```python
#Correction du TP de révisions - Révisions II

#EX1
def dichotomie(n):
    """dichotomie(n:int)->float
    """
    f=lambda x:x**2-2
    a0,b0=0,10
    #on programme le calcul des termes d'une suite récurrente à un pas
    for k in range(1,n+1):
        c0=(a0+b0)/2 #on identifie le point milieu
        if f(a0)*f(c0)<=0:
            a1,b1=a0,c0
        else:
            a1,b1=c0,b0
        a0,b0=a1,b1 #on replace les valeurs pour le prochain passage
    return a0,b0

def approx1(eps):
    """approx1(eps:float)->int
    """
    n=0
    while abs(dichotomie(n)[1]-dichotomie(n)[0])>eps:
        n=n+1
    return n

#EX2
def tribulles(L:list):
    for k in range(0,len(L)): #on va faire n passages pour replacer tous les éléments
        for i in range(0,len(L)-k-1): #puis, à chaque passage, on remonte la bulle si
nécessaire, par des échanges successifs.
            if L[i+1]<L[i]:
                L[i],L[i+1]=L[i+1],L[i]
            else:
                pass
    return L

def insertion(L:list,x:float):
    #on va chercher la place de x dans cette liste triée
    i=0
    while i<len(L) and x>L[i]: #attention avec la boucle while, il faut éviter de
sortir de la liste : ce sont des effets de bord qu'il faut gérer convenablement
        i=i+1
    return L[:i]+[x]+L[i:]

def triinsertion(L:list):
    L2=[L[0]] #on initialise notre liste triée
    for i in range(1,len(L)):
        L2=insertion(L2,L[i]) #puis on insère dans L2 les éléments à chaque étape
    return L2

def trirapide(L:list):
    if len(L)==0 or len(L)==1:
        return L
    else:
        m=len(L)//2
        L1,L2=[],[]
        for k in range(0,len(L)):
            if k!=m and L[k]<=L[m]:
                L1.append(L[k])
            elif k!=m and L[k]>L[m]:
                L2.append(L[k])
        return trirapide(L1)+[L[m]]+trirapide(L2)

#En fait, à chaque étape, on peut considérer qu'on a placé L[m], et ainsi on est
ramené à des sous-problèmes de taille n-1, puis n-2... ce qui permet d'accrocher la
condition d'arrêt à un moment. Le programme peut alors remonter la pile d'appels
```

```
récursifs pour renvoyer la liste triée.

#EX3
def evaluation(P:list,a:float):
    S=0
    for k in range(0,len(P)):
        S=S+P[k]*a**k
    return S

def derive(P:list):
    n=len(P)-1 #on définit ainsi le degré de P
    D=[]
    for k in range(0,n):
        D.append((k+1)*P[k+1])
    return D

#Une racine est multiple si elle annule P, et au moins P'... son ordre de
multiplicité dépendra alors du nombre de pôlynômes dérivés successifs qui
s'annuleront.

def multiple(P:list,a:float):
    D=derive(P)
    if evaluation(P,a)==0 and evaluation(D,a)==0:
        return True
    else:
        return False

def multiple2(P:list,a:float):
    val=0
    if evaluation(P,a)==0:
        val=val+1
        while evaluation(derive(P),a)==0:
            val=val+1
            P=derive(P)
        return True,val
    else:
        return False,val

#EX4
from numpy import *
G=array([[0,1,1,0,0,0],[1,0,0,1,0,0],[1,0,0,1,0,0],[0,1,1,0,1,1],[0,0,0,1,0,0],
[0,0,0,1,0,0]])

#On va parcourir le tableau er regarder la ième ligne.
def voisins(M,i):
    """voisins(M:array,i:int)->list"""
    p,q=shape(M)
    assert 0<=i<=p
    L=[]
    for j in range(0,q):
        if M[i,j]!=0: #si le poids est non nul, j est voisin de i
            L.append(j) #on stocke alors j
        else:
            pass
    return L

#On peut utiliser le programme précédent et additionner le nombre de voisins de
chaque sommet... le nombre d'arêtes sera alors immédiat.
def degretotal(M):
    """degretotal(M:int)->int"""
    p,q=shape(M)
    S=0
    for i in range(0,p):
        S=S+len(voisins(M,i))
    return S

def aretes(M):
```

```
    """aretes(M:int)->int"""
    deg=degretotal(M)
    return int(deg/2)

#EX5
from math import *
def bernstein(n,k,x):
    """bernstein(n:int,k:int,x:float)->float
    Renvoie l'image de x par le polynôme de Bernstein associé.
    """
    coeft=factorial(n)/(factorial(k)*factorial(n-k))
    return coeft*(x**k)*(1-x)**(n-k)

#on rappelle que pour représenter une fonction, on cherche à construire deux listes
X,Y, l'une contenant les abscisses, l'autre les images par la fonction, puis on trace
les points à l'aide de l'instruction : plot(X,Y).

from matplotlib.pylab import *
def representation(n):
    """representation(n:int)->None
    Affiche les courbes associées aux polynomes de Bernstein.
    """
    X=linspace(0,1,100)
    #on va alors faire une boucle pour afficher les graphes des polynômes B_n,0, B_n,
1,..., B_n,n.
    for k in range(0,n+1):
        Y=[bernstein(n,k,x) for x in X]
        plot(X,Y,label="B_"+str(n)+","+str(k))
    legend()
    show()

def polynome(n,x):
    """polynome(n:int,x:float)->float
    Renvoie l'image de x par le polynôme P_n.
    """
    #on définit la fonction f avec la commande lambda.
    f=lambda x:log(1+x)
    #on va faire une boucle pour calculer la somme demandée.
    S=0
    for k in range(0,n+1):
        S=S+f(k/n)*bernstein(n,k,x)
    return S

def approximation(n):
    """approximation(n:int)->None
    Affiche les graphes de f, P_n, P_5n, P_10n.
    """
    f=lambda x:log(1+x)
    X=linspace(0,1,100)
    Y=[f(x) for x in X]
    plot(X,Y,label="f")
    for N in [n,5*n,10*n]:
        Y=[polynome(N,x) for x in X]
        plot(X,Y,label="P_"+str(N))
    legend()
    show()
```