Rendu de monnaie et sac à dos

1. Définition et idée générale

Le **rendu de monnaie** et le **sac à dos** sont deux problèmes classiques d'**optimisation**. On cherche *toujours* la meilleure solution possible relativement à une contrainte donnée.

💭 **Remarque : point de vigilance ▼**

Ces problèmes peuvent *parfois* être résolus par un algorithme glouton. Pas *toujours*, donc !

⚠️ **Conseil important**

Revoir la notion d'algorithme glouton avant de lire la suite. En peu de mots : un algorithme glouton fait, à chaque étape, le meilleur choix *local*. Ce faisant, il donne généralement une *bonne* solution, mais qui **n'**est **pas** forcément *la **meilleure***. Trouver des solutions *locales* optimales peut donc aboutir à une solution *globale **non optimale***.

2. Problème du rendu de monnaie

Ce problème se pose lorsqu'un client paye en espèces : il donne une somme trop importante, et il faut lui rembourser le trop perçu. On doit donc atteindre une somme précise, avec une contrainte : utiliser un **minimum** de pièces.

💭 **Remarque : des pièces ou des billets ? ▼**

On assimile pièces et billets. On ne fait pas de distinction sur la nature, seulement sur la valeur de chaque « élément ».

✅ **Solution**

Le rendu de monnaie est souvent l'exemple de problème où un algorithme glouton fonctionne bien :

- on choisit la plus grande « pièce » inférieure à la somme à atteindre et on rend autant de fois que possible cette « pièce » ;
- on choisit la deuxième plus grande « pièce » inférieure à la somme *restant à atteindre* et on rend autant de fois que possible cette « pièce » ;
- etc. jusqu'à atteindre la somme voulue.

L'algorithme glouton peut donner un résultat ***optimal** garanti* **lorsque le « système de pièces » est bien choisi :**

- « optimal garanti » signifie ici que le glouton trouve **LA** meilleure solution ;
- la plupart des « systèmes de pièces » du monde conviennent : ils sont dits **canoniques**. Par exemple :
  - centimes : 1, 2, 5, 10, 20, 50 ;
  - pièces et billets : 1, 2, 5, 10, 20, 50, 100, 200, 500.

💭 **Remarque : un contre-exemple historique ▼**

Voir cette page de Wikipedia (https://fr.wikipedia.org/wiki/Probl%C3%A8me_du_rendu_de_monnaie#Exemples) qui évoque le système anglais avant sa réforme en 1971.

Voici le code Python du rendu de monnaie glouton.

```python
def rendu_glouton(montant : int, pieces : list) -> list:
    """
    Prend en paramètre un montant et une liste de pièces (des entiers).
    On considère qu'on dispose d'une quantité illimitée de pièces, ici.
    Renvoie la liste des pièces à rendre pour résoudre le problème.
    """
    resultat = []
    for piece in sorted(pieces, reverse=True):  # Prendre la plus grande pièce en 1er
        while montant >= piece:                 # Tant que c'est possible...
            resultat.append(piece)              # ...ajouter cette plus grande pièce
            montant -= piece                    # Actualiser le montant restant à rendre
    return resultat
```

3. Problème du sac à dos

On note parfois KP ce problème (de l'anglais *Knapsack Problem*). C'est un problème plus délicat à traiter que le rendu de monnaie...

On veut remplir un sac (à dos... ou pas) :

- *sans dépasser* une capacité donnée (en volume, ou en masse, ou ...) ;
- tout en *maximisant* la « valeur » transportée.

⚠️ **Attention aux variantes du sac à dos !**

Il existe au moins deux variantes de ce problème :

- le sac à dos « binaire » (ou « 0/1 ») : on prend un objet en totalité ou on ne le prend pas ;
- le sac à dos fractionnaire : on peut prendre une *portion* d'un objet.

