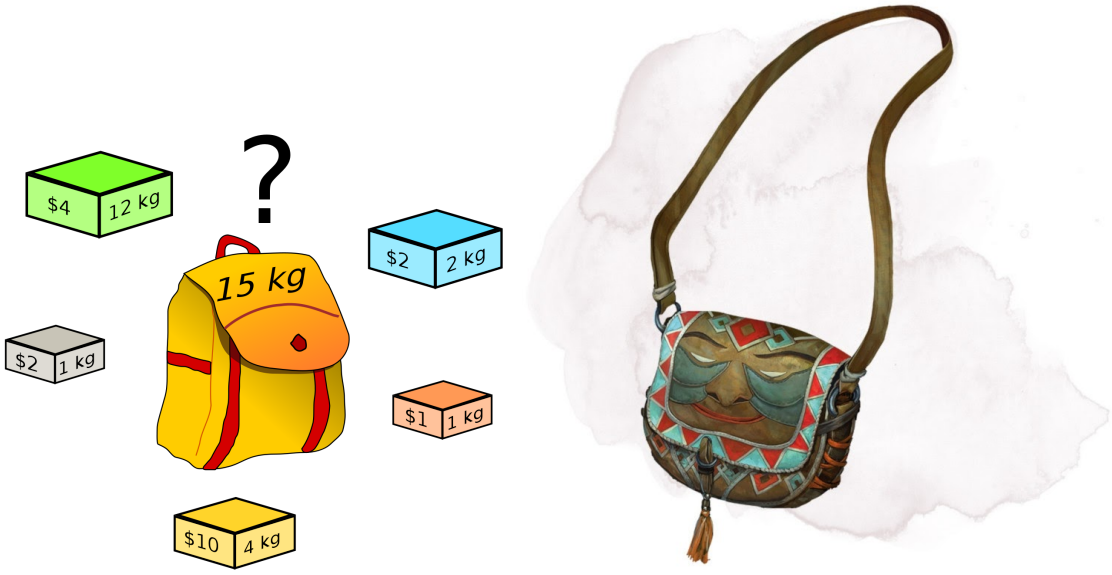
Projet : Le sac à dos

Contexte

Comme le dit la page wikipedia le problème du sac à dos (knapsack problem) :

En algorithmique, le **problème du sac à dos** est un problème d'optimisation combinatoire. Il modélise une situation analogue au remplissage d'un sac à dos, ne pouvant supporter plus d'un certain poids, avec tout ou partie d'un ensemble donné d'objets ayant chacun un poids et une valeur. Les objets mis dans le sac à dos doivent maximiser la valeur totale, sans dépasser le poids maximum.



Le problème du sac à dos est l'un des **21 problèmes NP-complets de Richard Karp**, exposés dans son article de 1972. La formulation du problème est fort simple, mais sa résolution est plus complexe. Les algorithmes existants peuvent résoudre des instances pratiques de taille importante. Cependant, la structure singulière du problème, et le fait qu'il soit présent en tant que sous-problème d'autres problèmes plus généraux, en font un sujet de choix pour la recherche.

But du jeu

Vous devez implémenter trois algorithmes différents ainsi que deux fonctions utilitaires.
Ces deux algorithmes ont les mêmes paramètres d'entrées, et renvoient la même chose (cf code).
Il est conseillé de commencer par le premier algorithme, ensemble, durant le TP.
Les paramètres d'entrées seront toujours :

- Un sac à dos vide :
    - Il a une capacité (un poids maximum)
    - Un contenu (un tableau vide)
- Un dictionnaire d'objet ayant chacun :
    - Une clé, qui est le nom, unique, de l'objet
    - Un tuple avec :
        - Une valeur (entier positif)
        - Un poid (entier positif)

Fonctions utilitaires

Il y a deux méthodes utilitaires à implémenter dans votre classe **Knapsack**.
(qui devra hériter de la mienne)

- Une méthode pour calculer la valeur et le poids d'un sac à dos.
- Une méthode imprimant tous les objets dans le sac

Premier algorithme : la naïveté

Le premier algorithme de résolution doit rester simple, il existe plusieurs algorithmes dit "glouton" ou "greedy" en anglais pour résoudre un problème.

Un exemple d'algorithme est de trier les objets selon leurs valeurs au kg, et d'ajouter progressivement au sac les éléments ayant la plus grande valeur/kg, jusqu'au moment où le sac est plein.
Ce premier algorithme ne sera pas optimal, mais doit remplir le sac à dos d'un certains nombres d'éléments, et vous permettra de prendre en main le problème.

