# Informatique Tronc Commun
# Révisions quotidiennes

Conseil : Vous pouvez cocher la case lorsque la question n'a pas été convenablement traitée et de prévoir de la refaire 3-4 jours plus tard.

## I Semaine 1

### I.1 Jeudi 2 avril

1–❑ Écrire une fonction `Moyenne(L)` qui renvoie la moyenne des éléments de la liste $L$ (liste non vide de nombres).

**Solution:**

```
def Moyenne(L):
    m=0
    for i in range(len(L)):
        m=m+L[i]
    return m/len(L)
def Moyenne(L):
    m=0
    for x in L
        m=m+x
    return m/len(L)
```

2–❑ Écrire une fonction `DepasseMoyenne(L)` qui renvoie la liste des éléments de $L$ strictement plus grand que la moyenne. Attention : la fonction `Moyenne(L)` devra être appelée une et une seule fois.

**Solution:**

```
def DepasseMoyenne(L):
    moy=Moyenne(L)
    dep=[] # liste des éléments > moyenne
    for i in range(len(L)):
       if L[i]>moy:
           dep.append(L[i])
    return dep
def DepasseMoyenne(L):
    moy=Moyenne(L)
    dep=[] # liste des éléments > moyenne
    for x in L:
       if x>moy:
           dep.append(x)
    return dep
```

3–❑ Quelle est la complexité de ces deux fonctions ?

**Solution:** Pour Moyenne, il y a une boucle for de $len(L)$ itérations, qui sont toutes à coût constant, cela donne une complexité linéaire en $O(len(L))$.
Pour DepasseMoyenne, l'appel à Moyenne est en $O(len(L))$ puis une boucle for de $len(L)$ itérations, qui sont toutes à coût constant.
Cela donne une complexité linéaire en $O(2 \times len(L)) = O(\times len(L))$.

## I.2 Vendredi 3 avril ♥

1–❑ Écrire une fonction `Maximum(L)` qui renvoie le maximum d'une liste non vide de nombres `L`.

**Solution:**

```
def Maximum(L):
    M=L[0] # maximum
    for i in range(1,len(L)):
        if L[i]>M: # nouveau maximum trouvé !
            M=L[i]
    return M
```

2–❑ Écrire une fonction `Indices(L)` qui renvoie la liste des indices des éléments égaux au maximum de `L`.
On n'appellera qu'une seule fois la fonction `Maximum`.

**Solution:**

```
def Indices(L):
    maxi=Maximum(L) # appel hors de la boucle for !
    li_maxi=[] # liste des maximums
    for i in range(len(L)):
        if L[i]==maxi:
            li_maxi.append(i)
    return li_maxi
```

## I.3 Samedi 4 avril ♥

1–❑ Écrire une fonction `Occurrences(chaine,x)` qui renvoie le nombre d'occurrences de `x` dans une chaine de caractères chaine.

**Solution:**

```
def Occurrences(chaine,x):
    compt=0
    for car in chaine:
        if car==x:
            compt+=1
    return compt
```

```
def Occurrences(chaine,x):
    compt=0
    for i in range(len(chaine)):
        if chaine[i]==x:
            compt+=1
    return compt
```

2–❑ Écrire la fonction `Minimum(d)` qui renvoie la clé du dictionnaire `d` dont la valeur associée est minimale.

**Solution:**

```
def Minimum(d):
    vmin=float('inf') # supérieur à toutes les valeurs de d
    for c in d:
        if d[c]<vmin: # valeur inférieure à vmin
            cmin=c # clé du minimum
            vmin=d[cmin]
    return cmin
```

## I.4 Dimanche 5 avril ♥

1–❑ Écrire une fonction `DictionnaireOccurrences(chaine)` qui prend comme paramètre une chaîne de caractères et renvoie un dictionnaire dont les clefs sont les caractères de la chaîne et la valeur associée est le nombre d'occurrences de ce caractère dans la chaine.

