TP 11
Algorithmes Gloutons – Sujet

Avant de commencer Avant toute chose, créer un répertoire TP11 dans votre répertoire "Informatique". Sur le site `https://ptsilamartin.github.io/info/TP.html`, télécharger `algorithmes_gloutons.py` que vous copiez dans votre répertoire TP11.

Rappel : On prendra bien soin, dans tout le TP, de documenter les fonctions écrites et de les tester. Les tests seront présentés en commentaires dans le script.

Le voyageur de commerce

A partir du travail de E. Detrez `https://info.faidherbe.org/`.

Lors des vacances, Solal souhaite aller passer un moment avec tous ses collègues de PTSI répartis dans toute la région. Il cherche l'itinéraire minimisant la distance totale parcourue. À la fin du périple, il souhaite retourner à la ville de départ.

re ville de la région Auvergne – Rhône - Alpes

	Lyon	Saint-Etienne	Grenoble	Clermont-Ferrand	Annecy	Valence
	0	64	106	165	138	102
	64	0	155	146	186	121
	106	155	0	270	105	94
d	165	146	270	0	300	263
	138	186	105	300	0	208
	102	121	94	263	208	0

Introduction

On appelle itinéraire un parcours de ville, par exemple : Lyon → Saint-Etienne → Grenoble → Clermont-Ferrand → Annecy → Valence.

Question 1 Déterminer le nombre d'itinéraires possibles au départ de Lyon. Déterminer le nombre d'itinéraires possibles pour un passage par n villes.

Correction

Pour 3 villes :

- 1, 2, 3
- 1, 3, 2

Pour 4 villes :

- 1, 2, 3, 4
- 1, 2, 4, 3
- 1, 3, 2, 4
- 1, 3, 4, 2
- 1, 4, 2, 3
- 1, 4, 3, 2

Conjecture :

- pour 3 villes, on peut faire 2×1 itinéraires ;
- pour 4 villes, on peut faire $3 \times 2 \times 1$ itinéraires ;

Pour n villes, il y a donc $(n-1)!$ itinéraires possibles.

Pour un grand nombre de villes, il n'est pas possible d'évaluer toutes les distances de tous les itinéraires et de prendre le plus court.

Définition – Algorithmes gloutons

Les algorithmes gloutons 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 et ainsi de suite.

Dans notre cas la solution optimale consiste en visiter la ville la plus proche.

Question 2 Appliquer la stratégie gloutonne à la main et donner la distance correspondante. Cette solution vous semble-elle être la meilleure ?

Correction

Lyon → Saint-Etienne → Valence → Grenoble → Annecy → Clermont-Ferrand → Lyon. (Soit `[0,1,5,2,4,3,0]`).
$d = 64 + 121 + 94 + 105 + 300 + 165 = 849$ km.
Lyon → Clermont-Ferrand → Saint-Etienne → Valence → Grenoble → Annecy → Lyon.
$d = 165 + 146 + 121 + 94 + 105 + 138 = 769$ km. (Minimum local... tant qu'on n'a pas prouvé qu'il était global).

Résolution

On associe un nombre (indice) à chaque ville : 0 à Lyon, 1 à Saint-Etienne, 2 à Grenoble, 3 à Clermont-Ferrand, 4 à Annecy, 5 à Valence. On appelle table des distances `distance` la liste de liste modélisant la table **??**.

Ainsi `distance[0][2]=distance[2][0]=106` la distance Lyon – Grenoble.

Question 3 Ecrire une fonction `enleve(L, e)` qui prend en paramètre une liste `L` et un élément `e` et renvoie une liste contenant les éléments de `L` à l'exception de `e`.

Correction

```
def enleve(L, e) :
    # renvoie une liste correspondant à L sans l'élément e
    L_modif = []
    for elt in L :
        if elt != e :
            L_modif.append(elt)
    return L_modif

def enleve(L, e) :
    # renvoie une liste correspondant à L sans l'élément e
    i = L.index (e)
    return L[0: i] + L [i+1:]
```

Question 4 Ecrire une fonction récursive `enleve_rec(L, e)` qui permet de réalise la même opération.

Correction

```python
def enleve_rec(L, e):
    if len(L) == 0 :  # Si la liste est vide
        return []
    elif L[0] == e:  # Si le premier élément est égal à e
        return enleve_rec(L[1:], e)
    else:  # Sinon, conservez le premier élément
        return [L[0]] + enleve_rec(L[1:], e)
```

Exemple –

Par exemple, `enleve([7, 5, 9, 3, 5], 3)` donnera `[7, 5, 9, 5]`.

