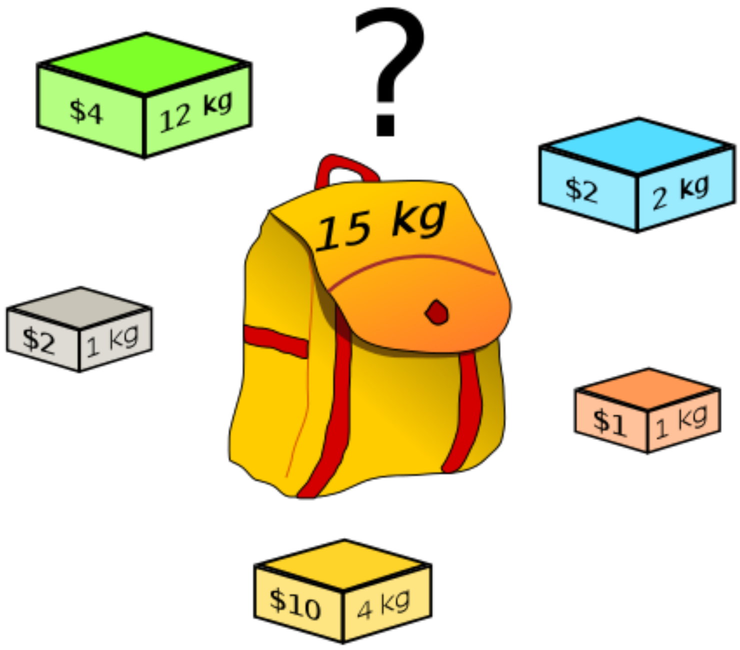
Algorithmes Gloutons (Greedy)



Sources :

[1] Cormen, Leiserson, Rivest and Stein : Introduction to algorithmns, third edition. The MIT press, 2009.

[2] Eduscol : https://eduscol.education.fr/2068/programmes-et-ressources-en-numerique-et-sciences-informatiques-voie-g

Les algorithmes gloutons

1. Introduction et généralité

Optimiser un problème, c'est déterminer les conditions dans lesquelles ce problème présente une caractéristique spécifique. Par exemple, déterminer le minimum ou le maximum d'une fonction est un problème d'optimisation. On peut également citer la répartition optimale de tâches suivant des critères précis, ***le problème du rendu de monnaie, le problème du sac à dos***, la recherche d'un plus court chemin dans un graphe, le problème du voyageur de commerce. De nombreuses techniques informatiques sont susceptibles d'apporter une solution exacte ou approchée à ces problèmes. Certaines de ces techniques, comme l'énumération exhaustive de toutes les solutions, ont un coût machine qui les rend souvent peu pertinentes au regard de contraintes extérieures imposées (temps de réponse de la solution imposé, moyens machines limités).

Les techniques de programmation dynamique ou d'optimisation linéaire, certains algorithmes numériques peuvent apporter une solution. Les ***algorithmes gloutons*** constituent une alternative dont le résultat n'est pas toujours optimal. Plus précisément, ces algorithmes déterminent une solution optimale en effectuant successivement des choix locaux, jamais remis en cause. Au cours de la construction de la solution, l'algorithme résout une partie du problème puis se focalise ensuite sur le sous-problème restant à résoudre. Une différence essentielle avec la programmation dynamique est que celle-ci peut remettre en cause des solutions déjà établies. Au lieu de se focaliser sur un seul sous-problème, elle explore les solutions de tous les sous-problèmes pour les combiner finalement de manière optimale.

Le principal avantage des algorithmes gloutons est leur facilité de mise en œuvre. En outre, dans certaines situations dites ***canoniques***, il arrive qu'ils renvoient non pas un optimum mais l'optimum d'un problème. Nous présentons de telles situations dans la suite de cet exposé, en montrant les avantages mais aussi les limites de la technique.

