Langages et Programmation

Calculabilité, Récursivité, Modularité

Exercice 1.

En mathématiques, la méthode de Héron ou méthode babylonienne est une méthode efficace d'extraction de racine carrée, c'est-à-dire de résolution de l'équation $x^2 = a$, avec a positif. Elle porte le nom du mathématicien Héron d'Alexandrie, qui l'expose dans le tome I de son ouvrage Metrica (Les métriques), découvert seulement en 1896 mais certains calculs antérieurs, notamment égyptiens, semblent prouver que la méthode est plus ancienne.[1]

Soit a le nombre dont on recherche la racine carrée et $approx$ une première valeur approchée de $\sqrt{a}$, la formule suivante nous fournit une valeur beaucoup plus précise que $approx$, que nous nommerons r_1:

$$r_1 = \frac{approx + \frac{a}{approx}}{2}$$

Par exemple, on devine facilement que la racine carrée de 5 ($\sqrt{5}$) est un nombre compris entre 2 et 3, puisque $2^2 < 5 < 3^2$ et intuitivement on se doute que $\sqrt{5}$ est plus près de 2 que de 3 (car 4 est plus près de 5 que 9) donc on peut raisonnablement prendre pour nombre de départ $approx = 2$.

1. Calculer r_1 en utilisant la formule ci-dessus avec $a = 5$ et $approx = 2$.
   Comparer les carrés de $approx$ et de $r_1$et vérifier qu'on a bien obtenu une valeur approchée plus précise de $\sqrt{5}$ que la valeur d' $approx$.
2. Appliquer à nouveau la même formule pour calculer $r_2 = \frac{r_1 + \frac{a}{r_1}}{2}$ et faire la même vérification.

Considérons maintenant la suite suivante définie par récurrence pour déterminer une valeur approchée de la racine carrée d'un nombre (positif) a:

$$\begin{cases} r_0 = approx & \text{où } approx \text{ est un nombre de départ si possible choisi « assez proche » de } \sqrt{a} \\ r_{n+1} = \dfrac{r_n + \frac{a}{r_n}}{2} & pour\ n \geq 1 \end{cases}$$

3. Compléter la fonction `heron(a, n, approx)` ci-dessous qui renvoie la valeur approchée de $\sqrt{a}$ obtenue à l'étape n (autrement dit la valeur de r_n) par la méthode de Héron calculée à partir d'un nombre de départ donné.
   La fonction admet trois paramètres : la valeur de `a`, le rang `n` et une valeur de départ `approx`:

```
def heron(a, n, approx):
    ''' (int, int, int) -> float
        Renvoie le terme de rang n (rn) de la valeur approchée de racine de a
        par la méthode de Héron, en utilisant un nombre de départ approx
    '''
    # rn est initialisé à la valeur de approx
    rn = approx.
    # boucle jusqu'à trouver rn
    for i in range(n                        #...à compléter
        #... à compléter
    return rn
```

4. Ecrire en Python une fonction équivalente en utilisant une approche récursive.

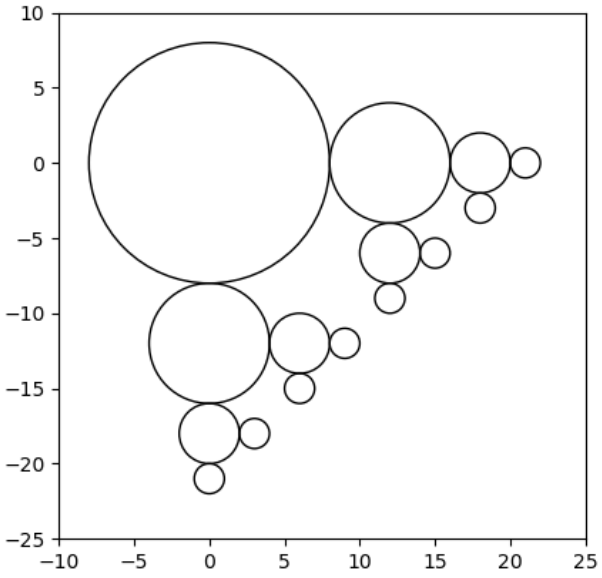
[1] Source : wikipedia

Exercice 2.

