Première Algorithmique	Algorithmes gloutons	

Objectifs de la séance :

- *Découvrir des problèmes d'optimisation : solution optimale, solution approchée*
- *Proposer une optimisation locale par une approche gloutonne*
- *Étudier deux problèmes célèbres : le problème du voyageur de commerce, le problème du rendu de monnaie*

Matériel et Logiciel nécessaire : PC

- *un environnement de programmation Python*

Sources : Wikipédia et le livre « Numérique et sciences informatiques » édition ellipse

Durée estimée : 3 h

A - Description générale des algorithmes gloutons

Les algorithmes gloutons sont utilisés pour répondre à des problèmes d'optimisation, c'est-à-dire des problèmes algorithmiques dans lesquels l'objectif est de trouver une solution « la meilleure possible » selon un certain critère, parmi un ensemble de solutions également valides mais potentiellement moins avantageuses :

- on considère un problème possédant un très grand nombre de solutions ;
- on dispose d'une fonction mathématique évaluant la qualité de chaque solution ;
- on cherche une solution qui soit bonne, voir la meilleure.

Les algorithmes gloutons s'appliquent lorsque de plus :

- la recherche d'une solution peut se ramener à une succession de choix qui produisent et précisent petit à petit une solution partielle ;
- on dispose d'une fonction mathématique évaluant la qualité de chaque solution partielle.

L'approche gloutonne consiste alors à construire une solution complète par une succession de choix en prenant systématiquement l'option donnant la meilleure solution partielle.

B - Problème : rendu de monnaie

Le **problème du rendu de monnaie** est un problème d'algorithmique. Il s'énonce de la façon suivante : étant donné un système de monnaie (pièces et billets), comment rendre une somme donnée de façon optimale, c'est-à-dire avec le nombre minimal de pièces et billets ?
Par exemple, la meilleure façon de rendre 9 euros est de rendre un billet de cinq et deux pièces de deux, même si d'autres façons existent :

Combinaison	*Nombre de valeurs*
9 x 1 €	9
7 x 1 € + 1 x 2 €	8
...	...
...	...
...	...
...	...
...	...
2 x 2 € + 1 x 5 €	3

Ce problème est NP-complet dans le cas général, c'est-à-dire difficile à résoudre. Cependant pour certains systèmes de monnaie dits canoniques, l'algorithme glouton est optimal, c'est-à-dire qu'il suffit de rendre systématiquement la pièce ou le billet de valeur maximale — ce tant qu'il reste quelque chose à

rendre. C'est la méthode employée en pratique, ce qui se justifie car la quasi-totalité des systèmes ayant cours dans le monde sont canoniques. Il n'existe pas, à ce jour, de caractérisation générale des systèmes canoniques, mais il existe une méthode efficace pour déterminer si un système donné est canonique.

Exercice 1

1. Compléter le tableau ci-dessus.
2. Le programme suivant (à compléter), définit une fonction Python calculant le nombre de pièces ou billets et leur valeurs qui doivent être utilisées pour une somme à rendre donnée, en utilisant la stratégie gloutonne. Jusqu'à ce que la somme totale ait été rendue, cette fonction considère tour à tour les différentes coupures de la plus grosse à la plus petite, identifiées par un indice `i` dans le tableau `euros`. Tant que la somme à rendre est supérieure à la coupure courante, on choisit de l'utiliser. Dès que la somme à rendre devient inférieure à la valeur de la coupure `euro[i]`, on poursuit avec la coupure immédiatement inférieure. On remarque que, la somme à rendre ne faisant que décroître, une coupure écartée à cette étape ne sera effectivement plus jamais utile.

