Correction CNC 2019 - MP

ESSADDOUKI Mostafa (essaddouki@gmail.com), WhatsApp (+212) 616 374 790

Exercice - Médian d'une liste de nombres

1. Ecrire la fonction ***grands(L,x)*** qui reçoit en paramètres une liste de nombre L, et un élément x de L. la fonction renvoie le nombre d'éléments de L qui sont supérieure strictement à x

```python
def grands(L, x):
    # initialiser le décompte (nb) avec 0
    nb = 0
    # Puis on compare x à chaque élément du tableau
    for i in range(len(L)):
        # Si un élément est supérieur à x, incrémentez nb par 1
        if L[i] > x:
            nb += 1
    return nb
```

2. Déterminer la complexité de la fonction $grands(L, x)$, et justifier votre réponse
la compléxité est $O(n)$, parce que nous bouclons sur tous les éléments, et à l'intérieur de la boucle, nous n'avons que des opérations élémentaires avec un coût $O(1)$
3. Ecrire la fonction ***petits(L,x)*** qui reçoit en paramètres une liste de nombre L, et un élément x de L. la fonction renvoie le nombre d'éléments de L qui sont inférieurs strictement à x

```python
def petits(L, x):
    # initialiser le décompte (nb) avec 0
    nb = 0
    # Puis on compare x à chaque élément du tableau
    for i in range(len(L)):
        # Si un élément est inférieur à x, incrémentez nb par 1
        if L[i] < x:
            nb += 1
    return nb
```

L est un une liste de taille n qui contient des nombres, et m un élément de L. L'élément m est un médian de L, si les deux conditions suivantes sont vérifiées :

— Le nombre d'éléments de L, qui sont supérieurs strictement à m, est inférieur ou égale à $n/2$

— Le nombre d'éléments de L, qui sont inférieurs strictement à m, est inférieur ou égale à $n/2$

4. Ecrire la fonction ***median(L)*** qui reçoit en paramètre une liste de nombres L non vide, et qui renvoie un élément médian de la liste L.

```python
def median(L):
    # Taille de la liste
    n = len(L)
    # Nous bouclons sur le tableau, et nous prenons chaque élément comme candidat pour la médiane
    for i in range(n):
        # L[i] est un candidat
        gd = grands(L, L[i])
        pt = petits(L, L[i])

        # Vérifiez si le candidat vérifie les propriétés
        if gd <= (n//2) and (pt <= n//2):
            # retourner la médiane
            return L[i]
```

5. Déterminer la complexité de la fonction ***median(L)***, et justifier votre réponse.
   $O(n^2)$, parce que nous bouclons sur tous les éléments, et à l'intérieur de la boucle, nous appelons les fonctions "grands" et "petits" et ces fonctions ont un coût O (n) chacune

Partie II :

II. 1- Calcul du déterminant d'une grille binaire carrée

Dans le but de calculer le déterminant d'une matrice carrée G qui représente une grille binaire carrée, on propose d'utiliser la méthode du ***pivot de Gauss***, dont le principe est le suivant :

— Créer une matrice C copie de la matrice G ;

— En utilisant la méthode du pivot de Gauss, transformer la matrice C en matrice triangulaire inférieure, ou bien en matrice triangulaire supérieure, en comptant le nombre d'échanges de lignes dans la matrice C. On pose k le nombre d'échanges de lignes dans la matrice C ;

— Calculer le déterminant D de la matrice triangulaire C ;

— Le déterminant de la matrice M est égal à : $D * (-1)^k$

1. a- Ecrire la fonction ***copie_matrice(G)***, qui reçoit en paramètre une matrice carrée G qui représente une grille binaire carrée, et qui renvoie une matrice C copie de la matrice G.

```python
def copie_matrice(G):
    # m et n sont respectivement le nombre de lignes et de colonnes de la matrice G
    m, n = len(G), len(G[0])
    # Créez une matrice C de même taille que G qui ne contient que des zéros
    C = zeros((m, n))
    # pour chaque ligne,
    for i in range(m):
        # pour chaque colonne
        for j in range(n):
            C[i, j] = G[i, j]

    # retourner la copie  C de G
    return C
```

