P15

ALGORITHMES GLOUTONS

Exercice 1 **Problème du sac à dos**

On dispose d'un sac à dos, supportant au maximum 30 kg, et d'objets de différentes valeurs :

Objet	0	1	2	3
Valeur (euros)	7	80	4	3
Poids (kg)	13	20	11	10
Valeur/Poids (euros/kg)	0.54	0.4	0.36	0.3

Il s'agit d'analyser le programme suivant, **sans l'exécuter**.

```
objets = [[7, 13], [80, 20], [4, 11], [3, 10]]
poids_max = 30

sac = []
poids = 0
valeur = 0
for i in range(len(objets)):
    objet = objets[i]
    poids_objet = objet[1]
    if poids + poids_objet <= poids_max:
        sac.append(objet)
        poids = poids + poids_objet
        valeur = valeur + objet[0]
print('sac =', sac, '; poids =', poids, '; valeur =', valeur)
```

1. Déterminer ce que ce programme affiche. Faire un tableau des valeurs successives.

2. On souhaite maximiser la valeur totale présente dans le cas à dos. Expliquer pourquoi l'ordre des objets est pertinent.

3. La solution trouvée est-elle optimale ?

Exercice 2 **Rendu de monnaie**

La personne en charge de la caisse d'un magasin doit rendre régulièrement de la monnaie aux clients qui payent en espèces.

On va écrire un programme permettant de préciser quelles pièces elle doit donner au client selon la somme à rendre.

On suppose que :

- ▷ la caisse dispose de pièces (ou billets) de valeurs entières $v_1 < v_2 < \cdots < v_n$ avec $v_1 = 1$ euro ;
- ▷ la quantité de monnaie disponible est très grande, elle ne sera pas à court ;
- ▷ la somme à rendre au client est une valeur entière.

Pour résoudre ce problème, on va utiliser un **algorithme glouton** : *cela consiste à construire une solution globale en avançant pas à pas selon un principe qui est localement le plus optimal.*

↪ *ici, cela veut dire qu'on commence par rembourser avec la pièce/le billet de plus grande valeur possible*

puis de répéter l'opération jusqu'à ce qu'il ne reste plus rien à rembourser

Par exemple : S'il y a 74 euros à rendre, avec des pièces de $v_1 = 1$ euro, $v_2 = 2$ euros et des billets de $v_3 = 5$ euros, $v_4 = 10$ euros et $v_5 = 20$ euros.

- ▷ parmi les pièces et les billets disponibles, la plus grande valeur inférieure à 74 est 20 : *on rend 1 billet de 20 euros*
- ▷ il reste 54 euros : *on rend donc encore 20 euros*
- ▷ il reste 34 euros : *on rend donc encore 20 euros*
- ▷ il reste 14 euros : *on rend donc 10 euros*
- ▷ il reste 4 euros : *on rend donc 2 euros*
- ▷ il reste 2 euros : *on rend donc 2 euros*
- ▷ il ne reste plus rien à rendre : *l'algorithme s'achève*

On va donc rendre : 3 billets de 20 euros, 1 billets de 10 euros, et 2 pièces de 2 euros.

À vous de jouer !

On dispose d'une liste `systeme` constituée des valeurs des différentes pièces (ou billets, on parlera de pièces en général) disponibles pour le rendu de monnaie, écrites dans l'ordre décroissant. Dans l'exemple précédent, `systeme = [20, 10, 5, 2, 1]`.

1. Pourquoi les pièces sont-elles dans l'ordre décroissant ?
2. Compléter le programme permettant de déterminer les pièces à rendre.

```
systeme = [20, 10, 5, 2, 1]

somme_à_rendre = 74 # un exemple, on pourra changer
rendu = [] # liste des pièces à rendre
i = 0

while ..................... :

    piece = ............... # valeur de la pièce numéro i

    if ..................... :

        rendu.append(...........)

        somme_à_rendre = ..............

    else:

        ...........

print(rendu)
```

3. **A propos de la « qualité » de la solution proposée**

   Un rendu de monnaie est considéré comme « optimal » lorsque, pour une somme donnée, il permet de rendre le moins possible de pièces et de billets.

