Chapitre 2

Corrigé des exercices

Exercice 1 Montrons que la fonction f retourne un résultat pour tout $n \in \mathbb{N}$.

- Ceci est clair pour $n \geqslant 101$.
- Si $n \in [\![90, 100]\!]$, $f(n+11) = n+1$ car $n+11 \geqslant 101$, donc $f(n) = f(n+1)$. On en déduit que $f(n) = f(101) = 91$, et f est donc bien définie et constante égale à 91 sur $[\![90, 100]\!]$.
- Si $n \in [\![0, 89]\!]$, il existe $p \in \mathbb{N}^*$ tel que $90 \leqslant n+11p \leqslant 100$. Alors $f(n) = \underbrace{f \circ \cdots \circ f}_{p \text{ fois}}(n+11p) = \underbrace{f \circ \cdots \circ f}_{p-1 \text{ fois}}(91) = 91$.

  f est donc définie sur $[\![0, 89]\!]$ et constante égale à 91 sur cet intervalle.

Cette fonction est connue sous le nom de fonction 91 de McCarthy.

Exercice 2 Montrons par récurrence sur n que $g(n)$ est calculable et que $g(n) \in [\![0, n]\!]$.

- C'est clair si $n = 0$.
- Si $n > 0$, supposons le résultat acquis jusqu'au rang $n-1$. Pour tout $k \in [\![0, n-1]\!]$, $g(k)$ est donc calculable et $g(k) \in [\![0, n-1]\!]$.
  En particulier, $g(g(n-1))$ est calculable et $g(g(n-1)) \in [\![0, n-1]\!]$. On en déduit que $g(n)$ est calculable, et que $g(n) \in [\![1, n]\!] \subset [\![0, n]\!]$.

Montrons maintenant par récurrence sur $n \in \mathbb{N}$ que $g(n) = \left\lfloor \dfrac{n+1}{\alpha} \right\rfloor$, avec $\alpha = \dfrac{1+\sqrt{5}}{2}$.

- C'est clair si $n = 0$ car $\alpha > 1$ donc $\lfloor 1/\alpha \rfloor = 0$.
- Si $n > 0$, supposons le résultat acquis jusqu'au rang $n-1$. En posant $p = g(n-1)$, on a donc par hypothèse de récurrence :
$$p = \left\lfloor \frac{n}{\alpha} \right\rfloor \quad \text{et} \quad g(n) = n - \left\lfloor \frac{p+1}{\alpha} \right\rfloor = \left\lceil n - \frac{p+1}{\alpha} \right\rceil$$
  Sachant que $p \leqslant \dfrac{n}{\alpha} < p+1$ et que $\alpha - \dfrac{1}{\alpha} = 1$, on obtient l'encadrement : $p - \dfrac{1}{\alpha} \leqslant n - \dfrac{p+1}{\alpha} < p+1$.
  Sachant que $0 < \dfrac{1}{\alpha} < 1$, on a donc : $p - 1 < n - \dfrac{p+1}{\alpha} < p+1$ et $p < \dfrac{n+1}{\alpha} < p+2$.
  Traitons alors deux cas.
  - Si $\dfrac{n+1}{\alpha} < p+1$, alors $\left\lfloor \dfrac{n+1}{\alpha} \right\rfloor = p$ et $n - \dfrac{p+1}{\alpha} < \left(\alpha - \dfrac{1}{\alpha}\right)(p+1) - 1 = p$ donc $\left\lceil n - \dfrac{p+1}{\alpha} \right\rceil = p$.
  - Si $\dfrac{n+1}{\alpha} > p+1$, alors $\left\lfloor \dfrac{n+1}{\alpha} \right\rfloor = p+1$ et $n - \dfrac{p+1}{\alpha} > \left(\alpha - \dfrac{1}{\alpha}\right)(p+1) - 1 = p$ donc $\left\lceil n - \dfrac{p+1}{\alpha} \right\rceil = p+1$.

  (On ne peut avoir l'égalité $\dfrac{n+1}{\alpha} = p+1$ car α est irrationnel.)
  Dans les deux cas, on a bien $\left\lceil n - \dfrac{p+1}{\alpha} \right\rceil = \left\lfloor \dfrac{n+1}{\alpha} \right\rfloor$, soit $g(n) = \left\lfloor \dfrac{n+1}{\alpha} \right\rfloor$.

