TP #25 — Programmation dynamique (exercices variés)

Éléments de programmation OCaml

On fournit ci-dessous quelques éléments de programmation OCaml utiles pour l'un ou plusieurs des exercices.

Module `Hashtbl`. La bibliothèque standard OCaml fournit une implémentation de table de hachage (implémentant une structure de données de tableau associatif) dans le module `Hashtbl`. Les fonctions de ce module au programme (à savoir utiliser si l'on vous en fournit une documentation) sont :

- `Hashtbl.create`
- `Hashtbl.add`
- `Hashtbl.remove`
- `Hashtbl.mem`
- `Hashtbl.find`
- `Hashtbl.find_opt`

Par ailleurs, l'usage de `Hashtbl.add` mérite en réalité d'être remplacé par celui de `Hashtbl.replace`.

Chaînes de caractères. Les chaînes de caractères sont typiquement représentées en OCaml *via* le type `string`, dont les éléments sont de type `char` : les `string`s sont des tableaux **immuables** de `char`.
Les littéraux de type `string` s'écrivent entre « `"` » (par exemple : `"hewwo"`) et ceux de type `char` entre « `'` » (par exemple : `'h'`).
L'accès au $\texttt{i}^{\text{ème}}$ (en partant de zéro) élément d'une `string` se fait par la syntaxe `.[i]` (une syntaxe différente de celle « `.(i)` des `array`s est nécessaire pour l'inférence de type) :

```
utop # let grt = "hewwo";;
val grt : string = "hewwo"
utop # grt.[0];;
- : char = 'h'
utop # grt.[2] <- 'l';;
Error: Syntax error: "strings are immutable, there is no
assignment syntax for them".
```

La bibliothèque standard OCaml fournit un certain nombre de fonctions opérant sur les `string`s dans le module `String`. Seules deux d'entre elles sont au programme : `String.length` et la concaténation entre deux chaînes `(^)`, ou `String.cat` :

```
utop # "hewwo?" ^ " OHAI!";;
- : string = "hewwo? OHAI!"
```
Dans ce sujet, il pourra cependant également être utile d'utiliser la fonction `String.make` de création de chaîne, et la fonction `String.concat` qui construit la chaîne donnée par la concaténation des chaînes se trouvant dans son second argument, séparées par la chaîne donnée dans son premier argument :
```
utop # String.concat " " ["il"; "fait"; "beau"];;
- : string = "il fait beau"
```

Exercice 1. *Calcul de coefficients binomiaux*

Les coefficients binomiaux $\binom{n}{k}$ vérifient la relation de récurrence suivante : $\binom{n}{0} = \binom{n}{n} = 1$ pour $n \geq 0$ et $\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$ pour $0 < k < n$.

1. Écrivez une fonction OCaml :
   `binom_nav : int -> int -> int`
   qui utilise les relations de récurrence ci-dessus telle que pour des arguments positifs, `binom_nav n k` calcule $\binom{n}{k}$ (si cette valeur est représentable par un `int`) récursivement **et naïvement** (sans aucune exploitation de la redondance des sous-problèmes).
2. Constatez l'inefficacité de cette fonction.
3. Écrivez une nouvelle version :
   `binom_mem : int -> int -> int`
   de votre fonction `binom_nav` qui utilise une table de hachage pour mémoïser les valeurs calculées par une fonction récursive auxiliaire.
4. Testez.
5. Quel est le coût en espace de votre fonction `binom_mem` ?
6. Écrivez une nouvelle fonction OCaml :
   `binom_iter : int -> int -> int`
   qui calcule les coefficients binomiaux sans utiliser aucune récursion et avec un coût en espace inférieur à celui de `binom_mem`.
7. Testez.

Exercice 2. *Résolution de sac à dos*

On pose la variante suivante de problème de sac-à-dos (unaire) : soit $\mathcal{O}$ un ensemble de n objets donnés sous la forme de couples $(p_i, v_i)_{0 \leq i < n}$ de leur *poids* p_i et de leur *valeur* v_i (supposés tous-deux être des entiers naturels), et soit T le poids total maximum (un *volume* maximum serait bien plus réaliste…) qui peut être transporté dans un sac-à-dos (sa « capacité »), on souhaite remplir le sac-à-dos avec un sous-ensemble $\mathcal{C} \subseteq \mathcal{O}$ de valeur maximale en respectant la contrainte de poids. Autrement dit, on cherche $\mathcal{C}$ maximisant $\sum_{(p_i,v_i)\in\mathcal{C}} v_i$ sous la contrainte $\sum_{(p_i,v_i)\in\mathcal{C}} p_i \leq T$; on note $V_{\max}$ la valeur maximale ainsi atteinte.

