Exercice 1 – Fonction mystère

Correction **Question 1** Lorsque l'on appelle `mystere([14,20,3,16])`, les appels récursifs successifs sont : `mystere([14,20,3])`, `mystere([14,20])`, et enfin `mystere([14])`. La condition d'arrêt assure alors que `mystere([14])` vaut 14. Ceci permet alors de calculer `mystere([14,20])` : puisque $14 < 20$, `mystere([14,20])` vaut 14. Puis, `mystere([14,20,3])` vaut 3, et enfin `mystere([14,20,3,16])` vaut 3.

Question 2 On peut alors conjecturer que `mystere` calcule le minimum d'une liste.

Question 3 Soit L une liste de taille n. Alors n est un variant de boucle car :
- si $n = 1$ ou $n = 0$, l'algorithme se termine ;
- si $n > 1$ chaque appel récursif est réalisé avec l'argument $L[0 : n-1]$, qui est de longueur $n-1$. Ainsi n décrit une suite strictement décroissante, jusqu'à ce que $n = 1$ (terminaison de l'algorithme).

Soit la propriété suivante : soit L une liste de taille k. L'appel à la fonction mystère retourne le plus petit élément de L. Montrons-le par récurrence.

Pour une liste de longueur 0 ou 1, le résultat est immédiat.

Soit une liste de taille $k+1$. Alors x reçoit le résultat de `mystere(L[0:k])`. D'après la propriété, x contient donc le plus petit élément de la liste `L[0:k]`. Ensuite x est comparé à l'élément `L[k]`. Si x est inférieur à cet élément, c'est donc le plus petit élément, et x est bien retourné. Sinon c'est que l'élément `L[k]` est le plus petit de la liste. C'est bien celui qui est retourné.

La propriété énoncée est donc bien héréditaire, et l'algorithme renvoie bien toujours le minimum de la liste entrée en argument.

Exercice 2 – Palindrome...

Correction **Question 1**

```
def miroir_it(mot):
    """ fonction itérative retournant le miroir d'une chaîne de caractères """
    tom=''
    for lettre in mot:
        tom=lettre+tom
    return (tom)
```

Question 2

```
def miroir_rec(mot):
    """ fonction récursive retournant le miroir d'une chaîne de caractères """
    n=len(mot)
    if len(mot) <= 1:
        return mot
    else:
        return mot[-1] + miroir_rec(mot[0:n-1])
```

Question 4 Soit `mot` une chaîne de caractères de longueur n.
Dans la version itérative, il y a exactement n tours de boucle. Il est facile de montrer par récurrence l'invariant suivant : au début du k-ème tour de boucle, `tom` est de longueur $k-1$. Or l'opération `tom = lettre + tom` consiste à créer une chaîne de k caractères dans laquelle sont recopiées `lettre` et `tom`. La complexité de cette opération est donc un $\mathcal{O}(k)$. Par sommation, on voit donc que `miroir_it` a une complexité en $\mathcal{O}(n^2)$.
On remarque que si l'on avait pu parcourir `mot` de droite à gauche et non de gauche à droite, on aurait pu rajouter à `tom` les lettres par la droite, et utiliser la syntaxe `tom += lettre`. Mais comme les chaînes ne sont pas de type *mutable*, cette opération a la même complexité que `tom = tom + lettre` (ce qui n'aurait pas été le cas avec des listes).
Dans la version récursive, on montre par récurrence que la complexité est également un $\mathcal{O}(n^2)$. Posons l'hypothèse de récurrence : $(\mathcal{P}_k)$: si `mot` est de longueur k, `miroir_rec (mot)` nécessite $\mathcal{O}(k^2)$ opérations.
Pour $k = 0$ ou 1, ceci est immédiat grâce aux conditions d'arrêt.

Si $(\mathscr{P}_k)$ est vraie. Il existe donc $K \in \mathbb{R}$ (que l'on peut supposer supérieur à 1) tel que si `mot` est de longueur $k+1$, alors `miroir_rec (mot[0,n-1])` nécessite moins de $K.k^2$ opérations. Or le calcul de `miroir_rec(mot[0,n-1])` nécessite $k+1$ opérations de plus, ce qui fait au total moins de $K.(k+1)^2$ opérations. L'hypothèse de récurrence est donc héréditaire.

Exercice 3 – Suite de Fibonacci

Correction **Question 1**

```
def fibonacci_it(n):
    """fibonacci itérative"""
    a=0
    b=1
    for i in range(n-1): # invariant en entrée : a=u_i, b=u_{i+1}
        c=a+b
        a=b
        b=c
        # invariant en sortie : a=u_{i+1}, b=u_{i+2}
    return (b) # àla fin de la boucle, i=n-2
```

Il y a $n-2$ tours de boucle dans la boucle de cet algorithme, et à chaque tour de boucle, une addition et trois affectations sont effectuées. En dehors de la boucle, il n'y a qu'un nombre constant d'opérations, et ainsi cet algorithme a une complexité linéaire.

