1) *Algorithme d'Euclide*

L'énoncé a rappelé les propriétés du PGCD :

$$\forall a \in \mathbb{N} \quad \text{PGCD}(a,0) = a \quad \text{et} \quad \forall b \in \mathbb{N}^* \quad \text{PGCD}(a,b) = \text{PGCD}(b, a \,\%\, b)\,.$$

Version itérative : ces propriétés incitent à utiliser une boucle `while`

- tant que $b > 0$, remplacer a, b par $b, a \,\%\, b$
- à la sortie de la boucle renvoyer a.

La boucle se termine bien puisque b est un entier qui décroît strictement. Le résultat est correct d'après l'invariant de boucle suivant : "à chaque étape, le PGCD recherché est le PGCD du couple (a, b)". Or à la sortie de la boucle b est nul et l'on renvoie a qui est bien $\text{PGCD}(a, 0)$.

D'où le programme Python :

```
def PGCDi(a,b):
    while b > 0:
        a,b = b,a%b
    return a
```

Version récursive : les mêmes propriétés peuvent s'interpréter récursivement

- si $b = 0$, renvoyer 0 ; sinon, renvoyer $\text{PGCD}(b, a \,\%\, b)$.

Les appels récursifs se terminent car les valeurs successives de b décroissent strictement dans $\mathbb{N}$, donc on arrivera à la condition d'arrêt $b = 0$ en un temps fini. On montre que le résultat est correct par récurrence sur b.

Programme Python :

```
def PGCDr(a,b):
    if b==0:
        return a
    else:
        return PGCDr(b,a%b)
```

2) *Recherche dichotomique* : comme dans le cours, je fais diminuer par dichotomie l'intervalle $[\![i, j[\![$ jusqu'à ce qu'il ne contienne plus qu'une valeur, en faisant en sorte que, **si x est présent dans le tableau initial, alors il est dans la tranche** $t[i : j]$.

Version itérative : l'adaptation du programme du cours est immédiate

```
def ChercheI(x,t):
    i=0
    j=len(t)
    while i<j-1:
        m=(i+j)//2
        if t[m]<=x:
            i=m
        else:
            j=m
    if t[i]==x:
        return i
    else:
        return -1
```

Version récursive : ici la version du cours doit être retouchée, car les indices dans le nouveau tableau peuvent être décalés lors de l'appel récursif (lorsque la recherche se poursuit dans le tableau de droite). Pour en tenir compte, il suffit d'introduire un paramètre supplémentaire i, initialisé à 0 et contenant l'indice dans le tableau initial du premier élément du tableau en cours d'examen. Cet indice i a juste besoin d'être augmenté de m lorsque je "laisse tomber" les éléments numérotés de 0 à $m-1$. Au lieu d'un paramètre supplémentaire (plus coûteux en espace mémoire), on peut stocker les valeurs successives de i dans une variable globale, mais il est préférable, de façon générale, d'éviter d'utiliser des variables globales.

Une autre idée consiste à écrire une fonction auxiliaire récursive recevant comme paramètres i et j et renvoyant un indice d'une occurrence de x dans $t[i:j]$ s'il s'y trouve, -1 sinon.

Première idée

```
def ChercheR1(x,t,i=0):
    if len(t)==1:
        if t[0]==x:
            return i
        else:
            return -1
    else:
        m=len(t)//2
        if t[m]<=x:
            return ChercheR1(x,t[m:],i+m)
        else:
            return ChercheR1(x,t[:m],i)
```

Deuxième idée

```
def ChercheR2(x,t):
    def indice(i,j):
        if i==j-1:
            if t[i]==x:
                return i
            else:
                return -1
        else:
            m=(i+j)//2
        if t[m]<=x:
            return indice(m,j)
        else:
            return indice(i,m)

    return indice(0,len(t))
```

Noter que la première présente l'inconvénient de recopier des moitiés de tableaux, d'où un complexité temporelle linéaire, bien que la complexité en nombre de comparaisons reste logarithmique.

3) *Évaluation d'une fonction polynomiale : algorithme de Horner*

Pour calculer $P(x) = \sum_{k=0}^{n} a_k x^k$ en programmant le calcul tel quel, on effectue n additions et un certain nombre de multiplications, qui dépend de la façon dont on calcule les puissances !

Avec l'exponentiation naïve, on effectue k multiplications pour calculer $a_k x^k$, soit au total $\dfrac{n(n+1)}{2}$ multiplications, ce qui est de l'ordre de n^2.

