Algorithmes gloutons

Issu du latin impérial glut(t)onem, « glouton », dérivé de glut(t)us, « gosier » : Qui mange avec avidité et excès, voracement.
Un enfant glouton, une faim gloutonne, cet homme est un glouton, une jeunesse gloutonne de plaisirs...etc
N. m. Mammifère carnivore de la famille des Mustélidés, un glouton vit essentiellement dans les régions arctiques.

I- Introduction :

La résolution d'un problème algorithmique peut parfois se faire à l'aide de techniques générales , des « paradigmes», qui sont selon le philosophe des sciences Thomas Samuel Kuhn des découvertes scientifiques universellement reconnues qui, pour un temps, fournissent à un groupe de chercheurs des problèmes types et des solutions. Ces techniques ont pour avantage d'être applicables à un grand nombre de situations.
Parmi ces méthodes on peut citer par exemple le fameux principe **"diviser pour régner"** :

1. **Diviser** : on divise les données initiales en **plusieurs sous-parties**.
2. **Régner** : on **résout récursivement** chacun des sous-problèmes associés (ou on les résout directement si leur taille est assez petite)
3. **Combiner** : on **combine** les différents résultats obtenus pour obtenir une solution au problème initial.

Ou encore la méthode de " **la programmation dynamique"**. Pour ce faire, et donc éviter de recalculer plusieurs fois les solutions des mêmes sous-problèmes, on va **mémoriser** ces solutions dans une sorte de **mémoire cache**. Celle-ci sera un **tableau** ou **liste**, selon le langage d'implémentation utilisé, et possédera une ou deux dimensions suivant les cas. Sa mise en pratique peut prendre **deux formes** :

1. Une forme **récursive "Top down"** dite de **mémoïsation** :
    - On utilise directement la **formule de récurrence**.
    - Lors d'un appel récursif, **avant d'effectuer un calcul on regarde dans le tableau de mémoire cache** si ce travail n'a pas déjà été effectué.
2. Une forme **itérative "Bottom Up"** :
    - On résout d'abord les sous problèmes de la plus "petite taille", puis ceux de la taille "d'au dessus", etc. Au fur et à mesure on stocke les résultats obtenus dans le **tableau de mémoire cache**.
    - On continue jusqu'à la taille voulue.

Les algorithmes gloutons, que l'on rencontre principalement pour résoudre des problèmes d'optimisation, constituent l'une de ces techniques générales de résolution.

II- Définition et principe:

Lors de la résolution d'un problème d'optimisation, la construction d'une solution se fait souvent de manière séquentielle, l'algorithme faisant à chaque étape un certain nombre de choix. Le principe glouton consiste à faire le choix qui semble le meilleur sur le moment (choix local), sans se préoccuper des conséquences dans l'avenir, et sans revenir en arrière.
Un algorithme glouton est donc un algorithme qui ne se remet jamais en question et qui se dirige le plus rapidement possible vers une solution. Aveuglé par son appétit démesuré, le glouton n'est pas sûr d'arriver à une solution optimale, mais il fournit un résultat rapidement. Même si la solution n'est pas optimale, il n'est pas rare que l'on s'en contente, on rentre alors dans le monde des algorithmes d'approximation.
Pour que la méthode gloutonne ait une chance de fonctionner, il faut que le choix local aboutisse à un problème similaire plus petit. La méthode gloutonne ressemble ainsi à la programmation dynamique. Mais la différence essentielle est que l'on fait d'abord un choix local et on résout ensuite un problème plus petit (progression descendante). En programmation dynamique, au contraire, on commence par résoudre des sous-problèmes, dont on combine ensuite les résultats (progression ascendante).

III- Le problème du rendu de monnaie :

1. Position du problème :

On considère un **système de pièces de monnaie**.

La question est la suivante : quel est le **nombre minimal de pièces** à utiliser pour **rendre une somme donnée** ? De plus, quelle est la **répartition des pièces** correspondante ?

La valeur de la solution optimale sera ce nombre minimal de pièces, et la solution en elle-même la liste des pièces nous permettant de rendre la somme.

