Corrigé

Récursivité

Exercice 1 On définit la fonction :

```
def power(x, n):
    if n == 0:
        return 1
    elif n == 1:
        return x
    y = power(x * x, n // 2)
    if n % 2 == 0:
        return y
    else:
        return x * y
```

Pour $n \geqslant 1$, considérons la décomposition de n en base 2 : $n = (b_k \cdots b_1 b_0)_2$ avec $b_k = 1$, et notons $C(n)$ le nombre de multiplications réalisées par cet algorithme pour calculer x^n. On dispose de la relation : $C(n) = C(\lfloor n/2 \rfloor) + 1 + b_0$ avec $\lfloor n/2 \rfloor = (b_k \cdots b_1)_2$ donc $C(n) = C(1) + k + b_{k-1} + \cdots + b_0 = k + b_{k-1} + \cdots + b_0$ et $k \leqslant C(n) \leqslant 2k$. Sachant que $k = \lfloor \log_2 n \rfloor$ on a donc $C(n) = O(\log n)$.

Exercice 2 Si $n \geqslant 1$ on résout le problème récursivement de la façon suivante :

- on déplace $n-1$ disques de la tige 1 à la tige 3 en respectant la contrainte ;
- on déplace le dernier disque de la tige 1 à la tige 2 ;
- on déplace $n-1$ disques de la tige 3 à la tige 1 en respectant la contrainte ;
- on déplace le dernier disque de la tige 2 à la tige 3 ;
- on déplace $n-1$ disques de la tige 1 à la tige 3 en respectant la contrainte.

Ce qui conduit à la fonction suivante :

```
def hanoi(n, i=1, j=3):
    if n > 0:
        hanoi(n-1, i, j)
        print("Déplacer le disque {} de la tige {} vers la tige 2.".format(n, i))
        hanoi(n-1, j, i)
        print("Déplacer le disque {} de la tige 2 vers la tige {}.".format(n, j))
        hanoi(n-1, i, j)
```

Si $C(n)$ désigne le nombre de déplacements, nous avons $C(0) = 0$ et $C(n) = 3C(n-1) + 2$, ce qui conduit à $C(n) = 3^n - 1$.

Exercice 3 Tout mot m de longueur 0 ou 1 est un palindrome ; tout mot $m = am'b$ de longueur supérieure ou égale à 2 (où a et b sont des lettres) est un palindrome si et seulement si $a = b$ et m' est un palindrome. D'où la fonction :

```
def palindrome(m):
    if len(m) < 2:
        return True
    return m[0] == m[-1] and palindrome(m[1:-1])
```

Remarque. Cette version a une complexité quadratique car le calcul `m[1:-1]` au sein de l'appel récursif a un coût linéaire. Pour obtenir une version de complexité linéaire il faut faire les appels récursifs sur les indices :

```
def palin(m, i, j):
    if j - i < 2:
        return True
    return m[i] == m[j-1] and palin(m, i+1, j-1)

def palindrome(m):
    return palin(m, 0, len(m))
```

Exercice 4 Si $n = 0$ ou $n = 1$ il n'y a rien à calculer ; si $n \geqslant 2$ on a $\lfloor n/2 \rfloor < n$ et $\lceil n/2 \rceil < n$, ce qui assure la terminaison de la fonction. D'où :

```
def power(x, n):
    if n == 0:
        return 1
    elif n == 1:
        return x
    return power(x, n//2) * power(x, n-n//2)
```

Notons $C(n)$ le nombre de multiplications effectuées pour calculer a^n. On dispose des relations :

$$C(0) = C(1) = 0 \qquad \text{et} \qquad C(n) = C(\lfloor n/2 \rfloor) + C(\lceil n/2 \rceil) + 1.$$

Montrons par récurrence sur $n \geqslant 1$ que $C(n) = n - 1$:

- c'est clair si $n = 1$ ou $n = 2$;
- si $n \geqslant 3$, supposons l'hypothèse de récurrence vérifiée jusqu'au rang $n - 1$. Alors :

$$C(n) = \lfloor n/2 \rfloor - 1 + \lceil n/2 \rceil - 1 + 1 = n - 1$$

ce qui prouve le résultat au rang n.

Autrement dit, cette version n'apporte rien comparé à un algorithme qui exploiterait la relation : $a^n = a \times a^{n-1}$.