**Solution:**

```
def DictionnaireOccurrences(chaine):
    dict={car:0 for car in chaine} # dictionnaire des caractères
    for car in chaine:
        dict[car]+=1
    return dict
def DictionnaireOccurrences(chaine):
    dict={} # dictionnaire des caractères
    for car in chaine:
        if car in dict: # s'il a déjà été rencontré
            dict[car]+=1
        else:
            dict[car]=1 # on crée la clé
    return dict
```

2–❑ Soit texte une chaîne de caractères, écrire une fonction `EltMajoritaire(texte)` qui renvoie l'un des caractères qui apparait le plus souvent souvent dans cette `chaîne`.

**Solution:**

```
def EltMajoritaire(texte):
    d_occ=DictionnaireOccurrences(texte)
    maxi=0
    for c in d_occ: # parcours du dictionnaire des occurrences
        if d_occ[c]>maxi:
            c_max=c # clé d'occurrence maximale
            maxi=d_occ[c] # mise à jour de la valeur maximale
    return c_max
```

# II Semaine 2

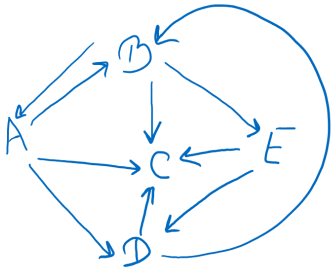
## II.1 Lundi 6 avril ♥

1–❑ Considérons le graphe définie le dictionnaire d'adjacence :

```
Dico={"A" :["B","C","D"], "B" :["A","C","E"], "C" :[], "D" :["B","C"], "E" :["C","D"]}
```

a) Faites un dessin du graphe.

**Solution:**




b) Donner la liste des points qui seront visités si on fait un parcours en largeur en partant de `A`.

**Solution:** Lors d'un parcours en largeur, on commence par visiter tous les sommets successeurs du sommet de départ, puis tous les successeurs (non encore visités) du sommet suivant... Soit : $A-B-C-D-E$

Informatiquement, on utilise une file (premier entré, premier sorti) :

| sommet | sommets en attente | liste des sommets visités |
|---|---|---|
| A | [B,C,D] | [A] |
| B | [C,D] | [A,B] |
| | [C,D,E] | [A,B] |
| C | [D,E] | [A,B,C ] |
| D | [E] | [A,B,C,D] |
| E | [] | [A,B,C,D,E] |

c) Donner la liste des points qui seront visités si on fait un parcours en profondeur partant de `B`.

**Solution:** Lors d'un parcours en profondeur, on commence par visiter un successeur du premier, puis un successeurs du deuxième sommet visité, ... puis on revient au premier et visiter le deuxième successeur, ... Soit : $B-E-D-C-A$

Informatiquement, on utilise une pile (premier entré, dernier sorti) :

| sommet | sommets en attente | liste des sommets visités |
|---|---|---|
| B | [A,C,E] | [B] |
| E | [A,C] | [B,E] |
| | [A,C,D] | [B,E] |
| D | [A,C] | [B,E,D] |
| C | [A] | [B,E,D,C] |
| A | [] | [B,E,D,C,A] |

2–❑ Écrire une fonction `sommets(G)` qui renvoie la liste des sommets du dictionnaire d'adjacence `G` d'un graphe.

**Solution:**

```
1 def sommets(G):
2     L=[c for c in G] # les clés sont les sommets du graphe
3     return L
```

## II.2 Mardi 7 avril ♥

1–❑ Écrire la fonction `degre_succ(G)` qui renvoie le dictionnaire dont les clés sont les sommets de `G` et les valeurs le nombre de successeurs du sommet.

**Solution:**

```
def degre_succ(G):
    return [c:len(G[c]) for c in G]
```

2–❑ Écrire la fonction `predecesseurs(G,x)` qui renvoie la liste des prédécesseurs de `x` dans `G`.

