Exercice 5. Déménagement

```
import random
```

Question 1 –

```
def etape_dem(cartons, p, c):
    for i in range(len(cartons)):
        if p + cartons[i] <= c:
            cartons[i] += p
            return
    cartons.append(p)
```

Question 2 –

```
def dem_glouton(n, c):
    cartons = []
    print()
    for _ in range(n):
        p = random.randint(1, c)
        etape_dem(cartons, p, c)
        print(p, "->" , cartons)
```

Question 3 – Dans la fonction `etape_dem`, il y a une boucle qui fait `len(cartons)` $\leqslant$ `n` tours de boucle. Chaque tour s'exécute en temps constant. Donc cette fonction est de complexité linéaire en `n`.

Dans la fonction `dem_glouton`, il y a `n` tours de boucle qui s'exécutent en temps linéaire en `n`.

On a donc une complexité quadratique en `n`.

Question 4 – Prenons l'exemple où $c = 5$ et les poids des cartons sont $1, 3, 2, 2, 2$. Voici l'évolution de la variable `cartons` :

```
1 -> [1]
3 -> [4]
2 -> [4, 2]
2 -> [4, 4]
2 -> [4, 4, 2]
```

L'algorithme glouton utilise donc 3 cartons alors qu'il est possible de n'en utiliser que deux :

`[1+2+2, 3+2]`.

Exercice 6. Le problème du sac à dos

Question 1 –

```
def make_P(obj):
    P = []
    for s in obj:
        v,p = obj[s]
        r = p/v
        P.append((r,s))
    return sorted(P)
```

Question 2 –

```
def sad_glouton(obj, poids_max):
    prio = make_P(obj)
    res = []
    poids = 0
    valeurs = 0
    for _, s in prio:
        v,p = obj[s]
        if p + poids <= poids_max:
            res.append(s)
            poids   +=  p
            valeurs += v
    return res, valeurs, poids
```

Question 3 – Sans compter l'appel à la fonction `sorted`, on a une complexité linéaire en `len(objets)`.

En effet, dans la fonction `make_P`, on a une boucle `for` qui s'exécute en temps linéaire.

Dans la fonction `sad_glouton`, l'appel à la fonction `make_P` s'exécute en temps linéaire. La boucle `for` s'exécute aussi en temps linéaire.

Question 4 – La solution n'est pas optimale pour $P \in \{8, 9, 18, 19, 20\}$. Voici les solutions de l'algorithme glouton et les solutions optimales :

P	Solution glouton	Valeur glouton	Solution optimale	Valeur optimale
8	`['smartphone', 'montre', 'lunettes', 'stylo']`	281	`['ordinateur']`	390
9	`['smartphone', 'montre', 'lunettes', 'gourde']`	290	`['ordinateur', 'lunettes']`	430
18	`["lingot", "smartphone", "montre", "lunettes", "stylo"]`	10 281	`["ordinateur", "lingot"]`	10 390
19	`["lingot", "smartphone", "montre", "lunettes", "gourde"]`	10 290	`["lingot", "lunettes", "ordinateur"]`	10 430
20	`["lingot", "smartphone", "montre", "lunettes", "gourde", "stylo"]`	10 291	`["lingot", "ordinateur", "montre"]`	10 480

Question 5 – On peut par exemple prendre $P = 4$ et

```
objets = {"objet 1": (5,3), "objet 2": (3,2), "objet 3": (3,2)}
```

Sur cet exemple, l'algorithme glouton sélectionne d'abord l'objet 1. Il n'y a alors plus de place pour les objets 2 et 3. La valeur totale de la solution donnée par l'algorithme glouton est donc de 5.

On peut faire mieux : en sélectionnant les objets 2 et 3, on obtient une valeur de 6.

Question 6 –

```
# Renvoie un couple (clé,valeur) du dictionnaire.
# Hypothese: d non vide.
def get_one_item(d):
    for c in d:
        return c, d[c]
```

```
# Renvoie une liste contenant toutes les listes M décrites dans l'énoncé ainsi
# que la somme des valeurs et la somme des poids des objets de M.
def get_M(obj, poids_max):
    if poids_max < 0:
        return []
    if len(obj) == 0:
        return [([],0,0)]
    s, (v,p) = get_one_item(obj)
    del obj[s]
    res1 = get_M(obj, poids_max-p)
    for i in range(len(res1)):
        M, v0, p0 = res1[i]
        M.append(s)
        res1[i] = M, v0 + v, p0 + p
    res2 = get_M(obj, poids_max)
    obj[s] = v,p  # Ne pas oublier cette ligne
    return res1 + res2
```

```
def sad_BF(obj, poids_max):
    L = get_M(obj, poids_max)
    M_max = []
    v_max = 0
    p_max = 0
    for M, v, p in L:
        if v > v_max:
            M_max = M
            v_max = v
            p_max = p
    return M_max, v_max, p_max
```