Informatique – TP8

Vésale Nicolas - Henrik Thys

Exercice 1: Rendu de monnaie: l'algorithme récursif.

On s'intéresse à un algorithme récursif qui permet de rendre la monnaie à partir d'une liste donnée de valeurs de pièces et de billets.
Le système monétaire est donné sous forme d'une liste

$$\texttt{valeurs} = [100, 50, 20, 10, 5, 2, 1].$$

On suppose que les pièces et les billets sont disponibles sans limitation. On cherche à utiliser un algorithme glouton **récursif** pour donner la liste des valeurs à rendre pour une somme donnée en argument.
Compléter à cet effet la fonction `renduRecursif` du fichiers de tests.

Voici quelques exemples:

```
>>> renduRecursif(67, 0)
[50, 10, 5, 2]
>>> renduRecursif(291, 0)
[100, 100, 50, 20, 20, 1]
>>> renduRecursif(291,1) # si on ne dispose pas de billets de 100
[50, 50, 50, 50, 50, 20, 20, 1]
```

Exercice 2: Allocations d'une salle pour des cours.

Le but de cet exercice est de donner un algorithme permettant de résoudre la question suivante: comment allouer une salle pour des cours de façon optimale ? Pour ceci, on suppose qu'on dispose de deux listes `D` et `F` qui représentent les horaires (que l'on supposera pour simplifier entiers positifs) demandés par les différents professeurs. Par exemple si:

$$\texttt{D} = [1, 2, 6, 6, 7, 10, 11, 17] \quad \text{et} \quad \texttt{F} = [5, 8, 11, 12, 8, 12, 15, 21]$$

alors un professeur demande une salle pour l'horaire de `1` à `5`, un autre pour l'horaire de `2` à `8` et ainsi de suite.

1. **On supposera dans cette question que comme dans l'exemple que les demandes sont triées par horaire croissant de début de demande.**
   Pour répondre à la question, on va utiliser l'algorithme glouton suivant: on prend la demande qui débute le plus tôt puis la première qui débute juste après la fin de celle-ci et ainsi de suite. Écrire une fonction `glouton1(D,F)` qui rend la liste des demandes attribuées pour cette salle par cet algorithme.
   Par exemple, si `D` et `F` sont les listes de l'exemple, `glouton1(D,F)` doit rendre:

   $$[[1, 5], [6, 11], [11, 15], [17, 21]]$$

2. Une stratégie gloutonne alternative consiste à choisir, au lieu du cours qui commence le plut tôt à chaque étape celui qui se termine le plus tôt parmi ceux qu'on peut accepter.

   (a) Pour faciliter son implémentation, il est donc nécessaire de trier les demandes non pas par horaire de début, mais par horaire de fin de façon croissante. En adaptant le tri à bulles d'une liste sont le code est rappelé sur le fichier de tests, donner une fonction `trieFin(D,F)` qui prend pour paramètres les listes `D` et `F` et qui rend le couple de ces deux listes de telle façon qu'elles correspondent toujours aux demandes des professeurs et que `F` soit triée de façon croissante.
   Par exemple, si `D` et `F` sont les listes de l'exemple, `trieFin(D,F)` doit rendre

   $$([1, 2, 7, 6, 6, 10, 11, 17], [5, 8, 8, 11, 12, 12, 15, 21])$$

   (b) En déduire une fonction `glouton2(D,F)` qui rend la liste des demandes attribuées pour cette salle par cet algorithme.
   Par exemple, si `D` et `F` sont les listes de l'exemple, `glouton2(D,F)` doit rendre:

   $$[[1, 5], [7, 8], [10, 12], [17, 21]]$$

Exercice 3: Empaquetage.

On considère un ensemble d'objets numérotés de 0 à $n-1$ de poids $\mathtt{p} = [\mathtt{p}_0, \mathtt{p}_1, \ldots, \mathtt{p}_{n-1}]$. On souhaite ranger l'ensemble de ces objets dans des boîtes identiques de telle manière que la somme des masses des objets contenus dans une boîte ne dépasse pas la capacité `C` de la boîte. On souhaite utiliser le moins de boîtes possibles pour ranger cet ensemble d'objets.
Pour résoudre ce problème, on utilisera un algorithme glouton consistant à placer chacun des objets dans la première boîte où cela est possible.
Par exemple, pour ranger dans des boîtes de capacité $\mathtt{C} = 5$ un ensemble de trois objets dont les masses sont représentées en Python par la liste $[1, 5, 2]$, on procède de la façon suivante :

- Le premier objet, de masse 1, va dans une première boîte.
- Le deuxième objet, de masse 5, ne peut pas aller dans la même boîte que le premier objet car cela dépasserait la capacité de la boîte. On place donc cet objet dans une deuxième boîte.
- Le troisième objet, de masse 2, va dans la première boîte.

1. Créer une fonction `empaquetage1` qui prend pour paramètres la liste `p` des poids des objets à empaqueter et l'entier `C` et qui rend une liste de liste `boites` où pour tout i, `boites`[`i`] désigne la liste des objets contenus dans la boîte numéro i.
   Si vous manquez de temps, vous pouvez vous contenter de rendre la nombre de boîtes utilisées. Attention, les tests du fichier de tests ne seront alors plus adaptés !

   Voici quelques exemples:

   ```
   >>> empaquetage1([1, 5,2],5)
   [[1,2],[5]]
   >>> empaquetage1([1, 5,3,2,2],5)
   [[1,3],[5],[2,2]]
   ```

2. Tester votre fonction pour $\mathtt{p} = [7, 6, 3, 4, 8, 5, 9, 2]$ et $\mathtt{C} = 11$. La solution proposée est-elle optimale ?
   Pour améliorer la fonction, on peut trier au préalable la liste des poids suivant l'ordre décroissant de manière à remplir les boîtes avec les objets les plus lourds d'abord.
   Proposer une fonction `empaquetage2` qui résout ce problème avec cette amélioration. Vérifiez qu'elle donne pour l'exemple ce cette question un meilleur résultat que `empaquetage1`.