Exercice 3 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(a, n):
    if n == 0:
        return 1
    elif n == 1:
        return a
    return power(a, n//2) * power(a, 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 : $a^n = a \times a^{n-1}$. Cependant, la remarque de l'énoncé permet de modifier la fonction initiale pour écrire :

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

On dispose cette fois des relations : $C(0) = C(1) = 0$ et $C(\lfloor n/2 \rfloor) + 1 \leqslant C(n) \leqslant C(\lfloor n/2 \rfloor) + 2$.
Montrons par récurrence sur $n \geqslant 1$ que $C(n) \leqslant 2\log n$:

– c'est clair si $n \leqslant 2$ car $C(2) = 1$;

– si $n \geqslant 3$, supposons le résultat acquis jusqu'au rang $n-1$. Alors :

$$C(n) \leqslant 2\log\lfloor n/2 \rfloor + 2 \leqslant 2\log(n/2) + 2 = 2\log n$$

ce qui prouve le résultat au rang n.

Ainsi, cet algorithme est de coût logarithmique ; il porte le nom d'algorithme d'*exponentiation rapide*.

Exercice 4

```
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 5

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 6

a) Posons $n = 2^p$ et notons a_p le nombre de fois que la fonction blit est utilisée. Alors $a_0 = 0$ (une image d'un seul pixel est invariante par rotation) et $a_p = 5 + 4a_{p-1}$ donc $a_p = 5(4^p - 1)/3$. Le nombre de fois que la fonction blit est utilisée est égale à $5(n^2 - 1)/3$.

b) Notons b_p le coût total lorsqu'un blit $k \times k$ a un coût égal à k^2. On a $b_0 = 0$ et $b_p = 5.4^{p-1} + 4b_{p-1}$.

Alors : $\dfrac{b_p}{4^p} = \dfrac{b_{p-1}}{4^{p-1}} + \dfrac{5}{4}$ donc par télescopage $b_p = 4^p \dfrac{5p}{4} = 5p4^{p-1} = \mathrm{O}(p4^p) = \mathrm{O}(n^2 \log n)$.

c) Notons c_p le coût total lorsqu'un blit $k \times k$ a un coût égal à k. On a $c_0 = 0$ et $c_p = 5.2^{p-1} + 4c_{p-1}$.

Alors : $\dfrac{c_p}{4^p} = \dfrac{c_{p-1}}{4^{p-1}} + \dfrac{5}{2^{p+1}}$ donc par télescopage $c_p = 4^p \displaystyle\sum_{k=1}^{p} \frac{5}{2^{k+1}} = 4^p \frac{5}{2}\Big(1 - \frac{1}{2^p}\Big) = \frac{5}{2}(4^p - 2^p) = \mathrm{O}(4^p) = \mathrm{O}(n^2)$.

Remarque. La bibliothèque `matplotlib.image` offre une instruction `imread` pour importer un fichier image au format PNG sous la forme d'une matrice numpy bi-dimensionnelle, chaque case de la matrice représentant un pixel (sous la forme d'un triplet RGB ou d'un quadruplet RGBA), et la bibliothèque `matplotlib.pyplot` une fonction `imshow` permettant d'afficher l'image associée à une matrice de ce type.
Bien que le procédé soit lent, il est possible de simuler un blit par la copie d'une partie d'une matrice vers une autre et ainsi mettre en œuvre le procédé de rotation décrit plus haut :

```
def rotate(a):
    n = a.shape[0]
    if n == 1:
        return None
    k = n // 2
    b = a[:k, :k].copy()
    a[:k, :k] = a[:k, k:]
    a[:k, k:] = a[k:, k:]
    a[k:, k:] = a[k:, :k]
    a[k:, :k] = b
    rotate(a[:k, :k])
    rotate(a[:k, k:])
    rotate(a[k:, k:])
    rotate(a[k:, :k])
```

Un exemple d'utilisation avec une image 256×256 :

```
import matplotlib.pyplot as plt
import matplotlib.image as img

a = img.imread('picasso.png')
b = a.copy()
rotate(b)
plt.subplot(1, 2, 1)
plt.imshow(a)
plt.axis('off')
plt.subplot(1, 2, 2)
plt.imshow(b)
plt.axis('off')
plt.show()
```


Exercice 7 Une version simple (mais peu économique) de définir la fonction `circle` consiste à écrire :

```
import matplotlib.pyplot as plt
import numpy as np

def circle(coords, r):
    X, Y = [], []
    for t in range(101):
        X.append(coords[0]+r*np.cos(t*np.pi/50))
        Y.append(coords[1]+r*np.sin(t*np.pi/50))
    plt.plot(X, Y, 'b')
```

On peut alors définir récursivement la fonction `bubble1` de la façon suivante :

```
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)
```

Pour définir la fonction `bubble2`, 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 8 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)
```

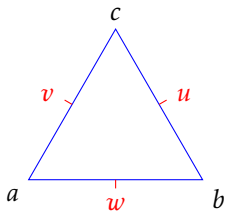

FIGURE 1 – La position relative des points u, v, w en fonction de a, b, c.

