Un gourmet ?... C'est un glouton qui se domine.

Francis Blanche

11 Algorithmes gloutons

Extrait du programme

Thème : Algorithmique

- **Contenus :**
  Algorithmes gloutons

Capacités attendus :
Résoudre un problème grâce à un algorithme glouton.

Commentaires :
Exemples : problèmes du sac à dos ou du rendu de monnaie.
Les algorithmes gloutons constituent une méthode algorithmique parmi d'autres qui seront vues en terminale.

I. Algorithmes gloutons

a) Définition

Un algorithme est dit **glouton** si
- il résout un problème pas à pas,
- en faisant à chaque pas le choix du meilleur gain (ou du moindre coût).

Il tente donc de résoudre des problèmes d'optimisation. Dans la spécification, le rôle sera de trouver le plus ou le moins ... et la postcondition concernera le maximum ou le minimum ...

b) Un premier exemple : conversion d'un nombre en binaire

Nous avons déjà utilisé un algorithme glouton, sans le savoir, lorsque nous avons voulu convertir un nombre décimal en un nombre binaire. Voyons plutôt.

Nous cherchons à convertir le nombre 1789 en binaire. Pour commencer nous allons dresser la liste des puissances de 2 strictement inférieure à 1789 :

$2^{10} = 1024$ | $2^9 = 512$ | $2^8 = 256$ | $2^7 = 128$ | $2^6 = 64$ | $2^5 = 32$ | $2^4 = 16$ | $2^3 = 8$ | $2^2 = 4$ | $2^1 = 2$ | $2^0 = 1$

Nous allons maintenant pouvoir commencer notre algorithme glouton. Nous allons extraire de 1789, la plus grande puissance de 2 possible. On a donc :

$1789 = 1024 + 765$

Nous allons maintenant faire de même avec le reste 765. Quelle est la plus grande puissance de 2 possible que l'on peut extraire de 765 :

$1789 = 1024 + 512 + 253$

On fait à nouveau la même chose avec 253, ... et ainsi de suite...

$1789 = 1024 + 512 + 128 + 125$

$1789 = 1024 + 512 + 128 + 64 + 61$

$1789 = 1024 + 512 + 128 + 64 + 32 + 29$

$1789 = 1024 + 512 + 128 + 64 + 32 + 16 + 13$

$1789 = 1024 + 512 + 128 + 64 + 32 + 16 + 8 + 5$

$1789 = 1024 + 512 + 128 + 64 + 32 + 16 + 8 + 4 + 1$

L'écriture binaire de 1789 est donc $110\ 1111\ 1101_2$.

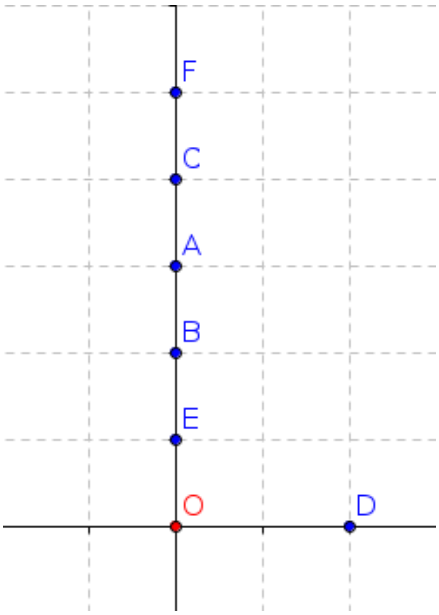
Nous avons bien procédé à l'aide d'un algorithme glouton car nous avons procédé **par étapes** en choisissant à chaque fois **la plus grande** puissance de 2 que l'on peut extraire.

c) Un deuxième exemple : le problème du voyageur de commerce

Mon ami wikipédia [1] me dit la chose suivante :
En informatique, le problème du voyageur de commerce, ou problème du commis voyageur, est un problème d'optimisation qui, étant donné une liste de villes, et des distances entre toutes les paires de villes, détermine un plus court chemin qui visite chaque ville une et une seule fois et qui termine dans la ville de départ. Malgré la simplicité de son énoncé, il s'agit d'un problème d'optimisation pour lequel on ne connaît pas d'algorithme permettant de trouver une solution exacte rapidement dans tous les cas. Plus précisément, on ne connaît pas d'algorithme en temps polynomial, et sa version décisionnelle (pour une distance D, existe-t-il un chemin plus court que D passant par toutes les villes et qui termine dans la ville de départ ?) est un problème ***NP-complet*** [2]*, ce qui est un indice de sa difficulté.*

Voici un problème du voyageur de commerce simple proposé par Gilles Lassus [3], professeur de NSI à Bordeaux.

