Soit $1,6.10^{50}/(24 \times 3600 \times 365)$ années : $\approx 5.10^{42}$ années $\approx 5.10^{33}$ milliards d'années...

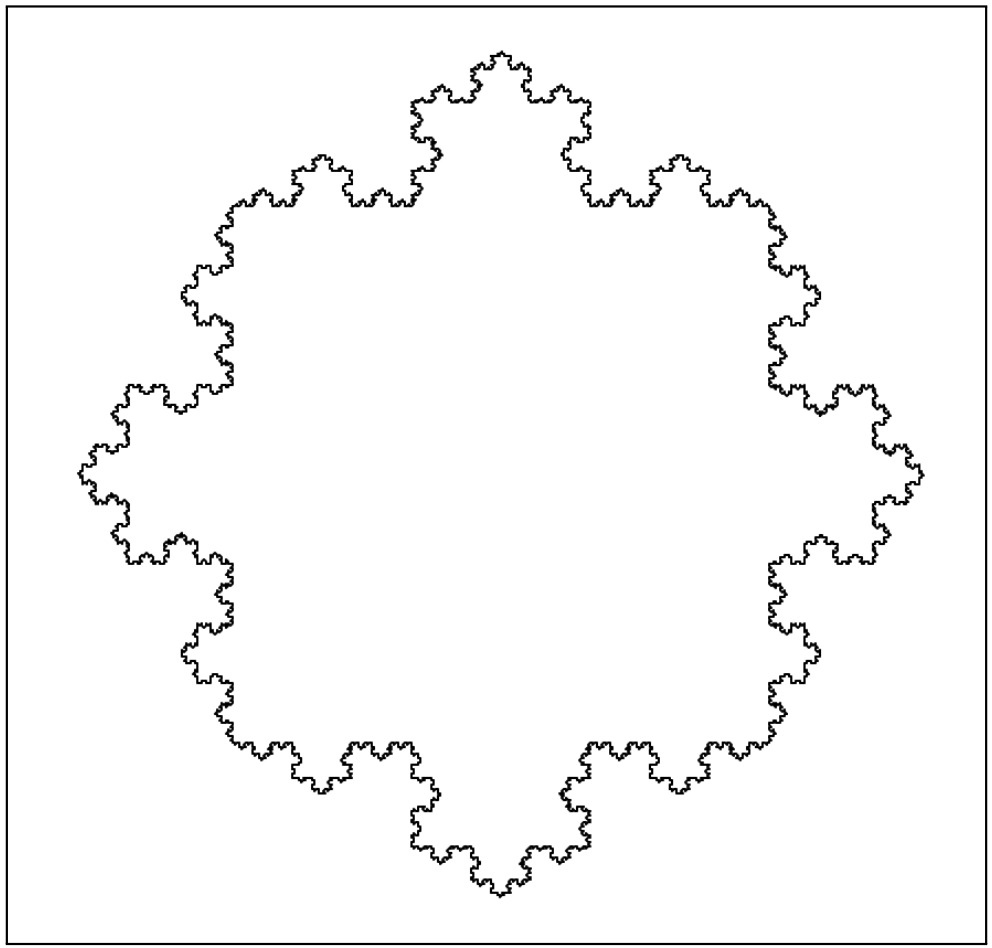
Personne n'en verra le résultat même si les processeurs atteignaient des performances des milliards de milliards de milliards (10^{27}) de fois meilleures qu'aujourd'hui. !

Exercice 6 - Correction

2) Pour le premier (on construit des courbes de Von-Koch sur un carré) :

