DS n°4

I. Fonctions récursives

1 - Fonction mystère

On s'intéresse à la fonction mystère ci-dessous.

```
def mystere(x, n):
    if n == 0:
        return 1
    elif n < 0:
        return 1 / mystere(x, -n)
    else:
        return x * mystere(x, n-1)
```

☐ **Q 1 -** Quel est le résultat de la fonction `mystère (4,3)`. On expliquera le calcul à l'aide d'une pile des appels récursifs.

☐ **Q 2 -** Quel est le résultat de la fonction `mystère (4,-2)`. On expliquera le calcul à l'aide d'une pile des appels récursifs.

☐ **Q 3 -** Proposer une écriture de cette fonction de manière non récursive.

2 - Inversion d'une liste

On rappelle que `L[:i]` contient une copie de la liste `L` des éléments indicés de 0 à i-1. De même `L[i:]` contient une copie de la liste `L` des éléments indicés à partir de i.

☐ **Q 4 -** Écrivez une fonction récursive `inverse(L)` qui retourne une liste `L` inversée.

☐ **Q 5 -** Proposer une version non récursive appelée `inv2(L)` retournant une nouvelle liste inversée, en utilisant la méthode `.pop()`.

On considère le code suivant :

```
L = [1,3,2,4]
L2 = inv2(L)
print(L)
print(L2)
```

☐ **Q 6 -** Qu'affiche la console lors de l'exécution de ces lignes ?

☐ **Q 7 -** Proposer une version non récursive mutant sur place la liste `L` pour obtenir une liste inversée.

3 - Modification d'une liste

On considère la fonction suivante :

```
def surprise(L):
    for i in range(len(L)) :
        for j in range(i+1,len(L)) :
            if L[i] == L[j] :
                return True
    return False
```

☐ **Q 8 -** Que retourne la fonction `surprise([1,3,2,5,2])`.

☐ **Q 9 -** Déterminer en la justifiant la complexité de cet algorithme.

☐ **Q 10 -** Proposer une fonction `no_double(L)` qui renvoit une liste issue de `L` sans doublon.

II. Algorithme de Fibonacci

Au XIIIe siècle, le mathématicien Fibonacci a proposé un problème célèbre sur la croissance d'une population de lapins :

Un couple de lapins adultes donne naissance à un nouveau couple chaque mois. Un couple de lapins devient adulte après un mois. Supposons que les lapins ne meurent jamais et qu'un couple commence seul au départ. On note $F(n)$ le nombre total de couples de lapins après n mois.

- Au mois 1 : 1 couple de lapins (initial).
- Au mois 2 : 1 couple (encore trop jeune pour se reproduire).
- Au mois 3 : 2 couples (le premier couple a eu une nouvelle paire).
- Au mois 4 : 3 couples (le couple initial a encore donné naissance).
- Au mois 5 : 5 couples, etc.

On obtient ainsi la célèbre **suite de Fibonacci** :

$$F(n) = F(n-1) + F(n-2)$$

avec $F(1) = 1$ et $F(2) = 1$.

On considère le code suivant :

```python
def fibo1(n):
    if n <= 1:
        return n
    return fibo1(n-1) + fibo1(n-2)
```

☐ **Q 11 -** Représenter l'arbre des appels récursifs pour `fibo1(4)`

☐ **Q 12 -** Déterminer en la justifiant, la complexité d'un tel algorithme.

On se propose d'utiliser l'algorithme suivant utilisant un dictionnaire. On rappelle qu'un dictionnaire est une collection d'éléments non ordonnés. L'ajout d'un élément au dictionnaire `memo`, associé à une clé `n` se fait simplement par : `memo[n]=elt`.

```python
def fibo2(n, memo={}):
    if n in memo:
        return memo[n]
    if n <= 1:
        return n
    memo[n] = fibo2(n-1, memo) + fibo2(n-2, memo)
    return memo[n]
```

☐ **Q 13 -** Représenter l'arbre des appels récursifs pour `fibo2(4)`. Conclure sur la complexité d'un tel algorithme.

III. Le tri du gnome

Cet algorithme a été d'abord proposé en 2000 sous le nom de "tri stupide" par le Dr. Hamid Sarbazi-Azad (Professeur à l'université de Sharif, une des plus grandes universités d'Iran) puis redécouvert plus tard par Dick Grune (le premier développeur de CVS) qui l'appela "gnome sort" (parce qu'il parait que c'est la méthode qu'utilisent les nains de jardin hollandais pour trier une rangée de pots de fleurs). L'algorithme est similaire au tri bulle. Dans le tri gnome, on commence par le début du tableau :

- on compare deux éléments consécutifs (i, i+1) tant que i+1 existe :
  - s'ils sont dans l'ordre on se déplace d'un cran vers la fin du tableau (`i` est incrémenté de 1)
  - sinon, on les permute et on se déplace d'un cran vers le début du tableau (`i` est décrémenté de 1) si cela est possible.

