Interrogation d'informatique n°4. *Barème sur 23 pts*

Exercice A. Tableaux binaires *Les deux questions sont indépendantes*

1) On rappelle que pour n entier, `n%2` et `n//2` renvoient le reste et le quotient de n par 2.

a) [1 pt] On considère la fonction suivante qui étant donnés deux entiers naturels k et n vérifiant $0 \leq k < 2^n$ renvoie la n-liste $[a_0, ..., a_{n-1}]$ telle que $k = \sum_{i=0}^{n-1} a_i 2^i$, avec $a_i \in \{0, 1\}$ pour tout i.

```
01    def binaire(k,n) :
02        L = [] ; q = k
03        for i in range(n) :
04            L.append(q%2)
05            ###  à compléter  ###
06        return L
```

Compléter *sans justification* la ligne 05 et expliciter la liste renvoyée par `binaire(6,6)`.

b) [1 pt] En déduire une fonction `listes(n)` qui renvoie la liste des n-listes de $\{0, 1\}^n$.

Par exemple, `listes(0)` renvoie `[[]]` et `liste(2)` renvoie `[[0,0],[1,0],[0,1],[1,1]]`.

2) On considère le programme suivant :

```
01    def listes_aux(k,tab_partiel,sol):
02        if k == len(tab_partiel) :
03            sol.append(tab_partiel.copy())
04        else :
05            tab_partiel[k] = 0
06            liste_aux(k+1,tab_partiel,sol):
07            tab_partiel[k] = 1
08            liste_aux(k+1,tab_partiel,sol)

09    def listes(n) :
10        sol = [] ; tab_partiel = [-1]*n
11        listes_aux(0,tab_partiel,sol)
12        return sol
```

La fonction `liste_aux` est une procédure qui modifie le tableau `tab_partiel` (aux lignes 05 et 07) et la liste `sol` (à la ligne 03).

a) [1.5 pt] Expliciter *sans justification* la liste `sol` renvoyée par l'instruction `listes(2)`.
On prendra soin à bien donner la liste dans le bon ordre.

b) [2 pts] Expliciter *sans justification* TOUTES les valeurs successives prises par le tableau `tab_partiel` lorsqu'on exécute `listes(2)`. Le tableau `tab_partiel` est créé et initialisé à la ligne 10.

b) [1 pt] Expliquer brièvement la nécessité de faire une copie dans `partiel.copy()` à la ligne 03.

Exercice B. Problème du sac-à-dos

On dispose d'un sac-à-dos et de n objets indexés de 0 à $n-1$.

La capacité du sac-à-dos, qui est le poids maximal que peut supporter le sac, est notée C.

A chaque objet d'indice i est associé deux réels strictement positifs : un poids p_i et une valeur v_i.

On cherche à remplir le sac en maximisant la somme des valeurs des objets contenus dans le sac en imposant que la somme des poids soit inférieure ou égale à C.

Les paramètres sont donc un réel positif C et la liste L des couples (*poids,valeur*) des n objets.

1) [1 pt] Écrire une fonction `depaq(L)` qui étant donnée une liste L de n couples (p_i, v_i), renvoie le couple de listes (p, v), où $p = [p_0, ..., p_{n-1}]$ et $v = [v_0, ..., v_{n-1}]$.

Par exemple, `depaq([(1,2),(3,4),(2,4)]` renvoie `([1,3,2],[2,4,4])`.

2) Algorithme glouton

La stratégie gloutonne consiste à classer les objets de sorte que le quotient valeur/poids soit décroissante, et ensuite à remplir le sac-à-dos par les objets de la liste pris dans l'ordre avec comme condition que la somme des poids des objets sélectionnés ne dépasse par C.

Autrement dit, on suppose que les objets sont classés de sorte que

$$\frac{v_0}{p_0} \geq \frac{v_1}{p_1} \geq ... \geq \frac{v_{n-1}}{p_{n-1}}$$

Exemple : On prend $C = 4$, $n = 5$ et les poids et valeurs :

$p_0 = 3$, $p_1 = p_2 = 2$, $p_3 = 1$ et $v_4 = 8$ et $v_1 = v_2 = 5$ et $v_3 = 1$.