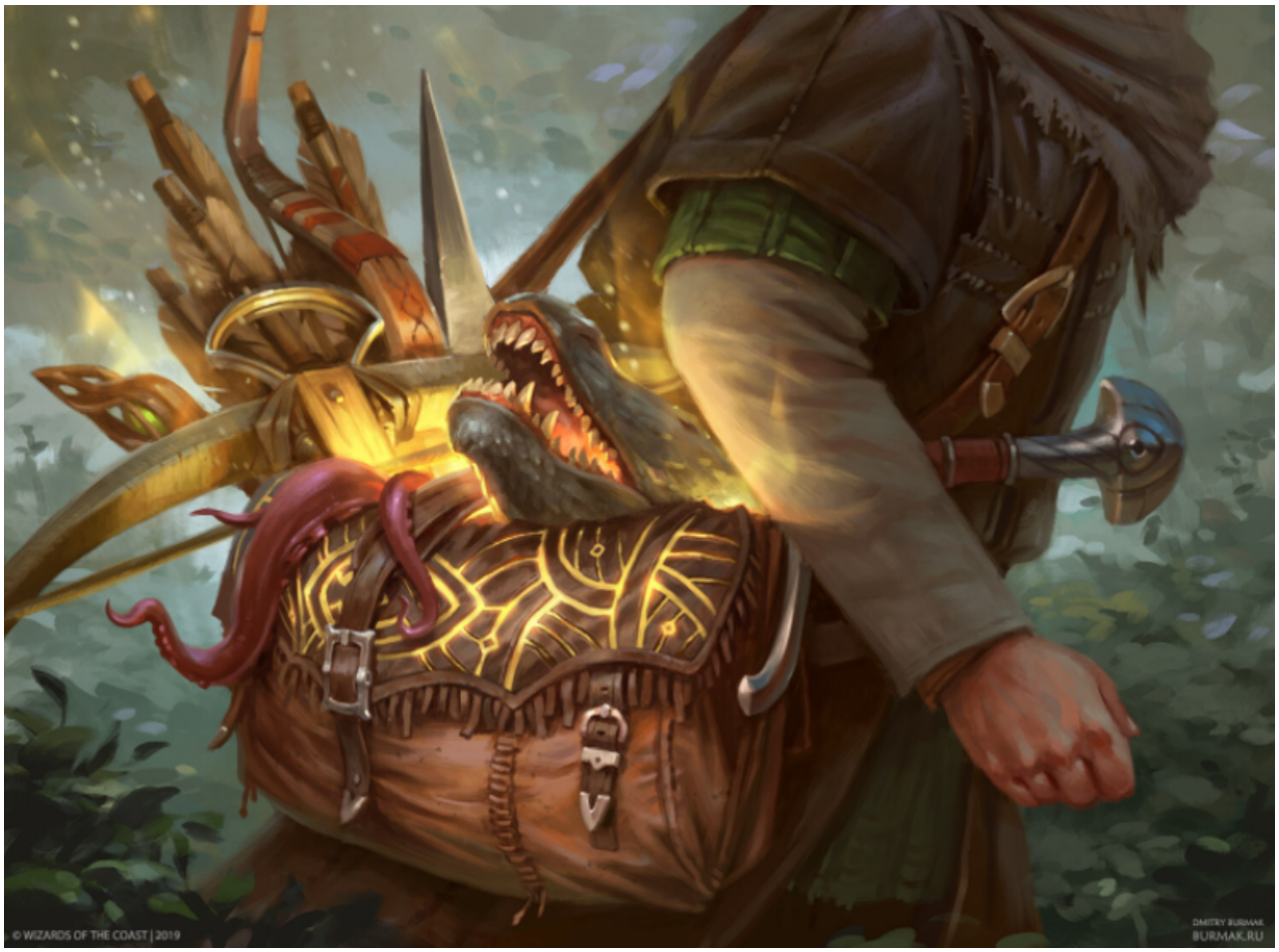
Pour tester cet algorithme, les tests unitaires sont fournis.

Deuxième algorithme : le meilleur !

