CCINP TSI 2021 - Correction

Q1. Quitte à énumérer toutes les cargaisons constituées d'un ou deux produits, on voit qu'il est toujours possible d'ajouter un produit supplémentaire tout en respectant la contrainte de poids maximal, aucune de ces solutions ne maximise donc le profit.

A contrario, une cargaison constituée de quatre produits a un poids de 10, ce qui dépasse la valeur $P_{\max}$, ce n'est donc pas une solution acceptable.

Q2. Trois cargaisons de trois produits respectent le poids maximal :

- La cargaison constituée des produits 1, 2 et 3 a un poids de 6 et donne un profit de 8.
- La cargaison constituée des produits 1, 3 et 4 a un poids de 8 et donne un profit de 14.
- La cargaison constituée des produits 2, 3 et 4 a un poids de 7 et donne un profit de 13.

Q3. D'après la question précédente, la cargaison constituée des produits 1, 3 et 4 maximise le profit et donne un profit de 14.

Q4. On peut procéder avec des `append` :

```
1 def ListeProduits(n):
2     prod = []
3     for i in range(1, n+1):
4         prod.append(i)
5     return prod
```

ou avec une compréhension de liste :

```
1 def ListeProduits(n):
2     return [i for i in range(1, n+1)]
```

Q5. La construction est analogue à celle de la question précédente.

```
1 def Ratio(P, V):
2     ratios = []
3     for i in range(len(P)):
4         ratios.append(V[i] / P[i])
5     return ratios
```

ou

```
1 def Ratio(P, V):
2     return [V[i] / P[i] for i in range(len(P))]
```

Q6. Comme `len(L)` vaut 4, i va prendre les valeurs de 1 à 3 soit 3 itérations de la boucle for.

En exécutant le code, on obtient :

- après $i = 1$, on a `[3, 5, 2, 1]` ;
- après $i = 2$, on a `[2, 3, 5, 1]` ;
- après $i = 3$, on a `[1, 2, 3, 5]`.

Autrement dit, à la fin de l'itération i les éléments jusqu'à l'indice i inclus sont triés.

Q7. Cette fonction de tri ne fonctionnerait pas pour une chaîne de caractères car celles-ci sont non mutables. Les lignes 7 (`L[j] = L[j-1]`) et 9 (`L[j] = x`) provoqueraient alors une erreur du type `'str' object does not support item assignment`

Q8. La fonction `Tri` repose sur le principe du tri par insertion. En notant n la longueur de la liste passée en argument, on a :

- Meilleur des cas : le tableau est déjà trié par ordre croissant. La boucle `for` est réalisée $n-1$ fois et la condition dans le `while` est toujours fausse, une seule comparaison est donc effectuée. On aboutit à une complexité en $\mathcal{O}(n)$, *i.e.* linéaire.
- Pire des cas : le tableau est trié par ordre décroissant. Pour l'itération `i` de la boucle `for`, la seconde condition du `while` est toujours vraie, l'arrêt se fait donc lorsque `j` prend la valeur 0, c'est-à-dire après `i` comparaisons. On a ainsi $\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$ comparaisons, on obtient donc une complexité en $\mathcal{O}(n^2)$, *i.e.* quadratique.

Q9. On procède de manière analogue aux questions **Q4** et **Q5**.

```
def Inverse(L):
    inv = []
    for i in range(len(L)):
        inv.append(L[len(L)-i-1])
    return inv
```

ou avec une compréhension de liste et des indices négatifs :

```
def Inverse(L):
    return [L[-i-1] for i in range(len(L))]
```

Q10. La fonction `Tri` permettrait de trier la liste obtenue avec la fonction `Ratios` mais pas les listes de poids et de valeurs ensemble.

Q11. On garde l'idée de la fonction `Tri` sur la liste des ratios mais à chaque étape on effectue les mêmes modifications sur les listes `P` et `V`.

```
def Tri2(P, V):
    R = ratio(P, V)
    for i in range(1, len(R)):
        p, v, r = P[i], V[i], R[i]
        j = i
        while j > 0 and r < R[j-1]:
            R[j], P[j], V[j] = R[j-1], P[j-1], V[j-1]
            j = j-1
        R[j], P[j], V[j] = r, p, v
    return inverse(P), inverse(V)
```

Q12. Tant qu'il reste des produits et que l'ajout du suivant ne fait pas dépasser `Pmax`, on prend ce produit :

```
def vmax(P, V, Pmax):
    P2, V2 = Tri2(P, V)
    SP = 0
    SV = 0
    i = 0
    while i < len(P) and SP + P2[i] <= Pmax:
        SP = SP + P2[i]
        SV = SV + V2[i]
        i = i+1
    return SV
```

Q13. Les ratios valent, dans l'ordre des produits, 4/3, 3/2, 1 et 9/4. Après la fonction `Tri2`, les ratios sont `[2.25, 1.5, 1.33, 1]`, les poids `[4, 2, 3, 1]` et les valeurs `[9, 3, 4, 1]`.
La fonction `Vmax` renvoie alors 12 (car $4+2 \leqslant 8$ mais $4+2+3 > 8$). Cette solution est non optimale d'après ce que l'on a vu en **Q3**.

Q14.

