CORRECTION DS 2 INFORMATIQUE

1) • La cargaison à 4 produits ne respecte pas le poids maximal.

• Une cargaison à 1 produit ne respecte pas l'optimisation du profit.

• La cargaison à 2 produits maximisant la valeur est constituée des produits 0 et 3 de valeur 13 et de poids 7.

• La cargaison à 3 produits maximisant la valeur et respectant le poids maximal est constituée des produits 0, 2 et 3 de valeur 14 et de poids 8.

$\boxed{\text{La cargaison est donc 0, 2 et 3 et le profit 14.}}$

2-a)

```python
def Ratio(P,V):
    n = len(P)
    R = []   # initialisation de la liste des ratios.
    for i in range(n):
        R.append( V[i]/P[i])
    return R.
```

La fonction est composée d'une boucle for de longueur n et d'un nombre constant d'opérations, sa complexité est donc $O(n)$ c'est-à-dire : $\boxed{\text{linéaire.}}$

b)

```python
def TriSelection(l):
    n = len(l)
    for i in range(n):
        imin = i      # recherche du minimum à partir
                      # du rang i
        m = l[i]
        for j in range(i+1,n):
            if l[j]<m:
                imin = j
                m = l[j]
        l[imin], l[i] = l[i], l[imin]   # échange avec
                                         # le minimum
    return l.
```

La complexité de cette fonction est : $\boxed{\text{quadratique.}}$

c)

```python
def Inverse(l):
    n = len(l)
    L = []    # initialisation de la liste inverse.
    for i in range(1,n+1):
        L.append (l[-i])   # on commence par regarder
                           # le dernier élément.
    return L
```

• Seule une boucle for de longueur n est utilisée, la fonction est donc bien de complexité linéaire.

• La liste l en argument n'est pas modifiée, la fonction n'a donc pas d'effet de bord.

d)
```
def TriRatio (P,V) :
    return Inverse (TriSelection ( Ratio (P,V)))
```

e)
```
def TriRatio2 (P,V) :
    l = Ratio (P,V)
    n = len (l)
    for i in range (n) :
        imax = i          # tri par ordre décroissant, on cherche
        m = l[i]          # donc le maximum à partir du rang i
        for j in range (i+1,n) :
            if l[j] > m :
                imax = j
                m = l[j]
        l[imax], l[i] = l[i], l[imax]     # échange dans toutes les listes.
        P[imax], P[i] = P[i], P[imax]
        V[imax], V[i] = V[i], V[imax]
    return P,V
```

f)
```
def Vmax (P,V, Pmax) :
    P,V = TriRatio2 (P,V)     # tri décroissant.
    poids = 0                 # initialisation du chargement
    valeur = 0
    i = 0
    while i < len(P) and poids+P[i]<=Pmax:   # on peut rajouter i
        poids += P[i]
        valeur += V[i]
        i += 1
    return valeur
```

Cette fonction est composée d'une boucle while de longueur au plus n contenant un nombre constant d'opérations et d'un appel à TriRatio2 de complexité quadratique.

Sa complexité est donc : $\boxed{\text{quadratique}}$

g) La liste des ratios est : $[\frac{4}{3}, \frac{3}{2}, 1, \frac{9}{4}]$.

Les listes triées sont donc : $P = [4,2,3,1]$ $V = [9,3,4,1]$

Le résultat renvoyé par Vmax correspond à la somme des 2 premières valeurs c'est-à-dire $\boxed{12}$.

En comparant avec la question 1, on voit que ce résultat n'est pas optimal.

On a utilisé un algorithme glouton non exact c'est-à-dire une $\boxed{\text{heuristique gloutonne.}}$

3-a) • Si $i=0$, il n'y a pas de produit à placer donc la valeur est nulle : $S(i,w) = 0$.

• Si $i>0$ et $p_{i-1} > w$, le i^e produit dépasse la charge maximale, il ne pourra pas être utilisé.