Formalisons un peu ce problème avec quelques **notations mathématiques** :

- Le système de pièces de monnaie peut être modélisé par un **n-uplet d'entiers naturels**

$S=(c_1,c_2,...,c_n)$, où c_i représente la valeur de la pièce ;

- On suppose que $c_1=1$ et que $c_1<c_2<...<c_n$;
- Une somme à rendre est un **entier naturel** X ;
- Une répartition de pièces est un **n-uplet d'entiers naturels** $(x_1,x_2,...,x_n)$, où x_1 représente le nombre de pièces c_1, x_2 le nombre de pièces c_2, ...etc. Le nombre total de pièces d'une telle répartition est donc $\sum_{i=1}^{i=n} x_i$.

On peut reformuler la question comme suit :

pour tout entier naturel X, on cherche un n-uplet d'entiers naturels $(x_1,x_2,...,x_n)$ qui **minimise** $\sum_{i=1}^{i=n} x_i$ **sous la contrainte** $\sum_{i=1}^{i=n} x_i.c_i = X$

2. Remarques :

a-L'hypothèse c1=1 garantit qu'un rendu est toujours possible puisque les sommes à rendre sont entières.

b-L'égalité $\sum_{i=1}^{i=n} x_i.c_i = X$ signifie juste que l'on rend la bonne somme.

c- $\sum_{i=1}^{i=n} x_i = x_1 + x_2 + ... + x_n$.

3. Exemple (Système de monnaie européen) :

Dans la zone euro, le système actuellement en circulation est S=(1,2,5,10,20,50,100,200,500),

100 centimes = 1€, 200 centimes = 2€, 500 centimes = billet de 5€ .

Avec les notations précédentes, on a ainsi $c_1 = 1$, $c_2 = 2$, $c_3 = 5$,..., $c_9 = 500$. Et bien sûr n = 9.

Pour rendre par exemple une somme de X = 6 euros, on doit considérer les n-uplets $(x_1,x_2,...,x_9)$ vérifiant $\sum_{i=1}^{i=9} x_i = 6$.

Pour $i \geq 4$ on a nécessairement $x_i = 0$ car les pièces correspondantes ont une valeur plus grande que la somme à rendre. La contrainte devient donc $x_1+2x_2+5x_3=6$

et il y a cinq triplés qui la vérifient :

- (6,0,0)
- (4,1,0)
- (2,2,0)
- (1,0,1)
- (0,3,0)

Le triplet qui minimise $x_1+x_2+x_3$ est alors la solution, il s'agit en l'occurrence de (1,0,1). Il faut ainsi deux pièces pour rendre une somme de 6 euros, une pièce de 1 et un billet de 5.

Pour terminer cette sous-partie, demandons-nous ce qu'un **humain** ferait dans une telle situation. Il commencerait sans doute par rendre la **plus grande pièce "possible"**, puis ferait de même avec le reste jusqu'à ce que la somme soit rendue. C'est d'ailleurs ce que font des millions de commerçants quotidiennement.

D'un **point de vue algorithmique** cela donne :

1. Choisir la plus grande pièce du système de monnaie inférieure ou égale à la somme à rendre.
2. Déduire cette pièce de la somme.
3. Si la somme n'est pas nulle recommencer à l'étape 1.

Ce point de vue est un algorithme glouton.

Cette méthode est séduisante, car simple, mais **malheureusement pas toujours optimale**.Voici un contre-exemple :

Si l'on doit rendre la somme de 6 avec le système (1,2,5), la méthode précédente fournit un résultat optimal à savoir un billet de 5 puis une pièce de 1, *i.e.* = deux pièces.

Par contre, pour rendre cette même somme avec le système (1,3,4) il n'y a pas optimalité. En effet on rendra d'abord une pièce de 4, puis une pièce de 1 et enfin une autre pièce de 1, c'est-à-dire trois pièces. Or on pouvait rendre de façon plus performante deux pièces de 3.

Dans **le système de pièces européen**, on peut montrer que l'algorithme glouton **donne toujours une solution optimale.**

4. Pseudo-code et code Python :

a- En pseudo-code, on peut écrire :

```
n ← longueur de la liste pieces
pour i allant de 0 à n-1
  Tant que somme >= pieces[i]
    somme ← somme – pieces[i]
    choisies[i] ← choisies[i] + 1
renvoyer choisies
```

Où :

pieces : les types de pièces possibles, une liste d'entiers classées dans le sens décroissant.

