# TP 7 : Algorithmes gloutons

## I. Principe général

Un problème d'optimisation est un problème algorithmique dans lequel l'objectif est de trouver la « meilleure » solution, selon un critère donné, parmi un ensemble de solutions également valides mais potentiellement moins bonnes. Par exemple, déterminer le minimum ou le maximum d'une fonction à une certaine précision est un problème d'optimisation. On peut également citer la répartition optimale de tâches suivant des critères précis, le problème du rendu de monnaie, le problème du sac à dos, la recherche du plus court chemin dans un graphe, le problème du voyageur de commerce.

Le contexte d'un problème d'optimisation est donc :

***1)*** un très grand nombre de solutions (dans le cas contraire, il n'y aurait pas de difficulté à trouver la meilleure),

***2)*** une fonction permettant d'évaluer la qualité de chaque solution (parfois appelée *fonction de coût*),

***3)*** l'existence d'une solution optimale (ou suffisamment satisfaisante).

**Exemple** *(Problème du sac à dos)*
On dispose d'une collection d'objets dont on connaît les valeurs et poids respectifs. On cherche à remplir un sac à dos de manière optimale, sans dépasser le poids maximal qu'il peut contenir.

De nombreuses techniques informatiques sont susceptibles d'apporter une solution exacte ou approchée à ces problèmes. Certaines de ces techniques, comme l'énumération exhaustive de toutes les solutions (on parle de méthode par **force brute**), ont un coût machine qui les rend souvent peu pertinentes au regard de contraintes extérieures imposées (temps de réponse de la solution imposé, moyens machines limités).

Un **algorithme glouton** (**greedy algorithm** en anglais) est un algorithme qui suit le principe de faire, étape par étape, un choix optimal *local*, dans l'espoir d'obtenir un résultat optimal *global*, et de ne jamais revenir en arrière sur un choix déjà effectué.

**Exemple** *(Application au problème du sac à dos)*
La méthode gloutonne pour ce problème consiste à mettre dans le sac à dos, à chaque étape, l'objet ayant le meilleur ratio valeur/poids parmi les objets ayant un poids inférieur à ce qu'il est encore possible de mettre dans le sac à dos.

Nous allons étudier quelques situations où l'on peut mettre en œuvre un algorithme glouton. Nous verrons que :

× le principal avantage des algorithmes gloutons est leur facilité de mise en œuvre,

× leur principal défaut est qu'ils ne renvoient pas toujours une solution optimale.

# II. Problème du rendu de monnaie

Vous allez à la boulangerie acheter du pain. Vous payez, et attendez votre monnaire. Il est souhaitable que cela se fasse avec un minimum de pièces, aussi bien pour le vendeur que pour le client.
De façon générale, les commerçants rendent la monnaie par méthode *gloutonne* :

***1)*** ils choisissent la plus grande valeur de pièce inférieure ou égale au montant à rendre,

***2)*** ils rendent cette pièce puis recommencent avec le reste à rendre.

On représente un système monétaire par la liste, triée dans l'ordre croissant, des différentes valeurs en circulation. Par exemple, `[1, 2, 5, 10, 20, 50, 100]` représente les pièces de 1 centime à 1 euro.
Ainsi, par exemple, en partant d'un montant de $1,43$ soit 143 centimes :

× la plus grande valeur inférieure à 143 est 100, on prend donc 1 pièce de 1. Il reste 43 centimes.

× la plus grande valeur inférieure à 43 est 20, on prend donc 2 pièces de 20 centimes. Il reste 3 centimes.

× la plus grande valeur inférieure à 3 est 2, on prend donc 1 pièce de 2 centimes. Il reste 1 centime.

× la plus grande valeur inférieure à 1 est 1, on prend donc 1 pièce de 1 centime. Il reste 0 centime. L'algorithme s'arrête.

Le rendu de 143 centimes se fera alors comme suite : $100 + 2 \times 20 + 2 + 1$, ce qu'on représentera par la liste `[1, 1, 0, 0, 2, 0, 1]` qui donne, pour chaque valeur, le nombre de pièces utilisées.

- Écrire une fonction `somme_rendue` qui prend en paramètres les listes `pieces` et `rendu`, et qui renvoie la somme correspondant au `rendu` de monnaie dans le système `pieces`. On s'en servira pour les tests.
- Écrire une fonction `i_max_inferieur` qui prend en paramètre une liste `t` et un entier naturel `x`, et qui renvoie l'indice du plus grand élément de la liste `t` parmi ceux qui sont inférieurs ou égaux à `x`. On supposera qu'il en existe au moins un et que la liste `t` est triée dans l'ordre croissant.
- En déduire une fonction *récursive* `rendu_rec`, qui prend en paramètres une liste `pieces` et un entier naturel `montant`, et qui renvoie la liste représentant le rendu du `montant` en argument dans le système `pieces` en appliquant la stratégie gloutonne.
- Écrire de même une version *impérative* de cette fonction. On la nommera `rendu_iter`.
- En considérant le système `[1, 2, 4, 5]`, montrer que l'algorithme glouton ne donne pas toujours le plus petit nombre possible de pièces.
  *On peut cependant démontrer que, pour le système* `[1, 2, 5, 10, 20]` *correspondant aux euros, l'algorithme glouton renvoie toujours une solution optimale.*