Question 5 Écrire une fonction `proche(v_actuelle, v_non_visit)` qui prend en paramètre la ville actuelle `v_actuelle` et la liste des villes à visiter `v_non_visit` et renvoie la ville la plus proche de `v_actuelle` parmi les villes de `v_non_visit` en prenant en compte les distances `dist` entre villes.

Correction

```python
def proche (v_actuelle, v_non_visit) :
    """ renvoie la ville la plus proche de la ville actuelle selon les
    distance dist et la liste des villes non encore visitées """
    n = len(v_non_visit) # nombre de villes à visiter
    v_proche = v_non_visit [0] # premi ère ville à visitée
    d_min = dist [v_actuelle][v_proche] # intialisation de la distance
    minimale
    for j in range (1, n) : # pour toutes les autres villes
        d = dist [v_actuelle][v_non_visit [j]] # récupération de la
    distance
        if d < d_min : # si on trouve une ville plus proche
            v_proche = v_non_visit[j] # mise à jour de la ville la plus
    proche
            d_min = d # mise à jour de la distance minimale.
    return v_proche
```

Exemple –

Par exemple, la ville la plus proche de Lyon est Saint-Etienne et `proche(0, [1, 2, 3, 4, 5])` qui donne `1` : Saint-Etienne.

Question 6 Écrire une fonction `glouton(depart)` qui prend en paramètre le numéro de la ville de départ, depart et qui renvoie l'itinéraire sous forme d'une liste ordonnée des numéros des villes à visiter. En appliquant cette fonction, répondre au problème de notre voyageur de commerce. Conclure.

Correction

```python
def glouton(depart) :
    itineraire = [depart] # initialisation à la ville de départ
    # ville actuelle, initialisation à la ville de départ
    v_actuelle = depart
```

```
    v_non_visit = enleve(villes,depart) # on enlève la ville actuelle des
    villes à visiter
    # Tant qu'il y a des villes à visiter
    while len( v_non_visit ) != 0:
        # recherche de la ville la plus proche des villes à visiter
        v_proche = proche(v_actuelle, v_non_visit)
        # ajout dans l'itinéraire
        itineraire.append (v_proche)

        # on enlève la ville la plus proche des villes à visiter
        v_non_visit = enleve(v_non_visit, v_proche)

        # mise à jour de la ville actuelle par la ville la plus proche
        v_actuelle = v_proche
    # fin de l'itinéraire avec retour à la ville de départ
    itineraire.append(depart)
    return itineraire
```

Version récursive à vérifier

```
def glouton_recursif(depart, v_non_visit, itineraire):
    if not v_non_visit:  # Condition de base : plus de villes à visiter
        itineraire.append(depart)  # Retour à la ville de départ
        return itineraire

    v_actuelle = itineraire[-1]  # La dernière ville ajoutée à l'itinéraire
    v_proche = proche(v_actuelle, v_non_visit)  # Trouver la ville la plus
    proche
    itineraire.append(v_proche)  # Ajouter la ville proche à l'itinéraire

    # Mettre à jour les villes non visitées
    v_non_visit = enleve(v_non_visit, v_proche)

    # Appel récursif avec les paramètres mis à jour
    return glouton_recursif(depart, v_non_visit, itineraire)
```

```
def glouton(depart):
    v_non_visit = enleve(villes, depart)  # Initialiser les villes non
    visitées
    itineraire = [depart]  # Initialiser l'itinéraire avec la ville de dé
    part
    return glouton_recursif(depart, v_non_visit, itineraire)
```

Question 7 Écrire une fonction `distance(itineraire)` qui prend en paramètre un itinéraire sous forme d'une liste ordonnée des villes à visiter et renvoie la distance totale de l'itinéraire.

Correction

```
def distance (itineraire) :
    n = len(itineraire)
    d = 0
    for i in range (1, n) :
        d = d + dist[itineraire[i - 1]][itineraire[i]]
    return d
```

Question 8 Écrire une version récursive de la fonction précédente.

Correction

Version récusive à tester.

```
def distance_recursive(itineraire, dist):
    # Cas de base : si l'itinéraire est vide ou ne contient qu'un seul point, la distance est 0
    if len(itineraire) < 2:
        return 0

    # Calculer la distance entre le premier point et le reste de l'itinéraire
    return dist[itineraire[0]][itineraire[1]] + distance_recursive(itineraire[1:], dist)
```

Algorithme Glouton - Remplir un sac à dos

A partir du travail de Nicolas VIDAL et de Nicolas COURRIER pour l'UPSTI.

Un promeneur souhaite transporter dans son sac à dos le fruit de sa cueillette. La cueillette est belle, mais trop lourde pour être entièrement transportée dans le sac à dos. Des choix doivent être faits. Il faut que la masse totale des fruits choisis ne dépasse pas la capacité maximale du sac à dos.

