MPSI

DS informatique N°6 – Eléments de correction

I- *Prod*

Q1-
```
i=0
while i<len(L):
    print(L[i])
    i=i+1
```

Q2-
```
i=0
p=1
while i<len(L):
    p=p*L[i]
    i=i+1
```

Q3-
```
def prod(L):
    if len(L)>0:
        return L[0]*prod(L[1:])
    else:
        return 1

A=[3, 5, 1, 4, 2]
resultat=prod(A)
```

Q4- A chaque appel récursif, la liste passée en argument est réduite d'un élément. Le dernier appel récursif se produit lorsque len(L)=0. Il y a donc autant d'appels récursifs que d'éléments dans la liste. A chaque appel, il y a un nombre constant d'opération. La complexité d'un appel est en $O(1)$, donc la complexité de la fonction récursive appelée avec une liste de longueur n est $C(n) = n . O(1) = O(n)$.

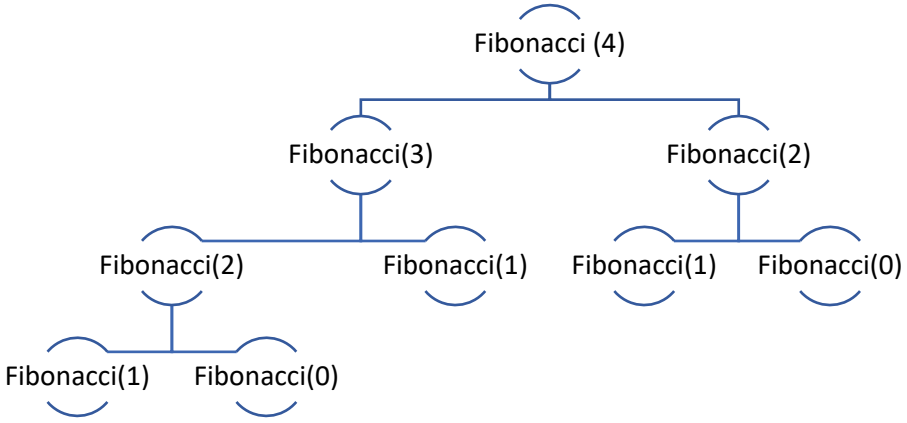
Q5- a chaque appel récursif, la valeur passée en argument est une nouvelle liste, contenant un élément de moins que la précédente. A chaque appel récursif, la variable locale L est affectée de cette nouvelle liste.
Si on exécute l'instruction `p=prod([3, 5, 1, 4, 2])` , cela impliquera la mémorisation des listes [3, 5, 1, 4, 2] , [5, 1, 4, 2] , [1, 4, 2] , [4, 2] , [2] , et []
Ainsi la complexité en mémoire est $C(n) = \sum_{i=1}^{n} i = \frac{n(n+1)}{2} = O(n^2)$

II- Fibonacci

Q1-
```
def Fibonacci(n):
    print("appel de la fonction avec l'argument : ",n) # utile uniquement pour la Q2
    if n==0:
        return 0
    elif n==1:
        return 1
    else:
        return Fibonacci(n-2)+Fibonacci(n-1)

res=Fibonacci(4)
```

Q2- l'appel de Fibonacci(4) implique l'arbre des appels récursifs suivant :



La fonction est donc appelée avec les arguments suivants :
appel de la fonction avec l'argument : 4
appel de la fonction avec l'argument : 2
appel de la fonction avec l'argument : 0
appel de la fonction avec l'argument : 1
appel de la fonction avec l'argument : 3
appel de la fonction avec l'argument : 1
appel de la fonction avec l'argument : 2
appel de la fonction avec l'argument : 0
appel de la fonction avec l'argument : 1

