TP 07

Fonctions récursives

Savoirs et compétences :

- ❒ *AA.C9 : Choisir un type de données en fonction d'un problème à résoudre*
- ❒ *AA.S11 : Manipulation de quelques structures de données.*
- ❒ *AA.S12 : Fichiers*

Activité 1 – Algorithme du rendu de monnaie

La société Sharp commercialise des caisses automatiques utilisées par exemple dans des boulangeries. Le client glisse directement les billets ou les pièces dans la machine qui se charge de rendre automatiquement la monnaie.

Objectif Afin de satisfaire les clients, on cherche à déterminer un algorithme qui va permettre de rendre le moins de monnaie possible.

La machine dispose de billets de 20€, 10€ et 5€ ainsi que des pièces de 2€, 1€, 50, 20, 10, 5, 2 et 1 centimes.

On se propose donc de concevoir un algorithme qui demande à l'utilisateur du programme la somme totale à payer ainsi que le montant donné par l'acheteur. L'algorithme doit alors déterminer quels sont les billets et les pièces à rendre par le vendeur.

Pour ne pas faire d'erreurs d'approximation, tous les calculs seront faits en centimes.

Le contenu de la caisse automatique et le contenu du porte-monnaie du client seront modélisés par un tableau ayant la forme suivante :

```
caisse = ↙
    [[2000,10],[1000,10],[500,10],[200,10],[100,10],[50,10],[20,10],[10,10],[5,10],[2,10],[1,10]]↙
```

Cela signifie que la caisse contient 10 billets de 20€, 10 billets de 10€...

Dans la prochaine question, on fait l'hypothèse que la caisse contient suffisamment de billets et de pièces de chaque valeur.

Question 1 *Ecrire une fonction* `rendre_monnaie(caisse:list, cout:float, somme_client:float)->list` *prenant en arguments deux flottants* `cout` *et* `somme_client` *représentant le coût d'un produit et la somme donnée par le client en € ainsi que le contenu de la caisse. Cette fonction renvoie une liste ayant la même structure que la caisse, mais contenant le nombre de chaque billet (ou pièce) à rendre au client.*

Pour un coût de 15,99€, et une somme donnée par le client de 17,50€, la machine devra rendre les pièves ou les billets suivants :

- 0 : billet de 20 euros ;
- 0 : billet de 10 euros ;
- 0 : billet de 5 euros ;
- 0 : pièce de 2 euros ;
- 1 : pièce de 1 euros ;
- 1 : pièce de 50 centimes ;
- 0 : pièce de 20 centimes ;
- 0 : pièce de 10 centimes ;
- 0 : pièce de 5 centimes ;
- 0 : pièce de 2 centimes ;
- 1 : pièce de 1 centimes.

`rendre_monnaie` devra donc renvoyer `[[2000,0],[1000,0],[500,0],[200,0],[100,1],[50,1],[20,0],[10,0],[5`

Question 2 *Ecrire une fonction* `rendre_monnaie_v2(caisse:list, cout:float, somme_client:float)->list` *ayant le même objectif que la précédente. En plus de la précédente, cette fonction devra mettre à jour la caisse. Elle devra prendre en compte que la caisse peut manquer de tels ou tels billets. Elle renverra une liste vide s'il n'est pas possible*

de rendre la monnaie.

On suppose que la caisse est maintenant la suivante.

```
caisse = [[5000,10],[2000,10],[1000,10],[800,10],[100,10]]
```

Le client achète un article de 36€ avec un billet de 50€.

Question 3 *Que retourne la fonction* `rendre_monnaie` *? Est-ce le rendu optimal ?*

Question 4 *Conclure « qualitativement ».*

Activité 2 – Problème du sac à dos

Un promeneur souhaite transporter dans son sac à dos le fruit de sa cueillette. La cueillette est belle, mais trop lourde pour être entièrement transportée dans le sac à dos. Des choix doivent être faits. Il faut que la masse totale des fruits choisis ne dépasse pas la capacité maximale du sac à dos.

Les fruits cueillis ont des valeurs différentes et le promeneur souhaite que son chargement soit de la plus grande valeur possible.

