Python – Algorithmes gloutons

Le rendu de monnaie

Pour les exercices de cette partie, vous pourrez utiliser la liste suivante pour avoir la valeur des billets et pièces d'euros :

```
euros = [500, 200, 100, 50, 20, 10, 5, 2, 1]
```

Exercice 1 : Compléter la fonction `rendu_monnaie(montant, liste_valeurs)` qui renvoie la liste des nombres de billets et pièces nécessaire pour obtenir `montant`, en en utilisant le moins possible. Les valeurs des billets et des pièces sont données dans l'ordre décroissant. Vous pourrez par exemple utiliser un algorithme glouton.

```
def rendu_monnaie(montant, liste_valeurs):
    liste_rendu = ...
    for val in liste_valeurs:
        while ...:
            liste_rendu.append(...)
            montant = ...
    return liste_rendu
```

```
>>> rendu_monnaie(358, euros)
[200, 100, 50, 5, 2, 1]
>>> sum([200, 100, 50, 5, 2, 1])
358
>>> rendu_monnaie(740, euros)
[500, 200, 20, 20]
>>> sum([500, 200, 20, 20])
740
```

La fonction `sum(liste)` renvoie la somme de tous les éléments d'une liste de nombres.

Exercice 2 : Compléter la fonction `rendu_monnaie2(montant, liste_valeurs)` qui renvoie la plus petite liste de couples `(nombre, valeur)` correspondant au nombre de fois qu'il faut utiliser `valeur` pour obtenir `montant`. Les valeurs des billets et des pièces sont données dans l'ordre décroissant.

```
def rendu_monnaie2(montant, liste_valeurs):
    liste_rendu = ...
    for val in liste_valeurs:
        n = ...
        while ...:
            montant = ...
            n = ...
        if ...:
            liste_rendu.append((n, val))
    return liste_rendu
```

```
>>> rendu_monnaie2(358, euros)
[(1, 200), (1, 100), (1, 50), (1, 5), (1, 2), (1, 1)]
```

```
>>> rendu_monnaie2(740, euros)
[(1, 500), (1, 200), (2, 20)]
>>> rendu_monnaie2(4567, euros)
[(9, 500), (1, 50), (1, 10), (1, 5), (1, 2)]
```

Remarques :

- Il n'est pas conseillé d'utiliser des centimes dans la liste des valeurs, à cause des erreurs d'arrondis sur les nombres réels. Ou alors, il faut que toutes les valeurs soient exprimées en centimes.
- Il est possible de simplifier l'algorithme en utilisant la division sur les entiers pour trouver directement le nombre de fois qu'il faut utiliser chacune des valeurs.

Le problème du sac à dos

Afin de pouvoir générer des listes d'objets et de faire des algorithmes de force brute, nous allons utiliser les modules suivants :

```
import random
import itertools
```

Pour ce problème, nous modéliserons les objets à l'aide de dictionnaires :

```
objets = [{"nom": "Objet 1", "valeur": 10, "poids": 9},
          {"nom": "Objet 2", "valeur": 7, "poids": 12},
          {"nom": "Objet 3", "valeur": 1, "poids": 2},
          {"nom": "Objet 4", "valeur": 3, "poids": 7},
          {"nom": "Objet 5", "valeur": 2, "poids": 5}]
```

Afin de tester toutes les solutions possilbles, nous allons générer des listes de booléens indiquant s'il faut prendre tel ou tel objet :

```
>>> list(itertools.product([False, True], repeat=3))
[(False, False, False), (False, False, True), (False, True, False),
  (False, True, True), (True, False, False), (True, False, True),
  (True, True, False), (True, True, True)]
```

Cette commande permet de faire toutes les listes de booléens de longueur donnée. Chaque booléen correspond à un objet de la liste d'objets et indique s'il faut prendre ou pas cet objet.

