Algorithmique	Chap 04	**Algorithmes gloutons**

4.1 Qu'est-ce qu'un algorithme glouton ?

Le principe

Les **algorithmes gloutons** (*greedy algorithms* en anglais) forment une catégorie d'algorithmes permettant de résoudre des **problèmes d'optimisation** qui visent à maximiser ou minimiser une quantité (plus court chemin, meilleure organisation d'emploi du temps, plus grand bénéfice…).

Le principe repose sur la stratégie suivante :

- on résout le problème **étape par étape** ;
- à chaque étape, on fait le **choix optimal local** (de moindre coût ou de meilleur gain) ;
- ce choix n'est **jamais remis en cause**.

C'est en ce sens que l'adjectif *glouton* caractérise ces algorithmes : ils terminent **rapidement** sans explorer toutes les possibilités.

Définition 1 — Algorithme glouton

Un **algorithme glouton** est un algorithme qui résout un problème d'optimisation en construisant une solution pas à pas, en faisant à chaque étape le **meilleur choix local** possible, sans jamais revenir sur les décisions prises.

Une question fondamentale

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

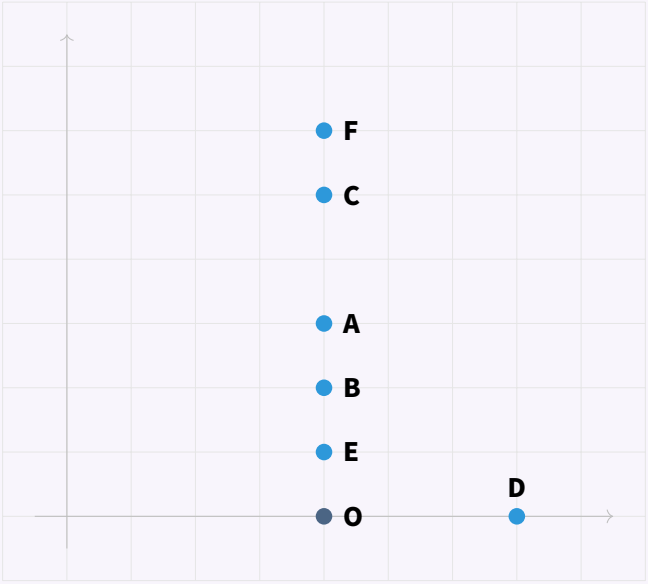
Nous allons explorer cette question sur deux exemples avant d'étudier des problèmes classiques.

4.2 Premiers exemples

Exercice 1 — Chemin le plus court

On part du point **O**. On doit atteindre les points **A, B, C, D, E, F** dans n'importe quel ordre, en cherchant à **minimiser la distance totale** parcourue.

La disposition des points sur le repère est la suivante :



1. En appliquant la **stratégie gloutonne**, à chaque étape, aller vers le point non encore visité le plus proche, déterminer l'ordre de visite des points depuis O. Calculer la distance totale parcourue.

2. Ce chemin est-il **optimal** ? Peut-on trouver un meilleur ordre de visite ?

Exercice 2 — Somme maximale sans voisins

On dispose de la liste de nombres suivante :

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

On cherche à sélectionner **cinq nombres** de cette liste pour avoir leur **somme la plus grande possible**, avec la contrainte suivante : on ne peut pas choisir deux nombres **voisins** dans la liste.

La stratégie gloutonne consiste à choisir à chaque étape le **plus grand nombre disponible** parmi les choix restants.

1. Appliquer cet algorithme glouton. Indiquer les cinq nombres choisis et leur somme.

   ---

2. Vérifier que la sélection $20,\ 18,\ 17,\ 16,\ 15$ est une solution valide. Calculer sa somme. Que peut-on conclure sur 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, il permet d'aboutir **rapidement** à une bonne solution, à défaut d'être la meilleure.

4.3 Le problème du rendu de monnaie

Le problème

Vous êtes marchand dans l'univers de The legend of Zelda et devez rendre de la monnaie à vos clients de façon optimale, c'est-à-dire avec le **nombre minimal de rubis**. Il existe différents types de rubis qui ont pour valeur une certains fois la valeur du rubis rouge.