☐ **Q 14 -** Proposer une fonction `gnome(L)` associée au tri gnome.

☐ **Q 15 -** Pour le tableau $T = [4,2,9,1]$, indiquer la séquence des valeurs prises par T de votre algorithme

☐ **Q 16 -** Donnez la complexité du tri gnome au meilleur cas et au pire cas ? Justifiez votre réponse.

☐ **Q 17 -** Le tri gnome est-il un tri en place ou non ? Justifier.

☐ **Q 18 -** Le tri gnome est-il stable ou non ? Justifier votre réponse à l'aide d'un exemple.

I. Fonctions récursives

1 - Fonction mystère

Q1Résultat de mystère(4,3)

— Appels successifs :

$$\begin{aligned}\text{mystère}(4,3) &= 4 \times \text{mystère}(4,2)\\ &= 4 \times (4 \times \text{mystère}(4,1))\\ &= 4 \times (4 \times (4 \times \text{mystère}(4,0)))\\ &= 4 \times (4 \times (4 \times 1))\\ &= 64\end{aligned}$$

Q2Résultat de mystère(4,-2)

— On applique la règle pour n ¡ 0 :

$$\begin{aligned}\text{mystère}(4,-2) &= \frac{1}{\text{mystère}(4,2)}\\ &= \frac{1}{4 \times 4}\\ &= \frac{1}{16}\end{aligned}$$

Q3Version itérative

```
def mystere_iter(x, n):
    r = 1
    if n < 0:
        for _ in range(-n):
            r *= 1/x
    else:
        for _ in range(n):
            r *= x
    return r
```

2 - Inversion d'une liste

Q4Fonction récursive inverse

```
def inverse(L):
    if len(L) == 0:
        return []
    return [L[-1]] + inverse(L[:-1])
```

Q5 Version non récursive avec pop

```
def inv2(L):
    L2 = []
    for i in range(len(L)):
        L2.append(L.pop() )
    return L2
```

Q6 La fonction affiche None pour L et [4,2,3,1] pour L2

Q7

```
def inv2(L):
    for i in range(len(L)//2):
                L[i],L[n-1-i] = L[n-1-i],L[i]
```

3 - Détection et suppression des doublons

Q8 Résultat de surprise([1,3,2,5,2])

— La fonction retourne **True** car l'élément 2 est présent en double.

Q9 On reconnaît une écriture de deux boucles imbriquées effectuant, 1, 2 puis n opérations soit :

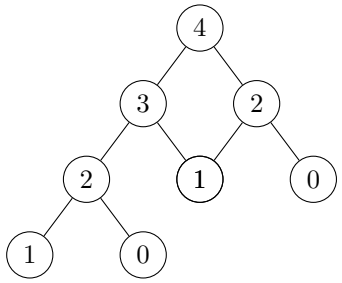
$$1+2+3+..n = \frac{n(n+1)}{2} = O(n^2)$$

Q10 Fonction pour supprimer les doublons

```
def no_double(L):
    L2 = []
    for elem in L:
        if elem not in L2:
            L2.append(elem)
    return L2
```

II. Algorithme de Fibonacci

Q11 Arbre des appels récursifs pour fibo1(4)



Q12 Complexité de fibo1 deux appels récursifs, on peut penser à une complexité $O(2^n)$ soit exponentielle.

En réalité, la complexité suit la même loi que la suite de Fibonnacci, elle suit une complexité donnée par son comportement asymptotique via le nombre d'or $O(\phi^n)$.

Q13 Optimisation avec mémorisation, la complexité est alors linéaire car il faut remplir la liste `memo` pour répondre au calcul demandé.

III. Tri du gnome

Q14 Implémentation

```
def gnome_sort(L):
    i = 0
    while i < len(L):
        if i == 0 or L[i] >= L[i}1]:
            i += 1
        else:
            L[i], L[i}1] = L[i}1], L[i]
            i -= 1
```

Q15 Suite des valeurs :

4,2,9,1

2,4,9,1

2,4,1,9

2,1,4,9

1,2,4,9

Q16 Complexité :

— Meilleur cas : $O(n)$ si la liste est presque triée.

— Pire cas : $O(n^2)$.

Q17 Le tri est en place : il ne nécessite pas de mémoire supplémentaire.

Q18 Le tri est-il stable ? oui car les élements sont maintenus dans l'ordre, ils ne sont déplacés que s'ils sont différents.