Exercice 5

```
def numerote(x, y):
    if x == 0 and y == 0:
        return 0
    if y > 0:
        return 1 + numerote(x+1, y-1)
    return 1 + numerote(0, x-1)
```

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

Exercice 6

a) Considérons la valeur minimale t_k du tableau. Si $k = 0$ alors $t_1 = t_0$ donc t_1 est un minimum local ; si $k = n - 1$ alors $t_{n-2} = t_{n-1}$ donc t_{n-2} est un minimum local. Dans les autres cas t_k est un minimum local.

b) On procède à une recherche dichotomique en considérant t_k avec $k = \lfloor n/2 \rfloor$:

- si $t_k \leqslant t_{k-1}$ et $t_k \leqslant t_{k+1}$, t_k est un minimum local ;
- si $t_k > t_{k-1}$, le tableau $t[0 \cdots k]$ possède la même propriété que le tableau initial donc la recherche peut s'y poursuivre ;
- de même, si $t_k > t_{k+1}$ la recherche se poursuit dans $t[k \cdots n-1]$.

```
def min_local(t, *args):
    if len(args) == 0:
        i, j = 0, len(t)
    else:
        i, j = args
    k = (i + j) // 2
    if t[k] <= t[k-1] and t[k] <= t[k+1]:
        return t[k]
    if t[k] > t[k-1]:
        return min_local(t, i, k)
    return min_local(t, k, j)
```

Exercice 7 Si a désigne un élément d'un ensemble S et $S' = S \setminus \{a\}$, les sous-ensembles de S sont :

- chacun des sous-ensembles de S' ;
- chacun des sous-ensembles de S' à qui on adjoint a.

D'où la solution :

```
def subset(l):
    if l == []:
        return [[]]
    else:
        m1 = subset(l[1:])
        m2 = [[l[0]] + s for s in m1]
        return m1 + m2
```

Exercice 8 La fonction `bubble1` nécessite deux appels récursifs, le premier pour tracer la génération précédente vers la droite du cercle initial, le second pour tracer la génération précédente vers le bas.

```
def bubble1(n, x=0, y=0, r=8):
    circle((x, y), r)
    if n > 1:
        bubble1(n-1, x+3*r/2, y, r/2)
        bubble1(n-1, x, y-3*r/2, r/2)
```

Le premier appel à la fonction `bubble2` est suivi de quatre appels récursifs, les suivants de seulement trois appels récursifs. C'est pourquoi on ajoute un paramètre supplémentaire qui indique la direction (nord, ouest, sud, est) de l'expansion.

```
def bubble2(n, x=0, y=0, r=8, d=''):
    circle((x, y), r)
    if n > 1:
        if d != 's':
            bubble2(n-1, x, y+3*r/2, r/2, 'n')
        if d != 'w':
            bubble2(n-1, x+3*r/2, y, r/2, 'e')
        if d != 'n':
            bubble2(n-1, x, y-3*r/2, r/2, 's')
        if d != 'e':
            bubble2(n-1, x-3*r/2, y, r/2, 'w')
```

Exercice 9 On peut définir la fonction `polygon` à l'aide de la fonction `fill` qui colorie l'intérieur d'une ligne polygonale :

```
def polygon(*args):
    X, Y = [], []
    for arg in args:
        X.append(arg[0])
        Y.append(arg[1])
    plt.fill(X, Y, 'b')
```

Pour obtenir les approximations souhaitées du triangle de Sierpiński on définit alors la fonction suivante :

```
from numpy import sqrt

def sierpinski(n, a=[0, 0], b=[1, 0], c=[.5, sqrt(3)/2]):
    if n == 1:
        polygon(a, b, c)
    else:
        u = [(b[0]+c[0])/2, (b[1]+c[1])/2]
        v = [(c[0]+a[0])/2, (c[1]+a[1])/2]
        w = [(a[0]+b[0])/2, (a[1]+b[1])/2]
        sierpinski(n-1, a, w, v)
        sierpinski(n-1, w, b, u)
        sierpinski(n-1, v, u, c)
```