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

def floconVonKochCarre(longueur, n):
    tt.pen(speed = 0)
    tt.hideturtle()
    tt.up()
    tt.goto(-longueur/2, longueur/2)
    tt.down()
    for i in range(4):
        vonkoch(longueur,n); tt.right(90)
```

Exercice 6 - Correction



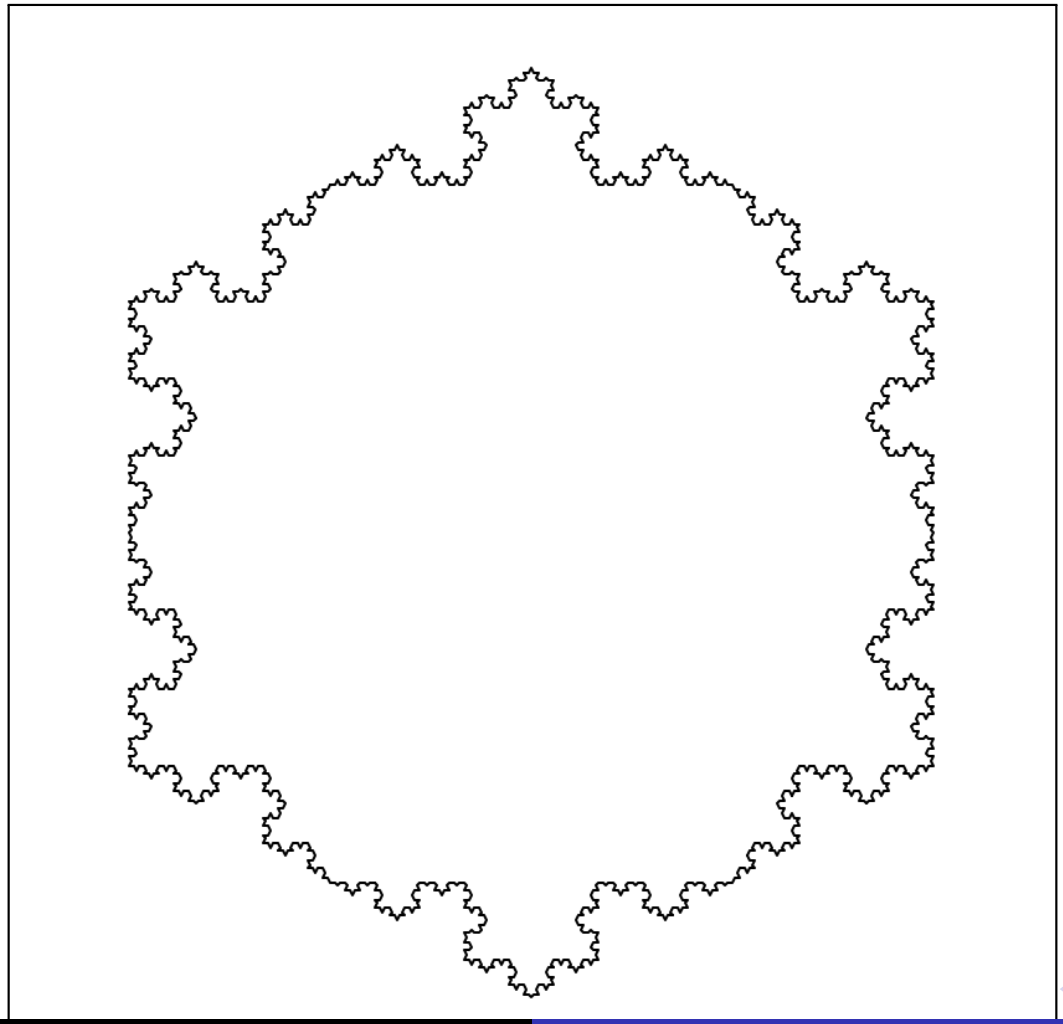
Exercice 6 - Correction

- Pour le deuxième (courbes de Von-Koch sur un hexagone) :

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

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

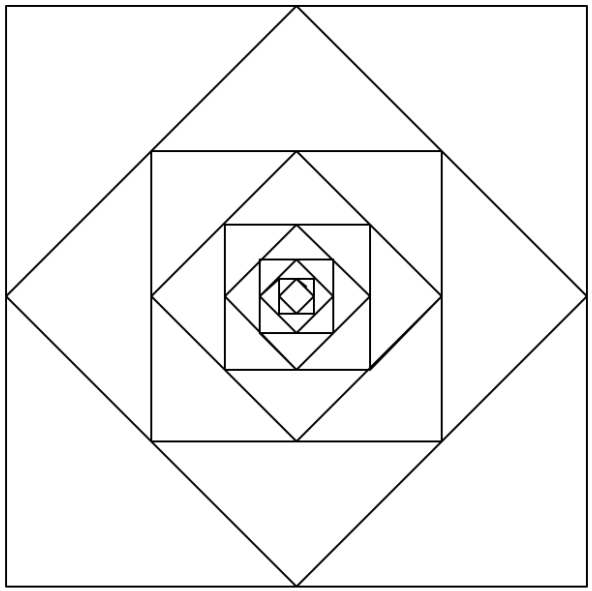
3. même argument $\Longrightarrow$ complexité exponentielle en $\Theta(4^n)$.

Exercice 6 - Correction



Exercice 7

Seriez-vous capable à l'aide du module turtle, et par récursivité, d'écrire une fonction prenant en paramètre un entier n et qui réalise le dessin suivant avec n carrés imbriqués (on pourra aussi passer en second paramètre la longueur des côtés du plus grand carré).



Quelle est sa complexité ?

Exercice 7

