Exercice 1

Fonctions pair / impair

On donne le script de la fonction `pair` :

```
def pair(N):
    if N==0 :
        return True
    else :
        return pair(N-2)
```

1.1 Que renvoie `pair(4)` ? Expliquez en faisant le tracé du résultat (représenter la pile d'appels).

1.2 Comment se comporte le programme si on fait : `pair(5)` ?

1.3 Modifier alors le script proposé pour tenir compte de ce problème (il faudra modifier la condition de base).

1.4 Ecrire le script de la fonction `impair`, qui renvoie True si l'argument est ...impair.

Exercice 2

Fonction mystère

```
def f(x,y):
  if x == y:
    return x
  elif x < y:
    return f(x,y-x)
  else:
    return f(x-y,y)
```

2.1 Sans executer le code, indiquer ce que renvoient f(5, 15) et f(8, 29)

2.2 Indiquer plus généralement ce que calcule f(x, y).

Exercice 3

Algorithme de calcul du PGCD d'Euclide

3.1 Historique

En mathématiques, l'algorithme d'Euclide est un algorithme qui calcule le plus grand commun diviseur (PGCD) de deux entiers, c'est-à-dire le plus grand entier qui divise les deux entiers, en laissant un reste nul.

l'algorithme d'Euclide est l'un des plus anciens algorithmes. Il est décrit dans le livre VII (Proposition 1-3) des Éléments d'Euclide (vers 300 avant JC). Cela correspond à une adaptation de la méthode naïve de calcul de la division euclidienne. *(source wikipedia)*

3.2 Principe

- Il s'agit de divisions en cascade de A par B : les résultats de l'une servent à poser la suivante :
  - Le diviseur B devient le dividende ; et
  - Le reste r1 devient le diviseur.
- Arrêt lorsque le reste de la division est nul.
- Le reste trouvé avant le reste nul est le PGCD des nombres A et B.

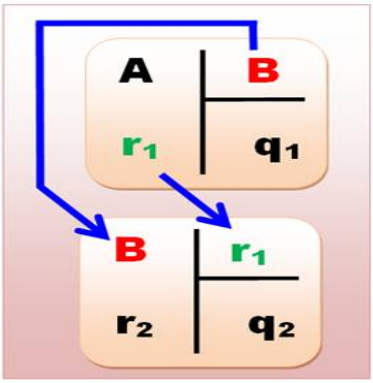

FIGURE 1 – division euclidienne

3.2.1 Compléter le tableau suivant montrant le tracé du calcul du PGCD de 264 par 228 :

division de A	par B	q	r
...			
...			
...			

3.2.2 Ecrire le script en python correspondant à la programmation itérative de cet algorithme.

```
def PGCD_it(a,b):
    """
    a et b sont des entiers, a > b
    euclide retourne un entier qui est le PGCD de a et b
    """

    ...
    ...
    ...
    ...
    ...
    ...
```

3.3 script récursif

```
def PGCD(a,b):
  """
  PGCD : entier correspondant au plus grand diviseur commun de a par b
  a et b : entiers tels que a > b
  """
  if b == 0 : return a
  else:
    c = a % b
    return PGCD(b,c)
```

3.3.1 Tracer le calcul de PGCD(264,228) en représentant la pile des appels successifs.

3.3.2 Prouver la terminaison de cette fonction recursive.

Exercice 4

Les vaches de Narayana

Une vache donne naissance à une autre tous les en debut d'année, qui elle-même donne naissance à une autre chaque année à partir de sa quatrième année.

Partant d'une vache qui vient de donner naissance (v1 = 1), les termes successifs de la suite (vi) sont donc :

$$1, 1, 1, 2, 3, 4, 6, \ldots$$

4.1 Ecrire une fonction récursive v(i) qui renvoie le i-eme terme de la suite.

4.2 La complexité est-elle exponentielle ou linéaire ?

Exercice 5

La fonction de Mc Carthy

La fonction 91 de McCarthy est une fonction récursive définie par McCarthy dans son étude de propriétés de programmes récursifs, et notamment de leur vérification formelle. *(https ://fr.wikipedia.org/wiki/Fonction_91_de_McCarthy)*

C'est une fonction dont l'image est égale à 91 pour tout entier n < 102.