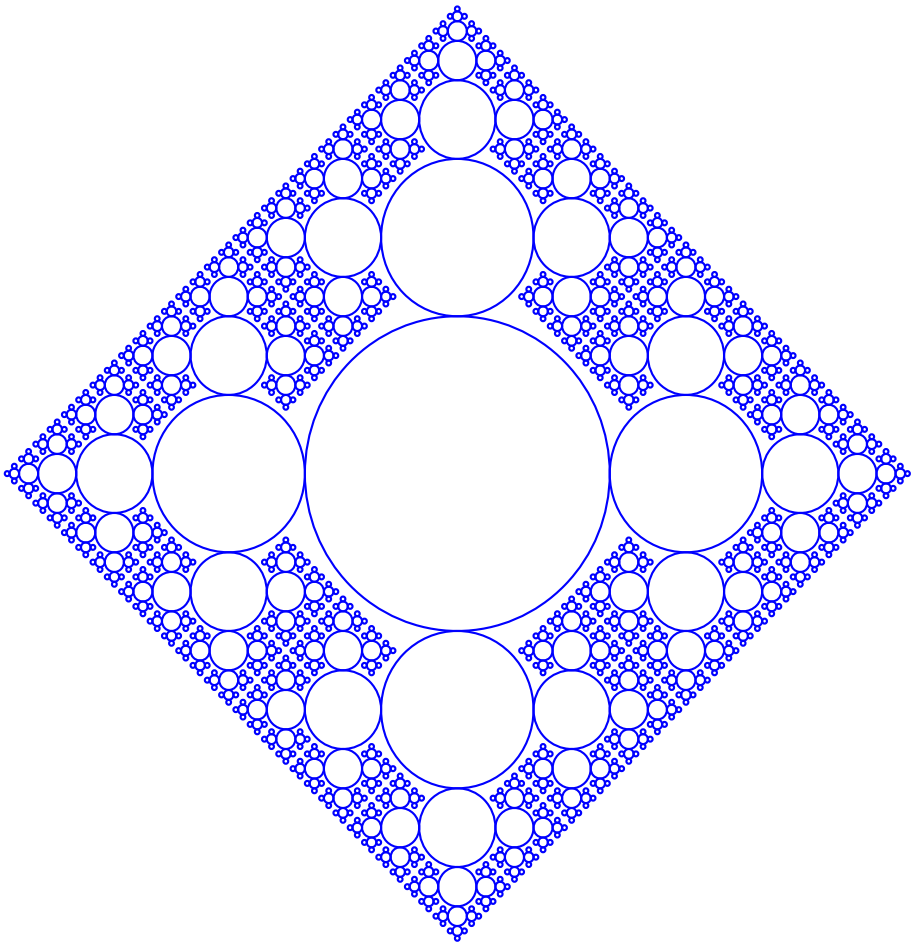

FIGURE 2 – Le résultat de `bubble2(7)`.

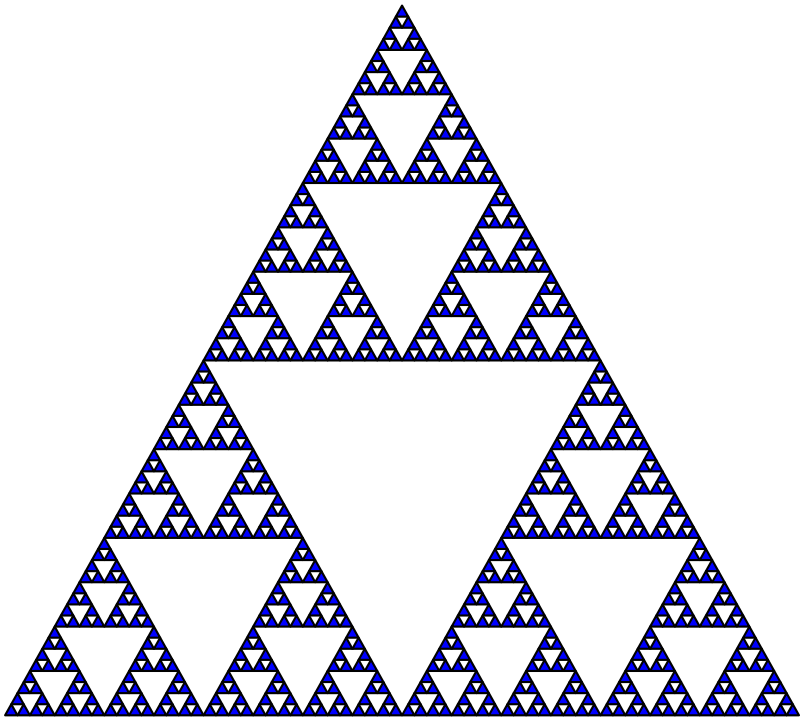

FIGURE 3 – Le résultat de `sierpinski(7)`.