Algorithmes gloutons

Algorithmique

Introduction

De nombreux problèmes en informatique consistent à trouver la **meilleure solution** parmi un grand nombre de possibilités. On parle de problèmes d'**optimisation**. Par exemple : rendre la monnaie avec le moins de pièces possible, trouver le chemin le plus court entre deux villes, remplir un sac à dos de la manière la plus rentable.

Une approche naïve consisterait à examiner **toutes** les solutions possibles, mais leur nombre explose très vite. Pour n objets, il y a 2^n combinaisons à tester : avec 20 objets, cela fait plus d'un million de possibilités ; avec 30, plus d'un milliard.

Comment faire quand tester toutes les solutions est impossible ? Commençons par un exemple concret.

Le rendu de monnaie

Le problème

Un caissier doit rendre 6,78 euros à un client. Il dispose des pièces de 2 euros, 1 euro, 50, 20, 10, 5, 2 et 1 centimes, soit, en centimes : 200, 100, 50, 20, 10, 5, 2 et 1. Comment rendre la monnaie avec le **nombre minimal** de pièces ?

La stratégie gloutonne

L'idée naturelle est de **commencer par la plus grosse pièce** possible, puis de recommencer avec le montant restant. On fait à chaque étape le choix qui réduit le plus le montant, sans jamais revenir en arrière. Suivons cette stratégie pas à pas :

```
Montant à rendre : 678 centimes

Étape 1 -> Pièce de 200 : 678 >= 200 ? Oui -> reste 478
Étape 2 -> Pièce de 200 : 478 >= 200 ? Oui -> reste 278
Étape 3 -> Pièce de 200 : 278 >= 200 ? Oui -> reste 78
Étape 4 -> 78 < 200, 78 < 100, 78 >= 50    -> reste 28
Étape 5 -> 28 >= 20                         -> reste 8
Étape 6 -> 8 >= 5                           -> reste 3
Étape 7 -> 3 >= 2                           -> reste 1
Étape 8 -> 1 >= 1                           -> reste 0

Résultat : [200, 200, 200, 50, 20, 5, 2, 1] -> 8 pièces
```

Cette stratégie qui consiste à faire, à chaque étape, le choix **localement le meilleur** sans jamais revenir en arrière s'appelle un **algorithme glouton** (*greedy algorithm*).

Implémentation en Python

Traduisons cette stratégie en code. L'algorithme suit exactement la démarche manuelle :

```python
def rendu_monnaie(montant, pieces):
    """Renvoie la liste des pièces utilisées pour rendre
    le montant donné, avec un algorithme glouton."""
    pieces_triees = sorted(pieces, reverse=True)
    resultat = []
    reste = montant
    for piece in pieces_triees:
        while reste >= piece:
            resultat.append(piece)
            reste -= piece
    return resultat

systeme = [200, 100, 50, 20, 10, 5, 2, 1]
solution = rendu_monnaie(678, systeme)
print(solution)      # [200, 200, 200, 50, 20, 5, 2, 1]
print(len(solution))  # 8 pièces
```

Le glouton est-il toujours optimal ?

Avec le système monétaire européen, l'algorithme glouton donne **toujours** la solution optimale. Mais ce n'est pas le cas pour tous les systèmes. Considérons un système fictif avec les pièces `[6, 4, 1]` et un montant de 8 centimes :

```
Glouton : 6 + 1 + 1 = 3 pièces   (choix de la plus grosse d'abord)
Optimal : 4 + 4     = 2 pièces   (meilleure solution)
```

L'algorithme glouton choisit d'abord la plus grosse pièce (6), puis est obligé de compléter avec des pièces de 1. En se focalisant sur le meilleur choix *local*, il rate la meilleure solution *globale* (deux pièces de 4). C'est la limite fondamentale de tout algorithme glouton : il ne regarde pas « plus loin » que l'étape en cours.