2. Rendu de monnaie

Un achat dit en espèces se traduit par un échange de pièces et de billets. Dans la suite, les pièces désignent indifféremment les véritables pièces que les billets. Supposons qu'un achat induise un rendu de 49 euros. Quelles pièces peuvent être rendues? La réponse, bien qu'évidente, n'est pas unique. Quatre pièces de 10 euros, 1 pièce de 5 euros et deux pièces de 2 euros conviennent. Mais quarante-neuf pièces de 1 euros conviennent également ! ***Si la question est de rendre la monnaie avec un minimum de pièces***, le problème change de nature. Mais la réponse reste simple : c'est la première solution proposée.

Toutefois, comment parvient-on à un tel résultat ? Quels choix ont été faits qui optimisent le nombre de pièces rendues ? C'est le problème du rendu de monnaie dont la solution dépend du système de monnaie utilisé.

Dans le système monétaire français, les pièces prennent les valeurs 1, 2, 5, 10, 20, 50, 100 euros (on oublie le billet de 500 euros). Rendre 49 euros avec un minimum de pièces est un problème d'optimisation. En pratique, sans s'en rendre compte généralement, tout individu met en œuvre un algorithme glouton. Il choisit d'abord la plus grandeur valeur de monnaie, parmi 1, 2, 5, 10, contenue dans 49 euros. En l'occurrence, quatre fois un pièce de 10 euros. La somme de 9 euros restant à rendre, il choisit une pièce de 5 euros, puis deux pièces de 2 euros. Cette stratégie gagnante pour la somme de 49 euros l'est-elle pour n'importe quelle somme à rendre ? On peut montrer que la réponse est positive pour le système monétaire français. Pour cette raison, un tel système de monnaie est ***qualifié de canonique***.

D'autres systèmes ne sont pas canoniques. L'algorithme glouton ne répond alors pas de manière optimale. Par exemple, avec le système {1, 3, 6, 12, 24, 30}, l'algorithme glouton répond en proposant le rendu 49 = 30 + 12 + 6 + 1, soit 4 pièces alors que la solution optimale est 49 = 2 × 24 + 1, soit 3 pièces. La réponse à cette difficulté passe par la programmation dynamique.

2.1. Un algorithme glouton

Considérons un ensemble de v pièces de monnaie de valeurs : $v_1 < v_2 < v_3 \cdots < v_n$ avec $v_1 = 1$. On suppose que ce système est canonique. On peut noter le système de pièces : $\Sigma_n = \left\{v_1, v_2 \ldots v_n\right\}$.

Désignons par S une somme à rendre avec le minimum de pièces de Σ_n.

L'algorithme glouton sélectionne la plus grande valeur v_n et la compare à S.

⇒ Si $S < v_n$, la pièce de valeur v_n ne peut pas être utilisée. On reprend l'algorithme avec le système de pièces Σ_{n-1}.

⇒ Si $S \geq v_n$, la pièce v_n peut être utilisée une première fois. Ce qui fait une première pièce à comptabiliser, de valeur v_n, la somme restant à rendre étant alors $S - v_n$. L'algorithme continue avec la même système de pièces Σ_n et cette nouvelle somme à rendre $S - v_n$.

L'algorithme est ainsi répété jusqu'à obtenir une somme à rendre nulle.

Il s'agit effectivement d'un ***algorithme glouton***, la plus grande valeur de pièce étant systématiquement choisie si sa valeur est inférieure à la somme à rendre. Ce choix ne garantit en rien l'optimalité globale de la solution. Le choix fait est considéré comme pertinent et permet d'avancer plus avant dans le calcul. Toutefois, comme nous l'écrivions plus haut, si le système monétaire est canonique, la solution est optimale

2.2. Code python

✔ On définit le système de pièces à l'aide d'un tableau de valeurs des pièces ***classées par valeurs croissantes.***

✔ Pour stocker les pièces à rendre, une liste Python initialement vide peut être utilisée.