Avec l'exponentiation rapide, le calcul de x^k nécessite un nombre de multiplications de l'ordre de $\log_2 k$, ce qui donne au total un nombre de multiplications de l'ordre de $\log_2(n!)$, ce qui est équivalent à $n\log_2 n$ grâce à la formule de Stirling qui donne $n! \sim \sqrt{2\pi n}\left(\dfrac{n}{e}\right)^n$. C'est mieux que n^2, mais moins bien que n.

a) *Version itérative*

Je pose, suivant l'énoncé : $y_0 = a_n$ et, pour i allant de 1 à n, $y_i = x \cdot y_{i-1} + a_{n-i}$.

Je vérifie alors aisément par récurrence sur i que : $\forall i \in [\![0,n]\!]$ $y_i = \sum_{k=0}^{i} a_{n-k} x^{i-k}$.

Pour $i = n$ j'obtiens bien

$$\boxed{y_n = P(x) \text{ et j'ai effectué } n \text{ additions et } n \text{ multiplications !}}$$

Pour implémenter cela en Python, je suppose que les coefficients sont stockés dans une liste p, de longueur $n+1$. Je traduis la boucle décrite ci-dessus en remarquant qu'une seule variable y suffit pour contenir les valeurs des y_i !

```
def HornerI(p,x):
    n=len(p)-1
    y=p[n]
    for i in range(1,n+1):
        y=x*y+p[n-i]
    return y
```

b) *Version récursive*

En notant $P_j(x) = \sum_{k=j}^{n} a_k x^{k-j}$, j'ai en effet $P_n(x) = a_n$ et, si $j < n$, $P_j(x) = a_j + x \cdot P_{j+1}(x)$.

Première idée : pour coller au plus près aux notations de l'énoncé, j'utilise un 3e paramètre j qui augmente au fil des appels récursifs, moyennant quoi la formulation de l'énoncé se traduit immédiatement :

```
def HornerR(p,x,j=0):
    n=len(p)-1
    if j==n:
        return p[n]
    else:
        return p[j]+x*HornerR(p,x,j+1)
```

La terminaison est assurée par la croissance stricte de j et la correction du résultat par récurrence descendante sur la valeur de j.

Deuxième idée : en réindexant, je constate que $P_j(x) = \sum_{i=0}^{n-j} a_{j+i}x^i$ est l'image de x par la fonction polynomiale associée à la liste $[a_j, a_{j+1}, \ldots, a_n]$. Autrement dit, si la liste p contient une seule valeur, je renvoie ladite valeur (polynôme constant !), sinon je renvoie le premier élément de p, plus x que multiplie le résultat associé à la liste p privée de son premier élément ! D'où le programme Python :

```
def HornerR(p,x):
    if len(p)==1:
        return p[0]
    else:
        return p[0]+x*HornerR(p[1:],x)
```

La terminaison est assurée par la décroissance stricte de la longueur de p et la correction du résultat par récurrence sur ladite longueur !

Noter que — là encore — la recopie du tableau est nuisible, bien que le nombre de multiplications reste n.

4) *Numérotation diagonale*

Le principe de cette numérotation et l'observation des cas particuliers conduit aux fonctions suivantes :

```
def entier(x,y):
    if y==0:
        if x==0:
            return 0
        else:
            return 1+entier(0,x-1)
    else:
        return 1+entier(x+1,y-1)
```

```
def couple(n):
    if n==0:
        return (0,0)
    else:
        (x,y)=couple(n-1)
        if x==0:
            return (y+1,0)
        else:
            return (x-1,y+1)
```

Pour la fonction `couple`, la terminaison (n décroît strictement à chaque appel récusif) et l'exactitude du résultat (par récurrence sur n) sont faciles à justifier.

C'est plus délicat pour la fonction `entier`... On peut utiliser l'expression entière $\frac{(x+y)(x+y+1)}{2}+y$, qui décroît strictement à chaque appel récursif (l'ironie de l'histoire étant que ladite expression donne directement la valeur n recherchée !).

5) a) La traduction est quasiment "mot à mot" :

```
def binom_a(n,p):
    if n<p:
        return 0
    elif p==0:
        return 1
    else:
        return binom_a(n-1,p-1)+binom_a(n-1,p)
```

La terminaison est garantie par la stricte décroissance de n à chaque appel récursif, avec arrêt dès que $n < p$ ou $p = 0$ (sachant que si n et p sont décrémentés tous les deux, p décroît strictement !).

