Algorithmes Gloutons

1) Un premier exemple : le rendu de monnaie

Rendu de monnaie

Étant donné un système de monnaie (pièces et billets), comment rendre une somme donnée de façon optimale, c'est à dire avec le nombre minimal de pièces et de billets ?

1. On souhaite acheter des livres pour un montant 41 euros, en payant avec un billet de 50 euros.

   Le libraire doit nous rendre 9 euros.

   La valeur des pièces et billets mis à sa disposition sont : 1, 2, 5, 10, 20, 50, 100 et 200 euros, et il dispose d'autant d'exemplaires qu'il le souhaite de chaque pièce et billet.

   Indiquez toutes les façons que le libraire a de rendre la monnaie.

   Quelle est celle qui est optimale ?

2. On souhaite acheter un livre supplémentaire qui coûte 23 euros, en payant avec un billet de 50 euros.

   Le libraire doit nous rendre 27 euros.

   Comment le libraire va-t-il nous rendre la monnaie pour minimiser le nombre de pièces et de billets ? Proposez une solution (aucune démonstration n'est demandée.)

   Remarque :

   Le fait d'énumérer toutes les possibilités pour trouver celle qui demande le moins de pièces est appelée méthode par **"force brute"**. Elle permet de trouver la **solution optimale globale** au problème, mais dès que la somme à rendre augmente, le nombre de possibilités devient vite très important. L'utilisation d'un tel algorithme dans le cas de ce problème est **très coûteux en temps** de calcul !

   Problèmes d'optimisation

   Les **problèmes d'optimisation** sont des problèmes algorithmiques dans lesquels l'objectif est de trouver **une solution "la meilleur possible"** selon un certain critère, parmi un ensemble de solutions également valides mais potentiellement moins avantageuses.

   Face à des problèmes d'optimisation impossibles à explorer exhaustivement (donner toutes les solutions possibles), il peut être utile de connaître **des algorithmes donnant rapidement une réponse qui, sans être nécessairement optimale globalement, resterait bonne**.

   La **méthodologie gloutonne** donne une approche simple pour concevoir de tels algorithmes. L'algorithme Glouton n'aboutit pas toujours à la meilleure solution, mais il en donne une bonne approximation et il est facile à mettre en oeuvre.

3. Pour mettre en oeuvre l'algorithme glouton pour résoudre notre problème, on va **sélectionner** les pièces et les billets à rendre **un à un**, et **faire décroître** progressivement la somme restant à rendre.

   Chaque choix, doit être celui qui paraît être **le meilleur au vu de la somme restant à rendre**. Pour limiter le nombre de pièces (ou billets) à rendre, on choisit de faire décroître cette somme **aussi vite que possible**, à chaque fois on va **sélectionner la plus grande valeur disponible** qui ne soit pas strictement supérieure à la somme à rendre.

On propose l'algorithme glouton suivant :

Rendu de monnaie

entrées : jeu_pieces est un tableau constitué des montants disponibles des pièces (ou billets) classés dans l'ordre décroissant des valeurs.
S est une somme entière à rendre
sorties : rendu est un tableau contenant toutes les pièces (ou billets) à rendre
nb_pieces est un entier indiquant le nombre de pièces et billets utilisés pour rendre S

```
initialiser le tableau rendu au vide
initialiser nb_pieces à 0
tant que S > 0
    on choisi dans jeu_pieces la plus grande valeur qui ne dépasse pas S
    on enlève cette valeur à S
    on ajoute cette valeur à rendu
    nb_pieces augmente du 1
retourner rendu et nb_pieces
```

(a) Déroulez cet algorithme **"à la main"** avec les données suivantes, et précisez à chaque étape les valeurs de **rendu** et de **somme**.

Quel est le résultat obtenu ?

★ $s = 27$ et jeu_pieces $= [200, 100, 50, 20, 10, 5, 2, 1]$

★ $s = 63$ et jeu_pieces $= [200, 100, 50, 20, 10, 5, 2, 1]$

(b) On va implémenter cet algorithme en Python, en créant une fonction **rendu_monnaie()**.

Précisez les spécifications de votre fonction (sa documentation en quelques mots).

(c) Commencez par écrire des **tests** que devra passer cette fonction.

(d) Codez le corps de la fonction et faites-lui passer les tests.

Si besoin, corrigez-là !

Remarque :

Regardons **la complexité** (temps d'exécution) dans le pire des cas, c'est à dire lorsque seule la pièce de 1€ est disponible :

à chaque étape de l'algorithme on soustrait la valeur de la pièce à la somme s, il faudra donc s étapes.

La **complexité** est donc **de l'ordre de** s, la somme à rendre, au pire des cas.

(e) Il est possible d'améliorer cet algorithme.

Pour cela on va calculer **la multiplicité** de chaque pièce, c'est-à-dire le nombre de fois où chaque pièce est utilisée.

Implémentez cette modification de l'algorithme.

Remarque :

À chaque étape, on va soustraire à s, la valeur de la pièce multipliée par sa multiplicité.

Le nombre d'étapes devient alors égal au nombre de pièces disponibles car au pire des cas on va utiliser au moins une pièce/billet de chaque sorte pour rendre la monnaie.

La **complexité** est de l'**ordre de la taille du jeu** de pièces donné. Elle est linéaire vis à vis de cette taille.

