TD 10 – Algo avancé: Programmation Dynamique (suite)

Exercice 1. *Le problème du sac à dos*

Le problème du sac à dos (discret) est le suivant : étant donné un sac à dos et plusieurs objets ayant chacun un poids (entier) et une valeur (entière), quelle est la valeur maximum d'un ensemble de ces objets qui ne dépasse pas la capacité du sac à dos ?
Dans votre problème, l'entrée est donnée sous la forme de deux tableaux p et v chacun de même taille n et tel que $p[i]$ et $v[i]$ sont respectivement le poids et la valeur du i-ème objet et d'un entier c qui correspond à la capacité du sac à dos.

1. Écrivez un programme naïf qui calcule $meilleure\_valeur(p, v, c)$ et estimez son temps de calcule.
   ☞ On peut se contenter d'énumérer toutes les 2^n possibilités, en s'arrêtant quand on dépasse la capacité du sac à dos.

```
def meilleure_valeur(p,v,c):
        if not p:
                return 0
        if p[0] > c:
                return meilleure_valeur(p[1:],v[1:],c)
        return max(meilleure_valeur(p[1:],v[1:],c), v[0]+meilleure_valeur(p[1:],v[1:],c-p[0]))
```

2. Générez aléatoirement des instances du problème par exemple avec le programme suivant :

```
import random

for n in range(1000):
        p = []
        v = []
        for _ in range(n):
                p.append(random.randint(1,100))
                v.append(random.randint(1,100))
        c = n*25
        print(n,meilleure_valeur(p,v,c))
```

   Jusqu'à quelle valeur de n votre programme est-il capable de décider en un temps raisonnable ? Estimez l'efficacité de votre programme sur ces entrées.
   ☞ Il devient lent autour de $n = 23$. De manière très grossière, on peut penser que le programme a une efficacité en $O(2^n)$: pour chaque objet, soit il est mis dans notre sac, soit il n'est pas mis dans le sac. On peut être plus précis en remarquant qu'en moyenne, un sous ensemble de plus de la moitié des objets ne rentre pas dans le sac, mais ça ne change pas beaucoup l'ordre de grandeur.

3. Le problème du sac à dos est NP-complet[1]. Vous ne trouverez donc probablement pas d'algorithme polynomial pour le résoudre dans tous les cas.
   En revanche, en utilisant la programmation dynamique, vous pouvez faire un algorithme efficace à condition que c ne soit pas trop grand (c'est-à-dire polynomial en n). Proposez un tel algorithme et donnez son efficacité.
   ☞ L'astuce consiste à remplir une matrice M de dimension $n * c$ où $\forall i \in [0, n], \forall j \in [0, c], M[i][j]$ représente la valeur maximum qu'on peut obtenir en utilisant les objets de numéro $< i$ avec un poids $\leq j$.
   Naturellement, $M[0][j] = M[i][0] = 0$ pour tout $i \in [0, n]$ et $j \in [0, c]$.

1. En réalité, le problème de décision associé au problème d'optimisation est NP-complet...

De plus, pour tout $i, j > 0$,

$$M[i][j] = \begin{cases} M[i-1][j] & \text{si } p[i-1] > j \\ max(M[i-1][j], M[i-1][j-p_{i-1}] + v_{i-1}) & \text{sinon.} \end{cases}$$

En effet, avec une capacité j et en choisissant parmi les objets 0 à $i-1$, je peux au moins obtenir la même valeur qu'avec seulement les objets de 0 à $i-2$. Mais en plus, si j'ai assez de place pour mettre l'objet $i-1$, je peux le prendre et additionner sa valeur à celle de la meilleure valeur que je peux obtenir avec une capacité $j-[p_{i-1}]$ et en choisissant parmi les objets 0 à $i-2$.

```
def meilleure_valeur2(p,v,c):
        M = [ [0]*(c+1) for _ in range(n+1)]

        for i in range(1,n+1):
                for j in range(1,c+1):
                        poids = p[i-1]
                        valeur = v[i-1]
                        if poids > j:
                                M[i][j] = M[i-1][j]
                        else:
                                M[i][j] = max(M[i-1][j],M[i-1][j-poids]+valeur)
        return M[n][c]
```

La taille de la matrice est M est $n * c$. Calculer $M[i][j]$ prend un temps $O(1)$. Donc la complexité est en $O(n * c)$.

Exercice 2. *Problème de partition*

Le problème de partition est le problème de décision qui, étant donné un multiensemble[2] S d'entiers naturels, détermine s'il est possible de le partionner en deux sous-multiensembles dont les sommes des éléments sont égaux.

PARTITION
entrée : Un multiensemble E d'entiers naturels $\mathbb{N}$.
question : Existe-t-il S un sous-multiensemble de $S \subseteq E$ tel que $\sum_{x \in S} x = \sum_{y \in E \setminus S} y$?