On voit bien venir (comme pour les suites de Fibonacci) une complexité catastrophique due aux

recalculs déraisonnables. Précisément, le nombre d'additions (hors indices) $T_a(n,p)$ vérifie :

$$T_a(n,0)=0 \quad \text{et, pour } 1\leq p\leq n, \quad T_a(n,p)=T_a(n-1,p-1)+1+T_a(n-1,p),$$

c'est-à-dire que $C(n,p)=T_a(n,p)+1$ vérifie, pour $1\leq p\leq n$:

$$C(n,0)=1 \quad \text{et} \quad C(n,p)=C(n-1,p-1)+C(n-1,p).$$

Je reconnais (!) la relation du triangle de Pascal, qui caractérise les coefficients du binôme.

Par conséquent, $C(n,p)$ n'est autre que $\binom{n}{p}$ et donc, pour $0\leq p\leq n$:

$$T_a(n,p)=\binom{n}{p}-1.$$

On constate la catastrophe annoncée : temps de calcul prohibitif, sachant par exemple que $\binom{2n}{n}\sim\frac{4^n}{\sqrt{\pi n}}$, cela d'après la formule de Stirling : $n!\sim\sqrt{2\pi n}\left(\frac{n}{e}\right)^n$.

b) Une première version (un peu naïve) consiste à calculer $\binom{i}{j}$ pour les couples (i,j) tels que $0\leq i\leq n$ et $0\leq j\leq p$. Pour cela je vais remplir ligne par ligne un tableau `numpy` d'entiers, initialisé à 0, à l'aide de la formule de Pascal, bien sûr ! Pour ne pas être trop naïf tout de même, je commence par remplacer p par $n-p$ dans le cas où ce dernier est plus petit, tenant compte de la relation de symétrie $\binom{n}{p}=\binom{n}{n-p}$.

```
import numpy as np

def binom_b(n,p):
    if n-p<p: p=n-p
    if p<0:
        return 0
    else:
        t=np.zeros((n+1,p+1),dtype=int)
        t[0,0]=1
        for i in range(1,n+1):
            t[i,0]=1
            for j in range(1,p+1):
                t[i,j]=t[i-1,j-1]+t[i-1,j]
        return t[n,p]
```

Ici, calcul direct du nombre d'additions : $T_b(n,p)=np$, donc $O\left(n^2\right)$, préférable à 4^n !

On peut optimiser le programme précédent en remarquant que les $t[i,j]$ du rectangle $[\![0,n]\!]\times[\![0,p]\!]$ rempli ci-dessus ne sont pas tous utiles. En effet, pour pouvoir calculer par exemple $t[n,p]$, nul besoin d'avoir rempli la ligne $n-1$ en entier, puisque $t[n-1,p-1]$ et $t[n-1,p]$ suffisent ! Plus généralement, il apparaît que les couples (i,j) "vraiment" utiles sont ceux pour lesquels (faire un dessin...)

$$0\leq j\leq p \quad \text{et} \quad j\leq i\leq n-p+j.$$

D'où la fonction, en calculant, pour $j=0..p$, les éléments de tableau $t[i,j]$ pour $i=j..n-p+j$.

```
def binom_b2(n,p):
    if n-p<p: p=n-p
    if p<0:
        return 0
    else:
        t=np.zeros((n+1,p+1),dtype=int)
        for i in range(n-p+1):
            t[i,0]=1
        for j in range(1,p+1):
            t[j,j]=1
            for i in range(j+1,n-p+j+1):
                t[i,j]=t[i-1,j-1]+t[i-1,j]
        return t[n,p]
```

Cette fois-ci le nombre d'additions est $p(n-p)$.

c) En regardant encore mieux, on s'aperçoit qu'il suffit de mémoriser une seule colonne du tableau contenant les valeurs précédentes. Précisément, $n-p+1$ valeurs suffiront dans un tableau c, initialisées à 1 et mises à jour pour $j = 1..p$. En effet, le calcul de $\binom{i}{j}$ n'utilise que $\binom{i-1}{j}$ et $\binom{i-1}{j-1}$, il n'est donc pas gênant d'avoir "écrasé" les $\binom{i-1}{k}$ pour $k < j-1$ en les remplaçant par les $\binom{i}{k}$.

```
def binom_c(n,p):
    if n-p<p: p=n-p
    if p<0:
        return 0
    else:
        c=np.ones(n-p+1,dtype=int)
        for j in range(1,p+1):
            for k in range(1,n-p+1):
                c[k]+=c[k-1]
        return c[n-p]
```

Le nombre d'additions est de nouveau $p\,(n-p)$, l'amélioration consiste ici en une économie d'espace mémoire.