Somme : la somme à rendre, un entier.

Choisies : une liste indiquant le nombre de pièces choisies pour chacune des pièces.

b- Code python :

On suppose que l'on dispose d'un nombre de pièces illimité et que l'on **a** 8€ à rendre , c-à-d 800 centimes, dans le système européen de monnaie :

```
def rendu_monnaie(somme,pieces) :
    n = len(pieces)
    choisies = [0]*n
    for i in range(n) :
        while somme >= pieces[i]
            somme = somme – pieces[i]
            choisies[i] = choisies[i] + 1
    return choisies
pieces = [500,200,100,50,20,10,5,2,1]
somme = 800
print("Les pièces choisies sont :")
print(rendu_monnaie(somme,pieces))
```

La résultat dans le shell est :

```
Les pièces choisies sont :
[1,1,1,0,0,0,0,0,0]
```

Il faut donc rendre :

Un billet de 5€,une pièce de 2€ et une pièce de 1€.

Supposons que l'on veuille rendre 6 unités d'un système fictif de monnaie , et, que dans notre caisse il ne reste que des pièces de 4 unités, 3 unités et 1 unités. On teste l'algorithme avec :

```
Pieces = [4,3,1]
somme = 6
print("Les pièces choisies sont :")
print(rendu_monnaie(somme,pieces))
```

La réponse dans Le shell est :

```
Les pièces choisies sont :
[1, 0, 2]
```

L'algorithme nous propose de rendre une pièce de 4 unités et deux pièces de 1 unité. Mais on voit bien qu'on aurait pu rendre deux pièces de 3 unités qui est un rendu plus optimal !

A.LOUGHANI

c- Test de satisfaction du client :

Il faut toujours s'assurer que le compte est bon et que le rendu au client est exact, donc la somme finale à rendre, c-à-d à la fin de l'algorithme est somme=0.

Pour cela on peut faire un test en fin de boucle for dans l'algorithme précédent :

```
def rendu_monnaie(somme,pieces) :
    n = len(pieces)
    choisies = [0]*n
    for i in range(n) :
        while somme >= pieces[i]
            somme = somme – pieces[i]
            choisies[i] = choisies[i] + 1
    assert somme == 0
    return choisies
```

On peut tester cet algorithme pour une somme à rendre de 31€ à partir des pièces 10€ ,5€ et 2€ et voir la réponse dans le shell.

IV- Le problème du sac à dos :

1. Historique :

Le problème du sac à dos est l'un des 21 problèmes NP-complets de Richard Karp, exposés dans un article de 1972. La formulation du problème est fort simple, mais sa résolution est plus complexe.

Richard Manning Karp
Né le 3 Janvier 1935
à Boston
Américain
Université Havard
Mathématicien,informaticien etprofesseur d'université

Étant donné plusieurs objets possédant chacun un poids et une valeur et étant donné un poids maximum pour le sac, quels objets faut-il mettre dans le sac de manière à maximiser la valeur totale sans dépasser le poids maximal autorisé pour le sac ?

2. Formulation du problème :

On commence par un énoncé des données. Dans notre cas, nous avons un sac à dos qui peut contenir un poids maximal P et n objets au plus. Pour chaque objet i, nous avons un poids p_i et une valeur v_i.
Par exemple, si on a quatre objets et un sac à dos d'un poids maximal de 30 kg alors on a $n = 4$ et $P = 30$, l'indice ,entier i, varie de 1 à 4. On résume cela dans le tableau suivant :

Objets	1	2	3	4
La valeur v_i	7	4	3	3
Le poids p_i	13	12	8	10

Ensuite, on définit les variables qui représentent en quelque sorte les actions ou les décisions qui amèneront à trouver une solution. On prend alors la variable x_i associée à un objet i telle que :

$x_i = 1$ si l'objet i est mis dans le sac, et $x_i = 0$ si l'objet i n'est pas mis dans le sac.

