Informatique Tronc Commun
Les indispensables

Ce polycopié reprend les algorithmes fondamentaux de 1re année à maitriser. Les graphes feront l'objet d'un autre polycopié. Toutes les fonctions ici doivent pouvoir écrites sans hésitation et sans erreur.

I Représentation des nombres

R1. Quels entiers peut-on coder sur 1 octet ?

Solution: 1 octet = 8 bits et 1 bit permet de coder 2 entiers (0 et 1).
Ainsi, 1 octet permet de coder $2^8 = 256$ entiers.

R2. Un pixel est codé sur trois niveaux de couleur (RGB), chacun représenté par un entier compris entre 0 et 255. Quelle est la mémoire nécessaire pour stocker une image de 1024 pixels par 768 ?

Solution: Chaque niveau de couleur nécessite 1 octet pour le coder. Ainsi, il faut $3 \times 1024 \times 768 = 2\ 359\ 296$ octets pour les stocker l'image.

II Listes (++++)

On considère une liste `L` de flottants.

R3. Écrire une fonction qui prend en entrée la liste `L` et qui renvoie la somme des éléments de `L`.

Solution:

```
def somme(L):
    """  Renvoie la somme des éléments de la liste L """
    S=0 #initialisation de la somme
    for i in range(len(L)):
        S=S+L[i]
    return S
```

R4. Écrire une fonction qui prend en entrée la liste `L` et qui renvoie la moyenne des éléments de `L`.

Solution:

```
def moyenne(L):
    """ Renvoie la moyenne des éléments de la liste L """
    S=0 #initialisation de la somme
    for i in range(len(L)): #calcul de la somme des éléments de L
        S=S+L[i]
    return S/len(L) #calcul de la moyenne
def moyenne(L):
    """ Renvoie la moyenne des éléments de la liste L """
    return somme(L)/len(L)
```

R5. Écrire une fonction qui prend en entrée la liste `L` et qui renvoie la variance des éléments de `L`.

Solution:

```
def variance(L):
    """ Renvoie la variance des éléments de la liste L """
    moy=moyenne(L)
    S=0 #initialisation de la somme
    for i in range(len(L)): #calcul de la somme des carrés des écarts à
     la moyenne
        S=S+(L[i]-moy)**2
    return S/len(L) #calcul de la variance
```

R6. Écrire une fonction qui prend en entrée la liste `L` et qui renvoie le maximum des éléments de `L`.

Solution:

```
def max(L):
    """ Renvoie le maximum de la liste L  """
    M=L[0] #initialisation du minmum
    for i in range(len(L)):
        if L[i]>M: #teste si L[i] est sup à M
            M=L[i]
    return M
```

R7. Écrire une fonction qui prend en entrée la liste `L` et qui renvoie le rang du minimum des éléments de `L`.

Solution:

```
def rang_min(L):
    """ Renvoie le rang du minimum de la liste L  """
    m=L[0] #initialisation du minmum
    rg=0 #initialisation du rang du minimum
    for i in range(len(L)):
        if L[i]<m: #teste si L[i] est inf à m
            m=L[i]
            rg=i #i est le nouveau rang du min
    return rg
```

R8. La liste de listes de deux éléments, `L`, est du type `[[ch1,a1],[ch2,a2],...]`, où les premiers éléments de chaque liste `ch1`, `ch2`, sont des chaines de caractères et les deuxièmes éléments `a1`, `a2`, de chaque sous liste est un flottant.

(a) Écrire une fonction qui prend en entrée la liste `L` et qui renvoie la somme des valeurs `ai`.

Solution:

```
def somme2(L):
    S=0 #initialisation de la somme
    for i in range(len(L)):
        S=S+L[i][1] #on somme les 2è éléments de chaque sous-liste
    return S
```

(b) Écrire une fonction qui prend en entrée la liste `L` et qui renvoie la moyenne des valeurs `ai`.

