Chapitre 20

# Algorithme Glouton

## I. Découverte de l'algorithme

**Exercice 1** Marc et Luc partent en randonnée pour plusieurs jours, ils ont chacun un sac de 40L et veulent mettre un maximum de choses dedans pour espérer ne manquer de rien et, au pire, ne pas avoir de regrets.

Ils ont tous les deux réalisé des petits paquets de même volume pour emballer leurs affaires. Voici le volume en litres des différents paquets : 20, 15, 10, 8, 7, 7, 3. On suppose dans la suite de l'exercice que les paquets peuvent être déformés pour s'emboîter parfaitement dans le sac (Tetris!).

Marc fait son sac très rapidement, va voir Luc et lui dit : "C'est fait, j'ai 38L dans mon sac, je ne peux rien mettre de plus". Marc l'ignore et continue de réfléchir à l'organisation de son sac. Luc s'impatiente : "Allez, viens, on y va!". Marc s'arrête alors de réfléchir, choisit ses paquets et remplit totalement son sac : "Ca y est, on peut partir, mon sac est plein à ras bord".

1. Quelle a été la stratégie de Luc pour remplir son sac si rapidement et obtenir un volume de 38 L?
2. Comment Marc a-t-il fait pour remplir totalement son sac?
3. Finalement, que doit-on penser de la stratégie de Luc?

**Définition 1:**

Dans un problème d'optimisation, utiliser un algorithme glouton revient à choisir systématiquement le plus grand (ou plus petit) élément disponible sans se préoccuper de la suite. Ainsi, on espère qu'en prenant le plus grand (ou plus petit) élément à chaque fois, on aura à la fin une solution optimale.
Utiliser un algorithme glouton n'est pas toujours une réussite.

**Exemple 1:**

Dans le problème précédent, on cherche à optimiser le volume total des paquets dans le sac. Ici Luc utilise un algorithme glouton pour parvenir à ses fins (il a choisi systématiquement le plus grand paquet qui pouvait entrer dans la place restante du sac). Malheureusement cet algorithme n'est pas optimal car Marc a fait mieux.

**Définition 2:**

Dans certains cas, la distribution des éléments implique que l'algorithme glouton est optimal. On dit alors que **le système est canonique** pour cet algorithme.

**Exemple 2:**

Dans le problème précédent, le système constitué des différents paquets : 20, 15, 10, 8, 7, 7, 3 n'est pas canonique car l'algorithme glouton choisi n'a pas produit une solution optimale.

**Exercice 2** On considère un nouveau problème. Une personne nous donne 30 chak (la monnaie locale) pour nous acheter un produit qui coûte 14 chak. On doit donc lui rendre 16 chak et on veut lui rendre cette monnaie en utilisant le moins de pièces possibles. On choisit d'utiliser un algorithme glouton qui donne à chaque fois que c'est possible la plus grande pièce.

1. Imaginons qu'on ait à disposition ce système de pièces : 10, 5, 2, 1. Ce système paraît-il canonique?
2. Imaginons qu'on ait à disposition ce système de pièces : 10, 4, 3, 1. Ce système paraît-il canonique?

**Remarque :**

1. Trouver un contre-exemple pour montrer qu'un système n'est pas canonique est parfois facile. Par contre, montrer qu'un système est canonique est systématiquement difficile.
2. Un système peut être canonique pour un algorithme glouton donné et pas canonique pour un autre algorithme glouton résolvant le même problème.
3. Presque tous les systèmes de pièces réels de par le monde sont canoniques (dont celui de l'euro). Voilà pourquoi la méthode usuelle de rendu de monnaie est efficace. Mais, le passage en canonique de certains systèmes est tout récent comme la Livre Sterling au Royaume-Uni depuis 1971.

# II. Codage d'un algorithme Glouton sur Python

**Situation :** Un achat dit en espèces se traduit par un échange de pièces et de billets. Dans la suite de ce T.P., les pièces désignent indifféremment les véritables pièces que les billets. Supposons qu'un achat induise un rendu de monnaie de 49 euros. Quelles pièces peuvent être rendues ? La réponse, bien qu'évidente, n'est pas unique. Quatre pièces de 10 euros, 1 pièce de 5 euros et deux pièces de 2 euros conviennent. Mais quarante-neuf pièces de 1 euros conviennent également ! Si la question est de rendre la monnaie avec un minimum de pièces, le problème change de nature. Mais la réponse reste simple : c'est la première solution proposée qui fonctionne ! Toutefois, comment parvient-on à un tel résultat ? Quels choix ont été faits qui optimisent le nombre de pièces rendues ? C'est le problème du rendu de monnaie dont la solution dépend du système de monnaie utilisé.

Dans le système monétaire français, les pièces prennent les valeurs 1, 2, 5, 10, 20, 50, 100 euros. Pour simplifier, nous nous intéressons seulement aux valeurs entières et oublions l'existence des billets de 200 et 500 euros. Rendre 49 euros avec un minimum de pièces est un problème d'optimisation. En pratique, sans s'en rendre compte généralement, tout individu met en œuvre un algorithme glouton. Implémentons-le à l'aide du langage Python !

**Exercice 3** *En langage naturel*

1. Rédiger un algorithme intuitif (en langage courant, pas en Python) permettant de rendre une somme nommée $S$.
2. Appliquer votre algorithme au rendu de 49€.
3. Appliquer votre algorithme au rendu de 49€ avec le système $\{1,3,6,12,24,30\}$ . Combien de pièces sont rendues ?
4. Peut-on trouver une façon différente de rendre la monnaie avec moins de pièces ? Ce nouveau système est-il canonique pour notre algorithme ?

La réponse à cette difficulté de l'optimisation passe par la programmation dynamique, thème qui sera abordé en spécialité NSI de classe de terminale.

**Exercice 4** Coder une fonction en Python qui permet de rendre la monnaie en connaissant la somme à rendre et le système de monnaie.

Vous pouvez faire cela seul ou en vous inspirant du guidage ci-dessous (il utilise les listes mais on peut aussi utiliser de manière pertinente des dictionnaires).

1. Définir le système de pièces à l'aide d'un tableau de valeurs `systeme_ monnaie` des pièces classées par valeurs croissantes.
2. Pour stocker les pièces à rendre, créer un tableau vide `lst_ pieces`.
3. La première pièce à rendre est potentiellement la dernière pièce du tableau `systeme_ monnaie`. Créer une variable $i$, initialisée avec l'indice du dernier élément de ce tableau.
4. Chaque fois qu'une pièce du tableau n'est plus utilisable, la valeur de $i$ sera diminuée de 1. Le programme doit s'arrêter quand la somme à rendre est nulle. Ce qui mène à l'écriture d'une boucle conditionnelle pour remplir la liste des pièces choisies. Créer et initialiser la variable `somme_ a_ rendre` avec une valeur (par exemple 49), puis créer la boucle citée.
5. Tester votre code.
6. Pour finir, le code précédent peut être encapsulé dans une fonction qui reçoit deux arguments « la somme à rendre et le système de monnaie » et qui renvoie la liste des pièces choisies par l'algorithme glouton (éventuellement l'affiche et affiche un nombre de pièces rendues).

**Exercice 5** *Preuve de terminaison*

1. En s'aidant du T.P. sur la dichotomie, chercher un variant de boucle (une quantité entière positive qui décroît à chaque itération de la boucle).
2. Effectuer une preuve de terminaison (ne pas s'inquiéter de la rédaction pour l'instant, donner l'idée générale).