Dans notre exemple, une solution réalisable est de mettre tous les objets dans le sac à dos sauf le premier, nous avons donc :

$$x_1 = 0, x_2 = 1, x_3 = 1, \text{ et } x_4 = 1.$$

Puis il faut définir les contraintes du problème. Ici, il n'y en a qu'une : la somme des poids de tous les objets dans le sac doit être inférieure ou égale au poids maximal du sac à dos.
Cela s'écrit ici :

$$x_1.p_1 + x_2.p_2 + x_3.p_3 + x_4.p_4 \leq P$$

et pour n objets :

$$\sum_{i=1}^{i=n} x_i.p_i \leq P$$

Pour vérifier que la contrainte est respectée dans notre exemple, il suffit de calculer cette somme : $0 \times 13 + 1 \times 12 + 1 \times 8 + 1 \times 10 = 30$, ce qui est bien inférieur ou égal à 30, donc la contrainte est respectée. Nous parlons alors de solution réalisable. Mais ce n'est pas nécessairement la meilleure solution.
Enfin, il faut exprimer la fonction qui traduit notre objectif qui est de maximiser la valeur totale des objets dans le sac c-à-d la somme $x_1.v_1 + x_2.v_2 + x_3.v_3 + x_4.v_4$ doit être la plus grande possible. Pour n objets, cela s'écrit :

$$\max \sum_{i=1}^{i=n} x_i.v_i$$

Dans notre exemple, la valeur totale contenue dans le sac est égale à 10. Cette solution n'est pas la meilleure, car il existe une autre solution de valeur plus grande que 10. En effet, il suffit de prendre seulement les objets 1 et 2 qui donneront une valeur totale de 11. Il n'existe pas de meilleure solution que cette dernière, nous dirons alors que cette solution est optimale.

3.Fonction remplir_sac :

a- Pseudo-code :

On dispose d'une liste de valeurs des objets et de leurs poids du type :
objets = $[[v_i, p_i]]_i$
= $[[v_1,p_1],[v_2,p_2],...,[v_n,p_n]]$
= [[*valeur , poids*],[*valeur , poids*],...]
On note le poids du sac P et le poids maximal poids_max.
On peut donc écrire le pseudo-code :

```
Trier la liste objets par ordre croissant selon la valeur de chaque objet
P ← 0
n ← nombre d'objets
objets_choisis = [0]*n
pour i allant de 0 à n-1
   si P + objets[i][1] <= poids_max alors
     objet_choisis[i] = 1
     P ← P + objet[i][1]
Renvoyer objet_choisis
```

b- Code Python :

Avant de lancer la fonction remplir_sac , on trie d'abord la liste des objets selon les valeurs décroissantes puis on remplit le sac.
La fonction remplir_sac est :

```
def  remplir_sac(objets,poids,poids_max) :
     P = 0
     n = len(objets)
     objets_choisis = [0]*n
     for i in range(n) :

         if P + objets[i][1] <= poids_max :

             objets_choisis[i] = 1

             P = P + objets[i][1]

     return objets_choisis
```

Prenons un cas, résumé par le tableau de valeurs ci-dessous :

Objets	1	2	3	4
La valeur v_i	2	5	1	3
Le poids p_i (en kg)	1	0.5	0.2	4

On suppose poids_max = 5, d'un autre côté on a objets = [[2,1],[5,0.5],[1,0.2],[3,4]] . On trie la liste objet et on lance la fonction remplir_sac(objets,poids_max) :

```
objets = [[2,1],[5,0.5],[1,0.2],[3,4]]
objets = list(reversed(sorted(objets)))
print(objets)
poids_max = 5.0
print("Les objets choisis sont :")
print(remplir_sac(objets,poids_max))
```

Dans le shell, on voit la liste triée d'abord ensuite les objets choisis :

```
[[5,0.5],[3,4],[2,1],[1,0.2]]
Les objets choisis sont :
[1,1,0,1]
```

On retient donc :
l'objet2 de valeur 5, l'objet4 de valeur et l'objet3 de valeur 1.

Références :

- Supinfo.com
- Cécile Canu,NSI 1re TRAVAIER EN AUTONOMIE , Nouveau programme, ellipse
- Pierre Begian, Begian.free.fr
- Schoolmouv.fr
- *https://fr.wikipedia.org › wiki › Mustelidae*