La correction de l'algorithme est ici non triviale à l'œil nu. Il est donc sage d'exhiber un invariant de boucle qui permettra de justifier l'exactitude du résultat (la terminaison est claire du fait des boucles `for`). En notant c_j la colonne obtenue après le j-ième passage dans la boucle, on prouve aisément par récurrence sur $j \in [\![0,p]\!]$ la propiété $\mathcal{P}_j$: "$\forall k \in [\![0,n-p]\!] \quad c_j[k] = \binom{j+k}{j}$". On vérifie en effet que l'instruction `c[k]+=c[k-1]` correspond au rang $j-1$ à

$$c_j[k] = c_{j-1}[k] + c_j[k-1] \quad \text{soit} \quad \binom{j+k}{j} = \binom{j+k-1}{j-1} + \binom{j+k-1}{j}$$

qui est bien vraie en vertu de la relation du triangle de Pascal !

N.B. On peut aussi adapter la première version du **b)** en ne stockant qu'une ligne ℓ de $p+1$ valeurs, à condition de la mettre à jour **de droite à gauche** pour ne pas écraser les valeurs encore utiles. C'est alors l'invariant de boucle Q_i : "$\forall j \in [\![0,p]\!] \quad \ell_i[j] = \binom{i}{j}$" qui permet de conclure. Plus simple à voir mais plus coûteux : np additions au lieu de $p\,(n-p)$.

```
def binom_c2(n,p):
    if n-p<p: p=n-p
    if p<0:
        return 0
    else:
        L=np.zeros(p+1,dtype=int)
        L[0]=1
        for i in range(1,n+1):
            for j in range(p,0,-1):
                L[j]+=L[j-1]
        return L[p]
```

6) Rappelons qu'un *palindrome* est une chaîne de caractères qui est identique lorsqu'on la lit de droite à gauche à celle qu'on lit de gauche à droite (par exemple ressasser, malayalam sont les deux plus longs palindromes acceptés au Scrabble). La vision récursive est naturelle : une chaîne d'au moins deux caractères est un palindrome si et seulement si la première et la dernière sont identiques et le mot obtenu en les supprimant est un palindrome (étant entendu qu'une chaîne d'au plus un caractère est un palindrome !). D'où le programme Python :

```
def est_palindrome(s):
    if len(s)<2:
        return True
    else:
        return (s[0]==s[-1]) and est_palindrome(s[1:-1])
```

Pour les anagrammes, deux chaînes non vides $s1$ et $s2$ sont des anagrammes si et seulement si le premier caractère de $s1$ est présent dans $s2$ et les chaînes obtenues en supprimant ce caractère de $s1$ et de $s2$ sont des anagrammes.

Pour gagner du temps, on peut commencer par tester l'égalité des longueurs des deux chaînes (inutile de lancer des appels récursifs si elles ne sont pas de même longueur !). Pour la recherche d'un caractère dans une chaîne, on peut la programmer soi-même, ou encore utiliser la méthode `find` :

```
def sont_anagrammes(s1,s2):
    if len(s1)!=len(s2):
        return False
    elif s1=="":
        return True
    else:
        k=s2.find(s1[0])
        if k==-1:
            return False
        else:
            return sont_anagrammes(s1[1:],s2[:k]+s2[k+1:])
```

7) *Les tours de Hanoï*

Pour une vision récursive du problème posé, je rebaptise les trois poteaux D, A et I (pour "départ", "arrivée" et "intermédiaire"), moyennant quoi le déplacement de n rondelles D vers A s'effectue de la façon suivante (pour $n \geq 1$, car pour $n = 0$ il n'y a rien à faire !) :

- déplacer $n-1$ rondelles de D vers I (appel récursif)
- afficher le déplacement d'une rondelle de D vers A
- déplacer $n-1$ rondelles de I vers A (appel récursif).

Cela se traduit directement en Python :

```
def hanoi(n,D,I,A):
    if n>0:
        hanoi(n-1,D,A,I)
        print(D,"->",A)
        hanoi(n-1,I,D,A)
```

La terminaison est assurée par la décroissance stricte de n à chaque appel récursif et la correction par récurrence sur n.

