TP algorithmes gloutons

1. Le problème du rendu de monnaie

Rappel : le problème du rendu de monnaie consiste à déterminer le nombre minimal de billets et pièces nécessaires pour rendre une somme donnée. Dans toute la suite, les « pièces » désignent indifféremment les véritables pièces que les billets. Par exemple, supposons qu'on dispose d'un système monétaire formé des valeurs 200, 100, 50, 20, 10, 5, 2 et 1 €. Dans cet exemple, pour simplifier, les centimes et le billet de 500 € sont ignorés.

On a mis en place l'algorithme suivant :

```
s=somme à rendre
tant que s>0 faire
    rendre la plus grande valeur v ≤ s
    s=s-v
```

Il s'agit bien d'un algorithme glouton car à chaque étape, on fait le meilleur choix sans revenir en arrière.

Question 1

Compléter la fonction `renduMonnaie` ci-dessous qui prend comme argument un entier `s` et qui renvoie la liste minimale des pièces à rendre de somme égale à `s`.

Par exemple, `renduMonnaie(43)` devra renvoyer la liste `[20,20,2,1]`.

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


def renduMonnaie(s):
    """In : un entier s
    Out: liste des pièces à rendre de somme s"""
    L = []
    i = 0
    while s > 0:
        if valeurs[i] <= s:
            ...
            ...
        else:
```

```
            ...
    return L

# Jeu de tests

# encore un test

# Test de la fonction précédente
# me permet de voir que votre fonction à l'air correcte, c'est le cas si
 ↪  cette cellule ne renvoie pas d'erreur
valeurs = [200, 100, 50, 20, 10, 5, 2, 1]
assert [renduMonnaie(s) for s in range(100)] == [
    [],
    [1],
    [2],
    [2, 1],
    [2, 2],
    [5],
    [5, 1],
    [5, 2],
    [5, 2, 1],
    [5, 2, 2],
    [10],
    [10, 1],
    [10, 2],
    [10, 2, 1],
    [10, 2, 2],
    [10, 5],
    [10, 5, 1],
    [10, 5, 2],
    [10, 5, 2, 1],
    [10, 5, 2, 2],
    [20],
    [20, 1],
    [20, 2],
    [20, 2, 1],
    [20, 2, 2],
    [20, 5],
    [20, 5, 1],
    [20, 5, 2],
    [20, 5, 2, 1],
    [20, 5, 2, 2],
    [20, 10],
    [20, 10, 1],
```

```
    [20, 10, 2],
    [20, 10, 2, 1],
    [20, 10, 2, 2],
    [20, 10, 5],
    [20, 10, 5, 1],
    [20, 10, 5, 2],
    [20, 10, 5, 2, 1],
    [20, 10, 5, 2, 2],
    [20, 20],
    [20, 20, 1],
    [20, 20, 2],
    [20, 20, 2, 1],
    [20, 20, 2, 2],
    [20, 20, 5],
    [20, 20, 5, 1],
    [20, 20, 5, 2],
    [20, 20, 5, 2, 1],
    [20, 20, 5, 2, 2],
    [50],
    [50, 1],
    [50, 2],
    [50, 2, 1],
    [50, 2, 2],
    [50, 5],
    [50, 5, 1],
    [50, 5, 2],
    [50, 5, 2, 1],
    [50, 5, 2, 2],
    [50, 10],
    [50, 10, 1],
    [50, 10, 2],
    [50, 10, 2, 1],
    [50, 10, 2, 2],
    [50, 10, 5],
    [50, 10, 5, 1],
    [50, 10, 5, 2],
    [50, 10, 5, 2, 1],
    [50, 10, 5, 2, 2],
    [50, 20],
    [50, 20, 1],
    [50, 20, 2],
    [50, 20, 2, 1],
    [50, 20, 2, 2],
```