On a réalisé un programme récursif pour construire une figure géométrique définie ainsi:

1. Tracer un cercle (appelé ‘grand cercle’).
2. Tracer deux autres cercles (appelés ‘petits cercles’) tels que :
   a. le rayon des petits cercles est la moitié du rayon du grand cercle ;
   b. les petits cercles sont tangents au grand cercle (rappel : deux cercles sont dit tangent si ils n’ont qu’un seul point commun) ; et
   c. les deux droites passant par le centre du grand cercle et les centres des petits cercles sont chacune parallèle à un axe du repère.
3. Ces deux petits cercles deviennent à leur tour “grand cercle” pour poursuivre la figure.

L’exécution du programme affiche la figure ci-contre:



Le code Python de ce programme est donné ci-dessou:

```
import matplotlib.pyplot as plt

fig = plt.gca()  # création d'un objet 'figure' fig

def mon_cercle(x, y, r):
    """ cercle de centre (x, y) et de rayon r """
    # création du cercle
    cir = plt.Circle ((x , y), radius=r, fill = False)
    # ajout du cercle à la figure
    fig.add_artist(cir)


def cercles_recursifs(x, y, r):
    """ construction de la figure """
    mon_cercle(x, y, r )
    if r > 1 :
        cercles_recursifs(x + 3 * r / 2, y, r / 2)
        cercles_recursifs(x, y - 3 * r / 2, r /2)

# appel de la fonction cercles_recursifs
cercles_recursifs(0, 0, 8)

# définition des axes et aspect de la figure
plt.xlim(-10, 25)
plt.ylim(-25, 10)
fig.set_aspect('equal', adjustable='box')

# affichage de la figure
plt.show()
```

1. Que désigne le mot « `plt` » qui apparait plusieurs fois dans ce programme ?
2. Expliquer pourquoi ce programme est considéré comme récursif.
3. Qu’est-ce qui garantit que ce programme se termine ?
4. Décrire la figure obtenue (le nombre de cercles obtenus et leur rayons respectifs) quand on change l’instruction `cercles_recursifs(0,0,8)` par `cercles_recursifs(0,0,32)` dans le programme?

Exercice 3.

Pour chacun des problèmes suivant, indiquer s'il est décidable ou indécidable (aucune justification):

- Problème 1 : Déterminer pour tout nombre s'il est premier ou pas.
- Problème 2 : Déterminer si la politique du Président Trump est bonne pour les Etats-Unis ou pas.
- Problème 3 : Déterminer pour tout programme informatique s'il s'arrête ou pas.
- Problème 4 : Déterminer pour toute liste de nombres si elle est triée en ordre décroissant ou pas.

Exercice 4.

On donne la définition suivante utilisée dans cet exercice: « Deux fonctions f() et g() sont dites équivalentes si pour tout paramètres d'entrée e, f(e) et g(e) donnent le même résultat : f(e) = g(e). »

1. Analyser les deux fonctions `f` et `g` données ci-dessous. Expliquer en quelques phrases le fonctionnement de chacune d'elles. Les deux fonctions sont-elles équivalentes ? Justifier.

```
def f(n) :
    assert (type(n) == int) & (n >= 0), 'n doit être un entier positif'
    if n == 0:
        return True
    elif n == 1:
        return False
    else:
        return f(n - 2)

def g(n) :
    assert (type(n) == int) & (n >= 0), 'n doit être un entier positif'
    return n%2 == 0
```

On cherche maintenant à écrire un programme `equivalent()` pour déterminer d'une façon systématique l'équivalence de deux programmes `p1` et `p2` pour certaines valeurs `x`. Les deux programmes `p1` et `p2` et les valeurs `x` sont des paramètres du programme `equivalent()` qui renvoie:

- `True` si les programmes `p1` et `p2` (qui utilisent les données `x`) produisent le même résultat ; et
- `False` si les résultats sont différents.

Admettons qu'on a déjà écrit entièrement la fonction `equivalent()` dont le code est en partie donné ici:

```
def equivalent(p1, p2, x):
    ''' p1 et p2 sont des fonctions, x est un paramètre -> bool
        renvoie True si p1 et p2 sont équivalentes
    '''
    if …… :                                  # teste si p1(x) == p2(x)
        return True
    else:
        return False
```

On réalise maintenant les programmes `boucle_infine()` et `test(p, x)` faisant appel à `equivalent()`.

```
def boucle_infine():
    while True :
        print('et ca continue et encore et encore…')

def test(p, x):
    ''' p est une fonction, x est un paramètre -> bool '''
    if equivalence(p, boucle_infinie, x):
        return False
    else:
        return True
```

2. Que renvoie la fonction `test(p, x)` quand `p` est un programme qui s'arrête? Même question quand `p` ne s'arrête jamais? En déduire ce que fait le programme `test()`? Que peut-on conclure sur le programme `equivalent()`?