- *Justification pour* $i = 0$: la valeur est nulle puisqu'il n'y a aucun produit.
- *Justification pour* $i > 0$ *et* $p_i > \omega$: le i-ème produit ayant un poids dépassant la capacité maximale, il ne sera jamais pris et la valeur maximale sera la même que celle avec les $i-1$ premiers produits.
- *Justification pour* $i > 0$ *et* $p_i \leqslant \omega$: on peut prendre le i-ème produit car son poids ne dépasse pas la capacité maximale. Il y a alors deux cas possibles parmi lesquels on prend l'optimal (le max). Premièrement, si on ne prend pas ce i-ème produit, la valeur maximale de la cargaison est la même qu'avec les $i-1$ premiers produits. Deuxièmement, si on prend le i-ème produit alors la valeur de la cargaison est la valeur de celui-ci (v_i) à laquelle on ajoute la valeur maximale obtenue avec les $i-1$ premiers produits et une capacité maximale de $\omega - p_i$ (puisque p_i est pris par le i-ème produit).

Q15. L'algorithme termine lorsque i vaut 0 (cas d'arrêt). Or à chaque appel récursif la valeur de i est décrémentée de 1. Ainsi en partant de $n \geqslant 0$, on parviendra toujours au cas d'arrêt, l'algorithme termine donc.

Q16.

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

Q17. Il suffit de retranscrire les trois cas de la relation de récursivité dans la fonction en faisant attention au décalage d'indice : les produits $p_1, \ldots, p_n$ correspondent à $P[0], \ldots, P[n-1]$.

```
def recur(P, V, i, w):
    if i == 0:
        return 0
    # On arrive ici seulement si i>0
    if 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]))
```

Q18.

```
recur([3, 2, 1, 4], [4, 3, 1, 9], 4, 8)
```

Q19. Le singulier dans l'énoncé force à utiliser deux compréhensions de liste imbriquées :

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

Si on s'autorise plusieurs instructions, on peut aussi proposer :

```
Memoire = []
for i in range(n+1):
    ligne = []
    for j in range(Pmax + 1):
        ligne.append(-1)
    Memoire.append(ligne)
```

Q20. *Remarque : on utilise le principe de mémoïsation qui consiste à garder en mémoire des calculs intermédiaires pour ne pas les effectuer plusieurs fois. Cela se traduit ici sur les tests pour voir si une case dans* `Memoire` *vaut* -1 *(correspond à un calcul jamais fait donc à faire) ou une valeur strictement plus grande (correspond à un calcul déjà effectué, on peut alors directement renvoyer la valeur).*

```
def recur2(P, V, i, w, Memoire):
    if i == 0:
```

```
        return 0
    if Memoire[i][w] > -1:  # le calcul a déjà été effectué
        return Memoire[i][w]  # donc on renvoie la valeur
    if P[i-1] > w:
        Memoire[i][w] = recur2(P, V, i-1, w, Memoire)
        return Memoire[i][w]
    else:
        if Memoire[i-1][w] == -1:  # jamais calculé donc on le fait
            Memoire[i-1][w] = recur2(P, V, i-1, w, Memoire)
        if Memoire[i-1][w-P[i-1]] == -1:  # idem
            Memoire[i-1][w-P[i-1]] = recur2(P, V, i-1,
                                            w-P[i-1], Memoire)
    a = max(Memoire[i-1][w], V[i-1] + Memoire[i-1][w-P[i-1]])
    Memoire[i][w] = a
    return Memoire[i][w]
```

Q21. Aucun attribut ne permet à lui-seul d'identifier une entrée. Une clé primaire est donnée par le triplet (`date`, `heure`, `id_client`) car il ne peut y avoir deux livraisons simultanées chez le même client.

Q22.
```
SELECT id_client
FROM livraison
WHERE date = "10-01-2021"
```

Q23.
```
SELECT date, heure
FROM livraison
JOIN client ON id_client = id
WHERE zone = 5 AND date = "02-03-2021"
```

Q24.
```
SELECT COUNT(*)
FROM livraison
JOIN local ON id_local = id
WHERE date = "03-02-2021" AND zone1 <= 10 AND zone2 <= 10
                                          AND zone3 <= 10
```

Q25. On a $39 = 32 + 4 + 2 + 1 = 2^5 + 2^2 + 2^1 + 2^0$ donc 39 est codé par 00100111.

Q26. Les deux erreurs se trouvent à la ligne 6 :

- `Bin[i]` est une chaîne de caractères, on doit la convertir en entier via `int(Bin[i])` pour pouvoir effectuer des calculs avec ;
- il y a un décalage dans l'exposant de 2 : il faut `len(Bin)-i-1` et non `len(Bin)-i` (si besoin se convaincre avec les cas limites `i = 0` et `i = len(Bin)-1`).

Q27. On calcule les puissances de 2 de proche en proche : on commence par 1 et on multiplie par 2 à chaque tour de boucle.

```
def Identifiant2(Bin):
    S = 0
    puissance = 1
    for i in range(len(Bin)):
        S = S + int(Bin[len(Bin)-1-i])*puissance
        puissance = 2*puissance
    return S
```

Q28. Il y a 30 zones donc il faut au moins $\lceil \log_2(30) \rceil$ bits, *i.e.* 5 bits (car $2^4 = 16 < 30 \leqslant 32 = 2^5$).