Exercice 3 : Compléter la fonction `evaluer_solution(liste_objets, solution)` qui renvoie le poids, la valeur et la liste des noms d'objets pris dans la solution `solution`.

```
def evaluer_solution(liste_objets, solution):
    poids = ...
    valeur = ...
    objets_selec = ...
    for i in range(...):
        if ...:
            poids = poids + liste_objets[i]["poids"]
            valeur = ...
            objets_selec.append(...)
    return poids, valeur, objets_selec
```

```
>>> evaluer_solution(objets, [True, False, False, True, False])
(16, 13, ['Objet 1', 'Objet 4'])
```

Exercice 4 : Compléter la fonction `sac_a_dos_brute(liste_objets, poids_max)` qui renvoie la valeur, le poids et les objets sélectionnés pour la meilleure solution.

```
def sac_a_dos_brute(liste_objets, poids_max):
    n = len(liste_objets)
    meilleur_val = ...
    meilleur_poids = ...
    meilleur_selec = ...
    solutions = itertools.product([False, True], repeat=n)
    for solution in solutions:
        poids, valeur, selec = evaluer_solution(..., ...)
        if ...:
            # C'est une solution valide et meilleure
            meilleur_val = ...
            meilleur_poids = ...
            meilleur_selec = ...
    return meilleur_val, meilleur_poids, meilleur_selec
```

```
>>> sac_a_dos_brute(objets, 15)
(12, 14, ['Objet 1', 'Objet 5'])
```

Dans cet exemple, on voit que la meilleure solution consiste à prendre les objets 1 et 5, pour une valeur totale de 12, et un poids de 14, avec une limite fixée à 15.
Le temps de calcul avec la méthode par force brute devient très rapidement trop long. C'est pour cela qu'on utilise un algorithme glouton.
On construit une liste de n-uplets (rentabilité, poids, valeur, nom) et on la classe dans l'ordre décroissant. On parcourt ensuite cette liste en regardant objet par objet s'il est possible de le prendre ou pas.

Exercice 5 : Compléter la fonction `sac_a_dos_glouton(liste_objets, poids_max)` qui prend une liste d'objets et une limite de poids et renvoie une solution obtenue avec l'algorithme glouton.

```
def sac_a_dos_glouton(liste_objets, poids_max):
    liste_rentabilite = []
    poids_restant = poids_max
    for objet in liste_objets:
        val = objet["valeur"]
        poids = ...
        nom = ...
        liste_rentabilite.append((..., poids, val, nom))
    # On trie du plus rentable au moins rentable
    liste_rentabilite.sort(reverse=True)
    objets_pris = ...
    valeur_totale = ...
    for rentabilite, poids, val, nom in liste_rentabilite:
        if ...:
            objets_pris.append(nom)
            poids_restant = ...
            valeur_totale = ...
    return valeur_totale, poids_max-poids_restant, objets_pris
```

Cette fonction est bien plus rapide que la méthode par force brute, mais ne donne pas forcément la meilleure solution :

```
>>> sac_a_dos_glouton(objets, 15)
(11, 11, ['Objet 1', 'Objet 3'])
```

La valeur obtenue est moins bonne qu'avec l'autre méthode, mais n'en est pas très loin.

Exercice 6 :

1) Comparer les résultats trouvés par les deux méthodes sur cette liste d'objets, avec un poids maximum de 10 :

```
objets = [{"nom": "Objet 1", "valeur": 80, "poids": 1},
          {"nom": "Objet 2", "valeur": 100, "poids": 10}]
```

2) Expliquer les différences observées.

Afin de tester les deux méthodes, nous allons générer des listes d'objets au hasard.

