TP Python – Structures de données et algorithmique 2

Objectifs :

- Comprendre les principes et l'intérêt de la **programmation dynamique**
- Découvrir les **heuristiques** avec un exemple simple d'**algorithme glouton**
- Comparer les algorithmes **exacts** et d'**approximation** pour la résolution de **problèmes d'optimisation combinatoire**

Le **problème du sac à dos** (*Knapsack*) consiste à choisir un sous-ensemble parmi n objets, de telle manière que la somme des **poids** des objets choisis ne dépasse pas la **capacité** c du sac à dos et que la somme de leurs **valeurs** soit **maximale**. Si on note $w(x)$ le poids de l'objet x et $v(x)$ sa valeur (ces deux fonctions renvoient des valeurs **entières strictement positives**), il faut trouver un sous-ensemble $X' \subseteq X$ de l'ensemble des objets $X = \{x_1, \ldots, x_n\}$ tel que $\sum_{x \in X'} w(x) \leq c$ et de valeur totale $\sum_{x \in X'} v(x)$ maximale.

On peut résoudre ce problème NP-difficile grâce à la relation de récurrence suivante sur $kp(i,j)$ définie comme la valeur totale maximale réalisable avec les i premiers objets $\{x_1, \ldots, x_i\}$ seulement, pour un sac à dos de capacité j :

$$\forall i \in [1,n],\ \forall j \in [1,c] \quad kp(i,j) = \begin{cases} kp(i-1,j) \text{ si } w(x_i) > j \\ \max(kp(i-1,j), kp(i-1, j-w(x_i)) + v(x_i)) \text{ sinon} \end{cases}$$

avec $\forall j, kp(0,j) = 0$ (l'ensemble vide a une valeur nulle) et $\forall i, kp(i,0) = 0$ (un sac de poids nul a une valeur nulle). La solution sera ainsi donnée par le calcul de $kp(n,c)$.

1. Définir une classe `Item` doté de trois attributs (entiers) (`id`, `w` et `v`).
2. Définir une fonction **récursive directe** `knap(items, c)` qui calcule $kp(n,c)$ en prenant en paramètres la liste des items ainsi que la capacité maximale c du sac à dos. **Tester** votre fonction avec l'instance suivante :

   ```
   >>> ws = [2,5,7,9,12]
   >>> vs = [1,2,3,10,7]
   >>> c = 15
   >>> items = [Item(id, ws[id], vs[id]) for id in range(len(ws))]
   >>> knap(items, c)
   12
   ```

   Indication : Utiliser une fonction récursive locale qui prend i et j en paramètres.
   Remarque : Les listes de n éléments en Python (comme les tableaux dans la plupart des langages de programmation) sont indexées de 0 à $n-1$. Il y aura donc un décalage entre la formule de l'énoncé et l'indexation des items.

3. Estimer (en commentaires dans le code) un minorant de la **complexité temporelle** en pire cas de cet algorithme[1] pour montrer qu'elle est exponentielle, puis calculer sa **complexité spatiale**.
4. Améliorer l'efficacité de la fonction précédente en définissant une fonction `knap_dyn(items, c)` qui utilise la technique de **programmation dynamique**.
   Indication : Utiliser une matrice (une liste de listes en compréhension – inutile d'utiliser NumPy quand il n'y a pas d'algèbre linéaire) de taille $(n+1) \times (c+1)$ initialisée à 0.
5. Calculer les **complexités temporelle et spatiale** en pire cas de `knap_dyn`.
6. Quand le problème est trop grand ou que le temps de réponse est trop court pour utiliser un algorithme de programmation dynamique, on peut utiliser un **algorithme glouton** utilisant l'**heuristique** suivante :
   — on trie au préalable les objets par « efficacité », i.e. le ratio valeur sur poids, décroissante ;
   — on initialise le poids restant à c et la valeur totale à 0 ;
   — on considère les objets un à un dans cet ordre tant que le poids restant est strictement supérieur à 0 :
      — si le poids de l'objet est inférieur au poids restant, on ajoute l'objet et on met à jour le poids restant et la valeur totale ;
      — on passe au suivant.

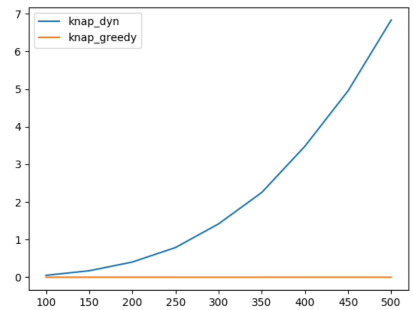
   Écrire une fonction `knap_greedy(items, c)` qui implémente cette heuristique, la tester sur l'exemple précédent et indiquer sa **complexité temporelle** en pire cas.
   Remarque : Cette heuristique, bien que pouvant paraître assez efficace sur de nombreuses instances, peut être arbitrairement mauvaise : e.g. avec un objet de poids 1 et de valeur 2, de meilleure efficacité qu'un second objet de poids c et de valeur c (valeur totale 2 avec l'heuristique alors que la solution optimale est c).
7. **Tester l'efficacité en temps de calcul et en optimalité** de la solution sur des **instances aléatoires de taille croissante** et vérifier s'ils confirment les calculs de complexité.
8. On souhaite également obtenir non seulement la valeur totale optimale mais aussi le **détail des objets faisant partie de la solution**. Écrire une nouvelle version des différents algorithmes qui **renvoie également les indices des objets utilisés** :
   — C'est très simple à faire pour l'algorithme glouton en ajoutant chaque objet retenu à une liste d'indices initialement vide.
   — C'est plus délicat pour la version récursive : il faut renvoyer la liste vide lors du cas d'arrêt et ajouter l'indice au résultat lorsque le maximum de la relation de récurrence sélectionne le second terme.
   — Pour l'algorithme de programmation dynamique, on peut retrouver après coup la décision prise à chaque étape en partant de la fin : à partir de la case `kp[n][c]`, si `kp[i][j] != kp[i-1][j]`, alors l'objet i (ou plutôt d'indice `i-1` à cause du décalage) a été utilisé dans la solution ; on l'ajoute alors à la liste d'indices et on

1. On supposera que les poids sont tous égaux à 1.

décrémente j du poids correspondant ; on décrémente systématiquement i pour passer à l'objet suivant et on réitère jusqu'à ce que i ou j soit nul.

Avec, par exemple, 80 objets aléatoires (de poids et valeur entre 1 et 80) et une capacité de 70 :

```
$ ./tp2.py 80 70
knap: 439 in 0.122422
knap2: 439 in 0.133179 wt: 70 vt: 439
[5, 12, 14, 26, 56, 62, 63, 78]
knap_dyn: 439 in 0.00156597
knap_dyn2: 439 in 0.00154746 wt: 70 vt: 439
[5, 12, 14, 26, 56, 62, 63, 78]
knap_greedy2: 424 in 5.3133e-05 wt: 64 vt: 4
[5, 12, 14, 25, 26, 56, 62, 78]
```



Note : un fichier source Python peut être rendu exécutable en copiant :

```
#!/usr/bin/env python3
```

en première ligne ("shebang") et en exécutant la commande suivante dans un terminal :

```
chmod a+x tp2.py
```