Exercice 1

Suite récurente et fonction recursive

1.1 Script recursif

Donner dans chacun des cas suivants, le scrip récursif qui calcule le terme de rang n de la suite :

a. $u_n = 2 \times u_{n-1} + 1, u_0 = 0$

b. $v_n = \frac{1}{v_{n-1}}, v_0 = 1$

c. $f_n = n \times f_{n-1}$

d. $fb_n = fb_{n-1} + fb_{n-2}$

1.2 Script itératif

Pour la série (u_n) de la question a., donner le script de la fonction itérative qui calcule le terme de rang n de la suite

Exercice 2

Fonctions pair / impair

On donne le script de la fonction `pair` :

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

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

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

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

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

Exercice 3

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)
```

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

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

Exercice 4

Les vaches de Narayana

Une vache de plus de 3 ans, donne naissance à une autre tous les ans, en debut d'année, qui elle-même donne naissance à une autre chaque année à partir de sa quatrième année (elle a 3 ans et 1 jour).

Partant d'une vache qui vient de naitre ($v_1 = 1$), les termes successifs de la suite (v_i), donnant le nombre de vaches, sont donc :

$$1, 1, 1, 2, 3, 4, 6, ...$$

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 son docstring (entre guillemets, au début).

6.2 Compléxité

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

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

6.2.2 En déduire la complexité en notation de Landau. Définir la classe de complexité.

6.3 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) ? Ecrire l'instruction de ce test d'assertion.

6.4 On propose un autre script pour cette fonction :

```
def u_rec (n) :
    if n==0:
        return 1
    else :
        x=u_rec (n-1) # variable locale
        return x+1/x
```

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

Exercice 7

Exponentiation (non donné en 2024)

Etudions l'exponentiation à travers deux exemples.

```
def exp1(n,x) :
  """
  programme qui donne x^n en sortie sans utiliser **
  n : entier
  x : reel
  exp1 : reel
  """
  acc=1
  for i in range(1,n+1):
    acc*=x
  return acc

def exp2(n,x):
    """
```

```
    n : entier
    x : reel
    exp2 : reel
    """
    if n==0 : return 1
    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.

Partie 2

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)
```

```
20          hanoi(1,d,i,a)
21          hanoi(N-1,i,d,a)
```

Résultat

```
1  >>> hanoi(3,1,2,3)
2  deplacement de 1 vers 3
3  deplacement de 1 vers 2
4  deplacement de 3 vers 2
5  deplacement de 1 vers 3
6  deplacement de 2 vers 1
7  deplacement de 2 vers 3
8  deplacement de 1 vers 3
```

8.2.1 Vérifier (experimentalement) 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 Retrouver la loi T(n) en fonction de T(n-1) en analysant le script de la fonction.

Exercice 9

Dessin recursif

Soit la relation de recurence suivante :

$$u_{n+1} = \frac{8u_n}{9} + \frac{1}{9}$$

Le premier terme de la suite est $u_0 = 0$

9.1 Script python

Ecrire en python le script de la fonction recursive correspondante, qui calcule au rang `n` le nombre u_n. Appeler cette fonction `f`.

9.2 Application géométrique

On considère un carré de côté 1. On le partage en 9 carrés égaux, et on colorie le carré central. Puis, pour chaque carré non-colorié, on réitère le procédé. On note u_n l'aire coloriée après l'étape `n`. La suite u_n est $u_{n+1} = \frac{8u_n}{9} + \frac{1}{9}$

a. Représenter le dessin obtenu à l'ordre 2.

b. Déterminer expérimentalement la limite de la suite u_n ?

Exercice 10

Longueur d'une liste

10.1 algorithme itératif :

La fonction suivante calcule la longueur d'une chaine de caractères `seq` passée en argument.

```python
def len_iterative(seq):
    """
    Return the length of a list (iterative)
    """
    count = 0
    for elt in seq:
        count = count + 1
    return count
```

Réaliser la **preuve** de cet algorithme. C'est à dire montrer que ce programme va bien retourner la longueur de la chaine de caractères.

10.2 algorithme récursif

a. Montrer que la relation de récurence $u_{n+1} = 1 + u_n$ permet de compter de 1 en 1.

b. Soit la chaine de caractères `s = "abcd"`. On veut supprimer le premier caractère de `s` à l'aide d'un *slice* python. Quelle est l'instruction python correspondante ?

c. Ecrire le script python de l'algorithme récursif

Aide pour l'écriture de l'algorithme recursif : la fonction récursive s'appelera `len_recursive`, et aura pour argument `seq`. Si on veut passer en argument la liste `seq` de laquelle on retire le premier élément, on fait : `len_recursive(seq[1:])`. Il faudra alors s'inspirer de la relation de récurence suivante lors de l'appel recursif :

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

d. Prouver la terminaison de l'algo recursif.

Exercice 11

Sujet Métropole Sept 2 2021 Exercice 4

Cet exercice porte sur la programmation en général et la récursivité en particulier.

On s'intéresse dans cet exercice à un algorithme de mélange des éléments d'une liste.

1. Pour la suite, il sera utile de disposer d'une fonction echange qui permet d'échanger dans une liste lst les éléments d'indice `i1` et `i2`.

Expliquer pourquoi le code Python ci-dessous ne réalise pas cet échange et en proposer une modification.

