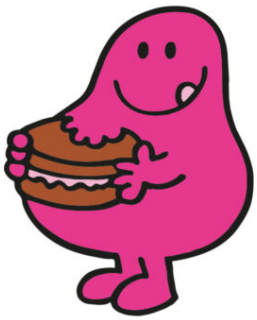
Algorithmes gloutons

Dernière mise à jour le : 20/03/2025

Introduction

On dit *greedy algorithms* en anglais, l'adjectif « greedy » signifiant avare/glouton.



Les **algorithmes gloutons** forment une catégorie d'algorithmes permettant de donner une solution à des *problèmes d'optimisation* qui visent à *maximiser/minimiser* une quantité (*plus court* chemin (GPS), *plus petit* temps d'exécution, *meilleure* organisation d'un emploi du temps, etc.)

Le principe d'un algorithme glouton repose sur la stratégie suivante :

- **résoudre un problème étape par étape**
- **à chaque étape, faire le *choix optimal*** (de moindre coût ou de meilleur gain)

Le choix (optimal) effectué à chaque étape n'est **jamais** remis en cause, ce qui fait que cette stratégie permet d'aboutir rapidement à une solution au problème de départ (car on n'explore pas toutes les possibilités).

C'est en ce sens que l'adjectif *greedy* (glouton/avare) caractérise ces algorithmes : ils terminent rapidement (*glouton*) sans fournir beaucoup d'efforts (*avare*).

Algorithmes gloutons et optimalité

Faire le meilleur choix (local) à chaque étape nous conduit-il nécessairement à la meilleure solution globale ?

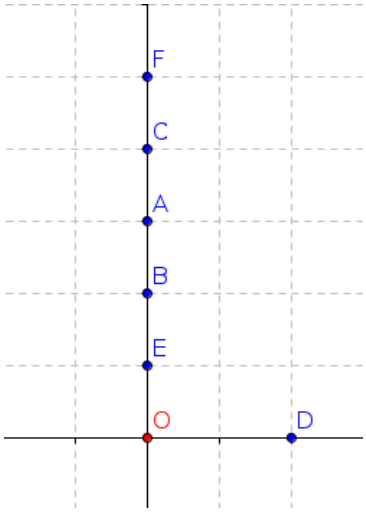
Commençons par quelques exemples pour répondre à cette question.

Exemples simples

✏️ Exemple 1

- On part du point O.

- On doit atteindre le plus rapidement possible tous les points A, B, C, D, E, F.
- L'ordre de parcours des points n'est pas important.



✍ **Question 1** : En appliquant une stratégie gloutonne, à chaque étape on doit aller vers le point le plus proche. Que donne le parcours dans ce cas ?

✍ **Question 2** : Ce chemin est-il optimal ?

✏️ Exemple 2

On cherche à sélectionner cinq nombres de la liste suivante en cherchant à avoir leur somme la plus grande possible (maximiser une grandeur) et en s'interdisant de choisir deux nombres voisins (contrainte).

$$15 - 4 - 20 - 17 - 11 - 8 - 11 - 16 - 7 - 14 - 2 - 7 - 5 - 17 - 19 - 18 - 4 - 5 - 13 - 8$$

Comme on souhaite avoir le plus grand résultat final, la stratégie gloutonne consiste à choisir à chaque étape le plus grand nombre possible dans les choix restants.

✍ **Question 1** : Appliquez cet algorithme glouton sur le tableau.

✍ **Question 2** : Vérifiez que $20, 18, 17, 16, 15$ est une autre solution possible.

✍ **Question 3** : Que dire de la solution gloutonne ?

Moralité

Un algorithme glouton, qui consiste à trouver la **solution optimale *locale*** à chaque étape, ne garantit pas de trouver la **solution optimale *globale***.

Néanmoins, une stratégie gloutonne permet d'aboutir rapidement à une solution, en espérant qu'il s'agisse d'une *bonne* solution globale (à défaut d'être optimale).

Pourquoi se contenter d'une solution non optimale ?
Un peu de patience, nous en discuterons à la fin.

Voyons deux problèmes classiques qui peuvent être résolus par un algorithme glouton.

Le problème du rendu de monnaie

Énoncé

✏️ Le problème

Vous êtes commerçant et devez rendre de la monnaie à vos clients de façon optimale, c'est-à-dire avec le **nombre minimal** de pièces et de billets.

- On suppose que les clients ne vous donnent que des sommes entières en euros (pas de centimes pour simplifier).
- Les valeurs des pièces et billets à votre disposition sont : 1, 2, 5, 10, 20, 50, 100, 200 et 500. On suppose que vous avez autant d'exemplaires de chaque pièce et billet
- Dans la suite, afin de simplifier, nous désignerons par "pièces" à la fois les pièces et les billets.