Principe général d'un algorithme glouton

L'exemple du rendu de monnaie illustre un schéma que l'on retrouve dans tous les algorithmes gloutons :

1. **Trier** les éléments selon un critère pertinent (ici, les pièces par valeur décroissante).
2. **Parcourir** les éléments dans cet ordre.
3. À chaque étape, **sélectionner** l'élément s'il satisfait les contraintes (ici, si la pièce ne dépasse pas le montant restant).
4. **Ne jamais revenir** sur un choix déjà fait.

L'avantage est la **simplicité** et la **rapidité** (souvent en $O(n \log n)$ grâce au tri initial). L'inconvénient, comme on l'a vu avec le système `[6, 4, 1]`, est que le résultat n'est pas toujours optimal.

Quand un algorithme glouton est-il fiable ? Lorsqu'on peut garantir que le meilleur choix local mène toujours au meilleur résultat global. Voyons un deuxième exemple.

Le problème du sac à dos

Énoncé

Un voleur possède un sac pouvant supporter au maximum 15 kg. Il se trouve face aux objets suivants :

Objet	1	2	3	4	5	6	7
Valeur (euros)	500	1 000	1 500	700	800	900	400
Poids (kg)	1	3	5	4	1	2	2

Il souhaite maximiser la valeur totale emportée sans dépasser la capacité du sac. Quel choix faire ?

Trois stratégies gloutonnes

Contrairement au rendu de monnaie où le critère de tri est évident (valeur de la pièce), le sac à dos offre plusieurs critères envisageables. Lequel choisir ?

Stratégie par valeurs décroissantes

On prend d'abord les objets les plus chers, et l'on continue avec les suivants tant qu'ils tiennent :

Objet	Valeur	Poids	Poids cumulé
3	1 500	5	5
2	1 000	3	8
6	900	2	10
5	800	1	11
4	700	4	15
1	500	1	$16 > 15$ (refusé)
7	400	2	$17 > 15$ (refusé)

Total : **4 900 euros** pour 15 kg.

Stratégie par poids croissants

On prend d'abord les objets les plus légers :

Objet	Valeur	Poids	Poids cumulé
1	500	1	1
5	800	1	2
6	900	2	4
7	400	2	6
2	1 000	3	9
4	700	4	13
3	1 500	5	$18 > 15$ (refusé)

Total : **4 300 euros** pour 13 kg.

Stratégie par rapport valeur/poids

On calcule d'abord la « rentabilité » de chaque objet (combien d'euros par kilogramme), puis on prend les plus rentables en premier :

Objet	Valeur	Poids	Ratio	Poids cumulé
5	800	1	800	1
1	500	1	500	2
6	900	2	450	4
2	1 000	3	333	7
3	1 500	5	300	12
7	400	2	200	14
4	700	4	175	$18 > 15$ (refusé)

Total : **5 100 euros** pour 14 kg. La stratégie du ratio valeur/poids donne ici le meilleur résultat, et c'est même l'optimum de cet exemple (on peut le vérifier en essayant toutes les combinaisons) : en tenant compte à la fois de la valeur et du poids, on fait un choix plus éclairé à chaque étape.

Variante fractionnaire (objets divisibles)

Imaginons que les objets soient des matériaux divisibles : poudre d'or, épices, minerai. Le voleur peut alors prendre une **fraction** d'un objet. Par exemple, s'il reste 2 kg de place et qu'un objet pèse 5 kg, il en emporte $\frac{2}{5}$.

Dans ce cas, la stratégie gloutonne par ratio valeur/poids **donne toujours** la solution optimale (on admettra ce résultat). L'intuition est simple : puisqu'on peut couper les objets, il n'y a jamais d'espace gaspillé et prendre le plus rentable en premier est toujours le meilleur choix.

Variante 0/1 (tout ou rien)

En pratique, les objets ne sont souvent pas divisibles : on prend un ordinateur entier ou on le laisse. C'est la variante **0/1** : chaque objet est pris en entier ou pas du tout. L'algorithme glouton ne garantit plus l'optimalité. Voici un contre-exemple avec les objets `A(165, 3)`, `B(100, 2)`, `C(100, 2)` et un sac de 4 kg :

```
Glouton (par ratio) : A seul -> 165 euros (3 kg, plus de place)
Optimal : B + C -> 200 euros (4 kg)
```

L'objet A est le plus « rentable », mais en le prenant, on n'a plus assez de place pour les deux autres. Le glouton est piégé par son propre choix : il ne peut pas *revenir en arrière*.