```
    [50, 20, 5],
    [50, 20, 5, 1],
    [50, 20, 5, 2],
    [50, 20, 5, 2, 1],
    [50, 20, 5, 2, 2],
    [50, 20, 10],
    [50, 20, 10, 1],
    [50, 20, 10, 2],
    [50, 20, 10, 2, 1],
    [50, 20, 10, 2, 2],
    [50, 20, 10, 5],
    [50, 20, 10, 5, 1],
    [50, 20, 10, 5, 2],
    [50, 20, 10, 5, 2, 1],
    [50, 20, 10, 5, 2, 2],
    [50, 20, 20],
    [50, 20, 20, 1],
    [50, 20, 20, 2],
    [50, 20, 20, 2, 1],
    [50, 20, 20, 2, 2],
    [50, 20, 20, 5],
    [50, 20, 20, 5, 1],
    [50, 20, 20, 5, 2],
    [50, 20, 20, 5, 2, 1],
    [50, 20, 20, 5, 2, 2],
]
```

Question 2

Tester l'algorithme glouton précédent sur le système monétaire `valeurs=[10,6,1]` pour rendre une somme égale à 12.

```
# Jeu de test
```

Est-ce la solution optimale ? Que peut-on en déduire ?

2- Le problème du sac à dos

Le problème du sac à dos, noté également KP (en anglais, Knapsack Problem) est un problème d'optimisation combinatoire. On dispose d'un sac à dos pouvant supporter un poids maximal donné

et de divers objets ayant chacun une valeur et un poids. Il s'agit de choisir les objets à emporter dans le sac afin d'obtenir la valeur totale la plus grande tout en respectant la contrainte du poids maximal.

Arsène Lupin a devant lui 7 objets dont les valeurs et masses sont indiquées dans le tableau ci-dessous. Il dispose d'un sac à dos de capacité maximale 10 kg. Il souhaite emporter dans son sac à dos les objets dont la valeurs est la plus grande possible tout en ne dépassant pas la capacité maximale du sac à dos.

Objet	A	B	C	D	E	F	G
valeur	10 €	11 €	23 €	9 €	30 €	6 €	8 €
masse	2 kg	4,5 kg	5 kg	4 kg	9 kg	1 kg	2,5 kg

Structure de données adaptée au problème du sac à dos

On peut donner une liste de noms, de valeurs et de masses. Ces trois variables resteront inchangées donc on pourra là aussi les considérer comme variables globales (elles seront utilisées dans les fonctions mais n'apparaitrons pas dans les arguments des fonctions).

Question 3

Exécuter la cellule suivante pour utiliser les variables globales `noms`, `valeurs` et `masses` qui serviront pour toute la suite.

```
noms = ["A", "B", "C", "D", "E", "F", "G"]
valeurs = [10, 11, 23, 9, 30, 6, 8]
masses = [2, 4.5, 5, 4, 9, 1, 2.5]
```

Question 4

Compléter la fonction ci-dessous qui prend comme arguments une liste de noms d'objets, et renvoie la valeur de ces objets. Elle devra renvoyer -1 si jamais la masse dépasse la masse maximale. On vérifiera cette fonction avec les résultats des trois premières questions de l'exercice 4 du cours (questions 1, 2 et 3).

```
def valeur(objets, masseMax):
    """In : une liste d'objets et un entier masseMax
    Out: valeur totale des objets de la liste objets
         si la masse des objets ne dépasse pas masseMax
         -1 sinon"""
```

```
    valeur = 0
    masse = 0
    for x in objets:
        i = noms.index(x)  # indice de l'objet x
        valeur = ...
        masse = ...
    if masse <= masseMax:
        return ...
    else:
        return ...

