Python - TP4
Algorithmes gloutons

Les problèmes d'optimisation (recherche d'une meilleure solution selon un critère bien défini) sont des problèmes courant en mathématiques et en informatique.

Définition 1 - Algorithme glouton

Un algorithme glouton est un algorithme dans lequel on procède étape par étape en faisant, à chaque étape, le meilleur choix possible (selon le critère défini initialement).

Important !

On ne remet jamais en cause les choix faits aux étapes passées.

On espère ainsi, qu'en faisant un choix optimal à chaque étape, la solution proposée soit également une solution optimale.

Petite remarque

C'est bien seulement une espérance... bien naïve ! En effet, rien ne garantit qu'un tel algorithme renvoie bien une solution optimale. On différencie parfois l'algorithme glouton (qui fournit bien une solution optimale) de l'heuristique gloutonne (qui procède de même, sans fournir une solution optimale).

I Problème du rendu de monnaie

L'algorithme de rendu de monnaie consiste en le choix, à chaque étape, de la plus grande pièce possible que l'on peut rendre, puis à réitérer sur la somme restante. C'est un exemple d'**algorithme glouton** qui permet de renvoyer le montant à rendre en **minimisant le nombre de pièces utilisées**. Nous avions également remarqué que l'algorithme ne renvoie pas toujours une solution optimale si le système de pièces est changé. Par conséquent, le fonctionnement de cet algorithme est basé sur le système de pièces choisi ; et il existe une démonstration prouvant que l'algorithme, pour un tel système de pièces, renvoie toujours la solution optimale recherchée.

Création de l'algorithme.

1. Créer une liste `L` contenant les différents montants des billets et pièces disponibles en euros.
2. Faire demander à l'utilisateur le prix à payer ainsi que la somme versée.
3. Créer une liste nommée `detail` telle que pour tout k, `detail[k]` stocke le nombre de billets/pièces correspondant au montant `L[k]`.
4. Faire afficher le détail à rendre de chaque billet/pièce.

```
import numpy as np

L=[500,200,100,50,20,10,5,2,1,0.5,0.2,0.1,0.05,0.02,0.01]

cout=float(input("Quel cout, en euros ? "))
somme=float(input("Quelle est la somme versee ? "))

rendu=somme-cout
detail=[]

for k in L:
    n=0
    while rendu>=k:
        rendu=rendu-k
        n=n+1
    detail.append(n)

print("Vous devez rendre : ")
for k in range(0,len(L)):
    if detail[k]!=0:
        if k<7:
            print(detail[k],"billets(s) de", L[k],"euros")
        else:
            print(detail[k],"piece(s) de", L[k],"euros")
```

II Problème des conférenciers

Des conférenciers sont invités à présenter leurs exposés dans une salle. Mais leurs disponibilités ne leur permettent d'intervenir qu'à des horaires bien définis. Le problème est de construire un planning d'occupation de la salle **avec le plus grand nombre de conférenciers**. Désignons par n, entier naturel non nul, le nombre de conférenciers. Chacun d'eux, identifié par la lettre C_i, avec $i \in [\![1;n]\!]$, est associé à un intervalle temporel $[d_i;f_i]$, où d_i et f_i désignent respectivement l'heure de début et l'heure de fin de l'intervention.

II.1 Choix de la stratégie

Afin de dégager une tactique de résolution du problème, commençons par analyser un cas particulier.
Trois conférenciers peuvent intervenir selon les créneaux ci-dessous :

Conférenciers	C_1	C_2	C_3
Créneaux	[12;15]	[13;14]	[14;16]

Question : comment le choix des conférenciers va-t-il se faire ?

II.2 Écriture de l'algorithme

Quinze conférenciers peuvent intervenir selon les créneaux ci-dessous :

Conférenciers	C_1	C_2	C_3	C_4	C_5	C_6	C_7	C_8	C_9
Créneaux	[12;19]	[14;17]	[18;20]	[13;14]	[17;18]	[12;14]	[16;19]	[12;13]	[15;18]

C_{10}	C_{11}	C_{12}	C_{13}	C_{14}	C_{15}
[13;15]	[16;17]	[18;20]	[12;14]	[17;19]	[13;16]

1. Trouver la solution optimale "à la main".

2. Écrire l'algorithme glouton permettant d'obtenir la solution optimale.

   Principe de l'algorithme glouton à mettre en place :

   2.a. Créer différentes listes contenant les identifiants des conférenciers, les heures de début et les heures de fin. ATTENTION : il faut ordonner ces listes par ordre d'heures de fin de conférence croissantes.

   2.b. Choisir alors le premier conférencier.

   2.c. Puis choisir le premier conférencier suivant, qui soit compatible avec le précédent.

   2.d. Réitérer.

Petite remarque

On admettra que l'algorithme dont le principe est détaillé ci-contre fournit bien une solution optimale...

```
fin=[13,14,14,14,15,16,17,17,18,18,19,19,19,20,20]
conf=['C8','C4','C6','C13','C10','C15','C2','C11','C5','C9','C1','C7','C14','C3','C12']
debut=[12,13,12,12,13,13,14,16,17,15,12,17,17,18,18]
planning=[conf[0]]
i=0
for k in range(1,len(conf)):
    if debut[k]>=fin[i]:
        planning.append(conf[k])
        i=k
print(planning)
```