Pour ce type de problème, des méthodes plus avancées comme la **programmation dynamique** (étudiée en terminale) sont nécessaires.

Implémentation du sac à dos fractionnaire

```python
def rapport(o):
    """Rapport valeur/poids de l'objet o = (nom, valeur, poids)."""
    return o[1] / o[2]

```

```python
def sac_a_dos_frac(poids_max, objets):
    """Résout le problème du sac à dos fractionnaire.
    objets : liste de tuples (nom, valeur, poids)
    Renvoie la liste des objets pris et la valeur totale."""
    tries = sorted(objets, key=rapport, reverse=True)

    poids_actuel = 0
    valeur_totale = 0
    sac = []

    for nom, valeur, poids in tries:
        if poids_actuel + poids <= poids_max:
            sac.append((nom, valeur, poids))
            poids_actuel += poids
            valeur_totale += valeur
        else:
            reste = poids_max - poids_actuel
            if reste > 0:
                fraction = reste / poids
                sac.append((nom, round(valeur * fraction, 2), reste))
                valeur_totale += valeur * fraction
            break

    return sac, round(valeur_totale, 2)
```

Résumé

Propriété	Algorithme glouton
Principe	Choix localement optimal, sans retour en arrière
Avantages	Simple, rapide ($O(n \log n)$)
Inconvénient	Pas toujours optimal
Rendu de monnaie (euros)	Optimal
Sac à dos fractionnaire	Optimal (tri par ratio)
Sac à dos 0/1	Pas toujours optimal

Quand utiliser un algorithme glouton ? Lorsqu'on peut prouver qu'un choix localement optimal mène à une solution globalement optimale, ou lorsqu'une solution approchée suffit.

Exercices

Exercice 1

1. Programmer la fonction `rendu_monnaie(montant, pieces)` qui renvoie la liste des pièces utilisées.
2. Tester avec le système `[200, 100, 50, 20, 10, 5, 2, 1]` pour rendre 678 centimes.
3. Tester avec le système `[6, 4, 1]` pour rendre 8 centimes. Comparer le résultat glouton avec la solution optimale.

Exercice 2 Modifier la fonction `rendu_monnaie` pour qu'elle renvoie un **dictionnaire** associant chaque valeur de pièce utilisée à son nombre d'occurrences. Par exemple, pour 678 centimes :

```
{200: 3, 50: 1, 20: 1, 5: 1, 2: 1, 1: 1}
```

Exercice 3

1. Programmer la fonction `sac_a_dos_frac(poids_max, objets)`.
2. Tester avec les objets de l'énoncé et un sac de 15 kg.
3. Déterminer la complexité temporelle de cette fonction.

Exercice 4 Pour le problème du sac à dos avec les données de l'énoncé, programmer et comparer les trois stratégies gloutonnes (tri par valeur, tri par poids, tri par ratio). Afficher le contenu du sac et la valeur totale pour chacune.

Exercice 5 Considérons les objets suivants et un sac de 4 kg :

Objet	A	B	C
Valeur	165	100	100
Poids	3	2	2

1. Appliquer l'algorithme glouton (tri par ratio). Quel résultat obtient-on ?
2. Trouver « à la main » une meilleure solution.
3. Expliquer pourquoi l'algorithme glouton échoue dans ce cas.

Exercice 6 Un élève dispose d'une journée et souhaite participer au maximum d'activités. Chaque activité a une heure de début et une heure de fin, et deux activités ne peuvent se chevaucher.

Activité	A	B	C	D	E	F
Début	8	9	10	11	13	14
Fin	10	12	11	14	15	16

1. Proposer une stratégie gloutonne pour maximiser le nombre d'activités.
2. Appliquer cette stratégie aux données ci-dessus.
3. Programmer cette stratégie en Python.

Indication : trier les activités par **heure de fin** croissante et sélectionner chaque activité dont l'heure de début est postérieure ou égale à la fin de la dernière activité choisie.