1. b- Ecrire la fonction ***echange_lignes(C,i,j)***, qui reçoit en paramètres une matrice carrée C qui représente une grille binaire carrée. La fonction échange les lignes i et j dans la matrice C.

```python
def echange_lignes(C, i, j):
    # le nombre de colonnes
    n = len(C[i])

    # boucler sur le nombre de colonnes
    for k in range(n):
        # échanger l'élément dans la colonne k de la ligne i avec
        # l'élément dans la même colonne mais dans la ligne j
        C[i, k], C[j, k] = C[j, k], C[i, k]
```

1. c- Ecrire la fonction ***triangulaire(C)***, qui reçoit en paramètre une matrice carrée C qui représente une grille binaire carrée. En utilisant la méthode du Pivot de Gauss, la fonction transforme la matrice C en matrice triangulaire inférieure ou bien triangulaire supérieure, tout en comptant le nombre de fois qu'il y a eu échange de lignes dans la matrice C. La fonction doit retourner le nombre d'échanges de lignes.

```python
# Cette fonction cherche dans les éléments C(k+1,k), C(k+2,k) .. C(m,k)
# le premier pivot non nul et qui renvoie l'indice de la ligne correspondante
# Sinon renvoie -1 donc la matrice est non inversible
def choix_pivot(C, k):
    m = len(C)
    for i in range(k+1, m):
        if C[i, k] != 0:
            return i
    return -1


def triangulaire(C):
    # Nombre de lignes
    m = len(C)

    # Compter le nombre d'échange
    k = 0

    # boucler sur les lignes (chaque ligne contient un pivot)
    for i in range(m):
        # Si le pivot de la ligne i contient 0
        if C[i][i] == 0:
            # Chercher un pivot dans les lignes sous la ligne i
            pivot = choix_pivot(C, i)
            # Si la matrice est non inversible (aucun pivot trouvé)
            if pivot == -1:
                return (-1)

            # échanger la ligne i avec la ligne du pivot
            echange_lignes(C, i, pivot)

            # Incrémenter le nombre d'échange
            k += 1
        # pour chaque ligne j sous la ligne i, réaliser l'opération L : Ligne
        # L(j)=L(j)-L(i)*(L(j,k)/L(i,i))
        # pour que les lignes sous la ligne pivot (mêmes colonnes que le pivot) soient égales à 0
        for j in range(i+1, m):
            C[j] = C[j]-C[i]*(C[j, i]/C[i, i])

    # retourner le nombre d'échange
    return k
```

1. d- Ecrire la fonction ***deteminant(G)***, qui reçoit en paramètre une matrice G qui représente une grille binaire carrée. En utilisant la méthode du pivot de Gauss, la fonction renvoie la valeur du déterminant de G,

```python
def deteminant(G):
    # Créer une copie de G
    C = copie_matrice(G)

    # Rendre C triangulaire supérieure
    k = triangulaire(C)

    # Si la matrice est non inversible
    if k == -1:
        return None
    # Sinon

    # Le déterminant de la matrice triangulaire est le produit d'éléments diagonaux
    diag = 1
    for i in range(len(C)):
        diag *= C[i, i]

    # le déterminant de la matrice est le déterminant de la matrice triangulaire
    # multiplié par (-1) puissance (nombre d'échange de lignes)
    d = diag*(-1)**(k)
    return d
```

Tester la fonction

```python
G = array([[1, 1, 1, 1, 0, 0, 1, 1, 1, 1], [1, 1, 1, 0, 0, 1, 1, 0, 1, 1], [1, 1, 0, 0, 1, 1, 0, 0, 1, 1], [1, 1, 1, 0, 0, 0, 0, 1, 1, 1], [1, 0, 1, 1, 1, 0, 1, 1, 1, 1], [1, 0, 0, 1, 1, 1, 0, 1, 1, 0], [1, 1, 0, 1, 1, 0, 1, 0, 0, 0], [1, 0, 1, 0, 0, 1, 1, 0, 0, 1], [1, 0, 0, 0, 1, 1, 1, 0, 1, 1],
          [1, 1, 1, 0, 0, 1, 1, 1, 1, 1]], float)

print(deteminant(G))
 # résultat  = -4
```