Les objets sont classés par valeur massique décroissante : $\frac{v_0}{p_0} = \frac{8}{3} \geq \frac{v_1}{p_1} = \frac{5}{2} = \frac{v_2}{p_2} \geq \frac{1}{1} = \frac{v_3}{p_3}$.

Par la stratégie gloutonne, on choisit d'abord l'objet d'indice 0 puis l'objet d'indice 3.

Mais la stratégie optimale consiste à choisir les objets 1 et 2, car on a $5 + 5 > 8 + 1$.

a) [2.5 pts] Écrire une fonction `glouton(L,C)` qui renvoie la valeur totale du sac-à-dos rempli selon la méthode gloutonne.

Par exemple, `glouton([(3,8),(2,5),(2,5),(1,1)],4)` renvoie `9`.

b) [1 pt] Préciser *sans justification* la complexité de la fonction `glouton`.

Préciser aussi la complexité totale si on doit préalablement classer les objets par valeur massique décroissante.

3) Force brute

On propose d'abord un algorithme consistant à explorer *toutes* les combinaisons possibles.

On pourrait exploiter les fonctions de l'exercice précédent, mais on va procéder directement.

a) [0.5 pt] Écrire une fonction `somme(v,a)` qui étant données deux listes v et a de longueur n composées de nombres entiers ou flottants, renvoie la somme $\sum_{i=0}^{n-1} v_i a_i$.

b) [2.5 pts] On considère un entier k tel que $0 \leq k < 2^n$. L'entier k s'écrit $\sum_{i=0}^{n-1} a_i 2^i$, où $a_i \in \{0,1\}$.
On dit que la liste $a = [a_0, ..., a_{n-1}]$ est le code de l'entier k (il s'agit de l'écriture de k en base 2).

Écrire une procédure `successeur(a)` qui étant donnée une liste a de longueur n codant un entier k vérifiant $0 \leq k < 2^n - 1$ modifie a en une liste codant l'entier $(k+1)$.

Par exemple, si a vaut `[1,1,0,1,0]`, `successeur(a)` modifie a en `[0,0,1,1,0]`.

On demande une fonction de complexité linéaire $O(n)$ en la longueur n de a.

Indication : Il s'agit en fait de remplacer dans a le premier chiffre 0 par un chiffre 1, et de remplacer tous les chiffres 1 qui le précèdent par un chiffre 0. Un tel chiffre 0 existe car $k \neq 2^n - 1$.

c) [2 pts] Écrire une fonction `optimal(L,C)` qui étant donnés la liste L des couples (*poids,valeur*) et un réel positif C renvoie la valeur maximale d'un sac-à-dos de capacité C contenant des objets de L.

Indication : Utiliser les fonctions définies en a) et b) ainsi qu'une boucle `for`.
La complexité attendue de la fonction `optimal(L,C)` est $O(n\ 2^n)$.

4) Programmation dynamique

Le problème du sac-à-dos dans le cas général est de complexité exponentielle.
Mais lorsque les p_i sont des entiers entiers, il existe un algorithme de complexité polynomiale.

On suppose ici que les p_i sont des entiers naturels non nuls et que C est lui aussi un entier naturel.

a) [2 pts] Pour $0 \leq i \leq n$ et $0 \leq k \leq C$, on note $M[i][k]$ la valeur maximale d'un sac-à-dos de capacité k contenant des objets choisis parmi les i premiers objets de L, c'est-à-dire parmi les objets (p_0, v_0), ..., (p_{i-1}, v_{i-1}).

Expliciter sans justification $M[0][k]$ pour tout $0 \leq k \leq C$.

Pour $0 \leq i < n$, exprimer sans justification $M[i+1][k]$ en fonction de $M[i][k]$, de $M[i][k-p_i]$ et de v_i.

