EXERCICE 2 (6 points)

Cet exercice porte sur la programmation orientée objets, les tris, les algorithmes gloutons, la récursivité et les assertions.

Cet exercice est composé de trois parties dont les deux dernières sont indépendantes entre elles.

Dans cet exercice, l'entête des fonctions est décrit avec le type des objets en paramètre et le type de l'objet renvoyé. Ainsi la fonction puissance qui prend un paramètre flottant `x` et un entier `n` puis qui renvoie le flottant `x**n`, a pour entête `puissance(x: float, n: int) -> float`

Une entreprise transporte des marchandises. Elle souhaite maximiser son profit en optimisant le remplissage de ses moyens de transport. On considère qu'un moyen de transport est limité par son volume (exprimé en litres). Chaque marchandise est caractérisée par son prix (en euros) et son volume indivisible (en litres).

Supposons qu'on ait trois marchandises caractérisées par les couples (prix, volume) suivants : $m_1 = (100, 10)$, $m_2 = (100, 10)$ et $m_3 = (250, 20)$. Si le moyen de transport peut encore charger 25 litres, il vaut mieux charger la marchandise numéro 3 qui rapporte 250 € à l'entreprise plutôt que charger les marchandises numéros 2 et 3 qui rapportent 200 € au total pour le même espace utilisé.

Partie A – Quelques outils

Nous souhaitons définir une classe `Marchandise` dont chaque instance définit une marchandise possédant deux attributs entiers `prix` et `volume`.

1. Compléter le constructeur qui renvoie un objet `Marchandise`. Utiliser le mot-clé `assert` afin qu'une exception soit levée si le paramètre `v` n'est pas strictement positif.
   On rappelle que si `condition` est une expression Python booléenne s'évaluant à `True` ou `False`, l'instruction `assert condition` déclenche une exception quand la condition s'évalue à `False`.

```
class Marchandise:
    def __init__(self, p: int, v: int) -> 'Marchandise':
      ...
```

2. Donner une instruction qui permet de créer une variable `m1` représentant une marchandise d'un volume de 7 litres coûtant 20 €.

3. Proposer une méthode `ratio(self) -> float` qui renvoie le ratio prix/volume d'une marchandise.

4. Proposer une fonction `prixListe(tab: list) -> int` qui renvoie le prix cumulé de l'ensemble des marchandises formant le tableau `tab`.

Partie B – Première approche de rangement

Le transporteur souhaite maximiser son profit. On considère que nous avons les quatre marchandises définies par les couples (prix, volume) suivants :

$$m_1 = (40, 20),\ m_2 = (210, 70),\ m_3 = (160, 40) \text{ et } m_4 = (50, 50).$$

5. Préciser toutes les combinaisons de marchandises possibles si on ne dépasse pas un volume de 100 litres et le prix associé. En déduire la combinaison de marchandises qui maximise le prix.

Une première méthode appelée `ChargementGlouton` consiste à trier les marchandises dans l'ordre décroissant de leur prix volumique (ratio prix/volume), puis transporter en priorité les marchandises avec le plus grand prix volumique. Si une marchandise est trop volumineuse pour être transportée, on essaie avec la marchandise ayant le prix volumique juste inférieur, ce jusqu'à ce qu'aucune marchandise ne puisse rentrer. Ainsi, en notant `v_restant` le volume disponible et `m_i` la $(i+1)^{e}$ marchandise une fois les marchandises triées, l'algorithme peut s'écrire :

`ChargementGlouton`

```
n = nombre de marchandises
POUR i ALLANT de 0 à n-1 FAIRE
| SI volume de m_i <= v_restant ALORS
|   | charger m_i
|   | v_restant = v_restant - volume de m_i
TRANSPORTER le chargement prévu
```

Le tri dans l'ordre décroissant des prix volumiques donne m_3 , m_2, m_1, m_4. Si le moyen de transport accepte 100 litres de chargement, l'algorithme charge m_3 et m_1 pour un prix de 200 € (à comparer avec la combinaison trouvée précédemment pour maximiser le prix).

Par la suite, on rappelle que dans l'implémentation Python, les marchandises sont définies par des instances de la classe `Marchandise`.

On donne quelques qualificatifs : dichotomique, glouton, graphique, insertion, maximum, récursif, tri.

6. Indiquer, sans justification, le qualificatif qui s'applique le mieux à l'algorithme précédent.

7. Recopier et compléter la fonction `tri(tab: list) -> None` ci-dessous afin qu'elle trie en place un tableau contenant des objets de type `Marchandise` selon l'ordre décroissant des ratios. Ainsi, `tab[0]` doit contenir la marchandise avec le plus haut ratio prix/volume après l'appel `tri(tab)`.

```
def tri(tab: list) -> None:
    n = len(tab)
    for i in range(1, n):
        marchandise = tab[i]
        j = i-1
        while ... and ... > ... :
            tab[j+1] = ...
            j = ...
        tab[j+1] = marchandise
```

8. Sans justifier, préciser le nom de ce tri, ainsi que son coût temporel dans le pire des cas (constant, logarithmique, linéaire, quasi-linéaire ($n \log_2 n$), quadratique, cubique ou exponentiel).

9. Recopier et compléter la fonction `charge` suivante qui applique l'algorithme `ChargementGlouton` décrit plus haut.

```
def charge(tab: list, volume: int) -> list:
    tri(tab)
    chargement = []
    n = len(tab)
    for ...
        if ...
           ...
           ...
    return ...
```

Partie C – Rangement optimisé par récursivité

L'algorithme précédent ne renvoie pas toujours une solution optimale. On peut donc suivre un algorithme récursif. On note `n` le nombre de marchandises et on souhaite implémenter la fonction récursive `chargeOptimale` d'entête :

```
chargeOptimale(tab: list, v_restant: int, i: int) -> list
```

Un appel à cette fonction doit permettre de calculer la charge optimale pour un transport de volume `v_restant` utilisant les marchandises à partir de l'indice `i` :

- si `i >= n`, toutes les marchandises ont été essayées et il n'en reste plus d'autres disponibles. L'appel récursif renvoie la liste vide ;
- si `i < n` et la marchandise d'indice `i` est de volume strictement supérieur au volume restant, l'appel récursif renvoie le résultat de l'appel effectué avec le même volume restant mais avec la marchandise suivante, c'est-à-dire `chargeOptimale(tab, v_restant, i+1)` ;

- si `i < n` et la marchandise d'indice `i` est de volume inférieur ou égal au volume restant, il existe deux options possibles :
  - Option 2 soit on utilise la marchandise `i`, auquel cas le chargement contiendra cette marchandise et celles du résultat de l'appel récursif à partir de la prochaine marchandise et d'un volume restant strictement inférieur,
  - Option 1 soit on n'utilise pas la marchandise `i`, auquel cas le chargement sera le résultat de l'appel récursif avec le même volume restant mais à partir de la marchandise suivante.

  On garde l'option de chargement qui maximise le prix transporté.

10. Compléter le code de la fonction `chargeOptimale` dont le principe a été décrit ci-avant.

```
def chargeOptimale(tab: list, v_restant: int, i: int) -> list:
    if i>=...:
        return ...
    else:
        if tab[i].volume > v_restant:
            return chargeOptimale(tab, v_restant, i+1)
        else:
            option1 = chargeOptimale(tab, ..., ...)
            option2 = [tab[i]] + chargeOptimale(tab, ..., ...)
            if prixListe(option1) > prixListe(option2):
                return ...
            else:
                return ...
```