— Les valeurs de rubis disponibles sont : **1, 2, 5, 10, 20, 50, 100, 200, 500**. On suppose avoir autant d'exemplaires de chaque rubis.

Exercice 3 — Appliquer la stratégie gloutonne

1. Link vous achète un objet qui coûte **53 rubis**. Elle paye avec un rubis d'une valeur de **200 rubis**. Que lui rendez-vous pour minimiser le nombre de rubis rendus ? Détailler chaque étape.

   ---

2. Compléter la phrase qui décrit la stratégie gloutonne appliquée :

   À chaque étape, on rend ---

3. En vous appuyant sur la définition, expliquer pourquoi cette stratégie est dite **gloutonne**.

Exercice 4 — La solution gloutonne est-elle toujours optimale ?

1. Appliquer la stratégie gloutonne pour rendre la somme **8 rubis** avec les rubis disponibles **1, 4 et 6**. Donner la liste des rubis rendues. Comparer avec la solution optimale.

   ---

2. Appliquer la stratégie gloutonne pour rendre la somme **8 rubis** avec les rubis **2 et 5** (sans rubis de 1). Que se passe-t-il ?

Moralité

L'algorithme glouton du rendu de monnaie :

— ne donne **pas nécessairement la solution optimale**, cela dépend du système monétaire ;

— ne donne **pas nécessairement une solution exacte** au problème, même lorsque celle-ci existe.

Remarque 1. Pour le système monétaire de la zone euro (pièces 1, 2, 5, 10, 20, 50, 100, 200, 500), on peut démontrer que l'algorithme glouton donne toujours la solution optimale. Ce n'est cependant **pas le cas en général**.

À coder 1 — Implémentation Python

Effectuer l'exercice 1 de l'activité capytale

4.4 Le problème du sac à dos

Le problème (*Knapsack Problem*)

Vous disposez d'un sac pouvant supporter une **masse maximale** fixée. Vous pouvez y mettre des objets, chacun ayant un **poids** et une **valeur**.

L'objectif est de choisir les objets pour **maximiser la valeur totale du butin** sans dépasser la capacité maximale du sac.

Il s'agit d'un problème d'**optimisation avec contrainte** (ici, la contrainte est le poids maximum). On l'appelle *Knapsack Problem* (KP) en anglais.

Données du problème

Link s'apprête à affronter Ganondorf et doit remplir son sac avec le plus précieux butin possible, sans dépasser la capacité maximale de **10 kg** qu'il peut porter.

Objet	Master Sword	Bouclier	Arc des Fées	Bombes	Fiole de Vie	Flèches
Masse (kg)	7	6	4	3	2	1
Valeur (rubis)	9100	7200	4800	2700	2600	200

4.4.1 Trois stratégies gloutonnes

Exercice 5 — Comparer les stratégies

Il existe plusieurs façons d'appliquer un algorithme glouton à ce problème. Pour chacune, trier les objets selon le critère indiqué, puis remplir le sac en prenant à chaque étape l'objet suivant qui tient encore dans le sac.

Stratégie 1 : par valeur décroissante : on prend toujours l'objet de plus **grande valeur** disponible.

Objets choisis	
Masse totale	
Valeur totale	

bigskip **Stratégie 2 : par masse croissante :** on prend toujours l'objet de plus **faible masse** disponible.

Objets choisis	
Masse totale	
Valeur totale	

Stratégie 3 : par taux de valeur décroissant : le **taux de valeur** d'un objet est défini par $\frac{\text{valeur}}{\text{masse}}$. On prend toujours l'objet de plus grand taux disponible.

Compléter d'abord le tableau des taux :

Objet	A	B	C	D	E	F
Taux (€/kg)						

Objets choisis	
Masse totale	
Valeur totale	

Laquelle des trois stratégies donne le meilleur résultat ? Peut-on faire encore mieux ?

⚠ Moralité

Résoudre le problème du sac à dos de manière gloutonne :

— peut se faire selon **plusieurs stratégies** ; selon les exemples, l'une ou l'autre se révélera meilleure.