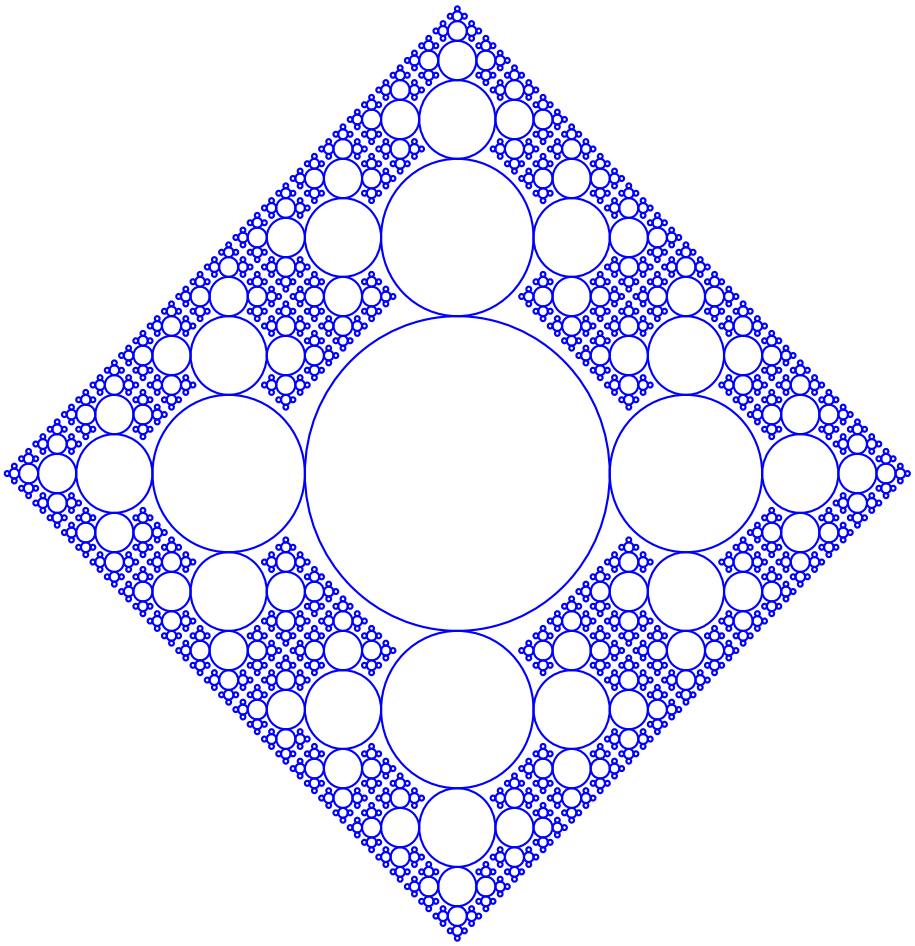

FIGURE 1 – Le résultat de `bubble2(7)`.

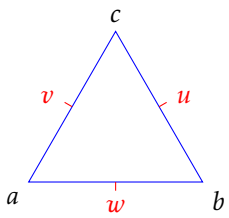

FIGURE 2 – La position relative des points u, v, w en fonction de a, b, c.

Exercice 10 La reine de la colonne k peut attaquer la reine de la colonne j dans l'un des trois cas suivants : $q_i = q_j$ (les deux reines sont dans le même rangée), $q_j - q_k = j - k$ ou $q_j - q_k = k - j$ (les deux reines partagent une même diagonale). D'où la fonction :

```
def reine(q, j):
    if j == len(q):
        print(q)
    for i in range(len(q)):
        legal = True
        for k in range(j):
            if q[k] in (i, i - j + k, i + j - k):
                legal = False
                break
        if legal:
            q[j] = i
            reine(q, j + 1)
```

Pour résoudre le problème initial, il suffit d'appliquer `reine(q, 0)` à un tableau de n cases :

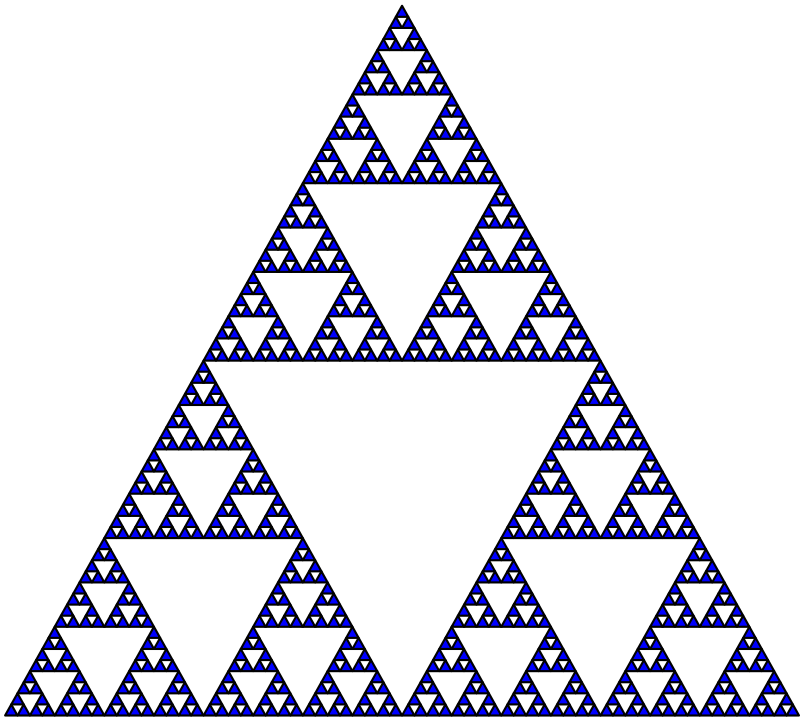

FIGURE 3 – Le résultat de `sierpinski(7)`.

```
>>> q = [None] * 8
>>> reine(q, 0)
[0, 4, 7, 5, 2, 6, 1, 3]
[0, 5, 7, 2, 6, 3, 1, 4]
........................
[7, 3, 0, 2, 5, 1, 6, 4]
```

(il y a 92 solutions).