TP n° 05 – Présentation des algorithmes gloutons avec le rendu de monnaie et l'empaquetage

I Introduction

Dans la résolution de problèmes numériques à l'aide d'algorithme. Nous avons pour le moment rencontré des problèmes pour lesquels la solution était unique. Cependant, dans certains cas, un problème peut avoir plusieurs solutions. (ex : Trouver la valeur minimale de la fonction $sin(x)$). Il est alors possible de donner des contraintes à cette solution (ex : $x \in [0, 2 \cdot \pi]$) ou de chercher une solution optimale en ajoutant un critère d'optimisation (ex : la solution la plus près de 42).
Nous allons ici étudier une méthode de résolution pour ce type d'exercice, la méthode des **Algorithmes Gloutons**.
L'exemple que nous allons utiliser ici est celui du rendu de monnaie.
Nous allons pour cela dans un premier temps nous placer dans le contexte du rendu de monnaie en euro pour des sommes entières. Afin de simplifier le sujet, nous appellerons *pièce* les pièces et les billets, ainsi, nous pourrons parler d'une pièce de 10€.

II Calcul d'un montant à partir d'une liste de pièces.

Exercice 1.
Créer une liste `monnaie_euro`, contenant l'ensemble des pièces (dont le montant est un entier) du système monétaire européen dans l'ordre décroissant.

Solution 1.

```
monnaie_euro=[500,200,100,50,20,10,5,2,1]
```

Exercice 2.
Créer une fonction `montant(monnaie,liste)` qui renvoie le montant d'une liste de pièces dans un système monétaire. Par exemple, `montant(monnaie_euro,[0,0,0,1,0,1,0,2,0])` doit renvoyer `64`.) Afficher la valeur donnée pour la liste proposée en exemple.

Solution 2.

```
def montant(monnaie,liste):
    somme=0
    for idx,piece in enumerate(liste):
        somme+=piece*monnaie[idx]
    return somme

liste_monnaie=[0,0,0,1,0,1,0,2,0]
print(montant(monnaie_euro,liste_monnaie))
```

Exercice 3.
Tester la fonction précédente sur la liste `[0,0,0,0,2,2,0,1,2]`. Qu'en conclure ?

Solution 3.

```
liste_monnaie=[0,0,0,0,2,2,0,1,2]
print(montant(monnaie_euro,liste_monnaie))
```

Il y a plusieurs solutions pour obtenir le montant de 64€avec le système monétaire européen.

III Optimisation du rendu de monnaie

Comme nous l'avons vu à la question précédente, il y a plusieurs combinaisons de pièces qui permettent d'obtenir une somme en euros. Dans le cadre du rendu de monnaie, nous sommes donc face à un problème

d'optimisation. Nous allons donc ajouter une contrainte pour y répondre : nous souhaitons rendre la monnaie avec le moins de pièces possible.
Nous allons dans cet objectif utiliser un algorithme Glouton. Comme son nom l'indique, cet algorithme tente d'obtenir l'optimum de la solution en cherchant un optimum à chaque itération. Il va donc essayer de converger le plus vite possible sans se demander s'il n'existe pas un chemin plus long mais plus efficace. Ainsi dans le cadre du rendu de monnaie, cet algorithme va proposer quand il le peut d'utiliser la plus grande pièce possible. Nous supposons qu'il est nécessaire de rendre une somme S en euros. Au fur et à mesure que l'on choisit des pièces pour rembourser cette somme, le restant dû est donc r_i, cette valeur diminue au fur et à mesure et $r_0 = S$.
Le pseudo-code de l'algorithme est donc le suivant :

- tant que le restant dû est > 0 ou $\neq 0$
  - si la pièce avec l'indice i, dans la liste `monnaie_euro`, est plus petite que r_i, on choisit cette pièce : on ajoute la pièce à la liste des pièces utilisées et on calcule la nouvelle valeur du reste dû,
  - si la pièce avec l'indice i, dans la liste `monnaie_euro`, est plus grande que r_i, on incrémente i et on revient à l'étape précédente.
- On retourne la liste des pièces à rendre.

Exercice 4.
Créer la fonction `rendre_monnaie_euro(restant_du)` correspondant au pseudo-code précédent. Utiliser cette fonction pour déterminer la liste des pièces nécessaire au remboursement de la somme de 242€.

Solution 4.

```
def rendre_monnaie_euro(restant_du):
    monnaie_euro=[500,200,100,50,20,10,5,2,1]
    monnaie_a_rendre=[0]*len(monnaie_euro)
    i=0
    while restant_du>0:
        if restant_du>=monnaie_euro[i]:
            monnaie_a_rendre[i]+=1
            restant_du+=-monnaie_euro[i]
        else:
            i+=1
    return(monnaie_a_rendre)

print(rendre_monnaie_euro(242))
```

Exercice 5.
Utiliser la fonction `montant(monnaie,liste)` afin de vérifier que la liste générée précédemment donne bien la somme demandée.

