```
>>> L = [4, 9, 8, 1, 6]
>>> tri_selection(L) #la fonction ne renvoie rien
>>> L
[1, 4, 6, 8, 9]
```

Question 9. Quelle est la complexité de `tri_selection(L)` en fonction de la taille n de `L` ?

Question 10. Indiquer sans coder comment modifier votre fonction `tri_selection(L)`, pour qu'elle puisse trier une liste de triplets (comme `('eau', 5000, 200)`), par valeurs décroissantes du ratio score/poids.

```
>>> L = [('eau', 5000, 200), ('barbecue', 20000, 80), ('telephone', 100, 50)]
>>> tri_selection(L)
>>> L
[('telephone', 100, 50), ('eau', 5000, 200), ('barbecue', 20000, 80)]
```

On suppose dans la suite de cette sous-section que la liste d'objets est triée par valeurs décroissantes de ce ratio.

Question 11. Écrire une fonction `remplissage_glouton(L, C)` prenant en entrée une liste `L` de triplets triées par valeurs décroissantes du ratio score/poids, un poids maximal et renvoyant la liste des objets choisis (on rappelle qu'un objet est gardé dès que son poids plus ceux des objets déja choisis n'excède pas `C`).

```
>>> remplissage_glouton(L, 20000) #avec L précédente.
['telephone', 'eau']
```

Question 12. Quelle est la complexité de l'approche gloutonne, sans compter le tri préalable de la liste ?

Question 13. Montrer par un exemple que le remplissage glouton ne donne pas nécessairement une solution optimale au problème : exhiber une liste d'objets triés par ratio score/poids décroissant, un poids maximal C, et tel que l'algorithme glouton ci-dessus propose un remplissage du sac qui ne soit pas maximal pour la somme des scores.

III. 2. Une recherche exhaustive

Pour résoudre notre problème, on choisit ici d'explorer toutes les possibilités. On considère toujours une liste de triplets comme `('eau', 5000, 200)`, mais on ne la suppose plus nécessairement triée par ratio score/poids décroissant.

Question 14. Écrire une fonction `toutes_listes(n)` prenant en entrée un entier n, et renvoyant une liste contenant les 2^n listes possibles constituées uniquement de zéro et de uns. Par exemple :

```
>>> toutes_listes(3)
[[0, 0, 0], [0, 0, 1], [0, 1, 0], [0, 1, 1], [1, 0, 0], [1, 0, 1], [1, 1, 0], [1, 1, 1]]
```

On pourra, sans que ce soit une obligation, procéder par récursivité.

Question 15. Écrire une fonction `poids_score(L, D)` prenant en entrée :

— une liste `L` de triplets comme précédemment ;

— une liste `D` de même taille, constituée uniquement de zéros et de uns.

Votre fonction renverra un couple (poids, score), associé aux sac à dos obtenu en prenant les éléments `L[i]` pour lesquels `D[i]` vaut 1. Par exemple :

```
>>> poids_score([('eau', 5000, 200), ('barbecue', 20000, 80), ('telephone', 100, 50)], [0, 1, 0])
(20000, 80)
>>> poids_score([('eau', 5000, 200), ('barbecue', 20000, 80), ('telephone', 100, 50)], [1, 1, 1])
(25100, 330)
```

Question 16. Déduire des questions précédentes une fonction `remplissage_optimal(L, C)`, ayant les mêmes entrées que `remplissage_glouton`, mais renvoyant une liste maximisant le score d'affection sans dépasser la capacité C du sac.

```
>>> remplissage_optimal(L,20000)
['telephone', 'eau']
```

Question 17. Que donne l'algorithme sur votre exemple de la question 13 ?

Question 18. Estimer la complexité de `remplissage_optimal(L,C)` en fonction de la taille n de la liste `L`.

III. 3. Une programmation dynamique

On explore dans cette sous-section une méthode différente pour la résolution du problème du sac à dos. On note dans la suite les objets $O_0, \dots, O_{n-1}$. Pour l'objet O_i, on note $p_i > 0$ son poids et $s_i > 0$ le score d'affection que Donald lui a attribué. On note également pour $0 \leq c \leq C$ et $0 \leq i \leq n$:

$M_{c,i} =$ score maximal d'affection que l'on peut obtenir avec les objets $O_0, \dots, O_{i-1}$ sans dépasser le poids c

Question 19. Dans cette question, on voit comment calculer tous les $M_{c,i}$, pour $0 \leq i \leq n$ et $0 \leq c \leq C$.

1. Que vaut $M_{c,0}$ pour $0 \leq c \leq C$?
2. Que vaut $M_{0,i}$ pour $0 \leq i \leq n$?
3. Pour $c > 0$ et $i > 0$, justifier brièvement que :

$$M_{c,i} = \begin{cases} M_{c,i-1} & \text{si } p_{i-1} > c \\ \max(M_{c,i-1}, s_{i-1} + M_{c-p_{i-1},i-1}) & \text{si } p_{i-1} \leq c \end{cases}$$

Question 20. Écrire une fonction `scores_optimaux(L,C)` prenant en entrée une liste de triplets (nom de l'objet, poids, score d'affection), et un entier strictement positif `C`, et renvoyant une liste de listes `M` telle que `M[c][i]` contienne précisément $M_{c,i}$. On pourra utiliser la fonction suivante :

```
def init(n,C):
    return [[0]*(n+1) for c in range(C+1)]
```

Question 21. Discuter brièvement de comment construire un sac à dos optimal à partir de la liste `M`. L'implémenter en une fonction `remplissage_optimal2(L,C)`, similaire à celle de la question 16.

Question 22. Quelle est la complexité de votre fonction ?

Question 23. Dans quels cas préférer l'approche de la section III.2, et celle de la section III.3 ?

III. 4. Variante fractionnaire

On considère dans cette sous-section une variante du problème du sac à dos, mais on s'autorise désormais à prendre des fractions d'objets plutôt que l'objet entier. Pour un objet de poids p et de score affectif s, prendre une fraction x de l'objet (avec $x \in [0, 1]$) apporte un poids $x \cdot p$ et un score affectif $x \cdot s$. On ne demande pas de coder dans cette section.

Question 24. On souhaite montrer que l'approche gloutonne, consistant à trier les objets par ratio $\frac{\text{score}}{\text{poids}}$ décroissant, puis à considérer les objets un à un, et prendre pour chacun une portion maximale entre 0 et 1 sans dépasser la capacité du sac, mène à un remplissage optimal du sac. Pour cela, on note les objets triés par ratio $r_i = \frac{s_i}{p_i}$.

1. Montrer qu'un remplissage du sac prenant une fraction $x_i < 1$ de l'objet numéro i et une fraction $x_j > 0$ de l'objet numéro j, avec $r_j < r_i$, n'est pas optimal.
2. Conclure sur l'optimalité de l'approche gloutonne.