# Jeu de tests (question 1 de l'ex 4 du cours)

# Jeu de tests (question 2 de l'ex 4 du cours)

# Jeu de tests (question 3 de l'ex 4 du cours)

# Jeu de tests (question 3 de l'ex 4 du cours)

# Test de la fonction précédente
# me permet de voir que votre fonction à l'air correcte, c'est le cas si
 ↪  cette cellule ne renvoie pas d'erreur
noms = ["A", "B", "C", "D", "E", "F", "G"]
valeurs = [10, 11, 23, 9, 30, 6, 8]
masses = [2, 4.5, 5, 4, 9, 1, 2.5]
assert valeur(["A", "B", "D"], 10) == -1
assert valeur(["A", "D", "F", "G"], 10) == 33
assert valeur(["A", "C", "F"], 10) == 39
assert valeur(["A", "C", "G"], 10) == 41
```

On choisit la stratégie gloutonne suivante : Tant qu'il reste un objet transportable : - prendre l'objet qui a la plus grande valeur massique $\mu = \frac{valeur}{masse}$

Pour appliquer cette stratégie, on va devoir trier par ordre décroissant les valeurs massiques. Mais pour ce faire, utilisons une fonction qui renvoie la liste des dictionnaires dont les clés sont le nom, la valeur, la masse et la valeur massique de chaque objet, triée par valeur massique décroissante.

Question 5

Compléter la fonction ci-dessous puis la tester. Elle doit renvoyer une liste triée (par ordre décroissant des valeurs massiques) des objets (donnés sous forme de dictionnaires).

On vérifera que cette liste est correctement triée et que les valeurs massiques correspondent au tableau suivant :

Objet	A	B	C	D	E	F	G
valeur	10 €	11 €	23 €	9 €	30 €	6 €	8 €
masse	2 kg	4,5 kg	5 kg	4 kg	9 kg	1 kg	2,5 kg
valeur massique	5€/kg	2,4€/kg	4,6€/kg	2,25€/kg	3,3€/kg	6 €/kg	3,2€/kg

```
def listeDictionnaires():
    """Out: liste de dictionnaires dont les clés sont
    nom, valeur, masse et valeurMassique"""
    n = len(noms)  # nb d'objets
    L = []
    for i in range(n):
        d = dict()
        d["nom"] = ...
        d["valeur"] = ...
        d["masse"] = ...
        d["valeurMassique"] = ...
        L.append(...)
    L.sort(key=lambda d: ..., reverse=True)
    return L

# Test
listeDictionnaires()

# Test en donnant chaque objet par ordre décroissant des valeurs massiques:
for d in listeDictionnaires():
    print(d)

# Test de la fonction précédente
# me permet de voir que votre fonction à l'air correcte, c'est le cas si
 ↪  cette cellule ne renvoie pas d'erreur
noms = ["A", "B", "C", "D", "E", "F", "G"]
valeurs = [10, 11, 23, 9, 30, 6, 8]
masses = [2, 4.5, 5, 4, 9, 1, 2.5]
assert listeDictionnaires() == [
    {"nom": "F", "valeur": 6, "masse": 1, "valeurMassique": 6.0},
    {"nom": "A", "valeur": 10, "masse": 2, "valeurMassique": 5.0},
    {"nom": "C", "valeur": 23, "masse": 5, "valeurMassique": 4.6},
    {"nom": "E", "valeur": 30, "masse": 9, "valeurMassique":
 ↪  3.3333333333333335},
    {"nom": "G", "valeur": 8, "masse": 2.5, "valeurMassique": 3.2},
    {"nom": "B", "valeur": 11, "masse": 4.5, "valeurMassique":
 ↪  2.4444444444444446},
```

```
    {"nom": "D", "valeur": 9, "masse": 4, "valeurMassique": 2.25},
]
```

Question 6

Compléter la fonction suivante qui renvoie la liste des noms des objets à prendre selon l'algorithme glouton. Vérifier cette fonction avec le résultat de la question 3 de l'exercice 5.

```
def sacAdosGlouton(masseMax):
    """In : un entier masseMax
    Out: la liste des objets renvoyée par l'algo glouton"""
    n = len(masses)  # nb d'objets
    L = listeDictionnaires()
    masseDuSac = 0
    objetsAprendre = []
    i = 0
    while masseDuSac + L[i]["masse"] <= masseMax:
        objetsAprendre.append(...)
        masseDuSac = ...
        i = ...
    return objetsAprendre

# Test

# Test qui donne la valeur des objets renvoyée par l'algorithme glouton

# Test de la fonction précédente
# me permet de voir que votre fonction à l'air correcte, c'est le cas si
 ↪  cette cellule ne renvoie pas d'erreur
noms = ["A", "B", "C", "D", "E", "F", "G"]
valeurs = [10, 11, 23, 9, 30, 6, 8]
masses = [2, 4.5, 5, 4, 9, 1, 2.5]
assert [sacAdosGlouton(x) for x in range(20)] == [
    [],
    ["F"],
    ["F"],
    ["F", "A"],
    ["F", "A"],
    ["F", "A"],
    ["F", "A"],
    ["F", "A"],
    ["F", "A", "C"],
    ["F", "A", "C"],
```