# III. Problème du sac à dos

On rappelle qu'on dispose d'une collection d'objets sous forme de deux listes `valeurs` et `poids` représentant respectivement leurs valeurs et leurs poids, supposés entiers. On cherche à remplir un sac à dos de manière optimale sans dépasser le poids maximal qu'il peut contenir. Ce poids maximal sera donné par un entier `poids_max`.
Pour cela, à chaque étape, on met dans le sac à dos l'objet ayant le meilleur ratio valeur/poids parmi les objets ayant un poids inférieur à ce qu'il est encore possible de mettre dans le sac à dos. On représente cette sélection par une liste de 0 et de 1 selon que les objets sont choisis ou non.

- Supposons qu'on a défini les variables suivantes.
  × `valeurs = [126, 32, 20, 5, 18, 80]`
  × `poids = [14, 2, 5, 1, 6, 9]`
  × `poids_max = 15`
  Vérifier alors que la sélection `[0, 1, 0, 1, 0, 1]` est valide. Est-elle optimale ?

- Écrire une fonction `objets_disponibles` qui prend en paramètres la liste `poids` des poids de tous les objets, un entier `capacite` correspondant à la charge maximale et une liste `selection` telle que définie précédemment, et qui renvoie la liste des objets qui n'ont pas déjà été sélectionnés et dont le poids est inférieur ou égal à la capacité.
- Écrire une fonction `choix_objet` qui prend en paramètres les listes `valeurs`, `poids` et `objets`, et qui renvoie un élément de la liste d'entiers `objets` donnant un ratio valeur/poids maximal.
- Écrire une fonction `sac_a_dos` qui prend en paramètres les listes `valeurs` et `poids`, et l'entier naturel `poids_max`, et qui renvoie, sous forme de liste, la sélection d'objets obtenue par stratégie gloutonne.

## IV. Problème du voyageur de commerce

Un voyageur doit visiter $n$ lieux, donnés par leurs coordonnées cartésiennes $(x_i, y_i)$, qu'on supposera entières, dans un repère orthonormé du plan euclidien. On cherche à relier tous ces lieux par une ligne brisée de longueur minimale. Le point de départ est le premier de la liste. La stratégie gloutonne consiste alors à se rendre, à chaque étape, au lieu le plus proche parmi les lieux que l'on n'a pas encore visités.

- Montrer sur un exemple que cette stratégie n'est pas toujours optimale. Pourquoi n'est-il pas raisonnable de tester toutes les possibilités lorsque $n$ est grand ?
- Importer le module `random` sous l'alias `rd` puis entrer `help(rd.randint)` dans la console pour obtenir de l'aide sur la fonction `rd.randint`.
  Écrire alors une fonction `points.aleatoires` qui prend en paramètres des entiers `n` et `m`, et qui renvoie une liste de `n` points dont les coordonnées sont tirées aléatoirement parmi les entiers de 0 à `m`.
- Pour mémoriser les lieux déjà visités, on utilisera un dictionnaire `deja_vus` qui associe à chaque couple $(x_i, y_i)$ un booléen.
  Écrire une fonction `initialise_deja_vus` qui prend en paramètre une liste `lieux` et qui renvoie un dictionnaire associant la valeur `False` à tous les lieux de la liste.
- Écrire une fonction `lieux_a_visiter` qui prend en paramètre un dictionnaire `deja_vus`, et qui renvoie la liste des lieux qui n'ont pas encore été visités.
- Écrire une fonction `d2` qui prend en paramètres deux tuples `a` et `b` et qui renvoie le carré de la distance euclidienne entre les lieux `a` $= (x_a, y_a)$ et `b` $= (x_b, y_b)$.
- Écrire une fonction `plus_proche` qui prend en paramètres un tuple `position` et un dictionnaire `deja_vus`, et qui renvoie les coordonnées du lieu le plus proche parmi ceux qui restent à visiter.
- En déduire une fonction `chemin_glouton` qui prend en paramètre une liste de lieux `lieux`, et qui renvoie la liste des lieux visités successivement par le voyageur.
- Importer le module `matplotlib.pyplot` sous l'alias `plt`. Ce module permet de tracer des courbes. On pourra ainsi utiliser les fonctions suivantes :
  - `plt.clf()` qui réinitialise la fenêtre graphique,
  - `plt.plot(x, y, 'bo')` qui marque d'un cercle bleu le point de coordonnées `(x, y)`.
  - `plt.plot([xa, xb], [ya, yb], 'r-')` qui relie les points `(xa, ya)` et `(xb, yb)` par un segment de couleur rouge.
  - `plt.axis('equal')` qui normalise l'échelle des abscisses et l'échelle des ordonnées.
  - `plt.show()` qui affiche la fenêtre graphique.

  Simuler et afficher le trajet du voyageur pour 50 lieux tirés aléatoirement dans $[\![0, 100]\!]^2$.