Solution:

```
def moyenne2(L):
    return somme2(L)/len(L)
```

(c) Écrire une fonction qui prend en entrée la liste `L` et qui renvoie la valeur maximale des `ai`.

Solution:

```
def max2(L):
    M=L[0][1] # initialisation du maximum (2è élément de L[0])
    for i in range(len(L)):
        if L[i][1]>M:
            M=L[i][1] #on a trouvé un nouveau max
    return M
```

(d) Écrire une fonction qui prend en entrée la liste `L` et qui renvoie la chaine `chi` de valeur `ai` minimale.

Solution:

```
def min3(L):
    ch_m=L[0][0] # chaine ch de valeur ai min
    m=L[0][1] # initialisation du minimum (2è élément de L[0])
    for i in range(len(L)):
        if L[i][1]<m:
            m=L[i][1] #on a trouvé un nouveau min
            ch_m=L[i][0]
    return ch_m
```

R9. Écrire une fonction booléenne qui prend en entrée une liste `L` d'entiers et un entier `e` si l'élément et qui renvoie si un élément ou non dans la liste. Quelle est la complexité de cette fonction ?

Solution:

```
def recherche_elt(L,e):
    """ Renvoie True si e est dans la liste L, False sinon """
    for i in range(len(L)):
        if L[i]==e:
            return True
    return False
```

Il y a un parcours complet de la liste, donc la complexité est linéaire en la longueur de la liste.

III Les dictionnaires

On considère un dictionnaire `D`, les clés sont des chaines de caractères, et les valeurs des entiers.

R10. Écrire une fonction qui prend en entrée le dictionnaire `D` et qui renvoie la valeur maximale.

Solution:

```
def max_dico(D):
    M=-float('inf')
```

```
    for c in D:
        if D[c]>M:
            M=D[c]
    return M
```

R11. Écrire une fonction qui prend en entrée le dictionnaire `D` et qui renvoie la clé de valeur minimale.

Solution:

```
def cle_min_dico(D):
    m=float('inf')
    for c in D:
        if D[c]<m:
            m=D[c]
            cmin=c
    return cmin
```

R12. Écrire une fonction qui prend en entrée le dictionnaire `D` et qui renvoie la moyenne des valeurs.

Solution:

```
def moy_dico(D):
    moy=0
    for c in D:
        moy=moy+D[c]
    return moy/len(D)
```

R13. Écrire une fonction qui prend en entrée le dictionnaire `D` et un entier `e` et qui renvoie la liste des clés de valeurs `e`.

Solution: (HS !) Fonction qui compte le nombre de fois où e apparait :

```
def valeur(D,e):
    compt=0
    for c in D:
        if D[c]==e:
            compt+=1
    return compt
```

La fonction réellement demandée :

```
def valeur(D,e):
    L=[]
    for c in D:
        if D[c]==e:
            L.append(c)
    return L
```

R14. Écrire une fonction `occurrences(texte:str)->dict` qui prend en argument une chaîne de caractère `texte` et renvoie un dictionnaire dont les clés sont les lettres qui apparaissent dans le texte et les valeurs le nombre d'occurrences de ces lettres.

Par exemple : `occurrences('ACCTAGCCCTA')` renverra `{'A':3,'C':5,'T':2,'G':1}`.

Solution:

```
def occurrences(texte):
    occ={}
    for x in texte:
        if x not in occ: # le caractère n'a pas été rencontré
            occ[x]=1 # on crée la clé
        else:
            occ[x]=occ[x]+1
    return occ
# ou
def occurrences(texte):
    occ={}
    for i in range(len(texte)):
        if texte[i] not in occ: # le caractère n'a pas été rencontré
            occ[texte[i]]=1 # on crée la clé
        else:
            occ[texte[i]]=occ[texte[i]]+1
    return occ
```

IV Tris

On souhaite trier une liste, par ordre croissant.