Ce deuxième algorithme nécessitera des recherches de votre part (des exemples d'algorithmes sont proposés sur wikipedia par exemple).
Il s'agira d'un algorithme donnant une solution meilleure que le glouton, en optimisant l'utilisation du sac pour en maximiser la valeur.
Lors de la soutenance, je vous demanderais de m'expliquer, voir de me représenter, cette solution. N'hésitez pas à fournir vos sources.

Troisième algorithme : l'optimal.

Ce troisième algorithme nécessitera aussi des recherches de votre part, il sera exhaustif et donnera la solution la meilleure, en cherchant parmi toutes les solutions possibles.
Il est attendu que cet algorithme soit long, trop long, et sera exécuté sur des sacs plus petits.



Règles du jeu

En utilisant le code python fourni, comme exemple, vous devez implémenter les différentes fonctions dont voici les signatures :

```
def solve_knapsack_greedy(knapsack, objects_dict) -> Knapsack:
def solve_knapsack_best(knapsack, objects_dict) -> Knapsack:
def solve_knapsack_optimal(knapsack, objects_dict) -> Knapsack:
```

Dans la **classe Knapsack**, il est attendu. :

```
class Knapsack:
    def __init__(self, capacity):
        self.capacity = capacity
        self.content = []
    def get_value_and_weight(self, objects_dict) -> (int, int):
    def print_content(self, objects_dict) -> None:
```

Vous pouvez ajouter des attributs à la classe, mais les signatures ne doivent pas changer.
Les tests unitaires fournis permettent de tester les méthodes utilitaires ainsi que l'algorithme Greedy.

Collaboration

- Vous pouvez être un ou deux, mais pas trois
- Vous pouvez vous aider les uns les autres, en particulier dans la recherche du meilleur algorithme. Sans copier évidemment.
- Vous pouvez vous échanger des fonctions/classes utilitaires non obligatoires. Mais si cela est fait, vous devrez le mentionner.
- Vous pouvez me demander de l'aide, de préférence par discord.

Respect du format

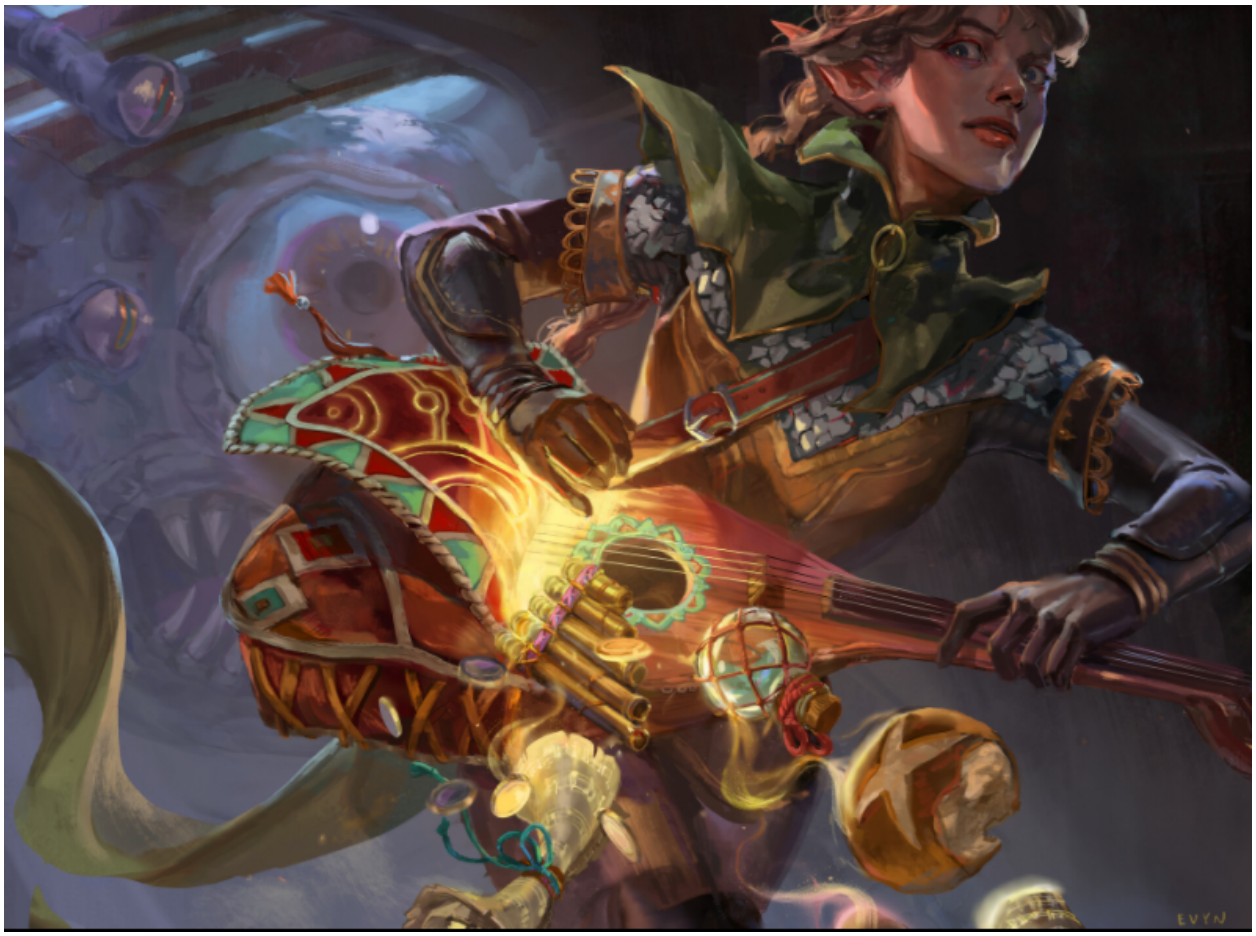
Tout manquement à ces règles sera sanctionné !

