TP n°13 - Introduction à la programmation dynamique

Notions abordées

- Principe de mémoïsation pour des suites récurentes (valeurs enregistrées dans tableau 1D)
- Établir une définition récursive de la solution optimale pour le sac-à-dos entier
- Résolution du problème du sac-à-dos (entier) par prog. dyn. (avec un tableau 2D)
- Amélioration de la complexité spatiale d'un algorithme avec mémoïsation ou de prog. dyn.
- Pratique de Ocaml : utilisation des tableaux (y compris 2D), boucles `for`

Toutes les fonctions doivent être commentées et **testées**. Elles doivent notamment être munies d'une description faisant suite à d'éventuelles hypothèses sur leurs arguments.

On pourra utiliser les fonctions du module `Array` dont la documentation est disponible à l'adresse suivante. `https://ocaml.org/releases/4.12/api/Array.html`

Exercice 0 Tableaux 2D en Ocaml

Question 1 💻

Dans l'interpréteur `utop`, créer `t` un tableau de 10 entiers initialisés à 0. Vérifier que cette création s'est passée comme prévu. Modifier la valeur d'indice 2 de ce tableau. Vérifier que cette modification s'est passée comme prévu.

Question 2 💻

Dans l'interpréteur `utop`, créer `t` un tableau de 5 tableaux de 3 entiers initialisés à 0. Vérifier que cette création s'est passée comme prévu. Modifier la valeur d'indice 2 du premier tableau de `t`. Vérifier que cette modification s'est passée comme prévu.

Question 3 💻

Dans l'interpréteur `utop`, créer `t` un tableau de 5 tableaux vides. À l'aide d'une boucle `for` remplacer chacun de ces 5 tableaux par un tableau de 3 entiers initialisés à 0. Vérifier que cette création s'est passée comme prévu. Modifier la valeur d'indice 2 du premier tableau de `t`. Vérifier que cette modification s'est passée comme prévu.

Question 4 💻

À l'aide de la fonction `Array.make_matrix`, créer directement un tableau de 5 tableaux de 3 entiers initialisés à 0. Modifier la valeur d'indice 2 du premier tableau de `t`. Vérifier que cette modification s'est passée comme prévu.

Question 5 ✍

Faire un bilan de la syntaxe à adopter pour la création de tableaux à une ou deux dimensions.

Exercice 1 Suites récursives et mémoïsation

Question 1 ✍
On rappelle que la suite de Fibonacci est définie par $F_0 = 0$, $F_1 = 0$ et $\forall n \in \mathbb{N}$, $F_{n+2} = F_{n+1} + F_n$.
Que dire de la fonction récursive `fibo_naif` qui traduit directement cette définition récursive ? Combien d'appels récursifs fait-elle apparaître ?

Question 2 ✍
Représenter sur un axe gradué les appels récursifs de l'appel `fibo_naif 8`. Combien de fois est appelée `fibo_naif 6` ? Combien de fois est appelée `fibo_naif 3` ?

Une idée générique pour gagner en efficacité dans ces cas là, c'est-à-dire lorsqu'on est face à une fonction qui réalise certains appels (récursifs) de nombreuses fois, est de stocker la valeur de ces appels. Ce principe s'appelle la **memoïsation**.

Dans le cas de la fonction `fibo_naif`, on pourrait à chaque appel sur un entier n enregistrer la valeur F_n calculée, dans la case d'indice n d'un tableau par exemple, et au prochain appel de la fonction sur ce même entier n, il suffira de récupérer la valeur enregistrée plutôt que de la recalculer.

Question 3 🖳
Définir une fonction `fibo_memo_unique` qui prend en argument un entier n et qui calcule F_n par memoïsation. *Avant de commencer les appels récursifs il faut de créer un tableau prêt à recevoir les valeurs calculées au fil des appels récursifs. De plus il faudra pouvoir déterminer si la valeur F_n a déjà été calculée, ou si au contraire la case correspondante est encore "vide".*

Question 4 🖳
Effectuer quelques appels aux fonctions `fibo_naif` et `fibo_memo_unique` directement dans `utop` afin de constater le gain d'efficacité dû à la mémoïsation.

Question 5 🖳 ∗
Définir une fonction `fibo_memo_dynamique` qui prend en argument un entier n et un tableau d'entiers de taille quelconque contenant les valeurs de la suite de Fibonacci précédemment calculées, qui calcule F_n et met à jour le tableau. *La mise à jour du tableau ne se limite pas toujours au remplissage de nouvelles cases, elle peut aussi nécessiter un redimensionnement du tableau, c'est pourquoi on ne peut se contenter d'un argument de type* `'a array`, *qui est une famille de* `'a ref` *de taille fixée. Il faut utiliser une référence vers un tableau, i.e. un objet de type* `'a array ref`.

Question 6 🖳
On considère la suite u définie par $u_0 = u_1 = u_2 = 1$ et $\forall n \in \mathbb{N}, n \geqslant 3, u_n = u_{n/3+r} + r u_{n-r}$ où r est le reste modulo 3 de n. Définir une fonction `u_memo` qui prend en argument un entier n et qui calcule u_n memoïsation.

Question 7 🖳
Lors d'un appel à `fibo_memo_unique`, un tableau est crée. Quelles cases de ce tableau sont remplies à la fin de cet appel ? En déduire `fibo_memo_for` une alternative itérative à `fibo_memo_unique`.

