1NSI	Algorithmique Algorithmes gloutons	TP – **Les algorithmes gloutons**

Objectifs :

- ⇨ Introduire la classe des problèmes d'optimisation
- ⇨ Prendre conscience que trouver une solution optimale à certains problèmes peut être impossible en pratique
- ⇨ Découvrir la méthode gloutonne et la mettre en œuvre sur différents problèmes.

Le glouton est un mammifère de la famille des mustélidés

I – Les algorithmes d'optimisation

Un **problème d'optimisation** est un problème pour lequel on cherche la meilleure solution (selon un critère défini) dans un ensemble de solutions possibles.

Exemples :
Détermination du meilleur itinéraire
Répartition optimale de tâches suivant des critères précis (emploi du temps avec plusieurs contraintes)
Rendu de monnaie
Ranger des éléments : colis dans un camion, sac à dos, etc.

On parle de **solution optimale** pour une solution qui fait partie des solutions possibles mais qui est la meilleure des solutions selon le critère défini.

Les algorithmes gloutons que nous allons détailler et mettre en œuvre ici sont un exemple d'algorithmes d'optimisation.

II – Le problème du voyageur de commerce

Le problème du voyageur de commerce est un grand classique des problèmes algorithmiques. Voici son énoncé :
Supposons que l'on ait déterminé une certaine liste de villes dans lesquelles nous devons nous rendre et que l'on cherche un itinéraire minimisant la distance parcourue. On peut visiter les différentes villes dans n'importe quel ordre, mais il faut absolument passer par toutes les villes et on doit à la fin revenir à la ville de départ.

Prenons un exemple concret :
Nous partons de Nancy et devons nous rendre à Metz, Paris, Reims et Troyes avant de retourner à Nancy. Le tableau des distances kilométriques entre ces villes est donné ci-contre.

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

Distances entre les villes en km d'après geoportail.gouv.fr

On peut ramener le problème à trouver l'ordre de visite des villes de Metz, Paris, Reims et Troyes qui minimise la distance totale parcourue.

On peut montrer en mathématique qu'il y a 4![1] = 24 permutations possibles entre ces 4 villes :

Itinéraire	Calcul de la longueur du trajet	Total
Metz, Paris, Reims, Troyes	55 + 306 + 142 + 123 + 183	809
Metz, Paris, Troyes, Reims	55 + 306 + 153 + 123 + 188	825
Metz, Reims, Paris, Troyes	55 + 176 + 142 + 153 + 183	**709**
Metz, Reims, Troyes, Paris	55 + 176 + 123 + 153 + 303	810
Metz, Troyes, Reims, Paris	55 + 203 + 123 + 142 + 303	826
Metz, Troyes, Paris, Reims	55 + 203 + 153 +142 + 188	741
Troyes, Metz, Paris, Reims	183 + 203 + 306 + 142 + 188	1022
Troyes, Metz, Reims, Paris	183 + 203 + 176 + 142 + 303	1007
Troyes, Reims, Metz, Paris	183 + 123 + 176 + 306 + 303	1091
Reims, Metz, Paris, Troyes	188 + 176 + 306 + 153 + 183	1006
Reims, Metz, Troyes, Paris	188 + 176 + 203 + 153 + 303	1023
Reims, Troyes, Metz, Paris	188 + 123 + 203 + 306 + 303	1123
Les 12 lignes suivantes sont les même que les précédentes mais en ordre inverse : les longueurs de trajets sont donc les mêmes et on ne les étudiera pas.		

On s'aperçoit que le trajet optimum fait 709 km et fait Nancy, Metz, Reims, Paris, Troyes et revient à Nancy.