**Solution:**

```
def predecesseurs(G,x):
    pred=[]
    for c in G: # sommets de G
        for v in G[c]: # v successeur de c
            if v==x: # si x est successeur c
                pred.append(c) # c est prédécesseur de x
    return pred
```

3–❑ Écrire la fonction `transpose(G)` qui renvoie la liste des prédécesseurs des sommets de `G`.

**Solution:**

```
def transpose(G):
    trans_G={}
    for x in G:
         trans_G[x]=predecesseurs(G,x)
    return trans_G
```

## II.3 Mercredi 8 avril ♥

1–❑ Écrire une fonction `ImageBlanche(h,w)` qui définit une image blanche de taille $h \times w$ ($h$ lignes et $w$ colonnes), ne comportant que des 255.

**Solution:**

```
def ImageBlance(h,w):
    return [ [255 for i in range(w) ] for j in range(h) ]
```

2–❑ Écrire une fonction `MaxCoeffMatrice(M)` qui, à une matrice `M`, renvoie le plus grand coefficient de `M`, sa ligne et sa colonne.

**Solution:**

```
def MaxCoeffMatrice(M):
    m,imax,jmax=M[0][0],0,0 # valeur du max, ligne et colonne du max
    for i in range(len(M)):
        for j in range(len(M[0])):
            if M[i][j]>m:
                imax,jmax=i,j
                m=M[i][j]
    return m
```

3–❑ On considère la matrice `M` d'adjacence d'un graphe orienté et pondéré. Écrire une fonction `Transposee(M)` qui renvoie la transposée de la matrice `M` (c'est-à-dire la matrice du graphe dont on a inversé le sens des flèches).

**Solution:** Il faut calculer la transposée de la matrice

```
def Transposee(M):
    Mt = [ [0 for i in range(len(M[0])) ] for j in range(len(M)) ]
    for i in range(len(M)):
        for j in range(len(M[0])):
            Mt[j][i]=M[i][j]
    return Mt
```

## II.4 Jeudi 9 avril

Soit G un graphe non orienté et pondéré, dont les sommets sont numérotés de 0 à $n-1$, ($n$ étant donc l'ordre du graphe) codé par sa matrice d'adjacente notée `M`.

1–❑ ♥ À quoi correspond la valeur de `M[i][j]` dans le graphe ?

**Solution:** `M[i][j]` correspond à la pondération de l'arc entre le sommet $i$ et le sommet $j$.

2–❑ ♥ Quelle propriété a la matrice `M` du fait que le graphe soit non orienté ?

**Solution:** La matrice est symétrique.

3–❑ ♥ Quelle commande permet de récupérer l'ordre du graphe ?

**Solution:** L'ordre du graphe, c'est-à-dire le nombre de sommet s'obtient avec `len(M)`

4–❑ ♪ ♪ ♪ Un graphe est complet si tous les sommets sont reliés à l'ensemble des autres sommets.
Écrire une fonction `EstComplet(M)` qui teste si graphe `G` représenté par sa matrice d'adjacence `M` est complet. La fonction renvoie `True` s'il est complet, `False` sinon.

**Solution:** Il faut vérifier qu'à partir de n'importe quel sommet on atteint n'importe quel sommet. Dans la matrice, il faut donc qu'il n'y ait aucun 0 hormis sur la diagonale.

```
def EstComplet(M):
    for i in range(len(M)):
        for j in range(len(M[0])):
            if i!=j and M[i][j]==0:
                return False # i non relié à j
    return True # on n'est jamais sorti, donc graphe complet
```

## II.5 Vendredi 10 avril ♥

On considère le graphe `G={0: [1,2,3], 1 : [0,2,4], 2 : [], 3 : [1,2], 4 : [2,3]}`

1–❑ Donner la matrice d'adjacence `M` de `G`.