À adapter pour trier une liste par ordre décroissant, ou une liste de liste selon l'un des éléments des sous listes...

On donne ici quelques exemples de tris classiques. Aucun n'est rigoureusement au programme, tout en l'étant tous... Les algorithmes ne sont pas à connaître mais il faudrait pouvoir les écrire une fois l'algorithme rappelé.

IV.1 Tri fusion

Le tri fusion est un **tri récursif** basé sur le principe de « diviser pour régner » : un problème initial sur n données est divisé en deux sous-problèmes portant sur $\frac{n}{2}$ données si possible.

Ici, il s'agit de passer du tri d'une liste de n éléments aux tris de deux listes contenant moitié moins d'éléments. On trie alors la première moitié de la liste, la seconde moitié de la liste, et on fusionne ces deux listes triées en une seule liste triée en utilisant une fonction annexe.

R15. Écrire une fonction `fusion(L1,L2)` qui prend en entrée deux listes triées dans le même ordre et renvoie une liste triée résultant de la fusion des deux listes.

Solution:

```
def fusion(L1,L2):
    """Fusionne deux listes triées en une seule liste triée"""
    n1,n2=len(L1),len(L2) #longueur des deux listes
    L=[] #liste vide qui contiendra les deux listes fusionnées
    i1,i2=0,0 #indices pour parcourir les listes L1, L2
    while i1<n1 and i2<n2 : #on reste dans les deux listes
        if L1[i1]<L2[i2]:
            L.append(L1[i1])
            i1=i1+1 #on passe à l'élément suivant de L1
        else:
            L.append(L2[i2])
            i2=i2+1 #on passe à l'élément suivant de L2
    if i1==n1:
```

```
    #première liste finie, il reste à placer tous les éléments de L2 à la fin de L
        while i2<n2:
            L.append(L2[i2])
            i2=i2+1 #on passe à l'élément suivant de t2
    else:
    #deuxième liste, il reste à placer tous les éléments de L1 à la fin de L
        while i1<n1:
            L.append(L1[i1])
            i1=i1+1 #on passe à l'élément suivant de t1
    return L
```

R16. Écrire la fonction récursive `tri_fusion(L)` qui renvoie la liste triée en utilisant l'algorithme du tri fusion.

Solution:

```
def tri_fusion(L):
    """tri fusion d'un tableau en récursif"""
    if len(L)<=1: #condition d'arrêt
        return L
    #fusion des deux demi-listes triées
    return fusion(tri_fusion(L[:len(L)//2]),tri_fusion(L[len(L)//2:]))
```

IV.2 Tri rapide

Le tri rapide est un tri récursif dans lequel on divise un problème initial en deux sous-problèmes.

On choisit un élément, que l'on notera `p`, appelé **pivot** (qui peut être, le premier élément de la liste, le dernier, ...), de l'enlever de la liste et de partitionner la liste en deux sous-tableaux : une liste contenant les éléments strictement inférieurs à `p` et une liste contenant les éléments strictement supérieurs à `p`. On trie récursivement chacune des deux listes et on rassemble tout.

R17. Écrire la fonction `tri_rapide` qui renvoie la liste `L` triée.

Solution:

```
def tri_rapide(L):
    """Tri rapide d'une liste en récursif"""
    if len(L)<=1 : # si la liste contient moins de 2 éléments, elle est déjà triée.
        return L
    p=L[0] #choix du pivot, le premier élément de L
    L1,L2,L3=[],[],[]#création des trois listes vides L1, L2, L3, qui contiendront respectivement les éléments strictement inférieurs au pivot, égaux au pivot, strictement supérieurs
    for x in L : #parcours des éléments de la liste
        if x<p:
            L1.append(x)
        elif x==p:
            L2.append(x)
        else:
            L3.append(x)
```

```
    return tri_rapide(L1)+L2+tri_rapide(L3)

def tri_rapide(L):
    """Tri rapide d'une liste en récursif"""
    if len(L)<=1 : # si la liste contient moins de 2 éléments, elle est déjà triée.
        return L
    p=L[0] #choix du pivot, le premier élément de L
    L1,L3=[],[]#création des deux listes vides L1 et L3 qui contiendront respectivement les éléments strictement inférieurs au pivot, strictement supérieurs
    for x in L : #parcours des éléments de la liste
        if x<p:
            L1.append(x)
        else:
            L3.append(x)
    return tri_rapide(L1)+[p]+tri_rapide(L3)
```