✔ La première pièce à rendre est potentiellement la dernière pièce du tableau `systeme_monnaie`. Une variable `i` de type entier est initialisée avec l'indice du dernier élément de ce tableau.

✔ Chaque fois qu'une pièce de Σ_n n'est plus utilisable, la valeur de `i` sera diminuée de 1. Le programme s'arrête quand `i` atteint la valeur 0. Ce qui mène à l'écriture d'une boucle conditionnelle pour remplir la liste des pièces choisies. La somme à rendre est initialement stockée dans la variable `somme_a_rendre`.

✔ Pour finir, le code peut être encapsulé dans une fonction qui reçoit deux arguments – la somme à rendre et le système de monnaie – et qui renvoie la liste des pièces choisies par l'algorithme glouton.

```python
def pieces_a_rendre(somme_a_rendre, systeme_monnaie):
# liste des pièces à rendre
    lst_pieces = []
# indice de la première pièce comparer à la somme à rendre
    i = len(systeme_monnaie) - 1
    while somme_a_rendre > 0:
        valeur = systeme_monnaie[i]
        if somme_a_rendre < valeur:
            i = i-1
        else:
            lst_pieces.append(valeur)
            somme_a_rendre = somme_a_rendre - valeur
    return lst_pieces


# valeurs des pièces
systeme_monnaie = [1,2,5,10,20,50,100]

# somme à rendre
somme_a_rendre = 87

# appel de la fonction
liste_piece_rendu= pieces_a_rendre(somme_a_rendre, systeme_monnaie)
print(liste_piece_rendu)
```

```
Système de monnaie:
[1, 2, 5, 10, 20, 50, 100]
Somme à rendre:
87
Pièces à rendre:
[50, 20, 10, 5, 2]
```

3. Un problème d'organisation

Des conférenciers sont invités à présenter leurs exposés dans une salle. Mais leurs disponibilités ne leur permettent d'intervenir qu'à des horaires bien définis. ***Le problème est de construire un planning d'occupation de la salle avec le plus grand nombre de conférenciers***.

Désignons par n, entier naturel non nul, le nombre de conférenciers. Chacun d'eux, identifié par une lettre C_i, où i est un entier compris entre 0 et $n-1$, est associé à un intervalle temporel $\left[d_i, f_i\right[$ [où d_i et f_i désignent respectivement l'heure de début et l'heure de fin de l'intervention. Afin de dégager une tactique de résolution du problème, commençons par analyser plusieurs situations.

3.1. Première analyse

Situation 1

Quatre conférenciers peuvent intervenir aux intervalles temporels suivants, illustrés sur la figure1.

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

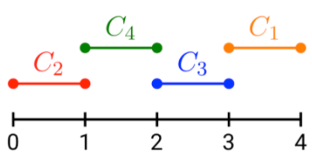


FIGURE 1 – Situation 1

Une telle situation est simple puisque tous les conférenciers peuvent intervenir sur des créneaux horaires disjoints. Le planning est donc défini par la suite $\left[C_2,C_4,C_3,C_1\right]$ de conférenciers. L'algorithme menant à ce résultat choisit les conférenciers par ordre croissant des heures de début ou de fin des conférences après s'être assuré que les intervalles sont disjoints.

Situation 2

On considère à nouveau quatre conférenciers dont les créneaux horaires ne sont plus toujours disjoints (figure 2).