- Vous partez du point O.
- Vous devez avoir atteint le plus rapidement possible tous les points A, B, C, D, E, F.
- L'ordre de parcours des points n'est pas important.



Comme le problème n'est pas aisé, nous allons le faire à l'aide d'un algorithme glouton.
Nous partons de O et à chaque étape nous irons vers le point le plus proche, et ainsi de suite.
La liste des points parcourus est alors :

$$O - E - B - A - C - F - D$$

Est-ce le chemin le plus court ?
Malheureusement non.
Ce chemin a pour longueur $5 + \sqrt{29} \approx 10,385$
En utilisant une autre méthode, la méthode de la **force brute** [4], on trouve la solution :

$$O - D - E - B - A - C - F$$

qui a pour longueur $6 + \sqrt{5} \approx 8,236$.

Première conclusion :
A l'aide d'un algorithme glouton, on peut trouver une solution, mais cette solution n'est pas nécessairement optimale.

1. https://fr.wikipedia.org/wiki/Problème_du_voyageur_de_commerce
2. En théorie de la complexité, un problème **NP-complet** est un problème de décision vérifiant les propriétés suivantes :
   - il est possible de vérifier une solution efficacement (en temps polynomial) ; la classe des problèmes vérifiant cette propriété est notée NP ;
   - tous les problèmes de la classe NP se ramènent à celui-ci via une réduction polynomiale ; cela signifie que le problème est au moins aussi difficile que tous les autres problèmes de la classe NP.

Un problème NP-difficile est un problème qui remplit la seconde condition, et donc peut être dans une classe de problème plus large et donc plus difficile que la classe NP.
Bien qu'on puisse vérifier rapidement toute solution proposée d'un problème **NP-complet**, on ne sait pas en trouver efficacement.
https://fr.wikipedia.org/wiki/Problème_NP-complet
3. https://github.com/glassus/nsi/blob/master/Premiere/Theme05_Algorithmique/06_Algorithmes_gloutons.ipynb
4. La recherche exhaustive ou recherche par force brute est une méthode algorithmique qui consiste principalement à essayer toutes les solutions possibles. Par exemple pour trouver le maximum d'un certain ensemble de valeurs, on consulte toutes les valeurs. En cryptanalyse on parle d'attaque par force brute, ou par recherche exhaustive pour les attaques utilisant cette méthode.
https://fr.wikipedia.org/wiki/Recherche_exhaustive

II. Le problème du rendu de monnaie

La question est de savoir comment rendre la monnaie en utilisant le moins de pièces possibles.
Si nous décontextualisons, cela revient à dire :
Comment écrire une somme égale à un certain entier N avec un minimum de termes pris, avec répétition, parmi un ensemble fini $P = \{p_1, ..., p_k\}$.
(N est la somme rendu et $P = \{p_1, ..., p_k\}$ est l'ensemble des pièces disponibles.)

On peut donner la spécification de notre problème :

Rôle :	Écrire N comme la somme d'un minimum de $P = [p_1, ..., p_k]$.
Entrées :	Un entier N (à rendre), Un tableau d'entiers dans $P = [p_1, ..., p_k]$ (pièces).
Précondition :	Tous les entiers en entrée sont strictement positifs.
Sortie :	Un tableau d'entiers $M = [m_1, ..., m_k]$ (multiplicité de chaque pièce).
Postconditions :	$m_1p_1 + ... + m_kp_k = N$ $m_1 + ... + m_k$ est minimal pour cette égalité.

Exemple
Si nous avons des pièces de 5€, 2€ et 1€ alors $P = [5, 2, 1]$.
Si nous devons rendre la somme $N = 13$€ alors la réponse optimale est : $M = [2, 1, 1]$, autrement dit :

- 2 pièces de 5€,
- 1 pièce de 2€,
- 1 pièce de 1€.

Nous avons bien $2 \times 5 + 1 \times 2 + 1 \times 1 = 13$€ et nous avons utilisé : $2 + 1 + 1 = 4$ pièces et nous ne pouvons pas faire moins.