**Solution:** Le graphe est orienté, la matrice n'est pas symétrique.

$$M = \begin{pmatrix} 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \end{pmatrix}$$

2–❑ Écrire une fonction `degre_mat(M:list,s:int)->int` qui prend en argument un graphe représenté par sa matrice d'adjacence `M` et un sommet `s`, et renvoie le degré du sommet `s`.

**Solution:** Il faut compter le nombre d'arcs qui partent de s.

```
def degre_mat(M,s):
    deg=0
    for j in range(len(M[s])):
        deg=deg+M[s][j] # ajoute 1 s'il y a un arc, 0 sinon
    return deg
```

3–❑ Écrire une fonction `voisin_mat(M:list,s:int)->list` qui prend en entrée une matrice d'adjacence et un sommet `s`, et renvoie la liste des voisins du sommet `s`.

**Solution:**

```
def voisin_mat(M,s):
    voi=[] # liste des voisins
    for j in range(len(M[s])):
        if s!=j and M[s][j]!=0: # il y a un arc entre s et j
            voi.append(j) # j est voisin de s
    return voi
```

4–❑ Écrire une fonction `Isole(G)` qui permet de renvoyer la liste des sommets isolés du graphe `G`.

**Solution:** Un graphe est isolé s'il n'y a aucun arc qui n'en part, à part éventuellement bouclé sur lui-même

```
def Isole(G):
    seuls=[] # liste des sommets isolés
    for i in range(len(M)):
        if voisin_mat(M,i)!=[]: # aucun voisin à i
            seuls.append(i)
    return seuls
```

## II.6 Samedi 11 avril ♥

1–❑ Écrire la fonction `distance(X:list,Y:list)->float` qui renvoie la distance entre les deux listes `X` et `Y`.

**Solution:**

```
def distance(X,Y):
    d=0
    for i in range(len(X)):
        d = d + (X[i]-Y[i])**2
    return sqrt(d)
```

2–❑ Écrire la fonction `Distances(X:list,Z:list[list])->list[float]` qui renvoie la liste des distances entre `X` et chaque liste de la liste `Z` en utilisant la fonction précédente.

**Solution:** Avec la fonction précédente.

```
def Distances(X,Z):
    LD=[] # liste des distances
    for i in range(len(Z)):
        dist_i=distance(X,Z[i]) # distance entre la liste Z[i] et X
        LD.append(dist_i)
    return LD
```

3–❑ Évaluer la complexité de cette dernière fonction en fonction du nombre d'éléments de `Z` et de la longueur de `X`.

**Solution:** Il y a une boucle de `len(Z)` itérations. Au sein de chaque itération, on appelle la fonction `distance` qui calcule la distance entre `X` et `Z[i]`. Au sein de `distance` il y a une boucle `for` de `len(X)` itérations, chaque itération est de coût constant.
Ainsi, `Distances` est de complexité $\mathcal{O}(\texttt{en(Z)}\times\texttt{len(X)})$

## II.7 Dimanche 12 avril

Soit `L=[['ville',temperature],... ]` qui donne la température (un flottant) un jour donné dans la ville `'ville'`.

1–❑ ♥ Écrire la fonction `moyenne(L:list[list])` qui renvoie la moyenne des températures du jour.

**Solution:**

```
def moyenne(L):
    S = 0
    for i in range(len(L)):
        S = S + L[i][1]
    return S/len(L)
```

2–❑ Écrire la fonction `ecart_type(L:list[list])` qui renvoie l'écart-type des températures du jour. On veillera à obtenir une fonction de complexité linéaire.

**Solution:**

```
def ecart_type(L):
    M = moyenne(L)
    S = 0
    for i in range(len(L)):
        S = S + ( L[i][1] - M )**2
    return sqrt(S/len(L))
```

3–❑ Écrire la fonction `froid(L:list[list])->str` qui renvoie la ville où la température a été la plus froide.