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

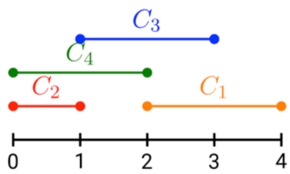


FIGURE 2 – Situation 2

Dans cette situation, les intervalles ne sont plus disjoints. Nous dirons que ***ces intervalles ne sont pas compatibles***. Des choix doivent être faits et certains conférenciers peuvent ne pas être retenus pour construire un planning. Plusieurs solutions peuvent être construites :

$$\left[C_2,C_1\right] \text{ ou } \left[C_2,C_3\right] \text{ ou } \left[C_4,C_1\right] \text{ ou } \left[C_3\right]$$

Seules les trois premières solutions sont retenues puisque la dernière ne maximise pas le nombre de conférenciers choisis. Mais laquelle de ces solutions retenir ? Pour y répondre, il convient de se demander comment un algorithme pourrait aboutir à la construction de ces solutions. Une idée serait de nouveau de classer par ordre croissant les heures de début des intervalles compatibles.

En procédant de la sorte :

$\Rightarrow$ C_1 et C_2 sont compatibles avec $d_2 < d_1$ ce qui mène au planning $\left[C_2,C_1\right]$.

$\Rightarrow$ C_2 et C_3 sont compatibles avec $d_2 < d_3$ ce qui mène à $\left[C_2,C_3\right]$.

$\Rightarrow$ C_4 et C_1 sont également compatibles avec $d_4 < d_1$ ce qui mène à $\left[C_4,C_1\right]$.

Une autre idée serait de construire les plannings en classant par ordre croissant les heures de fin des intervalles compatibles. En procédant de la sorte, on retrouve les trois propositions de plannings précédentes. Il semble donc que ces deux idées mènent à des résultats identiques. En outre, elles n'ont pas permis d'éliminer des solutions afin de n'en fournir qu'une seule. ***C'est à ce niveau que la stratégie gloutonne intervient***. Celle-ci va faire un premier choix de conférencier en suivant un critère à préciser. Ce choix ne sera jamais remis en question et la même stratégie sera appliquée pour trouver les conférenciers suivants.

Après avoir classé les intervalles par valeurs croissantes des heures de début, sélectionner l'intervalle de la première plus petite valeur, puis celui de la deuxième plus petite valeur compatible avec la précédente, et ainsi de suite. On observe que :

$$d_2 = d_4 < d_3 < d_1$$

et que :

$\Rightarrow C_2$ et C_4 ne sont pas compatibles.

$\Rightarrow C_3$ et C_4 ne sont pas compatibles.

$\Rightarrow C_1$ et C_3 ne sont pas compatibles.

Deux solutions $\left[C_2, C_1\right]$ et $\left[C_4, C_1\right]$ restent, en raison de l'égalité $d_2 = d_4$ qui ne permet pas de choisir le premier conférencier. On peut aussi classer les intervalles par valeurs croissantes des heures de fin, puis sélectionner l'intervalle de la première plus petite valeur, puis celui de la deuxième plus petite valeur compatible avec la précédente, et ainsi de suite. On observe à présent que :

$$f_2 < f_4 < f_3 < f_1$$

avec les mêmes incompatibilités que précédemment. Une seule solution est alors :

$$\left[C_2, C_3\right].$$

Cette solution était également proposée par la stratégie précédente. Il est alors légitime de se demander si ce dernier résultat relève d'une stratégie générale pertinente ou d'une situation trop particulière. La situation suivante apporte un premier élément de réponse.

Situation 3

Considérons à présent trois conférenciers (figure 3) et appliquons les deux stratégies précédentes.

$$C_1 : \left[0,0\right[\quad C_2 : \left[1,2\right[\quad C_3 : \left[2,3\right[$$

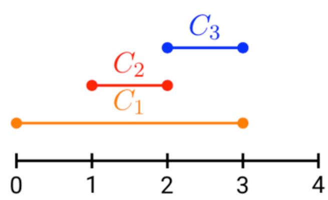


FIGURE 3 – Situation 3

$\Rightarrow$ En classant les heures de début, on a $d_1 < d_2 < d_3$. Seuls C_2 et C_3 sont compatibles. Mais puisque d_1 est la plus petite des heures de début, le planning proposé en suivant cette stratégie se réduit $\left[C_1\right]$ alors que $\left[C_2, C_3\right]$ est une meilleure solution puisqu'elle maximise le nombre de conférenciers.