— **ne garantit pas** d'obtenir une solution optimale.

</> À coder 2 — Implémentation python

Effectuer l'exercice 2 de l'activité capytale.

4.5 Pourquoi se contenter d'une solution non optimale ?

La stratégie de force brute

Comme nous venons de le voir, la stratégie gloutonne ne donne pas toujours une solution optimale. On peut alors se demander s'il n'est pas possible de trouver la meilleure solution à coup sûr.

Une telle approche existe : la **force brute**. Elle consiste à **passer en revue toutes les options possibles** et à retenir la meilleure.

Mais est-elle toujours applicable ?

Exercice 6 — Complexité de la force brute

Pour le problème du sac à dos, chaque objet peut être pris ou non : c'est une donnée **binaire**.

1. Combien de combinaisons différentes faut-il tester avec 3 objets ? avec 10 ? avec 30 ? avec 80 ?

2. Un ordinateur peut effectuer environ 10^9 opérations par seconde. Estimer le temps nécessaire pour tester toutes les combinaisons avec 30 objets, puis avec 80 objets. Convertir le résultat en années si nécessaire.

3. Conclure : pourquoi la force brute est-elle **inapplicable en pratique** pour de grands problèmes ?

Heuristique

Lorsqu'un algorithme glouton ne donne pas la solution optimale, on dit qu'il s'agit d'une **heuristique** de résolution : il fournit une **solution approchée**, rapidement, là où la force brute serait trop lente.

Pour le problème du sac à dos, la stratégie force brute est donc inapplicable si trop d'objets sont en jeu. Dans ce cas, une solution gloutonne peut être privilégiée, on s'en contente, faute de mieux.

4.6 (Bonus) Planification de tâches

Un cas où le glouton est optimal

Tous les problèmes ne se ressemblent pas : pour certains, il existe une stratégie gloutonne qui **garantit** la solution optimale. La planification d'activités en est un exemple classique.

Le problème

Un professeur doit planifier des activités dans une salle informatique. Chaque activité a un **créneau horaire** défini par une heure de début et une heure de fin. Deux activités sont **incompatibles** si leurs créneaux se chevauchent.

L'objectif est de sélectionner le **plus grand nombre d'activités compatibles**.

Données

Activité	A	B	C	D	E	F	G
Début	8h	9h	10h	11h	8h	12h	13h
Fin	10h	11h	12h	13h	13h	14h	15h

Exercice 7 — Planification gloutonne

La stratégie gloutonne consiste à toujours choisir l'activité qui se **termine le plus tôt** parmi celles compatibles avec les activités déjà choisies.

1. Appliquer cette stratégie à la main. Donner la liste des activités sélectionnées.

À coder 3 — Planification d'activités

1. Représenter les activités en Python sous forme de liste de tuples `(nom, debut, fin)`, les horaires étant des entiers (8 pour 8h, etc.).
2. Écrire une fonction `planifier(activites)` qui :
   - trie les activités par **heure de fin croissante** ;
   - sélectionne de façon gloutonne les activités compatibles (une activité est compatible si son heure de début est supérieure ou égale à l'heure de fin de la dernière activité choisie) ;
   - renvoie la liste des activités retenues.
3. Tester votre fonction sur les données du tableau. Le résultat correspond-il à la réponse de la question à la main ?

Bilan

Ce qu'il faut retenir

- Les **algorithmes gloutons** résolvent des problèmes d'optimisation en faisant à chaque étape le **meilleur choix local**, sans jamais revenir en arrière.
- Ils sont **rapides** mais ne garantissent **pas** la solution optimale en général.
- La **force brute** garantit l'optimalité mais devient **inapplicable** dès que la taille des données est trop importante (complexité exponentielle).
- Quand la solution gloutonne n'est pas optimale, on parle d'**heuristique** : une solution approchée, obtenue rapidement.
- Dans certains cas particuliers (ex. : rendu de monnaie en euros, planification par fin croissante), le glouton donne la solution **optimale**.
- Il existe d'autres méthodes algorithmiques pour résoudre des problèmes d'optimisation (programmation dynamique, algorithmes génétiques…) qui seront abordées en Terminale.