II. 2- Représentation de la grille binaire par une liste de listes

La grille binaire de la figure 1 est représentée par la liste G suivante, composée de 10 listes, de taille 12 chacune :

```
G =[ [1,1,1,1,0,0,1,1,1,1,1,1] ,[1,1,1,0,0,1,1,0,1,1,1,1] ,
    [1,1,0,0,1,1,0,0,1,1,1,1] ,[1,1,1,0,0,0,0,1,1,1,1,1] ,
    [1,0,1,1,1,0,1,1,1,1,1,1] ,[1,0,0,1,1,1,0,1,1,0,0,0] ,
    [1,1,0,1,1,1,0,0,0,0,1,0] ,[1,0,1,0,0,1,1,0,0,1,1,1] ,
    [1,0,0,0,1,1,1,0,1,1,1,1] ,[1,1,1,0,0,1,1,1,1,1,1,1]
]
```

— **2-** à partir de la liste G, de l'exemple ci-dessus, donner le résultat de chacune des expressions suivantes : G[4][2] , len(G[3]) , G[5] , len(G)

G[4][2] = > 1
len(G[3]) => 12 : nombre de colonnes de la matrice
G[5] = > [1,0,0,1,1,1,0,1,1,0,0,0] : ligne 6 de la matrice
len(G) => 10 : nombre de lignes de la matrice

II. 3- Position valide d'une case

3- Ecrire la fonction ***valide(i,j,G)***, qui reçoit en paramètres deux entiers i et j. La fonction renvoie True, si i et j sont positifs, et s'ils représentent, respectivement la ligne et la colonne d'une case qui existe dans la grille binaire G, sinon, la fonction renvoie False.

```
def valide(t, G):
    # t est un tuple, le premier élément fait référence à
    # la ligne et le deuxième élément à la colonne
    # i est la ligne, j et la colonne
    i, j = t

    # Vérifiez si i est compris entre 0 et le nombre de lignes dans G,
    # et j est compris entre 0 et le nombre de colonnes dans G
    if (0 <= i < len(G)) and (0 <= j < len(G[0])):
        return True

    return False
```

II. 4- Couleur d'une case

4- Ecrire la fonction ***couleur(t,G)***, qui reçoit en paramètres un tuple ***t*** qui représente une case dans la grille binaire G. La fonction renvoie 1 si la couleur de la case t est blanche, sinon, elle renvoie 0.

```
def couleur(t, G):
    # t est un tuple, le premier élément fait référence à
    # la ligne et le deuxième élément à la colonne
    # i est la ligne, j et la colonne
    i, j = t

    # si le cas est valide, renvoyez la valeur stockée,
    if valide(t, G):
        return G[i][j]

    # renvoie -1 si ce n'est pas une case valide
    return -1
```

II. 5- Cases voisines

Dans une grille binaire, deux cases sont voisines, si elles ont la même couleur et si elles ont un seul côté en commun.

5- Ecrire la fonction ***list_voisines(t,G)***, qui reçoit en paramètres un tuple ***t*** représentant une case dans la grille binaire G. La fonction renvoie la liste des cases voisines à la case t, dans la grille G.

```python
def list_voisines(t, G):
    # t est un tuple, le premier élément fait référence à
    # la ligne et le deuxième élément à la colonne
    # i est la ligne, j et la colonne
    i, j = t

    # L contient la liste des cases voisines de t
    L = []

    # ajoutez la case ci-dessus s'elle est valide et a la même couleur que t
    if couleur((i+1, j), G) == couleur(t, G):
        L.append((i+1, j))

    # ajoutez la case ci-dessous s'elle est valide et a la même couleur que t
    if couleur((i-1, j), G) == couleur(t, G):
        L.append((i-1, j))

    # ajoutez la case à droite s'elle est valide et a la même couleur que t
    if couleur((i, j+1), G) == couleur(t, G):
        L.append((i, j+1))

    # ajoutez la case à gauche s'elle est valide et a la même couleur que t
    if couleur((i, j-1), G) == couleur(t, G):
        L.append((i, j-1))

    # retourner L
    return L
```

