Devoir maison n°2 - Corrigé du problème

Q1.

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.

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. On remarque que la cargaison maximisant le profit est 1,3,4 avec une valeur $\mathrm{V} = 14$.

Q4.

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

Q5.

```python
def Ratio(P,V):
    rapports=[]
    for i in range(len(P)):
        rapports.append(V[i]/P[i])
    return rapports
```

Q6. Il y a 3 boucles for de 1 à $\operatorname{len}(L) - 1 = 4 - 1 = 3$

À la fin de la 1ère boucle $L = [3, 5, 2, 1]$

À la fin de la 2ème boucle $L = [2, 3, 5, 1]$

À la fin de la 3ème boucle $L = [1, 2, 3, 5]$

Q7. 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.

Q8.

```python
def Inverse(L):
    N=len(L)
```

```
    renverse=[]
    for i in range(N):
        renverse.append(L[N-1-i])
    return renverse
```

Q9. Tri trie la liste suivant ses valeurs, on pourra donc trier les ratios mais pas simultanément P et V en parallèle.

Q10. 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):
    rapports=Ratio(P,V)
    for i in range(len(P)):
        x=rapports[i]
        Pval=P[i]
        Vval=V[i]
        j=i
        while j>0 and x<rapports[j-1]:
            rapports[j]=rapports[j-1]
            P[j]=P[j-1]
            V[j]=V[j-1]
            j=j-1
        rapports[j]=x
        P[j]=Pval
        V[j]=Vval
    return Inverse(P),Inverse(V)
```

Q11. 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: #tant qu'on a des produits
        SP=SP+P2[i]                   #et que le poids ajouté ne fait
        SV=SV+V2[i]                   #pas dépasser Pmax
        i=i+1
    return SV
```

Q12. 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.

Q13.

- 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).

Q14. 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.

Q15.

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

Q16. 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 $P[0], \ldots, P[n-1]$.

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

Q17.

```python
Pex=[3,2,1,4]
Vex=[4,3,1,9]
print(recur(Pex,Vex,4,8))
```

Q18.

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

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

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

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

Q19. 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).

```python
def recur2(P,V,i,w,Memoire):
    if i==0:
        return 0
    if Memoire[i][w]>-1: #déjà calculé
        return Memoire[i][w]
    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: #doit etre calculé
            Memoire[i-1][w]=recur2(P,V,i-1,w,Memoire)
        if Memoire[i-1][w-P[i-1]]==-1: #doit etre calculé
            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]
```