Informatique — MPSI/PCSI, 2024–2025

TP n° 5 – Algorithmes gloutons

D'après des documents eduscol

Objectifs :

- Comprendre la philosophie des algorithmes gloutons.
- Comprendre que l'algorithme glouton choisi fournit parfois la solution optimale, MAIS parfois même pas une solution exacte.

1 Généralités

Optimiser un problème, c'est déterminer les conditions dans lesquelles ce problème présente *une caractéristique spécifique*. Par exemple, déterminer le minimum ou le maximum d'une fonction 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 d'un plus court chemin dans un graphe... 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, 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).

Les **algorithmes gloutons** constituent une alternative dont le résultat n'est pas toujours optimal. Plus précisément, ces algorithmes déterminent une solution optimale en effectuant successivement des choix **locaux**, jamais remis en cause. Au cours de la construction de la solution, l'algorithme résout une partie du problème puis se focalise ensuite sur **LE** sous-problème restant à résoudre. Une différence essentielle avec la *programmation dynamique* est que celle-ci peut remettre en cause des solutions déjà établies. Au lieu de se focaliser sur un seul sous-problème, elle explore les solutions de **TOUS** les sous-problèmes pour les combiner finalement de manière optimale.

Le principal avantage des **algorithmes gloutons** est leur facilité de mise en œuvre. En outre, dans certaines situations dites **canoniques**, il arrive qu'ils renvoient non pas **UN** optimum mais **L'**optimum d'un problème.

Le but de ce TP est de présenter de telles situation, en montrant les avantages mais aussi les limites de la technique.

2 Rendu de monnaie

Un achat dit en espèces se traduit par un échange de pièces et de billets. Dans la suite ce cet exposé, les pièces désignent indifféremment les véritables pièces que les billets. Supposons qu'un achat induise un rendu 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**. La réponse est la première solution proposée. Toutefois, comment parvient-on à un tel résultat ? Quels choix ont été faits qui optimisent le nombre de pièces rendus ? 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 0.01, 0.02, 0.05, 0.10, 0.20, 0.50, 1, 2, 5, 10, 20, 50, 100, 200, 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. Il choisit d'abord la plus grandeur valeur de monnaie, parmi 1, 2, 5, 10, contenue dans 49 euros. En l'occurrence, quatre fois une pièce de 10 euros. La somme de 9 euros restant à rendre, il choisit une pièce de 5 euros, puis deux pièces de 2 euros. Cette stratégie gagnante pour la somme de 49

euros l'est-elle pour n'importe quelle somme à rendre ? **On peut montrer que la réponse est positive pour le système monétaire français**. Pour cette raison, **un tel système de monnaie est qualifié de canonique**.

Exercice 5.1
Considérons une monnaie qui a pour liste de valeurs $[c_0, ..., c_{n-1}]$ avec $0 < c_0 < ... < c_{n-1}$.

1. Pour simplifier, nous nous intéressons seulement aux valeurs entières, dans un premier temps. Implémenter une fonction `rendu_monnaie_1(s,P)` qui étant donné une liste de valeurs P et une somme s à rendre, applique la stratégie gloutonne décrite plus haut et renvoie la liste des valeurs rendues dans l'ordre par le commerçant.
2. Tester la fonction écrite pour différents exemples.

```
>>> rendu_monnaie_1(49,EUR)
[20, 20, 5, 2, 2]
>>> rendu_monnaie_1(76,EUR)
[50, 20, 5, 1]
```

3. Adapter, si nécessaire, la fonction pour une liste complète de valeurs, ne se limitant plus aux valeurs entières (on se limitera à des valeurs décimales multiples du centime).
4. Commenter les éventuels problèmes rencontrés.

```
>>> rendu_monnaie_3(13.45,EURC)
[10, 2, 1, 0.2, 0.2, 0.05]
>>> rendu_monnaie_3(25.58,EURC)
[20, 5, 0.5, 0.05, 0.02, 0.01]
```

5. Tester la fonction `rendu_monnaie_1(s,P)` pour les valeurs $P = [1, 3, 6, 12, 24, 30]$ et $s = 49$. Conclure.
6. Effectuer un dernier test pour les valeurs $P = [2, 3]$ et $s = 7$. Conclure

Dans un problème, la stratégie gloutonne consiste à faire le choix le plus efficace **localement** (c'est-à-dire *à chaque étape*). Parfois, la stratégie conduit bien à *la solution optimale*. **MAIS** ce n'est pas toujours le cas. Il arrive que cela ne conduise même pas à la solution recherchée...

3 Le problème du sac à dos

3.1 Le problème type

On dispose d'un sac pouvant supporter un poids maximal donné et de divers objets ayant chacun une valeur et un poids. Il s'agit de choisir les objets à emporter dans le sac afin d'obtenir la valeur totale la plus grande tout en respectant la contrainte du poids maximal. C'est un *problème d'optimisation avec contrainte*.

Ce problème peut se résoudre par **force brute**, c'est-à-dire en testant tous les cas possibles. Mais ce type de résolution présente un problème d'efficacité. Son coût en fonction du nombre d'objets disponibles croît de manière exponentielle.

