1 Principe

> On appelle **algorithme glouton** un algorithme **produisant une solution pas à pas**, en faisant à chaque étape un choix qui optimise un **critère local**, dans l'espoir d'obtenir une **optimisation globale**.

Cependant, les algorithmes gloutons **ne fournissent pas toujours une solution optimale**.

Il existe tout de même des problèmes où ces algorithmes sont mis en œuvre car ils sont généralement assez simples à développer, et la solution exacte au problème peut être très compliquée à obtenir par un autre moyen.
On espère dans ces cas que les solutions apportées par les algorithmes restent suffisamment intéressantes.

2 Exemples

2.1 Le problème du rendu de monnaie

Le problème du rendu de monnaie consiste à **rendre une somme** d'argent **avec un minimum de pièces** (critère *global*).

Un algorithme glouton propose de répondre à ce problème en le décomposant étape par étape.
À chaque étape, la pièce rendue est celle de plus grande valeur possible afin de minimiser la somme restant à rendre (critère *local*).

Remarque : Cet algorithme optimise le nombre total de pièces rendues (critère global) si le système de pièces existantes est bien choisi (ex : 1, 2, 5 10). Mais ce n'est pas le cas pour n'importe quel système de pièces (ex : 1, 3, 4, 10).

Exemple d'implémentation :

```
def rendu_monnaie(somme_a_rendre, jeu_pieces):
    """ Renvoie la liste des pièces à rendre pour atteindre la somme à rendre.
            - Entrées :
                - somme_a_rendre est un entier : int
                - jeu_pieces contient la liste des pièces disponibles : list(int)
            - Sortie : pieces_rendues est la liste des pièces rendues : list(int)
    """
    jeu_pieces.sort(reverse=True) # tri décroissant des pièces disponibles
    pieces_rendues = []
    i = 0
    while somme_a_rendre > 0:
        piece = jeu_pieces[i]
        if piece <= somme_a_rendre :
            pieces_rendues.append(piece)
            somme_a_rendre = somme_a_rendre - piece
        else :
            i = i + 1
    return pieces_rendues
```

2.2 Le problème du sac à dos

Le problème du sac à dos consiste à choisir parmi une collection d'objets de valeurs différentes et de poids différents lesquels choisir pour remplir son sac à dos, en respectant une contrainte : le sac à dos ne peut supporter qu'une charge maximum limitée.

Le critère d'optimisation global est d'obtenir le sac à dos de plus grande valeur.

Un algorithme glouton peut essayer de répondre à ce problème pas à pas. À chaque étape on choisit un objet selon un critère local fixé ; par exemple choisir l'objet dont le rapport valeur/poids est maximum, ou plus simplement l'objet de valeur maximale.
Quelques tests permettent de se rendre compte qu'aucun de ces critères locaux ne garantit d'obtenir une solution optimale globale, mais on peut tout de même espérer qu'ils fournissent une solution suffisamment intéressante dans la plupart des cas.

Exemple d'implémentation :

```
def clef_tri(objet):
    """ Renvoie la clef de tri valeur/poids de l'objet.
    - Entrée  : objet est un tuple (valeur, poids)
    - Sortie : renvoie un nombre qui est le rapport valeur/poids : float
    """
    (valeur, poids) = objet
    return valeur/poids

def sac_a_dos(objets, poids_maxi):
    """ Renvoie la liste des objets retenus.
    - Entrées :
            - objets est une liste d'objets à choisir
              (chaque objet est une tuple (valeur, poids))
            - poids_maxi est un nombre : float
    - Sortie : la valeur renvoyée est la liste d'objets retenus : list(tuple)
    """
    objets_retenus = []
    objets.sort(key=clef_tri, reverse=True)  # tri de la liste d'objets
    for objet in objets:
        (valeur, poids) = objet
        if poids < poids_maxi:
            objets_retenus.append(objet)
            poids_maxi = poids_maxi - poids
    return objets_retenus
```