Question 2

```
def fibonacci_rec(n):
    """fibonacci recursive avec renvoi de u_n"""
    if n==0:
        return 0
    elif n==1:
        return 1
    else:
        return (fibonacci_rec(n-1)+fibonacci_rec(n-2))
```

Notons $T(n)$ le nombre d'opérations nécessaires pour calculer `fibonacci_rec(n)`. Nous avons alors très facilement : $T(0)=T(1)=1$, et pour tout $n \in \mathbb{N}$ tel que $n>1$, $T(n)=T(n-1)+T(n-2)+k \geq T(n-1)+T(n-2)$, où k est une constante. C'est une relation arithmético-géométrique que l'on sait résoudre. On trouve qu'il existe deux constantes a et b telles que $T(n)=a\left(\frac{1+\sqrt{5}}{2}\right)^n+b\left(\frac{1-\sqrt{5}}{2}\right)^n-k \sim a\left(\frac{1+\sqrt{5}}{2}\right)^n$. La complexité est donc exponentielle.
On remarque d'ailleurs que $T(n)$ est équivalente à la suite de Fibonacci elle-même.

Question 3

```
def fibonacci_rec2(n):
    """fibonacci recursive avec renvoi de deux valeurs de u_n"""
    if n==0:
        return (0,1)
    elif n==1:
        return (1,1)
    else:
        X = fibonacci_rec2(n-1)
        return (X[1],X[0]+X[1])
```

Si nous notons maintenant $X(n)$ le nombre d'opérations nécessaires pour calculer `fibonacci_rec2(n)`, nous avons $X(0)=X(1)=1$ et si $n>1$, $X(n)=X(n-1)+k$, où k est une constante. C'est une simple suite arithmétique, donc la complexité de `fibonacci_rec2` est linéaire.

Exercice 4 – Les tours de Hanoï

Question 1 *Résoudre le problème pour $n=1$.*

Question 2 *On suppose que l'on sait résoudre le problème pour $n-1$ disques. Donner alors une résolution du problème pour n disques.*

Question 3 *Écrivez maintenant un programme **récursif** résolvant ce problème. Il s'agit d'écrire un programme* `hanoi` *ayant quatre arguments A, B, C et n :*

- *le premier, A, est le piquet sur lequel se trouvent les disques au départ ;*
- *le second, B, est le piquet "de transition" ;*

- *le troisième, C, est le piquet sur lequel on veut récupérer les disques à la fin ;*
- *et enfin le quatrième et dernier argument, n, est le nombre de disques.*

On lui demande également d'afficher tous les déplacements effectués sous forme de chaînes : la chaîne ''a->b'' *signifie par exemple que le disque du dessus du piquet A est déplacé sur le piquet B. On utilisera le programme d'impression des déplacements suivant :*

```
def deplace (x, y) :
    print (x + ''->'' + y + ''\n'')
```

Attention : A, B, C, x et y sont ici de type `string`.

Écrire le programme `hanoi`. *Attention : il s'agit d'un programme récursif, qui va donc être très court ! En aucun cas le programme en lui-même ne fait apparaître les opérations effectuées.*

Correction **Question 4** Trivial : on déplace le disque du piquet A et on le place sur le piquet C.

Question 5 Si l'on sait transférer $n-1$ disques, il suffit de transférer $n-1$ disques vers le piquet B, puis le n-ème disque vers le piquet C et enfin de transférer les $n-1$ disques du piquet B vers le piquet C.

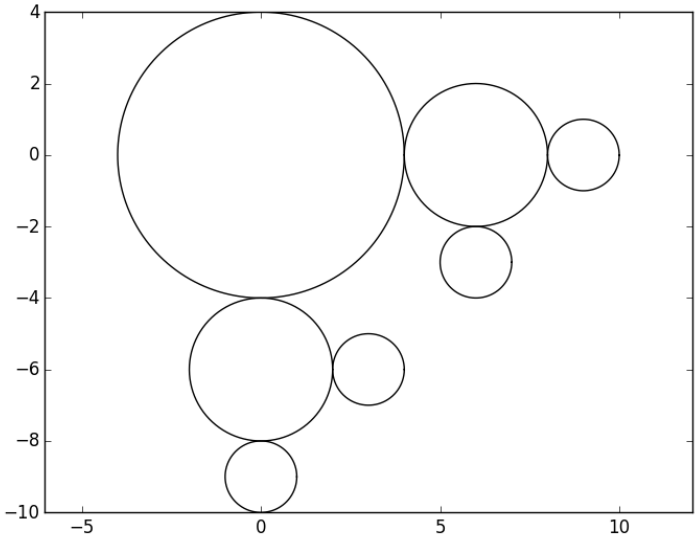
Question 6

```
def hanoi (A, B, C, n) :
    if n == 1 :
        deplace (A,C)
    else :
        hanoi (A,C,B,n-1)
        hanoi (A,B,C,1)
        hanoi (B,A,C,n-1)
```

