Algo 07 – Stratégie gloutonne

I – Principe

Certains problèmes sont DIFFICILES (coût exponentiel ou plus) à résoudre si on calcule toutes les possibilités pour récupérer la meilleure.

La **stratégie gloutonne** consiste à faire un choix local et définitif sans regarder la totalité du problème : on réduit donc la taille du problème. On continue de faire un choix jusqu'à avoir traité toutes les données.
On espère alors trouver une solution totale pas trop mauvaise.

Cela ne peut s'appliquer qu'aux **problèmes d'optimisation** : les problèmes qui ont plusieurs solutions possibles : de bonnes, des moyennes et des mauvaises.

II – Quelques exemples (voir le site pour plus de détails)

→ **Voyageur de commerce** : trouver un cycle entre plusieurs villes sans repasser par les villes déjà visitées en faisant le moins de km.

Stratégie gloutonne :
→ Choisir un point de départ au hasard.
→ Tant qu'on a pas visité toutes les villes, choisir la prochaine ville la plus proche non encore visitée.
→ On repart ensuite jusqu'à la ville de départ.

→ **Rendu du monnaie** : rendre de la monnaie en limitant le nombre de pièces rendues.

Stratégie gloutonne :
→ Tant que la somme à rendre n'est pas 0, choisir la prochaine pièce la plus grande possible pour réduire la somme à rendre.

→ **Problème du sac à dos** : comment remplir au mieux un sac de X litres connaissant une liste d'objets comportant le nom de l'objet, son prix et son volume.

Stratégie gloutonne :
→ Rajouter une colonne à la liste en calculant le rapport prix/volume de chaque objet.
→ Tant qu'il reste assez de place dans le sac pour rajouter un objet, place l'objet qui a le meilleur prix volumique ET qui rentre dans le sac.

CONLUSION : de nombreux pb sont résolus par cette méthode pour laquelle on ne revient jamais sur les décisions locales qu'on réalise. Le choix doit donc être un choix local qui logiquement devrait amener à une solution pas trop mauvaise. On parle d'heuristique.

III – Suppléments TABLEAUX

TABLEAUX DYNAMIQUES

Les tableaux Python sont dynamiques en réalité : on peut faire varier leurs nombres de cases au fur et à mesure.

La méthode **append(x)** permet de rajouter une case contenant x à la fin du tableau.
La méthode **pop()** permet de supprimer la dernière case et renvoie le contenu de cette case.

```
>>> t = [5]
>>> t.append(15)          → t référence [5, 15]
>>> t.append(3)           → t référence  [5, 15, 3]
>>> v = t.pop()           → t référence [5,  15] ET v référence 3.
```

Utiliser de cette façon, elles sont à coût CONSTANT.

ATTENTION : pop(0)

```
>>> t = [5, 10, 20, 30, 40]
>>> t.pop(0)        supprime la 1er case mais à coût LINEAIRE !
>>> t
[10, 20, 30, 40]
```

La méthode remove(x) supprime la 1er occurrence de x dans le tableau. Son action est également à coût LINEAIRE.

```
>>> t = [5, 10, 20, 30, 40, 10]
>>> t.remove(10)   → t référence [5, 20, 30, 40, 10], le 1er 10 a disparu.
```

CREATION PAR COMPREHENSION

```
>>> t = [x for x in range(5)]
>>> t
[0, 1, 2, 3, 4]
```

Traduction : Crée un nouveau tableau t où
pour chaque x compris entre 0 et 4
on crée une case contenant x

```
>>> t = [10*x for x in range(5)]
>>> t
[0, 10, 20, 30, 40]
```

Traduction : Crée un nouveau tableau t où
pour chaque x compris entre 0 et 4
on crée une case contenant 10*x

```
>>> t = [5, 12, 8, 15, 18, 3]
>>> t2 = [v for v in t if v >= 10]
>>> t2
[12, 15, 18]
```

Traduction : Crée un nouveau tableau t2 où
pour chaque valeur contenant dans le tableau t
si cette valeur est supérieure ou égale à 10
on crée une case contenant v

```
>>> t = [5, 12, 8]
>>> t2 = [2*v for v in t]
>>> t2
[10, 24, 16]
```

Traduction : Crée un nouveau tableau t2 où
pour chaque valeur contenant dans le tableau t
on crée une case contenant 2*v

IV – Rendu de monnaie en Python et utilisation de doctest

```
1    les_choix = [100, 50, 20, 10, 5, 2, 1]     # Variable globale
2
3    def rendre(a_rendre, choix=les_choix):     # Déclaration avec valeur par défaut
4        reponse = []                           # On crée le tableau des réponses
5        reste = a_rendre                       # Il reste tout à rendre au début
6        i = 0                                  # Commence par la plus grande valeur
7        while reste > 0:                       # Tant qu'il reste des sous à rendre
8            if reste >= choix[i]:              # SI la pièce actuelle convient
9                reponse.append(choix[i])       # Rajoute cette pièce au tableau
10               reste = reste - choix[i]       # Modifie le reste à rendre
11           else:                              # SINON c'est qu'elle est trop grande
12               i = i + 1                      # Passe à la pièce suivante
13       return reponse
```

Remarques explicatives : **`def rendre(a_rendre, choix=les_choix)`**
Dans cette fonction, le paramètre **`choix`** possède une valeur par défaut (**`choix=les_choix`**). Cela veut dire qu'on peut choisir :
→ d'envoyer un argument pour **choix** (appel du type **`rendre(18, [5,2,1]`**)
→ de ne rien lui envoyer (appel du type **`rendre(18`**). Dans ce cas, **choix** sera initialisé avec **`les_choix`**, la valeur par défaut.

Remarque L6 : comme les valeurs sont triées en commençant par la plus grande, en choisissant de commencer par la plus grande, on indique bien de rendre la plus grande pièce possible.

Bien entendu, cette fonction n'est absolument pas bien documentée et testée. VERSION FINALE :

```
1    les_choix = [100, 50, 20, 10, 5, 2, 1]     # Variable globale
2
3    def rendre(a_rendre, choix=les_choix):
4        """Renvoie le tableau des pièces pour rendre la monnaie
5
6        :: param a_rendre(int)     :: la somme qu'elle vaut rendre
7        :: param choix (list[int]) :: le tableau des pièces qu'on peut rendre
8        :: return (list[int])      :: le tableau des pièces à rendre
9
10       ..exemples..
11       >>> rendre(15)
12       [10, 5]
13       >>> rendre(15, les_choix)
14       [10, 5]
15       >>> rendre(17)
16       [10, 5, 2]
17
18       """
19       reponse = []                           # On crée le tableau des réponses
20       reste = a_rendre                       # Il reste tout à rendre au début
21       i = 0                                  # Commence par la plus grande valeur
22       while reste > 0:                       # Tant qu'il reste des sous à rendre
23           if reste >= choix[i]:              # SI la pièce actuelle convient
24               reponse.append(choix[i])       # Rajoute cette pièce au tableau
25               reste = reste - choix[i]       # Modifie le reste à rendre
26           else:                              # SINON c'est qu'elle est trop grande
27               i = i + 1                      # Passe à la pièce suivante
28       return reponse
29
30   if __name__ == "__main__":   # Si c'est bien le prog. Principal et pas un module
31       import doctest           # Importation du module doctest
32       doctest.testmod()        # Lancement des exemples comme tests automatiques
```