Les fruits cueillis ont des valeurs différentes, et le promeneur souhaite que son chargement soit de la plus grande valeur possible.

Fruits cueillis	Prix au kilo	Quantité ramassée
Framboises	24 €/kg	1 kg
Myrtilles	16 €/kg	3 kg
Fraises	6 €/kg	5 kg
Mures	3 €/kg	2 kg

La capacité du sac à dos n'est que de 5 kg.

On suppose que la masse d'un unique fruit est négligeable par rapport à la masse totale du sac en charge.

Présentation de l'algorithme et implémentation – Version fractionnaire

Le principe pour optimiser le chargement est de commencer par mettre dans le sac la quantité maximale de fruits les plus chers par unité de masse. S'il y a encore de la place dans le sac, on continue avec les fruits les plus chers par unité de masse parmi ceux restants, . . . et ainsi de suite jusqu'à ce que le sac soit plein.
Il est possible qu'on ne puisse prendre qu'une fraction de la quantité de fruits disponible, si la capacité maximale du sac est atteinte. L'algorithme sera donc constitué

d'une structure itérative, incluant une structure alternative.

Dans `algorithmes_gloutons.py`, nous avons ébauché l'algorithme de résolution du problème et implémenté `cueillette` par la liste des fruits **triée** par prix au kilo décroissant.
Un fruit est représenté par une liste `fruit=[prix au kilo,nom,quantité ramassée]`.

`cueillette = [[24,"framboises",1], [16,"myrtilles",3], [6,"fraises",5], [3,"mures",2]]`.

Question 9 Dans `algorithmes_gloutons.py`, compléter la fonction `sac_a_dos(L:list, capacite:int)` qui prend en argument une liste de listes `L` modélisant la cueillette et la capacité du sac à dos, et appliquant la méthode décrite précédemment. Cette fonction renvoie :

- une liste de listes des fruits contenus dans le sac à dos avec leur masse correspondante de façon à avoir le chargement de valeur maximale ;
- la valeur du chargement.

La liste `cueillette` ne doit pas être modifiée. Vous pourrez si nécessaire utiliser une copie de la liste `cueillette` que vous pourrez modifier. La copie de liste de listes se fait avec les instructions :

```
import copy # au début de votre script
L=copy.deepcopy(votreListe) # quand cela est nécessaire
```

Question 10 A la suite du script `algorithmes_gloutons.py`, taper les lignes de code permettant d'utiliser la fonction `sac_a_dos()` pour afficher les fruits et les quantités à choisir pour remplir le sac à dos, à partir de la liste `cueillette`. Exécuter votre programme et vérifier le résultat.

Une variante de ce problème ne trouve pas de solution optimale par la méthode gloutonne. Nous l'étudions ci-après.

Version non fractionnaire du problème du sac à dos

On suppose maintenant que les éléments à transporter ne sont pas fractionnables. Les fruits parmi lesquels choisir sont présentés dans le tableau suivant.

Fruits	Prix au kilo	Masse d'un fruit	Quantité disponible
Melon de cavaillon	3 €/kg	1 kg	1
Melon jaune	2.5 €/kg	2 kg	1
Pastèque	2 €/kg	3 kg	1

Cet ensemble de fruits est modélisé dans `algorithmes_gloutons.py` par :

- un fruit non fractionnable est représenté par une liste `prix au kilo,nom,masse, quantité disponible`
- une liste de listes `fruitsDisponibles = [[3,"melon de cavaillon",1,1], [2.5,"melon jaune",2,1], [2,"pastèque",3,1]]`.

L'objectif est toujours de placer dans le sac à dos le chargement de valeur maximale, de masse totale inférieure à 5kg. Par contre, les éléments n'étant fractionnables, il est possible que les choix successifs mène à un chargement qui ne remplit pas complètement le sac à dos.

On se propose de tester la méthode gloutonne pour cette nouvelle formulation.

Question 11 Dans `algorithmes_gloutons.py`, copier-coller `sac_a_dos(L, capacite)` que vous renommerez `sac_a_dos_V2(L, capacite)` et adapter la fonction de sorte qu'elle applique la méthode gloutonne à la version non fractionnaire du problème.

Question 12 Le résultat obtenu est-il optimal ? Comparer avec la solution `S = [2.5, "melon jaune",2,1], [2,"pastèque",3,1]]`. Quelle est la solution dont la valeur est maximale ?

Question 13 De la propriété du choix glouton et de la propriété de la sous-structure optimale, laquelle n'est pas respectée dans cette formulation du problème ? En quoi l'autre propriété l'est ?