DS 5

CORRIGÉ

I Réductions

▸ **Question 1** Pour 0/1-Knapsack, on prend la partie I comme certificat et l'on vérifie que $\sum_{i\in I} v_i \geqslant V$ et $\sum_{i\in I} w_i \leqslant W$, ce qui peut bien se faire en temps polynomial.
De même pour Knapsack en prenant cette fois le n-uplet $x_0, \dots, x_{n-1}$ comme certificat.

▸ **Question 2** Il reste à montrer que 0/1-Knapsack est NP-dur. Pour cela, on considère une instance (X, S) de SubsetSum, avec $X = x_0, \dots, x_{n-1}$ et l'on définit l'instance suivante de 0/1-Knapsack, que l'on peut facilement produire en temps polynomial en $|(X, S)|$:

- les n objets sont les couples (x_i, x_i) ;
- $W = S$;
- $V = S$.

Cette instance est positive si et seulement si il existe I tel que $\sum_{i\in I} x_i \leqslant W = S$ et $\sum_{i\in I} x_i \geqslant V = S$, donc si et seulement si il existe I tel que $\sum_{i\in I} x_i = S$, c'est-à-dire si et seulement si (X, S) est une instance positive de SubsetSum. On en déduit que SubsetSum $\leqslant_p$ 0/1-Knapsack. Étant donné que SubsetSum est NP-complet et que 0/1-Knapsack $\in$ NP, on conclut que 0/1-Knapsack est NP-complet.

▸ **Question 3** Soit $I \subset [0 \dots n-1]$ tel que $\sum_{i\in I} x_i = S$. Pour $0 \leqslant i < n$, on pose :

- $\lambda_i = 0$ si $i \in I$, $\lambda_i = 1$ sinon ;
- $\lambda'_i = 1$ si $i \in I$, $\lambda'_i = 0$ sinon.

On a alors :

$$\begin{aligned}
\sum_{i=0}^{n-1} \lambda_i y_i + \sum_{i=0}^{n-1} \lambda'_i y'_i &= \sum_{i\in I} y'_i + \sum_{i\notin I} y_i \\
&= \sum_{i=0}^{n-1} (2^n + 2^i) nB + \sum_{i\in I} x_i \\
&= nB \left(2^n + \sum_{i=0}^{n-1} 2^i \right) + S \\
&= S'
\end{aligned}$$

Comme $V = W = S'$, G est bien une instance positive de Knapsack.

▸ **Question 4**
Comme $n2^n B \leqslant y_i < y'_i$ pour tout $i \in [0 \dots n-1]$, on a :

$$\begin{aligned}
\sum_{i=0}^{n-1} \lambda_i y_i + \sum_{i=0}^{n-1} \lambda'_i y'_i &\geqslant n2^n B \sum_{i=0}^{n-1} (\lambda_i + \lambda'_i) \\
&= n2^n BN
\end{aligned}$$

D'après notre hypothèse, cela signifie que $S' \geqslant n2^n BN$. Or :

$$\begin{aligned}
S' &= (n2^n + 2^n - 1)nB + S \\
&< ((n+1)2^n - 1)nB + B && \text{par définition de B} \\
&\leqslant n(n+1)2^n B
\end{aligned}$$

Ainsi, $n2^n BN < n(n+1)2^n B$ et donc $N \leqslant n$ puisque N et $n+1$ sont entiers.

▶ **Question 5** On a :

$$\begin{aligned} S' &= \sum_{i=0}^{n-1} (\lambda_i y_i + \lambda'_i y'_i) \\ &= nB \sum_{i=0}^{n-1} (\lambda_i + \lambda'_i)(2^n + 2^i) + \sum_{i=0}^{n-1} \lambda'_i x_i \\ &= nB \left(N2^n + \sum_{i=0}^{n-1} (\lambda_i + \lambda'_i) 2^i \right) + \sum_{i=0}^{n-1} \lambda'_i x_i \end{aligned}$$

Or $\sum_{i=0}^{n-1} \lambda'_i \leqslant n$ d'après la question précédente, donc $\sum_{i=0}^{n-1} \lambda'_i x_i < nB$ par définition de B. L'égalité ci-dessus est donc une division euclidienne de S' par nB, tout comme la définition de S' (puisque $0 \leqslant S < B$), d'où :

- $S = \sum_{i=0}^{n-1} \lambda'_i x_i$
- $N2^n + \sum_{i=0}^{n-1} (\lambda_i + \lambda'_i) 2^i = n2^n + \sum_{i=0}^{n-1} 2^i$