```
    ["F", "A", "C"],
    ["F", "A", "C"],
    ["F", "A", "C"],
    ["F", "A", "C"],
    ["F", "A", "C"],
    ["F", "A", "C"],
    ["F", "A", "C"],
    ["F", "A", "C", "E"],
    ["F", "A", "C", "E"],
    ["F", "A", "C", "E"],
]
```

Question 7

L'algorithme glouton est-il optimal pour le problème du sac à dos ?

Question 8

Utiliser les fonctions précédentes pour savoir ce que renvoie l'algorithme glouton si le sac à dos a une capacité maximale de 17 kg ?

```
# Test

# Test qui donne la valeur des objets renvoyée par l'algorithme glouton
```

Question 9

La fonction `sacAdosForceBrut` ci-dessous teste tous les cas et renvoie la liste des objets de valeur maximale (on ne vous demande pas de comprendre le code).

On reste avec un sac à dos de capacité maximale 17kg.

Tester la fonction `sacAdosForceBrut` pour savoir si l'algorithme glouton a renvoyé ou non la valeur maximale.

Pour info, le module `itertools` met à disposisition des fonctions (ici `product`) qui permettent de faire en une fois plusieurs boucles (ici 7 pour tester tous les cas), cela facilite le code (imaginez 7 boucles `for` imbriquées) mais ne rend pas plus rapide l'exécution. Il y a, comme on l'a vu $2^7 = 128$ tests à effectuer. Donc on évitera de tester cette fonction avec 100 objets !

```
from itertools import product
```

```python
def listeDobjets(nUpletsDobjets):  # fonction qui sert à sacAdosForceBrut
    """In : un n uplets de 0 et 1, une liste de noms
    Out: une liste des noms d'objets correspondant aux 1"""
    objets = []
    for i in range(len(nUpletsDobjets)):
        if nUpletsDobjets[i] == 1:
            objets.append(noms[i])
    return objets


def objetsValide(nUpletsDobjets, masseMax):  # fonction qui sert à
↪   sacAdosForceBrut
    """In : un n uplets de 0 et 1, une liste de masses, un entier masseMax
    Out: vraie si tous les objets associés à 1 peuvent être mis dans le sac"""
    masse = 0
    for i in range(len(nUpletsDobjets)):
        if nUpletsDobjets[i] == 1:
            masse += masses[i]
    return masse <= masseMax


def valeurBis(nUpletsDobjets):  # fonction qui sert à sacAdosForceBrut
    """In : un n uplets de 0 et 1, une liste de valeurs
    Out: la valeur totale des objets asssociés aux 1"""
    val = 0
    for i in range(len(nUpletsDobjets)):
        if nUpletsDobjets[i] == 1:
            val += valeurs[i]
    return val


def sacAdosForceBrut(masseMax):
    """In : une liste de masses, une liste de valeurs, un entier masseMax
    Out: la liste des objets de valeur maximale en testant tous les cas"""
    n = len(masses)  # nb d'objets
    objets = []
    valeur_max = 0
    for p in product(
        [0, 1], repeat=n
    ):  # pour chaque élt p du produit cartésien [0,1]^n
        if objetsValide(p, masseMax):
            val = valeurBis(p)
```

```
            if val > valeur_max:
                valeur_max = val
                objets = listeDobjets(p)
    return objets

# Jeu de test

# Test qui donne la valeur des objets renvoyée par l'algorithme glouton
```

Question 10

En utilisant les fonctions précédentes, dire si l'algorithme glouton renvoie la valeur optimale pour une capacité maximale de 40 kg avec les six objets suivants :

Objet	A	B	C	D	E	F
valeur	700 €	500 €	200 €	300 €	600 €	800 €
masse	13 kg	12 kg	8 kg	10 kg	14 kg	18 kg