Dans ce problème, on voit très bien la différence entre une stratégie gloutonne et une stratégie fondée sur la

programmation dynamique :

- un choix local se fondant sur le meilleur rapport « valeur/poids » peut échouer ;
- une solution par *programmation dynamique* explore les combinaisons pertinentes.

💭 Remarque : un petit exemple où l'algorithme glouton échoue ? ▼

Considérons un sac (à dos) d'une capacité de 10 kg. Voici la liste des objets emportables (non fractionnables).

Objet	Masse	Valeur	Rapport valeur/masse
A	1 kg	2 €	2
B	5 kg	6 €	1,2
C	5 kg	6 €	1,2

L'approche gloutonne conduit à choisir A (meilleur rapport valeur/masse), puis B, puis C → 1 + 5 + 5 = 11 kg, supérieur à la capacité. Donc la solution gloutonne se limite à prendre A et B, d'une valeur de 8 € pour une masse de 6 kg. Pourtant, prendre B et C donne une masse de 10 kg et une valeur de 12 €, ce qui est meilleur : le glouton a échoué.

💭 Remarque : logique et apport de la programmation dynamique ▼

La programmation dynamique résout le problème en construisant un tableau de résultats intermédiaires.
L'idée : pour *chaque objet* et *chaque capacité possible*, on répond à la question : *« quelle est la valeur maximale atteignable avec les objets vus jusqu'ici, pour **au plus** cette capacité ? »* On remplit ce tableau ligne par ligne (un objet par ligne), colonne par colonne (une capacité différente par colonne). À chaque case, on choisit la meilleure option entre :

- ne pas prendre l'objet courant (on recopie la valeur de la ligne précédente) ;
- prendre l'objet courant (on ajoute sa valeur à la meilleure solution pour la capacité réduite).

À la fin, la case en bas à droite du tableau contient la valeur optimale. On n'a jamais fait de choix irréversible : on a exploré toutes les combinaisons pertinentes sans les énumérer toutes naïvement.

Avec l'exemple précédent, le tableau construit est :

	0	1	2	3	4	5	6	7	8	9	10
ø	0	0	0	0	0	0	0	0	0	0	0
+A	0	2	2	2	2	2	2	2	2	2	2
+B	0	2	2	2	2	6	8	8	8	8	8
+C	0	2	2	2	2	6	8	8	8	8	12

On obtient bien la meilleure solution ici.

✅ **Solution par programmation dynamique** ▼

La programmation dynamique construit un tableau où chaque case permet de répondre à la question *« quelle est la valeur maximale avec ces objets et cette capacité ? »*. On évite ainsi les mauvais choix irréversibles faits par un glouton, au prix d'un coût en mémoire et en temps plus élevé.

4. Approches top-down et bottom-up

Ce paragraphe est hors programme !

La programmation dynamique peut s'implémenter de deux façons :

- ***« Top-down »*** (approche descendante) : on part du problème global et on le décompose récursivement en sous-problèmes, en mémorisant les résultats au fur et à mesure. (mémoïsation). C'est l'approche la plus naturelle à écrire, on la voit en action dans le calcul des termes de la suite de Fibonacci, dans la fiche sur la programmation dynamique.
- ***« Bottom-up »*** (approche ascendante) : on part des sous-problèmes les plus simples et on remplit un tableau de résultats intermédiaires jusqu'à atteindre la solution globale. C'est l'approche utilisée implicitement dans le tableau du sac à dos ci-dessus.

Les deux approches donnent le même résultat. Le *« bottom-up »* est souvent plus efficace en mémoire (pas de pile d'appels récursifs), mais le *« top-down »* est plus proche de la définition mathématique du problème.

5. Liens avec d'autres notions

- Le **glouton** est souvent la première méthode essayée.
- La **programmation dynamique** est essentielle pour le sac à dos « binaire ».
- La **complexité** permet de comprendre pourquoi certaines approches sont préférables.
- La **récursivité** peut servir à formuler la version naïve.