I – Principe

Les algorithmes gloutons sont des techniques visant à résoudre des **problèmes d'optimisation**. Ces problèmes résolvent des questions du type : « comment obtenir le plus grand ... ? », « Comment aller le plus vite vers ... ? », etc.

> 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.

II – Exemples

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*).

```
1  Fonction renduMonnaie(sommeARendre, piecesDispo)
   Entrées :
   sommeARendre : entier
   piecesDispo : liste des valeurs des pieces accessibles, triée par ordre décroissant.
   Sorties : piecesRendues : liste des pieces rendues ; on souhaite que cette liste soit la plus
             courte possible.
2  piecesRendues ← [ ]
3  i ← 0                  /* indice de piecesDispo, i=0 désigne la plus grande pièce */
4  tant que sommeARendre > 0 faire
5  |  piece ← piecesDispo[i]
6  |  si piece <= sommeARendre alors
7  |  |  piecesRendues ← piecesRendues + [piece]      /* on garde la piece courante */
8  |  |  sommeARendre ← sommeARendre - piece
9  |  fin si
10 |  sinon
11 |  |  i ← i + 1                  /* pour passer à la plus grande pièce suivante */
12 |  fin si
13 fin tq
14 renvoyer piecesRendues
```

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).

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.

```
1  Fonction remplirSac(ressources, poidsMaxi)
   Entrées :
   ressources : liste des couples (valeur, poids) des objets convoités.
   poidsMaxi : charge maximale du sac à dos.
   Sorties : sac : liste des objets retenus.
2  trier (ressources, clefDeTri=choixLocal)     /* le choix local dépend de la stratégie
     gloutonne retenue */
3  sac ← [ ]
4  pour chaque objet in ressources faire
5  |  poids ← objet[poids]
6  |  si poids <= poidsMaxi alors
7  |  |  sac ← sac + [objet]                          /* on garde l'objet courant */
8  |  |  poidsMaxi ← poidsMaxi - poids
9  |  fin si
10 fin pour chaque
11 renvoyer sac
```

À cet algorithme principal, on associe une clef de tri pour trier les ressources disponibles selon le critère local de la stratégie gloutonne qu'on veut mettre en œuvre.

```
1  Fonction choixLocal(objet)
   Entrées :  objet : couple (valeur, poids) d'un objet.
   Sorties : définit une clef de tri pour une liste d'objets.
   /* ici on retient le rapport (valeur/poids) comme clef de tri */
2  renvoyer objet[valeur] / objet[poids]
```

III – Compléments

On a vu à travers ces exemples que les algorithmes gloutons ne conduisent pas forcément à la solution optimale globale. Cependant, il existe heureusement des situations où on peut **démontrer** qu'une stratégie gloutonne résout correctement le problème d'optimisation attendu : dans ce cas, l'algorithme glouton est souvent plus simple que d'autres algorithmes possibles.

En classe de Terminale, on étudie une autre approche algorithmique, appelée **programmation dynamique**, qui permet de résoudre correctement certains problèmes d'optimisation qui échouent avec les algorithmes gloutons.