   - Quel rendu de monnaie un algorithme glouton effectue-t-il si l'on dispose exclusivement de pièces de 1 euro, 3 euros et 4 euros, et qu'on doit rendre la somme de 10 euros ?

   - Quel rendu de monnaie un algorithme glouton effectue-t-il si l'on dispose exclusivement de pièces de 3 euros et 4 euros, et qu'on doit rendre la somme de 10 euros ?

Exercice 3 Planning d'un seminaire

Vous assistez à un séminaire présentant un certains nombre de conférences qui vous intéressent. Vous souhaitez voir **le maximum de conférences**. Il s'agit donc, en fonction des horaires des conférences, d'établir votre planning de la journée.

Exemple 1 : on a quatre conférences C_1, C_2, C_3 et C_4 et leurs créneaux sont indiquées sous la forme d'un intervalle.

$$C_1 : [3,4[, \quad C_2 : [0,1[, \quad C_3 : [2,3[, \quad C_4 : [1,2[.$$

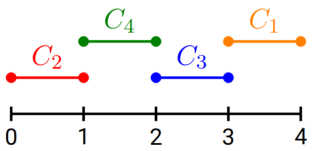

FIGURE 1 – Situation 1

Ici c'est très simple, vous allez pouvoir assister à toutes les conférences dans l'ordre suivant : C_2, C_4, C_3, C_1.

Exemple 2 : $C_1 : [2,4[, \quad C_2 : [0,1[, \quad C_3 : [1,3[, \quad C_4 : [0,2[.$

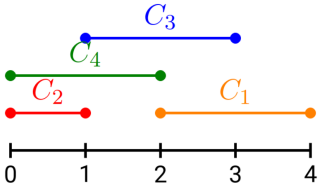

FIGURE 2 – Situation 2

Ici, plusieurs conférences se superposent, vous n'allez pas pouvoir toutes les voir. Il y a plusieurs solutions :

$$C_2 \text{ puis } C_1 ; \quad C_2 \text{ puis } C_3 ; \quad C_4 \text{ puis } C_1 ; \quad C_3.$$

Se posent alors plusieurs questions :

- Comment choisir une de ces solutions ?
  Idée 1 : on classe les conférences par horaire de début croissante, on voit la première puis on regarde quelle conférence on peut mettre à la suite. Ceci élimine C_3 et $[C_2, C_1]$.

- Problème : un algorithme doit renvoyer une réponse unique et bien déterminée. Or ici, il nous reste deux solutions : $[C_2, C_3]$ et $[C_4, C_1]$.

  Remarque : toutes les conférences vous intéressent autant, quelles que soient leurs durées. La solution $[C_4, C_1]$ n'est pas à privilégier de ce point de vue. On cherche un algorithme permettant de voir **le maximum de conférences** (en nombre, pas en durée).

- Comment choisir la première conférence entre C_2 et C_4 ?
  Idée 2 : on complète en regardant les horaires de fin : $fin_2 < fin_4$. Laquelle choisissez-vous ?

Exemple 3 : $C_1 : [0, 3[, \quad C_2 : [1, 2[, \quad C_3 : [2, 3[$.

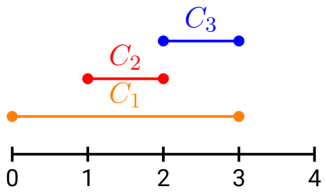

FIGURE 3 – Situation 3

Appliquons les principes précédents :

- On commence par C_1. Il n'y a pas de conférence possible après. Cette solution n'est pas satisfaisante car on préfère voir C_2 et C_3.
- Idée 3 : on procède uniquement par heure de fin croissante.

$$fin_2 \leqslant fin_1 \leqslant fin_3.$$

  On commence donc par C_2 puis on aura C_3 (C_1 est éliminée car incompatible avec C_2). C'est cette idée que nous allons programmer. On peut montrer qu'elle fournit une solution optimale.