IV.3 Tri bulles

On considère une liste `L` à n éléments. Le tri à bulles consiste à faire remonter les éléments les plus grands en permutant successivement les éléments du tableau : on parcourt le tableau, et à chaque fois que l'élément de gauche est strictement supérieur à l'élément de droite, on les permute. À la fin de ce parcours, le plus grand élément du tableau est en dernière position. On recommence le parcours du tableau pour trier les $n-1$ éléments, puis les $n-2$ éléments, etc. Le nom « tri à bulles » vient du fait que les éléments les plus grands remontent plus vite, comme les bulles dans l'eau.

R18. Écrire une fonction itérative qui trie une liste selon l'algorithme du tri à bulles.

Solution:

```
def tri_bulles(L):
    for i in range(len(L)):#n=len(L) étapes
        for j in range(len(L)-i-1): #on parcourt L du rang 0 au rang n-i-2 : on trie L[:n-i-1], les i+1 derniers éléments sont déjà triés
            if L[j+1]<L[j]:
                L[j],L[j+1]=L[j+1],L[j]
    return L
```

R19. Écrire une fonction récursive qui trie une liste selon l'algorithme du tri à bulles.

Solution:

```
def tri_bulles_rec(L):
    if len(L)<=1: #condition d'arrêt
        return L
    else:
        for j in range(len(L)-1):
            if L[j+1]<L[j]: #on fait remonter l'élément le plus grand
                L[j],L[j+1]=L[j+1],L[j]
        return tri_bulles_rec(L[:len(L)-1])+[L[len(L)-1]] #appel récursif, on applique la fonction sur la liste L sans le dernier élément, et on concatène avec le dernier élément
```

V Dichotomie

R20. Comment doit-être la liste pour utiliser l'algorithme de dichotomie pour déterminer si un entier est présent dans une liste d'entier ?

Solution:

R21. Rappeler le principe de la dichotomie pour déterminer si un entier est présent dans une liste d'entier.

Solution:

R22. Écrire une fonction booléenne qui prend en entrée une liste `L` d'entiers et un entier `e` si l'élément et qui renvoie si un élément ou non dans la liste par dichotomie.

Solution: La recherche dichotomique ne peut être faite que sur une liste TRIÉE!!

```
def dichotomie(L,e):
    """ Recherche dichotomique de e dans la liste L TRIEE ! """
    g=0
    d=len(L)-1
    while g<=d:
        m=(g+d)//2 #rang du milieu de la liste
        if L[m]==e:
            return m #on a trouvé e  !
        elif L[m]<e :
        #si e est + gd que l'élt du milieu, on va le chercher sur [m+1,
   d]
            g=m+1
        else:
        #si e est + petit que l'élt du milieu, on va le chercher sur [g
   ,m-1]
            d=m-1
    return False #on  n'a pas trouvé e !
```

La recherche dichotomique, sur une liste triée, est en $O(\log(n))$, soit plus efficace que la recherche « naïve » (mais qui nécessite de trier la liste avant).

R23. Quelle est la complexité de cette fonction ?

Solution: La recherche dichotomique, sur une liste triée, est en $O(\log(n))$, soit plus efficace que la recherche « naïve » (mais qui nécessite de trier la liste avant).