Pour trouver le meilleur trajet, on a donc examiné tous les trajets (ou plutôt la moitié à cause de la symétrie). Cela peut sembler assez simple car on a travaillé avec seulement 4 villes. Si nous doublons et prenons 8 villes, on aura alors 8! = 1×2×3×4×5×6×7×8 = 40320 permutations soit 20160 possibilités à examiner. Avec 15 villes, c'est 654 milliards de possibilités qu'il faut analyser…
On voit donc que lorsque le nombre de villes augmente trop, on ne peut pas envisager d'examiner tous les circuits possibles car ils sont trop nombreux et cela prendrait trop de temps.
Cela n'est pas dû au fait qu'on a choisi une approche assez directe. En effet dans l'état actuel des connaissances mathématiques, il n'existe pas d'algorithme donnant la solution à ce problème en un temps raisonnable.
On va donc faire appel à un algorithme capable de trouver une solution *assez bonne* au problème sans qu'elle soit nécessairement la meilleure. C'est ce que font les applications de guidage routier qui doivent trouver un itinéraire parmi des milliers ou même des millions de routes en un temps assez court : elles n'ont pas le temps de passer en revue toutes les possibilités pour s'assurer que leur chemin est le meilleur.

III – Algorithmes gloutons

On appelle algorithmes gloutons les algorithmes qui tentent de déterminer la meilleure solution globale à un problème en effectuant à chaque étape le choix qui semble le meilleur (le nom algorithme glouton[2] s'explique par le fait que, comme un gourmand, on s'occupe de prendre ce qui semble le meilleur sur le moment sans avoir de vision à long terme).

Cette méthode est simple à mettre en œuvre et donne souvent d'assez bons résultats, mais il existe des cas où la solution proposée par l'algorithme est loin d'être la meilleure globalement.

On a recours à un algorithme glouton dans les cas où :
- le problème à solutionner possède un très grand nombre de solutions,
- on dispose d'une fonction mathématique permettant d'évaluer la qualité de chaque solution,
- on cherche une solution qui soit bonne sans nécessairement être la meilleure.

[1] n! se lit « n factorielle » ou « factorielle n » et vaut 1×2×3×4×…×n donc 4! = 1×2×3×4 = 24
[2] En anglais on utilise le terme « greedy » qui signifie « avide ».

Pour qu'un algorithme glouton puisse être utilisé il faut de plus :

- que la recherche d'une solution puisse se faire petit à petit par une succession de choix qui construisent chacun des solutions partielles jusqu'à la solution finale ;
- qu'on dispose également d'une fonction mathématique permettant d'évaluer la qualité des solutions partielles (et pas uniquement de la solution finale).

Dans le cas de notre problème du voyageur de commerce, un algorithme glouton serait en partant de Nancy de prendre à chaque étape la ville non visitée la plus proche :

- Etape 1 :

Au départ de Nancy, c'est Metz qui est la plus proche. La solution partielle est donc Nancy, Metz et fait 55 km.

- Etape 2 :

Puis à partir de Metz il nous reste Paris, Reims et Troyes à visiter. Parmi ces 3 villes, c'est Reims qui est la plus proche. La solution partielle est donc à cette étape Nancy, Metz, Reims et fait 231 km.

- Etape 3 :

A partir de Reims il nous reste Paris et Troyes à visiter. Entre ces deux destinations, c'est Troyes qui est la plus proche. La solution partielle est donc à cette étape Nancy, Metz, Reims, Troyes et fait 354 km.

- Etape 4 :

Il ne reste plus que Paris, on l'ajoute donc d'office à notre circuit. La solution partielle est donc : Nancy, Metz, Reims, Troyes, Paris et fait 507 km.

- Etape 5 :

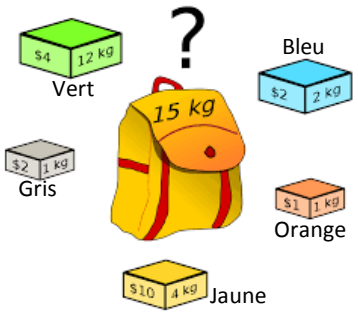
Il reste juste à rajouter le retour à Nancy pour obtenir le circuit Nancy, Metz, Reims, Troyes, Paris, Nancy qui fait donc 810 km.

L'itinéraire obtenu (810 km) n'est donc pas le plus court, mais il n'est pas très loin du meilleur et il est par contre assez loin des pires solutions qui dépassent les 1000 km. L'important c'est que son évaluation est très rapide et reste raisonnable même si le nombre de villes augmente fortement.