```
def dessin(l,n):
    tt.hideturtle()
    tt.down()
    if n==0 :
        return
    for i in range(4):    # pour chacun des 4 côtés...
        tt.forward(l)    # ...Tracé du côté...
        tt.right(90)    # ... puis rotation droite de 90°
    tt.forward(l/2.); tt.right(45) # Préparation carré suivant
    dessin(l/(2**0.5),n-1) # Appel récursif du carré suivant
```

La complexité est linéaire (en temps et en espace). Le nombre d'opérations à l'étape n satisfait la relation de récurrence :

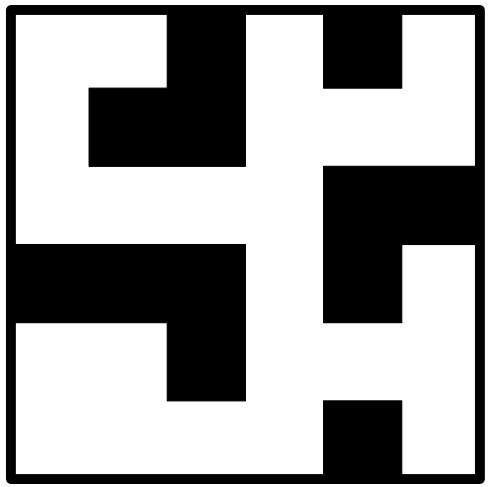
$$C(n) = C(n-1) + O(1) \quad , C(1) > 0$$
$$\Longrightarrow C(n) = \Theta(n)$$

Exercice 8

Reprendre le dernier exercice (le labyrinthe) du TD précédent sur les piles, et écrire une version récursive de l'algorithme de recherche d'un trajet de l'entrée à la sortie.

```
import numpy as
np T = np.array([
[0,0,1,0,1,0],
[0,1,1,0,0,0],
[0,0,0,0,1,1],
[1,1,1,0,1,0],
[0,0,1,0,0,0],
[0,0,0,0,1,0] ])
```

pour :



Exercice 8

On réutilise la fonction `voisin` (cf. TD5, on pourrait ne retourner qu'un seul voisin) :

```
def voisins(T,v):
    V = []
    N = np.shape(T)[0]
    i,j = v[0],v[1]
    for a in (-1,1):
        if 0<= i+a <N:
            if T[i+a,j] == 0:
                V.append((i+a,j))
        if 0<= j+a <N:
            if T[i,j+a] == 0:
                V.append((i,j+a))
    return V
```

Exercice 8

```
def laby_rec(T,P,sortie):
    if P == []:
        return False
    v = P[-1]
    if v == sortie:
        return P
    Vois = voisins(T,v)
    if Vois == []:
        P.pop()
        return laby_rec(T,P,sortie)
    v = Vois[0]
    T[v[0],v[1]]=-1
    P.append(v)
    return laby_rec(T,P,sortie)

def labyrinthe_rec(T,entree,sortie):
    from copy import deepcopy
    T = deepcopy(T)
    P = [entree]
    i,j = entree
    T[i,j] = -1
    return laby_rec(T,P,sortie)
```