VI Algorithme glouton

Les **algorithmes gloutons** suivent une stratégie simple : **lorsqu'à chaque étape, un choix doit être fait, c'est le choix optimal à ce moment qui est fait.**

Un problème d'optimisation a deux caractéristiques : une fonction que l'on doit maximiser ou minimiser et une série de contraintes auxquelles il faut satisfaire. On peut essayer de résoudre ce type de problème en écrivant un algorithme qui énumère toutes les possibilités afin de trouver la meilleure. C'est un algorithme très simple mais souvent inutilisable à cause du coût. L'objectif d'un algorithme glouton est d'obtenir une solution rapidement, mais qui ne sera pas toujours la solution optimale.

À chaque étape exécutée par un algorithme, se présente un ensemble de choix et un algorithme glouton fait le meilleur choix parmi les propositions. **Un choix glouton est donc un choix localement optimal. La question est de savoir si en faisant une série de choix localement optimaux, on finit par aboutir à une solution optimale. C'est parfois le cas mais pas toujours.**

Prenons l'exemple classique du rendu de monnaie. On considère le système de monnaie de la zone euro, et nous avons les pièces suivantes (en centimes) : $S = (1, 2, 5, 10, 20, 50, 100, 200)$.

Le problème du rendu de monnaie consiste à trouver le nombre de pièces de chaque type à rendre pour arriver à la somme à rendre, en minimisant le nombre de pièces utilisées.

La contrainte est le fait que le montant total des pièces rendues est égal à la somme à rendre.

On cherche, par valeur décroissante en partant de la pièce qui a la plus forte valeur, la première pièce qui a une valeur inférieure ou égale à la somme à rendre r. On prend cette pièce, on retranche sa valeur v à r. On recommence en partant de la pièce prise en cherchant cella qui a une valeur inférieure ou égale à la nouvelle somme à rendre $r - v$. On prend cette pièce, et ainsi de suite jusqu'à arriver à une somme à rendre nulle.

R24. Mettre en œuvre l'algorithme à la main pour rendre 8 centimes, 7 € ,12 €.

Solution:

R25. Compléter la fonction `monnaie(p,r)` en Python.

```
def monnaie(p,r):
    """
    Arguments :
    p : liste des pièces du système de monnaie
    r : somme à rendre
    Retour :
    sol : liste du nombre de pièces à rendre pour chaque pièce du système
    de monnaie
    """
    assert type(p)==list and type(r)==int
    assert r>=0
    n=len(p) # nbre de pièces
    i=........ # on part de la pièce la plus grosse
    sol=........ # initialisation du nbre de pièces dans chaque type
    while ........ : # tant que la somme à rendre n'est pas nulle
        while ........ : # tant que la pièce testée est plus grande que
  le montant à rendre, on passe à la suivante (plus petite)
            i=........
        sol[i] = ................ # on ajoute une pièce i à rendre
        r=................ # on enlève le montant de la pièce ajoutée à
  la somme à rendre
    return sol
```

On considère les valeurs des pièces et billets en euros `p=[1,2,5,10,20,50,100,200,500]`.

On représente par une liste `C` le nombre de chaque pièces/billets qu'un commerçant dispose dans sa caisse (par exemple : `[32,37,17,12,13,5,1,0,0]` signifie qu'il a 32 pièces de 1 €, 37 pièces de 2 €, etc).

R26. Écrire une fonction `renducaisse(p,C,r)` qui détermine la façon de rendre la somme `r`, avec ce qu'il y a de disponible dans la caisse. On renverra une liste avec la quantité de chaque pièce/billet qu'il devra rendre.