On distinguera deux cas selon que $p_i \leq k$ et $p_i > k$.

b) [2.5 pts] Écrire une fonction `opt_dyn(L,C)` qui étant donnés la liste L de n objets et un entier C renvoie la valeur maximale d'un sac-à-dos de capacité C.

On ititialisera `M` par `M = [[0 for k in range(C+1)] for i in range(n+1)]`.

On demande une fonction de complexité $O(nC)$.

c) [2.5 pts] La connaissance de la matrice des $M[i][k]$ permet de reconstituer les objets qui peuvent été sélectionnés pour obtenir un sac de capacité C et de valeur maximale $M[n][C]$.

Il convient de partir de $M[n][C]$. Le dernier objet (p_{n-1}, v_{n-1}) est sélectionné ssi $M[n-1][C] < M[n][C]$.
On peut alors déterminer de proche en proche la liste des objets à sélectionner.

Écrire une fonction `sac(M,L)` qui étant donné la matrice `M` obtenue au b) renvoie les indices des objets qui peuvent être sélectionnés pour obtenir la valeur maximale d'un sac de capacité C.

Préciser la complexité de la fonction proposée.

Remarque : On a ici `n = len(M)-1 ; C = len(M[0])-1`

5) Question supplémentaire. Recherche exhaustive avec back-tracking

La recherche exhaustive du 3) est généralement trop coûteuse.
On peut l'améliorer un tant soit peu en utilisant la technique dite du "back-tracking".

On suppose désormais que dans la liste L, **les objets classés par poids décroissant**.

Par exemple, considérons $L = [(12, 6), (7, 8), (4, 3), (3, 5), (2, 3)]$. On a bien $12 > 7 > 4 > 3 > 2$.

Une recherche exhaustive peut se représenter par l'arbre du schéma (1) : on lit les objets dans l'ordre et à chaque étape, on choisit ou non chaque nouvel objet ; sur le schéma, pour alléger, on a mentionné uniquement le poids des objets sélectionnés. Sur chaque branche, le poids décroît le long de la branche.

Lorsqu'on parcourt une branche depuis la racine, il arrive que la somme des poids des objets de la branche partielle dépasse la capacité totale du sac-à-dos. C'est pourquoi **on peut alors élaguer l'arbre** en tronquant les branches lorsqu'on ne peut plus ajouter d'objets sans dépasser la capacité.

Par exemple, si on prend $C = 10$, on ne peut pas prendre l'objet $(12, 6)$ de poids 12.
Si on choisit l'objet $(7, 8)$, on ne pourra considérer ensuite que l'un des objets $(3, 5)$ et $(2, 3)$.

On a représenté sur le schéma (2) l'arbre ainsi élagué. La valeur maximale recherchée est alors le maximum des sommes des valeurs (non indiquées sur le schéma) appartenant à une même branche.

a) Écrire, en suivant ce principe, une fonction récursive `opt_back(L,i,C)` qui étant donnés une liste L de n couples (*poids,valeur*) et un entier i (compris entre 0 et n) renvoie la valeur maximale des objets qu'il est possible de mettre dans un sac-à-dos de capacité C en prenant les objets parmi $L[i]$, $L[i+1]$, ... , $L[n-1]$.

On rappelle que dans la liste L, les éléments sont classés par ordre décroissant du poids.

Remarque : Autrement dit, on cherche à écrire une fonction dont la complexité est liée au nombre de sommets de l'arbre élagué (et non de l'arbre originel). Le choix de classer les éléments de L par poids décroissant permet d'obtenir un élagage plus efficace.

b) Écrire une version itérative `optimal(L,C)` en utilisant :

- une pile qui mémorise la liste (des indices) des objets qui appartiennent à la branche en cours d'étude
- des variables `v_pile` et `p_pile` qui donnent la valeur et le poids total des sommets de la branche
- une variable `v_max` qui donne la valeur maximale des branches valides déjà parcourues.

Indication : Noter que dans ce parcours en profondeur, on passe d'une branche codée par la pile $[..., i]$ à la suivante soit en ajoutant j à la pile soit en remplaçant i par j dans la pile, où $\boxed{j = i + 1}$.