Pour $m < n$ et $t \leq T$, on note $V(m,t)$ la valeur maximale que l'on peut atteindre en ne prenant que des objets parmi (p_0, v_0), …, (p_m, v_m), et avec un poids total maximum $\leq t$.

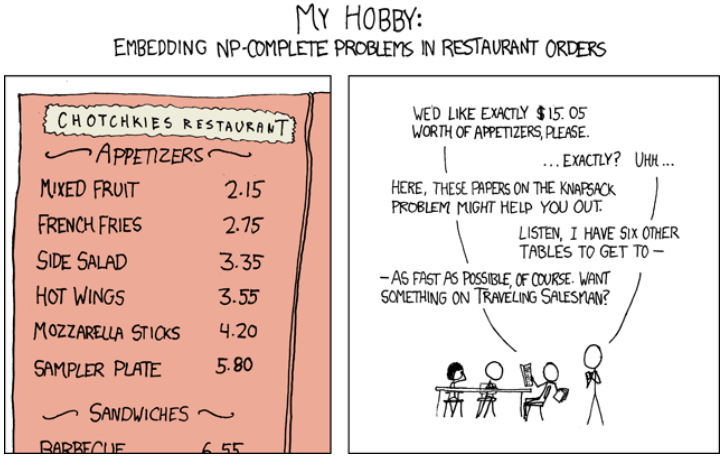
1. Donnez un couple m, t tel que $V_{\max} = V(m,t)$.
2. Que vaut $V(m,t)$ si $t < p_m$? Et si $m = 0$?
3. Déduisez de ce qui précède une formule récursive pour $V(m,t)$.
4. Écrivez une fonction OCaml :

   ```
   knapsack : (int * int) array -> int -> int
   ```

   telle que pour un ensemble d'objets `o` représenté par un `array` de couples *(poids, valeur)* et une capacité `t`, `knapsack o t` s'évalue en la valeur maximale $V_{\max}$ de l'instance correspondante du problème de sac-à-dos.
   Vous devez pour cela utiliser une implémentation récursive **avec mémoïsation** basée sur la formule établie à la question précédente.
5. Testez.
6. Analysez le coût spatial et temporel de votre algorithme, et comparez le avec celui d'une recherche exhaustive.
7. Écrivez une variante :

   ```
   knapsack2 : (int * int) array -> int
   -> int * ((int * int) list)
   ```

   qui s'évalue également en une représentation d'un sous-ensemble de son argument qui atteint la valeur maximale.
8. Testez.
9. Donnez des arguments `o`, `t`, qui permettent d'utiliser votre fonction `knapsack2` pour trouver une solution à la variante de sac-à-dos donnée dans la bande-dessinée ci-dessous. Pensez-vous que cette résolution sera efficace ?



https://xkcd.com/287/

10. Testez.

11. Comment pourriez-vous par la même approche calculer la capacité $T(m, v)$ minimale d'un sac-à-dos de valeur au moins v contenant des objets d'indice 0 à m ?

Exercice 3. *Distance d'édition*

Une *distance d'édition* entre deux chaînes de caractères mesure le nombre minimal d'application de transformations (pour un certain ensemble de transformations fixé) permettant de transformer une chaîne en l'autre et vice-versa. L'objectif de cet exercice est d'utiliser une approche par programmation dynamique pour calculer la distance d'édition quand les transformations autorisées sont la modification d'une lettre en une autre et l'insertion d'espaces. Dans un second temps on cherchera également à expliciter une suite minimale d'application des transformations permettant de passer d'une chaîne à l'autre ; on fera cela sous la forme du calcul d'un *alignement optimal*, c'est à dire une façon d'insérer des espaces dans chacune des chaînes qui minimise la somme du nombre d'espaces insérées et de désaccords parmi les lettres alignées.

N.B. Le module `String` de la bibliothèque standard OCaml (version 5.4) fournit une implémentation du calcul d'une distance d'édition, mais pour un plus grand ensemble de transformations. Elle ne donnera donc pas forcément le même résultat que la fonction que vous devrez implémenter.