II. 6- Chemin dans une grille

On considère une liste L de tuples qui représentent des cases dans une grille binaire G. La liste L est un chemin dans la grille G, si la liste L satisfait les trois conditions suivantes :

- La liste L contient au moins deux cases de la grille G ;
- Toutes les cases de L sont différentes deux à deux, (pas de doublon dans L) ;
- Deux cases consécutives dans L sont voisines.

6- Ecrire la fonction ***chemin(L,G)***, qui reçoit en paramètres une liste **L** de tuples représentant des cases dans la grille binaire **G**. La fonction renvoie **True** si la liste L est un chemin dans G, sinon, la fonction renvoie **False**.

```python
def chemin(L, G):
    # nombre d'éléments dans L
    m = len(L)
    # nous parcourons le chemin à partir de l'index 1 (case 2)
    for i in range(1, m):
        # si la case actuelle (i) n'est pas voisine de la case précédente (i-1), retourner False
        if L[i] not in list_voisines(L[i-1], G):
            return False

    # si nous n'atteignons aucune case violant le chemin, retourner True
    return True
```

II. 7- Compression d'un chemin dans une grille binaire

La compression d'un chemin dans une grille binaire, consiste à remplacer, dans ce chemin, chaque suite d'au moins trois cases voisines qui se trouvent sur la même ligne ou bien sur la même colonne, par deux cases seulement : la première case et la dernière case.

7- Ecrire la fonction ***compresse_chemin(L),*** qui reçoit en paramètres une liste **L** qui contient un chemin dans une grille binaire. La fonction renvoie la liste qui contient le chemin compressé de L.

```python
def compression(L):
    comp = []
    comp.append(L[0])
    direction = -1
    for k in range(1, len(L)):
        # coordonnée de la case actuelle
        m, n = L[k]

        # coordonnée de la case précédente
        i, j = L[k-1]

        # d mémoriser le sens du mouvement à partir de la case précédente (k-1)
        # jusqu'à la case actuelle (k)
        d = -1

        # si les cases k et k-1 dans la même ligne
        # vérifier si le mouvement est à gauche ou à droite
        if m == i:
            # mouvement à droite
            if j < n:
                d = 1

            # mouvement à gauche
            else:
                d = 2

        # si les cases k et k-1 dans la même colonne
        # vérifier si le mouvement est en haut ou en bas
        if n == j:
            # mouvement en bas
            if i < m:
                d = 4
            # mouvement en haut
            else:
                d = 3
        # si c'est la première fois que nous calculons la direction
        # Donc, la direction est égale à la direction actuelle (d)
        if direction == -1:
            direction = d

        # Sinon
        else:
            # Vérifiez si la direction actuelle (d) n'est pas la même que la direction précédente (direction)
            # parce que la direction a changée, nous ajoutons la case précédente au chemin compressé
            # et mettre à jour la direction
            if d != direction:
                comp.append(L[k-1])
                direction = d
    # ajouter la dernière case du chemin dans le chemin compressé
    comp.append(L[-1])
    # retourner le chemin compressé
    return comp
```

II. 8- Appartenance d'une case à un chemin compressé

8- Ecrire la fonction ***appartient(t,L)***, qui reçoit en paramètres une liste L contenant un chemin compressé dans une grille binaire, et un tuple **t** représentant une case. La fonction renvoie True si le chemin compressé **L** passe par la case **t**, sinon, la fonction renvoie False.

```
def appartient(t, L):
    i, j = t

    for k in range(1, len(L)):
        # coordonnée de la case actuelle
        m, n = L[k]

        # coordonnée de la case précédente
        a, b = L[k-1]

        # si la case actuelle, la case précédente et t dans la même ligne
        if (m == a) and (a == i):
            # si la colonne de la case précédente est supérieure à la case courante
            if n < b:
                if n <= j <= b:
                    return True
            else:
                if b <= j <= n:
                    return True
        # Sinon case précedente (L[k-1]) et courante (L[k]) dans la même colonne
        else:
            # si la case t dans la même colonne que L[k] ou L[k-1]
            if j == b:
                if m < a:
                    if m <= i <= a:
                        return True
                else:
                    if a <= i <= m:
                        return True

    # si nous ne trouvons pas "t", renvoyer Faux
    return False
```

II. 9- Longueur d'un chemin compressé

Dans une grille binaire, la longueur d'un chemin compressé, est égale au nombre total des cases par les quelles passe ce chemin.