▶ **Question 6** On applique le résultat admis à la deuxième égalité prise modulo 2^n, on obtient $\lambda_i + \lambda'_i = 1$ pour tout i, et donc $\lambda'_i \in \{0, 1\}$ pour tout i. En posant $I = \{i \in [0 \dots n-1] \mid \lambda'_i = 1\}$, la première égalité devient alors $S = \sum_{i \in I} x_i$, et F est donc une instance positive de SubsetSum. On a donc :

- $F \in I_+(\text{SubsetSum})$ si et seulement si $G \in I_+(\text{Knapsack})$;
- G peut être construit en temps polynomial à partir de F (la question éventuelle porte sur les multiplications par nB, mais elles se font en temps polynomial en $\log(nB)$, donc polynomial en $|F|$) ;
- Knapsack $\in$ NP et SubsetSum est NP-difficile.

On conclut donc que Knapsack est NP-complet.

II Algorithmes d'approximation

▶ **Question 7** En notant $\lambda^*_0, \dots, \lambda^*_{n-1}$ la solution optimale, on a :

$$\begin{aligned} v^* &= \sum_{i=0}^{n-1} \lambda^*_i v_i \\ &= \sum_{i=0}^{n-1} \lambda^*_i \frac{v_i}{w_i} w_i \\ &\leqslant \frac{v_0}{w_0} \sum_{i=0}^{n-1} \lambda^*_i w_i && \text{décroissance des } v_i/w_i \\ &\leqslant \frac{v_0}{w_0} W && \text{d'après la contrainte sur le poids total} \end{aligned}$$

▶ **Question 8** Notons $\lambda_0, \dots, \lambda_{n-1}$ la solution renvoyée par l'algorithme glouton et v la valeur totale associée. On a $(1 + \lambda_0) w_0 > W$ d'après l'algorithme, donc $2\lambda_0 w_0 > W$ puisque $\lambda_0 \geqslant 1$ (puisque $w_0 \leqslant W$). On en déduit :

$$\begin{aligned} v &\geqslant \lambda_0 v_0 \\ &> \frac{W}{2w_0} v_0 \\ &= \frac{1}{2} W \frac{v_0}{w_0} \\ &\geqslant \frac{1}{2} v^* \end{aligned}$$

L'algorithme fournit bien une 1/2-approximation.

▸ **Question 9** On crée un tableau $(0, \dots, n-1)$ que l'on trie par valeurs décroissantes de v_i/w_i à l'aide de `Array.sort`. On utilise le signe de $v_j w_i - v_i w_j$ et non celui de $\frac{v_j}{w_j} - \frac{v_i}{w_i}$ puisque les divisions seraient des divisions entières.

```
let by_ratio v w =
  let n = Array.length v in
  let indices = Array.init n (fun i -> i) in
  let cmp i j = v.(j) * w.(i) - v.(i) * w.(j) in
  Array.sort cmp indices;
  indices
```

▸ **Question 10** On remarque que λ_i correspond au quotient de la division euclidienne de la capacité restant disponible par w_i. La fonction `by_ratio` permet de traiter les objets par v/w décroissant.

```
let greedy v w capacity =
  let n = Array.length v in
  let lambda = Array.make n 0 in
  let r = ref capacity in
  let indices = by_ratio v w in
  for i = 0 to n - 1 do
    lambda.(indices.(i)) <- !r / w.(indices.(i));
    r := !r - lambda.(indices.(i)) * w.(indices.(i))
  done;
  lambda
```

▸ **Question 11** L'appel à `by_ratio` se fait en temps $O(n \log n)$, et la boucle s'exécute ensuite clairement en temps linéaire en n. Au total, on a donc une complexité en $O(n \log n)$.

▸ **Question 12** Considérons l'instance avec deux objets $(1,1)$ et $(N-1, N)$ et $W = N$. L'algorithme glouton choisira le premier objet puisque $\frac{N-1}{N} < 1$ et on obtiendra donc $v = 1$, alors qu'on a immédiatement $v^* = N-1$. Le rapport d'approximation est donc $\frac{1}{N-1}$. Si l'on fixe $\alpha > 0$, il suffit donc de choisir $N + 1 > 1/\alpha$ pour obtenir un rapport d'approximation strictement inférieur à α : l'algorithme ne fournit pas une α-approximation.

▸ **Question 13** C'est immédiat : toute solution admissible pour 0/1-Knapsack$_o$ est également admissible pour Fractional-0/1-Knapsack$_o$.

