CORRECTION DM 2

Partie 1 :

1)
```
def Verif(c, Obj):
    s = 0
    for x in s:
        s += x          # s = sum_{i=0}^{n-1} c[i]
    return s >= Obj
```

2).
```
def EstCroissante(c):
    n = len(c)
    for j in range(n-1):
        if c[j] > c[j+1]:   # croissance non vérifiée
            return False
    return True             # fin des tests.
```

- La fonction est composée d'une boucle for de longueur au plus $n-1$ et d'un nombre constant d'opérations. Sa complexité est donc $O(n)$ c'est-à-dire **linéaire.**

3). On a $t[i] = \sum_{j=0}^{i} c[j]$ et cette fonction renvoie la liste $[t[0], \dots, t[n-1]]$, c'est-à-dire la liste des sommes cumulées de c.

- La fonction est composée de :
  - affectation d'une liste de longueur n
  - boucle pour $i \in [\![0, n-1]\!]$, composée d'une boucle de longueur $i+1$ de 2 opérations

Ainsi, on effectue $n + \sum_{i=0}^{n-1} 2(i+1) = O(n^2)$ opérations.

Donc la complexité de fonction 1 est :

quadratique.

- fonction 2(c) :
```
n = len(c)
t = [ ]
s = 0
for i in range(n):
    s += c[i]           # s = sum_{k=0}^{i} c[k] = t[i]
    t.append(s)
return t.
```

La complexité de fonction 2 est : **linéaire**

Partie 2 :

1- La fonction vérifie que a et c aient même longueur et, si c'est le cas, elle renvoie la somme des chiffres d'affaire des entreprises de l'alliance.
Sa complexité est clairement : **linéaire.**

2- fonction renvoie True si l'alliance est réussie et False sinon. Sa complexité est la même que celle de fonction 3, c'est-à-dire : **linéaire.**

3-
```
def est_stable(c, a, Obj):
    if fonction4(c, a, Obj) == False:  # l'alliance doit être réunie
        return False
    else:
        s = fonction3(c, a)  # chiffre d'affaire de l'alliance
        for i in range(len(c)):
            if a[i] == True and s - c[i] >= Obj:
                # l'alliance reste réunie en enlevant une entreprise
                return False
        return True
```

4- Pour minimiser le nombre d'entreprises, on commence par choisir les entreprises ayant le chiffre d'affaire le plus élevé.
```
def alliance_Min(c, Obj):
    l = []
    s = 0
    for i in range(1, n+1):
        s += c[-i]  # somme en commençant par le dernier
        l.append(n-i)  # numéro de l'entreprise
        if s >= Obj:  # objectif atteint.
            return l
```

Partie 3 :

1) La représentation binaire est $\overline{10100}^2$ qui vaut $2^4 + 2^2 =$ **20**

2)
```
def suivante_De(a):
    s = 0
    n = len(a)
    for i in range(n):  # calcul du nombre décimal associé à a.
        if a[i] == True:
            s += 2**i
    s += 1  # suivant
    if s == 2**n:  # suivant est n+1 bits
        return False
    else:
        for i in range(n):
            r = s%2  # décomposition binaire de s
            s = s//2
            if r == 1:  # modification de a.
                a[i] = True
            else:
                a[i] = False
        return True
```

3)
```
def imprimer(a):
    l = []
    for i in range(len(a)):
        if a[i] == True:
            l.append(i)
    return l
```

4)
```
def imprimerStables(c, Obj):
    a = [False] * len(c)        # alliance vide
    l = []                      # liste des alliances stables
    b = True                    # successeur existe
    while b == True:
        b = suivantDe(a)        # on teste le successeur de a
        if estStable(c, a, Obj):    # l'alliance est stable.
            l.append(imprimer(a))
    return l
```

En notant $n = len(c)$, on a effectué une boucle while de longueur 2^n composé d'appels à des fonctions en $O(n)$. La complexité de cette fonction est donc $\boxed{O(n\,2^n)}$.

Cette complexité est beaucoup trop grande pour que la fonction soit utilisable.