Question 8 ✍ 🖳
Quelle est sa complexité temporelle de `fibo_memo_for` ? Et sa complexité spatiale ? Peut-on améliorer cette dernière ? Expliquer comment puis en déduire une fonction `fibo_malin`.

Question 9 ✍ 🖳
Peut-on faire de même pour le calcul des termes de la suite u ?

Exercice 2 Problème du sac-à-dos entier

Dans cet exercice on s'intéresse au problème du sac-à-dos en supposant que les poids des objets et du sac sont entiers.

Question 1 ✍ ∗
Expliquer comment le problème où les poids des objets sont des nombres rationnels peut se ramener au problème étudié ici avec des poids entiers. Expliquer pourquoi cette hypothèse d'intégrité des poids n'est pas réellement restrictive.

On considère $I = (p, v, P) \in \mathbb{N}^N \times (\mathbb{R}_+^*)^N \times \mathbb{N}$ une instance du problème du sac-à-dos entier à $N \in \mathbb{N}^*$ objets. On note $\mathcal{S}_I = \{\delta \in \{0,1\}^N \mid \delta \cdot p \leqslant P\}$ l'ensemble des solutions pour I. Pour une solution $\delta \in \mathcal{S}_I$, on appelle poids de la solution la quantité $\delta \cdot p$, et valeur de la solution la quantité $\delta \cdot v$.

Question 2 ✍
Que dire du poids d'une solution pour I ?

Pour $k \in [1..N]$, on note π_k la projection sur les k premières composantes (*Cf.* définition formelle ci-dessous) et pour $\delta \in \{0,1\}^k$ on appelle poids de δ la quantité $\delta \cdot \pi_k(p)$.

$$\pi_k = \left(\begin{array}{lcl} \mathbb{R}^N & \to & \mathbb{R}^k \\ (x_i)_{i\in[1..N]} & \mapsto & (x_i)_{i\in[1..k]} \end{array} \right)$$

Question 3 ✍
Pour $k \in [2..N]$, que dire du poids de $\pi_k(\delta)$ pour $\delta \in \mathcal{S}_I$ telle que $\delta_k = 1$?

Pour $k \in [0..N]$ et $W \in \mathbb{N}$, on note $V(k, W)$ la valeur maximale d'une solution de poids inférieur à W et constituée uniquement d'objets d'indice dans $[1..k]$.

Question 4 ✍
Donner une définition formelle explicite de $V(k, W)$ pour $k \in [0..N]$ et $W \in \mathbb{N}$.

Question 5 ✍
Pour $W \in \mathbb{N}$, que vaut $V(0, W)$?

Question 6 ✍
Pour $k \in [1..N]$ et $W \in \mathbb{N}$, exprimer $V(k, W)$ en fonction de valeurs de la forme $V(k', W')$ avec $k' < k$ et $W' \leqslant W$? *On pourra distinguer le cas où $p_k > W$, du cas où $p_k \leqslant W$, et dans ce dernier cas envisager les deux possibilités, à savoir mettre l'objet k dans le sac ou ne pas le mettre, et sélectionner la meilleure.*

Question 7 ✍
Quelle valeur de V donne valeur d'une solution optimale pour l'instance I ? *On attend des paramètres k et W tels que $V(k, W)$ donne cette valeur optimale.*

Question 8 ✍
Déduire des questions précédentes le pseudo-code d'un algorithme résolvant le problème du sac-à-dos entier. *On attend un algorithme itératif consistant principalement à remplir un tableau bien choisi.*

Question 9 🖳
Définir une fonction `val_opt_sac` qui prend en argument une instance du sac-à-dos sous la forme d'un entier pour le poids maximal du sac, et de deux tableaux de même taille pour les poids et les valeurs des objets, et qui calcule la valeur d'une solution optimale de ce problème.

Question 10 ✍
Exprimer la complexité temporelle de `val_opt_sac` en fonction des paramètres de l'instance (p, v, N et P). Est-ce que cette complexité est polynomiale en la taille de l'instance ?

Question 11 🖳
Définir une fonction `sol_opt_sac` qui prend en argument une instance du sac-à-dos (sous la même forme que précédemment) et qui calcule une solution optimale de ce problème sous la forme d'un tableau de booléens indiquant les objets pris par exemple.

Question 12 🖳
Définir, si c'est possible, une fonction `val_opt_sac_bis` qui a la même description que `val_opt_sac` mais qui a une meilleure complexité spatiale, mais la même complexité temporelle.

Question 13 🖳
Définir, si c'est possible, une fonction `sol_opt_sac_bis` qui a la même description que `sol_opt_sac` mais qui a une meilleure complexité spatiale, mais la même complexité temporelle.

Question 14 🖳 ∗∗
Coder l'algorithme glouton vu en classe et comparer les valeurs des solutions obtenues par les deux algorithmes sur des instances générées aléatoirement. *C'est l'occasion de tester l'implémentation d'un des algorithmes de tri vu en cours. Pour la génération aléatoire d'entiers, on pourra utiliser le module* `Random` *dont la documentation est disponible sur* `https://ocaml.org/releases/4.12/api/Random.html`

Question 15 🖳 ∗
Définir une fonction `val_opt_sac_bf` *"bf" pour "brute force"* qui donne la valeur optimale pour une instance du sac-à-dos entier en faisant une recherche exhaustive du maximum, puis comparer les temps d'exécution de cette fonction et de `val_opt_sac`. *On pourra s'inspirer de la génération des environnements propositionnels pour générer tous les sous-ensemble d'objets, puis ne considérer que ceux pouvant rentrer dans le sac.*