```
# Test de l'algorithme glouton

# Test qui donne la valeur des objets renvoyée par l'algorithme glouton

# Jeu de test de force brute

# Test qui donne la valeur des objets renvoyée par l'algorithme force brute
```

3- Le problème du voyageur de commerce

- En informatique, le problème du voyageur de commerce, est un des problèmes d'optimisation les plus connus.
- C'est déjà sous forme de jeu que William Rowan Hamilton a posé pour la première fois ce problème, dès 1859.
- Sous sa forme la plus classique, son énoncé est le suivant : « Un voyageur de commerce doit visiter une et une seule fois un nombre fini de villes et revenir à son point d'origine. Il s'agit de déterminer l'ordre de visite des villes qui minimise la distance totale parcourue par le voyageur ».
- Voici une situation concrète. Nous partons de Nancy et devons nous rendre à Metz, à Paris, à Reims et à Troyes avant de retourner à Nancy. Le tableau des distances routières kilométriques

entre ces différentes villes est le suivant :

	Nancy	Metz	Paris	Reims	Troyes
Nancy	0	55	303	188	183
Metz	55	0	306	176	203
Paris	303	306	0	142	153
Reims	188	176	142	0	123
Troyes	183	203	153	123	0

Une manière simple d'aborder le problème consiste à énumérer tous les ordres possibles, à calculer pour chacun la distance correspondante, pour sélectionner ensuite la plus petite. C'est ce qu'on appelle la force brute.

Question 11

Quelle distance correspond au trajet ci-dessous ?

Nancy -> Metz -> Paris -> Reims -> Troyes -> Nancy

Question 12

Proposer un autre trajet et calculer sa distance

Question 13

1. Donner un trajet qui vous semblerait le plus court et calculer sa distance

Question 14

Combien y a-t-il de trajets au total ?

Question 15

Plus généralement, pour *n* villes, combien y aurait-il de trajets à examiner ?

Question 16

On se place dans le cas particulier de $n = 20$ villes. En supposant que l'ordinateur mette 0,001 seconde pour examiner un trajet, combien de temps faudrait-il attendre pour examiner tous les trajets ?

Appliquons l'approche gloutonne à notre problème du voyageur de commerce. Le problème considéré est la visite de l'ensemble des villes, depuis une ville de départ déterminée et avec retour à cette ville. Une solution est donc n'importe quelle séquence passant par toutes les villes intermédiaires demandées, qu'on peut ramener à la succession de choix « quelle sera ma prochaine étape ? ».

Principe de l'algorithme glouton du voyageur de commerce

Question 17

Compléter : - Partant de la ville de départ, aller à la ville la plus . . . . . , - Puis aller à la ville la plus . . . . . de cette dernière parmi les villes non encore visitées, et ainsi de suite

Question 18

Quel va être le trajet que va renvoyer cet algorithme glouton et quelle est sa distance ?

Réponse à détailler par étapes de l'algorithme glouton.

Question 19

- Cette distance que renvoie l'algorithme glouton est-elle optimale ?
- Vous parait-elle acceptable ?

Structure de données adaptée au problème du voyageur de commerce

- On commence par numéroter les villes : 0 pour Nancy, 1 pour Metz, 2 pour Paris, 3 pour Reims et 4 pour Troyes
- On pourra donc créer la liste suivante qui permettra de faire l'association entre le numéro de la ville et la ville :

```
villes=['Nancy','Metz','Paris','Reims','Troyes']
```

- Ensuite, on stocke les distances entre chaque ville. Pour cela, on peut créer une liste de listes, comme ceci : l'association entre le numéro de la ville et la ville :

```
L=[
    [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]
]
```

- Ainsi, `L[i][j]` est la distance entre la ville numéro `i` et la ville numéro `j`.
- Un trajet sera une liste du numéro des villes. Par exemple, le trajet `[0,1,2,3,4,0]` correspond au trajet : Nancy -> Metz -> Paris -> Reims -> Troyes -> Nancy

Question 20

Exécuter la cellule suivante pour pouvoir utiliser les deux variables `villes` et `L`.