Solution:

```
def renducaisse(p,C,r):
    """
    Arguments:
    p : liste des valeurs des pièces et billets en euro
    C : liste des nombres de chaque pièce/billet disponible
    r : somme à rendre
    Retour :
    sol : liste de la quantité de chaque pièce/billet à rendre
    """
    assert type(p)==list and type(r)==int and type(C)==list
    assert r>=0
    n=len(p) # nbre de type de pièces/billets
    i=n-1 # on part de la pièce la plus grosse
    sol=n*[0] # initialisation du nbre de pièces/billets dans chaque type
    while r>0: # tant que la somme à rendre n'est pas nulle
        while p[i]>r:
            i=i-1 # tant que la pièce testée est plus grande que le montant à rendre, on passe à la suivante (plus petite)
        if C[i]>0: # s'il reste des pièces de ce type
            sol[i]=sol[i]+1 # on en rend une
            r=r-p[i] # on soustrait au montant qu'il reste à rendre
            C[i]=C[i]-1 # on enlève une pièce de la caisse
        else: # s'il ne reste pas de pièce de ce type
            i=i-1 #
    return sol
```

VII Algorithmes fondamentaux indispensables en 2e année

VII.1 Programmation dynamique

R27. Écrire une fonction `mat_nulle(n:int,m:int)->list[list]` qui crée un tableau de `n` lignes et de `m` colonnes remplies de 0.

Solution:

```
def mat_nulle(n,m):
    return [ [ 0 for j in range(m) ] for i in range(n) ]
```

Exemple simple : calcul du coefficient binomial avec $\binom{n}{k} = \begin{cases} 1 & \text{si } k = n \text{ ou } k = 0 \\ n & \text{si } k = 1 \\ \binom{n-1}{k-1} + \binom{n-1}{k} & \text{sinon} \end{cases}$

R28. En utilisant la mémoïsation, écrire la fonction `binome_mem1(k,n,d={})`.

Solution:

```
def binome_mem1(k,n,d={}):
    """
    Arguments :
    k, n : entiers
    dico : dictionnaire qui contient les valeurs déjà calculées
    Retours :
    c : valeur de k parmi n
    """
    if (k,n) in d: # il a déjà été calculé, on renvoie la valeur
        return d[(k,n)]
    # on traite les cas où la valeur n'a pas déjà été calculées
    elif k==n or k==0:
        c=1
    elif k==1:
        c=n
    else:
        c=binome_mem1(k-1,n-1,d)+binome_mem1(k,n-1,d) # appels
  récursifs
    d[(k,n)]=c # on ajoute  la valeur au dictionnaire
    return c
```

R29. En utilisant la mémoïsation, écrire la fonction `binome_mem2(k,n)`. On définira une sous-fonction `cb(i,j)` au sein de `binome_mem2` qui calcule $\binom{j}{i}$ et complète un dictionnaire `d` variable locale pour `binome_mem2`, globale pour `cb`.

Solution:

```
def binome_mem2(k,n):
    d={} # variable locale pour binome_mem2, variable globale pour cb
    def cb(i,j):
        """
        Fonction auxiliaire qui calcule la valeur de i parmi j et
   stocke dans dico les valeurs de i parmi j calculées
        """
        if (i,j) in d: # il a déjà été calculé, on renvoie la valeur
            return d[(i,j)]
    # on traite les cas où la valeur n'a pas déjà été calculées
        elif i==j or i==0:
            c=1
        elif i==1:
            c=j
        else:
            c=binome_mem2(i-1,j-1)+binome_mem2(i,j-1) # appels
   récursifs
        d[(i,j)]=c # on ajoute  la valeur au dictionnaire
        return c
    return cb(k,n) # calcule k parmi n
```

R30. Écrire une fonction `binome_asc(k:int,n:int)->list[list]` qui calcule le coefficient binomial en utilisant l'approche ascendante. Pour cela, on remplira un tableau `tab` (de `k+1` colonnes et de `n+1` lignes) après l'avoir initialisé avec des 0. La case ligne `j` et colonne `i` contient $\binom{j}{i}$.