Ce problème est NP-complet, donc vous ne trouverez probablement d'algorithme efficace dans tous les cas pour le résoudre au cours de ce TD...

1. Montrer que ce problème appartient à NP en proposant un certificat vérifiable en temps polynomial. Écrire une fonction $verification(E, certificat)$ qui vérifie ce certificat polynomial. Vous pouvez supposer que E vous est donné sous la forme d'une liste.
   ☞ Le certificat peut-être le sous-multiensemble S qui existe bien quand E est une instance positive du problème. On peut représenter S comme une liste d'entiers qui doit être une sous-liste de E.
   Notre fonction doit vérifier deux choses :
   — Que $\sum_{x \in S} x = \sum_{y \in E \setminus S} y$. Pour ça, il suffit de vérifier que $\sum_{z \in E} z = 2 \sum_{x \in S} x$ (car $\sum_{z \in E} z = \sum_{x \in S} x + \sum_{y \in E \setminus S} y$).
   — Que S est bien une sous-liste de E.

```
def verification(E,certificat):
        if sum(E) != 2*sum(certificat):
                return False
        i = j = 0
        while i < len(E) and j < len(certificat):
                if E[i] == E[j]:
                        j+=1
                i+=1
        return j == len(certificat)
```

2. Un multiensemble est comme un ensemble mais où un même élément peut-être présent plusieurs fois. Par exemple le multiensemble $\{\{1,1,2,3\}\}$ est égal $\{\{3,1,2,1\}\}$ mais pas à $\{\{1,2,3\}\}$.

2. Proposez un programme naïf $partition(E)$ qui décide le problème PARTITION.
   Vous pouvez utiliser $itertools.combinations(lst, t)$ qui prend en entrée une liste lst[3] et une taille t et renvoie un itérable sur toutes les sous-listes de lst de taille t.
   ☞

```
import itertools

def partition(E):
        for t in range(0,len(E)):
                for certificat in itertools.combinations(E,t):
                        if verification(E,certificat):
                                return True
        return False
```

3. Tester votre algorithme avec le programme suivant :

```
for n in range(1,301):
        E = [random.randint(0,2**n) for _ in range(n)]
        print(n,partition(E))
```

   Jusqu'à quelle valeur de n votre programme arrive-t-il à calculer ? Quel est la complexité de votre algorithme en fonction du nombre d'éléments n de E et de $s = \sum_{z \in E} z$?

   ☞ Environ 25. La complexité du programme est en $O(2^n)$ en négligent la taille des entiers de E et en $O(2^n \log(s))$ sinon.
   On peut dire que la valeur de s n'influe pas beaucoup sur la complexité de l'algorithme.

4. En utilisant la programmation dynamique, proposez une fonction $partition2(E)$ qui calcule efficacement PARTITION quand la valeur de s est polynomial en n.
   ☞ On va maintenir une liste d'ensembles t où $t[i]$ est l'ensemble des $\{sum(S) \mid S \subseteq E[0:i]\}$.
   Si $S \subseteq E[0:i]$ alors $S \subseteq E[0:i+1]$ et $S \cup \{E[i]\} \subseteq E[0:i+1]$. Donc, $t[i+1] = t[i] \cup \{s + E[i] \mid s \in t[i]\}$.
   La question est : est-ce que qu'il existe $S \subseteq E$ tel que $sum(S) = sum(E)/2$? Autrement dit, est-ce que $sum(E)/2 \in t[n]$?

```
def partition2(E):
        n = len(E)
        s = sum(E)
        if s % 2==1:
                return False

        t = [{0}]

        for i in range(len(E)):
                t.append( t[i].union( {s + E[i] for s in t[i] } ) )

        return s//2 in t[n]
```

   Petite optimisation possible : On a jamais besoin que de $t[i]$ pour calculer $t[i+1]$ alors autant ne garder en mémoire que le dernier $t[i]$.

```
def partition3(E):
        n = len(E)
        s = sum(E)
        if s % 2==1:
                return False

        sommes = {0}

        for i in range(len(E)):
                sommes = sommes.union( {s + E[i] for s in sommes } )

        return s//2 in sommes
```

   Autre optimisation possible : $sum(S)$ ne doit jamais dépasser $sum(E)/2$. Plus subtil : $sum(E \setminus S)$ non plus...

5. Donnez la complexité de votre algorithme en fonction de s et n et testez votre nouveau programme avec les exemples aléatoire suivants :

3. En réalité tout itérable... voir la page itertools de la doc Python

```
for n in range(1,301):
        E = [random.randint(0,n) for _ in range(n)]
        print(n,partition(E))
```

☞ $O(s*n)$ Ici $s < n^2$ donc $O(n^3)$.