9- Ecrire la fonction, de **complexité linéaire**, ***longueur_chemin(L)***, qui reçoit en paramètres une liste L qui contient un chemin compressé. La fonction renvoie la longueur du chemin L.

```python
def longueur_chemin(L):
    if L == []:
        return 0

    taille = 0
    for k in range(1, len(L)):
        # coordonnée de la case actuelle
        m, n = L[k]

        # coordonnée de la case précédente
        a, b = L[k-1]

        # si la case actuelle, la case précédente et t dans la même ligne
        if (m == a):
            # ajouter la différence entre les colonnes à la taille
            taille += abs(n-b)

        # Sinon case précedente (L[k-1]) et courante (L[k]) dans la même colonne
        else:
            # ajouter la différence entre les lignes à la taille
            taille += abs(m-a)

    return taille+1
```

II. 10- Chemin entre deux cases

On suppose que a et b sont deux tuples qui représentent deux cases distinctes, de même couleur, dans une grille binaire G.
Un chemin entre les cases a et b, s'il existe, est un chemin compressé, tel que le tuple a est son premier élément, et le tuple b est son dernier élément.
NB : On peut trouver plusieurs chemins entre les cases a et b.

10. a- Ecrire la fonction ***chemins(G,a,b,chemin)***, reçoit en paramètres deux tuples a et b distincts qui représentent deux cases de même couleur, dans la grille binaire G. Le paramètre chemin est une liste initialisée par la liste vide. Cette fonction renvoie une nouvelle liste contenant tous les chemins entre la case a et la case b dans la grille G.

```python
def chemins(G, a, b, chemin):

    # si la taille du chemin est égale à 0,
    # ajoutez une sous-liste pour stocker le premier chemin
    if len(chemin) == 0:
        chemin.append(list())
    # si nous atteignons la destination (a == b)
    if a == b:
        # ajouter la destination au dernier chemin
        chemin[-1].append(a)
        # Créez une copie du dernier chemin et ajoutez-la au chemin
        chemin.append(list())
        for elm in chemin[-2]:
            chemin[-1].append(elm)

    else:
        # "a" est la case actuelle
        i, j = a
        # trouver des voisines de a
        v = list_voisines(a, G)
        # marquer a comme visité
        # 2 est une valeur aléatoire différente de 0 et 1
        G[i][j] = 2

        # ajouter la case courante au chemin
        chemin[-1].append(a)

        # vérifier récursivement s'il y a un chemin
        # partant de chaque voisine de la case "a"
        for i in range(len(v)):
            # obtenir la ligne et la colonne de la case v[i]
            k, m = v[i]
            # s'il n'est pas déjà visité
            if G[k][m] != 2:
                chemins(G, v[i], b, chemin)

    # s'il y a un chemin ou que nous atteignons une case qui ne mène pas à la destination,
    # supprimez cette case du chemin et marquez-la comme visitée (récupérez la couleur d'origine)
    i, j = a
    # marquer la case comme visitée
    G[i][j] = couleur(b, G)
    # supprimer cette case du chemin
    chemin[-1].pop()

    # si le dernier chemin est vide, supprimez-le
    if chemin[-1] == []:
        chemin.pop()
```

10. b- Ecrire la fonction ***list_chemins(G,a,b)***, reçoit en paramètres deux tuples a et b qui représentent deux cases quelconques dans la grille binaire G. Cette fonction renvoie la liste de tous les chemins compressés entre la case a et la case b, dans la grille G.

```python
def list_chemins(G, a, b):
    # chemins compressés
    comp = []

    # liste des chemins
    L = []
    # Stocker les chemins de a à b dans L
    chemins(G, a, b, L)

    # compresser chaque chemin
    for chemin in L:
        comp.append(compresse_chemin(chemin))

    return comp
```