$\Rightarrow$ En classant les heures de fin, on obtient $f_2 < f_3 = f_1$. Cette fois-ci, le planning proposé en suivant la seconde stratégie fournit le planning $\left[C_2, C_3\right]$.

3.2. Un algorithme glouton

Une solution semble émerger des observations précédentes. On peut donc proposer la stratégie suivante :

$\Rightarrow$ Classer les intervalles par heures de fin croissantes.
$\Rightarrow$ Choisir le conférencier associé au premier intervalle.
$\Rightarrow$ Choisir parmi les intervalles suivants celui du conférencier dont l'intervalle est compatible avec celui du premier conférencier.
Recommencer ainsi avec les intervalles classés suivants jusqu'à ce qu'il n'y en ait plus à traiter. Illustrons la mise en œuvre de cet algorithme sur la situation de la figure 4.

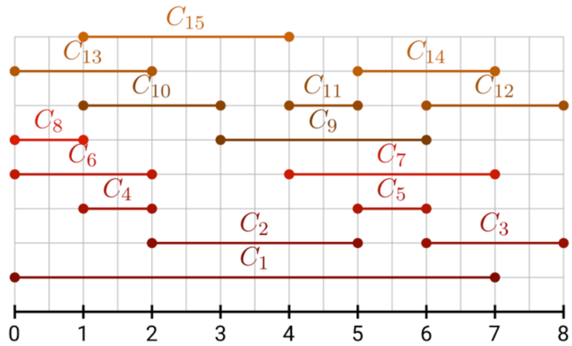


FIGURE 4 – Une situation plus complexe.

Commençons par classer les conférenciers par heures de fin croissantes en notant $\leq$ la relation d'ordre associée.

$$C_8 \leq C_4 \leq C_6 \leq C_{13} \leq C_{10} \leq C_{15} \leq C_2 \leq C_{11} \leq C_5 \leq C_9 \leq C_1 \leq C_7 \leq C_{14} \leq C_3 \leq C_{12}$$

Puis construisons petit à petit le planning.
$\Rightarrow$ Le premier conférencier est $C_8 \rightarrow [C_8]$.
$\Rightarrow$ Le conférencier suivant dont l'intervalle est compatible avec celui de C_8 est $C_4 \rightarrow [C_8, C_4]$.
$\Rightarrow$ Le conférencier suivant compatible avec C_4 est $C_2 \rightarrow [C_8, C_4, C_2]$.
$\Rightarrow$ Le conférencier suivant compatible avec C_2 est $C_5 \rightarrow [C_8, C_4, C_2, C_5]$.
$\Rightarrow$ Le conférencier suivant compatible avec C_5 est $C_3 \rightarrow [C_8, C_4, C_2, C_5, C_3]$.
Ce qui mène au planning final suivant.

$$[C_8, C_4, C_2, C_5, C_3]$$

Remarque 1. Un tel algorithme est effectivement de type glouton. Chaque choix fait sélectionne l'une des meilleures possibilités et ne remet jamais en cause les choix précédents.
Remarque 2. Pour conclure cette proposition d'algorithme, il conviendrait de montrer que la solution obtenue est optimale, c'est-à-dire de cardinal maximum.

3.3. Code Python

Avant de proposer un code Python qui renvoie un planning sous la forme d'un tableau, il convient de s'interroger sur la manière de stocker les intervalles horaires de chaque conférencier. Étant donné n conférenciers, une première idée consiste à placer à l'indice $i \in \{0,\ldots,n-1\}$ d'un tableau `tab_intervalles` l'intervalle $\left[d_{i+1}, f_{i+1}\right]$ du conférencier $i+1$ sous la forme d'un tableau. Reprenant l'exemple de la figure 4, cela donnerait le tableau suivant.

```
tab_intervalles = [[0,7],[2,5],[6,8],[1,2],[5,6],[0,2],[4,7],[0,1],[3,6],[1,3],[4,5],
    ↪ [6,8],[0,2],[5,7],[1,4]]
```

