Corrigé – Récursivité

Exercice 1

La factorielle $n!$ d'un nombre entier $n \geq 1$ est définie par $n! = 1 \times 2 \times 3 \times \cdots \times n$.

1.

n	1	2	3	4	5	6	7
$n!$	1	2	6	24	120	720	5040

2. $n \times (n-1)!$

3. (a) À l'exécution, on ne sort jamais de cette fonction car elle s'appelle à l'infini. Il faudrait une condition pour en sortir à un moment pertinent.

(b)
```
def factorielle(n):
    if n == 1:
        return 1

    return n*factorielle(n-1)
```

(c) Dans le corps du script :

```
assert factorielle(1) == 1
assert factorielle(2) == 2
assert factorielle(3) == 6
assert factorielle(4) == 24
assert factorielle(5) == 120
assert factorielle(6) == 720
assert factorielle(7) == 5040
```

(d) 1 appel initial puis 3 appels récursifs sont réalisés lors de l'appel `factorielle(4)`. Il y a donc 4 appels de la fonction en tout. On peut le vérifier en faisant un `print n` en tout début du corps de la fonction.

Exercice 2

Carrés imbriqués

```
# IMPORTS

from turtle import *
from math import sqrt

# FONCTIONS

def carre(cote):
    for _ in range(4):
        forward(cote)
        right(90)

```

```
def carres_imbriques(cote):
    if cote < 1:
        return

    # tracer le carré
    carre(cote)

    # déplacement
    forward(cote/2)
    right(45)

    # appel récursif
    carres_imbriques(cote/sqrt(2))

# SCRIPT

setup(800, 600)
# speed(10)

carres_imbriques(200)

# boucle maintenant la fenêtre ouverte
mainloop()
```

Triangles imbriqués

```
# IMPORTS

from turtle import *
from math import sqrt

# FONCTIONS

def triangle(cote):
    for _ in range(3):
        forward(cote)
        right(120)

def triangles_imbriques(cote):
    if cote < 1:
        return

    # tracer le triangle
    triangle(cote)

    # déplacement
    forward(cote/2)
    right(60)

    # appel récursif
    triangles_imbriques(cote/2)
```

```
# SCRIPT

setup(800, 600)
# speed(10)

triangles_imbriques(200)

# boucle maintenant la fenêtre ouverte
mainloop()
```

Cercles et carrés imbriqués

```
# IMPORTS

from turtle import *
from math import sqrt

# FONCTIONS

def carre(cote):
    for _ in range(4):
        forward(cote)
        right(90)

def carre_blanc(cote):
    begin_fill()
    color("white", "white")
    carre(cote)
    end_fill()

def disque_gris(cote):
    begin_fill()
    color("gray", "gray")
    circle(cote)
    end_fill()

def cercles_carres_imbriques(cote):
    if cote < 1:
        return

    # tracer le carré blanc
    carre_blanc(cote)

    # déplacement position initiale cercle
    right(90)
    forward(cote/2)

    # trace le cercle gris
    disque_gris(cote/2)

```

```
    # déplacement
    left(135)

    # appel récursif
    cercles_carres_imbriques(cote/sqrt(2))

# SCRIPT

setup(800, 600)
# speed(10)

cercles_carres_imbriques(200)

# boucle maintenant la fenêtre ouverte
mainloop()
```

Exercice 3

On considère la fonction récursive `fibo` ci-dessous, qui prend en argument un nombre entier.

```
def fibo(n):
    if n<=1: # condition d'arrêt
        return n
    return fibo(n-1)+fibo(n-2)
```

1. 8
2. 25
3. 144
4.
```
def fibo_iteratif(n):
    # on règle les deux premiers termes
    if n <= 1:
        return n

    n_moins_1 = 1
    n_moins_2 = 0

    for _ in range(2, n+1):
        total = n_moins_1 + n_moins_2
        n_moins_2 = n_moins_1 # attention à l'ordre des instructions !
        n_moins_1 = total

    return total
```

Exercice 4

```
# FONCTIONS
def est_palindrome_ite(mot):
    taille_mot = len(mot)
    for indice in range(taille_mot//2):
```

```
5           if mot[indice] != mot[taille_mot-1-indice]:
6               return False
7       return True
8
9   # Autre version
10  def est_palindrome_ite_compact(mot):
11      return mot == mot[::-1]
12
13  def est_palindrome(mot):
14      if len(mot) <= 1: # si le mot est vide ou n'a qu'un seul caractère, c'est un
        ↪  palindrome
15          return True
16
17      if mot[0] == mot[-1]: # comparaison du premier et du dernier caractère
18          return est_palindrome(mot[1:-1]) # on rappelle la fonction sur le mot sans
            ↪  ses lettres extrêmes
19
20      return False
21
22  # SCRIPT
23  # batterie de tests pour vérifier nos fonctions (ne pas oublier de tester plusieurs
    ↪  cas dont les cas de base !)
24  assert est_palindrome_ite("")
25  assert est_palindrome_ite("z")
26  assert est_palindrome_ite("été")
27  assert est_palindrome_ite("radar")
28  assert not est_palindrome_ite("Radar")
29  assert not est_palindrome_ite("spaghetti")
30
31  assert est_palindrome_ite_compact("")
32  assert est_palindrome_ite_compact("z")
33  assert est_palindrome_ite_compact("été")
34  assert est_palindrome_ite_compact("radar")
35  assert not est_palindrome_ite_compact("Radar")
36  assert not est_palindrome_ite_compact("spaghetti")
37
38  assert est_palindrome("")
39  assert est_palindrome("z")
40  assert est_palindrome("été")
41  assert est_palindrome("radar")
42  assert not est_palindrome("Radar")
43  assert not est_palindrome("spaghetti")
```

Exercice 5

```python
# FONCTIONS
def somme_it(lst):
    total = 0
    for element in lst:
        total += element
    return total

def somme_rec(lst):
    if len(lst) == 0:
        return 0

    return lst[0] + somme_rec(lst[1:])

# SCRIPT
assert somme_it([]) == 0
assert somme_it([9999]) == 9999
assert somme_it([1, 1, 1, 1]) == 4
assert somme_it([1, 2, 3]) == 6

assert somme_rec([]) == 0
assert somme_rec([9999]) == 9999
assert somme_rec([1, 1, 1, 1]) == 4
assert somme_rec([1, 2, 3]) == 6
```

Exercice 6

```python
# FONCTIONS
def rendu_glouton(a_rendre, pieces):
    """ rendu_glouton(int, int, list) -> list
    a_rendre : la somme d'argent à rendre
    pieces : un tableau de pièces possibles à rendre"""
    piece_max = pieces[0]

    if a_rendre == 0:
        return []

    if a_rendre >= piece_max:
        return [piece_max] + rendu_glouton(a_rendre - piece_max, pieces)

    return rendu_glouton(a_rendre, pieces[1:])

# SCRIPT
valeurs_possibles = [100, 50, 20, 10, 5, 2, 1]

assert rendu_glouton(67, valeurs_possibles) == [50, 10, 5, 2]
assert rendu_glouton(291, valeurs_possibles) == [100, 100, 50, 20, 20, 1]
# si on ne dispose pas de billets de 100 :
assert  rendu_glouton(291, valeurs_possibles[1:]) == [50, 50, 50, 50, 50, 20, 20, 1]
```