Donc $S(i,w) = S(i-1,w)$

• Si $i>0$ et $p_{i-1} \leq w$, si on n'utilise pas le i^e produit la valeur maximale est $S(i-1,w)$ et si on utilise le i^e produit, la valeur maximale est

$$v_{i-1} + S(i-1, w - p_{i-1})$$ car le poids maximal

en dehors du i^e produit est $w - p_{i-1}$. Donc, en optimisant :

$$S(i,w) = \max\left(S(i-1,w),\ v_{i-1} + S(i-1, w-p_{i-1})\right)$$

b) L'indice i décrit une suite strictement décroissante d'entiers de pas 1 donc le cas de base $i=0$ est atteint.

Ainsi, l'algorithme se termine.

c)
```
def Max(a, b):
    if a >= b:
        return a
    else:
        return b
```

d)
```
def recur(P, V, i, w):
    if i == 0:
        return 0
    elif P[i-1] > w:
        return recur(P, V, i-1, w)
    else:
        return Max(recur(P, V, i-1, w), V[i-1]
                   + recur(P, V, i-1, w-P[i-1]))
```

e) Pour répondre au problème, on exécute :

recur(P, V, n, Pmax)

où $n = \text{len}(P)$.

f) On note $C(i)$ le coût de l'appel à recur(P, V, i, w).

On a : $C(i) = 0$ ou $C(i-1)$ ou $2C(i-1)$ donc

dans le pire des cas : $C(i) = 2C(i-1)$

Ainsi $C(i) = 2^i C(0) = 2^i$.

Donc la complexité est $O(2^i)$ c'est-à-dire $\boxed{\text{exponentielle.}}$

En pratique, on ne peut pas utiliser cette fonction.

4-a) Memoire = [[-1 for j in range(Pmax+1)] for i in range(n+1)]

b)
```
def recur2(P, V, i, w, Memoire):
    if i == 0:
        return 0
    elif Memoire[i][w] > -1:   # cas où on a déjà calculé S(i,w)
        return Memoire[i][w]
    elif P[i-1] > w:
        Memoire[i][w] = recur2(P, V, i-1, w, Memoire)
                                  # on enregistre S(i,w)
        return Memoire[i][w]
```

```
else:
    if Memoire[i-1][w - P[i-1]] == -1 :
                              # S(i-1, w - p_{i-1}) non calculé
        Memoire[i-1][w - P[i-1]]
                = recur2(P, V, i-1, w - P[i-1], Memoire)
    if Memoire[i-1][w] == -1 :   # S(i-1, w) non calculé
        Memoire[i-1][w] = recur2(P, V, i-1, w, Memoire)
    Memoire[i][w] = Max( Memoire[i-1][w],
                    V[i-1] + Memoire[i-1][w - P[i-1]])
    return Memoire[i][w]
```

c). Les appels récursifs ne se font que si la valeur n'est pas déjà connue. Le nombre d'appels récursifs est donc majoré par la taille de la matrice Memoire, c'est-à-dire : $\boxed{(n+1) * (P_{max}+1)}$.

- La complexité dépend de P_{max} mais elle n'est plus exponentielle. Cette fonction est donc plus efficace que celle de la question 3.

Partie 2 :

1) $39 = 2 \times 19 + 1$, $19 = 2 \times 9 + 1$, $9 = 2 \times 4 + 1$, $4 = 2 \times 2 + 0$, $2 = 2 \times 1 + 0$, $1 = 2 \times 0 + 1$

Le codage est donc : $\boxed{\text{'00100111'}}$

2)
```
def identifiant(Bin):
    n = len(Bin)
    s = 0
    for i in range(n):
        s += int(Bin[i]) * 2 ** (n-i-1)   # Bin[i] est une chaîne de caractère.
    return s
```

Cette fonction est composée d'une boucle for sur $i \in [\![0, n-1]\!]$ composée d'une incrémentation, d'une conversion, d'un produit et d'une puissance $n-i-1$ soit de $n-i+2$ opérations. Son coût est donc : $\sum_{i=0}^{n-1} (n-i+2) = O(n^2)$.

Sa complexité est donc $\boxed{\text{quadratique.}}$

3)
```
def identifiant2(Bin):
    n = len(Bin)
    s = 0
    p = 1          # stockage des puissances.
    for i in range(n):
        s += int(Bin[n-i-1]) * p    # à partir du dernier
        p *= 2                      # p est une puissance de 2
    return s
```

La fonction n'est composée que d'une boucle for et d'un nombre constant d'opérations, elle est donc bien linéaire.