Voici l'algorithme glouton que nous allons utiliser :

L'algorithme de rendu de monnaie :

```
1  trier(P) #par ordre décroissant ;
2  M ← [0, ..., 0] #tableau de k cases nulles ;
3  somme ← 0 ;
4  i ← 0 ;
5  tant que somme < N et i<k faire
6  |  si somme + P[i] ≤ N alors
7  |  |  M[i] ← M[i] + 1 ;
8  |  |  somme ← somme + P[i] ;
9  |  sinon
10 |  |  i ← i + 1 ;
11 |  fin si
12 fin tq
13 retourner M
```

La version en langage Python :

```python
def rendu(P,N):
    #nombre de types de pièces
    k = len(P)
    #on range le pièces dans l'ordre dé
    croissant
    P.sort(reverse=True)
    #création du tableau des mutliplicité
    s
    M = [0]*k
    somme = 0
    i = 0
    while somme < N and i<k :
        if somme + P[i] <= N :
            #on rajoute la i-ème pièce
            M[i] = M[i] + 1
            somme = somme + P[i]
        else :
            #on passe à une autre pièce
            i = i + 1
    return M
```

Avec `P = [5, 2, 1]`, pour `rendu(P,13)` nous obtenons bien `[2, 1, 1]`.
On peut tester ce programme en ligne :
`https://repl.it/@lebonprof/rendumonnaie`.

Vous pouvez faire d'autres essais. La méthode gloutonne fonctionne bien car le système monétaire choisi **[5, 2, 1]** est **canonique**. Autrement dit, il n'y a pas de problème, on y arrivera toujours [5].

Pourtant, jusqu'en 1971, le Royaume Uni a utilisé un système monétaire non canonique [6].
Comme ce système est relativement complexe, on va en imaginer un peu plus simple.
Imaginons le système `P = [30, 24, 12, 6, 3, 1]`. Si nous souhaitons rendre la monnaie 49, l'algorithme glouton nous proposera `M=[1, 0, 1, 1, 0, 1]`. En effet :

$$49 = 1 \times 30 + 0 \times 24 + 1 \times 12 + 1 \times 6 + 0 \times 3 + 1 \times 1$$

Pourtant le rendu `M=[0, 2, 0, 0, 0, 1]` est plus intéressant car nous ne rendons que 3 pièces (au lieu de 4 pour la solution gloutonne).

Retour de la première conclusion :
A l'aide d'un algorithme glouton, on peut trouver une solution, mais **cette solution n'est pas nécessairement optimale**.

Imaginons maintenant que nous n'avons plus de pièce de 1. Notre système est le suivant `P = [30, 24, 12, 6, 3]`. Ce système n'est plus canonique et vous pouvez chercher, on ne pourra pas rendre la somme de 49.
L'algorithme glouton nous donne néanmoins une réponse : `M=[1, 0, 1, 1, 0]`. En effet :

$$49 > 1 \times 30 + 0 \times 24 + 1 \times 12 + 1 \times 6 + 0 \times 3 = 48$$

Et oui, dans le système Britannique utilisé jusqu'en 1971, il arrivait parfois que l'on ne pouvait pas rendre la monnaie. Mais comment faisait-il alors ? Personnellement, je n'en sais rien.

Deuxième conclusion :
A l'aide d'un algorithme glouton, **on peut ne pas trouver de solution**, mais la proposition faite par l'algorithme peut être considérée comme convenable.

5. Par contre, il n'existe pas de moyen simple de dire si un système est ou non canonique
6. Voir l'article suivant `https://omnilogie.fr/O/Livre,_shilling,_penny..._le_système_monétaire_anglais_avant_la_décimalisation`

III. Le problème du sac à dos (Knapsack Problem)

Le problème est celui-ci : vous vous appelez Arsène Lupin et vous avez décidé de cambrioler un château. Vous avez un sac pouvant supporter 40 kg et vous souhaitez maximiser la valeur totale des objets que vous mettrez dans votre sac. Évidemment, la somme de leur masse ne doit pas dépasser 40 kg.
Ce problème, malgré sa simplicité est un problème majeur d'optimisation. Actuellement :

- on sait trouver **LA** meilleure solution, mais en explorant toutes les combinaisons une par une. Cette méthode par force brute est inapplicable si beaucoup d'objets sont en jeu.
- on sait facilement trouver une bonne solution, mais pas forcément la meilleure, par exemple en adoptant une stratégie gloutonne.
- on ne sait pas trouver facilement (en temps polynomial) la meilleure solution. Si vous y arrivez, 1 Million de $ sont pour vous [7].

Nous allons donc regarder ici la stratégie gloutonne... Mais laquelle ?