▸ **Question 14** Il faut utiliser au maximum les objets ayant un bon ratio valeur/poids. On propose l'algorithme suivant :

Trier les objets par v_i/w_i décroissant.
$\lambda \leftarrow (0, \dots, 0)$ (taille n)
$R \leftarrow W$
pour $i = 0$ à $n-1$ **faire**
  $\lambda[i] \leftarrow \min(1, R/w_i)$
  $R \leftarrow R - \lambda[i] \cdot w_i$
renvoyer λ

Le tri se fait en $O(n \log n)$ et le reste en $O(n)$, donc $O(n \log n)$ au total.

▸ **Question 15** Considérons I et j en fin d'exécution. Si $j = -1$, alors on a pris tous les objets, ce qui est clairement optimal : on suppose donc à présent $j \neq -1$.
On constate que l'algorithme de la question précédente aurait renvoyé $\lambda_0 = \cdots = \lambda_{j-1} = 1$, $\lambda_j \in [0, 1[$ et $\lambda_i = 0$ pour $i > j$, pour une valeur totale $V_f^* \geqslant V^*$. On a donc :

$$\begin{aligned}\sum_{i \in I \cup \{j\}} v_i &= \underbrace{\sum_{i=0}^{j-1} v_i + \lambda_j v_j}_{V_f^*} + (1-\lambda_j) v_j + \sum_{i \in I, i > j} v_i \\ &\geqslant V_f^* \\ &\geqslant V^*\end{aligned}$$

Ainsi, $v_j + \sum_{i \in I} v_i \geqslant V^*$, et donc $\max\left(v_j, \sum_{i \in I} v_i\right) \geqslant \frac{V^*}{2}$. Or ce maximum est précisément la valeur de la solution renvoyée, donc on a bien une 1/2-approximation.

III Résolution exacte

▸ **Question 16** On implémente l'algorithme de la question 14, qui a clairement la complexité demandée :

```
let relax (v, w, rem_capacity) k =
  let n = Array.length v in
  let sum_v = ref 0. in
  let rem_w = ref (float rem_capacity) in
  for i = k to n - 1 do
    let lambda = min 1. (!rem_w /. float w.(i)) in
    sum_v := !sum_v +. lambda *. float v.(i);
    rem_w := !rem_w -. lambda *. float w.(i)
  done;
  !sum_v
```

▸ **Question 17** Étant donnée une solution partielle s de taille k, on note $\mathrm{cap}(s)$ la capacité restante et $\mathrm{val}(s)$ la valeur des objets présents. Si s' étend s, on a nécessairement $\mathrm{val}(s') \leqslant \mathrm{val}(s) + r^*$, où r^* est la valeur optimale de l'instance $(v_k, \dots, v_{n-1}), (w_k, \dots, w_{n-1}, \mathrm{cap}(s)$ de 0/1-Knapsack$_o$. Or `relax` fournit la valeur optimale r_f^* de cette même instance vue comme une instance de Fractional-0/1-Knapsack$_o$, et l'on a donc $r_f^* \geqslant r^*$, et même $\lfloor r_f^* \rfloor \geqslant r^*$ puisque r^* est entier. On en déduit la condition d'élagage.

On écrit ensuite la fonction, dans laquelle on a choisi de toujours s'autoriser à prendre l'objet k et donc de vérifier au début de la fonction `aux` que l'on n'a pas dépassé le poids total autorisé. Les objets étant classés par rapport valeur sur poids décroissant, il est plus intéressant de commencer par explorer la branche dans laquelle on prend l'objet, pour obtenir rapidement une solution « pas trop mauvaise ».

Dans la fonction `aux`, le paramètre `k` indique la taille actuelle de la solution partielle, `sum_v` la somme des valeurs des objets déjà choisis et `cap` la capacité restante étant donnés les choix déjà effectués.

```
let solve ((v, w, capacity) : instance) =
  let n = Array.length v in
  let sol = Array.make n false in
  let opt_v = ref 0 in
  let opt_sol = ref [||] in
  let rec aux k sum_v cap =
    if cap < 0 then ()
    else if k = n && sum_v > !opt_v then (
      opt_v := sum_v;
      opt_sol := Array.copy sol
    ) else if k < n
              && sum_v + int_of_float (relax (v, w, cap) k) > !opt_v then (
      sol.(k) <- true;
      aux (k + 1) (sum_v + v.(k)) (cap - w.(k));
      sol.(k) <- false;
      aux (k + 1) sum_v cap
    ) in
  aux 0 0 capacity;
  !opt_sol, !opt_v
```