✍️ **Question 1** : Myriam vous achète un objet qui coûte 53 euros. Elle paye avec un billet de 200 euros. Que lui rendez-vous pour minimiser le nombre de pièces rendues ?

✍️ **Question 3** : Complétez la phrase suivante qui explique la stratégie (gloutonne) à appliquer pour minimiser le nombre de pièces rendues :

À chaque étape, on rend ...

✍️ **Question 3** : En vous appuyant sur la définition, expliquez pourquoi cette stratégie est dite *gloutonne*.

✍️ **Question 4** : Appliquez la stratégie gloutonne pour rendre la somme 8 si on dispose des pièces 1, 4 et 6. Que pensez de cette solution ?

✍️ **Question 5** : Appliquez la stratégie gloutonne pour rendre la somme 8 si on dispose des pièces 2 et 5. Que pensez de cette solution ?

Moralité : L'algorithme glouton du rendu de monnaie :

- ne donne pas nécessairement la solution optimale au problème, cela dépend du système monétaire
- ne donne pas nécessairement une solution exacte au problème, même lorsque celle-ci existe

Implémentation

✏️ À faire

Faire l'exercice 1 pour implémenter l'algorithme glouton de rendu de monnaie.

Voici une implémentation possible :

```
def rendu_glouton(somme_a_rendre, pieces):
    solution = []
    i = 0  # i est l'indice de la pièce à tester
    while somme_a_rendre > 0:  # tant qu'on n'a pas tout rendu
        if pieces[i] <= somme_a_rendre:  # si on peut rendre une pièce
            solution.append(pieces[i])  # on le fait
            somme_a_rendre = somme_a_rendre - pieces[i]  # et on la déduit de la somme
        else:
            i = i + 1  # sinon on passe à la suivante
    return solution
```

On peut vérifier que cela fonctionne bien :

```
>>> rendu_glouton(147, [500, 200, 100, 50, 20, 10, 5, 2, 1])
[100, 20, 20, 5, 2]
```

Comme déjà évoqué, la solution gloutonne n'est pas toujours optimale :

```
>>> rendu_monnaie_glouton(8, [6, 4, 1])
[6, 1, 1]  # on pouvait rendre seulement 2 pièces : [4, 4]
```

Et s'il n'y a pas de pièce unité ?

Notre algorithme ne convient pas s'il n'y a pas de pièce unité. En effet, il se peut alors que l'on ne puisse pas rendre de manière exacte la somme avec notre système monétaire, ce qui impliquerait que l'indice `i` va dépasser la dernière pièce et on aura une erreur de type `IndexError: list index out of range`. Pour éviter ce problème, on peut rajouter une seconde condition à la boucle `while` pour la stopper après avoir testé la dernière pièce :

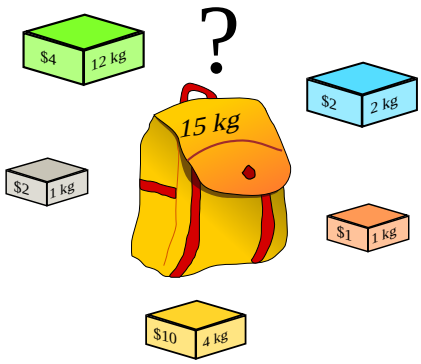
```
while somme_a_rendre > 0 and i < len(pieces):
```

Mais dans ce cas, la solution construite n'est pas toujours exacte :

```
>>> rendu_glouton(8, [5, 2])
[5, 2]  # on rend 7 au lieu de 8 (c'était possible !)
```

Le problème du sac à dos

On dit *Knapsack Problem* en anglais, souvent abrégé **KP**.



✏️ Le problème (version Lupin)

Vous êtes un voleur et disposez d'un sac pouvant supporter une masse maximale de (15 kg sur l'image) dans lequel vous allez mettre des objets qui ont un certain poids et une certaine valeur. Votre objectif est de choisir les objets pour **maximiser la valeur du butin mais sans dépasser la capacité maximale du sac** (on dit qu'il s'agit d'un problème d'*optimisation avec contrainte*, ici la contrainte de poids).

Considérons les objets suivants et un sac de capacité maximale 10 kg. Quels objets faut-il prendre ?

objet	A	B	C	D	E	F
masse (kg)	7	6	4	3	2	1
valeur (€)	9100	7200	4800	2700	2600	200

Il y a **plusieurs stratégies gloutonnes** possibles :

- **Stratégie 1** : prendre toujours l'objet de plus grande valeur possible sans dépasser la capacité maximale (il faut trier préalablement par valeur décroissante).
- **Stratégie 2** : prendre toujours l'objet de plus faible masse possible (il faut trier préalablement par masse croissante).
- **Stratégie 3** : prendre toujours l'objet de plus grand *taux de valeur* (= $\frac{\text{valeur}}{\text{masse}}$) n'excédant pas la capacité restante (il faut trier préalablement par rapport au taux de valeur décroissant).