Si une telle solution semble pertinente, elle présente un inconvénient lié à la phase de tri qui, en réorganisant les créneaux horaires, place à un indice i du tableau trié un conférencier qui n'est plus C_i. Sur le tableau ci-dessus, le tri mène au tableau suivant :

```
[[0,1],[1,2],[0,2],[0,2],[1,3],[1,4],[2,5],[4,5],[5,6],[3,6],[0,7],[4,7],
    ↪ [5,7],[6,8],[6,8]]
```

Pour éviter cette difficulté, plusieurs solutions sont envisageables. On peut ajouter un champ aux tableaux des créneaux horaires sous la forme d'une chaîne de caractères qui identifie le conférencier.

```
tab_intervalles = [[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']]
```

Le tableau trié est alors le suivant :

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

```python
import numpy as np

from random import randint

# fonction qui génère de façon aléatoire une liste des plages
# horaires des conférenciers entre 8H00 et 17H00 avec une durée max de 4H
# retourne une liste de liste
def intervalles(nb_intervalles):

    debut = 8
    fin = 17
    duree_max = 4
    tab_intervalles = []

    for i in range(nb_intervalles):
        duree = np.random.randint(1,duree_max)
        deb = np.random.randint(debut,fin-duree)
        tab_intervalles.append([deb, deb+duree, 'C'+str(i+1)])

    return tab_intervalles


# fonction qui génère le planning des conférences par un algorithme glouton
def planning(tab_intervalles):
    nb_intervalles = len(tab_intervalles)
    # tri des intervalles par valeurs croissantes de d'heures de fin
    tab_intervalles = sorted(tab_intervalles, key=lambda act: act[1])
    # tableau du planning
    tab_planning = [tab_intervalles[0]]
    j=0
    for i in range(1,nb_intervalles):
        if tab_intervalles[i][0] >= tab_intervalles[j][1]:
            tab_planning.append(tab_intervalles[i])
            j=i
    return tab_planning


n = 25
lst_intervalles = intervalles(n)
lst_org = planning(lst_intervalles)

print("-----------------------------------")
print("Plage horaire des conférenciers:")
print(lst_intervalles)
print("-----------------------------------")
print("Planning des conférences:")
print(lst_org)
```

```
-----------------------------------
Plage horaire des conférenciers:
[[10, 12, 'C1'], [8, 11, 'C2'], [10, 11, 'C3'], [13, 15, 'C4'], [10, 12, 'C5'], [14, 15, 'C6'], [8, 9, 'C7'],
[13, 14, 'C8'], [12, 14, 'C9'], [10, 12, 'C10'], [12, 13, 'C11'], [9, 10, 'C12'], [13, 16, 'C13'], [11, 13,
'C14'], [13, 15, 'C15'], [15, 16, 'C16'], [8, 11, 'C17'], [11, 14, 'C18'], [13, 14, 'C19'], [10, 13, 'C20'],
[11, 14, 'C21'], [14, 16, 'C22'], [11, 14, 'C23'], [9, 12, 'C24'], [14, 16, 'C25']]
-----------------------------------
Planning des conférences:
[[8, 9, 'C7'], [9, 10, 'C12'], [10, 11, 'C3'], [12, 13, 'C11'], [13, 14, 'C8'], [14, 15, 'C6'], [15, 16,
'C16']]
```

4. Exercices d'application

Exercice 1 : le problème du sac à dos

Vous avez un ensemble d'objets ayant chacun un nom n_i, une valeur v_i et un poids p_i. Vous souhaitez bien entendu maximiser la valeur de votre sac en gardant le poids inférieur à un poids p_{max} maximal, 10 kg par exemple.