**Solution:**

```
def froid(L):
    ville = L[0][0] # nom de la ville
    Tmin = L[0][1] # température minimale
    for i in range(1,len(L)):
        if L[i][1] < Tmin:
            Tmin = L[i][1]
            ville = L[i][0]
    return ville
```

# III Semaine 3

## III.1 Lundi 13 avril ♥

On considère une image `img` représentée par un tableau (=liste de listes) de $h$ lignes et de $w$ colonnes. Chaque élément du tableau est une liste de trois entiers $[r, g, b]$ (=pixel). Ces entiers sont codés sur 8 bits.

1–❑ Comment obtient-on le nombre de lignes et le nombre de colonnes de `img` ?

**Solution:** Nombre de lignes : `len(img)` ; Nombre de colonnes : `len(img[0])`

2–❑ Que contient `img[0][0][0]` ? `img[0][0][1]` ? `img[0][0]` ?

**Solution:** `img[0][0][0]` est le niveau de rouge du pixel ligne 0 et colonne 0.
`img[0][0][1]` est le niveau de vert du pixel ligne 0 et colonne 0.
`img[0][0]` est une liste de trois éléments : [niveau de rouge, niveau de vert, niveau de bleu]

3–❑ Écrire une fonction `inv(pixel)` qui renvoie le négatif d'un pixel (représenté par une liste de trois entiers $[r, g, b]$), c'est-à-dire les couleurs inversées (par exemple le niveau de rouge $r$ est remplacé par $255 - r$).

**Solution:**

```
def inv(pixel):
    pix_inv = [255-pixel[i] for i in range(3)]
```

4–❑ Écrire une fonction `negatif(img)` qui renvoie le négatif de l'image.

**Solution:**

```
def negatif(img):
    img_neg=[[0 for j in range(len(img[0]))] for i in range(len(img))]
    for i in range(len(img)):
        for j in range(len(img[0])):
            img_neg[i][j] = inv(img[i][j])
    return img
```

## III.2 Mardi 14 avril

On considère une image `img` en niveau de gris : c'est un tableau de taille $h \times w$ dont chaque élément est un entier codé sur 8 bits (le pixel est d'autant plus sombre que cet entier est petit)..

1–❑ ♥ Écrire la fonction `nb(img,seuil)` qui convertit l'image en noir et blanc selon le critère de seuil : le pixel est blanc (255) au-delà du seuil, il est noir (0) sinon.

**Solution:**

```
def nb(img,seuil):
    h,w=len(img),len(img[0]) # nombre de lignes et de colonnes
    img_nb=[[0 for j in range(w)] for i in range(h) ] # image noire
    for i in range(h):
        for j in range(w):
            if img[i][j]>seuil:
                img_nb[i][j]=255 # blanc codé avec 255
    return img_nb
```

2–❑ Écrire la fonction `sombre(img,seuil)` qui renvoie la liste des coordonnées (numéro de ligne, numéro de colonne) des pixels noirs.

**Solution:**

```
def sombre(img,seuil):
    h,w=len(img),len(img[0]) # nombre de lignes et de colonnes
    noir_blanc=nb(img,seuil) # conversion de img en noir et blanc
    L_noir=[]  # liste des pixels noirs
    for i in range(h):
        for j in range(w):
            if noir_blanc[i][j]==0:# pixel noir
                L_noir.append([i,j]) # liste des coordonnées du pixel noir
    return L_noir
```

## III.3 Mercredi 15 avril

1–❑ ♥ Soit un `texte` une chaîne de caractères, écrire une fonction `Presence(lettre,texte)` qui vérifie si `texte` contient le caractère `lettre`.

**Solution:**

```
def Presence(lettre,texte):
    for car in texte:
        if car==lettre: # car vaut lettre
            return True
    return False # on est arrivé au bout de texte, lettre non trouvée
```

2–❑ Écrire une fonction `RienAVoir(L1,L2)` qui renvoie `True` si les deux listes `L1` et `L2` n'ont aucun élément en commun et `False` sinon. La complexité est attendue en $\mathcal{O}\big(\max(\text{len}(L_1),\text{len}(L_2))\big)$.

