ALGORITHMES GLOUTONS

1 Problème du rendu de monnaie

Il s'agit du problème du monnayeur : comment rendre la monnaie en utilisant le plus petit nombre de pièces ?

1.1 Formalisation du problème

On appelle *système* un $m-$uplet d'entiers $s = (s_i)_{1 \leq i \leq m}$ vérifiant : $1 = s_1 < s_2 < \cdots < s_m$. Les s_i sont les valeurs faciales des pièces (ou billets) en service. Par exemple, pour le système en zone Euro c'est : $(1, 2, 5, 10, 20, 50, 100, 200, 500, 1000, 2000, 5000, 10000, 20000, 50000)$ si on prend en compte billets et pièces. Tout est exprimé en centimes d'euro.
Dans tout le TP, on supposera toujours qu'on possède une quantité illimitée de pièces de toutes valeurs.
Soit $x \in \mathbb{N}$, le montant à rendre. Une *représentation de x* dans ce système s est un $m-$uplet $k = (k_1, \ldots, k_m)$ vérifiant : $x = \sum_{i=1}^{m} k_i s_i$. k_i est donc le nombre de pièces de valeur s_i qui seront rendues. Pour épargner les poches des clients, on souhaite minimiser le poids de cette représentation, c'est-à-dire la quantité :

$$w(k) = \sum_{i=1}^{m} k_i,$$

ou encore le nombre de pièces rendues.

Les systèmes et les représentations seront donnés sous forme de tableau, `[3, 1, 4]` est par exemple une représentation de 30 dans le système $(1, 3, 6)$.

1. Écrire en Python une fonction `est_un_systeme` qui prend en argument un tableau et qui renvoie `True` si c'est un système et `False` sinon.

1.2 L'algorithme glouton

L'algorithme glouton pour rendre une somme $x > 0$ consiste à toujours prioriser les valeurs les plus grandes. Par exemple avec le système $s = (1, 2, 5, 10)$, l'algorithme décomposera 27 en $2 \times 10 + 5 + 2$.

2. Écrire une fonction `glouton_monnaie_rec` prenant en argument la somme à rendre `x`, le système `S` et un entier `i`, et qui renvoie la représentation selon l'algorithme glouton de `x` dans le système `S[0:i]`, c'est-à-dire en n'utilisant que les `i` premières valeurs du système `S`. Cette fonction devra être récursive en `i`.

3. En déduire une fonction `glouton_monnaie` prenant en argument `x` et `S` et renvoyant la représentation de `x` dans le système `S` selon l'algorithme glouton. Par exemple `glouton_monnaie(13, [1, 2, 5])` retournera `[1, 1, 2]`.

4. Au franprix, on ne dispose dans les caisses que des billets de valeurs inférieures à 20€ et toutes les pièces du système. Écrire une fonction `glouton_monnaie_euro` en Python qui prend en argument le montant à rendre en euros, et qui affiche la liste des billets et pièces à rendre selon le modèle ci dessous :

```
>>> glouton_monnaie_euro(36.31)
Il faut rendre :
- 1.0 billets de  20 euros
- 1.0 billets de  10 euros
- 1.0 billets de  5 euros
- 0.0 pieces de  2 euros
- 1.0 pieces de  1 euros
- 0.0 pieces de  50 centimes d euros
- 1.0 pieces de  20 centimes d euros
- 1.0 pieces de  10 centimes d euros
- 0.0 pieces de  5 centimes d euros
- 0.0 pieces de  2 centimes d euros
- 1.0 pieces de  1 centimes d euros
```

1.3 Système canonique

On dira que le système est *canonique* lorsque l'algorithme glouton donne toujours une représentation minimale.

5. Montrer que tout système (s_1, s_2) est canonique.

6. Exhiber un système (s_1, s_2, s_3) non canonique (justifier).

7. Avant la réforme de 1971 introduisant un système décimal, le Royaume-Uni utilisait le système $(1, 3, 6, 12, 24, 30)$. Montrer que ce système n'est pas canonique.

Heureusement, on peut montrer que le système monétaire de la zone euro est canonique.

2 Réservation d'une salle

La salle Médiathèque du lycée est partagée entre plusieurs classes. Chaque cours est caractérisé par une date de début d_i et une date de fin f_i. On souhaite que le maximum de cours ait lieu dans la salle, deux cours ne pouvant avoir lieu en même temps (leurs intervalles de temps doivent être disjoints).