Elle est définie pour tout $n \in \mathbb{N}$.

$$f(n) = \begin{cases} n - 10 & \text{si} \quad n > 100 \\ f(f(n + 11)) & \text{sinon} \end{cases}$$

5.0.1 Ecrire en python le script de cette fonction recursive

5.0.2 Représenter la pile d'appels pour f(99) par cette fonction. Converge t-elle bien vers 91 ?

Exercice 6

Prolongement : Une fonction x + 1/x

Considérons la suite u_n

$$n \in \mathbb{N}$$

définie par

$$u_0 = 1$$

et la relation de recurrence :

$$u_{n+1} = u_n + \frac{1}{u_n}$$

6.1 Compléter le script récursif de cette fonction que l'on nommera `u_rec`. Ecrire egalement sont docstring.

```
def u_rec(n):
    """...
          ...
          ...
          ...
          ...
          ...
    """
    ...
    ...
    ...
    ...
```

On utilisera les conventions suivantes pour le calcul de la complexité :

Opérations	Poids
+, −, ×, ÷	1 unité de temps
Affectation	1 unité de temps
Appel de fonction	1 unité de temps
Comparaison	1 unité de temps

6.1.1 Quelle est la loi de recurence sur le nombre d'instructions T(n) en fonction de T(n-1) pour cette fonction.

6.1.2 En déduire la complexité asymptotique O(g(n)).

6.2 Dans votre script, où pourrait-on ajouter un test d'assertion pour protéger la fonction d'une entrée non conforme (par exemple avec n<0)? S'agit-il d'une pré-condition ou d'une post-condition? Ecrire l'instruction de ce test d'assertion.

6.3 On propose un autre script pour cette fonction :

```
def u_rec (n) :
    if n==0:
        return 1
    else :
```

```
5
6           x=u_rec (n-1) # variable locale
7           return x+1/x
```

6.3.1 Cette fonction, est-elle plus efficace ? C'est à dire, est-elle de complexité inferieure ? Justifiez rapidement.

Exercice 7

Exponentiation

Etudions l'exponentiation à travers deux exemples.

```
1  def exp1(n,x) :
2    """
3    programme qui donne x^n en sortie sans utiliser **
4    n : entier
5    x : reel
6    exp1 : reel
7    """
8    acc=1
9    for i in range(1,n+1):
10     acc*=x
11   return acc
12
13 def exp2(n,x):
14     """
15     n : entier
16     x : reel
17     exp2 : reel
18     """
19     if n==0 : return 1
20     else : return exp2(n-1,x)*x
```

1. Combien de produits sont necessaires pour calculer une puissance n-ième avec la fonction `exp1` ?
2. Pour la fonction `exp2` : Soit u_n le nombre de produits nécessaires pour calculer une puissance n-ième. Quelle est la relation de récurrence vérifiée par u~n+1 ?

$$u_{n+1} = \ldots$$

3. En déduire la complexité pour ces 2 fonctions.

Exercice 8

Les tours de Hanoï

Voir le cours en ligne sur la compléxité. L'exemple y est longuement traité.

8.1 Principe

On considère trois tiges plantées dans une base. Au départ, sur la première tige sont enfilées N disques de plus en plus étroits. Le but du jeu est de transférer les N disques sur la troisième tige en conservant la configuration initiale.

8.2 algorithme récursif

L'algorithme récursif pour ce problème est étonnament réduit :

```
def hanoi(N,d,i,a):
    """N disques doivent être déplacés de d vers a
    Params:
    N : int
        nombre de disques
    d: int
        depart (vaut 1 au debut)
    i: int
        intermediaire (vaut 2 au debut)
    a: int
        fin (vaut 3 au debut)
    Exemple:
    lancer avec
    >>> hanoi(3,1,2,3)
    """
    if N==1 :
        print('deplacement de {} vers {}'.format(d,a))
    else:
        hanoi(N-1,d,a,i)
        hanoi(1,d,i,a)
        hanoi(N-1,i,d,a)
```

Résultat

```
>>> hanoi(3,1,2,3)
deplacement de 1 vers 3
deplacement de 1 vers 2
deplacement de 3 vers 2
deplacement de 1 vers 3
deplacement de 2 vers 1
deplacement de 2 vers 3
deplacement de 1 vers 3
```

8.2.1 Vérifier que pour N = 2 disques, il y a 3 déplacements, que pour 3 disques, il y en a 7, et que pour 4 disques, il y en a 15.

8.2.2 Proposez une loi de recurence entre le nombre de déplacements T(N) pour N disques, et le nombre de déplacements T(N-1) pour N-1 disques.

8.2.3 Cette loi, est-elle conforme à celle que l'on aurait déduite de l'etude de la complexité pour l'algorithme recursif?