Exercice 7 : Compléter la fonction ci-dessous qui permet de générer au hasard une liste d'objets en indiquant les valeurs minimales et maximales ainsi que les poids minimums et maximums. On rappelle que la fonction `random.randint(a, b)` renvoie une valeur entière comprise entre `a` et `b` inclus.

```
def genere_liste_objets(nb_objets, val_min, val_max, poids_min, poids_max):
    liste_objets = ...
    for i in range(...):
        objet = dict()
        objet["nom"] = "Objet " + str(...)
        objet["valeur"] = random.randint(..., ...)
        objet[...] = random.randint(..., ...)
        liste_objets.append(...)
    return ...
```

```
>>> genere_liste_objets(4, 100, 400, 1, 10)
[{'nom': 'Objet 1', 'valeur': 301, 'poids': 7},
 {'nom': 'Objet 2', 'valeur': 292, 'poids': 2},
 {'nom': 'Objet 3', 'valeur': 303, 'poids': 1},
 {'nom': 'Objet 4', 'valeur': 385, 'poids': 8}]
```

Exercice 8 : Générer des listes d'objets au hasard et comparer les résultats obtenus par les deux méthodes. Attention à ne pas utiliser des listes de longueurs supérieur à 20 pour la méthode par force brute.

Exercice 9 : Tester l'algorithme glouton avec des listes de centaines ou de milliers d'objets, pour voir l'efficacité au niveau du temps de calcul.

Le voyageur de commerce

Pour ce problème, nous allons utiliser un tableau de taille $n \times n$ pour représenter les distances entre les villes.

```
matrice_exemple = [[0, 55, 303, 188, 183],
                   [55, 0, 306, 176, 203],
                   [303, 306, 0, 142, 153],
                   [188, 176, 142, 0, 123],
                   [183, 203, 153, 123, 0]]
```

Exercice 10 : Compléter la fonction `voyageur_commerce_glouton(matrice)` qui cherche le plus court chemin en utilisant l'algorithme glouton. On part de la ville 0 et ensuite, à chaque fois, on cherche la ville non visitée la plus proche de la ville courante et on passe à cette ville. À la fin, on rajoute la distance entre la dernière et la première ville pour le chemin du retour. La fonction renvoie la distance trouvée et le chemin passant par toutes les villes. On ne rajoute pas la ville 0 à la fin de la liste, mais on pourrait.

```
def voyageur_commerce_glouton(matrice):
    parcours = [0]
    ville_courante = 0
    distance_totale = ...
    for i in range(len(matrice)-1):
        ville_la_plus_proche = 0
        d_min = 9999999999  # valeur arbitraire
        distances = matrice[ville_courante]
        for j in range(len(distances)):
            if j not in ... and ... < d_min:
                d_min = ...
                ville_la_plus_proche = ...
        distance_totale = ...
        parcours.append(...)
        ville_courante = ...
    # on rajoute la distance entre la première ville parcourue et la dernière
    distance_totale += matrice[...][...]
    return distance_totale, parcours
```

```
>> voyageur_commerce_glouton(matrice_exemple)
(810, [0, 1, 3, 4, 2])
```

Afin de chercher la solution optimale, nous allons tester tous les chemins possibles.

```
>>> list(itertools.permutations(range(3), 3))
[(0, 1, 2), (0, 2, 1), (1, 0, 2), (1, 2, 0), (2, 0, 1), (2, 1, 0)]
```

Cette commande permet de générer toutes les combinaisons possibles avec les nombres de 0 à 2. Comme nous l'avons vu, ce n'est pas la peine de tester tous les circuits. On peut fixer le point de départ. On prendra donc les valeurs à partir de 1 et on rajoutera le fait de partir de 0.

```
>>> list(itertools.permutations(range(1, 3), 2))
[(1, 2), (2, 1)]
```

Cela permet de réduire grandement le nombre de chemins à tester. Mais si lorsque le nombre de villes augmente, ce nombre deviendra quand même rapidement gigantesque. Il n'est pas conseillé de tester l'algorithme par force brute avec plus d'une dizaine de villes.