Fruits cueillis	Prix au kilo	Quantité ramassée
framboises	24 €/kg	1 kg
myrtilles	16 €/kg	3 kg
fraises	6 €/kg	5 kg
mures	3 €/kg	2 kg

TABLE 1

La capacité du sac à dos n'est que de 5 kg. On suppose que la masse d'un unique fruit est négligeable par rapport à la masse totale du sac en charge.

Présentation de l'algorithme et implémentation

Le principe pour optimiser le chargement est de commencer par mettre dans le sac la quantité maximale de fruits les plus chers par unité de masse. S'il y a encore de la place dans le sac, on continue avec les fruits les plus chers par unité de masse parmi ceux restants, ... et ainsi de suite jusqu'à ce que le sac soit plein. Il est possible qu'on ne puisse prendre qu'une fraction de la quantité de fruits disponible, si la capacité maximale du sac est atteinte. L'algorithme sera donc constitué d'une structure itérative, incluant une structure alternative.

Dans **Algorithmes_gloutons.py**, nous avons ébauché l'algorithme de résolution du problème et implémenté `cueillette` par la liste des fruits **triée** par prix au kilo décroissant.
Un fruit est représenté par une liste `fruit=[prix au kilo,nom,quantité ramassée]`

cueillette = [[24,"framboises",1], [16,"myrtilles",3], [6,"fraises",5], [3,"mures",2]]

Question 1 *Implémenter la fonction* `sac_a_dos(L:list, capacite:int)` *qui prend en argument une liste de listes* `L` *modélisant la cueillette et la capacité du sac à dos, et appliquant la méthode décrite précédemment. Cette fonction renvoie :*

- *une liste de listes des fruits contenus dans le sac à dos avec leur masse correspondante de façon à avoir le chargement de valeur maximale ;*

- *la valeur du chargement.*

La liste `cueillette` ne doit pas être modifiée. Vous pourrez si nécessaire utiliser une copie de la liste `cueillette` que vous pourrez modifier. La copie de liste de listes se fait avec les instructions :

```
import copy # au début de votre script
L=copy.deepcopy(votreListe) # quand cela est nécessaire
```

Question 2 *Exécuter votre programme et vérifier le résultat.*

Une variante de ce problème ne trouve pas de solution optimale par la méthode gloutonne. Nous l'étudions ci-après.

Version non fractionnaire du problème du sac à dos

On suppose maintenant que les éléments à transporter ne sont pas fractionnables. Les fruits parmi lesquels choisir sont présentés dans le tableau 2.

Fruits	Prix au kilo	Masse d'un fruit	Quantité disponible
melon de cavaillon	3 €/kg	1 kg	1
melon jaune	2.5 €/kg	2 kg	1
pastèque	2 €/kg	3 kg	1

TABLE 2

Cet ensemble de fruits est modélisé par :

- un fruit non fractionnable est représenté par une liste `[prix au kilo,nom,masse,quantité disponible]` ;
- une liste de listes `fruitsDisponibles = [[3,"melon de cavaillon",1,1], [2.5,"melon jaune",2,1, 2,"pastèque",3,1]]`.

L'objectif est toujours de placer dans le sac à dos le chargement de valeur maximale, de masse totale inférieure à 5kg. Par contre, les éléments n'étant fractionnables, il est possible que les choix successifs mène à un chargement qui ne remplit pas complètement le sac à dos.

On se propose de tester la méthode gloutonne pour cette nouvelle formulation.

Question 3 *En adaptant la fonction* `sac_a_dos(L , capacite)` *que vous renommerez* `sac_a_dos_V2(L , capacite)`*, adapter la fonction de sorte qu'elle applique la méthode gloutonne à la version non fractionnaire du problème. Exécuter la fonction en utilisant la liste* `fruitsDisponibles`.

Question 4 *Le résultat obtenu est-il optimal ? Comparer avec la solution S = [[2.5,"melon jaune",2,1], [2,"pastèque",3,1]]. Quelle est la solution dont la valeur est maximale ?*