IV – Première application : le problème du sac à dos

1) Problème

On souhaite remplir un sac à dos avec des objets dont la valeur est maximale mais dont la masse totale total doit rester inférieure à la masse maximale autorisés.
Le sac à dos peut contenir 15 kg et on peut y placer les boites ci-contre (chacune est disponible en autant d'exemplaires qu'on le souhaite).

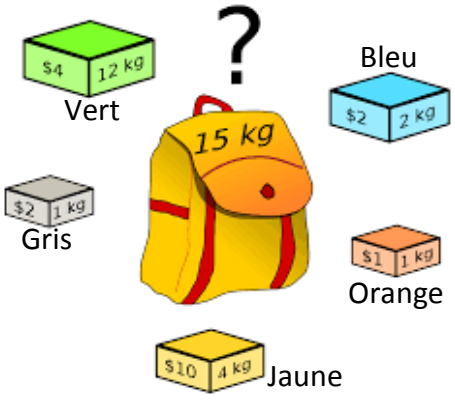


2) Algorithme

Question 1 : Algorithme glouton
Trouver et décrire (en français) un algorithme glouton le plus simple possible qui permette de traiter le problème. Après validation par le professeur, utiliser votre algorithme pour résoudre « à la main » les deux cas présentés ci-après.

Aide : En principe 2-3 phrases suffisent à décrire l'algorithme.

Cas favorable :

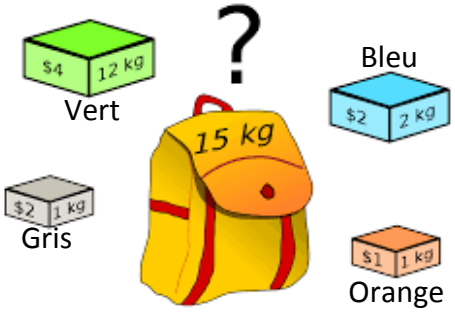


Couleur de la boîte	Masse totale (en kg)	Valeur totale (en $)

Dans ce cas particulier, on voit que l'algorithme glouton trouve sans difficulté la meilleure solution.

Mais voyons ce qui se passe si on supprime la boîte jaune…

Cas défavorable :



Couleur de la boîte	Masse totale (en kg)	Valeur totale (en $)

On voit sans peine que cette fois-ci la solution est loin d'être optimale car en ne mettant que des boîtes grises dans le sac on aurait eu 15 boîtes pour un total de 30 $...

3) Programmation de l'algorithme en python

Question 2 : Programmation de l'algorithme glouton
Ecrire une implémentation de votre algorithme en python. Vous pouvez utiliser comme base le programme « Question2.py » si vous souhaitez utiliser un dictionnaire pour représenter les boîtes disponibles ou « Question2_tableaux.py » si vous souhaitez utiliser des tableaux.

Aide : Si vous avez vraiment du mal, vous pouvez utiliser comme base le fichier « Question2_Trame.py ».

V – Problème du rendu de monnaie

On s'intéresse au problème suivant : étant donné un montant de monnaie à rendre à un client, comment lui rendre la monnaie en utilisant le moins de pièces et de billets possible.

On procédera avec un algorithme glouton, consistant à sélectionner à chaque étape la pièce ou le billet de plus grande valeur possible qui ne fait pas dépasser le montant à rendre au client.

On utilisera la constante VALEURS = [500, 200, 100, 50, 20, 10, 5, 2, 1, 0.50, 0.20, 0.10, 0.05, 0.02, 0.01] comme liste décroissantes des valeurs de pièces et billets disponibles.

1) Quantités disponibles illimités

Question 3 : Rendu de monnaie illimité

1) En appliquant un algorithme glouton, comment rendrait-on 0,30 € ?
2) Quelle est la valeur de vérité selon Python de `0.1 + 0.2 <= 0.3` ?
3) En faisant référence au cours sur la représentation des nombres décimaux, expliquer l'erreur commise par Python.
4) Ecrire deux fonctions :
   - « `conversion_centimes` » qui convertit des valeurs décimales (`float`) en euros en valeurs entières (`int`) en centimes d'euros
   - « `conversion_euros` » qui convertit des valeurs entières (`int`) en centimes d'euros en valeurs décimales (`float`) en euros

Dans la suite on utilisera ces deux fonctions aussi souvent que nécessaire afin d'effectuer des calculs exacts et d'éviter les problèmes similaires à celui qui a été identifié à la question 2.