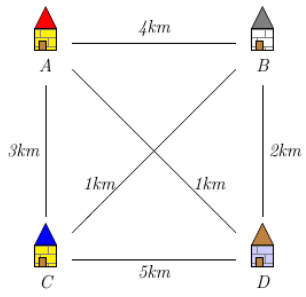
```
euros = [1, 2, 5, 10, 20, 50, 100, 200]

def rendu_monnaie(somme : int) -> list:
    """retourne une liste de tuples (v, n) où v est la valeur
    des pièces ou billets à rendre et n le nombre"""
    i = ...
    a_rendre = []
    n = 0
    monnaie = None
    while somme > 0:
        if somme >= euros[i]:
            n = ...
            monnaie = ...
            somme = ...
        else:
            n = ...
            i = ...
            if monnaie != None:
                ...
                monnaie = ...
    ...
    return a_rendre
```

C - Problème d'optimisation : le voyageur de commerce

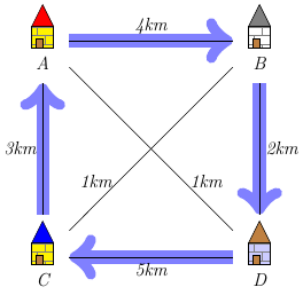
L'énoncé du *problème du voyageur de commerce* est le suivant : étant donné n points (des « villes ») et les distances séparant chaque point, trouver un chemin de longueur totale minimale qui passe exactement une fois par chaque point et revienne au point de départ.

C.1-Un exemple

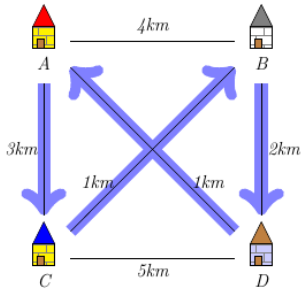
On considère la liste des villes A, B, C, D et les distances données par le dessin ci-dessous à gauche. Un premier chemin qui part de A, revient en A et qui visite toutes les villes est ABDCA. Un chemin plus court est ACBDA. Le chemin ACBDA est optimal.



Instance du problème du voyageur de commerce



Le chemin ABDCA de longueur :
4 + 2 + 5 + 3 = 14 km.



Le chemin ACBDA est le chemin le plus court qui part de A, revient A et passe par toutes les villes. Il est de longueur :
3 + 1 + 2 + 1 = 7 km.

C.2-Explosion combinatoire

Ce problème est plus compliqué qu'il n'y paraît ; on ne connaît pas de méthode de résolution permettant d'obtenir des solutions exactes en un temps raisonnable pour de grandes instances (grand nombre de villes) du problème. Pour ces grandes instances, on devra donc souvent se contenter de solutions *approchées*, car on se retrouve face à une explosion combinatoire.

Pour un ensemble de points, il existe au total chemins possibles $1\times2\times3\times...\times n=n!$. Le point de départ ne changeant pas la longueur du chemin, on peut choisir celui-ci de façon arbitraire, on a ainsi $(n-1)!$ chemins différents. Enfin, chaque chemin pouvant être parcouru dans deux sens et les deux possibilités ayant la même longueur, on peut diviser ce nombre par deux. Par exemple, si on nomme les points a, b, c, d les chemins abcd et dcba, cdab et badc, adcb et bcda, cbad et dabc ont tous la même longueur, seul le point de départ et le sens de parcours change.

On a donc $\frac{1}{2}(n-1)!$ chemins candidats à considérer.

Par exemple, pour 71 villes, le nombre de chemins candidats est supérieur à 5×10^{80} qui est environ le nombre d'atomes dans l'univers connu.

C.3-Une autre situation

Voici le tableau des distances routière entre Nancy, Metz, Paris, Reims et Troyes.

	Nancy	Metz	Paris	Reims	Troyes
Nancy	0	55	303	188	183
Metz	55	0	306	176	206
Paris	303	306	0	142	153
Reims	188	176	142	0	123
Troyes	183	203	153	123	0

Si l'on cherche à énumérer, partant par exemple de Nancy, tous les ordres possible, nous en trouverons 12 différents (et 12 autre dans le sens inverse mais avec les mêmes distances) et le meilleur est :

Nancy – Metz – Reims – Paris – Troyes avec 709 km

Exercice 2

1. Appliquer la méthode gloutonne à ce problème : partant de la ville de départ, aller à la ville la plus proche, puis à la ville la plus proche de cette dernière parmi les villes non encore visités et ainsi de suite. On parle de la *méthode du plus proche voisin*.