**Solution:** Une idée possible :

```
def RienAVoir(L1,L2):
    for x1 in L1:
        if Presence(x1,L2): # x1 est dans la liste L2
            return False
    return True
```

ne respecte pas la complexité demandée, car `Presence(x1,L2)` est de complexité $\mathcal{O}\big(\text{len}(L_2)\big)$. Appelée dans la boucle for sur `L1` ça donne une complexité quadratique en $\mathcal{O}\big(\text{len}(L_1)\times\text{len}(L_2)\big)$.
Pour diminuer la complexité, utilisons un dictionnaire pour stocker les caractères d'une des deux listes, puis tester s'ils sont ou non dans la liste $L_2$.

```
def RienAVoir(L1,L2):
    dict1={} # dictionnaire des éléments de L1
    for x1 in L1: # complexité O(len(L1))
        dict1[x1]=True
    for x2 in L2: # len(L2) itérations
        if x2 in dict1: # si x2 est présent dans dict1, il est dans L1 : recherche en cout constant car dans un dictionnaire
            return False
    return True # on a parcouru tout L2 sans trouver d'éléments aussi présents dans L1
```

3–❑ ♥ Soit `D` un dictionnaire dont les clefs et les valeurs sont toutes des chaînes de caractères. Écrire une fonction `PlusLong(D)` qui renvoie la clef dont la valeur associée est de longueur maximale.

**Solution:**

```
def PlusLong(D):
    lg_max=0
    for c in D: # clé de D
        if len(D[c])>lg_max: # nouvelle longueur max
            lg_max=len(D[c])
            cmax=c # clé dont la valeur est de lg max
    return cmax
```

## III.4 Jeudi 16 avril ♥

1–❑ Écrire la fonction `Distances(X:list,Z:list[list])->list[float]` qui renvoie la liste des distances entre `X` et chaque liste de la liste `Z`. On n'utilisera pas de fonctions auxiliaires.

**Solution:** sans la fonction précédente

```
def Distances(X,Z):
    LD=[] # liste des distances
    for i in range(len(Z)): # parcours des listes de Z
        # calcul de la distance entre Z[i] et X
        dist_i=0 # initialisation de la distance entre Z[i] et X
        for j in range(len(X)): # ou len(Z[i])
            dist_i=dist_i+(X[j]-Z[i][z])**2 # somme à calculer
        LD.append(sqrt(dist_i))
    return LD
```

2–❑ Écrire une fonction `EstTriee(L)` qui renvoie `True` si la liste de nombre `L` est triée par ordre croissant et `False` sinon.

**Solution:**

```
def EstTriee(L):
    for i in range(len(L)-1):
        if L[i+1]<L[i]: # suivant plus petit que le précédent
            return False
    return True # on est arrivé au bout de la liste avec toujours L[i+1]>=L[i] :
   elle est triée !
```

3–❑ Quelle est la complexité de cette fonction ?

**Solution:** On parcourt la liste une fois, le test est en coût constant $O(1)$. La complexité est linéaire en la longueur de la liste.

## III.5 Vendredi 17 avril ♪ ♪ ♪

On considère une liste `L` de flottants.

1–❑ Écrire une fonction `PlusProches(L)` qui prend en entrée une liste de flottants et renvoie un couple (tuple) d'indices distincts de valeurs les plus proches dans cette liste.

**Solution:**

```
def PlusProches(L):
    i1,i2=0,0 # couple d'indices des 2 éléments les plus proches
    d=float('inf') # écart minimal entre les deux éléments
    for j in range(len(L)):
        for k in range(len(L)):
            if j!=k:
                if abs(L[j]-L[k])<d: # ils sont plus proches
                    i1,i2=j,k
                    d=abs(L[j]-L[k]) # mise à jour du plus petit écart
    return (i1,i2)
```

2–❑ Écrire une fonction `SecondMax(L)` qui prend en entrée une liste de flottants et renvoie la valeur du second maximum de cette liste (la seconde valeur maximale après le maximum).