- prendre en priorité les objets les plus légers ?
- prendre en priorité les objets qui ont le plus de valeur ?
- prendre en priorité les objets qui ont le meilleur rapport valeur/masse ?

Nous allons étudier ces 3 cas.

a) Algorithme glouton avec les objets les plus légers en priorité

Considérons un sac de 40kg et les objets suivants :

objet	A	B	C	D	E	F
masse (en kg)	13	12	8	10	14	18
valeur (en €)	700	500	200	300	600	800

On peut implémenter ce tableau à l'aide d'un tableau :
`obj =[["A",13,700], ["B",12,500], ["C",8,200], ["D",10,300],["E", 14,600],["F",18,800]]`

Si on veut utiliser une stratégie gloutonne en fonction des masses, il faut d'abord ranger dans l'ordre croissant des masses de cette liste. Nous reverrons dans un prochain chapitre comment faire et pourquoi cette proposition fonctionne bien :

```
1 def masse(objet):
2     return objet[1]
3
4 obj.sort(key = masse)
```

Nous obtenons le résultat suivant : `[['C', 8, 200], ['D', 10, 300], ['B', 12, 500], ['A', 13, 700], ['E', 14, 600], ['F', 18, 800]]`

7. https://www.claymath.org/millennium-problems/p-vs-np-problem

Il ne nous reste plus qu'à utiliser l'algorithme glouton sur cette liste :

```
def sac(objets,limite):
    '''
    objets est la liste des objets que l'on peut voler
    limite est la masse maximale que peut supporter le sac
    '''
    #masse totale
    masse_tot = 0
    #valeur totale
    val_tot = 0
    #liste des objets volés
    vol = []
    #numéro de l'objet analysé
    i = 0
    while i < len(objets) :
        #est-ce que l'on peut prendre l'objet
        if masse_tot + objets[i][1] <= limite :
            #on rajoute l'objet dans le sac
            masse_tot = masse_tot + objets[i][1]
            val_tot = val_tot + objets[i][2]
            vol.append(objets[i][0])
        else :
            #on passe à un autre objet
            i = i + 1
    return vol, val_tot, masse_tot
```

En posant `sac(obj,40)` nous obtenons la réponse `(['C', 'C', 'C', 'C', 'C'], 1000, 40)`.
Autrement dit nous pouvons mettre 5 objets C pour une masse totale de 40 kg et un vol de 1000€.

b) Algorithme glouton avec les plus grandes valeurs en priorité

Il va falloir ordonner notre liste en fonction des valeurs :

```
def valeur(objet):
    return objet[2]

obj.sort(key = valeur, reverse = True)
```

Nous obtenons le résultat suivant : `[['F', 18, 800], ['A', 13, 700], ['E', 14, 600], ['B', 12, 500], ['D', 10, 300], ['C', 8, 200]]`

En utilisant le même algorithme glouton que précédemment sur cette liste nous obtenons le résultat `(['F', 'F'], 1600, 36)`.
Autrement dit nous pouvons mettre 2 objets F pour une masse totale de 36 kg et un vol de 1600€.

c) Algorithme glouton avec les plus grands rapports valeur/masse en priorité

Il va falloir ordonner notre liste en fonction des rapports valeur/masse :

```
def rapport(objet):
    return objet[2]/objet[1]

obj.sort(key = rapport, reverse = True)
```

Nous obtenons le résultat suivant : `[['A', 13, 700], ['F', 18, 800], ['E', 14, 600], ['B', 12, 500], ['D', 10, 300], ['C', 8, 200]]`

En utilisant le même algorithme glouton que précédemment sur cette liste nous obtenons le résultat `(['A', 'A', 'A'], 2100, 39)` autrement dit dans le sac il y a 3 objets A pour un poids total de 39 kg et un vol de 2100€.
Vous pouvez retrouver le programme en suivant le lien :
`https://repl.it/@lebonprof/sacados`

d) Mais quelle est la meilleure solution ?

Très bonne question. A l'aide d'un algorithme de force brute, nous trouvons la meilleure solution ... à vous de la découvrir.

IV. Exercices

Exercice 1 (**)
Programmer une fonction qui convertit un nombre décimal en binaire à l'aide d'un algorithme glouton.

Exercice 2 (**)
Reprendre le problème du voyageur de commerce proposé en exemple et programmer la recherche d'une solution

1. par un algorithme glouton ;
2. par la méthode de la force brute.

Exercice 3 (**)
Résoudre le problème du sac à dos par la méthode de la force brute.