5) Ecrire une fonction « `rendre_monnaie` » qui prend entrée le montant à rendre et qui renvoie en sortie un dictionnaire dont les clefs sont les valeurs de pièces ou billets utilisés et les valeurs le nombre de fois ou la pièce ou le billet est utilisé. Le nombre de pièces et billets rendus devra être minimisé selon un principe d'algorithme glouton. Si aucune solution n'est possible, la fonction doit renvoyer `None`.

2) Quantités disponibles limités

On considère maintenant le cas où on ne dispose pas d'un stock infini de pièces ou de billets de toutes les valeurs.
Pour représenter les valeurs de pièces disponibles et leur quantité, on va utiliser un dictionnaire caisse qui représente le stock de monnaie dans le tiroir caisse.
Les clés de ce dictionnaire sont les valeurs des pièces/billets et les valeurs du dictionnaire sont les quantités de chaque pièce.

Valeur	Nombre de pièces/billets
10	2
5	2
2	5
1	2
0.5	0
0.2	3
0.1	2
0.05	1
0.02	3
0.01	2

Exemple :
Représentation de la caisse donnée dans le tableau ci-contre.

```
caisse = {10:2, 5:2, 2:5, 1:2, 0.5:0, 0.2:3, 0.1:2, 0.05:1,
0.02:3, 0.01:2}
```

Question 4 : Rendu de monnaie limité
On va reprendre la question 5 de la partie précédente, avec la contrainte supplémentaire suivante :
La fonction `rendre_monnaie_disponible` prend en entrée le montant à rendre et un dictionnaire `caisse` comme celui décrit précédemment. Elle doit renvoyer :

- soit `None` si le rendu de monnaie est impossible. Dans ce cas le dictionnaire `caisse` doit rester inchangé.
- soit un dictionnaire donnant le rendu de monnaie minimisé par un algorithme glouton comme pour la question 3. Dans ce cas le dictionnaire caisse doit être modifié pour refléter le rendu de monnaie effectué.

Exemples :

- Rendu possible sur la somme de 27,83€ avec 1×20 + 1×5 + 1×2 + 3×0,2 + 2×0,1 + 1×0,02 + 1×0,01

```
Etat de la caisse avant rendu :
{20:3, 10:2, 5:2, 2:5, 1:2, 0.5:0, 0.2:3, 0.1:2, 0.05:1, 0.02:3, 0.01:2}
Rendu sur la somme de 27.83 :
{20: 1, 5: 1, 2: 1, 0.2: 3, 0.1: 2, 0.02: 1, 0.01: 1}
Etat de la caisse après rendu :
{20:2, 10:2, 5:1, 2:4, 1:2, 0.5:0, 0.2:0, 0.1:0, 0.05:1, 0.02:2, 0.01:1}
```

- Rendu impossible avec la même caisse sur 9,99€

```
Etat de la caisse avant rendu :
{20:3, 10:2, 5:2, 2:5, 1:2, 0.5:0, 0.2:3, 0.1:2, 0.05:1, 0.02:3, 0.01:2}
Rendu sur la somme de 9.99 :
None
Etat de la caisse après rendu :
{20:3, 10:2, 5:2, 2:5, 1:2, 0.5:0, 0.2:3, 0.1:2, 0.05:1, 0.02:3, 0.01:2}
```

3) Compléments

Suivant le système de monnaie utilisé, la méthode gloutonne peut donner de bons ou de mauvais résultats.
On peut montrer que pour le système monétaire français, la stratégie gloutonne donne toujours une solution optimale. Pour cette raison, un tel système de monnaie est qualifié de **canonique**.

D'autres systèmes ne sont pas canoniques : l'algorithme glouton ne répond pas toujours de manière optimale.

Par exemple, avec le système `{1, 3, 6, 12, 24, 30}`, l'algorithme glouton répond en proposant le rendu 49=30+12+6+1, soit 4 pièces alors que la solution optimale est 49=2×24+1, soit 3 pièces.