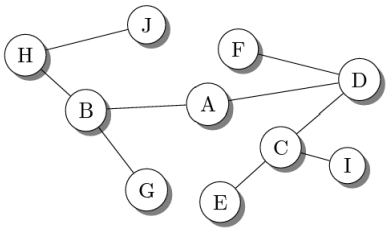
**Solution:**

```
def SecondMax(L):
    # recherche du 1er max
    max1=L[0] # 1er max
    for i in range(len(L)):
        if L[i]>max1:
            max1=L[i]
    print(max1)
    # recherche du 2è max
    max2=L[0]
    for i in range(len(L)):
        if L[i]!=max1:
            if L[i]>max2:
                max2=L[i]
    return max1,max2
```

## III.6 Samedi 18 avril ♪ ♪ ♪

1–❑ Qu'est-ce qu'une pile ? qu'est-ce qu'une file ? S'aider d'une représentation de la vie quotidienne !

**Solution:** Une pile est une structure informatique basée sur le principe « dernier arrivé, premier sorti ». Une liste peut être vue comme une pile : `L.append(x)` ajoute `x` en fin de liste, et `L.pop()` enlève le dernier élément de `L`.
Une file est une structure informatique basée sur le principe « premier arrivé, premier sorti. » `L.append(x)` ajoute `x` en fin de file, et `L.popleft()` enlève le premier élément de `L`.




2–❑ Effectuer le parcours en largeur et en profondeur du graphe suivant à partir de A :

**Solution:**
- Parcours en largeur : A - B - D - C - H - F - C - J - E - I
- Parcours en profondeur : A - B - H - J - D - C - E - G - F - E - I

3–❑ Compléter la fonction suivante qui effectue de façon itérative le parcours en largeur d'un graphe représenté par un dictionnaire l'adjacence.

**Solution:**
```
def parcours_largeur(G,s0):
    parcours=[] # liste décrivant le parcours
    vu={} # dictionnaire  des sommets visités
    file=deque([s0]) # file qui garde les sommets en attente
    while len(file)>0 : # tant que la file n'est pas vide
        s=file.popleft() # on enlève le sommet au début de la file (file.popleft())
        if s not in vu : # si le sommet s n'a pas déjà été visité
            parcours.append(s) # on l'ajoute à la liste du parcours
            vu[s]=True # on l'ajoute au dictionnaire vu
            for s in G[s] : # il faut visiter les voisins v du sommet s
                if s not in vu : # le voisin v de s n'a pas déjà été visité
                    file.append(s) # on le place dans la file des sommets en
   attente d'être visité (en fin de file)
    return visite
```

4–❑ Compléter la fonction suivante qui effectue de façon itérative le parcours en profondeur d'un graphe représenté par un dictionnaire l'adjacence.

**Solution:**
```
def parcours_profondeur(G,s0):
    parcours=[] # liste décrivant le parcours
    vu={} # dictionnaire  des sommets visités
    pile=[] # pile qui garde les sommets en attente
    pile.append(s0) # on ajoute au sommet de la pile le sommet de départ
    while len(pile)>0 : # tant que la pile n'est pas vide
        s=pile.pop() # on enlève et enlève le sommet au sommet de la pile (pile.pop
   ())
        if s not in vu : # si le sommet s  n'a pas déjà été visité
            parcours.append(s) # on l'ajoute à la liste des sommets visités
            vu[s]=True # on l'ajoute au dictionnaire des sommets vus
            for v in G[s] : # il faut visiter les voisins v du sommet s
                if v not in vu : # si le voisin v de s n'a pas déjà été visité
                    pile.append(v) # on le place dans la pile des sommets en
   attente d'être visité
    return parcours
```