```
villes = ["Nancy", "Metz", "Paris", "Reims", "Troyes"]
L = [
    [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],
]
```

Question 21

Écrire une fonction `distanceTrajet` qui prend comme argument une liste `trajet` et qui renvoie la distance totale du trajet. On vérifiera cette fonction avec les résultats obtenus par les questions 11, 12 et 13.

```
def distanceTrajet(trajet):
    """In : une liste trajet
    Out: la distance totale du trajet"""
    pass

# Jeu de test correspondant à la question 11

# Jeu de test correspondant à la question 12

# Jeu de test correspondant à la question 13
```

```
# Test de la fonction précédente
# me permet de voir que votre fonction à l'air correcte, c'est le cas si
 ↪  cette cellule ne renvoie pas d'erreur
villes = ["Nancy", "Metz", "Paris", "Reims", "Troyes"]
L = [
    [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],
]
assert distanceTrajet([0, 1, 2, 3, 4, 0]) == 809
assert distanceTrajet([0, 3, 2, 4, 1, 0]) == 741
assert distanceTrajet([0, 4, 2, 3, 1, 0]) == 709
```

Question 22

Compléter la fonction `afficheDistanceTrajet` qui prend comme arguments une liste `trajet` et qui affiche le trajet ainsi que sa distance totale. On vérifiera également cette fonction avec les résultats obtenus par les questions 11, 12 et 13.

```
def afficheDistanceTrajet(trajet):
    """In : une liste trajet
    Act: affiche le trajet et sa distance totale"""
    for i in trajet:
        print(..., "->", end="")  # seule ligne à compléter
    print(distanceTrajet(trajet), "km")

# Jeu de test correspondant à la question 11

# Jeu de test correspondant à la question 12

# Jeu de test correspondant à la question 13
```

Question 23

- La fonction `plusProche` aura pour argument une liste de booléens visitees indiquant pour chaque ville (donnée par son numéro) si elle a déjà été visitée ou non.
- Par exemple, la liste `visitees=[True,True,False,False,False]` signifie que seules les villes numéros 0 et 1 (Nancy et Metz) ont été visitées.

- Compléter puis tester la fonction `plusProche` qui prend comme arguments un numéro ville et une liste de booléens `visitees` et qui renvoie le numéro de la ville la plus proche de `ville` parmi celles non encore visitées (on suppose qu'il reste encore des villes à visiter).
- Par exemple, `plusProche(1,[True,True,False,False,False])` devra renvoyer 3 correspondant à Reims. En effet, Reims est la ville la plus proche de la ville numéro 1 (Metz) parmi les villes non encore visitées.
- Pour cela, pour chaque ville non encore visitée, on va calculer la distance à `ville`, et on mémorisera la distance la plus courte ainsi que la ville la plus proche. Voilà pourquoi, on part d'une distance `dist_min` (ici 10000) assez grande qu'on va modifier peu à peu ainsi qu'une villePlusProche qu'on met à -1 mais que l'on modifiera également.

```
def plusProche(ville, visitees):
    """In : un numéro ville et une liste de booléens visitees
    Out: le numéro de la ville la + proche de ville
         parmi les villes non visitées"""
    dist_min = 100000  # on part d'une distance_min très grande
    villePlusProche = -1
    for i in range(len(visitees)):
        if not ...:
            d = L[...][...]
            if ...:
                ...
                ...
    return villePlusProche

# Jeu de tests

# Jeu de tests

# Test de la fonction précédente
# me permet de voir que votre fonction à l'air correcte, c'est le cas si
 ↪  cette cellule ne renvoie pas d'erreur
villes = ["Nancy", "Metz", "Paris", "Reims", "Troyes"]
L = [
    [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],
]
assert [plusProche(i, [True, True, False, False, False]) for i in range(5)]
 ↪  == [
    4,
```

```
    3,
    2,
    3,
    4,
]
assert [plusProche(i, [False, False, False, False, False]) for i in
 ↪  range(5)] == [
    0,
    1,
    2,
    3,
    4,
]
```

Question 24

On arrive enfin à la fonction principale: `voyageurGlouton`.

Compléter et tester cette fonction qui renvoie le trajet renvoyé par la méthode gloutonne.