III- Miroir

```
def miroir(M):
    """
    Entrée : une matrice M représentant une image numérique.
    Sortie : la matrice de l'image transformée.
    """
    n = len(M) # Nombre de lignes de M.
    p = len(M[0]) # Nombre de colonnes de M.

    for i in range(n): # L'indice i va de 0 à n - 1,
    # c'est-à-dire que i parcourt les indices des lignes de M.
        for j in range(p): # L'indice j va de 0 à p - 1,
        # c'est-à-die que j parcourt les indices des colonnes de M.

            M[i].append( M[i][p - 1 - j] )

            # Lorsque que l'indice colonne vaut 0,
            # on rajoute l'élément M[i][p - 1],
            # c'est-à-dire le dernier élément de la ligne i initiale.
            # Lorsque que l'indice j vaut p - 1,
            # on rajoute l'élément M[i][0],
            # c'est-à-dire le premier élément de la ligne i initiale.

    return M
```

IV- Traitement de données de mesure

Q1 - nombre d'octets correspondant à 20 minutes d'enregistrement

8 caractères par lignes sur 8 bits = 8 octets par mesures
20 minutes à 2 Hz = 20*60*2 = 2400 mesures
soit 19200 octets en 20 minutes

Q2 nombre d'octets correspondant à la campagne de mesures

En 15 jours = 15 x 24 x 2 x 19 200 = 13 824 000 octets
soit 13.8 Mo
La carte mémoire est largement suffisante.

Q3 - Gain relatif d'espace mémoire

Un chiffre de moins donne 1 octet de moins par mesure soit 12.5% de moins (soit 1.8Mo de moins)

Q4 - Lecture du fichier

```
def liste_niveaux():
    fichier = open('donnees.txt','r')
    ligne = fichier.readline()
    liste_niveaux = []
    while ligne:
        ligne = fichier.readline()
        if ligne:
            liste_niveaux.append(float(ligne))
    fichier.close()
    return liste_niveaux
```

Q6 - Calcul de la moyenne

```
def moyenne(liste_niveaux):
    somme = 0
    for niveau in liste_niveaux:
        somme += niveau
    return somme / len(liste_niveaux)
```

Q7 - Calcul du maximum, du minimum Voir cours

Q8 - Détermination de l'indice du premier PND

```
def ind_premier_pzd(liste_niveaux):
    i = 0
    moy = moyenne(liste_niveaux)
    while: i < len(liste_niveaux)-1 and not(liste_niveaux[i] > moy and \
            liste_niveaux[i+1] < moy) :

        i = i + 1
    if liste_niveaux[i] > moy and i<len(liste_niveaux) :
        return i
    else:
        return -1
```

Q9 - Détermination de l'indice du dernier PND

```
def ind_dernier_pzd(liste_niveaux):
    i = len(liste_niveaux) - 2
    moy = moyenne(liste_niveaux)
    while i >0 and not(liste_niveaux[i] > moy and liste_niveaux[i+1] < moy):
        i = i - 1
    if liste_niveaux[i] > moy :
        return i
    else:
        return -2
```

Q10 - Compléter l'algorithme 1

```
def construction_successeurs(liste_niveaux):
    n = len(liste_niveaux)
    successeurs = []
    m = moyenne_precise(liste_niveaux)
    for i in range(n-1):
        if i== ind_premier_pzd( liste_niveaux[i :] ):
            successeurs.append(i+1)
    return successeurs
```

Q11 - Décomposition d'une liste de niveaux en liste de vagues

```
def decompose_vagues(liste_niveaux):
    m = moyenne(liste_niveaux)
    i_debut = ind_premier_pzd(liste_niveaux)
    i_fin = ind_dernier_pzd(liste_niveaux)
    vagues = []
    i = i_debut + 1
    while i < i_fin:
        i_vague = ind_premier_pzd(liste_niveaux[i:])
        vagues.append(liste_niveaux[i: i + i_vague + 1])
        i = i + i_vague + 1
    return vagues
```

Q12 - Caractérisation des vagues

```
def proprietes(liste_niveaux):
    vagues = decompose_vagues(liste_niveaux)
    liste = []
    for i in range(len(vagues)-1):
        Hi=max(vagues[i])-min(vagues[i+1])
        Ti=len(vagues[i]) * 0.5                # fréquence de 2 Hz
        liste.append([Hi, Ti])
    return liste
```