II. 11- Tri d'une liste de chemins

11- Ecrire la fonction ***tri_chemins(R)***, qui reçoit en paramètres une liste R contenant tous les chemins compressés entre deux cases dans une grille binaire. La fonction trie les chemins compressés de R dans l'ordre croissant des longueurs des chemins. La longueur d'un chemin telle qu'elle est définie dans la question II. 9

```python
# algorithme de tri par selection
def tri_chemins(L):
    # Parcourez tous les éléments du tableau
    for i in range(len(L)):
        # Trouver l'élément minimum dans le tableau non trié restant
        min_idx = i
        for j in range(i+1, len(L)):
            if longueur_chemin(L[min_idx]) > longueur_chemin(L[j]):
                min_idx = j

        # Échanger l'élément minimum trouvé avec le premier élément
        L[i], L[min_idx] = L[min_idx], L[i]
```

II. 12- Plus courts chemins entre deux cases

a et b sont deux tuples qui représentent deux cases dans une grille binaire G. Le plus court chemin entre a et b est le chemin compressé entre a et b ayant la plus petite longueur.

12- Ecrire la fonction ***plus_court_chemins(G,a,b)***, qui renvoie la liste de tous les plus courts chemins compressés entre deux cases a et b.

```python
def plus_court_chemins(G, a, b):
    # liste des chemins les plus courts
    short = []

    # obtenir tous les chemins compressés entre a et b
    chemins = list_chemins(G, a, b)

    # s'il existe un chemin entre a et b
    if len(chemins) > 0:
        # trier les chemins et renvoyer le premier élément des chemins triés
        tri_chemins(chemins)

        # ajouter le premier élément
        short.append(chemins[0])

        # vérifier s'il y a un autre chemin avec la même longueur que le premier
        for i in range(1, len(chemins)):
            if longueur_chemin(chemins[i-1]) == longueur_chemin(chemins[i]):
                short.append(chemins[i])
            # Sinon quitter
            else:
                break

    return short
```

II. 13- Zone d'une case dans une grille binaire

On considère un tuple t qui représente une case dans une grille binaire G. La zone de la case t est l'ensemble qui contient la case t, et toutes les cases de la grille binaire G, qu'on peut joindre par un chemin à partir de la case t.

> **13-** Ecrire la fonction ***tri_chemins(R)***, qui reçoit en paramètres une liste R contenant tous les chemins compressés entre deux cases dans une grille binaire. La fonction trie les chemins compressés de R dans l'ordre croissant des longueurs des chemins. La longueur d'un chemin telle qu'elle est définie dans la question II. 9

pour découvrir une région d'une case, nous utilisons un algorithme de parcours en largeur et affectons à ces cases un nombre (pour les marquer comme visités), chaque fois que nous terminons l'exécution de l'algorithme, nous incrémentons le nombre de régions et recherchons une autre case non visitée et faisons de même

```python
def compte_zone(G):
    # On commence par 2 car la matrice contient déjà la valeur 0 et 1
    cpt = 2

    # Parcourir la matrice
    for i in range(len(G)):
        for j in range(len(G[0])):
            # si la case n'est pas visitée, exécutez l'algorithme à partir de cette case
            if (G[i][j] < 2):
                print((i, j))
                file = []
                file.append((i, j))
                while file:
                    m, n = file.pop(0)
                    v = list_voisines((m, n), G)
                    # marquer comme visité
                    G[m][n] = cpt
                    for elm in v:
                        ii, jj = elm
                        # si la boîte n'est pas visitée, ajoutez-la à la file d'attente
                        if G[ii][jj] < 2:
                            file.append(elm)
                # une fois le parcours terminé, incrémenter le nombre de régions
                cpt += 1

    # retourner (cpt-2) parce que nous avons commencé à 2
    return (cpt-2)
```