

TP11

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	3	4	4
Poids (kg)	13	7	11	12
Valeur/poids	0.54	0.43	0.36	0.33

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

```
objets = [[7,13], [3,7], [4,11], [4,12]]
n = len(objets)

poids_max = 30
sac = []
poids = 0
valeur = 0

for i in range(n):
    poids_objet = objets[i][1]
    if poids + poids_objet <= poids_max:
        sac.append(objets[i])
        poids = poids + poids_objet
        valeur = valeur + objets[i][0]

print('sac =', sac, 'poids =', poids, 'valeur =', valeur)
```

1. Faire un tableau de valeurs successives.

On a ici $n = 4$.

i	poids_objet	sac	poids	valeur
		[]	0	0
0	13	[[7,13]]	13	7
1	7	[[7,13],[3,7]]	20	10
2	11	[[7,13],[3,7]]	20	10
3	12	[[7,13],[3,7]]	20	10

2. Que fait ce programme ?

La programme remplit le sac à dos, avec un poids maximal de 30kg. Les objets sont placés dans l'ordre du tableau.

3. L'ordre des objets a-t-elle une importance ?

Oui.

4. La solution obtenue est-elle optimale ?

Non, pour 30kg, on aurait pu mettre les trois derniers objets pour une valeur totale de 11 euros (contre 10 ici).

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 ici écrire un programme pour l'aider et lui dire précisément 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 $v_1 < v_2 < \dots < v_n$ 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 notée s ;

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 la plus grande possible.

Exemple : $s = 27$ euros à rendre.

Monnaie disponible : $v_1 = 1, v_2 = 2, v_3 = 5, v_4 = 10, v_5 = 20$.

- la plus grande valeur inférieure à 27 est 20 : on rend 1 billet de 20 euros ;
- il reste 7 euros : on rend donc 5 euros ;
- il reste 2 euros : on rend donc 2 euros.

On va donc rendre : 1 billet de 20 euros, 1 billet de 5 euros et 1 pièce de 2 euros.

1. Compléter le programme suivant :

```
s = 27 # somme à rendre
monnaie = [1,2,5,10,20] # monnaie par ordre croissant

rendu = [] # liste des pièces à rendre

# D'où démarre t-on ?
i = len(monnaie)-1 # indice dans la liste monnaie

while s > 0:
    # on cherche la plus grande pièce <= s
    piece = monnaie[i]
    if piece <= s :
        rendu.append(piece)
        s = s - piece
    else :
        i = i - 1

print('rendu =', rendu)
```

2. Vérifier que l'on obtient bien le résultat attendu, puis tester avec d'autres valeurs de s .
3. Changeons maintenant la liste des pièces pour $\text{monnaie} = [1, 3, 4]$. Déterminer quelles pièces rendre pour une somme de $s = 10$ euros.
 - 10 euros : on rend 4 euros ;
 - il reste 6 euros : on rend donc 4 euros ;
 - il reste 2 euros : on rend donc 1 euro.
 - il reste 1 euro : on rend donc 1 euro.

Cette solution est-elle optimale ?

Non, il y avait mieux, avec moins de pièces : $4 + 3 + 3$.

4. Que se passe-t-il si on enlève la pièce de 1 euro ?
On ne pourra pas toujours rendre la monnaie exacte.

Exercice 3 **Planning d'un séminaire**

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.

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

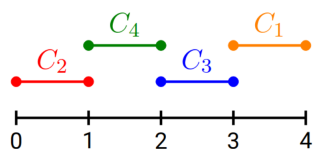


FIGURE 1 – Situation 1

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

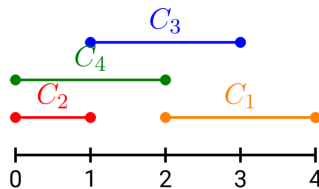


FIGURE 2 – Situation 2

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 ?
On choisit la conférence se terminant en premier, car cela permettra potentiellement de voir plus de choses.

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

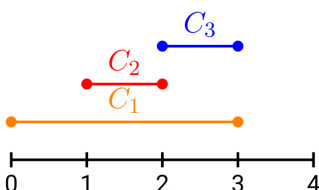


FIGURE 3 – Situation 3

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.

$$fin_2 \leq fin_1 \leq fin_3.$$

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 :

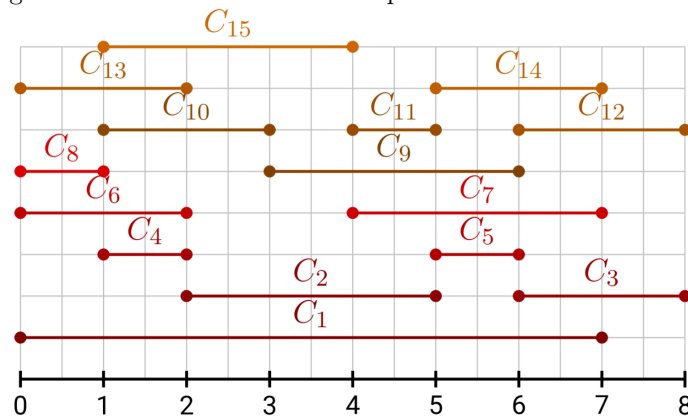


FIGURE 4 – Une situation plus complexe.

- Classer les conférences par heure de fin croissante : $C_8, C_4, C_6, C_{13}, C_{10}, C_{15}, C_2, C_{11}, C_5, C_9, C_1, C_7, C_{14}, C_3, C_{12}$.
- conférences_vues = $[C_8]$
- C_4 est compatible, donc conférences_vues = $[C_8, C_4]$
- La suivante est C_2 : conférences_vues = $[C_8, C_4, C_2]$ (remarquons que suivant l'ordre choisi, cela aurait pu être C_{11}).
- La suivante est C_5 : conférences_vues = $[C_8, C_4, C_2, C_5]$.
- La suivante est C_3 : conférences_vues = $[C_8, C_4, C_2, C_5, C_3]$ (suivant l'ordre choisi, cela aurait pu aussi être C_{12}).

En tout, on voit 5 conférences, $[C_8, C_4, C_2, C_5, C_3]$.

2. Compléter le programme suivant :

```
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']]

# Trier les conférences par date de fin croissante.
conferences_triees = sorted(conferences, key=lambda liste: liste[1])
print(conferences_triees)

n = len(conferences) # nombre de conférences

planning = []

i = 0 # indice conférence dans liste triée
planning.append(conferences_triees[i])

for j in range(1,n):
    fin = conferences_triees[i][1]
    debut_j = conferences_triees[j][0]
    if debut_j >= fin : # si conf_j est compatible
        i = j
        planning.append(conferences_triees[i])
    # sinon, on passe à la conférence suivante

print('planning :', planning)
```