TP 06

Algorithme glouton

Savoirs et compétences :

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

Proposition de corrigé

Activité 1 – Problème glouton du rendu de monnaie

Question 1 *Écrire 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 la liste des billets à rendre par le client.*

```
def rendre_monnaie(caisse:list, cout:float, somme_client:float) -> list :
    """
    Algorithme glouton du rendu de monnaie

    Parameters
    ----------
    caisse : list
        liste de liste contenant les valeurs des billets et leur quantité.
        On utilise une liste (plutôt qu'un dictionnaire) car les valeurs
        doivent être triées
    cout : float
        coÃ»t àpayer en euros
    somme_client : float
        somme donnée par le client (en euros)

    Returns
    -------
    list
        DESCRIPTION.

    """
    rendu = []
    somme = int(100*somme_client)
    cout = int(100*cout)
    delta = somme- cout
    while delta > 0 :
        for valeur in caisse :
            while valeur[0]<=delta :
                delta = delta - valeur[0]
                rendu.append(valeur[0])
    return rendu
```

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

```
def rendre_monnaie_v2(caisse:list, cout:float, somme_client:float) -> list :
    """
    Algorithme glouton du rendu de monnaie avec MAJ de la caisse.
    """
    rendu = []
    somme = int(100*somme_client)
    cout = int(100*cout)
    delta = somme- cout
    while delta > 0 :
        for i in range(len(caisse)):
            while caisse[i][0]<=delta and caisse[i][1]>0:
                delta = delta - caisse[i][0]
                caisse[i][1]=caisse[i][1]-1
                rendu.append(caisse[i][0])
        print(delta)
        if delta>0 :
            return []
    return rendu
```

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

La solution optimale serait [2000,500,100] et l'algorithme `rendre_monnaie_v2(caisse, 34, 50)`, l'algorithme renverra `[1000, 100, 100, 100, 100, 100, 100]`.

Question 4 *Conclure « qualitativement ».*

L'algorithme glouton renvoie un optimum local et non pas un optimum global comme le montre l'exemple précédent.

Activité 2 –

Question 5 *Définir la fonction* `def valeur(objet:list) -> float` *qui renvoie la valeur d'un objet.*

```
def valeur(obj):
    return obj[1]
```

Question 6 *Définir la fonction* `def poids(objet:list) -> float` *qui renvoie le poids d'un objet.*

```
def poids(obj):
    return obj[2]
```

Question 7 *Définir la fonction* `def rapport(objet:list) -> float` *qui renvoie le rapport valeur/poids d'un objet.*

```
def rapport(obj):
    return obj[1] / obj[2]
```

Question 8 *Implémenter la fonction* `glouton` *qui renvoie (dans l'ordre)* `reponse` *et* `valeur`.

```
def glouton(liste, poids_max, choix):
    copie = sorted(liste, key=choix, reverse=True)
    reponse = []
    valeur = 0
    poids = 0
    i = 0
    while i < len(liste) and poids <= poids_max:
        nom, val, pds = copie[i]
        if poids + pds <= poids_max:
            reponse.append(nom)
```

```
            poids = poids + pds
            valeur = valeur + val
        i = i + 1
    return reponse, valeur
```

Question 9 *Exécuter la fonction* `glouton` *pour les différents types de choix. On observant la nature des différents objets, le choix optimal est-il parmi les choix proposés ?*

```
>>>glouton(objets,15,valeur)
(['objet 1', 'objet 4'], 131)

>>> glouton(objets,15,poids)
(['objet 1', 'objet 4'], 131)

>>> glouton(objets,15,rapport)
(['objet 2', 'objet 6', 'objet 4'], 117)
```

Question 10 *Estimer le nombre de tours de boucle nécessaire pour exécuter l'algorithme* `glouton` *en fonction du nombre d'objets* `n`.

Le nombre de tour de boucle dépend des valeurs présentent mais dans le pire des cas il correspond au nombre de terme. On est donc dans le cas d'une complexité linéaire

Question 11 *Estimer (grossièrerement) le nombre de combinaisons possibles parmi la liste d'objets qui permettrait de remplir le sac à dos. Conclure sur l'intérêt d'un algorithme glouton.*

Le nombre de combinaison est en n! L'algorithme glouton est donc vraiment intéressant.

Question 12 *Proposer une variante récursive de l'algorithme glouton.*

```
def glouton_rec(liste, poids_max, reponse, valeur, poids, i):
    if i < len(liste) and poids <= poids_max:
        nom, val, pds = liste[i]
        if poids + pds <= poids_max:
            reponse.append(nom)
            poids = poids + pds
            valeur = valeur + val
        return glouton(liste, poids_max, reponse, valeur, poids, i+1)
    else:
        return reponse, valeur
```