Chaque objet peut être représenté par une liste de la forme `objet=(nom,valeur,poids)`. Le nom étant une chaîne de caractère, la valeur un entier de 1 à 10 (10 indiquant un objet très important) et le poids un entier en gramme.

On pourra travailler sur l'ensemble des objets suivants qui sera représenté par une liste de listes :

```
objets=[['book_organic_chemistry',1,1500],['corde',10,4000],['guitare',0,4
000],['carte',8,50],['eau',10,1000],['macbook',2,1500],['casque',9,300],['
crampons',9,800],['piolet',9,400],['doudou',5,200],['smartphone',3,200],['
nourriture',8,2000],['frontale',5,150],['longe',4,400],['trousse_pharmacie
',4,500]]
```

Ecrire une fonction `tri(objets)` qui, à partir de la liste de listes, renverra la liste de listes `objets_tries` des objets triés par ordre décroissant du rapport valeur/poids. A chaque élément de la liste, on ajoutera l'élément (valeur/poids). Vous devez obtenir la liste de listes suivante :

```
[['carte', 8, 50, 0.16], ['frontale', 5, 150, 0.03333333333333333], ['casque', 9, 300, 0.03], ['doudou', 5, 200, 0.025],
['piolet', 9, 400, 0.0225], ['smartphone', 3, 200, 0.015], ['crampons', 9, 800, 0.01125], ['longe', 4, 400, 0.01], ['eau', 10,
1000, 0.01], ['trousse_pharmacie', 4, 500, 0.008], ['nourriture', 8, 2000, 0.004], ['corde', 10, 4000, 0.0025], ['macbook', 2,
1500, 0.0013333333333333333], ['book_organic_chemistry', 1, 1500, 0.0006666666666666666], ['guitare', 0, 4000, 0.0]]
```

Ecrire une fonction `remplir_sac(objets_tries,p_max)` qui, à partir de la liste de listes précédente triée, retournera `(objets_emportes,valeur_sac,poids_sac)` où :

✔ `objets_emportes` est la liste des objets mis dans le sac en optimisant la valeur du sac pour le poids total imposé.

✔ `valeur_sac` est la valeur totale du sac.

✔ `poids_sac` est le poids du sac.

Votre fonction devra utiliser un algorithme glouton évidemment. Vous devez obtenir le résultat suivant pour un poids total de 10 kg maximal :

```
-----------------------------------
valeur du sac = 84
poids du sac (en g) = 10000
liste des objets a emporter:
['carte', 'frontale', 'casque', 'doudou', 'piolet', 'smartphone', 'crampons', 'longe', 'eau', 'trousse_pharmacie', 'nourriture',
'corde']
-----------------------------------
```

Si vous devez partir faire un sommet en montagne, le résultat obtenu est-il intéressant ? A discuter et méditer !

Modifier légèrement le programme pour que le tri s'effectue, de façon décroissante, par la « valeur » de chaque objet et non plus par la « valeur par unité de poids ». Le résultat obtenu est le suivant. Conclusion.

```
---------------------------------
valeur du sac = 84
poids du sac (en g) = 10000
liste des objets a emporter:
['eau', 'corde', 'piolet', 'crampons', 'casque', 'nourriture', 'carte', 'frontale', 'doudou',
'trousse_pharmacie', 'longe', 'smartphone']
---------------------------------
```

Exercice 2 : *Végéter*

A la télévision, on trouve un ensemble de programmes qui sont tous caractérisés par une liste `(ni, di,fi)` où `ni` est le nom du programme, `di` est l'heure de début du programme et `fi` l'heure de fin. Une personne souhaite regarder le plus grand nombre de programmes possible dans une journée.

En utilisant ce qui a été fait dans l'exercice précédent et le paragraphe 3 sur le problème d'organisation, écrire une fonction (qui utilise un algorithme glouton) `vegeter(programmes)` qui renvoie le liste des programmes qu'elle va regarder.