*Dans tout cet exercice, on considérera des chaînes de caractères **sans accents**.*

Exemples

— Les chaînes `"tourelle"` et `"tourterelle"` sont à distance d'édition trois. Un alignement optimal est :

```
"tou   relle"
"tourterelle"
```

— Les chaînes `"martre"` et `"matroïde"` sont à distance d'édition quatre. Un alignement optimal est :

```
"ma r tre"
"matroïde"
```

(malheureusement, cet exemple ne sera pas aisé à implémenter à cause de la variabilité de la taille d'encodage de ses caractères en OCaml).

— Les chaînes `"hewwo"` et `"ohai"` sont à distance d'édition cinq. Un alignement optimal est :

```
"hewwo"
" ohai"
```

— Les chaînes `"parapente"` et `"permanent"` sont à distance d'édition quatre. Un alignement optimal est :

```
"par apente"
"permanent "
```

La distance d'édition entre deux chaînes peut se calculer (assez) aisément en utilisant une approche par programmation dynamique. Soit deux chaînes w_1 et w_2 de longueurs respectivement n et m dont on souhaite calculer la distance d'édition, on introduit pour tout $0 \leq i \leq n$, $0 \leq j \leq m$ les solutions $ED_{i,j}$ aux sous-problèmes consistant à calculer la distance d'édition entre les préfixes $w_1[i]$ et $w_2[j]$ de w_1 et w_2 constitués respectivement de leurs i et j premières lettres.

1. Donnez une expression de la distance d'édition entre w_1 et w_2 en fonction des solutions $ED_{i,j}$ aux sous-problèmes.

Les sous-problèmes ci-dessus possèdent une propriété de sous-structure optimale qui permet de les résoudre efficacement.

2. Montrez que pour tout $0 \leq i \leq n$, $0 \leq j \leq m$, $ED_{i,0} = i$ et $ED_{0,j} = j$.
3. Soit $i, j > 0$, on pose $\delta = 1$ si les $(i-1)^{\text{ème}}$ et $(j-1)^{\text{ème}}$ (en partant de zéro) lettres de w_1 et w_2 sont différentes, et $\delta = 0$ sinon. Montrez alors que :

$$ED_{i,j} = \min(ED_{i-1,j-1} + \delta, ED_{i-1,j} + 1, ED_{i,j-1} + 1)$$

4. Déduisez de ce qui précède une fonction OCaml **non récursive** :
   `make_dist : string -> string -> int array array`
   telle que `make_dist w1 w2` s'évalue en un tableau à deux dimensions contenant $ED_{i,j}$ en sa case d'indices i, j. Cette fonction devra avoir un coût en $O(nm)$, produit de la longueur de ses arguments.
5. Écrivez une fonction OCaml :
   `edit_distance : string -> string -> int`
   qui calcule et s'évalue en la distance d'édition de ses arguments.
6. Testez.
7. Quel est le coût en espace de votre fonction `edit_distance` ? Comment celui-ci pourrait-il être amélioré ?

Le calcul d'un alignement optimal entre chaînes de caractères peut se faire efficacement à partir de la connaissance des solutions $ED_{i,j}$ aux sous-problèmes : il suffit d'« inverser » le sens du calcul en partant d'une solution $ED_{n,m}$ du problème complet et de déterminer à chaque solution visitée $ED_{i,j}$ *un* choix qui permet d'étendre un alignement optimal à *un* sous-problème plus petit. Concrètement, si $ED_{i,j} = ED_{i-1,j-1} + \delta$ (avec δ défini comme ci-dessus), alors un alignement optimal peut s'obtenir à partir d'un alignement optimal des i & j premières lettres de w_1 & w_2 et en ajoutant respectivement les lettres d'indice i et j de w_1 & w_2 à la fin de cet alignement ; sinon un alignement optimal peut s'obtenir à partir d'un alignement optimal des $i-1$ & j ou i & $j-1$ premières lettres de w_1 et w_2 et en ajoutant une espace à la fin d'une des deux chaînes de cet alignement.

7. Montrez que le coût de l'algorithme esquissé ci-dessus est un $O(n+m)$ (sans prendre en compte le calcul des solutions $ED_{i,j}$).
8. Écrivez une fonction OCaml :

   ```
   of_char : char -> string
   ```

   qui construit et s'évalue en une chaîne de caractères de longueur un dont l'unique caractère est égal à son argument.
9. Écrivez une fonction OCaml :

   ```
   optimal_alignment : string -> string -> string * string
   ```

   qui calcule et s'évalue en un alignement optimal de ses deux arguments. Par exemple, on peut avoir :

   ```
   utop # optimal_alignment "topaze" "tropique";;
   - : string * string = ("t op aze", "tropique")
   ```
10. Testez.