Solution 5.

```
print(montant(monnaie_euro,rendre_monnaie_euro(242)))
```

Exercice 6.
Créer une fonction `nombre_de_piece(liste)` qui a partir d'une liste de pièces retourne le nombre de pièces utilisées. (Ex :`nombre_de_piece([0,0,0,1,0,1,0,2,0])` doit retourner 4.) Tester cette fonction sur la liste précédente.

Solution 6.

```
def nombre_de_piece(liste):
    somme=0
    for piece in liste:
        somme+=piece
    return somme
print(nombre_de_piece(rendre_monnaie_euro(242)))
```

IV Utilisation d'un algorithme Glouton sur un ensemble non canonique

Le système monétaire est adapté au rendu de monnaie et par conséquent l'algorithme Glouton renvoie toujours la solution optimale. On dit que ce système est canonique.
Nous allons maintenant imaginer l'utilisation d'un système pour lequel la liste des pièces serait : `monnaie_non_canonique=[9,7,3,1]`.

Exercice 7.
Refaire la question 4 et déterminer la liste des pièces pour rembourser 14€.

Solution 7.

```
def rendre_monnaie_non_canonique(montant_du):
    monnaie_non_canonique=[9,7,3,1]
    monnaie_a_rendre=[0]*len(monnaie_non_canonique)
    i=0
    while montant_du>0:
        if montant_du>=monnaie_non_canonique[i]:
            monnaie_a_rendre[i]+=1
            montant_du+=-monnaie_non_canonique[i]
        else:
            i+=1
    return(monnaie_a_rendre)
```

Exercice 8.
La solution proposée par l'algorithme Glouton est elle optimale ? Proposer une nouvelle distribution de pièces si ce n'est pas le cas.

Solution 8.

```
monnaie_non_canonique=[9,7,3,1]

print("Dans la liste {}, il y a {} pièces, c'est mieux que le résultat précédent"\
        .format([0,2,0,0],nombre_de_piece([0,2,0,0])))

print("En donnant les pièces suivantes: {}, on rembourse {}\euro."\
        .format([0,2,0,0],montant(monnaie_non_canonique,[0,2,0,0])))
```

Selon le choix du système de monnaie, l'algorithme Glouton peut aussi renvoyer une solution fausse.

Exercice 9. A faire à la main.
Dans cet exercice, la liste de pièces est : `monnaie_faux=[6,5]`. On veut rembourser 15€. Que va renvoyer l'algorithme Glouton ? Quel est en fait le bon rendu de monnaie ?

Solution 9. Par l'algorithme Glouton, on ne peut pas faire un rendu de monnaie exact, puisqu'il rend comme pièces : 6 puis 6 et il reste 3.
La bonne réponse comme rendu est : 5 puis 5 puis 5.

V Empaquetage

On a n objets dont on connaît le poids, rassemblés dans une liste du type $[\text{poids}_1, \cdots, \text{poids}_n]$. On a plusieurs boites qui ont toutes la même capacité : C. L'objectif est de mettre tous les objets dans des boites, en respectant la limite de poids de chaque boite et en optimisant le nombre de boites utilisées.
Par exemple, on a 6 objets de poids respectifs $[7, 6, 3, 4, 8, 5]$ et des boites de capacité maximale $C = 11$.
La première méthode Glouton pour remplir les boites consiste à prendre les objets un par un et de les mettre dans la première boite qui puisse les contenir. Dans ce cas, on obtient :
Boite n°1 : 7,3
Boite n°2 : 6,4
Boite n°3 : 8
Boite n°4 : 5.

Exercice 10. Écrire une fonction `empaqueter` qui prend comme entrée une liste de poids L et une capacité C et qui renvoie la liste des boites contenant les poids des objets qu'elles contiennent.
Par exemple : `empaqueter([7,6,3,4,8,5],11)` renvoie `[[7,3] [6,4], [8],[5]]`.

Solution 10.
```
def empaqueter(Liste_objets,C):
    Boite=[[]]
    Capacite_Boites=[C]
    for objet in Objets:
        numero_boite=0
        while numero_boite<len(Boite) and objet>Capacite_Boites[numero_boite]:
            numero_boite=numero_boite+1
        if numero_boite>=len(Boite):
            Boite.append([objet])
            Capacite_Boites.append(C-objet)
        else:
            Boite[numero_boite].append(objet)
            Capacite_Boites[numero_boite]-=objet
    return(Boite)
```

Exercice 11. Pour optimiser la méthode précédente, on peut à chaque étape ranger l'objet le plus lourd. Écrire une fonction `empaqueter2` qui renvoie le rangement ainsi optimiser. On pourra utiliser la méthode `sort(reverse=True)`.

Solution 11. `empaqueter(Objets.sort(reverse=True),C)`

Exercice 12. A la main.
La méthode précédente n'est pas optimale. Que renvoie notre algorithme Glouton avec : $L = [6, 5, 5, 3, 3, 2]$ et $C = 12$?
Donner un rangement qui utilise moins de boites.

Solution 12. `[[6,5],[5,3,3],[2]]` est moins optimal que : `[[6,3,3],[5,5,2]]`.