TDD

La méthode d'implémentation ci-dessus, appelée **TDD** (Test-Driven Development, ou Développements Pilotés par les Tests en français), est une méthode de développement de logiciel, qui consiste à **concevoir un logiciel par petits pas**, **en écrivant chaque test avant d'écrire le code source** et **en remaniant le code continuellement**.

4. Supposons maintenant que le libraire ne possède que des pièces de 1, 2, 20 et 50 euros.
   (a) Quelle solution donne l'algorithme Glouton pour rendre une somme de 63 euros ?
   (b) Est-ce la solution optimale ?

Remarque :

L'algorithme glouton **ne donne pas toujours une solution globalement optimale**, mais elle en donne une bonne approximation ; dans le cas du rendu de monnaie, le jeu de pièces disponibles doit être particulier : canonique...

Algorithme Glouton

On considère un problème d'optimisation possédant un très grand nombre de solutions. On dispose d'une fonction Mathématiques évaluant la qualité de chaque solution et on cherche une solution qui soit bonne, voire la meilleure.

Un algorithme est glouton si, à chaque étape où une décision doit être prise, il fait le meilleur choix sur le moment, en espérant arriver à une solution globalement optimale.

2) Un deuxième exemple : le problème du sac à dos

Sac à dos

On a un certain nombres d'objets, dont on connait la valeur et le poids, que l'on doit transporter dans un sac à dos.

On doit choisir les objets mis dans le sac à dos afin de maximiser la valeur totale, sans dépasser le poids maximum que peut supporter le sac.

Nous disposons d'une clé USB qui est déjà bien remplie et sur laquelle il ne reste que 5 Go de libre. Nous souhaitons copier sur cette clé des fichiers vidéos pour l'emporter en voyage.

L'idée est d'avoir un maximum de vidéos à regarder, ainsi la durée totale de ces vidéos doit être la plus grande possible.

Chaque fichier a une taille et chaque vidéo a une durée. La durée n'est pas proportionnelle à la taille car les fichiers sont de format différents, certaines vidéos sont de grande qualité, d'autres sont très compressées.

Le tableau qui suit présente les fichiers disponibles avec les durées données en minutes.

Vidéos	Durées des vidéos	Tailles des fichiers
Vidéo 1	114	4,57 Go
Vidéo 2	32	630 Mo
Vidéo 3	20	1,65 Go
Vidéo 4	4	85 Mo
Vidéo 5	18	2,15 Go
Vidéo 6	80	2,71 Go
Vidéo 7	5	320 Mo

1. Vous devez commencer par représenter les données du problème.

   Pour cela, on va utiliser une liste de listes, de la forme [nom vidéo, durée, taille], en exprimant les tailles en Go.

   Donnez la liste **videos** correspondant au tableau ci-dessus.

   Remarque : Nous allons utiliser un algorithme glouton pour chercher une solution à ce problème.

   On peut prendre **différents critères de choix** pour chaque étape de notre algorithme :

   - **la durée** de la vidéo (on prendra celle qui dure le plus longtemps pour que la durée totale soit maximale) ;
   - le **rapport durée/taille** (on prendra le plus grand rapport).

2. Vous allez créer une fonction **cle_USB()** prenant en paramètres la **liste des fichiers vidéos** disponibles, et la **taille maximale disponible** sur la clé USB en Go.

   En retour, cette fonction donnera une **liste des vidéos** à placer sur la clé, ainsi que la **durée totale** qu'elles représentent.

   (a) Dans un premier temps, nous allons prendre pour critère de choix, celui de la durée des vidéos :

      i. Commencez par **trier la liste** des fichiers dans l'ordre décroissant des durées.

      Remarque :

      Pour cela vous pourrez utiliser la fonction python **sorted()**.

      La syntaxe de cette fonction est :

      sorted(liste, key = lambda liste : liste[0] , reverse = False)

      si l'on souhaite trier la liste selon l'élément d'indice 0 ;

      reverse = False veut dire que ce sera dans l'ordre croissante et **reverse = True** dans l'ordre décroissant.

      *Vérifiez au fur et à mesure que votre code fonctionne en insérant par exemple des **print()**.*

      ii. Ensuite vous devez parcourir la liste triée et ajouter les noms des fichiers que vous ajoutez, un par un dans la variable **reponse**, tant que la taille totale des fichiers ajoutés ne dépasse pas la taille maximale disponible sur la clé. La **durée totale** et la **taille totale** des fichiers ajoutés sont stockés dans deux variables nommées **duree_totale** et **taille_totale**.

   (b) Dans un second temps, nous allons prendre pour critère de choix, celui du rapport durée/taille des vidéos :

      Implémentez une nouvelle fonction **cle_USB2()** en fonction de ce choix.

3. Quel est le critère de choix permettant d'obtenir la meilleure solution au problème ?

   Est-ce la solution optimale globalement ?

Sources :

★ TP du DIU EIL de l'université d'Orleans par Mathieu Liedloff

★ Document d'accompagnement 1NSI, "le problème du sac à dos"

★ Numérique et Sciences Informatiques (30 leçons avec exercices corrigés) aux éditions ellipses par Thibaut Balabonski, Sylvain Conchon, Jean-Christophe Filliâtre et Kim Nguyen