Le problème est donc défini par :

- *en entrée* : n le nombre de cours et $(d_1, f_1), \ldots, (d_n, f_n)$ pour chacun d'entre eux, les dates de début et de fin.

- *en sortie* : les réservations retenues, un ensemble d'entiers $J \subset [\![1, n]\!]$. Les réservations retenues sont compatibles si :

$$\forall (i, j) \in J^2, i \neq j, \quad d_i \geq f_j \quad \text{ou} \quad f_i \leq d_j.$$

On aura satisfait le maximum de demandes si le cardinal de J est maximal.

Nous allons tenter de résoudre ce problème à l'aide d'un algorithme glouton. On a pu voir dans la première partie que le schéma général de l'algorithme est basé sur un critère local de sélection des éléments pour construire une solution optimale. On doit disposer pour cela d'un critère de sélection qui permet de choisir le meilleur élément restant à ajouter à la solution. Le choix de ce critère doit être bien réfléchi.

2.1 Le problème du choix du critère glouton

8. On considère 3 critères possibles :
   (a) la durée du cours ;
   (b) la date de début du cours ;
   (c) le nombre d'intersections du cours avec un autre cours.

   Pour chacun de ces critères, on obtient une stratégie en triant les cours dans l'ordre croissant, puis à chaque étape en choisissant le premier cours compatible avec les cours déjà sélectionnés.
   Pour chacune de ces stratégies, montrer par un exemple qu'elle n'est pas optimale.

2.2 Le bon choix, étude de l'optimalité et de la complexité

9. On décide à présent de trier les cours par dates de fin croissantes : on choisit le cours se terminant au plus tôt, puis le cours se terminant au plus tôt parmi ceux qui sont compatibles, etc.
   On représente l'entrée du problème par un tableau dont chaque élément est un triplet composé de deux entiers correspondant aux dates de début et de fin du cours, et d'une chaîne de caractère qui identifie la classe. L'instruction `list.sort(t, key =f)` permet de trier le tableau `t` selon la fonction `f`, de façon à ce qu'après l'instruction, deux éléments consécutifs a et b vérifient $f(a) \leq f(b)$
   Préciser et programmer la fonction f à utiliser dans notre cas.
10. Programmer en Python une version itérative de l'algorithme glouton utilisant ce critère. Cette fonction renverra un tableau au même format que l'entrée, mais ne contenant que les cours retenus dans le planning.
11. Étudier l'optimalité de la solution gloutonne pour cette stratégie.
12. Sachant qu'un tri peut se faire en $O(n \log n)$ opérations, estimer la complexité de l'algorithme glouton.

3 En résumé

Nous avons rencontré des problèmes d'optimisation. Les techniques de programmation dynamique ou d'optimisation linéaire peuvent apporter une solution mais sont parfois complexes à mettre en œuvre.
Les algorithmes gloutons (*greedy algorithms*) constituent une alternative beaucoup plus simple à programmer, relativement rapide à exécuter, mais dont le résultat n'est pas toujours optimal (sauf dans certaines situations dites *canoniques*).
L'approche gloutonne consiste à construire une solution complète par une succession de choix locaux donnant systématiquement la meilleure solution partielle.
Généralement, on peut employer un algorithme glouton lorsque :

- une solution complète peut être construite en passant par une succession de solutions partielles,
- chaque solution partielle est établie en faisant un choix local à partir de la solution partielle précédente,
- on dispose d'une fonction permettant d'évaluer la qualité de chaque solution partielle.

Les choix ne sont jamais remis en cause : une fois faits, on ne revient pas dessus. Cela constitue une différence essentielle avec la programmation dynamique qui au lieu de se focaliser sur un seul sous-problème, explore les solutions de tous les sous-problèmes pour les combiner finalement de manière optimale.

Fin de l'énoncé

Aide Python :

- reste de la division euclidienne : `666 %13`;
- quotient de la division euclidienne `666 //13`;
- ajouter un élément en fin de liste : `ma_liste.append(x)` ;
- longueur d'une liste : `len(ma_liste)` ;
- trier une liste selon un critère : L'instruction `list.sort(t, key =f)` permet de trier le tableau `t` selon la fonction `f`, de façon à ce qu'après l'instruction, deux élements consécutifs a et b vérifient $f(a) \leq f(b)$