- Vous devez respecter le format d'entrée et de sortie.
- Vous devez travailler dans d'autres fichiers.
- Vous pouvez ajouter des tests si vous le souhaitez, mais lors de la correction seul mon fichier de test sera exécuté.
- Vous devez coder en Python 3.8 ou supérieur
- Vous pouvez utiliser toutes les librairies intégrées au Python, à l'exception de pytest qui est autorisé, voir très fortement conseillé.

Liste des librairies intégrées à Python 3.9: https://docs.python.org/3.9/library/index.html
Si vous avez besoin d'une autre librairie, contactez moi et on en parlera.

Livraison

Vous devez me partager, au plus tôt, votre code dans un repository privé sur github (https://github.com/sophisur)
Afin que j'ai accès à tout l'historique de votre code. La version utilisée pour la correction sera celle de la branche principale le 18 décembre matin. Une soutenance, qui servira de pré-correction, le 15, après le DST, selon les conditions sanitaires. Vous permettant d'éviter des petits accidents.



Score final

Votre score final sera basé sur plusieurs éléments. Le barème est à titre indicatif, il vous sert de guide.

Est ce que tous les tests passent ? (14)

Tout simplement, est ce que tout fonctionne ? Plus ou moins bien ?

- Un algorithme : 5 points
- Deux algorithmes : 10 points
- Les trois fonctionnent : 14 points

Qualité du code : Le code respecte t'il les normes Python ? (4)

Ici je vais noter si vous respectez la norme PEP 8. Beaucoup d'IDE permettent de vérifier automatiquement si votre code respecte cette norme.
https://www.python.org/dev/peps/pep-0008/
En toute transparence, voici la ligne de code que j'utilise :

```
flake8 --exclude __pycache__,venv --ignore E501
```

Mais aussi la lisibilité du code, le nommage et la longueur de vos fonctions/méthodes…
Je prendrais aussi en compte votre bonne utilisation de Git, ne poussez pas autre chose que vos fichiers de code ou de tests. (pas de pyc !)

La perfection ! La compétition ! (2)

Pour finir, je vais comparer la performance de votre algorithme, en temps, et sur la qualité du résultat.

- Le temps d'exécution des algorithmes
- Jusqu'à quelle taille de sac à dos et d'objets je peux aller, sur votre meilleurs algorithme en gardant un temps raisonnable ?
- Si vous avez des fonctions utilitaires supplémentaires intéressantes.

Conseils

Une petite liste non exhaustive :

- Je vous conseille de coder en pair, au moins sur la mise en place du premier algorithme.
- N'attendez pas pour exécuter ! Dès que vous récupérez le code, avant même de commencer à coder. Allez y pas à pas, en exécutant au fur et à mesure. Le TDD est une bonne pratique https://fr.wikipedia.org/wiki/Test_driven_development
- Faites des recherches
- Posez moi des questions
- Mettez en place un environnement qui vous convienne, pour ma part, j'ai codé le projet sur Pycharm, en utilisant la librairie pytest.
- Faites des dessins.
- Si vous vous le demandez, les illustrations viennent de D&D5.