Nous pouvons envisager **une stratégie gloutonne**. Le principe d'un algorithme glouton est de faire le meilleur choix pour prendre le premier objet, puis le meilleur choix pour prendre le deuxième, et ainsi de suite. Que faut-il entendre par meilleur choix ? Est-ce prendre l'objet qui a la plus grande valeur, l'objet qui a le plus petit poids, l'objet qui a le rapport valeur/poids le plus grand ? Cela reste à définir.

3.2 Le problème à résoudre

Soit un sac à dos, dont le contenu ne doit pas dépasser une masse de 20 kg. Le propriétaire du sac dispose de plusieurs objets, caractérisés chacun par une masse et une valeur. Quels objets emporter dans le sac à dos pour maximiser la valeur totale, sans dépasser le poids maximum ?

Les tableaux 1 et 2 ci-dessous présentent les objets disponibles avec les valeurs en euros et les masses en kg pour chacun des deux sacs à dos.

Objets	Valeur	Masse
1	2.0	10.0
2	4.0	14.0
3	4.0	7.0
4	5.0	3.0

TABLE 1 – Sac à dos n°1, masse maximale autorisée : 20 kg

Objets	Valeur	Masse
1	2.0	10.0
2	4.0	11.0
3	8.0	7.0
4	5.0	5.0
5	8.0	6.0
6	10.0	15.0
7	11.0	8.0

TABLE 2 – Sac à dos n°2, masse maximale autorisée : 40 kg

Dans la suite du problème, un objet sera représenté par un triplet contenant son numéro de type `int`, sa valeur de type `float` et sa masse de type `float`. Les triplets obtenus sont les éléments d'une liste.

3.3 Force brute

Le principe est simple : il faut tester tous les cas possibles !
La mise en œuvre l'est moins. Comment obtenir tous les cas sans les répéter et sans en oublier un ? Cette question pose la difficulté principale.

Indication : *une méthode est d'associer le chiffre 1 à un objet s'il est choisi et le chiffre 0 sinon. Nous obtenons ainsi un nombre entier écrit en binaire avec 4 chiffres. Le nombre 1011 signifie que nous avons choisi les objets 1, 3 et 4. Le nombre 1111 signifie que nous avons choisi tous les objets. À chaque nombre correspond exactement une possibilité pour construire une partie de l'ensemble des 4 objets.*
Il apparaît alors que le nombre total de cas est 2^4 *puisqu'avec 4 chiffres (0 ou 1), nous pouvons écrire exactement* 2^4 *nombres.*

Exercice 5.2

1. En utilisant l'indication précédente, écrire une fonction `ens_des_parties` qui prend en paramètre un ensemble d'objets `objets` et renvoie une liste dont chaque élément est une liste d'entiers (0 ou 1) traduisant une partie de l'ensemble.
   Indication : on pourra utiliser la fonction `bin` qui renvoie la représentation binaire d'un entier sous forme de chaîne de caractères commençant par `'0b'`. Une coupe de la chaîne pourra être effectuée...
2. Implémenter deux fonctions `valeur_totale(objets, liste)` et `masse_totale(objets, liste)` qui renvoient respectivement la valeur et la masse totale du choix d'objets donnée par la liste.
3. Implémenter une fonction `force_brute(objets, masse_max)` qui, étant donné un ensemble d'objets, applique la stratégie de la *force brute* et renvoie la liste des objets à emporter dans le sac à dos pour maximiser la valeur de son contenu tout en respectant une exigence de masse maximale et sa valeur totale.
4. Afficher la liste des objets à emporter ainsi que la valeur du contenu du sac à dos optimal pour les deux exemples proposés dont les objets sont données dans le fichier `Info-TP05_cadeau.py` et sachant que l'on doit obtenir :

```
>>> force_brute(Sac1, 20)
([1, 3, 4], 11.0)
>>> force_brute(Sac2, 40)
([3, 5, 6, 7], 37.0)
```

3.4 Stratégie gloutonne

Exercice 5.3

1. Implémenter une fonction `glouton(objets, masse_max, choix)` (et les sous-fonctions utiles), qui prend en paramètres une liste de d'objets, une masse maximale `masse_max` (celle que peut contenir le sac à dos) et le type de choix utilisé (par valeur, par masse, ou par le rapport valeur/masse) pour la stratégie gloutonne et qui renvoie la liste des objets à emporter pour maximiser le critère choisi.

Indication : on pourra construire une nouvelle liste en triant la liste passée en paramètre suivant le type de choix utilisé avec la fonction `sorted`. L'instruction `help(sorted)` écrite dans l'interpréteur permet d'obtenir des informations sur la fonction `sorted`[1].
Les sous-fonctions `masse(objet)`, `valeur(objet)` et `rapport(objet)`, qui renvoient respectivement la masse, la valeur et le rapport valeur/masse d'un objet représenté par un triplet, pourront être implémentées.

2. Comparer les résultats obtenus pour chaque choix avec ceux obtenus à l'aide de la stratégie de la *force brute*. Conclure.

* *
*

1. pour des exemples, consultez https://pythonexamples.org/python-sorted/