Algorithme :

- Classer les conférences par heure de fin croissante.
- Choisir la première conférence de la liste, l'ajouter au planning.
- S'intéresser à la deuxième conférence de la liste. Si elle est compatible on l'ajoute au planning. Sinon, on passe à la conférence suivante.
- Recommencer jusqu'à avoir passé en revue toutes les conférences.

1. Appliquer l'algorithme à la main sur cet exemple :

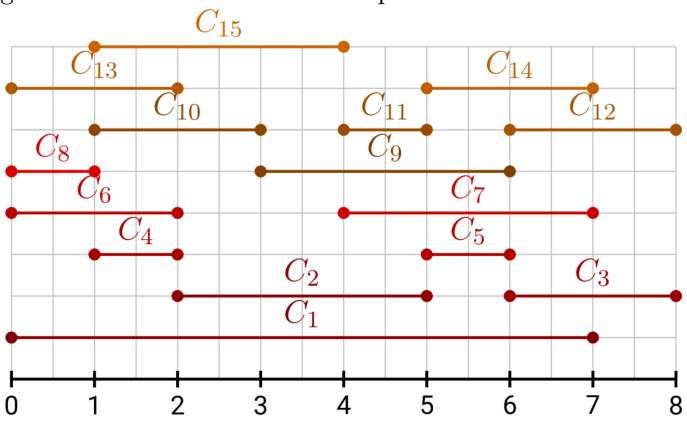

FIGURE 4 – Une situation plus complexe.

2. On dispose au début d'une liste de conférences.

```
conferences = [[0,7,'C1'], [2,5,'C2'], [6,8,'C3'], [1,2,'C4'],
[5,6,'C5'], [0,2,'C6'], [4,7,'C7'], [0,1,'C8'], [3,6,'C9'],
[1,3,'C10'],[4,5,'C11'], [6,8,'C12'], [0,2,'C13'], [5,7,'C14'],
[1,4,'C15']]
```

Comment obtenir l'heure de début de $C3$?

Comment obtenir l'heure de fin de $C14$?

3.

```
# Tri des conferences par date de fin croissante.
conferences_triees = sorted(conferences, key = lambda L:L[1] )
print(conferences_triees)

n = ..................... # nombre de conferences

planning = []

i = 0 # indice de la première conference consideree

planning.append(conference_triees[...][...])

for j in range(1,n):
    # heure de fin de la dernière conférence vue
    heure = conferences_triees[...][...]
    # heure de début de la conférence numéro j
    debut_j = conferences_triees[...][...]

    if .............. : # si on peut aller voir la conférence j

        planning.append(conferences_triees[...][...])

        i = ... # numéro de la dernière conférence vue



print(planning)
```

Exercice 4 **Déplacement sur une grille**

On dispose d'une grille carrée (liste de listes) comme ci-dessous. On souhaite, en partant du premier élément (en haut à gauche), arriver au dernier élément (en bas à droite) en cherchant à maximiser la somme des nombres rencontrés. À chaque étape, deux déplacements sont autorisés : vers la droite ou vers le bas (sans sortir de la grille).

3	4	1	5	1	2
2	2	4	6	1	2
3	5	6	2	2	4
6	5	3	6	1	1
4	5	1	2	2	3
4	6	6	6	3	4

Écrire un programme reposant sur un algorithme glouton pour répondre à ce problème. On donnera le chemin sous la forme d'une liste de 'bas' et 'droite' et on affichera la valeur totale.

```
grille = [[3,4,1,5,1,2],
          [2,2,4,6,1,2],
          [3,5,6,2,2,4],
          [6,5,3,6,1,1],
          [4,5,1,2,2,3],
          [4,6,6,6,3,4]]
```

La solution trouvée est-elle globalement optimale ?