ITC – TD n°6
Le problème du sac à dos

On s'intéresse ici à un problème classique d'algorithmique : le problème du sac à dos. La question est la suivante : disposant d'un sac à dos de contenance limitée, et d'un ensemble d'objets ayant un certain poids et un certain prix, comment remplir le sac à dos sans dépasser sa contenance et en maximisant la valeur emportée ? Il s'agit d'un problème dit NP complet, c'est-à-dire qu'il n'existe pas d'algorithme de complexité polynomiale trouvant la solution optimale au problème.

L'ensemble des objets sera stocké dans deux dictionnaires `poids` et `prix` ayant les mêmes clés (les noms des objets, de type `str`), et dont les valeurs seront des entiers représentant respectivement le poids et le prix de chaque objet. Par exemple, `poids['montre']` contient le poids d'une montre.

Ici, nous allons nous intéresser à trois approches pour ce problème :

- l'algorithme glouton, qui donne très rapidement une solution raisonablement correcte ;
- l'algorithme d'exploration systématique, très coûteux en temps de calcul mais qui garantit la solution optimale ;
- l'algorithme en programmation dynamique, qui exploite les spécificités du problème pour accélérer l'exploration en ne testant pas certaines options inutiles

Sépcification

Toutes ces fonctions `exhaustif`, `glouton` et `dynamique` prendront en paramètre les deux dictionnaires `poids` et `prix`, ainsi qu'un entier c représentant la contenance du sac, et renverront un dictionnaire contenant les objets sélectionnés : les clés seront les noms des objets et les valeurs les prix associés.

On fournit un préambule avec une fonction pour générer des collections d'objets (20 objets maximum), ainsi qu'une collection réduite (de taille 3) dont on donne les résultats attendus pour chaque test :

```
poids_test = {'montre': 1, 'livre': 2, 'boite': 3}
prix_test = {'montre': 3, 'livre': 5, 'boite': 6}
```

I. Une fonction utilitaire

1 – Coder une fonction `calculeEnsemble(objs: dict) -> int` qui prend en paramètre une caractéristique de la colection d'objets (les poids ou les prix) calcule le total correspondant.

II. Algorithme glouton

Ici, l'algorithme glouton prend à chaque instant l'objet de meilleur rapport valeur/poids qui rentre dans le sac, puis cherche à remplir la place restante. Cet algorithme est donc très rapide (la complexité est limitée par le tri des objets, donc en $\mathcal{O}(N\ln(N))$), mais dans ce problème il n'est pas du tout assuré que l'on obtienne la meilleure solution à la fin. On espère simplement que la solution obtenue soit assez bonne.

2 – Écrire la fonction `glouton(poids:dict, prix:dict, c: int) -> dict` correspondante qui renvoie la solution au problème en triant les objets par rapport prix/poids.

Aide : il y a plusieurs façons de procéder, mais si la vôtre nécessite une liste des noms d'objets triés selon le rapport prix/poids, on pourra utiliser les lignes suivantes pour générer une telle liste :

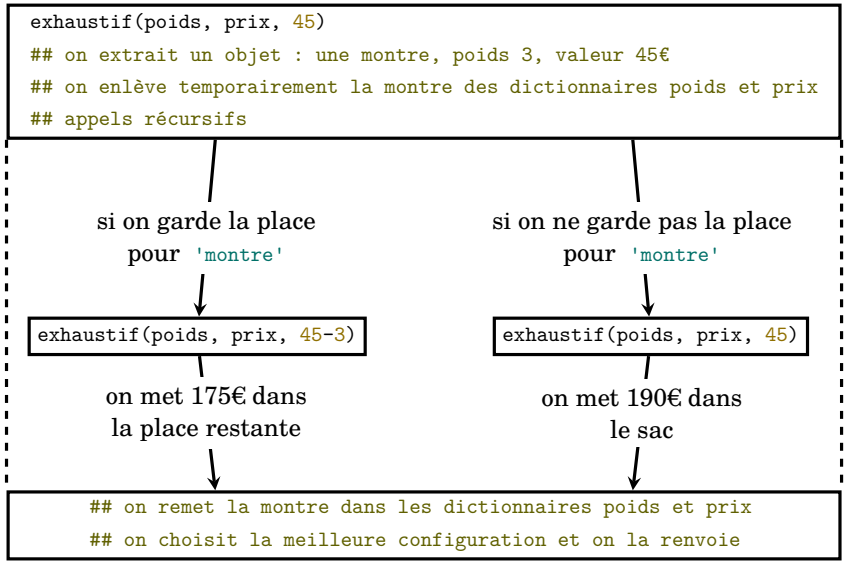
```
objets = [nom for nom in poids.keys()]
objets.sort(key=lambda e: prix[e]/poids[e], reverse=True)
```

III. Exploration exhaustive

On va explorer toutes les configurations possibles à l'aide d'un algorithme récursif testant toutes les possibilités. L'idée (inspirée de la génération exhaustive des permutations d'une liste) est la suivante :

- si la liste d'objets proposés est vide, on renvoie la liste vide avec une valeur de zéro ;
- sinon, on compare la valeur du meilleur remplissage sans le premier objet (mais avec tous les autres objets) avec la valeur du meilleur remplissage avec le premier objet imposé dans le sac (et potentiellement les autres dans la place restante), et on renvoie la meilleure des deux configurations.