```
def voyageurGlouton():
    """Out: le trajet renvoyé par l'algo glouton"""
    trajet = [0]
    n = len(L)
    visitees = [False for i in range(n)]
    ville = 0
    for i in range(n - 1):
        visitees[ville] = ...
        suivante = plusProche(..., ...)
        trajet.append(...)
        ville = ...
    trajet.append(0)
    return trajet

# Jeu de test

# affichage du trajet avec la fonction afficheDistanceTrajet

# Test de la fonction précédente
# me permet de voir que votre fonction à l'air correcte, c'est le cas si
 ↪  cette cellule ne renvoie pas d'erreur
villes = ["Nancy", "Metz", "Paris", "Reims", "Troyes"]
L = [
    [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],
]
assert voyageurGlouton() == [0, 1, 3, 4, 2, 0]
```

L'algorithme force brute

- Pour obtenir le trajet optimal en analysant toutes les possibilités (ce qui est faisable ici puisqu'il n'y a que 5 villes), il faudrait a priori 4 boucles emboîtées (correspondant aux numéros des 4 villes autres que Nancy) :

```
for i1 in range(1,5):
    for i2 in range(1,5):
        if i2!=i1:
            for i3 in range(1,5):
                if i3!=i1 and i3!=i2:
                    for i4 in range(1,5):
                        if i4!=i1 and i4!=i2 and i4!=i3:
```

- Les conditions `if i4!=i1 and i4!=i2 and i4!=i3:` sont là pour éviter d'avoir un trajet avec 2 fois la même ville (autre que Nancy).
- Plutôt que de faire ces 4 boucles emboîtées, on va utiliser la fonction permutations du module itertools. C'est un module que je vous conseille de retenir si un jour vous avez pleins de boucles emboîtées à faire.
- Après avoir importé la fonction `permutations` du module `itertools`, les 4 boucles emboîtées peuvent simplement se remplacer par l'instruction :

```
for i1,i2,i3,i4 in permutations([1,2,3,4]):
```

Question 25

Exécuter la cellule suivante pour tester la boucle et constater qu'on obtient toutes les permuations de l'ensemble {1,2,3,4} qui permettra de tester tous les trajets possibles.

```
from itertools import permutations

for i1, i2, i3, i4 in permutations([1, 2, 3, 4]):
    print(i1, i2, i3, i4)
```

Question 26

- Compléter puis tester la fonction ci-dessous pour qu'elle analyse tous les trajets et renvoie le trajet optimal.
- L'idée de cet algorithme est de partir d'un trajet `trajet_min` et d'une distance minimale `dist_min` et d'examiner toutes les possibilités de trajets en mettant à jour ces deux variables `trajet_min` et `dist_min` dès qu'on obtient un trajet plus court.

```
def forceBrute():
    """Out: le trajet de distance minimale"""
    trajet_min = [0, 1, 2, 3, 4, 0]
    dist_min = distanceTrajet(trajet_min)
    for i1, i2, i3, i4 in permutations([1, 2, 3, 4]):
        trajet = [0, i1, i2, i3, i4, 0]
        d = distanceTrajet(trajet)
        if d < dist_min:
            dist_min = ...
            trajet_min = ...
    return trajet_min

# Jeu de test

# affichage du trajet avec la fonction afficheDistanceTrajet
```

Question 27

Quelle est finalement le trajet optimale ?

Question 28

- En cherchant sur le WEB, déterminer les 8 villes françaises les plus peuplées puis déterminer leurs distances entre elles.
- Vous partez de Montpellier et vous devez vous rendre dans chacune de ces villes en revenant à Montpellier.
- Déterminer le trajet obtenu (et sa distance) par l'algorithme glouton.

```
# liste des villes et listes des distances

# affichage du trajet avec la fonction afficheDistanceTrajet
```

Question 29

Adapater la fonction `forceBrute` à 8 villes, au lieu de 5 puis donner le trajet minimale et le comparer avec celui renvoyé par l'algorithme glouton.

```
def forceBrute():
    """Out: le trajet de distance minimale"""
    pass

# affichage du trajet avec la fonction afficheDistanceTrajet
```