   Remarque : L'itinéraire obtenu est plus long que le circuit minimal de 709 km mais parmi les trois meilleurs solutions.

2. Les n villes à visiter sont numérotées de 0 à $n-1$ et les distances entre les villes sont stockées dans un tableau `dist` à deux dimensions tel que `dist[i][j]` donne la distance de la ville numéro `i` à la ville numéro `j`. Le tableau `villes` contient le nom des villes : l'indice de chaque nom dans le tableau est le numéro de la ville.

   Écrire les tableaux `dist` et `villes` associé à cet exemple.

3. Commencer par écrire la fonction `plus_proche` qui prend en paramètres la table des distances, le numéro de la ville courante et le tableau des villes visitées pour sélectionner la ville la plus proche de la `ville` courante parmi les villes non encore visitées. Elle utilise un tableau `visitees` de booléens indiquant pour chaque ville (donnée par son numéro) si elle a déjà été visité ou non.

```
def plus_proche(ville : int, dist : tuple, visitees : tuple) -> int:
    """Retourne le numéro de la ville non encore listée la plus
    Proche,  en supposant qu'il en existe au moins une."""
    pp = None
    for i un range(len(visitees)):
        if not visitees[i]:
            ...
    ...
```

4. Compléter la fonction principale `voyageur` qui prend en paramètres la liste des villes, la table des distances et le numéro de la ville de départ. Elle utilise une variable `courante` mémorisant le numéro de la ville actuelle.

```
def voyageur(villes : tuple, dist : tuple, depart : int) -> list:
    """Retourne l'ordre des villes à visité et
    en fin de tableau la distance de ce parcours"""
    n = len(villes)
    visitees = ...
    courante = ...
    distance =...
    trajet = ...
    for _ in range(n-1):
        visitees[courante] = ...
        suivante = plus_proche(courante, dist, visitees)
        ...
   ...
```

D - Problème du sac à dos

Arsène L. a devant lui un ensemble d'objets de valeurs et de poids variés. Il dispose d'un sac à dos dans lequel prendre une partie des objets, en essayant de maximiser la valeur totale emportée. Cependant, il ne pourra emporter le sac que si le poids total ne dépasse pas 10 kilogrammes.

D.1- Quelques situations

Exercice 3

Dans chacune des situations suivantes, indiquer les différentes combinaisons qui peuvent être formées et les valeurs correspondantes.

a)

	kg	€
A	8	4800
B	5	4000
C	4	3000
D	1	500

b)

	kg	€
A	6	4800
B	5	3500
C	4	3000
D	1	500

c)

	kg	€
A	9	8100
B	6	7200
C	5	5500
D	4	4000
E	1	800

d)

	kg	€
A	7	9100
B	6	7200
C	4	4800
D	3	2700
E	2	2600
F	1	200

D.2- Algorithmes gloutons

Exercice 4

Arsène L. ne voulant pas arriver en retard à son rendez vous avec la comtesse C. va devoir choisir très rapidement les objets à emporter.

1. Appliquer chacune des stratégies gloutonnes suivantes aux situation du paragraphe précédent :
   a) Choisir les objets par ordre de valeur décroissante parmi ceux qui ne dépasse pas la capacité restante.
   b) Choisir les objets par ordre de poids croissant.
   c) Choisir les objets par ordre de rapport valeur/poids décroissant parmi ceux qui ne dépasse pas la capacité restante.
2. Pour chacune des stratégies gloutonnes précédentes, trouver des situations dans lesquelles la valeur emportée est aussi éloignée que possible de la valeur optimale.