Il est à noter que tester une configuration contenant un objet imposé de poids p dans un sac de contenance c revient à tester une configuration dans un sac vide de contenance $c-p$. Cela donne, par exemple, le cas suivant :



renverra un sac à dos où on a choisi de garder l'objet `'montre'`.

Pour obtenir une clé valide d'un dictionnaire `poids`, on pourra utiliser la syntaxe suivante (non exigible car hors programme) :

```
nom_1 = next(iter(poids))
```

3 – Écrire la fonction `exhaustif(poids:dict, prix:dict, c: int) -> dict` correspondante qui renvoie la solution au problème.

IV. Programmation dynamique (optionnel)

L'algorithme par programmation dynamique consiste à explorer progressivement l'ensemble des solutions à des problèmes plus simples, et de plus en plus compliqués ; on gagne en efficacité en construisant les solutions aux problèmes compliqués à partir de celles des problèmes simples. L'idée est donc de construire :

- pour un ensemble vide d'objets, les solutions optimales pour un sac de taille 0, puis 1...jusqu'à c ;
- pour un ensemble contenant uniquement le premier objet les solutions optimales pour un sac de taille 0, puis 1...jusqu'à c ;
- pour un ensemble contenant uniquement deux objets, les solutions optimales pour un sac de taille 0, puis 1...jusqu'à c ;
- et ainsi de suite jusqu'à l'ensemble de tous les objets.

On stocke l'ensemble de ces réponses dans deux matrices de taille $(c+1)\times(N+1)$: $M_{i,j}$ contient la valeur maximale stockée dans un sac de taille i à partir des j premiers objets, et $\Gamma_{i,j}$ contient le dictionnaire des objets (nom et prix) à choisir pour obtenir cette valeur. Ces matrices se remplissent par récurrence, d'abord toutes les lignes à colonne j fixée puis on passe à la colonne suivante, comme suit :

- si $i=0$ ou $j=0$, $\Gamma_{i,j}=$ et $M_{i,j}=0$
- sinon, si le j^{e}objet est trop grand pour entrer dans le sac de taille i ($i<j$) : on reprend la configuration optimale dans le cas où on n'utilisait pas cet objet j : $\Gamma_{i,j}=\Gamma_{i,j-1}$ et $M_{i,j}=M_{i,j-1}$
- sinon, on compare deux possiblités : soit on prend l'objet j, auquel cas il reste la place $i-p_j$, et on peut lire la configuration optimale dans cet espace restant en $\Gamma_{i-p_j,j-1}$; soit on ne prend pas l'objet j, et la configuration reste celle de $\Gamma_{i,j-1}$; on inscrit alors dans $\Gamma_{i,j}$ la meilleure de ces deux configurations, et $M_{i,j}=\max(M_{i,j-1},M_{i-p_j,j-1}+v_j)$

La meilleure réponse au problème est donc celle de la case c,N de la matrice $\Gamma_{i,j}$.

Traitons un exemple : pour les objets

```
poids = {'a': 3, 'b': 2, 'c': 1}
prix = {'a': 4, 'b': 3, 'c': 2}
```

et un sac de taille $c=5$, on construira les matrices ci-dessous (on rappelle que les indices démarrent à 0) :

$$M_{i,j}=\begin{pmatrix}0&0&0&0\\0&0&0&2\\0&0&\underline{3}&3\\0&4&\underline{4}&\mathbf{5}\\0&4&4&6\\0&4&7&7\end{pmatrix}\quad\text{et}\quad\Gamma_{i,j}=\begin{pmatrix}\{\}&\{\}&\{\}&\{\}\\\{\}&\{\}&\{\}&\{c\}\\\{\}&\{\}&\{\underline{b}\}&\{b\}\\\{\}&\{a\}&\{\underline{a}\}&\{\mathbf{b},\mathbf{c}\}\\\{\}&\{a\}&\{a\}&\{a,c\}\\\{\}&\{b\}&\{a,b\}&\{a,b\}\end{pmatrix}$$

La case en gras étant obtenue en comparant les cases soulignées, et en ajoutant l'objet c ($j=3$) à la case de la ligne $i-p_j=3-1=2$.

4 – Écrire une fonction `initMatrices` qui prend les tailles du problème et créé, initialise à zéro et renvoie les deux matrices utiles.

5 – Écrire une fonction `comparaisonDynamique(matriceValeurs, i, j, poids_j, prix_j)` qui permet de décider s'il vaut mieux ajouter l'objet obj de la colonne j dans un sac de taille i : elle renvoie `True` s'il vaut mieux ajouter l'objet, et `False` sinon.

6 – Écrire la fonction `dynamique` qui renvoie la réponse au problème.