```
def echange(lst, i1, i2):
 lst[i2] = lst[i1]
 lst[i1] = lst[i2]
```

2. La documentation du module random de Python fournit les informations ci-dessous concernant la fonction `randint(a,b)` :

```
Renvoie un entier aléatoire N tel que a <= N <= b.
```

Parmi les valeurs ci-dessous, quelles sont celles qui peuvent être renvoyées par l'appel `randint(0, 10)` ?

```
0   1   3.5   9   10   11
```

3. Le mélange de Fischer Yates est un algorithme permettant de permuter aléatoirement les éléments d'une liste. On donne ci-dessous une mise en œuvre récursive de cet algorithme en Python.

```
from random import randint

def melange(lst, ind):
  print(lst)
  if ind > 0:
    j = randint(0, ind)
    echange(lst, ind, j)
    melange(lst, ind-1)
```

a. Expliquer pourquoi la fonction melange se termine toujours.

b. Lors de l'appel de la fonction melange, la valeur du paramètre `ind` doit être égale au plus grand indice possible de la liste `lst`.

Pour une liste de longueur `n`, **quel est le nombre d'appels récursifs de la fonction melange effectués, sans compter l'appel initial ?**

c. On considère le script ci-dessous :

```
lst = [v for v in range(5)]
melange(lst, 4)
```

On suppose que les valeurs successivement renvoyées par la fonction `randint` sont 2, 1, 2 et 0.

Les deux premiers affichages produits par l'instruction `print(lst)` de la fonction melange sont :

```
[0, 1, 2, 3, 4]
[0, 1, 4, 3, 2]
```

Donner les affichages suivants produits par la fonction melange.

d. Proposer une version itérative du mélange de Fischer Yates.

On rappelle que la fonction `range` accepte 1 à 3 paramètres :

- avec un seul paramètre n, le variant prend successivement les valeurs de tous les entiers compris entre 0 et n-1
- avec 3 paramètres, le variant i prend toutes les valeurs entières comprises entre le 1er (inclus) et 2e paramètre (exclus). Le dernier paramètre sert à préciser le sens (croissant ou décroissant). Par exemple :

```
for i in range(4, 1, -1):
    print(i)
# affiche
4
3
2
```

Exercice 12

(non donné en 2024) Algorithme de calcul du PGCD d'Euclide

12.1 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)*

12.2 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.

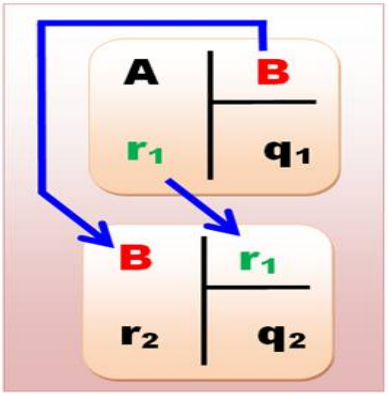

FIGURE 1 – division euclidienne

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

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

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

```
1 def PGCD_it(a,b):
2     """
3     a et b sont des entiers, a > b
4     euclide retourne un entier qui est le PGCD de a et b
5     """
```

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

12.3 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)
```

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

12.3.2 Prouver la terminaison de cette fonction recursive.

Exercice 13

(non donné en 2024) Dichotomie recursive

Soit `tab` un tableau non vide d'entiers triés dans l'ordre croissant et `n` un entier.

La fonction `chercher` ci-dessous doit renvoyer un indice où la valeur `n` apparaît dans `tab` si cette valeur y figure et `None` sinon.

Les paramètres de la fonction sont :

- `tab`, le tableau dans lequel s'effectue la recherche ;
- `x`, l'entier à chercher dans le tableau ;
- `i`, l'indice de début de la partie du tableau où s'effectue la recherche ;
- `j`, l'indice de fin de la partie du tableau où s'effectue la recherche.

L'algorithme demandé est une recherche dichotomique récursive.

Recopier et compléter le code de la fonction `chercher` suivante :

```
def chercher(tab, x, i, j):
    '''Renvoie l'indice de x dans tab, si x est dans tab,
    None sinon.
    On suppose que tab est trié dans l'ordre croissant.'''
    if i > j:
        return None
    m = (i + j) // ...
    if ... < x:
        return chercher(tab, x, ... , ...)
```

```
    elif tab[m] > x:
        return chercher(tab, x, ... , ...)
    else:
        return ...

# Exemples :


>>> chercher([1, 5, 6, 6, 9, 12], 7, 0, 10)

>>> chercher([1, 5, 6, 6, 9, 12], 7, 0, 5)

>>> chercher([1, 5, 6, 6, 9, 12], 9, 0, 5)
4
>>> chercher([1, 5, 6, 6, 9, 12], 6, 0, 5)
2
```