Solution:

```
def binome_asc(k,n):
    """
    Arguments :
    k, n : entiers
    Renvoi : tab[n][k], où tab est un tableau de n+1 lignes et k+1
   colonnes qui va stocker les valeurs successives de i parmi j
    """
    tab=[ [0 for i in range(k+1)] for j in range(n+1) ] # n+1 lignes de
    k+1 colonnes de 0
    for j in range(0,n+1): # remplissage de la première colonne (i=0)
        tab[j][0] = 1
    for j in range(0,k+1): # remplissage de la diagonale
        tab[j][j] = 1
    for j in range( 2 , n+1 ): # remplissage des lignes  (les 2
   premières sont déjà remplies)
        for i in range( 1, min(j-1,k)+1): # remplissage des colonnes
            tab[j][i]=tab[j-1][i]+tab[j-1][i-1]
    return tab[n][k]
```

VII.2 IA

R31. 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)
```

R32. 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 proposera deux versions : la première en utilisant la fonction précédente, la deuxième sans l'utiliser.

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
```

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
```

R33. Écrire la fonction `barycentre(L:list[list])->list` qui prend en entrée une liste de listes de coordonnées, et renvoie les coordonnées du barycentre de ces points sous la forme d'une liste.

Solution:

```
def barycentre(L:list)->list:
    """
    Entrée : liste L des données, liste de listes de m éléments (m
   dimensions)
    Renvoie : liste de m éléments :coordonnées du barycentre
    """
    m=len(L[0]) # dimension de l'espace
    coord_G=m*[0] # initialisation des coordonnées du barycentre
    for i in range(m) : # on calcule la coordonnés x_{Gi} de G
```

```
        for j in range(len(L)): # parcours de toutes les données
            coord_G[i]=coord_G[i]+L[j][i] # on somme les coordonnées
        coord_G[i]=coord_G[i]/len(L) # calcul de la moyenne.
    return coord_G
```

VII.3 Théorie des jeux

R34. Écrire la fonction`transpose(G:dict)->dict` qui prend en entrée un graphe `G` représenté sous la forme d'un dictionnaire des successeurs et renvoie le dictionnaire des précédesseurs.

Solution:

```
def transpose(G):
    """
    Argument : G : graphe, représenté par son dictionnaire d'adjacence
    Retour : Gt : transposé de G, les clés de Gt sont des sommets, la valeur correspondante est la liste des prédécesseurs
    """
    Gt={s:[] for s in G} # initialisation du graphe de la transposée avec toutes les clés de G
    for s in G: # pour chaque sommet de G
        for v in G[s] : # parcours des successeurs v de s
            Gt[v].append(s)  # s est un prédécesseur de v : ajout de s à Gt[v]
    return Gt
```

R35. Écrire la fonction `degres_sortants(G:dict)->dict` qui prend en entrée le graphe `G` représenté sous la forme d'un dictionnaire des successeurs et renvoie un dictionnaire des degrés sortants des sommets de `G`.

Solution:

```
def degres_sortants(G):
    """
    Argument :  G : graphe, représenté par son dictionnaire d'adjacence
    Retour : d, dictionnaire des degrés entrants des sommets du graphe transposé de G, donc des degrés sortants de G
    """
    deg_sort={} # dictionnaire des degrés sortants de G
    # calcul du degré sortant de chaque sommet de G
    for s in G:
        d_deg_sort[s]=len(G[s])
    return deg_sort
```

R36. Écrire une fonction `predecesseurs(G:dict,j:int)->list` qui renvoie la liste de tous les prédécesseurs du sommet `j` dans le graphe `G` représenté sous la forme d'un dictionnaire des successeurs.

Solution:

```
def predecesseurs(G,j):
    prec_j=[] liste des prédécesseurs de j
```

```
    for c in G: # parcours des clés du dictionnaires
        for x in G[c]: # parcours des successeurs de c
            if x==j: # j est un successeur de c
                prec_j.append(c) # c est un prédécesseur de j
    return prec_j
```