✍️ **Question 1** : Appliquez les trois stratégies sur l'exemple. Y en a-t-il une meilleure que les autres ?

✍️ **Question 2** : Trouvez une solution encore meilleure.

Moralité : Résoudre le problème du sac à dos de manière gloutonne :

- peut se faire selon plusieurs stratégies, et selon les exemples, l'une ou l'autre se révélera meilleure que les autres.
- ne garantit pas d'obtenir une solution optimale

✏️ À faire

Faire l'exercice 2 pour implémenter une résolution gloutonne du problème du sac à dos.

Pourquoi se contenter d'une solution non optimale ?

Comme nous venons de le voir dans les différents problèmes abordés, la stratégie gloutonne ne donne pas forcément un résultat optimal. On peut alors se demander s'il n'est pas possible de trouver la meilleure solution, à coup sûr, pour résoudre un problème d'optimisation.

Une telle approche existe, il s'agit de la stratégie de **force brute** (ou *énumérative*) qui consiste à **passer en revue toutes les options possibles** et retenir la meilleure.

Pourquoi n'utilise-t-on pas toujours la *force brute* ?

Car ce n'est pas toujours possible !

Le plus simple est de l'expliquer sur un exemple : prenons le problème du sac à dos.

Chaque objet est pris ou pas : il s'agit donc d'une donnée binaire. Avec 3 objets, il y a donc 2^3 combinaisons d'objets possibles, c'est-à-dire 8, ce qui est tout à fait acceptable.

De manière générale, avec n objets, il y aurait 2^n combinaisons à énumérer et tester. On obtient une complexité dite *exponentielle* et c'est là le problème : avec 80 objets, on obtient 2^{80} combinaisons à tester, c'est-à-dire environ 10^{24} combinaisons, soit de l'ordre de grandeur du nombre d'étoiles dans l'Univers observable, ou de gouttes d'eau dans la mer, ou du nombre de grains de sables au Sahara... (référence : https://fr.wikipedia.org/wiki/Ordres_de_grandeur_de_nombres).

Pour le problème du sac à dos, la stratégie *force brute* est donc inapplicable si trop d'objets sont en jeu. Il en est de même pour les autres problèmes d'optimisation dès que le taille des données est trop importante.

Autres problèmes classiques

Il existe de nombreux autres problèmes d'optimisation pouvant être résolu par un algorithme glouton (pas forcément de manière optimale) :

- problème du voyageur (plus court chemin)
- coloration d'un graphe

- coloriage de carte (voir le deuxième exemple de cette vidéo)
- organisation (de planning)
- ...

Certains sont abordés dans les exercices.

Conclusion

- Nous avons vu que les *algorithmes gloutons* fournissent une stratégie pour résoudre des problèmes d'optimisation : *à chaque étape, faire le meilleur choix (local)*.
- Ils donnent *rapidement* une *solution satisfaisante* à un problème mais pas nécessairement la *solution optimale* puisque les choix successifs ne sont jamais remis en cause.
- La stratégie de *force brute* permettrait à coup sûr d'obtenir une solution optimale mais devient inapplicable dès que la taille des données est trop importante. Dans ce cas, une solution gloutonne peut être privilégiée (on s'en contente, faute de mieux).
- Lorsque la solution gloutonne n'est pas optimale, on dit que l'algorithme glouton est une *heuristique* de résolution (il fournit une solution approchée).
- Il existe d'autres méthodes algorithmiques pour résoudre des problèmes d'optimisation : certaines seront abordées en Terminale.

Références :

- Document ressource de l'équipe éducative du DIU EIL de l'Université de Nantes, diffusé sous licence CC BY : Algorithmes gloutons (Bloc 2, Séance 5).
- Prepabac, spécialité NSI 1ère, C. ADOBET, G. CONNAN, G. ROZSAVOLGYI, L. SIGNAC.
- Numérique et Sciences Informatiques, 1re, T. BALABONSKI, S. CONCHON, J.-C. FILLIATRE, K. NGUYEN, éditions ELLIPSES : Site du livre
- Cours de G. LASSUS sur les algorithmes gloutons : lien Github vers son notebook
- Ressources Eduscol : Algorithmes gloutons et Le problème du sac à dos

---

Germain BECKER & Sébastien POINT, Lycée Mounier, ANGERS

Voir en ligne : info-mounier.fr/premiere_nsi/algorithmique/algorithmes-gloutons