Exercice 11 : Compléter la fonction `voyageur_commerce_brute(matrice)` qui cherche le plus court chemin en faisant une recherche exhaustive. Puisque les permutations sont données sous forme de n-uplets, on doit utiliser `(0,)` pour rajouter 0 au début du chemin considéré.

```
def voyageur_commerce_brute(matrice):
    n = len(matrice)
    combinaisons = itertools.permutations(range(1, n), n-1)
    meilleur = []
    pire = []
    longueur_min = 99999999999
    for parcours in combinaisons:
        # Ville de départ à la première visitée
        longueur = matrice[...][...]
        # On rajoute tous les déplacements du parcours
        for i in range(...):
            longueur = longueur + matrice[parcours[...]][parcours[...]]
        # Retour à la ville de départ
        longueur = longueur + matrice[parcours[...]][...]
        # On regarde si on a une meilleure solution
        if ...:
            meilleur = (0,) + parcours
            longueur_min = ...
    return longueur_min, meilleur
```

```
>>> voyageur_commerce_brute(matrice_exemple)
(709, (0, 1, 3, 2, 4))
```

Afin de tester ces algorithmes sur d'autres exemples, nous allons générer des points dans un repère et calculer les distances les séparant.

Exercice 12 : Écrire une fonction `genere_points(nb_points)` qui génère une liste de couples `(x, y)`, où les deux valeurs sont tirées au hasard dans l'intervalle $[0;100]$. Vous pouvez utiliser la commande `random.uniform(0, 100)` qui renvoie un réel choisi au hasard entre 0 et 100.

```
>>> random.uniform(0,100)
10.506382593738529
>>> genere_points(3)
[(98.35288913986197, 45.75239295229048),
 (69.75253012777488, 99.14961943360957),
 (11.91408689187744, 17.876060649733528)]
```

La fonction `distance(pt1, pt2)` permet de calculer la distance entre deux points, données sous la forme `(x, y)`.

```
import math

def distance(pt1, pt2):
    x1, y1 = pt1
    x2, y2 = pt2
    return math.sqrt((x2-x1)**2 + (y2-y1)**2)
```

Exercice 13 : Compléter la fonction `calcul_matrice_distances(liste_points)` qui renvoie la matrice des distances entre les points donnés.

```python
def calcul_matrice_distances(liste_points):
    n = len(liste_points)
    matrice = [[0 for i in range(n)] for j in range(n)]
    for i in range(n):
        for j in range(i+1, n):
            d = distance(..., ...)
            matrice[i][j] = d
            matrice[...][...] = d
    return matrice
```

```
>>> calcul_matrice_distances(genere_points(3))
[[0, 82.42860061936726, 36.90459181116363],
 [82.42860061936726, 0, 85.34081684848618],
 [36.90459181116363, 85.34081684848618, 0]]
```

Exercice 14 : Comparer les résultats trouvés avec les deux algorithmes avec des listes de longueur 10 au maximum. Vous pouvez aussi modifier la version par force brute pour qu'elle renvoie également la longueur du pire parcours.

```
>>> liste_points = genere_points(10)
>>> matrice = calcul_matrice_distances(liste_points)
>>> voyageur_commerce_glouton(matrice)
(383.07018202287094, [0, 5, 8, 9, 3, 4, 6, 1, 2, 7])
>>> voyageur_commerce_brute(matrice)
(315.9671698673584, (0, 5, 8, 1, 2, 6, 4, 9, 3, 7), 793.2125278743039)
```

Pour visualiser le trajet proposé, vous pouvez utiliser la fonction suivante :

```python
import matplotlib.pyplot as plt

def afficher_parcours(liste_points, parcours):
    parcours.append(parcours[0]) # on boucle
    p_x = [pt[i][0] for i in parcours] # liste des abscisses
    p_y = [pt[i][1] for i in parcours] # liste des ordonnées
    plt.plot(p_x, p_y, '.-') # on affiche en mettant des points
    plt.show()
```

```
>>> liste_points = genere_points(300)
>>> matrice = calcul_matrice_distances(liste_points)
>>> d, parcours = voyageur_commerce_glouton(matrice)
>>> afficher_parcours(liste_points, parcours)
```