DS 1 : lundi 3 novembre

Correction

Partie I - Données liées aux livraisons conservées par l'entreprise

Q1. La table livraison possède une clé primaire qui est id, elle possède deux clés étrangères qui sont id_client et id_local

Q2.
```
SELECT id_client
FROM livraison
WHERE date = "10-01-2021"
```

Q3.
```
SELECT id
FROM local
WHERE zone1 = 10 OR zone2 = 10 OR zone3 = 10
ORDER BY id ASC
```

Q4.
```
SELECT date, heure
FROM livraison
    JOIN client ON livraison.id_client = client.id
WHERE zone = 5 AND date = "02-03-2021"
```

Q5.
```
SELECT DISTINCT l.id_client
FROM livraison l
    JOIN local lo ON l.id_local = lo.id
WHERE lo.zone1 = 15;
```

Q6.
```
SELECT COUNT (*)
FROM livraison
    JOIN local ON id_local = id
WHERE date = "03-02-2021" AND zone1 <= 10 AND zone2 <= 10 AND zone3 <= 10
```

Q7.
```
SELECT c.id, COUNT(l.id_client) AS nombre_livraisons
FROM client c
    JOIN livraison l ON c.id = l.id_client
WHERE l.date LIKE '%-2021'
GROUP BY c.id
HAVING nombre_livraisons>=5;
```

Q8.
```
SELECT lo.id FROM local lo
WHERE lo.id NOT IN (SELECT DISTINCT l.id_local FROM livraison l
                        JOIN client c ON l.id_client = c.id
                    WHERE c.zone = lo.zone1 AND (l.date LIKE '%-01-2021' OR
                                                 l.date LIKE '%-02-2021' OR
                                                 l.date LIKE '%-03-2021' OR) );
```

Q9.
```
SELECT livraison.id
FROM livraison
    JOIN client ON livraison.id_client = client.id
WHERE livraison.date = '16-06-2020'
    AND client.zone > (SELECT AVG(client.zone)
                       FROM livraison
                           JOIN client ON livraison.id_client = client.id
                       WHERE livraison.date = '16-06-2020');
```

Partie II - Optimisation du chargement

II. 1 - Un exemple

Q10. Lorsqu'on ne met que 1 produit, on a une valeur maximale de 9
Lorsqu'on ne met que 2 produits, on a une valeur maximale de 13 (produits 1 et 4)
Or il est clair qu'en mettant les produits 1,3 et 4 on a une valeur plus élevée : 14
Lorsqu'on met 4 produits, le poids maximal est dépassé.

Q11. 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 .

Q12. D'après la question précédente, la cargaison constituée des produits 1,3 et 4 maximise le profit et donne un profit de 14.

II. 2 - Une méthode intuitive pour la résolution du problème

Q13.
```
def ListeProduits(n):
    prod = []
    for i in range(1, n+1):
        prod.append(i)
    return prod
```
ou avec une compréhension de liste :
```
def ListeProduits(n):
    return [i for i in range(1, n+1)]
```

Q14. La construction est analogue à celle de la question précédente.
```
def Ratio(P, V):
    ratios = []
    for i in range(len(P)):
        ratios.append(V[i] / P[i])
    return ratios
```
ou
```
def Ratio(P, V):
    return [V[i] / P[i] for i in range(len(P))]
```

Q15. 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.

Q16. 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.

Q17. 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 $sum_{i=1}^{n-1} i = \dfrac{n(n-1)}{2}$ comparaisons, on obtient donc une complexité en $\mathcal{O}\left(n^2\right)$, i.e. quadratique.

Q18. On procède de manière analogue aux questions Q13 et Q14.

```
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))]
```

Q19. La fonction Tri permettrait de trier la liste obtenue avec la fonction Ratios mais pas les listes de poids et de valeurs ensemble.

Q20. 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)
```

Q21. 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: #(A)
        SP = SP + P2[i]
        SV = SV + V2[i]
        i = i+1
    return SV #(B)
```

Q22. 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 Q12.

II. 3 - Une méthode récursive

Q23.
- 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).

Q24. 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.

Q25.
```
def Max(a, b):
    if a > b:
        return a
    else:
        return b
```

Q26. Il suffit de retranscrire les trois cas de la relation de récursivité dans la fonction en faisant attention au décalage d'indice : les produits $p_1, \dots, p_n$ correspondent à $P[0], \dots, P[n-1]$.

```
def recur(P, V, i, w):
    if i == 0:
        return 0 #(C)
    # 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])) #(D)
```

Q27. `recur([3, 2, 1, 4], [4, 3, 1, 9], 4, 8)`

II. 4 - Amélioration de la méthode récursive

Q28. 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)
```

Q29. 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] #(E) 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) #(F)
        if Memoire[i-1][w-P[i-1]] == -1: # idem (G)
            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] #(H)
```