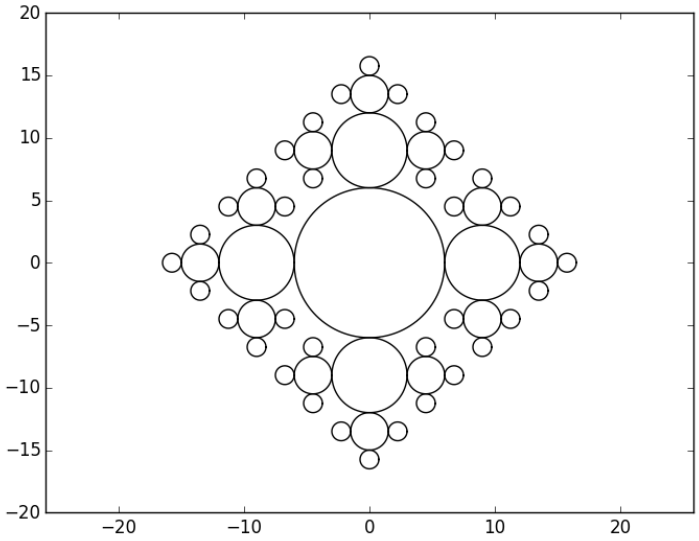
Exercice 5 – Faisons des Bulles

Question 1 *Ecrire une fonction* `cercle` *d'arguments* `x`, `y` *et* `r` *qui trace le cercle de centre le point $A(x,y)$ et de rayon r (supposé strictement positif).*

Question 2 *Ecrire une fonction récursive* `bubble` *d'arguments* `x`, `y`, `r` *et* `n`. *Elle effectuera la construction pour un nombre* `n` *d'étapes en ayant pour figure de base le cercle de centre $A(x,y)$ et de rayon* `r` *(supposé strictement positif). À chaque étape, le rayon du cercle est divisé par 2.*



Question 3 *Ecrire une fonction récursive* `bubbleComplet` *d'arguments* `x`, `y`, `r`, `n` *et une chaîne de caractères* `position`. *Elle effectuera la construction ci-dessous pour un nombre* `n` *d'étapes en ayant pour figure de base le cercle de centre $A(x,y)$ et de rayon* `r` *(supposé strictement positif).*



Correction

```
### tracé d'un cercle
import matplotlib.pyplot as plt
import numpy as np

def cercle(x,y,r):
    """tracé d'un cercle de centre A(x,y) et de rayon r"""
    angles=np.linspace(0,2*np.pi,500)
    les_X=[x+r*np.cos(angle) for angle in angles]
    les_Y=[y+r*np.sin(angle) for angle in angles]
    plt.plot(les_X,les_Y)
    plt.axis("equal")
    plt.show()

def cerclesRec(x,y,r,n):
    """tracé des cercles àdroite et en bas du cercle de centre A(x,y) et de rayon r"""
    cercle(x,y,r)
    if n>0:
        cerclesRec(x+1.5*r,y,r/2,n-1)
        cerclesRec(x,y-1.5*r,r/2,n-1)

def cerclesRec_2(x,y,r,n,position):
    """tracé des cercles autour du cercle de centre A(x,y) et de rayon r"""
    cercle(x,y,r)
    if n>0:
        if position=='centre':
            cerclesRec_2(x,y+1.5*r,r/2,n-1,'haut')
            cerclesRec_2(x-1.5*r,y,r/2,n-1,'gauche')
            cerclesRec_2(x+1.5*r,y,r/2,n-1,'droite')
            cerclesRec_2(x,y-1.5*r,r/2,n-1,'bas')
        if position=='droite':
            cerclesRec_2(x,y+1.5*r,r/2,n-1,'haut')
            cerclesRec_2(x+1.5*r,y,r/2,n-1,'droite')
            cerclesRec_2(x,y-1.5*r,r/2,n-1,'bas')
        if position=='bas':
            cerclesRec_2(x-1.5*r,y,r/2,n-1,'gauche')
            cerclesRec_2(x+1.5*r,y,r/2,n-1,'droite')
            cerclesRec_2(x,y-1.5*r,r/2,n-1,'bas')
        if position=='gauche':
            cerclesRec_2(x-1.5*r,y,r/2,n-1,'gauche')
            cerclesRec_2(x,y+1.5*r,r/2,n-1,'haut')
            cerclesRec_2(x,y-1.5*r,r/2,n-1,'bas')
        if position=='haut':
            cerclesRec_2(x-1.5*r,y,r/2,n-1,'gauche')
            cerclesRec_2(x+1.5*r,y,r/2,n-1,'droite')
            cerclesRec_2(x,y+1.5*r,r/2,n-1,'haut')
```