Du fait du double appel récursif, on se doute que la complexité est défavorable... Je note d_n le nombre de déplacements effectués pour déplacer n rondelles. La suite (d_n) vérifie d'après ce qui précède :

$$d_0 = 0 \quad \text{et} \quad \forall n \in \mathbb{N}^* \quad d_n = 2d_{n-1} + 1.$$

Il s'agit d'une suite arithmético-géométrique et -1 est le point fixe de $x \mapsto 2x+1$, donc $-1 = 2\times(-1)+1$ d'où, en soustrayant membre à membre :

$$d_0 + 1 = 1 \quad \text{et} \quad \forall n \in \mathbb{N}^* \quad d_n + 1 = 2\left(d_{n-1} + 1\right).$$

Autrement dit la suite de terme général $d_n + 1$ est géométrique de raison 2, d'où finalement

$$\boxed{\forall n \in \mathbb{N} \quad d_n = 2^n - 1.}$$

8) *Le flocon de Von Koch*

Comme le suggère l'énoncé, j'écris un programme récursif `flocon` traçant la "branche de flocon" (cf. la première figure) joignant un point A à un point B. Pour cela je commence par calculer d, la distance de A à B. Si d est inférieure à un seuil prédéfini (selon la précision souhaitée pour le dessin), je trace directement le segment $[A, B]$. Sinon, je détermine les points A_1, A_2 et A_3 décrits dans l'énoncé et j'appelle récursivement 4 fois le programme `flocon`...

Un peu de géométrie affine pour déterminer A_1 et A_3 qui sont de simples barycentres :

$$A_1 = \frac{1}{3}(2A + B) \quad \text{et} \quad A_3 = \frac{1}{3}(A + 2B).$$

C'est pour ce type de calculs que les tableaux `numpy.array` sont conseillés.

Et un peu de géométrie euclidienne pour déterminer A_2 !

Le vecteur $\overrightarrow{AB}$ a pour coordonnées $(x_B - x_A, y_B - y_A)$ et le vecteur $\vec{n}$ unitaire directement orthogonal

au vecteur $\overrightarrow{AB}$ a pour coordonnées $\frac{1}{d}(y_A - y_B, x_B - x_A)$ (car l'image du vecteur (λ, μ) par la rotation d'angle $\pi/2$ est $(-\mu, \lambda)$, la matrice de ladite rotation étant $\begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix}$!).

Or la hauteur de sommet A_2 dans le triangle équilatéral $A_1A_2A_3$ a pour pied le milieu $\frac{1}{2}(A+B)$ et pour longueur $\frac{\sqrt{3}}{2} \cdot \frac{d}{3}$ ($\frac{d}{3}$ étant la longueur du côté du triangle équilatéral).

Plutôt que de la recalculer sans cesse, je stocke dans une variable la constante $\frac{1}{2\sqrt{3}}$.

Ci-après le programme complet, avec la fonction `flocon` décrite ci-dessus et quelques options de `pyplot` :

- l'option `'k-'` dans `plt.plot` permet d'obtenir un tracé noir et un trait continu (`k` comme blac**k**... `b` comme **b**lue donne un tracé en bleu !)
- `plt.axis('equal')` fixe des échelles identiques sur les deux axes
- `plt.show()` affiche le dessin !

Le point `c` et les 3 appels de la fonction `flocon` permettent de tracer le flocon complet (cf. la seconde figure de l'énoncé).

```
from matplotlib import pyplot as plt
from math import sqrt
import numpy as np

k=0.5/sqrt(3)
xm=100
seuil=3

def flocon(a,b):
    d=sqrt((b[0]-a[0])**2+(b[1]-a[1])**2)
    if d<seuil:
        plt.plot([a[0],b[0]],[a[1],b[1]],'k-')
    else:
        a1=(2*a+b)/3
        a3=(a+2*b)/3
        a2=(a+b)/2+k*np.array([a[1]-b[1],b[0]-a[0]])
        flocon(a,a1)
        flocon(a1,a2)
        flocon(a2,a3)
        flocon(a3,b)

a=np.array([0,0])
b=np.array([xm,0])
c=np.array([xm/2,-xm*sqrt(3)/2])
plt.axis('equal')
flocon(a,b)
flocon(b,c)
flocon(c,a)
plt.show()
```