# 08 - Algorithmes gloutons et Programmation dynamique

Le but de ce cours est de présenter des méthodes pour **résoudre des problèmes d'optimisation**. Un problème d'optimisation consiste à déterminer le jeu de données d'entrée permettant de minimiser ou de maximiser une fonction objectif, tout en satisfaisant à certaines contraintes.

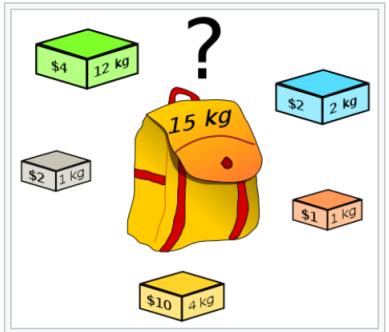
Nous avons déjà rencontrées des problèmes d'optimisation :

- recherche du plus grand carré blanc dans une image type QRcode *(cours 5 sur les dictionnaires et la mémoïsation)*
- recherche de la plus grande sous séquence commune entre deux mots *(DM n°2)*
- recherche d'un chemin de probabilité maximale dans un graphe *(DS n°2)*

Dans ce cours, nous travaillerons avec un problème très classique, appelé **problème du sac à dos**.

## 1 Présentation du problème du sac à dos

On dispose d'un sac à dos, que l'on doit remplir sans dépasser une masse maximale $M$ fixée à l'avance.
Pour remplir ce sac on dispose de $n$ d'objets, numérotés de 0 à $n-1$.
Pour tout $i$ entre 0 et $n$, l'objet $i$ a pour masse $m_i$, et pour valeur $v_i$.
L'objectif du problème du sac à dos est de remplir le sac avec **la plus grande valeur totale possible**, sous la contrainte que la masse totale des objets ne doit pas dépasser la capacité du sac à dos. *(on suppose pour simplifier que toutes les masses et toutes les valeurs sont des nombres entiers)*



Le problème du sac à dos : quelles boîtes choisir afin de maximiser la somme emportée tout en ne dépassant pas les 15 kg autorisés ?

Modélisation : On désigne par $[m_0,...,m_{n-1}]$ la liste des poids des objets, et $[v_0,...,v_{n-1}]$ la liste des valeurs des objets (ces objets étant numérotés de 0 à $(n-1)$).

## 2 Algorithme glouton

Un algorithme glouton est un algorithme permettant de **traiter des problèmes d'optimisation**. Son principe est de réaliser, **étape par étape, un choix optimum local**, afin d'essayer obtenir un résultat optimum global.
*Attention, rien ne dit qu'on obtiendra effectivement le meilleur résultat global, car une fois chaque étape effectuée, on ne revient jamais en arrière.*

### 2.1 Première version

Sur l'exemple du sac à dos, l'algorithme glouton le plus naïf consiste à prendre au fur et à mesure les objets de valeurs les plus élevés possibles, tant que la limite de poids n'est pas atteinte.

**Exemple 1 :** on a un sac à dos de capacité maximale 30 kg, on 4 objets numérotées de 0 à 3, la liste des masses est $[13,12,8,10]$, la liste des valeurs des objets est $[5,4,3,3]$.

**Q1** - En appliquant l'algorithme glouton naïf à cet exemple, donner les objets que l'on pourra ranger dans le sac, et la valeur totale du sac.

**Q2** - Programmer une fonction `Glouton1`, d'argument une masse maximale $M$, une liste de masses `ListeMasses`, une liste de valeurs `ListeValeurs`, qui retourne la liste des objets que contiendra le sac en suivant le principe ci-dessous, ainsi que la valeur totale du sac.

### 2.2 Deuxième version

Un algorithme glouton un peu plus "élaboré" consiste à d'abord calculer la liste des ratios $\frac{v_i}{m_i}$ (correspondant à la "rentabilité" du produit $i$, le prix au kilo si vous préférez !), à classer les objets du plus rentable au moins rentable, et à choisir les objets dans cet ordre, en vérifiant toujours que le poids maximal n'est pas dépassé.

**Q3** - Appliquer cette méthode à l'exemple 1, et donner les objets que l'on pourra ranger dans le sac, ainsi que la valeur totale.

**Q4** - Programmer une fonction `Glouton2`, d'argument une masse maximale $M$, une liste de masses `ListeMasses`, une liste de valeurs `ListeValeurs`, qui retourne la liste des objets que contiendra le sac en suivant le principe ci-dessous, ainsi que la valeur totale du sac.

## 3 Algorithmes de programmation dynamique

Comme on vient de le voir sur le dernier exemple, un algorithme glouton ne garantit pas de trouver la solution optimale. Pour pallier ce problème, on va mettre en place un algorithme dit *de programmation dynamique*.

Le principe est le suivant : nous allons trouver une solution optimale pour toute capacité de sac de $0$ à $M$.

Notons $V(k,m)$ la valeur maximale pour remplir un sac de capacité $m$ en utilisant les $k$ premiers objets On a alors les affirmations suivantes :

- Pour toute masse $m$, $V(0,m)=0$.
- Pour calculer $V(k+1,m)$, on peut dire que :
  - Soit l'objet numéro $k$ ne fait pas partie du chargement, et dans ce cas la valeur cherchée serait $V(k,m)$
  - Soit l'objet numéro $k$ fait partie du chargement (possible uniquement si $m_k \leq m$), et dans ce cas la valeur cherchée serait $V(k,m-m_k)+v_k$.

  Il faut donc calculer les deux nombres précédents et prendre le maximum des deux.

### 3.1 Version Bottom-up

**Q5** - En reprenant l'exemple 1, vous allez alors remplir progressivement le tableau suivant (dans chaque champ du tableau on indique $V(k,m)$) :

| m=masse maximale / k=nombre d'objets | 0 | 1 | 2 | ... | 30 |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | ... | 0 |
| 1 | | | | | |
| 2 | | | | | |
| 3 | | | | | |
| 4 | | | | | |

La valeur cherchée est alors $V(4,30)$.

**Q6** - Programmer une fonction `Dynamique1`, d'argument une masse maximale $M$, une liste de masses `ListeMasses`, une liste de valeurs `ListeValeurs`, qui retourne la valeur totale du sac. Dans cette fonction, vous construirez donc un tableau

(vous pouvez utiliser un tableau numpy), dont vous pourrez initialiser toutes les cases à $0$ ,et que vous remplirez ligne par ligne. Le résultat renvoyé sera la valeur contenue dans la dernière case du tableau.

### 3.2 Version Top-Down

**Q7** - En utilisant les relations de récurrence établies au début du paragraphe 3, programmer une fonction *récursive* `Dynamique2`, d'argument un nombre $k$ d'objets à placer, une masse maximale $M$, une liste de masses `ListeMasses`, une liste de valeurs `ListeValeurs`, qui retourne la valeur totale du sac.